Genetisch programmeren en codegroei

Bart Wyns

Promotor: prof. dr. ir. L. Boullart
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektrische Energie, Systemen en Automatisering
Voorzitter: prof. dr. ir. J. Melkebeek
Faculteit Ingenieurswetenschappen
Academiejaar 2006 - 2007

ISBN 978-90-8578-155-4
NUR 993
Wettelijk depot: D/2007/10.500/29

It is not the strongest of species that survive,
nor the most intelligent,
but the most responsive to change...
Charles Darwin

Aan mijn echtgenote
Karen Braeckman

Dankwoord

Doctoreren is een les in nederigheid, beseffen dat je eigenlijk niets weet wat je dacht te kennen. Doctoreren is tevens een zoektocht naar het onbekende, niet wetende wat de dag morgen brengen zal. Op mijn tocht kwam ik verschillende boeiende mensen tegen, welke allen op hun manier een bijzondere bijdrage hebben geleverd aan dit werk. Ook dit dankwoord dreigt te vervallen in een al te beperkte opsomming van namen. Toch is het mij een groot genoegen om de mensen te bedanken die mij geholpen hebben bij de totstandkoming van dit werk.

Vooreerst wil ik mij promotor, Professor doctor L. Boullart graag bedanken om me de kans te geven mezelf te bewijzen als wetenschapper en mens. Zijn steun en vooral grenzeloos vertrouwen zal ik me nog lang blijven herinneren. Ik zie hem dan ook als mijn professionele mentor en hoop nog vele jaren met hem te mogen samenwerken.

Verder wil ik ook graag de doctoraatscommissie en examencommissie bedanken, alsook Professor doctor M. Loccufier. Hun adviezen hebben in belangrijke mate dit werk verrijkt.

Ook mijn thesisstudenten Peter en Pieter Jan dank ik hartelijk voor de vruchtbare conversaties over mijn doctoraatsonderzoek.

Ook mijn collega's uit het gebouw Regeltechniek wil ik zeer hartelijk danken. Niet enkel voor alle middagpauze's die leidden tot een ontspannen en aangename werksfeer maar tevens voor de talloze discussies over ieders onderzoek die zeker bijgedragen hebben tot nieuwe inzichten. Speciale dank gaat hierbij uit naar Stijn Van Looy, alsook naar Professor doctor F. De Keyser en Bert Vander Cruyssen met wie ik gedurende twee jaar lang zeer vruchtbaar heb samengewerkt.

De talloze anonieme reviewers van verschillende wetenschappelijke bijdragen wil ik danken voor hun constructieve opmerkingen en suggesties.

Mijn ouders dank ik voor de waarden die ze me hebben meegegeven alsook de kansen die ik gekregen heb. Samen met hen, dank ik ook mijn zus, Inge, voor de steun gedurende al die jaren.

Tot slot dank ik ook de belangrijkste persoon voor mij: mijn allerliefste vrouw Karen.

Inhoud

Samenvatting

De complexiteit van de problemen waarmee een ingenieur heden in contact komt, neemt steeds toe. Hierdoor wordt de verleiding om een aantal concepten uit de biologie te lenen en zo het probleemoplossend vermogen van de ingenieur aan te scherpen, groter.

Reeds vanaf 1960 probeert men computationele systemen te ontwerpen volgens de principes van Charles Darwin. De achterliggende reden waarom men zo geïnteresseerd is in het gebruik van evolutionaire principes bij de organisatie van kunstmatige intelligentie systemen is de verbazende eigenschappen van natuurlijke (biologische) systemen. Darwin's theorie over natuurlijke selectie toonde zeer duidelijk aan dat men met behulp van een verzameling van eenvoudige grondbeginselen in staat is een compleet zelforganiserend, en steeds zichzelf verbeterend systeem te bouwen: een (r)evolutionair algoritme.

Genetisch programmeren (GP) is, in essentie, een evolutionaire optimalisatiemethode voor het automatisch creëren van computerprogramma's. Een kandidaat-oplossing is een uitvoerbaar computerprogramma, vaak voorgesteld als een boom, zonder vaste afmetingen. Structuren met variabele lengte zoals computerprogramma's bieden een groot voordeel: ze zijn immers zeer flexibel en passen zich dynamisch aan in functie van de moeilijkheidsgraad van de voorliggende toepassing. Door het vrijwel onbeperkte aantal mogelijke functies die aan de knopen kunnen toegewezen worden, kan genetisch programmeren toegepast worden in een zeer groot aantal toepassingsgebieden.

Helaas kent het gebruik van structuren met variabele lengte ook een belangrijk nadeel: codegroei. Wanneer GP wordt gebruikt om steeds complexere taken op te lossen, vertonen de geëvolueerde kandidaat-oplossingen een steeds sterker wordende drang om te groeien. De toename in grootte is vaak niet in verhouding met een verbetering van de kwaliteit. Codegroei vormt dan ook een ernstige bedreiging wanneer men GP wenst in te zetten voor com-

plexere en meer realistische taken. Zo nemen computerprogramma's met een groot aantal knopen meer geheugenruimte in beslag waardoor het aantal geheugentoegangen drastisch zal oplopen en de benodigde rekentijd zal stijgen. Bovendien laat de leesbaarheid van de bekomen oplossing sterk te wensen over. Een ander minstens even belangrijk nadeel is de uitvlakking van een verder positief verloop van de kwaliteit (*survival of the fattest*). Grote individuen dragen tevens veel code met zich mee die enkel geschikt is voor de evaluatieomgevingen op basis waarvan de kwaliteit werd toegekend. Verandert men iets aan deze voorbeelden dan zal de kwaliteit vaak negatief worden beïnvloed. Deze individuen zijn met andere woorden niet robuust in functie van verandering in de evaluatieomgeving.

De bestrijding van codegroei in al zijn facetten is dan ook de rode draad die doorheen dit proefschrift loopt. De belangrijkste doelstellingen van dit proefschrift zijn:

1. De ontwikkeling van een nieuwe methode om codegroei te bestrijden zonder de kwaliteit van de geëvolueerde computerprogramma's negatief te beïnvloeden.

2. Het ontwerpen van een nieuwe methode om structurele diversiteit op te meten. Deze methode houdt rekening met het fenomeen codegroei en de aanwezigheid van een groot aantal niet-functionele knopen. Deze methode wordt gebruikt om de invloed van de nieuwe codegroeibegrenzer op de structurele diversiteit te evalueren.

3. De ontwikkeling van adaptieve methoden om een aanvaardbare instelwaarde voor p_{lose} te vinden. Deze sturingsalgoritmen hebben als primair doel de instellingen voor de gebruiker te beperken en te vereenvoudigen alsook om de probleem- en deelboomselectiemethode-afhankelijkheden weg te werken.

4. De implementatie van een algemeen toepasbaar raamwerk waarmee de robuustheid kan opgemeten worden. Verder wensen we tevens de invloed die de nieuw ontwikkelde codegroeibegrenzer uitoefent op de robuustheid van de bekomen oplossingen te evalueren.

De methode die in dit proefschrift wordt ontwikkeld, is de lokale optimalisatieoperator, of kortweg LOSE. LOSE gaat op zoek naar geschikte deelbomen binnen de volledige boomstructuur. Zeer belangrijk hierbij zijn de voorwaarden die worden opgelegd aan een 'geschikte' deelboom. Zo moet de wortel van een deelboom steeds een interne knoop zijn, verschillend van de 'root'

van de volledige boom. Bovendien moet de fitheid van de deelboom, bepaald aan de hand van probleemspecifieke kennis (fitness cases), minstens gelijk zijn aan de fitheid van de volledige boom. Tot op heden bestaat er geen enkele methode die codegroei bestrijdt door hiervan gebruik te maken.

Het opzoeken van een geschikte deelboom kan op twee verschillende manieren gebeuren. Enerzijds zal 'beste deelboomselectie' steeds alle deelbomen bezoeken en evalueren, op zoek naar de deelboom met de hoogste fitheid. Anderzijds ontwerpen we een aantal methoden die stoppen zodra een deelboom wordt gevonden waarvan de fitheid minstens gelijk is aan de fitheid van de volledige boom. We introduceren vier verschillende dergelijke methoden: breedte-eerst zoeken, omgekeerd breedte-eerst zoeken, preorde zoeken en tenslotte postorde zoeken. Alle methoden verschillen onderling afhankelijk van de volgorde waarin ze deelbomen bezoeken.

Dankzij de voorwaarden opgelegd aan een deelboom, zal de gemiddelde boomgrootte gereduceerd worden. Hierdoor wordt de vereiste geheugenruimte om de individuen op te slaan drastisch gereduceerd. Minder benodigde geheugenruimte betekent ook minder geheugentoegangen die nodig zijn om en deel van de boomstructuur in te lezen. Bovendien leidt het gebruik van probleemspecifieke informatie bij de keuze van een geschikte deelboom tot een verdere positieve evolutie van fitheid. LOSE werd uitgetest op drie verschillende benchmarktoepassingen. Hieruit blijkt dat de lokale optimalisatieoperator er glansrijk in slaagt om de gemiddelde boomgrootte te beperken en de kwaliteit te verbeteren.

Het gebruik van probleemspecifieke kennis om de fitheid van de deelboom te bepalen en het bezoeken van een groot aantal deelbomen maakt LOSE een rekenintensieve operatie. Daarom besteden we ook bijzondere aandacht aan de beperking van de benodigde rekentijd. Met behulp van een algoritmische optimalisatie voor het opzoeken van een geschikte deelboom, slagen we erin om deze rekentijd sterk te reduceren. Aan de hand van enkele extra datastructuren worden steeds de fitheidswaarden van alle deelbomen bijgehouden. De verschillende genetische operatoren worden aangepast zodat, naast de structurele wijzigingen, ook de corresponderende stukken uit de rij van fitheidswaarden van een deelboom correct worden gekopieerd. Hierdoor worden er elke generatie, behoudens de initiële generatie, slechts een handvol deelbomen per individu opnieuw geëvalueerd.

Vroegtijdige convergentie is een belangrijk en vaak voorkomend probleem bij verschillende codegroeibegrenzers, ook bij LOSE. Vooreerst introduceren we een verbeterd algoritme om de structurele diversiteit nauwkeurig te kwantificeren. Codegroei beïnvloedt immers het aantal pseudo-isomorfen in de

populatie. Door de boomstructuur eerst te ontdoen van alle niet-functionele code en pas dan het aantal pseudo-isomorfen te berekenen, krijgen we een accurater beeld van de aanwezige genotypische diversiteit.

Toch is er nog een belangrijke hindernis die we moeten overwinnen indien we wensen te spreken van een autonoom werkend GP systeem: de starre statische instelling van de frequentie waarmee LOSE wordt toegepast. Om de invloed van deze parameterinstelling na te gaan, analyseren we eerst de samenstelling van de omvangrijkere zoekruimte van geschikte deelbomen en onderzoeken we de invloed van LOSE op de diversiteit binnen de populatie. Met deze kennis op zak stellen we twee adaptieve modellen voor om de frequentie van LOSE automatisch te bepalen.

Een eerste sterk geparametriseerde methode is gebaseerd op het gebruik van vaaglogische verzamelingen. Deze strategie bleek gelijkaardige resultaten op te leveren dan het gebruik van een vaste instelling. Dit model bleek bovendien complex en vereiste een aantal extra parameterinstellingen zoals het aantal vaagverzamelingen, de vage regels, etc. Om goede waarden voor deze parameters te kiezen, moet de applicatieprogrammeur nog steeds beschikken over voldoende kennis vanuit het probleemdomein. Bovendien bleef er een zekere toepassingsafhankelijkheid bestaan.

De tweede univariate methode gebaseerd op het aantal beschikbare deelbomen in de populatie, bleek bijzonder effectief. Deze methode biedt tal van voordelen zoals zijn eenvoud, de probleemonafhankelijkheid, en parametervrij. Resultaten op twee gekende benchmarktoepassingen hebben aangetoond dat het gebruik van het aantal deelbomen als voortplantingsfrequentie voor LOSE, een duidelijke meerwaarde betekent.

Dankzij codegroei hebben programma's de neiging om overgespecialiseerd te raken op het (meestal enige) voorbeeld dat tijdens de leerfase wordt aangeboden. De geëvolueerde kandidaat-oplossingen kunnen moeilijk overweg met soortgelijke ongeziene fitness cases en zijn niet robuust. Tot dusver is er weinig onderzoek gedaan naar robuustheid. Meestal gaat men ervan uit dat GP vanzelf robuuste programma's oplevert, maar dit wordt nooit gemeten.

In dit proefschrift ontwikkelen we eerst een strategie om nieuwe testvoorbeelden aan te maken. Voorbeelden die verwantschap vertonen met de reeds aanwezige leervoorbeelden. Deze technieken zijn uiteraard probleemspecifiek. Goede en representatieve testvoorbeelden worden bekomen via bouwblokken bij de kunstmatige mier (gebaseerd op de mogelijkheden van de functie- en terminalen verzameling!) en willekeurige selectie uit intervallen bij de 11-bit multiplexer.

Verder zijn we bijzonder geïnteresserd in de invloed van de lokale optimalisatieoperator op de robuustheid van de kandidaat-oplossingen. Specifiek voor de kunstmatige mier zijn we benieuwd naar het spoorvolggedrag van de geëvolueerde oplossingen bij gebruik van LOSE. Bij het multiplexerprobleem zijn we eerder benieuwd of met behulp van een kleiner aantal leervoorbeelden toch hetzelfde resultaat kan worden behaald. Zoals blijkt uit de resultaten verbetert LOSE, door rechtstreeks in te werken op het genotype, op een significante manier de fitheid en, belangrijker nog, ook de robuustheid (onder de vorm van de testfitheid). Ook het aantal leervoorbeelden dat noodzakelijk is om een goed resultaat te bekomen ligt beduidend lager dan wanneer LOSE niet wordt toegepast.

We beëindigen dit proefschrift met een algemeen besluit.

Summary

The complexity of problems which an engineer is facing nowadays, continuously increases. Because of this, the engineer is tempted to lend a number of concepts from biology. In this way the problem solving capacity of the engineer is sharpened.

Already from the early 1960 one tries to design computational systems according to the principles of Charles Darwin. The reason behind why one is so interested in using evolutionary principles with the organization of artificial intelligence systems, are the astonishing properties of natural (biological) systems. Darwin's theory concerning natural selection clearly showed that using a collection of simple rudiments, one is able to build a complete self-organizing, and continuously improving system: a truly (r)evolutionary algorithm.

Genetic programming (GP) is essentially an application of genetic algorithms to computer programs. Typically the genome is represented by a LISP tree expression, so that what evolves is a population of programs, rather than bit strings as in the case of a usual genetic algorithm. Such a program is usually unbounded: there are no restrictions in size or depth. Such variable length structures have a important advantage over fixed length bit strings: they are very flexible and can easily and dynamically adapt to the complexity of the underlying problem domain. By the nearly unlimited number of possible functions which can be assigned to the nodes of the program tree, genetic programming can be applied to a variety of problems.

Unfortunately variable-length structures also have a big disadvantage. While GP tries to solve realistic and more complex tasks, programs tend to collect more and more non-functional and problem specific code. Program trees start to grow beyond control during evolution, often without clear improvement in (best) fitness evolution. In genetic programming this phenomenon is called code growth or bloat. Bloating is an important issue in GP. There are several reasons why it is useful to take some measures against this phenomenon.

First, over time, program trees contain more and more non-functional code. These non-functional portions of code don't contribute to overall fitness. As a consequence, structure altering operations (such as crossover and mutation) are more likely to select and combine elements from degraded material. In this way, fitness evolution in the population will stagnate. Second, large programs consume a huge amount of memory and processing time because they require more space to store their nodes and more nodes are evaluated to calculate fitness. This will increasingly slow down the search process and could limit the usage of GP on problems requiring extensive search processes. Third, people have always tried to construct small and interpretable solutions rather than getting lost in a complex web of interactions between each of the parameters of the problem at hand. Some researchers have claimed that such compact solutions generalize better than large trees. Large solutions often contain large parts of code that are specifically designed for the task that is given.

Fighting code growth is the central theme of this dissertation. The main objectives can be summarized as follows:

1. The development of a new semantically driven method to (1) reduce program size and evolve compact solutions, and (2) increase program fitness.

2. The construction of a new method to accurately describe structural diversity. This method takes code growth (and the presence of a large number of non-functional nodes) into account. This method will then be used to analyze the effect of the new growth limiter on structural diversity.

3. The development of two adaptive methods to find an appropriate breeding rate setting for the semantically driven growth limiter. Both methods try to limit the input of prior knowledge supplied by the application programmer and try to eliminate existing subtree selection scheme dependencies.

4. The development of a general framework to check for train and test performance. The first part focusses on finding an unambiguous definition for similar problems. Second, the influence of code growth on solution quality is considered and we check if limiting bloat leads to better performance.

The new semantically driven method introduced in this research, is called local search operator (LOSE in short). LOSE will search for suited subtrees

within the program tree. A suited subtree must meet the following conditions: (1) a subtree is an internal (function) node, different from the root node of the whole tree, and (2) subtree fitness must be equal or larger than fitness of the whole tree. To date no growth limiter using subtree fitness (semantic information) exists in literature.

The only remaining question is how to find such a suited subtree? In general, we distinguish between (A) strategies searching for the best subtree and (B) strategies stopping when a good (but not necessarily the best) subtree is found. Best subtree selection will evaluate all possible subtree for each individual, searching for the subtree with best fitness. The second category of methods operates in a similar way but the search is 'aborted' when a better subtree is found. The selected subtree is not necessarily the best subtree. Additionally, we introduce four subtree selection strategies: breadth-first subtree selection, inverse breadth-first subtree selection, preorder subtree selection and finally, postorder subtree selection. All methods differ by the order in which subtrees are visited.

The local search operator succeeded in obtaining smaller sized programs with higher fitness. Solutions are smaller compared to GP systems without growth limiters no matter which selection strategy is chosen. The resulting highly fit individuals allow readable and compact solutions, which are easy to understand and can give people an inside look into the solution and its principle components without the need for digging into pages of code. Memory requirements are much lower in the GP system using the local search operator. LOSE was tested on three well-known benchmark applications. Results show that local search successfully reduces program size and increase best fitness in all three benchmarks.

It is obvious that the calculation of subtree fitness for each subtree introduces some computational overhead. But instead of re-evaluating the fitness of each subtree at each generation, we use incremental checking for better subtrees. Each individual contains some additional structures to store fitness of each subtree. All genetic operators are modified so that the corresponding parts of this collection of subtree fitness values are correctly copied to the offspring. Because a program tree is stored in memory in prefix order, only the subtrees with a depth smaller than the depth of the crossover point need to be recomputed. This leads to a large reduction in computational effort.

Premature convergence is often seen with many growth limiters, also with LOSE. First we introduce a new enhanced way of accurately measuring structural diversity. Code growth influences the number of pseudo-isomorphic tuples in the population because of the large quantities of non-functional

code. Our approach simplifies the tree structure using an expression simplifier before calculating the number of 'functional' pseudo-isomorphic tuples. Results show that this diversity measure more accurately describes the real structural diversity present in the population.

If we want our systeem to work autonomously, there is another problem that should be dealt with: the static breeding rate setting for LOSE. We first analyze the superset of (suited) subtrees and the evolution of population diversity to get a clear picture on the effects of this operator. Using this knowledge, we propose two different methods to adaptively determine the breeding rate for LOSE.

The first method was based on fuzzy logic principles. This strategy produced results similar to a static setting for the breeding rate. But fuzzy LOSE still has a few shortcomings. The model turned out to be quite complex (many other parameters were required such as the number of membership functions and their shape, etc.). To choose appropriate values for the fuzzy controller, the application programmer still needs to have some prior information on the problem at hand. As shown the resulting model was still dependent on the benchmark problem.

A second much simpler univariate algorithm was based on feedback given by the LOSE operator itself. This method uses the number of individuals containing a suited subtree as a direction for setting the breeding rate. We showed that this approach has many advantages like its simplicity, its application independence, and it doesn't require any other parameters to be set. Results on two well-known benchmark applications show that using the number of subtrees as a breeding rate for LOSE contributes to standard (static) LOSE's performance.

Bloating is also an important issue when searching for a general, widely applicable solution. Usually the fitness case(s) used for training program trees is learnt 'by heart'. When evaluating the individual on similar (test) fitness cases, performance drops dramatically, even if there is only a fine distinction between train and test fitness cases. Such individuals are often called 'fragile', 'brittle' or not robust. From the train fitness case no general rules of thumb are extracted, rules that are applicable on other fitness cases. So far, little research on robustness has been done. The evolution of robust individuals by genetic programming is often taken for granted. But in reality, robustness is hardly ever verified.

In this dissertation we start by developing a general framework to check for train and test performance. This part focusses on finding an unambiguous

definition for similar problems. These techniques are, of course, problem specific. We found that using buildings blocks (based on the characteristics of the function set) good representative test trails can be found for the artificial ant problem. Random selection from carefully defined intervals produced good train and test fitness cases in the 11-bit multiplexer.

Second, we are particularly interested if code growth influences program robustness (measured by test fitness) and we check if limiting code growth, using LOSE, leads to better performance. In the artificial ant we are curious about the trail-wise behavior that solutions will display. In the Boolean multiplexer problem we are more interested in the number of train fitness cases that are necessary to achieve robust solutions. As results showed, LOSE significantly improved program robustness (higher test fitness) by hoisting a highly fit subtree. The number of train fitness cases also significantly dropped when using local search compared to the standard GP algorithm.

We end this dissertation with a general conclusion.

Hoofdstuk 1

Inleiding

1.1 Zoekmethoden en Darwinisme

De complexiteit van de problemen waarmee een ingenieur heden in contact komt, neemt steeds toe. Hierdoor wordt de verleiding om een aantal concepten uit de biologie te lenen en zo het probleemoplossend vermogen van de ingenieur aan te scherpen, groter.

Reeds vanaf 1960 probeert men computationele systemen te ontwerpen volgens de principes van Charles Darwin. De achterliggende reden waarom men zo geïnteresseerd is in het gebruik van evolutionaire principes bij de organisatie van kunstmatige intelligentie systemen is de verbazende eigenschappen van natuurlijke (biologische) systemen. Darwin's theorie over natuurlijke selectie toonde zeer duidelijk aan dat men met behulp van een verzameling van eenvoudige grondbeginselen in staat is een compleet zelforganiserend, en steeds zichzelf verbeterend systeem te bouwen: een (r)evolutionair algoritme (EA). Dit in tegenstelling tot overgeparametriseerde, vaak breekbare kunstmatige modellen die na veel moeite (input van de applicatieprogrammeur) tot stand zijn gekomen. Mede dankzij de goede beschikbaarheid van voldoende computationele rekenkracht wordt het gebruik van Darwin's principes vereenvoudigd.

Evolutionaire algoritmen werken met een populatie van kandidaat-oplossingen (individuen). Door het bewerken van de genetische structuur van een individu en door een constante evolutionaire druk (*survival of the fittest*) worden steeds betere individuen 'gekweekt', tot uiteindelijk een bevredigende oplossing voor het probleem in behandeling gevonden is.

Genetisch programmeren (GP) is, in essentie, een evolutionaire optimalisatiemethode voor het automatisch creëren van computerprogramma's (Koza 1992). Een kandidaat-oplossing is een uitvoerbaar computerprogramma vaak voorgesteld als een boom, zonder vaste afmetingen. Structuren met variabele lengte zoals computerprogramma's bieden een groot voordeel: ze zijn immers zeer flexibel en passen zich dynamisch aan in functie van de moeilijkheidsgraad van de voorliggende toepassing. Door het vrijwel onbeperkte aantal mogelijke verschillende functies die aan de knopen kunnen toegewezen worden, kan het generieke GP-algoritme toegepast worden in een zeer groot aantal toepassingsgebieden.

1.2 Probleemstelling: genetisch programmeren en codegroei

Helaas kent het gebruik van structuren met variabele lengte ook een belangrijk nadeel. Wanneer GP wordt gebruikt om steeds complexere taken op te lossen, vertonen de geëvolueerde kandidaat-oplossingen een steeds sterker wordende drang om te groeien. De toename in grootte is zeer vaak niet in verhouding met een verbetering van de kwaliteit. Bij de start van het algoritme bedraagt de gemiddelde omvang van kandidaat-oplossingen vaak niet meer dan een handvol knopen (bijvoorbeeld 10). Na enkele generaties dikt de oplossing echter aan tot meer dan duizend knopen zonder dat men een duidelijke verbetering van de kwaliteit kan opmerken.

In de literatuur over GP is dit probleem beter gekend onder de noemer *bloat* of vrij vertaald codegroei. Codegroei vormt een serieuze bedreiging wanneer men GP wenst in te zetten voor complexere en meer realistische taken. Codegroei zorgt immers voor een heleboel nadelen. We geven hier een kort overzicht.

Computerprogramma's met een groot aantal knopen nemen meer geheugenruimte in beslag waardoor het aantal geheugentoegangen drastisch zal oplopen en de benodigde rekentijd zal stijgen. Vaak bestaat het logge programma voor een groot deel uit overbodige functies die helemaal geen bijdrage leveren aan de kwaliteit van de oplossing. Hierdoor wordt er zeer veel rekentijd (en geheugenruimte) verspild aan nutteloze operaties waardoor het vinden van nieuwe betere oplossingen vertraagt. Bovendien laat de leesbaarheid van de bekomen oplossing sterk te wensen over. Omvangrijke oplossingen met honderden of zelfs duizenden knopen zijn onhandelbaar en enige structuur in de oplossing ontbreekt.

Door de toevloed aan code wordt ook de werking van de verschillende genetische operatoren negatief beïnvloed. Zij combineren immers vaak nutteloos genetisch materiaal. De strijd tegen codegroei kan men dus opvatten als een race tegen de klok: zoveel mogelijk goede oplossingen trachten te vinden vooraleer codegroei de zoektocht vertraagt en uiteindelijk stopt.

Een ander minstens even belangrijk nadeel is de uitvlakking van een verder positief verloop van de kwaliteit. *Survival of the fattest* brengt de creatie van kwalitatief betere oplossingen abrupt tot stilstand.

Intuïtief kan men inzien dat grote individuen veel code met zich meedragen die enkel geschikt is voor de evaluatieomgevingen op basis waarvan de kwaliteit werd toegekend. Verandert men iets aan deze voorbeelden dan zal de

kwaliteit vaak negatief worden beïnvloed. Deze individuen zijn met andere woorden niet robuust in functie van verandering in de evaluatieomgeving.

1.3 Doelstellingen

De rode draad doorheen dit proefschrift is de bestrijding van codegroei. De belangrijkste doelstellingen kunnen we als volgt samenvatten.

- Bestaande methoden om codegroei te bestrijden slagen er enkel in om de groei van de computerprogramma's af te remmen of te stoppen. Helaas heeft de beperking van de boomstructuur vaak een negatieve impact op de kwaliteit van de geëvolueerde kandidaat-oplossingen. De doelstelling van dit proefschrift is de ontwikkeling van een nieuwe eenvoudige methode die zowel de omvang van de code beperkt alsook een positieve invloed uitoefent op de fitheid van de oplossingen.

- Een tweede belangrijk probleem binnen het domein van evolutionaire algoritmen in het algemeen en genetisch programmeren in het bijzonder is de voortijdige convergentie op minderwaardige oplossingen en het vastraken in lokale optima. Dit verschijnsel treedt ook zeer vaak op in het gezelschap van verschillende codegroeibegrenzers. Het verband tussen beide is nog niet helemaal duidelijk maar vaak wordt aangenomen dat het gebruik van codegroeibegrenzers de structurele diversiteit in de populatie snel doet afnemen. Het behouden van diversiteit binnen de populatie van individuen is volgens velen de aangewezen methode om dit probleem aan te pakken.

  Inherent hiermee verbonden is natuurlijk de vraag hoe men deze diversiteit kan meten. Een tweede doelstelling van dit proefschrift is het ontwerpen van een techniek die, in aanwezigheid van codegroei, beter dan de bestaande methoden de structurele diversiteit binnen een populatie kwantificeert. Deze aangepaste maat wordt verder gebruikt om de invloed van codegroeibegrenzers op de populatie na te gaan.

- Vrijwel alle codegroeibegrenzers vragen om één (en vaak meerdere) extra parameterinstelling(en) die een afweging tracht te maken tussen de twee belangrijkste doelcriteria bij GP: de kwaliteit van de oplossing enerzijds en de omvang anderzijds. Tot groot ongenoegen van de applicatieprogrammeur is deze parameterinstelling vaak afhankelijk van de voorliggende toepassing en soms zelfs van de codegroeibegrenzer zelf.

Een derde doelstelling van dit proefschrift is het ontwikkelen van alternatieven om deze extra parameterinstelling te omzeilen en zo de instellingen voor de gebruiker te beperken en te vereenvoudigen.

- Sommige auteurs suggereren dat kleine programma's beter bestand zijn tegen veranderingen in de evaluatieomgeving, of dat er minstens een verband bestaat tussen programmagrootte en de robuustheid van een kandidaat-oplossing. De rationale hierachter is dat veel *junk*-code in grote programma's delen van het probleem voorstellen die 'van buiten geleerd' zijn en dus de toepasbaarheid verminderen. Een veelgehoord argument hierbij is dat van Ockhams scheermes: "Entia non sunt multiplicanda praeter necessitatem"[1]. Ruwweg komt dit neer op stellen dat een kleiner programma een betere aanpak van zowel het gestelde probleem als gelijkaardige problemen zal vertonen dan een groter.

  Tot dusver is er weinig onderzoek gedaan naar de robuustheid van geëvolueerde computerprogramma's. Men leert en test nog steeds meestal op hetzelfde voorbeeld, terwijl een aparte leer- en testverzameling in andere takken van de kunstmatige intelligentie standaard zijn. Dit probleem is wel onderkend, maar onderzoeken dienaangaande zijn schaars en gebeuren meestal met ad-hoc methoden. Een laatste doelstelling van dit proefschrift is het opstellen van een algemeen toepasbaar raamwerk waarmee de robuustheid kan opgemeten worden. Verder wordt ook de invloed van de ontwikkelde codegroeibegrenzer op de robuustheid van de bekomen oplossingen vergeleken en worden enkele voorstellen voor de verdere verbetering van de robuustheid aangehaald en uitgewerkt.

De belangrijkste bijdragen van dit proefschrift kunnen we als volgt samenvatten.

- Het ontwikkelen van een nieuwe codegroeibegrenzer die behoudens de reductie van de boomgrootte eveneens zorgt voor een verbetering van de kwaliteit van de geëvolueerde oplossingen.
- De ontwikkeling van een nieuwe diversiteitsmaat die rekening houdt met het fenomeen codegroei en een duidelijker beeld geeft van de aanwezige diversiteit binnen de populatie.
- Het ontwerp van twee adaptieve regelaars om de parameterinstelling van de codegroeibegrenzer te vereenvoudigen.

[1] "Men moet entiteiten niet zonder noodzaak verveelvoudigen"

- De constructie van een raamwerk voor robuustheid en een grondige analyse van de invloed van de codegroeibegrenzer op de robuustheid van kandidaat-oplossingen.

Alle resultaten van de uitgevoerde simulatie in dit proefschrift werden bekomen via een eigen handgeschreven C programma. Dit programma is slechts gedeeltelijk gebaseerd op een bestaande implementatie (Punch & Zongker 1996) en werd grondig gewijzigd en uitgebreid in functie van de noden. Het uiteindelijke softwarepakket is het resultaat van vele uren intensief programmeer- en debugwerk. Deze simulator staat ter beschikking van eenieder die de methode genetisch programmeren wenst toe te passen voor zijn applicatie en bevat alle uitbreidingen die in dit proefschrift worden geïntroduceerd.

1.4 Overzicht van dit proefschrift

Hoofdstuk 2: Genetisch programmeren

Hoofdstuk 2 start met een beknopte beschrijving van de methode 'genetisch programmeren'. Vervolgens wordt de experimentele proefopstelling die in het vervolg van dit werk wordt gebruikt, nader toegelicht alsook de gebruikte instellingen en parameters. Er komen drie benchmarktoepassingen aan bod; de kunstmatige mier, het regressieprobleem, en de 11-bit multiplexer. Alle toepassingen worden kort voorgesteld. Tenslotte worden enkele eigen uitbreidingen van de GP-simulator besproken. We eindigen dit hoofdstuk met een korte beschrijving van de verschillende kenmerken van het paradigma van genetisch programmeren.

Hoofdstuk 3: Codegroei

In hoofdstuk 3 wordt het begrip codegroei gedefinieerd bij genetisch programmeren. Aan de hand van een uitgewerkt voorbeeld van codegroei bij de kunstmatige mier wordt duidelijk gemaakt welke nadelen verbonden zijn aan de explosieve groei van de boomstructuren in genetisch programmeren. Vervolgens geven we een kort overzicht van de verschillende principiële verklaringen voor codegroei, gevolgd door een uitgebreid overzicht van de verschillende technieken om codegroei te bestrijden.

Dit hoofdstuk bevat het leeuwendeel van de literatuurstudie omtrent codegroei. De beschrijving van de hypothesen die codegroei trachten te verklaren en het overzicht van de methoden om codegroei te bestrijden wordt steeds

gevolgd door een kritische nabeschouwing. Dit proefschrift heeft echter niet tot doel een zoveelste verklaring van codegroei te formuleren maar eerder een nieuwe methode te ontwikkelen die voldoet aan de voorwaarden uit 1.3. Een literatuuroverzicht zou echter niet volledig zijn indien de beschrijving van deze hypothesen wordt vergeten.

Hoofdstuk 4: Lokale optimalisatie

In hoofdstuk 4 introduceren we een nieuwe aanpak om codegroei te bestrijden door middel van lokale optimalisatie van de boomstructuur. Centraal staat het begrip 'geschikte deelboom'. Dit hoofdstuk begint met de beschrijving van de ingrediënten van de nieuwe methode. Vervolgens bespreken we kort de gebruikte experimentele proefopstelling om deze nieuwe methode te testen, gevolgd door een bespreking van de resultaten. De nieuwe operator wordt vergeleken met standaard genetisch programmeren (zonder codegroeibegrenzers) alsook met verschillende algemeen aanvaarde methoden om codegroei te bestrijden.

We introduceren tevens een algoritmische optimalisatie voor de bepaling van de fitheidswaarden van deelbomen, aangezien de benodigde rekentijd drastisch kan oplopen.

Hoofdstuk 5: Diversiteit

In hoofdstuk 5 wordt het begrip diversiteit bij genetisch programmeren gesitueerd. Vooreerst wordt besproken wat diversiteit precies is, gevolgd door een literatuuroverzicht van de verschillende types diversiteit. Codegroei beïnvloedt echter het beeld van de aanwezige structurele diversiteit waardoor klassieke structurele diversiteitsmaten ontoereikend zijn. Hierop volgt de bespreking van een verbeterde structurele diversiteitsmaat alsook de experimentele proefopstelling om de prestatie van deze maat na te gaan. We sluiten dit hoofdstuk af met de analyse en interpretatie van de resultaten en situeren kort welke rol de nieuwe maat zal spelen in de volgende hoofdstukken.

Hoofdstuk 6: adaptieve sturing en optimalisatie

Uit de resultaten die werden behaald op drie gekende benchmarktoepassingen (hoofdstuk 4) is echter duidelijk dat de nieuw ontwikkelde codegroeibegrenzer nog enkele kinderziektes vertoont. Zo resulteert het gebruik van een vaste instelwaarde voor de probabiliteit waarmee de codegroeibegrenzer wordt toegepast, soms in een verlies aan diversiteit.

In hoofdstuk 6 analyseren we de evolutie van de verzameling deelbomen en trachten we een verklaring te vinden waarom bepaalde instellingen voor de probabiliteit waarmee de nieuwe operator wordt toegepast, leiden tot verlies aan diversiteit. Vervolgens stellen we twee adaptieve regelaars voor die deze probabiliteit online (gedurende evolutie) zullen aanpassen.

Hoofdstuk 7: Robuustheid en dataselectie

Systemen gebaseerd op genetisch programmeren gebruiken totnogtoe veelal één enkele vaste leeromgeving om programma's te optimaliseren voor een bepaald, welomlijnd probleem. De geëvolueerde programma's vertonen dan ook vaak een optimaal (of bijna-optimaal) gedrag voor deze specifieke leeromgeving, maar falen jammerlijk wanneer een gelijkaardig probleem aangeboden wordt. Hoofdstuk 7 beschrijft een algemeen toepasbaar raamwerk waarmee robuustheid kan gemeten worden. Vervolgens onderzoeken we, door middel van een aantal experimenten op twee van bovenstaande benchmarkproblemen, wat nu de invloed is van de nieuwe codegroeibegrenzer op de robuustheid van de bekomen kandidaat-oplossingen. Vaak wordt immers gesteld dat kleinere (compacte) oplossingen beter in staat zijn om met nieuwe situaties om te gaan (zogenaamde Ockham's scheermes).

Hoofdstuk 8: Besluiten en verder onderzoek

We eindigen in hoofdstuk 8 met een algemeen besluit en enkele interessante denkpistes voor verder onderzoek.

Bijlage A: Genetisch programmeren in detail

Bijlage A geeft een gedetailleerde beschrijving van genetisch programmeren. Aangezien genetisch programmeren veel verwantschap vertoont met een genetisch algoritme wordt begonnen met de beschrijving hiervan. De evolutionaire cyclus die generatie na generatie wordt doorlopen, wordt in detail toegelicht. Deze bijlage is in hoofdzaak bedoeld voor lezers die niet vertrouwd zijn met genetisch programmeren.

Bijlage B: De parameterinstellingen voor FIS

Bijlage B geeft een overzicht van alle parameters en bijhorende instelwaarden voor de adaptieve vaaglogische sturing van de nieuwe codegroeibegrenzer uit hoofdstuk 6.

Bijlage C: Afkortingen, symbolen en vertalingen

Bijlage C geeft een overzicht van de afkortingen, symbolen en vertalingen die voorkomen in dit proefschrift.

Hoofdstuk 2

Genetisch programmeren

We starten dit hoofdstuk met een korte beschrijving van de methode genetisch programmeren en we benadrukken de elementen die genetisch programmeren van de andere evolutionaire algoritmen onderscheiden. We lichten eveneens de experimentele proefopstelling die in het vervolg van dit werk wordt gebruikt, nader toe alsook de gebruikte instellingen en parameters. Er komen drie benchmarktoepassingen aan bod; de kunstmatige mier, het regressieprobleem en de 11-bit multiplexer. Tenslotte worden de gebruikte software en enkele eigen uitbreidingen van de GP-simulator besproken. We eindigen dit hoofdstuk met een korte beschrijving van de verschillende kenmerken van het paradigma van genetisch programmeren.

Lezers die niet vertrouwd zijn met genetisch programmeren verwijzen we naar de bijlage A voor een meer uitgebreide bespreking van deze methode. Een verkorte versie kan men terugvinden in paragraaf 2.2.

2.1 Genetisch algoritme voor programma-inductie

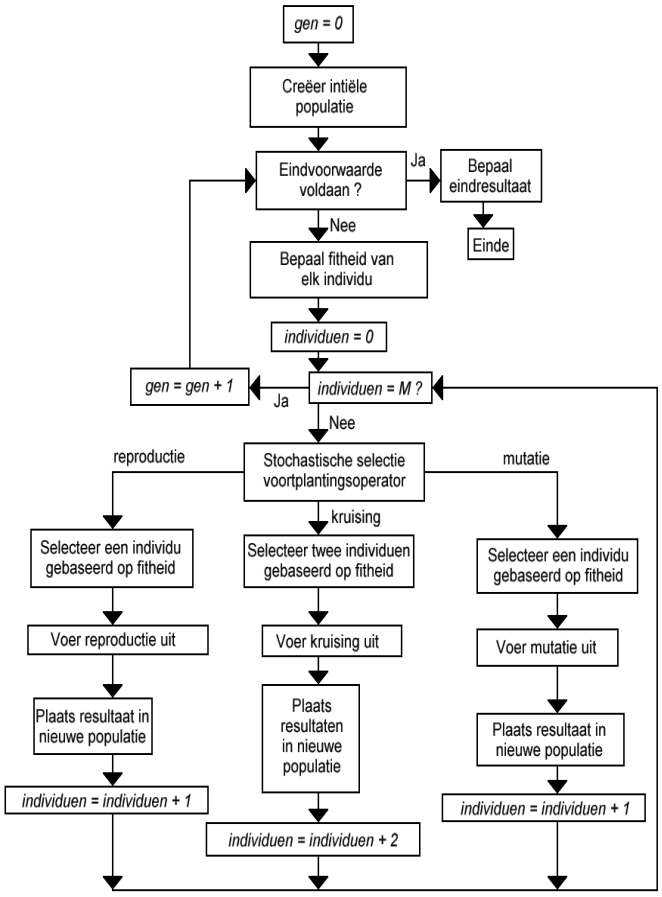
In tegenstelling tot de traditionele optimalisatiealgoritmen zijn evolutionaire algoritmen globale en robuuste optimalisatiemethoden, nauw verbonden met natuurlijke evolutionaire processen, zoals beschreven door Charles Darwin (Darwin 1959). Evolutionaire algoritmen vertrekken vanuit een populatie van willekeurig gekozen mogelijke oplossingen, in plaats van één enkele mogelijke oplossing. Op die manier verkleint ook de kans dat het algoritme convergeert naar een lokaal optimum. De overlevingkans van een individu en de creatie van nieuwe individuen, generatie na generatie, worden gegarandeerd door elementaire Darwinistische richtlijnen. Het is verbazingwekkend vast te stellen hoe uitgaande van willekeurige elementen en onder invloed van het principe *survival of the fittest*, EA's omwille van hun robuustheid en universaliteit efficiënte oplossingen genereren in een multidimensionele zoekruimte met heel wat ruis en lokale optima.

De meest bekende vorm van EA's is het genetische algoritme (GA). Deze methode werd bedacht tijdens de jaren '60 door onderzoeker John Holland aan de universiteit van Michigan (Holland 1975). Net zoals een genetisch algoritme, steunt ook genetisch programmeren (GP) op dezelfde Darwinistische principes. GP is echter in staat om computerprogramma's te kweken in plaats van bitstrings met een vaste lengte zoals bij klassieke implementaties van een GA[1]. Verder heeft GP enkel een aantal (invoer, uitvoer)-combinaties nodig op basis van dewelke het programma zal worden opgebouwd of geïnduceerd.

Een individu wordt vaak voorgesteld als een boom, zonder vaste afmetingen, die kan uitgevoerd worden als een programma. De evolutionaire cyclus is net dezelfde als die van een genetisch algoritme en begint met het aanmaken van een initiële populatie. De programmabomen in deze populatie zijn willekeurige samenstellingen van de gebruikte functies en terminalen. Vervolgens wordt elk programma in de populatie uitgevoerd, en wordt er een fitheid aan toegekend op basis van de prestatie. Met behulp van een aantal voortplantingsoperatoren wordt een nieuwe generatie programma's aangemaakt, waarbij de betere (meer fitte) individuen meer kans hebben om hun genetisch materiaal naar de volgende generatie over te dragen. Dit proces van creatie en evaluatie wordt verder herhaald, tot een bepaald eindcriterium voldaan is.

Een volledig overzicht van de evolutionaire cyclus bij genetisch programmeren kan men vinden in Figuur 2.1. In de volgende paragrafen verklaren we kort elke stap uit het hierboven samengevatte proces.

[1] Sommige types GA ondersteunen ook structuren met variabele lengte (Radcliffe & George 1993, Kargupta 1996, Goldberg *et al.* 1993, Fullmer & Miikkulainen 1992).



Figuur 2.1: *Cyclus van genetisch programmeren*

2.2 Ingrediënten van genetisch programmeren

2.2.1 Het genoom

Individuen kunnen op verschillende manieren voorgesteld worden (Nordin & Francone 1999, Banzhaf *et al.* 1997, Cramer 1985, Brameier & Banzhaf

2001) maar een boomstructuur komt het vaakst voor. Ook in dit proefschrift maken we enkel gebruik van de boomvoorstelling en worden alle concepten en ideeën verklaard en geïllustreerd aan de hand hiervan.

De allereerste stap bij het bouwen van een correcte boom is het bepalen van de functieset, dat is de verzameling mogelijke functies voor de interne knopen van een boom, en de terminalenset, de verzameling mogelijke eindkopen van een boom. De samenstelling en inhoud van beide verzamelingen wordt bepaald door de probleemstelling. Bij een regressieprobleem kan men bijvoorbeeld opteren voor rekenkundige operatoren ($+$, $-$, $*$ ...) en wiskundige functies (sin, cos, log, exp ...). Terminalen zijn dikwijls variabelen (zoals invoervariabelen, toestandsvariabelen, sensoren, detectoren ...) of constanten.

Om mogelijk te maken dat het GP-proces een aanvaardbare oplossing vindt voor een probleem, is het belangrijk om geschikte functies en te kiezen. In GP-terminologie noemt men dit vaak de *sufficiency* eigenschap. Ook al lijkt deze eigenschap triviaal in theorie, toch is het in de praktijk meestal niet eenvoudig om geschikte functies en terminalen te kiezen. Onnodige functies en onnodige terminalen opnemen in de sets kan een negatieve invloed uitoefenen op het snel vinden van een oplossing. Anderzijds, nodige functies of terminalen niet opnemen maakt het vinden van een oplossing onmogelijk.

Een tweede belangrijk criterium waar de functie- en terminalensets aan moeten voldoen, is sluiting (*closure*). Elke functie moet elke mogelijke waarde voortgebracht door een functie of terminaal kunnen aannemen als argumenten. Elke functie moet dus welgedefinieerd zijn voor elke combinatie van argumenten die ze mogelijkerwijs kan binnenkrijgen. Dit lijkt een grote beperking te zijn voor de expressieve vrijheid van deze genetische structuur, maar dat valt in de praktijk erg mee. Eenvoudige maatregelen voor een klein aantal uitzonderingsgevallen zijn over het algemeen voldoende om aan het principe van sluiting te voldoen.

Het construeren van een individu begint met het willekeurig kiezen van een functie voor de beginknoop. Voor elk van de argumenten van die beginknoop wordt vervolgens weer willekeurig een functie gekozen, en zo verder voor elke nieuw toegevoegde knoop. Is een gekozen functie een terminaal, dan vormt die een eindknoop. Er bestaan een aantal verschillende mogelijke implementaties om deze initiële populatie te construeren, die elk resulteren in een verschillende samenstelling van de populatie qua vorm en afmetingen.

Bij de *full* methode is in de uiteindelijke boom, elk pad tussen een eindknoop en de beginknoop gelijk aan een opgegeven maximale diepte. Bij de *grow* methode (groeimethode) worden bomen geconstrueerd waarbij elk pad tus-

sen een eindknoop en de beginknoop niet groter is dan de opgegeven maximale diepte.

Beide methodes worden dikwijls gecombineerd in een gemengde methode *ramped half-and-half*, waarbij voor elke diepte (bijvoorbeeld tussen 2 en 6) telkens de helft van de bomen via de *full* methode aangemaakt wordt en de andere helft via de *grow* methode. Deze methode produceert de grootste verscheidenheid aan vormen en dieptes, en is dus meest aangewezen om een grote structurele diversiteit te introduceren in de initiële populatie. Duplicaten worden uit de initiële populatie verwijderd.

2.2.2 De fitheid

Bij genetisch programmeren wordt elk programma (elke boom) uit de populatie uitgevoerd op één of meer vooraf bepaalde gevallen uit het probleemdomein. Daarna wordt gekeken hoe goed de gegenereerde uitvoer overeenstemt met de gewenste uitvoer. Daarbij is het dan vaak de bedoeling dat het uiteindelijke resultaat van de evolutie niet alleen deze bepaalde gevallen zal aankunnen, maar ook andere nog onbehandelde problemen uit hetzelfde domein. De vooraf bepaalde gevallen uit het probleemdomein waarop het programma getest wordt, noemen we de *fitness cases*, het uitvoeren van het programma op deze fitness cases is de evaluatie van het programma.

2.2.3 De selectie van geschikte individuen

Bij het aanmaken van een nieuwe generatie individuen moeten individuen uit de vorige generatie geselecteerd worden om hun genetisch materiaal door te geven. Vaak wordt roulettewielselectie gebruikt, waarbij de kans op selectie proportioneel is met de fitheidswaarde. Dit wordt vaak ook *fitness proportionate selection* (FPS) genoemd. Stel f_i gelijk aan de fitheid van één individu dan is de kans p_i waarmee een individu i wordt geselecteerd gelijk aan: $p_i = \frac{f_i}{\sum_{i=1}^{N} f_i}$.

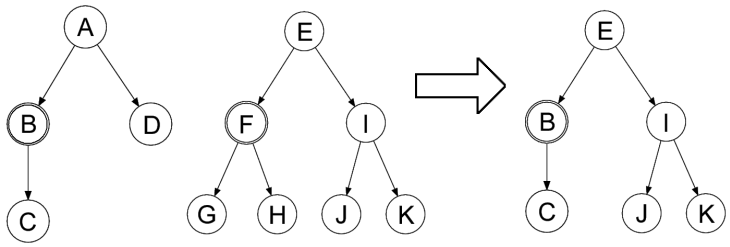
Ook tornooiselectie wordt vaak toegepast. Bij tornooiselectie selecteert men op willekeurige basis een aantal individuen uit de populatie. Dit aantal is een parameter die ingesteld moet worden. De geselecteerde individuen moeten zich met elkaar meten in een tornooi waar enkel het beste individu zal overleven. Een groot aantal deelnemers verkleint de kans dat minder goede individuen het tornooi overwinnen. De selectiedruk wordt groter.

We spreken van elitisme wanneer de selectiemethode enkel het beste individu in beschouwing neemt.

2.2.4 De operatoren

Het basismechanisme achter *survival of the fittest* is reproductie (duplicatie); de meest fitte kandidaat-oplossingen blijven leven, de minst fitte sterven af. De reproductieoperator kopieert een individu, geselecteerd met behulp van een selectiemethode gebaseerd op fitheid, ongewijzigd naar de volgende generatie.

Kruising of recombinatie zorgt voor de verspreiding en voor het ontstaan van andere combinaties van succesvol genetisch materiaal. De kruisingsoperator neemt twee individuen als ouders, beiden geselecteerd volgens een bepaalde selectiemethode, en creëert twee nakomelingen die elk bestaan uit delen van beide ouders (Fig. 2.2). De kruisingsoperatie begint met het willekeurig kiezen van een kruisingspunt in elk van de ouders. Dit kruisingspunt kan verschillend zijn in beide ouders! Vervolgens wordt de deelboom vertrekkend van het kruisingspunt van de eerste boom overgeplaatst naar het kruisingspunt van de tweede ouder, en omgekeerd. Merk op dat de ouders waarschijnlijk niet dezelfde vorm of afmetingen zullen hebben, en dat zulk een overgeplaatste deelboom soms slechts uit één terminale knoop kan bestaan.



Figuur 2.2: *Recombinatie bij genetisch programmeren; de linkse twee bomen zijn de ouders, de dubbel omlijnde knopen B en F zijn hun respectievelijke kruisingspunten. De rechtse boom is één van beide nakomelingen.*

Het is ook mogelijk dat het selectiemechanisme tweemaal hetzelfde individu als ouders gekozen heeft voor één kruising. Bij genetisch programme-

ren levert de kruising van een individu met zichzelf mogelijks twee nieuwe programma's op, doordat het kruisingspunt niet in beide ouders op dezelfde plaats moet liggen.

2.2.5 Eindcriterium

In theorie zou een GP-proces, zoals de natuurlijke evolutie, onbeperkt in de tijd kunnen blijven voortgaan. In de praktijk blijft het uiteraard wel nodig het proces ooit stil te leggen. Een mogelijk eindcriterium zou zijn de evolutie te stoppen eens een volledig correcte oplossing gevonden is. Een andere mogelijkheid is op voorhand een maximum aantal generaties vast te leggen en de evolutie te stoppen wanneer dat aantal bereikt is.

2.3 Benchmarktoepassingen

Het domeinonafhankelijke design van genetisch programmeren[2] biedt een enorme vrijheid bij de keuze van een specifieke toepassing. Eender welke toepassing waarvan potentiële oplossingen opgemeten kunnen worden, kan een kandidaat-toepassing zijn voor genetisch programmeren. De vrije (vrijwel ongelimiteerde) keuze van samenstelling van de functie- en eindknopenverzameling (door de applicatieprogrammeur) en de toch wel zeer lichte eisen die aan deze verzamelingen worden opgelegd ('sluiting', paragraaf 2.2.1) zorgen voor een bijkomend duwtje in de rug van GP. Het is dan ook niet verwonderlijk dat gedurende de laatste 10 jaar GP ingang gevonden heeft in een groot aantal toepassingen van zeer diverse aard (Koza 1992). GP heeft oplossingen geformuleerd die even efficiënt zijn als oplossingen die met behulp van andere methoden werden geconstrueerd in tal van toepassingen zoals optimale controle, cellulaire automaten, sturing van ruimtesatellieten, moleculaire biologie en het ontwerp van digitale circuits, etc. GP slaagt er zelfs in oplossingen te construeren die quasi even efficiënt zijn als de oplossingen gemaakt door mensenhanden.

Ondanks de voordelen van genetisch programmeren en de toevloed aan artikelen die de toepasbaarheid en de efficiëntie van GP trachtten te bewijzen, moeten we vanuit het standpunt van de applicatie toegeven dat GP niet de enige methode is die goede resultaten kan produceren op een aantal toepassingen uit bovenstaande domeinen. Zo bestaan er analytische zoekmethoden

[2]De evolutionaire GP cyclus steunt niet op de onderliggende toepassing die men wil optimaliseren. Enkel de bepaling van de fitheid gebruikt domeinspecifieke kennis.

of alternatieve heuristieken zoals simulated annealing en hill climbing die oplossingen produceren die kwalitatief minstens even goed zijn. Vaak vinden deze methoden deze oplossingen in minder tijd en/of vereisen ze minder voorkennis over het probleem.

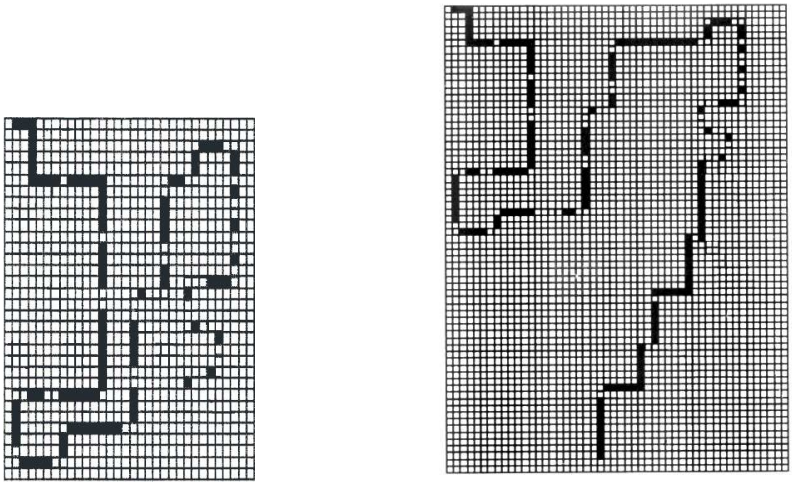
Toch zijn er een aantal domeinen waarop genetisch programmeren onbetwistbaar een duidelijke meerwaarde kan bieden: complex digitaal schakelontwerp, sturingsalgoritmen voor robotica doeleinden, en digitale signaalverwerking (Esparcia Alcazar & Sharman 1996). De gezochte oplossingen binnen deze domeinen zijn vaak te complex om handmatig te programmeren en wiskundige technieken schieten tekort om een éénduidige analytische oplossing aan te bieden.

Vooral op gebied van digitaal schakelontwerp volgen de successen elkaar in snel tempo op. Niet alleen bij het ontwerp van nieuwe schakelingen met een complexe functionaliteit worden belangrijke inspanningen geleverd maar ook de optimalisatie van bestaande circuits tot zelfs het evolutionair ontwerp van FPGA's (field programmable gate arrays) in het kader van evolvable hardware (Greenwood & Tyrrell 2006) zijn populaire onderwerpen. Regeltechnische problemen (vooral niet-lineaire systemen zoals bij optimale controle), op hun beurt, zijn bijzonder geschikt voor GP aangezien het met traditionele wiskundige methoden quasi onmogelijk is om analytische oplossingen af te leiden voor interessant praktische (en realistische) toepassingen (Koza 1992, broom balancing). Bovendien staan regeltechnici open voor benaderende oplossingen en worden kleine verbeteringen van de prestatie zeer hoog in waarde geschat.

Voor deze scriptie worden drie gekende standaardproblemen gebruikt. Voor elke verbetering ontwikkeld in het proefschrift selecteren we het standaardprobleem dat de werking het best illustreert.

2.3.1 De kunstmatige mier

Verschillende auteurs hebben de kunstmatige mier reeds gebruikt (Koza 1992, Langdon & Poli 1998*a*, 1997*a*, *b*, Chellapilla 1998). Er wordt een programma ontwikkeld dat een kunstmatige mier langs een voedselspoor laat bewegen zodat de mier zoveel mogelijk voedselkorreltjes opraapt. De fitness case is een welbepaald voedselspoor op een roostervormig terrein met welbepaalde afmetingen. Vaak gebruikte voedselsporen zijn het Santa Fe spoor, bestaande uit 89 voedselkorreltjes op een veld van 32 op 32 posities, en het Los Altos spoor van 105 voedselkorreltjes op een veld van 100 op 100 posities (Fig. 2.3). We gebruiken steeds het Santa Fe spoor.

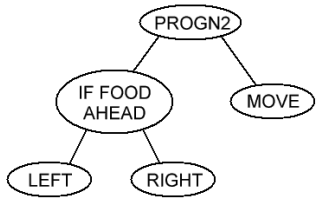


Figuur 2.3: *Fitness cases voor de kunstmatige mier; links het Santa Fe spoor op een gebied van 32 op 32 posities, rechts het Los Altos spoor, gelegen in een deel van 50 op 70 posities van een totaal gebied van 100 op 100.*

Het programma wordt ontwikkeld in de vorm van een eindige automaat. De evaluatie bestaat erin dat dit programma herhaaldelijk uitgevoerd wordt tot alle korreltjes opgeraapt zijn of totdat een bepaald aantal tijdseenheden overschreden is. Een voorbeeld van een dergelijke automaat is te zien in Figuur 2.4

De mier heeft op elk moment een positie (x, y) en een kijkrichting (noord, oost, zuid of west). De terminalenset bevat drie functies MOVE, LEFT, RIGHT, waarvan de uitvoering telkens één tijdseenheid in beslag neemt. De eerste functie verandert de positie van de mier door ze één stap in de kijkrichting te laten nemen. Als ze hierbij op de positie van een voedselkorreltje terechtkomt, wordt dit korreltje opgeraapt. De tweede en de derde veranderen de kijkrichting van de mier met respectievelijk een halve draai ($90°$) naar links of naar rechts. De functieset bevat om te beginnen de functie IF-FOOD-AHEAD, die twee argumenten heeft. Dit is een keuzefunctie, waarvan de linkse deelboom wordt uitgevoerd als een voedselkorreltje op de positie vóór die van de mier ligt. De rechter deelboom in het andere geval. Deze functie stelt de sensor van de mier voor. Merk op dat de mier een zeer beperkte weergave

heeft van wat er in de buurt te vinden is. Verder bevat de functieset een aantal functies die hun deelbomen één na één uitvoeren, van links naar rechts. Deze functies bestaan voor verschillende aantallen argumenten: PROGN2 voert twee deelbomen uit en PROGN3 drie deelbomen.



Figuur 2.4: *Een eenvoudige kunstmatige mier; wanneer deze mier voor zich uit voedsel ziet liggen, draait ze weg naar links, anders naar rechts. Na de draai zet ze steeds een stap vooruit, om vervolgens opnieuw te kijken of er voedsel voor haar ligt, en zo verder. Het spreekt voor zich dat dit geen zeer efficiënte mier is, maar ze demonstreert wel alle functies en terminalen van het probleem van de kunstmatige mier.*

De fitheid van een individu is gelijk aan het aantal opgeraapte voedselkorreltjes gedeeld door het aantal aanwezig voedselpakketten (=89). Het is duidelijk dat er slechts een beperkt aantal mogelijke verschillende fitheidswaarden bestaat, namelijk het aantal voedselkorreltjes in het spoor plus één (voor het geval dat er nul korreltjes opgeraapt zijn).

Verschillende onderzoekers hebben aangetoond dat de prestatie van genetisch programmeren niet veel beter is in vergelijking met simulated annealing, hill climbing of zelfs willekeurig zoeken. Willekeurig zoeken naar correcte oplossingen zal hoogstwaarschijnlijk geen resultaten geven op problemen waar er slechts één unieke oplossing bestaat. Bij de kunstmatige mier bestaan echter een exponentieel groeiend aantal oplossingen (Langdon & Poli 1998*b*). Dit verklaart de goede prestatie van andere stochastische zoektechnieken. Er wordt vaak beweerd dat het fitheidslandschap van de kunstmatige mier veel vergelijking vertoont met reële problemen. Het aantal mogelijk programma's is zeer groot en vormt een Karst-landschap met veel misleidende optima en plateau's doorkruisd door diepe dalen. Er bestaan verschillende conflicterende oplossingen voor het mierprobleem waarvan sommige veroorzaakt door symmetrie in de probleemdefinitie en andere door de samenstelling van de

primitieve verzameling functies en terminalen. In hun analyse met behulp van schema's (Poli & Langdon 2002), tonen Langdon en Poli aan dat het probleem misleidend is op verschillende niveau's. Zo zijn grotere programma's kwalitatief beter maar is de densiteit van oplossingen lichtjes kleiner (de verhouding tussen grote individuen en het aantal oplossingen tussen deze grote individuen is kleiner). Het aantal oplossingen binnen hyperschema's met dezelfde lengte is niet homogeen verdeeld. Er zijn veel lagere orde schema's die nodig zijn om tot een oplossing te komen maar zelf een lager dan gemiddelde fitheidswaarde hebben. En tot slot hebben de meeste schema's een hoge variantie waardoor ze een met ruis behept beeld geven van de verwachting van hun fitheid.

De combinatie van deze redenen maakt dat de kunstmatige mier een bijzondere uitdaging vormt voor genetisch programmeren.

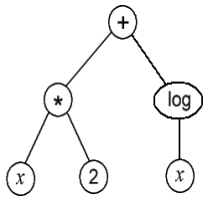
2.3.2 Regressie

Problemen waarbij men een wiskundig verband wenst af te leiden op basis van een aantal (invoer, uitvoer)-waarden, staan bekend onder de noemer symbolische regressie. De uitdrukking, in symbolische gedaante, zal doorheen de invoer en uitvoercombinaties lopen of toch op zijn minst een goede benadering zijn. Deze vorm van regressie wijkt af van de gekende meer traditionele lineaire regressie (en ook exponentiële en kwadratische regressie) in de zin dat de structuur van het model niet op voorhand door de gebruiker wordt vastgelegd terwijl dit bij bijvoorbeeld exponentiële regressie wel het geval is ($a \cdot e^{bx}$). Een aantal paren (abscis, ordinaat) vormen de fitness cases. De evaluatie bestaat erin dat het programma voor elke abscis éénmaal uitgevoerd wordt, in de hoop dat de waarden die het programma telkens oplevert, overeenkomen met de bijhorende ordinaatwaarden. De ordinaatwaarden wordt gegenereerd aan de hand van een vooraf bepaalde functie. De keuze van deze functie is vrij, maar vaak wordt geopteerd voor een vierde orde polynoom $x^4+x^3+x^2+x$. Ook in dit proefschrift wordt geopteerd voor deze doelfunctie. Merk op dat het genetisch programma enkel de (abscis, ordinaat) paren te zien krijgt. De gekozen wiskundige uitdrukking is enkel gekend door de applicatieprogrammeur.

De functieset bevat bij het regressieprobleem een aantal wiskundige operatoren. Standaardleden zijn de optelling, aftrekking, vermenigvuldiging en deling, evenals de wiskundige functies $\exp$, $\log$, $\sin$ en $\cos$. Merk op dat om aan het principe van sluiting te beantwoorden de deling en de logaritme 'beschermd' moeten zijn, dit wil in de praktijk zeggen dat een deling door nul

één oplevert, dat de logaritme steeds van de absolute waarde genomen wordt, en dat de logaritme van nul gelijk is aan nul.

De terminalenset heeft hier één of twee leden. Zeker bevat ze de invoerwaarde of variabele x, die bij evaluatie de waarde van de huidige abscis oplevert. Verder kan ze ook willekeurige constanten (ERC) bevatten. Een ERC is een terminaal met een bepaalde constante waarde. Wanneer de ERC aangemaakt wordt, bijvoorbeeld bij het genereren van een willekeurige boom voor de initiële populatie, of bij de mutatie van een bestaande boom, krijgt hij een willekeurige waarde. Deze waarde behoudt hij gedurende zijn volledige bestaan, ook als hij door kruising in een andere boom terechtkomt. Een voorbeeld van een boom is te zien in Figuur 2.5.



Figuur 2.5: *Bij het symbolische regressieprobleem zijn de kandidaatoplossingen wiskundige expressies. Dit individu stelt bijvoorbeeld de functie* $2x + \log(x)$ *voor.*

De fitheid moet uitdrukken hoe goed een individu de doelcurve benadert. In dit proefschrift gebruiken we als fitheid de som van de absolute waarden van de afwijkingen per abscis tussen de gegenereerde en de werkelijke ordinaat.

$$\text{fitheid} = \frac{1}{1 + \sum_{i=1}^{20} |\text{echte waarde} - \text{voorspelde waarde}|}$$

Vaak wordt een maximale toegelaten afwijking gebruikt, de afknijpwaarde (*cutoff value*). Als de afwijking voor een fitness case groter is dan die maximale waarde, wordt voor de berekening van de fitheid de afknijpwaarde in plaats van de ware afwijking gebruikt. In tegenstelling tot bij de kunstmatige mier (met zijn discreet aantal fitheidswaarden) is het fitheidsbereik hier continu.

2.3.3 De 11-bit multiplexer

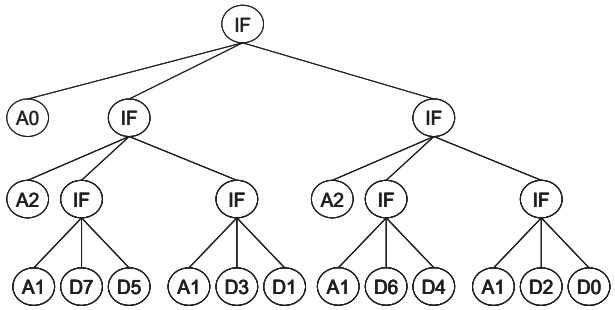
Booleaanse functies zijn de toepassingen bij uitstek om verschillende redenen (Koza 1992). Ten eerste, wanneer we een oplossing naderbij bekijken dan is het eenvoudig om te zien welke componenten van de boomstructuur bijdragen tot de prestatie van het individu alsook hun precieze aandeel daarin. Deze direkte relatie tussen structuur en kwaliteit (fitheid) is vaak veel moeilijker in te zien bij andere toepassingen. Ten tweede, er zijn veel minder praktische problemen die men moet overwinnen om deze Booleaanse problemen te implementeren en te testen. Bij de kunstmatige mier is er behoefte aan een simulatieomgeving waarin de mier op zoek gaat naar voedsel. De mier heeft ook een *wrapper* nodig die het programma opnieuw blijft uitvoeren tot het voedsel gevonden is of tot de tijd verstreken is. Verder hoeven we ook geen rekening te houden met illegale instructies zoals een deling door nul bij het regressieprobleem. Een laatste voordeel is de eindige gekende verzameling van fitness cases waardoor het mogelijk is om alle mogelijke invoer-uitvoer combinaties te testen.

De functieset bevat bij de 11-bit multiplexer een aantal logische operatoren: AND (twee argumenten), OR (twee argumenten) en NOT (één argument). Verder maakt men ook gebruik van een IF-THEN-ELSE constructie met drie argumenten waarvan het eerste argument de conditie voorstelt. Als deze conditie positief wordt geëvalueerd voert men het tweede argument uit, indien vals dan voert men argument drie uit. De verzameling terminalen bestaat uit 11 elementen: drie adreslijnen a_0, a_1, a_2 en 8 datalijnen $d_0, d_1 \ldots d_7$. De decimale voorstelling van de bitstring $a_0a_1a_2$ geeft het nummer aan van de datalijn die zijn inhoud zal kopiëren naar de uitvoer van de multiplexerschakeling.

De multiplexer beschikt over nog twee bijkomende terminalen: TRUE en FALSE. Beide eindknopen worden echter uitsluitend gebruikt door een vereenvoudigingsmodule die in de volgende paragraaf zal besproken worden. Bij het aanmaken van de initiële populatie (of bij het creëren van een nieuwe deelboom bij mutatie) zal het programma geen gebruik maken van beide eindknopen.

De 11-bit multiplexer heeft 2^{11} fitness cases. De fitheid is gedefinieerd als de verhouding tussen het aantal correct voorspelde fitness cases en het totaal aantal fitness cases ($= 2048$). De zoekruimte bij dit probleem is zeer groot. De gezochte 11-bit multiplexer is immers één functie uit $2^{2^{11}}$ ($\approx 10^{616}$) mogelijke Booleaanse functies met 11 invoerwaarden. Dit maakt het probleem bijzonder ongeschikt voor blind zoeken. De correcte oplossing bevat slechts

22 knopen en de boomstructuur vertoont heel wat gelijkenis (Koza 1992). Bovendien zijn de functies AND, OR en NOT overbodig (Fig. 2.6).



Figuur 2.6: *Een correcte oplossing voor het 11-bit multiplexerprobleem.*

2.3.4 Probleemspecifieke parameters

De verschillende probleemspecifieke parameters worden nogmaals opgesomd in Tabel 2.1. Tenzij anders vermeld, zal er in de volgende hoofdstukken steeds gebruikt gemaakt worden van deze instellingen.

2.3.5 Motivatie voor de keuze van de benchmarktoepassingen

Vooreerst merken we op dat het helemaal niet de bedoeling is van dit proefschrift om methoden voor te stellen die goede resultaten produceren enkel en alleen op de gebruikte benchmarktoepassingen. De centrale onderzoeksvraag die beantwoord wordt, is de bestrijding van codegroei in al zijn facetten. De gekozen toepassingen dienen enkel ter illustratie van de aangebrachte verbetering aan de GP methode. De meeste vaststellingen die worden gemaakt, zijn dan ook algemeen geldig.

Verschillende auteurs (o.a. Poli & Langdon (2002)) hebben aangetoond dat zowel de multiplexer alsook de kunstmatige mier beide moeilijke toepassingen zijn voor GP. De kunstmatige mier is een bijzonder interessante toepassing omdat de oplossingenruimte lijkt op een Karst-landschap en veel verwantschap vertoont met reële problemen. Denken we bijvoorbeeld aan robotica toepassingen en optimale controle, zelfs regeltechnische toepassingen zijn mogelijk. De mier is wel oplosbaar met behulp van hill climbing of simulated annealing, maar de prestatie van deze methoden is niet veel beter dan

Tabel 2.1: *Probleemspecifieke parameters*

	Kunstmatige mier	Regressie
fitness case(s)	Santa Fe spoor (89 voedselkorrels)	20 willekeurige punten $\in [-1, 1]$
tijdslimiet	600	/
doelfunctie	alle aanwezige voedsel verzamelen	$x^4 + x^3 + x^2 + x$
fitheid	$\frac{\text{verzameld voedsel}}{89}$	$\frac{1}{1+\sum_{i=1}^{20} \lvert\text{reëel - voorspeld}\rvert}$
afknijpwaarde	/	10^{15}
functieset	IF-FOOD-AHEAD, PROGN2, PROGN3	$+, -, *, /$, $\sin, \cos, \log, \exp$
terminalenset	LEFT, RIGHT, MOVE	$x, ERC \in [-1, 1]$

	11-bit multiplexer
fitness cases	2048
doelfunctie	multiplexerschakeling
fitheid	$\frac{\text{aantal hits}}{2048}$
functieset	AND, OR, NOT, IF-THEN-ELSE
terminalenset	a_0, a_1, a_2 $d_0, d_1 \ldots d_7$

de prestatie van willekeurig zoeken. Bij het multiplexer probleem zijn we op zoek naar één functie uit 10^{616} mogelijke Booleaanse functies met 11 invoerwaarden: een uitdaging voor GP gelet op het stochastische karakter. Deze toepassing is dan weer een representatief voorbeeld voor het ontwerp (en optimalisatie) van meer complexe digitale circuits en schakelingen. Dit heeft ontzettend veel toepassingen in de micro-elektronica. Koza zelf rapporteerde dat GP in staat was digitale schakelingen te ontwerpen die eenvoudiger waren (qua bedrading, omvang, ...) dan de oplossingen bekomen via traditionele technieken (Koza 1994).

Het regressieprobleem is qua moeilijkheidsgraad de eenvoudigste toepassing uit de benchmarksuite. Maar daarom is ze niet minder representatief voor de beoordeling van de aangebrachte verbeteringen aangezien standaard GP zeer vaak de neiging heeft om de oplossing te benaderen (omwille van het grote aantal overbodige functies) i.p.v. ze echt exact voor te stellen en gericht op zoek te gaan naar de meest compacte boomvoorstelling. Hoewel eenvou-

dig, kan het regressieprobleem uitgebreid toegepast worden op verschillende domeinen zoals statistiek (opstellen regressierechten waarvan de vorm niet op voorhand gekend hoeft te zijn), constructie van neurale netwerken, kernel constructie (classificatiemethoden gebaseerd op kernels) en talloze classificatieproblemen.

Elk van de gekozen benchmarktoepassingen is dus een (vereenvoudigde) vertegenwoordiger van een uiteenlopend aantal meer realistische problemen.

Een andere, tevens zeer belangrijke reden waarom we voor deze toepassingen hebben gekozen, is hun hoge frequentie van voorkomen in vergelijkbare studies. Dit heeft als grote voordeel dat er heel wat informatie over de evolutie van goede oplossingen gekend is. Dit liet ons toe om in dit onderzoek resultaten in vergelijkend perspectief te plaatsen en daardoor de resultaten hard te maken, zonder te moeten vervallen in speculatieve beschouwingen. Gelet op de vermelde representativiteit van de drie benchmarkproblemen van een groot aantal toepassingen in een breed domein, verhoogt dit tevens de universaliteit van het onderzoek. Recent werd zelfs een vergelijkende studie van verschillende codegroeibegrenzers gepubliceerd door Luke & Panait (2006). Zij gebruiken dezelfde benchmarktoepassingen.

2.4 De gebruikte software

Gelukkig bestaan er een heleboel kant-en-klare softwarepakketten die vrij beschikbaar zijn (Punch & Zongker 1996, Koza 1992, Yu & Clack 1998, 1997). Bovendien bestaan GP-simulatoren in een waaier van programmeertalen zoals LISP, C, C++, Java, en zelfs een rechtsreeks in de hardware (Draxton Labs) en via Xilinx FPGA's (Martin 2002). In dit proefschrift werd gekozen voor een implementatie die snel werkt en op een efficiënte manier gebruik maakt van het geheugen. Met efficiënt bedoelen we dat de software zo weinig mogelijk geheugen gebruikt om de boomstructuren van de individuen op te slaan. Beide eisen gecombineerd met mijn persoonlijke programmeerervaring in de programmeertaal C, hebben geleid tot de keuze voor het softwarepakket lil-gp (Punch & Zongker 1996).

In lil-gp wordt een boom voorgesteld als een rij van dataobjecten. Een dataobject kan verwijzen naar ofwel een functie, ofwel een willekeurige constante ofwel een geheel getal. Belangrijk hierbij is dat de boom in prefixnotatie wordt opgeslagen. Het eerste dataobject is steeds een functie. Wanneer de functie een willekeurige constante is dan wordt de waarde van deze constante opgeslagen in het dataobject dat erop volgt. Als de functie een verande-

ring van de volgorde van uitvoering van de verschillende knopen kan teweeg brengen (denk bijvoorbeeld aan een IF-FOOD-AHEAD of IF-THEN-ELSE constructie) dan wordt er een extra dataobject voor elk van de kinderen van deze functie geplaatst. Het dataobject bevat een gehele waarde die de lengte van het kind (in het bijzonder het aantal objecten) voorstelt. Wanneer een deelboom niet wordt uitgevoerd dan gebruikt de evaluatiefunctie deze waarde om het kind over te slaan. We geven een voorbeeld. Beschouwen we de volgende uitdrukking: $(+\ x\ (iflte\ x\ (\times\ 0.34\ x)\ 0.56))$. In lil-gp wordt deze uitdrukking als volgt voorgesteld:

$+$	wijzer naar de functie $+$
x	wijzer naar de functie x (eigenlijk een terminaal)
$iflte$	wijzer naar functie $iflte$
1	eerste argument van $iflte$ heeft één lnode nodig
x	
4	tweede argument van $iflte$ heeft 4 lnodes nodig
$\times$	wijzer naar de functie $\times$
C	wijzer naar een constante
0.34	de waarde van de constante
x	
2	derde argument van $iflte$ heeft 2 lnodes nodig
C	
0.56	

Hoewel deze voorstelling afwijkt van de klassieke gelinkte lijst, heeft ze toch een aantal voordelen. Zo wordt er minder geheugen verspild door het opslaan van de knopen in de volgorde waarin ze worden gebruikt. Een tweede belangrijk voordeel is de snelle evaluatie van de boomstructuur. Om dit aan te tonen beschouwen we het gebruik van de kruisingsoperator. Wanneer we gebruik zouden maken van gelinkte lijsten zal kruising slechts twee wijzers van richting moeten veranderen. Dit lijkt zeer eenvoudig en bijzonder snel maar naarmate de evolutie vordert, zullen de knopen van een boom verspreid raken over de beschikbare geheugenruimte. Bij een systeem met virtueel geheugen zal de boomstructuur van een individu worden uitgesmeerd over verschillende pagina's die continu in en uit het geheugen worden geladen waardoor de snelheid van de evaluatie zeer sterk afneemt. In lil-gp wordt de implementatie van de kruisingsoperator wat meer gecompliceerd maar blijft het genoom van het individu wel samengepakt in één aaneensluitend stuk geheugen dat perfect past op één (of misschien uitzonderlijk op twee) geheugenpagina's.

2.4.1 Instellingen van de GP-simulator

Het volledige GP systeem heeft een zeer groot aantal parameters die een bepaalde (soms nog ongekende) invloed uitoefenen op de evolutie. De ene parameter is uiteraard belangrijker dan de andere. Zo is de keuze van het selectiemechanisme of de instelling van de probabiliteit waarmee kruising wordt toegepast belangrijker dan de instelling van de initiële diepte van de individuen uit de startpopulatie. Uit praktische overwegingen is het echter onmogelijk om alle waarden en combinaties voor alle parameters van het systeem uit te proberen (rekentijd). Daarom baseren we ons vaak op instellingen die door een groot aantal auteurs worden gebruikt. Voor sommige parameters, waarvan we de exacte invloed op het algoritme niet op triviale wijze kunnen afleiden, werden aanvullende experimenten gedaan met andere parameterwaarden. De resultaten hiervan worden niet steeds in dit proefschrift vermeld omwille van twee redenen: ofwel zijn de resultaten niet wezenlijk verschillend van de gebruikte instellingen zoals beschreven in dit werk, ofwel heeft de parameter geen invloed op de vergelijking tussen het standaard genetisch programmeren systeem en het GP systeem met lokale optimalisatie[3]. De lezer overladen met grafieken is eveneens niet de bedoeling van deze studie.

In wat volgt geven we een kort overzicht van de verschillende parameters en hun (initiële) waarde. Tenzij anders vermeld worden steeds de volgende instellingen hiervoor gebruikt.

- Algemene instellingen:
  - aantal onafhankelijke simulaties per experiment (telkens met een andere willekeurige *seed*): 100
  - aantal generaties per simulatie: 100 (generatie 0 niet meegeteld)
  - aantal individuen in de populatie: 500. Er wordt vaak beweerd dat grotere populaties ($\geq$ 500) tot betere resultaten leiden dan kleinere populaties (Koza 1992, Poli & Langdon 2002, Poli 2003). Toch is een aantal van 500 individuen het maximum om de duurtijd van de simulaties beperkt te houden.
- Voor de invulling van de initiële populatie:
  - methode: half-and-half

[3]Er kan echter wel een verschil optreden in bepaalde opgemeten parameters maar vaak hebben deze opgemeten waarden betrekking op andere parameters, dan dewelke interessant zijn wanneer we spreken over het bestrijden van codegroei.

  - diepte-helling: tussen 2 en 6.

- Voor de voortplanting:
  - alle operatoren (behalve reproductie) gebruiken tornooiselectie (met 4 deelnemers)
  - kruising:
    * probabiliteit: 99%
    * selectieprobabiliteit van interne knopen: 0.9 (0.1 voor eindknopen)
  - reproductie:
    * probabiliteit: 1%
    * selectieprocedure:5 beste individuen (om convergentie te vrijwaren)
  - geen mutatie[4]
  - geen lokale optimalisatie bij standaard genetisch programmeren (zie hoofdstuk 4)
- Geen beperkingen van de boomstructuur.

Uiteraard bestaan er nog een heleboel andere parameters die echter geen belang hebben in dit proefschrift. Een volledig overzicht kan men vinden in (Punch & Zongker 1996).

2.4.2 De uitvoerbestanden

Tijdens elke simulatie schrijft het programma gegevens naar verschillende uitvoerbestanden. Per generatie worden de volgende parameters weergegeven: de minimum, maximum en gemiddelde waarden van de fitheid, van de fitheid van de deelbomen, van de absolute afmetingen (boomgrootte en boomdiepte), en van de functionele afmetingen, verder de fitheid en alle afmetingen van het beste individu van de generatie alsook van de beste deelboom in de populatie, de pseudo-isomorfe diversiteitsmaten (originele definitie en functionele versie), de entropie, het aantal unieke fitheidswaarden, en het aantal unieke (functionele) boomstructuren. Verder houden we ook een gedetailleerd overzicht bij van een aantal parameters in verband met de werking van de verschillende operatoren.

[4]Mutatie wordt niet vaak gebruikt in GP. Dit is te wijten aan een aantal factoren zoals beschreven in paragraaf A.6.4.

Merk tenslotte op dat aangezien elk experiment 100 simulaties telt, de grafieken in volgende hoofdstukken telkens uitgemiddeld zijn over 100 simulaties. Op deze manier heeft men voldoende herhalingen van het experiment om verschillende statistische methoden toe te passen.

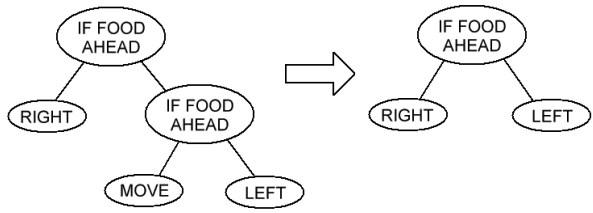
2.4.3 Uitbreiding van de simulator: de simplificatiemodule

Voor de bepaling van de hoeveelheid functionele code in een boomstructuur werd een *expression simplifier* ontwikkeld (Wyns *et al.* 2006). Dit houdt in dat een operator op een boom wordt toegepast die alle niet-functionele code verwijdert. Dit gebeurt echter zonder de boom te evalueren. Enkel op basis van de structuur van de boom en de betekenis van de knoopfuncties wordt bepaald welke knopen en deelbomen niet uitgevoerd zullen worden. Hiervoor is het nodig dat voor elke toepassing een aantal herleidingsregels opgesteld wordt. Voorbeelden van zulke regels zijn:

- voor het regressieprobleem:
  - als één van de kinderen van een vermenigvuldigingsknoop een constante met waarde nul is, dan kan deze deelboom door de constante nul vervangen worden
  - als één van de kinderen van een vermenigvuldigingsknoop een constante met waarde één is, dan kan deze deelboom door het andere kind worden vervangen. Voorbeeld: $S \times 1 = S$. Hetzelfde geldt voor een deling waarbij het tweede operand gelijk is aan één.
  - als beide kinderen van een aftrekking indentiek zijn, dan kan deze deelboom door de constante nul vervangen worden. Een analoge reductie vindt plaats met deelbomen
  - bij operatoren die enkel inwerken op constanten wordt het resultaat berekend en de deelboom vervangen door dit resultaat. Voorbeeld: $\log(\exp((3.65 \times 8.54)) - 4.50) = 11.58$.
- voor de kunstmatige mier:
  - als het eerste of het tweede kind van een IF-FOOD-AHEAD knoop ook een IF-FOOD-AHEAD knoop is, dan zal enkel de eerste of respectievelijk tweede deelboom van de onderste IF-FOOD-AHEAD knoop uitgevoerd worden. Een voorbeeld hiervan is te zien in Figuur 2.7

  - opeenvolgende LEFT en RIGHT (of in omgekeerde volgorde) eindknopen worden uit de boomstructuur verwijderd. Voorbeeld: (PROGN3 (LEFT) (RIGHT) (LEFT)) wordt gereduceerd tot (LEFT).
  - als beide acties van een IF-FOOD-AHEAD statement identiek zijn, wordt de uitdrukking gereduceerd tot enkel de actie overblijft. Voorbeeld: (IF-FOOD-AHEAD (MOVE) (MOVE)) = (MOVE).

- voor de Booleaanse 11-bit multiplexer:
  - stel S is een deelboom of eindknoop dan is (AND (s) (s)) of (OR (s) (s)) steeds gelijk aan (s)
  - de negatie van de negatie van een deelboom/eindknoop is gelijk aan die deelboom of eindknoop. Voorbeeld: (NOT (NOT (s))) = (s).
  - Een IF-THEN-ELSE uitdrukking waarbij beide acties (indien waar en indien vals) gelijk zijn, wordt gereduceerd tot de actie zelf. Voorbeeld: (IF-THEN-ELSE (CONDITION) (ACTION) (ACTION)) wordt gereduceerd tot (ACTION).



Figuur 2.7: *Voorbeeld van een herleidingsregel; als het tweede kind van een* IF-FOOD-AHEAD *knoop ook een* IF-FOOD-AHEAD *knoop is, dan zal enkel de tweede deelboom van de onderste* IF-FOOD-AHEAD *knoop uitgevoerd worden.*

Merk op dat men deze reductie nog verder kan doordrijven bij het regressieprobleem. Daar kan men bijvoorbeeld uitdrukkingen zoals $(2x - 1) - (x - 2)$ vervangen door $(x + 1)$. Dergelijk optimalisaties werden echter niet doorgevoerd aangezien deze reductieregels geen betrekking hebben op niet-functionele code. Alle code uit de eerste uidrukking is wel degelijk functioneel, alleen is ze niet geoptimaliseerd. Bovendien vereist de controle van de

boomstructuur op dergelijke gevallen computationeel zeer veel extra rekenkracht. Ook de commutatieve en associatieve eigenschappen van bepaalde functies werden niet in rekening gebracht, opnieuw om de vereenvoudigingsmodule niet te rekenintensief te maken. De getoonde hoeveelheid functionele code is dus steeds een bovengrens.

De implementatie van deze operator vormt een belangrijke uitbreiding van de kernel. Deze module omhelst drie onderdelen. Ten eerste is er het algemene raamwerk van de herleiding of simplificatie. Bij het simplificeren van een boom wordt niet de boom zelf aangepast, maar een kopie ervan, wat uiteraard noodzakelijk is om de evolutie niet te beïnvloeden. Het simplificatieraamwerk doet dus onder andere het geheugenonderhoud om deze kopie aan te maken en te vernietigen. Verder bevat dit raamwerk de eigenlijke simplificatiefunctie, die vanuit de kernel kan aangeroepen worden en die de gesimplificeerde boom produceert door de herleidingsregels achtereenvolgens (recursief) op alle knopen toe te passen.

Het tweede onderdeel van de module bestaat uit een groot aantal hulpfuncties die door de rest van de module gebruikt worden voor het manipuleren van de boom die herleid wordt. Ook hier wordt geen expliciet gelinkte boom gebruikt, maar dezelfde array-structuur als in de rest van lil-gp. Knopen worden gewist door ze in de array te vervangen door een ongeldige waarde. Aan het einde van de herleiding worden ze pas effectief uit de array verwijderd en wordt hun geheugen vrijgegeven.

Het derde onderdeel tenslotte voorziet een aantal functies die gebruikt kunnen worden om de herleidingsregels effectief te implementeren, bijvoorbeeld functies die een hele deelboom wissen, of die een deelboom die steeds een constante waarde oplevert vervangen door een ERC met die waarde.

Aangezien de herleidingsregels probleemspecifiek zijn, moeten zij in de toepassingscode geïmplementeerd worden en niet in de kernel. Daarvoor is een extra functieprototype voorzien in die toepassingscode. De implementatie van de regels vereist ook nog enig programmeerwerk, uiteraard meer naarmate de regels ingewikkelder worden, maar dankzij het voorhanden zijn van de functies uit het derde onderdeel van de herleidingsmodule blijft de code hier conceptueel zeer eenvoudig.

2.5 Besluiten

We overlopen tot slot enkele belangrijke kenmerken van genetisch programmeren. In Koza's eerste boek (1992) werden een groot aantal verschillende

toepassingen succesvol opgelost met behulp van genetisch programmeren. Zo gebruikte hij alle bovenstaande standaardtoepassingen maar ook op verscheidene andere domeinen zoals optimale controle, wiskundige problemen (afgeleiden, integralen...), regeltechnische problemen (balanceren van een bezem, achterwaarts parkeren van een vrachtwagen), problemen uit *artificial life* (A-Life) zoals het computerspel Pac-Man, cellulaire automata, ... werden uitstekende resultaten behaald. We kunnen dus stellen dat genetisch programmeren een generiek en breed toepasbaar paradigma is.

De invoerwaarden komen meestal rechtstreeks uit het probleemdomein. Genetisch programmeren gebruikt een natuurlijke voorstelling van dit domein. Er is geen nood aan *pre-processen* of voorverwerking van de invoer zoals dit vaak het geval is bij neurale netwerken, genetische algoritmen en andere algoritmen voor machinaal leren. Bovendien is er ook een sterke gelijkenis tussen de voorstelling van een individu als een boomstructuur en het probleemdomein. Een direkt gevolg van deze gelijkenissen is dat *post-processing* of naverwerking overbodig is. De geëvolueerde oplossingen zijn zeer eenvoudig te begrijpen. Zelfs al zou een dergelijke techniek nodig blijken, dan is deze vaak zeer eenvoudig.

Zoals reeds werd aangehaald in de inleiding kunnen evolutionaire algoritmen in het algemeen vrij goed overweg met onnauwkeurigheden in de invoerwaarden (ruis).

Aangezien het resultaat van genetisch programmeren een computerprogramma is, betekent dit dat dit programma onmiddellijk kan uitgevoerd worden op een andere computer. Het gegenereerde programma kan vaak ook op eenvoudige wijze gereduceerd (vereenvoudiging) en/of geoptimaliseerd worden.

Genetisch programmeren gebruikt weinig of geen voorkennis om de exacte grootte en diepte van de uiteindelijke oplossing te bepalen. Terwijl dit bij onder andere regelgebaseerde klassificatiesystemen wel vaak het geval is. Ook bij een eindige toestandsautomaat moet men het maximum aantal toestanden definieren. Bij een neuraal netwerk gaat het aantal instellingen nog verder: het aantal verborgen lagen, het aantal neuronen, het aantal verbindingen... Bij genetisch programmeren is de enige nodige informatie de samenstelling van de verzameling functies en eindknopen. Deze informatie is echter ook bij andere technieken uit machinaal leren noodzakelijk.

Het is ook mogelijk om een historiek op te stellen van de correcte oplossing. We kunnen met andere woorden nagaan hoe de oplossing precies geconstrueerd werd en welke componenten precies hebben bijgedragen tot het vinden van een correcte oplossing.

Vaak zijn genetische methoden in staat om incrementeel en op robuuste wijze een oplossing aan te passen zodat het verschillende situaties goed beheerst.

Genetische methoden lenen zich uitstekend om in parallel uitgevoerd te worden waardoor een quasi lineaire versnelling kan worden gerealiseerd.

. . . om het met de woorden van de grondlegger van genetisch programmeren te zeggen (Koza 1992):

> ". . . In conclusion, genetic programming is a robust and efficient paradigm for discovering computer programs using the expressiveness of symbolic representation. . . "

Een inleiding tot genetisch programmeren is te vinden in (Sette & Boullart 2001). De toepassing van genetisch programmeren voor de automatische generatie van beslissingsregels voor een vezel-garen productieproces is te vinden in (Sette *et al.* 2004).

Hoofdstuk 3

Codegroei

In dit hoofdstuk wordt het begrip codegroei gedefinieerd bij genetisch programmeren. Aan de hand van een uitgewerkt voorbeeld van codegroei bij de kunstmatige mier wordt duidelijk gemaakt welke nadelen verbonden zijn aan de explosieve groei van de boomstructuren in genetisch programmeren. Vervolgens geven we een kort overzicht van de verschillende principiële verklaringen voor codegroei, gevolgd door een uitgebreid overzicht van de verschillende technieken om codegroei te bestrijden.

3.1 Codegroei en zijn gevolgen

3.1.1 Codegroei en structuren met variabele lengte

In het voorgaande hoofdstuk werd er kort vermeld dat genetisch programmeren vaak gebruik maakt van boomstructuren om een individu voor te stellen. Ook voor genetisch programmeren bestaan er, behoudens de boomstructuur, verschillende andere coderingsstrategieën met variabele lengte. Stoffel & Spector (1996), Nordin (1997) en Banzhaf *et al.* (1997) gebruiken een lijst met machine instructies. Teller (1998) gebruikt een systeem met cyclische oproepgrafen. Sommige auteurs gebruiken ook gerichte acyclische grafen om de computerprogramma's voor te stellen (Keijzer 1996, Handley 1994). Andere voorbeelden van het creëren van grafen zijn de *push down* automaat van Zomorodian (1994), en programma's gebaseerd op grafen (Teller 1996). Een andere vaak gebruikte techniek is cellulaire codering (Gruau 1992) waarbij de boom een programma voorstelt dat een graaf opbouwt (vaak gebruikt voor een neuraal netwerk).

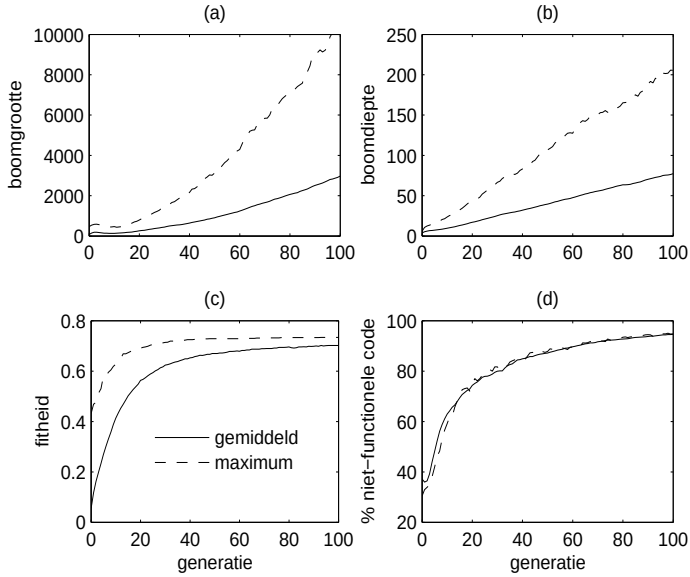
Ongeacht welke voorstellingswijze men ook kiest, is er steeds een belangrijk nadeel verbonden aan structuren met variabele lengte. Ze zijn vatbaar voor *bloat* of codegroei. Codegroei wordt vaak gedefinieerd als een niet controleerbare groei van het genoom gedurende de evolutie, onafhankelijk van een verhoging van de fitheid. Vooral genetisch programmeren met boom gebaseerde voorstellingen heeft erg te lijden onder codegroei (Koza 1992, Bickle & Thiele 1994, McPhee & Miller 1995, Angeline 1998, Langdon *et al.* 1999).

3.1.2 De gevolgen van codegroei: een voorbeeld

Wanneer de omvang van een boomstructuur zeer snel toeneemt in de loop van de evolutie, kunnen we onmiddellijk een aantal nadelen die hieraan verbonden zijn, bedenken. We verduidelijken deze nadelen aan de hand van een klein voorbeeld (kunstmatige mier). We gebruiken hier tornooiselectie met zeven individuen en 100% kruising (geen reproductie). We voeren ook slechts 50 onafhankelijke simulaties uit in plaats van de standaard 100. De andere instellingen worden overgenomen uit 2.3.4.

Figuur 3.1 geeft een overzicht (per generatie) van het gemiddeld en maximum aantal knopen (a), de gemiddelde en maximum diepte (b), de gemiddelde en maximum fitheid (c) en het procentuele aandeel van niet-functionele code in de boomstructuur (ten opzichte van het gemiddeld en maximum aantal knopen, d).

Boomstructuren met een groot aantal knopen (of met een grote diepte) zullen meer geheugenruimte in beslag nemen (Fig. 3.1a,b). Hierdoor zal het aantal geheugentoegangen drastisch oplopen waardoor de benodigde processortijd zal stijgen. Verder introduceren meer knopen ook meer functie-evaluaties wat opnieuw resulteert in een grotere vraag naar processortijd. Beide nadelen (geheugen en processortijd) zorgen ervoor dat men genetisch programmeren niet zomaar voor eender welke toepassing kan gebruiken (Banzhaf & Langdon 2002). Vooral toepassingen die vragen om een uitgebreide zoekopdracht zijn praktisch moeilijk haalbaar. Aangezien niet elke onderzoeksgroep beschikt over een cluster van rekenservers (Genetic Programming Inc., 1000 computers in een Beowulf cluster), of over een sterk geparallelliseerde versie (Chong 1999, Folino *et al.* 2001, Fernandez *et al.* 2005), of over een geheel of gedeeltelijke hardware implementatie (Martin 2002, Koza *et al.* 1998, Heywood & Zincir-Heywood 2000) is men genoodzaakt om te werken met beperkte populatiegroottes en een beperkt aantal generaties. De uitvoering van bovenstaand experiment op een x86 machine met Mandrake Linux als besturingssysteem en geen andere werkopdracht (processor hoeft enkel het GP programma uit te voeren, naast een zeer beperkte werklast afkomstig van



Figuur 3.1: *Codegroei bij de kunstmatige mier. De volle lijnen duiden steeds de gemiddelden aan terwijl de onderbroken lijnen de maxima voorstellen.*

het besturingssysteem) duurde ongeveer 180 minuten (voor 50 simulaties en slechts 50 generaties). Daarbij werd gemiddeld één gigabyte (GB) geheugen gebruikt. Tussen de verschillende generaties in werd er wel een deel van het gealloceerde geheugen opnieuw vrijgemaakt. Uiteraard willen deze cijfers enkel een idee geven van hoe rekenintensief genetisch programmeren is. De bekomen resultaten zijn dan ook afhankelijk van een aantal factoren zoals: de gekozen programmeertaal, compilerinstellingen, de machine waarop het programma wordt uitgevoerd, de werklast van deze machine, de parameterinstellingen van het algoritme, etc.

Zoals we ook kunnen zien op Figuur 3.1c, treedt er weinig of geen verbetering meer op van de maximum fitheid na generatie 20. De toename aan niet-functionele code hindert duidelijk de positieve evolutie van (maximum) fitheid. In plaats van continu betere individuen te ontdekken, stagneert maximum fitheid.

Een derde probleem is de leesbaarheid van de bekomen oplossing. Onderstaande code toont een deel van de boomstructuur van de oplossing uit simulatie nummer vijf (individu met de hoogste fitheidswaarde op generatie 50). De boomstructuur bevat maar liefst 4460 knopen en heeft een diepte van 111! Er wordt slechts een gedeelte van de boomstructuur getoond wegens plaatsgebrek. IFA is de verkorte notatie van de sensor IF-FOOD-AHEAD.

```
 (IFA (IFA (progn3 (IFA move move)
                   (progn2 right left) right)        (1)
                   (IFA (IFA (progn2 move move)      (2)
                             (progn2 move move))     (3)
... knip ...
                (progn3 (progn3 left move move)
                        (IFA left move)
                        (progn3 right move move)))
```

Door hun immense omvang zijn dergelijke oplossingen onhandelbaar. Men kan nauwelijks of geen structuur ontdekken binnen de boom.

Grote, omvangrijke structuren hebben vaak de neiging om meer code op te nemen dan functioneel nodig (zie Fig. 3.1d). Vaak is deze code zeer specifiek geschikt voor het oplossen van de getoonde verzameling fitness cases (Banzhaf *et al.* 1997, Rosca 1996, Kinnear 1993*b*). De fitheid van een dergelijk individu zal dan ook dalen wanneer men het evalueert op een andere (gelijkaardige) verzameling fitness cases.

We geven een aantal voorbeelden van niet-functionele code toegepast op de bovenstaande partiële boom. In coderegel (1) is het duidelijk dat de opeenvolging van de terminalen RIGHT en LEFT gereduceerd kan worden (de constructie (PROGN2 RIGHT LEFT) verdwijnt volledig). De tweede IFA constructie uit regel twee heeft twee identieke actie gedeelten (actie indien voedsel aanwezig is gelijk aan actie indien geen voedsel aanwezig). Daardoor kunnen we ook dit statement reduceren tot (PROGN2 MOVE MOVE). Er is echter ook een andere manier om regels twee en drie te vereenvoudigen. Twee opeenvolgende IFA functies kunnen immers gereduceerd worden tot één aangezien de conditie bij de tweede IFA functie zal falen als de conditie bij de eerste IFA functie reeds faalde (en ook indien er wel voedsel aanwezig is). Met behulp van de simplificatiemodule uit paragraaf 2.4.3 kunnen we het bovenstaande voorbeeld (weliswaar de volledige boomstructuur) herleiden tot:

```
(IFA (progn2 move right)
               (progn3 (progn3 left move move)
                       (IFA left move)
                       (progn3 right move move)))
```

Deze boom heeft slechts 16 knopen en een diepte van 3 (in vergelijking met 4460 knopen en een diepte van 111)! Toch is het gedrag van deze boomstructuur volledig equivalent met de grotere oplossing die standaard genetisch programmeren heeft gegenereerd.

Als we Figuur 3.1a aandachtig bekijken, stellen we vast dat de toename in boomgrootte probleemafhankelijk is. Langdon (2000) toonde aan dat voor de meeste discrete probleemdefinities de boomgrootte stijgt volgens $O(generaties^{1.2-1.5})$. Bij continue fitheidsruimten stijgt codegroei volgens $O(generaties^2)$.

3.2 Terminologie

Vooraleer we de verschillende verklaringen voor codegroei kort toelichten, definiëren we eerst enkele types code die vaak voorkomen in de boomstructuur van een kandidaat-oplossing. De indeling van programmacode in de onderstaande klassen is misschien niet altijd even voor de hand liggend en/of logisch. We hanteren deze indeling daarom enkel bij de bespreking van de verschillende theorieën die codegroei trachten te verklaren. Op deze manier vermijden we dat de auteurs van elke van deze theorieën verschillende types

codefragmenten voor ogen hebben. Een vollediger overzicht (met voorbeelden) is te vinden in (Luke 2000*b*).

Na de korte beschrijving van de verschillende theorieën voor codegroei zullen we echter enkel onderscheid maken tussen functionele code en niet-functionele code (introns). De omvang van de functionele code van een programmaboom wordt bepaald zoals beschreven in 2.4.3.

Intronen Intronen worden beschouwd als stukken programmacode uit het genoom die niet wezenlijk bijdragen tot de oplossing (Angeline 1994). Bijgevolg kan men deze stukken code verwijderen uit de boomstructuur. De term intronen omvat niet-levensvatbare code en code die nog verder geoptimaliseerd (vereenvoudigd) kan worden. Intronen staan in de literatuur ook bekend onder de noemer niet-functionele code.

Niet-levensvatbare code Niet-levensvatbare code is, zoals de naam reeds doet vermoeden, code die niet (nooit) kan bijdragen tot de functionaliteit van een individu. Wijzigingen die in deze codefragmenten worden aangebracht door bijvoorbeeld kruising of mutatie, veranderen in geen geval de fitheid van het individu. Een aantal voorbeelden van dit type code zijn: (*(- x x) *niet-levensvatbaar*), (or (or d_0 (not d_0)) *niet-levensvatbaar*), (if (and d_1 (not d_1)) *altijd levensvatbaar*), (if *niet-levensvatbaar* d_0 d_0), het niet verwijzen naar een automatisch gedefinieerde functie...

In de literatuur worden ook andere benamingen voor niet-levensvatbare code gebruikt: absolute intronen (Banzhaf *et al.* 1997), overbodige knopen (Blickle 1996*b*, Bickle & Thiele 1994), niet-effectieve code (Rosca 1996), 'type 1' en 'type 2' intronen (Nordin *et al.* 1995) en syntactische intronen (Angeline 1998).

Actieve code Actieve code is code die steeds wordt uitgevoerd en waar er geen nood is aan optimalisatie. Actieve code wordt ook vaak functionele code genoemd.

3.3 Wat veroorzaakt codegroei?

Er bestaan verschillende theorieën om codegroei te verklaren. In deze paragraaf bespreken we kort de meest gekende hypothesen.

3.3.1 Hitchhiking

In deze theorie staat de verspreiding van intronen via zogenaamde bouwblokken centraal. Deze bouwblokken zijn vaak kleine codefragmenten die een belangrijke functie vervullen. Tackett (1994) beweert dat intronen die verbonden zijn met deze bouwblokken (Tackett zelf sprak over belangrijke actieve code), worden verspreid door kruising en andere operatoren op voorwaarde dat de genetische operator de actieve code behoudt. Hij merkte op dat wanneer het systeem selectiever werd (verhoging van de selectiedruk), de bouwblokken belangrijker werden en zich sneller gingen verspreiden doorheen de populatie. Intronen die zich vasthechten aan deze bouwblokken zullen bijgevolg eveneens sneller verspreid worden doorheen de populatie waardoor de gemiddelde grootte stijgt.

3.3.2 Bescherming tegen structurele veranderingen

Kruising is de belangrijkste operator bij genetisch programmeren. Impliciet gaat men er vanuit dat bij kruising waardevolle structuren uitgewisseld worden. Dit is echter niet steeds het geval. Verschillende auteurs stellen vast dat kruising een eerder destructieve operator is (Luke 2000*b*, Langdon & Poli 1997*a*, Nordin & Banzhaf 1995). Kruising houdt immers geen rekening met de volgorde en positie van de knopen (diepte) en met hun onderlinge relaties[1]. Bestaande relaties worden afgebroken en nieuwe afhankelijkheden tussen de verschillende knopen worden mogelijks geïntroduceerd. Wellicht speelt dit een belangrijke rol bij de effectiviteit van kruising en is daarom de kans om een beter individu te genereren relatief klein (Luke 2000*b*).

De destructiviteit van kruising ligt aan de basis van een andere mogelijke verklaring voor codegroei. Codegroei zou, volgens deze theorie, immers het enige redmiddel zijn om deze destructieve effecten tegen te gaan. Indien een kruising optreedt in niet-levensvatbare code, dan is de kans dat deze kruising de fitheid van het individu negatief zal beïnvloeden onbestaande. Dus hoe meer niet-levensvatbare code een individu bevat, hoe beter het zich kan beschermen!

Deze theorie verwijst vaak enkel naar kruising. Nochtans is ze toepasbaar op alle operatoren die structurele wijzigingen aanbrengen aan het genoom van het individu (zoals bijvoorbeeld deelboom mutatie). Deze theorie werd zeer vaak beschreven in talloze artikelen (Bickle & Thiele 1994, McPhee & Miller

[1] Dit probleem is beter gekend als het *linkage* probleem (Mitchell 1996).

1995, Nordin *et al.* 1995, Rosca 1996, Langdon *et al.* 1999, Banzhaf *et al.* 1997, Andre & Teller 1996)[2].

3.3.3 Removal bias

Deze verklaring bewandelt grotendeels hetzelfde pad als de vorige theorie. Ook hier steunt men op het feit dat bomen met grote hoeveelheden niet-levensvatbare code een grotere kans hebben om destructieve kruisingen te overleven. Men maakt echter nog een aantal bijkomende veronderstellingen. Zo gaat men ervan uit dat knopen die verder van de beginknoop liggen (verder weg van de 'root') vaak intronen zijn. Deze veronderstelling is geldig voor niet-levensvatbare code (Soule & Foster 1998*b*). Er zijn (nog) geen empirische experimenten gekend die deze veronderstelling aantonen voor intronen. De resultaten van Igel & Chellapilla (1999) zorgen evenwel voor een extra duwtje in de rug. Zij toonden aan dat code verder verwijderd van de beginknoop, minder invloed uitoefent op de teruggeefwaarde van het programma.

Veronderstellen we bovendien dat niet-levensvatbare code zich concentreert rond de eindknopen van de boom, dan zal het verwijderen van een 'kleine' deelboom een grotere kans hebben om enkel de niet-levensvatbare code aan te tasten. Het verwijderen van een grotere deelboom zal eerder meer kans hebben om ook (delen) van de levensvatbare code te wijzigen. Wanneer we de redenering omdraaien —wanneer er dus een deelboom wordt toegevoegd ergens in niet-levensvatbare code— zal deze deelboom geen invloed uitoefenen, ongeacht de grootte van de toegevoegde deelboom. Deze theorie suggereert dat het verwijderen van kleine bomen een voorkeursbehandeling krijgt terwijl het verwijderen van een grotere deelboom wordt benadeeld. Bij het toevoegen van een deelboom is er geen corresponderende voorkeursbehandeling waardoor codegroei in de hand wordt gewerkt.

3.3.4 Spreiding van de fitheidswaarden

Deze hypothese is gebaseerd op de verdeling van goede boomstructuren in de zoekruimte en werd gesuggereerd door (Langdon & Poli 1998*a*, 1997*a*, Langdon *et al.* 1999, Langdon 1998*b*).

[2]Vele auteurs veralgemenen deze theorie tot intronen (in plaats van enkel niet-levensvatbare code). Toch heeft men weinig of geen experimenteel bewijs dat deze hypothese ook geldig is voor intronen. De meeste experimenten bestuderen immers enkel niet-levensvatbare code.

Experimenteel onderzoek heeft aangetoond dat, gegeven een programma met fitheid X en boomgrootte Y, er veel meer programma's bestaan met fitheid gelijk aan X en met afmetingen groter dan Y dan dat er programma's bestaan met diezelfde fitheid X maar kleiner dan Y. Het bestaan van een groter aantal grote boomstructuren kan men gedeeltelijk verhalen op intronen. Immers, door toevoeging van intronen kan men oneindig veel varianten van een programma ontwikkelen die groter zijn dan Y. Dus grotere programmabomen komen eenvoudigweg vaker voor.

De algemene redenering achter deze theorie is dat een stochastische zoekmachine zoals genetisch programmeren, de neiging heeft om de vaakst voorkomende programma's te selecteren. En die meest voorkomende programma's voor een bepaalde fitheidswaarde zijn eenmaal groter.

3.3.5 De diepte van het kruisingspunt

Luke (2003) berekende correlaties tussen de overlevingskans van het individu en een aantal afgeleide parameters zoals de diepte van het kruisingspunt, de boomgrootte van de ouder, de grootte van de verwijderde boom en de grootte van de toegevoegde boom. De overlevingskans van een individu werd gedefinieerd als het aantal keren dat een individu in de daarop volgende generaties door kruising wordt geselecteerd als ouder. Zijn bevindingen toonden een zeer sterke positieve correlatie met de diepte van het kruisingspunt. Het resulterende causale model is te zien in Figuur 3.2.

Figuur 3.2: *Een overzicht van de verschillende relaties in genetisch programmeren (uit Luke (2003)).*

Dieper gelegen knopen (en hun corresponderende deelbomen) oefenen een minder grote invloed uit op de functionaliteit van een individu in vergelijking met knopen die dichter bij de beginknoop liggen. De overlevingskans van een individu is dus groter wanneer kruising dieper gelegen knopen kiest.

Als gevolg daarvan worden vaker grotere individuen gekozen door kruising (aangezien de kruisingspunten daar dieper liggen). Bovendien zijn de verwijderde deelbomen met als beginknoop de dieper gelegen kruisingspunten meestal kleiner, in tegenstelling tot toegevoegde deelbomen die geen dergelijke bias vertonen (dit is een veralgemening van de removal bias theorie). De boomgrootte is op zichzelf een bescherming tegen structuur wijzigende genetische operatoren.

Analoge resultaten werden behaald door (Streeter 2003). Hij splitste codegroei op in twee afzonderlijke termen. De eerste term geeft aan in welke mate goede individuen groter zijn dan hun ouders terwijl de tweede term aangeeft in welke mate de ouders van goede individuen groter zijn dan de verzameling van alle ouders. Streeter definieerde verder ook nog de begrippen kwetsbaarheid (de mediaan van de verschillen in fitheid tussen individuen) en robuustheid[3] (=kwetsbaarheid $\times -1$). Streeter concludeerde dat GP op zoek is naar individuen met een hoge graad van robuustheid. Deze robuustheid kan bekomen worden door het toevoegen van niet-levensvatbare code en de diepte van de knopen (minder kwetsbaar). Grotere bomen zijn dus gemiddeld genomen minder kwetsbaar (dus meer robuust) dan kleinere exemplaren. Codegroei is dus een uiterlijk kenmerk van de zoektocht van GP naar robuuste individuen.

3.3.6 De relatie met de probleemdefinitie

Recent werd onderzocht of er een verband bestaat tussen codegroei en de probleemdefinitie (Burke *et al.* 2004). De auteurs hebben experimenteel aangetoond dat er een causaal verband bestaat tussen een toenemende moeilijkheidsgraad van de probleemdefinitie en de diversiteit[4] in de populatie. Dit causale verband leidt tot een 'verhoogde' codegroei.

De auteurs gebruiken een aantal wiskundige functies met toenemende moeilijkheidsgraad om hun hypothese te staven. Uit hun experimenten blijkt dat bij toenemende complexiteit van de probleemdefinitie ook de boomgrootte en boomdiepte toenemen alsook de tijd nodig om een goede oplossing te vinden. Nochtans is de stijging van de boomgrootte en -diepte niet echt noodzakelijk aangezien voor alle probleemdefinities compacte oplossingen bestaan.

[3] Robuustheid wordt ook vaak gebruikt om de tolerantie ten opzichte van ruis, e.d. uit te drukken. In deze context spreekt men echter enkel over de verandering in gedrag van een individu (fitheid).

[4] In hun studie maken zij gebruik van de entropie en de transformatieafstand om diversiteit op te meten. De exacte definitie en interpretatie van dit begrip wordt nader toegelicht in hoofdstuk 5.

Bij de eenvoudigere probleemdefinities bestaat er een groot aantal kwalitatief goede oplossingen, elk met een verschillende boomstructuur. Wanneer de populatie convergeert naar één of meerdere van deze (sub)optimale oplossingen verlaagt de selectiedruk waardoor in combinatie met het grote aantal structureel verschillende (sub)optimale oplossingen, de structurele diversiteit binnen de populatie groot is. De hoge diversiteit en lage selectiedruk zorgen ervoor dat de code trager groeit.

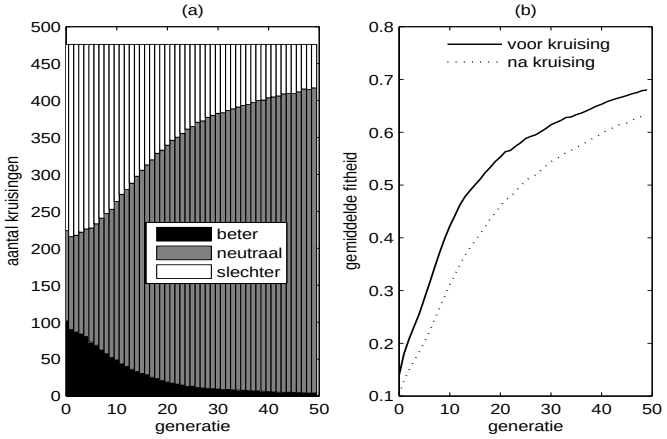
In de moeilijkere probleemstellingen is er een beperkter aantal goede oplossingen voorhanden waardoor de selectiedruk hoger is. Het bestaan van minder (sub)optimale oplossingen in combinatie met de hoge selectiedruk, zorgt voor minder structurele diversiteit (Tackett 1994). Dit leidt tot een snelle stijging van de boomgrootte.

3.3.7 Enkele algemene bedenkingen bij verklaringen van codegroei

De meeste theorieën en causale modellen belichten slechts een deelaspect van de oorzaken van codegroei. Sommige resultaten spreken de besluiten van andere theorieën tegen. Zo twijfelt Soule (2003) aan de correctheid van de theorie van (Langdon & Poli 1997*a*) terwijl Gelly *et al.* (2005) dan weer steunt op de statistische leertheorie van Vapnik (1999) om deze theorie te bekrachtigen. We zijn dan ook overtuigd dat codegroei te wijten is aan een samenspel van een aantal factoren.

De bespreking die volgt heeft zeker niet tot doel om één enkele theorie naar voren te schuiven en te aanvaarden als de allesomvattende verklaring van codegroei. Wel staan we kritisch tegenover de reeds ontwikkelde theorieën en hun tekortkomingen.

Bescherming tegen structurele veranderingen Om na te gaan of kruising wel degelijk zo destructief is als door vele auteurs wordt beweerd, tellen we, per generatie, het aantal kruisingen waarbij (A) er een verlies aan fitheid optreedt, (B) de fitheid niet verandert en tenslotte (C) er een hogere fitheidswaarde wordt bekomen. De resultaten zijn te zien in Figuur 3.3. We merken dat de meerderheid van de kruisingen neutraal (i.p.v. destructief) is. Wanneer we de gemiddelde fitheid bekijken van alle individuen die via kruising zijn gegenereerd (Figuur 3.3b), zien we dat de individuen na kruising gemiddeld slechter presteren dan ervoor. We moeten voorzichtig zijn met de interpretatie van deze resultaten aangezien ze slechts geldig zijn voor de kunstmatige mier (er is dus een zekere probleemafhankelijkheid) en voor de gebruikte parameterinstellingen.



Figuur 3.3: *Het effect van kruising bij de kunstmatige mier. De linkse grafiek toont het aantal destructieve kruisingen (kleinere fitheidswaarde), neutrale kruisingen en kruisingen waarbij een verhoging van de fitheidswaarde optreedt. De rechtse grafiek toont de gemiddelde fitheid van individuen voor en na kruising (alle types kruisingen). Het aantal generaties is beperkt tot 50. Alle andere parameterinstellingen zijn gekozen zoals beschreven in 2.3.4.*

Gebaseerd op deze resultaten kunnen we toch een aantal bedenkingen formuleren bij de theorie uit 3.3.2. Generatie na generatie wordt de boomstructuur van de kandidaat-oplossingen alsmaar groter. We verwachten dus dat de individuen beter beschermd zullen zijn tegen kruising. In zekere zin klopt dit aangezien het aantal kruisingen waarbij de fitheid daalt (zuiver destructief), sterk afneemt. Nochtans, wanneer men de hypothese omkeert, zou men verwachten dat de code minder snel toeneemt zodra er een voldoende niveau van bescherming wordt geboden. Dit werd evenwel niet vastgesteld.

Removal bias In de meeste softwarepakketten worden er steeds twee individuen geselecteerd voor kruising. De deelboom die bij het eerste individu wordt weggeknipt, wordt toegevoegd aan het tweede individu en omgekeerd. Indien er nu een bias bestaat om kleinere deelbomen uit een boomstructuur te verwijderen dan zal die kleine deelboom toegevoegd worden aan het tweede individu (en omgekeerd). Tijdens kruising wordt er geen extra informatie toegevoegd. Soule & Foster (1998*b*) insinueren dat er geen bias bestaat voor het toevoegen van een grotere boom. Maar aangezien de geselecteerde deelboom (eerder klein) uit het eerste individu overeenstemt met de toegevoegde deelboom (dus ook klein) in het tweede individu, zal de boomgrootte voor en na kruising identiek zijn. Rekening houdend met de vaak explosieve (sub-

kwadratische) groei van de gemiddelde boomgrootte (en boomdiepte) kan deze theorie 'alleen' niet verklaren waarom codegroei optreedt. We vermoeden dat vooral de plaats waar de deelbomen worden ingevoegd een belangrijkere rol speelt dan wel de omvang van de verwijderde/toegevoegde deelboom. Dit vermoeden wordt ook gesuggereerd in (Luke 2003).

Deze theorie werd bewezen door O'Reilly (1995) en later ook door Langdon & Poli (1998*c*) met behulp van een speciale variant op hillclimbing. De gebruikte techniek laat geen kruising toe waarvan de resulterende programmabomen de fitheid van een individu verlagen. Indien kruising toch destructief is, wordt het geselecteerde individu gedupliceerd. De auteurs stelden vast dat codegroei vermindert. Wanneer men eveneens alle neutrale kruisingen verbiedt, wordt codegroei nog verder afgeremd. Het verschil tussen de curves die de boomgrootte weergeven, is volgens hen te wijten aan het uitsluiten van kruisingen die zich voordoen in stukken niet-levensvatbare code.

Nochtans is deze conclusie niet geldig en is ook de bewijsvoering niet volledig correct. Zoals we kunnen afleiden uit Figuur 3.3 is het aantal destructieve en neutrale kruisingen vrij groot. We verwachten dus dat een dergelijke techniek veel kopieën van individuen zal maken. Aangezien de ouders en voorouders van de geselecteerde individuen in het algemeen kleiner zijn, zal deze techniek de gemiddelde boomgrootte beperken. We vrezen echter dat deze techniek niets meer is dan een lapmiddel om codegroei tegen te gaan, dan wel een methode die exact kan identificeren 'waarom' en 'waardoor' de code begint toe te nemen. Uit Luke (2003) blijkt dat de omvang van de code zelfs toeneemt indien men kruising de toegang tot niet-levensvatbare code ontzegde[5].

Spreiding van fitheidswaarden Wellicht bestaan er inderdaad meer grotere programmabomen van een gegeven fitheid dan kleinere exemplaren. Maar het is heel waarschijnlijk dat er ook veel (véél) meer grotere bomen bestaan maar met een kleinere fitheidswaarde. Dus in plaats van de verhouding te nemen tussen het aantal grote en kleine bomen voor een bepaalde fitheidswaarde lijkt het verstandiger om de verhouding te nemen tussen twee percentages. Namelijk de procentuele verhouding tussen het aantal kleinere individuen met hoge en lage fitheidswaarden en de procentuele verhouding tussen het aantal grotere individuen met hoge en lage fitheidswaarden. Verder is het ook eenvoudiger om uitgaande van twee kleine bomen met behulp van (bijvoorbeeld) kruising een grote boom te creëren dan een kleinere boom.

[5]S. Luke gaf wel toe dat deze toename niet statistisch significant was bij de gebruikte Booleaanse testproblemen (6 en 11-bit multiplexer). Er was wel een significant verschil bij het regressieprobleem (vierde orde polynoom).

De diepte van het kruisingspunt Recent werd het begrip robuustheid (zelfde definitie als M. Streeter) ook bestudeerd door T. Soule in (Soule *et al.* 2002, Besetti & Soule 2005, Soule 2003) en Harries & Smith (1998). Soule bevestigt de conclusies van Streeter, namelijk dat genetisch programmeren een balans moet vinden tussen de selectiedruk veroorzaakt door fitheid en de selectiedruk veroorzaakt door het zoeken naar robuuste oplossingen, en dat de robuustheid van oplossingen kan verhoogd worden door niet-levensvatbare code en intronen. Maar Soule ging verder en suggereerde ook dat de robuustheid kan worden verhoogd door de keuze van bepaalde genen (knopen in de boomstructuur). Soms wordt de robuustheid zelfs verhoogd door een inkrimping van het genoom (bij mutatie).

Dit is een bijzonder aangename vaststelling aangezien we in dit proefschrift codegroei wensen te onderdrukken met behulp van een lokale optimalisatietechniek die het genoom op intelligente wijze reduceert (zie hoofdstuk 4).

De relatie met de probleemdefinitie Deze theorie werd voorlopig enkel getest op een aantal regressieproblemen. Een verklaring voor codegroei die indirect steunt op de probleemstelling kan misschien niet geldig zijn voor een andere categorie van probleemdefinities zoals Booleaanse problemen. Bovendien zegt de vooropgestelde hypothese meer over de snelheid waarmee de code aangroeit in plaats van waarom codegroei optreedt. De resultaten en conclusies zijn eveneens gebonden aan het gebruik van de entropie en de transformatieafstand. Jammer genoeg verzaakt deze theorie om de relatie tussen de probleemdefinitie en de voorstelling ervan binnen genetisch programmeren verder te onderzoeken. Zo is het best mogelijk dat bepaalde keuzes van functies en terminalen minder optimaal zijn en rechtstreeks aanleiding geven tot een verhoogde codegroei.

3.4 Het bestrijden van codegroei

Samen met de zoektocht naar een verklaring voor codegroei is het bestrijden van dit fenomeen één van de meest gedocumenteerde onderwerpen uit de literatuur over GP. Het is dan ook niet verwonderlijk dat er talloze conceptueel verschillende manieren bestaan om codegroei te reduceren. Dit overaanbod maakt het zeker niet eenvoudiger om de verschillende methoden te catalogeren. Deze paragraaf geeft een overzicht van de meest gebruikte technieken om codegroei te bestrijden.

3.4.1 Beperking van de grootte en diepte van de boom

De meest klassieke manier om codegroei te beperken is het gebruik van een harde bovengrens voor de boomgrootte en de boomdiepte. Koza (1992) gebruikte een dieptelimiet maar tegenwoordig zijn beperkingen op de boomgrootte meer gangbaar. De methode zelf is zeer eenvoudig te implementeren en werkt bovendien bijzonder effectief. Een gelijkaardige manier om codegroei te bestrijden is door deze limiet gradueel te verhogen, vertrekkende van een kleine waarde. Deze aanpak wordt toegepast in ADATE (Automatic Design of Algorithms Through Evolution) (Olssen 1995).

In (Silva *et al.* 2005, Silva & Costa 2005*a*) wordt codegroei gereduceerd door globale beperkingen op te leggen op de bronnen van een GP systeem. Bij deze aanpak wordt het aantal bronnen dat ter beschikking staat van de individuen beperkt tot het populatieniveau (i.p.v. tot het niveau van het individu zoals bij (Wagner & Michalewicz 2001)). Een voorbeeld van een gemeenschappelijke bron is het aantal knopen. Men kan dan bijvoorbeeld eisen dat er binnen de populatie niet meer dan 10000 knopen mogen voorkomen. De methode werkt als volgt. Na de voortplantingsfase worden alle nakomelingen alsook alle ouders gesorteerd op basis van fitheid (elke groep afzonderlijk). Vervolgens worden beide groepen samengevoegd in één lijst waarbij de groep nakomelingen eerst komt. Tenslotte worden de bronnen toegewezen in een FIFO (first in, first out) volgorde. Naar het einde toe worden de individuen die meer bronnen vragen dan beschikbaar overgeslagen. We geven een voorbeeld. Nemen we aan dat de gesorteerde lijst er als volgt uitziet (O duidt op kind, P duidt op ouder): O1 (103 knopen), O2 (13 knopen), O3 (340 knopen), O4 (50 knopen), P1 (130 knopen), P2 (56 knopen), P3 (45 knopen), P4 (121 knopen). Wanneer het maximum aantal knopen (= bronnen) in de populatie niet meer dan 400 mag bedragen dan zal het algoritme de volgende individuen selecteren: O1, O2, O4, P1, P2 en P3. Enkel deze individuen zullen deel uitmaken van de nieuwe populatie. Aangezien het aantal knopen van individu O3 de opgelegde maximum overschrijdt, zal dit individu worden overgeslagen. De auteurs hebben deze methode uitgebreid om de bronnen op dynamische wijze toe te kennen (Silva & Costa 2005*b*). Hun aanpak reduceerde codegroei zonder de kwaliteit van de geëvolueerde oplossingen te beïnvloeden.

3.4.2 Genetische operatoren en selectieschema's

Langdon (1999) gebruikt twee varianten op de kruisingsoperator. Beide operatoren selecteren een willekeurige knoop in één ouder. De deelboom met als beginknoop de geselecteerde knoop wordt dan vervangen door een deelboom uit het tweede individu met ongeveer dezelfde lengte. Ekart (1999) gebruikt een speciale mutatieoperator (frameshift mutatie: toevoeging of verwijderen van één of meerdere basisparen en herstructurering van DNA sequenties) voor de simplificatie van programma's. Vereenvoudiging van de boomstructuur wordt ook gebruikt door (Hooper & Flann 1996, Blickle 1996*a*) bij symbolische regressie. Zij merken op dat de kwaliteit van de oplossingen stijgt terwijl de codegroei afneemt. Ook Brameier & Banzhaf (2001) gebruiken een vereenvoudigingsmodule in lineair genetisch programmeren. Zij beperken zich echter tot één bepaalde klasse van intronen.

Kinnear (1993*a*) gebruikt een mutatieoperator (*hoist*) om het aantal kleinere individuen in de populatie te verhogen. Hoist selecteert een knoop op willekeurige basis en promoveert de deelboom (met als beginknoop de geselecteerde knoop) tot een nieuw individu. Ook Angeline & Pollack (1993*b*) gebruiken een compressieoperator die een willekeurig punt in de boom selecteert. Bogen die een vooraf ingestelde maximale diepte overschrijden worden uit de boom verwijderd. De resulterende boomstructuur wordt gedefinieerd als een module en wordt toegevoegd aan een bibliotheek voor later gebruik. Poli reduceert codegroei door het gebruik van een constante mutatieprobabiliteit per knoop (Langdon & Poli 1997*b*). Hierdoor wordt het evolutionaire voordeel van niet-effectieve code teniet gedaan. Programma's met veel niet-effectieve code hebben geen succes, aangezien de kans om effectieve code te beschadigen onafhankelijk is van de hoeveelheid niet-effectieve code.

Ook automatisch gedefinieerde functies hebben het voordeel dat de gegenereerde programma's vaak veel expressiever en kleiner zijn dan wanneer men geen gebruik maakt van ADF (Koza 1994). Toch is het zo dat het gebruik van ADF's codegroei niet steeds kan voorkomen (Langdon 1995).

Fernandez *et al.* (2003) en Fernandez & Martin (2004) gaan dynamisch de populatiegrootte aanpassen om convergentie tegen te gaan. Om codegroei te reduceren gebruiken ze plagen om slechte individuen uit te roeien. In het bijzonder worden telkens een vast aantal individuen uit de populatie verwijderd (laagste fitheidswaarden). Onder de individuen met de laagste fitheidswaarden worden dan de grootste er eerst uitgegooid. Nieuwe individuen worden gecreëerd door mutaties toe te passen op de beste individuen die tot dan gekend zijn. Hun methode vergt echter eveneens een instelparameter die beslist

of er nu individuen moeten verwijderd of toegevoegd worden. Verder vraagt het algoritme ook instelwaarden voor het aantal toe te voegen en/of te verwijderen individuen. Hun techniek vond betere individuen en bovendien ook veel sneller in vergelijking met standaard GP.

Luke & Panait (2002) introduceren dubbele en proportionele tornooiselectie. Deze techniek is gelijkaardig aan VEGA (Vector Evaluated Genetic Algorithm (Schaffer 1985)). Beide methoden zijn gebaseerd op de klassieke tornooiselectie. In dubbele tornooiselectie moeten de individuen tweemaal duelleren: één keer op basis van fitheid en één keer op basis van boomgrootte. Bij proportionele tornooiselectie wordt er soms geduelleerd op basis van fitheid en soms op basis van boomgrootte. Ondanks de benaming van niet-parametrische *parsimony pressure* vereisen beide methoden parameterinstellingen die het belang van beide objectieven tegen elkaar afwegen.

3.4.3 De verandering van de fitheidsbepaling

In deze paragraaf bespreken we enkel de methodes die twee of meer objectieven (vaak zijn het er maar twee: boomgrootte en fitheid) combineren tot één enkele fitheidswaarde. Een objectief is het doel waarnaar men optimaliseert. Deze technieken worden ook vaak pseudo multi-objectieve methoden genoemd aangezien ze meerdere objectieven combineren tot één enkele via een transformatie. Vaak is een lineaire combinatie van de objectieven voldoende: $f(O_1, O_2, ..., O_n) = \alpha_1 O_1 + \alpha_2 O_2 + ... + \alpha_n O_n$. Deze aanpak wordt gebruikt in (Blickle 1996*a*) en staat gekend onder de naam *constant linear parsimony pressure*. Fitheid wordt bepaald door $f = E + \alpha N$ waarbij N de boomgrootte aanduidt, E de efficiëntie van de oplossing (=traditionele fitheid) en α de bijdrage van de boomgrootte tot de uiteindelijke fitheidswaarde[6]. Gathercole & Ross (1997) stellen voor om de bijdrage van de boomgrootte in de bepaling van fitheid te beperken zodat het enkel een invloed uitoefent bij het vergelijken van twee individuen met identieke score.

Iba *et al.* (1994) definiëren een fitheidsfunctie gebaseerd op het *Minimum Description Length* principe (MDL). De complexiteit van de boomstructuur wordt weerspiegeld in de fitheidswaarde. Soule *et al.* (1996) vergelijken het gebruik van een aangepaste fitheidsfunctie met verschillende manieren om intronen uit de boomstructuur te verwijderen. Zij concluderen dat de penali-

[6] α kan zowel positief als negatief zijn afhankelijk van de precieze definitie van fitheid. Grote individuen worden echter steeds bestraft. Bijvoorbeeld: indien hogere fitheidswaarden corresponderen met betere individuen dan zal α negatief zijn; omgekeerd, indien lagere fitheidswaarden corresponderen met betere individuen dan zal α positief zijn.

satie van grotere structuren steeds betere resultaten geven dan het vereenvoudigen van de boomstructuur.

Bij adaptieve *parsimony pressure* is α afhankelijk van de complexiteit van de huidige generatie. Een voorbeeld hiervan is te vinden in (Zhang & Mühlenbein 1995). Via een statistische aanpak wordt de relatie tussen de complexiteit van de boomstructuur en zijn fitheidswaarde beschreven. De fitheidsfunctie is gebaseerd op het MDL principe en de bijdrage van de complexiteit tot de fitheidsbepaling wordt adaptief gewijzigd (Zhang & Mühlenbein 1996).

Poli (2003) gebruikt een zeer eenvoudige techniek om de grotere individuen uit de populatie te verwijderen (*Tarpeian bloat control*). Boomstructuren groter dan het populatiegemiddelde worden vernietigd door hun fitheid gelijk te stellen aan nul (slechts één op de $ratio$ individuen). Het algoritme werkt als volgt.

```
if( (size(programma)>gemiddelde_populatiegrootte)
    AND
    (random()<ratio))
then return fitheid=0;
else return fitheid(programma);
```

3.4.4 De multi-objectieve methoden

Vele reële problemen behelzen het vinden van oplossingen waarbij gelijktijdig een aantal objectieven moeten worden gehaald. Deze objectieven zijn niet altijd vergelijkbaar en staan vaak met elkaar in competitie. Er is dan ook meestal meer dan één oplossing voor het probleem: een verzameling oplossingen die met elkaar onvergelijkbaar zijn. Er bestaan dus geen andere oplossingen uit de zoekruimte die, wanneer we alle objectieven in beschouwing nemen, even goed zijn als de oplossingen uit die verzameling. Dergelijke individuen worden vaak Pareto-optimale individuen of niet-gedomineerde individuen genoemd. We kunnen dit wiskundig als volgt formuleren. Veronderstel dat een individu x de waarden x_i heeft voor n objectieven en veronderstel bovendien een individu y met waarden y_i voor dezelfde n objectieven. Dan domineert x het individu y als en slechts als (in het geval van een maximalisatieprobleem):

$$x \succ y \Rightarrow \forall i \in [1..n] : x_i \geq y_i \wedge \exists i \in [1..n] : x_i > y_i$$

Het principe van multi-objectief optimalisatie bestaat reeds geruime tijd in genetische algoritmen (Fonseca & Fleming 1993, 1995). Recent wordt het ook steeds vaker in genetisch programmeren toegepast (Deb 2002, Zitzler & Thiele 1999, Zitzler *et al.* 2000, Ekárt & Németh 2001, De Jong *et al.* 2001, Rodríguez-Vásquez *et al.* 1997, Bleuler *et al.* 2001) waarbij vaak enkel fitheid en boomgrootte als objectieven worden gebruikt. Deze methoden bieden het voordeel dat een verzameling van oplossingen (de Pareto-optimale set) wordt gezocht in plaats van slechts één enkele. Zoals vele artikelen aantonen vormen multi-objectieve methoden een veelbelovende techniek om codegroei te bestrijden.

3.4.5 Enkele algemene bedenkingen bij methoden om codegroei te bestrijden

Quasi alle besproken methoden hebben één ding gemeenschappelijk: correct gebruik vereist één of meerdere extra parameterinstellingen. Op deze manier verplicht de auteur van de codegroeibegrenzer (= de persoon met de meeste ervaring en kennis over zijn algoritme en het GP proces) de applicatieprogrammeur tot het kiezen van een 'geschikte' waarde. Deze instelling is bijna altijd probleemafhankelijk en vereist het uitproberen van verschillende waarden. Niemand kan immers op voorhand zeggen hoe de uiteindelijke oplossing eruit zal zien (hoe diep? hoeveel en welke knopen?). Voor benchmarktoepassingen kan men een duidelijk afgelijnd interval opgeven waarbinnen de instelwaarde zich moet bevinden. Voor reële minder triviale problemen wordt de applicatieprogrammeur aan zijn lot overgelaten.

Ook bij pseudo multi-objectief methoden moet de gebruiker heel wat extra parameterinstellingen kiezen. Het grootste probleem met deze techniek is de bijna oneindige ruimte van mogelijke gewichtsfuncties om alle objectieven met elkaar te verbinden. Bovendien is het niet eenvoudig om, bijvoorbeeld bij *parametric parsimony pressure* methoden, een correcte waarde voor de gewichtsfactor α te kiezen. Hoge waarden zorgen er immers voor dat de populatie snel convergeert naar kleine bomen waardoor de kans om een goed individu te vinden kan dalen (Soule & Foster 1998*a*, Rosca 1997).

Technieken die de boomstructuur trachten te vereenvoudigen, zijn vaak rekenintensief waardoor hun gebruik wordt teruggedrongen. Meestal worden ze ingezet na de laatste generatie om de oplossing leesbaarder te maken. Deze oplossing creëert dan weer nieuwe problemen. Zo rapporteerde Haynes (1998) diversiteitsverlies bij het verwijderen van grote hoeveelheden niet-

functionele code. Ook het vinden van geschikte reductieregels is niet altijd vanzelfsprekend.

Ook de echte multi-objectief methoden kampen vaak met nadelen. Zo leidt het gebruik van Pareto-dominantie vaak tot convergentie richting zeer kleine individuen (Ekárt & Németh 2001, Langdon & Nordin 2000). Om dit probleem te omzeilen, gebruiken de Jong & Pollack (2003) een multi-objectief methode in combinatie met de controle van diversiteit om codegroei te reduceren. De auteurs omzeilen zo de problemen van vroegtijdige convergentie en codegroei terwijl fitheid positief evolueert. Zij introduceren eveneens het FOCUS algoritme dat diversiteit als een extra objectief opneemt en specifiek op zoek gaat naar oplossingen van de voorspelde minimale lengte.

Een ander belangrijk nadeel is dat men in de zoektocht naar compacte boomstructuren vaak de kwaliteit uit het oog verliest. Veel methoden produceren dan ook kleine en leesbare kandidaat-oplossingen maar met een lagere fitheid.

3.5 Besluiten en vooruitblik

Er bestaan verschillende theorieën die codegroei trachten te verklaren. We zijn ervan overtuigd dat codegroei te wijten is aan een samenspel van een aantal hypotheses. Dit proefschrift heeft niet tot doel een zoveelste hypothese in de rij te formuleren maar staat kritisch tegenover de reeds vooropgestelde verklaringen. Wel hopen we dat dankzij resultaten uit dit proefschrift bewijsmateriaal wordt gevonden om bepaalde hypothesen te bekrachtigen of ontkennen en dat de resultaten aanleiding kunnen geven tot nieuwe inzichten inzake het ontstaan van codegroei.

Gelukkig zijn alle onderzoekers het erover eens dat codegroei een vaak voorkomend fenomeen is en een heleboel ongewenste bijwerkingen introduceert die de schaalbaarheid van genetisch programmeren zeer negatief beïnvloeden. Het is dan ook niet verwonderlijk dat een groot aantal auteurs zich concentreren op het bestrijden van codegroei op talloze verschillende manieren.

Er zijn echter een aantal problemen met de voorgestelde oplossingen zoals beschreven in 3.4.5:

- Vaak wordt de gebruiker verplicht één of meerdere parameters op te geven die verantwoordelijk zijn voor de omvang van de reductie van de code: denk maar aan de gewichtsfunctie bij pseudo multi-objectief

methoden. De correcte instelling van deze parameters is in vele gevallen zeer probleemspecifiek en vergt enig uitproberen.

- Een tweede probleem dat optreedt, is het verlies aan diversiteit door een (te) sterke reductie van de boomstructuur.
- Een derde belangrijk probleem is het verlies aan fitheid. Wanneer men de boomstructuur verkleint, treden er ongewenste neveneffecten op bij de evolutie van gemiddelde en maximum fitheid. De reductie van codegroei kan vaak niet verhinderen dat de kwaliteit van de geëvolueerde oplossingen stagneert of erger nog verslecht. Misschien is dit een rechtstreeks of onrechtstreeks gevolg van het verlies aan diversiteit.

In hoofdstuk 4 introduceren we het concept van semantisch gedreven lokale optimalisatie om de boomstructuur te reduceren en toch eenzelfde niveau van fitheid te behouden. Idealiter zullen we zelfs trachten op de fitheid van kandidaat-oplossingen te verbeteren. De bevindingen uit hoofdstuk 4 en de studie van het begrip diversiteit (hoofdstuk 5) leiden uiteindelijk tot een parametervrije reductie van codegroei waarbij kandidaat-oplossingen opnieuw compact en leesbaar worden gemaakt (hoofdstuk 6).

Hoofdstuk 4

Lokale optimalisatie

Dit hoofdstuk beschrijft een nieuwe aanpak om codegroei te bestrijden gebaseerd op lokale optimalisatie van de boomstructuur. Centraal staat het begrip 'geschikte deelboom'. Dit hoofdstuk begint met de beschrijving van de ingrediënten van de nieuwe methode. Vervolgens bespreken we kort de gebruikte experimentele proefopstelling om deze nieuwe methode te testen, gevolgd door een bespreking van de resultaten. De nieuwe operator wordt vergeleken met standaard genetisch programmeren (zonder codegroeibegrenzers) alsook met verschillende algemeen aanvaarde methoden om codegroei te bestrijden.

We implementeren in dit hoofdstuk tevens een algoritmische optimalisatie voor de bepaling van de fitheidswaarden van deelbomen, aangezien de benodigde rekentijd drastisch kan oplopen.

4.1 Codegroei bestrijding

Vaak gaan codegroeibegrenzers uit van één of meerdere theorieën om codegroei te verklaren. Denk bijvoorbeeld maar aan de tientallen alternatieve implementaties voor kruising of andere genetische operatoren, allen gebaseerd op de hypothese van o.a. (Bickle & Thiele 1994, McPhee & Miller 1995, Nordin *et al.* 1995, Langdon *et al.* 1999, Banzhaf *et al.* 1997). Ook mechanismen om de selectiedruk te verlagen zijn zeer populair (gebaseerd op Langdon & Poli (1997*a*)). Maar aangezien GP zo snel mogelijk een optimale oplossing wil vinden, is het verlagen van de selectiedruk misschien een niet zo beste oplossing.

Er zijn echter nog een aantal andere problemen met de voorgestelde codegroeibegrenzers zoals beschreven in 3.4.5. Het verlies aan fitheid is ongetwijfeld het belangrijkste probleem. Veel codegroeibegrenzers kunnen vaak niet verhinderen dat de fitheid van de geëvolueerde oplossingen stagneert of erger nog verslecht. Optimalisatie, het vinden van steeds betere oplossingen, staat immers haaks op elke techniek die resulteert in een reductie van kwaliteit. Men moet steeds een positieve fitheidsevolutie als dé primaire doelstelling voor ogen houden. Vele groeiremmers verzwakken deze doelstelling door de reductie van de boomstructuur voorop te stellen.

Zelden gebruikt men kennis uit het probleemdomein om op een efficiënte manier aan codegroei reductie te doen. Nochtans is deze kennis steeds voorhanden aangezien ze wordt gebruikt bij de bepaling van de fitheid. Automatisch gedefinieerde functies en modules waarbij deelbomen worden geselecteerd en toegevoegd aan een bibliotheek (Rosca & Ballard 1994) zijn een stap in de goede richting maar hebben vooral tot doel om de oplossing meer gestructureerd en leesbaar te maken. Ze zijn ontworpen specifiek met proceduraal programmeren voor ogen. Bovendien kan men niet echt spreken van een reductie van de code aangezien, als we het aantal geëvalueerde functies tellen, een boom met of zonder ADF evenveel functie-evaluaties heeft.

In dit hoofdstuk introduceren we het concept van semantisch gedreven lokale optimalisatie om de boomstructuur te reduceren en toch eenzelfde niveau van fitheid te behouden. Idealiter zullen we zelfs trachten op de fitheid van kandidaat-oplossingen te verbeteren. We zullen de boomstructuur van een individu optimaliseren door gebruik te maken van de aanwezige kennis uit het probleemdomein (i.h.b. de fitness cases). Deze aanpak noemen we lokale optimalisatie aangezien de reductie onrechtstreeks gebeurt tijdens de evaluatie van de fitheid van een kandidaat-oplossing.

4.2 Lokale optimalisatie

4.2.1 Lokale optimalisatie van de boomstructuur

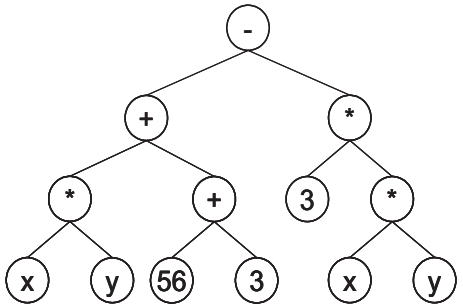
De meeste methoden om codegroei te bestrijden spitsen zich in hoofdzaak toe op de reductie van de boomgrootte. De hiervan afstammende codegroeiremmers koppelen de werking van de evolutionaire cyclus of de bepaling van de fitheid met de omvang van de kandidaat-oplossingen. GP is echter, zoals reeds herhaaldelijke keren werd vermeld, een optimalisatiemethode waarbij de evolutie van kwalitatief hoogstaande oplossingen steeds de voorkeur geniet.

Nochtans kan de honger naar steeds betere oplossingen adequaat worden gestild. Er is immers een grote hoeveelheid informatie over de toepassing aanwezig onder de vorm van een verzameling fitness cases. De methode die in deze paragraaf wordt geïntroduceerd, zal deze kennis zoveel mogelijk trachten te benutten en stelt de vrijwaring van een verder positief verloop van fitheid tot één van de belangrijkste doelen.

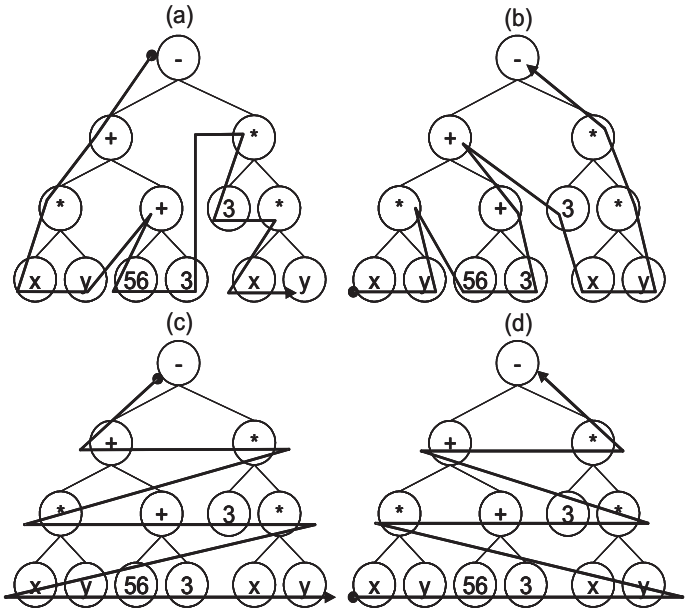
Lokale optimalisatie bestaat, zoals de benaming zelf al doet vermoeden, uit het zoeken naar een kwalitatief betere (en compacte) deeloplossing. Het opmeten van de kwaliteit van een deeloplossing geschiedt via de verzameling fitness cases. Concreet zal de boomstructuur van elk individu worden gescreend op de aanwezigheid van een deelboom met hogere (of minstens gelijke) fitheid in vergelijking met de fitheid van de volledige boom. De omvang van de deelboom (indien deze wordt gevonden) zal steeds kleiner zijn in vergelijking met de omvang van de volledige boomstructuur aangezien we de wortelknoop van de volledige boom buiten beschouwing laten. Op deze manier blijven we trouw aan de primaire doelstelling van een optimalisatiemethode en reduceren we toch de boomstructuur van de kandidaat-oplossingen.

Samengevat, lokale optimalisatie buit de aanwezige kennis over de te optimaliseren toepassing grondig uit door op zoek te gaan naar deeloplossingen met hogere fitheid. Lokale optimalisatie is dus een semantisch gestuurde methode die zich in zekere zin gedraagt als een dieptelimiet.

We geven in de volgende paragrafen eerst een kort overzicht van de benodigde ingrediënten van lokale optimalisatie, uiteindelijk gevolgd door de operator zelf.



Figuur 4.1: *De boom die de uitdrukking* $((x * y + (56 + 3)) - (3 * (x * y)))$ *voorstelt.*



Figuur 4.2: *Verschillende wandelingen doorheen een boomstructuur die de uitdrukking* $((x * y + (56 + 3)) - (3 * (x * y)))$ *voorstelt. (a) Preorde, (b) postorde, (c) breedte-eerst en tenslotte (d) omgekeerd breedte-eerst. De cirkel markeert het startpunt, de pijl het eindpunt.*

4.2.2 Een wandeling doorheen de boomstructuur

We kunnen boomstructuren op verschillende manieren doorlopen zodat alle knopen slechts éénmaal worden bezocht ('wandelingen'). Preorde, inorde[1]

[1]Een inorde wandeling is enkel mogelijk wanneer men beschikt over een binaire boom waarbij elke functie exact twee kinderen (operandi) heeft. Dit is niet steeds het geval voor

en postorde zijn gekende manieren om een boom te doorlopen maar er zijn nog andere wandelingen mogelijk. Aangezien een boom recursief gedefinieerd is, is het niet verwonderlijk dat deze wandelingen ook recursief zijn. In dit proefschrift worden vier verschillende wandelingen in beschouwing genomen. Om deze wandelingen te verduidelijken, gebruiken we de uitdrukking zoals weergegeven in Figuur 4.1. Een grafische voorstelling van de werking en volgorde van bezoeken wordt getoond in Figuur 4.2.

1. Preorde wandeling. Hierbij wordt eerst de wortel van de boom bezocht, vervolgens de linkerdeelboom (op preorde manier) en tenslotte de rechterdeelboom (eveneens op preorde manier). Toegepast op Figuur 4.1 geeft een preorde wandeling als resultaat: $- + \times x\ y + 56\ 3 \times 3 \times x\ y$.

2. Postorde wandeling. Hierbij worden eerst alle kinderen van de huidige knoop bezocht (startend bij de wortel), en pas daarna de huidige knoop zelf. Toegepast op Figuur 4.1 geeft een postorde wandeling als resultaat: $x\ y \times 56\ 3 + + 3\ x\ y \times \times -$.

3. Breedte-eerst wandeling. We bezoeken eerst alle knopen op diepte nul (enkel de wortel van de boom), vervolgens alle knopen op diepte één, enzovoort tot wanneer we alle knopen hebben bezocht. Een breedte-eerst wandeling toegepast op de boom uit Figuur 4.1 geeft de volgende volgorde: $- + \times \times + 3 \times x\ y\ 56\ 3\ x\ y$.

4. Omgekeerd breedte-eerst wandeling. Zoals de naam reeds doet vermoeden, verschilt deze methode nauwelijks van de voorgaande. We beginnen deze keer echter helemaal onderaan (bij de diepst gelegen knopen) en werken niveau per niveau naar de wortel toe. Toegepast op Figuur 4.1 geeft een omgekeerd breedte-eerst wandeling als resultaat: $x\ y\ 56\ 3\ x\ y \times + 3 \times + \times -$.

In paragraaf 2.4 hebben we vermeld dat het softwarepakket lil-gp de knopen opslaat in prefixnotatie. Deze voorstelling sluit zeer nauw aan bij een preorde wandeling doorheen de boomstructuur. Een preorde wandeling zal dus in staat zijn om zeer snel alle knopen van de boom te bezoeken in vergelijking met de andere methoden. Voor postorde, breedte-eerst en omgekeerd breedte-eerst wandelingen moeten alle knopen van de boom opnieuw

de benchmarktoepassingen die hier worden behandeld. We laten daarom deze manier verder buiten beschouwing.

geordend worden met behulp van een afzonderlijke sorteerprocedure. Deze procedure neemt extra tijd in beslag.

4.2.3 Het vinden van een geschikte deelboom: de deelboomselectiestrategieën

In deze paragraaf zal duidelijk worden waarom bovenstaande wandelingen doorheen een boom werden geïntroduceerd. Maar eerst definiëren we het begrip deelboom. Een deelboom wordt gevormd door een interne knoop, de kinderen van deze knoop, de kleinkinderen van deze knoop enzovoort — alle knopen die zich onder een bepaald niveau in de hiërarchische structuur bevinden. Met een interne knoop bedoelen we een knoop die minstens één kind heeft. Eindknopen (terminalen) zijn dus niet toegestaan. Verder sluiten we eveneens de wortel van de boomstructuur uit. De diepte van een deelboom is dus steeds minstens één minder dan de diepte van de volledige boom. We vermelden nog dat de knopen worden genummerd van 0 (wortel) tot het aantal interne knopen op dezelfde manier als de knopen opgeslaan worden in het geheugen (prefixnotatie). Toegepast op de boom in Figuur 4.1 geeft dit vijf geldige deelbomen: $(x * y + (56 + 3))$, $(3 * (x * y))$, $(56 + 3)$ en tweemaal $(x * y)$.

De kwaliteit van elk van deze deelbomen wordt op dezelfde wijze bepaald als voor de volledige boom, nl. door het uittesten van de deelboom op een aantal vooraf bepaalde fitness cases. Het evalueren van de fitheid van elke deelboom gebeurt in het applicatiegedeelte van het programma. Deze berekening gebruikt enkel de aanwezige kennis over het probleemdomein (fitness cases) en geen additionele informatie. Alle fitness cases worden steeds gebruikt.

Vervolgens wordt de fitheid van elke deelboom vergeleken met de fitheid van de volledige boom. Aangezien we spreken over een lokale 'optimalisatie' van de boomstructuur wensen we uiteraard enkel op zoek te gaan naar deelbomen waarvan de kwaliteit minstens even goed is als de fitheid van de volledige boom. Indien er een deelboom wordt gevonden die hieraan voldoet, zal dit individu een extra koppel waarden opslaan: <fitheid deelboom, het unieke nummer van de wortel van de deelboom>. Dit nummer ligt tussen één (nul zelf wordt uitgesloten aangezien dit de wortel van de volledige boom is) en het aantal interne knopen. Bij individuen die over geen deelboom beschikken waarvan de fitheid minstens even groot is als de fitheid van het volledige genoom, worden beide velden ongeldig gemaakt (waarde -1). Samengevat moet een goede deelboom dus voldoen aan twee voorwaarden:

1. Een deelboom is een interne knoop, verschillende van de wortel van de volledige boom.

2. De fitheid van de deelboom is minstens gelijk aan de fitheid van de volledige boom.

Nu rest uiteraard nog de vraag hoe we deze deelboom kunnen vinden? Er is dus nood aan een deelboomselectiestrategie. Algemeen onderscheiden we twee verschillende categoriën: zoekstrategieën die de beste deelboom trachten te vinden, en zoekstrategieën die een goede (maar niet noodzakelijk de beste) deelboom trachten te vinden. Vooral bij deze laatste categorie is de volgorde van knopen bezoeken zeer belangrijk en speelt de keuze van de wandeling een belangrijke rol. We verklaren nu kort de werking van elke categorie en introduceren vijf deelboomselectiemethoden die we in het vervolg van dit proefschrift zullen gebruiken.

1. **Beste deelboomselectie (BSS)**

   Tijdens de evaluatiefase worden de fitheidswaarden van 'alle' deelbomen berekend. Uit de lijst met geldige deelbomen (zie voorwaarden) wordt enkel de deelboom met de hoogste fitheidswaarde geselecteerd. Deze methode zal steeds alle deelbomen overlopen waardoor het geen rol speelt welk type wandeling we gebruiken om de knopen één voor één te bezoeken. Aangezien de knopen in prefix volgorde worden opgeslagen, passen we de preorde wandeling toe. Op deze manier wordt de computationele kost van een herordening van de knopen vermeden aangezien het evalueren van alle deelbomen sowieso veel rekentijd vraagt.

2. Bij de tweede categorie wordt de boom in een bepaalde volgorde doorlopen en stopt het algoritme zodra de fitheid van de deelboom minstens gelijk is aan de fitheid van de volledige boom. Indien een dergelijke deelboom wordt gevonden dan is dit niet noodzakelijk de 'beste' deelboom. De volgorde waarin de knopen worden bezocht speelt hier dus wel een belangrijke rol. Gebaseerd op de wandelingen uit de voorgaande deelparagraaf maken we volgend onderscheid:

   (a) **Preorde deelboomselectie (PRE)** We gebruiken een preorde wandeling. Deze strategie toegepast op Figuur 4.1 geeft als resultaat één van de volgende deelbomen (afhankelijk van welke deelboom als eerste aan de voorwaarden voldoet): $+ \times + \times \times$.

(b) **Postorde deelboomselectie (POST)** We gebruiken een postorde wandeling. Deze strategie toegepast op Figuur 4.1 geeft als resultaat één van de volgende deelbomen (afhankelijk van welke deelboom als eerste aan de voorwaarden voldoet): $\times + + \times \times$.

(c) **Breedte-eerst deelboomselectie (BFS)** We gebruiken een breedte-eerst wandeling. Deze strategie toegepast op Figuur 4.1 geeft als resultaat één van de volgende deelbomen (afhankelijk van welke deelboom als eerste aan de voorwaarden voldoet): $+ \times \times + \times$.

(d) **Omgekeerd breedte-eerst deelboomselectie (iBFS)** We gebruiken een omgekeerd breedte-eerst wandeling. Deze strategie toegepast op Figuur 4.1 geeft als resultaat één van de volgende deelbomen (afhankelijk van welke deelboom als eerste aan de voorwaarden voldoet): $\times + \times + \times$.

De drijfveer om deze categorie op te nemen in dit werk is de volgende. Op zoek gaan naar de beste deelboom vraagt het overlopen van alle mogelijke kandidaat deelbomen en vraagt dus heel wat rekentijd. Wanneer de zoektocht vroegtijdig wordt afgebroken (door te stoppen wanneer de fitheid van de deelboom groter of gelijk is aan de fitheid van het genoom) hopen we kostbare rekentijd uit te sparen. De prijs die we hiervoor betalen is dat we niet zeker zijn of we de beste deelboom hebben geselecteerd.

4.2.4 Een gepersonaliseerd selectiemechanisme

Alle operatoren (kruising, mutatie, etc.) gebruiken een selectiemechanisme om zich te voorzien van individuen die als ouders van de volgende generatie zullen fungeren. Vaak wordt er geopteerd voor tornooiselectie of voor roulettewielselectie. Ook de operator die voor de lokale optimalisatie van de boomstructuur zal zorgen, vraagt om een bijpassend selectiemechanisme. De werking verloopt als volgt.

Het selectiemechanisme gebruikt een lijst waaraan eerst alle geschikte deelbomen (zie voorwaarden) uit de populatie worden toegevoegd. Individuen waarvan de fitheid van de deelboom of het knoopnummer van de wortel gelijk is aan -1 worden uiteraard niet toegevoegd aan deze lijst. In de praktijk houdt de lijst enkel het unieke identificatienummer van het individu uit de populatie bij. Het individu bevat immers zelf een verwijzing naar de wortel en de fitheid van de deelboom.

Alle standaard aanwezige methoden om individuen uit deze lijst te selecteren, kunnen eenvoudigweg opnieuw gebruikt worden mits een kleine aanpassing. In plaats van de fitheid van de volledige boom te gebruiken, gebeurt de onderlinge afweging nu op basis van de fitheid van de deelboom. Concreet heeft men de keuze uit een aantal verschillende selectiemethoden: tornooiselectie, roullettewiel selectie, elitaire selectie of zelfs een multi-objectief (Pareto) aanpak.

In dit proefschrift opteren we steeds voor elitaire selectie. Hiervoor wordt de lijst eerst gesorteerd volgens de fitheid van de deelboom in aflopende volgorde; het individu met de beste deelboom staat vooraan. Merk op dat het selectiemechanisme waarvan sprake in deze deelparagraaf verschillend is van de zoekmethode uit voorgaande deelparagraaf. Het sorteren van de lijst gebeurt éénmaal per generatie nadat alle individuen geëvalueerd zijn en vóór de aanroep van de verschillende genetische operatoren. De zoektocht naar een geschikte deelboom wordt elke generatie opnieuw uitgevoerd voor alle individuen.

4.2.5 De operator

In de voorgaande paragrafen hebben we kort de verschillende benodigde elementen toegelicht om een lokale optimalisatie van de boomstructuur uit te voeren. In deze paragraaf combineren we deze afzonderlijke algoritmen tot één operator. We noemen deze operator kortweg LOSE wat staat voor *local search* operator. De werking verloopt als volgt.

Wanneer de lokale optimalisatieoperator wordt gekozen, zal deze operator het bijhorende selectiemechanisme oproepen die de operator bedient van een individu (i.e. elitaire selectie). De eerste maal dat elitaire selectie wordt aangeroepen, geeft de methode het individu met de beste deelboom als resultaat. De tweede maal, de tweede beste deelboom, enzovoort. Wanneer het einde van de lijst wordt bereikt, start men opnieuw bij het begin. De lijst zelf is dus circulair. De selectiemethode geeft enkel het unieke nummer van het individu als resultaat, waarmee de boomstructuur dan eenvoudigweg kan opgezocht worden.

Vervolgens wordt de positie van de deelboom opgevraagd (nummer van de wortel van de deelboom). De deelboom wordt als het ware opgehesen uit het ouder individu en zal als een volledig nieuw individu worden beschouwd. Alle andere knopen (die geen deel uitmaken van deze deelboom) worden verwijderd.

Tenslotte wordt dit individu toegevoegd aan de nieuwe populatie. Dankzij de voorwaarden die werden opgelegd aan een deelboom, bezit het nieuwe individu de volgende eigenschappen:

- Een geschikte deelboom is steeds een interne knoop verschillend van de wortel van de volledige boom. Hierdoor zal de diepte van het nieuwe individu steeds kleiner zijn dan de diepte van de volledige boom. De nieuwe boom zal dus minder knopen bevatten. Bij veelvuldige toepassing van de lokale optimalisatieoperator daalt de boomgrootte.
- De fitheid van een geschikte deelboom is steeds minstens gelijk aan de fitheid van de volledige boom. Hierdoor blijft de fitheid van de nieuwe populatie onaangeroerd.

De aanpassingen aan de evolutionaire cyclus die dienen te gebeuren, kunnen we kort samenvatten in onderstaande pseudo-code.

```
Initialiseer de populatie
herhaal
    evalueer de fitheid van de individuen
    zoek een goede deelboom m.b.v. BSS, PRE, POST,
    BFS of iBFS
    pas op probabilistische wijze de volgende
    operatoren toe
        kruising
    of
        mutatie
    of
        LOSE
    of
        ...
tot ((100% correct individu) of
     (maximum aantal generaties bereikt))
```

LOSE maakt gebruik van twee extra parameterinstellingen. Eerst en vooral moet een manier gedefinieerd worden om een geschikte deelboom te zoeken. Hiervoor hebben we vijf verschillende technieken geïntroduceerd: BSS (alle deelbomen worden bezocht en enkel de beste wordt gekozen), en de groep (PRE, POST, BFS, iBFS) waarbij men stopt zodra een geschikte deelboom wordt gevonden.

Een tweede parameter die door de gebruiker kan worden ingesteld, is de frequentie waarmee LOSE zal toegepast worden. Voorlopig zullen we hiervoor een statische instelwaarde gebruiken die niet verandert gedurende de evolutie. Hoe deze instelwaarde precies wordt bepaald, bespreken we in paragraaf 4.3.3.

4.3 De experimentele proefopstelling

In deze paragraaf wordt de experimentele proefopstelling nader toegelicht. LOSE wordt toegepast op de kunstmatige mier, het regressieprobleem en de 11-bit multiplexer. Voor een uitgebreide bespreking van de benchmarktoepassingen verwijzen we naar paragraaf 2.3. We bespreken de instellingen van de GP-simulator voor zover deze afwijken van de voorstelling in paragraaf 2.4.1.

4.3.1 De benchmarktoepassingen

Bij de kunstmatige mier en de 11-bit multiplexer worden alle parameterinstellingen gebruikt zoals gedefinieerd in Tabel 2.1.

Bij symbolische regressie kiezen we voor een vierde orde polynoom $x^4 + x^3 + x^2 + x$, analoog aan de studie in (Silva *et al.* 2005, Fernandez *et al.* 2003, Ekart 1999). De functie- en eindknopenverzameling bevat de volgende elementen: $\log$ (protected), $\exp$, $+$, $\div$, $\times$, $-$, x en een constante in het interval $[-1, 1]$. Twintig paren (abscis, ordinaat) vormen de fitness cases. De fitheid is gedefinieerd zoals in Tabel 2.1.

4.3.2 De GP-simulator

Het GP systeem evolueert gedurende 100 generaties (initiële generatie niet meegeteld) of tot wanneer een 100% correcte oplossing wordt gevonden. De initiële populatie bevat 500 individuen, gegenereerd volgens de methode uit paragraaf 2.4.1 met dieptes tussen twee en zes. Individuen worden geselecteerd door middel van tornooiselectie waarbij telkens vier individuen deelnemen (Poli 2003).

Kruising en reproductie worden toegepast zoals vermeld in paragraaf 2.4.1. Er wordt geen gebruik gemaakt van mutatie. In standaard GP wordt uiteraard ook geen gebruik gemaakt van de lokale optimalisatieoperator waardoor deze probabiliteit gelijk is aan nul. Wanneer we LOSE wel gebruiken, veranderen

eveneens de probabiliteiten van de andere operatoren. Reproductie zal echter steeds vijf individuen kopiëren naar de volgende generatie om convergentie te vrijwaren en het verlies van kwalitatief goede oplossingen te vermijden. De overgebleven plaatsen in de nieuwe populatie worden opgevuld door LOSE en kruising, en zijn afhankelijk van de instelling voor LOSE.

4.3.3 De instelling van de probabiliteit

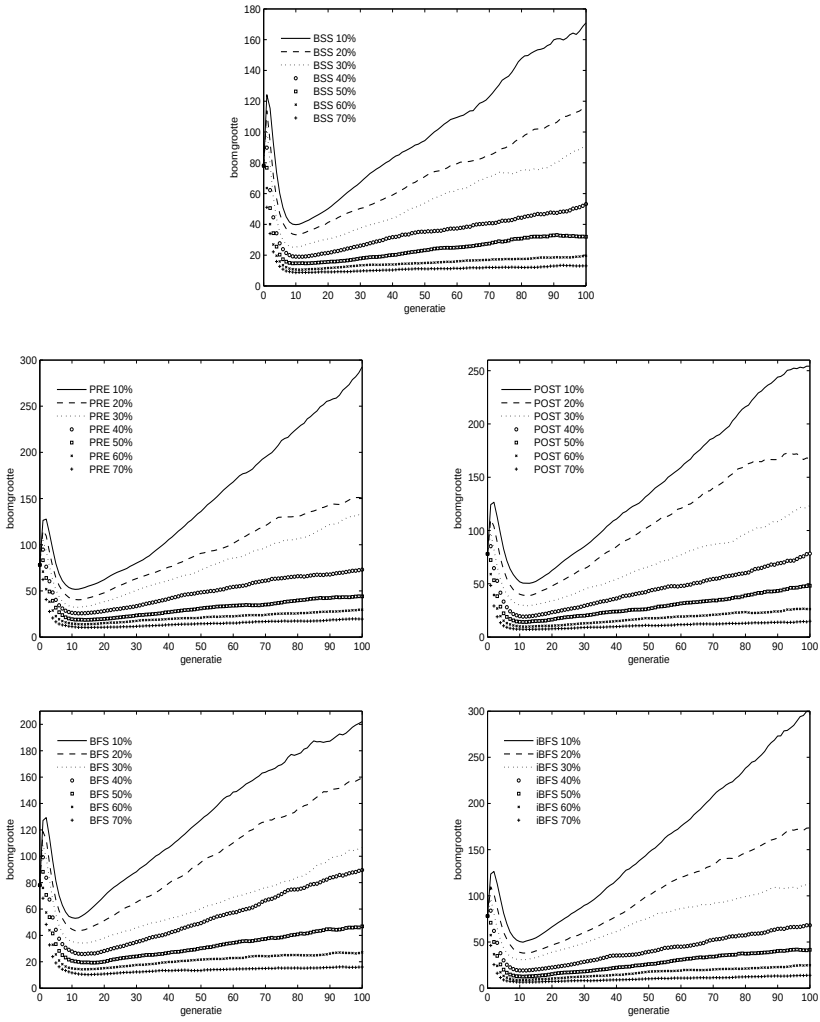
De belangrijkste instelling van de lokale optimalisatieoperator is de frequentie waarmee de operator wordt toegepast. Deze instelling moet doordacht gekozen worden aangezien LOSE de boomstructuur van een individu grondig kan wijzigen. Het is immers niet ondenkbaar dat, wanneer LOSE zeer frequent wordt toegepast, de gemiddelde boomgrootte snel zal afnemen en dat bijgevolg kostbare informatie uit de populatie zal verdwijnen.

experiment	p_{kruising}	$p_{\text{reproductie}}$	p_{lose}
1	90%	5 beste individuen	10%
2	80%	5 beste individuen	20%
3	70%	5 beste individuen	30%
4	60%	5 beste individuen	40%
5	50%	5 beste individuen	50%
6	40%	5 beste individuen	60%
7	30%	5 beste individuen	70%

Tabel 4.1: *De verschillende instellingen van de probabiliteit van LOSE, kruising en reproductie.*

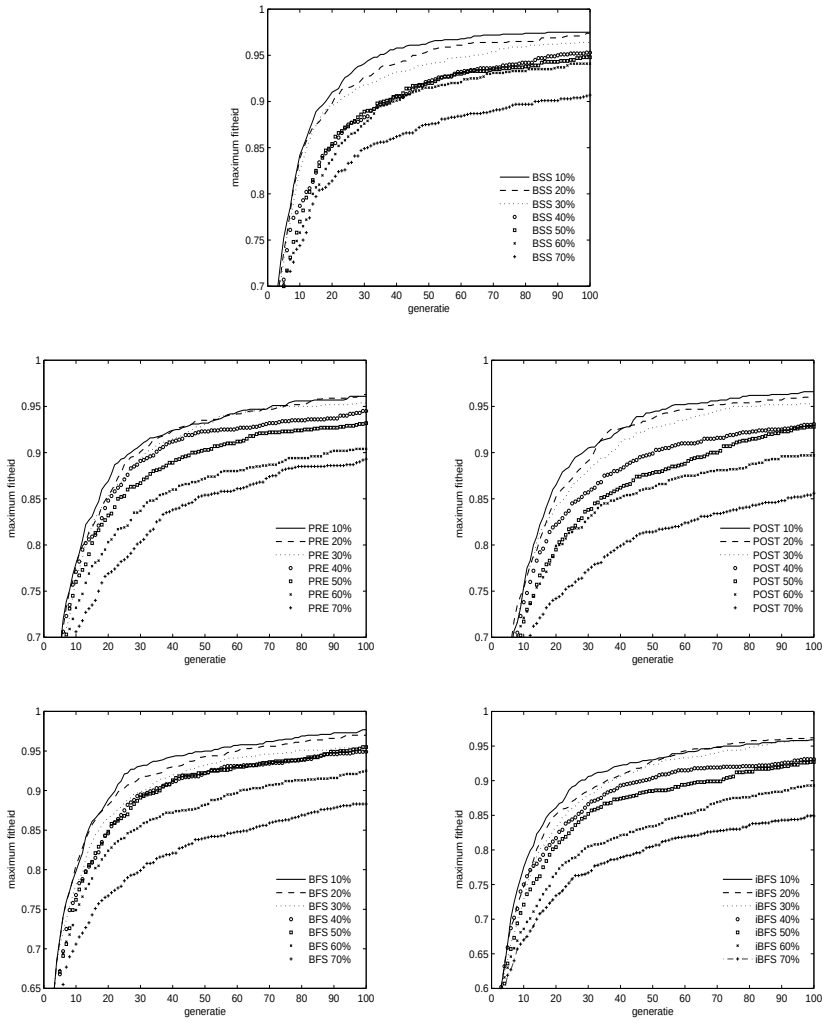
Om de invloed van deze parameter na te gaan en een ondoordachte keuze te vermijden, voeren we eerst een beperkt experiment uit op basis van slechts 50 verschillende simulaties (i.p.v. de normale 100). Zo beperken we de benodigde rekentijd maar hebben we toch voldoende experimentele data om een betrouwbare conclusie te formuleren. Voor de eenvoud wordt de probabiliteit waarmee LOSE wordt toegepast, afgekort tot p_{lose}. Voor elk van de vijf zoekstrategieën (BSS, PRE, POST, BFS en iBFS) laten we de probabiliteit van LOSE stapsgewijs variëren van 0% tot en met 70% in stapjes van 10% (zoals weergegeven in Tabel 4.1). Instelwaarden hoger dan 70% hebben geen nut aangezien de reductie in boomgrootte die LOSE teweegbrengt te ingrijpend is om nog competente individuen te genereren. Om de overdaad aan grafieken te beperken, tonen we enkel de resultaten bij de kunstmatige mier.

Figuren 4.3 en 4.4 tonen de gemiddelde boomgrootte en maximum fitheid voor alle vijf zoekstrategieën bij stijgende p_{lose}.



Figuur 4.3: *De gemiddelde boomgrootte bij verschillende instelwaarden van p_{lose} voor alle zoekmethoden bij de kunstmatige mier.*

Een eerste vaststelling (geldig voor alle benchmarktoepassingen) is dat bij toenemende waarde van p_{lose} de boomgrootte daalt (Fig. 4.3). Dit is uiteraard eenvoudig te verklaren aan de hand van de werking van de lokale optimalisatieoperator. Hoe frequenter LOSE wordt toegepast, hoe meer indi-



Figuur 4.4: *De maximum fitheid bij verschillende instelwaarden van p_{lose} voor alle zoekmethoden bij de kunstmatige mier.*

viduen vervangen worden door kleinere (en meer fitte) boomstructuren. Men mag echter niet verkeerdelijk besluiten dat hoe hoger de instelling van p_{lose}, hoe beter het resultaat. Want in combinatie met de grafieken uit Figuur 4.4 zien we ook dat maximum fitheid daalt naarmate p_{lose} stijgt.

Omgekeerd, een kleine instelwaarde voor p_{lose} zorgt in de eerste plaats dan weer voor een langere rekentijd. Vooral bij BSS kan dit onaanvaardbaar hoog

oplopen. Alhoewel dit niet geldig is bij alle toepassingen en voor alle deelboomselectiemethoden, lijkt het alsof maximum fitheid stijgt bij afnemende p_{lose}. De verschillen zijn echter vaak niet statistisch significant.
Een 'goede' waarde voor p_{lose} is vanzelfsprekend afhankelijk van de probleemstelling aangezien we de fitness cases gebruiken om geschikte deelbomen te vinden. Bij de kunstmatige mier stelt het GP systeem zich relatief tolerant op tegenover hoge probabiliteiten. Pas vanaf 70% kunnen we een duidelijke verlaging in maximum fitheid vaststellen en hebben we een duidelijke aanwijzing om te stellen dat er iets fout loopt. Bij de multiplexer en het regressieprobleem moeten we een lagere p_{lose} gebruiken. We gebruiken dan ook verschillende instelwaarden voor LOSE bij verschillende benchmarktoepassingen zoals weergegeven in Tabel 4.2. Om de verschillende deelboomselectiestrategieën onderling zo eerlijk mogelijk te kunnen vergelijken, kiezen voor een instelling van p_{lose} die enkel afhankelijk is van de toepassing (en dus gelijk voor elke deelboomselectiemethode).

kunstmatige mier

	p_{lose}	p_{kruising}
BSS, PRE, POST, BFS, iBFS	50%	50%

vierde orde regressie

	p_{lose}	p_{kruising}
BSS, PRE, POST, BFS, iBFS	40%	60%

11-bit multiplexer

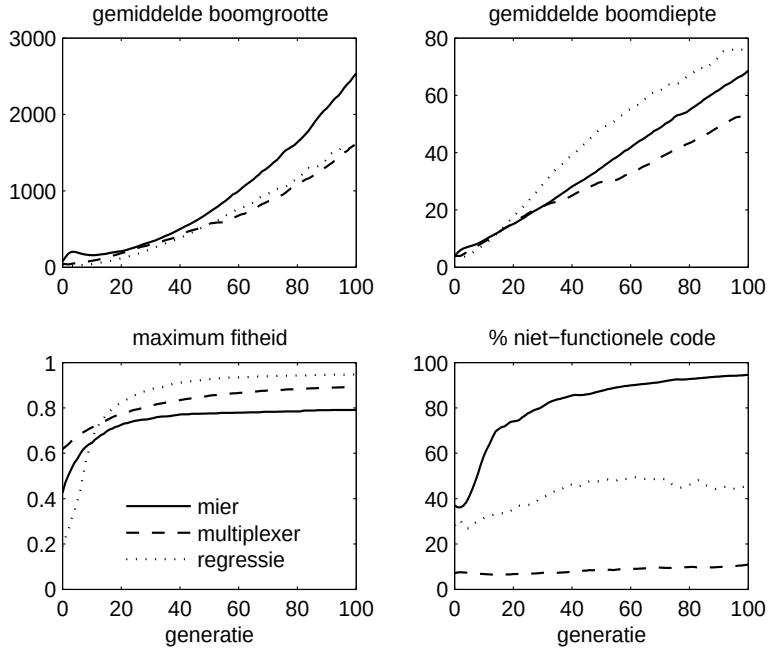
	p_{lose}	p_{kruising}
BSS, PRE, POST, BFS, iBFS	40%	60%

Tabel 4.2: *De instelwaarde van p_{lose} voor de verschillende benchmarktoepassingen. Reproductie zal steeds de 5 beste individuen uit de populatie kopiëren naar de volgende generatie.*

Een belangrijk detail: bij éénzelfde instelling voor p_{lose} produceert BSS steeds kleinere individuen in vergelijking met de andere methoden. Ook de fitheid is steeds minstens gelijk (en vaak zelfs hoger) aan de fitheid bij de andere methoden. Dit is niet zo verwonderlijk aangezien we bij BSS op zoek gaan naar de beste deelboom (d.m.v. een gulzig zoekalgoritme). Meer verwonderlijk is de kleinere omvang van de boomstructuren bij éénzelfde instelwaarde.

4.4 Standaard GP

We starten de bespreking van de resultaten met een overzicht van hoe een standaard GP (SGP) systeem zich gedraagt wanneer geen lokale optimalisatie noch diepte- of boomgroottebegrenzers worden gebruikt. Figuur 4.5 toont de gemiddelde boomgrootte en diepte, maximum fitheid en het aandeel niet-functionele code in de boomstructuur voor elk van de benchmarktoepassingen.



Figuur 4.5: *Gemiddelde boomgrootte en diepte, maximum fitheid en het aandeel niet-functionele code bij genetisch programmeren zonder groottebegrenzers voor alle benchmarktoepassingen.*

Vanaf generatie 20 stijgt de gemiddelde boomgrootte en boomdiepte zeer snel bij alle toepassingen. Deze snelle groei gaat niet gepaard met een corresponderende positieve evolutie van maximum fitheid. Codegroei doet duidelijk zijn intrede en remt het vinden van steeds betere kandidaat-oplossingen gevoelig af. We overlopen kort enkele toepassingsafhankelijke vaststellingen.

4.4.1 De kunstmatige mier

Bij de kunstmatige mier groeit het aantal knopen bijzonder snel. Dit is gedeeltelijk te wijten aan het grotere aantal kinderen per functieknoop (gemiddeld 2.3 per functie in vergelijking met 2 bij het regressieprobleem en de multiplexer). Hierdoor is de gemiddelde boomgrootte in de startpopulatie groter. In de initiële populatie moet men immers enkel rekening houden met een dieptehelling. Er staat geen maximum op het aantal knopen.

Een andere typerend kenmerk van deze toepassing is de initiële stijging gevolgd door een plotse daling tijdens de eerste generaties. Grotere individuen hebben meer knopen en bijgevolg meer eindknopen. Hierdoor zullen deze individuen meer terrein bestrijken dan kleinere en meer compacte kandidaatoplossingen waardoor hun fitheid groter zal zijn. Er worden immers meer posities op het rooster afgetast. Hierdoor worden deze individuen dan weer vaker geselecteerd als ouder waardoor de code initieel zal toenemen. Naarmate de selectiedruk echter stijgt, zal evolutie meer gericht zoeken naar individuen die het Santa-Fe spoor accurater volgen (individuen die dus een spoorvolggedrag vertonen). Dit gaat ten koste van grote, logge individuen die enkel een hoge fitheidswaarde behalen omdat ze veel knopen kunnen uitvoeren. Hierdoor zal de boomgrootte (tijdelijk) opnieuw dalen.

Naast een sterke toename van de boomgrootte, springt ook het procentueel aandeel niet-functionele code onmiddellijk in het oog. Op generatie 100 draagt maar liefst 95% van de code niet wezenlijk bij tot de functionaliteit van een individu. Dit is in hoofdzaak te wijten aan talloze opeenvolgende combinaties van IF-FOOD-AHEAD functieknopen zoals beschreven in 2.4.3.

Het is dan ook niet verwonderlijk dat maximum fitheid snel stijgt gedurende de eerste generaties en vervolgens (wanneer de boomgrootte toeneemt) snel uitvlakt en nauwelijks verandert. Slechts 11 van de 100 simulaties produceert een 100% correcte oplossing die alle voedsel verzamelt.

4.4.2 Het vierde orde regressieprobleem

Het regressieprobleem produceert vooral diepe bomen, veroorzaakt door het beperkte aantal kinderen per functieknoop. Wanneer we de boomstructuur van een aantal individuen naderbij bekijken, dan stellen we vast dat genetisch programmeren vooral probeert om de doelfunctie te benaderen dan wel exact voor te stellen. Een ellenlange combinatie van willekeurige constanten en wiskundige operatoren brengt de boomstructuur generatie na generatie dichter bij de correcte oplossing. Tussen twee opeenvolgende genera-

ties is de verbetering vaak miniem en in hoofdzaak te wijten aan code die nauwelijks bijdraagt tot de prestatie (bijvoorbeeld een zeer kleine constante 0.0000000234).

Het procentueel aantal niet-functionele knopen is relatief hoog maar wel beperkter dan bij de mier. De reductieregels zijn immers vaak complexer dan bij de mier en niet alle redundante codefragmenten worden uit de boom verwijderd (omwille van de sterk oplopende rekentijd bij verificatie van commutativiteit en associativiteit). Vaak voorkomende stukken niet-functionele code zijn: $x - x(= 0)$ en $\frac{x}{x}(= 1)$ en vooral de reductie van wiskundige operaties met enkel constanten.

Maximum fitheid bij het regressieprobleem neemt een bescheiden start en eindigt rond 0.95 wat een zeer hoge waarde is. Ondanks deze hoge waarde is het toch wat verwonderlijk dat slechts 47 van de 100 simulaties een 100% correcte oplossing voortbrengen.

4.4.3 De 11-bit multiplexer

Het multiplexerprobleem genereert individuen die gemiddeld meer knopen per niveau bevatten in vergelijking met de andere twee toepassingen (minder diepe individuen en beter opgevuld). Gelet op de structuur van de correcte oplossing uit paragraaf 2.3, zal GP 'brede' individuen een hogere kans op overleven bieden. Dergelijke individuen bevatten in de bovenste lagen van de boom de functies IFTE die de inhoud van de adreslijnen zal gebruiken om de correcte uitvoer te genereren.

De boomstructuren bevatten veel minder niet-functionele code in vergelijking met de andere benchmarktoepassingen. Het grotere aantal functies en vooral eindknopen (terminalen) speelt hierbij een belangrijke rol (Luke 2000*a*). Toch moeten we voorzichtig zijn bij de interpretatie van het procentueel aandeel niet-functionele knopen. In vergelijking met de twee andere toepassingen zijn er immers minder reductieregels van toepassing bij de multiplexer. Bovendien zijn deze regels vaak gecompliceerder en bijzonder computationeel intensief om te berekenen. Denk bijvoorbeeld maar aan (DEELBOOM AND (NOT DEELBOOM)) of aan (DEELBOOM OR (NOT DEELBOOM)). Gelukkig komen deze exotische gevallen in de praktijk bijzonder weinig voor.

Uit deze resultaten blijkt dat deze toepassing een zeer moeilijk op te lossen probleem is voor standaard GP. De fitheid start reeds vrij hoog maar evolueert zeer gestaag tot een eindwaarde van 0.89. Geen enkele simulatie is erin geslaagd om een correcte oplossing te produceren.

4.5 GP en LOSE met statische instelwaarde

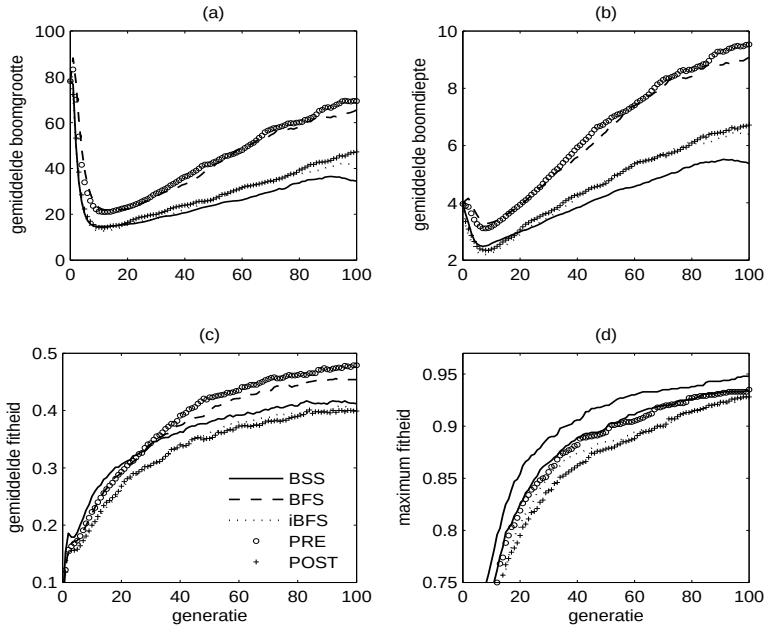
In deze paragraaf worden de resultaten van GP in combinatie met de lokale optimalisatieoperator besproken. We behandelen telkens de evolutie van de boomgrootte en maximum fitheid en maken een vergelijking van de verschillende zoekstrategieën voor het vinden van een geschikte deelboom. Tevens vergelijken we met standaard GP (zonder begrenzing van de boomgrootte). Pas in paragraaf 4.6 zullen we LOSE vergelijken met andere (recente) technieken om codegroei te bestrijden. De kunstmatig mier wordt gebruikt om de werking van LOSE te verduidelijken. Bij deze toepassing wordt een aantal zaken in meer detail toegelicht. Bij de andere toepassingen beperken we ons in hoofdzaak tot een weergave en bespreking van de resultaten. Uiteraard worden probleemspecifieke bevindingen wel uitgebreid toegelicht.

4.5.1 De statistische toetsen

Om verschillen van éénzelfde variabele (bijvoorbeeld boomgrootte) tussen twee methoden (bijvoorbeeld standaard GP en LOSE) te kunnen detecteren, gebruiken we statistische toetsen. We gebruiken hoofdzakelijk niet-parametrische methoden om ervoor te zorgen dat we ons zo onafhankelijk mogelijk opstellen ten aanzien van de onderliggende distributie van de data. Vaak is de data niet normaal verdeeld zodat we de meeste parametrische methoden niet kunnen aanwenden. De Wilcoxon rang toets wordt gebruikt om significante verschillen tussen twee populaties op te sporen. Voor meerdere groepen kiezen we voor de Kuskal-Wallis toets. Deze statistiek vormt een alternatief op een ANOVA analyse met één verklarende variabele. Om vervolgens na te gaan tussen welke twee groepen er een significant verschil bestaat gebruiken we opnieuw de Wilcoxon rang toets. Omwille van de herhaaldelijke significantietoetsen met dezelfde verklarende variabelen, is het aangewezen een strenger significantieniveau te hanteren. Daarom passen we de Bonferroni-correctie toe. Het significantieniveau wordt daarom gedeeld door het aantal paarsgewijze vergelijkingen. Aangezien weinig statistische methoden bestaan om tijdsreeksen onderling te vergelijken, passen we deze toetsen enkel toe op data uit de laatste generatie (generatie = 100) in plaats van op de volledige curve. We hanteren steeds een α waarde van 5%. Meer informatie over de gebruikte statistische toetsen is te vinden in (Walpole & Myers 1978, Higgins 2003).

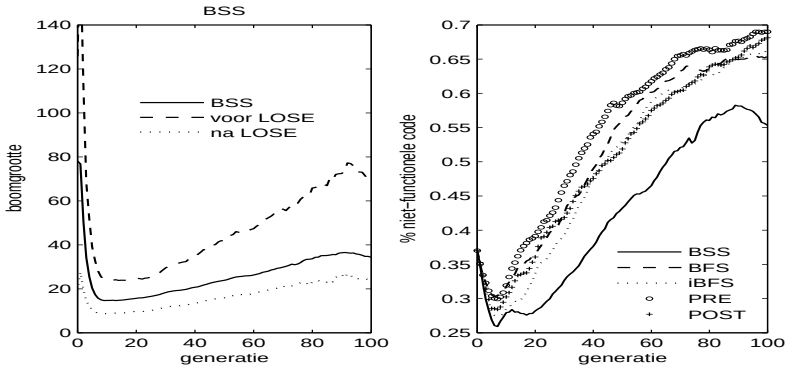
4.5.2 De kunstmatige mier: een modelvoorbeeld

Bij de bepaling van de instelprobabiliteit voor LOSE, is gebleken dat de kunstmatige mier zich zeer tolerant opstelt tegenover grote waarden voor p_{lose}. We hebben dan ook geopteerd om LOSE gedurende 50% van de tijd toe te passen (onafhankelijk van de keuze van de zoekstrategie). Figuur 4.6 toont de gemiddelde boomgrootte en boomdiepte, alsook de gemiddelde en maximum fitheid voor alle zoekstrategieën.



Figuur 4.6: *Gemiddelde boomgrootte (a) en boomdiepte (b), en gemiddelde (c) en maximum (d) fitheid bij de kunstmatige mier voor de verschillende zoekstrategieën.*

De boomgrootte Op deze figuur is een duidelijk verschil merkbaar met genetisch programmeren zonder gebruik van codegroeibegrenzers (Fig. 4.5). De gemiddelde boomgrootte (en boomdiepte) is bij alle vijf zoekstrategieën (BSS, PRE, POST, BFS en iBFS) significant kleiner in vergelijking met standaard GP (p-waarde is nul). Bij standaard GP bedraagt de gemiddelde boomgrootte op generatie 100 maar liefst 2536 knopen in tegenstelling tot GP in combinatie met LOSE waar de boomgrootte (uitgemiddeld over alle zoekstrategieën) slechts 51 knopen bedraagt! Dit is een reductie met een factor



Figuur 4.7: *De linkse grafiek toont de gemiddelde boomgrootte van individuen die door LOSE + BSS werden geselecteerd (voor en na verwerking) en de gemiddelde boomgrootte van alle individuen uit de populatie. Voor de resterende vier zoekstrategieën werden analoge resultaten behaald. De rechtse grafiek toont het procentueel aandeel niet-functionele code in de boomstructuur voor alle zoekstrategieën (*$\times 100\%$*).*

gelijk aan 49. De reductie is het grootst bij BSS en is significant verschillend van PRE en BFS.

Op Figuur 4.7 is te zien hoe de lokale optimalisatieoperator individuen selecteert waarvan de boomstructuur gemiddeld groter is dan de gemiddelde boomgrootte over alle individuen uit de populatie. LOSE zal de boomstructuur snoeien en verkleinen afhankelijk van de diepte waarop de wortel van de deelboom ligt die door één van bovenstaande zoekstrategieën werd gevonden. Aangezien, volgens de definitie van een deelboom, de wortel minstens één niveau dieper zal liggen dan de wortel van de volledige boom, zal de diepte van het nieuwe individu steeds kleiner zijn en zal dus ook de boomgrootte afnemen.

Waarom LOSE nu precies de groter-dan-gemiddelde individuen uit de populatie selecteert, kan als volgt worden verklaard. Een grote boomstructuur heeft meer interne knopen (en dus meer deelbomen) die in aanmerking komen om bezocht te worden door één van de vijf zoekstrategieën. Door dit grotere aantal kandidaat deelbomen vergroot uiteraard de kans dat er 'één' deelboom bestaat, die een fitheid heeft die groter is dan de fitheid van het volledige genoom. Dus grotere bomen hebben meer kans om opgenomen te worden in de selectielijst. Dit effect is zeer duidelijk in de eerste generaties aangezien daar de gemiddelde boomgrootte het grootst is (zie Fig. 4.6a).

In tegenstelling tot Figuur 4.5a treedt de initiële stijging van de boomgrootte nu niet meer op: door de lokale optimalisatie is er geen drijfveer die grotere individuen extra zal bemoedigen omdat ze dankzij het groter aantal knopen meer terrein bestrijken. Integendeel, LOSE selecteert de grotere individuen vaker omwille van de verhoogde kans op het vinden van een goede deelboom. Bovendien zorgt de hoge instelwaarde van p_{lose} (50%) er eveneens voor dat zeer veel individuen uit de selectielijst door LOSE effectief worden gereduceerd in grootte. Hierdoor zal de gemiddelde boomgrootte kleiner zijn in vergelijking met standaard GP.

Bij de bespreking van standaard genetisch programmeren uit voorgaande paragraaf, werd aangetoond dat de kunstmatige mier zeer veel code bevat die niet (of nauwelijks) bijdraagt tot de kwaliteit van het individu. Figuur 4.7 (rechtse grafiek) toont het procentueel aandeel niet-functionele code wanneer LOSE wordt gebruikt. Er valt op te merken dat, ondanks de aanhoudende aanwezigheid van veel overtollige knopen, het aandeel niet-functionele code lichtjes is gedaald (rond generatie 100 met gemiddeld 30%). Het is opmerkelijk dat bij het gebruik van BSS het aandeel niet-functionele knopen opvallend kleiner is dan bij de andere methoden.

Door de aanwezigheid van een groot aantal niet-functionele knopen is er dus nog ruimte om de boomstructuur verder te optimaliseren. Bovendien lijkt het alsof de code vanaf generatie 40 opnieuw lichtjes begint te groeien (alhoewel er een duidelijke stijging in maximum fitheid mee gepaard gaat). Misschien is het noodzakelijk om de probabiliteit p_{lose} alsnog op te drijven. We zullen deze mogelijkheid verder uitdiepen in hoofdstuk 6.

Maximum fitheid Een belangrijk nadeel verbonden aan veel codegroeiremmende methoden is de negatieve evolutie van fitheid. We zijn dus bijzonder geïnteresseerd in het effect van LOSE op de evolutie van (i.h.b. maximum) fitheid weergegeven in Figuur 4.6d.

Alle methoden hebben een maximum fitheid die significant hoger ligt dan de maximum fitheid bij standaard GP (p-waarde is nul). Tussen de verschillende deelboomselectiestrategieën onderling is er geen significant verschil, alhoewel BSS net iets beter scoort. Wanneer we het aantal simulaties (100 in totaal) vergelijken die een correcte oplossing geven voor de kunstmatige mier, merken we dat dit aantal maar liefst 67 bedraagt voor BSS in vergelijking met 11 bij standaard GP! Ook BFS (53), iBFS (54), PRE(59) en POST (57) doen opmerkelijk beter.

Uit de resultaten blijkt dat de gemiddelde en maximum fitheid van alle individuen uit de populatie hoger is dan de gemiddelde en maximum fitheid van alle deelbomen uit de populatie (enkel die deelbomen die door de deelboomselectiestrategie zijn geselecteerd om eventueel door LOSE gebruikt te worden). Dit betekent dat LOSE niet louter zorgt voor een optimalisatie van grote logge boomstructuren tot kleinere fittere exemplaren maar ook dat er in combinatie met kruising een meerwaarde wordt verkregen.

De prestatie van de verschillende deelboomselectiestrategieën Zoals we daarnet reeds hebben vermeld produceren alle zoekstrategieën zeer goede resultaten; maximum fitheid is bij alle methoden hoog en statistisch niet verschillend ten opzichte van elkaar. Wel kunnen we (statistisch gezien) een onderscheid maken tussen de gemiddelde boomgrootte bij (PRE, BFS) en bij de overige drie methoden BSS, iBFS en POST. Op zich is het niet zo verwonderlijk dat PRE en BFS in éénzelfde groep zitten aangezien hun manier van knopen bezoeken sterke verwantschap vertoont. Dezelfde stelling gaat op voor iBFS en POST.

We vermoeden dat er binnen éénzelfde individu verschillende deelbomen beschikbaar zijn waarvan de fitheid hoger is dan de fitheid van de volledige boom[2]. Veronderstellen we dat de geschikte deelbomen quasi uniform verspreid liggen binnen de boomstructuur, dan produceren iBFS en POST dus kleinere bomen dan PRE en BFS omdat beide methoden de lager gelegen knopen vroeger bezoeken. De instelling van p_{lose} is een bijkomende reden waarom de gemiddelde boomgrootte kleiner is. Experimenten met nog hogere instelwaarden hebben immers uitgewezen dat vanaf 60% de bekomen individuen bij POST en iBFS te klein waren om als oplossing te fungeren.

De deelbomen die door BSS worden gevonden, dragen minder niet-functionele code met zich mee in vergelijking met de andere strategieën (Fig. 4.7, rechtse grafiek). We verduidelijken dit aan de hand van volgend voorbeeld. Veronderstel de volgende boomstructuur (IFA = IF-FOOD-AHEAD):

```
(PROGN2 (IFA RIGHT
             (IFA MOVE
                  (IFA MOVE RIGHT)))
        LEFT)
```

Nemen we aan dat de fitheid van de (equivalente) deelbomen (IFA MOVE (IFA MOVE RIGHT)) en (IFA MOVE RIGHT) groter is dan de fitheid van de volledi-

[2]Dit vermoeden werd later bevestigd (zie hoofdstuk 6), althans voor de initiële generaties.

ge boom. Een methode die van boven naar onderen toe werkt en stopt zodra een deelboom wordt gevonden met een fitheid groter of gelijk aan de fitheid van de volledige boom zal in dit geval stoppen bij de deelboom (IFA MOVE (IFA MOVE RIGHT)). BSS zal nog één niveau dieper afdalen tot de deelboom (IFA MOVE RIGHT) die dezelfde fitheid heeft. Hierdoor is de boom niet alleen kleiner maar is de kans op niet-functionele knopen eveneens kleiner. De deelboom van BSS bevat in dit concrete voorbeeld geen niet-functionele knopen terwijl de andere deelboom kan gereduceerd worden tot (IFA MOVE RIGHT).

De compactheid van correcte oplossingen De gemiddelde boomgrootte van het beste individu op generatie 100 is gelijk aan 67 voor BSS, 133 voor BFS, 87 voor iBFS, 110 voor PRE en 98 voor POST. Wanneer we enkel de simulaties bekijken die een 100% correcte oplossing hebben voortgebracht dan zakt de boomgrootte van het beste individu tot 42 voor BSS, 54 voor BFS, 36 voor iBFS, 60 voor PRE en 49 voor POST. Met behulp van de vereenvoudigingsmodule uit paragraaf 2.4.3, kunnen we de boomstructuren ontdoen van alle niet-functionele code. Als gevolg hiervan bekomen we nog compactere individuen met een gemiddelde boomgrootte van 19 voor BSS, iBFS, PRE en POST en 22 voor BFS[3]. We kunnen veilig stellen dat LOSE in staat is om compacte en inherent ook leesbare programma's te creëren.

Samenvatting van de resultaten De kunstmatige mier is een toepassing die erg te lijden heeft onder een sterke toename van de code. Bovendien kan de boomstructuur drastisch vereenvoudigd worden, m.a.w. de boom bevat gemiddeld een groot aantal niet-functionele knopen. Vooral de impact van codegroei op maximum fitheid is zeer groot. De resultaten tonen aan dat LOSE (met om het even welke zoekstrategie) met succes slaagt in het reduceren van de boomgrootte terwijl maximum fitheid gevoelig wordt verhoogd. Wanneer enkel de simulaties die een correcte oplossing genereren in beschouwing genomen worden, geeft LOSE (eventueel in combinatie met een vereenvoudigingsmodule) compacte oplossingen die opnieuw leesbaar en makkelijk hanteerbaar zijn.

[3]De kleinste oplossing (met het minste aantal knopen) werd ontdekt door Poli & Langdon (2002) en bevat slechts 11 knopen.

4.5.3 Het vierde orde regressieprobleem: 90% kans op succes

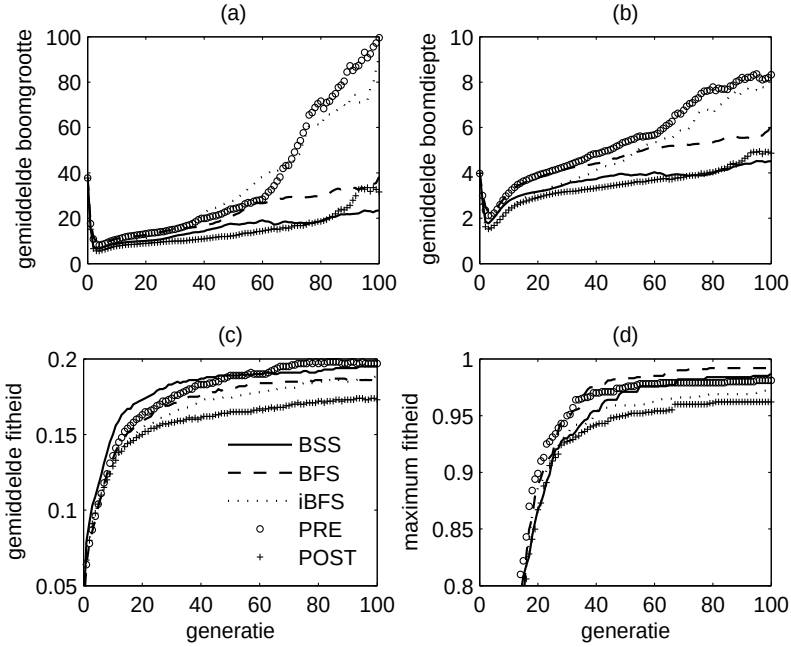
Wanneer we standaard genetisch programmeren toepassen op het vierde orde regressieprobleem dan worden reeds goede resultaten behaald (maximum fitheid van net geen 0.95). Afgaande op de gemiddelde boomgrootte en de boomstructuur van de bekomen oplossingen, stellen we echter vast dat standaard GP dergelijke goede resultaten behaalt door de doelfunctie te benaderen met behulp van een groot aantal constanten[4].

LOSE slaagt vrij snel in het vinden van een correcte oplossing voor dit probleem waardoor (onverwacht) de maximum fitheid nog hoger zal liggen in vergelijking met standaard GP. Op generatie 60 produceert nagenoeg 80% van alle simulaties een correcte oplossing. De simulaties die er niet in geslaagd zijn om een correcte oplossing te vinden: zijn ofwel vroegtijdig geconvergeerd naar een (te) kleine suboptimale oplossing onder invloed van een (te) hoge instelwaarde voor p_{lose}, ofwel tracht GP de doelfunctie te benaderen waardoor de boomgrootte enorm toeneemt. Deze laatste mogelijkheid komt vaker voor. Helaas heeft dit een enorme impact op de gemiddelde boomgrootte en boomdiepte zoals u zal merken in de grafische voorstelling van de resultaten. Niet alleen de grafieken maar eveneens de statistische toetsen die testen op significante verschillen tussen de gemiddelden worden hierdoor beïnvloed. Om dit probleem te vermijden, gebruiken we een gemiddelde dat berekend wordt op basis van een gereduceerde verzameling van simulaties. We zullen de sterk afwijkende waarden niet in beschouwing nemen bij het uitvoeren van de statistische toets.

De boomgrootte Figuur 4.8 toont de gemiddelde boomgrootte en boomdiepte en de gemiddelde en maximum fitheid voor alle vijf deelboomselectiestrategieën. Alle methoden produceren bomen die significant kleiner zijn (p-waarde is nul) dan de boomstructuren die door standaard GP worden gegenereerd. Uit de resultaten blijkt vooral de verschillende aanpak van beide genetische programma's. Standaard GP probeert om de doelfunctie te benaderen en overlaadt de boomstructuur met kleine stukjes code die zeer kleine verschuivingen in de fitheid veroorzaken en de geëvolueerde curve stapje voor stapje dichter bij de echte doelfunctie brengen. Het is dan ook niet verwonderlijk dat de boom gemiddeld niet minder dan 1578 knopen telt op generatie 100.

LOSE daarentegen zoekt naar compacte oplossingen die de doelfunctie exact voorstellen. Zeer vaak wordt een perfecte oplossing snel gevonden (op gene-

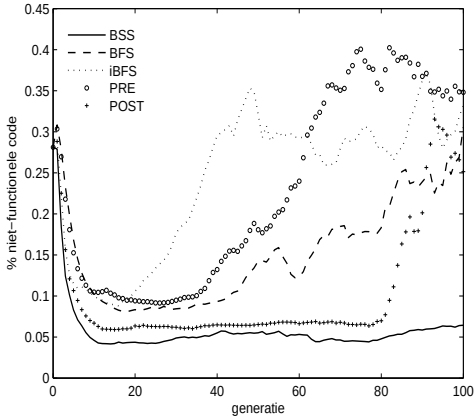
[4]De boom bevat een groot aantal pseudo-intronen (Luke 2003).



Figuur 4.8: *Gemiddelde boomgrootte (a) en boomdiepte (b), en gemiddelde (c) en maximum (d) fitheid bij vierde orde regressie voor de verschillende zoekstrategieën.*

ratie 15 reeds 50 correcte oplossingen) waardoor de gemiddelde boomgrootte zeer beperkt is. Het algoritme stopt immers zodra een perfecte oplossing wordt gevonden waardoor de overige individuen niet meer de kans krijgen om zich verder te ontwikkelen. Op generatie 100 is de gemiddelde boomgrootte gemiddeld over alle zoekstrategieën dan ook gelijk aan 57.

Na zoals bij de kunstmatige mier is het aandeel niet-functionele knopen na toepassing van LOSE lichtjes gedaald (Fig. 4.9), althans gedurende de eerste 50 generaties. Vanaf dan schommelt het procentueel aandeel ten gevolge van een aantal simulaties die de doelfunctie trachten te benaderen. Het aantal niet-functionele knopen in dat soort kandidaat-oplossing is natuurlijk bijzonder groot (soms zelfs tot 70%). Wanneer we ons enkel beperken tot de simulaties die een correcte oplossing genereren dan is het aandeel niet-functionele code zeer beperkt. De gemiddelde boomgrootte van alle individuen is immers zeer beperkt en zoals we in volgende paragrafen zullen zien zijn de meeste



Figuur 4.9: *Het procentuele aandeel niet-functionele code bij vierde orde regressie voor de verschillende zoekstrategieën. De schommelingen die optreden tijdens de latere generaties (vooral bij de methoden BFS, PRE en iBFS) zijn te wijten aan het beperkte aantal simulaties die de oplossing trachten te benaderen en zo variërende hoeveelheden niet-functionele code produceren.*

oplossingen van de minimum lengte. Deze bomen bevatten bijgevolg enkel functionele knopen.

Net zoals bij de kunstmatige mier selecteert LOSE ook hier individuen waarvan de boomgrootte groter is dan de gemiddelde boomgrootte van alle individuen uit de populatie. Deze resultaten zijn geldig voor alle selectiestrategieën maar worden hier niet in detail behandeld. Voor een uitgebreide discussie verwijzen we naar de bespreking van de kunstmatige mier.

Maximum fitheid Ook in deze toepassing produceert LOSE significant fittere individuen dan SGP (p-waarde gelijk aan nul). Tussen de verschillende zoekstrategieën bestaat er geen statistisch verschil (ook niet met POST zoals de grafiek doet vermoeden). Meer opvallend nog is het grote aantal simulaties dat een 100% correcte oplossing voortbrengt. In vergelijking met SGP (47 simulaties), bieden BSS (94), BFS (97), iBFS (91), PRE (91) en POST (86) duidelijk meer correcte oplossingen.

De prestatie van de verschillende zoekstrategiën Zoals we in vorige paragraaf reeds hebben opgemerkt is er geen statistisch verschil wat betreft maximum fitheid tussen de verschillende zoekstrategieën onderling.

Wat betreft de gemiddelde boomgrootte onderscheiden we deze keer twee verschillende groepen: (1) BSS, iBFS en POST en (2) PRE en BFS. De eerste groep produceert de kleinste individuen, PRE en BFS de grootste. Rekening houdend met het grote aantal simulaties dat een correcte oplossing heeft geproduceerd en het gebruikte stopcriterium, hechten we minder belang aan het onderscheid dat wordt gemaakt tussen de verschillende zoekmethoden op basis van boomgrootte. Men kan veilig stellen dat alle zoekmethoden erin slagen om snel en efficiënt de zoekruimte te doorkruisen en dat de bekomen kandidaat-oplossingen zeer klein zijn.

De compactheid van de oplossingen Tabel 4.3 geeft een overzicht van de gemiddelde boomgrootte van het beste individu uit elke simulatie en vergelijkt deze boomgrootte met de gemiddelde boomgrootte van alle individuen uit de populatie. Wat onmiddellijk opvalt, is dat LOSE (alle zoekmethoden) steeds kleinere oplossingen produceert dan SGP. In voorgaande paragrafen hebben we dit reeds besproken voor de gemiddelde boomgrootte van alle individuen uit de populatie maar ook wanneer we enkel de beste individuen beschouwen, moet SGP onderdoen in prestatie. Opnieuw wensen we op te merken dat de boomgrootte van de beste individuen wanneer we alle simulaties in acht nemen, minder representatief is aangezien de enkele simulaties die zeer grote individuen produceren een negatieve invloed uitoefenen op de gegeven resultaten. Wanneer we enkel de simulaties bekijken die een 100% correcte oplossing voortbrengen, dan ligt de boomgrootte (en zeker de boomgrootte van de vereenvoudigde boomstructuur, rijen C en D) zeer dicht in de buurt van de kleinste bestaande oplossing. Deze oplossing bevat slechts 13 knopen en ziet er uit als volgt:

```
(+ X
   (* X
      (+ X
         (* X
            (+ X
               (* X X))))))
```

Evenals bij de kunstmatige mier slaagt LOSE er ook bij het regressieprobleem in om compacte en leesbare oplossingen te produceren.

Samenvatting van de resultaten Het regressieprobleem is niet meteen de moeilijkste toepassing uit de gekozen benchmarksuite. Toch is deze toepas-

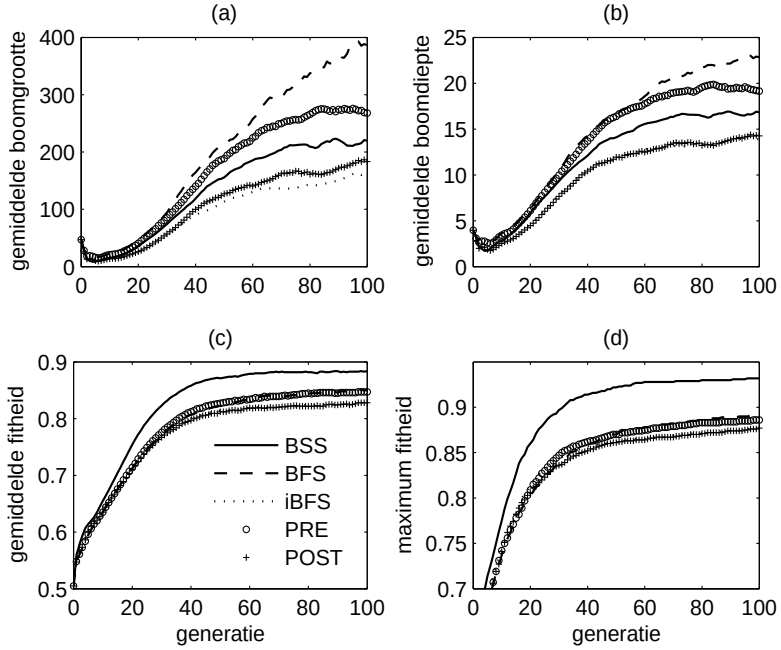
	SGP	BSS	BFS	iBFS	PRE	POST
A	1578	23	38	91	99	32
B	1336	19	49	94	126	46
C	33	14	14	15	15	14
D	21	14	14	13	14	13

Tabel 4.3: *Deze tabel geeft een overzicht van de gemiddelde boomgrootte van alle individuen uit de populatie (A), de boomgrootte van het beste individu (alle simulaties worden in rekening gebracht) (B), de boomgrootte van het beste individu (enkel correcte oplossingen) (C) en tenslotte (D) de boomgrootte van de vereenvoudigde boom uit (C).*

sing verraderlijk zoals standaard GP bewijst. De oplossing op deze toepassing is een uitstekend voorbeeld van Ockham's scheermes die zegt dat de meest eenvoudige voorstelling, meestal de beste (meest generieke) oplossing is. De verleiding is echter groot om de gekozen doelfunctie te benaderen met behulp van een combinatie van constanten en wiskundige operatoren. Standaard GP loopt in deze val en produceert obese individuen. In tegenstelling tot de kunstmatige mier waar de reductie en het behoud van maximum fitheid centraal staat, stellen we ons de vraag of LOSE in staat is om zeer snel en op efficiënte wijze de zoekruimte te doorkruisen op zoek naar een minimale oplossing (oplossing met een zeer beperkt aantal knopen). Uit de resultaten blijkt dat alle zoekmethoden deze doelstelling halen. GP in combinatie met LOSE heeft meestal weinig generaties nodig om een juiste en compacte oplossing te vinden.

4.5.4 De 11-bit multiplexer: een harde noot om te kraken

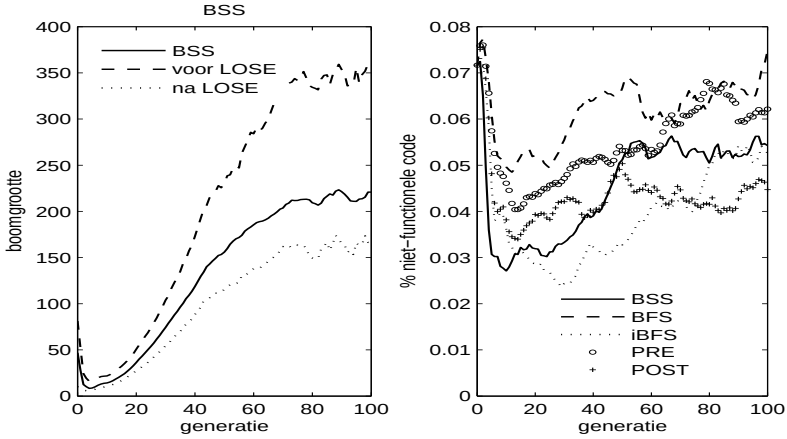
De multiplexer functie is een unieke functie uit een zoekruimte die 10^{616} groot is. Dat maakt deze toepassing meteen de moeilijkste uit het rijtje. Ondanks de verhoogde moeilijkheidsgraad behaalt standaard GP een goede fitheidswaarde op generatie 100. De gemiddelde boomgrootte is echter helemaal niet in verhouding met deze fitheidswaarde en wijkt gevoelig af van de kleinste correcte oplossing die slechts 22 knopen telt (Koza 1992). Niet zozeer de beperkte boomgrootte maar wel de gelijkvormigheid in de boomstructuur is zeer kenmerkend voor deze oplossing. Dit maakt de multiplexer een geschikte kandidaat voor het gebruik van ADF's. Het is de vraag of ook LOSE in staat in om deze similariteit op te merken.



Figuur 4.10: *Gemiddelde boomgrootte (a) en boomdiepte (b), en gemiddelde (c) en maximum (d) fitheid bij de 11-bit multiplexer voor de verschillende zoekstrategieën.*

De boomgrootte Figuur 4.10 toont de gemiddelde boomgrootte en boomdiepte en de gemiddelde en maximum fitheid voor alle vijf zoekstrategieën. Opnieuw produceren alle zoekmethoden genomen die significant kleiner zijn (p-waarde is nul) dan de boomstructuren die door SGP worden gegenereerd. Op generatie 100 is de gemiddelde boomgrootte bij SGP gelijk aan 1611 in vergelijking met 244 (gemiddeld over alle zoekstrategieën) wanneer LOSE wordt gebruikt. Dit is een meer bescheiden reductie met een factor 6.6. Ondanks de reductie in boomgrootte en -diepte is de uiteindelijke (gemiddelde) omvang van de bomen toch relatief hoog. We stellen een snelle initiële daling in de gemiddelde boomgrootte vast (wellicht te wijten aan het uitwieden, door LOSE, van een aantal zeer grote individuen met lage fitheidswaarde), gevolgd door een snelle stijging tot ongeveer generatie 45. Vanaf dan neemt de code nog steeds toe maar wat langzamer.

Uit experimentele resultaten blijkt dat de boomgrootte van het beste individu uit de populatie (per generatie) steeds groter is dan de gemiddelde



Figuur 4.11: *De linkse grafiek toont de gemiddelde boomgrootte van individuen die door LOSE + BSS werden geselecteerd (voor en na verwerking) en de gemiddelde boomgrootte van alle individuen uit de populatie. Voor de resterende vier zoekstrategieën werden analoge resultaten behaald. De rechtse grafiek toont het procentueel aandeel niet-functionele code in de boomstructuur voor alle zoekstrategieën (*$\times 100\%$*).*

boomgrootte. Wellicht is de combinatie van de stijgende boomgrootte van het meest fitte individu en de convergentie van andere oplossingen naar dit (sub)optimale individu de oorzaak van een sterke toename gedurende de eerste generaties (tot generatie 45). Dan neemt de selectiedruk af (veel individuen worden naar het suboptimale individu toegetrokken) en vertraagt de groei van de boomgrootte (gebaseerd op de theorie uit paragraaf 3.3.4). We vermoeden dat reeds vroeg in de evolutie een suboptimaal individu wordt ontdekt waar andere individuen worden naartoe getrokken.

Net als bij de kunstmatige mier selecteert LOSE ook bij het multiplexerprobleem individuen die groter zijn dan de gemiddelde boomgrootte over alle individuen uit de populatie (Fig. 4.11). De resultaten geven enkel de zoekmethode BSS weer maar zijn eveneens geldig voor de overige vier zoekstrategieën. Vooral in de tweede helft van de simulatie zal LOSE groter dan gemiddelde individuen selecteren en reduceren tot meer compacte genomen (dit gebeurt ook tijdens de eerste generatie en verklaart de snelle initiële daling zoals beschreven in voorgaande paragraaf). De discrepantie in boomgrootte tussen de individuen die door LOSE worden geselecteerd (voor en na) en de gemiddelde boomgrootte van alle individuen uit de populatie neemt toe. Dit

is één van de redenen waarom de groei van de code lichtjes afremt vanaf generatie 45 en de gemiddelde boomgrootte minder snel toeneemt dan ervoor.

Het aandeel niet-functionele code, alhoewel zeer beperkt, is eveneens gedaald in vergelijking met SGP. De curven vertonen heel wat schommeling, maar de veranderingen in amplitude zijn zeer beperkt gelet op de fijne schaalverdeling in Figuur 4.11 (rechtse grafiek).

Maximum fitheid De voorspelde verhoogde moeilijkheidsgraad van deze toepassing blijkt te kloppen. Enkel BSS slaagt erin om significant betere resultaten te behalen dan SGP. De andere methoden produceren analoge resultaten, vergelijkbaar met de maximum fitheid van SGP. Merk echter wel op dat de andere zoekmethoden individuen genereren die véél minder knopen bevatten dan de genomen die door SGP worden geproduceerd. Hierdoor kunnen we stellen dat ook de andere methoden betere resultaten produceren.

Wanneer we het aantal simulaties vergelijken (100 in totaal) die een correcte oplossing geven voor de multiplexerschakeling, merken we dat dit aantal 19 bedraagt voor BSS in vergelijking met nul bij standaard GP! BFS (6), iBFS (3), PRE(5) en POST (3) doen nauwelijks beter.

De prestatie van de verschillende zoekstrategieën BFS produceert de grootste individuen. De boomgrootte is steeds significant groter dan bij PRE, iBFS en POST (en uiteraard BSS). PRE produceert gelijkaardige resultaten als BSS qua boomgrootte alhoewel de fitheid gelijk is aan de maximum fitheid van SGP. iBFS en POST leunen zeer nauw aan bij elkaar zowel qua fitheid alsook qua boomgrootte.

De compactheid van de oplossingen In deze paragraaf behandelen we enkel BSS omdat het de enige methode is waarvan 19 simulaties een correcte oplossing hebben genereerd. De gemiddelde boomgrootte van het beste individu op generatie 100 is gelijk aan 284 voor BSS. Wanneer we enkel de simulaties bekijken die een 100% correcte oplossing hebben voortgebracht dan zakt de boomgrootte van het beste individu tot 150 voor BSS. Met behulp van de vereenvoudigingsmodule uit paragraaf 2.4.3, kunnen we de boomstructuren ontdoen van alle niet-functionele code. Als gevolg hiervan bekomen we nog compactere individuen met een gemiddelde boomgrootte van 131 voor BSS. De kleinste correcte oplossing telt slechts 22 knopen (Koza 1992). Dus ondanks het gebruik van LOSE kunnen we bij het multiplexerprobleem niet meteen spreken van compacte oplossingen.

Samenvatting van de resultaten De multiplexer is ongetwijfeld de moeilijkste toepassing uit de reeks. De ontzettend grote zoekruimte en het bestaan van slechts één unieke oplossing zorgen ervoor dat genetisch programmeren zeer moeizaam en gestaag betere oplossingen vindt. Bovendien loopt de gemiddelde boomgrootte bijzonder hoog op. LOSE slaagt erin om de groei van het aantal knopen in te perken. Doch worden er niet dezelfde resultaten behaald als bij de kunstmatige mier of het regressieprobleem. Wanneer we de kwaliteit van de verschillende zoekmethoden tegen elkaar afwegen, slaagt enkel BSS erin om significant betere oplossingen te evolueren dan SGP. Ondanks de betere resultaten bij BSS is de methode er onvoldoende in geslaagd om compacte oplossingen te genereren.

4.6 De vergelijking met bestaande methoden

In de voorgaande paragraaf hebben we aangetoond dat LOSE in combinatie met de BSS zoekstrategie de beste resultaten produceert. De bekomen reductie in boomgrootte en -diepte was het grootst en ook maximum fitheid was voor alle toepassingen het hoogst bij gebruik van BSS (steeds significant hoger dan SGP). We zullen daarom in deze paragraaf enkel BSS vergelijken met andere codegroeibegrenzers. We geven eerst een overzicht van de gebruikte methoden en bespreken vervolgens de resultaten voor elke benchmarktoepassing.

4.6.1 Enkele belangrijke concurrenten voor LOSE

We gebruiken vijf verschillende alternatieve methoden om codegroei te bestrijden: de boomgrootte en dieptebegrenzers, de verandering van de fitheidsbepaling (lineaire *parsimony pressure* en Tarpeiaanse codebegrenzing) en tenslotte een vereenvoudigingsoperator. Bij de 11-bit multiplexer worden ook ADF's opgenomen in de test suite. Een gedetailleerde beschrijving van een ADF tezamen met een voorbeeld, wordt opgegeven in paragraaf A.4.3. Alle codegroeiremmende methoden vereisen, net als LOSE, één of meerdere parameters die door de gebruiker moeten opgegeven worden. Het is niet steeds eenvoudig om een goede waarde te kiezen voor deze parameter. Ook hier is er nood aan het uitproberen van een aantal waarden zodat het resultaat optimaal is. De bekomen instellingen zijn probleemafhankelijk en worden steeds duidelijk in de tekst vermeld.

Diepte- en boomgroottebegrenzers zijn wellicht de oudste en meest gekende methoden om codegroei te bestrijden (Koza 1992). De werking ervan is zeer eenvoudig: van elk individu dat wordt gecreëerd door bijvoorbeeld kruising of mutatie zal het aantal knopen en/of de diepte worden bepaald. Indien de bekomen waarde afwijkt (groter is dan) van de opgegeven limiet dan zal het individu niet worden toegevoegd aan de populatie. Bij de generatie van de startpopulatie zal dit individu dan worden vervangen door een ander willekeurig gegenereerd individu dat wel aan de opgegeven limieten voldoet. Wanneer de simulatie echter is gestart, dan rijst er een bijkomende vraag: wat doen we met een individu dat niet beantwoordt aan de vooropgestelde limieten? Heel vaak wordt dit individu dan vervangen door één van de ouders. Sommige implementaties laten toe dat de operator blijft proberen om een geschikt kind af te leveren. Bij bijvoorbeeld kruising betekent dit dat er steeds opnieuw een nieuw kruisingspunt zal worden gekozen totdat een maximum aantal herhaling is bereikt of totdat een kind is geproduceerd dat aan de vereisten voldoet. Diepte- en boomgroottebegrenzers worden zelden gelijktijdig gebruikt. We verwijzen naar de beperking van de boomgrootte als SL en naar de dieptebegrenzer als DL.

Bij de tweede categorie van methoden staat de verandering van de fitheidsbepaling centraal. We gebruiken een alternatieve fitheidsfunctie die grotere individuen zal bestraffen door het gebruik van een zogenaamde *parsimony* factor (Soule & Foster 1998*a*). De nieuwe fitheidsfunctie ziet er als volgt uit:

$$f_{\text{nieuw}} = f_{\text{oud}} - \alpha \times \text{boomgrootte}$$

waarbij α het aandeel van de boomgrootte weergeeft bij de bepaling van de fitheid. Aangezien de fitheidswaarde genormaliseerd is tussen nul en één wordt ook de boomgrootte herschaald naar hetzelfde interval (op basis van de boomgrootte van het grootste individu). Een belangrijk nadeel bij het gebruik van deze fitheidsfunctie is dat bij hoge waarden van α de gemiddelde boomgrootte snel afneemt en de structurele diversiteit zeer klein wordt. Een doordachte keuze van α is dan ook onontbeerlijk. We noemen deze methode voor de eenvoud PP.

Een tweede manier om de codegroei af te remmen werd recent gesuggereerd door Poli (2003). De fitheidswaarde van individuen met een boomgrootte groter dan de gemiddelde boomgrootte van alle individuen uit de populatie, wordt gelijk gesteld aan nul. Uiteraard gebeurt dit niet met elk individu aangezien dit een negatieve invloed zou uitoefenen op de evolutie van de boomgrootte en maximum fitheid maar enkel met één op n individuen waarbij n een parameter is die door de gebruiker wordt ingesteld. Hoe groter n, hoe

minder individuen hun fitheid wordt aangetast. We noemen deze methode TP.

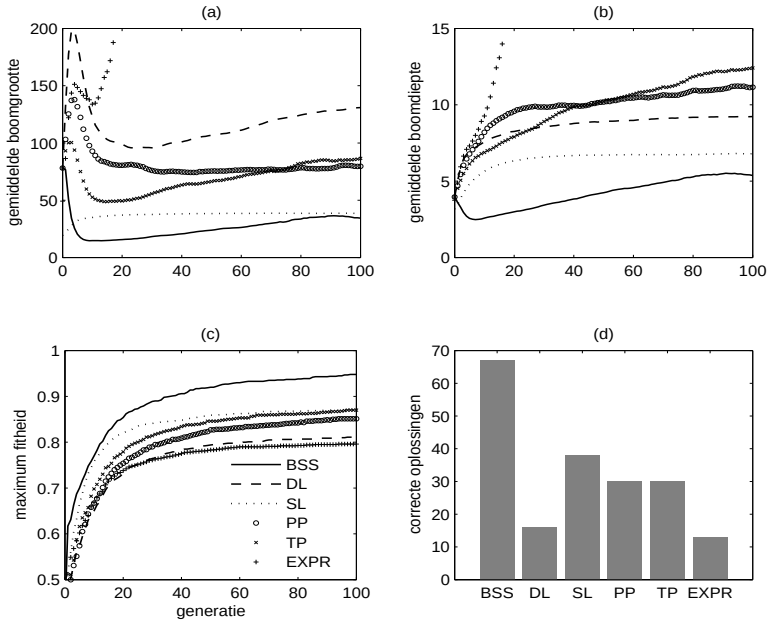
Een andere manier om codegroei te bestrijden is het gebruik van speciale operatoren. In het kader van dit proefschrift zullen we gebruik maken van een vereenvoudigingsoperator die de boomstructuur van individuen met behulp van een aantal eenvoudige regels tracht in te korten (Ekart 1999). Volgens Hooper & Flann (1996) geeft deze methode goede resultaten bij het regressieprobleem. Bij deze methode wordt aan de gebruiker de frequentie van toepassen van vereenvoudiging gevraagd. Dit wordt uitgedrukt in één om de s generaties. We noemen deze methode EXPR.

De reden waarom ADF's enkel bij het multiplexerprobleem worden gebruikt is de volgende. In hoofdstuk 2 werd de boomstructuur van de kleinste oplossing voor deze Booleaanse schakeling getoond. Zoals toen werd opgemerkt, vertoont deze boom heel wat gelijkenissen en bestaan er deelbomen die regelmatig terugkeren. Precies om deze gelijkvormigheden samen te vatten in één afzonderlijke functie werd ADF in het leven geroepen. Helaas valt het niet altijd uit de probleemstelling af te leiden of dergelijke gelijkvormigheid aanwezig zal binnen de boomstructuur van de geëvolueerde oplossing. Bij de overige twee benchmarktoepassingen is dit duidelijk niet het geval. Maar er zijn nog andere gevaren verbonden aan het gebruik van ADF's. Zo kan men zich de vraag stellen welke functies en eindknopen worden toegelaten in de resultaat-producerende boom en de ADF? Op welke bomen zal kruising een invloed uitoefenen? Het antwoord op al deze vragen zit vervat in bijkomende parameterinstellingen wat deze methode zeer kwetsbaar maakt. Bij de 11-bit multiplexer werden verschillende samenstellingen van de functie en terminalen verzameling uitgeprobeerd.

4.6.2 De kunstmatige mier

We gebruiken de volgende instellingen: de dieptelimiet bedraagt 10, de boomgroottelimiet bedraagt 50, $\alpha = 0.1$, elke vijf generaties wordt de vereenvoudigingsoperator toegepast en van één op twee individuen wordt de fitheid gelijk gesteld aan nul (Tarpeian methode). Figuur 4.12 toont de gemiddelde boomgrootte en -diepte en maximum fitheid voor alle methoden.

De Wilcoxon rang test wijst uit dat BSS voor beide opgemeten parameters betere resultaten produceert in vergelijking met alle andere methoden om codegroei te reduceren. BSS produceert dus kleinere individuen die bovendien een hogere fitheidswaarde behalen. Ook het aantal simulaties dat een correcte oplossing produceert ligt gevoelig hoger bij BSS dan bijvoorbeeld bij de



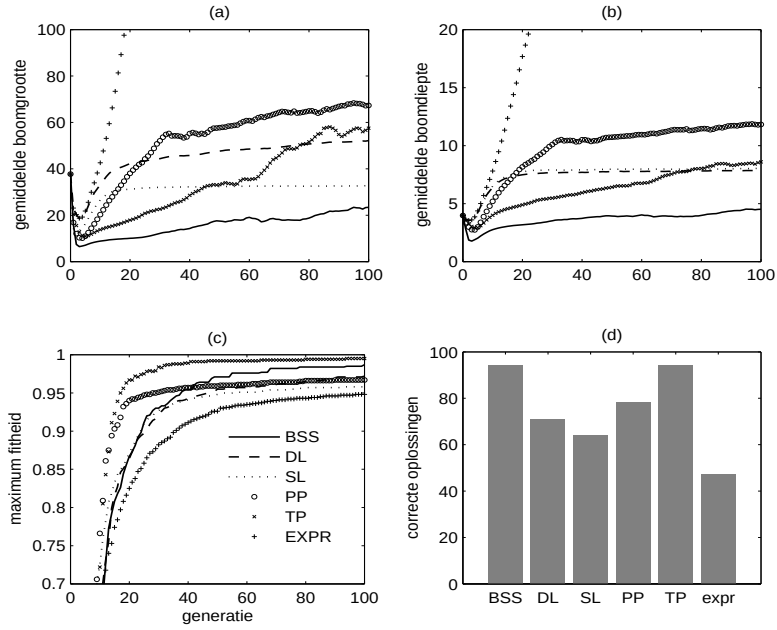
Figuur 4.12: *Gemiddelde boomgrootte en -diepte, en maximum fitheid van BSS en andere codegroeiremmende methoden bij de kunstmatige mier. Grafiek (d) duidt het aantal simulaties aan die een correcte oplossing hebben gegenereerd.*

methode PP. Enkel de EXPR slaagt er niet in om codegroei af te remmen. In tegendeel, de gemiddelde boomgrootte neemt zelfs sterker toe dan in vergelijking met SGP. Vele auteurs beweren dat niet-functionele code bescherming biedt tegen de mogelijks destructieve gevolgen van kruising of andere structuur veranderende genetische operatoren. In de generaties waar EXPR wordt toegepast, wordt alle niet-functionele code uit de boom verwijderd en dus ook alle bescherming tegen kruising. Wanneer we dan de hypothese 'bescherming tegen kruising' uit paragraaf 3.3.2 volgen, dan zou de code nog moeten toenemen, wat hier duidelijk gebeurt.

4.6.3 Vierde orde regressie

We gebruiken de volgende instellingen: de dieptelimiet bedraagt 10, de boomgroottelimiet bedraagt 50, $\alpha = 0.05$, elke vijf generaties wordt de vereenvoudigingsoperator toegepast en van één op drie individuen wordt de

fitheid gelijk gesteld aan nul (Tarpeian). Figuur 4.13 toont de gemiddelde boomgrootte en -diepte en maximum fitheid voor alle methoden.



Figuur 4.13: *Gemiddelde boomgrootte en -diepte, en maximum fitheid van BSS en andere codegroeiremmende methoden bij het regressieprobleem. Grafiek (d) duidt het aantal simulaties aan die een correcte oplossing hebben gegenereerd.*

Alle methoden behalve de EXPR produceren zeer goede resultaten. Ook bij deze toepassing stijgt de gemiddelde boomgrootte en -diepte wanneer men de boomstructuur elke vijf generaties vereenvoudigt. Statistische toetsen tonen significante verschillen aan tussen BSS en de overige codegroeiremmende methoden. We moeten wel oppassen met de interpretatie van deze p-waarden aangezien naast BSS ook de TP methode te lijden heeft onder het grote aantal simulaties die een correcte oplossing produceren waardoor de overblijvende simulaties een groter belang verwerven bij de bepaling van het gemiddelde. Wanneer we echter vergelijken op basis van het gemiddelde berekend zonder sterk afwijkende waarden, is er geen statistisch verschil.

Wanneer we kijken naar maximum fitheid is er een significant verschil tussen BSS en de vereenvoudigingsmodule, de boomgroottelimiet, de dieptelimiet en het gebruik van een aangepaste fitheidsfunctie. Enkel met de Tarpeian

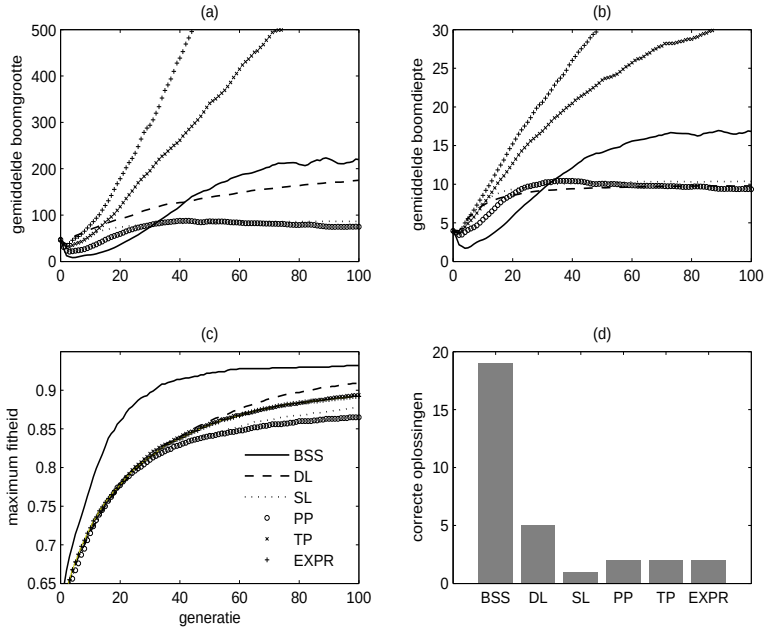
methode is er geen significant verschil. Wanneer we enkel deze methode bekijken, stellen we vast dat BSS de kleinste boomstructuren genereert. Wanneer we enkel de simulaties die een correcte oplossing hebben gegenereerd in beschouwing nemen dan is de gemiddelde boomgrootte bij BSS gelijk aan 14 in vergelijking met 18 voor de Tarpeian codegroeibegrenzer.

4.6.4 De 11-bit multiplexer

We gebruiken de volgende instellingen: de dieptelimiet bedraagt 10, de boomgroottelimiet bedraagt 100, $\alpha = 0.05$, elke tien generaties wordt de vereenvoudigingsoperator toegepast en van één op vijf individuen wordt de fitheid gelijk gesteld aan nul (TP methode). Figuur 4.14 toont de gemiddelde boomgrootte en -diepte en maximum fitheid voor alle methoden. Uit deze resultaten met behulp van ADF's bleek al gauw dat codegroei zowel binnen de RPB alsook binnen de afzonderlijke ADF bomen optrad. De resulterende boomgrootte was groter dan in vergelijking met standaard GP waardoor deze resultaten niet verder in dit hoofdstuk werden opgenomen.

Figuur 4.5 (aandeel niet-functionele code) geeft aan dat de multiplexer slechts weinig niet-functionele knopen bevat. We vermoedden dan ook dat de vereenvoudigingsmodule weinig of geen verandering zal brengen in de toenemende gemiddelde boomgrootte. Dit wordt bevestigd in Figuur 4.14. Alle methoden (met uitzondering van de vereenvoudigingsmodule) slagen erin om de hoeveelheid code te doen dalen. Tussen de gemiddelde boomgrootte bij BSS en de dieptelimiet bestaat er geen significant verschil. De oplossing voor het multiplexerprobleem bevat 22 knopen en vertoont een symmetrische structuur. De boom zelf is volledig opgevuld en heeft een beperkte diepte. Men kan intuïtief inzien dat een dieptelimiet de beste resultaten zal geven aangezien deze methode het groeien van diepe (niet volledig opgevulde bomen) verhindert en het creëren van brede individuen bevordert. Het verschil in gemiddelde boomgrootte tussen BSS en de andere methoden is wel significant. Enkel de groottelimiet en de alternatieve fitheidsbepaling produceren kleinere individuen.

Wanneer we echter kijken naar maximum fitheid dan produceert BSS de beste resultaten (p-waarde is nul wanneer we vergelijken met de vijf andere methoden). Bovendien is de maximum fitheid van de andere methoden niet significant verschillend van SGP. Behoudens de reductie in gemiddelde boomgrootte is de kwaliteit van de oplossing bij die methoden er niet op vooruit gegaan. Van de 100 simulaties zijn er 19 die leiden tot een correcte oplossing. Bij de andere methoden zijn er dit slechts enkele.



Figuur 4.14: *Gemiddelde boomgrootte en -diepte, en maximum fitheid van BSS en andere codegroeiremmende methoden bij de 11-bit multiplexer. Grafiek (d) duidt het aantal simulaties aan die een correcte oplossing hebben gegenereerd.*

4.7 Rekentijd optimalisatie

Het is duidelijk dat alle zoekstrategieën een zekere hoeveelheid extra rekentijd vragen om de boomstructuur te doorlopen op zoek naar een geschikte deelboom. Zeker ingeval van BSS, waar alle interne knopen worden bezocht, is de overhead vrij groot. BSS is immers een gulzig algoritme. Wanneer het aantal fitness cases hoog oploopt, kan dit leiden tot onaanvaardbaar lange rekentijden (multiplexer). Ook, wanneer gedurende latere generaties de code opnieuw groeit en bijgevolg het aantal deelbomen en de boomgrootte van de deelbomen eveneens toeneemt, kan dit aanleiding geven tot hoog oplopende rekentijden.

4.7.1 De rekentijd

Om de lezer een idee te geven van hoeveel extra rekentijd vereist is, beschouwen we een eenvoudige boomstructuur die volgens het *full* principe werd gecreëerd. Veronderstel dat de diepte k van deze boom bedraagt en dat elke functieknoop gemiddeld a kinderen heeft. Er moeten dus $\frac{1-a^k}{1-a}$ deelbomen worden geëvalueerd startend bij de wortel. Uit deze berekening lijkt het alsof de GP-simulator zeer zwaar belast zal worden, of toch niet?

Gelukkig maar zijn er een aantal verzachtende omstandigheden waardoor de overhead relatief beperkt blijft. Zo zorgt LOSE ervoor dat de diepte k van de boom bij een correcte instelling van p_{lose} gedurende de hele evolutie laag blijft waardoor het aantal te evalueren deelbomen eveneens beperkt is. Om een idee te geven van de benodigde rekentijd werd een extra experiment uitgevoerd. In tegenstelling tot de bovenstaande resultaten gebruiken we hier slechts één stopcriterium: een maximum aantal generaties. Tabel 4.4 toont de rekentijd in seconden van één uitvoering van het GP programma bij alle benchmarktoepassingen (gemiddeld over 100 simulaties). De resultaten van het regressieprobleem worden sterk beïnvloed door de simulaties die de oplossing trachten te benaderen. De simulaties werden uitgevoerd op een P4 2.6GHz processor met 1.5 GB intern geheugen en geen andere werklast.

Zoals verwacht is er een groot verschil merkbaar in de tijd nodig voor het evalueren van het individu (toekennen van de fitheid en zoeken naar een geschikte deelboom). De totale rekentijd omvat niet alleen de evaluatiefase maar ook de tijd nodig om nieuwe individuen te creëren met behulp van kruising, het vrijgeven van het geheugen, etc. Hier zien we vooral bij de kunstmatige mier een groot verschil tussen standaard GP en LOSE. Het werken met grote en omvangrijke boomstructuren kost immers meer tijd dan wanneer men enkel met kleinere boomstructuren omgaat. De talloze geheugentoegangen (lees- en schrijfoperaties) vertragen het systeem aanzienlijk. GP in combinatie met LOSE haalt hier een voordeel uit en zorgt er zelfs voor dat, bij de kunstmatige mier, de benodigde rekentijd (in zijn totaliteit) kleiner is dan bij SGP!

Een andere merkwaardige vaststelling is dat BFS, iBFS, PRE en POST niet bijzonder veel sneller zijn dan BSS. We hadden verwacht dat deze methoden sneller een geschikte deelboom zouden vinden. Bij de bespreking van de resultaten was duidelijk dat BSS de kleinste deelbomen vond. Wellicht zal dit een deel van de overhead die BSS introduceert wegwerken waardoor de methode quasi even snel kan werken als de overige vier selectiestrategieën. Ook het opnieuw ordenen bij de andere selectiestrategieën nemen extra rekentijd in beslag.

toepassing	SGP	BSS	BFS	iBFS	PRE	POST
mier (fitheid)	5.5	43.9	43.2	40.2	38.8	42.9
mier (totaal)	154.5	47.3	47.2	43.7	43.0	46.6
multiplexer (fitheid)	335.0	848.2	1979.0	782.6	1503.8	866.3
multiplexer (totaal)	394.9	856.7	1988.0	789.5	1517.2	873.8
regressie (fitheid)	22.9	6.5	7.9	85.9	59.1	8.8
regressie (totaal)	79.9	7.0	8.4	88.0	61.0	9.4

Tabel 4.4: *De gemiddelde rekentijd bij alle benchmarktoepassingen voor standaard GP en GP in combinatie met LOSE (alle zoekmethoden). Er werd een onderscheid gemaakt tussen de rekentijd nodig om de individuen te evalueren (alsook de deelbomen, aangeduid door 'fitheid') en de totale rekentijd (aangeduid door 'totaal').*

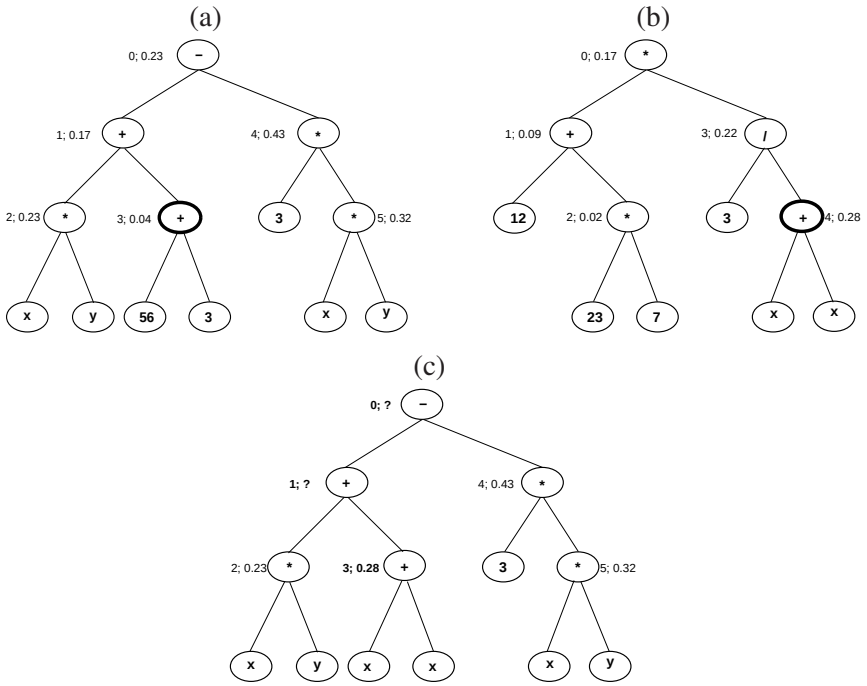
4.7.2 Een optimalisatie voor de rekentijd

In voorgaande paragraaf hebben we reeds aangetoond dat de overhead relatief beperkt blijft aangezien de diepte van de individuen progressief gereduceerd wordt door LOSE. Toch wensen we de benodigde rekentijd nog verder te reduceren.

In het bijzonder BSS, waarbij tijdens elke generatie alle deelbomen opnieuw worden geëvalueerd, leent zich uitstekend tot het incrementeel verifiëren of er nieuwe (betere) deelbomen werden aangemaakt. Om dit te realiseren zijn er uiteraard een heleboel aanpassingen nodig aan het bestaande programma. We zetten deze aanpassingen even op een rijtje en geven waar nodig extra toelichting.

De datastructuur van het individu bevatte reeds twee extra variabelen: de positie van de geschikte deelboom en de fitheid van deze deelboom. In plaats van enkel de fitheid van de geselecteerde deelboom houden we nu een rij van vlottende kommagetallen bij. Elk element uit deze rij bevat de fitheid van de deelboom met als wortel de index van het element uit deze rij. We noemen deze rij voor de eenvoud F_{subs}. Vroeger werden alle knopen geëvalueerd (BSS) maar werd enkel de fitheid en de positie van de beste deelboom bijgehouden in het geheugen. Nu is dit nog steeds het geval maar worden ook alle andere berekende fitheidswaarden opgeslagen in $F_{\text{subs}}[]$. Tijdens de initiële generatie zal geen enkele positie binnen $F_{\text{subs}}[]$ een geldige waarde bevatten en worden dus alle deelbomen geëvalueerd. Vanaf dan worden enkel die deelbomen opnieuw geëvalueerd die door structuur veranderende operatoren zoals kruising worden gewijzigd. Beschouwen we twee ouders en de nieuwe

boom na kruising zoals weergegeven in Figuur 4.15. $F_{\text{subs}}^{\text{ouder 1}}[3] = 0.04$ (ouder 1) betekent dat de fitheid van de deelboom met als wortel interne knoop nummer 3 gelijk is aan 0.04. Dit is de deelboom $(56 + 3)$.



Figuur 4.15: *Een voorbeeld van kruising waarbij ook de rij met fitheidswaarden ($F_{subs}[]$) wordt gekopieerd. De gehele getallen geven de index van de deelboom weer, de vlottende komma getallen duiden op de fitheid van de deelboom. (a) ouder 1, met als kruisingspunt knoop nummer 3 (b) ouder 2 met als kruisingspunt knoop nummer 4 en (c) het resultaat na kruising (slechts één van de kinderen wordt getoond).*

Alle genetische operatoren (reproductie, LOSE, mutatie en kruising) worden aangepast zodat de correcte waarden van $F_{\text{subs}}[]$ worden gekopieerd naar het nieuwe individu. Voor reproductie en LOSE is deze aanpassing zeer eenvoudig. Bij reproductie volstaat het om een exacte kopie van $F_{\text{subs}}[]$ te nemen. Bij LOSE wordt er slechts een gedeelte mee gekopieerd. De knopen boven de wortel van de geschikte deelboom worden immers door LOSE verwijderd. De andere knopen blijven ongewijzigd. Aangezien de positie van de deelboom wordt opgeslagen en dit precies de index is in $F_{\text{subs}}[]$, kan deze kopieeroperatie zeer snel gebeuren.

Bij de overige operatoren (kruising en mutatie) is het kopiëren minder eenvoudig. We bekijken eerst de mutatieoperator. Wanneer we een individu muteren dan kennen we de index k van de gekozen knoop, het aantal interne knopen (l_2, l_1) van de nieuwe vervangende deelboom en de deelboom die wordt verwijderd, alsook het aantal interne knopen in de nieuwe en oude boom (t_{old} en t_{new}). De rij $F_{\text{subs}}[]$ wordt dan als volgt opgevuld:

$$F^{\text{new}}_{\text{subs}}[0 \ldots k-1] = F^{\text{old}}_{\text{subs}}[0 \ldots k-1]$$

$$F^{\text{new}}_{\text{subs}}[k \ldots k+l_2-1] = -1.0$$

$$F^{\text{new}}_{\text{subs}}[k+l_2 \ldots t_{\text{new}}-1] = F^{\text{old}}_{\text{subs}}[k+l_1 \ldots t_{\text{old}}-1]$$

Het eerste deel is uiteraard gelijk en wordt ongewijzigd gekopieerd in de datastructuur van het nieuwe individu. De plaats in de rij voor nieuwe deelboom met l_2 interne knopen die door mutatie willekeurig wordt samengesteld, wordt opgevuld met ongeldige fitheidswaarden (-1.0). Voor alle knopen uit deze zopas gegenereerde deelboom werd er nog nooit een exacte fitheid berekend vandaar dat de waarden in $F_{\text{subs}}[]$ ongeldig worden gemaakt. Het laatste gedeelte (na l_1 interne knopen in de oude boomstructuur) wordt opnieuw ongewijzigd gekopieerd naar de nieuwe structuur.

Nu moeten we ervoor zorgen dat de fitheidswaarden die tijdens de volgende evaluatieronde opnieuw berekend moeten worden, ongeldig worden gemaakt. Met andere woorden, we zoeken de indices van de interne knopen waarvan de fitheid ongeldig wordt na toevoeging van de nieuwe deelboom. Uiteraard zijn dit de interne knopen van de nieuwe willekeurige gegenereerde deelboom (reeds gebeurd bij kopiëren) maar ook alle interne knopen waarvan de diepte kleiner of gelijk is aan de diepte van het mutatiepunt. De boomstructuur van elk van deze knopen werden immers gewijzigd door de nieuw toegevoegde deelboom. In de praktijk zijn dit enkel de knopen met een index kleiner dan de index van de wortel van de vervangen deelboom, dankzij het gebruik van prefix ordening. We kunnen zelfs een stapje verder gaan door de diepte adaptief te veranderen (te verlagen). Enkel de fitheid van de ouderknoop wordt aangepast, de fitheidswaarden van de 'broers' van deze knoop blijven ongewijzigd. Op deze manier worden de fitheidswaarden van enkel de ouder, grootouder, overgrootouder, etc. tot aan de wortel van het originele individu gewijzigd. Het algoritme ziet er als volgt uit:

```
deelboom_diepte=diepte_gekozen_mutatie_punt;
for(idx=(k-1);idx>=0;idx--){
 temp_diepte=diepte_deelboom_met_wortel_idx;
```

```
 if(temp_diepte<deelboom_diepte)
 {
  F_subs[idx]=-1.0;//ongeldig
  deelboom_diepte=temp_diepte; //adaptief aanpassen
                               //diepte
 }
}
```

Bij kruising zijn de uit te voeren operaties analoog. Voor het eerste kind (met als ouder het eerst geselecteerde individu) ziet $F_{\text{subs}}[]$ eruit als volgt:

$$F_{\text{subs}}^{\text{new1}}[0 \ldots k_1 - 1] = F_{\text{subs}}^{\text{old1}}[0 \ldots k_1 - 1]$$

$$F_{\text{subs}}^{\text{new1}}[k_1 \ldots k_1 + l_2 - 1] = F_{\text{subs}}^{\text{old2}}[k_2 \ldots k_2 + l_2 - 1]$$

$$F_{\text{subs}}^{\text{new1}}[k_1 + l_2 \ldots t_{\text{new1}} - 1] = F_{\text{subs}}^{\text{old1}}[k_1 + l_1 \ldots t_{\text{old1}} - 1]$$

waarbij k_1 gelijk is aan de index van de interne knoop die als eerste kruisingspunt werd gekozen (analoog voor k_2 maar dan de index van de interne knoop die als tweede kruisingspunt werd gekozen), l_1 en l_2 het aantal interne knopen van de te wisselen deelbomen, t_{new1} het aantal interne knopen in de nieuwe boomstructuur en t_{old1} het aantal interne knopen in de oude boomstructuur (ouder eerst geselecteerde individu).

We geven een voorbeeld aan de hand van de boomstructuren uit Figuur 4.15. Veronderstel dat in de eerste ouder knoop nummer 3 als kruisingspunt wordt gekozen en in de tweede ouder knoop nummer 4. De boomstructuur van het eerste kind na kruising is te zien in Figuur 4.15c. De fitheidswaarden van deelbomen met knoopnummers 4 en 5 veranderen niet en worden overgenomen van ouder 1 ($F_{\text{subs}}^{\text{new1}}[4 \ldots 5] = F_{\text{subs}}^{\text{old1}}[4 \ldots 5]$). De fitheid van de nieuw ingevoegde deelboom $(x + x)$ wordt overgenomen van ouder 2 en verandert uiteraard ook niet ($F_{\text{subs}}^{\text{new1}}[3] = F_{\text{subs}}^{\text{old2}}[4]$). De fitheidswaarden van de deelbomen met knoopnummers 0 (volledige boom) en 1 worden in eerste instantie overgenomen van ouder 1. Pas daarna, met behulp van bovenstaand algoritme, wordt de fitheid in $F_{\text{subs}}[]$ iteratief aangepast (ongeldig gemaakt). De fitheidswaarden van zowel de volledige boom (knoop 0) als deelboom 1 moeten opnieuw berekend worden aangezien deze zullen wijzigen door invoeging van een nieuwe deelboom afkomstig van ouder 2.

Voor het tweede kind (met als ouder het tweede geselecteerde individu) ziet $F_{\text{subs}}[]$ eruit als volgt:

$$F_{\text{subs}}^{\text{new2}}[0 \ldots k_2 - 1] = F_{\text{subs}}^{\text{old2}}[0 \ldots k_2 - 1]$$

$$F_{\text{subs}}^{\text{new2}}[k_2 \dots k_2 + l_1 - 1] = F_{\text{subs}}^{\text{old2}}[k_1 \dots k_1 + l_1 - 1]$$

$$F_{\text{subs}}^{\text{new2}}[k_2 + l_1 \dots t_{new2} - 1] = F_{\text{subs}}^{\text{old2}}[k_2 + l_2 \dots t_{\text{old2}} - 1]$$

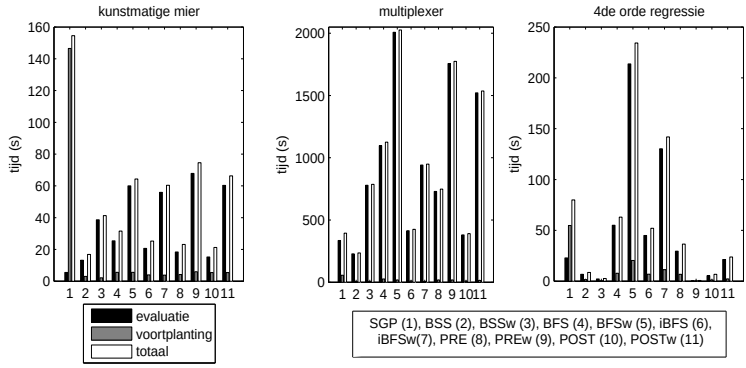
waarbij t_{new2} het aantal interne knopen in de nieuwe boomstructuur en t_{old2} het aantal interne knopen in de oude boomstructuur (ouder eerst geselecteerde individu). De bepaling van welke interne knopen hun fitheid ongeldig moet worden gemaakt, geschiedt zoals hierboven beschreven.

Veronderstel dat we een GP systeem gebruiken met enkel kruising en dat alle individuen volgens de *full* methode worden gegenereerd. Dan zijn er voor een gegeven diepte k, $\frac{1-a^k}{1-a}$ deelbomen waarvan de fitheid bepaald moet worden. a duidt het gemiddeld aantal kinderen per functieknoop aan. Bij een gegeven diepte l van het kruisingspunt moeten er steeds l knopen opnieuw geëvalueerd worden. Bijvoorbeeld wanneer het kruisingspunt ligt op diepte 3 dan zullen er drie knopen een nieuwe fitheidswaarde krijgen, namelijk de ouder van het kruisingspunt (diepte 2), de grootouder van het kruisingspunt (diepte 1) en tenslotte de wortel van de volledige boom (diepte 0). Onder de voorwaarde dat de verwijderde deelboom en de vervangende deelboom dezelfde afmetingen hebben (kruising maakt de volledige boom niet dieper), is de bekomen reductie dan gelijk aan:

$$\text{reductie factor} = \frac{\text{geëvalueerde knopen oud}}{\text{geëvalueerde knopen nieuw}} = \frac{\frac{1-a^k}{1-a}}{l}$$

Voor $k = 4$, $a = 2$ en $l = 2$ werden vroeger $\frac{1-2^4}{1-2} = \frac{-15}{-1} = 15$ bomen geëvalueerd, terwijl we nu slechts 2 deelbomen opnieuw evalueren. Uiteraard geeft dit een te optimistisch beeld van de te verwachten reductiefactor. Niet alle bomen zijn immers volledig opgevuld en vaak wordt ook mutatie gebruikt die ervoor zorgt dat tevens alle interne knopen uit de nieuwe deelboom geëvalueerd moeten worden. Bovendien is het zo dat kruising of mutatie met een vooraf instelbare kans ook eindknopen selecteren. Indien dit gebeurt dan moeten alle deelbomen die deze eindknoop gebruiken ook opnieuw geëvalueerd worden. Gelukkig is deze kans meestal relatief klein (meestal $\leq 10\%$).

Zoals men kan merken in Figuur 4.16 leidt deze algoritmische optimalisatie tot een aanzienlijke versnelling van de evaluatiefase. Wanneer we kijken naar de resultaten voor BSS dan werkte de niet geoptimaliseerde versie van het GP programma reeds sneller in vergelijking met standaard GP. Bij het vierde orde regressieprobleem vonden twee simulaties méér een 100% correcte oplossing dan bij LOSE met het incrementeel verifiëren van nieuwe

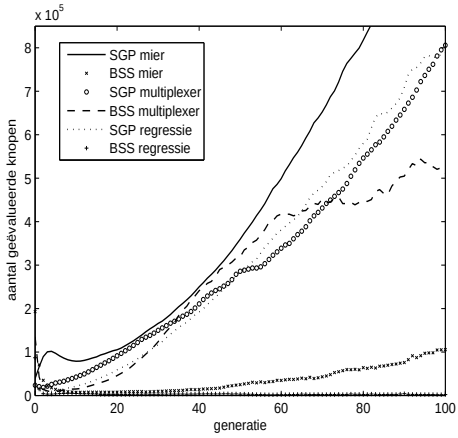


Figuur 4.16: *De rekentijd uitgedrukt in seconden voor alle deelboomselectiestrategieën voor alle benchmarktoepassingen opgesplitst per evaluatiefase, voortplantingsfase en de totale rekentijd. De deelboomselectiestrategieën met de vermelding 'w' maken geen gebruik van de beschreven algoritmische optimalisatie. We gebruiken dezelfde experimentele opstelling zoals beschreven in hoofdstuk 4.*

deelbomen. Tijdens deze laatste uitvoering trachtte het GP programma de functie te benaderen waardoor de benodigde rekentijd aanzienlijk toenam.

Voor het eerst werkt GP in combinatie met LOSE ook sneller bij het multiplexerprobleem wanneer BSS wordt toegepast. Wanneer we de niet-geoptimaliseerde en geoptimaliseerde versie van LOSE voor elk van de selectiemechanismen onderling vergelijken, dan is het zo dat het verbeterde zoekalgoritme steeds de rekentijd aanzienlijk reduceert. Een alternatieve voorstelling wordt gegeven in Figuur 4.17. Voor alle benchmarktoepassingen is het aantal geëvalueerde knopen (functies + eindknopen) steeds kleiner bij BSS dan bij standaard GP. Bij de kunstmatige mier en het multiplexerprobleem is het zo dat het aantal geëvalueerde knopen toeneemt naar het einde toe. Dit is te wijten aan het (gemiddeld) dalen van het kruisingspunt (dieper in de boom) waardoor het aantal deelbomen dat een nieuwe fitheid moet krijgen, zal stijgen. Enkel tijdens de initiële generatie is het aantal geëvalueerde knopen beduidend hoger in vergelijking met standaard GP. Dit is precies omdat dan voor de eerste maal alle deelbomen een fitheid toegewezen krijgen. De gemiddelde reductie van het aantal geëvalueerde knopen ten opzichte van standaard GP bedraagt 92% voor de kunstmatige mier, 15% voor de multiplexer en 95% voor het regressieprobleem (de initiële generatie werd niet in beschouwing genomen). Bij de multiplexer is dit gemiddelde aan

de lage kant aangezien gedurende generatie 30 tot 70 meer knopen worden geëvalueerd.



Figuur 4.17: *Het aantal geëvalueerde knopen wanneer LOSE in combinatie met BSS wordt gebruikt voor alle benchmarktoepassingen. Bij het regressieprobleem is dit aantal zeer klein aangezien 94 simulaties erin slagen een compacte oplossing te vinden (slechts 14 knopen, zie Tabel 4.3).*

Tenslotte merken we op dat ook voor de andere deelboomselectiemethoden (BFS, iBFS, PRE en POST) deze algoritmische optimalisatie zijn vruchten afwerpt. Het is evenwel zo dat de snelheidswinst uitgesmeerd wordt over meer generaties. Deze methoden stoppen immers met het evalueren van knopen wanneer een geschikte deelboom (fitheid groter dan fitheid volledige genoom) wordt gevonden. Hierdoor blijven er steeds een aantal interne knopen zonder fitheid achter.

4.7.3 LOSE en toenemende probleemgrootte

In dit proefschrift werd tevens onderzocht wat de invloed is van de moeilijkheidsgraad van de toepassing (analoog aan de studie van Daida *et al.* (2001)). Men zou immers verwachten dat bij een stijgende moeilijkheidsgraad ook de benodigde rekentijd zal toenemen aangezien de gezochte oplossingen complexer (omvangrijker) worden. De resultaten van deze experimenten tonen echter aan dat de moeilijkheidsgraad van de toepassing een minder grote invloed uitoefent op de rekentijd van GP in combinatie met LOSE (grafieken niet weergegeven). Ook het gebruikte aantal fitness cases speelt een ondergeschikte rol.

In figuur 4.3 merken we wel op dat bij een lage instelwaarde voor p_{lose} de gemiddelde boomgrootte (alsook de boomgrootte van het beste individu) relatief snel zal toenemen na een aantal generaties. Deze groei is enigszins vergelijkbaar met de situatie wanneer de gezochte oplossing voor de onderliggende toepassing complexer worden[5]. De zoektocht naar grotere, meer complexe structuren zal logischerwijs ook de rekentijd nodig om elke deelboom te evalueren, sterk doen toenemen. Dit lijkt de bruikbaarheid van LOSE enigszins te beperken. Toch zijn er een aantal oplossingen voorhanden om de rekentijd te beperken. We geven een kort overzicht:

- De keuze van een goede instelwaarde voor p_{lose} is zeer belangrijk (hetzij via parameter tuning, hetzij via een adaptieve methode uit hoofdstuk 6). Een goede instelwaarde laat de evolutie van logge individuen niet toe en zal steeds het vinden van compacte oplossingen promoten (zeker in vergelijking met standaard GP).

- In hoofdstuk 7 zullen we aantonen (bij de 11-bit multiplexer) dat GP in combinatie met LOSE niet steeds hoeft te beschikken over alle aanwezige fitness cases. Vaak zijn de prestaties van LOSE met minder fitness cases beter dan deze van standaard GP. Deze vaststelling is tevens geldig op basis van bovenstaand experiment.

- Er bestaan verschillende technieken om slechts een beperkt deel van de fitness cases te selecteren bij de bepaling van de fitheid van de verschillende deelbomen (Zhang & Cho 1999).

- LOSE kan zonder probleem gebruikt worden in combinatie met andere codegroeiremmende methoden

4.8 Verwante technieken en methoden

In de literatuur worden een aantal methoden beschreven nauw verwant met de lokale optimalisatieoperator. Deze paragraaf geeft een kort overzicht van deze technieken en vergelijkt deze methoden met LOSE.

[5]We bedoelen niet de moeilijkheidsgraad van de toepassing maar wel de complexiteit (boomstructuur, aantal knopen) van de correcte oplossing. Voorbeelden van dergelijke complexe oplossingen zijn onder meer te vinden in het domein van het ontwerpen en optimaliseren van digitale circuits. De gezochte oplossingen zijn vaak omvangrijk van structuur en bevatten een relatief groot aantal knopen.

Kinnear, Jr. (1994) definieerde een speciale mutatieoperator (*hoist mutation*) waarbij een 'willekeurig' geselecteerde deelboom als het ware wordt opgetakeld en omgevormd tot een nieuw individu. Deze operator diende om het aantal kleine individuen in de populatie te verhogen om codegroei van boomstructuren tegen te gaan[6].

Angeline & Pollack (1993*b*) gebruiken een compressieoperator die eveneens gebaseerd is op de selectie van een willekeurige knoop binnen de boom. Deze knoop vormt de wortel van de nieuwe boom. Vervolgens worden alle knopen die lager liggen dan een vooraf ingestelde grens uit deze nieuwe boom verwijderd. De maximale diepte van de boom wordt ad-hoc gekozen. Het resultaat wordt opgeslagen in een afzonderlijke bibliotheek en kan in toekomstige generaties worden aangewend door de andere operatoren.

Zowel hoist als de compressieoperator gebruiken geen kennis van het probleemdomein (geen fitness cases) om de positie van de deelboom te bepalen. LOSE gebruikt echter steeds informatie van de probleemstelling om een geschikte deelboom te bepalen. Ook de eisen die bij lokale optimalisatie aan de deelboom worden opgelegd zijn strikter. Bij de methode van (Kinnear, Jr. 1994) selecteert men een interne knoop verschillende van de wortel van de volledige boom. Bij (Angeline & Pollack 1993*b*) wordt er tevens een maximum diepte opgelegd aan de deelboom. Deze laatste methode had echter enkel tot doel om de functieverzameling verder uit te breiden en een bibliotheek van routines aan te leggen.

Rosca & Ballard (1994) gebruiken een adaptieve methode om de functieverzameling gedurende de evolutie uit te breiden (*adaptive representation scheme*, AR GP). Fitte deelbomen die worden beperkt in diepte zijn kandidaten voor nieuwe functies. De fitheid van een deelboom wordt bepaald door een zogenaamde blok fitheidsfunctie. Deze fitheidsfunctie was identiek aan de manier waarop fitheid voor de volledige boomstructuur werd geëvalueerd maar gebruikte een kleinere verzameling fitness cases en additionele informatie uit het probleemdomein[7].

Ook Rosca en Ballard gebruiken, net zoals LOSE, informatie van de probleemstelling om een goede deelboom te vinden. Uit hun resultaten is het niet meteen duidelijk of alle deelbomen werden overlopen of slechts enkele waaruit dan de beste deelboom wordt geselecteerd. Beide auteurs gebruiken ook een dieptebeperking waaraan de deelboom moet voldoen. Het is echter niet duidelijk hoe deze dieptelimiet wordt ingesteld. Verder wordt ARP en-

[6] Deze operator is gelijkaardig aan de *collapse* operator van Janikow (1996).

[7] Wat deze additionele informatie precies was, werd echter niet vermeld.

kel gebruikt om, net zoals de methode van (Angeline & Pollack 1993*b*), de functieset uit te breiden en hadden geenszins als hoofddoel om codegroei te bestrijden.

4.9 Besluiten

Vaak zijn de methoden om codegroei te bestrijden rechtstreeks of onrechtstreeks gebaseerd op één of meerdere theorieën voor codegroei. Dit hoofdstuk beschrijft een alternatieve aanpak gebaseerd op lokale optimalisatie van de boomstructuur. Centraal staat het begrip 'geschikte deelboom'.

Een geschikte deelboom moet voldoen aan twee belangrijke voorwaarden. Enerzijds is een geldige deelboom een interne knoop verschillend van de wortel van de volledige boom waardoor de diepte steeds ten minste één minder bedraagt. Anderzijds is de kwaliteit van de deelboom steeds minstens gelijk aan de fitheid van de volledige boom. Deze kwaliteit wordt bepaald door evaluatie van de deelboom op alle fitness cases.

Er bestaan verschillende strategieën om een geschikte deelboom te vinden. In dit hoofdstuk werden vijf verschillende deelboomselectiestrategieën voorgesteld: BSS (alle deelbomen worden geëvalueerd), en de groep (PRE, POST, BFS, iBFS) waarbij men stopt zodra een geschikte deelboom wordt gevonden. De combinatie van de deelboomselectiestrategieën en een aangepast selectiemechanisme gebaseerd op de fitheid van de deelboom, leidde tot de lokale optimalisatieoperator, kortweg LOSE.

Dankzij de voorwaarden opgelegd aan een deelboom, zal de gemiddelde boomgrootte gereduceerd worden. Bovendien heeft het gebruik van probleemspecifieke informatie bij de keuze van een geschikte deelboom als doel een positieve evolutie van maximum fitheid te vrijwaren. Ook de vereiste geheugenruimte om de individuen op te slaan wordt drastisch gereduceerd. Minder benodigde geheugenruimte betekent ook minder geheugentoegangen die nodig zijn om en deel van de boomstructuur in te lezen. Tot op heden bestaat er geen enkele methode die codegroei bestrijdt door gebruik te maken van de beschikbaar probleemkennis.

Uit de resultaten op drie verschillende benchmarktoepassingen blijkt dat de lokale optimalisatieoperator erin slaagt om codegroei tegen te gaan en de gemiddelde boomgrootte te beperken. Omvangrijke logge individuen worden omgevormd tot compacte en fittere exemplaren. Maar belangrijker: een verdere positieve evolutie van maximum fitheid wordt gerealiseerd. We geven een kort overzicht per toepassing van de belangrijkste bevindingen.

Bij de kunstmatige mier slaagt LOSE met glans in het reduceren van de boomgrootte terwijl maximum fitheid gevoelig wordt verhoogd. Wanneer we enkel de simulaties die een correcte oplossing genereren in beschouwing nemen, geeft LOSE compacte oplossingen die opnieuw leesbaar en bruikbaar zijn. In vergelijking met andere codegroeibegrenzers resulteert LOSE + BSS in de grootste reductie en de hoogste maximum fitheid.

Het regressieprobleem is de meest verraderlijke toepassing uit de benchmarksuite, zoals standaard GP aantoont. Toch blijkt uit de experimenten dat alle zoekmethoden zeer snel en op efficiënte wijze de zoekruimte doorkruisen op zoek naar een minimale oplossing. GP in combinatie met LOSE heeft meestal weinig generaties nodig om een juiste en compacte oplossing te vinden. De gekozen zoekstrategie speelt bij beide benchmarktoepassingen een ondergeschikte rol. Alle zoekmethoden produceren immers aanvaardbare resultaten, zowel qua boomgrootte alsook qua fitheid. In het bijzonder BSS herbergt de grootste reductie en slaagt er toch in om de beste oplossingen te genereren. Bij deze toepassing is er geen statistisch verschil tussen andere codegroeibegrenzers LOSE + BSS. Wanneer we enkel de correcte oplossingen in beschouwing nemen, genereert deze laatste wel de kleinste individuen.

De multiplexer is ongetwijfeld de moeilijkste toepassing uit de reeks. LOSE slaagt erin om de groei van het aantal knopen in te perken. Doch worden er niet dezelfde resultaten behaald als bij de kunstmatige mier of het regressieprobleem. Wanneer we de kwaliteit van de verschillende zoekmethoden tegen elkaar afwegen, slaagt enkel BSS erin om significant betere oplossingen te evolueren dan SGP. In vergelijking met andere methoden om codegroei tegen te gaan, geeft enkel LOSE een significant betere maximum fitheid.

BSS voert een globale zoektocht uit en overloopt alle interne knopen van de boomstructuur. In eerste instantie lijkt dit een zeer rekenintensieve operatie te zijn. Maar zoals we reeds hebben gezegd, blijft de diepte beperkt gedurende de evolutie waardoor ook de boomgrootte klein is. Hierdoor is het aantal knopen dat in aanmerking komt als wortel van een deelboom eveneens beperkt en zal de benodigde rekentijd meevallen. Enkel bij de multiplexer wordt een aanzienlijke overhead geïntroduceerd.

Met behulp van een algoritmische optimalisatie voor het opzoeken van een geschikte deelboom, slagen we erin om de benodigde rekentijd nog verder te reduceren. Aan de hand van enkele extra datastructuren worden steeds alle fitheidswaarden van alle deelbomen bijgehouden in het dataobject. De verschillende genetische operatoren worden aangepast zodat, naast de structurele wijzigingen, ook de corresponderende stukken uit de rij van fitheidswaarden van deelbomen correct worden gekopieerd. Het resultaat van deze

ingrepen is dat de rekentijd aanzienlijk geslonken is voor alle benchmarktoepassingen en dat het aantal geëvalueerde knopen aanzienlijk is afgenomen.

Delen van dit hoofdstuk werden gepubliceerd op een Europese conferentie en in *Lecture Notes in Computer Science* (Wyns *et al.* 2004).

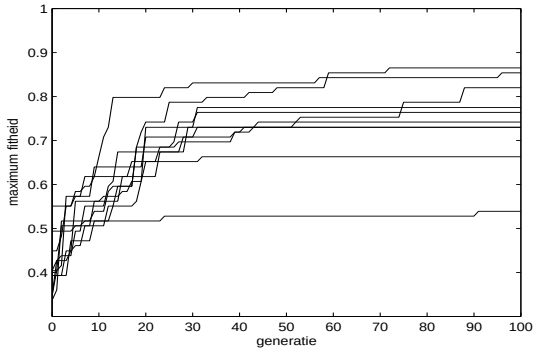
Hoofdstuk 5

Diversiteit

Een verlies aan diversiteit is een belangrijk en vaak voorkomend probleem bij genetisch programmeren wanneer codegroeibegrenzers worden gebruikt. Dit hoofdstuk start met een bespreking van wat diversiteit precies is, gevolgd door een literatuuroverzicht van de verschillende manieren om diversiteit op te meten. Codegroei beïnvloedt echter de resultaten van de klassieke structurele diversiteitsmaten. De aanwezigheid van grote hoeveelheden niet-functionele code vertekent immers het beeld van het aantal unieke boomstructuren aanwezig in de populatie. Daarom stellen we in dit hoofdstuk een verbeterde structurele diversiteitsmaat voor op basis van enkel de functionele knopen van een boom. Vervolgens bespreken we de resultaten van deze nieuwe aanpak bij twee benchmarktoepassingen. Tot slot, situeren we kort welke rol deze nieuwe maat zal spelen in de volgende hoofdstukken.

5.1 Vroegtijdige convergentie

De zoekstrategie van vele evolutionaire algoritmen bestaat uit twee grote fasen: exploratie en exploitatie. Enerzijds wensen we nieuwe gebieden in de zoekruimte te verkennen (exploratie) met behulp van mutatie en recombinatie. Daarnaast wensen we dat eigenschappen van goede kandidaat-oplossingen veelvuldig worden verspreid binnen de populatie. Het evenwicht tussen beide fasen is zeer gevoelig en de kleinste verstoring kan resulteren in een afremming van de zoekcapaciteiten van het algoritme. Wanneer steeds meer kandidaat-oplossingen eenzelfde vorm aannemen of eenzelfde gedrag vertonen, zullen hun kinderen (na kruising, mutatie, etc.) minder afwijkingen vertonen en zal het aantal 'unieke' individuen afnemen. Als dit verschijnsel zeer vroeg in de evolutionaire zoektocht optreedt, zal het algoritme eindigen in een suboptimale oplossing. Een voorbeeld van vroegtijdige convergentie is weergegeven in Figuur 5.1. In het begin volgen betere kandidaat-oplossingen elkaar in sneltempo op. Maar naarmate het aantal generaties verstrijkt, worden er minder snel nieuwe oplossingen gevonden.



Figuur 5.1: *Deze grafiek van de maximum fitheid per generatie bij 10 willekeurige GP-simulaties, toont hoe de fitheid binnen de populatie eerst snel stijgt, om vervolgens over te gaan op een veel vlakker verloop. De overgang tussen beide periodes duidt convergentie aan: er worden nauwelijks of geen betere kandidaat-oplossingen gevonden. De trapsgewijze stijging is te wijten aan een beperkt en discreet aantal mogelijke fitheidswaarden.*

Vroegtijdige convergentie is een belangrijk en vaak voorkomend probleem bij verschillende evolutionaire algoritmen en in het bijzonder bij verschillende codegroeibegrenzers (Ekárt & Németh 2001, Langdon & Nordin 2000). Om toch nog betere oplossingen te genereren probeert men vaak om de di-

versiteit kunstmatig te verhogen (Ryan 1994). Vooraleer men dit kan realiseren, moet men uiteraard beschikken over algoritmen die het begrip diversiteit accuraat kwantificeren. Ruwweg kan men een onderscheid maken tussen diversiteitsmaten die de boomstructuur van de kandidaat-oplossing analyseren en diversiteitsmaten die het gedrag van de individuen weergeven[1]. Met gedrag wordt bedoeld hoe een individu reageert in bepaalde vooraf gespecifieerde situaties. Bijvoorbeeld, bij de kunstmatige mier is dit het testspoor dat de mier moet bewandelen. Bij een regressieprobleem zijn dit een aantal (abscis, ordinaat)-waarden. De uitvoer bij elk van de invoerwaarden kenmerkt het gedrag van een individu.

Helaas vertroebelt codegroei het beeld op de aanwezige structurele diversiteit in de populatie. De toevloed aan niet-functionele code vergroot immers de kans dat twee boomstructuren van elkaar zullen verschillen. Wanneer men enkel de functionele delen van de boom overhoudt, komt het vaak voor dat beide bomen toch identiek zijn. Bijgevolg lokt codegroei de applicatieprogrammeur in de val via een verkeerde inschatting van de reële diversiteit (McPhee & Hopper 1999). In dit hoofdstuk introduceren we een nieuwe structurele diversiteitsmaat die rekening houdt met de bijwerkingen van codegroei.

5.2 Diversiteitsmaten

Wanneer men spreekt over diversiteit dan denken we vaak in de eerste plaats aan structurele verschillen. Diversiteit is echter een ruim begrip dat bij genetisch programmeren klassiek twee verschillende invalshoeken kent: diversiteit gebaseerd op de boomstructuur en diversiteit gebaseerd op het gedrag (fitheid). De volgende paragrafen geven een overzicht van de belangrijkste diversiteitsmaten.

5.2.1 Diversiteitsmaten gebaseerd op de boomstructuur

Vooral binnen deze categorie kan men een grote verscheidenheid aan technieken terugvinden. We kunnen deze technieken echter onderverdelen in drie grote groepen: de eigenschappen van een boomstructuur, transformatieafstanden en genealogische technieken.

[1] Vaak wordt er gesproken over genotypische diversiteit en fenotypische diversiteit, naar analogie met de biologie. Deze termen worden soms gebruikt om specifieke methoden aan te duiden, maar in dit proefschrift worden ze enkel gebruikt als namen van categorieën.

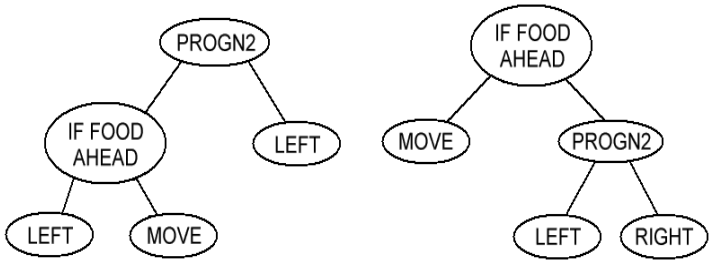
Diversiteit gebaseerd op de eigenschappen van de boomstructuur

De eerste voor de hand liggende diversiteitsmaat is de variëteit, die eenvoudigweg het aantal unieke programma's in de populatie telt (Koza 1992). Hiervoor worden alle individuen onderling structureel met elkaar vergeleken, twee programma's zijn verschillend als hun programmabomen niet exact gelijk zijn. Zowel de structuur (diepte, aantal knopen, positie van de knopen,...) alsook de inhoud van de knopen spelen hierbij een belangrijke rol. Deze diversiteitsmaat wordt vaak gebruikt omwille van zijn eenvoud (programmatorisch niet zo moeilijk te berekenen), en de intuïtieve interpretatie. In (Langdon 1998*a*) wordt aangetoond dat variëteit een bovengrens vormt voor de aanwezige fenotypische diversiteit binnen een populatie. In genetisch programmeren zullen twee bomen met exact dezelfde structuur, ook hetzelfde gedrag vertonen. Wanneer het aantal unieke boomstructuren vermindert, zal dus ook het aantal verschillende fitheidswaarden verminderen.

Er bestaan een aantal meer complexe afgeleide maten van deze variëteit, gebaseerd op het tellen van meer verfijnde kenmerken dan unieke bomen, zoals unieke deelbomen, unieke groottes, en aantallen knopen van een bepaald type in de populatie. Zo noemt Keijzer (1996) de hierboven vermelde variëteit de programma-variëteit, en wordt de deelboom-variëteit vastgelegd als de verhouding van het aantal unieke deelbomen tot het totaal aantal deelbomen.

Een andere aanpak om diversiteit te meten, is isomorfisme. Deze term is afkomstig vanuit de grafentheorie waar isomorfisme gedefinieerd is als een bijectie tussen de knopen die de structuur bewaart. Bij GP bedoelt men met isomorfisme een functie die één boomstructuur kan afbeelden op een andere boomstructuur zonder uiteraard de functionaliteit van beide boomstructuren te veranderen en zonder het aantal knopen te reduceren (Rosca 1995). De diversiteit is dan het aantal isomorfe klassen binnen de populatie.

In plaats van te bepalen of individuen isomorf zijn, kan men ook bepalen of individuen isomorf met elkaar 'kunnen' zijn. Dit doet men door een aantal eenvoudig te meten eigenschappen te kiezen, zoals bijvoorbeeld het aantal functieknopen, het aantal terminalen en de diepte. Als al deze eigenschappen voor twee individuen gelijk zijn, dan zijn deze twee 'mogelijk' isomorf, of pseudo-isomorf. Het volstaat dan het aantal unieke combinaties van de gekozen eigenschappen te tellen, en dit geeft ons het aantal pseudo-isomorfen in de populatie als diversiteitsmaat. Een voorbeeld is te zien in Figuur 5.2. Let wel: pseudo-isomorfisme houdt geen rekening met de inhoud van de knopen.



Figuur 5.2: *Pseudo-isomorfe bomen; deze beide individuen hebben elk 2 interne knopen, 3 eindknopen en een diepte 2, en zijn dus pseudo-isomorf. Beide bomen zijn echter in geen geval isomorf!*

De transformatieafstanden

Bij genetische algoritmen ligt het gebruik van de Hamming-afstand tussen individuen voor de hand om te weten te komen hoe verschillend twee individuen zijn. Bij genetisch programmeren zijn zulke afstandsmaten wat ingewikkelder. Er zijn in de literatuur een paar licht variante afstandsmaten, die wij verder ook transformatieafstanden noemen, terug te vinden. Zo is er de gemiddelde transformatieafstand van elke individu tot alle anderen (Burke *et al.* 2002*b*, *a*), een gewogen gemiddelde afstand waar op elkaar lijkende transformatieafstanden meer gewicht hebben dan geïsoleerde transformatieafstanden (Ekárt & Németh 2000), de gemiddelde kwadratische afstand (De Jong *et al.* 2001), of de gemiddelde afstand van het beste individu tot alle anderen (O'Reilly 1997). Toch hanteren al deze methoden dezelfde werkwijze.

De basisgedachte achter deze groep is het zoeken naar de minst kostende reeks transformaties van één boom naar een andere. Transformaties zijn daarbij het vervangen van een knoop door een andere, het verwijderen van een knoop uit de boom of het toevoegen van een knoop aan de boom.

Genealogische technieken

De belangrijkste vertegenwoordiger van de laatste groep om genotypische diversiteit te meten, is de zogeheten ID-memID methode (McPhee & Hopper 1999). Elke knoop in de hele populatie wordt gemerkt met twee labels: 'ID' en 'memID'. Elke knoop die tijdens de initiële populatie gegenereerd wordt, krijgt een unieke waarde als ID en diezelfde unieke waarde voor memID. Wanneer er een kruising plaatsvindt, krijgt elke knoop in het nieuw gemaakte individu die op het pad ligt tussen de beginknoop en het kruisingspunt hetzelfde ID als de overeenkomstige knoop in de ouder. Op die manier hebben knopen met dezelfde vooroudergeschiedenis hetzelfde ID. Diezelfde knopen op het pad tussen de beginknoop en het kruisingspunt krijgen ook een nieuw, uniek memID. Zo krijgen alle knopen die andere kinderknopen hebben na de kruising dus een nieuw memID.

De aantallen unieke ID's en memID's leveren een indicatie voor de verscheidenheid aan genetisch materiaal aanwezig in de populatie, en kunnen dus als diversiteitsmaat aangewend worden. McPhee en Hopper ontdekten dat het aantal unieke ID's, en dus de diversiteit, zeer sterk daalt na een klein aantal generaties. Door de ouders te volgen ontdekten ze bovendien dat, eveneens na een klein aantal generaties, alle individuen in de hele populatie dezelfde voorouder delen. Ze concludeerden dat het vastleggen van de bovenste knopen van alle programmabomen in zulk een vroeg stadium van de evolutie een groot nadeel vormt voor het ontwijken van lokale optima in latere generaties.

5.2.2 Diversiteitsmaten gebaseerd op het gedrag van een individu

Net zoals de eenvoudigste genotypische diversiteitsmaat het aantal unieke bomen telt (de variëteit), telt de eenvoudigste fenotypische maat het aantal unieke fitheidswaarden. Als je de zoektocht van het genetisch programmeren vergelijkt met het doorkruisen van een fitheidslandschap, dan geeft het aantal unieke fitheidswaarden een idee over hoe de populatie verspreid is over dit landschap. Met andere woorden, een hoger aantal verschillende fitheidswaarden betekent dat het programma een groter deel van de zoekruimte exploreert.

Buiten deze eenvoudige telling, kunnen op basis van de fitheidswaarden nog andere diversiteitsmaten geconstrueerd worden. De distributie van de fitheidswaarden, gevisualiseerd in fitheidshistogrammen, levert al iets meer informatie over de spreiding van de individuen over de fitheidsruimte dan enkel het aantal fitheidswaarden (Rosca & Ballard 1993).

Rosca introduceerde de entropie[2] in de wereld van het genetisch programmeren (Rosca 1995). De entropie is een maat voor de wanorde binnen de populatie, een hogere entropie komt dus overeen met een hogere diversiteit. Dalingen in de entropie correleren dan met het stagneren van de fitheidscurven en vormen een indicatie voor het bereiken van een (lokaal) optimum. Voor de berekening van de entropie worden de individuen op basis van hun gedrag opgedeeld in een aantal partities. Een partitie wordt over het algemeen bepaald door één unieke fitheidswaarde, alle individuen met eenzelfde fitheid vormen dan één partitie, maar in theorie kan een partitie ook door een verzameling fitheidswaarden bepaald worden (bijvoorbeeld een interval). p_k is de fractie van de populatie die in partitie k van de populatie valt, en de entropie wordt dan als volgt berekend: $-\sum_k p_k \cdot \log p_k$.

5.2.3 Enkele algemene opmerkingen bij de diversiteitsmaten

Niet alle beschreven diversiteitsmaten zijn even gemakkelijk in gebruik. Aan een aantal technieken zijn er duidelijk nadelen verbonden. We geven een kort overzicht.

Vooral bij genotypische diversiteitsmaten wordt de bruikbaarheid van een techniek vaak afgemeten op de benodigde rekentijd. Deze rekentijd vormt een belangrijk nadeel van het bepalen van isomorfe klassen voor een hele populatie. Bij deze techniek kan de rekentijd immers zeer sterk oplopen. We geven een kort voorbeeld. Veronderstel een regressieprobleem waarbij de boomstructuur enkel de functie $+$ bevat en de bomen gegenereerd zijn met de *full* methode met een diepte van vier. Een boom met dergelijke specificaties heeft precies 16 eindknopen. Er bestaan exact $16!$ verschillende ordeningen van deze 16 eindknopen en dus ook $16! \approx 21 \times 10^{12}$ verschillende bomen die isomorf zijn. Gelukkig maar zal men in de praktijk niet telkens een dergelijk groot aantal mogelijke isomorfismen moeten vergelijken. De meeste toepassingen hebben immers meer dan één enkele functie en niet alle functies delen de associativiteit en commutativiteit van de optelling.

Ook de transformatieafstanden delen ditzelfde nadeel. Boomstructuren één voor één met elkaar vergelijken, vergt zeer veel rekentijd. Bijkomend is het de taak van de applicatieprogrammeur om alle voorkomende transformaties te implementeren afhankelijk van de samenstelling van de functie- en eindknopenverzameling. Het is bovendien niet steeds duidelijk wat nu precies

[2]De definitie van Rosca's entropie is equivalent met de algemeen gekende Shannon-entropie. In het vervolg van dit proefschrift verwijzen we enkel nog naar Rosca's entropie.

het 'kleinste' aantal transformaties is, nodig om de ene boomstructuur om te vormen tot de andere. Dit is een zoektocht op zich.

Net zoals er verschillende factoren zijn die aan de basis van codegroei liggen, geven verschillende diversiteitsmaten ook een accurater beeld dan slechts één enkele maat. Verschillende auteurs waaronder Burke *et al.* (2004) pleiten dan ook voor het hanteren van een aantal verschillende diversiteitsmaten (zowel fenotypische als genotypische maten).

5.3 Een verbeterde weergave van structurele diversiteit: functionele pseudo-isomorfen

In McPhee & Hopper (1999) wordt opgemerkt dat de aanwezigheid van intronen (door bijvoorbeeld codegroei) tot een verkeerde inschatting van de diversiteit kan leiden. Intuïtief kan men ook aanvoelen dat wanneer men rekening zou houden met de aanwezigheid van niet-functionele code, men misschien wel nauwkeuriger meetresultaten voor de genotypische diversiteit kan bekomen.

Het aantal pseudo-isomorfen is eenvoudig te berekenen aangezien deze techniek gebruik maakt van enkele eenvoudige eigenschappen van de boomstructuur. De benodigde rekentijd wordt dus tot een absoluut minimum beperkt. In Burke *et al.* (2002*b*) wordt bovendien gemeld dat deze pseudo-isomorfenmaat vergelijkbare resultaten levert als het aantal unieke bomen, en dus de voorkeur geniet aangezien ze minder rekentijd vereist.

Op basis van de aanwezigheid van codegroei en de vertekening die dit fenomeen veroorzaakt werd besloten om de diversiteitsmaat op basis van pseudo-isomorfen aan te passen, en vervolgens te testen in hoeverre dit inderdaad een verbetering betekent. Het algoritme bestaat uit twee verschillende stappen.

1. Met behulp van een vereenvoudigingsmodule wordt de boomstructuur ontdaan van alle niet-functionele code. De vereenvoudigingsregels zijn uiteraard probleemafhankelijk en kan men raadplegen in paragraaf 2.4.3.

2. In overeenstemming met de originele bepaling van het aantal pseudo-isomorfen, wordt een aantal eenvoudig te meten karakteristieken van de vereenvoudigde boomstructuur berekend zoals de functionele grootte (aantal functies en aantal terminalen) en de functionele diepte.

Omdat een vereenvoudiging van de boomstructuur een reductie van het aantal functies, terminalen en diepte betekent, verwachten we dat het aantal functionele pseudo-isomorfen kleiner zal zijn dan het aantal pseudo-isomorfen. Bovendien is deze diversiteitsmaat eenvoudig te berekenen op basis van de gereduceerde boomstructuur. Enkel de simplificatie routine neemt wat tijd in beslag. De gereduceerde boom moet echter maar éénmaal berekend worden bij het begin van elke generatie en wordt eveneens opgeslaan in het geheugen. Beide voordelen maken functionele pseudo-isomorfen expressiever (codegroei en de rol van niet-functionele code worden geminimaliseerd) dan pseudo-isomorfen en computationeel minder zwaar dan een transformatieafstand.

5.4 De gebruikte experimentele opstelling

In deze paragraaf wordt de experimentele proefopstelling nader toegelicht. We gebruiken twee standaardtoepassingen: de kunstmatige mier en het regressieprobleem. Voor een uitgebreide bespreking van beide benchmarks verwijzen we naar paragraaf 2.3. In de volgende paragraaf worden de instellingen van de GP-simulator besproken voor zover deze afwijken van de voorstelling in paragraaf 2.4.1. De meeste gewijzigde parameterinstellingen werd overgenomen uit een vergelijkende studie van klassieke manieren om diversiteit op te meten (Burke *et al.* 2004). In tegenstelling tot de studie van Burke worden echter geen dieptelimieten gebruikt.

5.4.1 De benchmarktoepassingen

Bij de kunstmatige mier worden alle parameterinstellingen gebruikt zoals gedefinieerd in 2.3.4.

Bij symbolische regressie kiezen we voor een vierde orde polynoom $x^4 + x^3 + x^2 + x$. De functie- en eindknopenverzameling is samengesteld zoals weergegeven in Tabel 2.1.

5.4.2 De GP-simulator

Het GP systeem evolueert gedurende 50 generaties (initiële generatie niet meegeteld) of tot wanneer een 100% correcte oplossing wordt gevonden. De initiële populatie bevat 500 individuen die volgens de methode uit paragraaf 2.3.4 met dieptes tussen twee en vier. Individuen worden geselecteerd door

middel van roulettewielselectie. Kruising wordt toegepast in 100% van de gevallen (analoog aan Burke *et al.* (2004)). Er wordt geen gebruik gemaakt van lokale optimalisatie, reproductie of mutatie.

De volgende diversiteitsmaten worden opgemeten (per generatie): pseudo-isomorfen, functionele pseudo-isomorfen, Rosca's entropie, en het aantal unieke fitheidswaarden.

5.5 Resultaten met standaard GP

In deze paragraaf wordt het verloop van de vernieuwde structurele diversiteitsmaat geanalyseerd. Maar vooraleer daaraan te beginnen, wordt het verloop van de fitheid en de boomgrootte en diepte voor beide toepassingen kort herhaald. Tot slot bespreken we ook kort de evolutie van fenotypische diversiteit aan de hand van Rosca's entropie.

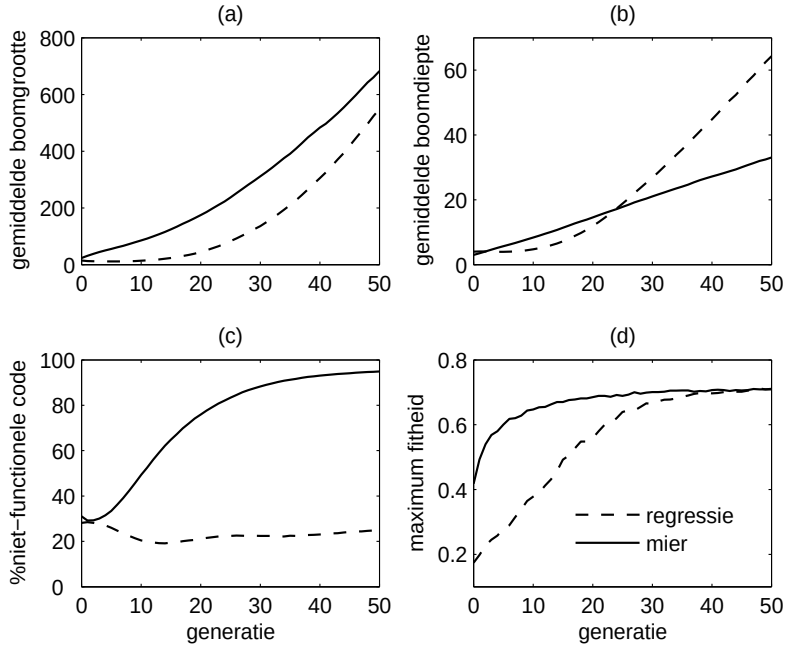
5.5.1 Boomgrootte en fitheid bij standaard genetisch programmeren

De drijfveer achter het ontwerp van de verbeterde structurele diversiteitsmaat uit dit hoofdstuk is gebaseerd op codegroei. Vandaar dat hier eerst wat aandacht besteed wordt aan de evolutie van boomgrootte en diepte bij beide toepassingen wanneer gebruik gemaakt wordt van standaard GP[3].

Figuur 5.3 toont het verloop van de verschillende grootte- en dieptematen voor beide toepassingen. We zien dat de hoeveelheid knopen duidelijk toeneemt in de loop van de evolutie. De kunstmatige mier genereert steeds meer knopen dan het regressieprobleem terwijl dit laatste probleem diepere bomen creëert. Dit is te wijten aan het (gemiddeld) kleinere aantal operandi per functie. Het regressieprobleem heeft gemiddeld: (2 $(+)$ + 2 $(-)$ + 2 $(\times)$ + 2 $(\div)$ + 1 $(\exp)$ + 1 $(\log)$ + 1 $(\cos)$ + 1 $(\sin)$) / 8 = 1.5 operandi per functie. De kunstmatige mier heeft er: (2 (IF-FOOD-AHEAD) + 2 (PROGN2) + 3 (PROGN3)) / 3 $\approx$ 2.3. Hierdoor zal de boomstructuur bij de mier eerder in de breedte groeien en eerder in de diepte bij regressie.

Figuur 5.3c toont het gemiddelde percentage niet-functionele code dat aanwezig is in de boomstructuren voor beide standaardtoepassingen. Het percentage niet-functionele code neemt duidelijk toe per generatie. Merk ook

[3]De gebruikte experimentele opstelling wijkt af van de opstelling die werd gebruikt in hoofdstuk 4.



Figuur 5.3: *Gemiddelde boomgrootte (a) en boomdiepte (b), het gemiddeld percentage niet-functionele code (c) en maximum fitheid (d) voor beide toepassingen uitgezet per generatie.*

op dat bij de kunstmatige mier de hoeveelheid functionele code blijft dalen zij het langzamer naar het einde toe (dit werd eveneens vastgesteld door Nordin & Banzhaf (1995)). Bij het regressie-experiment is dit minder uitgesproken dan bij de kunstmatige mier. De oorzaak daarvan is dat de kans dat er bij het regressieprobleem niet-functionele code ontstaat kleiner is dan de kans op het optreden van geneste IF-FOOD-AHEAD structuren bij de mier. Bij zulke constructies verdwijnen volledige deelbomen waardoor een grotere reductie mogelijk is. Bij het regressieprobleem zijn het voornamelijk deelbomen die bestaan uit constanten (ERC waarden), die gereduceerd kunnen worden. Voorbeeld: $\log(\exp(\cos(ERC_1 + ERC_2))) = ERC_3$. Bovendien zijn de reductieregels complexer dan bij de kunstmatige mier en worden niet alle redundante codefragmenten uit de boom verwijderd omwille van de beperkte rekentijd (bijvoorbeeld associativiteit en commutativiteit). Voorbeeld: $((ERC_1 + ERC_2) + (x + ERC_3)) = (ERC_4 + (x + ERC_3))$ (in plaats van $(x + ERC_4)$).

Maximum fitheid van elke toepassing is weergegeven op Figuur 5.3d. Beide fitheidscurven kennen een analoog verloop. Ze beginnen met een fase van snel stijgen, en gaan dan over naar een trage stijging (regressie) of naar een constante waarde (bij de kunstmatige mier). De populatie vertrekt vanuit een initieel willekeurige toestand en evolueert naar een geoptimaliseerde verzameling van kandidaat-oplossingen. De overgang van snel naar traag stijgen treedt over het algemeen reeds vroeg op (rond de tiende generatie bij de mier en rond de twintigste generatie bij het regressieprobleem). Wat opvalt bij het regressieprobleem, is dat maximum fitheid tot generatie 50 lichtjes blijft stijgen.

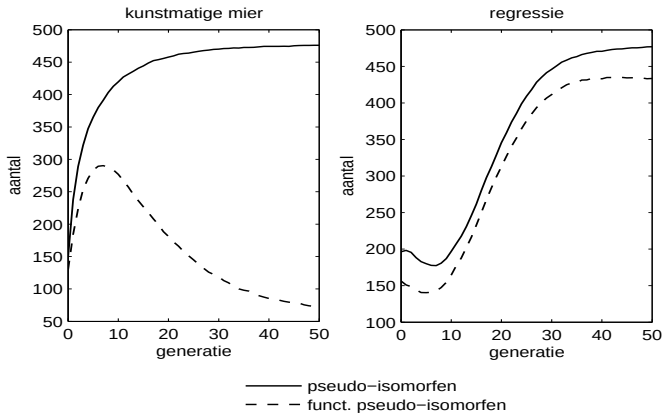
Op basis van dit eerste eenvoudige experiment merken we toch een verschil tussen beide toepassingen. Hoewel codegroei optreedt op bij beide toepassingen, ondervindt de kunstmatige mier toch meer hinder van de sterke toename in boomgrootte. Dit uit zich onder meer in de snelle uitvlakking van maximum fitheid. Door de aanwezigheid van grote hoeveelheden niet-functionele knopen (Fig. 5.3c, tot bijna 95%) houdt het GP algoritme zich in hoofdzaak bezig met het recombineren van waardeloos materiaal en worden er maar weinig nieuwe en meer fitte oplossingen gevonden.

Bij het regressieprobleem lijkt codegroei een minder negatieve invloed uit te oefenen op maximum fitheid. De geringe aanwezigheid van niet-functionele code ('slechts' 22% in tegenstelling tot bij de mier) en de stijging van de boomgrootte doen ons vermoeden dat GP de beoogde doelfunctie ($x^4 + x^3 + x^2 + x$) tracht te benaderen met behulp van een combinatie van de vele wiskundige operatoren en een constante. Inderdaad, wanneer we de boomgrootte van het best individu van elke simulatie bekijken, dan stellen we vast dat de geëvolueerde boomstructuur heel wat meer knopen telt dan de eenvoudigste oplossing met slechts 13 knopen. Dit verklaart ook waarom maximum fitheid lichtjes blijft stijgen op het einde van de simulatie.

5.5.2 Structurele diversiteit bij standaard GP

Figuur 5.4 toont het aantal (functionele) pseudo-isomorfen bij de kunstmatige mier en het regressieprobleem. Op beide grafieken kan men zien dat het aantal pseudo-isomorfen klein is bij de start van de simulatie. Omdat de initiële populatie bestaat uit kleine individuen (de dieptehelling ligt immers tussen twee en vier), zullen er maar weinig verschillende pseudo-isomorfe combinaties bestaan, en is het aantal pseudo-isomorfen dus beperkt. Na enkele generaties, gesteund door de onbeperkte diepte en boomgrootte, zal de boomstructuur van de meeste kandidaat-oplossingen aangroeien en uitdiepen

(zie Fig. 5.3). Dit geeft zeer duidelijk aanleiding tot een toename in het aantal pseudo-isomorfen. Dit is niet vanzelfsprekend aangezien de definitie gebaseerd is op afgeleide karakteristieken van de boomstructuur in plaats van exacte gelijkenis. In beide standaardproblemen neemt de klassieke definitie van de structurele diversiteitsmaat toe, bijna tot wanneer het maximum, de populatiegrootte, wordt bereikt.



Figuur 5.4: *Het aantal (functionele) pseudo-isomorfen bij de kunstmatige mier en het regressieprobleem uitgezet per generatie.*

Figuur 5.3c toont het gemiddeld percentage niet-functionele code binnen de boomstructuur van een kandidaat-oplossing. Zeker bij de kunstmatige mier bestaat er een enorm verschil tussen de functionele en niet-functionele delen van de boom. Dit geeft aan dat deze toepassing zeer gevoelig is voor de creatie van overtollige code. Wanneer we nu de boomstructuur vereenvoudigen aan de hand van de eerder vernoemde reductieregels, verwachten we dat veel individuen —ook al zijn ze structureel en inhoudelijk nog steeds verschillend— toch in dezelfde klasse van functionele pseudo-isomorfen zullen vallen waardoor het aantal unieke combinaties zal dalen. Naarmate het niet-functionele aandeel stijgt, zal dit effect steeds duidelijker worden. Deze intuïtieve aanpak wordt bevestigd in Figuur 5.4(links). Het aantal functionele pseudo-isomorfen stijgt gedurende de eerste generaties, met een maximum op generatie zeven, waarna het opnieuw daalt. Op deze generatie zit de maximum fitheid reeds op 90% van zijn hoogste waarde. Wanneer het aantal verschillende combinaties blijft afnemen, stagneert de maximum fitheid. Het is duidelijk dat in deze toepassing er een duidelijk verlies aan structurele diversiteit optreedt. Het aantal functionele pseudo-isomorfen geeft zonder twijfel

een accurater beeld van de reëele diversiteit in vergelijking met de originele diversiteitsmaten. Deze laatste blijven immers stijgen! Men zou verkeerdelijk de indruk krijgen dat er reeds een zeer grote structurele verscheidenheid aan individuen beschikbaar is terwijl dit duidelijk niet zo is.

Bovendien werd reeds vermeld dat na verloop van tijd een steeds groter deel van de populatie dezelfde wortelstructuur deelt (McPhee & Hopper 1999). Aangezien dichter bij de beginknoop gelegen code meer kans heeft om functionele code te zijn dan code verder van de beginknoop, zal dit effect zeker ook meespelen in het dalen van het aantal pseudo-isomorfe klassen wanneer niet-functionele code niet meegerekend wordt.

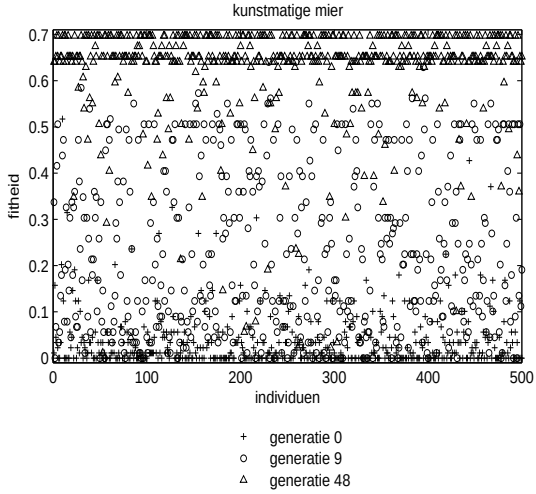
Deze bevindingen zijn sterk verschillend van de resultaten bij het regressieprobleem. Het aantal (functionele) pseudo-isomorfen stijgt snel gedurende de eerste helft van de simulatie (generaties 0 tot 30). Vervolgens stijgt het aantal pseudo-isomorfen langzaam terwijl het aantal functionele pseudo-isomorfen stagneert of zelfs lichtjes daalt. Het hoogteverschil tussen beide curven wordt wellicht veroorzaakt door de aanwezigheid van een beperkte hoeveelheid niet-functionele code zoals weergegeven in Figuur 5.3d. De kortstondige terugval gedurende de eerste generaties is wellicht te wijten aan een trage start van de groei van de programmacode zoals in Figuur 5.3a (generatie 0 tot 10).

In tegenstelling tot de kunstmatige mier, valt er geen sterke daling in het aantal functionele pseudo-isomorfen op te merken. We kunnen dit echter als volgt verklaren. Codegroei heeft een andere functie bij het regressieprobleem (zie bespreking resultaten standaard GP). De aanwezigheid van verschillende constanten, functies en hun eigenschappen (commutativiteit, associativiteit van de optelling en vermenigvuldiging) zorgen ervoor dat er heel veel varianten van de correcte oplossing bestaan die de beoogde doelfunctie trachten te benaderen. Dit grote aantal structureel verschillende kandidaat-oplossingen kan blijven bestaan aangezien hun fitheid hoog is en omdat onderlinge kruisingen even fitte individuen voortbrengen (zie ook Burke *et al.* (2004)).

We besluiten dat de vernieuwde genotypische diversiteitsmaat een duidelijker beeld geeft in vergelijking met de originele definitie van pseudo-isomorfen en dit vooral bij toepassingen die gebukt gaan onder het juk van niet-functionele code. Figuur 5.4 geeft duidelijk aan dat, in tegenstelling tot wat er vaak werd aangenomen, er wel degelijk structurele convergentie optreedt bij de kunstmatige mier. Dit kan verklaren waarom maximum fitheid stagneert. Bij het regressieprobleem blijven er echter zeer veel structureel verschillende oplossingen bestaan omwille van de redenen die hierboven werden beschreven, en kunnen we niet spreken over een structurele convergentie.

5.5.3 Fenotypische diversiteit bij standaard GP: de evolutie van de entropie

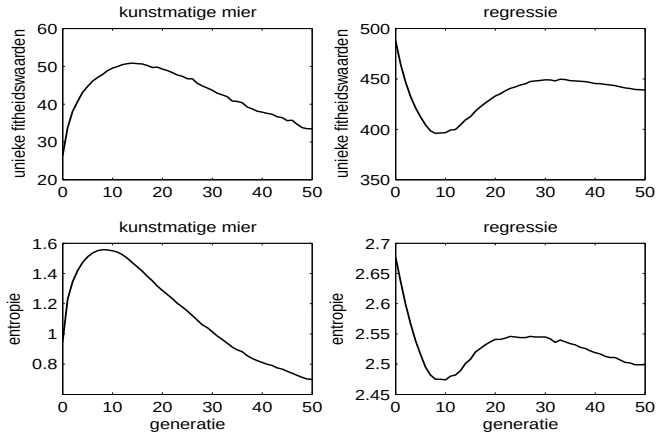
In deze paragraaf bespreken we kort de evolutie van de entropie bij beide benchmarktoepassingen. De entropie staat bekend als een zeer krachtig hulpmiddel om fenotypische diversiteit op te meten (Burke *et al.* 2004) en wordt door tal van auteurs gebruikt. Entropie kan men beschouwen als een maat die uitdrukt hoeveel chaos in de verzameling fitheidswaarden aanwezig is. We gebruiken Rosca's entropie (Rosca 1995): $-\sum_k^N p_k \cdot \log p_k$. Elke unieke fitheidswaarde vormt een afzonderlijke partitie (aangeduid door index k). Het aantal fitheidswaarden in een partitie k gedeeld door het totaal aantal fitheidswaarden (met N de grootte van de populatie, hier dus concreet gelijk aan 500) geeft p_k.



Figuur 5.5: *Spreiding van de verschillende fitheidswaarden bij de kunstmatige mier op generaties 0, 9 en 49. Bij generatie nul (+) situeert de fitheid van de meeste individuen zich rond lage waarden onder* 0.1*. Na enkele generaties optimaliseren worden de fitheidswaarden meer verspreid over het fitheidsbereik (•). Na het convergentiepunt verzamelen de fitheidswaarden zich rond de verschillende lokale optima (△).*

Bij toepassingen met een beperkte verzameling discrete fitheidswaarden zoals de mier, verwachten we in de beginfase van de simulatie weinig unieke fitheidswaarden (Fig. 5.5(+)), geconcentreerd rond lage waarden (weinig chaos). De entropie is dus relatief klein (Fig. 5.6). Na enkele rondjes optima-

liseren en kruisen van verschillende betere kandidaat-oplossingen, zullen de fitheidswaarden meer verspreid liggen tussen nul en één waardoor het aantal unieke fitheidswaarden zal stijgen (Fig. 5.5(•)). Ook de entropie zal stijgen aangezien aantal partities stijgt en de fitheidswaarden meer 'uitgesmeerd' worden (meer chaos). Dit kunnen we duidelijk vaststellen in Figuur 5.6. Naar het einde toe (of misschien beter: naar het convergentiepunt toe), zal het ontstaan van verschillende lokale optima ervoor zorgen dat de fitheidswaarden hiernaartoe getrokken worden waardoor een omgekeerd effect ontstaat (Fig. 5.5(△)): minder partities (minder chaos) en dus een lagere entropie.



Figuur 5.6: *Fenotypische diversiteit bij de kunstmatige mier en het regressieprobleem uitgezet per generatie.*

Bij toepassingen met een continu fitheidsbereik is de evolutie van de entropie sterk verschillend. Zowel de entropie alsook het aantal unieke fitheidswaarden bij het regressieprobleem (continue fitheidswaarden) vertonen een daling gedurende de eerste generaties, gevolgd door een stijging om tenslotte opnieuw lichtjes te dalen. De hoge startwaarde van de entropie is te wijten aan de willekeurige samenstelling van de populatie bij het opstarten van de simulatie (de dieptehelling zorgt voor een grote structurele en fenotypische verscheidenheid aan individuen). Hierdoor ligt de entropie voor die initiële populatie heel dicht bij de maximaal mogelijke entropie (500 verschillende fitheidswaarden):

$$entropie_{max} \quad = \quad -\sum_{k=1}^{500} \frac{1}{500} \cdot \log \frac{1}{500}$$

$$
\begin{aligned}
&= -\log \frac{1}{500} \\
&= 2.69897
\end{aligned}
$$

Vervolgens zal, onder invloed van de selectiedruk, een groot aantal uitzonderlijk slechte individuen uit de populatie worden verwijderd. Het aantal unieke fitheidswaarden zal bijgevolg dalen (en dus het aantal partities) waardoor ook de entropie daalt (meer orde en minder chaos). Codegroei zal het ontstaan van talloze structureel verschillende oplossingen in de hand werken. Deze oplossingen hebben vaak niet alleen een verschillende boomstructuur maar ook een verschillende fitheid (meer fitheidswaarden en dus meer chaos). Vandaar dat de entropie opnieuw zal stijgen tot wanneer (sub)optimale kandidaat-oplossingen worden gevonden die andere boomstructuren aantrekken.

We moeten wel opmerken dat de entropiewaarden vrij beperkt zijn gelet op de schaal van de $Y-$as in de rechtse grafiek van Figuur 5.6. Aangezien fitheid een reëel getal is tussen nul en één, bestaan er een quasi oneindig aantal verschillende waarden.

Een uitgebreide bespreking van de entropie en mogelijke alternatieve implementatie voor toepassingen met een continu fitheidsspectrum vallen buiten het bestek van dit proefschrift. De geïnteresseerde lezer verwijzen we naar (Wyns *et al.* 2006).

5.6 Besluiten

Voortijdige convergentie, gekoppeld aan een verlies van diversiteit binnen de populatie, is een belangrijk probleem bij genetisch programmeren. De literatuur herbergt talloze verschillende maten om diversiteit binnen een populatie te kwantificeren. Toch staan een aantal van deze maten nog niet volledig op punt.

Codegroei beïnvloedt het aantal pseudo-isomorfen in de populatie. Door de boomstructuur eerst te ontdoen van alle niet-functionele code en pas dan het aantal pseudo-isomorfen te berekenen, krijgen we een accurater beeld van de aanwezige genotypische diversiteit. Resultaten bij de kunstmatige mier tonen aan dat er wel degelijk structurele convergentie optreedt in tegenstelling tot wat de andere diversiteitsmaten doen vermoeden.

Deze experimenten zijn gebeurd op een beperkt aantal toepassingen en er treedt in de resultaten een zekere mate van probleemafhankelijkheid op. Toch

zijn we ervan overtuigd dat de structurele diversiteitsmaat op basis van functionele programmacode breed toepasbaar is, i.h.b. bij problemen waarbij de kandidaat-oplossingen in hoofdzaak bestaan uit niet-functionele code. Het bepalen van de functionele pseudo-isomorfen vereist het opstellen van herleidingsregels, maar dit enkel om onderscheid te maken tussen code die wel en die niet bijdraagt, een onderscheid dat bij elke toepassing bestaat.

Vaak is het zo dat het gebruik van multi-objectief technieken of het toepassen van simplificatieoperatoren bij de bestrijding van codegroei een negatieve invloed uitoefent op de diversiteit binnen een populatie (Ekárt & Németh 2001, Langdon & Nordin 2000). Ook het gebruik van de lokale optimalisatieoperator beschreven in hoofdstuk 4 veroorzaakt een verlies aan diversiteit bij hoge instelwaarden voor de probabiliteit waarmee deze operator toegepast wordt. In hoofdstuk 6 zullen we het aantal functionele pseudo-isomorfen en/of de entropie uit dit hoofdstuk gebruiken om deze invloed van dichterbij te onderzoeken.

Delen van dit hoofdstuk werden gepubliceerd op een Europese conferentie en in *Lecture Notes in Computer Science* (Wyns *et al.* 2006).

Hoofdstuk 6

Adaptieve sturing en optimalisatie van LOSE

Lokale optimalisatie van de boomstructuur blijkt een zeer effectief hulpmiddel in de strijd tegen codegroei (hoofdstuk 4). Toch vertoont de lokale optimalisatieoperator nog een belangrijke tekortkoming. Zo is het gebruik van een vaste instelwaarde voor de probabiliteit waarmee LOSE wordt toegepast, onverenigbaar met het dynamische karakter van evolutie.

In dit hoofdstuk analyseren we de evolutie van de verzameling geschikte deelbomen en trachten we een verklaring te vinden waarom bepaalde instellingen voor de probabiliteit waarmee LOSE wordt toegepast, minder geschikt zijn en leiden tot verlies aan diversiteit. Vervolgens stellen we twee adaptieve sturingsalgoritmen voor die deze probabiliteit gedurende de evolutie zullen aanpassen.

6.1 Problemen met lokale optimalisatie

De lokale optimalisatieoperator slaagt erin om opnieuw compacte individuen te evolueren en een verdere positieve evolutie van maximum fitheid te vrijwaren. Toch is er nog één belangrijke hindernis die we moeten overwinnen indien we wensen te spreken van een goed werkend GP systeem: de starre statische instelling van de probabiliteit waarmee LOSE wordt toegepast.

Tot hiertoe hebben we steeds een vaste instelling gehanteerd voor de probabiliteit waarmee lokale optimalisatie wordt toegepast (p_{lose}). Deze instelling wordt bekomen door wat men *parameter tuning* noemt, een veredelde term voor *trial and error*. Afhankelijk van de beschikbare rekentijd wordt slechts een 'handvol' waarden uitgeprobeerd.

Statische instellingen hebben heel wat nadelen. Zo is het niet altijd duidelijk welke instelling nu de beste resultaten geeft wanneer men verschillende maatstaven hanteert (bijvoorbeeld maximum fitheid en compactheid). Dit wordt algauw een multi-objectief optimalisatieprobleem waarbij men toegevingen zal moeten doen op beide maatstaven (Pareto-optimaal).

De instelling wordt gekozen op basis van de resultaten op het stopcriterium (generatie 100). Snelle stijgingen in maximum fitheid gedurende de eerste generaties kunnen ook een belangrijke prestatiemaat zijn in tegenstelling tot de waarde van maximum fitheid op het einde van de laatste generatie.

Een statische instelwaarde is weinig flexibel en past zich niet aan aan de actuele noden van het programma. Bij sommige benchmarktoepassingen groeit de gemiddelde boomgrootte opnieuw naar het einde toe. Met een vaste instelwaarde is het niet mogelijk om hierop in te pikken.

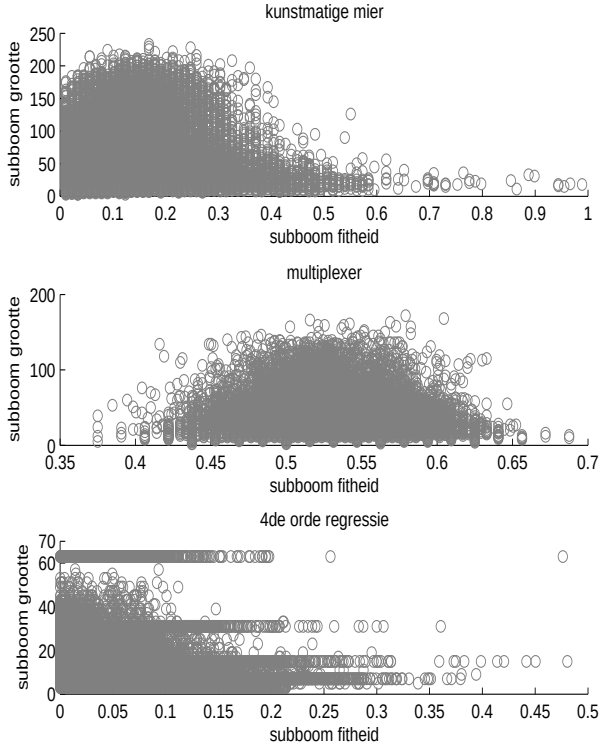
In dit hoofdstuk trachten we een antwoord te vinden op twee belangrijke vragen: "Hoe werken LOSE en de verschillende deelboomselectiestrategieën en wat is hun invloed op diversiteit (vanaf paragraaf 6.2)?" en "Kunnen we de kennis hieromtrent gebruiken om een betere instelling voor p_{lose} te kiezen (paragraaf 6.5)?".

6.2 Een uitgebreide zoekruimte

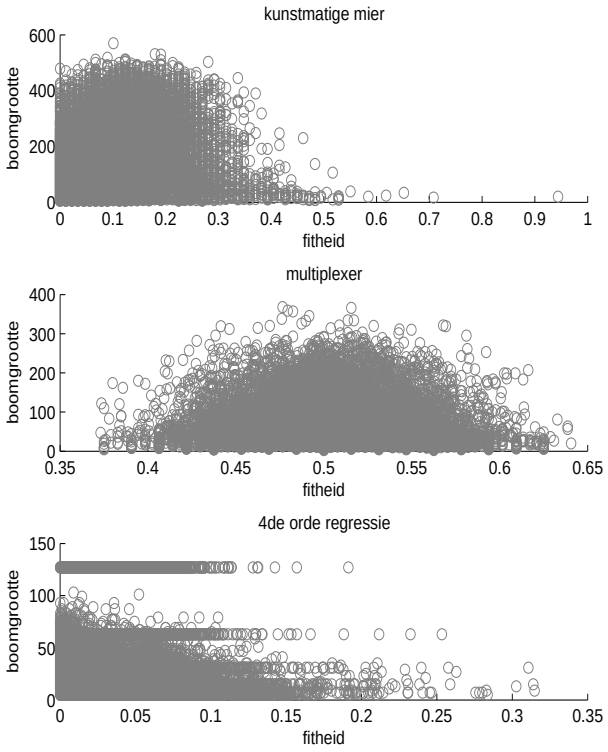
De lokale optimalisatieoperator zoekt naar een geschikte deelboom binnen de ruimte van alle mogelijke deelbomen. In hoofdstuk 4 werden twee belangrijke voorwaarden opgelegd aan een 'geschikte' deelboom: de wortel is steeds een interne knoop verschillend van de wortel van de volledige boom en de fitheid moet steeds minstens gelijk zijn aan de fitheid van de volledige boom.

In deze paragraaf tonen we de samenstelling van de uitgebreide zoekruimte van geschikte deelbomen en bespreken we enkele belangrijke algemene en probleemspecifieke eigenschappen. We beperken ons tot de verzameling deelbomen die voldoen aan bovenstaande voorwaarden.

Voor alle benchmarktoepassingen werd de samenstelling van deze zoekruimte getoond op de initiële generatie en op generatie 10. De getoonde figuren bevatten data van 100 verschillende simulaties.



Figuur 6.1: *Een spreidingsgrafiek van de grootte en de fitheid van alle geschikte deelbomen in de populatie tijdens de initiële generatie voor alle benchmarktoepassingen. De fitheid van de deelbomen moet groter of gelijk zijn aan de fitheid van het volledige individu.*



Figuur 6.2: *Een spreidingsgrafiek van boomgrootte en fitheid van alle individuen in de populatie tijdens de initiële generatie voor alle benchmarktoepassingen.*

6.2.1 De startpopulatie

Figuur 6.1 geeft een overzicht van de boomgrootte en fitheid van de verzameling van geschikte deelbomen op de initiële generatie voor alle benchmarktoepassingen. Figuur 6.2 geeft een aantal spreidingsgrafieken van de fitheid en boomgrootte van alle individuen uit de populatie (enkel volwaardige individuen, geen deelbomen). Tabel 6.1 geeft een overzicht van het totaal aantal deelbomen en maakt een opsplitsing afhankelijk van de fitheid van de deelboom.

probleem	mier	multiplexer	regressie
totaal	16019.0	11023.9	8683.6
$F_{\text{deelboom}} > F_{\text{ouder}}$	1826.4	6387.3	7211.8
$F_{\text{deelboom}} < F_{\text{ouder}}$	12771.2	4257.2	1471.8
$F_{\text{deelboom}} = F_{\text{ouder}}$	1421.4	379.4	0
geschikte deelbomen/totaal (%)	20%	61%	83%
geschikte deelbomen per individu	6.5	13.5	14.4

Tabel 6.1: *Deze Tabel geeft een overzicht van het totaal aantal deelbomen tijdens de initiële generatie (rij 2) en maakt vervolgens een opsplitsing afhankelijk van de fitheid van de deelboom ten opzicht van de fitheid van het volledige individu (rijen 3, 4 en 5). Rij 6 geeft het percentage van geschikte deelbomen weer ($F_{deelboom} >= F_{ouder}$). De laatste rij berekent het gemiddeld aantal geschikte deelbomen per individu (populatie bevat 500 individuen). De data is afkomstig van de initiële generatie en werd uitgemiddeld over 100 simulaties.*

Algemene vaststellingen

De boomgrootte van de individuen uit de populatie (Fig. 6.2) is steeds groter dan de boomgrootte van de geschikte deelbomen (Fig. 6.1). Dit is eenvoudig te verklaren aangezien een deelboom per definitie een interne knoop is, verschillend van de wortel van de volledige boom. Hierdoor zal de diepte steeds minstens één eenheid kleiner zijn waardoor inherent ook de boomgrootte daalt.

Tevens door de opgelegde voorwaarde aangaande de fitheid, zal de gemiddelde fitheid van de geschikte deelbomen hoger liggen dan de gemiddelde fitheid van de individuen uit de populatie. Wanneer we echter 'alle' deelbomen (dus het aantal deelbomen vermeld in de tweede rij uit Tabel 6.1) in beschouwing nemen dan is de gemiddelde fitheid niet significant verschillend in beide groepen. Beide populaties werden immers willekeurig samengesteld.

Wanneer we Figuren 6.1 en 6.2 met elkaar vergelijken dan zien we dat er tijdens de initiële generatie veel geschikte deelbomen bestaan. Vaak is het dan ook zo dat één individu gemiddeld verschillende deelbomen bevat waarvan de fitheid groter is dan de fitheid van de volledige boom (Tabel 6.1, laatste rij). In hoofdzaak omvangrijke individuen (dieptehelling 6) bevatten meer interne knopen en bijgevolg is de kans groot om minstens één geschikte deelboom te vinden.

Probleemspecifieke vaststellingen

Bij de kunstmatige mier zijn de geschikte deelbomen met een hoge fitheid (bijvoorbeeld groter dan 0.5) steeds zeer compact. Bij lagere fitheidswaarden is de variatie in boomgrootte dan weer groter. Geschikte subbomen met een hogere fitheid gaan gerichter op zoek naar voedsel en ontwikkelen een echt spoorvolggedrag. Dit resulteert in een compacte voorstelling en dus een beperkte boomgrootte. De aanwezigheid van grotere deelbomen is wijten aan het evolutionair voordeel van dergelijke structuren. Aangezien ze meer knopen bevatten, bestrijken ze meer terrein. Hierdoor is de kans om voedsel tegen te komen, groter waardoor toch een aanvaardbaar niveau van fitheid wordt behaald.

Bij de kunstmatige mier is het aanbod van geschikte deelbomen beperkt (slechts 20% en gemiddeld 6 per individu). Zoals reeds werd vermeld hebben grotere individuen een evolutionair voordeel omdat ze meer terrein kunnen bestrijken. Hierdoor is het moeilijker om een geschikte deelboom te vinden, vooral in een populatie van willekeurig gegenereerde individuen (en dus ook deelbomen).

Ook bij het regressieprobleem zien we dat geschikte deelbomen met hoge fitheid (groter dan 0.35) compact zijn en dat er meer variatie in boomgrootte optreedt bij lage fitheid. Bij deze toepassing zijn fitte geschikte deelbomen samengesteld uit elementaire bouwelementen die terugkomen in de correcte oplossing terwijl minder fitte exemplaren de oplossing trachten te benaderen.

Bij het regressieprobleem is het aantal deelbomen met exact dezelfde fitheid gelijk aan nul. Dit is te wijten aan de continue fitheidswaarde die wordt gebruikt waardoor de kans dat er twee exact dezelfde waarden voorkomen zeer klein (zoniet onbestaande) is.

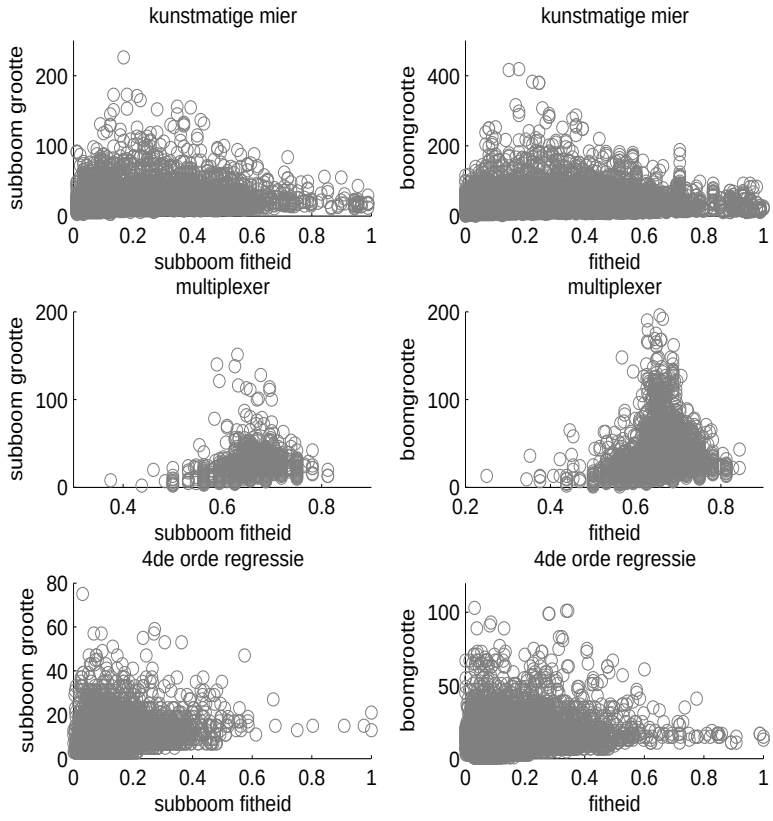
De grafiek bij het multiplexerprobleem is klokvormig. Bij hogere (> 0.65) en lagere fitheidswaarden (< 0.40) is de boomgrootte meestal beperkt. Tussenin bestaat er een zeer grote verscheidenheid aan geschikte deelbomen (verschillende groottes per fitheidswaarde).

6.2.2 De volgende generaties

Het beeld uit de startpopulatie blijft ook geldig gedurende de volgende generaties. Uiteraard is dit afhankelijk van de instelwaarde van de probabiliteit waarmee LOSE wordt toegepast en het gekozen deelboomselectiemechanisme. We nemen als voorbeeld de situatie op generatie 10, na toepassing van

LOSE met BSS als deelboomselectiestrategie en we gebruiken de parameterinstellingen uit Tabel 4.2, paragraaf 4.3.3.

Figuur 6.3 geeft, bij wijze van voorbeeld, een aantal spreidingsgrafieken weer van de boomgrootte en fitheid van alle individuen (rechtse grafieken) en geschikte deelbomen (linkse grafieken) in de populatie op generatie 10. Na tien generaties is het (positieve) effect van LOSE reeds zeer duidelijk zichtbaar op de evolutie van boomgrootte en fitheid en zijn we voldoende ver verwijderd van de initiële willekeurige populatie. Tenslotte zijn de negatieve bijwerkingen van LOSE ten gevolge van een vaste instelling voor p_{lose} niet of nauwelijks zichtbaar.



Figuur 6.3: *Een spreidingsgrafiek van boomgrootte en fitheid van alle geschikte deelbomen (links) en van alle individuen in de populatie (rechts) tijdens generatie 10 voor alle benchmarktoepassingen.*

Algemene vaststellingen

De gemiddelde boomgrootte van de verzameling geschikte deelbomen is nog steeds kleiner dan de gemiddelde boomgrootte van de individuen uit de populatie. Deze reductie is minder uitgesproken in vergelijking met de initiële generatie. De voorwaarden opgelegd aan een geschikte deelboom, het actief gebruiken van lokale optimalisatie waardoor de gemiddelde boomgrootte in de originele populatie afneemt en de aanwezigheid van compacte individuen zonder geschikte deelboom (BSS), zijn hier de onderliggende oorzaken van.

Na enkele rondjes van optimalisatie met behulp van LOSE lijkt het alsof de fitheid van de groep geschikte deelbomen lager is dan de fitheid van de verzameling individuen in de populatie. Toch is de gemiddelde fitheid van de geschikte deelbomen nog steeds groter dan de gemiddelde fitheid van de deelverzameling van individuen die over een geschikte deelboom beschikken (wegens de voorwaarden van een geschikte deelboom). Het stijgend aantal fitte individuen die te compact zijn om nog verder geoptimaliseerd te worden (en dus het dalend aantal geschikte deelbomen per individu), beïnvloeden in positieve zin het beeld dat men krijgt van de gemiddelde fitheid van de populatie. Op de rechtse grafieken staan immers alle individuen afgedrukt.

Het is misschien niet duidelijk af te leiden uit de spreidingsgrafieken, toch is het aantal geschikte deelbomen sterk afgenomen ten opzichte van de initiële generatie. Bij alle toepassingen bevat een individu gemiddeld minder dan één geschikte deelboom. Dit kunnen we als volgt verklaren. Afhankelijk van p_{lose} zal de lokale optimalisatieoperator continu de boomgrootte van individuen reduceren. Hierdoor zal de populatie in hoofdzaak bestaan uit kleine compacte individuen met hoge fitheid. Bovendien, door de toenemende selectiedruk en de stijging van de gemiddelde fitheid, is de kans dat één van deze interne knopen een deelboom met hogere fitheid voorstelt, kleiner waardoor het aantal geschikte deelbomen zal afnemen. Ook het gebruik van BSS in deze experimenten verhindert het vinden van nog betere deelbomen aangezien BSS enkel de beste deelboom behoudt.

Anderzijds bestaat er een minderheid (overgespecialiseerde) individuen die een ideale kweekbasis vormen voor nieuwe geschikte deelbomen. Deze beperkte groep is verantwoordelijk voor de meerderheid geschikte deelbomen maar kan toch niet verhinderen dat het steeds moeilijker wordt om een geschikte deelboom aan de haak te slaan. Dit werd experimenteel geverifieerd door het aantal geschikte deelbomen van groter dan gemiddelde individuen te tellen. Dit aantal lag vele malen hoger dan het aantal deelbomen bij kleinere individuen. Beschouwen we als voorbeeld het vierde orde regressieprobleem. In deze toepassing zullen een aantal simulaties de correcte functie trachten te

benaderen. Deze benaderende oplossingen bevatten zeer veel knopen. En dus is de kans dat men minstens één geschikte deelboom met een hogere fitheid vindt opnieuw groter waardoor het aantal geschikte deelbomen in deze groep van individuen zal stijgen.

Probleemspecifieke vaststellingen

Bij het multiplexerprobleem stijgt het aandeel geschikte deelbomen waarbij geen verbetering noch verslechtering van de fitheid wordt vastgesteld. De fitheid blijft onaangetast maar de boomgrootte wordt wel beperkt.

6.2.3 Belangrijke tendensen

Zoals aangetoond zijn er initieel veel geschikte deelbomen beschikbaar; gemiddeld méér dan één geschikte deelboom per individu. Dit aantal geschikte deelbomen neemt echter snel af naarmate het aantal generaties verstrijkt. Nochtans krijgt codegroei na een aantal generaties opnieuw de bovenhand (zoals vermeld in hoofdstuk 4) waardoor de gemiddelde boomgrootte stijgt. Hierdoor verwachten we een nieuwe stijging van het aantal deelbomen. Toch kunnen we dit niet afleiden uit de resultaten. Het aantal geschikte deelbomen neemt verder af of vlakt uit. We hebben experimenteel aangetoond dat er een aantal belangrijke verklaringen zijn waarom het aantal geschikte deelbomen generatie na generatie daalt.

Ten eerste, door de toenemende selectiedruk (stijging van de fitheid), wordt het steeds moeilijker om een deelboom te vinden waarvan de fitheid groter (of gelijk) is aan de fitheid van de volledige boomstructuur.

Ten tweede zal de LOSE operator (afhankelijk van de instelwaarde van p_{lose}) een deel van de individuen stelselmatig reduceren waardoor de boomgrootte kleiner wordt. Afhankelijk van de gekozen selectiestrategie bevatten deze individuen zelf geen geschikte deelbomen meer. Ingeval van BSS bijvoorbeeld wordt steeds de beste deelboom eruit gepikt. Deze deelboom (die door LOSE omgevormd wordt tot een volwaardig individu) zal tijdens de volgende generatie geen enkele andere deelboom bevatten waarvoor de fitheid minstens gelijk is.

Een derde reden hiervoor is dat in de simulaties waar codegroei zich opnieuw voordoet enkel de grootste boomstructuren verantwoordelijk zijn voor het vinden van ‘verschillende’ geschikte deelbomen. Merk op dat er uiteindelijk slechts één deelboom verder gebruikt zal worden door LOSE.

Het dalend aantal geschikte deelbomen vormt een belangrijke indicatie voor de optimale werking van LOSE. Bij de keuze van een geschikte instelwaarde van p_{lose} moeten we hiermee rekening houden.

6.3 De deelboomselectiemechanismen

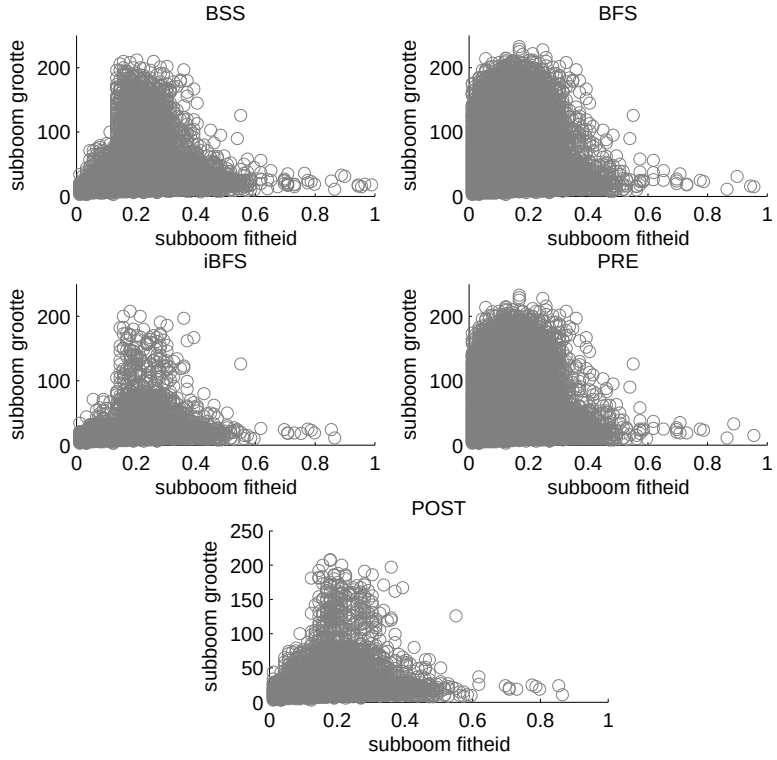
De vorige paragraaf beschreef de samenstelling en evolutie van de verzameling geschikte deelbomen. De verschillende deelboomselectiestrategieën (BSS, PRE, BFS, POST, iBFS) zullen slechts één enkele deelboom opslaan in hun datastructuur. Analoog aan de voorgaande paragraaf geven we in deze paragraaf een overzicht van de evolutie van de verschillende 'geselecteerde' deelbomen. Op basis hiervan trachten we een onderscheid te maken tussen de verschillende deelboomselectiemethoden en leggen we de voor- en nadelen van bepaalde methoden uit. We zijn minder geïnteresseerd in probleemafhankelijke verschillen.

6.3.1 De startpopulatie

Figuren 6.4, 6.5 en 6.6 tonen spreidingsgrafieken van de fitheid van de deelboom en de grootte van de deelboom voor alle deelboomselectiestrategieën tijdens de initiële generatie. Elke grafiek toont enkel de geschikte deelbomen die door de betreffende deelboomselectiestrategie werd geselecteerd. Dit aantal is steeds hoogstens gelijk aan één deelboom per individu. De verschillen tussen deze spreidingsgrafieken werden ook statistisch met elkaar vergeleken (multivariaat met Kruskall-Wallis test en univariaat met paarsgewijze Wilcoxon rangtesten met Tukey correctie). Figuur 6.7 toont een grafische weergave van enkele statistische eigenschappen met behulp van verschillende boxplot grafieken voor de fitheid en de grootte van de deelbomen.

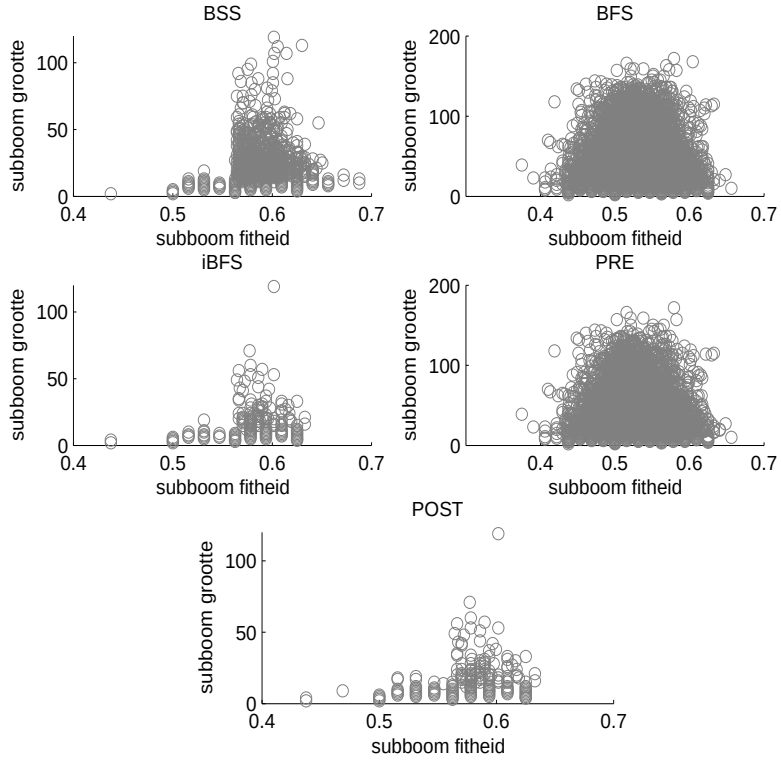
BSS De volgorde van knopen bezoeken in een boomstructuur is onbelangrijk bij de methode BSS. BSS zoekt enkel de deelboom met de hoogste fitheid. Wanneer we de spreidingsgrafieken van BSS met de andere strategieën onderling vergelijken dan ligt de gemiddelde boomgrootte bij BSS tussen de andere twee groepen (BFS en PRE enerzijds, en iBFS en POST anderzijds) in. Dit is ook wat met behulp van paarsgewijze Wilconxon rangtesten met Tukey correctie werd vastgesteld (Figuur 6.7).

De beste fitheid wordt vanzelfsprekend behaald met de methode BSS.



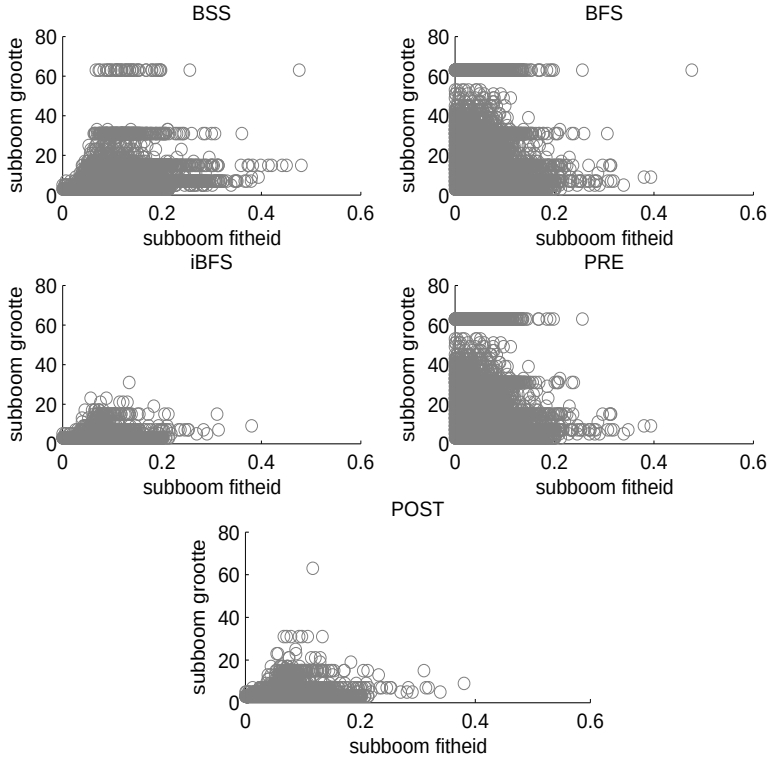
Figuur 6.4: *Een spreidingsgrafiek van boomgrootte en fitheid van alle geschikte deelbomen geselecteerd door de verschillende deelboomselectiestrategieën tijdens de initiële generatie voor de kunstmatige mier.*

PRE en BFS De verdeling van de puntenwolk bij BFS en PRE vertoont zeer veel gelijkenis met de verzameling van alle geschikte deelbomen (gelijke variantie, mediaan en gemiddelde). We proberen dit te verklaren aan de hand van de variatie binnen de originele populatie (Fig. 6.2). In de initiële populatie bestaan er een groot aantal structureel verschillende oplossingen. Deze grote verscheidenheid wordt veroorzaakt door gebruik van de half-en-half initialisatiemethode in combinatie met een dieptehelling tussen 2 en 6. Aangezien de populatie op willekeurige basis werd gegenereerd en dus de fitheid niet bijzonder geoptimaliseerd is naar de toepassing toe, is de kans om een deelboom te vinden die het minstens evengoed doet, relatief groot (zie Tabel 6.1, gemiddelde méér dan één geschikte deelboom per individu). Hierdoor kunnen we veronderstellen dat alle deelboomselectiestrategieën relatief snel



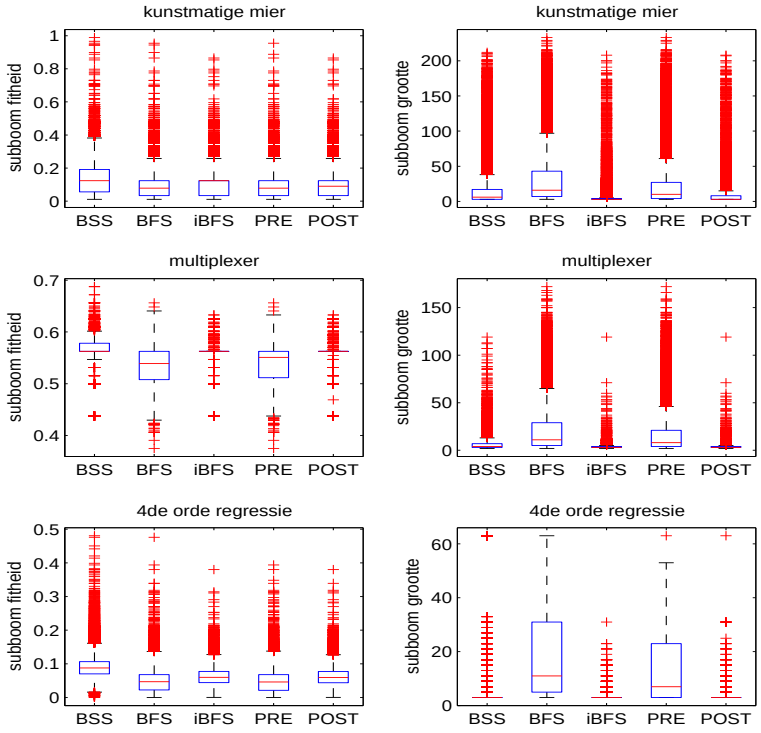
Figuur 6.5: *Een spreidingsgrafiek van boomgrootte en fitheid van alle geschikte deelbomen geselecteerd door de verschillende deelboomselectiestrategieën tijdens de initiële generatie voor de Boolean multiplexer.*

een geschikte deelboom zullen vinden. Bij PRE en BFS zal de wortel zich eerder bovenaan in de boom bevinden waardoor de diepte van de boom weinig afwijkt van de diepte van het ouder individu. De bekomen deelbomen zijn wel kleiner (omdat je steeds minstens één niveau afdaalt in de boom) maar de structurele verschillen daarom niet minder groot. Hierdoor zal de bestaande variatie bij een deelboomselectiestrategie die begint met het bezoeken van de bovenste knopen (zoals BFS en ook PRE) grotendeels behouden blijven. Dit werd geverifieerd door de variantie te berekenen van de deelboom grootte per verandering van fitheidswaarde. Voor PRE en BFS zijn deze waarden steeds groter dan voor de andere selectiestrategieën.



Figuur 6.6: *Een spreidingsgrafiek van boomgrootte en fitheid van alle geschikte deelbomen geselecteerd door de verschillende deelboomselectiestrategieën tijdens de initiële generatie voor het vierde orde regressieprobleem.*

Wat betreft fitheid presteren PRE en BFS het minst. Enkel bij de kunstmatige mier produceren beide selectiestrategieën beter dan POST en iBFS. PRE en BFS selecteren knopen die hoger in de boom gelegen zijn en produceren dus grotere deelbomen. We hebben reeds vermeld dat grotere boomstructuren gedurende de initiële generaties een evolutionair voordeel hebben aangezien ze over meer knopen beschikken en dus een groter terrein kunnen bestrijken. Hierdoor is de fitheid van deze geschikte deelbomen dan ook groter dan bij geschikte deelbomen geselecteerd door iBFS en POST (eerder onderaan de boom en dus kleiner).



Figuur 6.7: *Boxplots van de fitheid (links) en de grootte (rechts) van de geschikte deelbomen voor alle benchmarktoepassingen.*

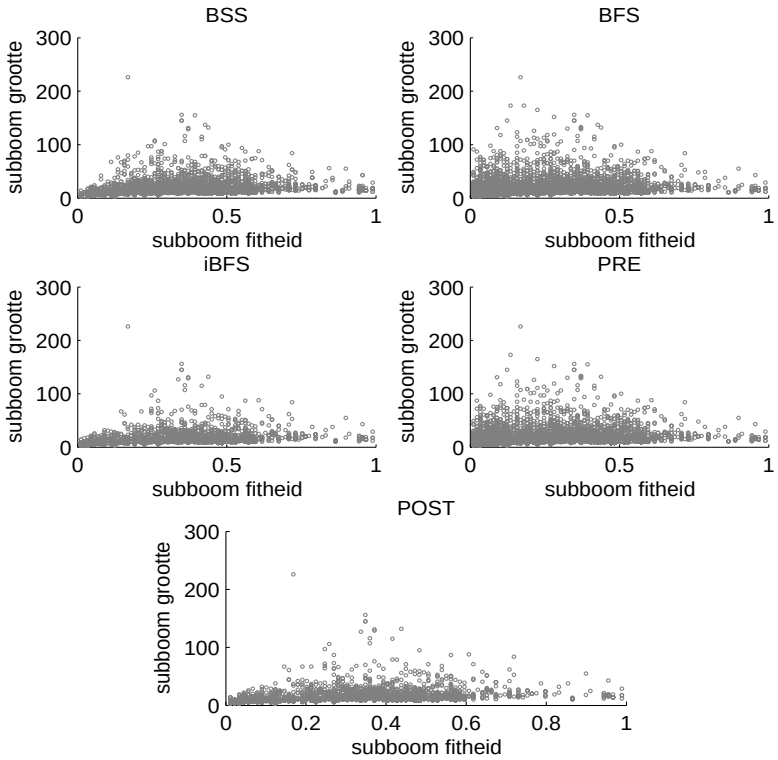
POST en iBFS Wat onmiddellijk opvalt, is dat de puntenwolken bij iBFS en POST veel meer geconcentreerd zijn in vergelijking met de overige methoden[1]. Binnen eenzelfde populatie worden er dus maar weinig verschillende geschikte deelbomen gevonden (louter op basis van het aantal knopen en de fitheid). Bovendien concentreren de puntenwolken zich rond lage waarden voor deelboom grootte. Vooral bij de multiplexer is dit onderscheid bijzonder duidelijk.

POST en iBFS selecteren geschikte deelbomen onderaan startend in de boomstructuur. In vorige paragraaf hebben we vastgesteld dat de startpopulatie meerdere geschikte deelbomen per individu bevat. Wellicht vinden beide methoden tijdens de initiële 'willekeurig' gegenereerde populatie voldoende

[1]Op elke grafiek staat exact hetzelfde aantal cirkels

geschikte deelbomen op lagere dieptes. Aangezien de zoektocht bij POST en iBFS onmiddellijk afgebroken wordt bij het vinden van een geschikte deelboom, worden hoger gelegen deelbomen niet verder geëvalueerd.

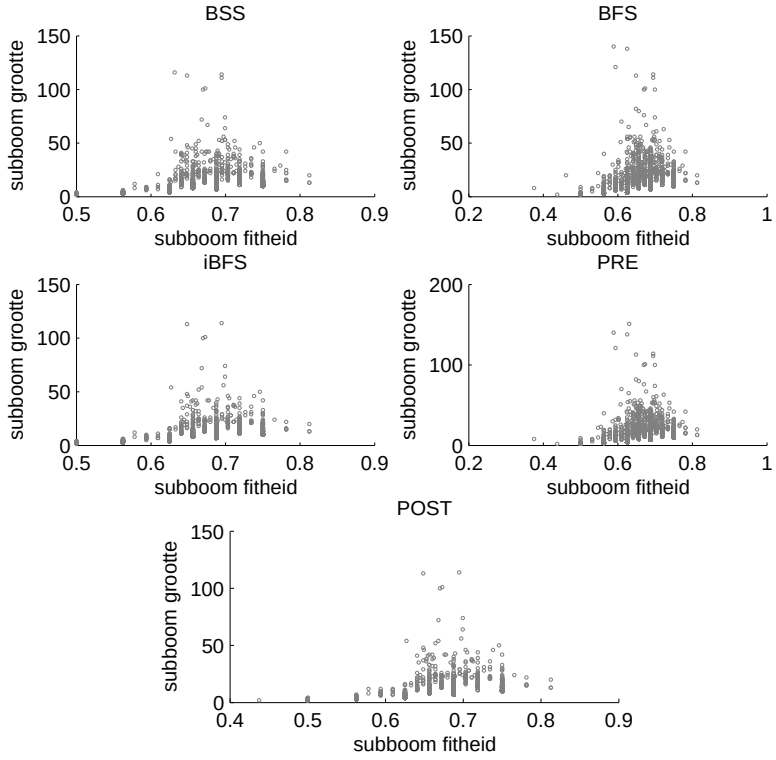
POST en iBFS presteren tussen beide methoden in. De kwaliteit van de deelbomen is slechter in vergelijking met BSS maar beter dan de groep (PRE, BFS). Gesteund door Ockham's theorie vermoeden we dat de kleine compacte deelbomen beter presteren.



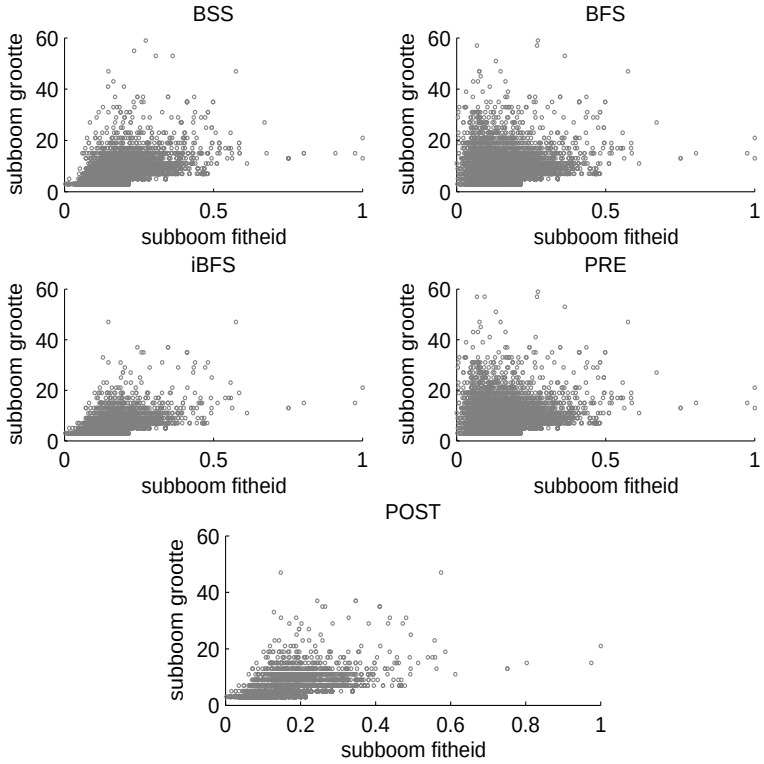
Figuur 6.8: *Een spreidingsgrafiek van boomgrootte en fitheid van alle geschikte deelbomen geselecteerd door de verschillende deelboomselectiestrategieën tijdens generatie 10 voor de kunstmatige mier.*

6.3.2 De volgende generaties

We gebruiken dezelfde instellingen als in Tabel 4.2, paragraaf 4.3.3. Ook nu vragen we ons af of er wezenlijke verschillen zullen optreden na enkele generaties van optimalisatie met behulp van LOSE. Figuren 6.8, 6.9 en 6.10 tonen spreidingsgrafieken van de fitheid en de grootte van de deelbomen op generatie 10 voor alle deelboomselectiestrategieën. Figuur 6.11 toont enkele statistische eigenschappen van de gegevens aan de hand van boxplots. Het onderscheid tussen de verschillende deelboomselectiestrategieën is nu minder duidelijk dan bij de initiële populatie. Toch springen er nog steeds een aantal bijzonderheden in het oog.



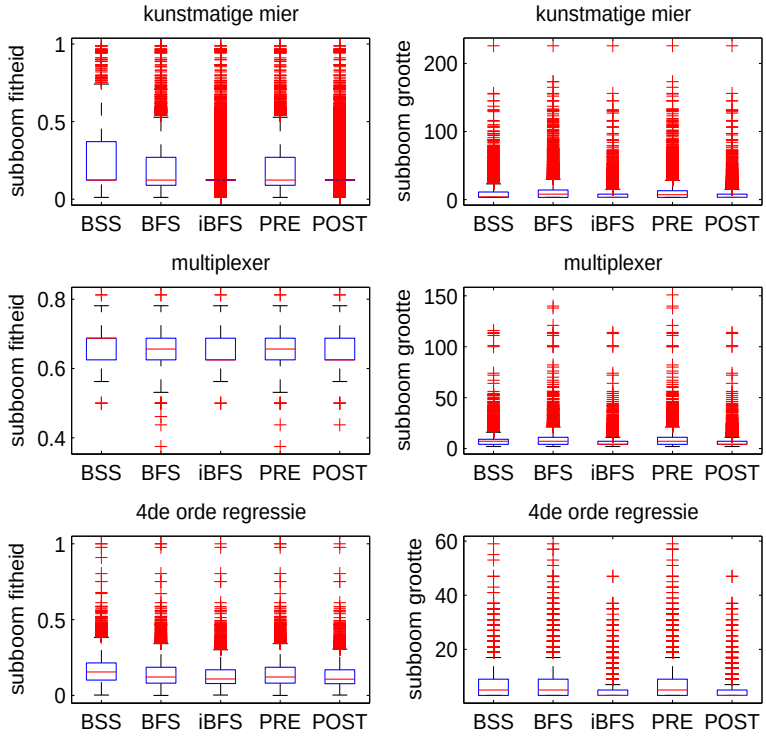
Figuur 6.9: *Een spreidingsgrafiek van boomgrootte en fitheid van alle geschikte deelbomen geselecteerd door de verschillende deelboomselectiestrategieën tijdens generatie 10 voor de Boolean multiplexer.*



Figuur 6.10: *Een spreidingsgrafiek van boomgrootte en fitheid van alle geschikte deelbomen geselecteerd door de verschillende deelboomselectiestrategieën tijdens generatie 10 voor het vierde orde regressieprobleem.*

BSS De deelboom grootte ligt tussen de andere methoden in. De fitheid van de deelbomen is nog steeds het grootst bij BSS (door het verschil in zoekalgoritme tussen de methoden).

PRE en BFS De samenstelling van de verzameling deelbomen geselecteerd door de methoden PRE en BFS is nog steeds gevarieerder in vergelijking met de andere methoden (i.h.b. iBFS en POST, in mindere mate ten opzichte van BSS). Dit kan men afleiden uit de hoogte van de *box* in Figuur 6.11. BFS en PRE selecteren ook steevast de grootste geschikte deelbomen (significant verschil met iBFS en POST en bij de kunstmatige mier en bij de Boolean multiplexer ook met BSS).



Figuur 6.11: *Boxplots van de fitheid (links) en de grootte (rechts) van alle deelbomen voor alle benchmarktoepassingen.*

PRE en BFS selecteren significant betere deelbomen dan POST en iBFS. PRE en BFS zijn minder geneigd om vast te raken in lokale optima (zeer compacte structuren) en zoeken eerder naar een totaaloplossing voor het probleem.

POST en iBFS iBFS en POST selecteren nog steeds de kleinste deelbomen (significant verschil ten opzichte van de andere methoden voor alle toepassingen). Toch is reeds op generatie 10 duidelijk dat de kwaliteit van de geschikte deelbomen hieronder te lijden heeft. Na veelvuldig toepassen van LOSE zal de gemiddelde boomgrootte sterk afgenomen zijn. Als men deze cirkel niet tijdig kan doorbreken, blijven er vaak enkel suboptimale kandidaatoplossingen over met een wel zeer beperkt aantal knopen. Beide methoden

zorgen dus voor weinig variëteit in de verzameling van deelbomen. Bij de methode iBFS is dit effect nog duidelijker dan bij POST. Dit is bijzonder duidelijk bij de 11-bit multiplexer waar vroeger reeds werd opgemerkt dat $p_{\text{lose}} = 40\%$ een snelle convergentie naar (te) compacte individuen tot gevolg had.

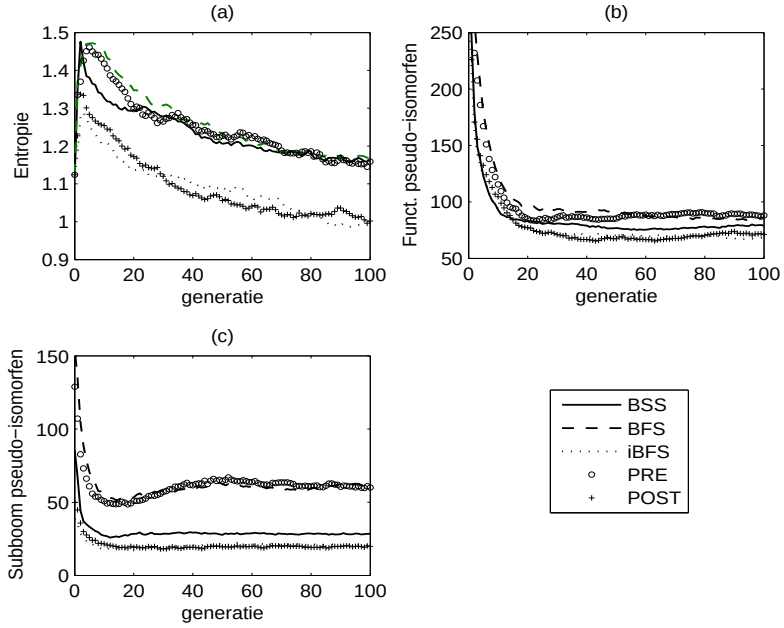
6.3.3 Belangrijke tendensen

Op basis van de selectie van een deelverzameling van deze geschikte deelbomen door de deelboomselectiestrategieën, kunnen we een aantal belangrijke tendensen in verband met de werking van LOSE onderscheiden, vooral tijdens de initiële generatie. Na een aantal generaties evolueren de geselecteerde deelbomen immers naar elkaar toe. De onderlinge verschillen tussen de deelboomselectiemethoden zijn dan miniem (op generatie 10 vertonen de meeste grafieken een zeer sterke verwantschap). Dit is te wijten aan het dalend aantal geschikte deelbomen. Subtiele verschillen kunnen dan wel eens grote gevolgen hebben.

De deelboomselectiemethoden iBFS en POST selecteren steeds kleine, compacte geschikte deelbomen. Dit is een logisch gevolg van de gehanteerde zoekstrategie. Beide methoden bezoeken immers de dieper gelegen knopen vroeger en er wordt gestopt zodra er een betere deelboom wordt gevonden. De gevonden deelbomen bieden aanvankelijk een goede fitheid die boven het populatiegemiddelde uitsteekt maar evolueren onvoldoende mee gedurende de rest van de evolutie. We kunnen deze initieel gevonden deelbomen ook beschouwen als hoogwaardige deeloplossingen van het gestelde probleem. POST en iBFS zullen echter steeds deze suboptimale oplossingen blijven selecteren (aangezien er een aanbod blijft) waardoor de reductie van de boomgrootte groot is maar de positieve evolutie van fitheid nauwelijks verder wordt gestimuleerd. Dit is vooral heel duidelijk bij de Boolean multiplexer en kan verklaren waarom de resultaten (zoals getoond in hoofdstuk 4) zo tegenvallen voor beide methoden. Met dit in het achterhoofd kunnen we in de toekomst een betere instelwaarde voor p_{lose} kiezen indien we gebruiken maken van *bottom-up* zoekstrategieën.

Omgekeerd zoeken PRE en BFS naar een totaaloplossing voor het voorliggend probleem. Helaas, door de ingebakken conditie dat de selectiemethode moet stoppen zodra de fitheid van de deelboom groter of gelijk is aan de fitheid van de volledige boom zullen beide methoden te vroeg stoppen met zoeken. De resulterende boom bevat nog vrij veel code die niet bijdraagt tot de correcte oplossing van het probleem. Vaak maar niet altijd kan men op

eenvoudige wijze deze code verder vereenvoudigen. BFS en PRE bieden een gevarieerd aanbod aan deelbomen en zijn minder gevoelig aan de instelwaarde van p_{lose} waardoor een nog grotere reductie mogelijk is.

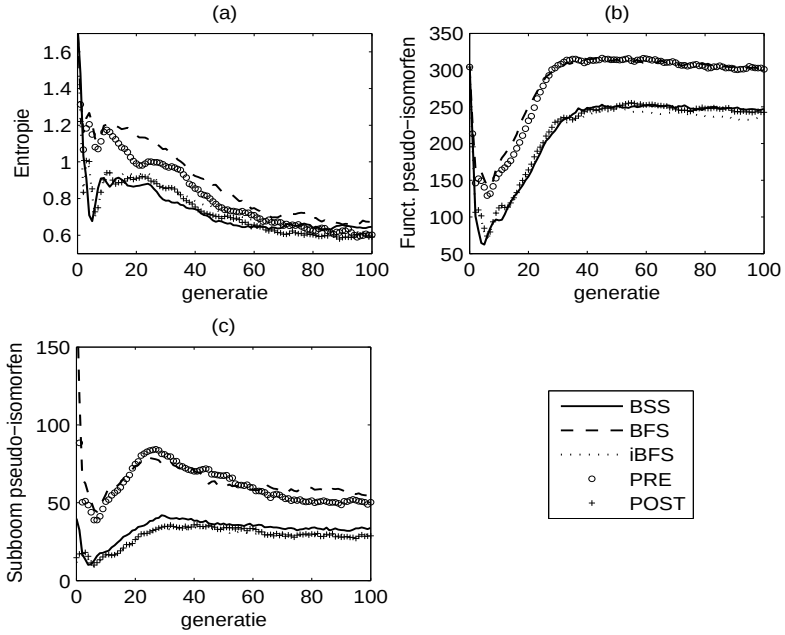


Figuur 6.12: *Entropie, het aantal functionele pseudo-isomorfen en het aantal pseudo-isomorfen in de verzameling deelbomen bij de kunstmatige mier.*

6.4 Diversiteit en een vaste p_{lose}

In deze paragraaf bespreken we eerst kort de evolutie van de verschillende diversiteitsmaten (zoals beschreven in hoofdstuk 5). We beschouwen zowel de structurele en fenotypische diversiteit binnen een populatie alsook de structurele diversiteit binnen de verzameling van geschikte deelbomen. We schenken bijzondere aandacht aan de structurele diversiteitsmaat gebaseerd op het aantal geschikte deelbomen. Voor de berekening van deze diversiteitsmaat werden enkel individuen die over een geldige deelboom beschikken, gebruikt. Deze geschikte deelboom werd niet vereenvoudigd. We hebben aangetoond dat het aantal geschikte deelbomen in de populatie gevoelig afneemt

na de eerste generatie (weergegeven voor 'alle' generaties in Fig. 6.15). Dit beïnvloedt uiteraard de evolutie van het aantal deelboom pseudo-isomorfen.



Figuur 6.13: *Entropie, het aantal functionele pseudo-isomorfen en het aantal pseudo-isomorfen in de verzameling deelbomen bij de Boolean multiplexer.*

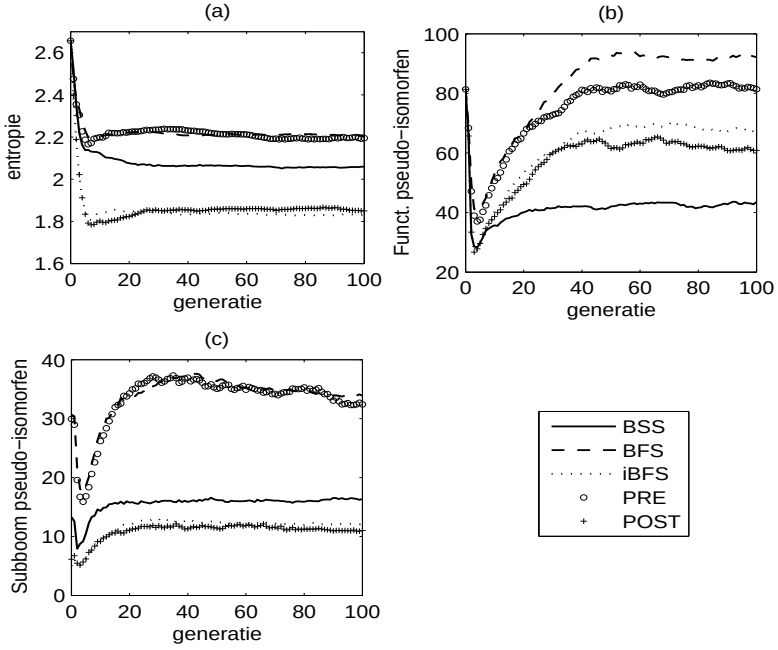
6.4.1 Diversiteit bij de gebruikte instellingen voor $p_{\mathbf{lose}}$ uit hoofdstuk 4

Figuren 6.12–6.14 tonen de entropie, het aantal functionele pseudo-isomorfen en het aantal deelboom pseudo-isomorfen voor alle benchmarktoepassingen. Figuur 6.15 toont het aantal individuen met een geschikte deelboom voor alle benchmark problemen.

Algemene vaststellingen

Op basis van deze grafieken komen we tot een aantal algemene vaststellingen:

1. Vooreerst merken we op dat bij alle benchmarktoepassingen een beperkt aantal deelboom pseudo-isormorfen aanwezig is (rekening hou-



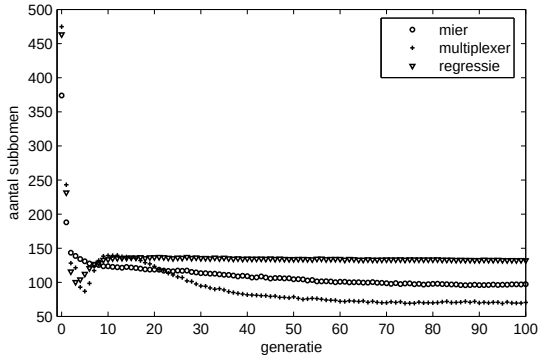
Figuur 6.14: *Entropie, het aantal functionele pseudo-isomorfen en het aantal pseudo-isomorfen in de verzameling deelbomen bij het vierde orde regressieprobleem.*

dend met het aantal deelbomen in de populatie, Fig. 6.15). Dit is vooral te wijten aan de beperkte boomgrootte van deze geschikte deelbomen (beperkte diepte, beperkt aantal eindknopen en functieknopen) waardoor het aantal pseudo-isomorfe klassen eveneens beperkt is.

2. Het kleine aantal deelboom pseudo-isomorfen heeft een invloed op de structurele diversiteit in de populatie van volwaardige individuen (telkens grafiek (b)). Ook het aantal functionele pseudo-isomorfen is kleiner in vergelijking met standaard GP (zonder LOSE). Dit kunnen we eenvoudig verklaren aan de hand van de werking van LOSE. De lokale optimalisatieoperator zal immers elke generatie een vast aantal geschikte deelbomen in de populatie brengen. Hierdoor zal de gemiddelde boomgrootte van de individuen kleiner zijn, alsook het aantal functionele pseudo-isomorfen. Dit wordt versterkt door de beperkte structurele diversiteit binnen de verzameling deelbomen.

3. We stellen tevens een snelle initiële daling vast van het aantal deelboom pseudo-isomorfen gedurende de eerste (vier) generaties. Vooral het sterk dalend aantal geschikte deelbomen dat beschikbaar is in de populatie (Fig. 6.15) is hier de oorzaak van (uiteraard speelt het effect van LOSE ook mee).

4. In paragraaf 6.3 stelden we op basis van de boxplots en de spreidingsgrafieken vast dat BFS en PRE steeds meer gevarieerde geschikte deelbomen selecteren in vergelijking met iBFS en POST. Dit werd nogmaals bevestigd in Figuren 6.7 en 6.11 waar we tevens opmerkten dat BFS en PRE grotere deelbomen selecteren. Ook de evolutie van het aantal deelboom pseudo-isomorfen bevestigen dit. Grotere geschikte deelbomen (dus meer knopen en diepere boomstructuur) leiden tot het bestaan van meer pseudo-isomorfe klassen en dus tot een hoger aantal pseudo-isomorfen (Fig. 6.12–6.14(b,c)). De structurele diversiteit bij BFS en PRE is dan ook steeds groter dan bij iBFS en POST. Beste deelboomselectie valt tussen beide klassen in (alhoewel het eerder aansluiting vindt bij POST en iBFS).



Figuur 6.15: *Het aantal individuen met een geschikte deelboom voor alle benchmarktoepassingen. De gebruikte deelboomselectiestrategie is BSS. De instelwaarde voor p_{lose} is weergegeven in Tabel 4.2.*

Probleemspecifieke vaststellingen

Na een aantal generaties treedt er bij de multiplexer en het regressieprobleem eerst een stijging op van het aantal (deelboom) pseudo-isomorfen, uiteinde-

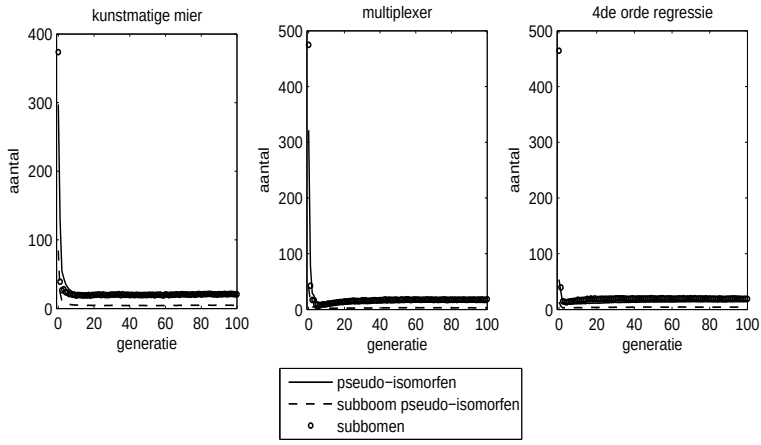
lijk gevolgd door een uitvlakking[2]. In hoofdstuk 4 hebben we vastgesteld dat bij beide toepassingen de code na enkele generaties opnieuw toeneemt. Hierdoor neemt de diepte en het aantal knopen toe wat uiteraard het aantal functionele pseudo-isomorfen verhoogt alsook het aantal deelboom pseudo-isomorfen (zeker wanneer BFS of PRE worden gebruikt). Wanneer we kijken naar de verhouding tussen de diepte van de volledige boomstructuur en de diepte waarop de geschikte deelboom zich bevindt (enkel voor de individuen die door LOSE worden gekozen), dan neemt deze verhouding af na enkele generaties. Met andere woorden, tijdens de initiële generaties is de geschikte deelboom dieper gelegen terwijl na verloop van tijd de diepte van het mutatiepunt (= de wortel van de deelboom) hoger komt te liggen. De resulterende deelbomen zijn groter waardoor ook het aantal deelboom pseudo-isomorfen stijgt. Bij het regressieprobleem merken we wel op dat de resultaten op de latere generaties (na generatie 40) gebaseerd zijn op een zeer beperkt aantal simulaties aangezien de meeste uitvoeringen van het GP programma reeds vroeger een correcte oplossing hebben gevonden. De simulaties die er niet in geslaagd zijn een correcte oplossing te vinden, trachtten de functie te benaderen (groot aantal knopen waarvan veel niet-functioneel).

Bij de kunstmatige mier krijgen we bij standaard GP zonder codegroeibegrenzers initieel een stijging, vrijwel onmiddellijk gevolgd door een daling. Wanneer LOSE wordt gebruikt, treedt de initiële stijging niet op ten gevolge van de reductie in boomgrootte veroorzaakt door LOSE. Er is geen reden meer om grotere boomstructuren met veel eindknopen (die veel terrein bestrijken) te promoten.

6.4.2 Problemen met een hoge statische instelwaarde

We gaan tevens de invloed van zeer hoge instelwaarden voor p_{lose} na op de populatie (kiezen we bijvoorbeeld voor een instelwaarde gelijk aan 80%). Voor de eenvoud beschouwen we enkel de methode BSS. Bij zeer hoge instelwaarden voor p_{lose} is het zo dat er maar weinig verschillen te ontdekken zijn tussen de verschillende deelboomselectiestrategieën. Voor een populatie met 500 individuen betekent dit dat er elke generatie steeds 400 individuen door LOSE worden gecreëerd. De resterende individuen worden via kruising (95 individuen) en reproductie (5 individuen) bekomen.

[2]De evolutie van de verschillende diversiteitsmaten bij het regressieprobleem is vrijwel analoog aan de situatie bij standaard GP, behoudens een verschil in amplitude veroorzaakt door het gebruik van LOSE (Fig. 5.4 en 5.6).



Figuur 6.16: *Het aantal pseudo-isomorfen binnen de populatie, het aantal pseudo-isomorfen binnen de verzameling geschikte deelbomen en het aantal individuen met een geschikte deelboom (voor 80% LOSE). De gebruikte deelboomselectiestrategie is BSS.*

Zoals Figuur 6.16 aantoont, heeft het gebruik van een dergelijke hoge waarden desastreuze gevolgen voor de diversiteit binnen de populatie en leidt dit tot zeer kleine boomstructuren met suboptimale (lage) fitheid. Quasi alle simulaties genereren individuen die heel wat kleiner zijn dat de kleinste 100% correcte oplossing.

Maar wat is nu de achterliggende reden dat de diversiteit zo snel uitsterft? Hiervoor bekijken we opnieuw de werking van de LOSE operator en het bijhorende selectiemechanisme, alsook het aantal geschikte deelbomen tijdens elke generatie (Fig. 6.16, ○).

Veronderstellen we even dat alle individuen over een geschikte deelboom beschikken. Wanneer BSS wordt gebruikt, zal, in de volgende generatie, slechts 20% (=100) van de individuen nog over een geschikte deelboom beschikken. BSS selecteert immers de beste deelboom waardoor tijdens de volgende evaluatiefase geen enkele deelboom meer kan gevonden worden met een fitheid die hoger is. Bij de andere deelboomselectiemethoden (BFS, PRE, iBFS en POST) kunnen we spreken van hetzelfde verschijnsel alhoewel dit meer uitgesmeerd wordt over een beperkt aantal generaties. Uit de grafiek die het aantal geschikte deelbomen weergeeft, is duidelijk af te leiden dat enkel tijdens de initiële generatie voldoende deelbomen aanwezig zijn om aan de vraag vanuit de LOSE operator te voldoen (met uitzondering van de mier). Vaak is het bijkomend ook zo dat de resterende 20% individuen die door kruising

worden gegenereerd niet steeds over een geschikte deelboom beschikken. In realiteit zijn er dus dikwijls té weinig geschikte deelbomen aanwezig.

Een tweede onderliggende reden wordt gegeven door de implementatie van het selectiealgoritme dat door LOSE wordt aangewend om zich te voorzien van geschikte individuen. In hoofdstuk 4, merkten we op dat de lijst bij de meeste selectiealgoritmen (elitaire deelboomselectie) circulair is. Op het einde van de lijst wordt opnieuw teruggekeerd naar het begin waardoor, indien de lijst heel kort is, vaak dezelfde individuen en dus dezelfde geschikte deelboom door LOSE in de nieuwe populatie wordt gebracht. We geven een voorbeeld. Wanneer er in generatie 1 slechts 40 geschikte individuen (met een fittere deelboom) worden gevonden en LOSE wordt gedurende 80% van de tijd toegepast, zullen van elk individu maar liefst 10 kopieën in de nieuwe populatie worden gebracht. Hierdoor zal de structurele diversiteit aanzienlijk afnemen en convergeert de populatie naar kleine suboptimale oplossingen.

Uiteraard geven we hier een extreem voorbeeld waarvan intuïtief te begrijpen valt dat $p_{lose} = 80\%$ te hoog is. Nochtans treedt structureel diversiteitsverlies op bij een groot aantal instelwaarden van LOSE, afhankelijk van de gekozen benchmarktoepassing. Bij de multiplexer en het regressieprobleem zien we soortgelijke problemen reeds vanaf een instelwaarde gelijk aan 50% opduiken. Bij de kunstmatige mier pas vanaf p_{lose} gelijk aan 70%. Zoals reeds herhaaldelijk opgemerkt, zijn sommige deelboomselectiemethoden hiervoor gevoeliger dan andere. Denken we maar aan POST of iBFS bij het multiplexerprobleem. Bovendien is deze instelling wel representatief gedurende latere generaties bij een dalend aantal geschikte deelbomen.

6.4.3 Enkele algemene besluiten

Op basis van deze eenvoudige experimentele proefopzet kunnen we de volgende besluiten trekken.

Hoge instelwaarden voor p_{lose} vernietigen alle bestaande structurele diversiteit binnen de populatie, ongeacht welke deelboomselectiemethode er wordt gekozen of over welke toepassing het gaat. De genetische operatoren zoals kruising die zorgen voor de toevoer van geschikte deelbomen kunnen onmogelijk de snelheid waarmee nieuwe deelbomen worden getransformeerd, bijhouden. De vraag is simpelweg te groot en het aanbod te klein.

Bovendien, door de wijze waarop het selectiealgoritme werkt, worden er verschillende identieke kopieën van deelbomen in de nieuwe populatie ingebracht waardoor de structurele diversiteit na enkele generaties tot een minimum wordt herleid.

We weten ook dat het aantal geschikte deelbomen daalt naarmate de generaties verstrijken. Hierdoor is het mogelijk dat een aanvaardbare instelling voor p_{lose} bekomen via parameter tuning toch een te hoge instelling blijkt te zijn gedurende de latere generaties. Uit een analyse van het aantal geschikte deelbomen blijkt tevens dat het aanbod niet steeds constant is en dat er op elke generatie niet evenveel deelbomen voorhanden zijn. De noodzaak voor een methode die de instelwaarde afstemt op het probleemtype en op de beschikbare hoeveelheid nuttig genetisch materiaal, dringt zich op. Een vaste en gedurende de evolutie onveranderbare instelwaarde voor p_{lose} is slechts ideaal gedurende een beperkt aantal generaties (in het begin is de waarde meestal te klein terwijl het op het einde net omgekeerd is).

Samengevat zijn er de volgende nadelen verbonden aan het gebruik van een vaste instelling voor p_{lose}:

- Het gebruik van p_{lose} vereist een extra parameter die de applicatieprogrammeur moet instellen. GP vraagt van nature reeds een heleboel parameterinstellingen en deze extra instelling bemoeilijkt nog maar eens de taak van de programmeur. Temeer omdat deze instelling afhankelijk is van het probleem waarop GP in combinatie met LOSE wordt toegepast. Bovendien is er een extra (beperkte) afhankelijkheid van de deelboomselectiemethode waarvoor wordt geopteerd.

- Evolutie is een tijdsafhankelijk proces. Op verschillende momenten tijdens de evolutie is er een verschillende noodzaak aan LOSE. Dit dynamische karakter is in tegenstrijd met de statische instelling.

- Er zijn niet altijd voldoende geschikte deelbomen aanwezig om aan de vraag van LOSE te voldoen. Bij te weinig geschikte deelbomen wordt de populatie opgevuld met identieke kopieën die negatieve gevolgen kunnen hebben voor de verdere positieve evolutie van fitheid. Ook dit is in tegenstrijd met het statische karakter van p_{lose}.

Om deze negatieve bijwerkingen op te lossen wordt in de volgende paragraaf op zoek gegaan naar geschikte alternatieven om een zo optimaal mogelijke instelling voor p_{lose} per generatie te bekomen.

6.4.4 Verwijderen van duplicaten

Vergelijken we nogmaals het aantal beschikbare deelbomen (Fig. 6.15) en het aantal deelboom pseudo-isomorfen (Fig. 6.12c–6.14c), dan merken we dat

het aantal deelboom pseudo-isomorfen vaak kleiner is dan het aantal geschikte deelbomen. De selectielijst zelf bevat dus mogelijks een aantal duplicaten. Ongeacht de invloed die de instelling van p_{lose} uitoefent (zie voorgaande paragrafen), zullen deze kopijen de structurele (en fenotypische) diversiteit doen afnemen. Om dit te vermijden werd het selectiemechanisme dat de lijst met geschikte deelbomen opbouwt, aangepast. Vooraleer het identificatienummer van een individu wordt toegevoegd aan de lijst, controleert men eerst of er reeds een identieke deelboom in de lijst werd opgenomen. Indien een dergelijke deelboom bestaat, wordt het individu verder buiten beschouwing gelaten. Op deze manier wordt de structurele diversiteit gemaximaliseerd.

Het verwijderen van duplicaten in de selectielijst wordt geactiveerd aan de hand van een parameterinstelling die wordt meegegeven met de LOSE operator. Tenzij anders vermeld wordt dit steeds toegepast.

6.5 Een adaptieve instelling voor p_{lose}

De nadelen van parameter tuning zijn onderhand reeds gekend (Eiben *et al.* 1999) en werden ook in dit hoofdstuk uitvoerig besproken. Een beter alternatief voor parameter tuning is *parameter control* (parametersturing); de parameters worden initieel gelijk gesteld aan een bepaalde waarde maar veranderen gedurende de uitvoering van een experiment.

Parametersturing, of het dynamisch aanpassen van de instelwaarden tijdens de evolutionaire uitvoering, kan men verder onderverdelen in drie categoriën: deterministische parametersturing, zelf-adaptieve en adaptieve parametersturing (Eiben *et al.* 1999).

De eerste methode gebruikt een deterministische regel om de waarde van een parameter aan te passen. Er wordt geen feedback vanuit de zoekstrategie gebruikt. Met mogelijks belangrijke randinformatie, zoals diversiteit, behaalde fitheid, boomgrootte, etc. wordt geen rekening gehouden. Een voorbeeld van een dergelijke strategie is de volgende regel om de probabiliteit van bijvoorbeeld mutatie aan te passen tijdens de evolutie:

$$p_{\text{mutatie}} = 1 - 0.9 \times \frac{t}{T}$$

waarbij T het maximum aantal generaties is en t de index van de huidige generatie. Deze methode heeft als nadeel dat ze enige voorkennis vereist van het te sturen proces opdat de deterministische regel optimaal afgestemd zou zijn. Deze voorkennis is, bij een stochastisch systeem zoals GP, moeilijk

exact te beschrijven. Verder vereist deze methode op zijn beurt een extra parameter, nl. de regel die wordt gebruikt. Een eerste belangrijke vraag is welke parameters we zullen opnemen in deze regel. Een tweede vraag is hoe we deze parameters zullen samenvoegen.

In vorige hoofdstukken hebben we gezien dat codegroei meestal na verloop van tijd zal optreden zodat we kunnen opteren voor de volgende (naïeve) formule:

$$p_{\text{lose}} = 0.1 + 0.9 \times \frac{t}{T}$$

De instelwaarde zal op deze manier lineair stijgen en zijn maximum bereiken zodra $t = T$. Deze methode gebruikt geen bijkomende informatie vanuit de GP populatie (enkel de index van de huidige generatie). Meteen merken we enkele tekortkomingen aan deze deterministische beslissingsregel. Zo leiden niet alle simulaties tot codegroei. Bepaalde experimenten evolueren compacte boomstructuren zonder dat codegroei aanwezig is.

Deze methode houdt tevens geen rekening met de intensiteit waarmee codegroei optreedt. Het verloop van de instelwaarde is per definitie vast en onveranderbaar. Soms is het echter nuttig om tijdelijk meer LOSE toe te passen (of net minder). Met bovenstaande formule is dit niet mogelijk.

Een derde reden waarom deze regel onvoldoende bescherming biedt, werd reeds aangetoond in voorgaande paragraaf. Daar wordt aangetoond dat het aantal geschikte deelbomen daalt naarmate de evolutie vordert. Er zijn dus niet steeds voldoende deelbomen beschikbaar waardoor, zeker tijdens de laatste generaties, het aantal kopieën van deelbomen die in de populatie worden gebracht snel toeneemt. LOSE op deze manier aansturen zal de instelwaarde net verhogen! Het is duidelijk dat deterministische beslissingsregels teveel voorkennis vragen van het codegroei fenomeen om praktisch bruikbaar te zijn. We moeten als het ware de evolutie van de boomgrootte op voorhand voorspellen en vastleggen in één formule. Meer nog, de analyses van de resultaten zijn vaak gebaseerd op het gemiddelde over 100 simulaties. De besluiten zijn wel geldig voor dit gemiddelde maar geven misschien niet altijd de beste resultaten per simulatie.

Een tweede categorie van methoden noemt men zelf-adaptieve parametersturing. De centrale idee is om evolutie te gebruiken om de evolutie aan te sturen en is vooral populair binnen het domein van genetische algoritmen (Bagley 1967, Brindle 1981, Goldberg & Smith 1987, Greene 1994, Hinterding 1995, 1997). De parameters die men wenst aan te passen, worden vaak gecodeerd in een afzonderlijk chromosoom. Deze chromosomen ondergaan de klassieke genetische operatoren zoals kruising, mutatie en reproductie. De

betere waarden voor de geëncodeerde parameters geven aanleiding tot een hogere fitheid en dus een betere overlevingskans. Een genetisch algoritme leent zich perfect hiervoor. Het chromosoom wordt uitgebreid met de gecodeerde waarden van de verschillende parameters die men op adaptieve wijze wenst te veranderen. Bij genetisch programmeren is het vaak veel moeilijker om een gemeenschappelijke voorstelling te vinden voor zowel de te sturen parameterwaarden alsook de kandidaat-oplossing op het voorliggende probleem. Heel vaak zijn de functies en eindknopen waaruit de boomstructuur bestaat, zeer probleemspecifiek. Denk bijvoorbeeld maar aan de kunstmatige mier waarbij men functies voorziet om de mier vooruit te laten bewegen of te laten draaien. Dergelijke functies kunnen onmogelijk gebruikt wordt om p_{lose} aan te sturen. Het is dan ook niet verwonderlijk dat er tot op heden weinig of geen artikels zijn verschenen die zelf-adaptieve sturing van parameters bij GP beschrijft. Een mogelijkheid om het probleem van het vinden van een geschikte voorstelling te omzeilen, bestaat erin om een genetisch algoritme te gebruiken dat enkel en alleen de parameterwaarden adaptief aanpast. Het is dan echter onduidelijk en verre van triviaal om te bepalen welke parameterwaarden beter presteren dan andere. De koppeling met het genetisch programma en de bepaling van de fitheid is een bijkomend probleem. Verder vereist deze aanpak een bijkomende hoeveelheid computationele rekentijd die het sowieso al zware GP programma mogelijks nog logger maakt. Het is bovendien ook niet duidelijk wat nu precies de invloed is van de evolutie van de parameterwaarden. Net zoals bij klassieke evolutie, heeft het algoritme enige tijd nodig om kwalitatief fitte individuen te produceren. Wellicht is dit analoog voor de evolutie van fitte parameterwaarden. Het is met andere woorden niet duidelijk of zelf-adaptieve parametersturing het evolutionaire proces en in het bijzonder de zoektocht naar een correcte oplossing zal versnellen.

De laatste manier om parameters dynamisch aan te passen noemt men adaptieve parametersturing. Hierbij wordt steeds feedback vanuit de populatie gebruikt om de amplitude of richting van de waarde voor de te sturen parameter aan te passen. Deze methode vereist minder voorkennis dan het gebruik van deterministische leerregels en omzeilt tegelijkertijd ook de bepaling van de fitheid en de operatoren die de parameterwaarden zullen aansturen zoals dit het geval is bij zelf-adaptieve methoden. De feedback die men krijgt vanuit de populatie zijn vaak opgemeten parameterwaarden zoals fitheid, diversiteit en boomgrootte. In het kader van dit proefschrift werden twee zulke adaptieve methoden ontworpen: een parametersturing op basis van vaaglogica en een alternatieve eenvoudige methode op basis van de bevindingen uit paragrafen 6.2–6.4.

6.5.1 Adaptieve sturing op basis van vaaglogica

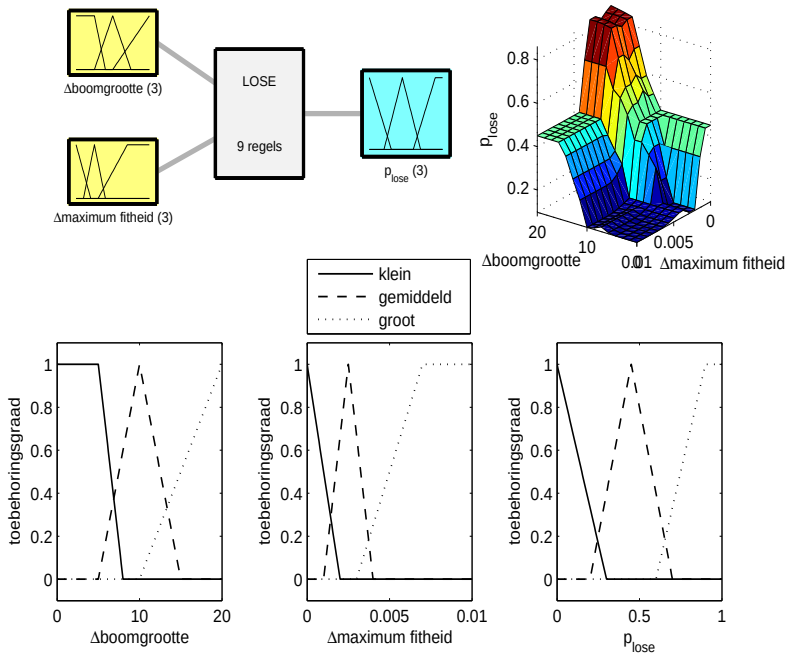
Vaaglogische regelaars (FLC) bieden een handige tool aan om linguïstische regelstrategieën die gebaseerd zijn op verworven expertise om te vormen tot een automatische regelstrategie (Driankow *et al.* 1993). Ze zijn bijzonder geschikt voor de modellering van relaties tussen variabelen waarbij de context zeer complex of moeilijk te definiëren valt. Ook codegroei is een dergelijk complex en moeilijk af te lijnen probleem. Er bestaat immers geen enkele deterministische formule waaruit men met hoge zekerheid kan afleiden dat codegroei optreedt (en in welke mate). Het vage begrip codegroei leent zich perfect tot het gebruik van dergelijke vaaglogische regelaars. De basisidee achter deze aanpak is om een FLC te gebruiken met aan de ingang een aantal opgemeten variabelen die worden afgeleid tijdens de evolutie en met aan de uitgang een voorspelling voor de instelwaarde van p_{lose}. Nadat de opgemeten invoervariabelen worden aangeboden aan de FLC, genereert deze één enkele uitvoerwaarde. Deze uitvoerwaarde wordt dan gebruikt om het aantal individuen dat door LOSE zal worden omgevormd, te bepalen. Vooraleer we de gebruikte FLC van naderbij toelichten, geven we eerst een kort overzicht van enkele kenmerken van een vaaglogische regelaar.

Ingrediënten van een FLC

Het belangrijkste onderdeel van een FLC is de verzameling van INDIEN $\Longrightarrow$ DAN regels, van welke de antecedenten en consequenties samengesteld zijn uit vage beweringen. De evaluatie van een dergelijke regel steunt op vaaglogische implicatie en compositie. Het proces noemt men vaak inferentie. Een typische FLC bestaat uit de volgende onderdelen:

- Fuzzificatie interface: deze module zal de invoervariabelen omvormen tot vage verzamelingen.
- Kennisbank (*Knowledge base*): deze databank bevat de informatie (expertise vanuit probleemdomein) onder de vorm van vaaglogische regelstructuren.
- Inferentie systeem: dit systeem gebruik de kennisbank en de vage verzamelingen (afkomstig van de fuzzificatie interface) met als doel een vaaglogische gevolgtrekking voorop te stellen.
- Defuzzificatie interface: hier gebeurt net het omgekeerde als bij de fuzzificatie interface. De vage uitvoer zal worden omgevormd tot een reële waarde die verder in het systeem kan worden gebruikt.

De kennisbank bestaat verder uit twee componenten: de databank met de linguïstische termen (zoals klein, gemiddeld, groot) en de toebehoringsgraden. Een tweede onderdeel zijn de verzameling (linguïstische) regels alsook de operator die de regels zal samenvoegen. Het is best mogelijk dat verschillende regels tegelijkertijd reageren op een bepaalde invoercombinatie. Voor meer informatie en een gedetailleerde beschrijving verwijzen we graag naar (Cordon *et al.* 2001, Driankow *et al.* 1993).



Figuur 6.17: *Een grafische voorstelling van de gebruikte vaaglogische sturing voor p_{lose}. De grafiek rechtsboven geeft het beslissingsoppervlak weer van het gebruikte model. De onderste grafieken geven de verschillende toebehoringsgraden weer voor beide invoervariabelen (Δmaximum fitheid en Δboomgrootte) en de uitvoervariabele (p_{lose}).*

De invoerparameters van een FLC

Een eerste belangrijke vraag bij adaptieve parametersturing is welke parameters we zullen gebruiken als invoervariabelen voor de vaaglogische sturing. Wat is met andere woorden de feedback die we wensen te ontvangen van-

uit de omgeving? Hiervoor hernemen we even de definitie van codegroei uit hoofdstuk 3. Codegroei beschrijft de ongecontroleerde toename in boomgrootte van de boomstructuren uit de populatie. Vaak wordt dit ook onmiddellijk in verband gebracht met een niet gecorreleerde toename in de fitheid van de kandidaat-oplossingen. Beide parameters, maximum fitheid (F_{max}) en gemiddelde boomgrootte, vormen dus een uitstekend vertrekpunt. Aangezien we vooral geïnteresseerd zijn in de toename van beide opgemeten variabelen, houden we het verschil tussen de huidige en de voorgaande generatie van beide parameterwaarden bij.

$$\Delta F_{\text{max}} = (F^{t}_{\text{max}} - F^{t-1}_{\text{max}})$$

$$\Delta\text{boomgrootte} = \begin{cases} (\text{boomgrootte}^{t} - \text{boomgrootte}^{t-1}) & \text{verschil} \geq 0 \\ 0 & \text{verschil} < 0 \end{cases}$$

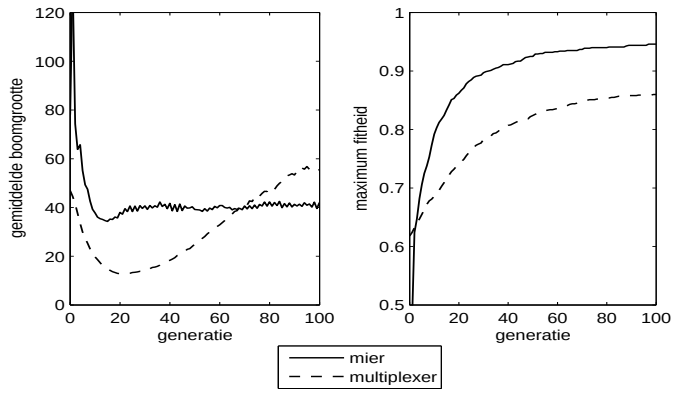
waarbij t de index van de huidige generatie aanduidt. De uitvoervariabele van de FLC is uiteraard de instelwaarde voor p_{lose}.

Vervolgens kennen we aan elke invoer- en uitvoervariabele een aantal vage klassen toe. Hier opteren we voor drie onderverdelingen (kleine toename, gemiddelde toename, grote toename) waarbij we in hoofdzaak driehoekige functies gebruiken. De instelling van de grenzen van deze functies gebeurde op basis van de analyse van de data van de verschillende benchmarktoepassingen. Er werden in totaal 9 regels gedefinieerd die de invoervariabelen afbeelden op de uitvoerwaarde voor p_{lose} (Tabel 6.2). Het resulterende beslissingsoppervlak is weergegeven rechtsboven Figuur 6.17. Eenvoudig samengevat zal de instelwaarde voor p_{lose} snel stijgen zodra de toename in gemiddelde boomgrootte groter wordt (voorbeeld: regels 1, 4, 5, 7 en 8). Een grote toename in maximum fitheid remt een stijgende p_{lose} en zorgt op deze manier voor een dempende factor (voorbeeld: regels 3, 6 en 9). Een volledig overzicht van het resulterende vaaglogische systeem is te zien in Figuur 6.17.

Verschillende instellingen voor het aantal toebehoringsgraden, de vorm van deze functies, het aantal en de samenstelling van de regels, etc. werden uitgeprobeerd. Voor alle experimenten toonde het gebruik van de vaaglogische sturing zich heel tolerant ten aanzien van deze wijzigingen. De gekozen parameterinstellingen zijn identiek voor elke toepassing (probleemonafhankelijk). Een volledig overzicht van alle parameterinstellingen is te vinden in bijlage B. Voor de eenvoud verwijzen we naar de vaaglogische sturing van p_{lose} onder de noemer FuLOSE (Fuzzy LOSE).

regel	Δ boomgrootte	$\Delta F_{\max}$	p_{lose}
1	klein	klein	gemiddeld
2	klein	gemiddeld	klein
3	klein	groot	klein
4	gemiddeld	klein	gemiddeld
5	gemiddeld	gemiddeld	gemiddeld
6	gemiddeld	groot	klein
7	groot	klein	groot
8	groot	gemiddeld	groot
9	groot	groot	gemiddeld

Tabel 6.2: *Een overzicht van de verschillende regels die de invoervariabelen afbeelden op p_{lose}. Beide invoervariabelen worden met de AND operatie gekoppeld.*



Figuur 6.18: *Gemiddelde boomgrootte en maximum fitheid bij gebruik van een vaaglogische sturing voor p_{lose} voor de kunstmatige mier en de 11-bit multiplexer.*

Resultaten

Deze aanpak werd uitgeprobeerd op alle benchmarktoepassingen. We geven echter enkel de resultaten voor de kunstmatige mier en de Booleaanse multiplexer (Figuur 6.18) aangezien er bij het regressieprobleem quasi identieke resultaten werden behaald als met een vaste p_{lose}. Telkens werd de methode BSS gebruikt om geschikte deelbomen te selecteren. De resultaten werden

vergeleken met de klassieke proefopzet (deelboomselectie eveneens via BSS en een vaste instelling voor p_{lose}, Figuren 4.6 en 4.10).

Bij de kunstmatige mier produceert FuLOSE vergelijkbare resultaten wat betreft maximum fitheid en gemiddelde boomgrootte. De eindwaarde op generatie 100 van deze laatste is wel lichtjes toegenomen (significant). Het is zeer duidelijk te merken dat het FuLOSE de groei van de code gedurende de tweede helft van de simulatie onder controle houdt (uitvlakkende curve), terwijl bij LOSE duidelijk te zien is dat de code opnieuw gestaag begint te groeien.

Bij het multiplexerprobleem zijn de bekomen oplossingen bijzonder compact maar is de maximum fitheid wel significant lager (vergelijkbaar met standaard GP)! De evolutie van de gemiddelde boomgrootte wordt vooral gekenmerkt door de zeer scherpe daling gedurende de eerste 20 generaties. Deze daling is te wijten aan een combinatie van een aantal factoren. We sommen deze kort op:

- De toename van maximum fitheid (ΔF_{max}) ligt tijdens de initiële generaties meestal rond de waarde 0.005 (groot). Deze toename zorgt enerzijds voor een beperkte demping van de instelwaarde voor p_{lose} (zie Tabel 6.2), maar anderzijds betekent dit ook dat in hoofdzaak de toename van de boomgrootte een voorname rol speelt bij de instelling van p_{lose}.

- De toename in boomgrootte bedraagt meestal een tiental knopen (gemiddeld tot groot). Hierdoor zal de uiteindelijk waarde van p_{lose} zich (vooral) bevinden in de vage verzameling 'gemiddeld'. In de praktijk vertaalt zich dit in instelwaarden tussen 0.2 en 0.45 wat bijzonder hoog is voor een toepassing zoals de multiplexer. Echter wanneer de toename in maximum fitheid eerder beperkt is, zal de instelwaarde onmiddellijk de hoogte inschieten (> 0.6) met de gekende negatieve gevolgen.

- De hoge instelwaarde voor LOSE en de werking van LOSE zelf zorgen voor een onmiddellijke daling van de boomgrootte. In de generatie die erop volgt, zal nauwelijks of zelfs helemaal geen lokale optimalisatie worden toegepast ($p_{lose} = 0.0$ indien het verschil in boomgrootte negatief is).

Vergelijken we deze resultaten met de statische instelling van LOSE (hoofdstuk 4) en paragraaf 6.4 dan kunnen we besluiten dat het probleem schuilt in

de definitie van de vage klasse "gemiddeld" voor p_{lose}. Specifiek voor de multiplexer zijn deze teruggeefwaarden te hoog.

Enkele bemerkingen bij het gebruik van FuLOSE

In deze paragraaf hebben we aangetoond dat het adaptief instellen van p_{lose} met behulp van een vaaglogische sturing een heleboel nadelen van de statische instelwaarde omzeilt. Zo wordt er actief ingegrepen op het moment dat codegroei optreedt (bijvoorbeeld bij de kunstmatige mier tijdens de tweede helft van de evolutie) en blijkt FuLOSE minder vatbaar voor schommelingen van de prestatie ten gevolge van wijzigingen van de verschillende parameterinstellingen eigen aan het FuLOSE.

Toch zijn er nog een aantal andere factoren die het gebruik van FuLOSE bemoeilijken. Zoals gezegd zijn we bij FuLOSE verplicht om een aantal extra parameters te kiezen. Zo moeten we beslissen hoeveel lidmaatschapsfuncties we wensen, alsook hun vorm en daaraan gekoppelde parameters (bijvoorbeeld bij een driehoekige lidmaatschapsfunctie de coördinaten van de drie hoeken van de driehoek), het aantal regels alsook hun concrete invulling, de defuzzificatie procedure etc. Opdat FuLOSE optimaal zou presteren moeten deze parameters (of toch een aantal ervan) afgestemd worden op de onderliggende toepassing. De resultaten bij het multiplexerprobleem tonen duidelijk aan dat de definitie van de vage klassen voor p_{lose} niet optimaal zijn. Er bestaan verschillende methoden en vuistregels om ook deze parameters, eigen aan het FuLOSE, op adaptieve wijze optimaal af te stellen (een voorbeeld gebruik makend van een genetisch algoritme wordt gegeven in (Dalci *et al.* 2004)). Helaas maakt dit het systeem alsmaar complexer.

Tevens moet er voldoende aandacht besteed worden aan de keuze van de gebruikte invoervariabelen (zowel het aantal als welke variabelen). Hier wensen we op te merken dat er niet meteen andere opgemeten parameters zijn die sterk verband houden met het fenomeen codegroei (hetzij door statische validatie, hetzij uit de literatuur). Toch is het mogelijk dat een weloverwogen keuze of combinatie van verschillende invoervariabelen de prestatie van FuLOSE in positieve zin kan beïnvloeden.

6.5.2 Sturing op basis van het aantal geschikte deelbomen

Naast het reduceren van de boomgrootte en het verhogen van de fitheid van kandidaat-oplossingen is gebruiksgemak een ander belangrijk aspect van LOSE. We wensen namelijk de gebruiker niet te overladen met tal van complexe

parameterinstellingen eigen aan deze methode. Tot nu toe valt dit best mee: we hebben aangetoond dat BSS steeds de beste resultaten geeft voor ongeacht welke benchmarktoepassing en de enige overblijvende parameter die de gebruiker moet instellen is p_{lose}. De bepaling van p_{lose} met behulp van een FuLOSE introduceert net nog meer extra parameters die manuele afstelling vereisen en pleegt een inbreuk op het gebruiksgemak. In deze paragraaf ontwikkelen we een alternatieve methode die de waarde van de probabiliteit waarmee LOSE wordt toegepast, automatisch en op adaptieve wijze instelt.

Op zoek naar geschikte invoerparameters

Vooreerst moeten we zoeken naar een zo beperkt mogelijke verzameling opgemeten parameters die ons een indicatie kunnen geven van hoe p_{lose} nu best gestuurd wordt. Hoe kleiner deze verzameling, hoe eenvoudiger de regelaar en hoe minder interacties tussen de verschillende parameters onderling. Wanneer we veel parameters in beschouwing nemen, stelt zich immers de vraag hoe deze parameters in een niet-geparametriseerd model te gieten?

Verder moeten we bij de sturing van de probabiliteit waarmee LOSE wordt toegepast ook rekening houden met wat de mogelijkheden zijn van de operator. Hiermee bedoelen we welke en hoeveel individuen over een geschikte deelboom beschikken. Zoals reeds aangetoond in voorgaande paragrafen kan een hoge instelwaarde voor p_{lose} in combinatie met een beperkte aanwezigheid van geschikte deelbomen leiden tot een sterk verlies aan structurele diversiteit met convergentie naar suboptimale compacte individuen tot gevolg. Met dit verlies aan structurele diversiteit werd geen rekening gehouden bij de vaste instelwaarde voor p_{lose} en evenmin bij de aanpak met FuLOSE!

In deze paragraaf kiezen we voor een totaal andere aanpak. In plaats van te zoeken naar goede verklarende parameters voor codegroei en een geschikte formule om deze parameters te combineren, richten we onze aandacht naar feedback die door de LOSE operator zelf wordt aangereikt. Er zijn verschillende parameters die hiervoor in aanmerking komen. Zo meten we het aantal individuen met een geschikte deelboom, de gemiddelde boomgrootte en boomdiepte van deze deelboom, de diepte waarop de deelboom te vinden is, de gemiddelde en maximum fitheid van de verzameling deelbomen, etc.

LOSE heeft als doel geschikte deelbomen te identificeren en deze deelbomen om te vormen tot volwaardige individuen. LOSE zelf tracht geen verklaring voor codegroei te geven maar geeft enkel een idee van hoeveel boomstructuren er op intelligente wijze verder geoptimaliseerd kunnen worden met een stijging van maximum fitheid en een daling van de boomgrootte als gevolg.

Het loont dus zeker de moeite om in het bijzonder naar het aantal geschikte deelbomen aanwezig in de populatie te kijken. Een sturing van p_{lose} op basis van deze parameter kan heel wat problemen verhinderen.

Eerst en vooral is het niet volledig correct om meer deelbomen dan effectief aanwezig in de populatie in te brengen zoals FuLOSE en een vaste p_{lose} wel doen. Hierdoor worden verschillende duplicaten in de populatie ingebracht waardoor de gemiddelde boomgrootte zal afnemen met structureel diversiteitsverlies tot gevolg. Maar de duplicaten hebben ook een nadelige invloed op de getoonde gemiddelde boomgrootte. Identieke compacte boomstructuren doen de gemiddelde boomgrootte sterk dalen waardoor men, valselijk, de indruk krijgt dat codegroei zeer sterk wordt gereduceerd.

Zoals we reeds hebben opgemerkt is het zeer moeilijk om een instelling te kiezen die voor alle deelboomselectiestrategieën en alle toepassingen past maar ook voor elke simulatie afzonderlijk. Sommige simulaties bevatten van nature (bij de initiële generatie) reeds minder structurele diversiteit dan andere waardoor de kans dat, onder druk van LOSE, die simulatie convergeert naar compacte suboptimale oplossingen groter is. Sturing op basis van het aantal beschikbare deelbomen kan een gepersonaliseerde aanpak bieden op maat van de toepassing, de deelboomselectiestrategie en de samenstelling van de populatie.

Wanneer we telkens het maximaal aantal geschikte deelbomen als richtlijn nemen voor de instelling van LOSE, behouden we, in de mate van het mogelijke, de structurele diversiteit binnen de populatie. Zowieso zal de diversiteit gedurende de evolutie automatisch afnemen (eveneens door het kleiner aantal knopen en de beperkte diepte). Samengevat, kiezen we voor het gebruik van het aantal geschikte deelbomen in de populatie voor de adaptieve aansturing van p_{lose} omwille van de volgende redenen:

1. Deze parameter hoeft niet herschaald te worden en bepaalt rechtstreeks het aantal nieuwe individuen in de populatie. Dit in tegenstelling tot bijvoorbeeld het gebruik van de diepte van de wortel van de deelboom waar een dergelijke herschaling (naar een percentage of het aantal individuen dat door LOSE wordt gegenereerd) wel noodzakelijk is.

2. Deze parameter wordt door het GP systeem zelf gegenereerd en vereist geen enkele instelling die door de gebruiker moet worden opgegeven. Zelfs een initiële waarde hoeft men niet te voorzien.

3. Deze parameter vormt zelf de bovengrens voor het aantal nieuwe individuen dat in de nieuwe populatie wordt ingebracht. Meer kopieën

inbrengen dan aanwezig is niet mogelijk en alzo wordt de structurele diversiteit binnen de populatie gedeeltelijk gevrijwaard.

4. Het gebruik van één enkele parameter biedt voordelen ten opzicht van meerdere parameters: de gebruiker hoeft zich immers geen zorgen te maken over de gebruikte logica om verschillende parameters te combineren tot één formule.

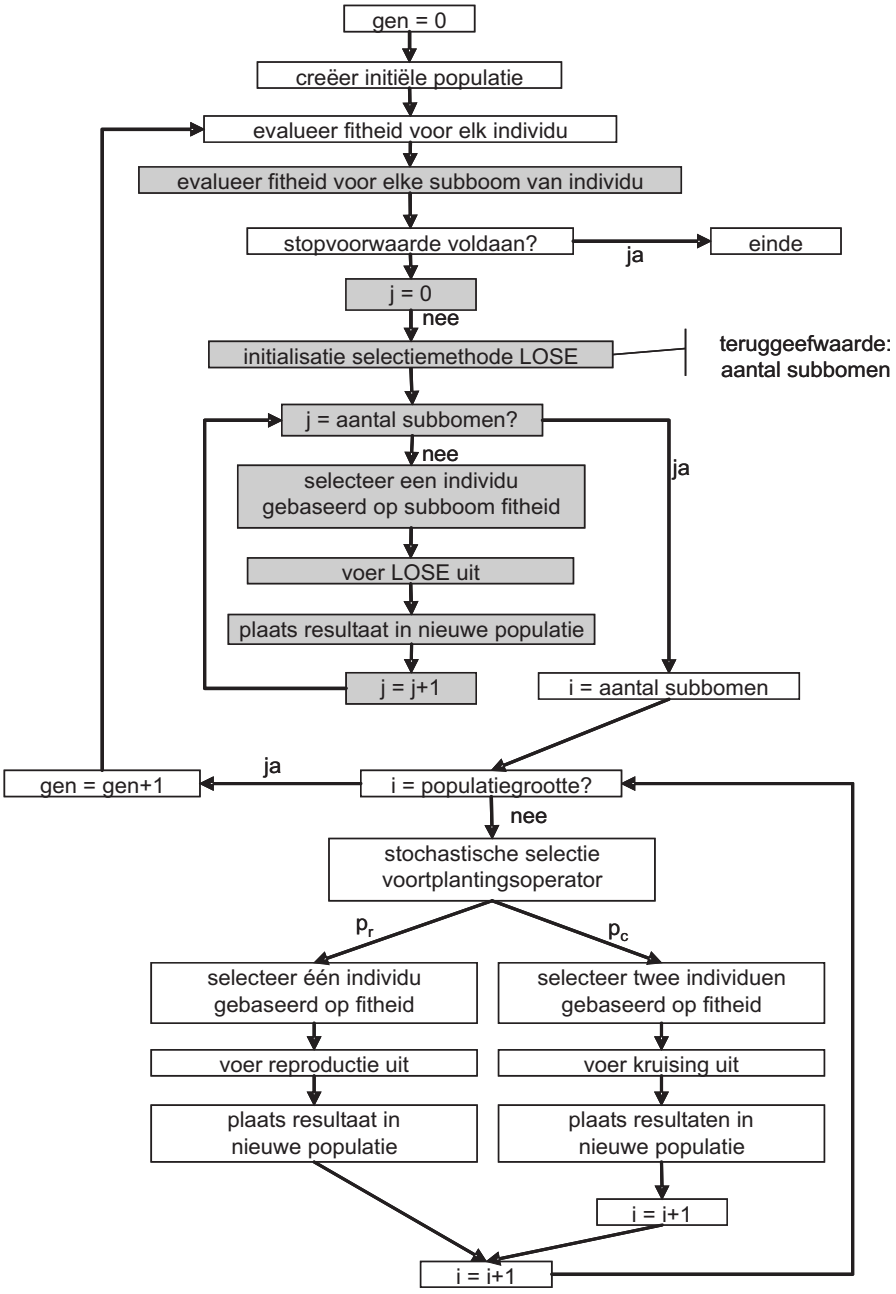
5. De parameter is onafhankelijk van de benchmarktoepassing (of eender welke andere toepassing die men wenst te optimaliseren) of van de gebruikte deelboomselectiestrategie. Bij moeilijkere problemen zullen er wellicht minder geschikte deelbomen zijn waardoor kruising (en mutatie) een groter aandeel krijgen in het exploreren van de zoekruimte (het verkennen van nieuwe gebieden). Bij eenvoudigere problemen zijn er wellicht meer geschikte deelbomen te vinden waardoor LOSE meer individuen in de populatie zal brengen en de gemiddelde boomgrootte zal beperken.

De aanpassingen die uitgevoerd worden aan de evolutionaire cyclus van genetisch programmeren zijn te zien in Figuur 6.20. We overlopen kort de werking van het adaptieve algoritme.

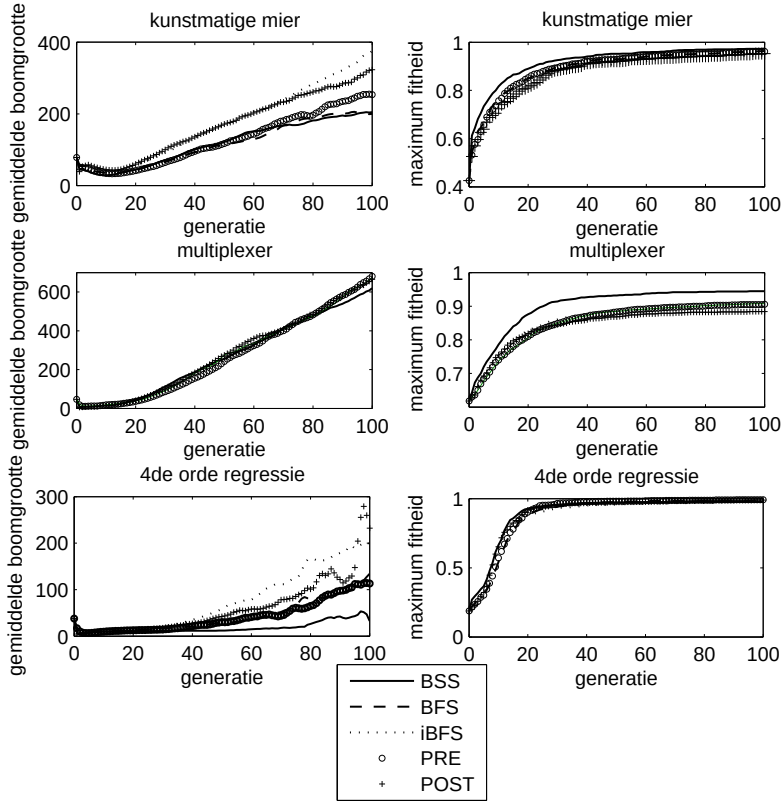
Tijdens de evaluatiefase wordt niet alleen de fitheid van de volledige boom maar ook de fitheid van elke deelboom bepaald (zie hoofdstuk 4). Vooraleer de voortplantingsfase wordt gestart, initialiseren we de selectiemethode die door LOSE zal worden gebruikt. Deze methode bouwt een lijst op met enkel de individuen die over een geschikte deelboom beschikken en verwijdert eventuele duplicaten (diversiteit). Het aantal individuen in deze lijst wordt geteld en als resultaat teruggegeven.

De eigenlijke voortplanting gebeurt in twee fazen. Een aantal ('aantal geschikte deelbomen') individuen wordt gecreëerd door LOSE (eerste fase). Iteratief worden individuen uit de selectielijst gehaald en omgevormd tot volwaardige kandidaat-oplossingen. Wanneer deze lijst ten einde is (en dus de teller j gelijk is aan het aantal geschikte deelbomen in de huidige populatie) stopt het eerste deel van de voortplantingsfase.

Indien er nog plaats is in de nieuwe populatie, worden er nog individuen toegevoegd met behulp van kruising of reproductie (tweede fase). Deze cyclus herhaalt zich tot wanneer het stopcriterium is vervuld.



Figuur 6.19: *De gewijzigde cyclus van genetisch programmeren bij gebruik van LOSE, adaptief gestuurd door het aantal deelbomen.*



Figuur 6.20: *Gemiddelde boomgrootte en maximum fitheid bij gebruik van het aantal geschikte deelbomen als sturing voor p_{lose} voor alle benchmarktoepassingen.*

Resultaten

Figuur 6.20 toont de resultaten wanneer het aantal geschikte deelbomen beschikbaar in de populatie wordt gebruikt om LOSE aan te sturen.

Bij de kunstmatige mier geeft de adaptieve aansturing van p_{lose} een significante meerwaarde bij de evolutie van maximum fitheid (behalve de methode iBFS).

Voor het vierde orde regressieprobleem is er nauwelijks of geen verschil in de evolutie van maximum fitheid. Gemiddeld over alle deelboomselectiemethoden behalen 98 van de 100 simulaties een correcte oplossing. Dit aantal is lichtjes hoger dan bij het gebruik van een vaste instelling maar is niet statistisch significant. De gemiddelde boomgrootte is hoger dan bij het gebruik

van een vaste p_{lose} maar dit is in hoofdzaak te wijten aan de simulaties die de oplossing trachten te benaderen. Van deze simulaties convergeert er geen enkele naar kleine suboptimale oplossingen zoals het geval was bij vaste p_{lose}. Dus indien er geen 100% correcte oplossing werd gevonden, zal het GP programma de oplossing trachten te benaderen door immense boomstructuren met soms enkele duizenden knopen.

Ook bij het multiplexerprobleem behalen alle deelboomselectiemethoden goede resultaten voor maximum fitheid. Toch is het enkel BSS die significant betere resultaten produceert. De gemiddelde boomgrootte bij beide toepassingen neemt wel toe. Dit is te wijten aan het gebruik van het aantal geschikte deelbomen in de adaptieve sturing. Deze parameter vormt een bovengrens voor p_{lose} in tegenstelling tot een vaste instelling voor p_{lose} waarbij veel kopieën van deelbomen werden ingebracht die de structurele diversiteit nadelig beïnvloeden. Hierdoor daalt de gemiddelde boomgrootte onmiddellijk door het inbrengen van een aantal kopieën maar ook op termijn door het onvermogen te ontsnappen wegens te weinig structurele variatie.

De toename in gemiddelde boomgrootte lijkt erger dan het in werkelijkheid is. Wanneer we het aantal knopen in het beste individu van elke generatie bekijken, merken we een lichte stijging die echter vaak niet significant verschillend is van de afmetingen van ditzelfde individu bij gebruik van een vaste p_{lose}.

6.6 Besluiten

Het actief inbrengen van compacte deelbomen met hoge fitheid heeft talrijke voordelen zoals de resultaten uit hoofdstuk 4 duidelijk weergeven. Een belangrijk probleem is echter de instelling van de probabiliteit waarmee LOSE wordt toegepast. Deze vereist enige voorkennis van het probleemdomein en wordt mogelijks nog meer gecompliceerd door de afhankelijkheid van de verschillende deelboomselectiestrategieën.

In hoofdstuk 4 gebruikten we de techniek parameter tuning om een 'aantal' instelwaarden uit te proberen en dan uiteindelijk te kiezen voor de waarde die het beste resultaat geeft. We merken op dat de stapgrootte bij het offline trachten te vinden van de optimale waarde opnieuw een extra parameter is die hoofdzakelijk wordt gekozen in functie van de beschikbare rekentijd. Immers, parameter tuning kost zelf ook heel wat rekentijd en zodra een geschikte waarde is gevonden moet men pas starten met de uitvoering van het programma gedurende een aantal onafhankelijke simulaties.

In dit hoofdstuk hebben we twee adaptieve modellen voorgesteld om de instelwaarde van p_{lose} automatisch te bepalen. Een geparametriseerde methode gebaseerd op vaaglogische verzamelingen bleek gelijkaardige resultaten op te leveren dan het gebruik van een vaste instelling voor p_{lose}. Dit model bleek bovendien complex en vereiste een aantal extra parameterinstellingen zoals het aantal vaagverzamelingen, de vage regels, etc. Om goede waarden voor deze parameters te kiezen, moet de applicatieprogrammeur nog steeds beschikken over voldoende kennis vanuit het probleemdomein. Bovendien bleef er een zekere afhankelijkheid ten aanzien van de deelboomselectiemethode bestaan.

De tweede univariate methode gebaseerd op het aantal geschikte deelbomen in de populatie, bleek bijzonder effectief. Maximum fitheid werd nogmaals significant hoger voor het multiplexerprobleem en de kunstmatige mier. Ook de gemiddelde boomgrootte nam toe, toch bleef de boomgrootte van het beste individu voor elke generatie quasi onveranderd. Deze methode biedt tal van voordelen zoals zijn eenvoud, betere resultaten, probleemonafhankelijk, en parametervrij.

Hoofdstuk 7

Robuustheid en dataselectie

Systemen gebaseerd op genetisch programmeren gebruiken totnogtoe veelal één enkele vaste leeromgeving om programma's te optimaliseren voor een bepaald, welomlijnd probleem. De geëvolueerde programma's vertonen dan ook vaak een optimaal (of bijna-optimaal) gedrag voor deze specifieke leeromgeving, maar falen jammerlijk wanneer een gelijkaardig probleem aangeboden wordt. Het doel van dit hoofdstuk is een algemeen toepasbaar raamwerk te ontwikkelen waarmee robuustheid kan gemeten worden. Vervolgens onderzoeken we wat nu de invloed is van de lokale optimalisatieoperator op de robuustheid van de bekomen kandidaat-oplossingen. Vaak wordt immers gesteld dat kleinere (compacte) oplossingen beter in staat zijn om met nieuwe situaties om te gaan (zogenaamde Ockham's scheermes).

7.1 De noodzaak voor robuuste programma's

Genetisch programmeren wordt meestal beschouwd als een optimalisatiemethode, een manier om één optimale oplossing voor een welbepaald probleem te verkrijgen. Het nadeel bij GP is dat, doorheen de evolutie, programma's de neiging hebben overgespecialiseerd te raken op het (meestal enige) voorbeeld dat tijdens de leerfase wordt aangeboden. Onder invloed van de toenemende selectiedruk zal een programma uiteindelijk het leervoorbeeld als het ware 'van buiten leren'. Wanneer men het programma vervolgens een gelijkaardig probleem voorschotelt, gebeurt het dikwijls dat de oplossing volkomen de mist ingaat, zelfs al zijn de afwijkingen tegenover het oorspronkelijke probleem minimaal. Men zegt dan dat het programma 'broos' of 'niet robuust' is. Het programma kan moeilijk toegepast worden op een soortgelijke opgave. Het creëert, uitgaande van het leervoorbeeld, geen hypothese die op een meer algemene klasse van leervoorbeelden van toepassing is. Dit veelvoorkomend gebrek aan robuustheid is een hinderpaal bij de creatie van kunstmatige intelligentie. Als GP meer wenst te worden dan 'slechts een optimalisatiemethode' is het belangrijk aandacht te schenken aan de robuustheid of toepasbaarheid op soortgelijke problemen.

Waarom is dit zo belangrijk? Robuustheid is één van de belangrijkste prestatiecriteria voor kunstmatige leersystemen. Een essentieel kenmerk van intelligentie is immers het vermogen een hypothese te kunnen afleiden uit een beperkt aantal gegeven voorbeelden. De kracht van leersystemen ligt in het feit dat ze impliciete verbanden in de invoer-uitvoer-mapping of tussen zichzelf en hun omgeving kunnen ontdekken. Het gebeurt vaak dat het onmogelijk of te omslachtig is om deze verbanden rechtstreeks aan het systeem aan te bieden. Het reduceren van de 'breekbaarheid' van een programma verbetert de kans dat het systeem succesvol blijft wanneer nieuwe feiten over het probleem bekend raken. Ook wanneer de omgeving veranderlijk is moet het systeem zo ontworpen worden dat het zich, binnen redelijke grenzen, kan aanpassen. Dit vermijdt de kost om het systeem telkens aan te passen of te herontwerpen.

Tot dusver is er weinig onderzoek gedaan naar robuustheid. In (Koza 1992) gaat men ervan uit dat genetisch programmeren als techniek vanzelf robuuste programma's oplevert, maar dit wordt niet gemeten. De geldigheid van deze bewering wordt zelfs niet nagegaan. Men leert en test nog steeds meestal op hetzelfde voorbeeld, terwijl een aparte leer- en testverzameling in andere takken van de kunstmatige intelligentie standaard zijn. Dit probleem is

wel onderkend, maar onderzoeken dienaangaande zijn schaars en gebeuren meestal met ad-hoc methoden.

In dit hoofdstuk ontwikkelen we eerst en vooral een algemeen toepasbaar raamwerk waarmee we de toepasbaarheid van de geëvolueerde oplossingen kunnen opmeten. Dit deel spitst zich in hoofdzaak toe op het vinden van een eenduidige definitie voor soortgelijke problemen. Vervolgens onderzoeken we, door middel van een aantal experimenten op twee benchmarkproblemen, het effect van lokale optimalisatie om de robuustheid te verbeteren en stellen we ons de vraag hoeveel verschillende leervoorbeelden minimaal nodig zijn om een aanvaardbare oplossing te bekomen.

7.2 Robuustheid en genetisch programmeren

7.2.1 De definitie van robuustheid

Om verwarring te vermijden, definiëren we eerst het begrip robuustheid zoals we het in dit proefschrift zullen gebruiken.

Robuustheid wordt gedefinieerd als de mate waarin een individu toepasbaar is op een ongeziene verzameling fitness cases[1]. Deze fitness cases vertonen een zekere verwantschap met de leervoorbeelden.

In dit hoofdstuk splitsen we de verzameling fitness cases op in twee afzonderlijke verzamelingen: de leervoorbeelden die worden gebruikt in het GP algoritme (door de selectiemechanismen, etc.) en de ongeziene testvoorbeelden (die een idee geven van hoe goed een kandidaat-oplossing in staat is soortgelijke problemen aan te pakken).

7.2.2 Het opmeten van robuustheid

Tot op heden bestaat er slechts een handvol methoden om de robuustheid van kandidaat-oplossingen op te meten. Vaak zijn deze methoden zeer probleemspecifiek maar toch kunnen we een aantal algemene strekkingen onderscheiden.

[1] In de literatuur worden verschillende andere benamingen gebruikt. Zo spreekt Kuschchu (2002) over evolutionaire ‘generalisatie’, Moore & Garcia (1997) over ‘breekbaarheid’ en Rosca (1996) over ‘generality’.

Ruis

Het toevoegen van ruis wordt vaak gebruikt om toevallige verbanden in een beperkte leerverzameling te vermijden. Deze verbanden worden niet opzettelijk expliciet toegevoegd aan de leerverzameling maar indien GP deze zal vinden en uitbuiten, kan de robuustheid van de geëvolueerde oplossingen nadelig beïnvloed worden. Deze ruis zal de neiging hebben toevallige verbanden te verstoren en zo de evolutie verhinderen hiervan gebruik te maken. Deze techniek wordt vaak toegepast bij sensor- en stuurwaarden bij robottoepassingen (Chongstitvatana 1998, 1999, Reynolds 1994).

Meerdere leervoorbeelden

Intuïtief kan men aanvoelen dat blootstelling aan bijvoorbeeld 20 verschillende leervoorbeelden tijdens de evolutie, beter generaliserende programma's zal opleveren dan wanneer telkens hetzelfde leervoorbeeld gebruikt wordt. De ruimte van alle mogelijke leervoorbeelden kan op verschillende manieren gesampled worden. Een voor de hand liggende aanpak is in het begin een vast aantal leervoorbeelden te nemen en deze aan het systeem te presenteren tot deze voorbeelden kunnen opgelost worden. Ook andere, dynamische manieren om voorbeelden te selecteren zijn mogelijk (Gathercole & Ross 1994).

Wisselen van leerverzameling

Een andere aanpak is het wisselen van leervoorbeelden gedurende de evolutie. Dit heeft als voordeel dat convergentie naar een voorafgedefinieerde leerverzameling verhinderd wordt. Intuïtief zou men denken dat deze techniek sterk lijkt op de voorgaande: wanneer na 5 generaties van leervoorbeeld wordt gewisseld, is het programma na 100 generaties ook op 20 leervoorbeelden geëvalueerd. Moore & Garcia (1997) suggereren dat de broosheid van geëvolueerde programma's gereduceerd kan worden door voor de evaluatie van elke generatie programma's, willekeurig nieuwe leervoorbeelden te genereren. Deze methode is natuurlijk zeer geschikt wanneer een vaste leerverzameling het volledige bereik van mogelijke situaties niet adequaat representeert.

Co-evolutie

De basisgedachte bij co-evolutie is een analogie te gebruiken van de 'wapenwedloop' zoals die zich in de natuur tussen prooi- en roofdieren soms voordoet. Bij co-evolutionaire methoden evolueert de invoer tegelijkertijd met de kandidaat-oplossingen: de fitheid van een bepaalde invoer is gebaseerd op de

hoeveelheid kandidaat-oplossingen die niet correct met deze invoer kunnen omgaan. Bij co-evolutie hebben we dus twee populaties die naast elkaar bestaan: een populatie van oplossingen en een populatie van problemen. Een voorbeeld maakt dit duidelijker: stel dat we een programma, een kunstmatige mier, zoeken dat een spoor van voedselkorreltjes volgt. De ene populatie bestaat dan uit mieren die zo goed mogelijk een spoor proberen te volgen. De andere populatie is er een van sporenconstructoren die zoveel mogelijk mieren proberen te misleiden (Ronge & Nordahl 1996, Browne 1996, Angeline & Pollack 1993*a*).

7.2.3 Enkele algemene bedenkingen bij methoden om robuustheid op te meten

De ideale leerverzameling bestaat uit alle mogelijke situaties die het programma onder ogen zou kunnen krijgen. Helaas is dit wegens de hoge kost niet wenselijk en heel dikwijls zelfs niet mogelijk. We moeten ons dus behelpen met methoden uit voorgaande paragraaf (of een combinatie ervan). Elk van deze methoden heeft echter voor- en nadelen[2].

De ruis-methode is minder geschikt bij benchmarkproblemen met discrete fitheidswaarden. Daar zorgt het toevoegen van ruis direct voor een bruuske verandering in het probleem. We verwachten dat deze techniek meer verdiensten heeft bij continue problemen, waar bijvoorbeeld met afstanden of hoeken gewerkt wordt, zodat toevoegen van ruis voor slechts kleine veranderingen zorgt.

Het aanbieden van meerdere leervoorbeelden is zonder twijfel de meest efficiënte manier om de robuustheid op te drijven. Een programma dat tijdens de leerfase met vele situaties in aanraking komt, zal naderhand ook beter in staat zijn ongeziene problemen het hoofd te bieden. Deze methode is zeer geschikt wanneer men voldoende rekenkracht en geheugen ter beschikking heeft. Belangrijk is echter wel te onderzoeken hoeveel leervoorbeelden noodzakelijk zijn. Een te kleine leerverzameling zorgt voor onvolledige en broze oplossingen, een te grote leerverzameling is een verspilling van tijd en geheugenruimte. Bovendien zijn er grenzen aan het aantal leervoorbeelden dat men kan toevoegen aan de leerverzameling opdat de robuustheid zou stijgen. Hier geldt de ‘wet van dalende meeropbrengst’.

[2]Deze voor- en nadelen werden uitvoerig onderzocht in een Master thesis onder begeleiding van de auteur van dit proefschrift (De Smedt 2006). We vermelden hier de voornaamste bevindingen.

Het telkens weer veranderen van leerverzameling na één of enkele generaties tijdens de evolutie is een bijzonder geschikte techniek wanneer men, door gebrek aan rekenkracht of geheugen, slechts op een kleine leerverzameling kan evalueren. Een vaste leerverzameling kan het volledige bereik van mogelijke situaties in dat geval niet adequaat genoeg representeren. Een steeds veranderende leerverzameling geeft significant betere resultaten dan slechts één of enkele leervoorbeelden te gebruiken. Wanneer men echter de leerverzameling groter maakt, vermindert de toegevoegde waarde van deze methode. Een constant veranderende leerverzameling kan echter elke zoekgradiënt naar een hogere fitheid wegnemen. Men moet dus afwegen in welke mate de leerverzameling verandert. Daarbij loont het misschien de moeite om moeilijkere leervoorbeelden vaker aan te bieden.

Ook aan co-evolutie zijn belangrijke nadelen verbonden. Ten eerste moeten er ook voor leervoorbeelden genetische operatoren gedefinieerd worden. Het is niet altijd evident om het probleem zelf als een boom voor te stellen, of een gelijkaardig 'organisme' waarmee eenvoudig te werken valt. Dit is dus geen algemeen toepasbare methode. Een tweede belangrijk nadeel is dat er dubbel zoveel werk moet verricht worden, terwijl de helft van dit werk, met name het evolueren en evalueren van de leervoorbeelden, op zich nutteloos is. We zoeken immers oplossingen, geen problemen.

7.2.4 Robuustheid en codegroei

Bepaalde auteurs zoals (Rosca 1996), (Kinnear 1993*b*) en (Banzhaf *et al.* 1997) hebben gesuggereerd dat kleine programma's beter toepasbaar zijn, of dat er minstens een verband bestaat tussen programmagrootte en de robuustheid. De rationale hierachter is dat veel *junk*-code in grote programma's delen van het probleem voorstellen die 'van buiten geleerd' zijn en dus de inzetbaarheid op soortgelijke problemen hinderen.

Een veelgehoord argument hierbij is dat van Ockhams scheermes: "Entia non sunt multiplicanda praeter necessitatem"[3], waarbij het begrip 'entiteiten' meestal staat voor 'gepostuleerde objecten binnen een hypothese'. De hypothese is dan de oplossing van het gestelde probleem, dus het programma; de gepostuleerde objecten zijn de instructies van dit programma, bij GP de knopen van de boom. Ruwweg komt dit neer op het stellen dat een kleiner programma een betere aanpak van zowel het gestelde probleem als gelijkaardige problemen zal vertonen dan een groter.

[3] "Men moet entiteiten niet zonder noodzaak verveelvoudigen"

Rosca (1996) geeft echter zelf toe dat meer experimenten nodig zijn om dit verband te bewijzen. Ook Mahler *et al.* (2005) bestudeerden de invloed *Tarpeian bloat control* op de robuustheid van de bekomen oplossingen. Hun resultaten waren op zijn zachtst uitgedrukt zeer uiteenlopend. De code werd zeer sterk gereduceerd, toch steeg de prestatie op de testverzameling nauwelijks en in sommige problemen zelfs helemaal niet.

Ockhams scheermes mag dan wel geschikt zijn als praktisch principe bij het onderscheiden van hypothesen, de natuur geeft ons een ander beeld. Zo vinden we amoeben die 200 keer zoveel genetisch materiaal hebben als mensen, zonder zich daarom beter te kunnen aanpassen aan veranderende omstandigheden. De vraag blijft dus of we via rechtstreekse manipulatie van het genotype bepaalde fenotypische eigenschappen, in casu de robuustheid, kunnen uitlokken.

In dit hoofdstuk zijn we bijzonder geïnteresseerd in de relatie tussen LOSE en de robuustheid van de kandidaat-oplossingen. We hopen een antwoord te kunnen geven op de vraag of kleinere boomstructuren gemiddeld genomen betere prestaties afleveren op soortgelijke problemen zoals vaak wordt beweerd maar nooit ten gronde werd onderzocht. We kunnen deze doelstelling uitsplitsen in twee afzonderlijke delen.

Enerzijds stellen we ons de vraag of compacte individuen beter in staat zijn soortgelijke problemen aan te pakken (of het gebruik van LOSE dus meer robuuste individuen creëert dan standaard GP). De precieze betekenis van soortgelijke problemen wordt in volgende paragraaf verduidelijkt. Een typisch voorbeeld hiervan is de kunstmatige mier. Het is immers handig meegenomen indien de kunstmatige mier ook andere sporen, behalve Sante-Fe, foutloos zou kunnen bewandelen. Denk bijvoorbeeld ook aan een zoekrobot die steeds geleerd werd op een vlakke ondergrond. Wanneer deze robot zich op ruwer terrein zou begeven (rotsblokken, putten, etc.), is de kans groot dat hij zich vastrijdt.

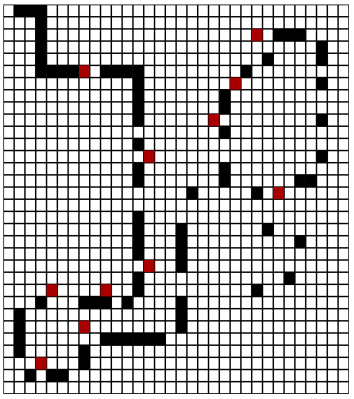
Anderzijds, in problemen waarbij er van nature reeds zeer veel leervoorbeelden aanwezig zijn, kunnen we ons de vraag stellen of het gebruiken van een beperkte deelverzameling ook niet volstaat om eenzelfde niveau van fitheid te bereiken? En hebben we minder voorbeelden nodig wanneer LOSE wordt toegepast of niet? Een gepast voorbeeld hiervan is de 11-bit multiplexer. Zijn alle 2048 mogelijke invoercombinaties noodzakelijk om een goed resultaat te bekomen of is de helft reeds voldoende? Zo ja, hoe moeten we deze dan selecteren om een optimaal resultaat te bekomen?

7.3 De creatie van nieuwe omgevingen

Een eerste stap om de robuustheid te kunnen opmeten is uiteraard de implementatie van een raamwerk waarbinnen leer- en testfitheid worden opgemeten. Dit behelst onder meer de creatie soortgelijke leer- en testvoorbeelden. Wat dit precies is, wordt voor elke toepassing afzonderlijk uitgelegd.

7.3.1 De kunstmatige mier

De meeste van de technieken die we willen onderzoeken gaan ervan uit dat er meerdere leervoorbeelden beschikbaar zijn. Bij de kunstmatige mier leert en test men echter traditioneel op één en hetzelfde voorbeeld, het Santa Fe spoor of het Los Altos spoor. We stellen hier twee methoden voor om alternatieven voor het Sante-Fe spoor te creëren.



Figuur 7.1: *Spoorgeneratie door ruis toe te voegen aan Santa Fe. De ten opzichte van het Santa Fe spoor verplaatste voedselpakketjes zijn in rood weergegeven.*

Ruis

We kunnen om nieuwe leervoorbeelden te creëren uitgaan van het Santa Fe spoor en hieraan ruis toevoegen. Een bepaald percentage van de voedselkorreltjes wordt verplaatst over een bepaalde afstand. Op het eerste zicht lijken deze sporen bijzonder goed op het Santa Fe spoor (zie Fig. 7.1).

Wanneer het verplaatste voedselkorreltje bovenop een bestaand korreltje terecht zou komen, wordt een nieuw, leeg vakje gezocht. Als parameterwaarden gebruiken we

- 10% kans om een voedselkorreltje te verplaatsen;
- over een afstand (horizontaal, verticaal of diagonaal) van 1 positie.

Bouwblokken

Een andere manier om sporen te vormen is niet uit te gaan van macroscopische kenmerken van Santa Fe, maar op te bouwen uit elementaire onderdelen. Kushchu (2002) beschrijft hoe uit het Santa Fe spoor elementaire bouwblokken kunnen gehaald worden. Deze bouwblokken zijn de 'eigenaardigheden' in het Santa Fe spoor, zoals een onderbreking in het spoor, een bocht of een combinatie van beide. Na een volledige analyse van Santa Fe komt men tot een tabel van bouwblokken, gerangschikt volgens de richting waarin ze optreden (zie Tabel 7.1).

Tabel 7.1: *Elementaire bouwblokken van het Santa Fe spoor.*

Richting	Rechte lijn		Bocht			
	1 open	2 open	1 open	2 open	3 open	0 open
←						
↑						
→						
↓						

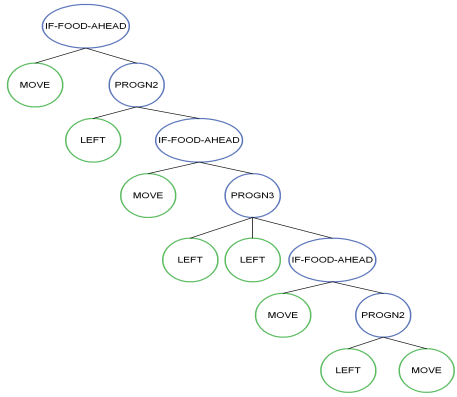
Kushchu (2002) rapporteert zeer goede resultaten wanneer tijdens de leerfase voorbeelden gebruikt worden die opgebouwd zijn uit enkel deze bouwblokken, afgewisseld met een willekeurig aantal voedselpakketjes. Wanneer andere blokken geïntroduceerd worden tijdens de testfase, hebben de geëvolueerde programma's echter nog steeds de neiging te falen. Men besluit dat, gegeven een 'klasse van sporen' en een leerproces dat sporen uit deze klasse gebruikt, het mogelijk is robuust gedrag te bekomen voor ongeziene sporen binnen die klasse.

Hoewel deze methode op het eerste zicht valabel lijkt, zijn er toch een aantal nadelen aan verbonden. Ten eerste moet men het oorspronkelijke Santa Fe spoor volledig analyseren en blijft men toch beperkt tot de uitzonderingsgevallen die hierbij optreden. Het lijkt erop dat een werkend programma eerder deze uitzonderingsgevallen van buiten leert dan dat het werkelijk een algemeen spoorvolgend gedrag vertoont. Ook lijkt de definitie van een klasse van sporen gebaseerd op enkel deze bouwblokken arbitrair en beperkend. We erkennen dat er bepaalde klassen van sporen zijn, maar de onderverdeling in klassen is ruimer dan het opsommen van de samenstellende bouwblokken. We moeten twee zaken voor ogen houden:

- we zoeken een mier die in staat is een spoor te volgen;
- de mier wordt zwaar beperkt door zijn eenvoudige functie- en terminalensets.

Het feit dat de mier een sensor heeft die slechts één positie vooruit kan kijken is de belangrijkste beperking. Er zijn diverse manieren voorgesteld om deze functie te verbeteren, zoals 360° zicht of een soort reukzin die voedselpakketjes vanop grotere afstand kan detecteren. Het veranderen van de functieset verandert echter dit benchmarkprobleem zelf: de bedoeling is juist, middels de zeer eenvoudige functieset, een spoorvolgend gedrag te ontwikkelen. Wanneer we de functieset zouden veranderen, wordt vergelijken met ander onderzoek omtrent de kunstmatige mier onmogelijk.

Een mogelijk programma dat het Santa Fe spoor correct volgt is weergegeven in Figuur 7.2. De mier blijft rechtdoor lopen tenzij ze links of rechts voedsel ziet liggen. In principe is dit een vrij goede definitie van 'spoorvolgen': wanneer verdere voedselkorrels even niet zichtbaar zijn, blijven rechtdoor lopen tot er weer voedsel in zicht is. Het terug oppikken van het spoor gebeurt door rond te kijken, met als gevolg dat het lastig is een spoor terug op te pikken dat herbegint op plaatsen die verder liggen dan het oog — of in het geval van de mier, de functie IF-FOOD-AHEAD — reikt. Een andere mogelijkheid is wanneer men het spoor bijster is in steeds wijdere concentrische cirkels rond te



Figuur 7.2: *Deze mier kan Santa Fe correct oplossen in 408 stappen. Wanneer ze geen voedsel voor zich ziet liggen, kijkt ze ofwel naar links ofwel naar rechts of er voedsel ligt. Indien dit het geval is, loopt ze vooruit. Indien dit niet het geval is loopt ze in de oorspronkelijke richting verder.*

lopen. Dit vergt echter veel meer tijdseenheden en zal dan ook bij een spoor met een vrij hoog percentage open plekken onvermijdelijk afgestraft worden.

Dit programma vertoont een spoorvolgend gedrag voor gelijk welk spoor dat opgebouwd is uit de bouwblokken van (Kuschchu 2002), zolang het zichzelf niet kruist. Dit ligt echter 'niet' aan de bouwblokken zelf, maar aan de manier waarop deze opgebouwd zijn. We merken op dat bij elke afwijking het spoor verdergaat in dezelfde richting, ofwel op een plaats die één positie afwijkt van de rechte lijn.

We besluiten dan ook dat de in Tabel 7.1 opgesomde bouwblokkken slechts een kleine arbitraire deelverzameling zijn van alle blokken die kunnen beschreven worden door de formule: "een aantal lege plaatsen, gevolgd door eventueel een bocht, gevolgd door een voedselpakketje", of in EBNF-notatie[4]:

$$bouwblok \quad ::= \quad (\square) * \quad (L|R)? \quad \blacksquare \tag{7.1}$$

met
- $\blacksquare$ een voedselkorreltje
- $\square$ een open plek in het spoor

Deze formule geeft ons niet alleen de aanvulling en uitbreiding van Tabel 7.1, maar definieert meteen ook de gehele klasse van sporen die door een robuust

[4]EBNF, voluit geschreven *Extended Backus-Naur Form*, voegt een syntax voor reguliere uitdrukkingen toe aan de grammatica voor context vrije talen.

programma moeten kunnen bewandeld worden, uitgaande van de functieset van dit benchmarkprobleem. We zullen dan ook snel ondervinden dat deze klasse van sporen de enige is die robuustheid kan bewerkstelligen. De toegemeten tijd wordt voor elk voorbeeld dynamisch bepaald en is onder andere afhankelijk van de grootte van de wereld en het aantal bochten in het spoor. Als parameterwaarden gebruiken we:

- 25% kans op een bouwblok, anders een voedselkorreltje leggen;
- 20×20 werelden als leervoorbeeld;
- 4 bouwblokken in een leervoorbeeld;
- 32×32 werelden als testvoorbeeld;
- 7 bouwblokken in een testvoorbeeld.

Verschil tussen de ruis- en de bouwblokken methode

Bekijken we aandachtig Figuur 7.1 dan merken we op dat de ruis-methode niet noodzakelijk sporen aanmaakt die voldoen aan de bovenstaande formule. Wanneer we even vooruitblikken op de resultaten zoals weergegeven in paragraaf 7.4, dan kunnen we dit experimenteel vaststellen. Willekeurige voedselpakketjes verwijderen, verplaatsen of toevoegen leidt soms tot situaties waar het geëvolueerde programma niet mee om kan gaan. De mier bevindt zich dan in een gebied waar er geen voedselpakketje in één van de aangrenzende gebieden is gepositioneerd. De kunstmatige mier heeft dus geen enkel aanknopingspunt in verband met het te volgen spoor en zal wellicht verdwalen of in concentrische cirkels bewegen tot wanneer een nieuw voedselpakket wordt gevonden. Net zoals Kuschchu (2002) kunnen we stellen dat ruis sporen creëert die niet altijd tot eenzelfde klasse van oplosbare sporen behoren. Vandaar dat de fitheid (gemiddeld genomen) steeds lager zal zijn in vergelijking met de bouwblokken methode.

7.3.2 De multiplexer

Vermits alle leervoorbeelden op voorhand gekend zijn, is hier de hoeveelheid leervoorbeelden en de manier van selecteren belangrijk.

Ruis

De multiplexer is een probleem waarbij ruis toevoegen aan de leervoorbeelden bijzonder voor de hand ligt. De leervoorbeelden bestaan uit een sequentie van 11 bits. Wanneer men bits verandert, bekomt men gewoon een ander leervoorbeeld van de 2048. Een mogelijkheid is in het datawoord bits te veranderen. Het veranderde datawoord heeft dan een bepaalde Hamming-afstand tot het originele datawoord. Wanneer het uit te voeren bit niet veranderd wordt door de ruis, zal een goede multiplexer zich dan ook niet van de wijs laten brengen door deze extra ruis en nog steeds dezelfde uitvoer als daarvoor geven. Aangezien deze methode datawoorden converteert naar andere (bestaande) datawoorden, wordt deze methode niet verder in beschouwing genomen.

Willekeurige getallen

Wanneer we de probleemruimte samplen kunnen we volledig willekeurig te werk gaan. Als we N leervoorbeelden nodig hebben genereren we N willekeurige getallen tussen 0 en 2047 zoeken, waarmee telkens een leervoorbeeld geassocieerd is.

Als we wat meer controle willen kunnen we het interval $[0, 2047]$ onderverdelen in N gelijke deelintervallen (op 1 getal nauwkeurig) en uit elk deelinterval een willekeurig getal kiezen. Zo verzekeren we ons ervan dat er geen dubbele leervoorbeelden zullen optreden. De gebruikte implementatie wordt hierna weergegeven:

```
testcases * aanmaken_testvoorbeelden (
 testcases * testcase, int aantal) {

 int x, y = 0,e = 0;
 int remember = -1;
 int randomgetal;

 for (x = 0; x < 2048; x++)
 {
  if( (e + aantal) < 2048 )
  {
   e += aantal;
  }
  else
  {
```

```
    randomgetal = (rand()%(x-remember))+(remember+1);
    testcase[y].a = randomgetal & 7;
    testcase[y].d = randomgetal >> 3;

    remember = x;
    y++;
    e += (aantal-2048);
   }
  }
 return testcase;
}
```

7.3.3 Aanpassingen aan de software

Om de experimenten te kunnen uitvoeren was het nodig lil-gp aan te passen. Eerst moest het mogelijk gemaakt worden dat elk individu meerdere fitheidswaarden kon opslaan, met name één waarde voor elk van de voorbeelden die aangeboden worden tijdens de leerfase, de validatie[5] en het testen. De fitheidswaarden worden opgeslagen in arrays waarvan de grootte bij het starten van de simulator dynamisch wordt bepaald uit het parameterbestand. Verder wordt ook de gemiddelde fitheid over de hele leer-, validatie- en testverzameling opgeslagen.

Ook was het nodig een aantal functies te schrijven die instaan voor de creatie van de leer-, validatie- en testvoorbeelden. Voor de mier kunnen deze voorbeelden ingelezen worden uit een bestand, kan er aan bestaande voorbeelden ruis toegevoegd worden waarvan de hoeveelheid en de amplitude kan geregeld worden, kan er ruis aan de begintoestand van de mier worden toegevoegd, zowel voor oriëntatie als voor positie. Er kunnen tevens sporen gegenereerd worden uit bouwblokken die uit een apart bestand worden ingelezen. Voor de multiplexer kunnen de voorbeelden ingelezen worden uit een bestand, kan er aan de datalijnen van de voorbeelden ruis toegevoegd worden tot een bepaalde Hamming-afstand, kunnen er volledig willekeurige voorbeelden gegenereerd worden of kan het interval [0, 2047] opgesplitst worden in gelijke delen waaruit een willekeurig voorbeeld genomen wordt, kan het complement genomen worden van de huidige voorbeeldverzameling of kan de volledige verzameling van alle 2048 voorbeelden genomen worden.

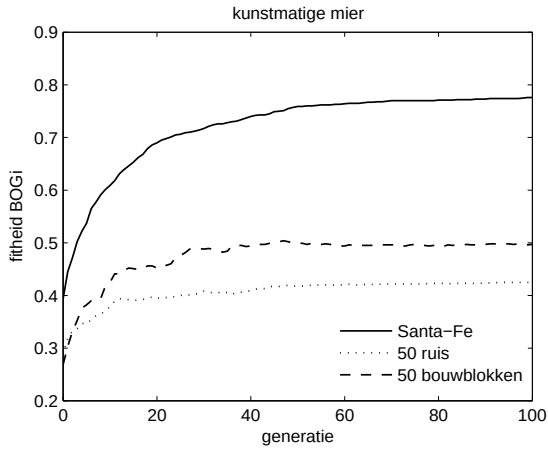
Het is bovendien mogelijk de leerverzameling te veranderen na elk in het parameterbestand opgegeven aantal generaties. Hiervoor zijn dezelfde me-

[5] Deze fase wordt nog niet gebruikt in dit proefschrift.

thoden als tijdens de initialisatie van toepassing. Ook de werking van de LOSE-operator werd aangepast zodat deze de fitheid van de deelboom berekent voor de volledige 'leer'verzameling.

7.4 Resultaten bij standaard GP

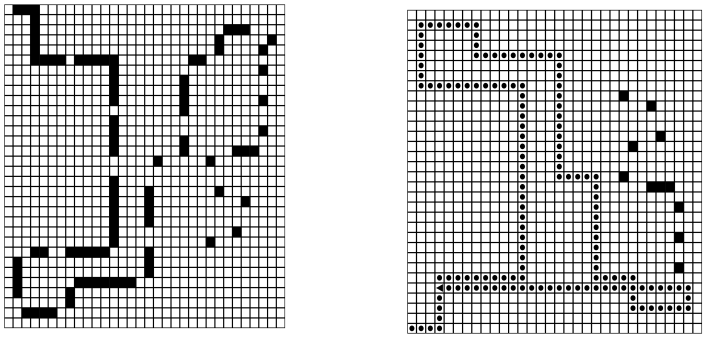
Vooraleer we de invloed van LOSE op de robuustheid van de geëvolueerde programma's onderzoeken, bekijken we wat de kwaliteit is van computerprogramma's geëvolueerd met standaard GP wanneer er enkel geleerd wordt op het Santa Fe spoor (Fig. 7.3). We gebruiken dezelfde experimentele proefopzet zoals gedefinieerd in hoofdstuk 4. De leerfitheid bekomen op het Santa Fe spoor wordt getoond samen met de gemiddelde testfitheid die bekomen wordt op 2×50 sporen die gecreëerd zijn met de methoden beschreven in 7.3 (de getoonde fitheidswaarden op de andere types sporen zijn dus gemiddelden over 50 instanties van die klasse). Enkel de prestatie van het beste individu op de huidige generatie (BOGi) wordt weergegeven.



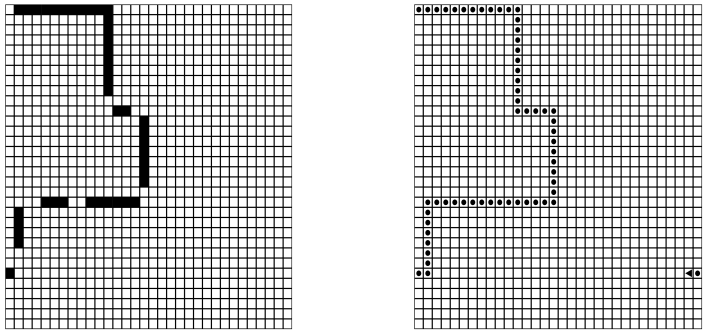
Figuur 7.3: *Wat is de prestatie van een programma op verschillende types van sporen wanneer het enkel op het Santa Fe spoor geleerd is (standaard GP)?*

We zien duidelijk dat een programma dat enkel het originele Santa Fe spoor onder ogen gekregen heeft, waar het na 100 generaties gemiddeld 77% van het voedsel kan vinden, ook min of meer de andere types sporen kan volgen, zij het met veel minder succes. De toepasbaarheid blijft met andere woorden beperkt. Opvallend is wel dat sporen opgebouwd uit bouwblokken een beter

resultaat geven. Daarna volgen sporen die ontstaan door ruis toe te voegen aan Santa Fe.



Figuur 7.4: *Een mier, die perfect scoort op Santa Fe, bewandelt een Santa Fe spoor met ruis.*



Figuur 7.5: *Een mier, die perfect scoort op Santa Fe, bewandelt ook dit spoor opgebouwd uit 7 bouwblokken perfect.*

Om beter te begrijpen wat er gebeurt bekijken we de Figuren 7.4 en 7.5, waarin een mier wordt getoond die een perfecte score haalt op het Santa Fe spoor. Dergelijke individuen komen helaas zelden voor bij standaard GP. Voor de sporen opgebouwd uit bouwblokken of ruis namen we als voorbeelden die sporen die nog het best bewandeld werden.

We zien in overeenkomst met grafiek 7.3 dat de mier vrij goed overweg kan met sporen die uit bouwblokken bestaan (Fig. 7.5). We merken op dat de mier na de leerfase op het Santa Fe spoor inderdaad geleerd heeft in dezelfde

richting te blijven doorlopen als ze geen voedsel ziet, tenzij ze direct links of direct rechts voedsel ziet liggen. Van een 'in-rondjes-zoeken'-gedrag of een andere zoekmethode is geen sprake.

Waarbij ruis aan het originele Santa Fe spoor werd toegevoegd (Fig. 7.4, rechts), volgt de mier dezelfde strategie tot wanneer er geen voedsel te vinden is in de onmiddellijke omgeving. Daar raakt de mier het spoor bijster en loopt gewoon rechtdoor.

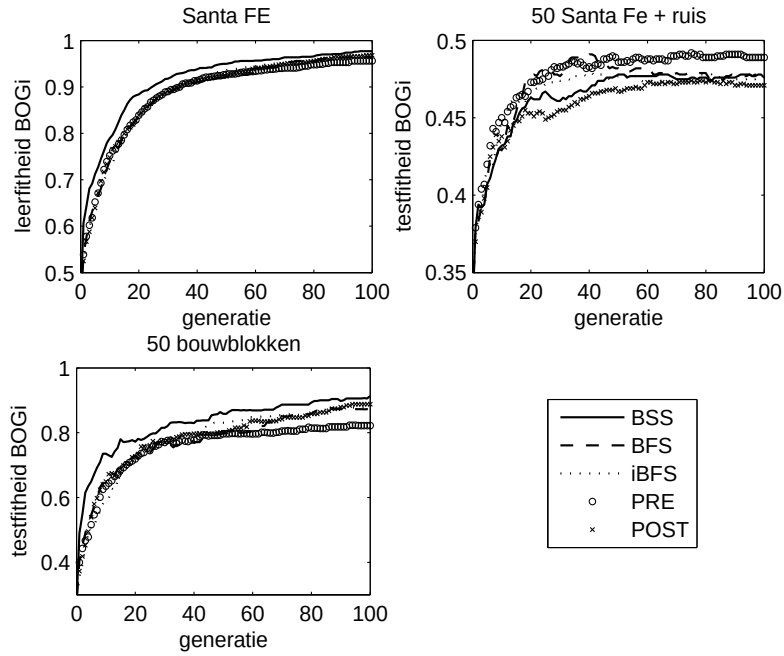
Bij het multiplexerprobleem wordt er standaard geleerd met behulp van alle 2048 leervoorbeelden. De resultaten zijn te zien in Figuur 4.5 (hoofdstuk 4) en worden hier niet nodeloos herhaald. De centrale vraag bij dit benchmarkprobleem is of met het gebruik van minder leervoorbeelden het GP programma in staat is om kwalitatief even goede oplossingen te evolueren en tevens of het gebruik van LOSE dit aantal nog verder kan reduceren.

7.5 LOSE en de robuustheid bij de mier

Bij de kunstmatige mier bekijken we eerst wat de invloed van codegroeiremmende methoden is bij GP, wanneer we leren op enkel het Santa Fe spoor. We hernemen de experimenten uit 7.4, met als enige verschil dat we nu LOSE toepassen. De probabiliteit waarmee LOSE wordt toegepast wordt automatisch bepaald aan de hand van het aantal beschikbare deelbomen (zie hoofdstuk 6). De vergelijking tussen de experimenten zonder LOSE en deze met LOSE is weergegeven in figuren 7.3 (standaard GP) en 7.6.

Het blijkt dat LOSE, ongeacht welke deelboomselectiestrategie wordt gebruikt, steeds een positieve invloed heeft op de robuustheid van de geëvolueerde programma's, ook al zijn de aangeboden testvoorbeelden niet gemakkelijk zoals bij de sporen die met ruis zijn gegenereerd. Het grootste voordeel wordt echter wel behaald bij sporen die opgesteld zijn volgens de formule (7.1). De testfitheid op basis van de bouwblokkenmethoden stijgt hierbij gemiddeld meer dan 76%, zelfs tot boven de gemiddelde maximale leerfitheid behaald in de experimenten zonder LOSE. Alle verschillen zijn significant, zoals blijkt uit de Wilcoxon p-waarden voor de bouwblokkenmethode (0) en ruis (0,0106)[6].

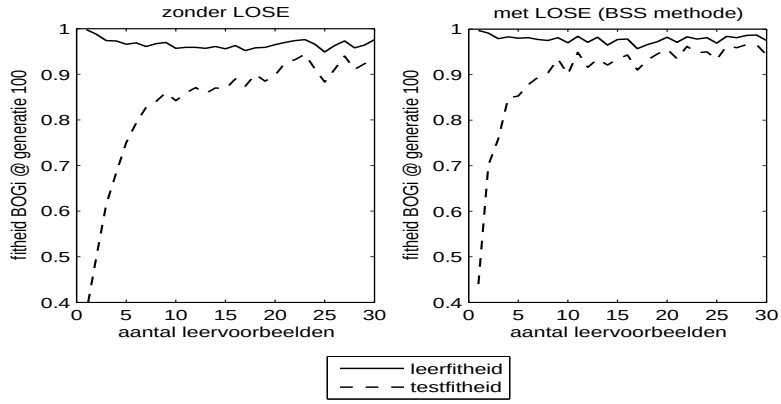
[6]Ditzelfde experiment werd tevens herhaald met behulp van een vaste instelling voor p_{lose}. Ook hieruit bleek dat LOSE een positieve invloed heeft op de robuustheid van de bekomen oplossingen. De bekomen testfitheidswaarden liggen echter lager dan wanneer een automatische instelling voor p_{lose} wordt gehanteerd.



Figuur 7.6: *De invloed van LOSE op de robuustheid wanneer enkel het Santa Fe spoor wordt gebruikt als leervoorbeeld.*

Indien we de resultaten van de verschillende deelboomselectiemethoden van naderbij bekijken (enkel getest op de bouwblokken methode) dan merken we dat BSS ook hier zorgt voor de beste resultaten. Tevens evolueert BSS ook de kleinste kandidaat-oplossingen. Het omgekeerde is geldig voor PRE. Deze methode produceert de grootste individuen, en de testfitheid is hier het kleinst. We stellen dus experimenteel vast dat compacte oplossingen beter in staat zijn om ongeziene testsporen te bewandelen.

Wanneer we nu meerdere leervoorbeelden gebruiken (i.p.v. enkel het Santa Fe spoor), nu met LOSE, krijgen we gelijklopende resultaten (Fig. 7.7): programma's bereiken betere prestaties op voorbeelden die tijdens de leerfase niet gezien zijn. Zowel leer- als testvoorbeelden zijn gegenereerd met de bouwblokkenmethoden (aantal testvoorbeelden = 50). Wanneer we de Wilcoxon-toets uitvoeren voor 30 leervoorbeelden met en zonder LOSE, zijn de verschillen voor alle testfitheidswaarden significant (p-waarde 0). Het gebruik van LOSE impliceert dus nogmaals een meerwaarde ook wanneer meerdere leervoorbeelden worden gebruikt tijdens de leerfase! Hier laat het



Figuur 7.7: *De invloed van LOSE op de robuustheid wanneer meerdere leervoorbeelden aangeboden worden.*

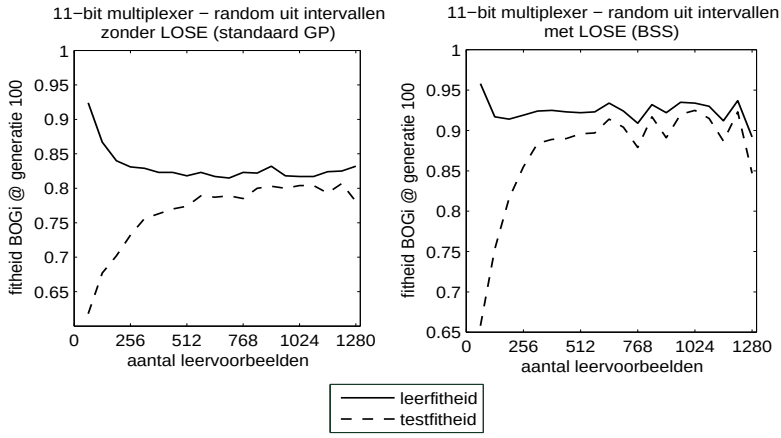
gebruik van LOSE echter nog een bijkomend voordeel zien: goede resultaten worden bekomen met een veel kleiner aantal leervoorbeelden (let bijvoorbeeld op het bereiken van de 90%-grens door de testfitheden). De rechtse curve stijgt veel sneller dan de linkse. Dit brengt een reductie van de evaluatietijd nodig voor de bepaling van de fitheid met zich mee.

Wanneer we de prestaties van de beste individuen op de sporen gegenereerd met behulp van ruis nagaan (figuur niet weergegeven), dan worden er helaas maar weinig voedselpakketjes gevonden. Ook al is de fitheid van deze individuen beter wanneer LOSE wordt gebruikt, toch blijft deze waarde bedroevend laag (onder 0.5). De individuen slagen er beter in om de stukken uit het spoor die voldoen aan de formule 7.1 te volgen, maar raken vaak het noorden kwijt precies op die plaatsen waar het spoor de formule verbreekt. De mier begint dan vaak willekeurig en zonder een duidelijke strategie het rooster af te tasten op zoek naar een nieuw aanknopingspunt om van daaruit verder te gaan. Wanneer dit vaak gebeurt (en in combinatie met de tijdslimiet) blijft de fitheidswaarde zeer laag.

7.6 LOSE en dataselectie bij de multiplexer

Dezelfde experimenten en vergelijkingen kunnen we uitvoeren voor de 11-bit multiplexer. Hier ligt de nadruk echter op de volgende vraagstelling: kan men met behulp van minder leervoorbeelden toch een zo optimaal mogelijke

oplossing genereren? We concludeerden reeds dat het toepassen van LOSE zorgt voor een beter resultaat. Toch eist de evaluatie van een deelboom op alle leervoorbeelden (2048) zeer veel van de machine waarop de simulatie wordt gedraaid. Het is dan ook niet verwonderlijk dat het multiplexerprobleem het traagste probleem uit onze benchmarksuite is. Bovendien is het ook niet mogelijk om op die manier te beschikken over een gescheiden testverzameling aangezien alle beschikbare voorbeelden zijn opgebruikt. Indien we kunnen besluiten dat we slechts een handvol leervoorbeelden nodig hebben (i.p.v. 2048) kan dit een aanzienlijke (extra) snelheidswinst opleveren!



Figuur 7.8: *De invloed van LOSE op de robuustheid wanneer meerdere leervoorbeelden aangeboden worden.*

De leer- en testvoorbeelden worden gegenereerd met behulp van het algoritme beschreven in paragraaf 7.3.2. We laten het aantal leervoorbeelden variëren tussen 64 en 1280 met een stapgrootte gelijk aan 64 (in totaal werden 20 verschillende waarden uitgeprobeerd). De testfitheid wordt bepaald aan de hand van de overgebleven datawoorden. De methode BSS werd gebruikt om deelbomen te selecteren aangezien dit de enige methode is die de leerfitheid significant verhoogt. Het aantal beschikbare deelbomen bepaalt de probabiliteit waarmee LOSE wordt toegepast (automatische instelling). De resultaten van dit experiment zijn te zien in Figuur 7.8.

Ongeacht hoeveel leervoorbeelden precies worden gebruikt, merken we een algemene verbetering van zowel leer- alsook (en belangrijker nog) testfit-

heid[7]. Beide fitheidswaarden liggen rond de waarde 0.9 in plaats van 0.8 bij standaard GP. De verhoging van de leerfitheid werd reeds aangetoond in hoofdstuk 4 (vaste instelling voor p_{lose}) en hoofdstuk 6 (automatische bepaling van p_{lose}) maar dat LOSE ook bij minder dan het maximaal aantal leervoorbeelden uitstekende prestaties behaald op de testvoorbeelden is opzienbarend.

Bij gebruik van een beperkt aantal leervoorbeelden (< 320) produceert GP met LOSE slechte waarden voor de testfitheid (overfitting). Ook bij standaard GP treedt dit verschijnsel op maar daar is er pas vanaf meer dan 512 leervoorbeelden een verbetering merkbaar. LOSE presteert dus merkelijk beter en sneller. Vanaf meer dan 750 leervoorbeelden treden er een aantal schommelingen op voor beide fitheidsmaten. Deze schommelingen zijn eerder beperkt gelet op de amplitude.

Opmerkelijk is ook dat bij gebruik van minder leervoorbeelden de code toeneemt. Zo is de gemiddelde boomgrootte van een individu gekweekt met behulp van 64 leervoorbeelden vele malen groter dan een individu gekweekt met behulp van 1024 voorbeelden. Dit onderscheid is bijzonder groot voor individuen gegenereerd op basis van minder dan 320 leervoorbeelden. De groei van deze code verklaart wellicht waarom de testfitheid zo laag is. Een kleinere dataset biedt dus meer kans om te overspecialiseren dan een grotere dataset.

7.7 Besluiten

Vooraleer zich te buigen over de vraag hoe we robuustheid zullen meten en bevorderen, moet men eerst een strategie ontwikkelen om nieuwe testvoorbeelden aan te maken. Voorbeelden die bij voorkeur gelijkend zijn op (of verwantschap vertonen met) de reeds aanwezige leervoorbeelden. In dit hoofdstuk hebben we een aantal methoden ontworpen om extra leervoorbeelden op een automatische manier aan te maken bij de kunstmatige mier. Zo kunnen we ruis toevoegen aan het Santa Fe spoor. Bij de 11-bit multiplexer verdelen we eerst zo efficiënt mogelijk alle beschikbare voorbeelden (2048) in intervallen waarna uit elk interval een aantal voorbeelden worden geselecteerd en dubbels worden vermeden.

[7]Net zoals bij de kunstmatige mier werd ook dit experiment herhaald met behulp van een 'vaste' instelling voor p_{lose}. Deze resultaten waren echter minder positief. Er trad nauwelijks verbetering op van de leer- en testfitheid en bij voldoende leervoorbeelden (vanaf 512) traden er nog maar weinig verschillen in prestatie op.

Waar mogelijk, is het echter beter na te denken voor men de leerverzameling opstelt in plaats van voor de eenvoudigste oplossing te kiezen. Zo kan men uit de resultaten afleiden dat de ruis methode om sporen te genereren voor de kunstmatige mier, niet altijd sporen produceert die door een computer programma foutloos kan worden bewandeld. Omwille van opgelegde beperkingen in de functie en terminalenverzameling, is het voor de mier onmogelijk om verder dan één positie op het rooster te kijken. Ook is de mier beperkt wat betreft de kijkrichting (slechts vier richtingen mogelijk). Zo produceert de techniek van spoorgeneratie uit bouwblokken bij de kunstmatige mier (gebaseerd op de mogelijkheden van de functie- en terminalen verzameling!) of selectie uit intervallen bij de 11-bit multiplexer betere testvoorbeelden dan door enkel gebruik te maken van ruis.

Een tweede belangrijke vraag was de invloed van de lokale optimalisatieoperator op de robuustheid van de kandidaat-oplossingen. Specifiek voor de kunstmatige mier is men benieuwd naar het spoorvolggedrag van de geëvolueerde oplossingen bij gebruik van LOSE. Bij het multiplexerprobleem zijn we eerder benieuwd of met behulp van een kleiner aantal leervoorbeelden toch hetzelfde resultaat kan worden behaald. Zoals blijkt uit de resultaten verbetert LOSE (met behulp van BSS en een adaptieve instelling van p_{lose}), door rechtstreeks in te werken op het genotype, in de meeste gevallen op een significante manier de fitheid en, belangrijker nog, ook de robuustheid (onder de vorm van de testfitheid). Bovendien stelden we experimenteel vast dat bij de kunstmatige mier kleinere kandidaat-oplossingen steeds beter presteerden dan grotere individuen, wat experimenteel de theorie van Ockham's scheermes aantoont. Het is prettig vast te stellen dat LOSE de robuustheid van kandidaat-oplossingen niet tegenwerkt maar het in veel gevallen zelfs, eigenlijk als neveneffect, bevordert. Ook het aantal sporen dat noodzakelijk is om een goed resultaat te bekomen ligt beduidend lager dan wanneer LOSE niet wordt toegepast. Bij het multiplexerprobleem kan men besluiten dat niet alle 2048 voorbeelden ook daadwerkelijk noodzakelijk zijn om tot een goede oplossing te komen. Vaak volstaat minder dan de helft van het aantal beschikbare voorbeelden. Dit aantal ligt tevens ook lager dan wanneer LOSE niet wordt gebruikt.

Men kan veilig besluiten dat het gebruik van LOSE een stijging van de robuustheid van de kandidaat-oplossingen impliceert en bovendien minder leervoorbeelden nodig heeft om tot een goede oplossing te komen. Dit laatste voordeel uit zich dan in een sterke reductie van de rekentijd nodig voor de fitheidsbepaling van de individuen. Tot slot merken we op dat de lokale optimalisatieoperator zonder probleem samen met andere methoden zoals meerdere

leervoorbeelden, veranderen van leerverzameling om de zoveel generaties, etc. kan toegepast worden, om de robuustheid van kandidaat-oplossingen nóg sterker te verbeteren.

Hoofdstuk 8

Algemene besluiten

In dit hoofdstuk herhalen we kort de probleemstelling en de doelstellingen die in het begin van dit proefschrift werden vooropgesteld. Vervolgens vormen we enkele algemene besluiten op basis van de experimenten en voorlopige conclusies uit voorgaande hoofdstukken. Tenslotte formuleren we enkele beloftevolle ideeën voor verder onderzoek.

8.1 Genetisch programmeren en codegroei

Genetisch programmeren zoekt een computerprogramma dat zo goed mogelijk een bepaalde taak vervult. Dit programma kan om het even welke vorm en grootte aannemen zolang het uiteraard voldoet aan de beperkingen opgelegd door de toepassing. Het gebruik van zulke computerprogramma's biedt heel wat voordelen. Structuren met variabele lengte zijn immers zeer flexibel en passen zich dynamisch aan in functie van de moeilijkheidsgraad van de voorliggende toepassing. Maar wanneer GP wordt gebruikt om steeds complexere taken op te lossen, doemt een schijnbaar onverzetbare hindernis op: codegroei. Codegroei zorgt immers voor een heleboel bijwerkingen. We herhalen kort de belangrijkste nadelen.

Computerprogramma's met een groot aantal knopen (waarvan het merendeel vaak bestaat uit overbodige functies) nemen meer geheugenruimte in beslag waardoor ook het aantal geheugentoegangen en dus de benodigde rekentijd zal toenemen. Bovendien laat de leesbaarheid van de bekomen oplossing sterk te wensen over.

Door de toevloed aan code wordt ook de werking van de verschillende genetische operatoren negatief beïnvloed. Zij combineren immers vaak nutteloos genetisch materiaal. De strijd tegen codegroei kan men dus opvatten als een race tegen de klok: zoveel mogelijk goede oplossingen trachten te vinden vooraleer codegroei de zoektocht vertraagt en uiteindelijk stopt.

Een ander minstens even belangrijk nadeel is de uitvlakking van een verder positief verloop van de kwaliteit. 'Survival of the fattest' brengt de creatie van kwalitatief betere oplossingen abrupt tot stilstand.

Intuïtief kan men inzien dat grote individuen veel code met zich meedragen die enkel geschikt is voor de evaluatieomgevingen op basis waarvan de kwaliteit werd toegekend. Verandert men iets aan deze voorbeelden dan zal de kwaliteit vaak negatief worden beïnvloed. Deze individuen zijn met andere woorden niet robuust in functie van verandering in de evaluatieomgeving.

8.2 Korte herhaling van de beoogde doelstellingen

We herhalen eerst kort de beoogde doelstellingen zoals vermeld in hoofdstuk 1.

1. De ontwikkeling van een nieuwe methode om codegroei te bestrijden zonder de kwaliteit van de geëvolueerde computerprogramma's negatief te beïnvloeden.

2. Het ontwerpen van een nieuwe techniek die beter dan de bestaande methoden de structurele diversiteit binnen een populatie kan kwantificeren en kan gebruikt worden om eigenschappen van de evolutie te koppelen aan de diversiteit binnen de populatie. Deze techniek houdt rekening met het fenomeen codegroei.

3. De ontwikkeling van parametervrije methoden om een aanvaardbare instelwaarde voor p_{lose} te vinden. Deze sturingsalgoritmen hebben als primair doel de instellingen voor de gebruiker te beperken en te vereenvoudigen alsook om de probleem- en deelboomselectiemethode-afhankelijkheden weg te werken.

4. De implementatie van een algemeen toepasbaar raamwerk waarmee de robuustheid kan opgemeten worden. Verder wensen we tevens de invloed die de nieuw ontwikkelde codegroeibegrenzer uitoefent op de robuustheid van de bekomen oplossingen te vergelijken.

8.3 Besluiten

We tonen aan hoe elke doelstelling werd gehaald en plaatsen de belangrijkste besluiten nogmaals in de schijnwerpers.

8.3.1 Doelstelling 1: het gebruik van lokale optimalisatie

De lokale optimalisatieoperator muteert een boomstructuur aan de hand van de zorgvuldige selectie van een deelboom en het gebruik van kennis uit het probleemdomein (i.h.b. de fitness cases). Aan deze deelbomen worden twee belangrijke eisen opgelegd:

1. Een deelboom is een interne knoop van de boomstructuur, verschillend van de wortel.

2. De fitheid van de deelboom is minstens gelijk (of hoger) aan de fitheid van de volledige boomstructuur.

Hierdoor slaagt de lokale optimalisatieoperator erin om codegroei tegen te gaan en de gemiddelde boomgrootte te beperken. Omvangrijke individuen

worden zo omgevormd tot compacte exemplaren. En belangrijker nog: men hoeft niet in te boeten aan kwaliteit van de kandidaat-oplossingen. Er treedt dus geen verlies aan fitheid op zoals vaak het geval is bij andere codegroei-begrenzers.

Om de deelbomen uit de boomstructuur te lichten, werden er vijf verschillende methoden ontworpen:

- BSS: Beste deelboom zoeken waarbij tijdens de evaluatiefase de fitheidswaarden van alle deelbomen worden berekend. Enkel de beste deelboom wordt behouden.
- PRE: Preorde deelboom zoeken waarbij elke boomstructuur in preorde wordt doorzocht en de zoektocht stopt zodra een beter deelboom wordt gevonden.
- POST: Postorde deelboom zoeken waarbij elke boomstructuur in postorde wordt doorzocht en de zoektocht stopt zodra een beter deelboom wordt gevonden.
- BFS: Breedte-eerst deelboom zoeken waarbij elke boomstructuur in breedte-eerst volgorde wordt doorzocht en de zoektocht stopt zodra een beter deelboom wordt gevonden.
- iBFS: Omgekeerd breedte-eerst deelboom zoeken waarbij elke boomstructuur in omgekeerde breedte-eerst volgorde wordt doorzocht en de zoektocht stopt zodra een beter deelboom wordt gevonden.

Alhoewel elk van deze zoekmethoden aanvaardbare resultaten produceren zowel qua boomgrootte alsook qua fitheid, herbergt BSS steeds de beste verhouding tussen kwaliteit en boomgrootte reductie.

LOSE reduceert drastisch de vereiste geheugenruimte om de individuen op te slaan. Compacte individuen hebben immers minder geheugenlocaties nodig om hun structuur te herbergen. Ook de bijhorende selectiemechanismen en deelboomselectiestrategieën zijn zodanig geïmplementeerd dat een minimum aan extra geheugen vereist is.

Minder benodigde geheugenruimte betekent ook minder geheugentoegangen die nodig zijn om een deel van de boomstructuur in te lezen. Dit heeft een positieve invloed op de benodigde rekentijd. In hoofdstuk 4 werd echter opgemerkt dat het zoeken naar een geschikte deelboom heel wat rekentijd in beslag neemt. Daarom werd er ook aandacht besteed aan een algoritmische

optimalisatie voor de verschillende deelboomselectiemethoden. Aan de hand van enkele extra datastructuren worden steeds alle fitheidswaarden van alle deelbomen bijgehouden in het dataobject. De verschillende genetische operatoren worden aangepast zodat, naast de structurele wijzigingen, ook de corresponderende stukken uit de rij van fitheidswaarden van deelbomen correct worden gekopieerd. Het resultaat van deze ingrepen doet de rekentijd aanzienlijk slinken. Ook het aantal geëvalueerde knopen duidelijk is afgenomen.

Ook over de benchmarktoepassingen valt een en ander op te merken. Bij de kunstmatige mier slaagt LOSE met glans in het reduceren van de boomgrootte terwijl maximum fitheid gevoelig wordt verhoogd. LOSE presteert ook merkelijk beter in vergelijking met andere codegroeibegrenzers. Wanneer we enkel de simulaties die een correcte oplossing genereren in beschouwing nemen, geeft LOSE (eventueel in combinatie met een vereenvoudigingsmodule) compacte oplossingen die opnieuw leesbaar en bruikbaar zijn.

Het regressieprobleem is de meest verraderlijke toepassing uit de benchmarksuite, zoals standaard GP aantoont. GP in combinatie met LOSE heeft meestal weinig generaties nodig om een optimale oplossing te vinden. Vrijwel alle zoekmethoden vinden compacte oplossingen van de voorspelde minimale lengte. Bij dit probleem is er geen statistisch verschil tussen de andere codegroeibegrenzers en BSS. Wel genereert BSS de kleinste individuen wanneer we enkel de correcte oplossingen in beschouwing nemen.

De multiplexer is ongetwijfeld de moeilijkste toepassing uit de reeks. LOSE slaagt erin om de groei van het aantal knopen in te perken. Doch worden er niet dezelfde resultaten behaald als bij de twee voorgaande problemen. Enkel BSS slaagt erin om significant betere oplossingen te evolueren dan standaard GP of andere codegroeibegrenzers.

Tot op heden is LOSE de enige methode die erin slaagt om zowel de boomgrootte te reduceren alsook de kwaliteit van de kandidaat-oplossing te verhogen bij verschillende toepassingen.

8.3.2 Doelstelling 2: diversiteit bij GP en LOSE

Bestaande maten om diversiteit op te meten staan nog niet volledig op punt. Zo beïnvloedt codegroei het aantal pseudo-isomorfen in de populatie. In dit proefschrift werd een nieuwe structurele diversiteitsmaat geïntroduceerd: functionele pseudo-isomorfen waarbij twee vereenvoudigde boomstructuren met elkaar worden vergeleken op basis van het aantal functionele functieknopen, het aantal functionele terminalen en de diepte.

Functionele pseudo-isomorfen zijn een zeer geschikt alternatief voor toepassingen die zeer veel niet-functionele code genereren zoals de kunstmatige mier. In tegenstelling tot de originele genotypische diversiteitsmaat treedt er weldegelijk structurele convergentie op bij de mier.

Natuurlijk zijn deze experimenten maar gebeurd op een klein aantal toepassingen, en er treedt in de resultaten een zekere mate van probleemafhankelijkheid op. Toch zijn we ervan overtuigd dat de vernieuwde maat toepasbaar is op soortgelijke problemen, i.h.b. problemen die erg te lijden hebben onder het fenomeen codegroei. Het bepalen van de functionele pseudo-isomorfen vereist het opstellen van herleidingsregels voor de programmabomen, maar dit enkel om onderscheid te maken tussen code die wel en die niet bijdraagt tot de fitheid, een onderscheid dat bij elke toepassing bestaat. Het is duidelijk dat de wijze van opstellen van de diversiteitsmaten niet probleemafhankelijk is.

Zoals aangetoond in hoofdstuk 6 veroorzaakt het gebruik van de lokale optimalisatieoperator een verlies aan diversiteit bij hoge instelwaarden voor de probabiliteit waarmee deze operator toegepast wordt. Op basis van een eenvoudige experimentele proefopzet konden we besluiten dat hoge instelwaarden voor p_{lose} alle bestaande structurele diversiteit binnen de populatie vernietigen, ongeacht welke deelboomselectiestrategie er wordt gekozen of over welke toepassing het gaat. Door de gehanteerde werkwijze van het selectiealgoritme, worden er verschillende identieke kopieën van deelbomen in de nieuwe populatie ingebracht waardoor de structurele diversiteit na enkele generaties tot een minimum wordt herleid. De bekomen oplossingen zijn in het beste geval deeloplossingen die slechts een deelaspect van de volledige probleemstelling behandelen.

Uit een analyse van het aantal beschikbare deelbomen blijkt echter dat het aanbod niet steeds constant is en dat er op elke generatie niet evenveel geschikte deelbomen voorhanden zijn. Hierdoor is het mogelijk dat een aanvaardbare instelling voor p_{lose} bekomen via parameter tuning toch een te hoge instelling blijkt te zijn gedurende de latere generaties.

Verder merken we ook op dat vooral tijdens de eerste generaties de impact van de LOSE operator zeer groot is. Na enkele generaties is het immers zo dat de meeste diversiteitsmaten hun eindwaarde hebben bereikt. De instelling die men in het begin gebruikt speelt een belangrijkere rol dan de instelling halfweg de uitvoering van het programma.

8.3.3 Doelstelling 3: de instelling van $p_{\mathbf{lose}}$

Een belangrijk probleem waarmee de gebruiker van LOSE wordt geconfronteerd, is de instelling van de probabiliteit waarmee LOSE wordt toegepast. Deze vereist enige voorkennis van het probleemdomein en wordt mogelijks nog meer gecompliceerd door de afhankelijkheid van de verschillende deelboomselectiestrategieën. In eerste instantie werd de techniek parameter tuning gebruikt om een geschikte instelwaarde te vinden. De stapgrootte bij het off-line trachten te vinden van de optimale waarde is opnieuw een extra parameter die hoofdzakelijk wordt gekozen in functie van de beschikbare rekentijd.

In dit proefschrift worden twee adaptieve modellen voorgesteld om de instelwaarde automatisch te bepalen. Beide methoden maken gebruik van feedback vanuit het zoekproces.

1. De sterk geparametriseerde methode gebaseerd op vaaglogische verzamelingen bleek gelijkaardige resultaten op te leveren dan het gebruik van een vaste instelling voor p_{lose}. Dit model bleek echter complex en vereiste een aantal extra parameterinstellingen zoals het aantal vaagverzamelingen, de vage regels, etc. Om goede waarden voor deze parameters te kiezen, moet de applicatieprogrammeur nog steeds beschikken over voldoende kennis vanuit het probleemdomein. Bovendien bleef er een zekere deelboomselectiemethodeafhankelijkheid bestaan.

2. De tweede univariate methode gebaseerd op het aantal beschikbare deelbomen in de populatie, bleek bijzonder effectief. Maximum fitheid werd nogmaals significant hoger voor het multiplexerprobleem en de kunstmatige mier. Ook de gemiddelde boomgrootte nam toe, toch bleef de boomgrootte van het beste individu voor elke generatie quasi onveranderd. Deze methode biedt tal van voordelen zoals zijn eenvoud, betere resultaten, probleemonafhankelijk, en parametervrij.

8.3.4 Doelstelling 4: de robuustheid van individuen bij GP en lokale optimalisatie

Robuustheid is een belangrijke maatstaf bij kunstmatige intelligentie methoden. Ook bij genetisch programmeren wordt de evolutie van robuuste, breed toepasbare programma vooropgesteld. Helaas gaat er men er vaak simpelweg vanuit dat GP robuuste individuen produceert en wordt deze bewering zelden geverifieerd.

In dit proefschrift hebben we een aantal methoden ontworpen om extra leervoorbeelden op een automatische manier aan te maken. We gingen ook de invloed na van de lokale optimalisatieoperator op de robuustheid van de kandidaat-oplossingen.

Bij de kunstmatige mier kunnen we ruis toevoegen aan het Santa Fe spoor. Deze methode produceert echter niet altijd sporen die door een computer programma foutloos kan worden bewandeld omwille van opgelegde beperkingen in de functie en terminalenverzameling. Een tweede techniek, spoorgeneratie uit bouwblokken (gebaseerd op de mogelijkheden van de IF-FOOD-AHEAD sensor), produceert betere testvoorbeelden. Specifiek voor deze toepassing zijn we benieuwd naar het spoorvolggedrag van de geëvolueerde oplossingen na toepassing van lokale optimalisatie. Door rechtstreeks in te werken op het genotype verbetert LOSE in de meeste gevallen op een significante manier ook de robuustheid (onder de vorm van de testfitheid). Bovendien stelden we experimenteel vast dat bij de kunstmatige mier kleinere kandidaat-oplossingen steeds beter presteerden dan grotere individuen. Ook het aantal sporen dat noodzakelijk is om een goed resultaat te bekomen ligt beduidend lager dan wanneer LOSE niet wordt toegepast.

Bij het multiplexerprobleem verdelen we alle beschikbare voorbeelden (2048) op een zo efficiënt mogelijke manier over het aantal intervallen. Vervolgens wordt er, op willekeurige basis, één leervoorbeeld uit elk interval geselecteerd. Bij deze toepassing zijn we benieuwd of met behulp van een kleiner aantal leervoorbeelden toch hetzelfde resultaat kan worden behaald. We besluiten dat niet alle 2048 voorbeelden ook daadwerkelijk noodzakelijk zijn om tot een goede oplossing te komen. Vaak volstaat minder dan de helft van het aantal beschikbare voorbeelden. Dit aantal ligt tevens ook lager dan wanneer LOSE niet wordt gebruikt.

We besluiten dat het gebruik van LOSE een stijging van de robuustheid van de kandidaat-oplossingen impliceert en bovendien minder leervoorbeelden nodig heeft om tot een goede oplossing te komen. Dit laatste voordeel uit zich dan in een sterke reductie van de rekentijd nodig voor de fitheidsbepaling van de individuen. Tot slot merken we op dat de lokale optimalisatieoperator samen met andere methoden om de robuustheid van kandidaat-oplossing te verhogen, kan toegepast worden, om de robuustheid nóg sterker te verbeteren.

8.3.5 . . . en wat met andere evolutionaire algoritmen?

De experimenten zijn verricht op genetisch programmeren, wat slechts één type evolutionair algoritme is. Nochtans is genetisch programmeren niet de enige methode waarop de voorgestelde algoritmen en methoden uit dit proefschrift toepasbaar zijn.

LOSE zal zich nuttig kunnen maken in verschillende variabele-lengte zoekmethoden. In eerste instantie denken we uiteraard aan zijn rol als codegroeibegrenzer maar LOSE kan ook worden gebruikt als zuivere optimalisatieoperator om de fitheid van de individuen te verbeteren. LOSE kan immers leiden tot de identificatie van compacte stukjes code met hoe fitheid. Deze codefragmenten kunnen bijvoorbeeld toegevoegd worden aan een bibliotheek met subroutines waaruit andere genetisch operatoren informatie kunnen putten. Zo kan mutatie een codefragment uit deze bibliotheek gebruiken in plaats van een nieuw (willekeurig) stukje code te creëren.

Ook de verbeterde structurele diversiteitsmaat is van toepassing op meer dan enkel GP. Er zijn nog andere evolutionaire algoritmen met genotypische representaties met variabele afmetingen. Sommige genetische algoritmen werken met bitstrings met variabele lengte, andere werken zelfs met bomen. In deze domeinen kunnen dus genotypische diversiteitsmaten analoog aan de pseudo-isomorfen gebruikt worden. De verbeteringen die aan deze maat zijn aangebracht in dit proefschrift zijn weliswaar expliciet gebaseerd op het voorkomen van codegroei en niet-functionele code, maar de uiteindelijk bekomen genotypische diversiteitsmaat presteert zo goed dat het zeker de moeite waard moet zijn een vergelijkbare maat, gebaseerd op vormklassen, te ontwikkelen voor die evolutionaire algoritmen waar de genetische structuur dat mogelijk maakt.

8.4 Verder onderzoek

Tot slot suggereren we nog enkele interessante ideeën voor verder onderzoek.

8.4.1 Op weg naar parameterloos genetisch programmeren

Het gebruik van feedback vanuit het algoritme om bepaalde parameterinstellingen *at run time* te bepalen is niet nieuw. Nochtans worden deze concepten nauwelijks aangewend bij genetisch programmeren. Op zich is dit wel vreemd aangezien precies genetisch programmeren zeer veel invoer van de

applicatieprogrammeur vraagt: er zijn immers vele tientallen parameters die een correcte instelling vereisen. Dit legt een serieuze last op de schouders van de gebruiker van het GP systeem. Het lijkt zelfs zo dat de kennis die de gebruiker heeft over de toepassing onder de vorm van de talloze parameterinstellingen moet ingevoerd worden in het GP algoritme. Helaas vereist dit een doorgedreven kennis van de werking van een GP systeem (het effect van een verandering van de instelwaarde van de parameter is niet gekend) en is deze kennis vaak niet voorhanden bij de eindgebruiker. Omgekeerd zal iemand die de werking van een GP systeem kent zoals zijn broekzak, vaak weinig of geen kennis hebben van het op te lossen probleem waardoor de gekozen instellingen minder goed (of soms zelfs helemaal niet) werken (al is dit niet de taak van de ontwerper).

De sturing op basis van eenvoudige feedback vanuit het algoritme (zoals het gebruik van het aantal beschikbare deelbomen bij de lokale optimalisatieoperator) of de toepassing van complexere regelaars (zoals de vaaglogische regelaar) kunnen eveneens aangewend worden bij het instellen van andere parameters uit het GP systeem. Een zeer nauwkeurige opmeting van verschillende prestatiemeters dringt zich op. Zo meten we bij de lokale optimalisatieoperator het aantal beschikbare deelbomen, de diepte van het wijzigingspunt, de boomgrootte voor en na het uitvoeren van de operator en nog een aantal andere parameterwaarden. We kunnen hetzelfde doen voor de andere genetische operatoren zoals kruising en mutatie. Zo kan men de diepte van het wijzigingspunt bij kruising laten afhangen van het procentueel aantal kruisingen waarbij de fitheid van de geproduceerde kinderen stijgt. Uit de beperkte informatie die hierover in de literatuur te vinden is, blijkt duidelijk dat er nog heel wat onderzoek kan worden verricht naar een parametervrije vorm van GP.

8.4.2 Diversiteitsbevorderende methoden en LOSE

Vele van de genotypische en fenotypische diversiteitsmaten uit hoofdstuk 5 meten de verscheidenheid binnen een populatie als geheel. Andere diversiteitsmaten zoals de transformatieafstand dienen om het verschil tussen aparte individuen te meten en situeren zich op het niveau van het individu. Dit type kan direct toegepast worden om te proberen de diversiteit hoog te houden tijdens de evolutie. We hebben immers aangetoond dat LOSE, door het inbrengen van compacte deelbomen, voor een verlies aan diversiteit zorgt. Dit werd gedeeltelijk opgelost dankzij het gebruik van adaptieve sturingsalgoritmen voor de instelwaarde van p_{lose}. Toch is er nog verbetering mogelijk met

behulp van diversiteitsbevorderende methoden. Er bestaan meerdere methoden die de selectie van individuen voor voortplanting beïnvloeden op basis van de verschillen tussen individuen, waarbij het dan de bedoeling is die kandidaat-oplossingen te bevoordelen die voor een gevarieerder nageslacht kunnen zorgen. Daarbij mag natuurlijk ook weer niet overdreven worden, het blijft zaak om een evenwicht te vinden tussen exploratie en exploitatie. Teveel diversiteit zou immers de exploitatie fase grondig kunnen verstoren.

8.4.3 Een constructieve kruisingoperator

Het gebruik van de fitheid van de deelboom (i.e. de kwaliteit van elke interne knoop van een individu) opent heel wat perspectieven. In dit proefschrift werden deze waarden hoofdzakelijk gebruikt voor de lokalisatie van het wijzigingspunt bij lokale optimalisatie. Maar ook andere operatoren kunnen hier dankbaar gebruik van maken. Recent wordt er veel onderzoek verricht naar de bepaling van de diepte van het kruisingspunt. Zoals aangetoond in hoofdstuk 3 vermindert het aantal constructieve kruisingen (kruisingen waarbij de fitheid van de kinderen groter is dan de fitheid van de ouderindividuen) zeer sterk naarmate het aantal generaties toeneemt. Hiermee gaat een stijging van het aantal destructieve (en neutrale) kruisingen gepaard wat uiteraard de zoeksnelheid —namelijk het vinden van nieuwe betere kandidaat-oplossingen— niet meteen ten goede komt. Een mogelijke oplossing bestaat erin om de berekende deelboom fitheden te gebruiken bij de selectie van het kruisingspunt. We kunnen de geëvalueerde deelbomen beschouwen als probabiliteiten of waarschijnlijkheden. Veronderstel dat een boom beschikt over drie interne knopen waarvan de fitheden zijn: 0.5; 0.6 en 0.9. De totale som is gelijk aan 2. In plaats van deze waarden louter als fitheid te beschouwen, kunnen we ze interpreteren als probabiliteiten, nl. 0.25; 0.30 en 0.45. We kunnen nu de keuze van het kruisingspunt laten afhangen van deze probabiliteit. Met andere woorden de kans om een interne knoop te kiezen wiens deelboom een hoge fitheid heeft zal groter zijn dan een interne knoop met een lage fitheid. In de andere ouder kan men dan net andersom tewerk gaan: daar kiest men bijvoorbeeld eerder voor knopen met een lage fitheidswaarde. Tenslotte worden beide deelbomen uitgewisseld in de hoop een nieuw individu te creëren met een hogere fitheidswaarde dan de ouders. We hopen een verband te ontdekken tussen het uitwisselen van fitte deelbomen en het ontstaan van nieuwe kinderindividuen wiens fitheid groter is dan de fitheid van de ouders. Uiteraard is het zeer voorbarig om enkel op basis hiervan een voorspelling te doen van de kwaliteit van de nakomelingen en verder experimenteel onderzoek is

nodig. Toch geven de resultaten op een beperkt aantal proefexperimenten nu reeds aan dat het aantal constructieve kruisingen stijgt. Verder onderzoek is nodig om de invloed op de finale kwaliteit van de kandidaat-oplossingen na te gaan. We hopen tevens dat de analyse van de diepte van het kruisingspunt nuttige informatie zal opleveren die leidt tot een dieper inzicht in de werking van het GP algoritme.

8.4.4 Standaardisatie van een raamwerk voor robuustheid

Talloze boeken en video's van de hand van Koza tonen aan dat de toepassingsmogelijkheden van genetisch programmeren quasi eindeloos zijn. Men kan het zo gek niet bedenken of GP wordt gebruikt voor de optimalisatie van een grote verscheidenheid aan reële toepassingen zoals DNA sequencing, ontwerp van regelaars, model selectie, etc. Helaas bestaat er tot op heden geen algemeen bruikbaar platform voor het leren en testen op verschillende (gescheiden) datavoorbeelden. In dit proefschrift werd een eerste stap gezet in de richting van het ontwikkelen van robuuste fout-tolerante programma's die in soortgelijke omgeving goed blijven presteren. Maar er is nog heel wat werk aan de winkel. We hopen dat dankzij het gratis aanbieden van alle ontwikkelde software het gebruik van afzonderlijke leer- en testdata een standaard gebruik wordt in de GP gemeenschap.

Bijlage A

Genetisch programmeren in detail

A.1 Evolutionaire zoekalgoritmen

Er bestaat een brede waaier aan technieken om optimalisatieproblemen op te lossen. Elk algoritme of heuristiek heeft zijn eigen voor- en nadelen en het is onmogelijk te zeggen welke zoektechniek nu de beste prestaties zal leveren (Wolpert & Macready 1997). Toch hebben evolutionaire zoekalgoritmen een stapje voor op de meer traditionele technieken.

Optimalisatiealgoritmen zoals hill climbing en simulated annealing (Kirkpatrick *et al.* 1983) werken met één enkele mogelijke oplossing. Ze vertrekken vanuit één punt in de zoekruimte, en verplaatsen dat punt stap per stap in de hoop een beter punt te vinden. Deze algoritmen werken zeer lokaal. Als het vertrekpunt toevallig in de buurt van een lokaal optimum ligt, kan dit het vinden van het globale optimum grondig vertragen of zelfs onmogelijk maken. Ook rekenkundige technieken zoals bijvoorbeeld gradiënt gebaseerde optimalisatiemethoden, convergeren naar het dichtstbijzijnde lokale optimum (indien deze er zijn) uitgaande van de initiële keuze van de beginvoorwaarden. Om er zeker van te zijn dat men het globale optimum zal vinden, dient bijgevolg initiële informatie over dit optimum reeds voorhanden te zijn. Dit is niet altijd het geval.

Een andere mogelijkheid zijn de exhaustieve en *random search* optimalisatietechnieken, eventueel in combinatie met bovenvermelde methoden. Aangezien deze algoritmen de volledige zoekruimte (of grote delen ervan) doorlopen, zijn deze technieken zeer rekenintensief. De rekentijd kan zeer hoog

oplopen wanneer het aantal dimensies in de zoekruimte toeneemt[1] (Bellman 1961). Deze technieken presteren relatief goed als er slechts één globaal optimum is. Vele reële problemen bezitten echter vaak verschillende globale en lokale optima waardoor de efficiëntie van deze technieken daalt. Reële problemen zijn ook dikwijls onderhevig aan ruis. Dit compliceert verder het efficiënt (robuust) zoeken naar het globale optimum.

In tegenstelling tot de 'traditionele' optimalisatiealgoritmen zijn evolutionaire algoritmen globale en robuuste optimalisatiemethoden, nauw verbonden met natuurlijke evolutionaire processen, zoals beschreven door Charles Darwin (Darwin 1959). Evolutionaire algoritmen vertrekken vanuit een populatie van willekeurig gekozen mogelijke oplossingen, in plaats van één enkele mogelijke oplossing. Op die manier verkleint ook de kans dat het algoritme convergeert naar een lokaal optimum. De overlevingkans van een individu en de creatie van nieuwe individuen, generatie na generatie, worden gegarandeerd door elementaire Darwinistische richtlijnen. Aan de hand van hoofdzakelijk drie belangrijke operatoren: reproductie, kruising en mutatie, worden nieuwe populaties gecreëerd uit hun voorgangers. De nieuwe populatie heeft daarbij een grote kans om gemiddeld 'fitter' te zijn dan de vorige. Het is verbazingwekkend vast te stellen hoe uitgaande van willekeurige (random) elementen en onder invloed van het principe 'survival of the fittest', EA's omwille van hun robuustheid en universaliteit efficiënte oplossingen genereren in een multidimensionele zoekruimte met heel wat ruis en lokale optima.

De meest bekende vorm van EA's is het genetische algoritme (GA). Deze methode werd bedacht tijdens de jaren '60 door onderzoeker John Holland aan de universiteit van Michigan (Holland 1975). Lange tijd werkten enkel hij en zijn studenten ermee, maar in de loop van de jaren raakten de GA's meer verspreid, en in 1989 bracht het boek "Genetic Algorithms in Search, Optimisation and Machine Learning" van David Goldberg (universiteit van Alabama) de GA's algemene bekendheid. Sindsdien worden de GA's beschouwd als veelbelovende optimalisatietechnieken en worden ze meer en meer gebruikt voor een grote verscheidenheid aan optimalisatietoepassingen.

We eindigen deze inleiding met een uitspraak van David Goldberg (Goldberg 1989):

> "... where robust performance is desired (and where is it not), nature does it better; the secrets of adaptation and survival are best learned from a careful study of the biological example. Yet we do not accept the genetic algorithm method by appeal to this

[1] *Curse of dimensionality*

beauty-of-nature argument alone. Genetic algorithms are theoretically and empirically proven to provide robust search in complex spaces."

A.2 Genetische algoritmen

A.2.1 Verschil met andere zoektechnieken

Zoals reeds in de inleiding werd verteld, verschilt een genetisch algoritme van een traditionele zoektechniek op de volgende aspecten.

Het genetisch algoritme manipuleert de te optimaliseren parameters (meestal) niet rechtstreeks, maar bewerkt een gecodeerde versie van de parameters. Heel dikwijls worden de parameters gecodeerd in een bitstring. In het standaard genetisch algoritme heeft deze bitstring een vaste lengte. Wanneer men een bepaalde functie $f(x)$ wenst te maximaliseren naar x (binnen een bepaald interval), dan is de eerste stap het opstellen van een efficiënte bitcodering voor de parameter x. Het aantal bits is afhankelijk van het gewenste bereik en de gewenste nauwkeurigheid. De bitstring wordt het "genoom" of het "chromosoom" van het individu genoemd.

Het vinden van een goede codering is niet altijd even vanzelfsprekend. Er moet immers op gelet worden dat de decodering altijd kan lukken. Er mogen geen individuen gevormd worden met een bitstring waaraan geen betekenis kan toegewezen worden. Het is immers mogelijk dat bij een bepaalde coderingsmethode niet alle willekeurige combinaties van bits een geldige kandidaat-oplossing vormen. Ook de voortplantingsoperatoren mogen geen ongeldige bitstrings produceren.

Genetische algoritmen starten met een populatie van kandidaat-oplossingen in plaats van één enkel punt.

Op basis van het gedrag van een bitstring[2] krijgt elk individu een fitheidswaarde toegekend, een getal dat aanduidt in hoeverre het individu een goede oplossing is voor het behandelde probleem. Dit getal is het resultaat van de toepassing van de "fitheidsfunctie" (of "doelfunctie", "objective function") op het genoom van het individu. In het voornoemde maximalisatieprobleem, zou een goede keuze voor deze fitheidsfunctie de functiewaarde $f(x)$ van de gedecodeerde waarde x van de kandidaat-oplossing zijn: hoe hoger deze functiewaarde, hoe beter x het optimum benadert. Deze fitheidsfunctie

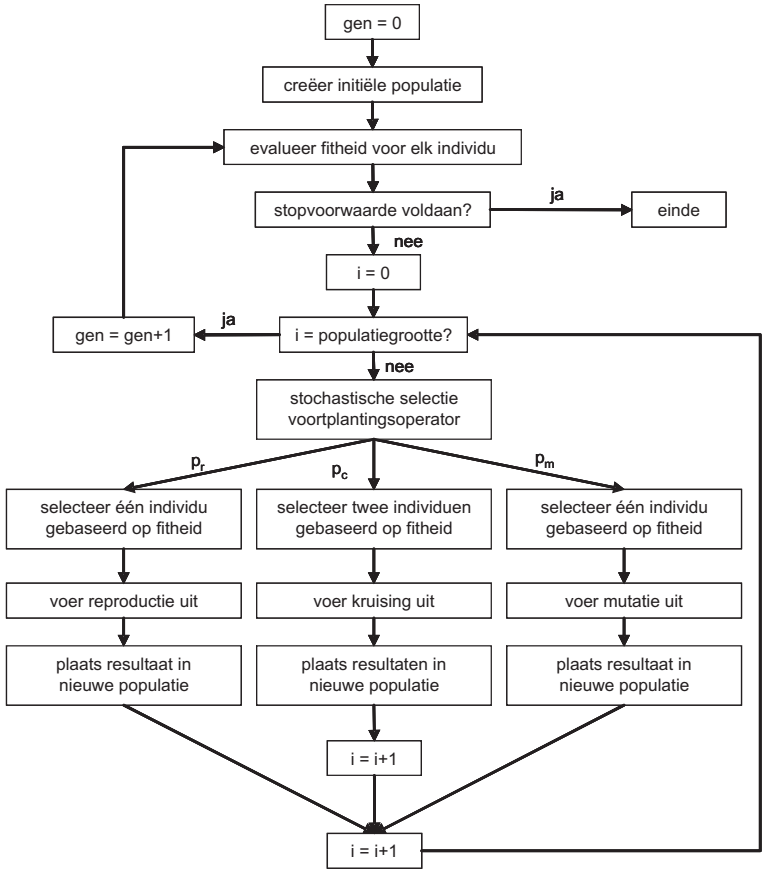
[2]Het gedrag van een bitstring wordt vaak de fenotypische uiting of kortweg het 'fenotype' genoemd. De bitstring zelf noemt men vaak de genotypische voorstelling of het 'genotype'.

is de enige vorm van informatie die het GA nodig heeft om te kunnen werken. Geen bijkomende informatie is nodig om een individu te evalueren. In klassieke optimalisatiemethoden is dikwijls wel zulke extra informatie nodig (zoals bijvoorbeeld afgeleiden). Deze methoden zijn dan ook dikwijls geschreven voor één specifiek probleem. GA's zijn meer algemeen toepasbaar.

De individuen die later structurele operaties zullen ondergaan, worden stochastisch gekozen. Individuen met een betere fitheid hebben een grotere kans hebben om gekozen te worden. Daaruit volgt dat nieuwe generaties gemiddeld fitter zijn dan de vorige generaties, wat leidt tot steeds betere oplossingen. De keuze van een selectiemethode werkte het ontstaan van een groot aantal verschillende methodes in de hand. De meest voorkomende methode is *roulette wheel selection* (roulettewielselectie), waarbij de kans dat een individu gekozen wordt gelijk is aan de verhouding van zijn fitheid tot de totale fitheid van de populatie. Naar analogie met een roulettespel is de grootte van de vakjes waar het balletje in kan vallen evenredig met de fitheid van het individu horend bij dat vakje. Andere bekende methoden zijn *rank selection* (rangselectie), waarbij met de rang van de individuen (gesorteerd op fitheid) gewerkt wordt, en *tournament selection* (tornooiselectie), waarbij een groep van (twee of meer) willekeurige individuen samengesteld wordt en uit deze groep het meest fitte individu gekozen wordt.

Naar analogie met natuurlijke evolutie, ontstaan nieuwe generaties individuen door reproductie, kruising (of recombinatie) en mutatie van hun genotypes. Bij klassieke optimalisatietechnieken vinden de optimalisatiestappen op deterministische wijze plaats (zeker bij *hill climbing*, ook in de latere fasen van een *simulated annealing* proces). Bij genetische algoritmen echter worden de overgangen naar nieuwe individuen op probabilistische wijze geregeld. Het precieze effect dat deze operaties op de genomen hebben is toevalsafhankelijk, evenals het al of niet toepassen van één van de drie operaties. Zo gebeuren reproductie en recombinatie dikwijls met een zo goed als zekere waarschijnlijkheid, terwijl mutatie een veel lagere waarschijnlijkheid heeft. Eigenschappen van het algoritme zoals snelheid van convergentie en stilvallen van de optimalisatie kunnen afhangen van deze waarschijnlijkheden. Soms is enige afstelling van deze parameters nodig om tot aanvaardbare resultaten te komen.

Een overzicht van een volledige evolutionaire cyclus bij een GA is te vinden in Figuur A.1.



Figuur A.1: *Cyclus van het genetisch algoritme. p_r is de kans op reproductie, p_m is de kans op mutatie en p_c geeft de kans op kruising weer.*

A.2.2 De operatoren bij een GA

Typisch genetisch gedrag van biologische soorten kan samengevat worden door drie essentiële operatoren (inwerkende op de chromosomen): reproductie, kruising en mutatie. Een computerimplementatie van dezelfde drie operatoren wordt gebruikt in een GA (inwerkende op de gecodeerde bitstrings). De keuze van welke operator wanneer toegepast wordt, gebeurt meestal op een probabilistische basis. Aanvaardbare probabiliteitswaarden (voor een specifiek probleem) worden vaak op een empirische wijze gevonden (*trial and error*).

Reproductie behelst het louter dupliceren van een chromosoom. Een duplicaat van een individu zal een grotere probabiliteit hebben om in de volgende generatie voor te komen indien de fitheid groot is. In tegenstelling tot de biologische wereld wordt nu echter het begrip fitheid (meestal) gereduceerd tot een enkelvoudige functiewaarde.

De kruisingsoperator kan gezien worden als de voortplanting tussen twee geselecteerde individuen. Het nieuwe chromosoom een recombinatie is van zijn ouders. In het meest eenvoudige geval van kruising, kiest men willekeurig eenzelfde positie uit bij de oorspronkelijke chromosomen. Vanaf deze positie worden dan de beide (bitstring-) helften met elkaar verwisseld.

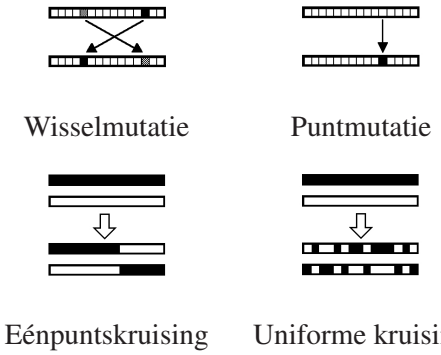
Reproductie en kruising worden beschouwd als de belangrijkste mechanismen voor het beschrijven van het genetisch gedrag. Als gevolg echter van de probabilistische richtlijnen, kan belangrijke informatie verloren gaan bij het toepassen van beide operatoren. Bijvoorbeeld een belangrijke '1' kan omgezet worden naar een nutteloze '0' m.b.v. bijvoorbeeld kruising. Het is tevens mogelijk dat deze informatie definitief verloren gaat in de populatie. Daarom is een bijkomende operator voorzien: de mutatieoperator. Mutatie heeft een ondergeschikte rol t.o.v. reproductie en kruising en heeft meestal een zeer lage probabiliteit van voorkomen (typisch 1:1000). Toch vervult hij een niet onbelangrijke rol in het behouden van de diversiteit van genetische informatie. Bij de meest voorkomende vorm van mutatie wordt een arbitraire bit veranderd van '1' naar '0' of van '0' naar '1'. Het kan beschouwd worden als een willekeurig doorlopen van alle posities van het chromosoom. Bij matig (voorzichtig) gebruik, kan deze operator vermijden dat het genetisch systeem vroegtijdig (en foutief) convergeert of wegens gebrek aan informatie niet convergeert.

Er bestaat een brede waaier aan types mutaties en kruisingen. Enkele voorbeelden van standaard operatoren zijn te zien op Figuur A.2.

A.2.3 Een eenvoudig voorbeeld (Goldberg 1989)

Tabellen A.1 en A.2 geven het verloop van de reproductie en de kruising voor één cyclus, uitgaande van een eenvoudig functie optimalisatieprobleem. Het doel is een waarde voor x te vinden waarvoor de doelfunctie $f(x) = x^2$ maximaal is. De variabele x ligt binnen het gesloten interval $[0, 31]$. Daarbij worden de volgende stappen doorlopen:

1. De populatie bestaat uit vier verschillende individuen. Elk van deze individuen worden op willekeurige basis geïnitialiseerd. Voor de een-



Figuur A.2: *Standaard vormen van mutatie en kruising op bitstrings.*

voud beperken we de waarden tussen nul en 31. De bitvoorstelling bestaat dus uit vijf bits.

2. Roullettewiel selectie wordt gebruikt om individuen te selecteren die vervolgens door kruising worden aangepast. Het resultaat is weergegeven in de laatste kolom van Tabel A.1.

3. Paren bitstrings worden geselecteerd door kruising. Er wordt eveneens een kruisingspunt bepaald (op willekeurige basis) waarna de kruising wordt uigevoerd.

4. De nieuwe populatie wordt neergeschreven (einde kruisingsfase) en de corresponderende fitheid wordt berekend (Tabel A.2).

De totale fitheid van de nieuwe populatie (1754) is beduidend hoger dan die van de vorige populatie (1170). Zelfs de fitheid van de populatie na reproductie (zonder kruising) ligt hoger (1682) dan de fitheid van de initiële populatie.

Het is duidelijk dat toevalseffecten een belangrijke rol spelen bij genetische algoritmen. Toch is het zoekproces niet geheel willekeurig, aangezien het hele proces erop gericht is de geschikte kenmerken vervat in de populatie in zijn geheel te bewaren. Het is zowel theoretisch als empirisch aangetoond dat genetische algoritmen een robuuste zoektechniek vormen binnen een verscheidenheid aan zoekruimtes (Goldberg 1989).

individu	bitstring	x	fitheid ($f(x)$)	% fitheid	geselecteerde exemplaren
1	01101	13	169	14%	1
2	11000	24	576	49%	2
3	01000	8	64	6%	0
4	10011	19	361	31%	1
totaal			1170	100%	4

Tabel A.1: *De initiële populatie bestaat uit vier individuen. De evaluatiefunctie is $f(x) = x^2$. De voorlaatste kolom geeft de verhouding weer tussen de fitheid van een individu en de totale fitheid. Bij roulettewielselectie duidt dit getal de kans aan dat een individu geselecteerd wordt. Het uiteindelijke aantal exemplaren van elk individu staat in de laatste kolom.*

paren voor reproductie	kruisingspunt	nieuwe bitstring	x	$f(x)$
011|01	3	01100	12	144
110|00	3	11001	25	625
10|011	2	10000	16	256
11|000	2	11011	27	729
Totaal				1754

Tabel A.2: *Dezelfde populatie na kruising.*

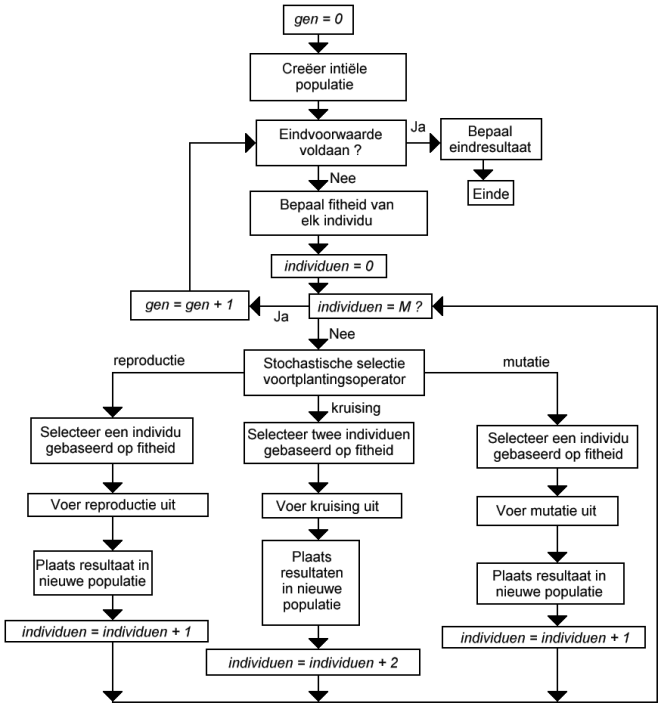
A.3 Genetisch algoritme voor programma-inductie

Net zoals een genetisch algoritme, steunt ook genetisch programmeren (GP) op dezelfde Darwinistische principes. GP is echter in staat om computerprogramma's te kweken in plaats van bitstrings met een vaste lengte zoals bij klassieke implementaties van een GA[3]. Verder heeft GP enkel een aantal (invoer, uitvoer)-combinaties nodig op basis van dewelke het programma zal worden opgebouwd of geïnduceerd.

Een individu wordt vaak voorgesteld als een boom, zonder vaste afmetingen, die kan uitgevoerd worden als een programma. De evolutionaire cyclus is net dezelfde als die van een genetisch algoritme en begint met het aanmaken van een initiële populatie. De programmabomen in deze populatie zijn willekeurige samenstellingen van de gebruikte functies en terminalen. Vervolgens wordt elk programma in de populatie uitgevoerd, en wordt er een fitheid aan

[3]Sommige types GA ondersteunen ook structuren met variabele lengte (Radcliffe & George 1993, Kargupta 1996, Goldberg *et al.* 1993, Fullmer & Miikkulainen 1992).

toegekend op basis van de prestatie. Met behulp van de drie gekende voortplantingsoperatoren, reproductie, recombinatie en mutatie, wordt een nieuwe generatie programma's aangemaakt, waarbij de betere (meer fitte) individuen meer kans hebben om hun genetisch materiaal naar de volgende generatie over te dragen. Dit proces van creatie en evaluatie wordt verder herhaald, tot een bepaald eindcriterium voldaan is. Tijdens de evolutie wordt de tot dan toe beste kandidaat-oplossing bijgehouden, en op het einde van de evolutie is deze het uiteindelijke resultaat van het GP algoritme.



Figuur A.3: *Cyclus van genetisch programmeren*

Een volledig overzicht van de evolutionaire cyclus bij genetisch programmeren kan men vinden in Figuur A.3. In de volgende paragrafen verklaren we elke stap uit het hierboven kort samengevatte proces in detail.

A.4 Het genoom

A.4.1 Functieset en terminalenset

De allereerste stap is het bepalen van de functieset, dat is de verzameling mogelijke functies voor de interne knopen van een boom, en de terminalenset, de verzameling mogelijke eindkopen van een boom. De samenstelling en inhoud van beide verzamelingen wordt bepaald door de probleemstelling. Bij een regressieprobleem kan men bijvoorbeeld opteren voor rekenkundige operatoren ($+$, $-$, $*$ ...) en wiskundige functies (sin, cos, log, exp ...). Bij Booleaanse problemen zoals een multiplexer of pariteitsprobleem kiest men eerder voor logische operatoren zoals (and, or, not ...). Maar ook voorwaardelijke operatoren (if-then-else ...), itererende functies (while, do-until ...), functies die recursie veroorzaken, en andere domeinspecifieke functies kunnen gedefinieerd worden. Terminalen of eindknopen zijn dikwijls variabelen (zoals invoervariabelen, toestandsvariabelen, sensoren, detectoren ...) of constanten. Het kunnen ook functies zijn die geen argumenten nemen, en die een invloed op de omgeving uitoefenen.

Om mogelijk te maken dat het GP-proces een aanvaardbare oplossing vindt voor een probleem, is het belangrijk om geschikte functies en te kiezen. In GP-terminologie noemt men dit vaak de *sufficiency* eigenschap. Ook al lijkt deze eigenschap een conditio sine qua non toch is het in de praktijk meestal niet eenvoudig om geschikte functies en terminalen te kiezen. Bijvoorbeeld, als we symbolische regressie willen uitvoeren, waarbij we een programma zoeken dat een wiskundig verband uitdrukt tussen een gegeven verzameling waarden, dan is het meestal niet op voorhand gekend welke wiskundige functies nodig zijn. Moeten de goniometrische functies in de functieset, of de logaritmische functies? Onnodige functies en onnodige terminalen opnemen in de sets kan een negatieve invloed uitoefenen op het snel vinden van een oplossing. Anderzijds, nodige functies of terminalen niet opnemen maakt het vinden van een oplossing onmogelijk.

Een tweede belangrijk criterium waar de functie- en terminalensets aan moeten voldoen, is sluiting (*closure*). Elke functie in de functieset moet elke mogelijke waarde voortgebracht door een functie of terminaal in de functie- en terminalenset kunnen aannemen als argumenten. Elke functie moet dus welgedefinieerd zijn voor elke combinatie van argumenten die ze mogelijkerwijs kan binnenkrijgen. Dit lijkt een grote beperking te zijn voor de expressieve vrijheid van deze genetische structuur, maar dat valt in de praktijk erg mee. Eenvoudige maatregelen voor een klein aantal uitzonderingsgeval-

len zijn over het algemeen voldoende om aan het principe van sluiting te voldoen. Bijvoorbeeld, als de functieset de delingsoperator omvat, moet het gedrag hiervan ook gedefinieerd zijn bij een deling door nul. De standaard maatregel hierbij is om een beschermde deling te gebruiken, waarbij het resultaat bij een deling door nul altijd één is (ook bij nul gedeeld door nul).

A.4.2 De uiteindelijke voorstelling van het genoom

Heel vaak worden individuen voorgesteld als boomstructuren maar er bestaan ook andere manieren. Nordin's idee om machine code te gebruiken bij de voorstelling van GP genomen was de meest radicale aanpak (Nordin & Francone 1999). Het concept werd een aantal keren aangepast (Nordin 1997) en leidde uiteindelijke tot Automatische Inductie van Machine code door Genetisch Programmeren (AIMGP) (Banzhaf *et al.* 1997). In AIMGP worden de verschillende genetische operatoren op de binaire machine code toegepast, zonder tussenkomst van een "interpreter" (vertaler) gedurende de bepaling van de fitheid. Dit resulteerde in een significante versnelling tegenover GP-simulatoren die wel een interpreter nodig hebben. Het nadeel is dat dergelijke systemen hardware afhankelijk zijn en minder overdraagbaar naar andere computerarchitecturen. Cramer (1985) gebruikt lineaire bitsequenties. Een meer algemene lineaire voorstelling werd geïntroduceerd door Banzhaf (1993). Brameier & Banzhaf (2001) gebruiken een variant van linear GP waarbij elk individu wordt voorgesteld als een string met variabele lengte met C-functies waarbij de aanwezigheid van overbodige code zeer eenvoudig en snel *at-runtime* kan gedetecteerd en verwijderd worden.

Een groot aantal onderzoekers maakt echter gebruik van genetisch programmeren waarbij de individuen m.b.v. een boom worden voorgesteld. In het vervolg van dit proefschrift maken we dan ook enkel gebruik van de boomvoorstelling en worden alle concepten en ideeën verklaard en geïllustreerd aan de hand hiervan.

A.4.3 Automatisch gedefinieerde functies

Vaak vertoont een oplossing voor een bepaald probleem heel wat symmetrie en kan men gelijkaardige stukken code herkennen in de boomstructuur. In plaats van de volledige oplossing onder te brengen in één enkele boom, zal elke bekwame programmeur eerst trachten de gelijkaardige stukken code te abstraheren en onder te brengen in bijvoorbeeld een afzonderlijke functie. Ook genetisch programmeren biedt hiervoor ondersteuning in de vorm van

automatisch gedefinieerde functies (ADF). We geven eerst een kort voorbeeld van de noodzaak aan een dergelijk mechanisme.

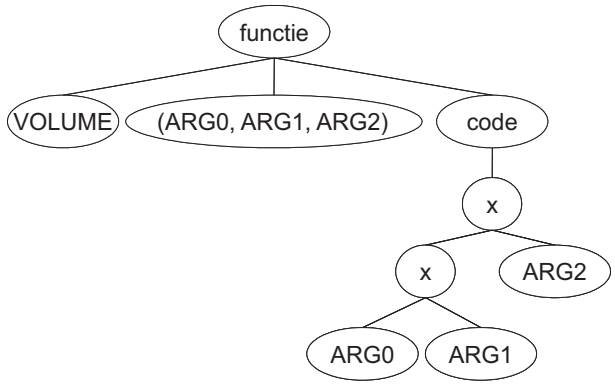
Een typisch voorbeeld dat door Koza werd gebruikt om het nut van automatische gedefinieerde functies aan te tonen is het "twee-dozen" probleem (Koza 1994). Het doel bestaat erin om het verschil in volume van twee dozen te berekenen: $D = L_0B_0H_0 - L_1B_1H_1$ waarbij respectievelijk L_i, B_i, H_i, de lengte, breedte en hoogte van doos i voorstelt. Genetisch programmeren zonder ADF zal de aanwezige symmetrie en het repetitief gedeelte (de bepaling van het volume) van de oplossing moeilijk herkennen. Het algoritme zal bijgevolg proberen om de getoonde invoerwaarden zo goed mogelijk af te beelden op de bijhorende uitvoerwaarden door middel van combinaties van verschillende elementaire rekenkundige functies zoals $+$, $-$, $\times$, $\div$. Een menselijke programmeur zal echter eerst een subroutine schrijven die het volume berekent. Pas dan zal hij beide volumes van elkaar aftrekken.

Bij genetisch programmeren met automatisch gedefinieerde functies wordt een individu niet voorgesteld door één maar meerdere bomen. Het hoofdprogramma (vergelijkbaar met het hoofdprogramma van een zelfgeschreven stuk programma) wordt vaak de resultaat-producerende boom (RPB) genoemd en geeft, na evaluatie, steeds een teruggeefwaarde (het resultaat) weer. De repetitieve delen in de code worden meestal ondergebracht in één of meerdere aparte functiebomen en kunnen geparametriseerd worden. Dit is evenwel geen verplichting. Deze functiebomen kunnen enkel aangeroepen worden vanuit de resultaat-producerende boom. Een functieboom voor het twee-dozen probleem uit bovenstaand voorbeeld ziet u in Figuur A.4.

Deze manier van werken, heeft een heleboel voordelen. Vooreerst biedt het ADF mechanisme de mogelijkheid om functies te definieren die herhaaldelijk vanuit de resultaat-producerende boom aangeroepen kunnen worden. Het is bijgevolg niet nodig om telkens opnieuw quasi dezelfde stukken code te herschrijven. Ten tweede zorgt ADF voor kleinere afmeting van de boomstructuur[4]. Een derde voordeel van ADF is de ondersteuning die inherent geboden wordt aan de functionele decompositie van een probleem in deelprobleem en een gestructureerde manier om functies aan te roepen. Tenslotte is het dankzij herbruikbare functiebomen overbodig om hetzelfde gedrag of concept opnieuw aan te leren wanneer dit elders nodig blijkt.

In GP met ADF's beschikt elk individu over zijn eigen functiebomen. Er wordt dus geen pool aangelegd die interessante functies bevat en die bruik-

[4]Het aantal functies dat effectief wordt uitgevoerd is echter wel hetzelfde. Toch vergroot het gebruik van ADF's de leesbaarheid van de oplossing.



Figuur A.4: *Een voorbeeld van een functieboom voor het twee-dozen probleem bij GP met ondersteuning voor automatisch gedefineerde functies. ARG0, ARG1 en ARG2 zijn drie argumenten die men met deze functie kan meegeven (lengte, breedte en hoogte).*

baar zijn door alle individuen. De functiebomen worden, net als de resultaatproducerende boom, op analoge manier gegenereerd en geëvolueerd (kruising, mutatie...).

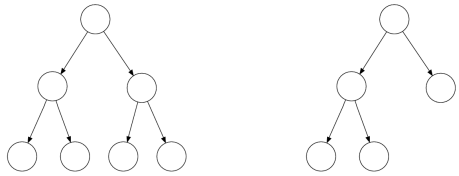
A.4.4 De initiële populatie

De initiële populatie bestaat uit willekeurig samengestelde individuen. Het construeren van een individu begint met het willekeurig kiezen van een functie voor de beginknoop. Voor elk van de argumenten van die beginknoop wordt vervolgens weer willekeurig een functie gekozen, en zo verder voor elke nieuw toegevoegde knoop. Is een gekozen functie een terminaal, dan vormt die een eindknoop. Er bestaan een aantal verschillende mogelijke implementaties om deze initiële populatie te construeren, die elk resulteren in een verschillende samenstelling van de populatie qua vorm en afmetingen.

Bij de *full* methode is in de uiteindelijke boom, elk pad tussen een eindknoop en de beginknoop gelijk aan een opgegeven maximale diepte. Deze voorwaarde wordt vervuld door bij het kiezen van functies voor de knopen die niet op die maximale diepte liggen geen terminalen toe te laten, en voor de andere knopen wel.

Bij de *grow* methode (groeimethode) worden bomen geconstrueerd waarbij elk pad tussen een eindknoop en de beginknoop niet groter is dan de opgegeven maximale diepte. Hierbij kunnen dus ook terminalen geselecteerd

worden voor knopen die niet op de maximale diepte liggen. Het resultaat van deze methode is een initiële populatie bestaande uit bomen met gevarieerde vormen en verschillende dieptes. Een voorbeeld van beide types bomen is te vinden in Figuur A.5.



Figuur A.5: *Mogelijke bomen uit de initiële populatie bij genetisch programmeren; de linkse boom is gevormd volgens de full methode, de rechtse volgens de grow methode.*

Beide methodes kunnen ook gecombineerd worden in een gemengde methode (*half-and-half*), waarbij de helft van de bomen met de *full* methode aangemaakt wordt en de andere helft met de *grow* methode. Verder is er ook nog het gebruik van een dieptehelling (*depth ramp*). Hierbij wordt een kleinste en een grootste waarde opgegeven voor de maximale diepte, en worden voor elke diepte tussen die kleinste en grootste waarde een even groot aantal bomen aangemaakt. Bij de *ramped half-and-half* methode tenslotte wordt bij dat even grote aantal bomen voor elke maximale diepte, telkens de helft via de *full* methode aangemaakt en de andere helft via de *grow* methode. Deze laatste methode, de *ramped half-and-half*, produceert de grootste verscheidenheid aan vormen en dieptes, en is dus de meest aangewezen methode om een grote structurele diversiteit te introduceren in de initiële populatie.

Bij genetisch programmeren zijn exact gelijke individuen in een populatie ongewenst, ze verspillen rekenkracht en geheugen, en verminderen de diversiteit van de populatie. Bij de initiële populatie zijn de bomen over het algmeen klein, en is de kans dus groter dat er duplicaten voorkomen. Daarom wordt bij het creëren van deze initiële populatie ook dikwijls gecontroleerd of er geen dubbele bomen in voorkomen, en worden eventuele dubbele bomen verwijderd en vervangen.

A.5 Bepalen van de fitheid

A.5.1 De fitness cases

Bij programma-inductie zoeken we een computerprogramma dat, wanneer het een bepaalde invoer krijgt, een bepaalde gewenste uitvoer genereert. Bij genetisch programmeren betekent dit dat elk programma uit de populatie moet uitgevoerd worden op één of meer vooraf bepaalde gevallen uit het probleemdomein, en dat daarna gekeken wordt hoe goed de gegenereerde uitvoer overeenstemt met de gewenste uitvoer. Daarbij is het dan vaak de bedoeling dat het uiteindelijke resultaat van de evolutie niet alleen deze bepaalde gevallen zal aankunnen, maar ook andere nog onbehandelde problemen uit hetzelfde domein. De vooraf bepaalde gevallen uit het probleemdomein waarop het programma getest wordt, noemen we de *fitness cases*, het uitvoeren van het programma op deze fitness cases is de evaluatie van het programma. Om inconsistentie te vermijden zullen we in de rest van dit proefschrift steeds deze terminologie hanteren.

A.5.2 Fitheid

Op basis van het uitvoeren van het programma op de fitness cases moet het een bepaalde waarde als fitheid toegewezen krijgen. Er zijn een aantal mogelijkheden om deze fitheid uit te drukken: zuivere fitheid, gestandaardizeerde fitheid, aangepaste fitheid, en genormeerde fitheid.

De basis wordt gevormd door de zuivere fitheid (*raw fitness*). Dat is de fitheid uitgedrukt in een maat die direct verband houdt met het probleemdomein waarvoor we een programma zoeken. Hierbij kan het zowel het geval zijn dat betere individuen een hogere zuivere fitheid hebben, als dat betere individuen een lagere zuivere fitheid hebben, afhankelijk van het probleemdomein. Bijvoorbeeld, bij het probleem van de kunstmatige mier zoeken we een automaat die zo veel mogelijke voedselkorreltjes vindt en opraapt in een bepaald gebied. Het aantal opgeraapte voedselkorreltjes is een voor de hand liggende keuze voor de zuivere fitheid, zodat hier geldt dat een hogere zuivere fitheid beter is. Dikwijls kiest men echter de fout tussen het doel en het effect van het programma als zuivere fitheid, zodat dan geldt dat een lagere zuivere fitheid beter is. Bij het regressieprobleem kan zo de som van de afwijkingen tussen de berekende en de werkelijke functiewaarden als zuivere fitheid gekozen worden.

De gestandaardizeerde fitheid (*standardized fitness*) wordt bekomen door de zuivere fitheid zodanig te herschrijven dat een lagere waarde altijd beter is en dat nul de best mogelijke waarde is. Zo is voor symbolische regressie de gestandaardizeerde fitheid gelijk aan de zuivere fitheid. Voor de kunstmatige mier echter moet de zuivere fitheid geïnverteerd worden, door ze af te trekken van het maximum aantal op te rapen voedselkorreltjes.

De volgende fitheidsvariant is de aangepaste fitheid (*adjusted fitness*). Als we de gestandaardizeerde fitheid f_s noemen, dan wordt de aangepaste fitheid f_a berekend als:

$$f_a = \frac{1}{1 + f_s}$$

De aangepaste fitheid ligt dus tussen nul en één, en hogere waarden duiden betere individuen aan. Een voordeel van deze fitheidsmaat is dat de kleine verschillen tussen goede en zeer goede individuen benadrukt worden.

De laatste fitheid is de genormeerde fitheid (*normalized fitness*), de genormeerde versie van de aangepaste fitheid, berekend als:

$$f_n = \frac{f_a}{S_{f_a}}$$

met S_{f_a} de som van de aangepaste fitheden van alle individuen in de populatie. Ook hier liggen fitheidswaarden tussen nul en één en zijn hogere waarden beter, maar bovendien zijn alle waarden genormeerd zodat hun som gelijk is aan één.

A.6 De voortplanting

A.6.1 Selectie

Bij het aanmaken van een nieuwe generatie individuen moeten individuen uit de vorige generatie geselecteerd worden om hun genetisch materiaal door te geven. Genetisch programmeren gebruikt hiervoor dezelfde selectiemethoden als genetische algoritmen. De standaard methode is de roulettewielselectie, waarbij de kans op selectie proportioneel is met de fitheidswaarde. Dit wordt vaak ook *fitness proportionate selection* (FPS) genoemd. Hierbij is de genormeerde fitheid zeer bruikbaar, de kans op selectie is dan immers gelijk aan deze genormeerde fitheid. Stel f_i gelijk aan de fitheid van één individu

en $\overline{f}$ de gemiddelde fitheid van de populatie ($=\frac{\sum_{i=1}^{N} f_i}{N}$). De kans waarmee een individu i wordt geselecteerd (p_i) is dan gelijk aan:

$$p_i = \frac{f_i}{\sum_{i=1}^{N} f_i} = \frac{f_i}{N\overline{f}}$$

Maar ook andere methoden zoals rangselectie en tornooiselectie kunnen worden toegepast. Bij tornooiselectie selecteert men op willekeurige basis een aantal individuen uit de populatie. Dit aantal (k) is een parameter die ingesteld moet worden. De geselecteerde individuen moeten zich met elkaar meten in een tornooi waar enkel het beste individu zal overleven. Een hoge waarde voor k, verkleint de kans dat minder goede individuen het tornooi overwinnen. De selectiedruk wordt groter.

Bij rangselectie worden alle individuen eerst gesorteerd op basis van fitheid. Vervolgens baseert het selectiemechanisme zich op de rang van het individu in plaats van de fitheidswaarde. Dit heeft het voordeel dat deze populatie niet gedomineerd kan worden door één zeer fit individu.

Tenslotte spreken we van elitisme wanneer de selectiemethode enkel het beste individu in beschouwing neemt. Dit wordt vaak gebruikt in combinatie met een reproductieoperator om ervoor te zorgen dat het beste individu steeds meegenomen wordt naar de volgende generatie.

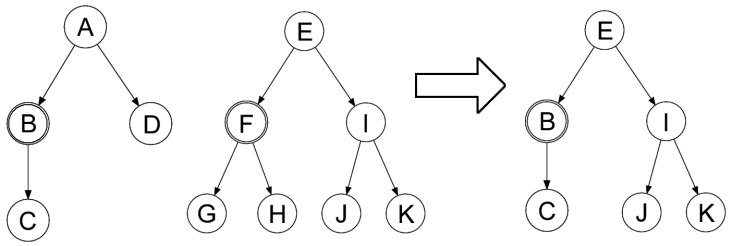
A.6.2 Reproductie

Dit is eigenlijk het basismechanisme achter *survival of the fittest*; de meest fitte kandidaat-oplossingen blijven leven, de minst fitte sterven af. De reproductieoperator kopieert een individu, geselecteerd met behulp van een selectiemethode gebaseerd op fitheid, ongewijzigd naar de volgende generatie. Dit is een aseksuele voortplantingsoperator, in die zin dat één enkele ouder één enkele nakomeling, namelijk een exacte kopie van zichzelf, voortbrengt.

A.6.3 Recombinatie

Kruising of recombinatie zorgt voor de verspreiding en voor het ontstaan van andere combinaties van succesvol genetisch materiaal. De kruisingsoperator neemt twee individuen als ouders, beiden geselecteerd volgens een bepaalde selectiemethode, en creëert twee nakomelingen die elk bestaan uit delen van beide ouders (Fig. A.6). Kruising is dus een seksuele operatie, in tegenstelling tot reproductie en mutatie.

De kruisingsoperatie begint met het willekeurig kiezen van een kruisingspunt in elk van de ouders. Dit kruisingspunt kan verschillend zijn in beide ouders! Vervolgens wordt de deelboom vertrekkend van het kruisingspunt van de eerste boom overgeplaatst naar het kruisingspunt van de tweede ouder, en omgekeerd. Merk op dat de ouders waarschijnlijk niet dezelfde vorm of afmetingen zullen hebben, en dat zulk een overgeplaatste deelboom soms slechts uit één terminale knoop kan bestaan.



Figuur A.6: *Recombinatie bij genetisch programmeren; de linkse twee bomen zijn de ouders, de dubbel omlijnde knopen B en F zijn hun respectievelijke kruisingspunten. De rechtse boom is één van beide nakomelingen.*

Het is ook mogelijk dat het selectiemechanisme tweemaal hetzelfde individu als ouders gekozen heeft voor één kruising. Bij genetische algoritmen levert de kruising van een individu met zichzelf geen gewijzigde individuen op, maar GP heeft het voordeel op GA dat hier wel twee nieuwe programma's uit de kruising van een programma met zichzelf kunnen voortkomen, doordat het kruisingspunt niet in beide ouders op dezelfde plaats moet liggen.

A.6.4 Een aantal secundaire operatoren

Vaak worden ook een aantal secundaire (soms probleemspecifieke) operatoren gebruikt. Een volledig overzicht geven van alle bestaande operatoren zou ons te ver leiden. Daarom lichten we enkel de belangrijkste methoden (in het opzicht van dit proefschrift) kort toe.

Een in GP vaak verguisde secundaire operator is mutatie. De werking van de mutatieoperator bestaat om te beginnen in het willekeurig kiezen van een mutatiepunt in de boom. Vervolgens verwijdert de operatie die knoop en al zijn eventuele deelbomen uit de boom, en brengt op het mutatiepunt een nieuwe,

willekeurig samengestelde deelboom in. Wanneer het mutatiepunt op een terminaal valt en de nieuw ingevoegde deelboom ook enkel uit een terminaal bestaat, zodat één terminaal vervangen is door een andere terminaal, spreken we van een puntmutatie.

Mutatie is, in tegenstelling tot reproductie en recombinatie, slechts een bijkomstige operator bij GP. Deze bijkomstigheid kan verklaard worden door de volgende twee bevindingen. Ten eerste is bij genetisch programmeren het aantal mogelijke functies en terminalen veel kleiner dan het aantal keren dat ze kunnen optreden in de populatie, zodat de kans dat één bepaalde functie of terminaal volledig uit de populatie verdwijnt heel klein is. Ten tweede, wanneer bij kruising beide gekozen kruisingspunten eindknopen van beide ouders zijn, dan is het effect van zulk een kruising heel gelijkaardig aan dat van puntmutatie, dus zorgt het bestaan van een kruisingsoperator impliciet ook voor mutatie.

Kinnear (1993*b*) definieerde een speciale kruisingsoperator waarbij slechts één individu wordt geselecteerd met behulp van roulettewielselectie. Binnen dit individu wordt op willekeurige basis een knoop geselecteerd. De deelboom met als ouderknoop dit geselecteerde punt wordt dan als het ware opgetakeld en vormt een nieuw individu. Deze operator diende om het aantal kleine individuen in de populatie te verhogen om codegroei van boomstructuren tegen te gaan. Deze operator is gelijkaardig aan de *collapse* operator van Janikow (1996). Beide operatoren gebruiken geen kennis van het probleemdomein (geen fitness cases) om de positie van de deelboom te bepalen.

In zijn eerste boek (Koza 1992), definieerde Koza een encapsulatieoperator die op een automatische wijze een nuttige deelboom binnen een individu selecteerde waarnaar men achteraf kon verwijzen. Deze operator is, net zoals de vorige twee operatoren, een asexuele operator en gebruikt bijgevolg slechts één ouder individu. De operator selecteert op willekeurige basis een functieknoop uit de boom waarna de geselecteerde deelboom wordt vervangen door een nieuwe functie. Deze nieuwe functie heeft geen argumenten en de code ervan wordt gevormd door de geselecteerde deelboom. Deze nieuwe functie zal de bestaande functieset uitbreiden zodat ook andere individuen van deze functie gebruik kunnen maken. Deze operator was de voorloper van ADF (Koza 1994).

Vooral reductieoperatoren zijn erg in trek bij genetisch programmeren. Ekart (1999) gebruikt een speciale mutatieoperator om overtollige en niet-geoptimaliseerde code te reduceren bij een regressieprobleem. Ook deze operator wordt gebruikt om codegroei te bestrijden.

A.7 Eindcriterium

In theorie zou een GP-proces, zoals de natuurlijke evolutie, onbeperkt in de tijd kunnen blijven voortgaan. In de praktijk blijft het uiteraard wel nodig het proces ooit stil te leggen. Een mogelijk eindcriterium zou zijn de evolutie te stoppen eens een volledig correcte oplossing gevonden is. Een andere mogelijkheid is op voorhand een maximum aantal generaties vast te leggen en de evolutie te stoppen wanneer dat aantal bereikt is. Natuurlijk kunnen we ook een combinatie nemen, namelijk stoppen bij het eerst optredende van beide eindcriteria. Vaak is het niet a priori mogelijk om de eigenschappen van een volledig correcte oplossing te bepalen, of verwachten we niet ooit een volledig correcte oplossing te bekomen. In zo'n gevallen is een maximum aantal generaties een bruikbaar alternatief.

A.8 Lil-gp

Gelukkig bestaan er een heleboel kant-en-klare softwarepakketten die vrij beschikbaar zijn (Punch & Zongker 1996, Koza 1992, Yu & Clack 1998, 1997). Bovendien bestaan GP-simulatoren in een waaier van programmeertalen zoals LISP, C, C++, Java, en zelfs een rechtsreeks in de hardware (Draxton Labs) en via Xilinx FPGA's (Martin 2002). In dit proefschrift werd gekozen voor een implementatie die snel werkt en op een efficiënte manier gebruik maakt van het geheugen. Met efficiënt bedoelen we dat de software zo weinig mogelijk geheugen gebruikt om de boomstructuren van de individuen op te slaan. Beide eisen gecombineerd met mijn persoonlijke programmeerervaring in de programmeertaal C, hebben geleid tot de keuze voor het softwarepakket lil-gp (Punch & Zongker 1996).

Lil-gp is een freeware C-programma, oorspronkelijk ontwikkeld en geschreven door Douglas Zongker en Dr. Bill Punch. De voordelen van dit programma zijn onder andere de snelheid waarmee de simulatie uitgevoerd wordt, de overdraagbaarheid van het programma naar verschillende software-platforms (Unix, Windows, Mac, ...) en het feit dat er een diverse selectie van bruikbare GP elementen in opgenomen is. Deze laatste omvatten bijvoorbeeld een handige implementatie van constanten, de mogelijkheid om met meerdere populaties te werken, om automatisch gedefinieerde functies te gebruiken en om resultaten uit te wisselen met andere GP-simulatoren in de vorm van de gestandaardizeerde eenvoudige LISP code uit de appendices van Koza's ori-

ginele boek (Koza 1992). Verder bevat deze simulator broncode van maar liefst vijf verschillende standaardtoepassingen.

Het programma bestaat uit twee onderdelen: enerzijds de kernel, die het eigenlijke GP algoritme verzorgt, anderzijds de toepassingen, die probleemspecifieke implementaties bevatten.

De kernel is opgebouwd rond een efficiënte boomvoorstelling: de programma's worden niet als expliciet gelinkte bomen bijgehouden, maar als arrays. De arrays bevatten de knopen in preorde, de boom kan dus uit de array afgeleid worden op basis van de volgorde van de knopen en het aantal kinderen van elke knoop. Deze compacte array-voorstelling laat toe om snel door de knopen te navigeren, maakt het mogelijk om grotere populaties in het geheugen op te slaan en beperkt het aantal wisselingen van geheugenpagina's, wat uiteraard allemaal de performantie ten goede komt. De in de arrays opgeslagen knopen zijn pointers naar C-functies, wat betekent dat de evaluatie van de programma's volledig met gecompileerde code gebeurt. Dit brengt ook weer een zeer grote snelheidswinst met zich mee. Op het implementatieniveau zijn de functieset en de terminalenset samengenomen tot één verzameling: er is in de voorstellingswijze geen noemenswaardig verschil tussen een functie en een terminaal, een terminaal is eenvoudigweg een functie zonder argumenten.

Het GP algoritme kan geconfigureerd worden met preprocessor-gedefinieerde constanten, met ingelezen parameterbestanden of met commandolijn-argumenten. De laatste twee mogelijkheden hebben het voordeel dat het programma maar één keer moet gecompileerd worden en dan toch met verschillende parameters kan uitgevoerd worden. Er is een zeer uitgebreid gamma aan instelbare eigenschappen, waarvan een overzicht in de lil-gp handleiding kan teruggevonden worden. De parameters controleren onder andere voor de hand liggende zaken zoals een eventuele groottelimiet, het aantal generaties en de grootte van een populatie, maar ook meer in het algoritme ingrijpende instellingen zoals het aantal populaties, de keuze en het gebruik van selectie-operatoren en van de voortplantingsoperatoren. Een gedetailleerde beschrijving van deze parameterinstellingen volgt in de onderstaande paragraaf.

Naast de kernel omvat lil-gp ook een aantal standaardtoepassingen, zoals het regressieprobleem of het probleem van de kunstmatige mier. Om zulke toepassingen voor lil-gp te schrijven is slechts een minimum aan programmeerwerk vereist. Een C-procedure voor elke functie en terminaal die gebruikt wordt voor het probleem in kwestie, en een fitheidsfunctie die vastlegt hoe de fitheid berekend wordt op basis van de evaluatie van een individu, is vol-

doende. Verder zijn een aantal prototypes voorzien voor functies die op bepaalde momenten in de evolutie opgeroepen worden, en die kunnen dienen voor initialisatie van de toepassing, aangepaste uitvoer of controle van de eindvoorwaarde.

Bijlage B

Het parameter bestand van het FIS

Deze bijlage geeft een overzicht van alle parameterinstellingen zoals gebruikt door het FIS.

```
[System]
Name='LOSE1'
Type='mamdani'
Version=2.0
NumInputs=2
NumOutputs=1
NumRules=9
AndMethod='min'
OrMethod='max'
ImpMethod='min'
AggMethod='max'
DefuzzMethod='centroid'

[Input1]
Name='size.increase'
Range=[0 20]
NumMFs=3
MF1='klein':'trapmf',[0 0 5 8]
MF2='gemiddeld':'trimf',[5 10 15]
MF3='groot':'trimf',[10 20 20]
```

```
[Input2]
Name='bfit.increase'
Range=[0 0.01]
NumMFs=3
MF1='klein':'trimf',[0 0 0.002]
MF2='gemiddeld':'trimf',[0.001
0.0025 0.004]
MF3='hoog':'trapmf',[0.003 0.007 0.01 0.01]

[Output1]
Name='LOSE.rate'
Range=[0 1]
NumMFs=3
MF1='klein':'trimf',[0 0 0.3]
MF2='gemiddeld':'trimf',[0.2 0.45
0.7] MF3='groot':'trapmf',[0.6 0.9 1 1]

[Rules]
1 1, 2 (1) : 1

1 2, 1 (1) : 1

1 3, 1 (1) : 1

2 1, 2 (1) : 1

2 2, 2 (1) : 1

2 3, 1 (1) : 1

3 1, 3 (1) : 1

3 2, 3 (1) : 1

3 3, 2 (1) : 1
```

Bijlage C

Afkortingen, vertalingen en symbolen

Deze bijlage geeft een overzicht van de gebruikte afkortingen en symbolen alsook een aantal vertalingen van gekende Engelstalige begrippen uit het onderzoeksdomein.

C.1 Afkortingen

A-life	artificial life, kunstmatig leven
ADATE	automatic design of algorithms through evolution
ADF	automatische gedefinieerde functie
AIMGP	automatische inductie van machine code door genetisch programmeren
BFS	breedte-eerst zoeken
BSS	beste deelboomselectie
DNA	desoxyribonucleïnezuur
EA	evolutionair algoritme
EP	evolutionair programmeren
ERC	ephemerische willekeurige constante
ES	evolutiestrategieën
FIFO	first in, first out
FuLOSE	fuzzy LOSE
FLC	vaaglogische regelaar
FPGA	field programmable gate array
FPS	roulettewielselectie

FSM	finite state machines, eindige toestandsautomaten
GA	genetisch algoritme
GEMGA	gene expression messy genetic algorithm
GP	genetisch programmeren
GNARL	generalized acquisition of recurrent links
iBFS	omgekeerd breedte-eerst zoeken
LOSE	lokale optimalisatieoperator
LISP	programmeertaal, list processor
MDL	minimum description length
POST	post-orde wandeling
PRE	pre-orde wandeling
RNA	ribonucleïnezuur
SA	simulated annealing
RPB	resultaat producerende boom

C.2 Vertalingen

adjusted fitness	aangepaste fitheid
at runtime	tijdens de uitvoering van het algoritme
array	rij van objecten
building block	bouwblok
closure	sluiting
code bloat, code growth	codegroei, een ongecontroleerde toename van de boomgrootte (of boomdiepte)
constant parsimony pressure	constante onderdrukking van de boomgrootte d.m.v. penalisatiefactor
cutoff value	afknijpwaarde
depth ramp	dieptehelling
edit distance	vormafstand
ephemeral random constant	ephemerische willekeurige constante
fitness	fitheid
fitness sharing	fitheidsdeling
full method	methode om een boom op te bouwen (alle eindknopen liggen op dezelfde diepte)
function node	functieknoop, functie
fuzzy logic	vaaglogica
grow method	methode om boom op te bouwen (probabilistische knoopselectie)

half-and-half	helft volgens methode 1 en andere helft volgens methode 2
hitchhiking	liften
hill climbing	iteratieve éénpunts zoekmethode die steeds dichtsbijzijnde volgende knoop kiest
interpreter	vertaler
intron(s)	intronen
inviable code	niet-levensvatbare code
knowledge base	kennisbank
leaf node	eindknoop
linkage problem	koppelingsprobleem
lnode	knoop zoals voorgesteld in lil-gp
normalized fitness	genormeerde fitheid
objective function	doelfunctie
parameter control	parametersturing
parameter tuning	(handmatig) afstellen van parameterinstellingen
random search	willekeurig zoeken
rank selection	rangselectie
raw fitness	zuivere fitheid
removal bias	verhoogde kans op het verwijderen van een kleine deelboom
root node	beginknoop, wortel
roulette wheel selection	roulettewielselectie
run	simulatie
seed	initiële waarde
standardized fitness	gestandardizeerde fitheid
subtree	deelboom
sufficiency	eigenschap dat terminalen en functieverzameling voldoende informatie bevat om probleem op te lossen
terminal node	eindknoop
tree depth	boomdiepte, de lengte van het pad van de wortel van de boom (diepte 0) tot de diepste eindknoop
tree size	boomgrootte, het aantal knopen (functies en eindknopen) in de boom
trial and error	uitproberen van een aantal waarden
fitness case	leervoorbeeld
tournament selection	tornooiselectie
wrapper	programma dat een ander programma omwikkelt

C.3 Symbolen

Δ	verschil tussen twee parameters
f_i	fitheid van een individu
$\overline{f}$	gemiddelde fitheid van de populatie
ΔF_{max}	verschil in maximum fitheid tussen huidige en vorige generatie
$F_{\text{subs}}[]$	rij met fitheidswaarden van deelbomen
p_i	kans waarmee individu i wordt geselecteerd
p_{kruising}	probabiliteit waarmee kruising wordt toegepast
p_{lose}	probabiliteit waarmee LOSE wordt toegepast
p_{mutatie}	probabiliteit waarmee mutatie wordt toegepast
$I(M, i, z)$	de benodigde computationele rekenkracht

Bibliografie

Andre, D. & Teller, A. (1996), "A Study in Program Response and the Negative Effects of Introns in Genetic Programming", in : J. Koza, D. Goldberg & D. F. R. Riolo, eds., Genetic Programming 1996: Proceedings of the First Annual Conference, 12–20, MIT Press, Kinsale, Ireland.

Angeline, P. (1994), "Genetic programming and emergent intelligence", Advances in genetic programming I, MIT Press, Cambridge, Massachusetts, hoofdstuk 4, 75–97.

Angeline, P. (1998), "Subtree crossover causes bloat", in : J. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. Fogel, M. Garzon, D. Goldberg, H. Iba & R. Riolo, eds., Genetic Programming 1998: Proceedings of the Third Annual Conference, 745–752, Morgan Kaufmann.

Angeline, P. & Pollack, J. (1993*a*), "Competitive Environments Evolve Better Solutions for Complex Tasks", in : S. Forrest, ed., Proceedings of the 5th International Conference on Genetic Algorithms, 264–270, Morgan Kauffman.

Angeline, P. J. & Pollack, J. B. (1993*b*), "Coevolving High-Level Representations", July Technical report 92-PA-COEVOLVE, Laboratory for Artificial Intelligence. The Ohio State University.

Bagley, J. (1967), "The behavior of adaptive systems which employ genetic and correlation algorithms", proefschrift, University of Michigan, Michigan, USA.

Banzhaf, W. (1993), "Genetic programming for pedestrians", in : S. Forrest, ed., Proceedings of the fifth international conference on genetic algorithms, 628, Morgan Kaufmann.

Banzhaf, W., Francone, F., Keller, R. & Nordin, P. (1997), "Genetic Programming: An Introduction: On the Automatic Evolution of Computer

Programs and Its Applications", Morgan Kaufmann, San Francisco, California, USA.

Banzhaf, W. & Langdon, W. (2002), "Some considerations on the reasons for bloat", Genetic programming and evolvable machines, 3 (1), 81–91.

Bellman, R. (1961), "Adaptive control processes", Princeton University Press, Pinceton, NJ.

Besetti, S. & Soule, T. (2005), "Function choice, resiliency and growth in genetic programming", in : H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, W. Banzhaf, C. Blum, E. W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G. R. Raidl, T. Soule, A. M. Tyrrell, J.-P. Watson & E. Zitzler, eds., GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, 2, 1771–1772, ACM Press, Washington DC, USA.

Bickle, T. & Thiele, L. (1994), "Genetic Programming and redundancy", in : J. Hopf, ed., Genetic algorithms within the framework of evolutionary computation (Workshop at KI-94n Saarbrücken), 33–38.

Bleuler, S., Brack, M., Thiele, L. & Zitzler, E. (2001), "Multiobjective Genetic Programming: Reducing Bloat Using SPEA2", in : Proceedings of the IEEE 2001 Congress on Evolutionary Computation, 536–543, IEEE.

Blickle, T. (1996*a*), "Evolving compact solutions in genetic programming: a case study", in : H.-M. Voigt, W. Ebeling, I. Rechenberg & H.-P. Schwefel, eds., Parallel Problem Solving from Nature-PPSNIV, Lecture Notes in Computer Science, 1141, 564–573, Springer-Verlag.

Blickle, T. (1996*b*), "Theory of Evolutionary Algorithms and Application to System Synthesis", proefschrift, Swiss Federal Institute of Technology, Zurich, Switzerland.

Brameier, M. & Banzhaf, W. (2001), "A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining", IEEE Transactions on Evolutionary Computation, 5 (1), 17–26.

Brindle, A. (1981), "Genetic algorithms for function optimization", proefschrift, University of Alberta, Edmonton.

Browne, D. (1996), "Vision-Based Obstacle Avoidance: A Coevolutionary Approach", .

Burke, E., Gustafson, S. & Kendall, G. (2002*a*), "A survey and analysis of diversity measures in genetic programming", in : W. Langdon, E. Cantu-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan & V. Honavar, eds., Proceedings of the Genetic and Evolutionary Computation Conference, 716–723, Morgan Kaufmann.

Burke, E., Gustafson, S. & Kendall, G. (2004), "Diversity in Genetic Programming: An Analysis of Measures and Correlation with Fitness", IEEE Transactions on Evolutionary Computation, 8 (1), 47–62.

Burke, E., Gustafson, S., Kendall, G. & Krasnogor, N. (2002*b*), "Advanced Population Diversity Measures in Genetic Programming", in : J. M. Guervos, P. Adamidis, H.-G. Beyer, J.-L. Fernandez-Villacanas & H.-P. Schwefel, eds., Parallel Problem Solving from Nature-PPSNVII, Lecture Notes in Computer Science, 2439, 341–350, Springer-Verlag.

Chellapilla, K. (1998), "Evolving Computer Programs without Subtree Crossover", IEEE Transactions on Evolutionary Computation, 1 (3), 209–216.

Chong, F. S. (1999), "Java based Distributed Genetic Programming on the Internet", in : E. Cantu-Paz & B. Punch, eds., Evolutionary computation and parallel processing, 163–166, Orlando, Florida, USA.

Chongstitvatana, P. (1998), "Improving robustness of robot programs generated by genetic programming for dynamic environments", in : Proceedings of the 1998 IEEE Asia-Pacific Conference on Circuits and Systems, Microelectronics and Integrating Systems, 523–526, IEEE Press.

Chongstitvatana, P. (1999), "Using Perturbation to Improve Robustness of Solutions Generated by Genetic Programming for Robot Learning", Journal of Circuits, Systems, and Computers, 9, 133–143.

Cordon, O., Herrera, F., Hoffman, F. & Magdalena, L. (2001), "Genetic Fuzzy Systems: Evolutionary Tuning and Learning of Fuzzy Knowledge Bases", World Scientific.

Cramer, N. (1985), "A representation for the adaptive generation of simple sequential programs", in : J. Grefenstette, ed., Proceedings of the First International Conference on Genetic Algorithms, 183–187, Lawrence Erlbaum Associates.

Daida, J. M., Bertram, R. R., Stanhope, S. A., Khoo, J. C., Chaudhary, S. A., Chaudhri, O. A. & Polito II, J. A. (2001), "What makes a problem GP-hard? analysis of a tunably difficult problem in genetic programming", Genetic Programming and Evolvable Machines, 2 (2), 165–191.

Dalci, K., Uzunoglu, M. & Kucukdemiral, I. (2004), "Genetic algorithm based optimal self-tuning fuzzy logic controller for power system static VAR stabiliser", International Journal of Electrical Engineering Education, 41 (1), 71–89.

Darwin, C. (1959), "The origin of species", John Murray, London, UK.

de Jong, E. & Pollack, J. (2003), "Multi-Objective Methods for Tree Size Control", Genetic Programming and Evolvable Machines, 4 (3), 211–233.

De Jong, E., Watson, R. & Pollack, J. (2001), "Reducing Bloat and Promoting Diversity using Multi-Objective Methods", in : L. Spector, E. Goodman, A. Wu, W. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. Garzon & E. Burke, eds., Proceedings of the Genetic and Evolutionary Computation Conference, 11–18.

De Smedt, P. (2006), "Generalisatievermogen van kandidaat-oplossingen bij genetisch programmeren", scriptie, Ghent University.

Deb, K. (2002), "Multi-objective Optimization using Evolutionary Algorithms", John Wiley & Sons, New York, USA.

Draxton Labs ().
URL: *http://www.daxtron.com/*

Driankow, D., Hellendoorn, H. & Reinfrank, M. (1993), "An introduction to fuzzy control", Springer-Verlag, Berlin.

Eiben, A., Hinterding, R. & Michalewicz, Z. (1999), "Parameter Control in Evolutionary Algorithms", IEEE Transactions on Evolutionary Computation, 3 (2), 124–141.

Ekart, A. (1999), "Controlling code growth in genetic programming by mutation", in : W. Langdon, R. Poli, P. Nordin & T. Fogarty, eds., Late breaking papers of Euro-GP 99, 3–12.

Ekárt, A. & Németh, S. (2000), "A metric for genetic programs and fitness sharing", in : R. Poli, W. Banzhaf, W. Langdon, J. Miller, P. Nordin

& T. Fogarty, eds., Genetic programming, Proceedings of the 3rd European Conference, Lecture Notes in Computer Science, 1802, 259–270, Springer-Verlag.

Ekárt, A. & Németh, S. (2001), "Selection based on the pareto nondominance criterion for controlling code growth in genetic programming", Genetic Programming and Evolvable Machines, 2 (1), 61–73.

Esparcia Alcazar, A. I. & Sharman, K. C. (1996), "Some Applications of Genetic Programming in Digital Signal Processing", in : J. R. Koza, ed., Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996, 24–31, Stanford Bookstore.

Fernandez, F. & Martin, A. (2004), "Saving Effort in Parallel GP by means of Plagues", in : M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa & T. Soule, eds., Genetic Programming 7th European Conference, EuroGP 2004, Proceedings, LNCS, 3003, 269–278, Springer-Verlag, Coimbra, Portugal.

Fernandez, F., Spezzano, G., Tomassini, M. & Vanneschi, L. (2005), "Parallel Genetic Programming", in : E. Alba, ed., Parallel Metaheuristics, Parallel and Distributed Computing, hoofdstuk 6, 127–153, Wiley-Interscience, Hoboken, New Jersey, USA.

Fernandez, F., Vanneschi, L. & Tomassini, M. (2003), "The Effect of Plagues in Genetic Programming: A Study of Variable-Size Populations", in : C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli & E. Costa, eds., Genetic Programming, Proceedings of EuroGP'2003, LNCS, 2610, 317–326, Springer-Verlag, Essex.

Folino, G., Pizzuti, C. & Spezzano, G. (2001), "CAGE: A Tool for Parallel Genetic Programming Applications", in : J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi & W. B. Langdon, eds., Genetic Programming, Proceedings of EuroGP'2001, LNCS, 2038, 64–73, Springer-Verlag, Lake Como, Italy.

Fonseca, C. & Fleming, P. (1993), "Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization", in : S. Forrest, ed., Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann.

Fonseca, C. & Fleming, P. (1995), "An overview of evolutionary algorithms in multiobjective optimization", Evolutionary Computation, 3 (1), 1–16.

Fullmer, B. & Miikkulainen, R. (1992), "Using marker-based genetic encoding of neural networks to evolve finite-state behavior", in : F. J. Varela & P. Bourgine, eds., Proceedings of the First European Conference on Artificial Life, 255–262 310–314, MIT Press.

Gathercole, C. & Ross, P. (1994), "Dynamic Training Subset Selection for Supervised Learning in Genetic Programming", in : Y. Davidor, H.-P. Schwefel & R. Manner, eds., Parallel Problem Solving from Nature-PPSNIII, Lecture Notes in Computer Science, 866, 312–321, Springer-Verlag.

Gathercole, C. & Ross, P. (1997), "Tackling the Boolean Even N Parity Problem with Genetic Programming and Limited-Error Fitness", in : J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba & R. Riolo, eds., Genetic Programming 1997: Proceedings of the Second Annual Conference, 119–127.

Gelly, S., Teytaud, O., Bredeche, N. & Schoenauer, M. (2005), "A Statistical Learning Theory Approach of Bloat", in : H.-G. Beyer, ed., Proceedings of the 2005 conference on Genetic and evolutionary computation, 1783–1784.

Goldberg, D. (1989), "Genetic Algorithms in Search, Optimization, and Machine Learning", Addison-Wesley, Reading, Massachusetts.

Goldberg, D., Deb, K., Kargupta, H. & Harik, G. (1993), "Rapid, accuate optimization of difficult problems using fast messy genetic algorithms", in : Proceedings of the Fifth Annual International Conference of Genetic Algorithms, 56–64.

Goldberg, D. & Smith, R. (1987), "Nonstationary function optimization using genetic algorithms with dominance and diploidy", in : J. Grefenstette, ed., Proceedings of the second international conference on genetic algorithms, 59–68.

Greene, F. (1994), "A method for utilizing diploid and dominance in genetic search", in : Proceedings of the first IEEE conference on evolutionary compuation, 439–444, IEEE Press.

Greenwood, G. W. & Tyrrell, A. M. (2006), "Introduction to Evolvable Hardware: A Practical Guide for Designing Self-Adaptive Systems", Wiley-IEEE Press.

Gruau, F. (1992), "Genetic synthesis of boolean neural networks with a cell rewriting developmental process", in : J. Schaffer & D. Whitley, eds., Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks, 2, 55–74, IEEE Press.

Handley, S. (1994), "On the use of a directed acyclic graph to represent a population of computer programs", in : Proceedings of the 1994 IEEE World Congress on Computational Intelligence, 154–159, IEEE Press.

Harries, K. & Smith, P. W. H. (1998), "Code Growth, Explicitly Defined Introns and Alternative Selection Schemes", .

Haynes, T. (1998), "Collective adaptation: The exchange of coding segments", Evolutionary Computation, 6 (4), 311–338.

Heywood, M. I. & Zincir-Heywood, A. N. (2000), "Register Based Genetic Programming on FPGA Computing Platforms", in : R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin & T. C. Fogarty, eds., Genetic Programming, Proceedings of EuroGP'2000, LNCS, 1802, 44–59, Springer-Verlag, Edinburgh.

Higgins, J. (2003), "Introduction to Modern Nonparametric Statistics", Duxburry Press.

Hinterding, R. (1995), "Gaussian utation and self-adaptation in numeric genetic algorithms", in : Proceedings of the 2nd IEEE conference on evolutionary computation, 384–389, IEEE Press.

Hinterding, R. (1997), "Self-adaptation using multi-chromosomes", in : Proceedings of the 4th IEEE conference on evolutionary computation, 87–91, IEEE Press.

Holland, J. (1975), "Adaptation in Natural and Artificial Systems", MIT Press, Cambridge, Massachusetts.

Hooper, D. & Flann, N. (1996), "Improving the Accuracy and Robustness of Genetic Programming through Expression Simplification", in : J. Koza, D. Goldberg, D. Fogel & R. Riolo, eds., Genetic Programming 1996: Proceedings of the First Annual Conference, 28–31.

Iba, H., de Garis, H. & Sato, T. (1994), "Genetic Programming Using a Minimum Description Length Principle", Advances in Genetic Programming I, MIT Press, Cambridge, Massachusetts, hoofdstuk 12, 265–284.

Igel, C. & Chellapilla, K. (1999), "Investigating the Influence of Depth and Degree of Genotypic Change on Fitness in Genetic Programming", in : W. Banzhaf, J. Daida, A. Eiben, M. Garzon, V. Honavar, M. Jakiela & R. Smith, eds., Proceedings of the genetic and evolutionary computation conference, 2, 1061–1068, Morgan Kaufmann.

Janikow, C. (1996), "A methodology for processing problem constraints in genetic programming", Computers and Mathematics with Applications, 32 (8), 97–113.

Kargupta, H. (1996), "The gene expression messy genetic algorithm", in : Proceedings of the IEEE International Conference on Evolutionary Computation, 814–819.

Keijzer, M. (1996), "Efficiently representing populations in genetic programming", Advances in Genetic Programming II, MIT Press, Cambridge, Massachusetts, hoofdstuk 13, 259–278.

Kinnear, K. (1993*a*), "Evolving a Sort: Lessons in Genetic Programming", in : Proceedings of the 1993 International Conference on Neural Networks, 881–888.

Kinnear, K. (1993*b*), "Generality and difficulty in genetic programming: Evolving a sort", in : S. Forrest, ed., Proceedings of the Fifth international conference on genetic algorithms, 287–294, Morgan Kaufmann.

Kinnear, Jr., K. E. (1994), "Alternatives in Automatic Function Definition: A Comparison Of Performance", in : K. E. Kinnear, Jr., ed., Advances in Genetic Programming, hoofdstuk 6, 119–141, MIT Press.

Kirkpatrick, S., Gelatt, Jr., C. & Vecchi, M. (1983), "Optimisation by Simulated Annealing", Science, 220 (4598), 671–680.

Koza, J. (1992), "Genetic Programming: On the programming of computers by Means of natural selection", MIT Press, Cambridge, Massachusetts.

Koza, J. (1994), "Genetic Programming II: Automatic discovery of reusable programs", MIT Press, Cambridge, Massachusetts.

Koza, J. R., Bennett III, F. H., Hutchings, J. L., Bade, S. L., Keane, M. A. & Andre, D. (1998), "Evolving Computer Programs using Rapidly Reconfigurable FPGAs and Genetic Programming", in : J. Cong, ed., FPGA'98 Sixth International Symposium on Field Programmable Gate Arrays, 209–219, ACM Press, Doubletree Hotel, Monterey, California, USA.

Kuschchu, I. (2002), "Genetic Programming and Evolutionary Generalisation", IEEE Transactions on Evolutionary Computation, 6 (5), 431–442.

Kushchu, I. (2002), "An Evaluation of Evolutionary Generalisation in Genetic Programming", Artif. Intell. Rev., 18 (1), 3–14.

Langdon, W. (1998*a*), "Data Structures and Genetic Programming: genetic programming + data structures = Automatic programming!", Kluwer, Boston.

Langdon, W. (1998*b*), "The evolution of size in variable length representations", in : 1998 IEEE International Conference on Evolutionary Computation, 633–638, IEEE Press.

Langdon, W. (1999), "Size fair and homologous tree genetic programming crossovers", in : W. Banzhaf, J. Daida, A. Eiben, M. Garzon, V. Honavar, M. Jakiela & R. Smith, eds., Proceedings of the genetic and evolutionary computation conference, 2, 1092–1097, Morgan Kaufmann.

Langdon, W. & Poli, R. (1997*a*), "Fitness Causes Bloat", in : P. Chawdhry, R. Roy & R. Pant, eds., Soft Computing in Engineering Design and Manufacturing, 13–22, Springer-Verlag.

Langdon, W. & Poli, R. (1997*b*), "Fitness causes bloat: mutation", in : J. Koza, ed., Late breaking papers at the GP-97 conference, 132–140, Stanford Bookstore.

Langdon, W. & Poli, R. (1998*a*), "Fitness causes bloat: Mutation", in : W. Banzhaf, R. Poli, M. Schoenauer & T. C. Fogarty, eds., Proceedings of the First European Workshop on Genetic Programming, Lecture Notes in Computer Science, 1391, 37–48, Springer-Verlag.

Langdon, W. & Poli, R. (1998*b*), "Why Ants are Hard", in : J. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. Fogel, M. Garzon, D. Goldberg, H. Iba & R. Riolo, eds., Genetic Programming 1998: Proceedings of the Third Annual Conference, 193–201.

Langdon, W., Soule, T., Poli, R. & Foster, J. (1999), "The evolution of size and shape", Advances in genetic programming III, MIT Press, Cambridge, Massachusetts, hoofdstuk 8, 163–190.

Langdon, W. B. (1995), "Evolving Data Structures Using Genetic Programming", in : L. J. Eshelman, ed., Genetic Algorithms: Proceedings of the

Sixth International Conference (ICGA95), 295–302, Morgan Kaufmann, Pittsburgh, PA, USA.

Langdon, W. B. (2000), "Quadratic bloat in genetic programming", in : D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee & H.-G. Beyer, eds., GECCO2000: Proceedings of the genetic and evolutionary computation conference, 451–458, Morgan Kaufmann.

Langdon, W. B. & Nordin, J. P. (2000), "Seeding GP Populations", in : R. Poli, W. Banzhaf, W. Langdon, J. Miller, P. Nordin & T. Fogarty, eds., Genetic Programming, Proceedings of EuroGP'2000, Lecture Notes in Computer Science, 1802, 304–315, Springer-Verlag.

Langdon, W. B. & Poli, R. (1998*c*), "Genetic Programming Bloat with Dynamic Fitness", in : W. Banzhaf, R. Poli, M. Schoenauer & T. Fogarty, eds., Proceedings of the First European Workshop on Genetic Programming, Lecture Notes in Computer Science, 1391, 96–112, Springer-Verlag.

Luke, S. (2000*a*), "Code growth is not caused by introns", in : D. Whitley, ed., Late Breaking Papers of the 2000 Genetic and Evolutionary Computation Conference, 228–235.

Luke, S. (2000*b*), "Issues in Scaling genetic programming: breeding strategies, tree generation and code bloat", proefschrift, University of Maryland, , USA.

Luke, S. (2003), "Modification Point Depth and Genome Growth in Genetic Programming", Evolutionary Computation, 11 (1), 67–106.

Luke, S. & Panait, L. (2002), "Fighting Bloat with Nonparametric Parsimony Pressure", in : J. M. Guervos, P. Adamidis, H.-G. Beyer, J.-L. Fernandez-Villacanas & H.-P. Schwefel, eds., Parallel Problem Solving from Nature-PPSNVII, Lecture Notes in Computer Science, 2439, 411–421, Springer-Verlag.

Luke, S. & Panait, L. (2006), "A Comparison of Bloat Control Methods for Genetic Programming", Evolutionary Computation, 14 (3), 309–344.

Mahler, S., Robilliard, D. & Fonlupt, C. (2005), "Tarpeian Bloat Control and Generalization Accuracy", in : M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert & M. Tomassini, eds., Proceedings of the 8th European Conference on Genetic Programming, Lecture Notes in Computer Science, 3447, 203–214, Springer, Lausanne, Switzerland.

Martin, P. (2002), "Genetic Programming in Hardware", proefschrift, University of Essex, Essex, UK.

McPhee, N. & Hopper, N. (1999), "Analysis of genetic diversity through population history", in : W. Banzhaf, ed., Proceedings of the Genetic and Evolutionary Computation Conference, 1112–1120, Morgan Kaufmann.

McPhee, N. & Miller, J. (1995), "Accurate replication in genetic programming", in : L. Eshelman, ed., Proceedings of the Sixth International conference on genetic algorithms, 303–309, Morgan Kauffman.

Mitchell, M. (1996), "An introduction to genetic algorithms", MIT Press.

Moore, F. & Garcia, O. (1997), "New Methodology for Reducing Brittleness in Genetic Programming", in : E. Pohl, ed., Proceedings of the National Aerospace and Electronics 1997 Conference, 2, 757–763, IEEE Press.

Nordin, P. (1997), "Evolutionary Program Induction of Binary Machine Code and its Application", proefschrift, der Universitat Dortmund am Fachereich Informatik, Munster, Germany.

Nordin, P. & Banzhaf, W. (1995), "Complexity compression and evolution", in : L. Eshelman, ed., Proceedings of the Sixth International Conference on Genetic Algorithms, 240–245, Morgan Kauffman.

Nordin, P., Francone, F. & Banzhaf, W. (1995), "Explicitly defined introns and destructive crossover in genetic programming", in : J. P. Rosca, ed., Proceedings of the Workshop on genetic programming: from theory to real-world applications, 6–22.

Nordin, P. & Francone, W. B. F. (1999), "Compression of effective size in genetic programming", in : A. Wu, ed., Proceedings of the Genetic and Evolutionary Computation Conference, 57–60, Morgan Kauffman.

Olssen, R. (1995), "Inductive functional programming using incremental program transformation", Artificial Intelligence, 74 (1), 55–81.

O'Reilly, U.-M. (1995), "An Analysis of Genetic Programming", proefschrift, Carleton University, Ottawa, Ontario, Canada.

O'Reilly, U.-M. (1997), "Using a Distance Metric on Genetic Programs to Understand Genetic Operators", in : J. Koza, ed., Late Breaking Papers at

the 1997 Genetic Programming Conference, 199–206, Stanford University, California.

Poli, R. (2003), "A Simple but Theoretically-motivated Method to Control Bloat in Genetic Programming", in : C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli & E. Costa, eds., Genetic Programming, Proceedings of EuroGP'2003, LNCS, 2610, 204–217, Springer-Verlag, Essex.

Poli, R. & Langdon, W. (2002), "Foundations of Genetic Programming", Springer Verlag, Berlin, Heidelberg, New York.

Punch, B. & Zongker, D. (1996), "Lilgp 1.01 user's manual", Technisch rapport, Michigan State University, Michigan, USA.

Radcliffe, N. & George, F. (1993), "A study in set recombination", in : Proceedings of the Fifth Annual International Conference of Genetic Algorithms, 22–30.

Reynolds, C. (1994), "Evolution of Corridor Following Behavior in a Noisy World", in : D. Cliff, P. Husbands, J.-A. Meyer & S. Wilson, eds., From Animals to Animats 3: Proceedings of the third International Conference on Simulation of Adaptive Behavior, 402–410, MIT Press.

Rodríguez-Vásquez, K., Fonseca, C. & Fleming, P. (1997), "Multiobjective genetic programming: A nonlinear system identification application", in : J. Koza, ed., Late Breaking papers at the GP-97 conference, 207–212, Stanford Bookstore.

Ronge, A. & Nordahl, M. G. (1996), "Genetic Programs and Co-Evolution Developing robust general purpose controllers using local mating in two dimensional populations", in : H.-M. Voigt, W. Ebeling, I. Rechenberg & H.-P. Schwefel, eds., Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computation, LNCS, 1141, 81–90, Springer Verlag, Berlin, Germany.

Rosca, J. (1995), "Entropy-Driven Adaptive Representation", in : J. Rosca, ed., Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications, 23–32.

Rosca, J. (1996), "Generality versus size in genetic programming", in : J. Koza, D. Goldberg, D. Fogel & R. Riolo, eds., Genetic Programming 1996: Proceedings of the First Annual Conference, 381–387.

Rosca, J. (1997), "Analysis of complexity drift in genetic programming", in : J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba & R. Riolo, eds., Genetic Programming 1997: Proceedings of the Second Annual Conference, 286–294, Morgan Kaufmann.

Rosca, J. & Ballard, D. (1993), "Causality in Genetic Programming", in : S. Forrest, ed., Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann.

Rosca, J. & Ballard, D. (1994), "Hierarchical Self-Organization in Genetic Programming", in : Proceedings of the Eleventh International Conference on Machine Learning, 251–258.

Ryan, C. (1994), "Pygmies and civil servants", Advances in Genetic Programming I, MIT Press, Cambridge, Massachusetts, hoofdstuk 11, 243–263.

Schaffer, J. (1985), "Multiple objective optimization with vector evaluated genetic algorithms", in : J. Grefestette, ed., Proceedings of 1st International Conference on genetic algorithms, 93–100, Lawrence Erlbaum Associates.

Sette, S. & Boullart, L. (2001), "Genetic Programming: principles and applications", Engineering Applications of Artificial Intelligence, 14, 727–736.

Sette, S., Wyns, B. & Boullart, L. (2004), "Comparing Learning classifier systems and genetic programming: a case study", Engineering Applications of Artificial Intelligence, 17 (2), 199–204.

Silva, S. & Costa, E. (2005*a*), "Comparing Tree Depth Limits and Resource-Limited GP", in : D. Corne, Z. Michalewicz, M. Dorigo, G. Eiben, D. Fogel, C. Fonseca, G. Greenwood, T. K. Chen, G. Raidl, A. Zalzala, S. Lucas, B. Paechter, J. Willies, J. J. M. Guervos, E. Eberbach, B. McKay, A. Channon, A. Tiwari, L. G. Volkert, D. Ashlock & M. Schoenauer, eds., Proceedings of the 2005 IEEE Congress on Evolutionary Computation, 1, 920–927, IEEE Press, Edinburgh, UK.

Silva, S. & Costa, E. (2005*b*), "Resource-limited genetic programming: the dynamic approach", in : H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, W. Banzhaf, C. Blum, E. W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S. Mancoridis,

M. Pelikan, G. R. Raidl, T. Soule, A. M. Tyrrell, J.-P. Watson & E. Zitzler, eds., GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, 2, 1673–1680, ACM Press, Washington DC, USA.

Silva, S., Silva, P. J. N. & Costa, E. (2005), "Resource-Limited Genetic Programming: Replacing Tree Depth Limits", in : B. Ribeiro, R. F. Albrecht, A. Dobnikar, D. W. Pearson & N. C. Steele, eds., Adaptive and Natural Computing Algorithms, Springer Computer Series, 243–246, Springer, Coimbra, Portugal.

Soule, T. (2003), "Operator Choice and the Evolution of Robust Solutions", in : R. L. Riolo & B. Worzel, eds., Genetic Programming Theory and Practise, hoofdstuk 16, 257–270, Kluwer.

Soule, T. & Foster, J. (1998*a*), "Effects of code growth and parsimony pressure on populations in genetic programming", Evolutionary Computation, 6 (4), 293–309.

Soule, T. & Foster, J. (1998*b*), "Removal bias: a new cause of code growth in tree based evolutionary programming", in : 1998 IEEE International Conference on Evolutionary Computation, 781–786, IEEE Press.

Soule, T., Foster, J. & Dickinson, J. (1996), "Code growth in genetic programming", in : J. Koza, D. Goldberg, D. Fogel & R. Riolo, eds., Genetic Programming 1996: Proceedings of the first annual conference, 215–223.

Soule, T., Heckendorn, R. B. & Shen, J. (2002), "Solution Stability in Evolutionary Computation", in : N. Cicekli, ed., ISCIS XVII Seventeenth International Symposium On Computer and Information Sciences, 237–241, CRC Press, University of Central Florida, Orlando, Florida.

Stoffel, K. & Spector, L. (1996), "High-performance, parallel, stack-based genetic programming", in : J. Koza, D. Goldberg, D. Fogel & R. Riolo, eds., Genetic Programming 1996: Proceedings of the First Annual Conference, 224–229, MIT Press.

Streeter, M. J. (2003), "The Root Causes of Code Growth in Genetic Programming", in : C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli & E. Costa, eds., Genetic Programming, Proceedings of EuroGP'2003, LNCS, 2610, 443–454, Springer-Verlag, Essex.

Tackett, W. (1994), "Recombination, Selection, and the Genetic Construction of Computer Programs", proefschrift, University of Southern California, California, USA.

Teller, A. (1996), "Evolving programmers: The co-evolution of intelligent recombination operators", MIT Press, 45–68.

Teller, A. (1998), "Algorithm Evolution with Internal Reinforcement for Signal Understanding", proefschrift, School of Computer Science, Carnegie Mellon University.

Vapnik, V. N. (1999), "The Nature of Statistical Learning Theory", Springer-Verlag, Berlin.

Wagner, N. & Michalewicz, Z. (2001), "Genetic Programming with Efficient Population Control for Financial Time Series Prediction", in : E. D. Goodman, ed., 2001 Genetic and Evolutionary Computation Conference Late Breaking Papers, 458–462, San Francisco, California, USA.

Walpole, R. & Myers, R. (1978), "Probability and Statistics for Engineers and Scientists", Macmillan.

Wolpert, D. H. & Macready, W. G. (1997), "No Free Lunch Theorems for Optimization", IEEE Transactions on Evolutionary Computation, 1 (1), 67–82.

Wyns, B., De Bruyne, P. & Boullart, L. (2006), "Characterizing diversity in genetic programming", in : P. Liardet, P. Collet, C. Fonlupt, E. Lutton & M. Schoenauer, eds., Proceedings of the 9th European Conference on Genetic Programming, Lecture Notes in Computer Science, 3905, 250–259, Springer-Verlag.

Wyns, B., Sette, S. & Boullart, L. (2004), "Self-improvement to control code growth in genetic programming", in : P. Collet, M. Tomassini, M. Ebner, S. Gustafson & A. Ekárt, eds., Artificial Evolution, Proceedings of the 6th International Conference, selected papers, Lecture Notes in Computer Science, 2936, 256–266, Springer-Verlag.

Yu, T. & Clack, C. (1997), "PolyGP: A Polymorphic Genetic Programming System in Haskell", in : J. Koza, ed., Late Breaking Papers at the GP-97 Conference, 264–273, Stanford Bookstore, Stanford, CA, USA.

Yu, T. & Clack, C. (1998), "PolyGP: A Polymorphic Genetic Programming System in Haskell", in : J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba & R. Riolo, eds., Genetic Programming 1998: Proceedings of the Third Annual Conference, 416–421, Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA.

Zhang, B.-T. & Cho, D.-Y. (1999), "Genetic Programming with Active Data Selection", in : B. McKay, X. Yao, C. Newton, J.-H. Kim & T. Furuhashi, eds., Simulated Evolution and Learning: Second Asia-Pacific Conference on Simulated Evolution and Learning, Selected papers, Lectures Notes in Computer Science, 1585, 146–153, Springer-Verlag.

Zhang, B.-T. & Mühlenbein, H. (1995), "Balancing Accuracy and Parsimony in Genetic Programming", Evolutionary Computation, 3 (1), 17–38.

Zhang, B.-T. & Mühlenbein, H. (1996), "Adaptive fitness functions for dynamic growing/pruning of program trees", Advances in Genetic Programming II, MIT Press, Cambridge, Massachusetts, hoofdstuk 12, 241–256.

Zitzler, E., Deb, K. & Thiele, L. (2000), "Comparison of multiobjective evolutionary algorithms: Empirical results", Evolutionary Computation, 8 (2), 173–195.

Zitzler, E. & Thiele, L. (1999), "Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach", IEEE Transactions on Evolutionary Computation, 3 (4), 257–271.

Zomorodian, A. (1994), "Context-free language induction by evolution of deterministic push-down automata using genetic programming", Stanford Bookstore, 184–193.