Universidade de Coimbra
Faculdade de Ciências e Tecnologia
Departamento de Engenharia Informática

Controlling Bloat: Individual and Population Based Approaches in Genetic Programming

Sara Guilherme Oliveira da Silva

Coimbra
April 2008

Controlling Bloat: Individual and Population Based Approaches in Genetic Programming

A dissertation submitted to the
University of Coimbra
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Informatics Engineering

by

Sara Guilherme Oliveira da Silva

University of Coimbra
Faculty of Sciences and Tecnology
Department of Informatics Engineering

April 2008

Financial support by Fundação para a Ciência e a Tecnologia
through the PhD grant SFRH/BD/14167/2003

Controlling Bloat:
Individual and Population Based
Approaches in Genetic Programming
© 2008 Sara Silva

ISBN 978-989-20-1137-0

This dissertation was prepared
under the supervision of

Ernesto Jorge Fernandes Costa
Full Professor
of the Department of Informatics Engineering
of the Faculty of Sciences and Tecnology
of the University of Coimbra

To my dear little brat Daniel,
the product of a very successful crossover

Acknowledgments

First things first, many thanks to John Koza and Wolfgang Banzhaf et al. for writing the books that definitely put me in the path of Genetic Programming. Thank you to all the authors who have sent me the papers that I could not find on the internet or in the library, and Leonardo Vanneschi and Denis Rochat for additional material and some enlightening discussions about their work. Many thanks to Penousal Machado and Susana Vinga for lending me the LaTeX templates of their PhD theses.

Thank you so much to my advisor Ernesto Costa for allowing me so much freedom and for always supporting and contributing to my ideas, and for sometimes throwing in a much needed dose of entropy. Thank you also to my unofficial (and very personal) advisor Pedro J.N. Silva for keeping a constant interest in my work and always finding the time for pleasant and fruitful discussions about it. A global thank you to the ECOS – Evolutionary and Complex Systems group for being such a nice group of people to work with. Many thanks to Francisco and Leonor for the car rides, meals, and cheerful company in Coimbra.

Last but not least, a big thank you to my family, for always being there and providing a wonderful sense of security. Most important of all, little Daniel and his daddy, "my two men". Daniel was born right in the middle of my PhD, creating considerable havoc both in my work and in my mind, and I thank him dearly for revealing a wonderful new world and for constantly reminding me that a thesis can always wait a little longer. Thank you Pedro for being my everything, and such a wonderful daddy for our son.

Sara Silva

Lisbon, April 2008

Abstract

Genetic Programming (GP) is the automated learning of computer programs. Basically a search process, it is capable of solving complex problems by evolving populations of computer programs, using Darwinian evolution and Mendelian genetics as inspiration. Theoretically, GP can solve any problem whose candidate solutions can be measured and compared, making it a widely applicable technique. Furthermore, the solutions found by GP are usually provided in a format that users can understand and modify to their needs. But its high versatility is also the cause of some difficulties. The search space of GP is virtually unlimited and programs tend to grow in size during the evolutionary process. Code growth is a healthy result of genetic operators in search of better solutions, but it also permits the appearance of pieces of redundant code that increase the size of programs without improving their fitness. Besides consuming precious time in an already computationally intensive process, redundant code may start growing rapidly, a phenomenon known as *bloat*. Bloat can be defined as an excess of code growth without a corresponding improvement in fitness. This is a serious problem in GP, often leading to the stagnation of the evolutionary process. Although many bloat control methods have been proposed so far, a definitive solution is yet to be found.

This thesis provides a comprehensive description of all the bloat theories proposed so far, and a detailed taxonomy of available bloat control methods. Then it proposes two novel bloat control approaches, Dynamic Limits and Resource-Limited GP, both implemented on the GPLAB software package, also developed in the context of this thesis. Dynamic Limits restricts the size or depth allowed at the individual level, while Resource-Limited GP imposes restrictions only at the population level, regardless of the particularities of the individuals within. Four different problems were used as a benchmark to study the efficiency of both Dynamic Limits and Resource-Limited GP. They represent a varied selection of problems in terms of bloat dynamics and response to different bloat control techniques: Symbolic Regression of the quartic polynomial, Artificial Ant on the Santa Fe food trail, 5-Bit Even Parity, and 11-Bit Boolean Multiplexer. The results of exhaustive experiments have shown that, although Dynamic Limits was a more efficient bloat control method than Resource-Limited GP across the set of problems studied, both approaches successfully performed the task they were designed to. Without adding any parameters to the search process, it was possible to match the performance of some of the best state-of-the-art methods available so far.

Contents

List of Tables

List of Figures

Resumo Alargado em Português

Introdução

A computação evolucionária é uma área de investigação no âmbito das ciências da computação. Historicamente, é composta por quatro diferentes paradigmas inspirados na biologia: Programação Evolutiva, Algoritmos Genéticos, Estratégias Evolutivas, e a mais recente Programação Genética.

Motivação

Programação Genética (PG) é a aprendizagem automatizada de programas de computador tendo em vista a resolução de problemas complexos. Basicamente um processo de procura, resolve os problemas fazendo evoluir populações de programas, usando a evolução Darwiniana e a genética Mendeliana como fontes de inspiração. A partir de uma população inicial de programas criados aleatoriamente, representando potenciais soluções para o problema em causa, avalia e atribui um valor de aptidão a cada um, quantificando quão bem o programa resolve o problema. Novas gerações de programas são criadas iterativamente selecionando os progenitores com base na sua aptidão e reproduzindo-os usando operadores genéticos como a recombinação e a mutação. Porque os indivíduos mais aptos são selecionados mais frequentemente, sendo-lhes dada a oportunidade para passar as suas melhores características aos seus descendentes, a população tende a melhorar qualitativamente ao longo de sucessivas gerações. Este processo evolutivo continua até que se verifique uma dada condição de paragem.

Teoricamente, a PG pode resolver qualquer problema cujas possíveis soluções possam ser medidas e comparadas, tornando-a uma técnica largamente aplicável. Para mais, as soluções encontradas pela PG são geralmente fornecidas num formato que os utilizadores conseguem perceber e modificar de acordo com as suas necessidades. Mas a sua grande versatilidade é também a causa de algumas dificuldades. Os utilizadores são obrigados a especificar uma série de parâmetros relacionados com os vários aspectos do processo evolutivo, sendo que alguns desses parâmetros podem influenciar o processo de procura ao ponto de impedir que seja encontrada uma solução óptima, se escolhidos incorrectamente. E mesmo na presença de um casamento perfeito entre problemas e parâmetros,

um importante problema subsiste, que tem sido estudado durante mais de uma década: o crescimento do código.

O espaço de procura da PG é virtualmente ilimitado, e os programas tendem a crescer em tamanho durante o processo evolutivo. O crescimento do código é um resultado natural dos operadores genéticos em busca de melhores soluções, mas também permite o aparecimento de segmentos de código redundante que aumentam o tamanho dos programas sem melhorar a sua aptidão. Para além de consumir tempo precioso num processo já de si computacionalmente intensivo, o código redundante pode começar a crescer rapidamente, um fenómeno conhecido por *bloat*[1,2]. O *bloat* pode ser definido como um crescimento excessivo do código sem uma correspondente melhoria da aptidão. Isto é um problema grave em PG, frequentemente levando à estagnação do processo evolutivo. Embora vários métodos para controlar o *bloat* tenham sido propostos até agora, ainda não foi encontrada uma solução definitiva.

Contribuições

Este trabalho introduz duas novas abordagens para o controlo do *bloat*. Ao contrário de muitas outras, estas abordagens não requerem operadores genéticos específicos, modificações na avaliação da aptidão ou esquemas de seleção diferentes, nem adicionam parâmetros ao processo de procura. A primeira abordagem é inspirada na técnica mais tradicional de impor um limite fixo na profundidade dos indivíduos admitidos na população, introduzida por Koza na PG baseada em árvores. Implementa um limite dinâmico que pode ser aumentado ou diminuido, dependendo da melhor solução encontrada até então, e pode ser aplicado tanto à profundidade como ao tamanho dos programas, tornando-a numa abordagem utilizável também em PG linear. As diferentes variantes desta abordagem serão colectivamente designadas como *Limites Dinâmicos*.

A segunda abordagem para controlar o *bloat* também usa um limite dinâmico, mas este actua a um nível diferente do paradigma da PG. Um único limite é imposto ao número total de nós das árvores ou linhas de código que a população pode usar. Os nós das árvores ou as linhas de código podem ser encarados como os recursos de que cada indivíduo necessita para sobreviver, e quando se tornam insuficientes para todos, alguns indivíduos são descartados e a população é redimensionada. O limite de recursos pode ser aumentado ou diminuído como nos Limites Dinâmicos, dependendo da aptidão média da população. As diferentes variantes desta abordagem são colectivamente designadas por *PG de Recursos Limitados*.

Enquanto que os Limites Dinâmicos actuam ao nível do indivíduo, impondo uma condição que cada indivíduo deve verificar para que seja aceite na população, a PG de Recursos Limitados actua ao nível da população, impondo uma restrição global que a população como um todo deve respeitar, independentemente das particularidades dos seus indivíduos. Cada uma destas abordagens tem as suas vantagens e os seus contras. Esta tese reporta o seu desempenho em diferentes problemas, combinadas entre si e uma contra a outra, e em com-

[1] Traduzido à letra para Português, "inchaço"!

[2] Segundo Bill Langdon, "sobrevivência dos mais gordos" (do Inglês *survival of the fattest*).

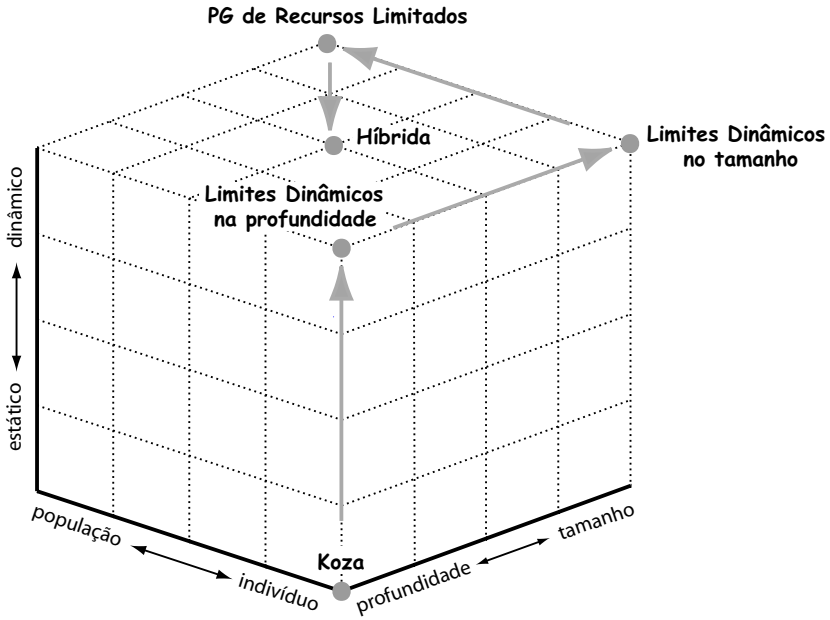

Figura 1: Arranjo das técnicas que usam limites. Os limites podem ser estáticos ou dinâmicos, impostos à profundidade ou ao tamanho, ao nível do indivíduo ou da população.

paração com os melhores métodos do estado da arte para controlo do *bloat*.

A Figura 1 mostra os Limites Dinâmicos e a PG de Recursos Limitados dispostos visualmente num cubo em que os três eixos representam limites estáticos ou dinâmicos, impostos à profundidade ou ao tamanho, ao nível do indivíduo ou da população. As setas cinzentas entre os pontos representam o historial das técnicas. Do tradicional limite estático na profundidade ao nível do indivíduo (Koza), veio a inspiração para os Limites Dinâmicos na profundidade, o que originou os Limites Dinâmicos no tamanho. Estes evoluíram para limites dinâmicos no tamanho ao nível da população, a PG de Recursos Limitados. Finalmente, uma abordagem híbrida foi implementada combinando os limites dinâmicos individuais no tamanho com os limites populacionais da PG de Recursos Limitados.

Outra contribuição desta tese é um pacote de software denominado *GPLAB – A Genetic Programming Toolbox for MATLAB*[3]. O MATLAB é um ambiente de programação largamente usado, e disponível para um grande número de plataformas computacionais. A sua linguagem de programação é simples e fácil de aprender, sendo no entanto rápida e poderosa em cálculos matemáticos. Adicionalmente, as suas ferramentas de visualização são extensas e directas, tornando-o um ambiente de programação muito apelativo. As bibliotecas são coleções de funções dedicadas e optimizadas para determinadas aplicações, que estendem o ambiente do MATLAB e providenciam uma sólida fundação onde

[3]Em Português, "Uma Biblioteca de Programação Genética para MATLAB".

desenvolver trabalho adicional. Disponibilizada primeiramente em 2003 sob a licença de software livre *GNU General Public Licence*, GPLAB foi a primeira biblioteca livremente disponível para o MATLAB, e foi rapidamente adoptada por investigadores de todo o mundo. Versátil, generalista e facilmente extensível, a GPLAB pode ser usada por todos os tipos de utilizadores, desde o leigo até ao investigador avançado. Com um continuado desenvolvimento ao longo dos anos de investigação e trabalho experimental, e também algumas contribuições de utilizadores, a GPLAB atingiu uma dimensão bem para além dos seus objectivos iniciais. A versão 3.0, disponibilizada em Abril de 2007, é composta por mais de 150 ficheiros que implementam todas as funcionalidades básicas da PG baseada em árvores mais a multitude de métodos de controlo de *bloat* com todas as variantes aqui referidas, assim como algumas funcionalidades extra que normalmente não são encontradas noutros pacotes de PG. Os ficheiros da biblioteca, assim como o manual de utilizador de 73 páginas, estão incluídos como material adicional no CD que acompanha esta tese. Também podem ser descarregados do endereço Internet da GPLAB[4].

Limites Dinâmicos

Aqui descreve-se o conjunto de técnicas de controlo de *bloat* colectivamente designadas por Limites Dinâmicos, desde a ideia inicial de aplicar um limite dinâmico à profundidade das árvores em evolução, chamado Profundidade Máxima Dinâmica das Árvores, até às variantes onde o limite pode ser aplicado tanto à profundidade como ao tamanho, e é-lhe permitido aumentar ou diminuir durante a execução.

Profundidade Máxima Dinâmica das Árvores

A PG baseada em árvores usa tradicionalmente um limite na profundidade para evitar o crescimento excessivo dos seus indivíduos. Quando um indivíduo é criado que viola este limite, em vez dele um dos seus progenitores é escolhido para a nova geração. Esta técnica evita eficazmente o crescimento das árvores para além de um certo ponto, mas não faz nada para controlar o *bloat* até o limite ser atingido. A natureza estática deste limite também pode impedir que a solução óptima seja encontrada em problemas de insuspeitada grande complexidade.

Limite Dinâmico na Profundidade

A Profundidade Máxima Dinâmica das Árvores é uma técnica de controlo de *bloat* inspirada no tradicional limite estático. Também impõe um limite de profundidade aos indivíduos aceites na população, mas este é dinâmico, o que significa que pode ser alterado durante a execução. O limite dinâmico é inicialmente muito baixo, mas pelo menos tão alto quanto a profundidade máxima das árvores aleatórias iniciais. Qualquer novo indivíduo que quebre este limite é rejeitado e substituído por um dos seus progenitores (tal como no tradicional

[4] `http://gplab.sourceforge.net`

limite estático), a não ser que seja o melhor indivíduo encontrado até então. Nesse caso, o limite dinâmico é aumentado de forma a ser igual à profundidade desse novo melhor indivíduo, e permitir a sua entrada na população. O resultado é uma sucessão de incrementos do limite, à medida que a melhor solução se vai tornando mais exacta e mais complexa.

A Profundidade Máxima Dinâmica das Árvores não substitui necessariamente o tradicional limite de profundidade - ambos os limites dinâmico e estático podem ser usados ao mesmo tempo. Quando isto acontece, o limite dinâmico situa-se sempre entre a profundidade máxima das árvores iniciais e o limite estático da profundidade. A simplicidade da Profundidade Máxima Dinâmica das Árvores torna-a fácil de usar com quaisquer valores de parâmetros e/ou combinada com outras técnicas de controlo de *bloat*.

O limite dinâmico pode ainda ser usado para outro propósito para além do controlo de *bloat*. Em aplicações do mundo real, pode não se estar interessado ou não se poder investir muito tempo em obter a melhor solução possível, particularmente em problemas de aproximação. Em vez disso, pode considerar-se que uma solução é aceitável se for suficientemente simples para ser compreendida, mesmo que a sua exactidão seja reconhecidamente mais baixa do que a exactidão de outras soluções mais complexas. Para além disso, as soluções pequenas tendem a generalizar melhor. Escolher uma condição de paragem menos exigente por forma a que o algoritmo páre mais cedo não é suficiente para garantir que a solução resultante seja aceitável, uma vez que não se pode prever a sua complexidade. Começando com um limite dinâmico baixo na produndidade das árvores, e aumentando-o repetidamente à medida que as soluções mais complexas provam ser melhores do que as mais simples, a técnica da Profundidade Máxima Dinâmica das Árvores pode de facto fornecer uma série de soluções de complexidade e exactidão crescentes, entre as quais o utilizador pode escolher a mais adequada.

Resultados Prévios

Testes anteriores mostraram que a Profundidade Máxima Dinâmica das Árvores é capaz de conter eficazmente o crescimento do código num problema de Regressão Simbólica e num problema de Paridade Par de 3 Bits[5]. Duas especificações diferentes para o valor inicial do limite dinâmico foram experimentadas (6 e 9), sendo a mais baixa exactamente a profundidade máxima das árvores aleatórias iniciais. Este valor mais restritivo resultou num tamanho médio das árvores mais baixo durante a execução, sem qualquer impedimento à capacidade de convergência para boas soluções. A Profundidade Máxima Dinâmica das Árvores também foi testada contra e em combinação com outra técnica de controlo de *bloat*, a Pressão de Parsimónia Lexicográfica[6]. As experiências mostraram uma clara superioridade do limite dinâmico, tendo os melhores resultados sido obtidos quando ambas as técnicas foram usadas em conjunto.

[5]Do Inglês *3-Bit Even Parity*.
[6]Do Inglês *Lexicographic Parsimony Pressure*.

```
para todos os novos indivíduos

  tamanho_i = tamanho (profundidade) do indivíduo
  aptidão_i = aptidão do indivíduo

  se tamanho_i ≤ limite_dinâmico
    aceita indivíduo

    se aptidão_i > melhor_aptidão
      melhor_aptidão = aptidão_i

    se MuitoPesado
      ou Pesado e tamanho_i ≥ limite_dinâmico_inicial
        limite_dinâmico = tamanho_i

  se tamanho_i > limite_dinâmico
   e aptidão_i > melhor_aptidão
    aceita indivíduo

    melhor_aptidão = aptidão_i
    limite_dinâmico = tamanho_i
```

Figura 2: Pseudo-código do procedimento de aceitação dos Limites Dinâmicos.

Variações no Tamanho e na Profundidade

A Profundidade Máxima Dinâmica das Árvores original foi rapidamente estendida de forma a incluir funcionalidades adicionais: uma variação *pesada* do limite dinâmico, que pode ser diminuído para além de aumentado, e um limite dinâmico no *tamanho* em vez da profundidade.

A Figura 2 mostra o procedimento geral de aceitação (incluindo todas as variantes) por que todos os novos indivíduos devem passar antes de serem aceites na nova geração. Qualquer indivíduo que não cumpra os requisitos de tamanho/profundidade/aptidão dos Limites Dinâmicos não será aceite por este procedimento, sendo em vez disso substituído por um dos seus progenitores.

Limite Dinâmico Pesado

A Profundidade Máxima Dinâmica das Árvores é capaz de suportar uma quantidade considerável de pressão de parsimónia, como provam os resultados obtidos inicializando o limite dinâmico com o valor mais baixo possível, a profundidade máxima das árvores aleatórias iniciais. Parece pois não haver razão para não permitir que o limite volte a cair para valores mais baixos caso a profundidade do novo melhor indivíduo fique mais baixa do que o limite actual, uma ocorrência que é na verdade muito comum. Então a primeira variação introduzida na Profundidade Máxima Dinâmica das Árvores original é o limite dinâmico *Pesado*, que acompanha a profundidade do melhor indivíduo, para cima e para baixo,

com a única restrição de não baixar mais do que o seu valor inicial. Uma outra variante do limite dinâmico pesado é o limite *MuitoPesado*, que pode diminuir mesmo abaixo do seu valor inicial.

Como esperado, sempre que o limite cai para um valor mais baixo, alguns indíviduos já pertencentes à população passam imediatamente a quebrar o novo limite, tornando-se 'ilegais'. Havia uma vasta gama de opções sobre como lidar com eles, sendo a mais drástica a sua imediata remoção da população, possivelmente substituindo-os por novos indivíduos aleatórios. No entanto, uma vez que estes 'ilegais' podem ser aqueles que conseguiram produzir o novo melhor indivíduo, removê-los não parece ser uma boa ideia. Uma opção mais branda foi adoptada: aos 'ilegais' é-lhes permitido permanecer na população tal como se não quebrassem o limite, mas quando se reproduzem os seus descendentes não podem ter mais profundidade do que o progenitor mais profundo. Isto coloca a população de novo dentro dos limites de forma natural e gradual.

Limite Dinâmico no Tamanho

Apesar de o *bloat* reconhecidamente afectar muitos outros processos de procura que usam representações de tamanho variável, os limites de profundidade não podem ser usados em PG não baseada em árvores. Estender a ideia de um limite dinâmico a outros domínios deve começar pela remoção do conceito de profundidade, e sua substituição pelo conceito de tamanho. A segunda variação à Profundidade Máxima Dinâmica das Árvores original é a utilização de um limite dinâmico no *tamanho*, em que o tamanho é o número de nós. Caso um limite estático seja usado juntamente com o limite dinâmico, deve ser também no tamanho, não na profundidade.

Quando se usa o limite dinâmico no tamanho, não faz sentido continuar a usar a profundidade como restrição na inicialização das árvores. Foi pois criada uma versão modificada do método de inicialização *Meio-por-Meio em Rampa*[7], onde um número igual de indivíduos são inicializados com tamanhos entre 2 e o valor inicial do limite dinâmico no tamanho. Para cada tamanho, um número igual de indivíduos são inicializados com o métodos *Crescente*[8] e *Completo*[9], que também foram modificados de modo a obedecer apenas a restrições no tamanho. No método Crescente modificado, os indivíduos são formados por adição de nós aleatórios (internos ou terminais) sem exceder o tamanho máximo especificado; o método Completo modificado escolhe apenas nós internos até o tamanho estar próximo do especificado, e só então escolhe terminais. Ao contrário do método Completo original, pode não conseguir criar indivíduos com exactamente o tamanho especificado, mas apenas próximo (e nunca excedendo).

Resultados Prévios

Ambas as variações pesada e de tamanho foram testadas nos mesmos problemas que a técnica da Profundidade Máxima Dinâmica das Árvores original

[7] Do Inglês *Ramped Half-and-Half*.
[8] Do Inglês *Grow*.
[9] Do Inglês *Full*.

(Regressão Simbólica e Paridade Par de 3 Bits) contra e combinadas com a Pressão de Parsimónia Lexicográfica. O limite dinâmico pesado aumenta a pressão de parsimónia durante a execução. Mesmo sem tomar medidas drásticas em relação aos indivíduos que subitamente passam a quebrar o limite mais baixo, o tamanho médio das árvores ao longo da execução foi mantido significativamente mais baixo do que com a Profundidade Máxima Dinâmica das Árvores original ou apenas a Pressão de Parsimónia Lexicográfica. Mais uma vez os melhores resultados foram obtidos juntando ambas as técnicas, e a aptidão continuou a não ser afectada por valores tão elevados de pressão de parsimónia. O tamanho dinâmico, no entanto, não teve um desempenho tão bom num dos problemas (Paridade Par de 3 Bits), onde a capacidade de encontrar boas soluções ficou comprometida. Estes resultados prévios não incluiram a variante MuitoPesado.

PG de Recursos Limitados

Aqui descreve-se o conceito e implementação da PG de Recursos Limitados, desde a ideia original de substituir os limites de profundidade/tamanho ao nível do indivíduo por um limite global nos recursos usados por toda a população, até à inspiração nos Limites Dinâmicos para criar um limite de recursos dinâmico, e a necessária comparação com a abordagem original ao nível do indivíduo.

Substituindo os Limites de Profundidade das Árvores

A PG de Recursos Limitados baseia-se num único limite imposto à quantidade de recursos disponíveis para toda a população de PG, onde os recursos são os nós das árvores ou outros elementos em PG não baseada em árvores, como linhas de código. Pode-se pensar nesta abordagem como a limitação dos recursos naturais disponíveis para uma dada população biológica, onde cada indivíduo compete com os outros pela sua parte, e os indivíduos mais fracos sucumbem quando os recursos são escassos. Na PG de Recursos Limitados, os recursos tornam-se escassos quando o número total de nós da população excede o limite predefinido. Para lá deste ponto, não há garantias de que todos os descendentes sejam aceites na nova geração. A alocação de recursos aos indivíduos (garantindo a sua sobrevivência) é principalmente baseada na aptidão, desempenhando o tamanho um papel secundário.

Os candidatos à nova geração são os descendentes, seguidos pelos seus progenitores. Cada um destes grupos é ordenado pelos valores de aptidão, independentemente do tamanho. Aos indivíduos em fila de espera são-lhes dados os recursos de que necessitam (o seu número de nós), sendo os primeiros da fila os primeiros a serem servidos. Os indivíduos que requerem mais recursos do que aqueles ainda disponíveis são ignorados (não sobrevivem) e o processo de alocação continua até ao fim da fila, ou até que se verifiquem restrições relativas ao tamanho da população. Alguns recursos podem ficar por usar. Alguns progenitores podem sobreviver enquanto os seus descendentes sucumbem. Deste procedimento emerge uma regra que promove a sobrevivência dos melhores indivíduos e a rejeição daqueles que 'não são suficientemente bons para o

seu tamanho', em que a relação entre tamanho e aptidão não é explicitamente programada, mas sim um produto do processo evolutivo.

A PG de Recursos Limitados remove a maioria das desvantagens dos limites de profundidade ao nível do indivíduo, ao mesmo tempo que introduz o redimensionamento automático da população, um efeito secundário naturalmente resultante de usar uma abordagem ao nível da população. Quando o limite de recursos é alcançado, e desde que o código continue a crescer, o tamanho da população (definido como o número de indivíduos) começa a decrescer progressivamente, algo que pode na verdade melhorar a convergência para boas soluções. É possível que um único indivíduo tenha que ser artificialmente mantido na população para evitar a extinção, mas este risco é não-existente se a recombinação de árvores for o único operador genético usado. Depois de os recursos terem atingido o ponto de exaustão e o tamanho da população ter sido reduzido, mais cedo ou mais tarde uma nova geração de indivíduos usará os recursos de forma mais poupada e deixará suficientes recursos para permitir que o tamanho da população aumente outra vez. Duas opções de implementação foram consideradas sobre como lidar com esta ocorrência: Depois de aceitar tantos indivíduos quanto o anterior tamanho da população, (1) usa-se os restantes recursos para permitir a sobrevivência de mais indivíduos da geração anterior - os progenitores que ainda não tinham sido aceites - continuando o procedimento de alocação até à exaustão dos recursos, ou até que seja atingido o tamanho inicial da população, ou (2) não se usa os recursos que sobram, desta forma nunca permitindo à população que volte a crescer. A primeira opção foi designada por *Estável*, porque obriga a uma utilização estável dos recursos , e a segunda foi denominada *Baixa*, porque permite uma possível baixa utilização de recursos. A Figura 3 mostra o pseudo-código do procedimento de alocação de recursos. Ver Figura 5 para um exemplo.

Resultados Prévios

A PG de Recursos Limitados foi testada num problema de Regressão Simbólica simples. Para comparar o seu desempenho com o uso tradicional de limites de profundidade ao nível do indivíduo, um limite de recursos estático (14500) foi encontrado que permitiu que a quantidade cumulativa de recursos (usados ao longo de toda a execução) fosse semelhante à quantidade cumulativa obtida aquando da utilização do tradicional limite estático na profundidade 17, introduzido por Koza. Os resultados mostraram que as técnicas Estável e Baixa se comportaram de forma semelhante, conseguindo o mesmo desempenho que o limite de profundidade, apesar de usarem diferentes estratégias de controlo de *bloat* e produzirem uma dinâmica evolutiva radicalmente diferente. A PG de Recursos Limitados foi bem sucedida como substituto do popular limite da profundidade das árvores.

A Abordagem Dinâmica

Tal como o tradicional limite de profundidade, a PG de Recursos Limitados original baseia-se num limite estático, imposto no início da execução e nunca alterado até ao fim. Isto dificilmente reflecte as necessidades de um processo

```
ordena descendentes por aptidão
ordena progenitores por aptidão
lista = descendentes seguidos dos progenitores

se Estável, minha_dimpop = dimensão_pop_inicial
se Baixa,   minha_dimpop = dimensão_pop_anterior
```

```
recursos_usados = 0
lista_aceites = vazia

para todos os indivíduos em lista

  recursos_i = recursos requisitados pelo indivíduo

  se recursos_usados + recursos_i ≤ limite_recursos
    lista_aceites = lista_aceites + indivíduo
    recursos_usados = recursos_usados + recursos_i

  se comprimento de lista_aceites = minha_dimpop
    termina para

nova geração = lista_aceites
```

Figura 3: Pseudo-código do procedimento de alocação de recursos.

de procura que tem que fazer crescer os seus indivíduos na procura de soluções melhores. Embora a PG de Recursos Limitados possua uma capacidade natural para compensar um maior tamanho das árvores com uma menor dimensão da população, em problemas complexos isso pode levar a uma perigosa redução do tamanho da população, à medida que prossegue o crescimento do código. Por outro lado, fornecer recursos estáticos suficientes para toda a execução pode levar à ocorrência de *bloat* desde o início. Torna-se óbvia a necessidade de um limite de recursos dinâmico.

Limite de Recursos Dinâmico

A abordagem dinâmica à PG de Recursos Limitados surge naturalmente da hibridização dos Limites Dinâmicos com o limite de recursos estático original. É implementado um limite de recursos dinâmico, inicializado com um valor baixo e aumentado sempre que tal resultar numa melhor aptidão média da população.

Depois da geração de descendentes, os candidatos à nova geração são ordenados e são-lhes dados os recursos disponíveis, segundo o procedimento descrito anteriormente, ilustrado na Figura 3. A alocação continua até à exaustão dos recursos, ou até a dimensão inicial da população ser atingida, de acordo com a opção Estável. Também é possível parar a alocação assim que a anterior dimensão da população seja atingida, de acordo com a opção Baixa. Até agora,

isto é a PG de Recursos Limitados original, mas agora vem a decisão acerca de quando aumentar o limite de recursos.

Aos indivíduos rejeitados é agora dada uma segunda oportunidade. À vez, cada um deles é reconsiderado como candidato à nova geração, e tantos quantos possível são agora aceites, desde que a sua inclusão provoque um melhoramento da aptidão média da população. Este melhoramento pode ser relativo à melhor aptidão média da população durante toda a execução, ou relativo à aptidão média da população na geração anterior, criando duas opções de implementação diferentes designadas por ResDin (de recursos dinâmicos) e ResDinLeve, respectivamente. Espera-se que ResDinLeve implemente um limite que sobe muito mais facilmente, e daí o nome. Assim que um dos indivíduos previamente rejeitados for rejeitado novamente, o processo de reseleção pára e o limite de recursos é aumentado de modo a fornecer os recursos adicionais necessários.

Como nos Limites Dinâmicos, a PG de Recursos Limitados também inclui um limite *Pesado* que volta a cair para valores mais baixos quando alguns recursos ficam por usar, e um limite *MuitoPesado* que pode mesmo cair abaixo do seu valor inicial. A Figura 4 mostra o pseudo-código do procedimento de reseleção. Ver a Figura 5 para um exemplo.

Resultados Prévios

Testada em dois problemas diferentes e comparada com ambos os limites de profundidade estático e dinâmico, o desempenho da variante dinâmica da PG de Recursos Limitados (a opção Estável) variou entre bom e excelente. Num problema de Regressão Simbólica a técnica resultou numa aptidão semelhante com uma utilização de recursos significativamente mais baixa, apesar de a taxa de sucesso (medida como a percentagem de experiências que convergiram para uma solução óptima) ter sido um pouco mais baixa do que usando o limite de recursos estático. Num problema mais complexo da Formiga Artificial, o limite de recursos dinâmico alcançou a mesma aptidão usando uma quantidade significativamente menor de recursos, sendo que os resultados revelaram óptimas perspectivas de se obter um óptimo muito mais facilmente do que com as outras técnicas. Estes resultados prévios não incluiram a opção Baixa, nem as variantes Pesado ou MuitoPesado.

Comparação com os Limites Dinâmicos

Os Limites Dinâmicos e a PG de Recursos Limitados operam em diferentes níveis do paradigma da PG: um actua ao nível do indivíduo, o outro ao nível da população. Ambos conseguiram anteriormente resultados promissores no controlo do *bloat* sem piorar o desempenho mas, porque apontam para alvos diferentes, produzem diferentes dinâmicas no processo evolutivo. Qual deles obtém melhores resultados?

Resultados Prévios

Resultados comparativos prévios entre os Limites Dinâmicos e a PG de Recursos Limitados elegeram a versão não leve do limite de recursos dinâmico como a

```
se ResDin     , minha_aptmedpop = melhor_aptidão_média_pop
se ResDinLeve, minha_aptmedpop = anterior_aptidão_média_pop
```

```
lista_rejeitados = lista - lista_aceites
actual_aptmedpop = aptidão média de lista_aceites

se actual_aptmedpop melhor que melhor_aptmedpop
  melhor_aptmedpop = actual_aptmedpop

para todos os indivíduos em lista_rejeitados

  lista_aceites_tmp = lista_aceites + indivíduo
  nova_aptmedpop = aptidão média de lista_aceites_tmp

  se nova_aptmedpop melhor que minha_aptmedpop
    lista_aceites = lista_aceites_tmp
    recursos_i = recursos requisitados pelo indivíduo
    recursos_usados = recursos_usados + recursos_i

  senão
    termina para

  se comprimento de lista_aceites = dimensão_pop_inicial
    termina para

nova geração = lista_aceites

se recursos_usados > limite_recursos
  ou Pesado e recursos_usados ≥ limite_recursos_inicial
  ou MuitoPesado
    limite_recursos = recursos_usados
```

Figura 4: Pseudo-código do procedimento de reseleção.

técnica com melhor desempenho. Nos dois problemas simples de Regressão Simbólica e Paridade Par de 3 Bits, conseguiu atingir a mesma melhor aptidão utilizando significativamente menos recursos do que as restantes técnicas. No problema mais complexo da Formiga Artificial, obteve significativamente melhor aptidão utilizando o mesmo número de recursos. De acordo com o mesmo critério, a técnica que ficou em segundo lugar foi a da profundidade dinâmica não pesada. O tradicional limite estático de profundidade foi o último. No entanto, ao contrário dos Limites Dinâmicos, a PG de Recursos Limitados não manteve um desempenho constante ao longo da evolução. Conseguiu resultados muito bons no início da execução, mas as melhorias foram abrandando gradualmente à medida que a execução prosseguia, levantando a dúvida se serão de facto a melhor abordagem.

Variáveis:

```
dimpop_inicial = 10
dimpop_anterior = 6
limite_recursos = 400
melhor_aptmedpop = 42
anterior_aptmedpop = 35
```

Candidatos à nova geração:
(da esquerda para a direita, 6 descendentes seguidos de 6 progenitores,
cada grupo ordenado por aptidão - valores mais altos são melhores)

Id	C1	C2	C3	C4	C5	C6	P1	P2	P3	P4	P5	P6
Aptidão	80	70	60	60	50	10	90	40	40	20	10	10
Tamanho	90	100	50	100	80	80	80	70	70	10	20	10

Nova geração obtida com cada técnica:
(ResDin and ResDinLeve começam a reseleção
a partir dos resultados de Estável)

Id	C1	C2	C3	C4	C5	C6	P1	P2	P3	P4	P5	P6
Estável	✓	✓	✓	✓	✗[1]	✗[1]	✗[1]	✗[1]	✗[1]	✓	✓	✓
Baixa	✓	✓	✓	✓	✗[1]	✗[1]	✗[1]	✗[1]	✗[1]	✓	✓	✗[2]
ResDin	✓	✓	✓	✓	✔	✘[3]	–	–	–	✓	✓	✓
ResDinLeve	✓	✓	✓	✓	✔	✔	✔	✘[4]	–	✓	✓	✓

Motivos para a não aceitação do indivíduo:
[1]Recursos não disponíveis
[2]`dimpop_anterior` excedida - terminação do procedimento
[3]`nova_aptmedpop` pior que `melhor_aptmedpop` - terminação do procedimento
[4]`dimpop_inicial` excedida - terminação do procedimento

Figura 5: Exemplo dos procedimentos de alocação de recursos e reseleção.

Experiências e Resultados

Aqui descreve-se os problemas usados para testar as duas abordagens originais descritas anteriormente, Limites Dinâmicos e PG de Recursos Limitados, indicando também quais as comparações efectuadas nas experiências. Finalmente, resume-se os resultados obtidos em todas as comparações.

Problemas e Comparações

Quatro problemas diferentes foram usados como bancada de testes no estudo da eficiência dos Limites Dinâmicos e da PG de Recursos Limitados. Representam uma seleção variada de problemas em termos de dinâmica de *bloat* e resposta a diferentes técnicas de controlo de *bloat*: Regressão Simbólica de um polinómio de quarto grau, Formiga Artificial no trilho de comida de Santa Fé, Paridade Par de 5 Bits, e Multiplexor Booleano de 11 Bits.

O objectivo do problema de Regressão Simbólica é fazer evoluir uma função que melhor aproxime um conjunto de pontos. Neste caso particular, são usados 21 pontos equidistantes do polinómio de quarto grau $x^4 + x^3 + x^2 + x$ no intervalo entre -1 e 1. No problema da Formiga Artificial o objectivo é fazer evoluir uma estratégia que permita à formiga consumir o maior número possível de unidades de comida dispostas numa grelha (toroidal) de 32×32 células. O trilho de comida apresenta várias interrupções, e a formiga tem tempo limitado para realizar a sua tarefa. O problema da Paridade Par de 5 Bits é na verdade um problema de regressão simbólica em que a função a encontrar aceita cinco argumentos booleanos e retorna um único valor indicando a paridade dos argumentos: 1 (ou verdadeiro) se há um número par de argumentos 1, e 0 (falso) caso contrário. Também selelhante a um problema de regressão simbólica, o problema do Multiplexor Booleano de 11 Bits pode no entanto ser encarado como um problema de desenho de circuitos electrónicos. A função a encontrar aceita três argumentos representando um endereço, mais oito argumentos de dados, todos booleanos. O valor retornado pela função é exactamente o bit de dados especificado pelos bits de endereço.

Todas as experiências foram efectuadas com PG baseada em árvores, utilizando a biblioteca GPLAB. A significância estatística da hipótese nula de não haver diferença foi determinada usando a análise de variância (ANOVA) de Kruskal-Wallis com $p = 0.01$. Foi usada uma ANOVA não paramétrica pois não é garantido que os dados sigam uma distribuição normal.

Uma primeira comparação de resultados foi efectuada entre as técnicas de Limites Dinâmicos, seguida de uma comparação entre as técnicas de PG de Recursos Limitados. As melhores técnicas de ambas as abordagens tomaram então parte num terceiro grupo de experiências, comparadas entre si e combinadas de modo a formar uma nova técnica híbrida. Uma última comparação foi efectuada com alguns dos melhores métodos de controlo de *bloat* do estado da arte. A base de comparação foi sempre a tradicional técnica de Koza baseada num limite máximo estático na profundidade das árvores. Todas as técnicas foram testadas com e sem este limite fixo. O objectivo foi verificar se estas conseguem operar sem o limite máximo estático, ou se mesmo assim beneficiam de serem combinadas com a técnica base, em vez de apenas a substituirem.

Resultados dos Limites Dinâmicos

De entre os Limites Dinâmicos, sempre que diferenças estatisticamente significativas permitiram que as técnicas fossem ordenadas segundo o seu desempenho, uma das variantes baseadas em profundidade conseguiu sempre o primeiro lugar em todos os problemas, nunca se qualificando atrás, e muitas vezes apresentando um desempenho significativamente melhor, do que a bem sucedida técnica base Koza. Todas as técnicas de Limites Dinâmicos conseguiram controlar o *bloat* sem necessitarem do limite máximo estático, uma propriedade extremamente desejável.

Resultados da PG de Recursos Limitados

De entre as técnicas de PG de Recursos Limitados, uma das variantes utilizando o limite máximo estático distinguiu-se por conseguir obter um desempenho semelhante a Koza, e encontrar soluções óptimas rápida e frequentemente nos casos em que não houve diferenças significativas em termos da melhor aptidão alcançada no final. Em muitos casos, esta abordagem tende a colapsar a população em apenas uns poucos indivíduos (geralmente só um, pequeno), independentemente da dificuldade do problema. Para problemas fáceis como a Regressão, isto significa um bom desempenho, pois encontra-se um óptimo poupando uma grande quantidade de recursos. Mas, para problemas difíceis como a Paridade, a população tende a colapsar antes de encontrar um óptimo, tornando toda a procura subsequente inútil devido a falta de diversidade genética. Logo, o desempenho da PG de Recursos Limitados não foi igual para todos os problemas, tendo-se obtido resultados excepcionais no problema da Regressão, e apenas um modesto êxito nos restantes problemas.

Comparação entre Limites Dinâmicos e PG de Recursos Limitados

Na comparação entre os Limites Dinâmicos e a PG de Recursos Limitados, foram os limites ao nível do indivíduo que provaram deter o papel principal no controlo do crescimento das árvores. Entre as melhores técnicas de ambas as abordagens observaram-se diferenças significativas no tamanho médio das árvores. Mas quando numa técnica de recursos limitados se substitui o limite individual estático por um limite individual dinâmico, a nova técnica híbrida produz curvas de crescimento semelhantes às obtidas com os Limites Dinâmicos, não com a PG de Recursos Limitados. Independentemente do crescimento das árvores, os limites ao nível do indivíduo também provaram ser os principais responsáveis por melhores valores de aptidão. Quer se use um limite estático ou dinâmico, passar de uma população de tamanho fixo para uma população de tamanho variável não melhorou o desempenho; mas quer se use uma população fixa ou variável, passar de um limite estático para um limite dinâmico quase sempre melhorou os resultados. O mérito pertence ao limite dinâmico, não à população de tamanho variável.

Comparação com o Estado da Arte

Relativamente à comparação com o estado da arte, as técnicas que conseguiram os melhores valores de aptidão no conjunto dos quatro problemas foram a técnica base de Koza, o limite dinâmico não pesado na profundidade, e uma técnica do estado da arte que efectua a seleção para reprodução utilizando um torneio duplo, primeiro baseado no tamanho dos indivíduos e depois nos valores de aptidão, ou vice versa. Apesar de o torneio duplo ter geralmente conseguido chegar a uma solução óptima mais facilmente, tem a desvantagem de utilizar um método de seleção cujo desempenho pode depender da correcta especificação dos seus parâmetros. Já era esperado que os valores de parâmetros escolhidos para estas experiências dessem bons resultados, uma vez que foram precisamente os melhores valores encontrados em estudos anteriores utilizando o mesmo conjunto de problemas. Outras técnicas do estado da arte baseadas em populações de tamanho variável tiveram um fraco desempenho.

Discussão

Aqui resume-se a análise efectuada aos possíveis motivos que levaram a um desempenho desigual das duas abordagens nos vários problemas considerados. São também apresentadas algumas preocupações e considerações acerca de como alguns detalhes de implementação podem influenciar a dinâmica de procura, em particular à luz da mais recente teoria que explica o *bloat*.

Análise por Problema

Foi efectuada uma análise mais profunda relativamente aos motivos pelos quais cada uma das abordagens obteve resultados diferentes nos vários problemas considerados, mais precisamente, porque é que o PG de Recursos Limitados teve um desempenho excepcional no problema da Regressão Simbólica, precisamente o problema onde a pontuação dos Limites Dinâmicos não foi tão brilhante. Foram considerados dois indicadores da dificuldade dos problemas, nomeadamente a taxa de sucesso e a velocidade de convergência, e multiplicados por forma a obter um único valor de dificuldade para cada problema. A percentagem de código inviável presente na população ao longo de 50 gerações de evolução também foi estudado, assim como a diversidade (genotípica) da população, e a diversidade fenotípica na última geração. A principal observação foi que o problema da Regressão Simbólica é de facto diferente dos outros. É um problema mais fácil, menos dado a código inviável, e é capaz de manter uma diversidade fenotípica muito mais elevada, mesmo na presença de uma diversidade genotípica mais baixa. Em geral, as experiências com o problema da Regressão demonstraram uma gama mais vasta de comportamentos do que as experiências com os outros problemas. Foi sugerido que, apesar de tudo, é o tamanho da população relativamente às necessidades do problema que mais influencia o nível de sucesso da PG de Recursos Limitados.

Considerações Finais

Existe uma preocupação considerável relativamente à utilização de limites de tamanho ou profundidade em PG. A forma mais tradicional de implementar os limites é rejeitando os indivíduos inválidos e substituindo-os por um dos seus progenitores. Embora isto impeça eficazmente os indivíduos de crescerem demasiado, a replicação de progenitores pode ter efeitos indesejáveis. Os progenitores maiores são aqueles que geralmente produzem descendência inválida, por isso tendem a ser replicados mais frequentemente do que os progenitores mais pequenos. A população fica cheia dos indivíduos maiores, sendo rapidamente levada ao limite. Formas alternativas de implementar os limites de tamanho ou profundidade são: 1) repetir o operador genético até que um descendente válido seja criado, com os mesmos progenitores ou com outros; 2) aceitar os indivíduos inválidos mas atribuindo-lhes um valor de aptidão nulo, por forma a que estes não sejam selecionados para reprodução na geração seguinte.

A mais recente teoria sobre o *bloat* defende que é o operador de recombinação o responsável pelo rápido crescimento do código, pela forma como influencia a distribuição de tamanhos das árvores na população. Sempre que a recombinação é usada, a quantidade de material genético removido do primeiro progenitor é exactamente a mesma quantidade inserida no segundo progenitor, e vice versa. O tamanho médio das árvores mantém-se inalterado. No entanto, à medida que a população sofre repetidas operações de recombinação, aproxima-se de uma distribuição particular de tamanhos de árvores em que os indivíduos pequenos são muito mais abundantes do que os indivíduos maiores. Por exemplo, a recombinação gera uma grande quantidade de indivíduos com apenas um nó. Porque os indivíduos muito pequenos costumam ser pouco aptos, a seleção tende a preteri-los em favor dos indivíduos maiores, causando um aumento do tamanho médio das árvores na população. É a proliferação destes pequenos indivíduos pouco aptos, perpetuada pela recombinação, que acaba por causar o *bloat*.

À luz desta teoria, repetir o operador de recombinação quando um indivíduo inválido é produzido pode não ser aconselhável, uma vez que oferece mais uma oportunidade para criar pequenos indivíduos pouco aptos. Aceitar os indivíduos grandes demais parece ser uma medida melhor contra o efeito indesejável da recombinação, uma vez que estes grandes indivíduos de aptidão nula nunca procriarão. No entanto, na presença de um limite altamente restritivo que não sobe facilmente, a replicação de progenitores pode ainda ter algumas vantagens sobre as outras opções.

Na actual implementação dos Limites Dinâmicos, a tradicional replicação de progenitores é de facto a opção tomada quando os descendentes violam o limite. Mas, ao contrário dos limites típicos (estáticos), o valor inicial do limite dinâmico é muito baixo, tão baixo quanto o tamanho/profundidade máxima das árvores iniciais, e não será aumentado até que um indivíduo maior prove ser melhor do que qualquer outro encontrado até ao momento. Isto restringe seriamente o espaço de procura, e para a maioria dos problemas é sabido que as boas soluções se situam acima deste limite, não abaixo. Quando um indivíduo maior e melhor empurra o limite para cima, isto significa que o processo entrou num terreno melhor de procura - soluções melhores podem ser encontradas nos novos limites de tamanho/profundidade permitidos. Assim, encaminhar a população na

direção deste novo, mas ainda muito restritivo, espaço de procura, pode mesmo acelerar a convergência para soluções melhores. A replicação de progenitores, juntamente com um limite que sobe lentamente, não sofre necessariamente das desvantagens inerentes ao uso de um limite estático elevado.

Na PG de Recursos Limitados, embora muitas das experiências realizadas tenham usado um limite estático ao nível do indivíduo, o maior esforço no sentido de evitar o crescimento do código é exercido ao nível da população. Mas ao fazer isso, a PG de Recursos Limitados pode estar a criar condições desfavoráveis que estimulam ainda mais o *bloat*, de acordo com a mais recente teoria aqui explicada. Durante o processo de alocação de recursos aos indivíduos em fila de espera, chega-se invariavelmente a um ponto em que os recursos disponíveis já são tão escassos que só os indivíduos mais pequenos podem ainda ser aceites. O procedimento de alocação continua a garantir a sobrevivência destes pequenos indivíduos pouco aptos até que os recursos se acabem ou se verifiquem restrições ao tamanho da população (em termos de número de indivíduos). Em condições específicas, isto pode resultar na aceitação de todos os pequenos indivíduos criados no último ciclo de reprodução, assim como aqueles da geração anterior, agravando assim o efeito que causa o *bloat*.

Conclusões

Apesar de os limites dinâmicos se terem revelado mais eficientes no controlo do *bloat* do que a PG de Recursos Limitados no conjunto dos quatro problemas considerados, ambas as abordagens desempenharam com sucesso a tarefa para que foram desenhadas. Um bom método de controlo de *bloat* deve ser capaz de lidar com qualquer tipo de problema, e ser praticamente insensível à escolha dos valores dos parâmetros e mesmo à combinação de diferentes elementos algorítmicos como os procedimentos de avaliação, seleção e reprodução. Ambas as contribuições originais desta tese seguem estas directivas. São livres de parâmetros e suficientemente flexíveis para adaptarem o seu comportamento às particularidades de cada situação, sem esgotarem os recursos computacionais disponíveis.

Chapter 1

Introduction

Evolutionary Computation is a research area within computer science [27]. Historically, it is composed of four different biologically inspired paradigms: Evolutionary Programming, Genetic Algorithms, Evolution Strategies, and the most recent Genetic Programming.

1.1 Motivation

Genetic Programming (GP) is the automated learning of computer programs [9, 45]. Basically a search process, it is capable of solving complex problems by evolving populations of computer programs, using Darwinian evolution and Mendelian genetics as inspiration. Starting from an initial population of randomly created programs representing the potential solutions to a given problem, it evaluates and attributes a fitness value to each, quantifying how well the program solves the problem. New generations of programs are iteratively created by selecting parents based on their fitness and breeding them using genetic operators like crossover and mutation. Because fitter individuals are selected more often and given the chance to pass their best characteristics to their offspring, the population tends to improve in quality along successive generations. This evolutionary process continues until a given stop condition is verified.

Theoretically, GP can solve any problem whose candidate solutions can be measured and compared, making it a widely applicable technique. Furthermore, the solutions found by GP are usually provided in a format that users can understand and modify to their needs. But its high versatility is also the cause of some difficulties. Users must set a number of parameters related to several aspects of the evolutionary process, some of which may influence the search process so strongly as to actually prevent an optimal solution to be found, if set incorrectly. And even when the perfect match between problem and parameters is achieved, a major problem remains, one that has been studied for more than a decade: code growth.

The search space of GP is virtually unlimited and programs tend to grow in size during the evolutionary process. Code growth is a healthy result of genetic operators in search of better solutions, but it also permits the appearance of

pieces of redundant code that increase the size of programs without improving their fitness. Besides consuming precious time in an already computationally intensive process, redundant code may start growing rapidly, a phenomenon known as *bloat*[1] [9, Chap. 7], [58, Chap. 11]. Bloat can be defined as an excess of code growth without a corresponding improvement in fitness. This is a serious problem in GP, often leading to the stagnation of the evolutionary process. Although many bloat control methods have been proposed so far, a definitive solution is yet to be found.

1.2 Contributions

This work introduces two new approaches to bloat control. Unlike many others available, these do not require specific genetic operators, modifications in fitness evaluation or different selection schemes, nor do they add any parameters to the search process. The first approach is inspired on the most traditional technique of imposing a fixed limit on the depth of the individuals allowed in the population, introduced by Koza in tree-based GP [45]. It implements a dynamic limit that can be raised or lowered, depending on the best solution found so far, and can be applied either to the depth or size of the programs being evolved, thus making it suitable also for linear GP [9]. The different variants of this approach will be collectively referred to as *Dynamic Limits* [101, 102].

The second approach to bloat control also uses a dynamic limit, but one that acts at a different level of the GP paradigm. A single limit is imposed on the total amount of tree nodes or code lines that the entire population can use. Tree nodes or code lines can be regarded as the resources that each individual needs to survive, and when they become insufficient for all, some individuals are discarded and the population is resized. The resource limit can be raised or lowered as in Dynamic Limits, depending on the mean population fitness. The several variants of this approach are called *Resource-Limited GP* [103–105].

While the Dynamic Limits act at the individual level, imposing a condition that each individual must verify in order to be accepted into the population, Resource-Limited GP acts at the population level, enforcing a global restriction that the population as a whole must respect, regardless of the particular individuals within. Each has its advantages and drawbacks. This thesis reports how they performed in different problems, combined and against each other, and in comparison with the best state-of-the-art bloat control methods.

Figure 1.1 shows the Dynamic Limits and Resource-Limited GP visually arranged in a cube where the three axes represent static or dynamic limits, imposed on depth or size, at the individual or population level. The gray arrows between the dots represent the history of the techniques. From the traditional static depth limit at the individual level (Koza) came the inspiration for the Dynamic Limits on depth, and this originated the Dynamic Limits on size. These evolved to dynamic limits on size at the population level, Resource-Limited GP. Finally, a hybrid approach was implemented by joining the individual dynamic depth limits with the population limits of Resource-Limited GP.

[1] Or, as Bill Langdon put it, the "survival of the fattest".

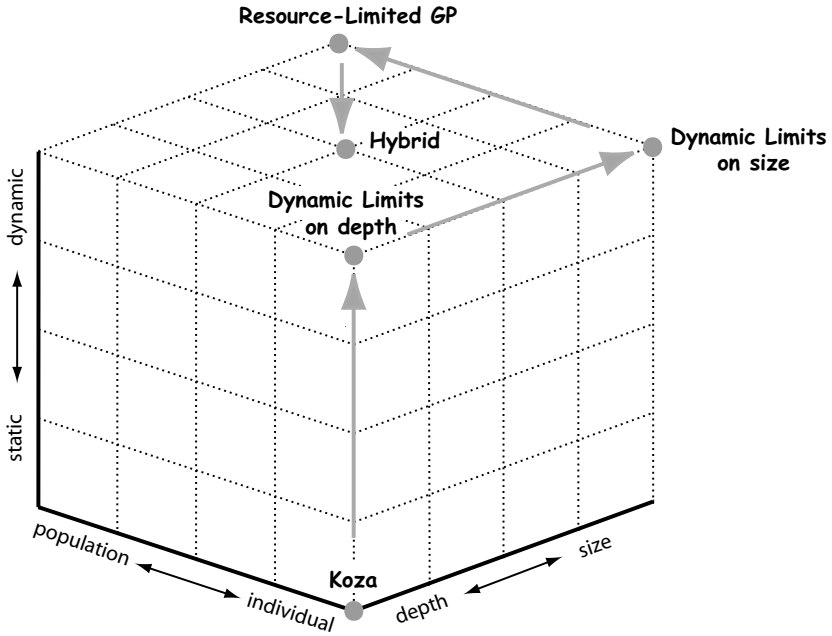

Figure 1.1: Arrangement of techniques that use limits. Limits can be static or dynamic, on depth or size, and at the individual or population level.

Another contribution of this thesis is a GP software package denominated *GPLAB – A Genetic Programming Toolbox for MATLAB* [100]. MATLAB[2] is a widely used programming environment available for a large number of computer platforms. Its programming language is simple and easy to learn, yet fast and powerful in mathematical calculus. Furthermore, its extensive and straightforward data visualization tools make it a very appealing programming environment. Toolboxes are collections of optimized, application-specific functions, which extend the MATLAB environment and provide a solid foundation on which to build. First released in 2003 under the GNU General Public License, GPLAB was the first GP toolbox freely available for MATLAB, and was quickly embraced by researchers around the world (e.g. [8, 18, 36, 61, 69, 119]). Versatile, generalist and easily extendable, GPLAB can be used by all types of users, from the layman to the advanced researcher. With continued development along the years of research and experimental work, and also a few contributions from helpful users, GPLAB has now reached a dimension well beyond its initial goals. Version 3.0, released in April 2007, is made of more than 150 files that implement all the basic tree-based GP functionalities plus the multitude of bloat control methods with all the variants herein referred, along with some extra functionalities not usually found in other GP packages. The toolbox files, along with the 73 page user's manual, are included as additional material in the CD accompanying this thesis. They can also be downloaded from the GPLAB website[3].

[2] `http://www.mathworks.com/matlab`
[3] `http://gplab.sourceforge.net`

1.3 Structure

The next chapter deals with bloat. It describes the main theories regarding why it occurs, and lists many of the bloat control methods that have been proposed in the literature so far. Chapter 3 describes the Dynamic Limits in detail, and Chapter 4 does the same with Resource-Limited GP. Chapter 5 describes the problems and specifies the general parameters used along the experiments, and introduces the plots that are later used to present the results. Chapter 6 reports the results of the comparisons among the Dynamic Limits variants, whereas Chapter 7 presents the results of comparing among Resource-Limited GP. Chapter 8 compares the two approaches with each other and with a hybrid technique. Chapter 9 ends the results by setting the best techniques from the previous comparisons against some successful state-of-the-art methods. Chapter 10 discusses the results, and Chapter 11 concludes and points towards future developments of this work.

Chapter 2

Bloat

When Koza published the first book on GP [45], most of the evolved programs therein contained pieces of code that did not contribute to the solution and could be removed without altering the results produced. Besides imposing a depth limit to the trees created by crossover to prevent spending computer resources on extremely large programs, Koza also routinely edited the solutions provided at the end of each run to simplify some expressions while removing the redundant code.

Two years later, Angeline remarked the ubiquity of these redundant code segments and, based on a slight biological similarity, called them *introns* [4]. In spite of classifying them as extraneous, unnecessary and superfluous, Angeline noted that they provided crossover with syntactically redundant constructions where splitting could be performed without altering the semantics of the swapped subtrees. Referring some studies where the introduction of artificial introns was helpful or even essential to the success of genetic algorithms, Angeline revels in the fact that introns emerge naturally from the dynamics of GP, and goes as far as to state that "it is important then to not impede this emergent property as it may be crucial to the successful development of genetic programs" [4].

It is possible that introns may provide some benefits. A non-intuitive effect that introns may have in GP is code compression and parsimony. Not the bloated code full of redundant segments, but the effective code that remains after removing the introns. Under specific conditions, particularly in the presence of destructive crossover, there is evidence that the existence of introns in the population results in shorter and less complex effective solutions [76, 108, 109], ones that are thought to be more robust and generalize better [44,76,96,116,126]. Introns also do seem to provide some protection against the destructive effects of crossover and other genetic operators [1,15,76,106,108] although this may not always be helpful. The usage of explicitly defined artificial introns has yielded generally good results in linear GP [60, 78, 79], but in tree-based GP it usually degraded the performance of the search process [3, 13, 106].

Regardless of its possible benefits to GP, the side effects of intron proliferation are very serious. Computational resources may be totally exhausted in the storage, evaluation and swapping of code that contributes nothing to the final

solution, preventing GP from performing the effective search needed to find better solutions. Bloat is now widely recognized as a pernicious phenomenon that plagues most progressive search techniques based on discrete variable-length representations and using fixed evaluation functions [11, 49, 55, 57, 59]. Bloat control has become a very active research area in GP, already subject to different theoretic and analytic studies [53, 60, 73, 84, 85, 96–98]. Several theories concerning why bloat occurs have been advanced, and many different bloat control methods have been proposed.

2.1 Theories

This section briefly describes the six main theories concerning the reasons why bloat occurs, along with some related ideas that are presented alongside the main theories. The different explanations for code growth are not necessarily contradictory. Some appear to be generalizations or refinements of others, and several most certainly complement each other. They are not presented in precise chronological order.

2.1.1 Hitchhiking

One of the first explanations to the multiplication of introns among GP programs, advanced by Tackett, was the *hitchhiking* phenomenon [116]. This is a common and undesirable occurrence in genetic algorithms, where unfit building blocks propagate throughout the population simply because they happen to adjoin highly fit building blocks. The introduction of artificial introns in genetic algorithms was partly an attempt to counteract the deleterious effects of hitchhiking.

According to the hitchhiking explanation, the reason why naturally emerging introns in GP become so abundant is that they, too, are hitchhikers. Tackett refutes the hypothetical protection against crossover (see Section 2.1.2) as the explanation for intron multiplication, based on the fact that the usage of *brood recombination* (Section 2.2.3), a less destructive recombination strategy, did not result in less code growth [116]. An additional hypothesis for code growth, advanced by Altenberg and somewhat related to the *removal bias* theory later advanced by Soule (Section 2.1.3), suggested that it was caused by an "asymmetry between addition and deletion of code at the lower boundary of program size", inherent to the recombination operator and not dependent on selection pressure [2]. Tackett also refutes this hypothesis by showing that, on the contrary, code growth is directly proportional to selection pressure, and the only time bloat does not occur is when fitness is totally disregarded along the search process [116]. These results have later been reinforced by other experiments showing the absence of bloat when selection is random [10, 55, 60].

2.1.2 Defense Against Crossover

Although early disputed, the idea of *defense against crossover* as being the explanation for bloat has persisted in the literature for a long time [1, 15, 71, 76,

106, 108], also stated and referred to as the *replication accuracy theory* [71, 85], *intron theory* [34, 35, 115] and *protection theory* [17]. It is based on the fact that standard crossover is usually very destructive [9, Chap. 6], [76, 78, 79]. In face of a genetic operator that seldom creates offspring better than their parents, particularly in more advanced stages of the evolution, the advantage belongs to the individuals that at least have the same fitness as their parents, those who were created by neutral variations. Introns provide standard crossover and other genetic operators with genetic material where swapping can be performed without harming the effective code.

Curiously, most of the theory devoted to the defense against crossover was developed in the context of linear GP [76] and may not be completely applicable to tree-based GP [62,63,65,96]. More specifically, introns can be roughly divided in two categories: inviable code and unoptimized code (or syntactic/structural and semantic introns [6, 17]). The former is code that cannot contribute to the fitness no matter how many changes it suffers, either because it is never executed or because its return value is ignored. The latter is viable code containing redundant elements whose removal would not change the return value [65]. Defense against crossover does not differentiate both types of introns, which is fine when considering only linear GP. But in tree-based GP the effects of regular genetic operators are very different in each type of intron. While inviable code effectively protects the individual from having its fitness changed, unoptimized code is highly susceptible to variations of its structure and its return value may greatly influence the fitness of the individual. It is not surprising to verify that the experiments supporting the defense against crossover in tree-based GP do not take into consideration any other type of intron besides inviable code.

Some of these experiments were performed by Soule and Foster, using a form of non-destructive *hill-climbing crossover* [80, 109] (Section 2.2.3) and studying its effects on code growth. In this crossover the offspring are kept only if they are strictly better than their parents in terms of fitness. Specifics apart, when offspring do not rise to these standards they are replaced by their parents. This crossover resulted in a strong limitation of code growth when compared to standard tree crossover, thus supporting the defense theory, but Luke suggests that code growth is just being delayed by the large amount of parents replicated along the generations [65, 66]. Additional experiments by Soule and Heckendorn using single node mutations have however suggested that code growth does occur in response to destructive operators [112].

Luke indeed rejects the defense theory in the context of tree-based GP [65] by using a simple procedure called *marking* [15]. Inviable code is identified and marked so that individuals cannot perform crossover within the inviable regions, thus removing the hypothetical advantage conferred by intron multiplication. The results showed a significant reduction of inviable code, but unoptimized code caused tree growth to persist and even increase. The defense theory seems to be correct when applied to those "syntactically redundant constructions" that Angeline called introns, but clearly does not apply to unoptimized code in tree-based GP. And even in linear GP, Brameier and Banzhaf have recently identified neutral crossover, not destructive crossover, as the main cause of code growth [17].

2.1.3 Removal Bias

Although presenting evidence to support the theory of defense against crossover (Section 2.1.2), Soule performed additional experiments with another non-destructive but less "rigorous" hill-climbing crossover [107, 108] (Section 2.2.3). While the previous crossover retained only the offspring that were strictly better than their parents [109], this one retains all the offspring that are equal or better in terms of fitness. Both are non-destructive operators and yet the less rigorous one produces a substantial amount of code growth, although smaller than with standard crossover. Soule concludes that there must be a second cause for code growth besides the defense against crossover, and presents a theory called *removal bias* [59, 107, 108].

Given the general destructive nature of standard crossover, offspring having the same fitness as their parents often benefit from a selective advantage over their siblings. The presence of inviable code provides regions where removal or addition of genetic material does not modify the fitness of the individual. According to the removal bias, to maintain fitness the removed branches must be contained within the inviable region, meaning they cannot be deeper than the inviable subtree. On the other hand, the addition of a branch inside an inviable region cannot affect fitness regardless of how deep the new branch is. This asymmetry can explain code growth, even in the absence of destructive genetic operators. A related explanation had already been advanced by Altenberg (see Section 2.1.1).

When using the more rigorous non-destructive crossover that only allows offspring with better fitness than their parents (Section 2.1.2), removal bias is disabled and code growth effectively drops to a minimum, lending support to the theory. However, Luke holds the argument that an even larger amount of parent replication may be stalling the evolution and that this may be the only cause for the suppression of bloat [65,66]. The improvement of mean population fitness does slow down with the introduction of the more rigorous crossover, but it is still faster than using standard crossover [107], suggesting that the evolution is not hampered by parent replication. Soule and Heckendorn provided additional support to the removal bias by showing that crossover destructiveness is positively correlated with removed branch size, but mostly unaffected by inserted branch size [112].

2.1.4 Fitness Causes Bloat

The first theory that does not make introns responsible for bloat was first advanced by Langdon and Poli [49, 55, 57, 59]. Also called *solution distribution* [108], *diffusion theory* [62,63,115], *drift* [17,112], *nature of search spaces* [85] and *entropy random walk* [60], it has recently been identified simply by its main claim, *fitness causes bloat* [68]. Given its general characteristics, this theory is applicable to any progressive search technique using a discrete variable-length representation and a static evaluation function.

The fitness causes bloat theory basically states that with a variable-length representation there are many different ways to represent the same program, long and short, and a static evaluation function will attribute the same fitness

to all, as long as their behavior is the same. Given the inherent destructiveness of crossover, when better solutions become hard to find there is a selection bias towards programs that have the same fitness as their parents. Because there are many more longer ways to represent a program than shorter ways, a natural drift towards longer solutions occurs, causing bloat. Although this explanation does not directly implicate introns in the process, the odds are that the code growth observed in the progressively longer alternative representations is ultimately caused by introns, either inviable or unoptimized code. Fitness causes bloat is strongly supported by theoretical evidence [58, Chap. 8].

If selection did not punish individuals worse than their parents, there would be no need to search for alternative representations for the same solutions, and bloat would not occur. So, fitness causes bloat. Confirming previous results by Tackett [116], experiments have shown that code growth does not occur when using random selection [10, 55, 60], not even when standard mutation is the only genetic operator [57]. Selection pressure has been further linked to code growth by Gustafson et al., who have found that increased problem difficulty induces higher selection pressure and loss of diversity, which together lead to bloat [37]. Studying bloat from a statistical learning theory viewpoint, Zhang and Mühlenbein have stated that programs tend to grow until they fit the fitness data perfectly [126], and Gelly et al., have also found evidence to support the claim that fitness causes bloat [34, 35].

2.1.5 Modification Point Depth

Another explanation for bloat in tree-based GP was advanced by Luke [62, 63, 65]. It has been called *depth-correlation* theory [115], but can also be referred to as *depth-based* theory or simply *modification point depth* [65].

Confirming previous results [42], Luke has observed that when a genetic operator modifies a parent to create an offspring, there is a correlation between the depth of the modified node and its effect on the fitness of the offspring when compared to the parent: the deeper the modification point, the smaller the change in fitness. Once again, because of the destructive nature of crossover, small changes will eventually benefit from a selective advantage over large changes, so there is a preference for deeper modification points. The larger the individual, the deeper its nodes can be, so large parents have an advantage over small parents. Plus, the deeper the modification point, the smaller the branch that is removed, thus creating a removal bias (Section 2.1.3). This may be regarded as a generalization of the original removal bias theory [112].

Luke denies that introns cause bloat [62]. In fact, according to the theory of modification point depth, size is a consequence of fitness, and Luke adds that size itself is what allows the propagation of inviable code [63, 65, 68]. Streeter also suggests that code growth may be related to a measure of resilience, where resilience is directly related to tree size [115].

2.1.6 Crossover Bias

The most recent theory concerning bloat is the *crossover bias* theory by Poli et al. [24, 25, 89, 90]. It explains code growth in tree-based GP by the effect that

standard subtree crossover has on the distribution of tree sizes in the population. Whenever subtree crossover is applied, the amount of genetic material removed from the first parent is the exact same amount inserted in the second parent, and vice versa. The mean tree size remains unchanged. However, as the population undergoes repeated crossover operations, it approaches a particular distribution of tree sizes (a *Lagrange distribution of the second kind*), where small individuals are much more frequent than the larger ones. For example, crossover generates a high amount of single-node individuals. Because very small individuals are generally unfit, selection tends to reject them in favor of the larger individuals, causing an increase in mean tree size. It is the proliferation of these small unfit individuals, perpetuated by crossover, that ultimately causes bloat. The theory also holds for the popular 10/90% crossover that uses a non-uniform selection of crossover nodes, preferring non-terminal nodes with 90% probability.

Strong theoretical and empirical evidence supports the crossover bias theory. It has been shown that the bias towards smaller individuals is more intense when the population mean tree size is low, and that the initial populations resembling the Lagrange distribution bloat more easily than the ones initialized with traditional methods [24]. A somewhat unexpected finding was that one common bloat control method, the usage of size limits, actually speeds code growth in the early stages of the run. The reason is that size limits promote the proliferation of the smaller individuals, thus biasing the population towards the Lagrange distribution [25]. Along with further theoretical developments, it has also been shown that smaller populations bloat more slowly [90], and that elitism reduces bloat [91].

2.1.7 Discussion

Looking back at all the bloat explanations suggested so far, one cannot help but notice the one thing that all the theories have in common, the one thing that if removed would cause bloat to disappear, ironically the one thing that cannot be removed without rendering the whole process useless: the search for fitness.

Remove fitness from the hitchhiking theory, and redundant code no longer propagates because the building blocks to which it associates cease to be selectively advantageous. Remove fitness from the defense theory, and individuals no longer need protection from a crossover that ceases to be destructive. Remove fitness from the removal bias theory, and the bias to remove small branches disappears. Remove fitness from the fitness causes bloat theory, and the drift towards longer alternative solutions no longer occurs. Remove fitness from the modification point depth theory, and deeper individuals no longer hold any advantage. Remove fitness from the crossover bias theory, and selection no longer rejects the numerous small individuals created by crossover. In short, remove fitness from the search process and bloat vanishes.

All this may sound as obvious as saying that if you climb a mountain high enough you will suffer from lack of oxygen. Altitude causes lack of oxygen, so to avoid it you must not climb. But the goal *is* to climb! And yet the question remains: what causes the lack of oxygen, the particular way you climb or the climbing itself? Can we find a climbing technique to avoid lack of oxygen? Can we find the GP equivalent of the oxygen bottle?

2.2 Taxonomy of Bloat Control Methods

Bloat control methods are so numerous and varied that it is hard to define a taxonomy to classify them, let alone enumerate them all. Because different control measures act at different stages of the evolutionary process, the following taxonomy reflects precisely that: where in the iterative GP process does the method apply. *Evaluation*, *selection*, *breeding* and *survival* are the four evolutionary stages considered. Given the extreme diversity of existent bloat control methods, and to avoid a further discretization of the evolution, some methods are simply included in a heterogeneous group called *others*. Although extensive, the following list of bloat control methods is not exhaustive. Other taxonomies can be found in the literature [58, Sect. 11.6], [67, 68, 127, 128].

2.2.1 Evaluation

Applying bloat control at the level of fitness evaluation is a very common practice, even in other evolutionary computation paradigms besides GP. Like all techniques where the size of an individual affects its probability of being selected for reproduction, the following belong to the wide family of *parsimony pressure* methods.

In *parametric* parsimony pressure, the fitness of each individual is a function of its raw fitness and its size, penalizing larger individuals. It has been used in GP since early times, also under different alternative names and variants like *Occam's razor* and *Minimum Description Length* [14, 40, 44, 45, 111, 126]. Parsimony pressure is usually linear and constant along the evolution, but some techniques apply adaptive pressure where the adjustment of the fitness penalty varies along the evolution [126]. A major difficulty of all parametric methods is precisely its dependency on the correct setting of the parsimony coefficient, worsened by the fact that what is correct in the beginning of the evolutionary process may actually handicap the search later in the evolution. The effects of parametric parsimony pressure have been studied, and it was found that although it may speed the evolution, it may also cause the search process to converge on local optima [96–98, 110].

Different parsimony pressure methods have also been used, like the *tarpeian* method [85], where a fraction of individuals with above-average size are periodically "killed" by giving them such an extreme fitness value that effectively prevents them from being selected for reproduction. Notably one of the few theoretically-motivated bloat control techniques, the tarpeian method has not however been able to beat some older and newer techniques [68].

Most recently, a very efficient way of setting the parsimony coefficient dynamically during the run has been developed for parametric parsimony pressure methods. Based on solid theory, the method achieves complete control over the evolution of the average size of the individuals in the population, even forcing its shrinkage if necessary [92].

2.2.2 Selection

Unlike parametric parsimony, *pareto-based* parsimony does not modify the fitness of the individuals. Instead, parsimony is applied by selecting based on two objectives: fitness and size. Multi-objective parsimony methods do not introduce any parameters to the search process, but unfortunately their results have not been consistently good [12, 21, 22, 29, 54, 68, 82].

Other bloat control methods that act at the selection level include several types of tournament. *Lexicographic parsimony pressure* [67, 68] uses a tournament that always selects the smaller contestant from among individuals having the same fitness. In *double tournament* [66, 68] the contestants are already the winners of a previous tournament, the first based on size and the second based on fitness, or vice versa. In *proportional tournament* [66,68] a proportion of tournaments will select winners based on size instead of fitness. Double tournament has recently proven to be one of the best bloat control methods available [68].

2.2.3 Breeding

Bloat control at the breeding level is performed with specific genetic operators that attempt to restrict code growth. Many different non-standard operators have been advanced, most of them searching for better performance [9, Chap. 6], but some specifically looking to end bloat.

Brood recombination (Section 2.1.1), also called *soft brood selection* and *greedy recombination* [1, 116], is a crossover operator that creates many more offspring than needed, by selecting different crossover points in both parents. From all the individuals created in each crossover, only the best become full-fledged offspring and candidate to the new generation. The number of effective offspring returned by this operator is always the same regardless of the size of the brood. The role of a large brood is to reduce the crossover destructiveness.

One-point crossover [86,87], very similar to *context preserving crossover* [23], chooses crossover points that are common to both parents. Because parents are seldom identical in size and shape, crossover points are restricted to parental structurally identical regions (in terms of function arity from the root node), usually meaning that relatively large subtrees are swapped. A variant named *strict one-point crossover* restricts identical regions to having both the same arity and the exact same functions. Less restrictive than one-point crossover, the *same depths* crossover [106] begins by choosing a random depth in the least deep parent, and then selects random crossover points at this depth, in both parents. This also increases the chances that larger subtrees are swapped. The popular 10/90% crossover that chooses non-terminals with 90% probability [45] is another attempt at swapping larger branches. *Uniform crossover* [88] is similar to one-point crossover in that swapping of genetic material can only be done within common parental regions, and it also includes a strict variant. The difference is that in uniform crossover only single nodes are swapped, not entire branches (except when the swapped node is a non-terminal at the boundary of the common region). *Smooth uniform crossover* (and the related *smooth point mutation*) [81] is a variant that does not swap entire nodes and instead interpolates their behavior to allow smaller movements around the solution space.

One-point, same depth and uniform crossovers all have one thing in common: they maintain the depth of the offspring within the limits established by the initial population.

Size fair crossover [51,52] behaves much like standard tree crossover, except in regard to the choice of the crossover point in the second parent. A bound is placed on the amount of genetic material exchanged in a single operation, so the selected branch on the second parent must not exceed a certain size limit related to the size of the selected branch on the first parent. *Size fair mutation* [49, 59] behaves in a similar manner regarding the size of the newly created subtree. Based on fair crossover, *homologous crossover* [51] attempts at preserving the context in which the subtrees are swapped, by selecting the second branch as similar as possible to the branch in the first parent. The *aligned homologous crossover* [77] and the *maximum homologous crossover* [83] are homologous crossovers specifically developed for linear GP.

Other bloat control operators include a crossover that truncates excess depth [70], and also specialized genetic operators that depend on the size or depth of the parents: large trees are modified with operators likely to reduce their size, while smaller trees are likely to grow [43].

Finally, two techniques where the number of operations performed on a single (pair of) parent(s) is dependent on the size of the breeding trees. *Uniform subtree mutation* [120] applies several rounds of standard mutation to the same individual. The larger the parent tree, the higher the number of operations. Following the same rationale, multiple crossovers have also been used as an attempt to break the tree resilience that is correlated to code growth [114] (Section 2.1.5).

2.2.4 Survival

Bloat control at the survival level can be applied on an individual basis, where each candidate to the new generation must conform to some standards if it is to become an effective population member, or on a population basis, where the population as a whole must obey some restrictions, regardless of the particular characteristics of each individual.

Some of the individual-based techniques are usually described as specific crossover operators, and as such could be classified as breeding restrictions (Section 2.2.3). But this really depends on the implementation details, and it is best to keep things separate. Any genetic operator can be used along with these bloat control techniques, intact and unrestricted. Only after the breeding process is finished are the new individuals filtered as part of the survival process.

Individual-based The first bloat control method ever used on tree-based GP is still the most popular and commonly used. Traditionally it imposes a fixed tree depth limit on the individuals accepted into the population. When a genetic operator creates an offspring that violates this limit, one of its parents is chosen for the new generation instead [45]. Alternatively, the operator can be retried until it produces a valid individual [70], or the invalid individuals accepted but given null fitness [72]. Size limits have been used instead of depth limits, where

size is the number of tree nodes [47, 53, 55, 59]. However, studies have suggested that the usage of size or depth limits can interfere with search efficiency once the average program size approaches the limit, leading to premature convergence [58, Chap. 10], [33,56]. Another study deals with the impact of size limits on the average size of the individuals in the population [72], and recent work related to the crossover bias theory reports that size limits actually speed code growth in the early stages of the run [25] (see Section 2.1.6).

Dynamic Maximum Tree Depth [101] is a bloat control technique inspired in the traditional tree depth limit. It also imposes a depth limit on the individuals accepted into the population, but this one is dynamic and able to increase during the run. Variants of this technique include the implementation of a heavy limit that can also decrease during the run, and the usage of a limit on size instead of depth [102]. Dynamic Maximum Tree Depth and its variants are collectively called *Dynamic Limits* (Chapter 3).

Another technique that replaces offspring with their parents when restrictions are not respected is the *hill-climbing crossover* [107–109] (Section 2.1.2), also called *pseudo-hillclimbing* [66, 68]. In this technique, only the individuals that are not worse than their parents, or are strictly better than their parents, are allowed to enter the population. Individuals who do not conform to these standards are replaced by their parents. This technique counteracts the natural destructiveness of crossover. An identical technique is called *improved fitness selection*, while *changed fitness selection* only accepts individuals with a different fitness from their parents, better or worse [106]. A similar approach at diversity pressure has been used in a different manner, much like parametric parsimony pressure (Section 2.2.1), by including a penalty on the fitness of the offspring that have the same fitness as their parents [50].

Developed alongside the crossover bias theory (Section 2.1.6), a new method for bloat control has been proposed, called *operator equalisation* [26]. It is capable of accurately controlling the distribution of sizes inside the population by probabilistically accepting each individual based on its size, where the probabilities are calculated considering the target distribution. This method can be used to counteract the crossover bias that ultimately causes bloat.

Population-based The first attempt to control bloat using restrictions at the population level was made with the implementation of a fixed limit on the total number of nodes of the entire population [125]. This idea was further developed and tree nodes were regarded as the natural resources that individuals need to survive [103]. Its hybridization with Dynamic Limits (Chapter 3) resulted in the implementation of a dynamic limit on the amount of resources the population can use [104,105], whose variations depend on the evolution of mean population fitness. The concept of natural resources in GP was named *Resource-Limited GP* (Chapter 4). The introduction of limits at the population level results in automatic population resizing.

Simpler approaches like the systematic shrinking of the population have also been used in order to save computational effort and thus counter the effects of bloat [30, 31, 64]. More sophisticated, another variable population approach is implemented by explicitly inserting or deleting individuals depending on how

the best fitness is evolving [121, Sect. 7.1] [19, 94, 95, 118]. Individuals are suppressed as long as the best individual in the population keeps improving, and new individuals are added when the best fitness stagnates.

2.2.5 Others

Some bloat control methods do not really fit into any of the previous categories, like the *waiting room* and *death by size* [68, 82]. The waiting room implements a queue where newly created individuals must wait until they can enter the population. The larger the individual, the longer it must wait. Death by size is a technique designed specifically for steady-state GP, as opposed to generational GP. At each time step some individuals are selected to be removed from the population and replaced by the new children. Larger individuals are more likely to be removed.

Code editing is the plain removal of redundant code, both in tree-based and in linear GP. It has been performed since early times to clean and simplify the final solution [45], and has also been used along the evolutionary process as an attempt to counteract bloat. Code editing can be done before [16] or after [41] evaluating an individual, with a mutation operator that simplifies [28] or a crossover operator that deletes redundant regions [13] or through any other simplification system that acts periodically on the individuals [39,111]. However, it has been shown that code editing can lead to premature convergence [38].

Explicitly defined introns are intended as substitutes of naturally occurring introns [5, 13, 60, 78, 79, 106]. This technique consists on the inclusion of special nodes with added functionality that adapt the likelihood of crossover or mutation to operate at specific locations within the code.

Dynamic fitness, where the fitness measure is based on co-evolution, or calculated on a variable set of fitness cases, has been extensively used as an attempt to improve the convergence ability of GP [48, Sect. 2.4.2], and only rarely as a bloat control technique [50].

Although not initially developed as a bloat control method, promoting the modularization and reusability of GP structures often results in more parsimonious code. Some modularization techniques are the creation of Automatically Defined Functions [46] and Automatically Defined Macros [113], Module Acquisition [7] and Adaptive Representation Learning [99].

Finally, some less common forms of GP seem to have the ability of reducing code growth, like *relaxed GP* [20], or do not seem to be affected by bloat at all, like *stochastic grammar-based GP* [93] and *cartesian GP* [74]. In relaxed GP, the values of the desired solution are *relaxed* so that any value within a specified interval will be considered correct. Stochastic grammar-based GP is a grammar-based GP framework where the information is stored as a probability distribution on the grammar rules, rather than in a population. In cartesian GP, a program is represented as an indexed graph.

Chapter 3

Dynamic Limits

This chapter describes the set of bloat control techniques collectively designated as Dynamic Limits, from the initial idea of applying a dynamic limit to the depth of evolving trees, called Dynamic Maximum Tree Depth [101], to the variants where the limit can be applied to either depth or size, and is allowed to increase or decrease during the run [102].

3.1 Dynamic Maximum Tree Depth

Tree-based GP traditionally uses a depth limit to avoid excessive growth of its individuals. When an individual is created that violates this limit, one of its parents is chosen for the new generation instead [45]. This technique effectively avoids the growth of trees beyond a certain point, but it does nothing to control bloat until the limit is reached. The static nature of the limit may also prevent the optimal solution to be found for problems of unsuspected high complexity.

3.1.1 Dynamic Depth Limit

Dynamic Maximum Tree Depth [101] is a bloat control technique inspired in the traditional static limit. It also imposes a depth limit on the individuals accepted into the population, but this one is dynamic, meaning it can be changed during the run. The dynamic limit is initially set with a low value, but at least as high as the maximum depth of the initial random trees. Any new individual who breaks this limit is rejected and replaced by one of its parents instead (as with the traditional static limit), unless it is the best individual found so far. In this case, the dynamic limit is raised to match the depth of the new best-of-run and allow it into the population. The result is a succession of limit risings, as the best solution becomes more accurate and more complex.

Dynamic Maximum Tree Depth does not necessarily replace the traditional depth limit – both dynamic and fixed limits can be used at the same time. When this happens, the dynamic limit always lies somewhere between the initial tree depth and the fixed depth limit. The simplicity of Dynamic Maximum Tree Depth make it easy to use with any set of parameters and/or coupled with other techniques for controlling bloat (Section 2.2).

The dynamic limit may also be used for another purpose besides controlling bloat. In real world applications, one may not be interested or able to invest a large amount of time in achieving the best possible solution, particularly in approximation problems. Instead, one may only consider a solution to be acceptable if it is sufficiently simple to be understood, even if its accuracy is known to be worse than the accuracy of other more complex solutions. Plus, shorter solutions tend to generalize better (Chapter 2). Choosing less stringent stop conditions to allow the algorithm to stop sooner is not enough to ensure that the resulting solution will be acceptable, as it cannot predict its complexity. By starting with a low dynamic limit for tree depth and repeatedly raising it as more complex solutions prove to be better than simpler ones, the Dynamic Maximum Tree Depth technique can in fact provide a series of solutions of increasing complexity and accuracy, from which the user may choose the most adequate one.

3.1.2 Early Results

Early tests have shown that Dynamic Maximum Tree Depth is able to effectively contain code growth in a Symbolic Regression and the Even-3 Parity problems [101]. Two different settings for the initial value of the dynamic limit were tried (6 and 9), the lowest being exactly the maximum depth of the initial random trees. This most restrictive value resulted in lower mean tree size along the run without any impairment on the ability to converge to good solutions. Dynamic Maximum Tree Depth was also tested against and together with another bloat control technique, Lexicographic Parsimony Pressure (Section 2.2.2). The experiments showed a clear superiority of the dynamic limit, with the best results achieved when both techniques were coupled together.

3.2 Variations on Size and Depth

The original Dynamic Maximum Tree Depth was soon extended to include additional functionalities: a *heavy* variation of the dynamic limit that can be lowered as well as raised, and a dynamic limit on *size* instead of depth.

Figure 3.1 shows the general acceptance procedure (including all the variants) that all newly created individuals must pass before being accepted into the new generation. Any individual that does not meet the size/depth/fitness requirements of the Dynamic Limits method will not be accepted by this procedure, but instead replaced by one of its parents.

3.2.1 Heavy Dynamic Limit

Dynamic Maximum Tree Depth is capable of withstanding a considerable amount of parsimony pressure, as proven by the results obtained by initializing the dynamic limit with the lowest possible value, the maximum depth of the initial random trees [101] (Section 3.1.2). So there seems to be no reason why the limit should not be allowed to fall back to lower values in case the depth of the new best individual becomes lower than the current limit, an occurrence which

```
for all newly created individuals

  size_i = size (depth) of individual
  fitness_i = fitness of individual

  if size_i ≤ dynamic_limit
    accept individual

    if fitness_i > best_fitness
      best_fitness = fitness_i

    if VeryHeavy
      or Heavy and size_i ≥ initial_dynamic_limit
        dynamic_limit = size_i

  if size_i > dynamic_limit and fitness_i > best_fitness
    accept individual

    best_fitness = fitness_i
    dynamic_limit = size_i
```

Figure 3.1: Pseudo code of the Dynamic Limits acceptance procedure.

is actually very common. So the first variation introduced to the original Dynamic Maximum Tree Depth is the *Heavy* dynamic limit, one that accompanies the depth of the best individual, up or down, with the sole constraint of not going lower than its initialization value [102]. An additional variation to the heavy dynamic limit is the *VeryHeavy* limit, one that can even fall back below its initialization value.

As expected, whenever the limit falls back to a lower value, some individuals already in the population immediately break the new limit, becoming 'illegals'. There was a vast range of options to deal with them, the more drastic being their immediate removal from the population, possibly replacing them by new random individuals. However, since these new 'illegals' could be the ones who managed to produce the new best individual, this did not seem like a very good idea. A much softer option was adopted: the 'illegals' are allowed to remain in the population as if they were not breaking the limit, but when breeding, their children cannot be deeper than the deepest parent. This naturally and gradually places the population within limits again.

3.2.2 Dynamic Size Limit

Even though bloat is known to affect many other search processes using variable-length representations (Chapter 2), depth limits cannot be used on non tree-based GP systems. Extending the idea of a dynamic limit to other domains must begin with the removal of the concept of depth, replacing it with the concept of size. The second variation on the original Dynamic Maximum Tree Depth is

the usage of a dynamic *size* limit, where size is the number of nodes [102]. If a static limit is to be used along with this dynamic limit, it should also be on size, not depth.

When using the dynamic size limit, it makes no sense to keep using depth as a restriction on tree initialization. So a modified version of the Ramped Half-and-Half initialization method [45] was created [102], where an equal number of individuals are initialized with sizes ranging between 2 and the initial value of the dynamic size limit. For each size, an equal number of individuals are initialized with the Grow method and with the Full method [45], that have also been modified to fit the size constraints only. In the modified Grow method, the individual grows by addition of random nodes (internal or terminal) without exceeding the maximum size specified; the modified Full method chooses only internal nodes until the size is close to the specified, and only then chooses terminals. Unlike the original Full version, it may not be able to create individuals with the exact size specified, but only close (and never exceeding).

3.2.3 Early Results

Both heavy and size variations have been tested on the same problems as the original Dynamic Maximum Tree Depth technique (Symbolic Regression and Even-3 Parity) against and coupled with Lexicographic Parsimony Pressure [102] (Section 3.1.2). The heavy dynamic limit adds parsimony pressure during the run. Even without taking drastic measures towards the individuals that suddenly break the lower limit, the mean tree size along the run was kept significantly lower than with either the original Dynamic Maximum Tree Depth or Lexicographic Parsimony Pressure alone. Once again the best results were achieved by joining both techniques, and fitness was still not affected by such high levels of parsimony pressure. The dynamic size, however, did not perform as well in one of the problems (Parity), where the ability to find good solutions was compromised. These early results did not yet include the VeryHeavy variation.

Chapter 4

Resource-Limited GP

This chapter describes the concept and implementation details of Resource-Limited GP, from the initial idea of replacing tree depth/size limits at the individual level by a global limit on the resources used by the entire population [103], to the inspiration on Dynamic Limits to create a dynamic resource limit [104], and necessary comparison with the original Dynamic Limits individual-level approach [105].

4.1 Replacing Tree Depth Limits

Dynamic Limits can effectively control bloat without impairing performance (Chapter 3). However, previous attempts at applying them to size instead of depth were somewhat unsuccessful [102], rendering them useless for non tree-based GP systems.

4.1.1 Static Resource Limit

Resource-Limited GP is based on a single limit imposed on the amount of resources available to the entire GP population, where resources are the tree nodes or other elements in non tree-based GP, like code lines. We can think about it as limiting the amount of natural resources available to a given biological population, where each individual competes with the others for its share, and the weakest individuals perish when resources are scarce. In Resource-Limited GP, resources become scarce when the total number of nodes in the population exceeds the predefined limit. Beyond this point, not all offspring are guaranteed to be accepted into the new generation. The allocation of resources to individuals (ensuring their survival) is mainly based on fitness, with size playing a secondary role.

The candidates to the new generation are the offspring, followed by their parents. Each of these groups is sorted by fitness, regardless of size. The queued candidates are then given the resources they need (their number of nodes) in a first come, first served basis. The individuals requiring more resources than the amount still available are skipped (do not survive) and the allocation continues until the end of the queue, or until population size restrictions apply. Some

resources may remain unused. Some parents may survive while their offspring perish. A rule emerges from this procedure, promoting the survival of the best individuals and the rejection of 'not good enough for their size' individuals, where the relationship between size and fitness is not explicitly programmed, but a product of the evolutionary process.

Resource-Limited GP removes most of the disadvantages of using depth limits at the individual level, while introducing automatic population resizing, a natural side-effect of using an approach at the population level. When the resource limit is reached, and as long as code growth continues, the population size (defined as the number of individuals) begins to steadily decrease, something that may actually improve convergence to good solutions [30,31,64]. It is possible that a single individual may have to be artificially kept in the population to avoid extinction, but this risk is non-existent if tree crossover is the only genetic operator used. After the resources have reached the exhaustion point and the population size has been reduced, eventually some new generation of individuals will use them more sparingly and leave enough unused to allow the population size to increase again. Two implementation options have been considered on how to deal with this occurrence: After accepting as many individuals as the previous population size, (1) use the remaining resources to allow the survival of additional individuals of the previous generation - the parents who have not yet been accepted - by continuing the resource allocation procedure until the resources are exhausted, or until the initial population size is reached, or (2) do not use them, thus never allowing the population size to increase. The first option was designated as *Steady*, for it enforces a steady usage of resources, and the second was called *Low*, because it allows a possible low usage of resources. Figure 4.1 shows the pseudo code of the resource allocation procedure. See Figure 4.3 for an example.

4.1.2 Early Results

Resource-Limited GP was tested on a simple Symbolic Regression problem. To compare its performance with the traditional usage of depth limits at the individual level, a static resource limit (14500) was found that would provide an amount of cumulative resources (used during the entire run) similar to the cumulative amount obtained when using the traditional static depth limit of 17 introduced by Koza [45]. The results showed that the Steady and Low techniques behaved in a similar manner, achieving the same performance as the depth limit, although using different bloat control strategies and producing radically different evolutionary dynamics [103]. Resource-Limited GP was a successful replacement for the popular tree depth limit.

4.2 The Dynamic Approach

Like the traditional depth limit, the original Resource-Limited GP relies on a static limit, imposed in the beginning of the run and never changed until the end. This hardly reflects the needs of a search process that must grow its individuals in search of better solutions. Although Resource-Limited GP has

```
sort offspring by fitness
sort parents by fitness
list = offspring followed by parents

if Steady, my_popsize = initial_popsize
if Low,    my_popsize = previous_popsize
```

```
resources_used = 0
accept_list = empty

for all individuals in list

  resources_i = resources needed by individual

  if resources_used + resources_i ≤ resource_limit
    accept_list = accept_list + individual
    resources_used = resources_used + resources_i

  if length of accept_list = my_popsize
    break for

new generation = accept_list
```

Figure 4.1: Pseudo code of the resource allocation procedure.

the natural ability to compensate higher tree size with lower population size, in complex problems this may lead to a dangerous shrinking of population size, as code growth proceeds. On the other hand, providing enough static resources to last until the end of the run may lead to the occurrence of bloat from the very beginning. The need for a dynamic resource limit becomes obvious.

4.2.1 Dynamic Resource Limit

The dynamic approach to Resource-Limited GP naturally arises from the hybridization of Dynamic Limits with the original static resource limit. A dynamic resource limit is implemented, one that is initially set with a low value, and raised whenever it results in better mean population fitness.

After generating the offspring, the candidates to the new generation are sorted and given the available resources, following the procedure described in Section 4.1 and Figure 4.1. The allocation continues until the resources are exhausted, or until the initial population size is reached, according to the Steady option. It is also possible to stop allocating once the previous population size is reached, according to the Low option. So far, this is the original Resource-Limited GP, but now comes the decision on whether to raise the resource limit.

The rejected individuals are now given a second chance. In turn, each of them is reconsidered as a candidate for the new generation, and as many as

```
if DynRes       , my_meanpopfit = best_meanpopfit
if DynResLight , my_meanpopfit = previous_meanpopfit
```

```
reject_list = list - accept_list
current_meanpopfit = mean fitness of accept_list

if current_meanpopfit better than best_meanpopfit
  best_meanpopfit = current_meanpopfit

for all individuals in reject_list

  tmp_accept_list = accept_list + individual
  new_meanpopfit = mean fitness of tmp_accept_list

  if new_meanpopfit better than my_meanpopfit
    accept_list = tmp_accept_list
    resources_i = resources needed by individual
    resources_used = resources_used + resources_i

  else
    break for

  if length of accept_list = initial_popsize
    break for

new generation = accept_list

if resources_used > resource_limit
  or Heavy and resources_used ≥ initial_resource_limit
  or VeryHeavy
    resource_limit = resources_used
```

Figure 4.2: Pseudo code of the reselection procedure.

possible successive individuals are accepted, as long as their inclusion causes an improvement of the mean population fitness. This improvement may be relative to the best mean population fitness of the run, or to the mean population fitness of the previous generation, creating two different implementation options called *DynRes* (for dynamic resources) and *DynResLight*, respectively. DynResLight is expected to implement a limit that is raised much easier, hence the name. As soon as one of the previously rejected individuals is rejected again, the process of reselection stops and the resource limit is increased to provide the additional needed resources.

As in Dynamic Limits, Resource-Limited GP also includes a *Heavy* limit that falls back to lower values when resources remain unused, and a *VeryHeavy* limit that can even fall below its initialization value. Figure 4.2 shows the pseudo code of the reselection procedure. See Figure 4.3 for an example.

Variables:

```
initial_popsize = 10
previous_popsize = 6
resource_limit = 400
best_meanpopfit = 42
previous_meanpopfit = 35
```

Candidates to the new generation:
(from left to right, 6 children followed by 6 parents,
each group sorted by fitness - higher is better)

Id	C1	C2	C3	C4	C5	C6	P1	P2	P3	P4	P5	P6
Fitness	80	70	60	60	50	10	90	40	40	20	10	10
Size	90	100	50	100	80	80	80	70	70	10	20	10

New generation obtained with each technique:
(DynRes and DynResLight begin the reselection
from the Steady results)

Id	C1	C2	C3	C4	C5	C6	P1	P2	P3	P4	P5	P6
Steady	✓	✓	✓	✓	✗[1]	✗[1]	✗[1]	✗[1]	✗[1]	✓	✓	✓
Low	✓	✓	✓	✓	✗[1]	✗[1]	✗[1]	✗[1]	✗[1]	✓	✓	✗[2]
DynRes	✓	✓	✓	✓	✔	✘[3]	–	–	–	✓	✓	✓
DynResLight	✓	✓	✓	✓	✔	✔	✔	✘[4]	–	✓	✓	✓

Reasons for not accepting individual:
[1]Resources not available
[2]`previous_popsize` exceeded - stop procedure
[3]`new_meanpopfit` worse than `best_meanpopfit` - stop procedure
[4]`initial_popsize` exceeded - stop procedure

Figure 4.3: Example of resource allocation and reselection procedures.

4.2.2 Early Results

Tested on two different problems and compared both with the static and dynamic depth limits, the performance of the dynamic variant of Resource-Limited GP (the Steady option) ranged from good to excellent. In a Symbolic Regression problem the technique provided similar fitness with significantly lower resource usage, although the success rate (measured as the percentage of runs that converged to an optimal solution) was a bit lower than using the static resource limit. In a more complex problem of Artificial Ant the dynamic resource limit was able to achieve the same fitness level using a significantly lower amount of resources, with the results showing fine prospects of reaching the optimal much easier than the other techniques [104]. These early results did not yet include the Low option, nor the Heavy or VeryHeavy variants.

4.3 Comparison with Dynamic Limits

Dynamic Limits and Resource-Limited GP operate at different levels of the GP paradigm: one acts at the individual level, the other at the population level. Both have previously achieved promising results in controlling bloat without impairing performance but, because they aim at different targets, they produce different dynamics of the evolutionary process. Which one can achieve better results?

4.3.1 Early Results

Early comparative results between Dynamic Limits and Resource-Limited GP have elected the non-light version of the dynamic resource limit as the best performing technique. On the two simple problems of Symbolic Regression and Even-3 Parity, it managed to reach the same best fitness using significantly less resources than the remaining techniques. On the more complex problem of the Artificial Ant, if reached significantly higher fitness using the same amount of resources. According to the same criteria, the technique scoring the second place was the non-heavy dynamic tree depth. The traditional static depth limit was last. However, unlike Dynamic Limits, Resource-Limited GP did not maintain a constant performance along the evolution. It managed to achieve very good results in the beginning of the run, but the improvements gradually slowed down as the run proceeded, casting a doubt on whether it is in fact the best approach [105].

Chapter 5

Experiments

This chapter introduces the experiments that were performed in order to study the efficiency of Dynamic Limits and Resource-Limited GP as bloat control methods. It provides a description of the problems used, the general parameter settings common to all the techniques, and how the results are presented. The procedures and parameters specifically related to each set of experiments will be detailed in the corresponding later chapters.

All the experiments were performed with tree-based GP in generational mode using the GPLAB toolbox (Section 1.2). Statistical significance of the null hypothesis of no difference was determined with (pairwise) Kruskal-Wallis ANOVAs at $p = 0.01$. A non-parametric ANOVA was used because the data is not guaranteed to follow a normal distribution. For the same reason, the median was preferred over the mean. The median is also more robust to outliers, and has already been used in similar studies [114].

5.1 Problems

Four different problems were chosen to test Dynamic Limits and Resource-Limited GP: Symbolic Regression, Artificial Ant, 5-Bit Even Parity and 11-Bit Boolean Multiplexer. This particular set of problems was chosen because it has been widely used in the literature [19, 20, 29–31, 52, 53, 62–68, 94, 95, 101–105, 118, 120, 121], as it represents a varied selection in terms of bloat dynamics and response to different bloat control techniques.

5.1.1 Symbolic Regression

The goal of the Symbolic Regression problem is to evolve a function that best approximates a set of points. In this particular case, 21 equidistant points of the quartic polynomial $(x^4 + x^3 + x^2 + x)$ in the interval -1 to $+1$ are used. Figure 5.1 plots this function along with the 21 chosen points.

The function and terminal sets for this problem are, respectively, $\{+, -, \times, \div, sin, cos, log, exp\}$ and $\{x\}$ (no random constants are used). The division and logarithm are protected as in [45]: the division returns 1 whenever the denominator is 0, and the argument of the logarithm is always converted to its absolute

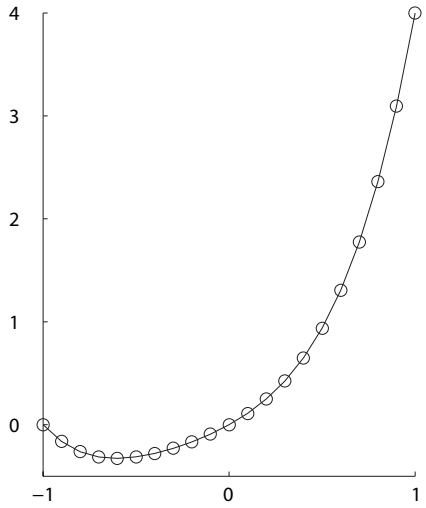

Figure 5.1: Plotting of the quartic polynomial $(x^4 + x^3 + x^2 + x)$ along with 21 equidistant points in the interval -1 to $+1$.

value. Fitness is measured as the sum of the absolute differences between the expected and predicted values of each point. It can take any real non-negative number, so there is a potentially infinite number of possible fitness values. Early results [101, 102] have suggested that the Symbolic Regression problem is not prone to the propagation of inviable code, but very much affected by unoptimized code, although these notions are contradicted in [68]. For simplicity, from now on this problem will be referred to simply as Regression.

5.1.2 Artificial Ant

In the Artificial Ant problem the goal is to evolve a strategy to follow a food trail. In this particular case, the Santa Fe trail is used, shown in Figure 5.2. The trail is represented on a 32×32 (toroidal) grid where black cells are the food pellets and gray cells are the gaps in the trail. The ant begins its search on the upper left corner, facing east.

The function and terminal sets for the Artificial Ant problem are, respectively, $\{$*if-food-ahead*$, progn2, progn3\}$ and $\{left, right, move\}$, as defined in [45]. With the *if-food-ahead* function, the ant checks the cell directly in front of it and performs a certain action in case it finds a food pellet there. $progn2$ and $progn3$ allow the ant to perform any two or three consecutive actions. With the terminals $left$ and $right$ the ant can turn around 90 degrees without moving from its cell. *move* allows the ant to move to the adjacent cell it is facing. When the ant stands on a cell containing a food pellet, it immediately eats it. A foraging strategy is built using these functions and terminals and each ant is given 400 time steps to apply it repeatedly in search of the 89 food pellets available in the trail. Fitness is measured as the number of pellets remaining

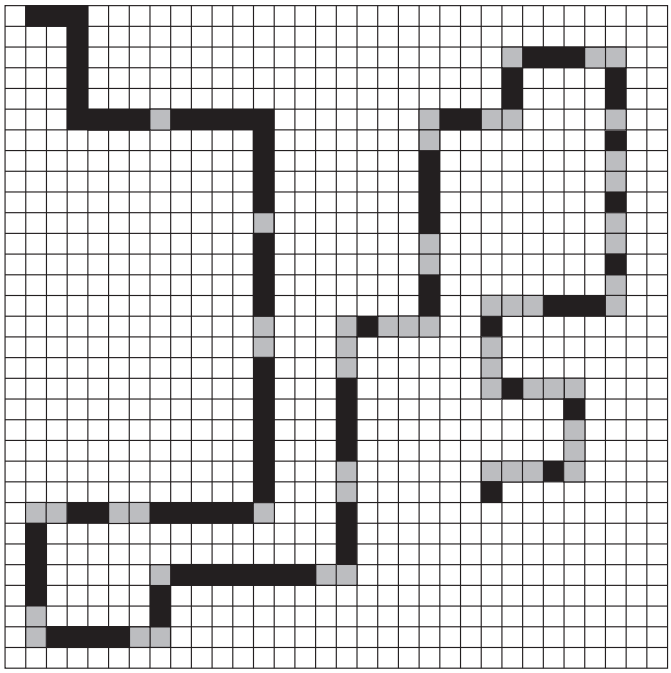

Figure 5.2: Santa Fe food trail for the Artificial Ant problem. Black cells are the food pellets and gray cells are the gaps in the trail.

afterwards. Many different foraging strategies may result in the same fitness, and this seems to be correlated to the proliferation of inviable code.

5.1.3 5-Bit Even Parity

The 5-Bit Even Parity problem is in fact a symbolic regression problem where the function to evolve takes five boolean arguments and returns a single output indicating the parity of the arguments: 1 (or true) if an even number of arguments are 1, and 0 (or false) otherwise.

This problem uses the function and terminal sets $\{and, or, nand, nor\}$ and $\{x_1, \ldots, x_5\}$, respectively. Fitness is measured as the number of misclassified cases, so it may only take values between 0 and 32, even fewer than in the Artificial Ant problem. Once again, a large amount of structurally distinct individuals may have the same fitness. For simplicity, from now on this problem will be referred to simply as Parity.

5.1.4 11-Bit Boolean Multiplexer

Also similar to a symbolic regression problem, the 11-Bit Boolean Multiplexer problem can however be viewed as a problem of electronic circuit design. The function to evolve takes three address arguments (a_0,a_1,a_2) plus eight data arguments (d_0,...,d_7), all boolean. The value returned by the function is the particular data bit that is singled out by the address bits. Figure 5.3 shows

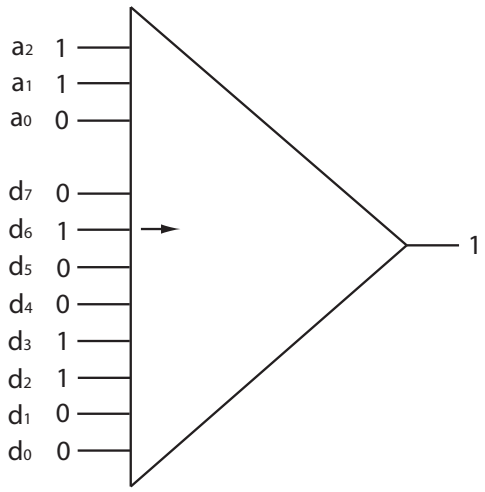

Figure 5.3: Example of the 11-Bit Boolean Multiplexer with the input 11001001100. The first three bits are the address arguments whose binary value indicates the data bit d_6 as the output.

an example where the input is 110 01001100. The first three bits (underlined) are the address arguments $a_2a_1a_0$ whose binary value indicates the data bit d_6 (bold) as the output.

The 11-Bit Boolean Multiplexer problem uses the function and terminal sets $\{and, or, not, if\}$ and $\{a_0, a_1, a_2, d_0, \ldots, d_7\}$, respectively. Note that both address and data arguments are simply treated as terminals, undistinguishable from one another. Fitness is measured as the number of misclassified cases. This theoretically allows fitness values between 0 and 2048, but in practical terms the values usually fall into multiples of 32 [68]. The 11-Bit Boolean Multiplexer problem suffers from relatively little inviable code [68]. For simplicity, from now on it will be referred to simply as Multiplexer.

5.2 Settings

Table 5.1 lists the parameters used in the experiments. A total of 30 runs were performed with each technique for each problem. All the runs used initial populations of 1000 individuals allowed to evolve for as many generations as necessary to consume a certain amount of resources. Since many of the studied techniques dynamically change the number of individuals in the population, a fixed number of generations would not provide a fair comparison of results. Even among the fixed population techniques, some use much fewer resources than others because their individuals remain much smaller along the run, and so they would also require additional generations. A fair comparison is achieved by allowing all the techniques to consume the same amount of resources during the run. Measured as the total number of nodes used by all the individuals in the population along the entire run, this amount is given by the median number of resources used along 50 generations (over 30 runs) by the baseline technique, the

Table 5.1: Settings used in the experiments.

Runs	30
Generations	(variable)
Initial population size	1000
Population initialization	Ramped Half-and-Half
Initial maximum depth	6
Final maximum depth	17 (when applicable)
Selection for reproduction	tournament size 7 (initial)
Genetic operators	tree crossover, no mutation
Reproduction rate	0.1
Selection for survival	no elitism

traditional static limit on tree depth, here identified as Koza. After establishing this amount of resources for each problem, all the runs evolved until reaching it, regardless of the number of generations required, even if an optimal solution was found earlier. The amount of resources given was 1163900 for The Regression problem, 3906594 for the Artificial Ant, 7640390 for the Parity problem, and 5220500 for the Multiplexer.

Most of the remaining parameters follow the settings indicated in [45] and [68]. The initial populations were generated with the Ramped Half-and-Half procedure [45], modified when using the dynamic size limit (see Section 3.2.2, page 19, for details). Although some effort was put into promoting the diversity of the initial population, the tree initialization procedure does not guarantee that all individuals are distinct from one another. For each newly created individual that is structurally identical to any of the members already in the population, the process is retried until a different individual is generated or until 20 attempts have been made. For 1000 individuals with maximum initial depth of 6, this resulted in a diversity of roughly 75% (Regression), 75% (Artificial Ant), 80% (Parity) and 85% (Multiplexer), where diversity is the percentage of distinct individuals in the population (based on the variety measure [48]).

In all four problems, fitness was calculated such that lower values represent better fitness. Selection for reproduction was made with tournaments of size 7, in one case modified to implement a specific technique, double tournament (Section 9.1.2). When using variable size populations the tournament size represents a proportion of the population (0.7%, with a minimum of 2 individuals) so that selection pressure is kept constant when the population is resized.

A reproduction rate of 0.1 was used, meaning that there was a 10% probability of copying an individual intact into the next generation instead of choosing a genetic operator to create new individuals. Standard tree crossover was used, but with uniform distribution of the random crossover points, instead of the more typical 10/90% choice of terminal/internal nodes. It has been suggested that a leaf crossover higher than 10% may be beneficial [5], and total random selection of crossover points may not even affect the results [63]. No mutation was used.

Selection for survival was not elitist (in the traditional sense only, since the techniques using variable size populations can be considered highly elitist, and the reproduction rate is also a form of elitism), meaning that the best individual of a given generation is not guaranteed to survive into the next generation. When using variable size populations, the selection for survival was modified in order to implement the particularities of each technique, where some offspring may not survive their parents, or simply be discarded when the population is resized.

5.3 Plots

The results of each experiment will be presented by four different plots. See, for example, Figure 6.1 (page 36). The first plot presents, for each technique, the amount of resources actually used on each run. The amount of *used* resources is lower than the amount of *given* resources when there is convergence to an optimal solution before the end of the run. In practical terms, when a technique succeeds in finding an optimum it does not need to consume the remaining resources. Techniques that succeed sooner and more often are the ones that save more resources. In this plot, the circles drawn near the top correspond to runs that did not succeed, and so used all the resources given (see Section 5.2 for the amount given to each problem). The numbers above the circles indicate, for each technique, the median number of generations needed to exhaust all the resources given, regardless of the success in finding an optimum. The circles drawn lower correspond to runs that succeeded before exhausting all the resources. The lower the circle, the sooner the run succeeded. The numbers near the bottom indicate the success rate of each technique, calculated as the percentage of runs that found an optimal solution before exhausting the resources.

The second plot presents the evolution of the average tree size inside the population, one line per technique. The median value of the 30 runs is used. Lines that rise faster and higher represent techniques that sooner allow larger individuals into the population. This is directly related to the number of generations needed to end the run, indicated in the previous plot, since larger trees consume more resources, that consequently get exhausted in fewer generations.

One of the most important plots, the third is a boxplot[1] that presents, for each technique, the best fitness achieved by any individual on each run. Any comparative statement regarding the performance of the techniques appearing on this boxplot is supported by statistical evidence. The several techniques are ranked according to this plot whenever there are statistically significant differences between them.

Also very important, the last plot presents the evolution of the best fitness as a function of the resources used, one line per technique. Once again the median value of the 30 runs is used. Lines that drop earlier in the run represent techniques that achieve better fitness with less resources.

[1] In the boxplot each technique is represented by a box and pair of whiskers. Each box has lines at the lower quartile, median, and upper quartile values, and the whiskers mark the furthest value within 1.5 of the quartile ranges. Outliers are represented by $+$, and $\times$ marks the mean.

Chapter 6

Comparison within Dynamic Limits

This chapter studies the efficiency of Dynamic Limits for bloat control. It lists the techniques involved and specifies the procedures and parameters used in the comparisons. Then it presents the results as described in Section 5.3, and concludes by summing up the major findings.

6.1 Techniques

Table 6.1 summarizes all the techniques compared within the Dynamic Limits approach. Koza is the baseline technique, because of its popularity, because all the original techniques of this thesis were directly or indirectly inspired on this traditional static limit (see Section 1.2), and also to stress the improvements introduced when a dynamic limit is used instead. The names (and acronyms) of the other techniques are composed of several parts to help their identification: *Dyn* stands for Dynamic Limits; *Depth (D)* and *Nodes (N)* relate to depth and size limits, respectively; *h* and *vh* identify the respective Heavy and VeryHeavy variants; *l* stands for limited, meaning that a static upper limit is used along with the dynamic limit, in which case the dynamic limit cannot go beyond the static limit. The purpose of testing the limited techniques was to check whether the Dynamic Limits can do without the static upper limit, or still benefit from joining, instead of just replacing, the baseline technique.

6.2 Depth Limits

Table 6.2 specifies the maximum depth/size of the individuals on the initial population, as well as the minimum and maximum limit values, for all the techniques compared within the Dynamic Limits approach. The limit of the Koza technique has the same minimum and maximum value, as it remains static along the run. Heavy and non-heavy variants use the same limit range (hence they appear together), with the only difference that the non-heavy techniques can only increase the limit, while the heavy techniques can also decrease it as

Table 6.1: Techniques compared within the Dynamic Limits approach.

Technique	Acronym	Short description
Koza	K	static depth limit
DynDepth	D	dynamic depth limit
DynNodes	N	dynamic size limit
hDynDepth	hD	heavy dynamic depth limit
hDynNodes	hN	heavy dynamic size limit
vhDynDepth	vhD	very heavy dynamic depth limit
vhDynNodes	vhN	very heavy dynamic size limit
lDynDepth	lD	limited dynamic depth limit
lDynNodes	lN	limited dynamic size limit
lhDynDepth	lhD	limited heavy dynamic depth limit
lhDynNodes	lhN	limited heavy dynamic size limit
lvhDynDepth	lvhD	limited very heavy dynamic depth limit
lvhDynNodes	lvhN	limited very heavy dynamic size limit

low as the maximum depth/size allowed on the initial population. The limit of the very heavy variants has no lower bound. An upper bound exists only in the limited techniques (including Koza), with the traditional value of 17. The maximum depth of the individuals on the initial population is the also traditional value of 6. Whenever a dynamic limit is used, its initial value is exactly the same as the maximum depth/size allowed on the initial population.

6.3 Size Limits

In terms of size instead of depth, appropriate values had to be found that somehow produced the same behavior as their corresponding values for depth. Tree initialization was performed using the modified Ramped Half-and-Half procedure described in Section 3.2.2. The maximum size of the initial individuals is such a number (in multiples of 5) that the median amount of resources used by the initial population is most similar to the median amount used by the corresponding depth-limited initial populations. This value turned out to be different for each problem, because of the distinct function and terminal sets they use: 20 nodes for the Regression problem, 105 nodes for the Artificial Ant, and 50 nodes for the Parity and Multiplexer problems. These were the values that created initial populations using the amount of resources most similar to the amount used by a depth limit of 6. The maximum size limit was chosen to be the rounded value (in multiples of 50) that most closely matches the maximum number of nodes found in 17-depth trees evolved with the baseline technique. Once again such value was different for each problem: 200 nodes for the Regression problem, 500 nodes for the Artificial Ant and Multiplexer problems, and 150 nodes for the Parity problem.

Table 6.2: Limits used within the Dynamic Limits approach.

Technique	Initial population	Minimum limit	Maximum limit
Koza	6	17	17
(h)DynDepth	6	6	-
(h)DynNodes	20/105/50‡	20/105/50‡	-
vhDynDepth	6	-	-
vhDynNodes	20/105/50‡	-	-
l(h)DynDepth	6	6	17
l(h)DynNodes	20/105/50‡	20/105/50‡	200/500/150§
lvhDynDepth	6	-	17
lvhDynNodes	20/105/50‡	-	200/500/150§

‡Regression / Artificial Ant / Parity & Multiplexer
§Regression / Artificial Ant & Multiplexer / Parity

6.4 Results

This section presents the results of the comparisons within the Dynamic Limits approach, divided in the four problems considered (see Section 5.1). The results are first presented without using a static upper limit, and then using the upper limit, except for the baseline technique, Koza. Short concluding remarks are inserted after each problem, highlighting the best performing techniques and describing how the introduction of the upper limit affected the performance.

6.4.1 Symbolic Regression

Without upper limit Figure 6.1 shows the results of the comparison among the dynamic limit techniques on the Regression problem, without using an upper limit. The first plot (a) presents the resources used and the number of generations needed to complete the run, as well as the success rate, for each technique. It reveals that convergence to an optimal solution often happens in the Regression problem, and when it does it occurs very early in the run. The success rates were lower for the dynamic size techniques (7–17%) than for the baseline Koza (40%) or the dynamic depth techniques (23–47%). Koza was the technique that used the most resources per generation, requiring fewer generations than the other techniques to exhaust the amount given. In this case, it used the expected number of exactly 50 generations (see Section 5.2), although this does not always happen. vhDynNodes was the most sparing technique, taking the highest number of generations to exhaust the resources (92), followed closely by the dynamic depth techniques (84–90).

The second plot (b) presents the growth of the average tree size along the run, for each technique. It shows a much quicker and larger growth in Koza than in any other technique, the reason why the resources were exhausted in

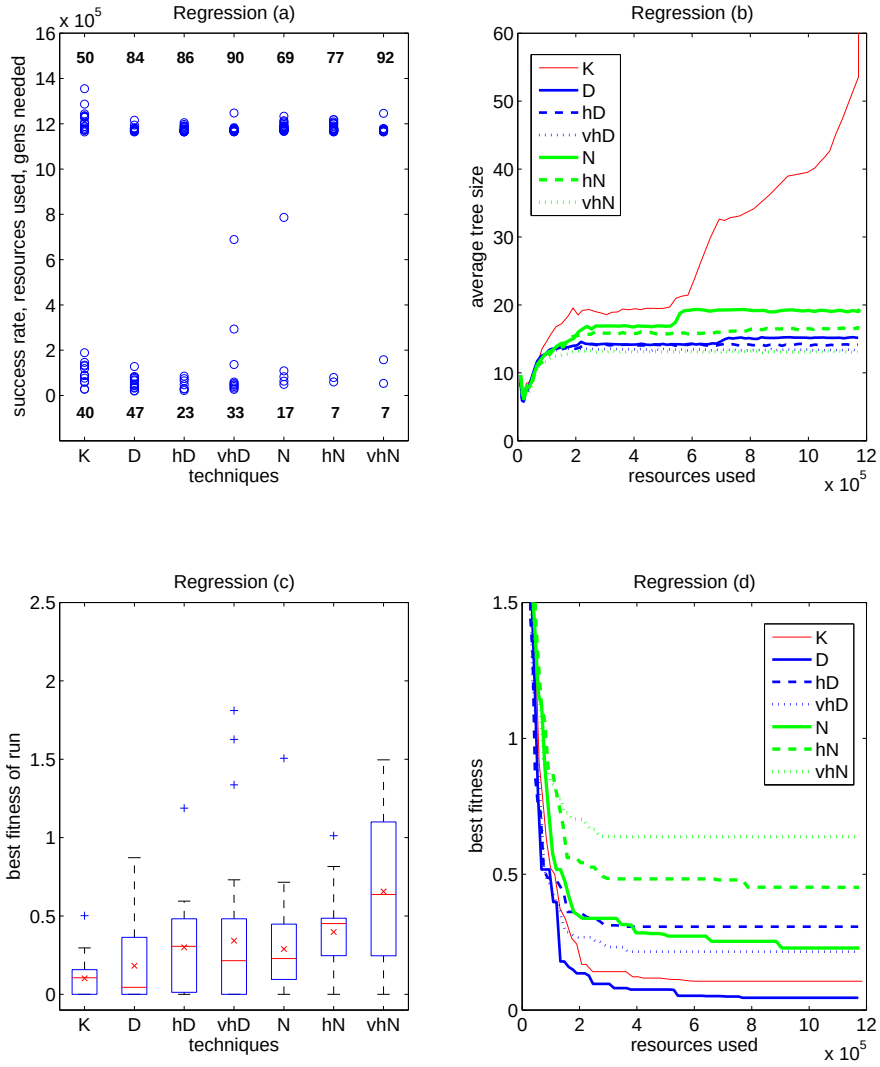

Figure 6.1: Results of the dynamic limit techniques on the Regression problem, without upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

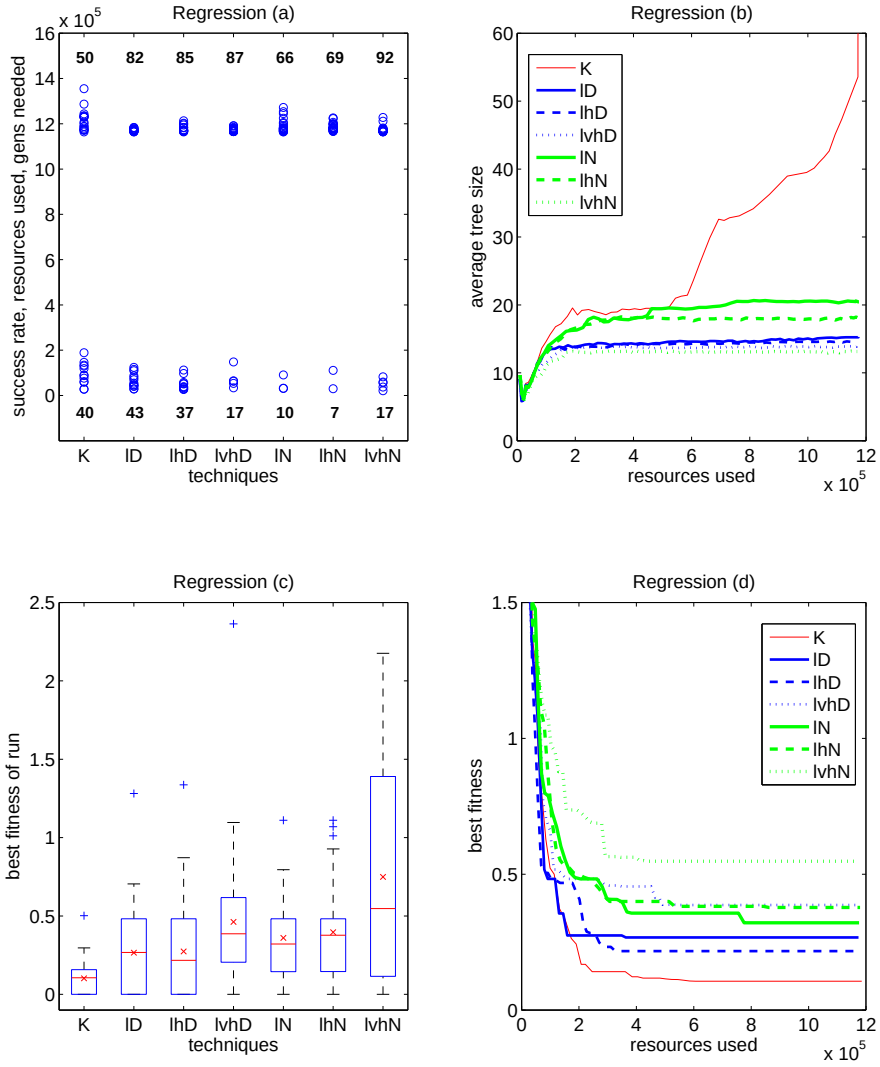

Figure 6.2: Results of the dynamic limit techniques on the Regression problem, with upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

fewer generations. The other techniques did not differ much from each other in terms of growth of average tree size.

The third plot (c) is a boxplot of the best fitness of run obtained with each technique. Only DynDepth and vhDynDepth were not worse than the baseline, Koza. There were no significant differences among the dynamic depth techniques. Among the dynamic size techniques, the performance seemed to degrade as the parsimony pressure increased, with vhDynNodes performing significantly worse than DynNodes. In fact, vhDynNodes was significantly worse than all other techniques except hDynNodes. hDynNodes was worse than DynDepth. Based on these results, a rough ranking of the several techniques could be: 1) Koza, DynDepth and vhDynDepth, 2) hDynDepth, DynNodes and hDynNodes, 3) vhDynNodes.

The last plot (d) presents the evolution of the best fitness as a function of the resources used, for each technique. DynDepth achieved better fitness using less resources, early in the run, followed closely by Koza.

With upper limit Figure 6.2 shows the results of the comparison among the dynamic limit techniques on the Regression problem, using a static upper limit. The first plot (a) once again reveals that, when convergence to an optimum happens, it happens early in the run. The success rates were again lower for the dynamic size techniques (7–17%) than for the baseline Koza (40%) or the dynamic depth techniques (17–43%). Once again, Koza was the technique that used the most resources per generation, requiring the fewest generations to end the run (50). As before, lvhDynNodes was the technique requiring the most generations (92), once again followed by the dynamic depth techniques (82–87).

The second plot (b) again shows a much quicker and larger growth in Koza than in any other technique. The remaining techniques once again did not differ much from each other.

The third plot (c) is the boxplot. The results were somewhat similar to the previous experiment. Only lDynDepth and lhDynDepth were not worse than the baseline. There were no significant differences among the dynamic depth techniques, neither among the dynamic size techniques. lvhDynNodes performed significantly worse than lDynDepth and lhDynDepth. Based on these results, a rough ranking of the several techniques could be: 1) Koza, lDynDepth and lhDynDepth, 2) lvhDynDepth and all dynamic size techniques.

In the last plot (d), Koza and lDynDepth improved fitness sooner in the run. Even though Koza appears to have achieved better values, the difference is not significant.

Remarks The above results show that Koza and (l)DynDepth are the best bloat control techniques in the Regression problem. The dynamic size techniques tend to perform worse. The introduction of the static upper limit increased some success rates but lowered others (although the general tendency was maintained), and did not cause any significant changes in the best fitness of run.

6.4.2 Artificial Ant

Without upper limit Figure 6.3 shows the results of the comparison among the dynamic limit techniques on the Artificial Ant problem, without using an upper limit. The first plot (a) presents the resources used and the number of generations needed to complete the run, as well as the success rate, for each technique. It reveals that convergence to an optimum often happens in the Artificial Ant problem, anytime during the run. The success rates were the highest for DynDepth, DynNodes and vhDynNodes (30%), followed by hDynNodes (23%) and hDynDepth (13%), and finally Koza and vhDynDepth (10%). Koza was the technique that exhausted the resources in the fewest generations (51), followed closely by DynNodes and hDynNodes (56–57). vhDynNodes took the highest number of generations (90).

The second plot (b) presents the growth of the average tree size along the run, for each technique. All the techniques present a drop of average tree size after an initial growth, some more markedly than others. Koza was the technique allowing the largest individuals in the population, with a steep growth of average tree size until the end of the run. The dynamic depth techniques followed the same behavior, but with much lower values. The dynamic size techniques, particularly DynNodes and hDynNodes, present a steep increase of average tree size in the beginning, but with a tendency for stabilization later in the run. vhDynNodes had the same behavior, but with much lower values. All in all, by the end of the run Koza reached the highest average tree size, and vhDynNodes the lowest, with all other techniques presenting values similar to each other.

The third plot (c) is a boxplot of the best fitness of run obtained with each technique. There were no significant differences between the techniques, except for hDynDepth being significantly worse than vhDynNodes. A rough ranking of the techniques could be: 1) all techniques except 2) hDynDepth.

The last plot (d) presents the evolution of the best fitness as a function of the resources used, for each technique. vhDynNodes achieved better fitness using less resources.

With upper limit Figure 6.4 shows the results of the comparison among the dynamic limit techniques on the Artificial Ant problem, using a static upper limit. The first plot (a) once again reveals that convergence to an optimum often happens in the Artificial Ant problem, anytime during the run. The success rates were the highest for the dynamic size techniques (23–33%), followed by Koza and the dynamic depth techniques (10–23%). Once again Koza was the technique that exhausted the resources in the fewest generations (51), followed closely by lDynNodes and lhDynNodes (56–57), as in the previous experiment. Also as before, lvhDynNodes took the highest number of generations (87).

The second plot (b) shows similar behavior to what had been observed without using the upper limit, with Koza reaching the highest average tree size by the end of the run, and lvhDynNodes the lowest.

The third plot (c), the boxplot, presents no significant differences between any of the techniques, therefore no ranking is possible.

The last plot (d) shows that all the dynamic size techniques were able to improve fitness sooner than the remaining techniques.

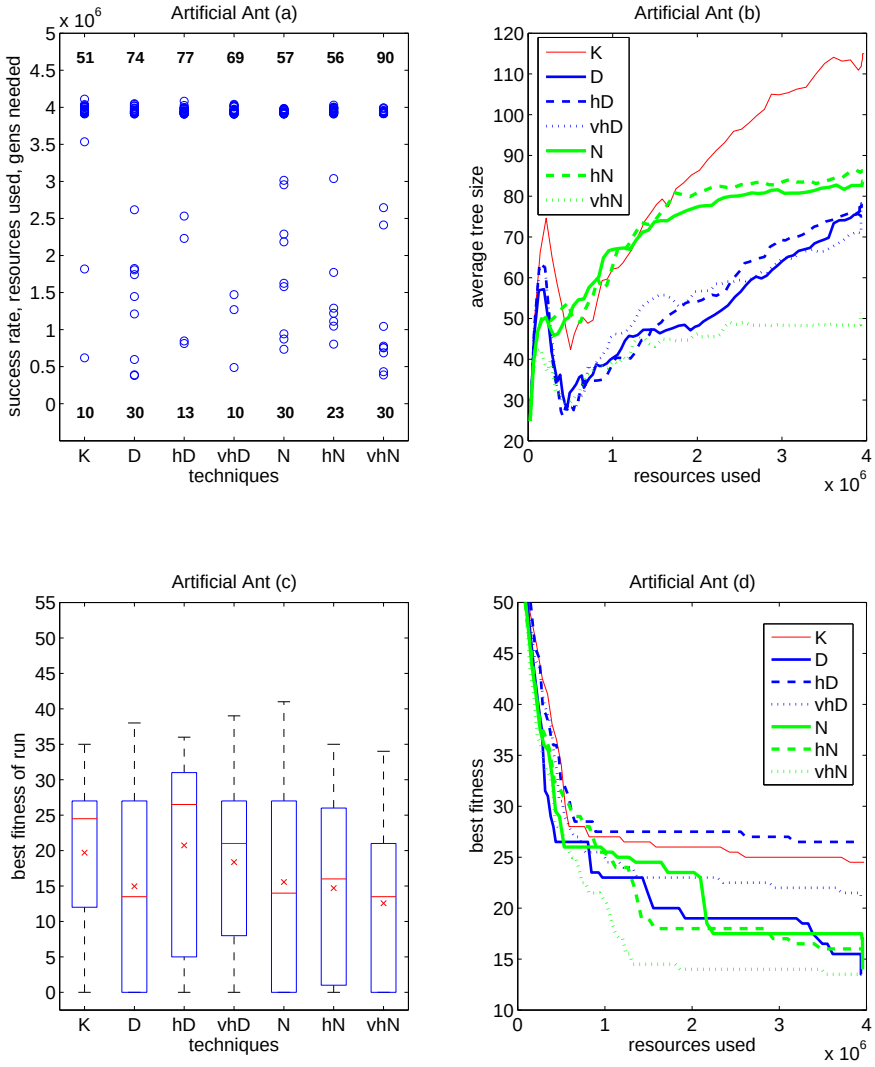

Figure 6.3: Results of the dynamic limit techniques on the Artificial Ant problem, without upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

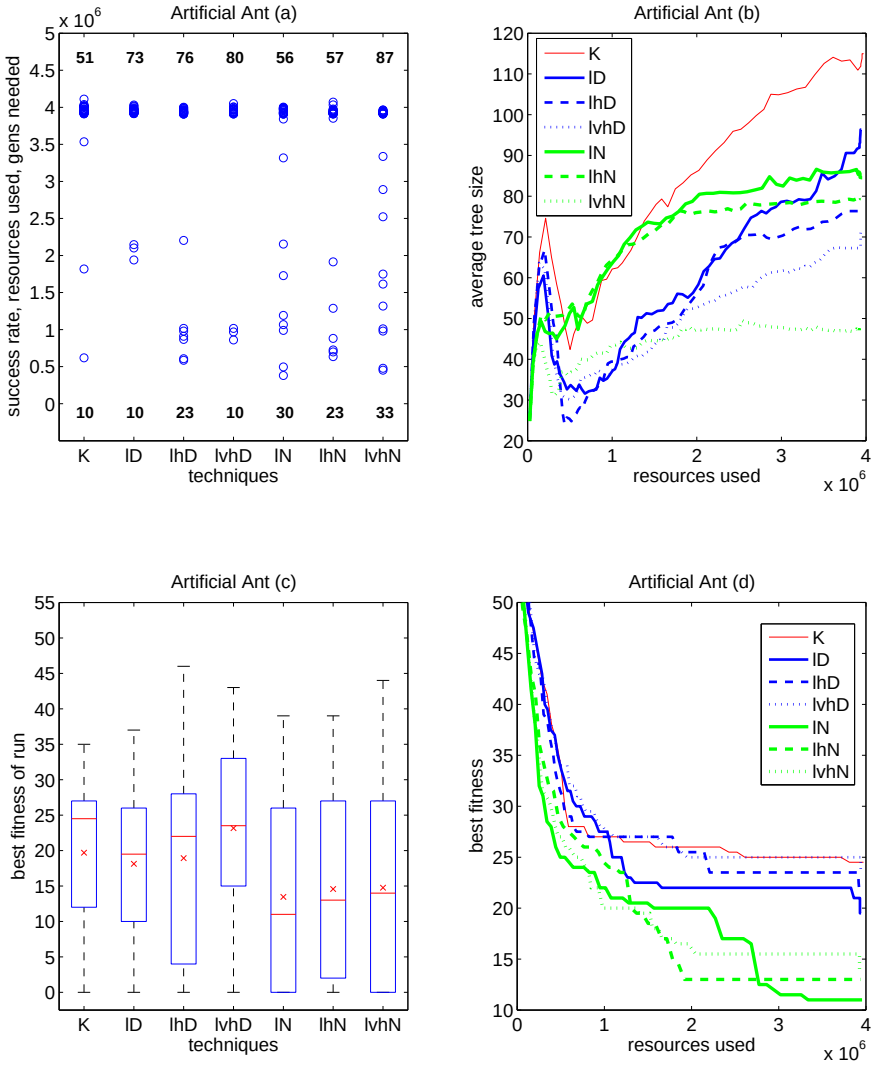

Figure 6.4: Results of the dynamic limit techniques on the Artificial Ant problem, with upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

Remarks The above results suggest that, although the differences in best fitness of run were not significant (except one), the dynamic size techniques have a better performance in the Artificial Ant problem, converging to an optimal solution more often and improving fitness more quickly. In addition, the (l)vhDynNodes techniques are also able to maintain a lower average tree size. The introduction of the static upper limit harmed the DynDepth technique by lowering its success rate, although the best fitness of run did not suffer any significant changes.

6.4.3 5-Bit Even Parity

Without upper limit Figure 6.5 shows the results of the comparison among the dynamic limit techniques on the Parity problem, without using an upper limit. The first plot (a) presents the resources used and the number of generations needed to complete the run, as well as the success rate, for each technique. There was no success in finding an optimal solution for the Parity problem, except once. Koza was the technique that exhausted the resources in the fewest generations (48), followed by the dynamic depth techniques (64–71) and the dynamic size techniques (97–136), with vhDynNodes being the most sparing technique.

The second plot (b) presents the growth of the average tree size along the run, for each technique. Koza presents the largest growth, followed by the dynamic depth techniques and finally the dynamic size techniques, the last ones never increasing the average tree size much, and stabilizing its growth early in the run.

The third plot (c) is a boxplot of the best fitness of run obtained with each technique. hDynDepth was the only technique that reached significantly better fitness than Koza. Among the dynamic depth techniques, hDynDepth also performed significantly better than vhDynDepth. There were no significant differences among the dynamic size techniques, but hDynNodes and vhDynNodes were outperformed by both DynDepth and hDynDepth. Based on these results, a rough ranking of the several techniques could be: 1) DynDepth, hDynDepth, DynNodes 2) Koza, vhDynDepth, hDynNodes and vhDynNodes.

The last plot (d) presents the evolution of the best fitness as a function of the resources used, for each technique. hDynDepth was able to achieve better fitness early in the run, sparing resources.

With upper limit Figure 6.6 shows the results of the comparison among the dynamic limit techniques on the Parity problem, using a static upper limit. The first plot (a) once again shows that there was no success in finding an optimal solution for the Parity problem. Once again Koza was the technique that exhausted the resources in the fewest generations (48), again followed by the dynamic depth techniques (64–70) and the dynamic size techniques (112–145), with lvhDynNodes being the most sparing technique.

The second plot (b) shows similar behavior to what had been observed without using the upper limit, with Koza presenting the largest growth, followed by the dynamic depth techniques and finally the dynamic size techniques, the last

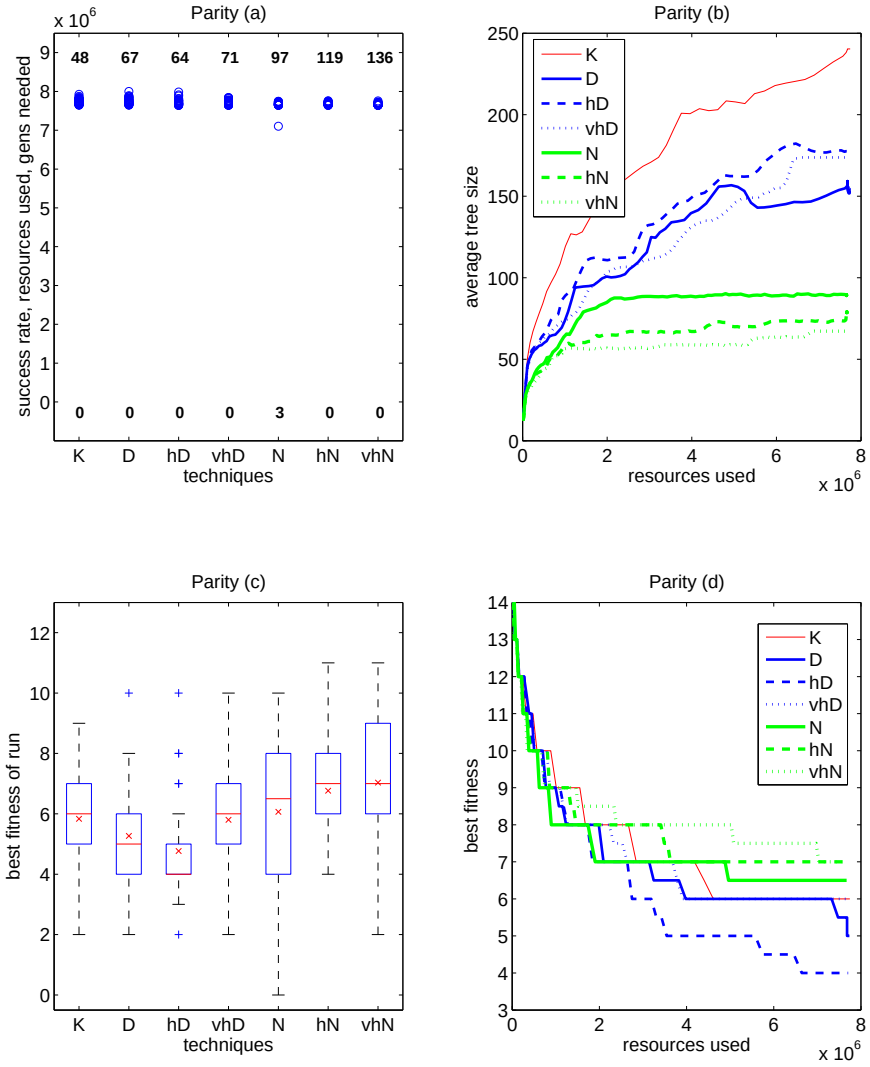

Figure 6.5: Results of the dynamic limit techniques on the Parity problem, without upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

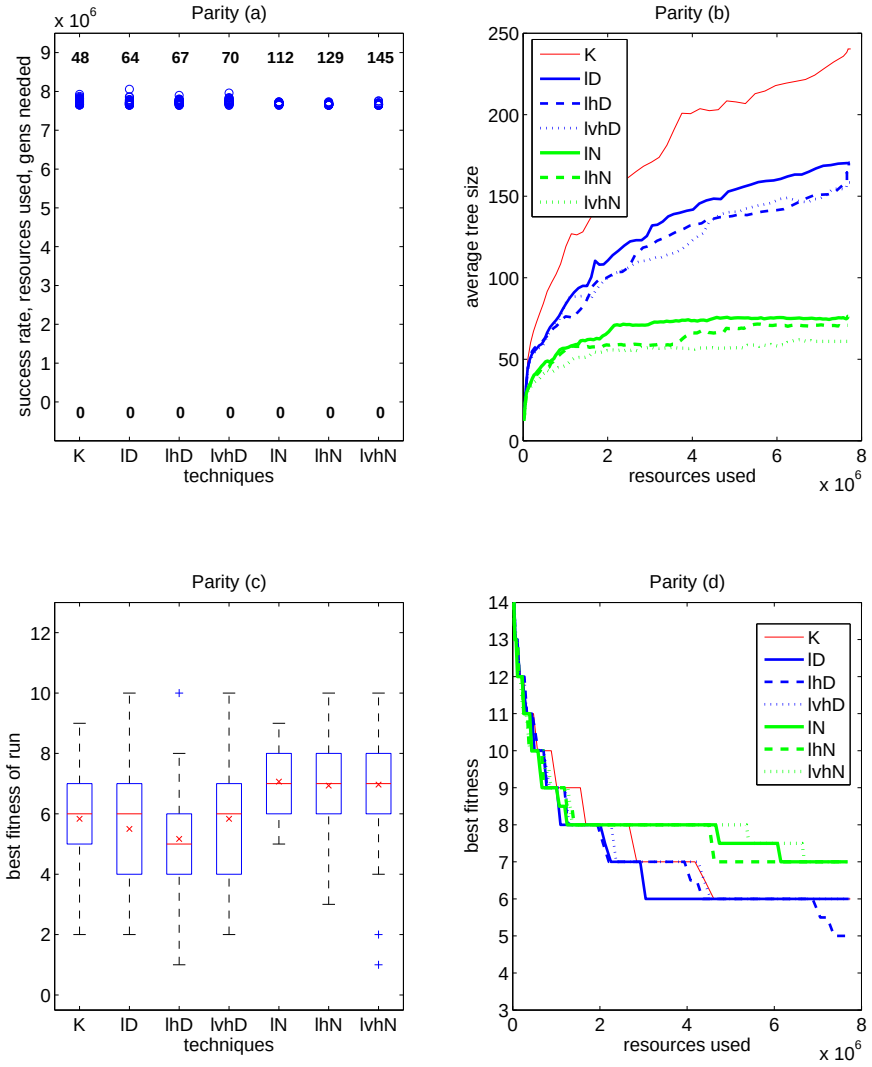

Figure 6.6: Results of the dynamic limit techniques on the Parity problem, with upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

ones presenting stable and relatively low average tree size early in the run.

In the third plot (c), the boxplot, only lDynNodes and lvhDynNodes were worse than the baseline, Koza. There were no significant differences among the dynamic depth or the dynamic size techniques, but all the dynamic size techniques were worse than at least two dynamic depth techniques. Based on these results, a rough ranking of the several techniques could be: 1) Koza and all dynamic depth techniques 2) all dynamic size techniques.

The last plot (d) supports the previous ranking, showing that Koza and the dynamic depth techniques achieved better fitness with less resources than the dynamic size techniques.

Remarks The above results reveal that the dynamic depth techniques, in particular DynDepth and hDynDepth, are the best performing techniques among Dynamic Limits, in the Parity problem. The introduction of the static upper limit evened the performance of the dynamic depth techniques, although it did not introduce any significant differences in the best fitness of run.

6.4.4 11-Bit Boolean Multiplexer

Without upper limit Figure 6.7 shows the results of the comparison among the dynamic limit techniques on the Multiplexer problem, without using an upper limit. The first plot (a) presents the resources used and the number of generations needed to complete the run, as well as the success rate, for each technique. It reveals that there is often success in finding an optimal solution for the Multiplexer problem (success rates of 23–40%), except with the dynamic size techniques (success rates of 3–10%). Koza was the technique that exhausted the resources in the fewest generations (51), while vhDynNodes took the highest number of generations (118).

The second plot (b) presents the growth of the average tree size along the run, for each technique. Koza presents the largest growth, followed by the non-heavy variants of both dynamic depth and dynamic size techniques. Next come the remaining dynamic depth techniques, and finally the remaining dynamic size techniques, that were able to stabilize the growth of the average tree size early in the run.

The third plot (c) is a boxplot of the best fitness of run obtained with each technique. There were no significant differences among the dynamic depth techniques, all as good as Koza. There were also no significant differences among the dynamic size techniques, but these performed worse than Koza and the dynamic depth techniques. Based on these results, a rough ranking of the several techniques could be: 1) Koza and all the dynamic depth techniques 2) dynamic size techniques.

The last plot (d) presents the evolution of the best fitness as a function of the resources used, for each technique. hDynDepth and vhDynDepth performed better than the rest, achieving improved fitness earlier in the run.

With upper limit Figure 6.8 shows the results of the comparison among the dynamic limit techniques on the Multiplexer problem, using a static upper limit.

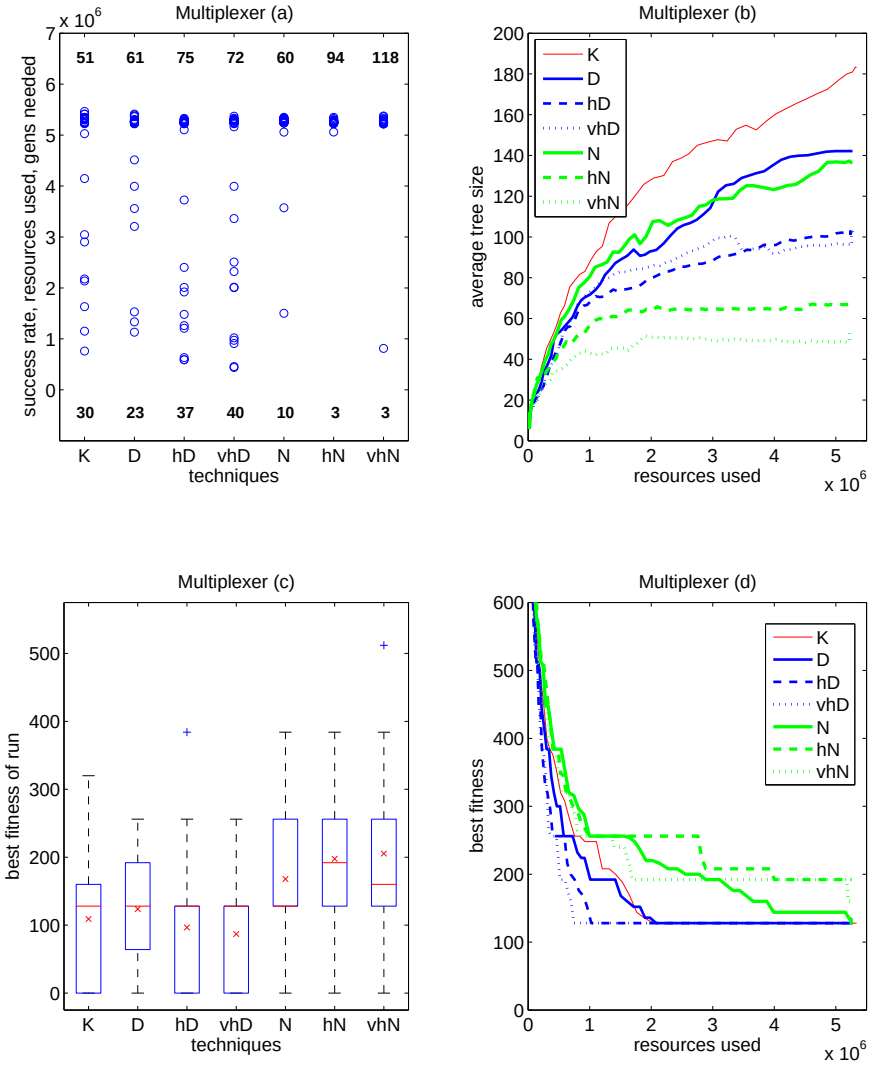

Figure 6.7: Results of the dynamic limit techniques on the Multiplexer problem, without upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

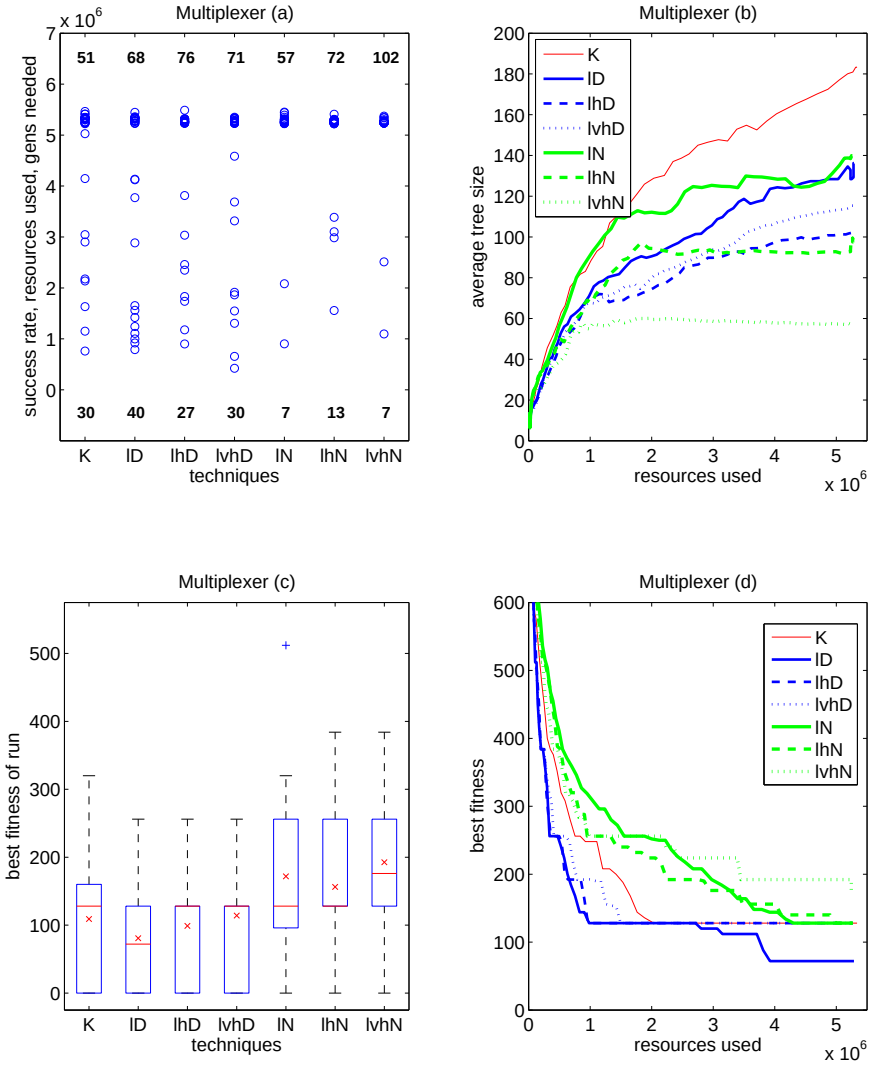

Figure 6.8: Results of the dynamic limit techniques on the Multiplexer problem, with upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

As in the previous experiment, the first plot (a) shows that there is often success in finding an optimal solution for the Multiplexer problem (success rates of 27–40%), but much less with the dynamic size techniques (success rates of 7–13%). Koza was the technique that exhausted the resources in the fewest generations (51), followed closely by lDynNodes (57). As previously, lvhDynNodes required the highest number of generations (102).

The second plot (b) shows that the Koza technique presents, once again, the largest growth of average tree size, and that lvhDynNodes presents the lowest. The remaining techniques present similar behavior between each other.

In the third plot (c), the boxplot, once again there were no significant differences among the dynamic depth techniques, all as good as Koza. There were also no significant differences among the dynamic size techniques, but lvhDynNodes performed worse than Koza and all the dynamic depth variants. The non-heavy depth variant lDynDepth outperformed all the dynamic size techniques. Based on these results, a rough ranking of the several techniques could be the same as in the previous experiment: 1) Koza and all the dynamic depth techniques 2) dynamic size techniques.

The last plot (d) shows that the dynamic depth techniques, followed by Koza, achieved better fitness sooner than the dynamic size techniques. lDynDepth was able to achieve better fitness than the rest.

Remarks The above results reveal that, along with the baseline Koza, the dynamic depth techniques are the best performing techniques among Dynamic Limits, in the Multiplexer problem. It is also clear that the introduction of the static upper limit benefited the DynDepth technique, increasing its success rate and allowing it to reach better fitness values, although not statistically significant.

6.5 Conclusions

In all problems, whenever statistically significant differences allowed for a ranking of the techniques, DynDepth was the only one that always scored number one, never performing worse, and sometimes significantly better, than the successful baseline Koza. The dynamic size techniques generally performed worse than the rest, except on the Artificial Ant problem. All the techniques within the Dynamic Limits are apparently able to control bloat without relying on the static upper limit, a very desirable property.

Chapter 7

Comparison within Resource-Limited GP

This chapter studies the efficiency of Resource-Limited GP for bloat control. It lists the techniques involved and specifies the procedures and parameters used in the comparisons. Then it presents the results as described in Section 5.3, and concludes by summing up the major findings.

7.1 Techniques

Table 7.1 summarizes all the techniques compared within the Resource-Limited GP approach (see Chapter 4 for details). As in Section 6.1, Koza is once again the baseline technique. The names (and acronyms) of the other techniques are composed of several parts to help their identification: *Res* stands for Resources; *Steady (S)* and *Low (L)* relate to the Steady and Low implementations, respectively; a following *L* stands for Light; *h* and *vh* identify the respective Heavy and VeryHeavy variants; *l* stands for limited, meaning that a static upper limit is used along with the resource limit, in which case there are restrictions both at the individual and population level. The purpose of testing the limited techniques was once again to check whether the new techniques can do without the static upper limit, or still benefit from joining, instead of just replacing, the baseline technique.

7.2 Resource Limit

The initial resource limit is always equal to the amount of resources used by the initial population. This random generation of individuals is created exactly like the populations of the Koza technique, using the Ramped Half-and-Half procedure with maximum depth 6. As in Dynamic Limits, the non-heavy variants can only increase the limit, while the heavy variants can also decrease it as low as the initial resource limit established by the first population, or even lower in the case of the very heavy variants. No upper bound exists for the resource limit. Like the baseline Koza, the limited techniques use the static upper limit of depth 17 at the individual level.

Table 7.1: Techniques compared within the Resource-Limited GP approach.

Technique	**Acronym**	**Short description**
Koza	K	static depth limit
ResSteady	S	steady resource limit
ResSteadyL	SL	steady-light resource limit
hResSteady	hS	heavy steady resource limit
hResSteadyL	hSL	heavy steady-light resource limit
vhResSteady	vhS	very heavy steady resource limit
vhResSteadyL	vhSL	very heavy steady-light resource limit
lResSteady	lS	limited steady resource limit
lResSteadyL	lSL	limited steady-light resource limit
lhResSteady	lhS	limited heavy steady resource limit
lhResSteadyL	lhSL	limited heavy steady-light resource limit
lvhResSteady	lvhS	limited very heavy steady resource limit
lvhResSteadyL	lvhSL	limited very heavy steady-light resource limit
ResLow	L	low resource limit
ResLowL	LL	low-light resource limit
hResLow	hL	heavy low resource limit
hResLowL	hLL	heavy low-light resource limit
vhResLow	vhL	very heavy low resource limit
vhResLowL	vhLL	very heavy low-light resource limit
lResLow	lL	limited low resource limit
lResLowL	lLL	limited low-light resource limit
lhResLow	lhL	limited heavy low resource limit
lhResLowL	lhLL	limited heavy low-light resource limit
lvhResLow	lvhL	limited very heavy low resource limit
lvhResLowL	lvhLL	limited very heavy low-light resource limit

7.3 Results

This section presents the results of the comparisons within Resource-Limited GP, divided in the four problems considered (see Section 5.1). As in Section 6.4, the results are first presented without using a static upper limit, and then using the upper limit, except for the baseline technique, Koza. Short concluding remarks are inserted after each problem, highlighting the best performing techniques and describing how the introduction of the upper limit affected the performance.

7.3.1 Symbolic Regression

Steady implementation, without upper limit Figure 7.1 shows the results of the comparison among the resource-limited techniques (Steady implementation) on the Regression problem, without using an upper limit. The first plot (a) presents the resources used and the number of generations needed to complete the run, as well as the success rate, for each technique. It reveals that early convergence to an optimum is a very common occurrence in the Regression problem, with success rates as high as 70% in the ResSteady and hResSteady-Light techniques. Koza achieved the lowest success rate (40%), followed by the remaining techniques (43–50%). Koza was the technique that exhausted the resources in the fewest generations (50), while vhResSteady needed the highest number of generations (221). The non-light variants were much more sparing, taking many more generations to exhaust the resources.

The second plot (b) presents the growth of the average tree size along the run, for each technique. Without using an upper limit, the trees of the resource-limited techniques grew very large, particularly the light variants. This growth was accompanied by a prompt decrease in population size (not shown). Koza was the technique with the lowest growth in average tree size, except for ResSteady and hResSteadyLight, which present very short growth lines (ending before reaching 2×10^5 resources) because they converge to an optimum very early in the run (see plot (d)).

The third plot (c) is a boxplot of the best fitness of run obtained with each technique. There were no significant differences among the resource-limited techniques, all as good as Koza, therefore it is not possible to rank the techniques.

The last plot (d) presents the evolution of the best fitness as a function of the resources used, for each technique. As already stated, ResSteady and hResSteadyLight converged to an optimum early in the run, while using only a small amount of resources. Not all runs succeeded, but 70% did (see plot (a)), so the median presents itself as a line that rapidly drops to the best fitness. Although the differences in best fitness of run were not significant (see plot (c)), ResSteady and hResSteadyLight appear to be the best techniques, followed by Koza and the remaining light techniques, and finally the remaining non-light techniques.

Steady implementation, with upper limit Figure 7.2 shows the results of the comparison among the resource-limited techniques (Steady implementation)

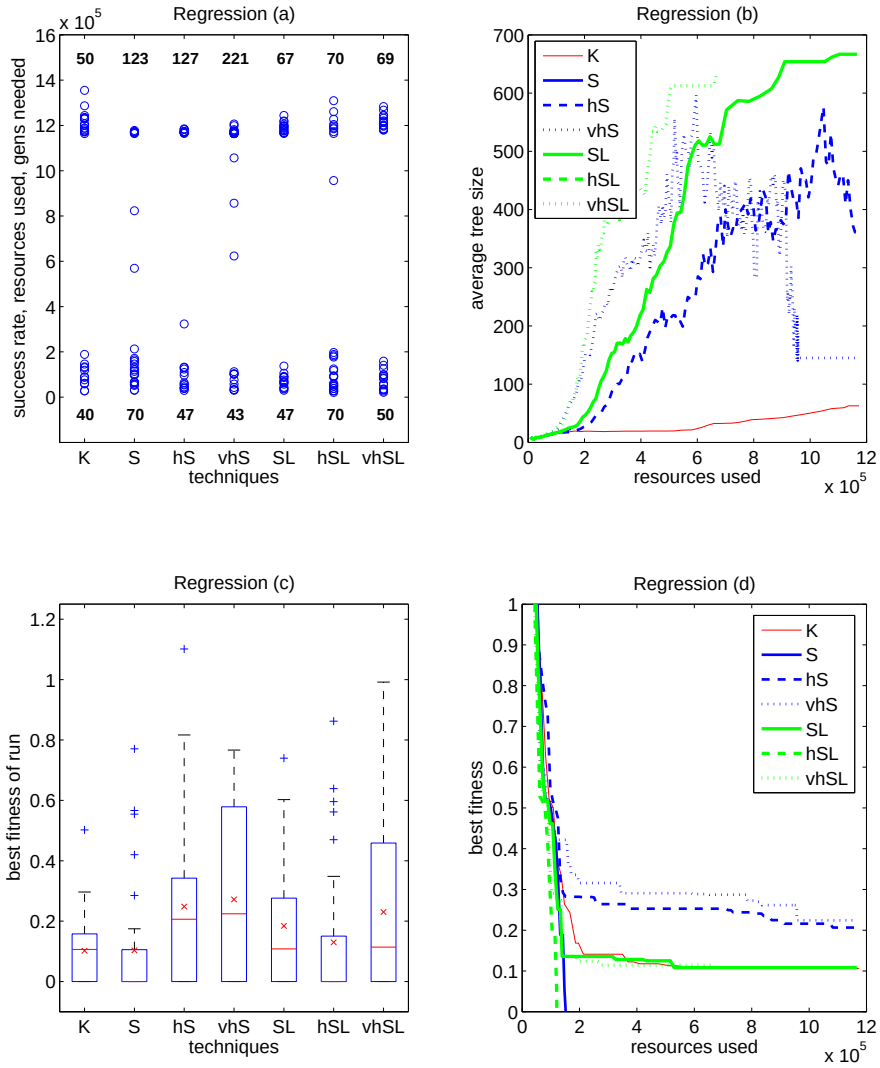

Figure 7.1: Results of the resource-limited techniques (Steady implementation) on the Regression problem, without upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

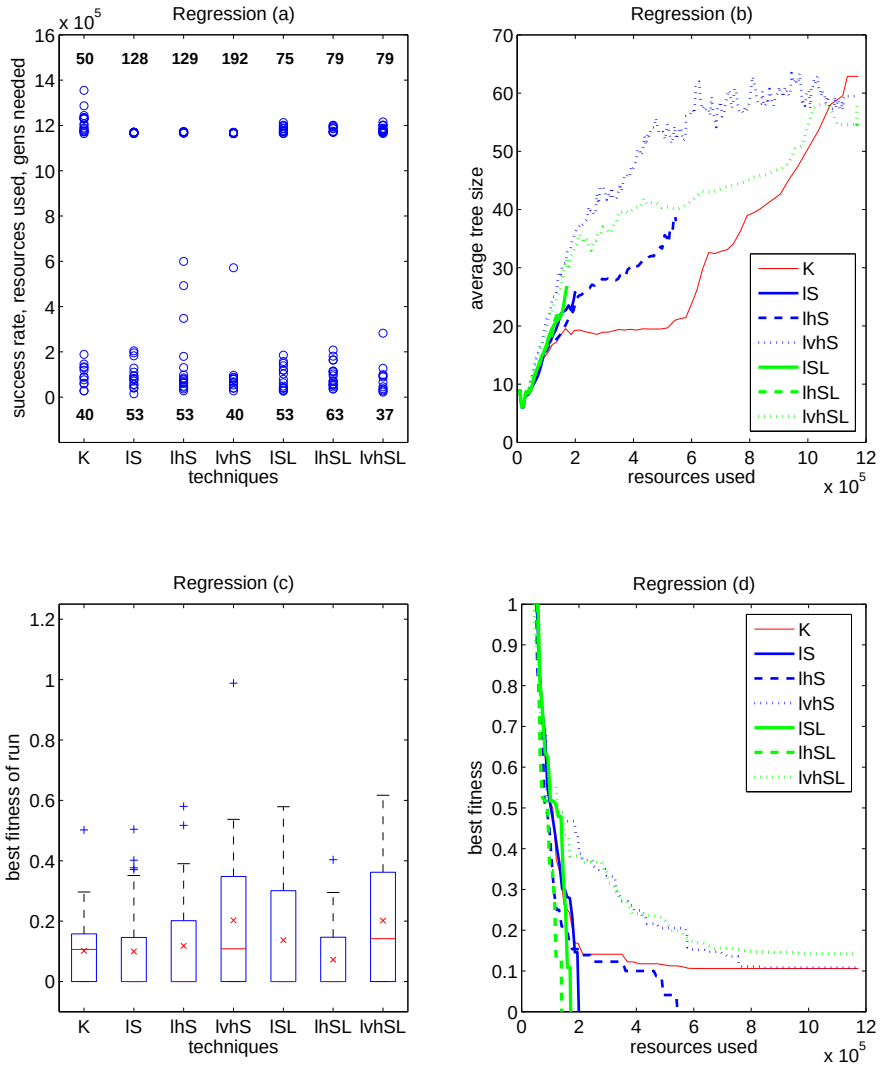

Figure 7.2: Results of the resource-limited techniques (Steady implementation) on the Regression problem, with upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

on the Regression problem, using a static upper limit. As in the previous experiment, the first plot (a) shows that early convergence to an optimal solution is very common in the Regression problem (success rates of 37–63%), with Koza presenting one of the lowest rates (40%). The number of generations required to exhaust the resources follows a similar pattern to the previous experiment, with Koza requiring the lowest number of generations (50), followed by the light techniques and finally the non-light techniques, with lvhResSteady requiring the highest number of generation (192).

The second plot (b) shows very different results from the previous experiment. The usage of the upper limit produced a much lower growth of average tree size on the resource-limited techniques. Many of the techniques quickly converged to an optimum and produced short growth lines. The remaining resource-limited techniques present a quick growth that by the end of the run reaches no higher than Koza.

In the third plot (c), the boxplot, once again there were no significant differences among any of the techniques, therefore no ranking is possible.

The last plot (d) shows that most of the resource-limited techniques succeeded in finding an optimal solution, some earlier than others, and the remaining had a slower convergence but by the end of the run achieved the same fitness as Koza.

Low implementation, without upper limit Figure 7.3 shows the results of the comparison among the resource-limited techniques (Low implementation) on the Regression problem, without using an upper limit. The first plot (a) presents the resources used and the number of generations needed to complete the run, as well as the success rate, for each technique. As in the Steady implementation, early convergence to an optimal solution is a common occurrence in the Regression problem, with success rates as high as 63% in the ResLowLight technique and the minimum of 27% in the vhResLow technique, the only one with a lower success rate than Koza (40%). Koza was the technique that exhausted the resources in the fewest generations (50). Both very heavy variants, vhResLow and vhResLowLight, collapsed the population into only a few individuals (possibly just one, small), so they required hundreds of thousands of generations to exhaust the resources.

The second plot (b) presents the growth of the average tree size along the run, for each technique. Without using an upper limit, the trees of the resource-limited techniques (Low implementation) grew very large. ResLowLight and vhResLowLight present short growth lines because they quickly converged to an optimum, followed by hResLowLight (see plot (d)). vhResLow presents a sudden drop of average tree size when the population collapses (see plot (a)). Koza was the technique with the lowest growth in average tree size.

The third plot (c) is a boxplot of the best fitness of run obtained with each technique. There were no significant differences except for vhResLow being worse than Koza and ResLowLight. Based on these results, a rough ranking of the techniques could be: 1) all techniques except 2) vhResLow.

The last plot (d) presents the evolution of the best fitness as a function of the resources used, for each technique. As already stated, ResLowLight and

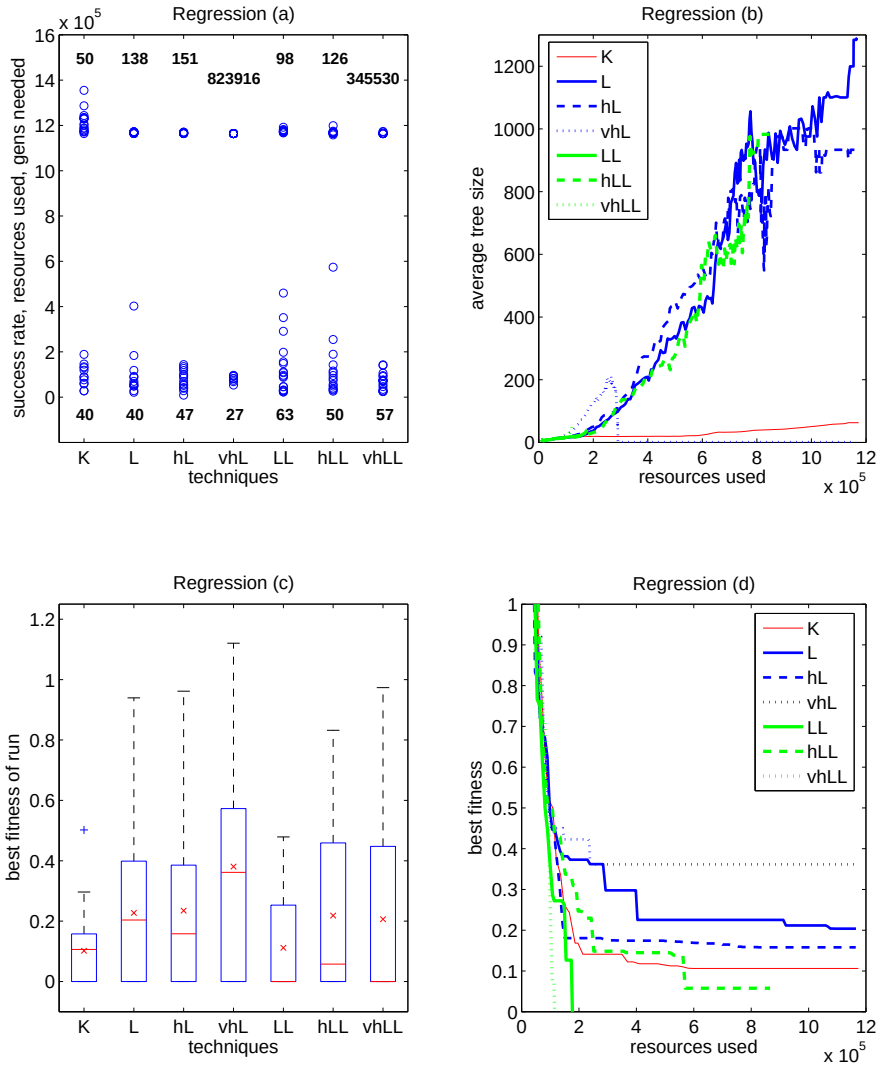

Figure 7.3: Results of the resource-limited techniques (Low implementation) on the Regression problem, without upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

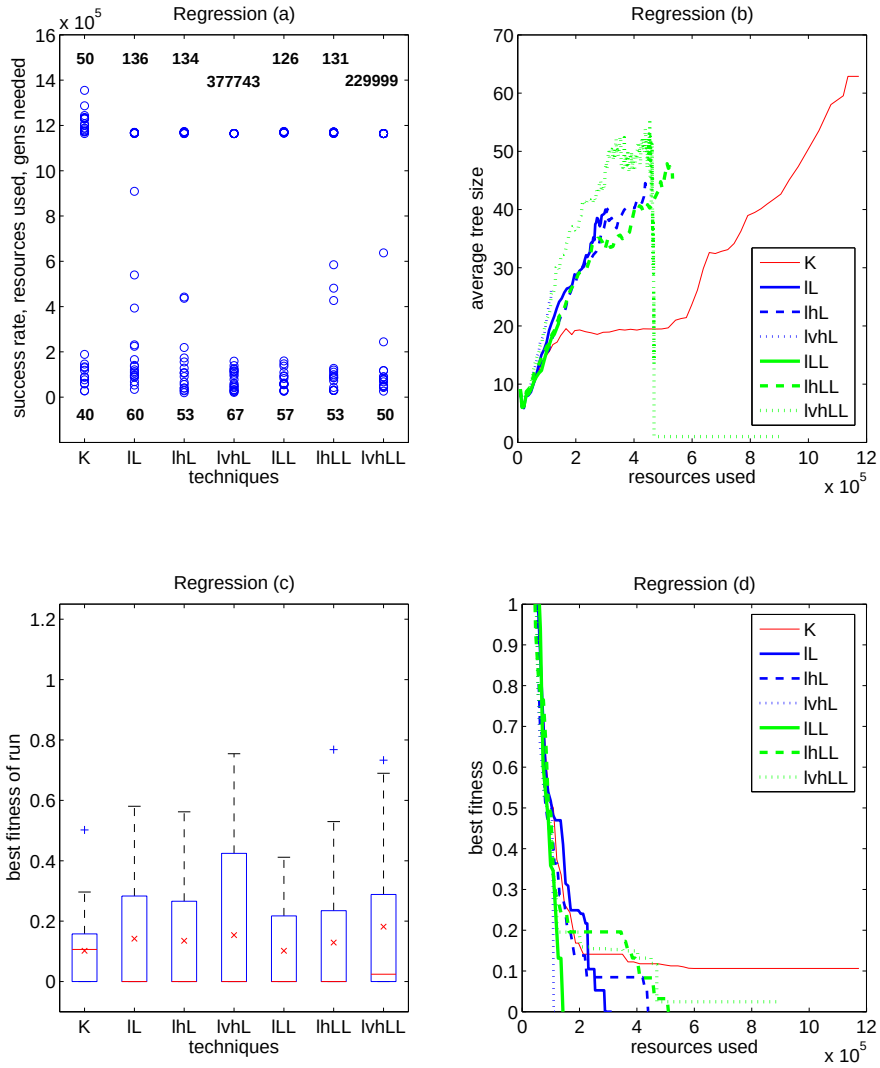

Figure 7.4: Results of the resource-limited techniques (Low implementation) on the Regression problem, with upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

vhResLowLight succeeded in finding an optimum early in the run, followed by hResLowLight.

Low implementation, with upper limit Figure 7.4 shows the results of the comparison among the resource-limited techniques (Low implementation) on the Regression problem, using a static upper limit. The first plot (a) once again shows that early convergence to an optimum is very common in the Regression problem (success rates of 40–67%), with Koza presenting the lowest rate. The number of generations required to exhaust the resources is the lowest for Koza (50), with both very heavy techniques once again collapsing the population into only a few small individuals, thus requiring hundreds of thousands of generations to exhaust the resources.

The second plot (b) shows that, as in the Steady implementation, the usage of the upper limit produced a much lower growth of average tree size. All the techniques except Koza converged to an optimum early in the run, presenting short growth lines. lvhResLowLight presents a sudden drop of average tree size when the population collapses (see plot (a)).

In the third plot (c), the boxplot, there were no significant differences among any of the techniques, therefore no ranking is possible.

The last plot (d) shows again that all the resource-limited techniques converged to an optimum early in the run, first lResLowLight and lvhResLowLight, followed by the remaining non-light techniques, and finally the remaining light techniques.

Remarks The above results show that, in the Steady implementation, all the techniques perform equally well, but (l)ResSteady and (l)hResSteadyLight tend to succeed in finding an optimum earlier and more often, with or without using the static upper limit. Among the Low implementation techniques, there are very few significant differences, with one of the very heavy techniques (vhResLow) performing less reliably, and (l)ResLowLight and (l)vhResLowLight consistently succeeding earlier than the rest. The collapsing of the population into only a few individuals does not necessarily mean a bad performance. There are no significant differences between the best techniques of the Steady and Low implementations in the Regression problem. The introduction of the static upper limit greatly reduced the growth of average tree size on the resource-limited techniques, but did not influence their performance, except for significantly improving vhResLow.

7.3.2 Artificial Ant

Steady implementation, without upper limit Figure 7.5 shows the results of the comparison among the resource-limited techniques (Steady implementation) on the Artificial Ant problem, without using an upper limit. The first plot (a) presents the resources used and the number of generations needed to complete the run, as well as the success rate, for each technique. It shows that convergence to an optimal solution seldom occurred in the Artificial Ant problem, with success rates of 0–10%. Koza was the technique that exhausted the

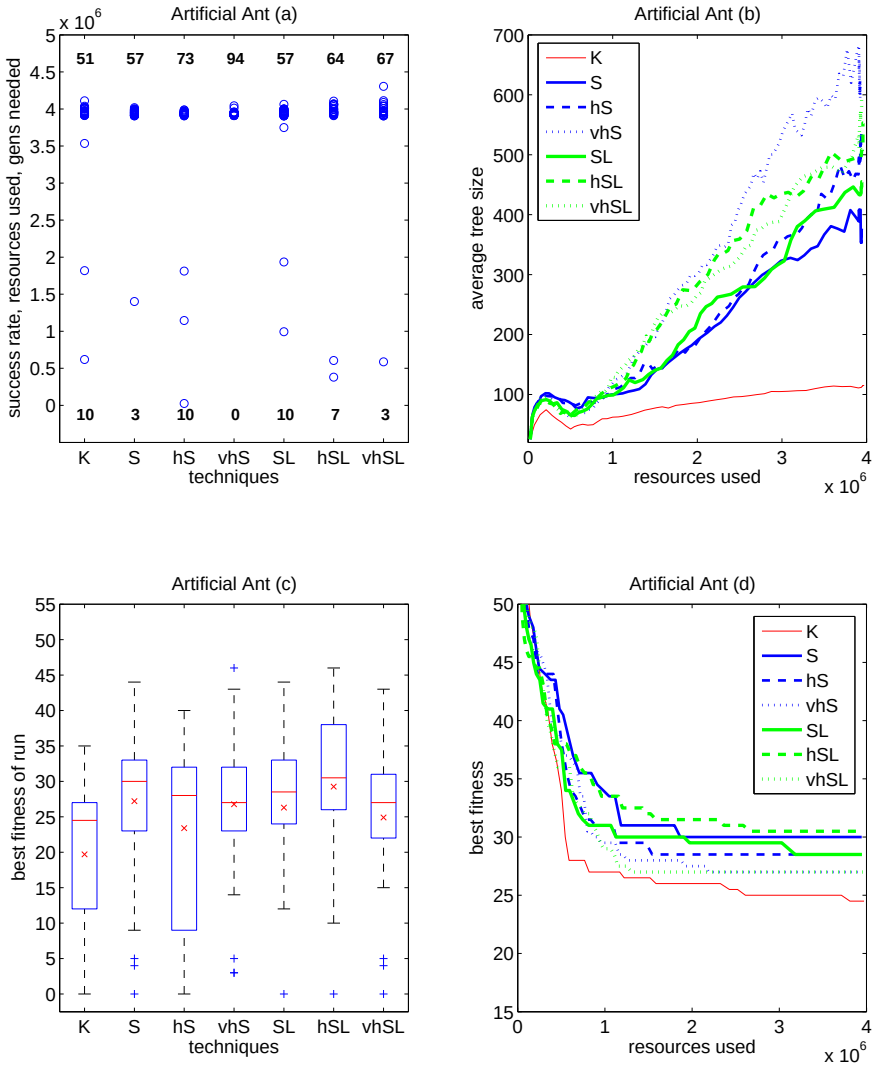

Figure 7.5: Results of the resource-limited techniques (Steady implementation) on the Artificial Ant problem, without upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

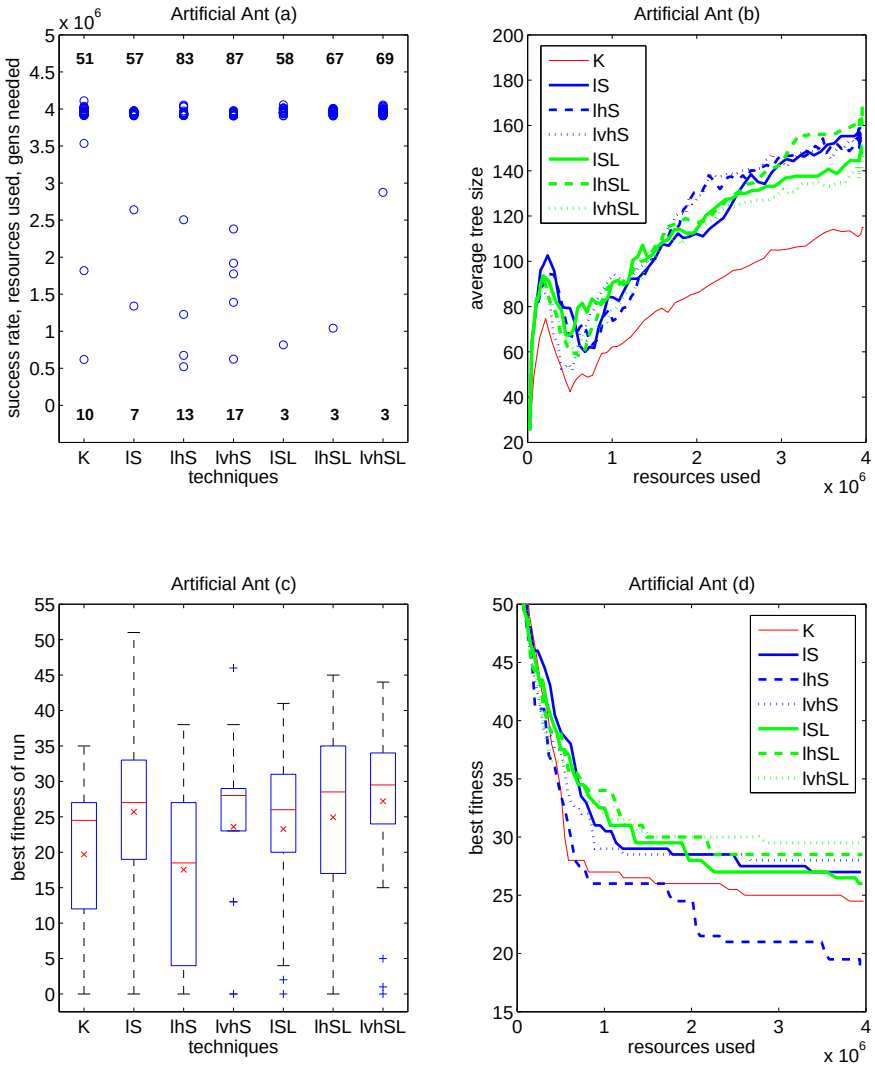

Figure 7.6: Results of the resource-limited techniques (Steady implementation) on the Artificial Ant problem, with upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

resources in the fewest generations (51), with vhResSteady taking the highest number of generations (94).

The second plot (b) presents the growth of the average tree size along the run, for each technique. As in the previous problem, without using an upper limit, the trees of the resource-limited techniques grew very large, whit Koza presenting the smallest growth.

The third plot (c) is a boxplot of the best fitness of run obtained with each technique. There were no significant differences among the techniques, except for hSteadyLight being worse than Koza. Based on these results, a rough ranking of the techniques could be: 1) all techniques except 2) hSteadyLight.

The last plot (d) presents the evolution of the best fitness as a function of the resources used, for each technique. Koza was the technique achieving better fitness earlier in the run.

Steady implementation, with upper limit Figure 7.6 shows the results of the comparison among the resource-limited techniques (Steady implementation) on the Artificial Ant problem, using a static upper limit. The success rates were somewhat higher than without using the upper limit, reaching values between 3% and 17%. Koza was once again the technique that exhausted the resources in the fewest generations (51), with the highest number of generations being taken by lvhResSteady (87).

The second plot (b) shows that, although the upper limit caused the resource-limited techniques to have a much more limited tree growth, Koza was still the technique presenting the smallest growth of average tree size.

In the third plot (c), the boxplot, there were no significant differences among the dynamic depth techniques, except for lvhResSteadyLight being worse than Koza and lhResSteady. Based on these results, a rough ranking of the several techniques could be: 1) all techniques except 2) lvhResSteadyLight.

The last plot (d) shows that lhResSteady was able to reach better fitness than the other techniques, although the differences are not significant. lhResSteady and Koza were the techniques achieving better fitness earlier in the run.

Low implementation, without upper limit Figure 7.7 shows the results of the comparison among the resource-limited techniques (Low implementation) on the Artificial Ant problem, without using an upper limit. The first plot (a) presents the resources used and the number of generations needed to complete the run, as well as the success rate, for each technique. As in the Steady implementation, convergence to an optimal solution seldom occurred in the Artificial Ant problem, with success rates of 0–10%, Koza having the highest rate. Koza was also the technique that exhausted the resources in the fewest generations (51), while both very heavy techniques collapsed the population into just a few small individuals, thus requiring millions of generations to exhaust the resources.

The second plot (b) presents the growth of the average tree size along the run, for each technique. As in the previous problem, without using an upper limit, the trees of the resource-limited techniques grew very large, while Koza

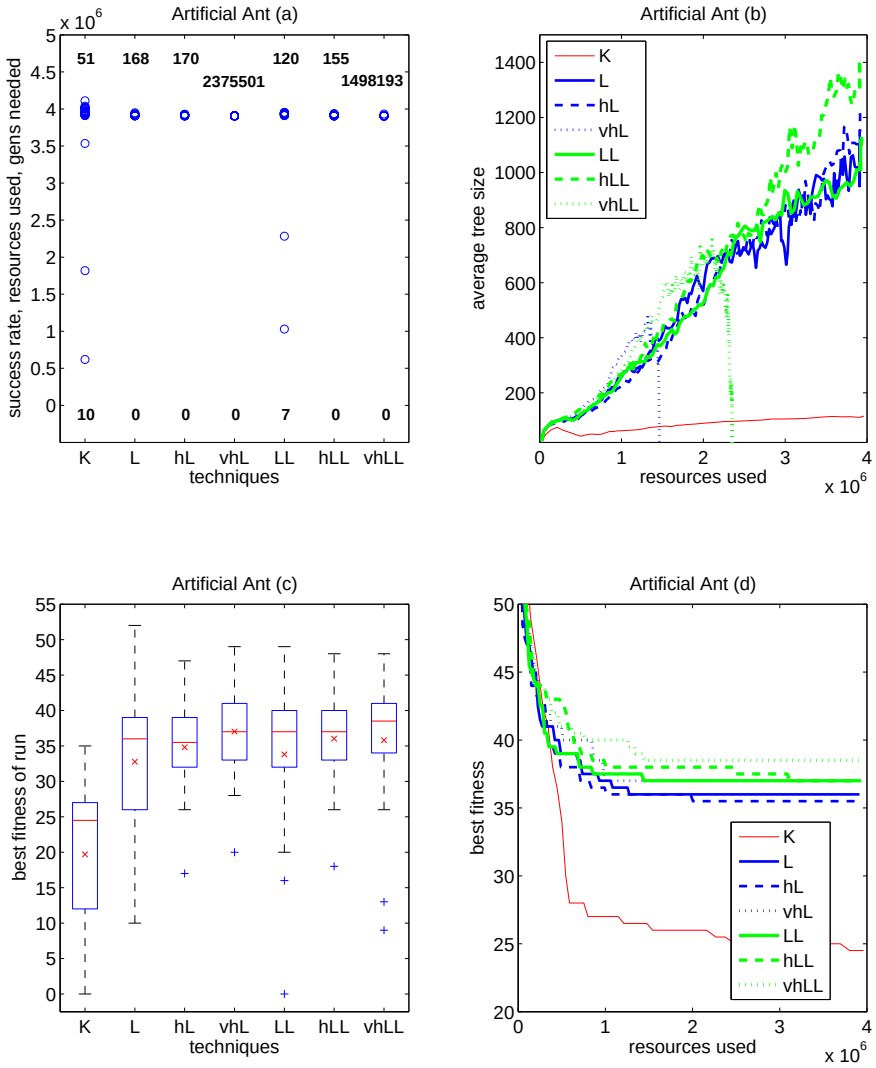

Figure 7.7: Results of the resource-limited techniques (Low implementation) on the Artificial Ant problem, without upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

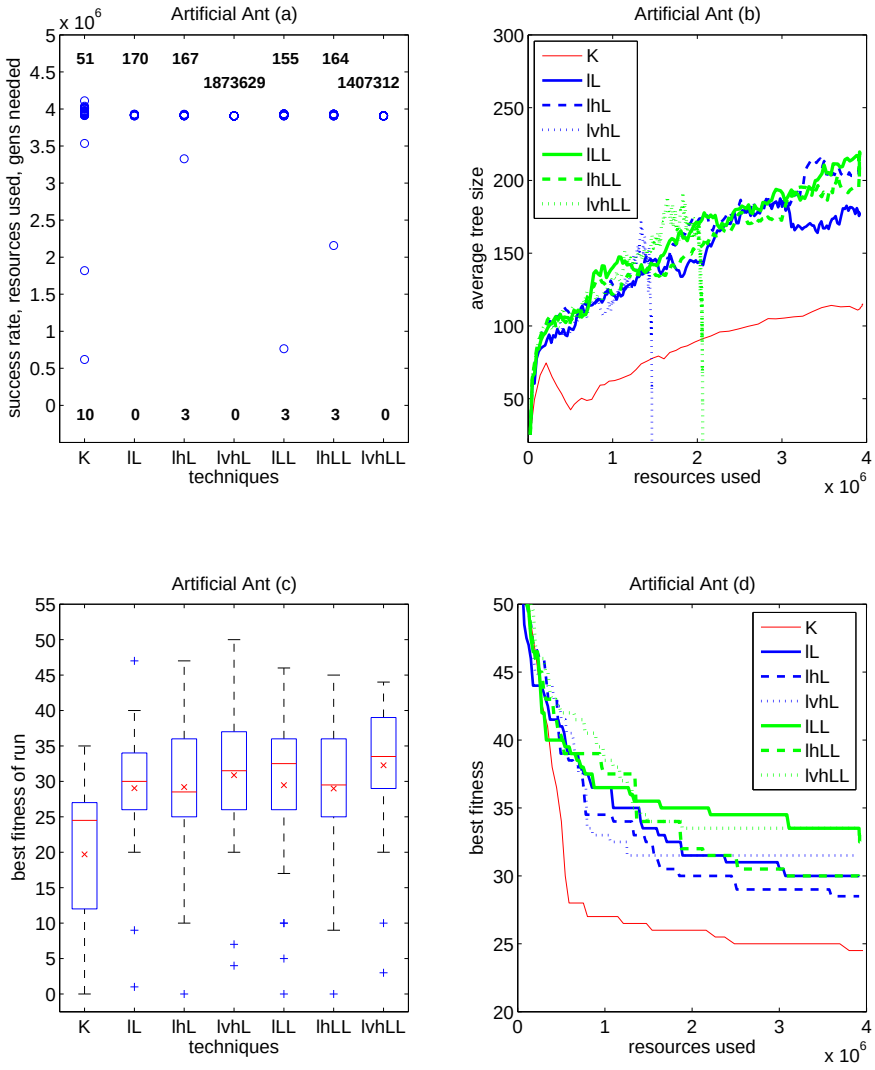

Figure 7.8: Results of the resource-limited techniques (Low implementation) on the Artificial Ant problem, with upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

maintained a small growth. Both very heavy variants present a dropping growth line where the population collapsed.

The third plot (c) is a boxplot of the best fitness of run obtained with each technique. All the resource-limited techniques performed significantly worse than the baseline, Koza. Based on these results, a ranking of the techniques could be: 1) Koza 2) all resource-limited techniques.

The last plot (d) presents the evolution of the best fitness as a function of the resources used, for each technique. Koza was able to achieve a much better fitness than all the rest.

Low implementation, with upper limit Figure 7.8 shows the results of the comparison among the resource-limited techniques (Low implementation) on the Artificial Ant problem, using a static upper limit. The success rates were equally low when compared to the previous experiment, with Koza again reaching the highest rate of 10%. Koza was once again the technique that exhausted the resources in the fewest generations (51), with both very heavy techniques once again collapsing the population and requiring more than a million generations.

The second plot (b) shows that, although the resource-limited techniques had a much more limited tree growth than in the previous experiment, Koza was still the technique presenting the smallest growth of average tree size. Once again, both very heavy techniques present a sudden drop of tree growth where the population collapsed into only a few small individuals.

In the third plot (c), the boxplot, once again Koza performed better than all the rest. Based on these results, the ranking of the techniques is the same as without the upper limit: 1) Koza 2) all resource-limited techniques.

The last plot (d) shows that Koza quickly achieved better fitness than all the other techniques.

Remarks The above results reveal that, in the Steady implementation, the resource-limited techniques are as good as Koza in the Artificial Ant problem, except some of the light variants, which appear less reliable. The introduction of the static upper limit increased the success rates and greatly reduced the growth of average tree size on the resource-limited techniques of the Steady implementation, but did not significantly influence their performance. In the Low implementation, all the techniques perform worse than Koza, even when using the static upper limit.

7.3.3 5-Bit Even Parity

Steady implementation, without upper limit Figure 7.9 shows the results of the comparison among the resource-limited techniques (Steady implementation) on the Parity problem, without using an upper limit. The first plot (a) presents the resources used and the number of generations needed to complete the run, as well as the success rate, for each technique. It reveals that an optimal solution was never found for the hard Parity problem. Koza was the technique that exhausted the resources in the fewest generations (48),

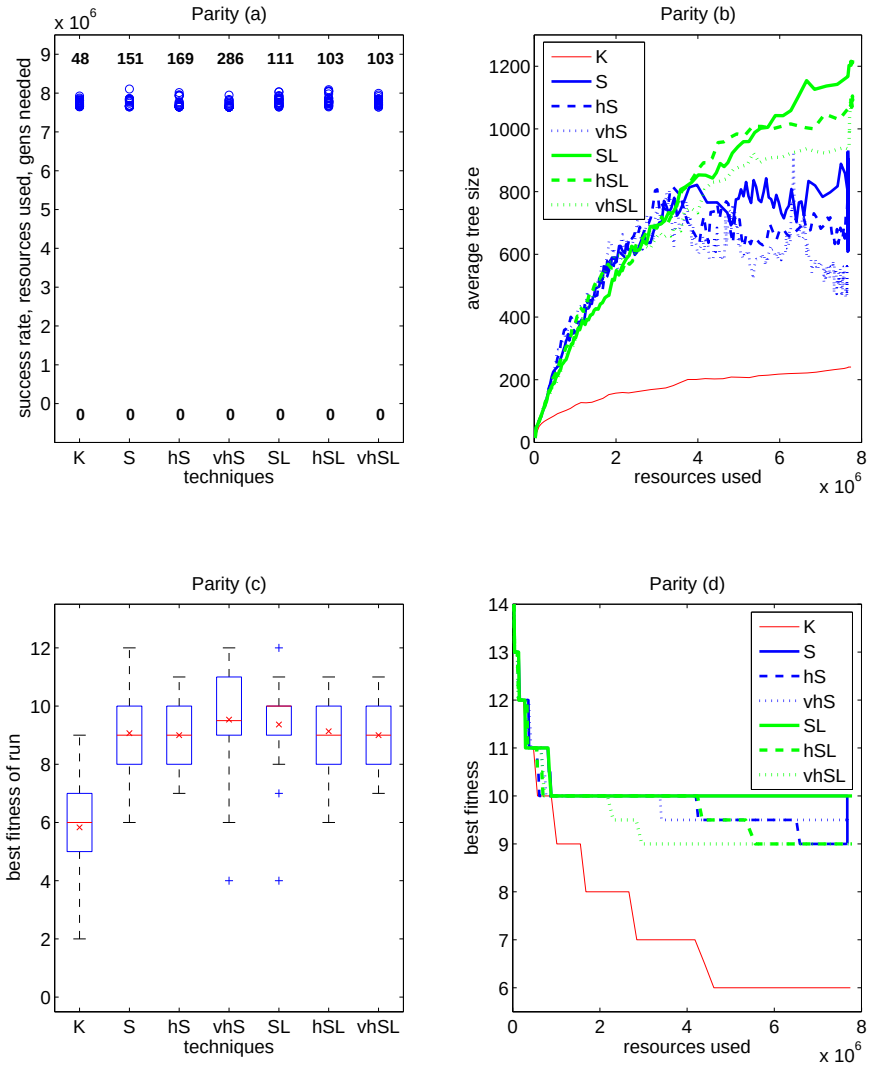

Figure 7.9: Results of the resource-limited techniques (Steady implementation) on the Parity problem, without upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

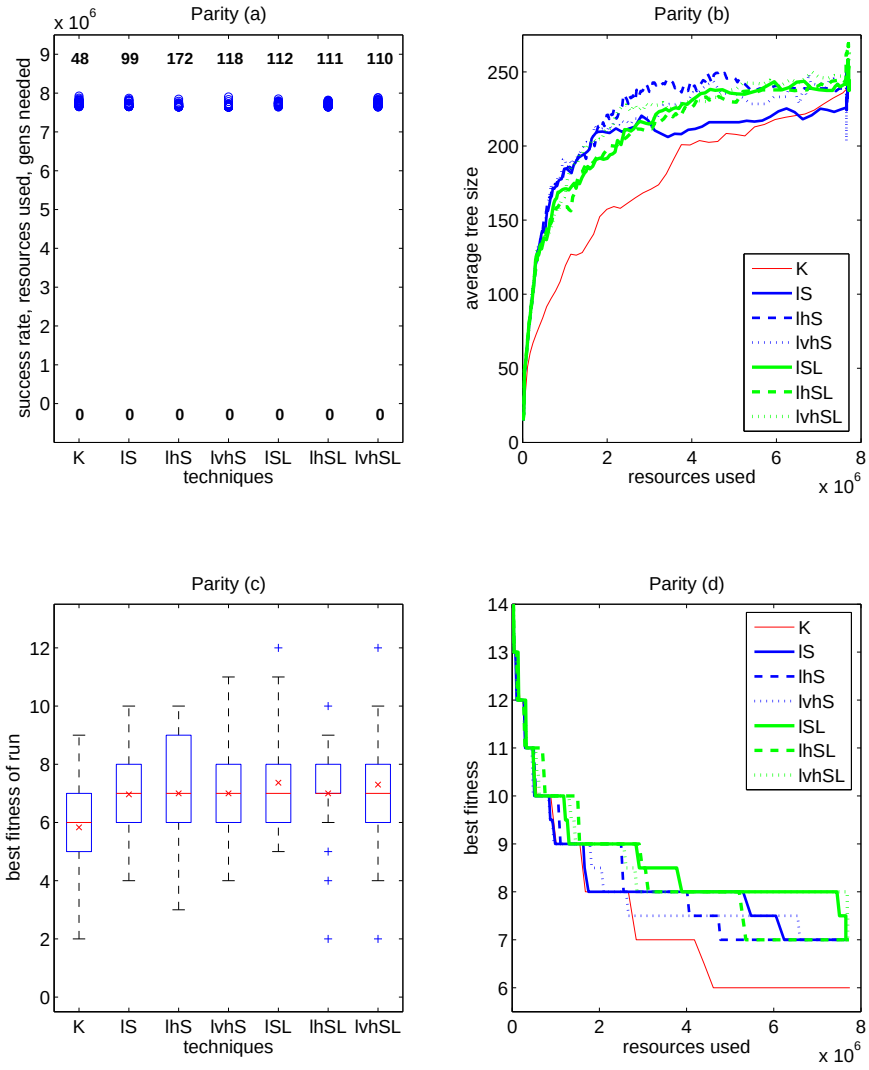

Figure 7.10: Results of the resource-limited techniques (Steady implementation) on the Parity problem, with upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

followed by the light resource-limited techniques (103-111), and finally the non-light counterparts (151-286).

The second plot (b) presents the growth of the average tree size along the run, for each technique. As in the previous two problems, all the resource-limited techniques present a large unbounded growth of average tree size (with a tendency for stabilization in the non-light variants), while Koza presents the smallest growth.

The third plot (c) is a boxplot of the best fitness of run obtained with each technique. All the resource-limited techniques performed significantly worse than Koza. Based on these results, a ranking of the several techniques could be: 1) Koza 2) all resource-limited techniques.

The last plot (d) presents the evolution of the best fitness as a function of the resources used, for each technique. Koza was able to achieve a much better fitness than any of the other techniques, from early in the run.

Steady implementation, with upper limit Figure 7.10 shows the results of the comparison among the resource-limited techniques (Steady implementation) on the Parity problem, using a static upper limit. As in the previous experiment, the first plot (a) shows that convergence to an optimum never happened in this problem. Koza was once again the technique that exhausted the resources in the fewest generations (48), while lhResSteady required the highest number (172).

The second plot (b) shows a similar average tree size growth for all the techniques, with Koza presenting a slower growth in the beginning but reaching the same values as the remaining techniques by the end of the run.

In the third plot (c), the boxplot, only the light techniques performed significantly worse than Koza. Based on these results, a ranking of the different techniques could be: 1) Koza and all non-light resource-limited techniques 2) light techniques.

The last plot (d) shows that Koza still achieved better fitness than the remaining techniques, although the difference was not always significant.

Low implementation, without upper limit Figure 7.11 shows the results of the comparison among the resource-limited techniques (Low implementation) on the Parity problem, without using an upper limit. The first plot (a) presents the resources used and the number of generations needed to complete the run, as well as the success rate, for each technique. It reveals that, as in the Steady implementation, convergence to an optimum never happened in the Parity problem. Koza was the technique that exhausted the resources in the fewest generations (48), with both very heavy techniques collapsing the population and taking more than six million generations to exhaust the resources.

The second plot (b) presents the growth of the average tree size along the run, for each technique. As in the previous problems, all the resource-limited techniques present a large unbounded growth of average tree size, while Koza presents the smallest growth. Both very heavy techniques present a sudden drop in growth where the population collapsed.

The third plot (c) is a boxplot of the best fitness of run obtained with each technique. All the resource-limited techniques performed significantly worse

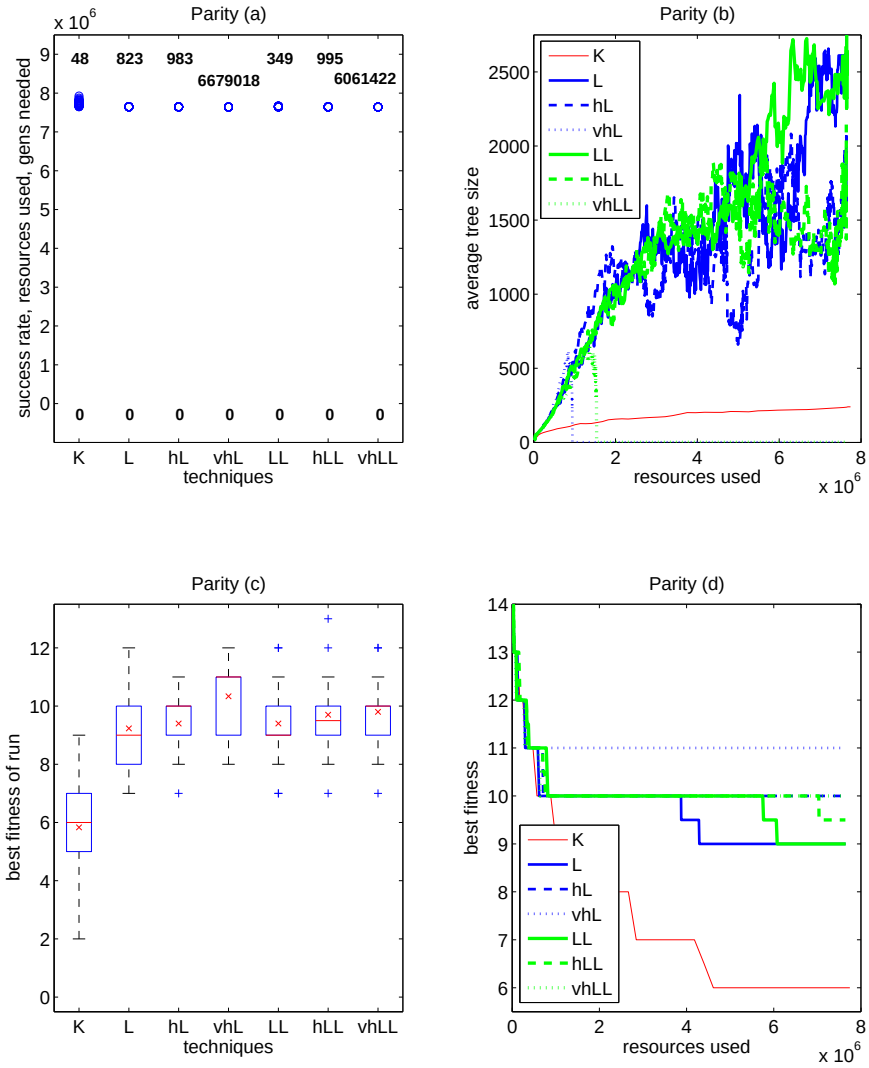

Figure 7.11: Results of the resource-limited techniques (Low implementation) on the Parity problem, without upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

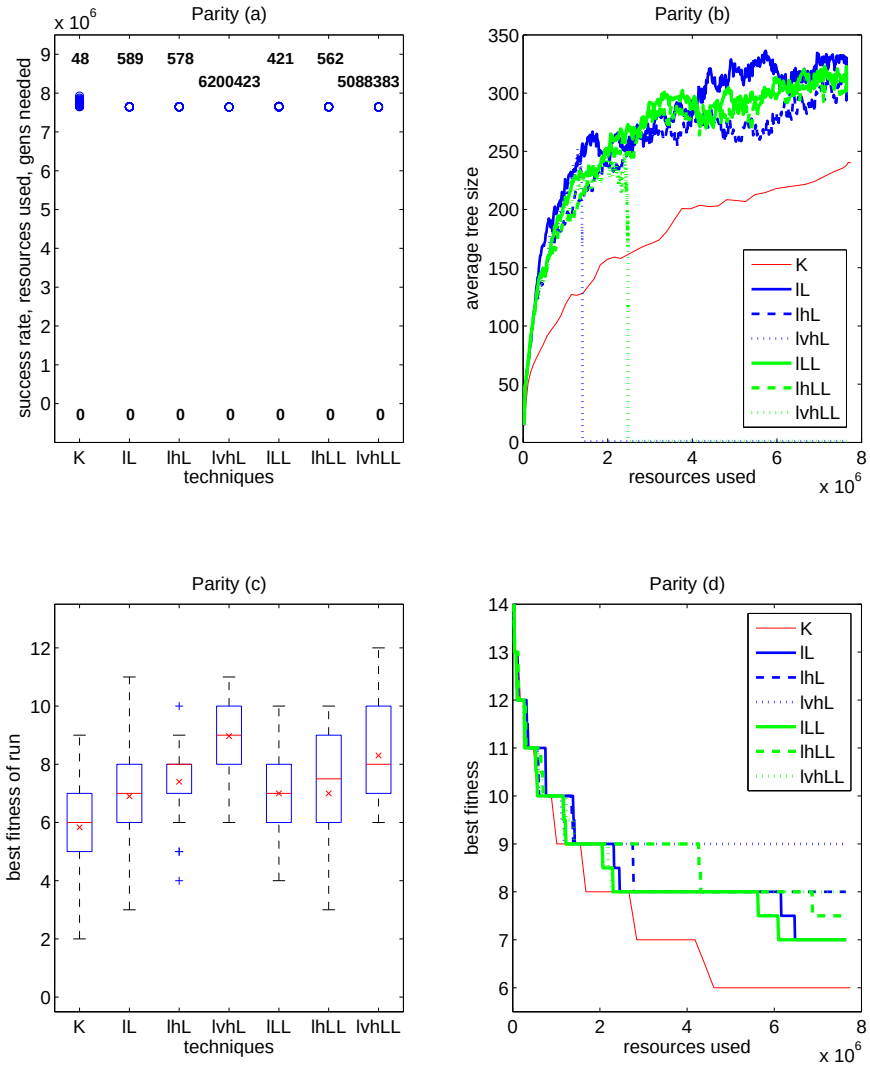

Figure 7.12: Results of the resource-limited techniques (Low implementation) on the Parity problem, with upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

than Koza. vhResLow performed worse than three other techniques. Based on these results, a ranking of the several techniques could be: 1) Koza 2) all the resource-limited techniques except 3) vhResLow.

The last plot (d) presents the evolution of the best fitness as a function of the resources used, for each technique. Koza was able to achieve a much better fitness than the other techniques, from early in the run. vhResLow only achieved a slight improvement in the beginning of the run, and then stagnated.

Low implementation, with upper limit Figure 7.12 shows the results of the comparison among the resource-limited techniques (Low implementation) on the Parity problem, using a static upper limit. As in the previous experiment, the first plot (a) shows that convergence to an optimum never happened in this problem. Koza was once again the technique that exhausted the resources in the fewest generations (48), and once again both very heavy techniques collapsed the population and required millions of generations.

The second plot (b) shows that Koza still presents the smallest growth of average tree size, and both very heavy techniques present a sudden drop where the population collapsed.

In the third plot (c), the boxplot, Koza performed better than lhResLow, lvhResLow, and lvhResLowLight. vhResLow performed significantly worse than all other techniques. Based on these results, a rough ranking of the techniques could be: 1) Koza, lResLow, lResLowLight, lhResLowLight 2) lhResLow, lvhResLowLight 3) lvhResLow.

The last plot (d) shows that Koza achieved better fitness sooner than the other techniques. The difference in performance was not as marked as when using no upper limit.

Remarks The above results reveal that the static upper limit is an essential element for the resource-limited techniques in the Parity problem, among the Steady implementation techniques. Without it, all the techniques failed when compared to Koza. Even when using it, only the non-light variants were able to achieve similar performance to Koza. Among the Low implementation techniques, a few were able to achieve similar results to Koza (lResLow, lResLowLight, lhResLowLight), but only when using the static upper limit.

7.3.4 11-Bit Boolean Multiplexer

Steady implementation, without upper limit Figure 7.13 shows the results of the comparison among the resource-limited techniques (Steady implementation) on the Multiplexer problem, without using an upper limit. The first plot (a) presents the resources used and the number of generations needed to complete the run, as well as the success rate, for each technique. It reveals that convergence to an optimum almost never happened except with the baseline technique (success rate of 30%). Koza was the technique that exhausted the resources in the fewest generations (51), while vhResSteady required the highest number of generations (106).

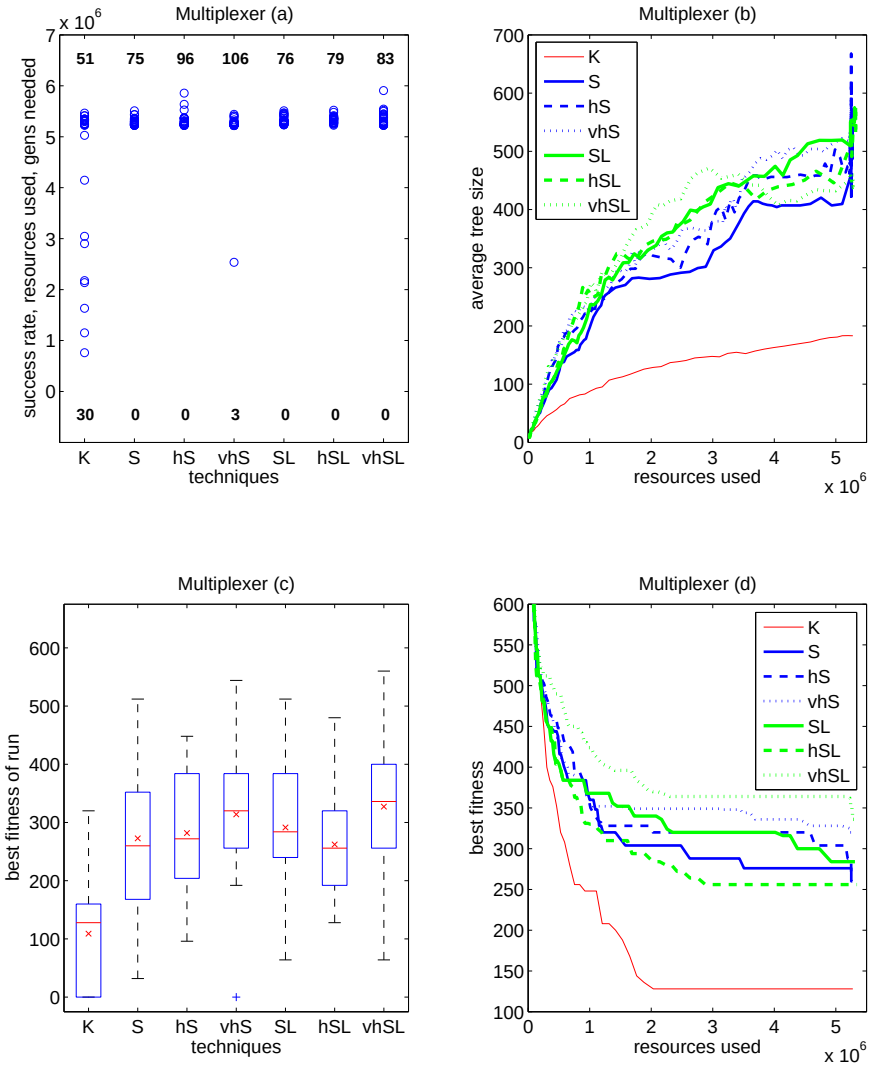

Figure 7.13: Results of the resource-limited techniques (Steady implementation) on the Multiplexer problem, without upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

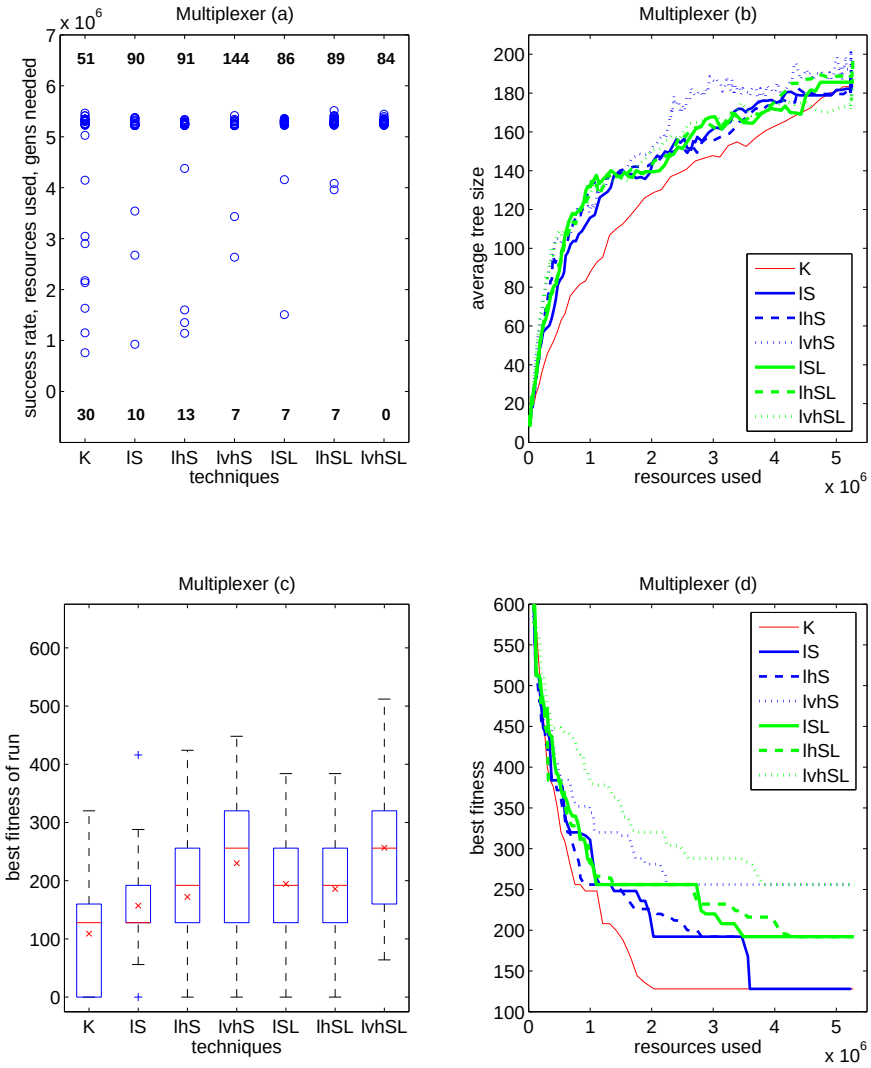

Figure 7.14: Results of the resource-limited techniques (Steady implementation) on the Multiplexer problem, with upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

The second plot (b) presents the growth of the average tree size along the run, for each technique. As in the previous three problems, all the resource-limited techniques present a large unbounded growth of average tree size, while Koza presents the smallest growth.

The third plot (c) is a boxplot of the best fitness of run obtained with each technique. As in the previous problem, all the resource-limited techniques performed significantly worse than Koza. Based on these results, a ranking of the several techniques could be: 1) Koza 2) all resource-limited techniques.

The last plot (d) presents the evolution of the best fitness as a function of the resources used, for each technique. As in the previous problem, Koza was able to achieve a much better fitness than the other techniques, from early in the run.

Steady implementation, with upper limit Figure 7.14 shows the results of the comparison among the resource-limited techniques (Steady implementation) on the Multiplexer problem, using a static upper limit. The first plot (a) shows that the success rates were somewhat higher than without using the upper limit, reaching values between 0% and 13%, except for the baseline (30%). Koza was once again the technique that exhausted the resources in the fewest generations (51), with the highest number of generations being again taken by lvhResSteady (144).

The second plot (b) shows a similar average tree size growth for all the techniques, with Koza presenting a slower growth in the beginning but reaching the same values as the remaining techniques by the end of the run.

In the third plot (c), the boxplot, only two of the non-light techniques, lResSteady and lhResSteady, performed as well as Koza, the rest performing significantly worse. lvhResSteady was worse than lResSteady, and lvhSteadyLight was worse than lResSteady and lhResSteady. Based on these results, a rough ranking of the different techniques could be: 1) Koza, lResSteady, lhResSteady 2) the remaining techniques.

The last plot (d) shows that Koza, lResSteady and lhResSteady achieved better fitness than the remaining techniques, with Koza improving the fitness sooner.

Low implementation, without upper limit Figure 7.15 shows the results of the comparison among the resource-limited techniques (Low implementation) on the Multiplexer problem, without using an upper limit. The first plot (a) presents the resources used and the number of generations needed to complete the run, as well as the success rate, for each technique. It reveals that convergence to an optimum never happened except with the baseline technique (success rate of 30%). Koza was the technique that exhausted the resources in the fewest generations (51), while both very heavy techniques collapsed the population and required over four million generations to exhaust the resources.

The second plot (b) presents the growth of the average tree size along the run, for each technique. As in the previous problems, all the resource-limited techniques present a large unbounded growth of average tree size, while Koza

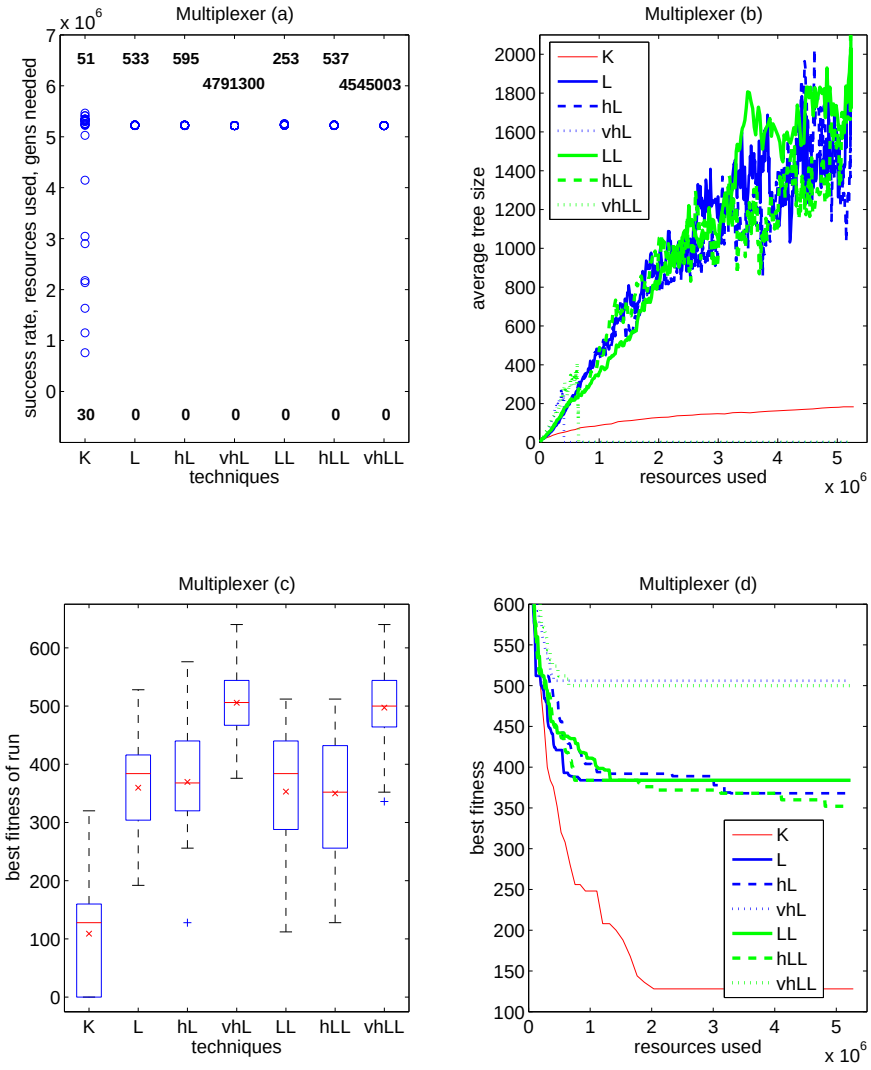

Figure 7.15: Results of the resource-limited techniques (Low implementation) on the Multiplexer problem, without upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

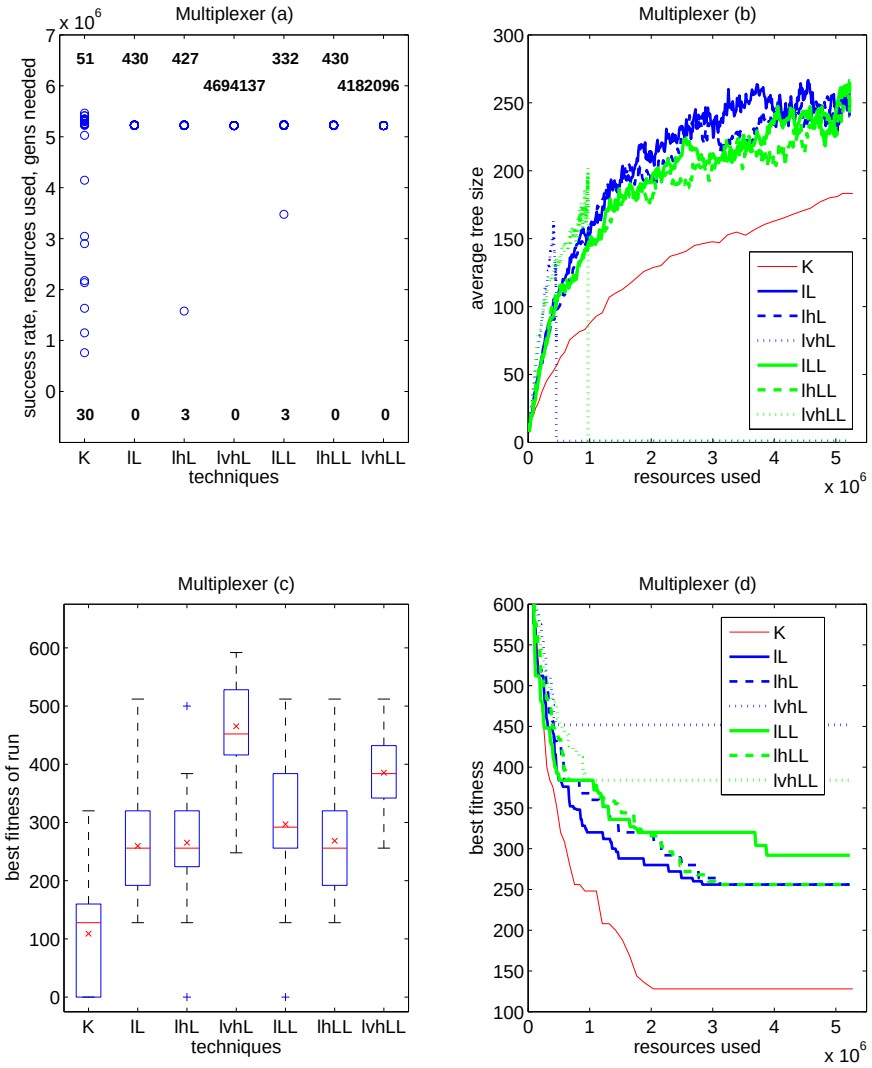

Figure 7.16: Results of the resource-limited techniques (Low implementation) on the Multiplexer problem, with upper limit: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

presents the smallest growth. Both very heavy techniques present a dropping line that represents the collapsing of the population.

The third plot (c) is a boxplot of the best fitness of run obtained with each technique. As before, all the resource-limited techniques performed significantly worse than Koza. Both very heavy variants performed worse than many of the other techniques. Based on these results, a ranking of the several techniques could be: 1) Koza 2) all resource-limited techniques except 3) vhResLow, vhResLowLight.

The last plot (d) presents the evolution of the best fitness as a function of the resources used, for each technique. As before, Koza was able to achieve a much better fitness, from early in the run. The worst performance belongs to both very heavy techniques.

Low implementation, with upper limit Figure 7.16 shows the results of the comparison among the resource-limited techniques (Low implementation) on the Multiplexer problem, using a static upper limit. The first plot (a) shows that convergence to an optimum was again a rare occurrence, except for Koza (30%). Koza was once again the technique that exhausted the resources in the fewest generations (51), with both very heavy techniques again taking millions of generations.

The second plot (b) shows that Koza still presents the smallest growth of average tree size, and both very heavy techniques present a sudden drop where the population collapsed.

In the third plot (c), the boxplot, once again all the resource-limited techniques performed significantly worse than Koza, and both very heavy variants performed worse than many of the other techniques. The ranking of the techniques is the same as without using the upper limit: 1) Koza 2) all resource-limited techniques except 3) lvhResLow, lvhResLowLight.

The last plot (d) shows that Koza still achieved better fitness than the remaining techniques, with the worst performance again belonging to both very heavy techniques.

Remarks The above results reveal that, as in the previous problem, the static upper limit is an essential element for the resource-limited techniques in the Multiplexer problem, without which all the techniques fail when compared to Koza, in the Steady implementation. When using the upper limit, lResSteady and lhResSteady were able to achieve similar performance to Koza. Among the Low implementation techniques, all of them are worse than Koza, with or without using the static upper limit.

7.4 Conclusions

In all problems, among the Steady implementation techniques, lResSteady distinguished itself by achieving similar performance to Koza when using the static upper limit, and converging to an optimal solution early and often when there were no significant differences in terms of best fitness of run. Among the Low

implementation techniques, the heavier variants tend to collapse the population into only a few individuals (generally just one, small), regardless of the difficulty of the problem. For easy problems like Regression, this means good performance, by finding an optimum while saving a large amount of resources. But, for difficult problems like Parity, the population collapses before finding any optima, rendering all further search useless for lack of genetic diversity. Among the Low implementation techniques, none of the variants can be considered a reliable bloat control technique.

Chapter 8

Dynamic Limits and Resource-Limited GP

This chapter compares Dynamic Limits with Resource-Limited GP. It provides a list of the techniques involved and presents the results as described in Section 5.3, also introducing additional plots. It concludes by summing up the major findings.

8.1 Techniques

For comparing Dynamic Limits and Resource-Limited GP, the best techniques from both approaches are selected and compared with each other, and a new hybrid technique is introduced by joining both approaches, thus applying size/depth restrictions both at the individual and at the population level. Table 8.1 lists the names, acronyms and main characteristics of the four techniques involved. Some of these techniques use the static upper limits, and others do not. Some use a variable size population, others do not. All of them are compared to each other, and everything is once again compared to the baseline.

Table 8.1: Techniques involved in the comparison of Dynamic Limits and Resource-Limited GP.

Technique	**Acronym**	**Main characteristics**
Koza	K	static limit, fixed population
DynDepth	D	dynamic limit, fixed population
lResSteady	lS	static limit, dynamic population
Hybrid	H	dynamic limit, dynamic population

8.2 Results

This section presents the results of the comparison between Dynamic Limits and Resource-Limited GP, divided in the four problems considered (see Section 5.1). Short concluding remarks are inserted after each problem, highlighting the best performing techniques.

Additional plots are introduced, showing the evolution of the population size, expressed as the number of individuals, for the techniques using variable size populations. One plot is presented for each problem/technique pair, and each plot contains 30 evolution lines, one line per run.

8.2.1 Symbolic Regression

Figure 8.1 shows the results of comparing among the dynamic limit and resource-limited techniques, and a hybrid, on the Regression problem. The first plot (a) presents the resources used and the number of generations needed to complete the run, as well as the success rate, for each technique. It was already known that convergence to an optimal solution often happens in the Regression problem, and usually very early in the run. lResSteady presents a higher success rate (53%) than DynDepth (47%), and the Hybrid technique presents the lowest success rate, along with Koza (40%). Koza was the technique that used the most resources per generation, requiring fewer generations (50) than the other techniques to exhaust the resources. The techniques using restrictions at the population level, lResSteady and Hybrid, were the most sparing, requiring the highest number of generations (128).

The second plot (b) presents the growth of the average tree size along the run, for each technique. lResSteady presents a short growth line because most of the time it quickly finds an optimal solution. Koza was the technique with the largest growth, while both DynDepth and Hybrid present very similar stabilizing growth curves.

The third plot (c) is a boxplot of the best fitness of run obtained with each technique. There were no significant differences between any of the techniques, therefore it is not possible to rank them.

The last plot (d) presents the evolution of the best fitness as a function of the resources used, for each technique. lResSteady converged to an optimum early in the run, and the Hybrid technique appears to present the worst performance, although not significantly different from the rest.

Figure 8.2 shows the evolution of population size, expressed as the number of individuals, by both techniques that use variable size populations, lResSteady and Hybrid. In both cases there was a rapid shrinking of the population from the very beginning, sometimes reaching only 10% of its initial size. This shrinking was more pronounced in the lResSteady technique. The sudden jump observed in one of the Hybrid runs was the result of a large increase of the resource limit.

Remarks The above results reveal that the Hybrid technique does not improve performance when compared to the other techniques in the Regression problem. All the techniques achieve similar fitness. lResSteady achieves it first, but DynDepth and Hybrid achieve it with lower average tree size.

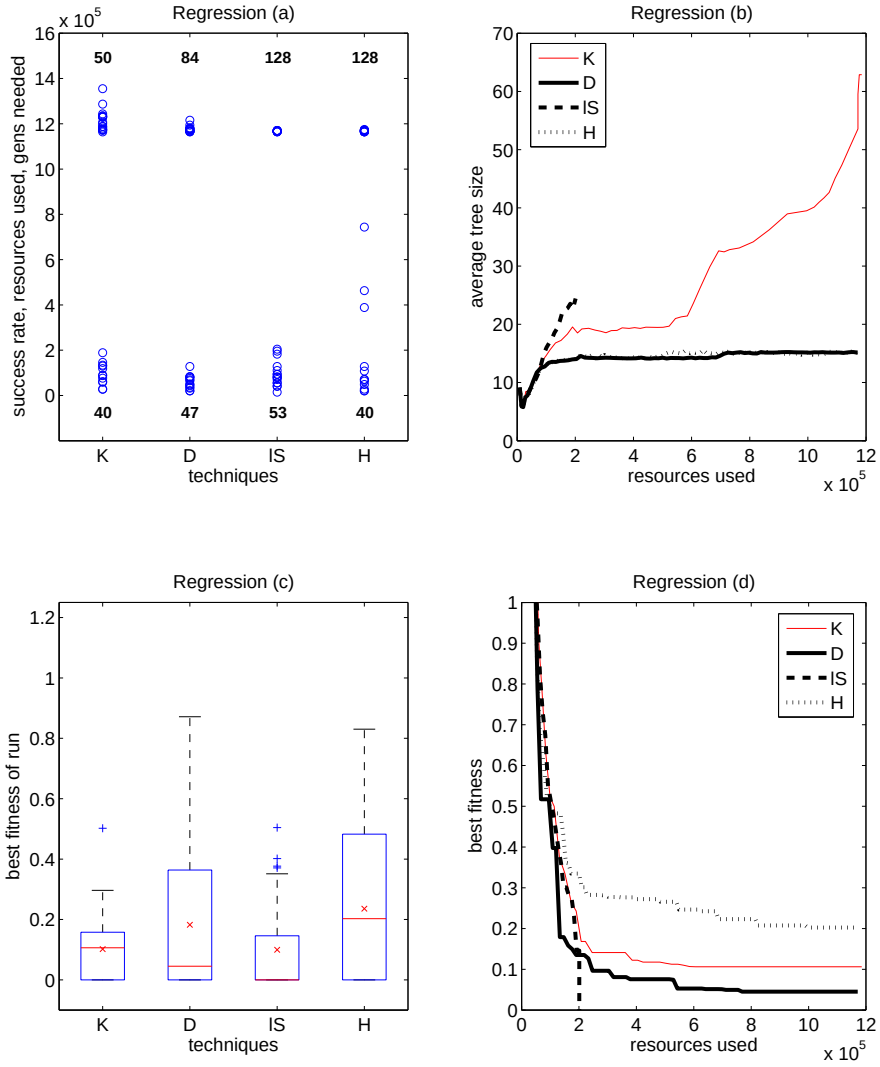

Figure 8.1: Results of comparison among both approaches and a hybrid on the Regression problem: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

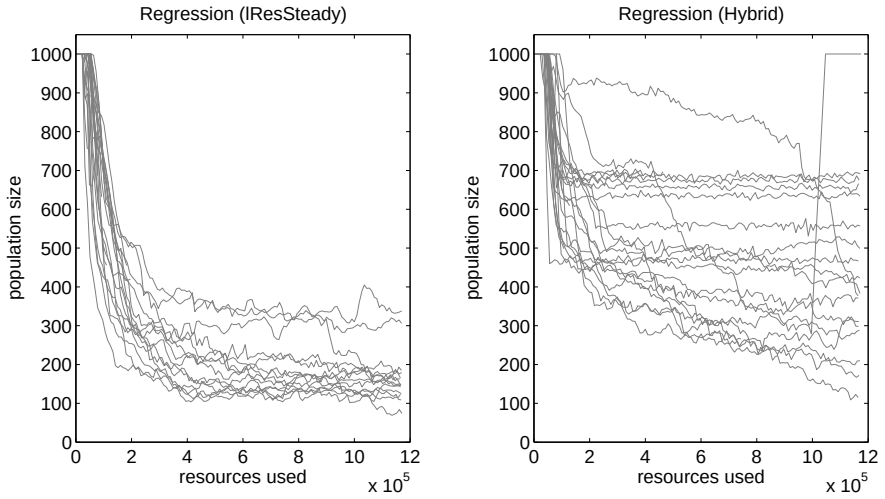

Figure 8.2: Evolution of population size by the limited ResSteady and Hybrid techniques on the Regression problem.

8.2.2 Artificial Ant

Figure 8.3 shows the results of comparing among the dynamic limit and resource-limited techniques, and a hybrid, on the Artificial Ant problem. The first plot (a) shows that success rates were the highest for the techniques using dynamic limits, DynDepth and Hybrid (success rates of 20–30%), followed by the techniques using a static limit, Koza and lResSteady (7–10%). Koza was the technique that required the fewest generations to exhaust the resources (51), while the Hybrid technique was the most resource sparing, requiring the highest number of generations (90).

The second plot (b) shows that lResSteady was the technique allowing the largest growth in average tree size, followed by Koza. DynDepth and Hybrid present the lower tree size values, very similar to each other.

The boxplot (c) reveals that lResSteady achieved significantly worse fitness than DynDepth and Hybrid. Based on this, a rough ranking of the techniques could be: 1) all except 2) lResSteady.

The last plot (d) shows that lResSteady had indeed the worst performance, with a slow fitness improvement from the beginning of the run.

Figure 8.4 shows that, after an initial sudden drop of population size, it tended to rapidly grow larger again, followed by a more gradual shrinking. Some of the runs ended with the initial number of 1000 individuals, more often in the Hybrid technique. In both techniques there was a wide dispersion of behaviors.

Remarks The above results reveal that the lResSteady technique is worse than the others in the Artificial Ant problem. Replacing its static limit with the dynamic limit, creating Hybrid, improved the performance to the same level of DynDepth and Koza.

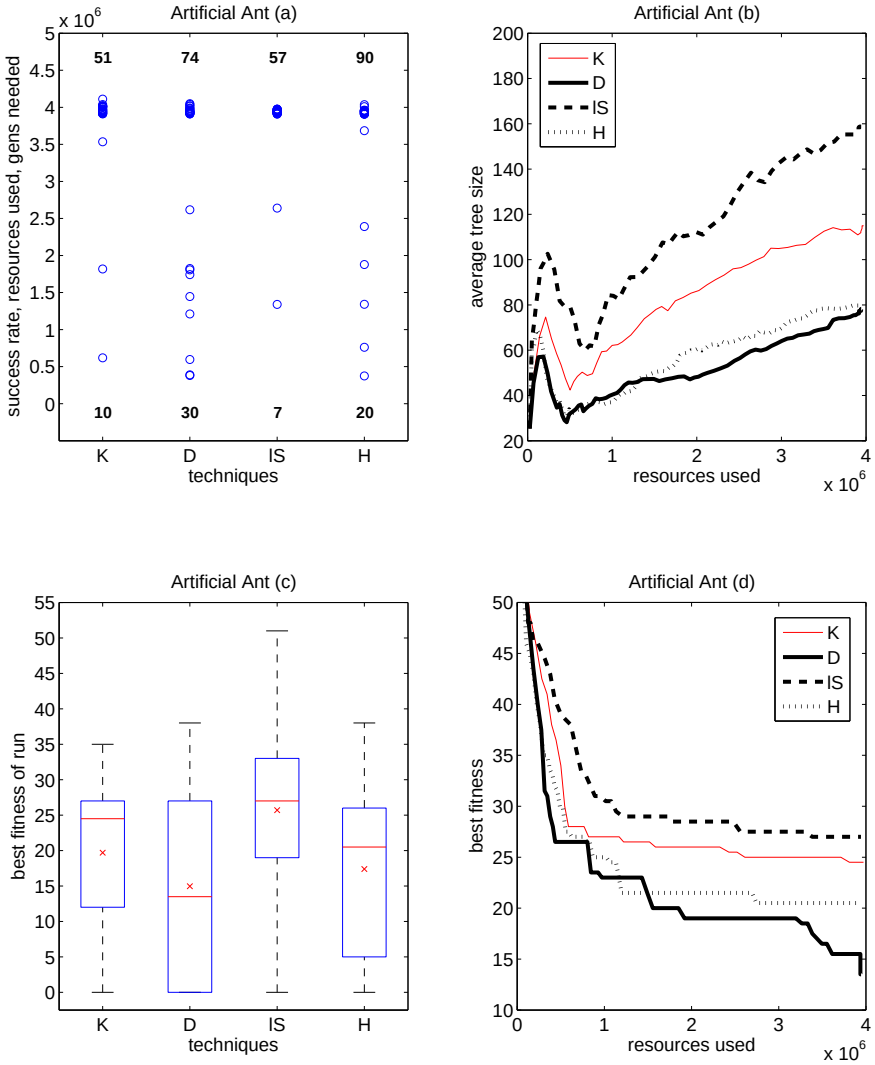

Figure 8.3: Results of comparison among both approaches and a hybrid on the Artificial Ant problem: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

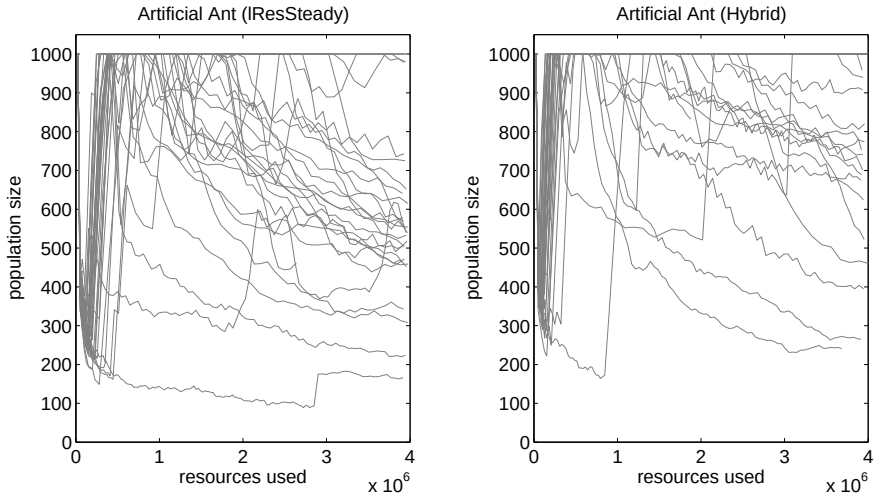

Figure 8.4: Evolution of population size by the limited ResSteady and Hybrid techniques on the Artificial Ant problem.

8.2.3 5-Bit Even Parity

Figure 8.5 shows the results of comparing among the dynamic limit and resource-limited techniques, and a hybrid, on the Parity problem. The first plot (a) shows that that convergence to an optimum never happened, not even for the new technique Hybrid. This was the technique requiring the most generations to exhaust the resources (132), while Koza required the lowest number of generations (48).

The second plot (b) shows that lResSteady was the technique with the fastest growth of average tree size, with Koza presenting a less steep curve that reached the same values by the end of the run. DynDepth and Hybrid present the lowest values, very similar to each other.

The boxplot (c) reveals that, as in the previous problem, lResSteady achieved significantly worse fitness than DynDepth and Hybrid. The proposed ranking of techniques is the same as previously: 1) all except 2) lResSteady.

The last plot (d) confirms the results of the boxplot, with lResSteady performing worse than the remaining techniques.

Figure 8.6 shows that, after an initial sudden drop of population size, it eventually climbed rapidly to high values, often followed by a gradual shrinking. In both techniques there was a wide dispersion of the final values of population size.

Remarks The above results reveal that the lResSteady technique is worse than the others in the Parity problem. The new Hybrid technique performs at the same level as DynDepth and Koza.

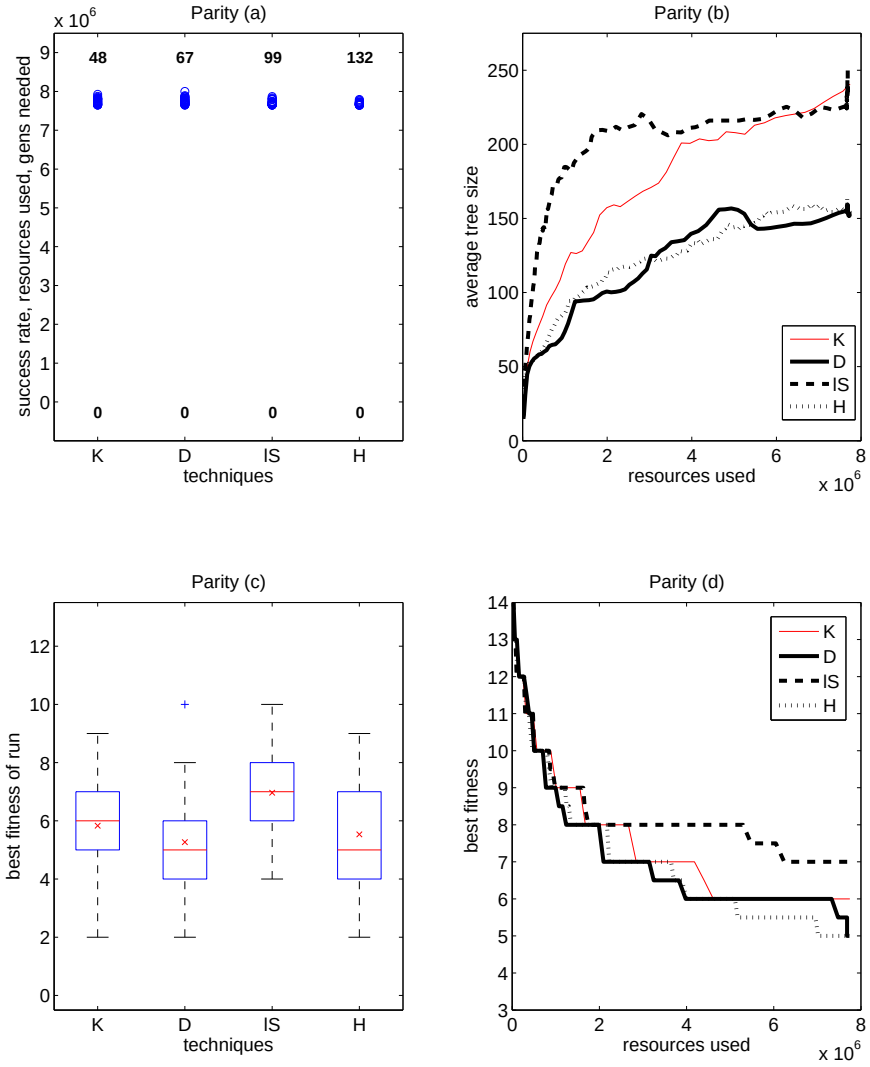

Figure 8.5: Results of comparison among both approaches and a hybrid on the Parity problem: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

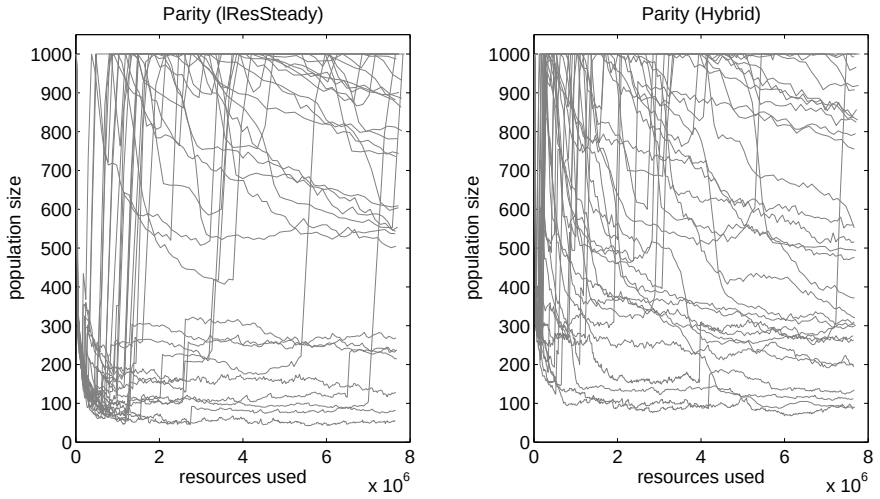

Figure 8.6: Evolution of population size by the limited ResSteady and Hybrid techniques on the Parity problem.

8.2.4 11-Bit Boolean Multiplexer

Figure 8.7 shows the results of comparing among the dynamic limit and resource-limited techniques, and a hybrid, on the Multiplexer problem. The first plot (a) shows that success rates were higher for Koza and DynDepth (23–30%) than for the techniques using dynamic populations (10–13%). Koza was the technique exhausting the resources in the fewest generations (51), while Hybrid required the highest number of generations (116).

The second plot (b) shows a similar behavior to the previous problem, with lResSteady and Koza reaching the highest average tree size, followed by the similar DynDepth and Hybrid techniques.

The boxplot (c) contains no significant differences between any of the techniques, therefore no ranking is possible.

The last plot (d) confirms the results of the boxplot, also revealing a somewhat slower fitness improvement by the lResSteady technique.

Figure 8.8 reveals an initial sudden drop of population size, followed by a rapid climb back to high values, and a tendency for a slow decrease afterwards, with the occasional climbing back, more frequent in the lResSteady technique. Once again there was a wide dispersion of the final values of population size in both techniques.

Remarks The above results reveal that the Hybrid technique does not improve performance when compared to the other techniques in the Multiplexer problem. All the techniques achieve similar fitness, but lResSteady produces the undesirable higher average tree size.

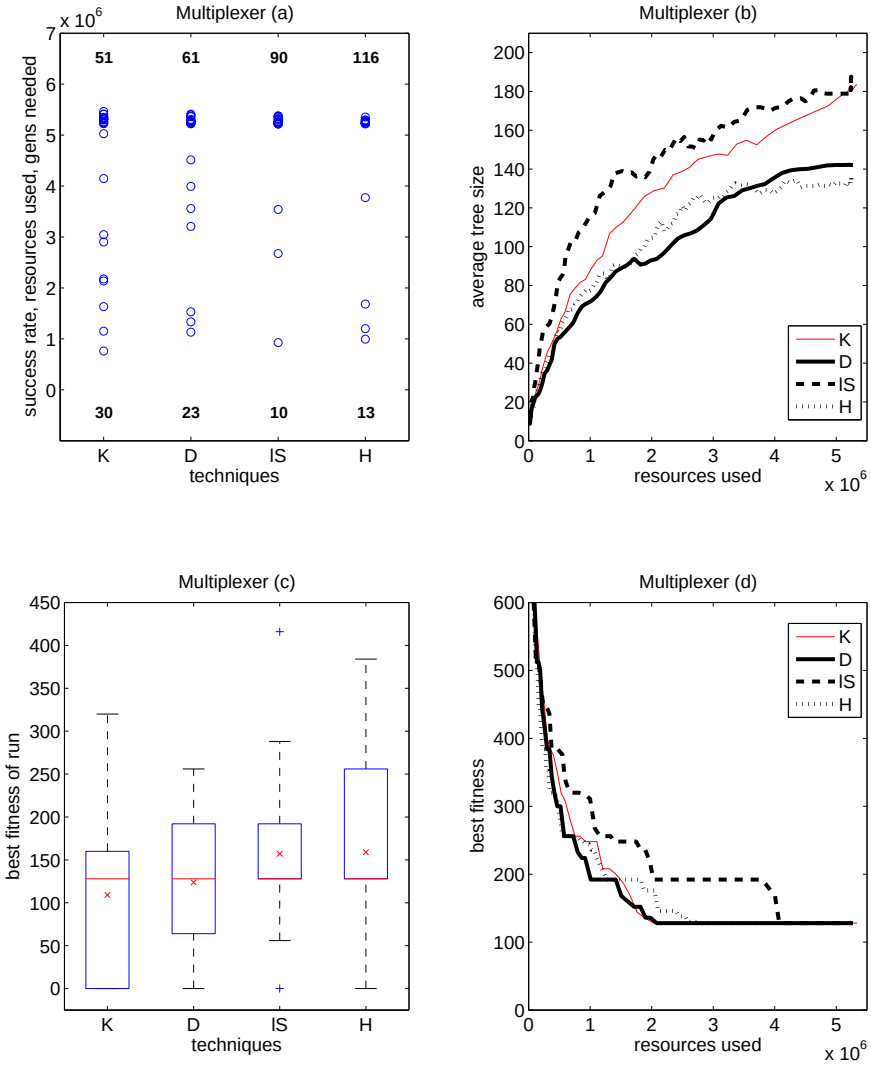

Figure 8.7: Results of comparison among both approaches and a hybrid on the Multiplexer problem: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

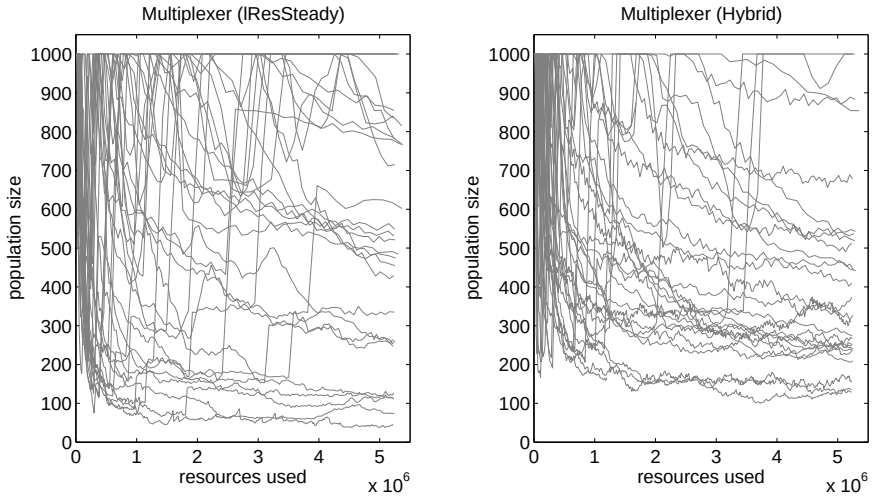

Figure 8.8: Evolution of population size by the limited ResSteady and Hybrid techniques on the Multiplexer problem.

8.3 Conclusions

In all problems, the new technique Hybrid performed as well as the baseline Koza and the very successful DynDepth. lResSteady was almost always the worst technique, either achieving worse fitness than DynDepth and Hybrid, converging to an optimum later, or producing larger trees than the other techniques. The evolution of the population size reveals wide differences between the several problems. In the Regression problem there is mainly a constant smooth decrease of population size, whereas in the other problems, particularly Parity and Multiplexer, the evolution of population size follows a jumpy and unpredictable pattern.

Chapter 9

Comparison with State of the Art

This chapter compares the best techniques from the previous experiments with some successful state-of-the-art techniques. It lists and describes the state-of-the-art techniques involved, presents the results as described in Section 5.3, and concludes by summing up the major findings.

9.1 Techniques

After selecting the best techniques from the previous experiments, these are compared with the following state-of-the-art bloat control methods: Linear Parametric Parsimony Pressure, Double Tournament, and Dynamic Populations. The first two were the best performing of a fairly extensive set of techniques tested by Luke and Panait [68], and the third is based on the few examples of recent work dealing with dynamic populations [121, Sect. 7.1] [19,94,95,118]. Table 9.1 shows the names and acronyms of the techniques compared, and a description of the involved state-of-the-art techniques follows.

9.1.1 Linear Parametric Parsimony Pressure

Parsimony pressure is a family of bloat control methods where the size of an individual affects its probability of being selected for reproduction. In linear parametric parsimony pressure, the adjusted fitness of an individual (g) is computed as a function of its raw fitness (f) and its size (s), that is, $g = xf + ys$. The experiments of this thesis using linear parametric parsimony pressure follow the settings used in [68], where the adjustment function always considers $y = 1$, and the best cross-problem performance was obtained with $x = 32$. When a fitness adjustment function is used, the selection of individuals (for reproduction and for survival) is based on the adjusted fitness, but all the results are presented using the raw fitness.

Table 9.1: Techniques involved in the comparison with state of the art.

Technique	Acronym	Brief description
Koza	K	traditional static upper limit
DynDepth	D	dynamic limit on depth
Hybrid	H	DynDepth with dynamic populations
Linear	Lnr	linear parametric parsimony pressure
Double	Dbl	double tournament
Dynamic Populations 1	M1	dynamic populations, variant M1
Dynamic Populations 2	M2	dynamic populations, variant M2

9.1.2 Double Tournament

This technique uses a double tournament that applies two layers of tournaments in series, first for fitness and then for parsimony (or the other way around) [68]. A parameter determines whether the first tournament selects based on fitness and the second one based on size (number of nodes), or vice versa. A second parameter is D, where $0.5 \leq D \leq 1$. When two individuals participate in the parsimony tournament, the smaller one wins with probability D, else the larger wins. $D = 0.5$ is random selection, while $D = 1$ is a plain parsimony tournament of size 2. The experiments using double tournament follow the settings used in [68], where the first tournament is based on size and the second on fitness, and $D = 0.7$. The fitness tournament is the same used with all the other techniques, with size 7 (0.7% of the population, with a minimum of 2, for the variable population techniques).

9.1.3 Dynamic Populations

Dynamic population techniques allow the population size to vary along the generations, by adding or suppressing individuals from the population depending on how well fitness is evolving. Basically, individuals are suppressed as long as the best individual keeps improving, and new individuals are added when the fitness stagnates, in all the variants developed so far [121, Sect. 7.1] [19, 94, 95, 118].

The experiments of this thesis using dynamic population techniques follow a mixture of settings from the different variants, some selected after conversations with the respective authors. In the beginning of the run, a parameter *pivot* is calculated by dividing the best fitness at the initial generation (f_0) by the maximum allowed number of generations for that particular run (g_{max}). During the run, every *period* generations the difference between the current best fitness (f_g) and the best fitness *period* generations back ($f_{g-period}$) is computed, and divided by *period*. The result is stored in the parameter *delta*. Every generation, if *delta* is larger than *pivot*, individuals are deleted, otherwise individuals are

added. All experiments used $period = 1$, as this parameter does not seem to be very influential [95].

The number of individuals to delete from the population is a portion of the current population size, proportional to the gain in fitness from generation $g - period$ to generation g. It is calculated as $P_g * period * (f_{g-period} - f_g)/f_{g-period}$, where P_g is the population size at the current generation. The worst individuals are deleted. At least one individual is kept in the population, to avoid extinction. The number of individuals to add is calculated in order to achieve a certain population size in case the fitness stagnation continues. Different choices regarding this intended population size created two implementation options called *M1* and *M2*. In M1 the aim is to reach the exact initial population size by the end of the run, so the number of individuals to add is calculated as $(P_0 - P_g)/(g_{max} - g)$, where P_0 is the initial population size and g is the current generation. In M2 the intended population size is a proportion of the initial population size. The number of individuals to add is calculated as $(c * P_0 - P_g)/(g_{max} - g)$, where $c = \sqrt{f_g/f_0}$. Individuals are added by mutating the best individuals in the population, using shrink and swap mutation with equal probability.

As in Resource-Limited GP, dynamic populations vary their number of individuals along the run, but the philosophies behind these two methods are notably different. If bloat occurs in Resource-Limited GP, this immediately results in the removal of the individuals 'not good enough for their size' and the population size is reduced. But in dynamic populations, if bloat happens to cause fitness stagnation, paradoxically this causes the population size to increase. On the other hand, when convergence to better solutions is good and the population quickly moves to the optimum, dynamic populations readily speed this process by continuously reducing the population size.

The immediate handicap of the dynamic population techniques is, however, the need to use the maximum number of generations to calculate *pivot*. In practical terms, this may not be viable and has even caused some trouble in simple experiments like the ones in this thesis, since the runs do not have a fixed number of generations. So *pivot* was calculated using the maximum number of generations 50, and for those runs where this was not enough to exhaust the resources, *pivot* was recalculated using an increased maximum number of generations (in multiples of 10), and the runs continued.

9.2 Results

This section presents the results of the comparison with the state of the art, divided in the four problems considered (see Section 5.1). Short concluding remarks are inserted after each problem, highlighting the best performing techniques.

9.2.1 Symbolic Regression

Figure 9.1 shows the results of comparing the best techniques from Dynamic Limits and Resource-Limited GP with the best state-of-the-art techniques, on the Regression problem. The first plot (a) presents the resources used and

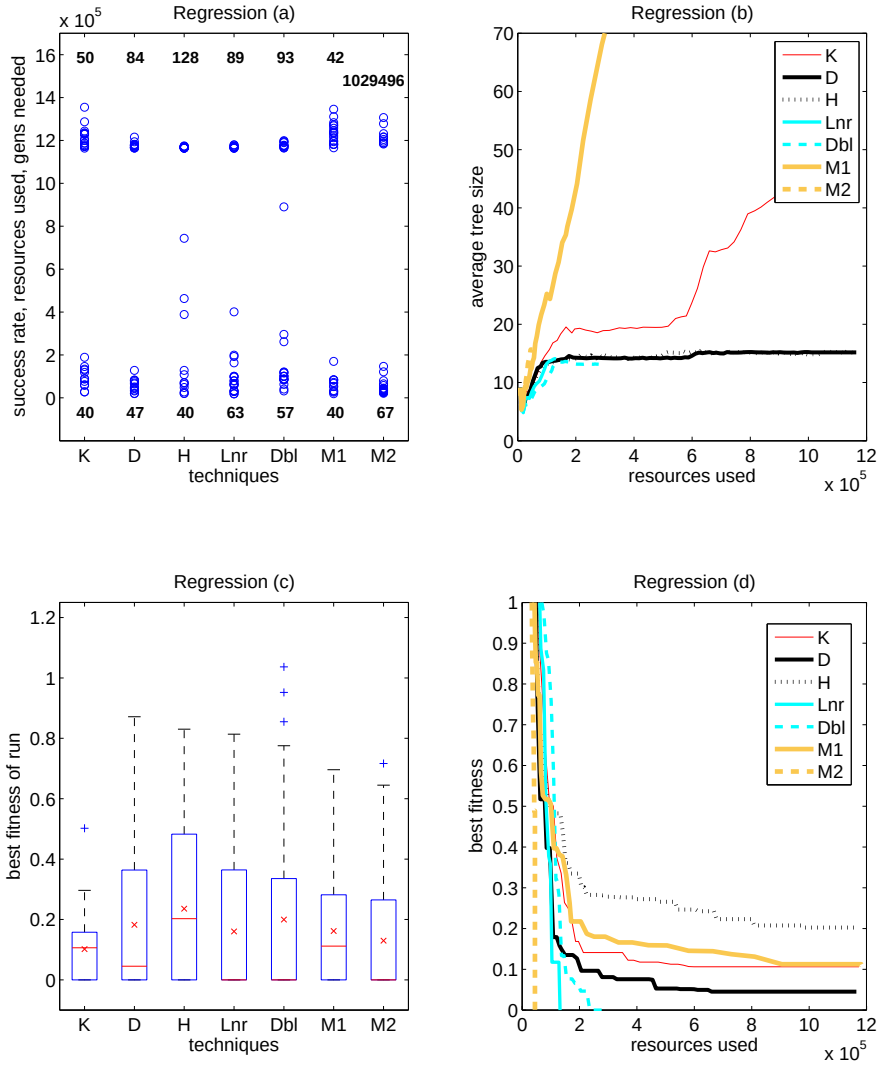

Figure 9.1: Results of comparison with state-of-the-art techniques on the Regression problem: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

the number of generations needed to complete the run, as well as the success rate, for each technique. Convergence to an optimal solution happens often and early in the Regression problem. Except for M1, the state-of-the-art techniques achieved the highest success rates (57–67%), followed by DynDepth (47%) and the remaining techniques (40%). M1 was the technique that used the most resources per generation, requiring less generations (42) than the other techniques to exhaust the resources. M2, on the other hand, collapsed the population into only a few individuals, requiring more than a million generations to exhaust the resources.

The second plot (b) presents the growth of the average tree size along the run, for each technique. M1 presents the largest growth, followed by the distant second Koza, and finally the remaining techniques. M2, Linear and Double found optimal solutions early, presenting short growth lines.

The third plot (c) is a boxplot of the best fitness of run obtained with each technique. There were no significant differences between any of the techniques, therefore it is not possible to rank them.

The last plot (d) presents the evolution of the best fitness as a function of the resources used, for each technique. M2, Linear and Double found optimal solutions very rapidly; Hybrid achieved the worst results, although not significantly different from the rest.

Remarks The above results do not permit a choice of best bloat control technique in the Regression problem. M1 is considered the worst technique because it reaches a very large average tree size.

9.2.2 Artificial Ant

Figure 9.2 shows the results of comparing the best techniques from Dynamic Limits and Resource-Limited GP with the best state-of-the-art techniques, on the Artificial Ant problem. The first plot (a) shows that success rates were very low for the M1 and M2 techniques (0–7%). Linear and Double once again reached the highest success rates (33–37%), followed the remaining techniques (10–30%). M1 and M2 were the techniques requiring the fewest generations to exhaust the resources (41–47), while Linear was the most resource sparing, requiring the most generations (108).

The second plot (b) shows that M1 and M2 were the techniques allowing the largest growth in average tree size, followed by the distant second Koza. Double presents similar growth to DynDepth and Hybrid, while Linear presents the smallest growth.

In the boxplot (c), M1 was worse than all other techniques, and M2 was worse than Linear and Double. Based on these results, a rough ranking of the techniques could be: 1) all except 2) M2 and 3) M1.

The last plot (d) shows that Double succeeded in finding an optimum faster than the other techniques, followed closely by Linear. M1 had indeed the worst performance, with a slow improvement from the beginning of the run.

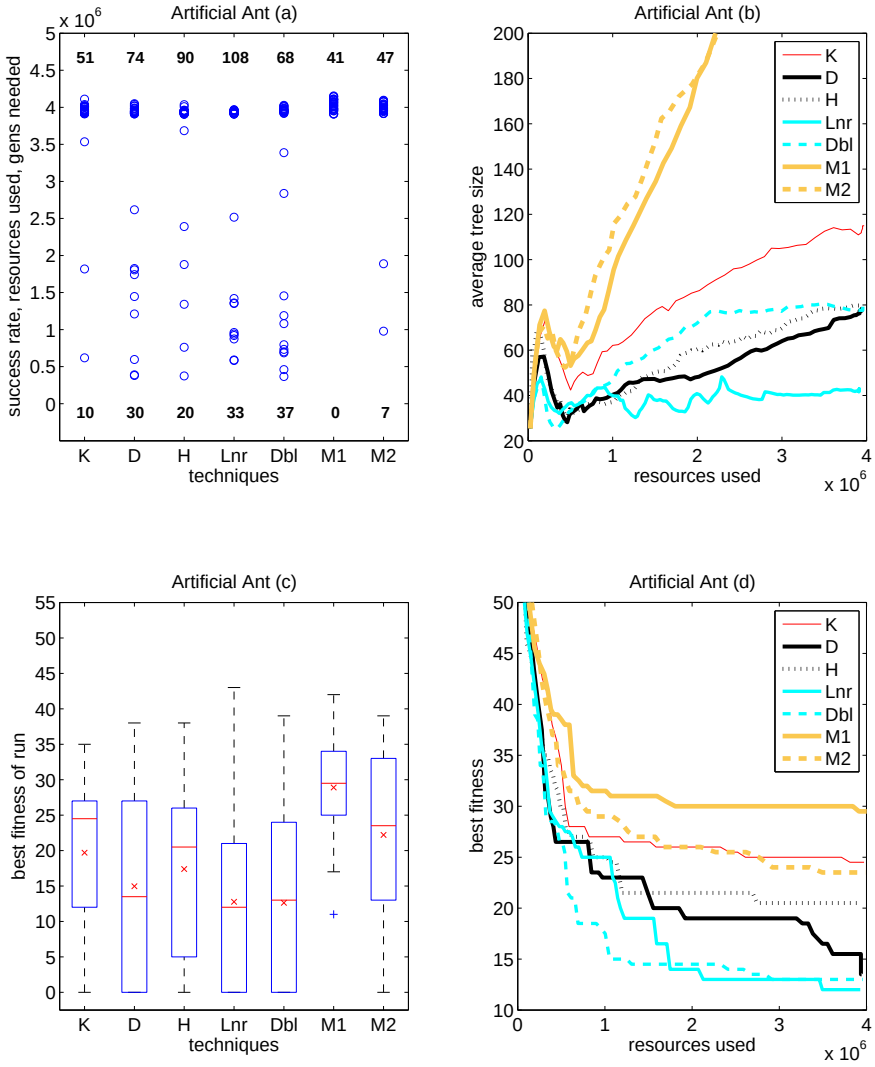

Figure 9.2: Results of comparison with state-of-the-art techniques on the Artificial Ant problem: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

Remarks The above results do not permit a choice of best bloat control technique in the Artificial Ant problem. M1 and M2, however, perform worse than the rest.

9.2.3 5-Bit Even Parity

Figure 9.3 shows the results of comparing the best techniques from Dynamic Limits and Resource-Limited GP with the best state-of-the-art techniques, on the Parity problem. The first plot (a) shows that convergence to an optimum never happened, not even for the state-of-the-art techniques. M1 was once again the technique requiring the fewest generations to exhaust the resources (44), followed closely by Koza and M2 (48). Hybrid and Linear were the techniques requiring the most generations (130–132).

The second plot (b) shows that, once again, M1 and M2 were the techniques with the fastest growth of average tree size, followed by the distant second Koza, then Double, then DynDepth and Hybrid, very similar to each other. Linear presents the lowest growth.

In the boxplot (c), M1 and M2 present worse performance than DynDepth. Based on these results, a rough ranking of the techniques could be: 1) all except 2) M1 and M2.

The last plot (d) shows that most techniques achieved similar fitness with approximately the same amount of resources.

Remarks The above results only permit the exclusion of M1 and M2 as adequate bloat control methods, in the Parity problem.

9.2.4 11-Bit Boolean Multiplexer

Figure 9.4 shows the results of comparing the best techniques from Dynamic Limits and Resource-Limited GP with the best state-of-the-art techniques, on the Multiplexer problem. The first plot (a) shows that convergence to an optimum almost never happened for M1 and M2 (success rates of 0–3%). Double reached the highest success rate (40%), followed by the remaining techniques (13–30%). Koza was the technique exhausting the resources in the fewest generations (51), followed closely by M1 and M2 (54–57). Linear required the highest number of generations (141).

The second plot (b) shows that, once again, M1 and M2 present the largest growth in average tree size, followed by the distant second Koza. DynDepth and Hybrid once again reached similar values, followed by Double and finally the Linear technique, where the average tree size increased and then began to decrease and stabilize.

In the boxplot (c), M1 and M2 present worse performance than Koza, DynDepth, and Double. Hybrid and Linear were also worse than Double. Based on these results, a rough ranking of the techniques could be: 1) Koza, DynDepth, Double 2) Hybrid, Linear, M1, M2.

The last plot (d) clearly shows the worse performance of the M1 and M2 techniques, and also reveals that Double achieved better fitness earlier than the rest.

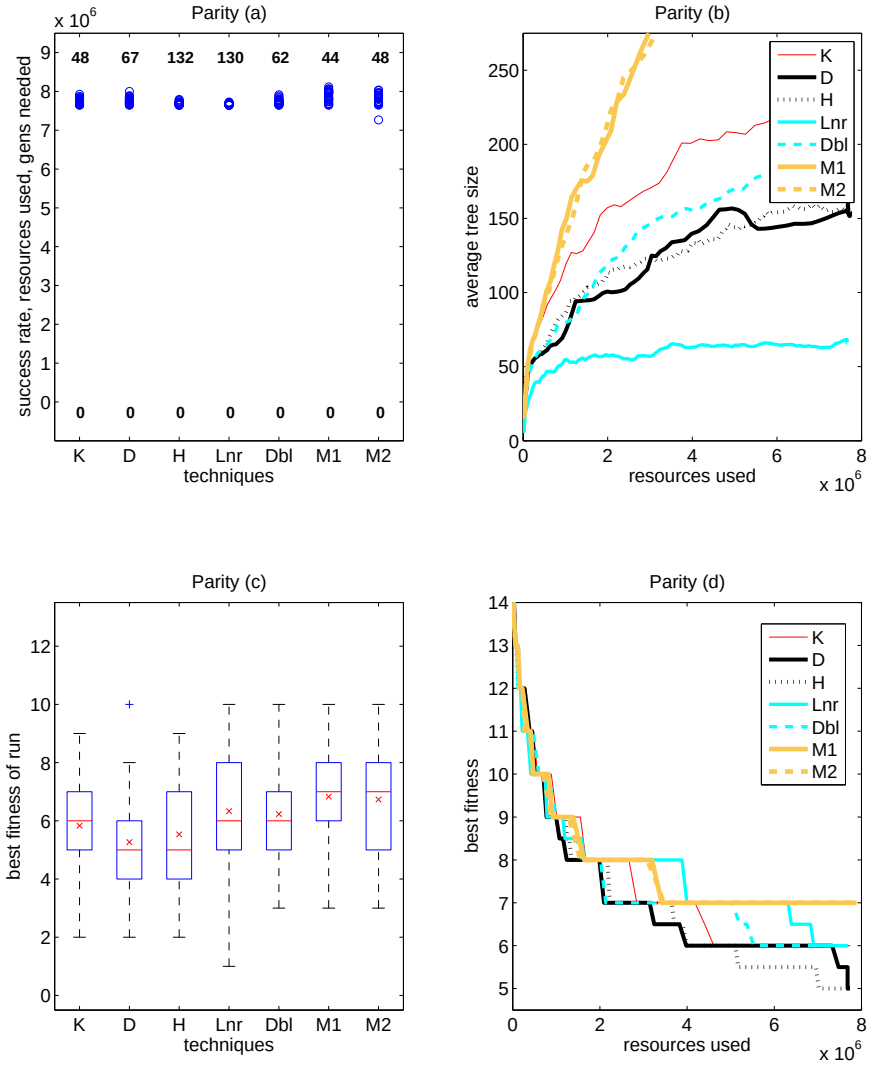

Figure 9.3: Results of comparison with state-of-the-art techniques on the Parity problem: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

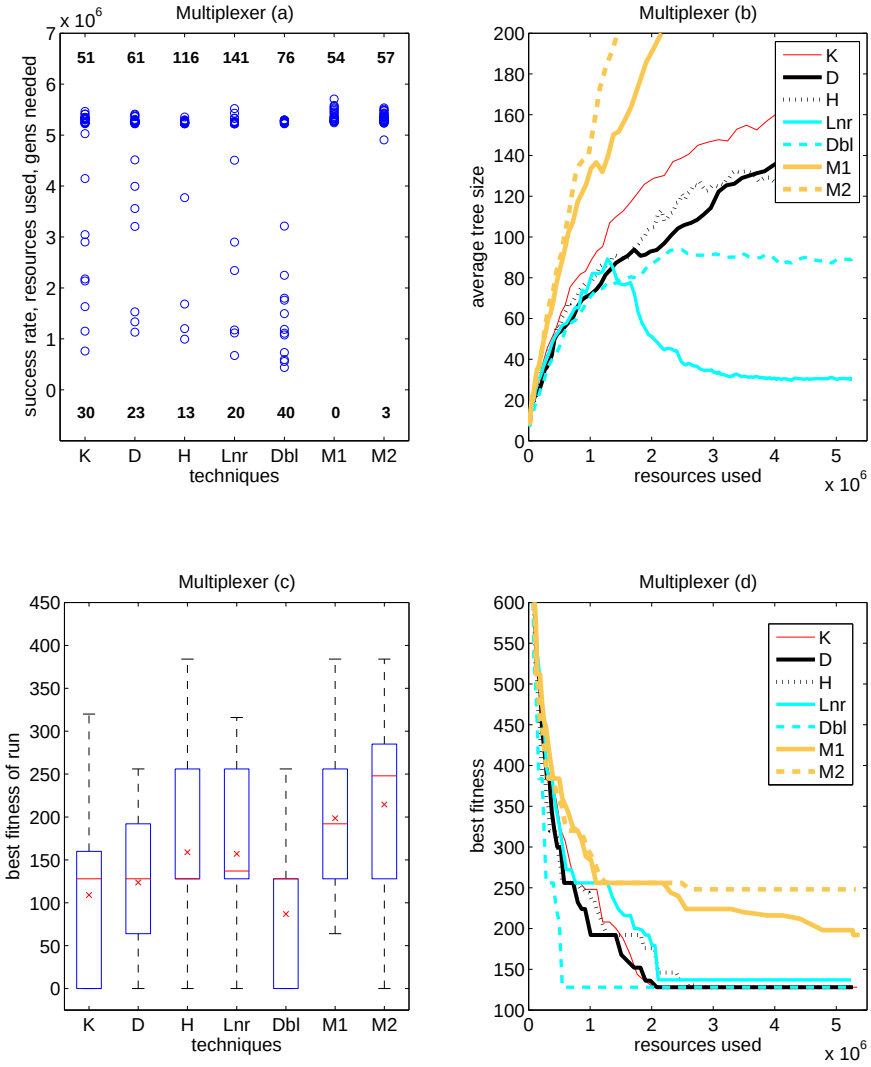

Figure 9.4: Results of comparison with state-of-the-art techniques on the Multiplexer problem: (a) success rate, resources used, and number of generations needed; (b) growth of average tree size; (c) best fitness of run; (d) evolution of best fitness.

Remarks The above results reveal that DynDepth and Double, along with the baseline technique Koza, are the best bloat control techniques in the Multiplexer problem.

9.3 Conclusions

In all problems, the techniques achieving best fitness of run were the baseline Koza, the dynamic depth technique DynDepth, and the state-of-the-art technique Double. This last technique was also able to consistently achieve higher success rates and find an optimal solution earlier than the rest. In terms of average tree size, Koza always reached higher values, while the relative results of DynDepth and Double depended on the problem considered.

Chapter 10

Discussion

In the previous chapters, the two original approaches to bloat control, Dynamic Limits and Resource-Limited GP, have been compared within themselves and between each other, and also against some of the best state-of-the-art techniques. This chapter discusses their global performance, and also presents a deeper analysis concerning the reasons why each approach achieved different results in the various problems considered. Closing the chapter are some brief concerns and considerations on how some implementation details may influence the search dynamics, particularly in the light of the most recent explanation for bloat, the crossover bias theory.

10.1 Global Performance

From all the experiments reported in the previous chapters, one of the techniques belonging to the Dynamic Limits approach, DynDepth, has proven to always achieve results as good as the best state-of-the-art technique that was tested, Double. Although Double was generally able to reach the optimum more easily, it has the disadvantage of being constrained by a tournament selection method whose performance may be dependent on its settings (see Section 9.1.2). The settings chosen for these experiments were expected to yield good results, as they were the best previously found across the set of studied problems [68]. DynDepth, on the other hand, is totally parameterless and, because it only acts during the survival phase, can be used along with any selection method. This, and the fact that it does not require the presence of the static upper limit for a good performance, are the two characteristics that make the Dynamic Limits an appealing bloat control approach.

Although already expected from the observation of early results [102], the poorer performance of the dynamic limit on size does not have to be final. This variant managed to produce good results in the Artificial Ant problem, and with the right improvement (Section 11.2) it could still be a capable replacement for the dynamic limit on depth, one that would be suited for linear as well as tree-based GP. It may not be a coincidence that it was precisely in the Artificial Ant problem that the average tree size was considerably higher for the dynamic size

than for the dynamic depth. Could it be that the poor results of the dynamic limit on size were simply caused by too much parsimony pressure?

It is arguable whether limiting size instead of depth increases the amount of parsimony pressure. When dealing with depth, there is usually the opportunity to add more nodes without breaking the limit, because trees are always far from full [102]. On the other hand, looking at size regardless of depth removes a very important restriction from the search: the shape of the trees. Regardless of the number of necessary nodes, low-depth solutions may be harder to find than their deeper equivalents. For example, in symbolic regression of the quartic polynomial, $x^4 + x^3 + x^2 + x$, tree-based GP usually finds a solution of depth 7, whereas solutions of depth 6, 5, and even 4[1] are rarely found. When using the size limit instead of the depth limit, trees are comparatively less full, and a higher population diversity is achieved [102].

Resource-Limited GP did not perform as well as Dynamic Limits in most of the problems considered. The Low implementation did not achieve acceptable results in most problems, while the Steady implementation greatly relied on the presence of the static upper limit to achieve a performance comparable to the Koza technique. The lResSteady technique, one of the best resource-limited techniques, was in most problems worse than DynDepth and the Hybrid technique, achieving worse fitness, converging to an optimum later, or producing larger trees. In some cases it produced trees larger than the Koza technique.

So, is Resource-Limited GP preventing bloat, or simply learning to live with it? The emergent rule that prevents the survival of 'not good enough for their size' individuals (see Section 4.1) actually ensures that bloat does not occur, as long as we define bloat as "an excess of code growth without a corresponding improvement in fitness" (from Section 1.1). But how much code growth is considered *excessive*? And how much fitness gain is considered a *corresponding* improvement? The fact is, Resource-Limited GP allows seemingly free code growth, and the modest resource usage is only obtained thanks to a prompt reduction of population size. In the absence of restrictions at the individual level, nothing is done to directly counteract bloat.

No doubt it is the limits at the individual level that have the main role in controlling tree growth. Between the DynDepth and lResSteady techniques there are considerable differences in average tree size. But when the static depth limit of lResSteady is replaced by a dynamic limit, the new Hybrid technique produces similar growth curves to DynDepth, not to lResSteady. Regardless of tree growth, the limits at the individual level also seem to be the main contributors to an improved best fitness. When using either a static or dynamic limit, replacing a fixed population with a dynamic population (e.g. Koza versus lResSteady, DynDepth versus Hybrid) did not improve the performance; but when using either a fixed or dynamic population, replacing a static limit with a dynamic limit (e.g. Koza versus DynDepth, lResSteady versus Hybrid) did improve the results in many cases. The merit belongs to the dynamic limit, not to the variable size population.

Still, the Resource-Limited GP techniques, both Steady and Low imple-

[1]The factored form of the polynomial, $(x^2 + 1)(x^2 + x)$, can be represented with a tree of depth 4.

mentations, produced excellent results on the Regression problem (see Section 10.2.1), the only problem where Dynamic Limits sometimes failed to perform at the same level as the Koza technique. In the next section an attempt is made to find a reason for such behavior.

Despite the good results obtained by M1 and M2 in previous work [94, 95], here they failed to pass the tests. These techniques seem to be highly dependent on a static upper limit, or maybe on a specific set of parameters that was not used here. The Linear technique, although producing good results, performed worse than DynDepth and Double, even though it had been the best technique in previous comparisons [68]. It, too, seems to be highly dependent on the choice of parameters. Finally, Koza remains a very successful technique, performing as well as the best, but the static nature of its limit may prevent it from finding the solution for problems of unsuspected high complexity.

10.2 Problemwise Analysis

This section summarizes the performance of Dynamic Limits and Resource-Limited GP on each of the four problems considered, and proceeds to describe some of the characteristics that may explain the differences observed in the quality of the results.

10.2.1 Performance

Tables 10.1 through 10.4 show the relative performance of all the techniques among Dynamic Limits and Resource-Limited GP when compared to the baseline Koza, in terms of success rate. The success rates of all the techniques, expressed as the percentage of runs where an optimal solution was found, were presented in Chapters 6 through 9. The success rates for the Koza technique were 40, 10, 0, and 30 for the Regression, Artificial Ant, Parity, and Multiplexer problems, respectively. The present tables only indicate, for each technique, whether the success rate increased ($\nearrow$), decreased ($\searrow$), or was maintained ($\rightarrow$), not the absolute values. Bold arrows indicate changes equal to or larger than 10 percentual points. The absence of an arrow indicates the technique achieved significantly worse fitness than Koza. For each group of techniques, the first column of arrows refers to the ones not using a fixed limit, and the second column (identified by (l)) refers to the ones using the static upper limit.

Table 10.1 clearly shows what has already been stated in the previous section: both Steady and Low implementations of Resource-Limited GP produced excellent results on the Regression problem. In the vast majority of cases, the success rate was increased, very often 10 or more percentual points. Dynamic Limits, on the other hand, produced poor results on this problem. The techniques based on size achieved significantly worse fitness than Koza, as did most of the depth-based techniques not using the upper limit. Only DynDepth was able to produce similar fitness to Koza, and slightly increase the success rate.

Table 10.2 shows that Dynamic Limits performed very well on the Artificial Ant problem. Most of the techniques were able to increase the success rate, in particular the variants based on size, with improvements of 10 or more per-

Table 10.1: Relative performance in the Regression problem. Arrows indicate whether the success rate increased, decreased, or was maintained when compared to Koza. Bold arrows indicate changes equal to or larger than 10 percentual points. The absence of an arrow indicates the technique achieved significantly worse fitness than Koza. Techniques using the static upper limit are identified by (l).

Dynamic Limits		**Resource-Limited (Steady)**		**Resource-Limited (Low)**	
	(l)		(l)		(l)
DynDepth	↗ ↗	ResSteady	**↗** **↗**	ResLow	→ **↗**
hDynDepth	↘	hResSteady	↗ **↗**	hResLow	↗ **↗**
vhDynDepth	**↘**	vhResSteady	↗ →	vhResLow	**↗**
DynNodes		ResSteadyLight	↗ **↗**	ResLowLight	**↗** **↗**
hDynNodes		hResSteadyLight	**↗** **↗**	hResLowLight	**↗** **↗**
vhDynNodes		vhResSteadyLight	**↗** ↗	vhResLowLight	**↗** **↗**

Table 10.2: Relative performance in the Artificial Ant problem. For details see caption of Table 10.1.

Dynamic Limits		**Resource-Limited (Steady)**		**Resource-Limited (Low)**	
	(l)		(l)		(l)
DynDepth	**↗** →	ResSteady	↘ ↘	ResLow	
hDynDepth	↗ **↗**	hResSteady	→ ↗	hResLow	
vhDynDepth	→ →	vhResSteady	↘ ↗	vhResLow	
DynNodes	**↗** **↗**	ResSteadyLight	→ ↘	ResLowLight	
hDynNodes	**↗** **↗**	hResSteadyLight	↘	hResLowLight	
vhDynNodes	**↗** **↗**	vhResSteadyLight	↘	vhResLowLight	

Table 10.3: Relative performance in the Parity problem. For details see caption of Table 10.1.

Dynamic Limits		**Resource-Limited (Steady)**		**Resource-Limited (Low)**	
	(l)		(l)		(l)
DynDepth	→ →	ResSteady	→	ResLow	→
hDynDepth	→ →	hResSteady	→	hResLow	
vhDynDepth	→ →	vhResSteady	→	vhResLow	
DynNodes	↗	ResSteadyLight		ResLowLight	→
hDynNodes	→ →	hResSteadyLight		hResLowLight	→
vhDynNodes	→	vhResSteadyLight		vhResLowLight	

Table 10.4: Relative performance in the Multiplexer problem. For details see caption of Table 10.1.

Dynamic Limits			**Resource-Limited (Steady)**		**Resource-Limited (Low)**	
		(l)		(l)		(l)
DynDepth	↘	↗	ResSteady	↘	ResLow	
hDynDepth	↗	↘	hResSteady	↘	hResLow	
vhDynDepth	↗	→	vhResSteady		vhResLow	
DynNodes	↘	↘	ResSteadyLight		ResLowLight	
hDynNodes		↘	hResSteadyLight		hResLowLight	
vhDynNodes			vhResSteadyLight		vhResLowLight	

centual points. In terms of Resource-Limited GP, the non-light techniques of the Steady implementation did fairly well, all of them achieving the same fitness as Koza and maintaining or slightly changing the success rate. Some of the light variants of the Steady implementation, as well as all the variants of the Low implementation, achieved significantly worse fitness than Koza.

Table 10.3 shows that most of the success rates on the Parity problem remained unchanged, and none decreased. This is not surprising, since the success rate of the baseline Koza is null. All the depth-based variants of Dynamic Limits performed well, whereas some of the size-based variants achieved significantly worse fitness than Koza. In terms of Resource-Limited GP, only the limited non-light variants of the Steady implementation performed well, the rest being unreliable in terms of best fitness achieved.

Table 10.4 shows that the depth-based variants of Dynamic Limits performed well on the Multiplexer problem, whereas the size-based variants either achieved significantly worse fitness when compared to Koza, or decreased the success rate 10 or more percentual points. Resource-Limited GP did not have a good performance, with most techniques achieving significantly worse fitness than Koza.

Summing up, Resource-Limited GP is excellent for the Regression problem, and the non-light Steady implementation can be safely used on the Artificial Ant problem and on the Parity problem with the upper limit. Dynamic Limits are excellent for the Artificial Ant problem, and the depth variants are good for the Parity and Multiplexer problems. The limited depth variants can be safely used on the Regression problem. All in all, some variants of the Dynamic Limits are a reliable option for all types of problems, while Resource-Limited GP should be used with a bit of caution, except in Regression problems like the one in this thesis, where its performance was outstanding. The next sections attempt to find what makes the Regression problem different from the others, something that may explain the reason for this isolated success of the variable population approach.

10.2.2 Problem Difficulty

The study of problem difficulty in GP is still a young field of research, and so far has proven to be a hard one [121, Chap. 3], [122]. One way to access the difficulty of a problem is by means of its fitness landscape, or search space. In a setting where the goal is to maximize fitness, a smooth and regular landscape with a single hill top is typical of an easy problem where the optimum is easily reached. On the contrary, difficult problems may be characterized by rugged landscapes with many peaks that can cause the search to get stuck on the local optima and never reach the highest peak, or by landscapes where most of the area is flat, with nothing to guide the search towards the points with higher fitness. Other difficulties may be the presence of deep valleys surrounding the highest peak, or the need to ascend an exceedingly long path to reach it [121, Chap. 3]. One of the factors that strongly determines the characteristics of the landscape is the neighborhood relationship. This relationship is determined by the genetic operators used to move around the search space: two individuals are neighbors if one can be obtained from the other by direct application of a genetic operator. Change this relationship, i.e., change the genetic operators, and the hills, valleys, and plateaus will probably suffer a major rearrangement. Likewise, change the representation of the individuals and the landscape will also be modified. Fitness landscapes are very hard to define when typical GP operators like crossover are used. Langdon and Poli have extensively studied the fitness landscapes of several classes of problems in GP [58]. Tavares has studied how the genetic representation influences the evolvability in different optimization problems [117].

Vanneschi has proposed two indicators of problem hardness for GP, both based on sampling the fitness landscape. *Fitness Distance Correlation* [121, Chap. 3], [122], first used for genetic algorithms, measures the extent to which the fitness of the individuals is correlated to their distance to an optimal solution. It is not a good predictive measure for all problems, as it requires the prior knowledge of an optimal solution. *Negative Slope Coefficient* [121, Chap. 3], [122,123] is based on the concept of *fitness cloud*, a scatterplot where the fitness of each individual is plotted against the fitness of its neighbors, thus providing and idea of how capable are the genetic operators of improving the fitness quality. Negative Slope Coefficient is an algebraic measure that quantifies the difficulty in performing this improvement. Although showing promising results, including real-life applications [124], the main drawback of this measure is that it does not return a normalized value that allows a comparison of difficulty among different problems. Both Fitness Distance Correlation and Negative Slope Coefficient are still experimental measures. Vanneschi validates them using a performance measure defined as being the proportion of runs for which the global optimum has been found in less than 500 generations over 100 runs [121, Chap. 3]. This is no different than the success rate, which seems to be, after all, the best measure of problem difficulty.

The first indicator of problem difficulty used in this thesis is indeed the success rate, measured as the percentage of runs where an optimal solution was found. The higher the rate, the lower the difficulty. The second indicator is convergence speed, measured as the proportion of resources that was left

Table 10.5: Indicators of problem difficulty. Numbers obtained by the Koza technique. The higher the number on the last column, the lower the difficulty of the problem.

Problem	**Success rate** (c)	**Resources saved** (r)	$c \times r$
Regression	40	0.92	36.8
Artificial Ant	10	0.49	4.9
Parity	0	–	0
Multiplexer	30	0.51	15.3

unused when convergence to an optimum happened. The average of the 30 runs was used. The more resources saved, the lower the difficulty. Table 10.5 shows both these indicators for all the problems considered, obtained by the baseline technique, Koza. An additional column contains the product of the two indicators. The higher this number, the lower the difficulty. According to the last column, Regression is clearly the easiest problem, followed by the Multiplexer, the Artificial Ant, and finally the Parity as the hardest problem.

10.2.3 Inviable Code

As stated in Section 5.1 (page 27), the amount of inviable code present in the individuals may be a distinctive feature among the several problems considered. Figure 10.1 shows the percentage of inviable code present in the individuals along 50 generations of evolution. These lines were obtained by 30 runs of the baseline technique, Koza.

Confirming early results [101, 102], the figure shows that most of the runs of the Regression problem have practically no inviable code, although the variability is high. The Multiplexer problem also keeps inviable code to low levels, followed by the Parity problem, and finally the Artificial Ant problem with a very high percentage of inviable code. The median percentages of inviable code measured on the last generation are 1.8% (Regression), 78.7% (Artificial Ant), 28.4% (Parity), and 11.5% (Multiplexer).

10.2.4 Diversity

Another feature that may be distinctive among the several problems considered is the diversity of the population. Figure 10.2 shows the percentage of distinct individuals present in the population along 50 generations of evolution. These lines were obtained by 30 runs of the baseline technique, Koza. This is the structural, or genotypic, diversity. A different kind of diversity, the phenotypic diversity, may be measured as the percentage of distinct fitness values present in the population. Figure 10.3 shows a boxplot of the phenotypic diversity measured on the last generation, for the several problems considered.

In both kinds of diversity, the differences between Regression and the remaining problems are striking. In terms of genotypic diversity, on the Regression problem a large range of behaviors can be observed. In some runs the

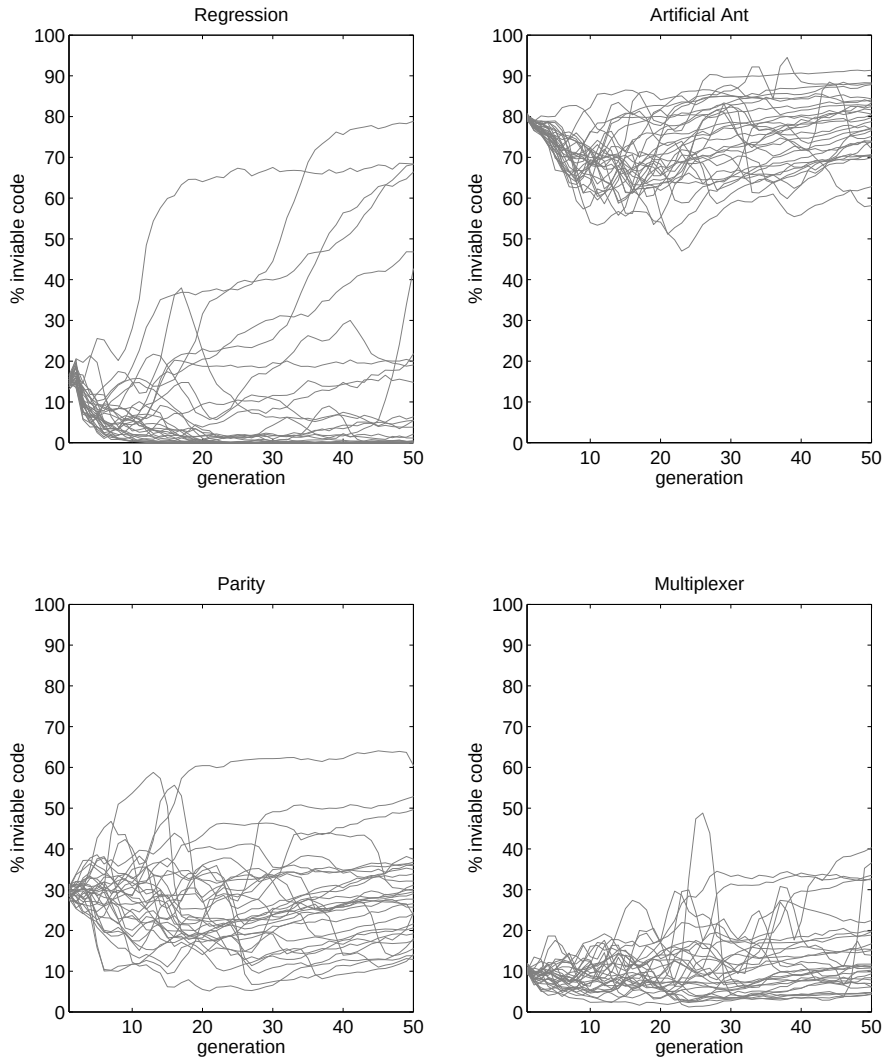

Figure 10.1: Percentage of inviable code along 50 generations of evolution. Numbers obtained by 30 runs of the Koza technique.

diversity quickly drops from the early stages of the evolution, while in others it increases and remains high until the end of the run. Other runs keep diversity to median values. On the last generation, the distribution of values is very wide. The other problems keep the values much more constrained, in particular the Parity and Multiplexer problems, where in all the runs the diversity quickly rises in the beginning of the evolution and remains close to the maximum value until the end. The median values of the genotypic diversity measured on the last generation are 38.0% (Regression), 96.3% (Artificial Ant), 97.3% (Parity), and 98.2% (Multiplexer).

In terms of the phenotypic diversity measured on the last generation, once again the Regression problem is the only one that displays a large range of values, with the remaining problems finishing with very constrained values. It could be expected that a higher genotypic diversity would naturally lead to a higher phenotypic diversity, but that is not observed, due to the characteristics of the problems (see Section 5.1, page 27). The Regression problem is where the genotypic diversity reached lower, and yet it is the problem with the highest phenotypic diversity (median of 33.7%), due to its wide range of possible fitness values. In the remaining problems, where the genotypic diversity remained close to the maximum value, the phenotypic diversity finished with median values as low as 4.8% (Artificial Ant), 1.1% (Parity), and 3.2% (Multiplexer). Note that the maximum allowed values would be 9.0% (Artificial Ant), 3.3% (Parity), and 6.5% (Multiplexer, hypothetical maximum), according to the distribution of possible fitness values described in Section 5.1.

10.2.5 Conclusions

The previous sections have described some of the characteristics that may explain why the Resource-Limited approach performed exceptionally well on the Regression problem, precisely the one problem where the score achieved by the Dynamic Limits was not so brilliant. The main observation is that the Regression problem is indeed different from the others. It is an easier problem than the rest, it is generally less prone to inviable code, and it is able to maintain a much higher phenotypic population diversity even in the presence of lower structural (genotypic) diversity. In general, the runs of the Regression problem exhibit a wider range of behaviors than the runs of the other problems.

Some of these characteristics may be related to each other. For example, the reduced amount of inviable code is usually related to a higher phenotypic diversity. Another related factor that plays an important role, maybe the main reason for the major success of Resource-Limited GP in the Regression problem, is the population size. A small population of 250 or 500 individuals would be enough to solve this problem [32, 118]. With 1000 individuals the structural diversity drops. The problem becomes so easy that Resource-Limited GP can steadily shrink the population until close to 10% of its original size (see Figure 8.2, left, page 80) and hardly ever increase the resource limit beyond its initial value, still achieving performance levels well above those of the Koza technique (see Table 10.1). Regression was in fact the only problem where the evolution of the population size showed this orderly behavior (compare with Figures 8.4, 8.6, and 8.8, left, pages 82, 84, and 86).

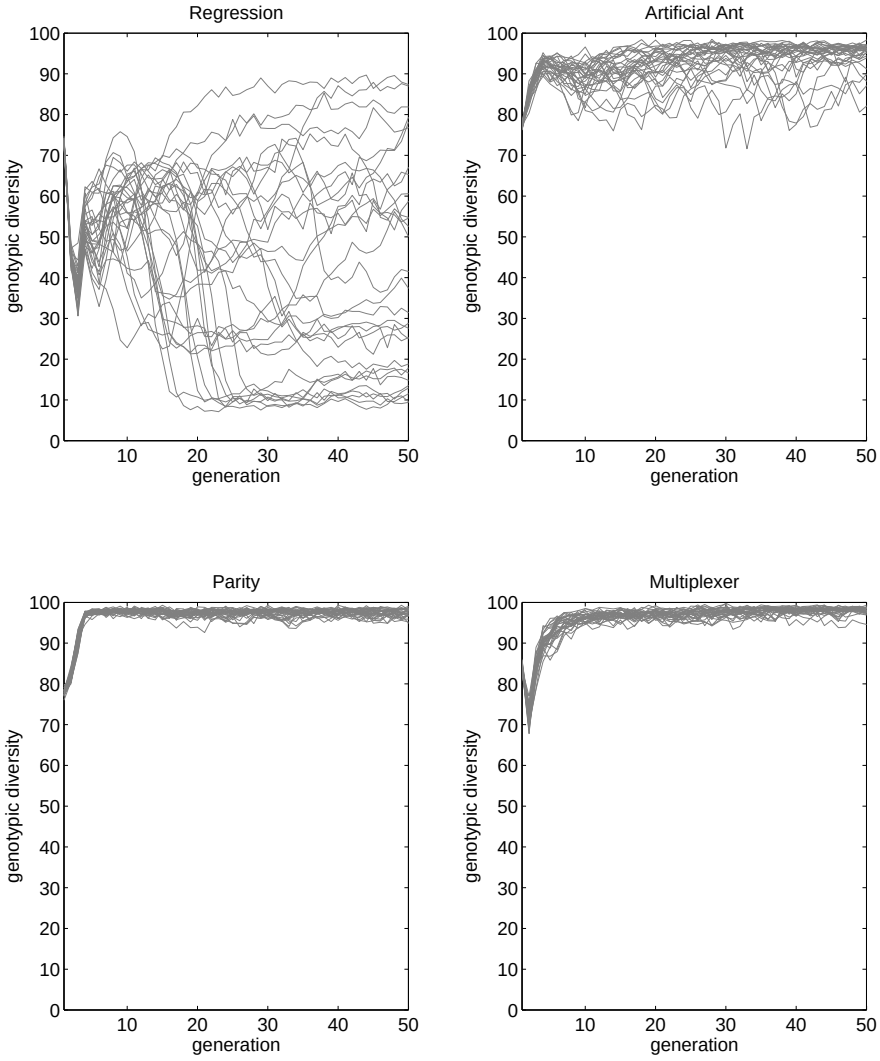

Figure 10.2: Genotypic diversity of the population along 50 generations of evolution. Numbers obtained by 30 runs of the Koza technique.

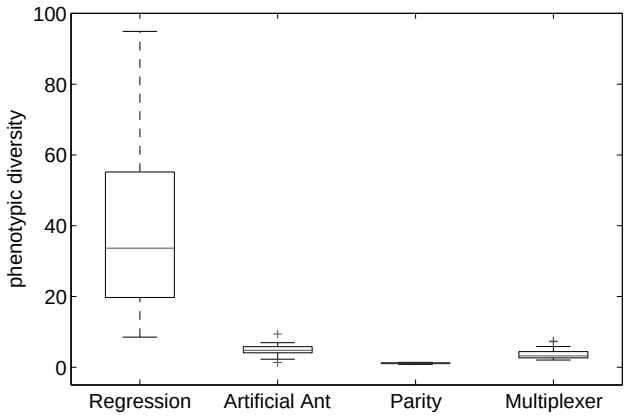

Figure 10.3: Phenotypic diversity of the population on the last generation. Numbers obtained by 30 runs of the Koza technique.

This may be the biggest strength of Resource-Limited GP: the ability to deal with too large a population, by quickly discarding the less promising individuals while maintaining the ones with a good fitness/size ratio. On the other hand, in problems where the population may be already too small to cope with the problem, the Resource-Limited approach may actually degrade the convergence ability of the search process. Some successful studies based on shrinking the population to reduce computational effort and counter the effects of bloat have in fact used larger initial populations. Luke et al. imploded populations from an initial size of 2048 individuals until they reached only a single individual, for the same four problems used in this thesis [64]. Fernandez et al. started with 5000 to 10000 individuals for the Artificial Ant and Parity problems [30, 31], stating that the larger the initial population, the greater the advantage of employing reductions [30]. Nevertheless, even with a modest population size, some variants of the Resource-Limited approach are able to achieve a good performance in most of the problems considered.

All things considered, both original approaches to bloat control presented in this thesis successfully performed the task they were designed to. Dynamic Limits have proven to be an excellent option for all the cases studied, capable of self adapting to the different characteristics inherent to such a diverse set of problems. Resource-Limited GP has achieved a performance ranging from modest to excellent, and is apparently more susceptible to the different characteristics of the problems. With additional work, both approaches can be further improved.

10.3 Final Considerations

There is a considerable amount of concern regarding the usage of depth or size limits in GP (see Section 2.2.4). The most traditional way of implementing the limits is to reject the invalid individuals and replace them with one of their

parents. Although this effectively prevents the individuals from growing too large, the replication of the parents may have undesirable effects. The larger parents are the ones that usually produce invalid offspring, so they tend to be replicated more often than the smaller parents. The population is filled with the largest individuals, and quickly rushes to the limit.

Alternative ways of implementing the depth or size limits are: 1) retry the genetic operator until a valid offspring is produced, either with the same parents or using different ones; 2) accept the invalid individuals but null their fitness so they will not be selected for reproduction in the next generation. In the light of the crossover bias theory (Section 2.1.6), retrying the genetic operator may not be advisable, since it provides another opportunity for the creation of more small unfit individuals. Accepting the large invalid offspring seems like a better measure against the undesirable crossover bias, since the large nullified individuals will never reproduce. However, in the presence of a dynamic and highly constrained limit that does not rise easily, parent replication may still provide advantages over these other options.

In the present implementation of Dynamic Limits, the traditional parent replication is indeed the action taken when the offspring violate the limit. But, unlike typical (static) limits, the initial dynamic limit is very low, as low as the maximum depth/size of the initial trees, and it will not be increased until a deeper/longer individual proves to be better than any other found so far. This highly constrains the search space, and for most problems it is known that the good solutions lie somewhere beyond this limit, not below. When a larger and better individual pushes the limit up, it means the process has entered a better searching ground - better solutions can be found within the new allowed depth/size. So, rushing the population towards this better, but still highly constrained, search space, may actually speed the convergence to better solutions. Parent replication along with a slowly increasing limit does not necessarily entail the drawbacks of using a high static limit.

In Resource-Limited GP, although many of the experiments presented here used a static limit at the individual level, the main effort against code growth is exerted at the population level. But in doing so, Resource-Limited GP may be creating unfavorable conditions that further stimulate bloat, according to the crossover bias theory (Section 2.1.6). During the process of allocating resources to the individuals in the queue, eventually the remaining available resources will become so scarce that only the smallest individuals can be further accepted. The allocation procedure continues to grant survival to these small unfit individuals until the resources are exhausted or the restrictions on population size (in terms of number of individuals) apply. In specific conditions, this may result in the acceptance of all the small individuals created in the last breeding cycle, as well as the ones from the previous generation, thus exacerbating the bias that causes bloat.

Chapter 11

Conclusion

This final chapter summarizes the main contributions of this thesis and the results obtained in the experiments, and points toward future directions of this work.

11.1 Summary

The search space of GP is virtually unlimited and programs tend to grow in size during the evolutionary process. Code growth is a healthy result of genetic operators in search of better solutions, but it also permits the appearance of pieces of redundant code that increase the size of programs without improving their fitness. Besides consuming precious time in an already computationally intensive process, redundant code may start growing rapidly, a phenomenon known as bloat. This is a serious problem in GP, often leading to the stagnation of the evolutionary process. Although many bloat control methods have been proposed so far, a definitive solution is yet to be found.

This work has introduced two new approaches to bloat control, called Dynamic Limits and Resource-Limited GP. Unlike many others available, these do not require specific genetic operators, modifications in fitness evaluation or different selection schemes, nor do they add any parameters to the search process. Dynamic Limits is inspired on the most traditional technique of imposing a fixed limit on the depth of the individuals allowed in the population, introduced by Koza in tree-based GP. It implements a dynamic limit that can be raised or lowered, depending on the best solution found so far, and can be applied either to the depth or size of the programs being evolved, thus making it suitable also for linear GP. Resource-Limited GP also uses a dynamic limit, but one that acts at a different level of the GP paradigm. A single limit is imposed on the total amount of tree nodes or code lines that the entire population can use. Tree nodes or code lines can be regarded as the resources that each individual needs to survive, and when they become insufficient for all, some individuals are discarded and the population is resized. The resource limit can be raised or lowered as in Dynamic Limits, depending on the mean population fitness. While the Dynamic Limits act at the individual level, imposing a condition

that each individual must verify in order to be accepted into the population, Resource-Limited GP acts at the population level, enforcing a global restriction that the population as a whole must respect, regardless of the particular individuals within.

Four different problems were used as a benchmark to study the efficiency of both Dynamic Limits and Resource-Limited GP. They represent a varied selection of problems in terms of bloat dynamics and response to different bloat control techniques: Symbolic Regression of the quartic polynomial, Artificial Ant on the Santa Fe food trail, 5-Bit Even Parity, and 11-Bit Boolean Multiplexer. A first comparison of results was performed among the Dynamic Limits techniques, followed by a comparison among the Resource-Limited GP techniques. The best techniques from both approaches were then involved in a third set of experiments, compared between each other and joined together to form a new hybrid technique. A final comparison was performed with some of the best state-of-the-art bloat control methods. The baseline for comparison was always the traditional Koza technique based on the static upper limit on tree depth. All the techniques were tested with and without using this fixed limit. The purpose was to check whether they can do without the static upper limit, or still benefit from joining, instead of just replacing, the baseline technique.

Among the Dynamic Limits, whenever statistically significant differences allowed for a ranking of the techniques, one of the depth-based variants (DynDepth) always scored number one in all problems, never performing worse, and sometimes significantly better, than the successful baseline Koza. The size-based variants generally performed worse than the rest. All the techniques within the Dynamic Limits were able to control bloat without relying on the static upper limit, a very desirable property.

Among Resource-Limited GP, one of the variants using the static upper limit (lResSteady) distinguished itself by achieving similar performance to Koza, and finding optima early and often when there were no significant differences in terms of best fitness of run. In many cases, this approach tended to collapse the population into only a few individuals (generally just one, small), regardless of the difficulty of the problem. For easy problems like Regression, this meant good performance, by finding an optimum while saving a large amount of resources. But, for difficult problems like Parity, the population collapsed before finding any optima, rendering all further search useless for lack of genetic diversity. So the performance of Resource-Limited GP was not even for all problems, with exceptional results on the Regression problem and only modest achievements on the remaining problems.

When comparing Dynamic Limits and Resource-Limited GP, it was the limits at the individual level that proved to have the main role in controlling tree growth. Between the DynDepth and lResSteady techniques there were considerable differences in average tree size. But when the static depth limit of lResSteady was replaced by a dynamic limit, the new Hybrid technique produced similar growth curves to DynDepth, not to lResSteady. Regardless of tree growth, the limits at the individual level also proved to be the main contributors to an improved best fitness. When using either a static or dynamic limit, replacing a fixed population with a dynamic population (e.g. Koza versus

lResSteady, DynDepth versus Hybrid) did not improve the performance; but when using either a fixed or dynamic population, replacing a static limit with a dynamic limit (e.g. Koza versus DynDepth, lResSteady versus Hybrid) did improve the results in many cases. The merit belonged to the dynamic limit, not to the variable size population.

Regarding the comparison with the state-of-the-art, the techniques achieving best fitness of run across all problems were the baseline Koza, the dynamic depth DynDepth, and the state-of-the-art Double, a technique based on the usage of a double tournament. Although Double was generally able to reach the optimum more easily, it has the disadvantage of being constrained by a selection method whose performance may be dependent on its settings. The settings chosen for these experiments were expected to yield good results, as they were the best previously found across the set of studied problems. Other state-of-the-art techniques based on dynamic populations had a very poor performance.

A deeper analysis was performed concerning the reasons why each approach achieved different results in the various problems considered, more precisely, why Resource-Limited GP performed exceptionally well on the Regression problem, precisely the one problem where the score achieved by the Dynamic Limits was not so brilliant. Two indicators of problem difficulty were considered, namely, the success rate and the success speed, and multiplied to provide a single difficulty value for each problem. The percentage of inviable code present in the population along 50 generations of evolution was also studied, as well as the structural (genotypic) diversity of the population, and the phenotypic diversity in the last generation. The main observation was that the Regression problem is indeed different from the others. It is an easier problem than the rest, it is generally less prone to inviable code, and it is able to maintain a much higher phenotypic diversity even in the presence of lower genotypic diversity. In general, the runs of the Regression problem exhibit a wider range of behaviors than the runs of the other problems. It was suggested that it is, however, the size of the population relative to the needs of the problem that mostly influences the level of success of the Resource-Limited GP.

Although Dynamic Limits was a more efficient bloat control method than Resource-Limited GP across the set of four problems studied, both approaches successfully performed the task they were designed to. A strong bloat control method should be able to deal with any type of problem, and be quite insensitive to the choice of parameters and even the combination of algorithmic elements like the evaluation, selection, and breeding procedures. Both original contributions in this thesis follow these guidelines. They are parameterless and flexible enough to adapt their behavior to the particularities of the situation at hand without draining the available computational resources. Both deserve to be further tested in order to identify their weaknesses, and improved in order to make them stronger.

11.2 Future Work

The generally poor fitness results achieved by the dynamic size techniques may be related to the difficulty in exploring around the current solution. Close to the

limit, any slight change in the number of nodes usually conflicts with the limit, and the new individuals are rejected. The freedom of exploration, that is present in all dynamic depth techniques, may be the crucial factor that the dynamic size lacks. If this supposition is true, a simple modification to the dynamic size techniques may easily solve the problem, for example by allowing individuals to always grow a little beyond the limit, regardless of their fitness. One way to do this is to allow the parents who do not exceed the limit to produce offspring of any size; parents who are already above the limit can only produce offspring smaller or equal to themselves. This is work in progress.

A comparison between the several implementation options presented when an offspring violates the limit (see Section 10.3) would be an interesting study. When using static limits, the best option appears to be the acceptance of the invalid offspring while giving it null fitness, but parent replication may hold some advantages when using dynamic limits.

The resource-limited approach could also be improved. When the individuals are placed in the queue for receiving the resources, they are ranked exclusively by fitness, regardless of size. If the individuals with the same fitness were ordered by size, the results achieved could be very different. A simple modification of a regular tournament to include this second ranking, called lexicographic parsimony pressure (Section 2.2.2), has already proven to produce very good results [67, 101, 102]. By implementing other types of queuing, ones that give a little more priority to size and less to fitness, it would be possible to increase the amount of parsimony pressure of the resource-limited techniques, and possibly improve their results in the problems where the performance was weaker. These are precisely the problems with lower phenotypic diversity, where the usage of different types of ranking is expected to cause more impact. This subject is currently under study.

In the light of the recent crossover bias theory, another aspect of the resource allocation procedure is also in need of improvement. In specific conditions, large numbers of small unfit individuals are accepted into the new generation, increasing the bias that causes bloat (see Section 10.3). The resource allocation procedure is a powerful tool in shaping the characteristics of the population. Carefully crafted, and possibly drawing inspiration from the operator equalisation bloat control method [26] (Section 2.2.4), it may provide the means to control the distribution of tree sizes inside the population, and totally suppress the bias that causes bloat.

Bibliography

[1] Altenberg, L.: The Evolution of Evolvability in Genetic Programming. In Kinnear Jr., K.E., editor, Advances in Genetic Programming. MIT Press (1994) C3:47–74

[2] Altenberg, L.: Emergent phenomena in genetic programming. In Sebald, A.V. and Fogel, L.J., editors, Proceedings of the 3rd Conference on Evolutionary Programming. World Scientific Publishing (1994) 233–241

[3] Andre, D. and Teller, A.: A study in program response and the negative effects of introns in genetic programming. In Koza, J.R. et al., editors, Proceedings of GP'96. MIT Press (1996) 28–31

[4] Angeline, P.J.: Genetic programming and emergent intelligence. In Kinnear Jr., K.E., editor, Advances in Genetic Programming. MIT Press (1994) C4:75–98

[5] Angeline, P.J.: Two self-adaptive crossover operators for genetic programming. In Angeline, P.J. and Kinnear Jr., K.E., editors, Advances in Genetic Programming 2. MIT Press (1996) C5:89–110

[6] Angeline, P.J.: A Historical Perspective on the Evolution of Executable Structures. Fundamenta Informaticae 35(1-4): 179–195 (1998)

[7] Angeline, P.J. and Pollack, J.B.: Coevolving high-level representations. In Langton, C.G., editor, Proceedings of Artificial Life III. Addison-Wesley (1994) 55–71

[8] Arakawaa, M., Hasegawab, K., Funatsu, K.: QSAR study of anti-HIV HEPT analogues based on multi-objective genetic programming and counter-propagation neural network. Chemometrics and Intelligent Laboratory Systems 83(2): 91–98 (2006)

[9] Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic programming – an introduction. dpunkt.verlag and Morgan Kaufmann (1998)

[10] Banzhaf, W. and Langdon, W.B.: Some considerations on the reason for bloat. Genetic Programming and Evolvable Machines 3(1): 81-91 (2002)

[11] Banzhaf, W., Francone, F.D., Nordin, P.: Some Emergent Properties of Variable Size EAs. Position paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97 (1997)

[12] Bleuler, S., Brack, M., Thiele, L., Zitzler, E.: Multiobjective Genetic Programming: Reducing Bloat Using SPEA2. In Proceedings of CEC-2001. IEEE Press (2001) 536–543

[13] Blickle, T.: Theory of evolutionary algorithms and applications to system design. PhD thesis, Swiss Federal Institute of Technology, Computer Engineering and Networks Laboratory (1996)

[14] Blickle, T.: Evolving Compact Solutions in Genetic Programming: A Case Study. In Voigt, H.-M. et al., editors, Proceedings of Parallel Problem Solving From Nature IV. Springer (1996) 564–573

[15] Blickle, T. and Thiele, L.: Genetic Programming and Redundancy. In Hopf, J., editor, Genetic Algorithms within the Framework of Evolutionary Computation. Max-Planck-Institut für Informatik (1994) 33–38

[16] Brameier, M. and Banzhaf, W.: A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining. IEEE Transactions on Evolutionary Computation 5(1): 17–26 (2001)

[17] Brameier, M. and Banzhaf, W.: Neutral variations cause bloat in linear GP. In Ryan, C. et al., editors, Proceedings of EuroGP-2003. Springer (2003) 286–296

[18] Brazdilova, S.L.: Autofocusing in Automated Microscopy. RNDr thesis, Faculty of Informatics, Masaryk University, Czech Republic (2006)

[19] Cuendet, J.: Populations dynamiques en programmation génétique. MSc thesis, Université de Lausanne, Université de Genève (2004)

[20] Da Costa, L.E. and Landry, J.A.: Relaxed genetic programming. In Keijzer, M. et al., editors, Proceedings of GECCO-2006. ACM Press (2006) 937–938

[21] De Jong, E.D., Watson, R.A., Pollack, J.B.: Reducing Bloat and Promoting Diversity using Multi-Objective Methods. In Spector, L. et al., editors, Proceedings of GECCO-2001. Morgan Kaufmann (2001) 11–18

[22] De Jong, E.D. and Pollack, J.B.: Multi-objective methods for tree size control. Genetic Programming and Evolvable Machines, 4(3): 211–233 (2003)

[23] D'haeseleer, P.: Context preserving crossover in genetic programming. In Proceedings of the 1994 IEEE World Congress on Computational Intelligence. IEEE Press (1994) 256–261

[24] Dignum, S. and Poli, R.: Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In Thierens, D. et al., editors, Proceedings of GECCO-2007. ACM Press (2007) 1588–1595

[25] Dignum, S. and Poli, R.: Crossover, sampling, bloat and the harmful effects of size limits. In O'Neill, M. et al., editors, Proceedings of EuroGP-2008. Springer (2008) 158–169

[26] Dignum, S. and Poli, R.: Operator equalisation and bloat free GP. In O'Neill, M. et al., editors, Proceedings of EuroGP-2008. Springer (2008) 110–121

[27] Eiben, A.E. and Smith, J.E.: Introduction to Evolutionary Computing. Springer (2003)

[28] Ekart, A.: Shorter Fitness Preserving Genetic Programs. In Fonlupt, C. et al., editors, Proceedings of AE-1999. Springer (2000) 73–83

[29] Ekart, A. and Németh, S.Z.: Selection based on the pareto nondomination criterion for controlling code growth in genetic programming. Genetic Programming and Evolvable Machines 2(1): 61–73 (2001)

[30] Fernandez, F., Vanneschi, L., Tomassini, M.: The effect of plagues in genetic programming: A study of variable-size populations. In Ryan, C. et al., editors, Proceedings of EuroGP-2003. Springer (2003) 317–326

[31] Fernandez, F., Tomassini, M., Vanneschi, L.: Saving computational effort in genetic programming by means of plagues. In Sarker, R. et al., editors, Proceedings of CEC-2003. IEEE Press (2003) 2042–2049

[32] Fernandez, F., Tomassini, M., Vanneschi, L.: An empirical study of multipopulation genetic programming. Genetic Programming and Evolvable Machines 4(1): 21–51 (2003)

[33] Gathercole, C. and Ross, P.: An adverse interaction between crossover and restricted tree depth in genetic programming. In Koza, J.R. et al., editors, Proceedings of GP'96. MIT Press (1996) 291–296

[34] Gelly, S., Teytaud, O., Bredeche, N., Schoenauer, M.: A statistical learning theory approach of bloat. In Beyer, H.-G. et al., editors, Proceedings of GECCO-2005. ACM Press (2005) 1783–1784

[35] Gelly, S., Teytaud, O., Bredeche, N., Schoenauer, M.: Universal Consistency and Bloat in GP. Revue d'Intelligence Artificielle 20(6): 805–827 (2006)

[36] Güroglu, S.: An Evolutionary Methodology for Conceptual Design. PhD thesis. Graduate School of Natural and Applied Sciences of Middle East Technical University (2005)

[37] Gustafson, S., Ekart, A., Burke, E., Kendall, G.: Problem difficulty and code growth in genetic programming. Genetic Programming and Evolvable Machines 5(3): 271–290 (2004)

[38] Haynes, T.: Collective Adaptation: The Exchange of Coding Segments. Evolutionary Computation 6(4): 311–338 (1998)

[39] Hooper, D. and Flann, N.S.: Improving the Accuracy and Robustness of Genetic Programming through Expression Simplification. In Koza, J.R. et al., editors, Proceedings of GP'96. MIT Press (1996) 428–428

[40] Iba, H., de Garis, H., Sato, T.: Genetic Programming Using a Minimum Description Length Principle. Kinnear Jr., K.E., editor, Advances in Genetic Programming. MIT Press (1994) 265–284

[41] Iba, H. and Terao, M.: Controlling Effective Introns for Multi-Agent Learning by Genetic Programming. In Whitley, D. et al., editors, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000). Morgan Kaufmann (2000) 419–426

[42] Igel, C. and Chellapilla, K.: Investigating the Influence of Depth and Degree of Genotypic Change on Fitness in Genetic Programming. In Banzhaf, W. et al., editors, Proceedings of GECCO-1999. Morgan Kaufmann (1999) 1061–1068

[43] Kennedy, C.J. and Giraud-Carrier, C.: A depth controlling strategy for strongly typed evolutionary programming. In Banzhaf, W. et al., editors, Proceedings of GECCO-1999. Morgan Kaufmann (1999) 879–885

[44] Kinnear Jr., K.E.: Generality and Difficulty in Genetic Programming: Evolving a Sort. In Forrest, S., editor, Proceedings of ICGA'93. Morgan Kaufmann (1993) 287–294

[45] Koza, J.R.: Genetic programming – on the programming of computers by means of natural selection. MIT Press (1992)

[46] Koza, J.R.: Genetic Programming II – Automatic discovery of reusable programs. MIT Press (1994)

[47] Koza, J.R., Bennett III, F.H., Andre, D., Keane, M.A.: Genetic Programming III – Darwinian Invention and Problem Solving. Morgan Kaufmann (1999)

[48] Langdon, W.B.: Genetic Programming + Data Structures = Automatic Programming! Kluwer Academic Publishers (1998)

[49] Langdon, W.B.: The Evolution of Size in Variable Length Representations. In Proceedings of the 1998 IEEE International Conference on Evolutionary Computation. IEEE Press (1998) 633–638

[50] Langdon, W.B.: Genetic Programming Bloat with Dynamic Fitness. In Banzhaf, W. et al., editors, Proceedings of EuroGP-1998. Springer (1998) 96–112

[51] Langdon, W.B.: Size fair and homologous tree genetic programming crossovers. In Banzhaf, W. et al., editors, Proceedings of GECCO-1999. Morgan Kaufmann (1999) 1092–1097

[52] Langdon, W.B.: Size fair and homologous tree genetic programming crossovers. Genetic Programming and Evolvable Machines, 1(1/2): 95–119 (2000)

[53] Langdon, W.B.: Quadratic Bloat in Genetic Programming. In Whitley, D. et al., editors, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000). Morgan Kaufmann (2000) 451–458

[54] Langdon, W.B. and Nordin, J.P.: Seeding GP Populations. In Poli, R. et al., editors, Proceedings of EuroGP-2000. Springer (2000) 304–315

[55] Langdon, W.B., Poli, R.: Fitness Causes Bloat. Technical Report CSRP-97-09, School of Computer Science, University of Birmingham (1997)

[56] Langdon, W.B. and Poli, R.: An analysis of the MAX problem in genetic programming. In Koza, J.R. et al., editors, Proceedings of GP'97. Morgan Kaufman (1997) 222–230

[57] Langdon, W.B., Poli, R.: Fitness Causes Bloat: Mutation. In Banzhaf, W. et al., editors, Proceedings of EuroGP'98. Springer (1998) 37–48

[58] Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer (2002)

[59] Langdon, W.B., Soule, T., Poli, R., Foster, J.A.: The evolution of size and shape. In Spector, L. et al., editors, Advances in Genetic Programming 3. MIT Press (1999) C8:163–190

[60] Langdon, W.B. and Banzhaf, W.: Genetic programming bloat without semantics. In Schoenauer, M. et al., editors, Proceedings of PPSN-2000. Springer (2000) 201–210

[61] Luerssen, M.H.: Graph Grammar Encoding and Evolution of Automata Networks. In Proceedings of the 28th Australasian Computer Science Conference (ACSC-2005), ACS CRPIT Series, vol 38 (2005) 229–238

[62] Luke, S.: Code Growth is Not Caused by Introns. In Late Breaking Papers at GECCO-2000 (2000) 228–235

[63] Luke, S.: Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat. PhD thesis, Department of Computer Science, University of Maryland (2000)

[64] Luke, S., Balan, G.C., Panait, L.: Population implosion in genetic programming. In Cantú-Paz, E. et al., editors, Proceedings of GECCO-2003. Springer (2003) 1729–1739

[65] Luke, S.: Modification Point Depth and Genome Growth in Genetic Programming. Evolutionary Computation 11(1): 67–106 (2003)

[66] Luke, S. and Panait, L.: Fighting Bloat With Nonparametric Parsimony Pressure. In Guervos, J.M. et al., editors, Proceedings of PPSN-2002. Springer (2002) 411–420

[67] Luke, S. and Panait, L.: Lexicographic parsimony pressure. In Langdon, W.B. et al., editors, Proceedings of GECCO-2002. Morgan Kaufmann (2002) 829–836

[68] Luke, S. and Panait, L.: A comparison of bloat control methods for genetic programming. Evolutionary Computation 14(3): 309–344 (2006)

[69] Majid, A., Khan, A., Mirza, A.M.: Improving performance of nearest neighborhood classifier using genetic programming. In Proceedings of the 2004 IEEE International Conference on Machine Learning and Applications. IEEE Press (2004) 469–476

[70] Martin, P. and Poli, R.: Crossover operators for a hardware implementation of genetic programming using FPGAs and Handel-C. In Langdon, W.B. et al., editors, Proceedings of GECCO-2002. Morgan Kaufmann (2002) 845–852

[71] McPhee, N.F. and Miller, J.D.: Accurate replication in Genetic Programming. In Eshelman, L., editor, Proceedings of ICGA'95. Morgan Kaufmann (1995) 303–309

[72] McPhee, N.F., Jarvis, A., Crane, E.F.: On the strength of size limits in linear genetic programming. In Deb, K. et al., editors, Proceedings of GECCO-2004. Springer (2004) 593–604

[73] McPhee, N.F. and Poli, R.: A schema theory analysis of the evolution of size in genetic programming with linear representations. In Miller, J. et al., editors, Proceedings of EuroGP-2001. Springer (2001) 108–125

[74] Miller, J.: What Bloat? Cartesian Genetic Programming on Boolean Problems. In Late Breaking Papers at GECCO-2001 (2001) 295–302

[75] Montana, D.J. and Davis, L.: Training feedforward neural networks using genetic algorithms. In Proceedings of the International Joint Conference on Artificial Intelligence (1989) 762-767

[76] Nordin, P. and Banzhaf, W.: Complexity compression and evolution. In Eshelman, L., editor, Proceedings of ICGA'95. Morgan Kaufmann (1995) 318–325

[77] Nordin, P., Banzhaf, W., Francone, F.D.: Efficient Evolution of Machine Code for CISC Architectures using Instruction Blocks and Homologous Crossover. Spector, L. et al., editors, Advances in Genetic Programming 3. MIT Press (1999) 275–299

[78] Nordin, P., Francone, F., Banzhaf, W.: Explicitly Defined Introns and Destructive Crossover in Genetic Programming. In Rosca, J.P., editor, Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications (1995) 6–22

[79] Nordin, P., Francone, F., Banzhaf, W.: Explicitly Defined Introns and Destructive Crossover in Genetic Programming. In Angeline, P.J. and Kinnear Jr., K.E., editors, Advances in Genetic Programming 2. MIT Press (1996) 111–134

[80] O'Reilly, U.-M. and Oppacher, F.: Hybridized Crossover-Based Search Techniques for Program Discovery. In Proceedings of the 1995 World Conference on Evolutionary Computation. IEEE Press (1995) 573–578

[81] Page , J., Poli, R., Langdon, W.B.: Smooth Uniform Crossover with Smooth Point Mutation in Genetic Programming: A Preliminary Study. In Poli, R. et al., editors, Proceedings of EuroGP-1999. Springer (1999) 39–49

[82] Panait, L. and Luke, S.: Alternative Bloat Control Methods. In Deb, K. et al., editors, Proceedings of GECCO-2004. Springer (2004) 630-641

[83] Platel, M.D., Clergue, M., Collard, P.: Maximum Homologous Crossover for Linear Genetic Programming. In Ryan, C. et al., editors, Proceedings of EuroGP-2003. Springer (2003) 194–203

[84] Poli, R.: General schema theory for genetic programming with subtree-swapping crossover. In Miller, J. et al., editors, Proceedings of EuroGP-2001. Springer (2001) 143–159

[85] Poli, R.: A simple but theoretically-motivated method to control bloat in genetic programming. In Ryan, C. et al., editors, Proceedings of EuroGP-2003. Springer (2003) 200–210

[86] Poli, R. and Langdon, W.B.: Genetic Programming with One-Point Crossover. In Chawdhry, P.K. et al., editors, Proceedings of the Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing. Springer London (1997) 180–189

[87] Poli, R. and Langdon, W.B.: A New Schema Theory for Genetic Programming with One-point Crossover and Point Mutation. In Koza, J. et al., editors, Proceedings of GP'97. Morgan Kaufmann (1997) 278–285

[88] Poli, R. and Langdon, W.B.: On the Search Properties of Different Crossover Operators in Genetic Programming. In Koza, J. et al., editors, Proceedings of GP'98. Morgan Kaufmann (1998) 293–301

[89] Poli, R., Langdon, W.B., Dignum, S.: On the limiting distribution of program sizes in tree-based genetic programming. In Ebner, M. et al., editors, Proceedings of EuroGP-2007. Springer (2007) 193–204

[90] Poli, R., McPhee, N.F., Vanneschi, L.: The impact of population size on code growth in GP: analysis and empirical validation. In Proceedings of GECCO-2008. ACM Press (2008), to appear

[91] Poli, R., McPhee, N.F., Vanneschi, L.: Elitism reduces bloat in GP: a theoretical analysis with empirical corroboration. In Proceedings of GECCO-2008. ACM Press (2008), to appear

[92] Poli, R. and McPhee, N.F.: Parsimony pressure made easy. In Proceedings of GECCO-2008. ACM Press (2008), to appear

[93] Ratle, A. and Sebag, M.: Avoiding the bloat with Probabilistic Grammar-guided Genetic Programming. In Collet, P. et al., editors, Proceedings of the Artificial Evolution 5th International Conference (Evolution Artificielle, EA-2001). Springer (2001) 255–266

[94] Rochat, D.: Programmation Génétique Parallèle: Opérateurs Génétiques Variés et Populations Dynamiques. MSc thesis, Université de Lausanne, Université de Genève (2004)

[95] Rochat, D., Tomassini, M., Vanneschi, L.: Dynamic Size Populations in Distributed Genetic Pogramming. In Keijzer, M. et al., editors, Proceedings of EuroGP-2005. Springer (2005) 50–61

[96] Rosca, J.P.: Generality versus Size in Genetic Programming. In Koza, J.R. et al., editors, Proceedings of GP'96. MIT Press (1996) 381–387

[97] Rosca, J.P.: Analysis of Complexity Drift in Genetic Programming. In Koza, J.R. et al., editors, Proceedings of GP'97. Morgan Kaufmann (1997) 286–294

[98] Rosca, J.P. and Ballard, D.H.: Complexity Drift in Evolutionary Computation with Tree Representations. Technical Report NRL96.5, Computer Science Department, The University of Rochester (1996)

[99] Rosca, J.P. and Ballard, D.H.:: Discovery of Subroutines in Genetic Programming. In Angeline, P.J. and Kinnear Jr., K.E., editors, Advances in Genetic Programming 2. MIT Press (1996) 177–202

[100] Silva, S. and Almeida, J.: GPLAB – A Genetic Programming Toolbox for MATLAB. In Gregersen, L., editor, Proceedings of the Nordic MATLAB Conference (2003) 273–278

[101] Silva, S. and Almeida, J.: Dynamic maximum tree depth - a simple technique for avoiding bloat in tree-based GP. In Cantú-Paz, E. et al., editors, Proceedings of GECCO-2003. Springer (2003) 1776–1787

[102] Silva, S. and Costa, E.: Dynamic limits for bloat control - variations on size and depth. In Deb, K. et al., editors, Proceedings of GECCO-2004. Springer (2004) 666–677

[103] Silva, S., Silva, P.J.N., Costa, E.: Resource-Limited Genetic Programming: Replacing Tree Depth Limits. In Ribeiro, B. et al., editors, Proceedings of ICANNGA-2005. Springer (2005) 243–246

[104] Silva, S. and Costa, E.: Resource-Limited Genetic Programming: The Dynamic Approach. In Beyer, H.-G. et al., editors, Proceedings og GECCO-2005. ACM Press (2005) 1673–1680

[105] Silva, S. and Costa, E.: Comparing tree depth-limits and resource-limited GP. In Corne, D. et al., editors, Proceedings of CEC-2005. IEEE Press (2005) 920–927

[106] Smith, P.W.H. and Harries, K.: Code Growth, Explicitly Defined Introns, and Alternative Selection Schemes. Evolutionary Computation 6(4): 339–360 (1998)

[107] Soule, T. and Foster, J.A.: Removal Bias: a New Cause of Code Growth in Tree Based Evolutionary Programming. In Proceedings of the 1998 IEEE International Conference on Evolutionary Computation. IEEE Press (1998) 781–786

[108] Soule, T.: Code growth in genetic programming. PhD thesis, College of Graduate Studies, University of Idaho (1998)

[109] Soule, T. and Foster, J.: Code size and depth flows in genetic programming. In Koza, J. et al., editors, Proceedings of GP'97. Morgan Kaufmann (1997) 313–320

[110] Soule, T. and Foster, J.A.: Effects of Code Growth and Parsimony Pressure on Populations in Genetic Programming. Evolutionary Computation 6(4): 293–309 (1998)

[111] Soule, T., Foster, J., Dickinson, J.: Code growth in genetic programming. In Koza, J. et al., editors, Proceedings of GP'96. MIT Press (1996) 215–223

[112] Soule, T. and Heckendorn, R.B.: An analysis of the causes of code growth in genetic programming. Genetic Programming and Evolvable Machines 3(1): 283–309 (2002)

[113] Spector, L.: Simultaneous evolution of programs and their control structures. In Angeline, P.J. and Kinnear Jr., K.E., editors, Advances in Genetic Programming 2. MIT Press (1996) 137–154

[114] Stevens, J., Heckendorn, R.B., Soule, T.: Exploiting disruption aversion to control code bloat. In Beyer, H.-G. et al., editors, Proceedings of GECCO-2005. ACM Press (2005) 1605–1612

[115] Streeter, M.J.: The Root Causes of Code Growth in Genetic Programming. In Ryan, C. et al., editors, Proceedings of EuroGP-2003. Springer (2003) 443–454

[116] Tackett, W.A.: Recombination, Selection, and the Genetic Construction of Genetic Programs. PhD thesis, Department of Electrical Engineering Systems, University of Southern California (1994)

[117] Tavares, J.: Evolvability in Optimization Problems: The Role of Representations and Heuristics. PhD thesis, Department of Informatics Engineering, University of Coimbra (2007)

[118] Tomassini, M., Vanneschi, L., Cuendet, J., Fernandez, F.: A New Technique for Dynamic Size Populations in Genetic Programming. In Proceedings of CEC-2004. IEEE Press (2004) 486–493

[119] Trujillo, L. and Olague, G.: Synthesis of interest point detectors through genetic programming. In Keijzer, M. et al., editors, Proceedings of GECCO-2006. ACM Press (2006) 887–894

[120] Van Belle, T. and Ackley, D.H.: Uniform subtree mutation. In Foster, J.A. et al., editors, Proceedings of EuroGP-2002. Springer (2002) 152–161

[121] Vanneschi, L.: Theory and Practice for Efficient Genetic Programming. PhD thesis, Faculty of Sciences, University of Lausanne (2004)

[122] Vanneschi, L., Tomassini, M., Collard, P., Clergue, M.: A Survey of Problem Difficulty in Genetic Programming. In Bandini, S. and Manzoni, S., editors, Proceedings of the 9th Congress of the Italian Association for Artificial Intelligence. Springer (2005) 66–77

[123] Vanneschi, L., Tomassini, M., Collard, P., Vérel, S.: Negative slope coefficient. A measure to characterize genetic programming. In Collet, P. et al., editors, Proceedings of the 9th European Conference on Genetic Programming. Springer (2006) 178–189

[124] Vanneschi, L.: Investigating Problem Hardness of Real Life Applications. In Riolo, R.L. et al., editors, Genetic Programming Theory and Practice V. Springer (2007) 107–125

[125] Wagner, N., Michalewicz, Z.: Genetic programming with efficient population control for financial time series prediction. In Late Breaking Papers at GECCO-2001 (2001) 458–462

[126] Zhang, B.-T. and Mühlenbein, H.: Balancing Accuracy and Parsimony in Genetic Programming. Evolutionary Computation 3(1): 17–38 (1995)

[127] Zhang, B.-T.: A Taxonomy of Control Schemes for Genetic Code Growth. Position paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97 (1997)

[128] Zhang, B.-T.: Bayesian methods for efficient genetic programming. Genetic Programming and Evolving Machines 1(1): 217–242 (2000)

Index