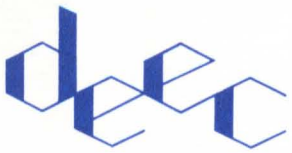

DEPARTAMENTO DE

ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

ALGORITMOS GENÉTICOS NA OPTIMIZAÇÃO DE PERGUNTAS RELACIONAIS

João Vasco Tavares da Mota Ranito

FACULDADE DE ENGENHARIA

UNIVERSIDADE DO PORTO

Rua dos Bragas, 4099 Porto Codex – PORTUGAL

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Departamento de Engenharia Electrotécnica e de Computadores

ALGORITMOS GENÉTICOS NA OPTIMIZAÇÃO DE PERGUNTAS RELACIONAIS

João Vasco Tavares da Mota Ranito

Licenciado em Engenharia Electrotécnica
pela Faculdade de Engenharia da
Universidade do Porto

Faculdade de Engenharia
BIBLIOTECA
N.º
CDU

Dissertação submetida para satisfação parcial dos
requisitos do programa do curso de mestrado
em
Engenharia Electrotécnica e de Computadores
(Área de especialização de Informática Industrial)

PORTO, Abril de 1993

UNIVERSIDADE DO PORTO
Faculdade de Engenharia
BIBLIOTECA
N.º 67122
CDU 004(043)
Data 13 / 5 / 2003

Dissertação realizada sob a supervisão de

Doutor João Bernardo Falcão e Cunha,

Professor Auxiliar do Departamento de Engenharia Mecânica e Gestão Industrial da Faculdade de Engenharia da Universidade do Porto

Resumo

Nos últimos tempos, tem ganho alguma notoriedade uma técnica de optimização combinatória inspirada na evolução natural. Esta técnica, denominada "Algoritmos Genéticos", tem sido usada com sucesso na resolução de problemas em diversas áreas, que vão desde a Investigação Operacional até à Inteligência Artificial. A abordagem genética fornece um meio simples de atacar problemas complexos, uma vez que é muito genérica e pouco dependente da estrutura do problema.

O trabalho descrito nesta tese consiste na aplicação dos Algoritmos Genéticos ao problema da Optimização Relacional, ou seja, o problema de encontrar um bom algoritmo para implementação de uma pergunta relacional posta a um SGBD, por exemplo em SQL. Foi construído um protótipo do optimizador relacional, que irá ser integrado num interpretador de SQL, desenvolvido no âmbito do projecto SAGA do INESC.

Abstract

In the last few years, we have been watching the rise of a combinatorial optimisation technique, inspired by natural selection. This technique, "Genetic Algorithms", has been successfully used to solve problems from several areas, from Operations Research to Artificial Intelligence. Because is mainly independent from the problem structure, the genetic approach provides a simple way of atacking complex problems.

This thesis describes the application of Genetic Algorithms to the Relational Optimisation problem. This problem amounts to finding a good algorithm to implement a relational query to a DBMS, for instance in SQL. A prototype has been built and is being integrated in a SQL interpreter, developed in the scope of the SAGA project from INESC.

Palavras chave

Algoritmos Genéticos

Optimização Relacional

SQL

Manipulação Simbólica

Reescrita automática

Keywords

Genetic Algorithms

Relational Optimisation

SQL

Symbolic Manipulation

Automatic rewriting

Agradecimentos

Gostaria de agradecer a uma série de pessoas e instituições que contribuíram muito para que esta tese pudesse estar pronta a tempo. Em primeiro lugar, queria agradecer aos Profs. Falcão e Cunha e Raul Vidal pela orientação científica, condições de trabalho e paciência. Sem qualquer uma destas coisas, seria difícil fazer uma tese. Também agradeço ao INESC, pelos meios que pôs à minha disposição e à JNICT, que suportou financeiramente este trabalho com a bolsa BM/153/90-IA.

Em segundo lugar, queria agradecer a todos os meus colegas de projecto pelo bom ambiente proporcionado, pelas sugestões, discussões frutuosas e piadas desanuviantes. Particularmente, gostaria de agradecer ao José Bonnet, ao Jorge Simões, ao Ademar Aguiar e ao Mário Santos (meus colegas de mestrado) pelo apoio e documentação, ao João Pascoal pelas ajudas em Teoria das Probabilidades, ao João Neta pelas partes "não genéticas" do optimizador e ao João Neto pelo interface para o GÉNIO e por todas as muitas boas ideias que me deu.

Ao Valter Pinho, ao José Moreira, ao Jorge Serro, ao Luís Neves, e ao João Correia Lopes (por via electrónica) e a todas as pessoas do Centro de Sistemas Computacionais do INESC, agradeço a disponibilidade para trocar impressões, enquanto que à Ana Paula agradeço a ajuda na "burocracia".

Por fim, quero ainda agradecer à minha família, em geral, e à minha mulher, Alexandra, em particular, pela paciência que demonstraram para com a minha atitude alterada pelo trabalho e pelo apoio e compreensão que sempre me deram.

Prefácio

Na realidade, nem sei o que me passou pela cabeça para iniciar uma tese em Algoritmos Genéticos, um campo que ainda nem sequer conseguiu estabelecer um nicho próprio no panorama das Ciências da Computação. Mas a verdade é que a ideia é fascinante: será que é mesmo possível fazer evoluir programas de computador como se fossem seres vivos? E ainda para mais de um modo tão simples, apenas com selecção, cruzamento e mutação? E basta lidar com cadeias de bits? Parece bom de mais para ser verdade!

Os Algoritmos Genéticos têm veleidades de ser uma técnica genérica de resolução de problemas, trazendo uma lufada de ar fresco ao ambiente um pouco asfixiante da hiper-especialização que invadiu as técnicas de Inteligência Artificial. Claro que esta generalização traz desvantagens: não se pode ser 100% eficiente em todos os domínios. Mas traz-nos também um consolo: se houver um problema que não se resolve com técnicas tradicionais, talvez uma abordagem genética possa funcionar. E há também o factor novidade. Os Algoritmos Genéticos são recentes e atacam frontalmente alguns dogmas. Isso faz com que tenham associada uma aura de novidade e iconoclastia que não pode deixar de atrair qualquer pessoa que tenha um mínimo de imaginação e de neofilia.

Mas aquilo que mais me atraiu nos Algoritmos Genéticos é o facto de representarem o fecho de um ciclo: provavelmente, o Homem é o resultado de alguns milhares de milhões de anos de evolução genética e propõe-se agora aplicar a mesma receita a programas de computador, uma criação sua. Este "Ciclo Estranho" (usando a nomenclatura do livro "Göedel, Escher, Bach", a obra-prima de Hofstadter) pode vir a ter como resultado uma geração de programas com uma arquitectura diferente da que nos habituámos desde que os computadores apareceram.

A mera possibilidade de participar desta mudança, nem que seja apenas com uma tese de mestrado, é, no mínimo, empolgante.

Índice

1. Introdução 1
   1.1. Objectivo, motivação e enquadramento 1
   1.2. Algoritmos Genéticos 2
   1.3. Optimização relacional 6
   1.4. Optimização relacional usando algoritmos genéticos 7
   1.5. Estrutura da tese 7

2. Algoritmos Genéticos 9
   2.1. Propriedades gerais de um AG 9
   2.2. Posicionamento face à optimização clássica 15

3. Perspectiva histórica 20

4. Fundamento matemático dos Algoritmos Genéticos 23
   4.1. Motivação 23
   4.2. O Teorema Fundamental 24
   4.3. Problemas enganadores 30

5. Codificação e Paramerização 33
   5.1. Codificação 33
   5.2. Parametrização 35
   5.3. Fim da evolução: a espécie perfeita? 39

6. Selecção e cruzamento revisitados 41
   6.1. Selecção 41
   6.2. Cruzamento 45

7. Utilização avançada 48
   7.1. Recessão e Domínio 48
   7.2. Partilha 49
   7.3. Inversão 51
   7.4. Programação Genética 53
   7.5. Permutações 55

8. Algumas experiências usando AG 58
   8.1. Job-Shop 58
   8.2. Problema do Caixeiro Viajante 62
   8.3. Determinação de Arquitecturas de Redes Neuronais 62

9. Optimização relacional 66
   9.1. Linguagens relacionais 66
   9.2. Álgebra relacional 67
   9.3. Métodos de acesso 70

10. Optimização relacional clássica 74
   10.1. Fases da optimização clássica 74
   10.2. Optimização Algébrica 75
   10.3. Optimização física 79

10.4. Restrições 83

11. Contexto do optimizador: o interpretador de SQL 85
11.1. Arquitectura do interpretador 85

12. O optimizador relacional genético 99
12.1. Introdução 99
12.2. Codificação 99
12.3. População inicial e operadores genéticos 106
12.4. Função de avaliação 107
12.5. Ensaios e resultados obtidos 109
12.6. Comparação com métodos clássicos 113

13. Conclusões 114
13.1. Trabalho realizado 114
13.2. Trabalhos futuros e possibilidades de extensão 115

Anexo A: "GÉNIO", uma ferramenta genética genérica 116
A.1. Arquitectura do simulador 116
A.2. Tipos de cromossoma suportados 118
A.3. Operadores relacionais 118

Anexo B: "GAS", um simulador simplificado 120

Bibliografia 125

1. Introdução

Neste primeiro capítulo descrevem-se o objectivo, a motivação deste trabalho e o seu enquadramento. Apresentam-se os Algoritmos Genéticos (AG), a Optimização Relacional e a ligação entre os dois conceitos. Finalmente, apresenta-se a organização por capítulos desta tese.

1.1. Objectivo, motivação e enquadramento

É importante, quando se começa um trabalho, perceber qual o seu objectivo. Mas é talvez tão importante como isso perceber qual a sua necessidade e como é que ele se enquadra em ideias, projectos e realizações já existentes. Faremos aqui uma pequena abordagem ao porquê de aplicar uma nova técnica à Optimização Relacional.

Objectivo

O objectivo desta tese é a aplicação de Algoritmos Genéticos ao problema da Optimização Relacional. Este problema resume-se a encontrar o melhor algoritmo de acesso a uma base de dados relacional, para implementar uma pergunta expressa numa linguagem não procedimental, como por exemplo o SQL.

Motivação

O problema da Optimização Relacional é um problema complexo e difícil de resolver. As técnicas tradicionais de optimização [Ullman89] têm obtido bons resultados utilizando algoritmos heurísticos, embora sofram de algumas limitações. As mais importantes provêm do facto de a optimização tradicional ser feita em duas fases disjuntas, a fase *lógica* e a fase *física*. Esta separação pode implicar um desperdício de oportunidades de optimização combinada.

Por outro lado, os algoritmos tradicionais são algoritmos não-interruptíveis, ou seja, é necessário esperar até ao fim do algoritmo para obter uma solução. Em ambientes em tempo real, é importante obter a melhor solução num determinado espaço de tempo, mesmo que não seja a óptima. Isto faz com que os optimizadores tradicionais possam não ser uma boa opção para ambientes deste tipo.

Por fim, são algoritmos heurísticos. Isto leva a que as soluções não sejam forçosamente óptimos globais, o que não constitui um problema de maior, mas que faz com que o programador de um algoritmo deste tipo tenha que ter muita experiência de utilização de bases de dados relacionais, por forma a conseguir inferir as melhores heurísticas.

Por isto tudo, pareceu-nos que seria interessante aplicar AG na resolução deste problema, tentando evitar essas limitações.

Enquadramento

Os Algoritmos Genéticos têm vindo a impôr-se cada vez mais como uma técnica genérica de optimização combinatória, levando à existência de numerosos projectos de investigação em curso nesta área [Goldberg89], além de aplicações em situações reais [Davis91], [Goldberg89], [Yin91] e ferramentas genéricas [Bayer91].

Dos projectos de investigação destacamos a utilização de AG em arquitecturas paralelas [Bessière92], a programação genética [Koza91a], a determinação de arquitecturas para redes neuronais [Martí92a], [Martí92b], [Belew90], [Chalmers90] e a aquisição de regras por sistemas baseados em conhecimento [Angeline92], [Grefenstette89], [DeJong91a], [Goldberg89].

Em Portugal anda não há experiência apreciável nesta área.

A necessidade de um optimizador relacional foi sentida durante o projecto SAGA, do INESC [Neta92] para poder obter uma ferramenta *lower*-CASE independente do SGBD. O optimizador genético irá ser integrado no interpretador de SQL da ferramenta SAGA. Para uma descrição mais completa do interpretador, ver o capítulo 11.

1.2. Algoritmos Genéticos

Algoritmos Genéticos (AG) são algoritmos de optimização inspirados nos mecanismos de selecção e genética naturais [Goldberg89]. Baseiam-se na evolução de uma *população* de *indivíduos*, formada por *cadeias* de informação (cromossomas artificiais constituídos por bits). Cada indivíduo representa uma solução potencial para o problema a optimizar, e possui um determinado *valor*, dado por uma *função de avaliação*. Esta população evolui em *gerações* e cada nova geração é constituída pela *recombinação* de pedaços dos cromossomas dos indivíduos *com maior valor* e pela *introdução aleatória* de algum material genético novo.

Na descrição anterior foram salientados alguns conceitos, em itálico. Estes conceitos formam a base dos algoritmos genéticos e irão ser explicados mais em pormenor nos capítulos seguintes. No entanto, valerá a pena fazer aqui uma descrição sumária do que são e como interactuam.

Funcionamento dos AG

O processo de optimização genética começa sempre pela criação de uma população inicial, pela geração de um conjunto de cadeias de bits, de uma forma completamente aleatória. As cadeias assim geradas representam uma amostragem uniforme do espaço das soluções para o problema a optimizar.

Em seguida, cada elemento da população é avaliado pela função de avaliação e recebe uma medida do seu valor, ou *adaptação*[1]. Esta função deve ser tal que pontue melhor os melhores indivíduos, do ponto de vista da resolução do problema a optimizar. Um indivíduo possui assim uma cadeia de bits que representa uma solução para o probelma (o cromossoma artificial) e uma medida da sua adaptação (o valor retornado pela função de avaliação).

Uma nova geração é então criada pela aplicação de três *operadores genéticos*: a *selecção*, o *cruzamento* e a *mutação*. A selecção escolhe os indivíduos que se vão reproduzir para criar a nova geração, de uma forma viciada para os melhores, ou seja, aos melhores indivíduos (no sentido da adaptação) é dada maior probabilidade de escolha do que aos piores.

Os indivíduos são emparelhados e cada par vai colocar dois descendentes na nova geração, pela aplicação dos operadores de cruzamento e mutação. O cruzamento combina, com determinada probabilidade, pedaços dos cromossomas dos "pais" para gerar os cromossomas dos "filhos", enquanto que a mutação altera aleatoriamente alguns bits dos cromossomas, de preferência em baixo número para cada geração.

A nova geração é avaliada e o ciclo recomeça, criando uma série de novas populações, sempre com o mesmo número de elementos.

---

[1] O termo *adaptação* é usado pelos biólogos e corresponde a uma medida da capacidade de um indivíduo sobreviver e gerar descendência

Figura 1. Fluxograma de um Algoritmo Genético

Esta definição simples dá origem a algoritmos de pesquisa e optimização que têm como principal característica a *robustez* [Goldberg89], ou seja, a capacidade de endereçar eficientemente problemas muito diferentes com apenas uma técnica básica: desde que um problema possa ser representável por uma cadeia de informação, é possível, sem alteração do algoritmo, chegar a uma solução próxima do óptimo.

Esta capacidade é encontrada em quase todos os sistemas naturais [Hofstadter79], enquanto que é extremamente incipiente mesmo nos sistemas artificiais mais elaborados. O que se verifica é que existem técnicas optimizadas para uma gama estreita de problemas mas que raramente têm aplicabilidade fora dessa faixa. Claro que a vasta aplicabilidade não chega: é necessário que o algoritmo seja eficiente. Verifica-se que os algoritmos genéticos se podem, muitas vezes, bater de igual para igual com técnicas específicas desenvolvidas para uma área de aplicação restrita, além de que é quase sempre possível [Davis91] combinar o melhor de ambas as abordagens por forma a conseguir um "híbrido" que tem melhor desempenho que qualquer dos "pais".

No entanto, há alguns problemas, os chamados problemas "enganadores" [Goldberg91d], [Grefenstette91b] que podem levar a que um AG convirja para sub-óptimos. A existência destes problemas está ligada à codificação escolhida para o cromossoma e a sua importância pode ser atenuada pelo uso de técnicas mais avançadas, como iremos ver em capítulos posteriores.

Diferenças essenciais para os métodos clássicos de optimização

Os AG possuem uma série de características que os diferenciam das técnicas de optimização mais tradicionais. Este assunto será visto mais em pormenor no capítulo 2 mas vale a pena fazer aqui um sumário das diferenças :

- Usam uma codificação dos parâmetros numa cadeia de bits, ao invés dos valores dos parâmetros. O significado dos bits é completamente indiferente para os AG.
- Pesquisam a partir de uma população de pontos e não a partir de um só ponto
- Usam apenas a função de avaliação e não necessitam de derivadas ou outro conhecimento acessório sobre a estrutura do problema
- Usam de regras probabilísticas de transição, em vez de regras determinísticas, representadas pelos operadores de selecção, cruzamento e mutação.

Os algoritmos genéticos obrigam a que o conjunto natural de parâmetros que definem o problema possa ser codificado numa cadeia finita de símbolos pertencentes a um alfabeto finito. É de notar que qualquer problema que possa ser codificado para resolução automática tem que possuir esta possibilidade [Ullman81]. Normalmente, e por razões que se prendem com a maximização da informação disponível [Goldberg91b], usa-se uma codificação binária explícita, embora essa codificação possa tomar muitas formas [Davis91] (base binária, códigos de Gray, matriz de conectividade, etc). Como iremos ver, e ao contrário dos métodos tradicionais, os algoritmos genéticos aproveitam as *semelhanças* introduzidas pela codificação.

Na maioria dos métodos de optimização, a zona de pesquisa move-se de um ponto para outro usando regras determinísticas para gerar o próximo ponto a partir do anterior. Este método é perigoso por ser a receita perfeita para convergir para picos menores num espaço de estados com mais do que um máximo local. Em contraste, os algoritmos genéticos pesquisam a partir de uma população de pontos, explorando vários picos em paralelo e diminuindo assim a probabilidade de encontrar um falso pico. Além disso, esta abordagem populacional faz com que seja possível implementar algoritmos genéticos com grande facilidade e sem "engarrafamentos" em máquinas muito paralelas [Bessière92], [Goldberg91a], facilidade que não é para desprezar, pois não é oferecida pelos métodos tradicionais de pesquisa e nem sequer por alguns menos tradicionais, como o *simulated annealing* [Kirkpatrick83].

Muitas técnicas de pesquisa, por outro lado, exigem que a função a optimizar seja contínua e derivável. Os algoritmos genéticos não exigem nada disto, bastando-lhe uma função de avaliação capaz de classificar os pontos do espaço de estados, representados

pelos indivíduos da população. Isto faz com que os algoritmos genéticos sejam um processo de pesquisa mais genérico do que os baseados em derivadas. Por outro lado, esta simplicidade dos AG provoca alguma falta de competitividade em comparação com os métodos que exploram informação adicional. No entanto, usando técnicas de *hibridização* [Davis91], é possível conjugar as duas coisas sem sacrificar totalmente a robustez inerente dos AG.

Ao contrário de outros métodos, os AG usam regras probabilísticas de transição para guiar a pesquisa. Na realidade, os AG não são métodos de pesquisa aleatórios, apenas usam a escolha aleatória como um meio de explorar zonas do espaço de estados que parecem promissoras.

1.3. Optimização relacional

O problema da optimização de perguntas relacionais surge sempre que se quer pôr na prática um interpretador de uma linguagem semelhante ao SQL [Ullman89]. De facto, as linguagens de manipulação de dados relacionais, como iremos ver mais em detalhe no capítulo 10, são linguagens com características muito próprias e que podem ser extraordinariamente pouco eficientes [Ranito91] quando implementadas de um modo simplista.

Os métodos tradicionais de optimização relacional baseiam-se numa separação em duas fases: a optimização lógica e a optimização física. Estas fases são totalmente disjuntas, quer no tempo, quer nos objectivos.

Durante a fase de optimização lógica, a pergunta inicial é transformada para uma forma canónica qualquer, normalmente álgebra relacional [Ullman89]. Em seguida, a expressão em álgebra relacional é manipulada, por forma a eliminar operações muito demoradas (como o produto cartesiano) e a diminuir o tamanho das relações em causa. Esta optimização é heurística e o resultado é uma expressão equivalente mas mais eficientemente implementável.

O processo de reescrita que transforma a expressão original na expressão optimizada é baseada nas propriedades de equivalência da álgebra relacional (ver capítulo 10).

A fase de optimização física decorre a seguir, sobre a expressão encontrada na primeira fase. A expressão optimizada é transformada num algoritmo de acesso aos dados, tendo em conta a existência dos diversos métodos de acesso normalmente

encontrados num SGBD relacional, como índices, tabelas de *hash*, acesso sequencial, etc.

O problema da optimização relacional é muito complexo, em termos temporais, se for usada uma abordagem enumerativa sem recurso a heurísticas, uma vez que o número de soluções possível cresce exponencialmente com a complexidade do esquema relacional [Ullman89].

1.4. Optimização relacional usando algoritmos genéticos

Como em qualquer outro problema, a utilização de AG para optimização relacional passa pela possibilidade de mapeamento de uma solução num cromossoma binário. Este mapeamento é sempre possível, pelo menos para problemas que podem ser resolvidos por computadores [Ullman81], mas é necessário encontrar uma codificação que preencha alguns requisitos, como iremos ver mais em detalhe no capítulo 5.

No nosso caso, em vez de uma codificação directa de uma solução, foi codificado um programa de *reescrita* que transforma a expressão inicial numa expressão optimizada (ver capítulo 12). As operações de reescrita usadas são derivadas a partir das equivalências básicas da álgebra relacional. Assim, cada cromossoma representa uma lista de transformações a aplicar à expressão original.

Apesar de este processo parecer muito semelhante à fase de optimização lógica, a verdade é que aglutina também a fase de optimização física. Isto é feito através da extensão da noção de álgebra relacional por forma a incluir informação sobre os métodos físicos de acesso, permitindo assim uma optimização simultânea e complementar das duas vertentes tradicionais.

1.5. Estrutura da tese

Embora estruturada em capítulos, existem três partes diferentes na tese, que correspondem à expansão dos tópicos abordados nas secções anteriores: Algoritmos Genéticos, optimização de perguntas relacionais e aplicação de AG à optimização relacional. Esta última parte corresponde também à descrição do protótipo que foi implementado.

Tendo isto em conta, vamos ver do que fala cada um dos capítulos.

No capítulo 2 iremos ver em mais detalhe como funciona um algoritmo genético, qual a sua composição e descrever cada um dos operadores. Falaremos de *esquemas*, um conceito fundamental para perceber porque funciona um AG e faremos uma revisão dos métodos clássicos de optimização e da sua relação com os AG. O capítulo 3 faz um resumo da história dos AG, enquanto que o capítulo 4 analisa as suas bases matemáticas, fala do Teorema Fundamental e de problemas enganadores. Os problemas da *codificação* e da *parametrização* são discutidos no capítulo 5. O capítulo 6 faz uma análise mais aprofundada dos operadores de selecção e de cruzamento e o capítulo 7 refere alguns melhoramentos que se podem fazer ao AG canónico, nomeadamente para tratar cromossomas não lineares. O capítulo 8 é dedicado à descrição de alguns ensaios realizados pelo autor na aplicação de AG a problemas práticos, que englobaram *job-shop*, TSP, Redes Neuronais e regressão simbólica. Estes capítulos constituem a primeira parte da tese, dedicada aos AG.

A segunda parte trata de optimização relacional. No capítulo 9 é exposta a necessidade da optimização relacional, é apresentada a Álgebra Relacional e são referidos os métodos de acesso que podem ser usados num Sistema de Gestão de Bases de Dados Relacional (SGBDR). O capítulo 10 descreve e comenta os algoritmos tradicionais de optimização relacional, e as suas fases principais, a optimização lógica e a optimização física.

A última parte trata da aplicação de AG à optimização relacional. O capítulo 11 descreve a arquitectura do interpretador de SQL em que se integra o optimizador genético, o "AGORA", e explica o interface deste com os restantes módulos. O capítulo 12 explica as opções tomadas para a implementação do "AGORA" e descreve a sua arquitectura interna, além de apresentar e discutir os resultados obtidos. Por fim, o capítulo 13 contém algumas notas sobre o trabalho realizado, o trabalho que poderá ser realizado no futuro e considerações finais várias.

Há ainda 2 anexos em que são apresentadas as ferramentas genéricas implementadas durante este trabalho, o "GÉNIO" e o "GAS". O primeiro é uma ferramenta que incorpora vários tipos de cromossoma e de operadores genéticos e possui um interface em NextStep [Neto93]. O segundo é uma ferramenta mais simples de usar e que apenas contempla AG canónicos. No entanto, apesar de mais simples, foi o "GAS" o simulador usado para a implementação do "AGORA".

2. Algoritmos Genéticos

Neste capítulo vamos ver em mais detalhe como funciona um algoritmo genético, qual a sua composição e descrever em mais pormenor cada um dos operadores. Faremos ainda uma leve abordagem ao problema do processamento de *esquemas*, um conceito fundamental para perceber porque funciona um AG. Por fim, faremos uma revisão dos métodos clássicos de optimização e da sua relação com os AG.

2.1. Propriedades gerais de um AG

Para a discussão que se vai seguir, admitamos que queremos optimizar a função $f(x) = x^2$ no intervalo inteiro $I = [0,31]$. A codificação a usar será a representação em base binária pura de x. Necessitamos para isso de 5 bits.

Em essência, o AG canónico[2] é composto por uma população de N cadeias de bits que evolui em gerações. Cada cadeia representa uma solução para o problema a optimizar. A geração de ordem $t+1$ é construída pela aplicação probabilística, à geração de ordem t, de três operadores:

- Selecção
- Cruzamento
- Mutação

A *selecção* é o processo pelo qual cadeias individuais são escolhidas para reprodução, de acordo com o seu valor na função de avaliação, f. Intuitivamente, podemos ver a função f como uma medida de lucro, utilidade ou qualidade que se quer maximizar. "De acordo com o seu valor" significa que a indivíduos com maior valor da função de avaliação será dada maior probabilidade de contribuir geneticamente para a criação de um ou mais descendentes na geração seguinte. Este operador é uma versão artificial do processo de selecção natural de Darwin, baseado na adaptação ao meio e na

[2]Há algumas variações sobre o AG canónico, que iremos abordar em capítulos posteriores. No entanto, para efeito de apresentação, achámos preferível começar por descrever apenas o AG mais simples.

capacidade de sobreviver a predadores, doenças e outros obstáculos ao crescimento e posterior reprodução. Num sistema artificial como o nosso, a função de avaliação é o árbitro final quanto à sobrevivência ou morte dos elementos da população.

Nº	Cadeia	Valor de *f*	%Total
1	01101	169	14.4
2	11000	576	49.2
3	01000	64	5.5
4	10011	361	30.9
Total		1170	100.0

Tabela 1. Exemplo de cadeias e valores para um problema de optimização

Há várias formas [Goldberg89], [Goldberg91c] de implementar algoritmicamente o operador de selecção. O mais simples é construir uma roleta "viciada", em que cada cadeia da população tem direito a uma ranhura cujo tamanho é ajustado proporcionalmente ao seu valor. Para percebermos melhor, tomemos a tabela anterior, aonde é mostrada, como exemplo, uma população para a resolução do nosso problema. A roleta correspondente a esta tabela é mostrada na figura seguinte e podemos ver que, para determinar o tamanho de cada ranhura, calculamos qual é a percentagem do valor de cada indivíduo no valor total da população (soma de todos os valores). Assim, cada elemento da população recebe uma área da roleta proporcional ao seu valor.

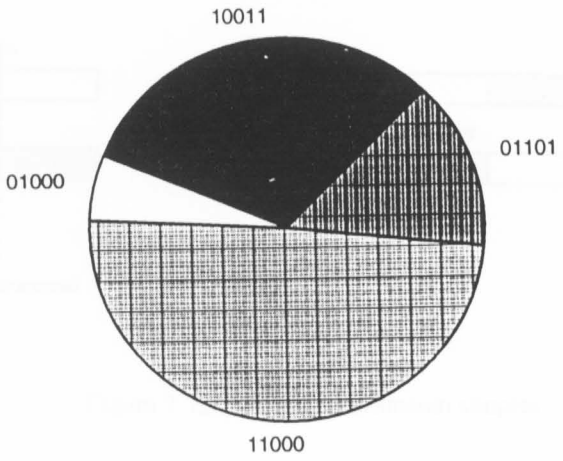

Figura 2. Roleta viciada correspondente à Tabela 1

De cada vez que é necessário escolher para reprodução um elemento da população, fazemos girar esta roleta, dando assim maior número de descendentes aos indivíduos com maior valor. Após a escolha, uma cópia exacta do indivíduo seleccionado é colocada num grupo de reprodução pronto a ser submetida a posterior processamento genético e incorporado na geração seguinte.

Um dos operadores que pode actuar no grupo de reprodução é o operador de *cruzamento*, que opera em duas fases. Primeiro, são escolhidos ao acaso dois elementos do grupo reprodutor, elementos esses que vão dar origem a dois descendentes. A seguir, é escolhida uma posição k, aleatória, no cromossoma, dando a mesma probabilidade a qualquer uma das posições da cadeia. Os dois novos elementos são formados pela troca dos caracteres que ficam entre a posição k e o fim da cadeia. Por exemplo, consideremos os elementos A_1 e A_2, da nossa população inicial:

$A_1 = 0\ 1\ 1\ 0\ 1$
$A_2 = \mathbf{1\ 1\ 0\ 1\ 0}$

Suponhamos que, ao acaso, obtemos uma posição de cruzamento, $k = 3$. O cruzamento dá origem a dois novos indivíduos, em que o símbolo ' significa que pertencem à nova geração:

$A'_1 = 0\ 1\ 1\ \mathbf{1\ 0}$
$A'_2 = \mathbf{1\ 1\ 0}\ 0\ 1$

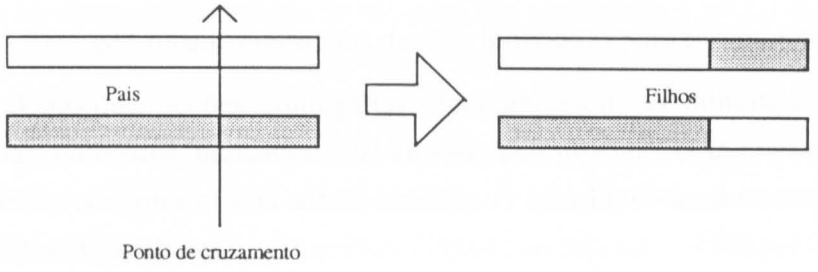

Figura 3. Esquema do cruzamento simples

O operador de cruzamento não é usado em todos os pares, sendo a sua aplicação ditada por um valor, a *probabilidade de cruzamento*. Os indivíduos que não sofrem cruzamento são copiados integralmente para a nova população.

O funcionamento destes operadores, reprodução e cruzamento, é surpreendentemente simples: afinal de contas, limitamo-nos a trocar pedaços de cadeias de informação de elementos escolhidos ao acaso. No entanto, verifica-se que muito do poder dos algoritmos genéticos advém precisamente da combinação destes dois operadores aleatórios, como iremos ver quando analisarmos o teorema fundamental dos algoritmos genéticos, o Teorema dos Esquemas. De qualquer forma, podemos tentar dar uma analogia, para facilitar a criação de uma noção intuitiva de como um AG funciona:

O matemático J. Hadamard (1949) propôs que a descoberta de novas ideias não pode ser atribuída unicamente ao acaso, antes surgindo pela combinação de "bocados" de ideias já existentes e que deram bons resultados no passado. Sugere ainda que o factor acaso intervém, principalmente, no modo como as noções anteriores se conjugam para dar origem a novas ideias [Goldberg91h], [Goldberg89]. Este é o modo como funcionam os algoritmos genéticos, combinando pedaços de informação que foram úteis, na esperança de que o resultado seja melhor que qualquer dos antepassados.

Se olharmos de perto para uma população qualquer, definida por cadeias de caracteres, vemos que existem semelhanças entre os diversos indivíduos e que essas semelhanças estão ligadas ao desempenho dos elementos, ou seja, é possível detectar regularidades em *subcadeias* de informação, ou padrões, que conduzem a um bom desempenho dos indivíduos. Os algoritmos genéticos exploram esta informação e:

1) seleccionam preferencialmente os bons indivíduos para reprodução, ou seja, aqueles que contêm, provavelmente, subcadeias que conduzem a um bom desempenho,
2) combinam subcadeias de dois indivíduos para formar um novo indivíduo.

Estas duas acções, conjugadas, têm como efeito a combinação muito provável de cadeias, ou blocos, indicadoras de elevado desempenho, com boas probabilidades de o resultado ser uma cadeia ainda melhor. A esta hipótese é costume dar o nome de "Hipótese dos Blocos".

O valor desta troca de informação útil, para criar informação ainda mais útil, sempre foi reconhecida por todos os cientistas, donde a existência de congressos, seminários, etc, aonde as pessoas vão "cruzar" os seus trabalhos, a que correspondem "subcadeias" de informação que levam a um bom desempenho, na esperança que algum ouvinte os saiba combinar por forma a obter um avanço na ciência.

Sendo assim, podemos dizer que os algoritmos genéticos tendem a construir uma solução boa pela combinação de "blocos" que revelaram bom desempenho anteriormente. Esta construção por blocos é que dá aos AG a sua força, embora possa, se mal explorada por uma má codificação do problema, contribuir também para a sua fraqueza.

Quanto à *mutação*, o último operador genético, limita-se a trocar, com baixa probabilidade, o valor de um bit de uma cadeia, escolhido aleatoriamente. O seu papel é mais limitado do que o dos outros operadores. De facto, nos AG, a mutação é apenas um mecanismo de recurso, servindo para recuperar material genético eliminado por uma selecção zelosa de mais ou para introduzir alguma diversidade numa população que está a estagnar. A mutação deve actuar em pequena quantidade, sob pena de transformar a pesquisa genética em pesquisa aleatória.

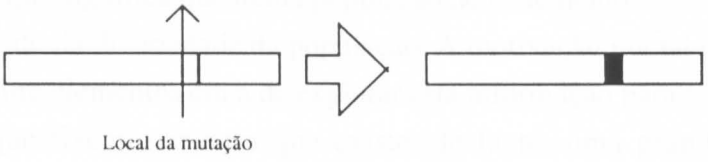

Figura 4. Esquema da mutação simples

Semelhanças e esquemas

A forma como os AG conseguem "escolher" os melhores blocos para reprodução pode ser apresentada de uma forma intuitiva, recorrendo à noção de *esquema*. Segundo Holland [Holland75], um esquema é um padrão de semelhança que descreve um conjunto de cadeias com similaridades em determinadas posições. Para podermos lidar com esquemas, introduzamos mais um carácter no nosso alfabeto, o carácter "*", com o significado de "0 ou 1". É um *meta-carácter* que pode representar qualquer carácter do alfabeto. Por exemplo, o esquema *0000 corresponde ao conjunto {00000, 10000}, enquanto que o esquema *111* corresponde ao conjunto {01110, 01111, 11110, 11111}. Como veremos, os esquemas são uma forma simples e poderosa para falar acerca da semelhança entre cadeias.

Para contar quantos esquemas possíveis existem, basta ver que numa cadeia de comprimento l podemos ter 3^l esquemas diferentes, uma vez que temos três símbolos (1, 0 e *) e l posições. Em geral, podemos ver que o número de esquemas possíveis para um alfabeto com k caracteres é $(k + 1)^l$. Numa primeira análise, parece que os

esquemas apenas vêm tornar a pesquisa mais difícil: já não bastava ter um espaço de estados de k^l, agora passamos a ter um espaço de $(k + 1)^l$.

Na verdade, a introdução dos esquema vem facilitar as coisas. O que acontece é que, se considerarmos uma cadeia de cada vez, apenas podemos contar com n pedaços de informação, em que n é o número de cadeias na população. Se passarmos a compará-las umas com as outras, olhando para valores da função de custo e para as semelhanças entre elas, estamos a incorporar uma grande fonte de informação. Já agora, que quantidade de informação é que introduzimos com a comparação dos vários elementos da população?

Para arranjar um valor limite para a resposta à pergunta anterior (que depende de quais são as cadeias presentes na população), vamos considerar uma cadeia de comprimento 5: 11111, por exemplo. Esta cadeia pertence a 2^5 esquemas, pois cada posição pode conter um valor constante ou o símbolo "*". Em geral, uma cadeia contém 2^l esquemas, o que significa que numa população de n elementos, se encontram entre 2^l e $n2^l$, dependendo da diversidade da população. A motivação inicial para considerar as semelhanças entre elementos era a de explorar esta informação para facilitar a pesquisa. A contagem que fizemos mostra que existe, de facto, uma grande quantidade de informação a explorar.

No entanto, até que ponto é que estes esquemas todos são efectivamente processados por um algoritmo genético? Não será necessário algum paralalelismo para explorar eficientemente toda esta quantidade de informação? Para obter uma resposta a estas questões temos que considerar o efeito conjugado da reprodução, cruzamento e mutação.

A reprodução é fácil de analisar: uma vez que a reprodução dos melhores indivíduos é preferida, é dado, em média, um número cada vez maior de amostras aos melhores esquemas; é bom lembrar, no entanto, que a reprodução, por si só, não é capaz de gerar novos pontos, apenas escolher os melhores. O que é que acontece com o cruzamento, o operador responsável pela recombinação do material genético?

O cruzamento tanto pode deixar o esquema exactamente como está, como pode rompê-lo, se o ponto de corte cair no meio do esquema. Consideremos, por exemplo, os esquemas 1***0 e **11*. O primeiro está bastante sujeito a destruição, enquanto o segundo tem uma probabilidade muito maior de sobrevivência. Isto leva a que os esquemas com um *comprimento* (distância entre os elementos mais exteriores diferentes de "*") pequeno sejam deixados em paz pelo operador de cruzamento e estejam aptos a ver o seu número aumentar, graças ao operador de reprodução.

Por fim, a mutação, desde que em pequenas proporções, não tem muitas hipóteses de destruir um esquema[3], o que nos leva a uma conclusão interessante: esquemas curtos, com bom desempenho, são rapidamente propagados para gerações seguintes e o número de indivíduos que os contêm cresce rapidamente com o número da geração. Isto acontece sem nenhum registo ou memória especiais: apenas usando a nossa população de n cadeias. Prova-se [Goldberg89] que o número de esquemas processados com alguma utilidade ronda os n^3. Isto compara-se muito favoravelmente com o número de avaliações da função de avaliação (n) correspondentes, o que induz uma espécie de *paralelismo implícito*, uma das características distintivas dos algoritmos genéticos [Grefenstette91].

2.2. Posicionamento face à optimização clássica

Embora o objectivo desta dissertação não seja o estudo comparativo das diversas técnicas de pesquisa e optimização, é importante ver algumas características dos algoritmos de optimização tradicional que implicam limitações. Esta percepção permite-nos ver melhor as vantagens de uma abordagem genética, para obter algoritmos robustos. Vamos apenas rever aqueles que são genéricos, i.e., que podem ser aplicados sem necessidade de conhecer a natureza do problema. Assim, não iremos falar de algoritmos *heurísticos* [Oliveira88], por exemplo, ou de métodos que tiram partido da estrutura do problema, como algoritmos de *afectação* ou de *transporte* [Soeiro88].

A literatura actual [Goldberg89] identifica três tipos principais de métodos de pesquisa genéricos: *baseados em cálculo*, *enumerativos* e *aleatórios*. Examinemos cada um destes tipos para ver que conclusões podem ser retiradas sem uma análise formal.

Métodos baseados em cálculo

Os métodos baseados em cálculo foram estudados em pormenor e subdividem-se em duas classes: *indirectos* e *directos*. Os métodos indirectos procuram extremos locais pela resolução do sistema de equações, normalmente não linear, que resulta de igualar a zero o gradiente da função a optimizar. Isto é a generalização multi-dimensional da noção elementar de pontos extremos, como é ilustrado na figura 5. Dada uma função contínua e derivável, e para encontrar um máximo sem restrições, devemos começar por restringir a pesquisa aos pontos com inclinação zero em todas as direcções. No caso

[3]Como veremos em pormenor no capítulo 4

genérico, a resolução do sistema de equações resultante é extraordinariamente complexo e impossível de automatizar.

Por outro lado, os métodos directos procuram os óptimos locais "saltando" pela função e movendo-se numa direcção relacionada com o gradiente. Esta noção é simplesmente a de "subida por gradiente" ou *hill-climbing*: para encontrar um óptimo local, subir pela função na direcção admissível mais inclinada. Apesar dos melhoramentos sofridos com o tempo, algum raciocínio simples mostra as suas limitações.

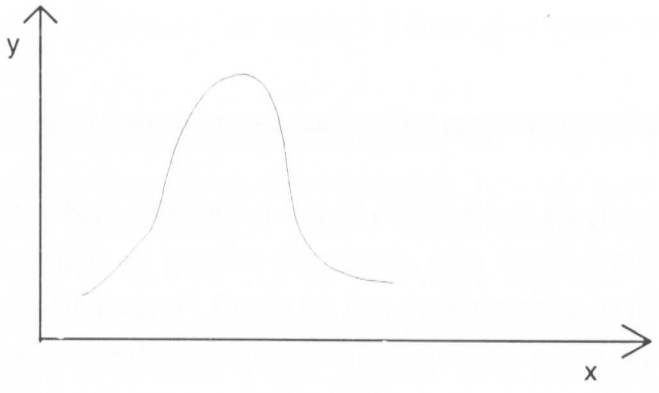

Figura 5. Função fácil para os métodos baseados em cálculo

Em primeiro lugar, estes algoritmos são de âmbito local, ou seja, os óptimos encontrados são apenas óptimos numa vizinhança do ponto corrente. Por exemplo, suponhamos que a figura 5 mostra uma porção de um domínio de interesse e que a imagem mais completa é dada pela figura 6. Claramente, começando a pesquisa na vizinhança do menor dos picos, é seguro de que o algoritmo falha o maior dos picos. Além disso, uma vez encontrado o menor pico, só é possível atingir o maior pico através do recomeço do algoritmo.

Em segundo lugar, este tipo de métodos depende da existência de derivadas (valores de inclinação bem definidos), o que é uma grave limitação, mesmo considerando a possibilidade da sua aproximação numérica. Na prática, muitos dos espaços paramétricos que nos interessam não têm o mínimo respeito pela noção de derivada. Os teóricos interessados na optimização basearam-se no legado dos grandes matemáticos dos séculos XVIII e XIX, que estudaram um mundo feito de funções-objectivo quadráticas e de baixa dimensão, restrições ideais e derivadas omnipresentes.

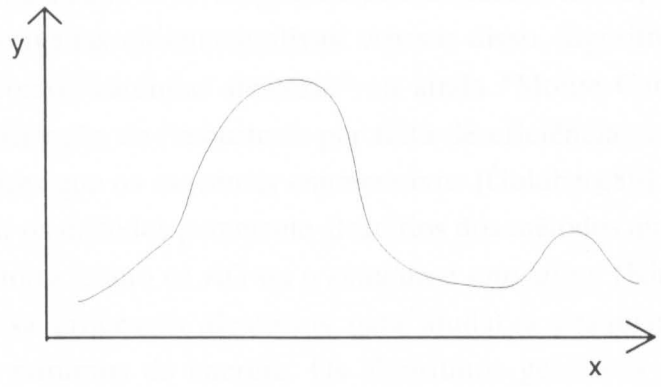

Figura 6. Exemplo de uma função difícil para métodos baseados em cálculo

A realidade é bem diferente: o mundo está cheio de funções com descontinuidades e de espaços de estado de grandes dimensões, com vários óptimos locais e ruidosos, sendo tudo menos "derivável". Como tal, não é surpresa que os métodos que dependem de funções contínuas e deriváveis sejam inaplicáveis a não ser num domínio muito pequeno. Por isto e por serem métodos locais, temos que considerar que têm algumas limitações.

Métodos enumerativos

Doutro lado, temos os algoritmos enumerativos. Estes existem em várias formas e tamanhos, embora tenham todos um modo de funcionamento comum: dado um espaço de estados discreto ou discretizável, percorremos todos os pontos, avaliando a função-objectivo em cada um, verificando qual é o melhor. Embora a simplicidade deste tipo de algoritmo seja atractiva, e apesar de a enumeração ser um método de pesquisa bem humano[4], estes algoritmos também não podem ser considerados como robustos por uma simples razão: falta de eficiência. Mesmo esquemas inteligentes de enumeração, como a *programação dinâmica* [Hu90], sofrem da "maldição da dimensão", como lhe chamou o seu criador, Bellman: o tempo de pesquisa cresce exponencialmente com o número de dimensões ou, em última análise, com o número de bits necessário para representar uma solução do problema.

---

[4]Pelo menos, quando o número de possibilidades é pequeno.

Por fim, temos os algoritmos aleatórios, que têm ganho cada vez mais popularidade, à medida que os investigadores reconhecem as limitações das abordagens orientadas por derivadas ou enumerativas. Apesar disso, algoritmos simplesmente aleatórios, tais como "caminho aleatório" ou ainda "Monte Carlo", também não merecem a classificação de "robustos", por falta de eficiência: a longo prazo, não podem ser melhores que os esquemas enumerativos [Goldberg89]. É importante, no entanto, distinguir os métodos puramente aleatórios dos métodos que incorporam uma componente aleatória, como os AG ou o *simulated annealing* [Kirkpatrick83]. Esta última técnica usa processos aleatórios para ajudar a pesquisa de estados que correspondem a mínimos de energia. Os algoritmos genéticos podem, mediante algumas pequenas alterações, ser vistos [Goldberg91] como uma generalização de *simulated annealing*.

Esta discussão, forçosamente não exaustiva, leva-nos a uma conclusão: os métodos tradicionais de pesquisa têm limitações. Isto não quer dizer que não sejam úteis, apenas que o seu desempenho não é o melhor, fora dos problemas para os quais foram concebidos. É evidente que estes métodos têm sido usados, e com muito sucesso, para resolver problemas reais mas a verdade é que não são facilmente "ampliáveis" para problemas muito grandes ou muito complexos. A figura 7 [Goldberg89] dá uma ideia de como deveria ser o desempenho de um método ideal, ou método *robusto*.

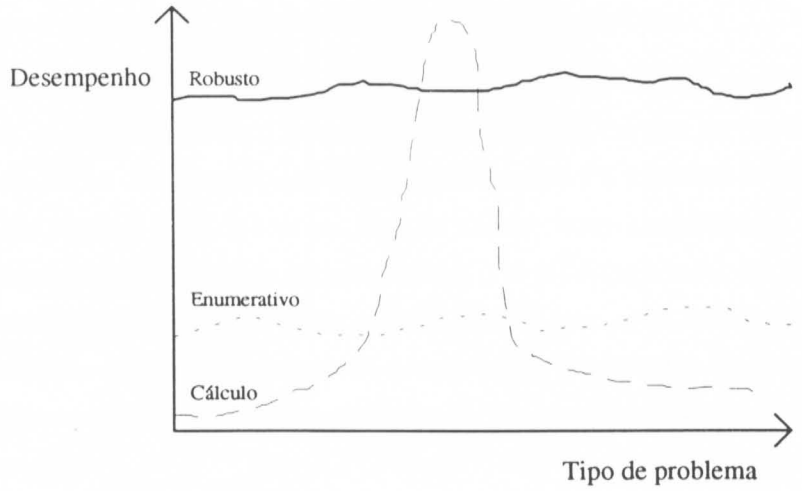

Figura 7. Comparação do desempenho de métodos de optimização

A curva mais equilibrada é a correspondente ao método robusto, dando a ideia de que, se calhar, vale a pena sacrificar um pouco de *performance* de pico para conseguir um método que dê bons resultados em quase todos os casos. Os métodos enumerativos funcionam com ineficiência igual para todos os casos. Por fim, os métodos baseados em gradiente dão óptimos resultados numa faixa estreita de utilização e bastante piores fora dessa área.

3. Perspectiva histórica

Sem querer fazer aqui um inventário completo da história dos Algoritmos Genéticos (para tal objectivo, ver [Goldberg89]), vale a pena citar os passos mais importantes que foram dados neste campo, desde o seu nascimento em 1962, pela mão de John Holland.

Antes dessa data já tinham surgido algumas tentativas de modelização de sistemas genéticos em computador (Barricelli, 1957, 1962; Fraser, 1960, 1962; Martin e Cockerham, 1960). Apesar de estes estudos terem como objectivo fundamental o de compreender uma série de fenómenos biológicos, o trabalho de Fraser, por exemplo, já estava perto do que se considera hoje um algoritmo genético.

No entanto, foi John Holland e os seus alunos na Universidade de Michigan quem reconheceu que seria proveitoso usar operadores com base genética em problemas artificiais de adaptação. Holland, em 1962, foi o primeiro a dar ênfase ao papel fundamental da selecção artificial como meio de escolher os melhores, e a preconizar o uso de *populações* em vez da tradicional abordagem "ponto-por-ponto". Além disso, cerca de três anos mais tarde, foi também pioneiro no reconhecimento do cruzamento como motor primário de exploração nos sistemas adaptativos.

Foi Bagley, em 1967, quem primeiro mencionou a expressão "algoritmo genético" e quem mostrou a primeira aplicação prática dos recentes conhecimentos. Bagley escreveu uma dissertação pioneira em que usava algoritmos genéticos para fazer evoluir um jogador artificial de um jogo de xadrez muito simplificado. A sua dissertação foi fundamental para assentar algumas ideias que já conhecemos: um algoritmo genético é composto por uma população de cadeias de bits, que sofre a acção de três operadores fundamentais: selecção, cruzamento e mutação. Além disso, Bagley foi mais longe e previu a necessidade, em alguns problemas, de operadores mais poderosos do que os três referidos.

Após uma série de trabalhos que envolveram biologia (Rosenberg, 1967), reconhecimento de padrões (Cavichio, 1970) e simulação de células (Weinberg, 1970), foi Hollstien, em 1971, o primeiro a usar algoritmos genéticos num problema de optimização matemática pura. Hollstien interessou-se principalmente por analisar as

diferenças entre diversos esquemas reprodutivos, baseados na experiência com a criação de animais em quintas. Foi o primeiro a propor uma codificação diferente (códigos de Gray) para os números inteiros representados em bits, como forma de evitar alguns problemas associados à representação em base binária pura.

Cerca de 1971, John Holland dá outro grande passo em frente na solidificação da base teórica dos algoritmos genéticos e propõe o Teorema dos Esquemas, um teorema fundamental que, entre outras coisas, preconiza o uso de representações binárias como forma de maximizar a utilização da informação disponível. A partir deste marco, os trabalhos divergem: uma linha, a minimalista, começa a explorar os caminhos binários abertos pelo Teorema dos Esquemas, enquanto outra facção continua a investigar sistemas genéticos baseados em codificações não binárias.

O marco seguinte é a publicação, em 1975, de duas obras muito importantes: "Adaptation in Natural and Artificial Systems", de Holland e "An Analysis of the Behavior of a Class of Genetic Adaptive Systems", a dissertação de De Jong. Esta dissertação foi fulcral, pois combinou a teoria dos esquemas de Holland com um conjunto muito cuidadoso de experiências, em que os algoritmos genéticos, o ambiente e os critérios de *performance* foram "dissecados" até ao essencial, permitindo a formação de um conjunto de resultados que serviu de base para a grande maioria dos estudos subsequentes.

A evolução seguinte foi a utilização dos algoritmos genéticos como motor para aquisição de regras através de experiências. Por exemplo, Goldberg, em 1985, utilizou algoritmos genéticos para determinar um conjunto de regras para controlo de uma rede de gás, conjunto esse capaz de rivalizar com os peritos na matéria. Com este tipo de abordagem pode criar-se um sistema pericial sem passar pela parte morosa de extracção do conhecimento de um perito, antes abstraindo regras de experiências. A utilização de algoritmos genéticos é comparável a técnicas específicas, semelhantes ao ID5R [Spears91a].

Cerca de 1987, e numa generalização deste tipo de problemas, Koza [Koza91a] propõe a utilização da "programação genética", em que algoritmos descritos em LISP são cruzados e reproduzidos por forma a dar origem a programas que resolvam problemas difíceis (comportamento emergente, controlo óptimo, regressão simbólica, etc).

Do ponto de vista teórico, começam-se a explorar os algoritmos genéticos com cromossomas de tamanho variável e operadores mais poderosos, conhecidos como "messy genetic algorithms" ou MGA [Goldberg90]. Este tipo de algoritmo genético

poderá ultrapassar as limitações de que sofrem os algoritmos genéticos convencionais, ou seja, a existência de funções que os "enganam", conduzindo o sistema para uma má solução.

De qualquer forma, os algoritmos genéticos estão a passar por uma fase quase "adulta", isto é, estão a ser utilizados com sucesso em aplicações, se não horizontais, pelo menos verticais. Os campos de aplicação são muito vastos [Davis91], pois os AG têm-se revelado uma boa abordagem a qualquer problema de optimização combinatória para que se não conheça uma solução computacionalmente viável. Por outro lado, são também bons candidatos para paralelização [Bessière92] ou para problemas de decisão multi-critério [Yin91], graças à sua abordagem populacional.

4. Fundamento matemático dos Algoritmos Genéticos

Neste capítulo são analisadas as bases matemáticas que explicam o funcionamento dos AG. Falamos também do Teorema dos Esquemas, o teorema fundamental dos AG. Por fim, é dada alguma atenção a problemas *enganadores*.

4.1. Motivação

Da leitura dos capítulos anteriores não ficou claro, de maneira nenhuma, porquê ou como é que os AG convergem para uma solução, nem como essa solução se relaciona com a solução óptima.

Na realidade, existem teoremas que demostram o porquê da convergência dos algoritmos genéticos para uma boa solução. O teorema fundamental, demonstrado por Holland, serviu de base a novos avanços no domínio da justificação teórica das propriedades dos algoritmos genéticos e é hoje possível prever se uma determinada função é "fácil" ou "difícil" para um AG tradicional, usando métdos como a "transformada de Walsh" [Goldberg91d], análoga à transformada de Fourier.

Neste capítulo vamos procurar mostrar (para uma demonstração mais completa, ver [Goldberg89]) alguns factos matemáticos que servem de base de apoio às nossas afirmações anteriores, sobre a convergência dos AG para boas soluções. Além disso, vamos elaborar melhor o conceito de "semelhança" ou *esquema*, introduzido no capítulo 2.

Quantitativamente, descobrimos que existe um grande número de semelhanças para explorar numa população de cadeias de informação. Intuitivamente, vimos como é que os AG exploram, em paralelo, as semelhanças encontradas entre blocos, ou esquemas, de elevado desempenho. Por forma a formalizar um pouco mais estas observações, vamos proceder à contagem dos esquemas representados numa população e considerar o seu crescimento ou decaimento, para uma determinada geração. Para isto, vamos considerar o efeito dos operadores de selecção, cruzamento e mutação. Esta

análise vai levar-nos ao teorema fundamental dos algoritmos genéticos, que quantifica estas taxas de crescimento e decaimento de uma forma mais precisa.

4.2. O Teorema Fundamental

O funcionamento de um algoritmo genético parece simples. Afinal de contas, começamos com uma população aleatória de cadeias de informação, copiamos algumas, combinamos e fazemos trocas parciais com material genético, além da mutação de um bit aqui e outro acolá. Apesar de este processamento explícito ser muito simples, começámos a reconhecer, numa secção anterior, que existe um processamento implícito de muitos esquemas em cada geração. Para analisar matematicamente o crescimento e decaimento dos variados esquemas contidos numa população, necessitamos de alguma notação para dar rigor à nossa discussão.

Notação e pressupostos

Vamos considerar que as nossas cadeias de informação são constituídas por elementos do alfabeto binário $V = \{0, 1\}$, sem perca de generalidade. Por conveniência notacional, designamos as cadeias completas por letras maiúsculas e os caracteres individuais por letras minúsculas em subscrito. Por exemplo, a cadeia $A = 0111000$ pode ser representada simbolicamente como:

$$A = a_1a_2a_3a_4a_5a_6a_7$$

Aqui, cada a_i representa uma característica[5] ou detector binário, em que cada característica pode tomar o valor[6] 0 ou 1. É também possível ter cadeias em que a informação não está ordenada, havendo algoritmos genéticos mais sofisticados que tiram partido desse facto para que o local de uma característica seja independente da sua função. Para já, vamos assumir que a posição de um gene determina a sua função.

Qualquer pesquisa genética exige uma população de cadeias individuais, A_i, $i = 1,2,...,n$, contidas na população $\boldsymbol{A}(t)$, na geração t, em que o tipo a cheio indica uma população. Além disso, vamos recuperar a nossa nomenclatura anterior para esquemas, ou seja, cadeias com o carácter adicional "*" que "vale" por qualquer

---

[5]Em linguagem "naturalista" chamar-se-ia "gene"

[6]Os biólogos dão o nome de "alelo" ao valor de um gene

carácter do alfabeto. Recordemos ainda que existem $(k + 1)^l$ esquemas numa cadeia de comprimento l e construída a partir um alfabeto de cardinalidade k. Além disso, cada cadeia é representante de 2^l esquemas e que podem, no máximo, existir $n2^l$ esquemas numa população de n elementos.

Propriedades dos esquemas

Nem todos os esquemas são iguais. Alguns são mais específicos que outros, como por exemplo o esquema 011*1** é mais específico acerca da semelhança do que o esquema 0*****. Além disso, alguns esquemas ocupam uma percentagem maior do comprimento total da cadeia do que outros, como podemos ver nos esquemas seguintes: 1*****1 é mais "comprido" do que 1*1****. Para quantificar estas ideias, vamos introduzir as noções de *ordem* e *comprimento* de um esquema.

A *ordem* de um esquema H, denotada por $o(H)$, é simplesmente o número de caracteres fixos (diferentes de "*") que o esquema contém. Por exemplo, o esquema **1*101 é de ordem 4, enquanto o esquema ******1 é de ordem 1.

O *comprimento* de um esquema H, denotado por $\delta(H)$, é a distância entre as duas ocorrências mais externas de um carácter fixo (diferente de "*"). Por exemplo, o esquema 011*1** tem comprimento 4, enquanto o esquema 0***** tem comprimento 0.

Os operadores genéticos e os esquemas

Os esquemas e as suas propriedades podem ser usados para determinar o efeito dos operadores de selecção, cruzamento e mutação sobre os blocos contidos na população. Começaremos por ver qual é o efeito de cada um dos operadores em separado, para depois considerar o efeito conjunto.

O efeito da selecção no número esperado de esquemas numa população é particularmente fácil de determinar. Suponhamos que, num determinado instante de tempo t, existem m exemplares de um esquema particular H, contidos na população $A(t)$, denotados por $m(H, t)$. Durante o processo de selecção, uma cadeia é copiada para a nova população com probabilidade $p_i = f_i / \Sigma f_j$. , em que f é a função de adaptação (ou de avaliação)ao meio e j varia entre 1 e n, o número de elementos da população. Após escolher uma nova população de tamanho n, com reposição na primeira população, esperamos ter $m(H, t + 1)$ representantes do esquema H na nova população, dado por

$$m(H, t+1) = m(H, t) \cdot n \cdot \frac{f(H)}{\sum_i f_i} \tag{1}$$

em que $f(H)$ é a adaptação média das cadeias que representam o esquema H. Se reconhecermos que a adaptação média da população pode ser reescrita como

$$\bar{f} = \frac{\sum_i f_i}{n}$$

então podemos reescrever (1) como:

$$m(H, t+1) = m(H, t) \frac{f(H)}{\bar{f}} \tag{2}$$

Por palavras, isto significa que o número de ocorrências de um esquema particular cresce na razão entre o valor médio do esquema e o valor médio da população, ou de outra forma, esquemas com valor médio acima da média da população recebem um número crescente de cópias em cada geração, enquanto que esquemas com valor médio abaixo da média da população recebem um número decrescente de cópias em cada geração. Resolvendo a equação recorrente (2) chegamos a uma forma fechada que indica que os esquemas acima da média recebem um número de cópias que cresce exponencialmente, enquanto os esquemas abaixo da média recebem um número de cópias que decresce exponencialmente, com o número de geração:

$$m(H, t) = m(H, 0) \cdot (1 + c)^t \tag{3}$$

A constante c é o factor pelo qual a adaptação média do esquema H se mantém acima ou abaixo da média da população, ou seja, a razão entre a adaptação média de um esquema e a adaptação média da população.

Consideremos agora o cruzamento. O cruzamento é um processo estruturado mas com uma componente aleatória que faz com que ocorram trocas de informação entre cadeias. O cruzamento é capaz de criar novas estruturas com um mínimo de disrupção

da estratégia de distribuição de elementos ditada pela selecção. Isto resulta no crescimento (ou decrescimento) exponencial de alguns dos esquemas presentes na população.

Como já vimos, nem todos os esquema têm igual probabilidade de serem mantidos intactos durante o cruzamento. Quanto maior for o comprimento de um esquema, menor é a probabilidade de sobreviver a um cruzamento. Assumindo que o cruzamento ocorre em cada "acasalamento" com probabilidade p_C, a probabilidade de sobrevivência de um determinado esquema pode ser dado por:

$$p_s \geq 1 - p_c \frac{\delta(H)}{l-1} \tag{4}$$

Podemos agora considerar o efeito combinado dos operadores de selecção e cruzamento, assumindo que estes são independentes. Se esta condição se verificar, obtemos então uma estimativa para o número de representantes de um determinado esquema H que são esperados numa dada geração:

$$m(H, t+1) \geq m(H, t) \frac{f(H)}{\bar{f}} \left(1 - p_c \frac{\delta(H)}{l-1}\right) \tag{5}$$

Isto significa que a expressão obtida é a que tínhamos para o operador de selecção, pesada pela probabilidade de sobrevivência do esquema ao operador de cruzamento.

Por fim, consideremos a mutação. Para que um determinado esquema sobreviva, é necessário que sobrevivam todas as posições fixas que o constituem, ou seja, a sua ordem. Assim, se a mutação ocorrer com probabilidade p_m, a probabilidade de sobrevivência do valor de um gene é $(1 - p_m)$. Se as mutações nas várias posições forem estatisticamente independentes, um esquema sobrevive com probabilidade $(1 - p_m)^{o(H)}$, expressão que pode ser aproximada para $1 - o(H)\, p_m$ para valores pequenos de p_m. Podemos então concluir que um esquema recebe um número de cópias na geração seguinte, sob selecção, cruzamento e mutação, dado pela seguinte equação, que não é mais do que a expressão matemática do Teorema Fundamental:

$$m(H, t+1) \geq m(H, t) \frac{f(H)}{\bar{f}} \left(1 - p_c \frac{\delta(H)}{l-1} - o(H) \cdot p_m\right) \quad (6)$$

Com a introdução da mutação, as nossas conclusões não mudaram muito: esquemas curtos e de pequena ordem recebem um número de cópias exponencialmente crescente (decrescente) se a sua adaptação média for superior (inferior) à adaptação média da população. Este crescimento (decrescimento) é tanto maior quanto maior (menor) for a razão entre a adaptação média do esquema e a da população.

Isto significa que os algoritmos genéticos favorecem o processamento e recombinação de pequenos blocos de informação que provaram levar a um bom desempenho no passado. A este teorema, desenvolvido por John Holland, dá-se o nome de Teorema dos Esquemas, ou Teorema Fundamental dos Algoritmos Genéticos.

Evolução temporal da qualidade das soluções

Para termos uma ideia mais concreta da evolução da qualidade das soluções ao longo das gerações, podemos observar a figura seguinte. Esta mostra as soluções obtidas por um algoritmo genético num problema muito simples, em que a função de avaliação é simplesmente maximizar $f(x) = x$ no intervalo inteiro [0,65535], ou seja, com uma solução óptima de 65535:

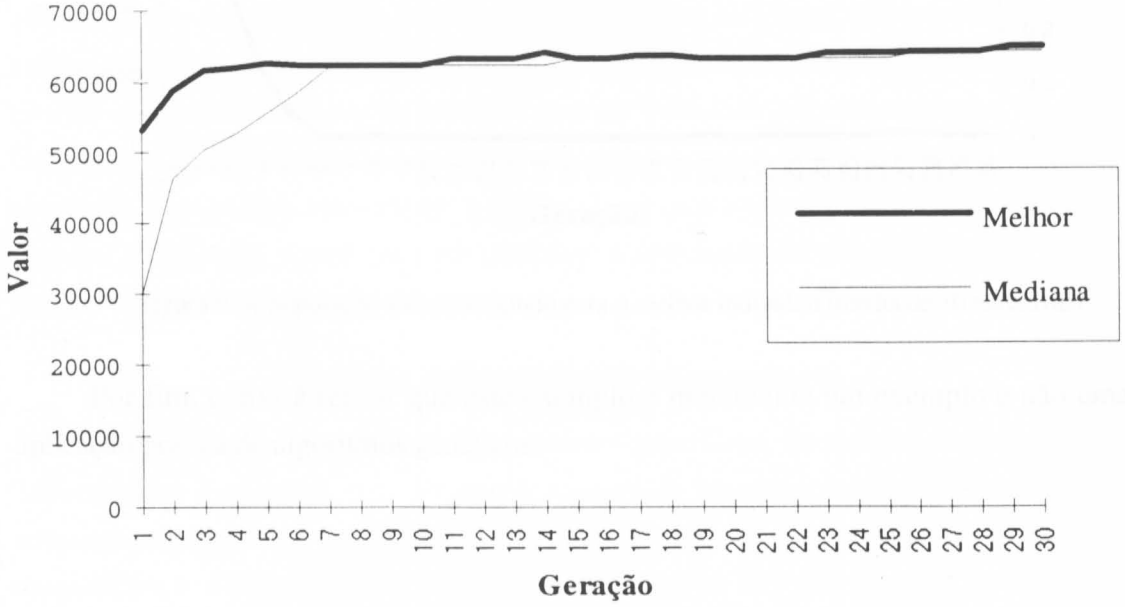

Figura 8. Evolução da solução num algoritmo genético (média de 10 tentativas)

No gráfico podemos ver 2 linhas de evolução, que correspondem à melhor solução e à mediana das soluções de cada geração. Pode ver-se claramente neste exemplo que a qualidade da solução obtida pelo AG vai melhorando e tendendo para um valor limite. Esta curva é típica dos resultados obtidos por AG, convergindo mais ou menos lentamente, com uma diferença mais ou menos acentuada entre o melhor valor e o pior, dependendo de diversos factores, alguns dos quais já mencionados. Os valores mostrados correspondem à média de 10 tentativas, para tentar atenuar o efeito aleatório, sempre presente num algoritmo deste tipo.

Nota-se também que a qualidade média da população, dada aqui pela mediana e não pela média, uma vez que encontramos valores muito díspares na população, alcança rapidamente o melhor da geração, ou seja, a população é rapidamente dominada pelos melhores indivíduos. Isto pode ser melhor observada na figura seguinte, em que se vê a evolução do intervalo inter-quartis normalizado, que é aqui usado como medida de dispersão em vez do desvio padrão, pelas razões apontadas para a mediana:

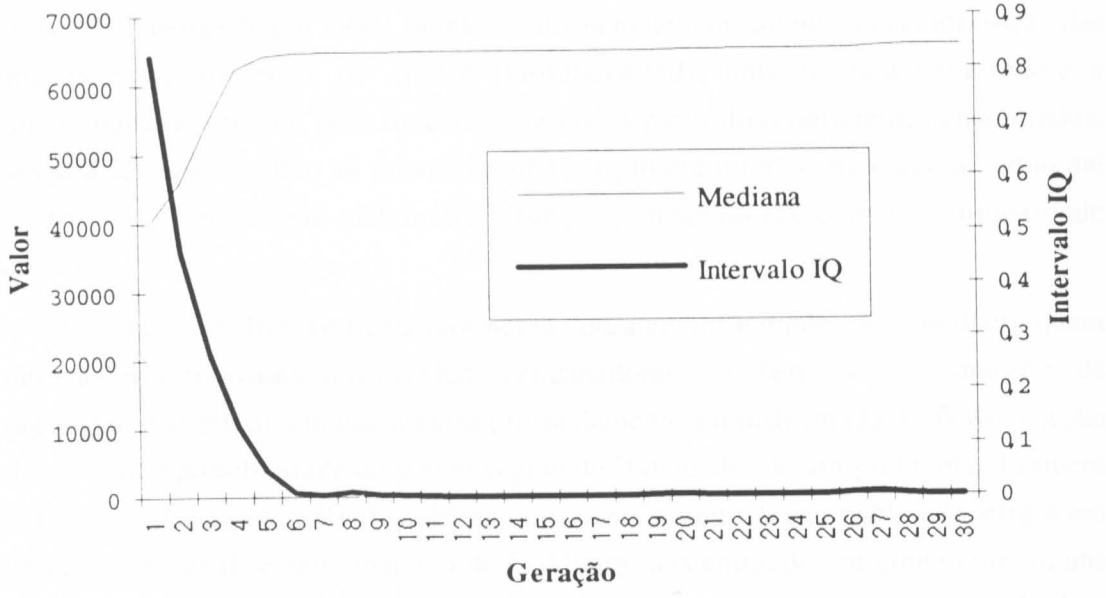

Figura 9. A população vai convergindo para o melhor indivíduo (média de 10 tentativas)

Por fim, convirá referir que este exemplo é meramente um exemplo e não uma aplicação prática de algoritmos genéticos.

4.3. Problemas enganadores

Na secção anterior vimos que os blocos se combinam por forma a conseguir juntar pedaços de duas soluções boas, na esperança de encontrar uma muito boa. No entanto, há problemas em que isto não se verifica. Embora sejam casos raros, a verdade é que existe um tipo de problemas, denominados na literatura como problemas *enganadores* (*deceptive*) [Goldberg91d], em que a combinação de bons blocos pode levar a soluções sub-óptimas.

Perspectiva histórica

Na verdade, os primeiros investigadores depararam com funções que faziam com que os AG convergissem para pontos que não eram óptimos globais. Numa primeira fase, foram tentadas soluções *ad-hoc* para cada problema e com mudanças da representação era possível superar o problema.

Mas a primeira abordagem genérica e formal veio da tese de doutoramento de Bethke [Goldberg89], em 1981. Bethke analisou matematicamente a decomposição das funções em *polinómios de Walsh* [Goldberg91d], uma técnica semelhante à transformada de Fourier, para funções binárias. A partir dos coeficientes encontrados, passou a ser possível não só prever se uma função era ou não enganadora, como até construir, de uma forma sistemática, funções enganadoras com a complexidade desejada.

O trabalho de Bethke ficou adormecido durante uns tempos e foi Goldberg quem voltou a pegar no assunto dos problemas enganadores. Goldberg escreveu uma série de artigos [Goldberg91d], em que analisa profundamente a transformada de Walsh e a sua relação com a possibilidade de prever o grau de "falsidade" de um problema. Também Grefenstette [Grefenstette91b] se debruçou sobre o assunto, tendo publicado artigos em que fazia uma análise semelhante à de Goldberg mas entrando em conta com alguns factores adicionais, o que fez com chegasse à conclusão de que Goldberg e Bethke tinham obtido resultados algo inexactos.

Análise dos problemas enganadores

Para fazermos uma análise, forçosamente superficial, do assunto, relembremos o Teorema dos Esquemas, que diz que o número de representantes de esquemas com adaptação média acima da média da população cresce exponencialmente com o número da geração, e a Hipótese dos Blocos, que diz que os AG são capazes de combinar blocos associados a boa adaptação, por forma a criar novos blocos, maiores e com

ainda melhor adaptação. Definamos, também, *adaptação média estática de um esquema* como sendo a média da adaptação de *todos* os elementos que constituem o esquema, independentemente de estarem ou não representados na população.

Segundo Goldberg, para que um AG consiga atingir o óptimo global, a Hipótese dos Blocos obriga a que os esquemas associados com o óptimo tenham uma adaptação média estática maior do que a dos esquemas competidores, nas suas diversas partições. Por exemplo, suponhamos que 000...0 é o óptimo para uma determinada função binária. Então, uma função *fácil* para os AG teria:

f(0*...*) > f(1*...*)

f(00*...*) > f(01*...*)

f(00*...*) > f(10*...*)

f(00*...*) > f(11*...*)

e assim por diante para cada partição do espaço de pesquisa. Pelo contrário, uma função que não obedeça a esta condição pode ser difícil de optimizar. Uma função deste tipo, em que os esquemas associados com o óptimo possuem adaptação média estática *inferior* à dos esquemas concorrentes, pode levar um AG para um ponto que não é o óptimo:

f(0*...*) < f(1*...*)

f(00*...*) < f(01*...*)

f(00*...*) < f(10*...*)

f(00*...*) < f(11*...*)

A este tipo de funções, é costume dar o nome de *enganadoras*.

Então, podemos pensar, aonde é que está a dificuldade? Basta calcular a adaptação média estática de cada esquema e verificar se a função é fácil ou difícil. Mas o problema reside precisamente aí: é que é muito difícil calcular a adaptação média estática de um esquema e é aí que a transformada de Walsh é útil.

A transformada de Walsh permite reescrever qualquer função binária sob a forma de uma soma pesada de polinómios, os polinómios de Walsh, que formam como que uma base canónica para a expressão de funções binárias. É possível analisar os coeficientes obtidos para estimar a adaptação média estática de um esquema.

Mas a verdade é que nem sempre a transformada de Walsh se revela como uma medida fiável para a predição da convergência dos AG, como o verificou o próprio

Goldberg. Há muitos problemas que são classificados como difíceis e que os Algoritmos Genéticos resolvem sem dificuldade. Segundo Grefenstette [Grefenstette91b], isto acontece pelo facto da Hipótese dos Blocos, que depende da adaptação média *estática*, não estar de acordo com o Teorema dos Esquemas, em que é considerada a adaptação média *observada*, ou seja, a adaptação média de cada geração.

Quanto maior for a discrepância entre a adaptação média estática (calculada com base em *todos* os possíveis representantes de um esquema) e a observada (calculada com base apenas nos elementos *presentes* na população), pior será a qualidade da previsão dada pela análise dos coeficientes de Walsh. A adaptação média observada pode variar muito, dependendo de factores críticos como o tamanho da população, a distribuição da função, etc. A sugestão de Grefenstette é de que se reveja a dificuldade das funções numa óptica *dinâmica* e não estática.

De qualquer das formas, não há dúvida que este é um dos campos fundamentais de investigação teórica, uma vez que nos ajuda a perceber os mecanismos menos aparentes dos AG.

5. Codificação e Paramerização

Neste capítulo, vamos falar sobre a *codificação* e *parametrização*, os dois problemas menos automatizáveis com que um investigador se depara, ao tentar aplicar AG a um novo problema. São aqueles que fazem com que a utilização de AG ainda possua características de arte. No entanto, mesmo estes problemas podem ter solução num enquadramento genético.

5.1. Codificação

Não restam dúvidas de que a questão da codificação, ou seja, o mapeamento dos parâmetros do problema num cromossoma, é fundamental para o sucesso duma aplicação de AG. A qualidade da solução final vai depender, de alguma forma, da escolha de uma representação que torne evidentes as semelhanças e diferenças entre os diversos elementos da população. Vale a pena referir, no entanto, que os AG não são, de modo nenhum, tão sensíveis à questão da representação como a maioria das técnicas clássicas de optimização. Sucede apenas que é possível arranjar representações mais ou menos eficientes.

Qualquer problema resolúvel em computador tem que ter a sua solução codificada como uma concatenação finita dos valores de vários parâmetros por forma a criar uma cadeia de informação [Hofstadter79], [Ullman81]. Essa é a base da teoria da computabilidade, desenvolvida por Alan Turing no início da era dos computadores. No caso dos AG, essa capacidade deve ser aproveitada para criar um cromossoma artificial, em que cada gene codifica um parâmetro e o cromossoma inteiro representa uma solução.

Boas codificações

Para determinar uma boa codificação do problema num cromossoma artificial, há que ter um conta alguns aspectos:

- Utilização de um alfabeto de cardinalidade (nº de caracteres) mínimo, preferencialmente o binário. Prova-se [Goldberg91g] que, quanto menor for a

cardinalidade de um alfabeto, maior é a quantidade de informação passível de ser utilizada eficientemente pelo algoritmo genético. A razão essencial é fácil de apreender, pelo menos intuitivamente: uma representação num alfabeto mínimo, desde que bem escolhida, provoca o aparecimento de maior número de semelhanças eventuais, como por exemplo 10 e 11 têm mais pontos em comum do que 2 e 3.

- Colocação dos genes de tal forma que o operador de cruzamento seja capaz de construir blocos compactos com significado e com um bom desempenho. Isto significa que as características que possam estar eventualmente relacionadas devem estar fisicamente próximas no cromossoma ou será necessário utilizar operadores mais avançados, como a *inversão*[7]. Este problema não é tão premente como parece, pois os AG são capazes de ultrapassar mesmo dificuldades deste tipo.
- Uso de uma representação que faça com que os operadores de mutação e cruzamento sejam "fechados", ou seja, que a utilização de qualquer um dos operadores sobre qualquer cromossoma deve produzir um cromossoma "legal". Esta restrição pode ser relaxada, penalizando fortemente [Pereira91] os indivíduos "impossíveis", ou seja, que violem restrições não relaxáveis[8], e deixando a selecção artificial fazer o seu papel. No entanto, se a criação de "nados-mortos" é muito frequente, o rendimento do AG pode baixar drasticamente.
- Utilização de codificações que preservem a semelhança entre conceitos semelhantes, como o faz, por exemplo, a codificação de Gray em oposição à representação binária pura. Na codificação de Gray, números adjacentes são representados por padrões binários que apenas diferem de um bit, evitando os chamados "penhascos de Hamming" [Davis91], ou seja, números adjacentes que possuem representações binárias totalmente diferentes, como por exemplo o 255 (01111111) e o 256 (10000000).

Seguindo estas indicações, é normalmente fácil encontrar uma representação para um problema numa cadeia finita de informação. No entanto, nem sempre é necessário proceder a "tentativas no escuro" para encontrar uma boa representação. Como veremos, a teoria existente permite conceber operadores especiais que alteram a própria representação, tornando-a parte do problema a resolver pelo AG e aliviando o utilizador dessa tarefa.

---

[7]Ver capítulo 7

[8]Cf. método "penalidade" de algoritmos mais tradicionais

5.2. Parametrização

Por outro lado, o desempenho do AG básico depende dos valores de três parâmetros de que já falámos: o tamanho da população, a probabilidade de cruzamento e a probabilidade de mutação. Após alguma investigação [Goldberg89], [Goldberg91a], é possível especificar alguns valores robustos para estas três variáveis:

- O tamanho da população não deve ser muito grande, pois demorará muito a convergir, nem muito pequeno, pois terá grandes hipóteses de sofrer do fenómeno denominado "deriva genética" [DeJong91b]. Este fenómeno, que provoca uma convergência prematura da população, é consequência de uma amostragem deficiente do espaço das soluções, ao gerar a primeira população. Assim, é normalmente aconselhado um tamanho de população entre 40 e 100 elementos para problemas de dimensão pequena, podendo atingir a casa dos 1000 para problemas muito grandes. Um valor típico andará nos 60 elementos.
- A probabilidade de cruzamento deve ser elevada, por forma a garantir um processamento rápido dos blocos presentes na população inicial. Por outro lado, uma probabilidade de cruzamento muito alta [DeJong91], [Spears91b], [Spears91] pode provocar muita *disrupção* dos esquemas, ou seja, incapacidade de propagar esquemas intactos para novas gerações. O valor óptimo depende da aplicação mas é aconselhado um valor da ordem dos 0.8.
- A probabilidade de mutação, pelo contrário, deve ser mantida baixa, ou o algoritmo tenderá para uma pesquisa aleatória [Goldberg89], o que acontece para a probabilidade de mutação igual a 0.5, independentemente de quaisquer outros parâmetros. Um valor correcto, sempre difícil de determinar, deverá andar na ordem de 0.01. Como referência, são obtidos bons efeitos quando, em média, um elemento de cada geração sofre uma mutação.

Os gráficos seguintes mostram como diferentes valores para os parâmetros referidos podem afectar a evolução de um algoritmo genético. Estes valores foram obtidos para a optimização da função $f(x) = x$, no intervalo inteiro [0,65535]. O valor óptimo é 65535.

Probabilidade de cruzamento

Começamos por olhar para a influência da probabilidade de cruzamento, considerando uma probabilidade de mutação constante e igual a 0.001, um valor muito baixo:

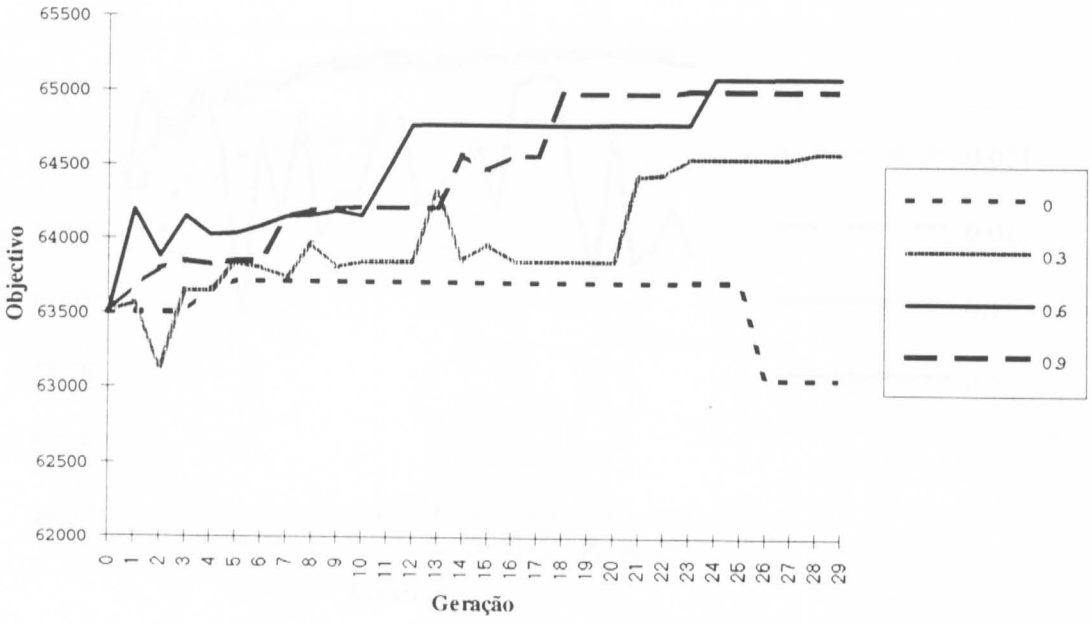

Figura 10. Influência da probabilidade de cruzamento na evolução

Atentando na figura anterior, podemos verificar que existe uma grande diferença de comportamento do algoritmo para diferentes valores de probabilidade de cruzamento. Em geral, quanto maior esta é, maior é a facilidade de recombinação dos blocos e mais rapidamente o algoritmo converge para a solução. No entanto, um valor muito elevado ($p = 0.9$) é pior que um valor médio ($p = 0.6$), pois começa a provocar muita disrupção. Por outro lado, se omitirmos o operador de cruzamento completamente ($p = 0.0$), a exploração de novas soluções fica totalmente a cargo da mutação, o que provoca o "oscilar" da qualidade da solução.

Probabilidade de mutação

Quanto à mutação, também ela interfere bastante na qualidade da solução final e no evoluir das populações. Afinal de contas, é o operador responsável por restaurar a diversidade, quando o domínio dos melhores indivíduos é provocado pela selecção e pelo cruzamento. No gráfico seguinte, podemos observar como é que a probabilidade de mutação interfere com um AG, mantendo a probabilidade de cruzamento constante:

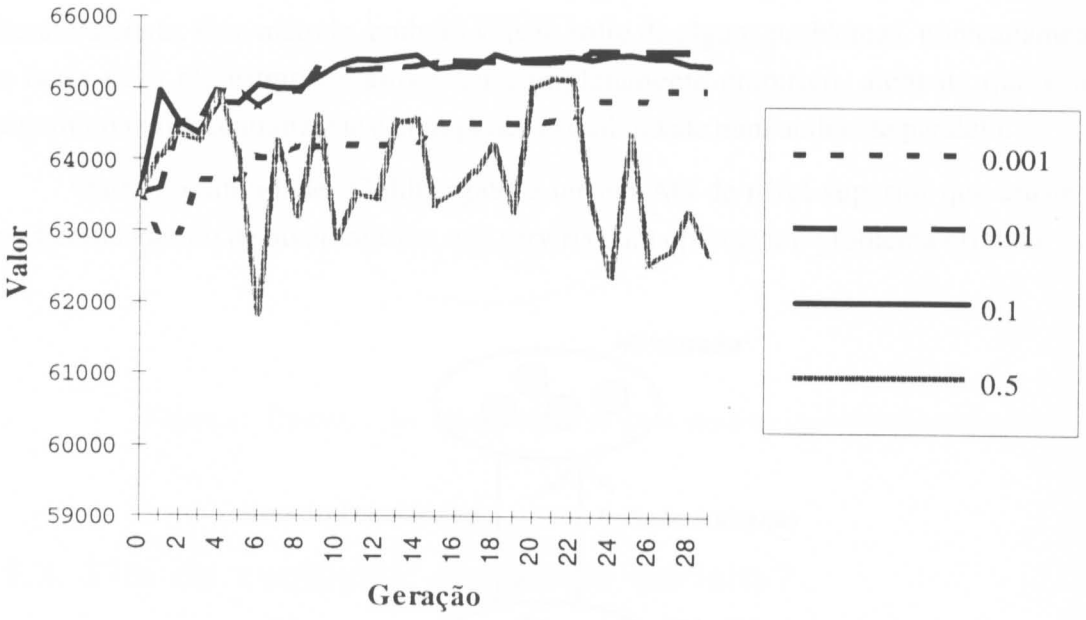

Figura 11. Influência da probabilidade de mutação na evolução

Também aqui os dados experimentais estão de acordo com a teoria: quantidades baixas (p = 0.01-0.1) de mutação fazem com que o operador cumpra perfeitamente o seu papel de "seguro de vida", restaurando blocos eliminados por uma selecção zelosa de mais e mantendo a diversidade da população. Pelo contrário, valores muito altos (p ~= 0.5) fazem com que a pesquisa genética se transforme em pesquisa aleatória, quase que "destruindo" o trabalho dos outros operadores genéticos. Um valor muito baixo (p = 0.001) também prejudica ligeiramente a *performance*, pois não permite a restauração do material genético perdido.

Estes gráficos foram obtidos pela aplicação dos AG ao nosso problema já conhecido, a maximização de $f(x) = x$ no intervalo inteiro [0,65535] e representam a média de 10 tentativas.

Ajuste automático das probabilidades

As figuras que vimos mostram que a escolha correcta dos parâmetros pode influir decisivamente no rendimento de um AG. Foram propostas algumas soluções para a automatização do ajuste das probabilidades, para evitar uma abordagem do tipo "tentativa e erro".

Uma das mais simples [Davis91] é ir monitorizando a evolução da adaptação e as alterações provocadas pelos diversos operadores e ajustar as probabilidades iterativamente. Este método, embora viável, sofre de alguns problemas, nomeadamente o facto de o algoritmo de ajuste ser completamente empírico, além de que é um algoritmo muito centralizante e com pouca exquibilidade num ambiente paralelo.

Outra possibilidade [Goldberg89] é ter um AG de nível superior que ajuste as probabilidades do de nível inferior, que serviria para optimizar o problema original:

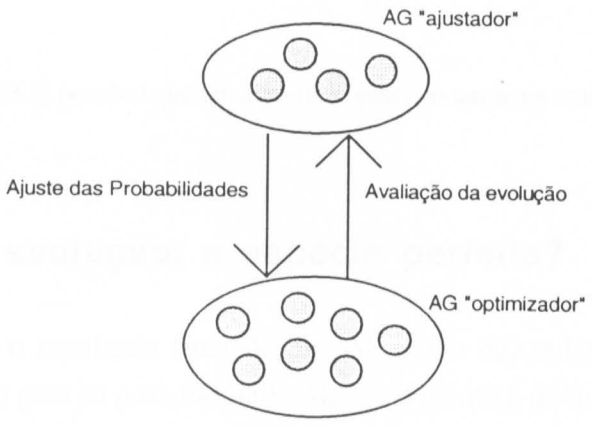

Figura 12. Ajuste dos parâmetros genéticos por um algoritmo genético

Esta opção parece lógica, uma vez que a optimização do valor das probabilidades é um novo problema de optimização. No entanto, esta solução sofre de um problema: o AG "de ajuste" também tem parâmetros para ajustar. Seria necessário um novo AG para controlar o controlador e o ciclo repete-se.

Mas a ideia de utilizar um AG para controlar as probabilidades de cruzamento e de mutação pode ser aproveitada, de um modo ainda mais elegante, codificando estes parâmetros no cromossoma e deixá-los sujeitos a evolução, exactamente como os parâmetros naturais do problema. Os AG passam a optimizar simultaneamente o problema original e o ajuste das probabilidades. Esta abordagem foi já tentada com sucesso por diversos investigadores [Goldberg89].

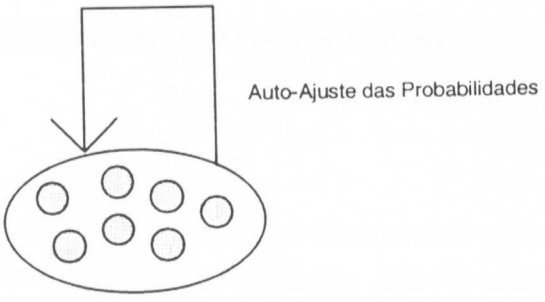

Figura 13. É possível que um algoritmo genético ajuste os seus próprios parâmetros

5.3. Fim da evolução: a espécie perfeita?

Vimos que o resultado final de um AG pode depender em grande parte dos valores escolhidos para as probabilidades de cruzamento e de mutação. Mas falta ainda pensar num ponto: quando é que o AG chega a um resultado final?

Na utilização de AG para problemas reais temos que estabelecer algum critério de paragem. Existem vários critérios para determinar um ponto útil de paragem e os mais usados são os seguintes:

- Terminar ao fim de um determinado número de gerações ou ao fim de um determinado tempo limite. Esta opção é boa para ambientes em tempo real, em que interessa não só a qualidade da solução, mas também o instante de tempo em que esta é encontrada. Os sistemas em tempo real são um campo com óptimas perspectivas de aplicação dos AG.
- Terminar quando a melhor solução encontrada até ao momento é melhor do que um determinado limite de qualidade que é imposto pelo utilizador.
- Terminar quando a população tiver convergido, ou seja, quando a percentagem de indivíduos iguais for maior que um determinado limite. Algumas das experiências feitas pelo autor basearam-se em parar a simulação quando o desvio padrão das adaptações da população diminui abaixo de determinado valor ou quando a média ou mediana da população não sofre grande alteração durante várias gerações consecutivas. Este tipo de convergência enfatiza a convergência *fenotípica*, ou seja,

o valor das soluções, e não a convergência *genotípica*, ou seja, a sua estrutura interna.

Evidentemente, é possível usar estas regras em acumulação.

6. Selecção e cruzamento revisitados

Neste capítulo, vamos rever as nossas noções sobre os dois principais operadores genéticos e ver como é possível expandi-los por forma a serem mais flexíveis e mais eficientes. Vamos ver que há várias possibilidades de implementação para o operador de selecção e vamos discutir qual á a melhor. Veremos também como é possível generalizar o operador de cruzamento por forma a minimizar a possibilidade de disrupção dos esquemas.

6.1. Selecção

A selecção é o operador responsável por fazer com que a qualidade média da população vá evoluindo, mesmo que não haja cruzamento nem mutação. Até agora, falámos unicamente numa implementação possível da selecção, nomeadamente a *roleta viciada*, algoritmo que foi analisado já num capítulo anterior. No entanto, é possível implementar o conceito de selecção com algoritmos muito diferentes, cada um com as suas características. Nesta secção vamos expor brevemente cada um dos tipos principais de implementação e por fim compará-los quanto a alguns parâmetros.

Reprodução proporcional

O nome "reprodução proporcional" descreve um grupo de funções de selecção que escolhe indivíduos, para colocar na geração seguinte, de acordo com a sua adaptação. Para estes algoritmos, a probabilidade de selecção do elemento i, é igual a f_i / f, em que f_i é a sua adaptação e f é a adaptação média da população. Já vimos, no capítulo 4, que este facto leva a um (de)crescimento geométrico, com factor de progressão f_i / f. Isto quer dizer que os "bons" elementos recebem cada vez mais cópias, enquanto que os "maus" elementos recebem cada vez menos cópias (ver equação 2, capítulo 4).

Com uma solução não recursiva [Goldberg91c] para esta equação é possível calcular o *tempo de convergência* (o tempo em que a proporção do melhor indivíduo na população atinge $(n - 1) / n$), e que é aproximadamente $\mathbf{O(n \ln n)}$.

Uma implementação possível de um algoritmo de reprodução proporcional é a roleta viciada. A sua complexidade temporal é má: no pior caso poderá ser $\mathbf{O(n^2)}$ para gerar uma nova população com n elementos e, na melhor das hipóteses, $\mathbf{O(n \log_2 n)}$, se forem utilizadas estruturas de dados mais sofisticadas.

Uma outra implementação da reprodução proporcional é a *selecção estocástica pelo resto*. Neste método, o número de cópias que um elemento terá na nova população é calculado deterministicamente como a parte inteira do quociente f_i / f. Evidentemente, só por muita sorte é que a nova população terá todos os elementos preenchidos deste modo. Os elementos que faltam podem ser determinados de duas maneiras: com reposição, em que a parte fraccionária do quociente é usado como peso para uma moeda viciada que determina se um indivíduo recebe mais uma cópia, ou sem reposição, em que as partes fraccionárias dos quocientes são usadas para implementar uma roleta viciada. A complexidade temporal da primeira abordagem (com reposição) é **O(n)**, pois pode ser calculada numa só passagem e a parte probabilística termina normalmente em **O(1)** passos. A segunda abordagem tem a mesma complexidade temporal da roleta viciada, uma vez que a primeira passagem é **O(n)**.

Por fim, temos a *selecção estocástica universal*, em que uma roleta viciada é rodeada com marcas equidistantes e é feita uma rotação da roleta. Cada indivíduo recebe tantos descendentes quantas as marcas que ficam em cada ranhura (de tamanho proporcional à adaptação). A complexidade temporal deste algoritmo é **O(n)**.

Selecção por posição (*rank*)

Nesta abordagem, os indivíduos são ordenados pela sua adaptação e o número de descendentes é dado por uma função da sua *posição*. Claro que esta função de atribuição de descendentes não pode ser qualquer, tendo que obedecer a algumas condições [Goldberg91c]. Essas condições têm a ver, essencialmente, com a existência de uma atribuição de descendentes para qualquer posição e com a necessidade de atribuir todos os elementos da nova população, i.e., o número total de elementos a atribuir terá que ser igual ao número de elementos da população.

Foram tentadas várias funções de atribuição, sendo mais usada a linear, em que o número de descendentes cresce inversa e linearmente com a posição. Com uma função deste tipo, a solução não recursiva para o problema da convergência tem um tempo de convergência de aproximadamente:

$$t^* = 2 \ln (n - 1)$$

A complexidade temporal deste método também não é muito boa, uma vez que uma implementação simples terá complexidade $\mathbf{O(n^2)}$, enquanto que uma implementação sofisticada não poderá ser melhor que **O(n log n)**.

Torneio

A ideia de *torneio* é talvez a mais simples e uma das mais eficientes [Goldberg91c], [Goldberg91]. O torneio mais simples é o *binário*, em que para cada elemento novo a gerar, pegamos em dois elementos à sorte e tomamos o melhor com uma probabilidade p qualquer, desde que $.5 < p \leq 1$. É possível estender esta noção a torneios de ordem superior e, de facto, para muitos problemas o número ideal anda perto de torneios de ordem 4.

O tempo de convergência é da forma:

$$t^* = 1/(\ln n)(\ln n + \ln(\ln n))$$

A complexidade temporal deste algoritmo é muito favorável: **O(n)**. Além disso, presta-se muito bem a implementação paralela, com muito pouca comunicação global, ao contrário dos outros métodos (cálculo da adaptação média, por exemplo). Para um torneio de ordem p, temos apenas ligações entre p processadores[9] adjacentes, possivelmente com algumas trocas esporádicas de elementos entre processadores não adjacentes.

Além disso, se o método de escolha e de "combate" dos elementos for ligeiramente alterado, podemos obter uma selecção por *torneio de Boltzman*, que é equivalente [Goldberg91] a uma versão paralela do método de optimização *simmulated annealing* [Kirkpatrick83]. Neste método, provavelmente com grande futuro, a probabilidade de vitória do melhor elemento não é fixa, dependendo da razão entre as adaptações dos elementos em combate e do número da geração. Estas duas dependências permitem mapear nos AG os conceitos do algoritmo de *simmulated annealing*, ou seja, o sistema pode passar para um estado de energia mais elevada dependendo de duas condições: diferença entre os níveis de energia (razão entre as adaptações dos elementos em combate) e "temperatura" do sistema (probabilidade de vitória do melhor aumentando com o número da geração).

---

[9]Se cada processador simular um indivíduo. Se cada processador simular uma pequena população, então o número de ligações é ainda menor [Bessière92]

Comparação

A comparação entre os vários métodos é difícil, pois é uma escolha que involve diversos parâmetros contraditórios, e que são basicamente os seguintes:

- Eficiência temporal
- Taxa de convergência
- Optimalidade
- Sequencialidade
- Plausibilidade biológica

A eficiência temporal de um operador não é muito importante na maioria dos casos, uma vez que é normal que o tempo gasto na avaliação dos indivíduos seja superior, em uma ou mais ordens de grandeza, ao tempo gasto nos operadores genéticos. No entanto, se tivermos a eficiência em conta, o torneio e a selecção estocástica universal parecem ser os melhores, uma vez que conseguem uma complexidade **O(n)**, enquanto que os outros métodos apenas conseguem **O(n log n)**.

Quanto à taxa, ou velocidade, de convergência, essa está relacionada profundamente com o parâmetro seguinte, a optimalidade. De facto, verifica-se que os algoritmos que produzem uma melhor taxa de convergência são também aqueles que são mais facilmente atraídos para pontos não óptimos, uma vez que não conseguem manter a diversidade da população. Por outro lado, os métodos proporcionais são métodos muito sensíveis à relação entre as adaptações dos diversos elementos da população. Isto significa que, para o fim de um AG, quando só temos praticamente bons indivíduos, os algoritmos proporcionais são incapazes de manter a competitividade entre os elementos[10], o que faz com que convirjam prematuramente [Goldberg89]. O torneio e a selecção por posição são mais imunes a este problema e conjugam tempos de convergência aceitáveis com convergência mais fácil para um óptimo global.

Alguns destes algoritmos são notoriamente mais fáceis de implementar em paralelo. Qualquer algoritmo que necessite de informação global (como a adaptação média da população, por exemplo) está condenado a não ser muito eficiente quando implementado numa arquitectura paralela, pois o *overhead* de comunicações entre processadores não adjacentes é normalmente proibitivo. Os métodos proporcionais e os

---

[10]As ranhuras da roleta viciada tendem a ser praticamente todas do mesmo tamanho, tornando a selecção em escolha aleatória

métodos de posição são algoritmos sequenciais, deixando como "vencedor" o torneio, que apenas necessita de informação local.

E, por fim, um parâmetro que não é determinante mas que, do ponto de vista conceptual, pode ser importante, a plausibilidade biológica. Também neste campo, o torneio parece ser o vencedor. Na realidade, a capacidade de reprodução de um elemento de uma população natural não é determinada pela população inteira mas apenas pelos indivíduos mais próximos. Aliás, muitos animais usam torneios de ordem 2 (duelos) como forma de conseguir a hipótese de acasalar e assim transmitir os seus genes à nova população, o que só vem confirmar a nossa intuição de que o torneio é o método mais plausível, biologicamente.

Por isto tudo, parece-nos que o torneio, em qualquer das suas variantes, é o melhor método de selecção que podemos usar. E o facto de o torneio de Boltzman poder "herdar" os resultados analíticos obtidos para o *simmulated annealing*, no que diz respeito a optimalidade, convergência, etc., é mais um ponto a seu favor.

6.2. Cruzamento

O cruzamento é o operador que permite a recombinação de blocos constituintes, que aparecem nos elementos com melhor desempenho, e a formação de indivíduos que partilhem boas características de cada um dos pais. E o operador de cruzamento mais simples que se pode imaginar, o que vimos até agora, dá bons resultados na maioria dos casos.

Mas este operador é muito *assimétrico*, como iremos ver, e alguns autores preconizam a sua rejeição [DeJong91], trocando-o por outros mais equilibrados.

Disrupção

Para termos uma ideia da utilidade abstracta de um operador de cruzamento, vamos definir duas probabilidades. A primeira é a *probabilidade de disrupção de um esquema*. Esta probabilidade já foi analisada antes, no capítulo sobre os fundamentos matemáticos dos AG e tínhamos chegado à conclusão que, para o operador de cruzamento convencional, ela depende do *comprimento* do esquema: quanto maior é o esquema, maior é a probabilidade de ser destruído, ou cortado. Esta probabilidade é definida, assim, em função do comprimento do esquema.

A segunda é uma espécie de "resumo" da primeira e é denominada *probabilidade média de disrupção*, que nos dá uma medida de como, de uma forma geral, o operador de cruzamento vai actuar em relação aos esquemas. No caso do cruzamento convencional, com um só ponto, esta probabilidade média é baixa, uma vez que temos mais esquemas curtos do que compridos.

Cruzamentos alternativos

O facto de o operador de cruzamento tradicional ser "viciado" para esquemas pequenos pode trazer-lhe alguns dissabores, principalmente quando a ordenação dos genes no cromossoma não é a melhor.

Uma abordagem alternativa seria a de criar operadores com mais pontos de cruzamento, o que foi tentado [Goldberg89] por diversos investigadores, com 2, 3 ou mais pontos de corte. Levando esta teoria ao limite, teríamos um cruzamento *uniforme*, ou seja, poderíamos, em cada gene, escolher qual dos pais (com probabilidade de 50%) doaria o material genético ao filho. Isto seria equivalente a ter tantos pontos de cruzamento quantos os genes, ou posições do cromossoma.

Esta abordagem foi tentada anteriormente mas sem sucesso, uma vez que a probabilidade média de disrupção é muito alta e é inaceitável na maioria dos casos, embora se distribua uniformemente pelos diversos comprimentos de esquema, ou seja, não é dependente do comprimento. Mas é possível combinar o melhor de cada uma das abordagens [Spears91], [Spears91b], [Spears91c], se considerarmos outro parâmetro importante: a probabilidade de um pai ceder o seu material genético a um filho, que tínhamos fixado arbitrariamente em 50%. Se deixarmos variar este valor, demonstra-se que, para valores próximos de 0.1, obtemos disrupção uniforme e com probabilidade média inferior à dada pelo cruzamento de um só ponto.

Isto significa que, usando cruzamento uniforme, o utilizador do AG está mais livre para escolher uma representação, uma vez que será mais fácil combinar os blocos importantes, mesmo que estes estejam longe no cromossoma. E, de um ponto de vista intuitivo, esta abordagem está mais próxima do que acontece no cruzamento natural, uma vez que os cromossomas dos filhos são constituídos à custa de grande número de pedaços dos cromossomas dos pais e não apenas por duas partes.

Mas toda esta análise está a partir de um pressuposto, o de que o cromossoma é uma cadeia simples de bits, o que pode não ser verdade em representações mais complexas. Por exemplo, e como iremos ver mais adiante, os problemas baseados em ordem não podem ser tratados por um cruzamento normal, se o cromossoma

representar linearmente a ordem de uma solução. Pode ser necessário, pois, inventar novos operadores de cruzamento, mais adequados ao problema. A única restrição é que o operador deve poder combinar partes importantes dos cromossomas, por forma a criar elementos com melhor adaptação. Evidentemente, tal generalização deve ser efectuada também para mutação, se tal for necessário.

7. Utilização avançada

Até agora, falámos sempre do AG canónico, i.e., usando cromossomas lineares binários, com tamanho fixo e alteráveis por três operadores, a selecção, o cruzamento e a mutação. Na realidade, há problemas em que é vantajoso usar mais alguns operadores, que existem também na evolução e reprodução naturais.

Por outro lado, embora os cromossomas binários possam representar uma solução para qualquer problema [Ullman81] resolúvel por computador, há problemas que se encaixam melhor noutro tipo de representação, mais estruturada.

Neste capítulo vamos olhar para algumas extensões e generalizações do próprio conceito de AG.

7.1. Recessão e Domínio

É vulgar encontrar, na Natureza, cromossomas *diplóides*, ou seja, que têm duas cópias das mesmas características, possivelmente com valores diferentes para o mesmo gene. Esta estrutura dá origem a que, quando da expressão física das características representadas, seja usado um mecanismo de *recessão* e *domínio*, por forma a eliminar a ambiguidade de dois valores para o mesmo gene.

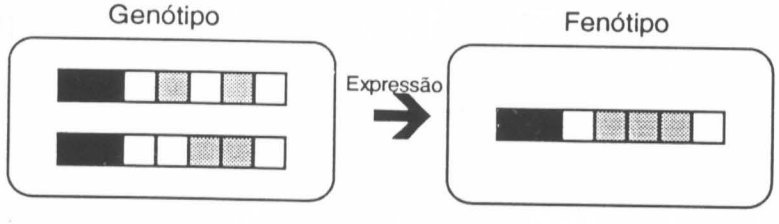

Figura 14. Um cromossoma diplóide e o fenótipo que resulta do domínio dos genes mais escuros

A necessidade do uso desta abordagem em algoritmos genéticos ainda não foi muito sentida mas foi proposto por Holstien e Holland [Goldberg89] que seria mais

fácil modelizar funções não estacionárias (variáveis no tempo) recorrendo a mecanismos de domínio e recessão. Um modo fácil de obter este efeito seria os cromossomas passarem a ser compostos de um alfabeto triplo {-1, 0, +1}, com -1 e +1 a representarem 1 e +1 a dominar 0, que domina -1.

Este esquema simples foi suficiente para que fosse demonstrado, por Goldberg e Smith, em 1987, a utilidade da recessão e do domínio na modelização de funções não estacionárias periódicas, embora não seja de prever que esta técnica possa ter uma aplicabilidade generalizada.

Os cromossomas diplóides constituem uma espécie de "regra de segurança" que diz que, se um valor de um gene é útil agora, então poderá vir a ser útil num futuro próximo. Esta "opção" da natureza permite que uma população seja muito mais rápida a reagir a um ambiente periódico, pois o valor alternativo para o gene está em "reserva", pronto a actuar pela mão da selecção.

7.2. Partilha

A necessidade de introduzir a noção de *partilha*, ou seja, o facto de elementos semelhantes dependerem de recursos semelhantes, surgiu de um facto simples: se não for tomada alguma medida, o esquema simples que demos para a construção de uma roleta viciada não funciona muito bem na fase final do processo de optimização [Goldberg91c], quando quase todos os indivíduos têm uma adaptação semelhante. Neste caso, começa a ser difícil distinguir os diferentes elementos, pois as "casas" da roleta são quase todas do mesmo tamanho. Isto tem como resultado um rápido crescimento das melhores cadeias, "abafando" o resto da população e fazendo com que o algoritmo convirja para um óptimo local, em vez de terminar num óptimo global [Goldberg89].

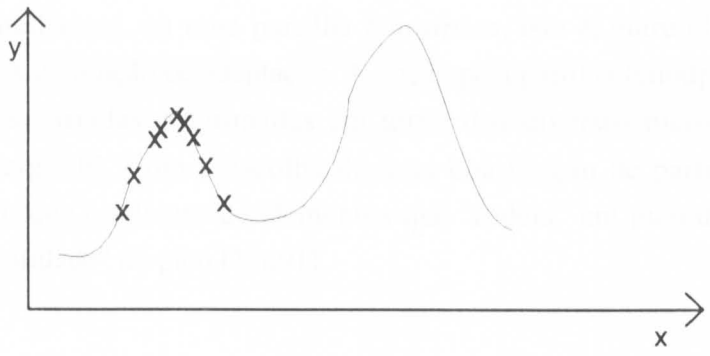

Figura 15. Convergência para um mínimo local (x = indivíduo)

Uma hipótese para obviar a este problema é "ajustar" a função de adaptação com o desenrolar do processo de convergência, fazendo com que no fim do processo as diferenças entre as adaptações sejam "ampliadas". No entanto, esta opção sofre de alguns problemas, sendo o menor deles o facto de ser necessário, *a priori*, determinar como fazer variar o factor de escala da função. Bastante pior é que esta abordagem provoca um "estrangulamento série" que pode fazer perigar a implementação paralela de um algoritmo genético, além de não possuir credibilidade natural.

Em vez disso, é possível fazer com que a adaptação de um indivíduo seja pesada pelo grau de *partilha* que ele tem com outros elementos [Goldberg89]. Esta opção tem uma contrapartida na Natureza: uma espécie não pode ver aumentar indefinidamente os seus membros, uma vez que a quantidade de recursos é limitada. Assim, existe pressão para que se desenvolvam várias espécies para ocupar "nichos" de sobrevivência, evitando a luta pelos mesmos recursos.

Em AG, uma implementação possível é fazer reflectir negativamente, na função de adaptação, a quantidade de elementos iguais: quanto maior for o número mais penalizado é o indivíduo. Esta pressão provoca uma amostragem maior do espaço de estados, uma vez que zonas muito densamente povoadas tendem a ter um valor mais baixo. Assim, há zonas do espaço de estados que se revelam promissoras, mesmo que aparentemente o não sejam. É possível utilizar técnicas [Goldberg89], designadas por "*crowding*", que não sofrem de estrangulamento série, para o cálculo da adaptação corrigida pelo factor de partilha. Estas técnicas baseiam-se em amostragem aleatória da população.

É possível definir uma partilha *genotípica*, ou seja, de elementos com cromossomas semelhantes, ou uma partilha *fenotípica*, isto é, entre elementos com valores semelhantes da função de adaptação. A opção pela partilha fenotípica dá origem a populações mais variadas e agrupadas em torno dos diversos picos da função a optimizar [Goldberg91b]. Com a escolha de uma boa função de partilha é mesmo possível fazer com que o número de elementos que "rodeia" um pico da função seja proporcional à "qualidade" do pico [Yin91].

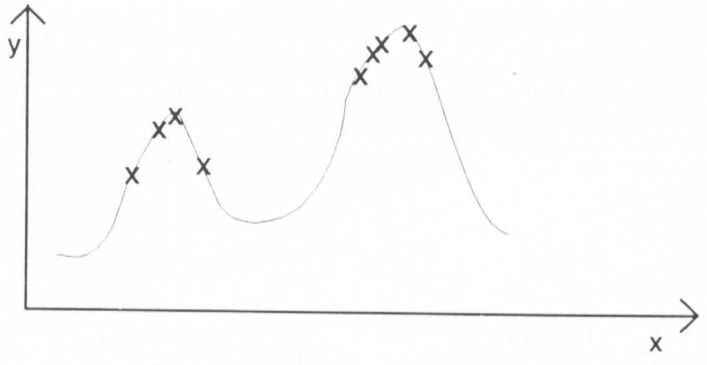

Figura 16. Se houver partilha, a população distribui-se pelos picos da função

7.3. Inversão

Como vimos em capítulos anteriores, os algoritmos genéticos processam essencialmente pequenos blocos contíguos de informação, os esquemas. Mas o que é que acontece quando a solução de um problema involve a correlação de informação que está situada em locais distantes do cromossoma? Na realidade, o que acontece é que os algoritmos genéticos podem ter alguma dificuldade em determinar o ponto óptimo. Isto é um fenómeno raro, o que significa que os AG conseguem mesmo "passar por cima" de codificações que aparentemente não potenciam a criação de blocos [Goldberg89], [Grefenstette91b].

No entanto, se alterarmos um pouco o modo como concebemos os cromossomas, é possível utilizar uma técnica, a *inversão*, que permite reordenar os genes, optimizando a representação escolhida. Para tal, basta pensar que a função do gene pode deixar de estar ligada ao local aonde este se encontra no cromossoma. Assim, basta rotular cada gene com a sua função e aquele passa a poder estar situado em qualquer posição, sem ambiguidade:

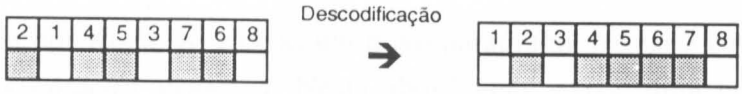

Figura 17. É possível separar a função do gene da sua localização no cromossoma

A seguir, fazemos entrar em cena o operador de *inversão*, cuja missão é reordenar os genes dentro do cromossoma. O seu funcionamento é muito simples: é escolhido um ponto de corte à sorte, inverte-se um dos pedaços do cromossoma e volta-se a "colá-lo" no mesmo sítio:

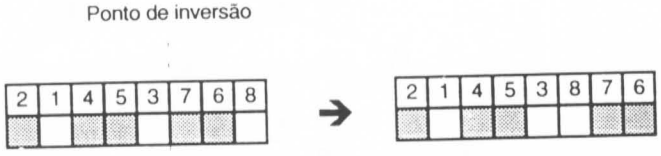

Figura 18. O operador de inversão reordena os genes dentro do cromossoma

Claro que é necessário construir operadores de cruzamento mais sofisticados, normalmente generalizações do cruzamento simples que expusemos em secções anteriores. Estes operadores encontram eco na Natureza: a fase de cruzamento natural é bastante complexa, passando pelo emparelhamento e alinhamento dos cromossomas a recombinar e posterior troca de informação.

Uma generalização da inversão, os algoritmos genéticos "desordenados" ("messy") foram propostos por Goldberg [Goldberg90] e as investigações em curso indicam que esta poderá ser uma abordagem que é capaz de, em simultâneo, optimizar o problema em questão, bem como a sua representação numa cadeia de bits. De qualquer forma, é difícil arranjar problemas em que o efeito da inversão seja notado, para lá de meras flutuações estatísticas.

7.4. Programação Genética

Foi sugerido por Koza [Koza91a] um novo paradigma baseado em algoritmos genéticos, a *programação genética*. Nesta abordagem, existe uma população de programas que é feita evoluir por forma a conseguir um programa que obedeça a determinadas especificações.

Cada programa é representado por uma estrutura em árvore, como na linguagem LISP: cada nó da árvore representa uma função, ou *não-terminal*, e cada folha representa uma constante ou variável, ou *terminal*. A população inicial é construída pela criação aleatória de árvores, compostas dos terminais e não-terminais listados pelo utilizador e que são relevantes para a modelização do domínio em causa.

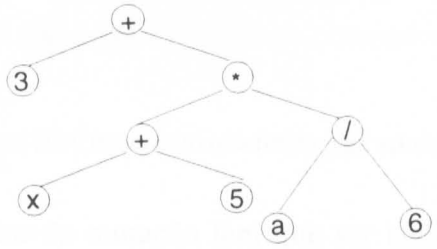

Figura 19. Representação, em LISP, da expressão (x+5)*a/6+3

Os operadores de cruzamento e mutação são redefinidos para actuarem sobre uma estrutura hierárquica e não sobre uma estrutura linear. Assim, o operador de cruzamento começa por escolher dois pontos para o corte, um em cada árvore que participa na operação. As sub-árvore assim obtidas são então trocadas, como se pode observar na figura seguinte:

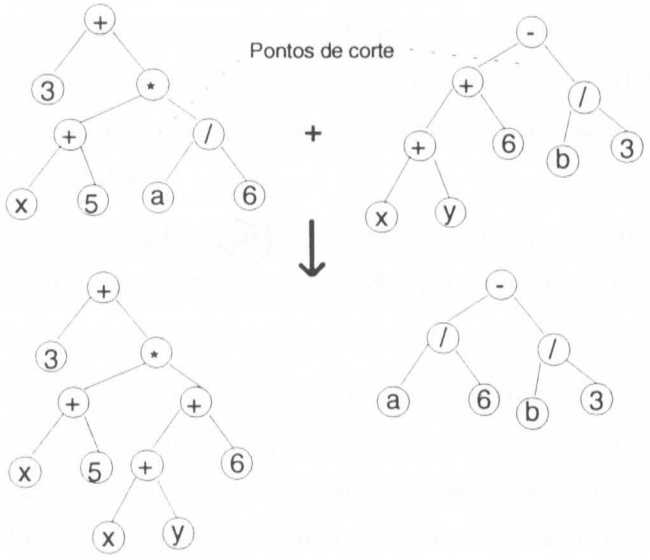

Figura 20. Cruzamento redefinido para actuar sobre árvores

Também o operador de mutação tem que ser alterado para poder lidar com árvores. A mutação consiste em alterar um nó da árvore, substituindo um não-terminal por outro ou um terminal por outro terminal. No caso dos não-terminais, é necessário ter cuidado para que o número de argumentos coincida com a sua aridade: por exemplo, se um nó dominado pelo operador unário **seno(x)** fosse mudado para um operador binário, como o +, faltaria um argumento. Há mais do que uma abordagem para resolver este assunto, embora a mais simples seja cada operador ignorar os argumentos em excesso, quando diminui a aridade do operador, e a geração de argumentos aleatórios, quando a aridade aumenta.

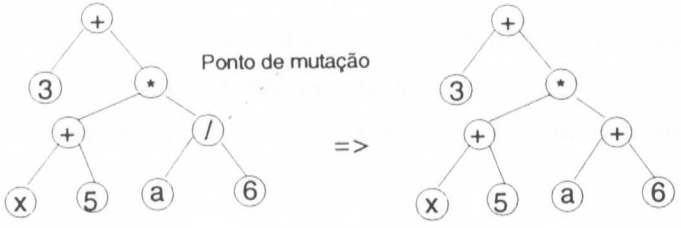

Figura 21. Mutação redefinida para actuar sobre árvores

Para evitar que os cruzamentos se dêem essencialmente nas folhas, Koza propôs que a probabilidade de cruzamento não seja distribuída uniformemente pelos nós e folhas mas sim assimetricamente: 80% de probabilidade de efectuar um cruzamento num nó e apenas 20% de o efectuar numa folha. Isto faz com que o cruzamento cumpra o seu papel de recombinação e não degenere em simples mutação, o que acontece quando actua ao nível das folhas.

A avaliação de cada indivíduo consiste, normalmente, na execução do programa que ele representa, para um determinado número de casos de teste e na acumulação dos erros.

Os resultados obtidos por Koza são verdadeiramente surpreendentes, principalmente na área de controlo óptimo, regressão simbólica, programação automática ou ainda na criação de comportamentos globais emergentes de comportamentos celulares (formigas/formigueiro, células/organismo, etc.).

7.5. Permutações

Existe uma série de problemas, que se podem definir como sendo "baseados em permutações", que não são triviais de resolver através do algoritmo genético canónico. Este tipo de problemas é muito comum no dia-a-dia e involve encontrar a melhor ordenação para um conjunto de coisas. Problemas típicos de optimização que se encaixam neste enquadramento são: *job-shop*, caixeiro viajante, coloração de mapas, etc.

O problema com o algoritmo canónico surge por causa da representação: para uma optimização baseada em ordem, uma representação possível seria cada cromossoma indicar a ordem pela qual as tarefas são executadas. Assim, o cromossoma (1 3 2 4) indica a ordem de execução 1, 3, 2, 4, enquanto o cromossoma (3 2 1 4) indica a ordem de execução 3, 2, 1, 4. Acontece que esta representação simples não sobrevive sequer a um simples cruzamento: se este ocorresse, por exemplo, na posição dois, os descendentes gerados seriam (3 2 2 4) e (1 3 1 4), que contêm elementos repetidos e que não são "legais".

A investigação que se tem desenvolvido tem incidido mais sobre a concepção de novos operadores de cruzamento, adaptados a permutações, do que sobre a concepção de novas representações, uma abordagem que seria talvez mais proveitosa para manter a robustez dos algoritmos genéticos. Foram propostos muitos destes operadores

especiais, entre os quais destacamos o "Cycle Crossover", o "Partial Matched Crossover" e o "Order Crossover" [Goldberg89], [Davis91].

No entanto e como iremos ver, a abordagem oposta, ou seja, definir uma nova representação para permutações, foi seguida pelo autor e foi "inventada" uma representação que é imune aos problemas encontrados com a representação linear tradicional.

Para analisarmos a raiz do problema com a representação tradicional, contemos o número de cadeias possíveis, para uma permutação de N objectos: uma vez que a lista tem que ter N posições e cada posição tem que representar, potencialmente, N objectos diferentes, o número total de cadeias possível é N^N. Por outro lado, o espaço de estados que queremos realmente representar é muito menor, sendo igual a $N!$. Isto significa que a probabilidade de gerar uma cadeia ilegal é igual a $\mathbf{1}\text{-}N!/N^N$. Este valor tem o seu mínimo para $N = 2$ (para não referir a permutação de um elemento, que não interessa) e cresce, tendendo para 1 quando N tende para infinito. Mesmo para $N = 2$ o valor de 1/2 não é aceitável: significa que, em média, metade das cadeias geradas são ilegais.

Para percebermos como aparece a nova representação, vamos definir primeiro alguns conceitos, ligados a permutações, conceitos que já não são novos, tendo sido introduzidos por G. Cramer, em 1750, ligados à famosa regra para resolver sistemas de equações lineares. Para uma exposição mais completa deste assunto, ver [Knuth73a].

Seja $a_1a_2...a_n$ uma permutação do conjunto $\{1, 2,..., n\}$. Se $i <\supseteq j$ e $a_i > a_j$, ao par (a_i, a_j) dá-se o nome de *inversão*[11] de uma permutação. Por exemplo, a permutação (3 1 4 2) tem três inversões: (3,1), (3,2) e (4,2). Cada inversão é um par que está "fora de ordem" na permutação.

A *tabela de inversões* $b_1b_2...b_n$ da permutação $a_1a_2...a_n$ é obtida fazendo b_i igual ao números de elementos para a *esquerda* de b_i que são *maiores* que i, ou por outras palavras, b_i é o número de inversões que têm i como segundo componente. Por exemplo, a permutação (5 9 1 8 2 6 4 7 3) tem a tabela de inversões (2 3 6 4 0 2 2 1 0). Pela definição, uma tabela de inversões tem sempre números decrescentes da esquerda para a direita e o número mais à esquerda é, no máximo, n - 1.

[11]Não confundir com a permutação inversa de uma outra permutação

A correspondência entre tabelas de inversões e permutações é de um para um, o que significa que são representações alternativas, existindo algoritmos para conversão entre os dois formatos, de complexidade temporal aceitável e nunca superior a $\mathbf{O(n \log_2 n)}$. Muitos problemas sobre permutações passam a ser triviais se estas forem representadas pelas suas tabelas de inversões.

Acontece que a representação por tabela de inversões se adapta muito bem a ser processada pelo operador de cruzamento convencional dos algoritmos genéticos. De facto, e uma vez que são sempre trocadas partes correspondentes dos cromossomas, nunca há hipótese de uma cromossoma gerado pela recombinação de dois cromossomas legais dê origem a uma tabela de inversões ilegal. Por outro lado, esta representação gasta menos espaço do que a baseada na ordem dos elementos, uma vez que sabemos, *a priori*, que os elementos vão decrescendo da esquerda para a direita. Esta vantagem pode ser substancial para n grande. Assim, para cada posição basta considerar o número de bits para guardar o máximo valor presente nessa posição.

E quanto à mutação, será que não levanta problemas? De facto, a mutação levanta um problema: pode ocorrer uma mutação que faça com que um dos elementos ultrapasse o valor máximo possível para cada posição do cromossoma (igual a i - 1, em que i é a posição a contar da esquerda, começando em 1). No entanto, este problema acontece sempre em algoritmos genéticos quando há um parâmetro com uma restrição superior ou inferior que não é uma potência de 2. Não é, portanto, um defeito inerente a esta representação, mas sim à representação binária dos diversos números (que pode ser melhorada, por exemplo usando códigos de Gray).

No entanto, é de notar que a probabilidade de geração de uma cadeia ilegal é muito baixa, por duas razões: o número de cadeias ilegais é muito baixo e a probabilidade de mutação é normalmente muitíssimo mais baixa que a do cruzamento. É normal, quando uma mutação ilegal acontece, remediar este facto fazendo uma truncagem do valor, quando usado para determinar o valor da função custo, ou seja, viciando um pouco o resultado para as soluções que involvem números maiores ou ainda impondo grandes penalidades na função de adaptação para os indivíduos "impossíveis".

Assim, a representação de uma permutação pela sua tabela de inversões resolve os problemas tradicionalmente encontrados durante a optimização de problemas baseados em ordem, por algoritmos genéticos canónicos. A abordagem foi usada, com sucesso, na resolução dos problemas dos tipos *job-shop* e caixeiro-viajante.

8. Algumas experiências usando AG

Após algumas experiências iniciais simples, os algoritmos genéticos foram aplicados a dois problemas tradicionais de optimização, nomeadamente o problema do *job-shop* e o problema do Caixeiro-Viajante. Foram também aplicados à determinação de uma topologia [Ranito,Simões92] para uma rede neuronal que tivesse um bom desempenho no reconhecimento de um conjunto de caracteres destorcidos. Por fim, foram feitas algumas tentativas na área da Programação Genética, em problemas de regressão simbólica. Estas experiências serviram, essencialmente, três objectivos distintos:

- Validação e depuração das ferramentas de simulação desenvolvidas
- Ensaio de representações alternativas para os problemas a resolver
- Ensaio dos diversos operadores de cruzamento, selecção e mutação, por forma a escolher os mais eficientes

Qualquer um dos problemas escolhidos tem as suas particularidades e é interessante mas o mais aliciante foi a determinação da arquitectura para uma rede neuronal. Este é um problema não trivial e com uma dimensão (60 neurónios) que ultrapassa a referida em geral pela bibliografia (5-10 neurónios). Além disso, o resultado obtido foi melhor do que qualquer das topologias encontradas "à mão" pelos autores do trabalho.

8.1. Job-Shop

O problema de *job-shop* é um dos problemas tradicionais de optimização e pode pôr-se do seguinte modo: existe uma lista de P produtos, uma lista de M máquinas e uma lista de T tarefas, em que uma tarefa é um quádruplo (produto, máquina, ordem, duração). Para cada produto, as tarefas têm que ser realizadas pela ordem indicada, não podendo começar a tarefa i+1 antes de terminar a tarefa i, enquanto que, para cada máquina, não pode haver sobreposição de tarefas. O problema consiste em encontrar um programa (lista de tarefas) para cada máquina de tal modo que obedeça às duas

restrições mencionadas e que minimize uma determinada quantidade, normalmente o tempo de saída do último produto.

Este problema está muito longe de ser um problema académico (qualquer fábrica tem que o resolver) e sofre da habitual explosão combinatória: o tamanho do espaço de estados é, como veremos, no mínimo proporcional a **T!**, o que, mesmo para dimensões moderadas (10 produtos e 10 máquinas e cada produto tem tarefas em todas as máquinas), atinge valores de 100!, já muito grandes para abordagens do tipo enumerativo ou puramente aleatórias [Hu90]. Se considerarmos que uma grande empresa pode ter 1000 produtos e 50 máquinas, temos uma ideia do que pode acontecer a algoritmos enumerativos. Neste caso, a tentativa de resolução do problema foi provocada por alguma exposição [Cunha92] a um problema real de escalonamento numa fábrica de têxteis de grandes dimensões.

Para a explicação que se segue, vamos assumir um pequeno problema de escalonamento, com 3 máquinas e 3 produtos e em que a unidade temporal mínima é a hora. A lista de tarefas a realizar para cada produto é indicada na tabela seguinte:

Tarefa	Produto	Máquina	Ordem	Duração
1	1	1	1	2
2	1	2	2	3
3	1	3	3	1
4	2	1	1	1
5	2	3	2	2
6	2	2	3	3
7	3	2	1	1
8	3	1	2	3
9	3	3	3	2

Tabela 2 - Lista de tarefas para um problema de job-shop

Um *escalonamento*[12] possível para esta lista de tarefas será dado pelo diagrama temporal seguinte (P*n* significa Produto *n*):

[12]Um *escalonamento* é uma lista de operações, por máquina, que leva à conclusão de todas as tarefas de todos os produtos

		Tempo										
		1	2	3	4	5	6	7	8	9	10	11
M	1	P1	P1	P2	P3	P3	P3					
á	2	P3		P1	P1	P1				P2	P2	P2
q	3						P1	P2	P2	P3	P3	

Codificação genética directa

O primeiro passo para resolver este problema por algoritmos genéticos foi o de arranjar uma codificação de um escalonamento num cromossoma.

Uma abordagem directa não foi muito bem sucedida: como a sequência e a duração das tarefas são fixas, para cada produto, tentou-se uma representação em que a posição no cromossoma significava a tarefa e o valor do gene era o tempo de atraso do começo da tarefa, ou seja, quanto tempo é que a tarefa tinha que esperar, após a finalização da tarefa anterior do mesmo produto. Para a primeira tarefa, o valor é simplesmente a hora de começo.

Por exemplo, o cromossoma (2 3 4 1 0 2 2 1 3) corresponderia a colocar a tarefa 1 a partir da hora 2, a tarefa 2 teria que esperar 3 horas após a finalização da tarefa 1, ou seja, começaria na hora 6, etc. Como outro exemplo, o escalonamento anterior seria representado pelo cromossoma (0 0 0 2 3 0 0 2 2).

Com esta representação era possível gerar escalonamentos em que havia tarefas de produtos diferentes sobrepostas na mesma máquina. Por exemplo, o escalonamento (0 0 0 2 3 0 0 2 1) faz sobrepor temporalmente a tarefa 9 e a tarefa 6 na máquina 3.

Para eliminar este tipo de soluções, a função de avaliação foi então pesada por um termo que penalizava fortemente a sobreposição de tarefas. Esta representação funcionou bem para dimensões moderadas (cerca de 5×5), embora para dimensões maiores o algoritmo convergisse muitas vezes para soluções nem sequer eram admissíveis, havendo tarefas sobrepostas. Esta codificação violava uma das "regras de ouro" das codificações para AG, ou seja, era muito fácil gerar um elemento "impossível".

Codificação baseada em permutações

Para uma segunda tentativa, bastou reconhecer que o problema de *job-shop* se pode enquadrar nos problemas baseados em permutações. De facto, considerando o

conjunto de tarefas, se a ordem estiver pré-determinada por uma sua permutação, é possível gerar um escalonamento óptimo[13], sem pesquisa: basta colocar cada tarefa o mais perto possível do início da escala de tempo. Esta noção de "o mais perto possível" tem a ver com as duas restrições que mencionámos: não pode haver sobreposição de tarefas numa máquina e uma tarefa só pode começar quando a tarefa anterior do produto correspondente já terminou. A colocação das tarefas pode ser realizada em tempo **O(N log2 N)**, em que **N** é o número de tarefas, se forem usadas árvores binárias para representar a lista de tarefas que vai sendo construída em cada máquina.

Por exemplo, o cromossoma perfeitamente ordenado (1 2 3 4 5 6 7 8 9) representa o seguinte escalonamento:

		Tempo										
		1	2	3	4	5	6	7	8	9	10	11
M	1	P1	P1	P2	P3	P3	P3					
á	2	P3		P1	P1	P1	P2	P2	P2			
q	3				P2	P2	P1	P3	P3			

Havia apenas um pequeno problema: a ordem das tarefas é fixa para cada produto, o que faz que nem todas as permutações das tarefas sejam permitidas, como por exemplo na permutação (2 1 3 4 5 6 7 8 9). Ainda era possível gerar escalonamentos proibidos.

Este problema foi resolvido pelo uso de permutações com elementos repetidos: cada cromossoma passou a ser uma lista ordenada de "hipóteses" dadas aos produtos de escalonar a sua próxima tarefa, havendo tantas cópias do produto quantas as tarefas que lhe correspondem. Por exemplo, o cromossoma (2 7 1 3 4 8 5 9 6) é transformado no cromossoma (1 7 2 3 4 8 5 9 6) antes de colocar as tarefas e de avaliar a solução. Esta transformação consiste em reordenar as tarefas de cada produto, de forma a que a sua ordem corresponda à ordem pedida. Esta abordagem é equivalente a ter permutações com elementos repetidos

[13]Para essa permutação de tarefas

Codificação das permutações

Para tentar preservar o AG canónico, tentámos encontrar uma codificação para permutações que fosse passível de ser processada por AG com cromossomas binários lineares e pelos operadores normais. Essa representação foi encontrada, como foi visto no capítulo 7, e verificou-se que a sua combinação com a mutação e o cruzamento normais, embora dando resultados ligeiramente piores do que alguns dos operadores especializados, chegava a ser melhor do que outros, com a vantagem de manter um algoritmo genético canónico.

De qualquer forma, a abordagem baseada em permutações foi um sucesso parcial. Foi possível lidar com problemas que, embora não atingindo os 1000×50, eram já de alguma dimensão: foram ensaiados testes em problemas de 40 produtos × 10 máquinas, o que já dá um espaço de estados muito grande. Os resultados, em termos de qualidade da solução final, são francamente melhores do que os obtidos com recurso a métodos aleatórios ou mesmo "à mão", embora não tivessem sido comparados com a solução óptima, que era desconhecida.

8.2. Problema do Caixeiro Viajante

A nova representação foi também aplicada ao problema do Caixeiro-Viajante, com resultados foram bem melhores que o esperado e ainda melhores do que no problema de *job-shop*: dos métodos especializados [Davis91] que foram ensaiados, apenas um teve melhor desempenho do que a representação nova e apenas se deixarmos evoluir a população por muitas gerações, ou seja, a solução final obtida com a nova representação, embora ligeiramente pior, é obtida bastante mais rapidamente. Isto significa uma convergência prematura para um mínimo não global, o que pode estar relacionado com o fenómeno da deriva genética, uma vez que estávamos a lidar com populações pequenas.

8.3. Determinação de Arquitecturas de Redes Neuronais

Um outro problema em que a abordagem genética foi tentada foi a determinação de uma arquitectura para uma rede neuronal treinada por retro-propagação [Rumelhart88], que demonstrasse bom desempenho no problema do reconhecimento "visual" de alguns caracteres, deliberadamente destorcidos [Ranito,Simões92].

Para realizar essa tarefa, a rede deveria ser treinada, usando um algoritmo de retro-propagação modificada [Silva90], com um conjunto de padrões, sendo-lhe depois exigido o reconhecimento de alguns padrões que não entraram no treino, forçando a que a rede exibisse propriedades de generalização. A rede deveria ter 25 neurónios de entrada e 4 de saída e ser uma rede *feedforward*, ou seja, não conter ciclos. Estas eram as únicas restrições à topologia a implementar.

Será curioso referir que foram tentadas várias topologias "à mão" pelo autor e pela outra pessoa que realizou o trabalho, durante cerca de três semanas. Os resultados não foram brilhantes: o melhor que se conseguiu foi uma rede, com cerca de 120 ligações, que acertava nos padrões de teste em cerca de 92% dos casos, embora com uma grande variância.

Mais uma vez, o problema fulcral residia na representação do problema num cromossoma. Várias abordagens foram já tentadas ([Davis91], [Martí92a], [Martí92b], [Belew90], [Chalmers90]) com mais ou menos sucesso. Algumas destas representações colocam mais ênfase em aspectos locais das ligações, enquanto outras exploram aspectos globais. Algumas usam topologias fixas, enquanto outras permitem uma maior flexibilidade em relação ao número de camadas e neurónios. Por fim, algumas tentam mesmo ajustar parâmetros do treino da rede, enquanto outras se limitam a esboçar um plano para a rede, deixando um módulo posterior definir o que falta. Como se vê, o consenso ainda está muito longe e este é um problema muito interessante para explorar.

Para começar, foi adoptada uma representação simples, baseada numa rede com 25 entradas, uma camada escondida com 15 neurónios e uma camada de saída com 4 neurónios. O cromossoma determinava, apenas, quais as ligações entre as camadas adjacentes, sendo impossível qualquer ligação intra-camada ou entre camadas não adjacentes. Cada rede gerada era então treinada durante cerca de 60 iterações e testada. O processo de treino repetia-se, a partir de um novo ponto, e a função de adaptação foi escolhida como a média das duas tentativas. Os resultados foram bons, tendo sido encontrada uma rede com cerca de 95% de acertos nos padrões de teste, com uma variância bem menor que na rede encontrada "à mão" [Ranito,Simões 92].

Uma segunda abordagem permitia já redes mais complexas: era usada, de novo, uma camada escondida de 15 neurónios, mas agora foi dada a possibilidade de qualquer neurónio estar ligado a qualquer outro que o preceda em ordem numérica. A topologia encontrada por este método foi a melhor topologia que conseguimos, com percentagens

de acerto de cerca de 97%. A rede encontrada pelo algoritmo era bastante complexa, sendo difícil descortinar qual a função das ligações.

Por fim, e sabendo que o processo de treino de uma rede neuronal é bastante dependente das condições iniciais[14] foi feita uma última tentativa: pusemos um algoritmo genético a tentar encontrar o melhor ponto de partida para o treino da melhor rede encontrada. O facto é que foi possível, por várias vezes, encontrar acertos de 99.2%, o que indica um erro de apenas um padrão.

Se olharmos à dimensão (2^{150}) do espaço de estados com que lidávamos, temos que considerar que os resultados encontrados foram animadores. Na verdade, recorrendo a um algoritmo genético muito simples e usando técnicas de representação muito pouco sofisticadas, foi possível, em algumas horas de CPU (cerca de 40 horas úteis, no total), encontrar uma topologia muito melhor do que a encontrada "à mão" em três semanas ou mais, por duas pessoas. Este é um bom exemplo das vantagens dos algoritmos genéticos em problemas difíceis de optimização combinatória para os quais o implementador não conhece uma solução computacionalmente viável.

Regressão simbólica

Dentro do paradigma da programação genética, uma das aplicações mais interessantes é a da regressão simbólica, ou seja, determinar qual a função que melhor aproxima um determinado conjunto de pontos [Koza91a]. Para tal, o utilizador escolhe um conjunto de não-terminais que ache que são suficientes para modelizar a função desconhecida. A lista de terminais deve conter um símbolo para cada variável da função, i.e., para uma função unidimensional deve existir um símbolo, para uma função bidimensional devem existir dois símbolos, etc. A população inicial é gerada, a partir das listas de terminais e de não-terminais e é feita evoluir, de acordo com os operadores de selecção, cruzamento e mutação.

A adaptação é calculada pela avaliação da função representada pelo indivíduo, nos pontos conhecidos da função a modelizar, e é acumulado o erro quadrático. Uma vez que, em AG, interessa que os melhores indivíduos tenham melhor adaptação, é necessário transformar a medida de erro numa medida de qualidade, o que pode ser obtido pela subtracção, a um número qualquer, do erro total obtido.

É importante que os operadores usados como não-terminais sejam capazes de utilizar qualquer resultado gerado por outro operador, ou seja, os operadores serem

---

[14]É um processo de pesquisa por gradiente

fechados. Assim, houve necessidade de redefinir, por exemplo, o operador de divisão, por forma a ser capaz de tratar a divisão por zero. A opção tomada, neste caso, for fazer a divisão retornar um número muito grande em vez de falhar, quando da tentativa de uma divisão por zero.

Esta abordagem foi ensaiada com sucesso para modelizar algumas funções simples, conhecidas *a priori*, como quadráticas, cúbicas e composição de senos e cossenos. Curiosamente, os AG foram capazes de gerar constantes numéricas, mesmo quando na população inicial estas não eram incluídas: por exemplo, ao modelizar a função **x^2+0.5x+1**, o algoritmo foi capaz de gerar a constante **1**, usando **x/x** e a constante **0.5**, usando a expressão **x/(x+x)**.

O algoritmo foi sempre capaz de convergir para funções que eram quase óptimos globais, embora se notasse uma certa dificuldade em modelizar os termos menos representativos das funções, o que não deixa de ser normal, pois têm um peso muito menor na função de adaptação. Em muitos casos, o algoritmo convergiu mesmo para o óptimo, encontrando a função exacta.

No caso de funções periódicas, foi possível mesmo obter uma aproximação à série de Fourier, deixando o algoritmo usar apenas somas, produtos e cossenos, embora a aproximação obtida nunca tivesse mais do que os três primeiros harmónicos.

Embora estes resultados não sejam muito impressionantes, a verdade é que não nos podemos esquecer de que o AG não tem a mínima "noção" do tipo de função que vai modelizar, o que torna este tipo de tentativa muito mais aliciante do que a simples determinação de coeficientes da regressão numérica convencional.

Estas experiências foram muito úteis para testar o protótipo de simulação desenvolvido pelo autor para esta dissertação, simulador esse que será descrito mais em pormenor no anexo B. Além disso, foram feitas algumas experiências para determinar qual a melhor função de selecção e para encontrar valores robustos para as probabilidades de mutação e cruzamento, bem como para o tamanho da população e número de gerações.

9. Optimização relacional

Neste capítulo, é exposto o problema da optimização relacional. Assim, fala das características das linguagens relacionais que levam ao problema. Fala também um pouco sobre álgebra relacional e sobre o tamanho que as relações podem atingir se for usada uma aproximação simplista. Por fim, debruça-se sobre os diversos métodos de acesso que estão à disposição do implementador de um optimizador relacional.

9.1. Linguagens relacionais

Com o advento dos Sistemas de Gestão de Bases de Dados Relacionais e a consequente utilização de linguagens de muito alto nível, o problema de optimizar o modo de recuperar determinada informação tornou-se crucial. Na realidade, as novas linguagens de interrogação (SQL, Quel, etc), baseadas em Álgebra ou Cálculo Relacional [Ullman90], têm algumas características que as distinguem das suas congéneres mais tradicionais (C, Pascal, etc):

- Passam a ser declarativas e não procedimentais, ou seja, o programador expressa apenas QUE informação quer recuperar e não COMO a quer recuperar
- Usam uma unidade de informação (a relação) muito maior do que as anteriores linguagens, cuja unidade de informação era normalmente o registo
- São mais abstractas, o que significa que existe um "trajecto" muito maior entre a pergunta feita pelo programador e um algoritmo para a resolver
- São, em geral, interpretadas em vez de compiladas

Este conjunto de características leva a que as novas linguagens sejam, além de mais poderosas, potencialmente muito ineficientes: se o interpretador de SQL, por exemplo, não decidir correctamente qual a estratégia de acesso para uma determinada pergunta, o resultado pode ser catastrófico, sendo possível, para bases de dados muito grandes, chegar a tempos de resposta da ordem dos séculos [Ranito91].

Este problema teve que ser resolvido por forma a que os SGBD relacionais pudessem competir num mercado aonde a eficiência é fundamental. Para tal, e uma vez que o problema de decidir qual o melhor modo de acesso para uma determinada pergunta é NP-completo, as implementações tradicionais usam métodos heurísticos, que nem sequer seguem as optimizações "normais" que aparecem durante a compilação de uma linguagem imperativa [Aho87]. Estes métodos heurísticos irão ser observados mais em pormenor numa secção seguinte.

É interessante pensar que o problema da optimização de perguntas relacionais é equivalente a encontrar o melhor programa, para uma máquina virtual de selecção, que implemente uma determinada recolha de dados, definidos por condições não procedimentais.

Quando Codd propôs o modelo relacional, no princípio dos anos 70, apresentou também duas "linguagens" de interrogação, *álgebra relacional* e *cálculo relacional*, este em duas variantes, de *n-úplos* e de *domínios*. Estas linguagens possuem um mapeamento um para um entre elas e servem como bitola para o ajuizar da "relacionalidade" de uma linguagem de interrogação. Qualquer SGBD, para se poder intitular "relacional", tem que fornecer uma linguagem com, pelo menos, o poder expressivo de qualquer uma destas liguagens padrão.

Destas linguagens abstractas, a mais útil é talvez a álgebra relacional, usada como linguagem intermédia na maioria dos interpretadores comerciais de linguagens de interrogação de SGBDs relacionais. Por isso, é à álgebra relacional que iremos dedicar um pouco mais de atenção, na secção seguinte (para uma consulta mais detalhada, ver [Ranito91]).

9.2. Álgebra relacional

A álgebra relacional [Ullman89], como qualquer álgebra, baseia-se em operandos relacionados por operadores. No nosso caso, a base matemática é a Teoria de Conjuntos e a álgebra relacional importa dela vários conceitos, nomeadamente:

- Domínio - conjunto de valores, como por exemplo {1,2,3,4} ou o conjunto dos números reais
- Produto Cartesiano de *n* domínios - um novo domínio, em que cada elemento é um *n*-úplo ordenado, obtido pela concatenação de elementos dos

diversos *n* domínios, como por exemplo {a,b} × {1,2} = {(a,1),(a,2),(b,1),(b,2)}

- Relação *n*-ária - Subconjunto finito do produto Cartesiano de *n* domínios, como por exemplo {(a,1),(b2)}

A unidade fundamental de processamento da informação, para a álgebra relacional, é a *relação*, ou seja, um conjunto de n-úplos. Uma relação pode ser vista como uma *tabela*, em que cada *linha* é um n-úplo. À tabela também se dá o nome de *ficheiro* e a cada linha o nome de *registo*. Aos componentes de um registos dá-se o nome de *atributos*, *colunas* ou *campos*. Os nomes mais vulgares são tabela, registo e campo, que serão os que iremos utilizar preferencialmente durante o desenvolvimento do trabalho. É normal usar a abreviatura **T.c**, quando nos queremos referir ao campo **c** da tabela **T**.

Mas, para haver uma álgebra, não basta definir os operandos, é também necessário definir os operadores e a sua semântica. Na álgebra relacional existe um conjunto de cinco operadores denominados *fundamentais*, pois cada um não pode ser definido por qualquer combinação dos outros, além de vários outros operadores derivados, muitas vezes definidos como primitivas dos SGBDs por uma questão de facilidade de utilização e de eficiência.

Os cinco operadores fundamentais são os seguintes (letras maiúsculas em **bold** representam relações):

- União, operador binário, denotado por **R** ⌈ **S**. A união de duas relações, **R** e **S**, é uma nova relação cujos n-úplos pertencem a **R** ou pertencem a **S**. Como uma relação é um conjunto, não tem elementos repetidos.
- Diferença, operador binário, denotado por **R** − **S**. A diferença de duas relações, **R** e **S**, é uma nova relação cujos n-úplos pertencem a **R** mas não pertencem a **S**.
- Produto Cartesiano, operador binário, denotado por **R** × **S**. O produto Cartesiano de duas relações, **R** e **S**, de aridade **r** e **s**, respectivamente, é uma nova relação de aridade **r+s**, cujos n-úplos são todos os pares ordenados que é possível formar com **R** e **S**, por esta ordem.
- Selecção por uma expressão booleana **F**, operador unário, denotado por **Sel[F](R)**. A selecção, por uma fórmula **F**, de uma relação **R**, é uma nova relação, cujos n-úplos são os n-úplos de **R** cujos valores tornam a formula **F** verdadeira.

- Projecção numa lista de colunas, $\mathbf{c_1,...,c_n}$, operador unário, denotado por $\mathbf{Proj[c_1,...,c_n](R)}$. A projecção de uma relação **R**, numa lista de colunas $\mathbf{c_1,...,c_n}$, é uma nova relação, de aridade **n**, cujos n-úplos são os n-úplos de **R** mas apenas com as colunas $\mathbf{c_1,...,c_n}$.

Além destes operadores fundamentais, existem alguns operadores derivados, dos quais realçamos a *junção* e a *junção natural*:

- Junção de duas relações, pelas colunas **R.r** e **S.s** e por um operador de comparação Θ, operador binário, denotado por **Join[r Θ s](R,S).** Este operador pode ser construído pela composição da selecção e do produto Cartesiano: **Join[r Θ s](R,S) = Sel[r Θ s](R $\times$ S)**.

- Junção natural, operador binário, denotado por **R >< S**, um caso particular da junção quando o operador Θ é o de igualdade e as colunas têm o mesmo significado no esquema da base de dados. É normal eliminar uma das colunas redundantes, usando uma projecção.

Qualquer um destes operadores tem como resultado uma relação, o que significa que se podem combinar por forma a criar expressões complexas. Aliás, é desta complexidade que advém uma das partes da dificuldade da optimização de perguntas expressas em álgebra relacional: existem muitas expressões equivalentes e algumas delas podem ser implementadas muito mais eficientemente que outras.

Complexidade da álgebra relacional

Por outro lado, a implementação de alguns dos operadores da álgebra relacional têm um comportamento muito "mau", no que diz respeito à complexidade espacio-temporal. O produto Cartesiano, por exemplo, demora um tempo e ocupa um espaço proporcional ao produto do número de registos das relações argumentos. Para termos uma ideia do caos a que uma abordagem "ingénua" do problema nos pode levar, admitamos que queríamos saber quantos carros Renault havia na região de Lisboa e que a base de dados estava normalizada (e muito simplificada), possuindo três tabelas:

habitante(nome, matrícula, morada)

habitações(morada, local)

carro(matrícula, marca)

Para responder a esta pergunta, teríamos, entre outras coisas, que fazer a junção entre a tabela **carro** e **habitante** e, em seguida, a junção entre a tabela resultante e a tabela **habitante** e **habitações**. Isto seria equivalente (numa interpretação "ingénua" e

à letra da definição) a ter que realizar o produto Cartesiano entre a tabela dos habitantes portugueses (10 000 000 de registos), a tabela de habitações (1 000 000 de registos) e a tabela de automóveis (500 000 registos), produto Cartesiano esse que teria:

10000000 * 1000000 * 500000 = 5000000000000000000 registos!

Para efectuar este produto, o sistema teria que fazer cerca de 10000000000000000000 leituras e escritas de registos. Assumindo que o SGBD consegue escrever ou ler 100 000 registos por segundo (é preciso algum optimismo...), o tempo que levaria a efectuar apenas o produto Cartesiano (e ainda faltaria a selecção, por exemplo!) seria:

10000000000000000000 /100000 = 100000000000000 s

~= 31710 séculos

Claro que há maneiras mais inteligentes e sofisticadas de calcular o mesmo resultado em poucos segundos (ou as bases de dados relacionais nunca teriam saído das universidades), recorrendo a transformações na expressão algébrica por forma a encontrar uma expressão equivalente mas que seja mais eficientemente calculável.

9.3. Métodos de acesso

Apesar do facto de serem baseados em conceitos matemáticos abstractos, como a álgebra relacional, os SGBDs relacionais foram buscar muita da sua base de implementação aos sistemas de ficheiros baseados em abordagens mais convencionais. Assim, é normal ver, a baixo nível de implementação, soluções que deram boas provas em termos de eficiência, que espacial, quer temporal.

Após uma série de evoluções, quase todos os SGBDs actuais fornecem vários métodos de acesso aos dados, cada um deles optimizado para um tipo específico de pesquisa [Knuth73a]. Para que um optimizador possa tirar partido deles é necessário que a pessoa que o implementa tenha consciência da sua existência, defeitos e virtudes. Assim sendo, vamo-nos debruçar um pouco sobre que métodos de acesso são normalmente providenciados pelos SGBDs actuais e como tirar partido das suas virtudes, rodeando os seus defeitos. Vamos, no fundo, estudar as *estruturas de dados persistentes* que formam a base de um moderno SGBD.

Estes vários métodos de acesso correspondem a especializações de um conceito fundamental: recuperar informação tendo como base condições que a definem, como

por exemplo "Quais os clientes que, nos últimos dez dias, ainda não pagaram as contas" ou "Quais são os elementos que compõem o circuito impresso nº 20".

Acesso Sequencial

O método de acesso mais simples, fornecido por qualquer gestor de bases de dados, é o *acesso sequencial*. Para recuperar uma série de registos que obedecem a uma condição, o SGBD percorre *todos* os registos de uma relação, anotando quais é que interessam.

Este método de acesso é muito simples, embora muito ineficiente: a sua complexidade temporal é **O(N)**, em que **N** é o número de registos da relação. É normal os SGBDs usarem o acesso sequencial como método de acesso por omissão, até o programador indicar outra forma mais eficiente de selecção.

No entanto, há casos em que o acesso sequencial é o método mais eficiente para recuperar registos: se a relação for muito pequena, ou quisermos recuperar todos os registos, não vale a pena usar métodos mais sofisticados.

Relações ordenadas

Se a pesquisa for feita essencialmente por um dos campos, é possível melhorar substancialmente o desempenho da pesquisa sequencial, mantendo a relação ordenada por um dos campos. Esta medida simples faz com que a complexidade temporal da pesquisa, desde que feita pelo campo chave da ordenação, seja $\mathbf{O(\log_2 N)}$. Pesquisas que não involvam condições sobre a chave de ordenação funcionam exactamente como no acesso sequencial, ou seja, em tempo **O(N)**.

Acesso indexado

Gastando mais um pouco de espaço em disco, é possível criar *índices*, ou tabelas de localização, para cada campo que possa servir para especificar condições. A utilização de índices é uma técnica muito antiga, muito anterior ao uso de computadores e, curiosamente, sobreviveu muito bem à passagem para o processamento automático de informação, sendo uma sua variante ("B-trees") a base de muitos sistemas de gestão de bases de dados relacionais.

Esta abordagem permite um tempo de pesquisa $\mathbf{O(\log_b N)}$, desde que a condição de pesquisa incida sobre qualquer um dos campos com índice, em que **b** é o número de descendentes de cada nó da árvore, no mínimo 2. Além disso, é igualmente fácil e rápido responder a dois tipos de pergunta fundamentais: comparação por igualdade e por desigualdade.

Tabelas de "Hash"

Uma vez que quase todas as tabelas possuem uma *chave*, ou seja, um conjunto de campos cujo valor identifica univocamente o registo, e que, muitas das vezes, o acesso é baseado no conhecimento exacto deste conjunto de campos, desenvolveram-se técnicas para aceder muito rapidamente a um registo, sabendo o valor exacto da sua chave.

O método mais utilizado é a criação de uma tabela de "hash". Para tal, é criada uma tabela auxiliar com vários "contentores", numerados de 0 até um limite superior, normalmente um número primo grande. A cada chave possível é feito corresponder um contentor, por uma função chamada de "hashing", que deve "espalhar" uniformemente as chaves pelos contentores. É possível que mais do que uma chave corresponda ao mesmo contentor, o que provoca a possível coexistência de mais do que um registo por contentor. Uma vez que os contentores são (para aplicações baseadas em disco) de tamanho fixo, usa-se normalmente uma área de *overflow* na qual são colocados os registos que já não cabem no contentor.

Este método garante tempos de acesso **O(1)**, ou seja, constantes com o número de registos da tabela, se a função de "hash" for bem escolhida, por forma a não formar agrupamentos de registos em contentores específicos. Quanto à pesquisa, ela só pode ser efectuada quando se conhece o valor exacto da chave, sendo portanto inútil usar uma tabela de "hash" quando se quer fazer uma pesquisa baseada em gamas.

Relações explícitas

Como iremos ver mais em detalhe, a operação relacional mais usada e potencialmente mais ineficiente é a junção natural, que involve, se for mal executada, um produto Cartesiano seguido de uma selecção e de uma projecção. Para obviar a tal problema, alguns SGBDs usam uma técnica, herdada do modelo *rede*, em que cada relação do tipo M/1 ou 1/M é modelizada recorrendo a *apontadores*, ou seja, ao *endereço* dos registos que se querem relacionar. Esta técnica requer apenas que seja possível aceder a um registo em tempo **O(1)**, dada a sua localização física, operação suportada por quase todos os SGBDs. Como as relações M/1 e 1/M são a grande maioria, é possível efectuar um grande número de junções em apenas **O(N)** em vez de **O(N + M)**, o melhor que se consegue se não houver relações guardadas de forma explícita.

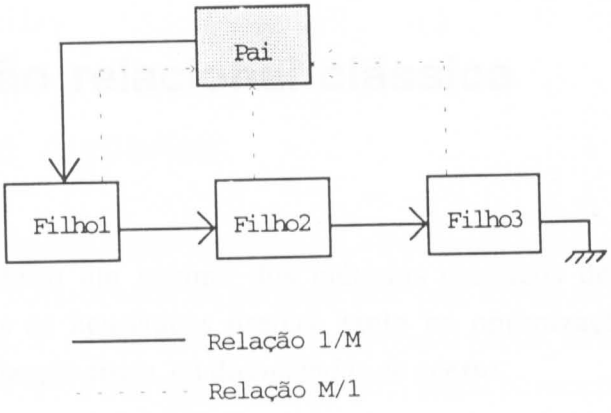

Figura 22- Implementação Baseada no Modelo Rede Para Relações 1/M e M/1

Este tipo de armazenamento é semelhante ao uso de apontadores nas linguagens convencionais de programação. Para implementar relações M/1, cada "filho" tem um apontador para o "pai" enquanto que a relação inversa, 1/M, é implementada com um apontador do "pai" para o primeiro "filho" e colocando, em cada "filho", um apontador para o seguinte. O último da lista tem um valor especial no campo do apontador para indicar que a lista terminou.

10. Optimização relacional clássica

Neste capítulo, é feito um resumo dos métodos clássicos de optimização relacional. São descritas as heurísticas usadas, tanto na optimização lógica, ou *algébrica*, como na optimização física, ou dos *métodos de acesso*.

10.1. Fases da optimização clássica

Tradicionalmente, o problema da optimização de perguntas relacionais passa por uma série de fases:

1. Tradução da pergunta, de SQL (ou equivalente) para álgebra relacional
2. Transformação da expressão de álgebra relacional na expressão com custo mínimo de implementação
3. Optimização da estratégia física de acesso correspondente à expressão encontrada no passo número 2

Cada um destes passos é completamente estanque, sendo usada álgebra relacional para comunicação entre as fases 1 e 2 e uma linguagem intermédia entre as fases 2 e 3, linguagem essa que consiste num conjunto de instruções para uma máquina virtual especializada na recolha de informação guardada numa base de dados. Tipicamente, esta máquina virtual é capaz de utilizar qualquer um dos métodos de acesso referidos atrás, embora tenha que ser o optimizador a determinar a sequência das operações. Por outro lado, deverá ser suficientemente poderosa para exprimir os cinco operadores fundamentais, além de que será muito vantajoso se a junção for considerada como uma operação primitiva, implementada eficientemente.

Cada uma das fases de optimização usa um certo número de heurísticas por forma a que o problema seja resolvido num tempo aceitável. É de notar que o uso de heurísticas, juntamente com a separação absoluta entre as fases, não garante de forma alguma que o resultado seja a melhor implementação física para uma determinada questão relacional. Qualquer uma destas fases é baseada no conteúdo do dicionário de dados, além de estatísticas referentes ao número de registos em cada tabela, etc.

Vamos analisar, em mais detalhe, a motivação e as soluções para cada uma das fases de optimização referidas.

10.2. Optimização Algébrica

Durante a fase de optimização algébrica, o objectivo é o de conseguir uma expressão equivalente à representação da pergunta mas cujo tempo de execução seja mínimo. Para isso, aplicam-se *regras de reescrita* à expressão inicial, na esperança de se obter uma expressão equivalente mas mais eficiente. Esta regras de reescrita baseiam-se numa série de leis de equivalência que se verificam na álgebra relacional e que serão aqui referidas, embora não demonstradas. Um estudo completo poderá ser visto em [Ullman89]:

- Comutatividade de produtos e junções. O produto Cartesiano e a junção gozam da propriedade comutativa, desde que a ordem das colunas não seja significativa. Isto acontece nas linguagens de interrogação modernas, aonde os campos são referidos pelo seu nome e não pela sua posição. Sejam **A** e **B** expressões de álgebra relacional. Então:

$$A \times B \ldots B \times A \tag{7}$$

$$A >< B \ldots B >< A \tag{8}$$

- Associatividade de produtos e junções. O produto Cartesiano e a junção gozam da propriedade associativa, nomeadamente se **A**, **B** e **C** forem expressões de álgebra relacional, então:

$$(A \times B) \times C \ldots A \times (B \times C) \tag{9}$$

$$(A >< B) >< C \ldots A >< (B >< C) \tag{10}$$

- "Cascata" de projecções. Uma série de projecções em cascata pode ser reduzida a uma única projecção em que as colunas a projectar são as correspondentes à mais exterior:

$$Proj[a1,...,an](Proj[b1,...,bm](E)) \ldots Proj[a1,...,an](E) \tag{11}$$

- "Cascata" de selecções. Uma série de selecções em cascata pode ser reduzida a uma selecção que combina todas as condições de todas as selecções:

$$\text{Sel[F1](Sel[F2](E))} \ldots \text{Sel[F1} \mid \text{F2](E)} \qquad (12)$$

- Comutação entre selecções e projecções. É sempre possível passar uma selecção "para dentro" de uma projecção ou "para fora", desde que só use atributos que são projectados:

$$\text{Proj[a1,...,an](Sel[F](E))} \ldots \text{Sel[F](Proj[a1,...,an](E))} \qquad (13)$$

- Comutação de uma selecção com um produto Cartesiano. Se a condição de selecção usar apenas campos de $\mathbf{E_1}$, então:

$$\text{Sel[F](E1} \times \text{E2)} \ldots \text{Sel[F](E1)} \times \text{E2} \qquad (14)$$

- Comutação de uma selecção com uma união. Se a condição de selecção usar apenas campos de $\mathbf{E_1}$, então:

$$\text{Sel[F](E1} \lceil \text{E2)} \ldots \text{Sel[F](E1)} \lceil \text{E2} \qquad (15)$$

- Comutação de uma selecção com uma diferença. Se a condição de selecção usar apenas campos de $\mathbf{E_1}$, então:

$$\text{Sel[F](E1} - \text{E2)} \ldots \text{Sel[F](E1)} - \text{E2} \qquad (16)$$

- Comutação de uma projecção com um produto Cartesiano.

$$\text{Proj[a1,...,an](E1} \times \text{E2)} \ldots \text{Proj[b1,...,bm](E1)} \times \text{Proj[c1,...,ck]E2} \qquad (17)$$

em que $b_1,...,b_m$ são os atributos, contidos em $a_1,...,a_n$, que pertencem a E_1, enquanto $c_1,...,c_k$ são os atributos, contidos em $a_1,...,a_n$, que pertencem a E_2.

- Comutação de uma projecção com uma união.

$$\text{Proj[a1,...,an](E1} \lceil \text{E2)} \ldots \text{Proj[b1,...,bm](E1)} \lceil \text{Proj[c1,...,ck]E2} \qquad (18)$$

em que $b_1,...,b_m$ são os atributos, contidos em $a_1,...,a_n$, que pertencem a E_1, enquanto $c_1,...,c_k$ são os atributos, contidos em $a_1,...,a_n$, que pertencem a E_2.

Destas leis fundamentais, podem ainda derivar-se leis acessórias para combinação de uma selecção com uma junção, por exemplo.

Heurísticas de aplicação das leis de equivalência

Dadas estas úteis leis de equivalência, podemos agora formular as heurísticas que dizem respeito à sua aplicação na direcção correcta. Embora não haja um algoritmo que garanta a produção da expressão algébrica óptima, os princípios seguintes são geralmente úteis:

1. Fazer selecções o mais cedo possível. Esta transformação nas perguntas, mais do que qualquer outra, é responsável pela poupança de ordens de grandeza no tempo de execução, uma vez que tende a fazer criar relações intermédias muito mais pequenas.

2. Combinar selecções com produtos Cartesianos para criar junções. Os produtos Cartesianos são o principal "mau da fita" das perguntas relacionais, pois o seu tamanho, bem como o tempo de execução, cresce proporcionalmente ao produto do número de registos das tabelas argumentos, ou $\mathbf{O(N^2)}$. Por outro lado, uma junção pode sempre ser calculada em tempo $\mathbf{O(N \log_2 N)}$, e a maioria das vezes em tempo $\mathbf{O(N + M)}$, o que é bem melhor para grandes **N**.

3. Combinar sequências de operadores unários. Uma série de operadores unários, selecções e projecções, pode ser combinada para ser executada numa só passagem pela tabela.

4. Descobrir subexpressões comuns numa expressão. Quando são usadas *vistas relacionais*, ou tabelas virtuais obtidas por operações sobre outras tabelas, é normal aparecerem subexpressões comuns numa expressão. Estas subexpressões devem ser avaliadas apenas uma vez, sendo o seu resultado guardado para mais tarde.

Um algoritmo de optimização relacional

Estas heurísticas podem ser combinadas com as leis algébrica, para obter um algoritmo que, para uma expressão de álgebra relacional, encontra uma expressão equivalente e com implementação substancialmente mais eficiente. Este algoritmo tem como saída um *programa*, consistindo nos seguintes passos:

1. Aplicação de uma projecção ou selecção

2. Aplicação de uma selecção seguida de uma projecção ou

3. Aplicação de um produto Cartesiano, união ou diferença de dois operandos, possivelmente precedidas por uma série de selecções e projecções em ambos os operandos, e possivelmente seguidos de várias selecções ou projecções

O algoritmo, não determinístico, é o seguinte:

ENTRADA: Uma expressão de álgebra relacional

SAÍDA: Um programa para avaliar eficientemente essa expressão

MÉTODO: Realizar cada um dos seguintes passos, por esta ordem:

1. Usar a regra (12) para separar cada selecção $Sel[F_1 \ ... \ F_n](E)$ na cascata de selecções $Sel[F_1](...Sel[F_n](E)...)$

2. Para cada selecção, usar as regras (12) a (16) para mover a selecção o mais para baixo possível na árvore da expressão.

3. Para cada projecção, usar a regras (11), (17) e (18) e a regra (13), por forma a mover a projecção o mais para baixo possível na árvore da expressão. É de notar que a regras (5) pode fazer desaparecer algumas projecções, enquanto que podemos eliminar projecções que projectem uma relação em todos os seus atributos.

4. Usar as regras (11)-(13) para combinar cascatas de projecções e de selecções numa só projecção, uma só selecção ou numa selecção seguida de uma projecção. É de notar que é mais eficiente agrupar todas as selecções e depois fazer todas as projecções do que entremeá-las, como seríamos levados a fazer pela heurística "fazer selecções o mais cedo possível".

5. Partir os nós interiores da árvore resultante em *grupos*, do modo seguinte: Cada nó interior que representa um operador binário fica no mesmo grupo que qualquer um dos seus antepassados que têm como anotação um operador unário. Deve-se incluir também no grupo qualquer cadeia de descendentes rotulados por operadores unários e terminando numa folha, excepto quando o operador binário é um produto Cartesiano e não é seguido por uma selecção que, combinada com o produto, dê uma junção com o operador de igualdade (*equijunção*).

6. Produzir um programa consistindo num passo para a avaliação de cada grupo em qualquer ordem tal que nenhum grupo seja avaliado depois dos

seus descendentes. Este programa pode ser gerado recorrendo a *ordenação topológica* [Knuth73], por exemplo.

Este algoritmo produz um programa que pode ser avaliado numa máquina virtual para executar as primitivas algébricas. No entanto, este algoritmo sofre de alguns defeitos:

- É não determinístico, ou seja, em cada passo pode aplicar-se mais do que uma regra e os resultados podem ser diferentes consoante a ordem de aplicação das regras
- É heurístico, ou seja, não garante que a expressão obtida seja de facto de custo mínimo.
- Separa radicalmente a optimização algébrica da optimização física, o que pode levar a erros de optimização.
- Não incorpora, de forma alguma, a semântica que é acessível nos modelo conceptuais mais evoluídos, tais como ER expandido ou OMT [Rumbaugh91], o que lhe garante um grande leque de aplicabilidade mas provoca o negligenciar de algumas fortes possibilidades de optimização.
- A primeira solução que encontra é a melhor, o que significa que não pode ser interrompido em qualquer altura e dar um resultado coerente. Isto pode inibir a sua utilização em ambientes que exigem resposta em tempos determinados (ambientes difíceis em tempo real).

Este não é o único algoritmo que se conhece para optimização algébrica mas os outros algoritmos, tais como o algoritmo de Wong-Youssefi [Ullman89], são extensões a esta abordagem essencial, apenas usando mais heurísticas ou heurísticas mais elaboradas

10.3. Optimização física

Após a fase de optimização algébrica, segue-se tradicionalmente a fase da optimização física, em que os operadores relacionais são implementados fisicamente, recorrendo aos métodos de acesso postos à disposição do programador pelo SGBD. Para vermos em que consiste esta fase, vamos dar uma vista de olhos à forma como os diversos operadores fundamentais podem ser implementados recorrendo aos vários métodos de acesso.

Pressupostos

Qualquer que seja o método de acesso escolhido, é muito importante ter em mente que a limitação essencial de velocidade num SGBD convencional é o acesso aos periféricos lentos, tais como o disco. Nesta análise sumária que vamos efectuar, é admitido que o tempo de pesquisa em memória, uma vez colocada lá a informação, é negligenciável em comparação com o tempo que demora a aceder e a ler a informação do disco.

Além disso, admitimos que a informação está guardada no disco sob a forma de *blocos* e que cada leitura/escrita de um bloco é uma operação que demora sempre o mesmo tempo, que depende do tipo de dispositivo que se está a utilizar. Para um termo de comparação, a maioria dos discos comerciais tem um bloco de cerca de 1024 bytes e um tempo de acesso médio da ordem dos 15 ms. No entanto, o tempo de acesso a vários blocos consecutivos é muito mais pequeno, o que faz com que seja benéfico usar sistemas de *buffering* [Knuth73], [Bach86].

Por outro lado, vamos também admitir que a memória rápida pode conter, simultaneamente, um determinado número pequeno de blocos $\mathbf{N_m}$ (da ordem do milhar) e que cada bloco tem capacidade para guardar um número pequeno de registos, $\mathbf{N_{rb}}$.

Projecção

A projecção tem que ser efectuada através de um acesso sequencial, uma vez que todos os registos nos interessam. Assim, a projecção não tem muitas hipóteses de optimização a não ser quando ligada a outro operador, como a selecção.

Produto Cartesiano

O produto Cartesiano é o operador relacional mais ineficiente. De facto, se mal implementado, o produto Cartesiano pode gastar um número de leituras e escritas de blocos proporcional ao produto do número de registos das tabelas operandos. No entanto, é possível fazer melhor do que isto, lendo sequencialmente grupos de registos da menor relação para a memória e percorrendo a maior relação um menor número de vezes. Se a relação menor couber toda em memória, o algoritmo pode ser executado em tempo proporcional à soma dos números de registos das duas tabelas, em vez de ser proporcional ao produto. Se esta última condição não se verificar, teremos que efectuar $\mathbf{(N/N_{rb})/N_b}$ leituras, em que **N** é o número de registos da relação menor da maior relação, o que mesmo assim é muito melhor, principalmente se o

número de blocos que cabe em memória não for muito menor que o número de registos da menor relação.

União e diferença

A união e a diferença de relações também se podem tornar preocupantes, tendo um comportamento semelhante ao produto, se não forem tomadas algumas precauções. Para efectuar rapidamente a união ou a diferença de duas relações, é conveniente preprocessá-las, ordenando-as para que seja possível, numa só passagem, verificar quais os registos que são iguais. A ordenação demora um tempo $\mathbf{O(N \log_2 N + M \log_2 M)}$, em que **N** e **M** são o número de registos das relações argumentos. A união e a diferença podem, então, ser efectuadas em tempo **O(N + M)**, o que dá um tempo total, se **N** e **M** forem semelhantes, de $\mathbf{O(N \log_2 N)}$, compensador em relação a $\mathbf{O(N^2)}$, mesmo com coeficientes desfavoráveis, se **N** for muito grande (como é típico em BDs). No caso de haver um índice sobre a chave, é possível realizar estas operações em tempo **O(N + M)**.

Selecção

Quanto à selecção, ela constitui a base da optimização, como vimos na secção anterior: se conseguirmos reduzir o tamanho das relações intermédias, por selecções eficientes e realizadas o mais cedo possível, o efeito multiplicativo dos operadores binários não se faz sentir de uma forma tão aguda. Assim sendo, vale a pena dedicar um pouco de tempo a estudar que tipos de restrições aparecem quando vamos avaliar uma selecção.

Antes de mais, convém separar as selecções *conjuntivas* das que o não são. Uma selecção conjuntiva é aquela em que a condição global é, simplesmente, uma conjunção de subcondições, que não contêm disjunções. Este tipo de selecções, de longe a mais frequente, pode ser resolvida eficientemente pela aplicação de algumas regras, pela seguinte ordem:

1. Se houver uma condição de igualdade sobre um campo que serve de chave a uma tabela de *hash*, seleccionar o registo usando a tabela de *hash* e verificar as condições restantes. Esta abordagem demora um tempo **O(1)**.

2. Se houver uma condição de igualdade sobre um campo que tem um índice, pesquisar por esse campo e verificar as condições restantes. Esta abordagem usa um tempo de $\mathbf{O(\log_2 N)}$.

3. Se houver uma condição que involve uma gama sobre um campo que tem um índice, pesquisar por esse campo e verificar as condições restantes. Esta abordagem usa um tempo de $\mathbf{O(\log_2 N)}$, embora possam surgir mais registos do que no passo anterior.

4. Se houver uma condição que involve uma desigualdade sobre um campo que tem um índice, pesquisar por esse campo e verificar as condições restantes. Esta abordagem usa um tempo de $\mathbf{O(\log_2 N)}$, embora possam surgir mais registos do que no passo anterior.

5. Se não se verificar nenhuma das condições anteriores, usar acesso sequencial e verificar todas as condições. Esta abordagem gasta um tempo $\mathbf{O(N)}$.

A heurística por trás destas regras é simples: ler do disco e aplicar testes ao menor número possível de registos. Claro que seria possível refinar o algoritmo anterior se houvesse informação mais relevante sobre algumas estatísticas, tais como distribuição dos valores de um campo. No entanto, praticamente nenhum gestor de bases de dados guarda informação estatística tão detalhada.

O outro tipo de perguntas, *não conjuntivas*, é muito mais difícil de tratar, uma vez que são *inclusivas*, não podendo, no caso geral, ser aplicada a heurística anterior sem mais processamento. No entanto, em casos particulares, é possível separar as condições por forma a que apareça uma condição *dominante*, pela qual seja possível pesquisar, aplicando o algoritmo anterior.

Junção

Por fim, falta considerar a junção. Tal como as selecções foram separadas em dois tipos, também aqui é proveitoso considerar um caso particular da junção, a *equijunção*, que compara dois campos de tabelas distintas pelo operador de igualdade. Este caso é o mais típico, uma vez que é a operação inversa da *decomposição relacional*, o processo que leva à normalização de um esquema, e a única forma de reunir a informação separada durante a decomposição.

No pior caso possível da equijunção, é necessário fazer $\mathbf{O(N \times M)}$ leituras e escritas, em que $\mathbf{N}$ e $\mathbf{M}$ são os números de registos das tabelas a juntar. No entanto, é possível, tal como para a diferença e a união, preprocessar as relações, ordenando-as pelos campos por onde se efectua a junção. Esta operação pode ser concluída em tempo $\mathbf{O(N \log_2 N)}$. Em seguida, o processo de junção pode decorrer num tempo $\mathbf{O(N +}$

M), fazendo com que o tempo total seja **O(N $\log_2$ N)**. Este tempo pode compensar largamente, se **n** e **m** forem suficientemente grandes.

No entanto, a fase de ordenação pode ser omitida ser houver um índice em ambos os campos a juntar, pois é possível seguir o índice e realizar a operação de junção em tempo **O(N + M)**. Esta é a opção normal para fazer a equijunção de tabelas básicas, embora seja difícil de implementar para relações intermédias, derivadas da avaliação da árvore que representa a expressão algébrica.

Por fim, a equijunção pode ser ainda mais eficiente em dois casos. Um deles é se houver uma tabela de "hash" sobre alguns dos campos que entra na junção, é possível realizar a junção em tempo **O(N)**, uma vez que, para cada registo de uma das relações, saber qual é o registo correspondente na outra relação pode ser feito em tempo **O(1)**, desde que a ordem da junção seja da tabela "filho" para a tabela "pai", ou seja, de M para 1.

O outro caso é quando existe uma relação explícita, sob a forma de cadeias de apontadores, como vimos, em que a junção pode também ser realizada em tempo **O(N)**. Esta deverá ser a opção preferida para realizar junções entre tabelas básicas, pois é a mais eficiente, sendo até mais eficiente do que usar a tabela de "hash": embora o tempo seja da mesma ordem de grandeza, a constante de proporcionalidade é menor no segundo caso, uma vez que lidamos directamente com os endereços, poupando-se assim vários acessos a disco por cada registo.

O que foi dito para a equijunção aplica-se à junção genérica, com a ressalva de que métodos baseados em valores exactos, como as relações explícitas e as tabelas de "hash", não podem ser aqui usados.

10.4. Restrições

As heurísticas referidas podem ser incorporadas num algoritmo que obtenha um programa "óptimo" (leia-se "muito bom") para uma determinada expressão algébrica. No entanto, é necessário notar alguns pontos:

- Seria errado aplicar cegamente as regras referidas, uma vez que o desempenho depende enormemente do número de registos a processar: se as relações forem pequenas, mais vale lê-las para memória e não usar algoritmos sofisticados.

- A optimização do método físico de acesso está completamente separada da optimização lógica, o que significa que se perdem algumas oportunidades de optimização cruzada.
- É extremamente importante considerar as várias alternativas possíveis, avaliando o seu desempenho, antes de se decidir por uma determinada implementação.

• Muitas vezes, o preprocessamento das relações (como a ordenação) pode fazer a diferença, se o número de registos for muito grande.

11. Contexto do optimizador: o interpretador de SQL

O optimizador relacional é o "coração" de qualquer interpretador de SQL. No entanto, um optimizador não pode fazer tudo sozinho. Normalmente, o trabalho desempenhado por um interpretador de SQL é distribuído por diversos módulos, cada um realizando uma tarefa bem específica. Este optimizador não é excepção. De facto, ele foi feito para ser integrado num interpretador de SQL que está a ser desenvolvido no âmbito projecto SAGA, do INESC, e que começou por ser o trabalho de fim de curso de um aluno [Neta92].

11.1. Arquitectura do interpretador

O interpretador de SQL que foi implementado tem, essencialmente, três módulos: um *parser*, um *optimizador* (o AGORA, Algoritmos Genéticos na Optimização Relacional Aproximada) e um *executor*. Estes três módulos são completamente independentes e comunicam entre si através de linguagens intermédias, a linguagem ARRE (Álgebra Relacional Reduzida ao Essencial) e a linguagem LORPA (Linguagem Optimizada de Representação de Perguntas Algébricas), como podemos ver na figura seguinte:

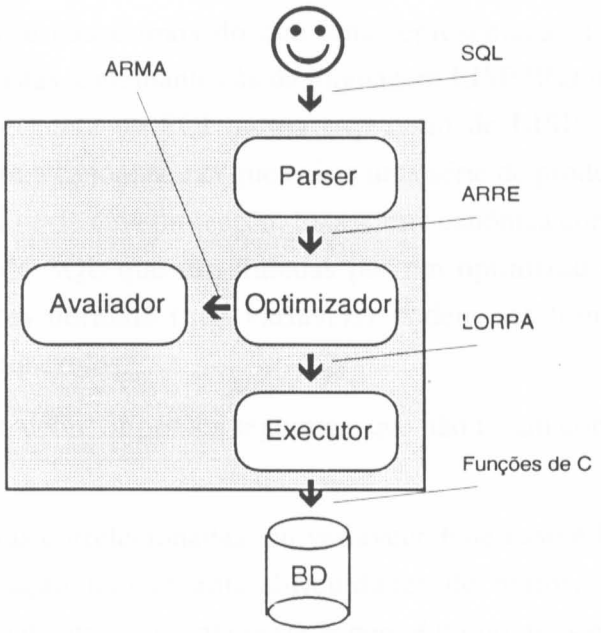

Figura 23. Arquitectura do interpretador de SQL

Existe também um módulo de avaliação, responsável por determinar o custo, em termos de número de acessos ao disco, de uma determinada solução, proposta pelo optimizador. Este módulo comunica com o optimizador através de uma linguagem especializada.

O *parser* traduz o SQL, introduzido pelo utilizador, para uma expressão em álgebra relacional, ainda não optimizada e passa-a ao optimizador. Este processa a expressão e gera uma nova expressão, optimizada e anotada, por forma a que o executor possa consultar a base de dados de uma forma eficiente. As anotações inseridas pelo optimizador são essencialmente ligadas à optimização física, ou seja, utilização de índices, necessidade de ordenação, etc.

A separação absoluta entre os módulos permitiu que estes pudessem ser desenvolvidos de uma forma completamente autónoma, apenas com o cuidado de obedecer ao interface definido. Os módulos de *parsing*, avaliação e execução foram realizados por [Neta92], em C, UNIX e utilizando o LEX e o YACC. O executor está corre sobre numa biblioteca pública de funções para manutenção de dados, o D/ISAM, completamente compatível com bases de dados geradas pelo INFORMIX. O avaliador é totalmente independente do optimizador, o que significa que é fácil ensaiar novas abordagens.

A linguagem ARRE

A linguagem ARRE é o veículo de comunicação entre o módulo de *parsing* e o módulo de optimização e não é mais do que uma representação física de álgebra relacional, baseada em listas semelhantes às da linguagem LISP [Ranito92a]. Assim, o *parser* transforma a pergunta de SQL numa expressão de LISP, depois de a ter modificado para uma forma canónica, em que existe uma série de produtos Cartesianos, seguidos de uma selecção e de uma projecção. Esta forma canónica contempla a grande maioria das perguntas de SQL que são tratadas por um optimizador, uma vez que mesmo as sub-perguntas normais (não variáveis) podem ser transformadas para obedecer aos critérios enunciados.

Por outro lado, não cobre os pontos seguintes, que não foram considerados muito importantes:

- Sub-perguntas correlacionadas, ou variáveis. Este caso é bastante raro e a sua optimização não levanta dificuldades de maior, uma vez que a interpretação de uma pergunta deste género é muito mais procedimental do que uma pergunta normal;
- Funções de agregação, ou de grupo (avg, sum, min, max, count). As funções de agregação são triviais de calcular, bastando ordenar a relação resultante e agrupar os registos antes de aplicar as funções nos pontos de quebra;
- Uniões e intersecções. Como iremos ver quando da descrição do funcionamento do optimizador, veremos que é extraordinariamente simples incluir a união e a intersecção no enquadramento definido;
- Modificação da base de dados. As cláusulas **insert**, **delete** e **update** não foram consideradas, uma vez que não se trata propriamente de optimização relacional, embora possam tirar partido do optimizador;
- Criação de bases de dados e manutenção da segurança. Este tópico não foi endereçado pois não contribui, de nenhuma forma, para a optimização relacional.

Apesar do optimizador não tratar nenhum destes tipos de pergunta, é de notar que o *parser* suporta [Neta 92] a sintaxe completa do ANSI SQL, podendo por isso ser usado para construir um interpretador completo, se disso houver necessidade.

Tendo tudo isto em conta, uma expressão em ARRE será sempre algo semelhante a:

$Proj[C_1,...,C_n](Sel[F]\ (T_1 \times ... \times T_n)$

Uma expressão destas é representada em LISP do seguinte modo:

(proj (c_1 ... c_n) (sel F (prod t_1 (prod t_2 ...) t_n))

Para esclarecer eventuais dúvidas, vamos analisar uma pergunta concreta de SQL e ver como é que ela se passa para álgebra relacional e para LISP. Imaginemos que o utilizador digita a seguinte pergunta:

```
select nome, morada, localidade, salario
from empregado, codigo_postal
where codigo_postal = codigo and
nome = "Faria" and
salario < 100;
```

Em álgebra relacional, esta pergunta seria representada por:

```
Proj[nome, morada, localidade, salario]
(
    Sel[codigo_postal = codigo and nome = "Faria" and salario < 100]
    (empregado × codigo_postal)
)
```

Em ARRE, seria representada da seguinte forma:

```
(proj (nome morada localidade salario) (sel (and (< salario 100) (and (=
codigo_postal codigo) (= nome "Faria"))) (prod empregado
codigo_postal)))
```

ou mais visualmente:

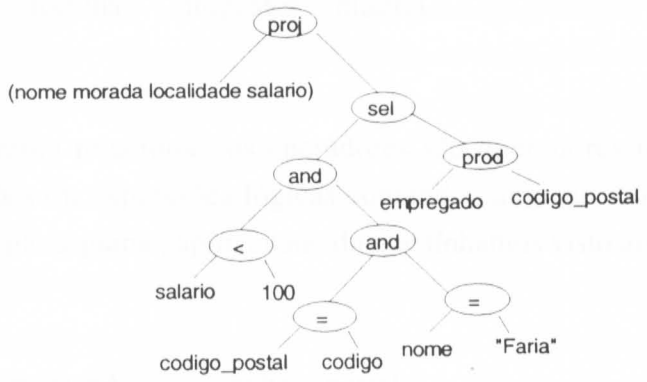

Figura 24. Representação gráfica da linguagem ARRE

As funções que dominam cada nó não-terminal são os operadores relacionais tradicionais: projecção (proj), selecção (sel) e produto cartesiano (prod) ou operadores lógicos. Os nós terminais correspondem a constantes explícitas (datas, nomes, números), tabelas ou campos.

É de notar que esta linguagem não inclui nenhuma informação de ordem física, no que diz respeito aos métodos de acesso à base de dados, sendo por isso independente da plataforma escolhida e completamente não optimizada.

A linguagem LORPA

A outra linguagem intermédia, que liga o optimizador ao executor, contém já referência aos métodos físicos de acesso à base de dados. O suporte físico desta linguagem é em tudo semelhante ao da linguagem ARRE, usando também o módulo de LISP mas a estrutura da linguagem é plana e não hierárquica, constituindo um programa a ser corrido pelo executor, sem nenhuma "inteligência" da parte deste. Este programa indica ao executor qual a forma como deve realizar as operações relacionais que constituem a implementação da pergunta.

Em essência, a linguagem LORPA é uma variante de uma "linguagem de três endereços" [Aho87], ou seja, é constituída por instruções que contêm um operador, o nome do local de dois operandos (possivelmente apenas um) e o nome do local aonde colocar o resultado, como por exemplo:

```
...
soma      dinheiro    accoes      fortuna
subtrai   fortuna     impostos    miseria
...
```

No caso da álgebra relacional, os operadores são operadores relacionais e os operandos são relações ou expressões lógicas contendo campos e constantes, como podemos ver no exemplo seguinte, aproveitado do que tínhamos visto anteriormente:

```
...
prod    empregado    codigo_postal                    temp1
sel     temp1        [codigo_postal = codigo]         temp2
sel     temp2        [salario < 100]                  temp3
sel     temp3        [nome = "Faria"]                 temp4
proj    temp4        [nome, morada, localidade, salario]  resultado
...
```

Os operadores que o executor conhece são os essenciais da álgebra relacional[15] e mais alguns que são imprescindíveis para um bom rendimento e eficiência:

Operador	**Função**
Selecção	select
Projecção	proj
Produto Cartesiano	prod
Ordenação	ord
Criação de índice	index
Junção	join

Tabela 3. Operadores relacionais suportados pelo executor

Um programa em LORPA é, então, uma sequência de instruções deste tipo para o executor relacional, que as traduz para funções em C.

[15]Menos a união e a intersecção

No entanto, para que a tradução possa ser o mais eficiente possível, é necessário acrescentar mais alguns parâmetros, por forma a que o executor saiba, por exemplo, qual é o método de acesso que deve usar para fazer uma determinada selecção. Assim, cada instrução pode ter um número variável de parâmetros, que dependem do operador e que vamos ver em mais detalhe. Qualquer uma das funções coloca o resultado numa relação temporária.

Selecção

A função **(select FP FS R Q OUT)**, responsável por efectuar a selecção, actua sobre uma relação e cria uma nova relação, que é formada pelos registos da primeira que obedecem a um conjunto de condições de selecção, a *fórmula*. Se as condições forem conjuntivas e houver uma condição que pode ser implementada através de um método de acesso rápido, como um índice, então o executor deve ser alertado para o facto. Assim, os argumentos para a função **select** são os seguintes:

Sigla	Nome	Descrição	Exemplos
FP	Fórmula principal	Uma expressão simples em LISP, com apenas um operador, um campo e um valor constante	(== salario 3) (> codigo 5)
FS	Fórmula secundária	Uma expressão complexa em LISP, que pode involver quaisquer operadores aritméticos, lógicos ou de comparação	(and (== b a)(not (> c (+ 1 3)))) (<= nome "joao")
R	Relação	Átomo de LISP que representa o nome de uma relação	empregados
Q	Qualificador	Pode ser um átomo ou uma lista de LISP, indicando parâmetros adicionais para a pesquisa, como o nome do índice ou o tipo de acesso a usar	seq ord (btree bt001) (hash h230)

OUT	Relação	Átomo de LISP que representa o nome de uma relação temporária	tmp1

Projecção

A projecção é efectuada pela função **(proj LC R OUT)**, que tem apenas dois argumentos: uma relação e uma lista de campos, em que a relação irá ser projectada:

Sigla	Nome	Descrição	Exemplos
LC	Lista de campos	Lista de átomos de LISP, simbolizando os campos em que a relação **R** vai ser projectada	(nome morada telefone) (codigo_postal salario)
R	Relação	Átomo de LISP que representa o nome de uma relação	empregados
OUT	Relação	Átomo de LISP que representa o nome de uma relação temporária	tmp1

Ordenação

Por vezes, torna-se mais eficiente ordenar uma relação antes de efectuar uma junção. Nestes casos, o optimizador indica o facto ao executor através da função **(ord LC R OUT)**. Esta função ordena a relação **R** pelos campos **LC**. Para o efeito (junção) é irrelevante se a ordenação é ascendente ou descendente, desde que se use sempre a mesma ordem. Os argumentos são os seguintes:

Sigla	Nome	Descrição	Exemplos
LC	Lista de campos	Lista de átomos de LISP, simbolizando os campos pelos quais a relação **R** vai ser ordenada	(nome morada telefone) (codigo_postal salario)

R	Relação	Átomo de LISP que representa o nome de uma relação	empregados
OUT	Relação	Átomo de LISP que representa o nome de uma relação temporária	tmp1

Indexação

Uma alternativa à ordenação é a criação de um índice, implementada pela função **(index LC R Q OUT)**. Esta função cria um índice de pesquisa numa tabela,numa determinada lista de campos. Os argumentos necessários são os seguintes:

Sigla	Nome	Descrição	Exemplos
LC	Lista de campos	Lista de átomos de LISP, simbolizando os campos para criação do índice	(nome morada telefone) (codigo_postal salario)
R	Relação	Átomo de LISP que representa o nome de uma relação	empregados
Q	Qualificador	um atomo que indica qual o tipo de índice a criar	btree hash
OUT	Relação	Átomo de LISP que representa o nome de uma relação temporária	tmp1

Produto Cartesiano

O produto Cartesiano é implementado por uma função muito simples, a função **(prod R S OUT)**. Uma vez que não há muito por onde melhorar um produto, os argumentos a passar são apenas os esperados:

Sigla	Nome	Descrição	Exemplos

R	Relação	Átomo de LISP que representa o nome de uma relação	empregados
S	Relação	Átomo de LISP que representa o nome de uma relação	salarios
OUT	Relação	Átomo de LISP que representa o nome de uma relação temporária	tmp1

Junção

A junção é a função mais importante para a optimização de perguntas relacionais, pois é a principal culpada do crescimento exponencial das relações intermédias. Assim, é compreensível que seja descrita ao executor com um certo detalhe, donde o número elevado de argumentos da função **(join E R S QR QS OUT):**

Sigla	Nome	Descrição	Exemplos
E	Expressão	Expressão simples de LISP para a junção, involvendo um operador de comparação e dois campos	(== salario1 salario2) (> codigo produto)
R	Relação	Átomo de LISP que representa o nome de uma relação	empregados
S	Relação	Átomo de LISP que representa o nome de uma relação	salarios
QR	Qualificador para a relação R	Indicação para ajudar o executor a escolher o método de acesso à relação R	seq (btree bt100) (hash h67)

QS	Qualificador para a relação S	Indicação para ajudar o executor a escolher o método de acesso à relação R	seq (btree bt100) (hash h67)
OUT	Relação	Átomo de LISP que representa o nome de uma relação temporária	tmp1

A linguagem ARMA

Um dos pontos fulcrais de um AG é a função de avaliação, que atribui um valor de adaptação a cada indivíduo da população. E, nos problemas muito simples, a função de avaliação pode actuar directamente sobre o cromossoma, ou *genótipo*. No entanto, num problema mais complexo, é necessário passar por uma fase de *descodificação*[16], que transforma o genótipo num *fenótipo*. A adaptação ao meio de qualquer ser vivo depende muito mais directamente do seu fenótipo (se tem as patas grandes ou um faro apurado) do que do seu genótipo (o facto de os genes de um animal indicarem que ele será muito forte poderá não se concretizar por restrições ambientais, por exemplo).

No caso da optimização relacional, optou-se por esta via, ou seja, ter uma representação fenotípica que não coincide com o genótipo. Esta opção permite maior liberdade, tanto da parte da representação escolhida para o cromossoma, que se pode adaptar mais facilmente às exigências de um AG, como da parte da função de avaliação. Este ponto é muito importante, se tivermos em conta que a função de avaliação poderá ser usada [Neta92] num optimizador clássico.

Assim, para a comunicação entre o módulo de optimização e o de avaliação, foi concebida uma outra linguagem, a linguagem ARMA (Álgebra Relacional Muito Anotada) que obedece aos requisitos descritos acima e que possui duas características muito importantes:

- É hierárquica, tal como a linguagem ARRE, o que faz com que seja preservada a estrutura em níveis da pergunta original
- Desce ao nível de detalhe da linguagem LORPA, o que permite que a optimização se faça simultaneamente no nível lógico e no nível físico

[16]Na Natureza, a descodificação do genótipo corresponde ao desenvolvimento do ser vivo

A estrutura escolhida para a linguagem não é mais do que a linguagem ARRE, enriquecida com os operadores e argumentos presentes na linguagem LORPA. Uma solução para o problema, ainda em fase de optimização, será representada em ARMA.

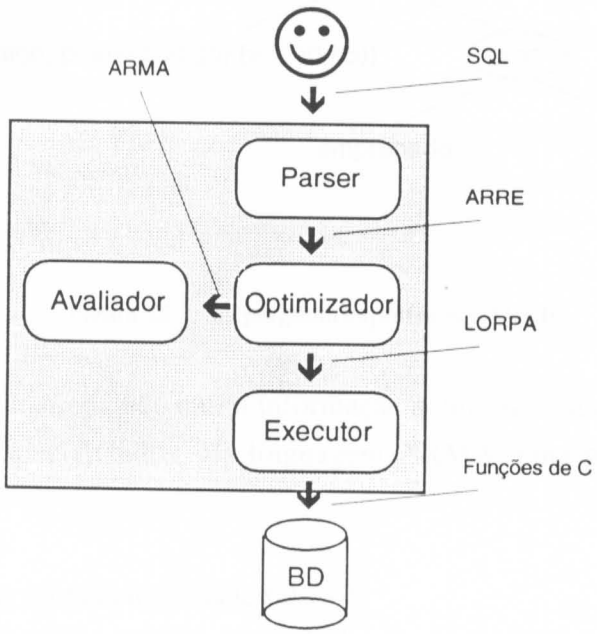

Figura 25. Arquitectura do interpretador de SQL (repetição)

Para percebermos melhor a equivalência entre as linguagens, vamos observar um exemplo, expresso em ARRE:

> (proj (nome cidade) (select (and (= codigo_postal) (> idade 5))) (prod empregado cpostal)))

ou mais graficamente:

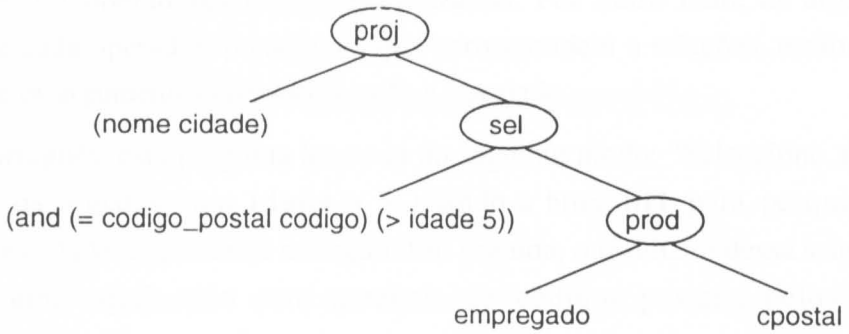

Figura 26. Uma pergunta expressa em ARRE

Esta representação ainda não inclui informação sobre os métodos de acesso nem os operadores mais especializados. Na linguagem ARMA, uma possível expressão equivalente seria:

(proj (nome morada localidade salario)
(join (= codigo_postal codigo) (select (> idade 5) () empregado (btree bt1)) seq seq))

ou, ainda, graficamente:

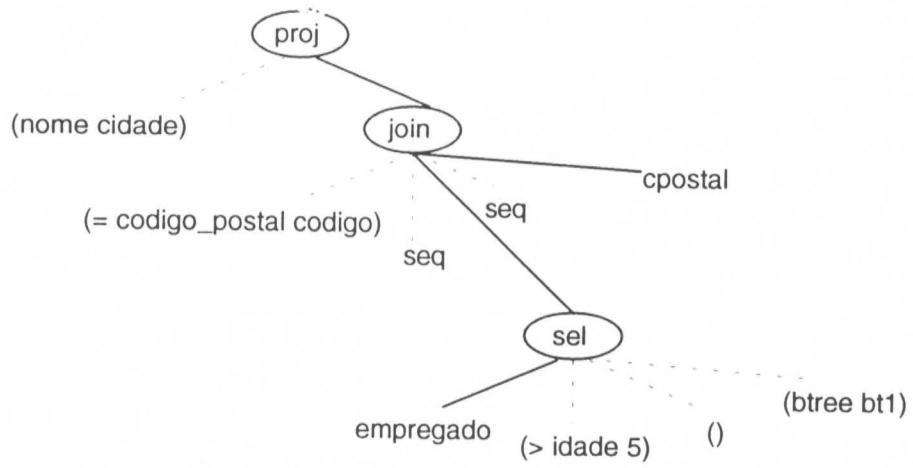

Figura 27. Pergunta expressa em ARMA

Note-se que esta representação, embora preserve a estrutura hierárquica da expressão original, passa já a incluir informação sobre os métodos de acesso a utilizar,

além de que usa operadores mais especializados. Por outro lado, os argumentos principais de cada operador, ou seja, os que correspondem a relações, estão a cheio, enquanto que os argumentos acessórios estão a tracejado.

Em Português, esta pergunta ler-se-ia do seguinte modo: "Seleccione, na tabela empregado, os registos com **idade** > 5, usando a btree **bt1** para pesquisar, não havendo necessidade de posterior selecção. Em seguida, o resultado dessa selecção vai participar numa equijunção com a tabela de códigos postais, pelos campos **codigo_postal** e **codigo**, usando acesso sequencial em ambas as tabelas para realizar a junção. Por fim, o resultado desta junção é para ser projectado nas componentes **nome** e **cidade**".

A tradução de ARMA para LORPA é trivial, bastando uma visita à árvore, pelos ramos principais, para gerar as instruções sequenciais em LORPA, além dos nomes automáticos para as relações intermédias.

12. O optimizador relacional genético

Este capítulo descreve a implementação do optimizador relacional genético e os resultados obtidos. É discutida a representação usada, representação que pode ser utilizada em geral para problemas de manipulação simbólica.

12.1. Introdução

Após a definição clara dos interfaces com os diversos módulos que constituem o interpretador de SQL, chegou a altura de nos debruçarmos mais cuidadosamente sobre o optimizador.

Qualquer aplicação de AG a um problema real passa por diversas fases, de entre as quais as mais importantes são:

- Codificação do problema num cromossoma
- Geração da população inicial
- Adaptação dos operadores genéticos ao problema, se necessário
- Ajuste do algoritmo até encontrar uma boa solução
- Análise dos resultados

Em qualquer uma destas fases há diversas opções e caminhos que podem ser escolhidos. Estas escolhas, que no fundo representam o trabalho criativo de uma dissertação como esta, são o tema descrito nas secções seguintes.

12.2. Codificação

Em qualquer aplicação de AG, a primeira fase de trabalho, e talvez a mais importante, é decidir qual a representação que vai ser usada para mapear uma solução do problema num cromossoma. Já vimos que as opções são muitas e que a escolha pode condicionar muito os resultados finais da aplicação.

Cromossomas lineares ou estruturados?

O primeiro passo é decidir em que paradigma genético vamos "encaixar" o nosso problema. Essencialmente, temos 2 opções [Koza91a]:

- Cromossoma linear
- Cromossoma estruturado, como na programação genética

Dentro de cada opção principal ainda há muitas variantes (comprimento fixo/alterável para o cromossoma, posições fixas/alteráveis para os genes, etc.) mas estas são as principais categorias em que podemos classificar um AG e que definem, por arrastamento, uma série de varáveis, como o tipo de operadores, probabilidades de actuação destes, etc.

À primeira vista, pareceu-nos que a abordagem a seguir seria a da programação genética, uma vez que os cromossomas são listas, o que coincide com o formato escolhido para as linguagens intermédias do interpretador. No entanto, o problema da optimização relacional tem algumas características que o tornam diferente, por exemplo, de um problema de regressão simbólica:

- Pretende-se encontrar a melhor expressão equivalente a uma outra dada, em vez de encontrar a melhor aproximação a um conjunto de pontos
- O tipo de dados principal é a relação, em vez de tipos simples como um número
- Pretende-se optimizar a própria avaliação, em vez desta ser usada como uma mera ferramenta

A primeira característica é importante, pelo menos de um ponto de vista teórico: temos que garantir que qualquer alteração genética a uma solução produza uma nova solução, uma vez que não temos meios algorítmicos de garantir a equivalência de duas expressões arbitrárias [Hofstadter79] [Ullman81], a menos da enumeração exaustiva de todos os pares entrada/saída possíveis.

A segunda característica elimina a hipótese da comparação exaustiva: comparar duas relações é tão pesado computacionalmente como qualquer operador relacional, o que inviabiliza a sua utilização. Além do mais, qualquer relação pode ser virtualmente infinita[17], o que faz com que haja dúvidas sobre a exequibilidade de uma tal comparação.

---

[17]Pelo menos se tomarmos em conta o tamanho dos discos actuais

A terceira característica é a mais importante: admitindo (por absurdo) que queríamos usar o método da comparação de resultados, o facto da avaliação relacional ser o que queremos evitar, não temos sequer hipótese de usar o resultado da expressão a optimizar como termo de comparação (ou então já não valia a pena optimizar, pois tínhamos o resultado).

Com isto tudo somado, o paradigma da programação genética já não parece tão sedutor. Além do mais, é extremamente difícil encontrar um operador de cruzamento útil e que mantenha a equivalência com a expressão original.

Assim, escolhemos o AG canónico, ou seja, usando cromossomas binários de comprimento fixo.

Álgebra relacional numa cadeia binária

A segunda fase foi encontrar uma codificação de uma expressão de álgebra relacional num cromossoma binário. Para isso, foi usado um pequeno "truque", semelhante ao que vimos para as permutações: em vez de usar uma representação literal da permutação, usámos uma lista de inversões da permutação e actuámos sobre essa lista. A permutação representada pela lista de inversões pode sempre ser recuperada pela "aplicação" das inversões à permutação ordenada.

No caso da optimização relacional, partimos de uma expressão em álgebra relacional, que sabemos ser uma solução do problema (embora ineficiente) e guardamos uma lista de alterações, ou *reescritas*, obtidas pela aplicação das leis de equivalência que vimos na parte sobre optimização relacional tradicional. Para obter a expressão equivalente, aplicamos a lista das transformações à expressão original. Esta lista de transformações é o nosso cromossoma.

Assim, cada gene representa uma oportunidade de transformação relacional, enquanto que o valor do gene representa qual a transformação a efectuar. Para obviar à necessidade de ter cromossomas de tamanho variável (mais difíceis de implementar e muito piores, em termos de fragmentação de memória) podemos ter um cromossoma muito grande e introduzir uma transformação nula, que corresponde a não fazer nada. O número de alelos não nulos representa, assim, o número de transformações aplicadas à expressão original.

BIBLIOTECA F.E. PORTO

Simplificação de expressões algébricas

Este método pode ser aplicado a qualquer problema, como por exemplo na simplificação de expressões algébricas: admitamos que temos as seguintes funções de transformação:

1. a + 0 = a
2. a * 1 = a
3. a * 0 = 0
4. a + b = b + a
5. a * b = b * a
6. a - a = 0

Imaginemos que temos a expressão **(a-a)*a**. Uma das soluções óptimas para este problema seria o cromossoma (6 5 3), que significa: aplicar a transformação 6, para obter 0*a, depois a 5, obtendo a*0 e por fim a 3, obtendo 0. Claro que o número dos operadores poderia ser representado por um número binário.

Este modelo foi experimentado na prática, usando como regras de reescrita as propriedades algébricas fundamentais (comutatividade, elemento absorvente, etc.) e com um cromossoma de 15 posições de 4 bits cada. Cada posição guarda o número de uma regra de reescrita a utilizar. Uma das expressões a simplificar era a seguinte, que é equivalente ao número 1:

(/ (+ (* (+ (* (+ 0 (* 1 5)) (* b (- 1 1))) x) (- 5 (+ 3 1))) 0) x)

O algoritmo convergiu para a solução óptima diversas vezes e convergiu muitas vezes para uma solução sub-óptima em que a divisão mais exterior não era eliminada. A função de adaptação foi escolhida como **f(x) = 40 + n**, em que **n** é o simétrico do número de elementos (átomos e listas) que compõem uma expressão. Para esta expressão em particular, o pior resultado é 0, enquanto que o melhor é 40. Podemos observer o gráfico de evolução de uma das vezes que o algoritmo convergiu para o óptimo:

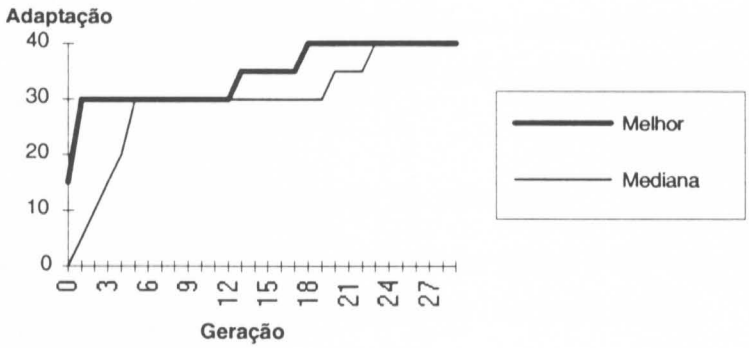

Figura 28. Evolução de um AG no problema da simplificação algébrica

Vale a pena notar que o espaço de estados para este problema já é grande, com 2^{60} soluções possíveis, uma vez que estamos a considerar cromossomas com 15 operações e 4 bits por cada operador (15 operadores possíveis e o operador nulo). O AG convergiu para a solução óptima à 18ª geração, ou seja, após ter amostrado cerca 18*40=720 soluções, uma ínfima parte do espaço completo.

Esta técnica de conversão é uma ideia que mereceria ser explorada mais em detalhe, pois parece-nos uma abordagem genérica a problemas simbólicos difíceis de modelizar, como prova automática de teoremas, derivação, integração, etc.

O motor de reescrita

Para facilitar a definição das regras de reescrita, foi desenvolvido um pequeno motor, em que é possível especificar os padrões, condições e acções a tomar, numa linguagem semelhante ao LISP [Ranito 92a]. Esta linguagem é independente do domínio, embora seja possível acrescentar novas funções em C, de forma a adaptá-la melhor a um determinado ambiente. No exemplo que demos em cima, de simplificação de expressões algébricas, as regras poderiam ser escritas do seguinte modo:

```
(+ $1 0)
     1
     (' $1)
(* $1 1)
     1
     (' $1)
(* $1 0)
     1
     0
(+ $1 $2)
     1
     (' (+ $2 $1))
(* $1 $2)
     1
     (' (* $2 $1))
(- $1 $2)
     1
     0
```

Cada regra de reescrita é constituída por três partes:

- Um *padrão*, que pode conter *variáveis* (indicadas por **$*n***, em que ***n*** pode ser um número qualquer). Se o padrão encaixar na expressão que queremos reescrever, as variáveis tomam o valor das sub-expressões correspondentes. Por exemplo, ao encaixar o padrão **(+ $1 $2)** na expressão **(+ (* 2 3) a)**, a variável **$1** passa a valer **(* 2 3)**, enquanto que a variável **$2** passa a conter **a**.
- Uma *condição*, que, se o padrão encaixar na expressão, é *substituída* e avaliada. Se o valor retornado for diferente de **0**, a reescrita será executada. A fase de *substituição* consiste em substituir as variáveis pelos seus valores, obtidos durante a fase de teste do padrão. Tomando o padrão anterior, por exemplo, se a condição fosse **(atomp $1)**, o teste falharia, uma vez que **$1** não é um átomo mas sim uma lista.
- Uma *acção*, ou valor reescrito. Se o padrão encaixar na expressão e a condição suceder, a acção é substituída e avaliada. O resultado da avaliação vai substituir a expressão original, efectuando-se assim uma reescrita. Vale a pena referir a função especial ', do CLISP, que indica **quote**, ou literalização do seu argumento, que impede a tentativa de avaliação dos argumentos, feita pelo CLISP. Esta é a forma de indicar, por exemplo, listas constantes e não representando funções.

A utilização desta mini-linguagem foi imprescindível para levar a bom termo as experiências, uma vez que evitou a escrita, sempre tediosa, de dezenas de funções especializadas de reescrita. Além do mais, uma vez que é interpretada, permite a adição e apagamento dinâmico de regras de reescrita, o que é sempre bom num ambiente experimental.

Regras relacionais de reescrita

No caso da optimização relacional, os operadores usados correspondem às regras de equivalência da álgebra relacional e são os seguintes (ver capítulo 10):

```
(prod $1 $2)
     1
     (' (prod $2 $1))

(join (== $4 $5) $2 $3)
     1
     (' (join (== $5 $4) $3 $2))

(prod $1 (prod $2 $3))
     1
     (' (prod (prod $1 $2) $3))

(sel $1 (sel $2 $3))
     1
     (' (sel $2 (sel $1 $3)))

(sel (and $1 $2) $3)
     1
     (' (sel $1 (sel $2 $3)))

(sel (== $1 $2) (prod $3 $4))
     (and (or (constp $2) (fieldq $2 $3)) (fieldq $1 $3))
     (' (prod (sel (== $1 $2) $3) $4))

(sel (== $1 $2) (prod $3 $4))
     (and (or (constp $2) (fieldq $2 $4)) (fieldq $1 $4))
     (' (prod $3 (sel (== $1 $2) $4)))

(sel (== $1 $2) (join $5 $3 $4))
     (and (or (constp $2) (fieldq $2 $3)) (fieldq $1 $3))
     (' (join $5 (sel (== $1 $2) $3) $4))

(sel (== $1 $2) (join $5 $3 $4))
     (and (or (constp $2) (fieldq $2 $4)) (fieldq $1 $4))
     (' (join $5 $3 (sel (== $1 $2) $4)))

(sel (== $2 $3) (prod $4 $5))
(and (fieldq $2 $4) (fieldq $3 $5))
     (' (join (ord $2 $3) $4 $5))

(sel (== $2 $3) (prod $4 $5))
     (and (fieldq $2 $5) (fieldq $3 $4))
     (' (join (ord $3 $2) $4 $5))
```

```
(and $1 $2)
     1
     (' (and $2 $1))
(and $1 (and $2 $3))
     1
     (' (and (and $1 $2) $3))
(join ($5 $1 $2) $3 $4)
     1
     (' (join (ord $1 $2) $3 $4))
(join ($5 $1 $2) $3 $4)
     (and (atomp $4) (btreeq $2))
     (' (join (btree $1 $2) $3 $4))
```

É de notar que, neste caso, as condições são um pouco mais complicadas do que no caso da simplificação algébrica, uma vez que também estamos a lidar com estruturas de dados muito mais complexas (relações). Foi necessário escrever algumas funções dedicadas em C, juntando-as ao CLISP, para testar, por exemplo, se um campo pertence a uma tabela ou não, etc.

12.3. População inicial e operadores genéticos

No nosso caso, a geração da população inicial não levanta problemas de maior, uma vez que qualquer cromossoma legal dá origem a um indivíduo legal. Foi arbitráriamente escolhida uma população com 30 elementos e um número máximo de gerações de 30.

Quanto à selecção, o método escolhido foi o torneio estocástico, de ordem 4, pelas razões apontadas no capítulo dedicado à selecção. O método tradicional de selecção, a roleta viciada, é eficiente "demais", fazendo com que a população seja totalmente dominada pelos melhores indivíduos das primeiras gerações, sem sequer dar tempo a obter uma boa solução. Embora convergindo mais lentamente, os resultados obtidos pelo torneio são melhores.

Além disso, adapta-se melhor a um ambiente distribuído, o que facilita eventuais experiências utilizando o NextStep, que é um ambiente distribuído por excelência. A utlização de ambientes distrtibuídos permite tirar partido da facilidade de implementação paralela dos AG [Neto93], [Bessière92] e resolver problemas em menos tempo, se houver várias máquinas acessíveis. Foram ensaiadas várias ordens para o torneio, sendo a ordem 4 a que apresenta o melhor compromisso entre competição e velocidade de convergência.

Os operadores de cruzamento e mutação, uma vez que estamos a lidar com cromossomas binários e lineares, podem ser os normais. Foram ensaiados vários valores para as probabilidades de cruzamento e de mutação, tendo sido escolhida a combinação $p_m = 0.0125$ e $p_c = 0.9$, uma vez que foi a que deu melhores resultados.

12.4. Função de avaliação

Um dos pontos fulcrais da aplicação de AG a um problema é a definição de uma função de avaliação. No nosso caso, a função retorna simplesmente uma estimativa do número de acessos ao disco a que uma determinada expressão obriga. Na primeira versão, foram deixados de fora os operadores **idx** (criação de índice) e os métodos de acesso por *hash* e directo ("pai e filho"). Além disso, foram consideradas apenas condições conjuntivas e apenas um operador de comparação, a igualdade.

Nenhuma destas omissões é fundamental, no sentido de ajudar a compreender ou a estender os conceitos aplicados. Para introduzir o resto dos operadores, bastaria alterar ligeiramente a função de adaptação e definir as regras de reescrita correspondentes. Esta opção não foi tomada por levar algum tempo e tornar o conjunto de regras muito maior e muito mais complexo.

Descodificação do cromossoma

A função de avaliação do indivíduo começa por descodificar o cromossoma, aplicando as transformações à expressão original, e depois percorre a árvore obtida, executando "abstractamente" as operações e acumulando o número de acessos ao disco. Para este cálculo, foram usadas as contas já efectuadas no capítulo 10, na determinação do custo de cada operador e seus métodos de acesso físico.

Por exemplo, imaginemos que temos a expressão inicial, não optimizada:

```
(proj (p_nome v_marca cp_desc) (sel (and (== p_bi 3) (and (==
p_carro v_matricula) (and (== p_casa c_endereco) (and (==
c_cpostal cp_codigo) (== v_matricula 1))))) (prod casas (prod
pessoas (prod viaturas cpostais)))))
```

que é equivalente à pergunta de SQL:

```
select p_nome, v_marca, cp_desc
from casas, pessoas, viaturas, cpostais
where p_bi = 3 and
p_carro = v_matricula and
v_matricula = 1 and
c_postal = cp_codigo and
p_casa = c_endereco;
```

Imaginemos também o indivíduo, já transformado num cromossoma de inteiros:

```
(5 7 8 7 12 9 0 4 7 0 10 3 3 3 6 5 9 13 2 4)
```

Se fossem aplicados os operadores de reescrita indicados no cromossoma, da esquerda para a direita, a expressão inicial seria transformada na expressão seguinte, em que já se nota alguma optimização:

```
(proj (p_nome v_marca cp_desc) (sel (== p_casa c_endereco)
(sel (== p_carro v_matricula) (join (ord c_cpostal cp_codigo)
casas (prod (sel (== p_bi 3) pessoas) (prod (sel (==
v_matricula 1) viaturas) cpostais))))))
```

A adaptação correspondente a esta expressão, pela nossa função de avaliação seria igual a -47519.207685, ou seja, teriam que ser efectuados cerca de 47519 acessos ao disco para implementar esta expressão. A adaptação de um indivíduo é, assim, o simétrico do número de acessos, uma vez que a função de adaptação deve dar maior valor aos melhores indivíduos. Uma vez que usámos o torneio estocástico como função de selecção, não nos temos que preocupar com o facto de haver adaptações negativas, o que já não aconteceria se tivéssemos usado a roleta viciada, por exemplo.

Por curiosidade, refira-se que a adaptação do cromossoma correspondente à pergunta inicial (só transformações nulas) é de cerca de 6402000200000.

12.5. Ensaios e resultados obtidos

Após alguns ensaios iniciais, que permitiram testar o funcionamento do simulador e que não vale a pena referir, foram efectuados ensaios sobre muitas perguntas relacionais, cobrindo diversos tipos de pergunta. Como não haveria tempo (nem espaço...) para descrever todos os ensaios feitos, foram escolhidas duas perguntas relacionais complexas para servir como caso de estudo para os AG. Foi criada uma base de dados fictícia, com as seguintes tabelas e campos:

Tabela(n)	Campo	Chave	Indice
cpostais (100)			
	cp_codigo	1	1
	cp_desc	0	0
pessoas (10000)			
	p_bi	1	1
	p_nome	0	0
	p_casa	0	0
	p_carro	0	0
casas (2000)			
	c_endereco	1	1
	c_quartos	0	0
	c_cpostal	0	0
viaturas (1000)			
	v_matricula	1	1
	v_marca	0	0
	v_modelo	0	0
	v_tipo	0	0

Tabela 4. Tabelas e campos da base de dados para os ensaios

Para cada tabela, está indicado o número (fictício) de registos, entre parêntesis.

Uma das perguntas escolhidas foi a que já vimos na secção anterior:

```
1)
(proj (p_nome v_marca cp_desc) (sel (and (== p_bi 3) (and (==
p_carro v_matricula) (and (== p_casa c_endereco) (and (==
c_cpostal cp_codigo) (== v_matricula 1))))) (prod casas (prod
pessoas (prod viaturas cpostais)))))
```

A outra foi escolhida para salientar alguns pontos importantes, que iremos ver mais à frente:

2)

```
(proj (p_nome v_marca cp_desc) (sel (and (== p_nome 3) (and
(== p_carro v_matricula) (and (== p_casa c_endereco) (==
c_cpostal 4200)))) (prod casas (prod pessoas viaturas))))
```

Foram efectuadas 20 tentativas para as combinações possíveis de diversos valores de probabilidade para os operadores de cruzamento (p_c = {0.5, 0.8, 0.9}) e mutação (p_m = {0.005, 0.0125, 0.05}). Como seria de esperar, valores altos de p_c e valores baixos de p_m foram os melhores, fazendo com que tenham sido escolhidos os valores já referidos de $p_c = 0.9$ e $p_m = 0.0125$. Curiosamente, o valor escolhido empíricamente para a probabilidade de mutação corresponde exactamente ao valor predito pela teoria, ou seja, uma mutação por individuo por geração.

Usando estes valores, o AG convergiu consistentemente para o óptimo na maioria dos problemas, sendo algumas das vezes atraído para soluções sub-óptimas. Vamos ver os gráficos de evolução do melhor indivíduo de cada geração para as duas perguntas escolhidas, em comparação com o mesmo número de tentativas aleatórias.

A figura seguinte mostra a evolução típica do AG, ao resolver o problema 1):

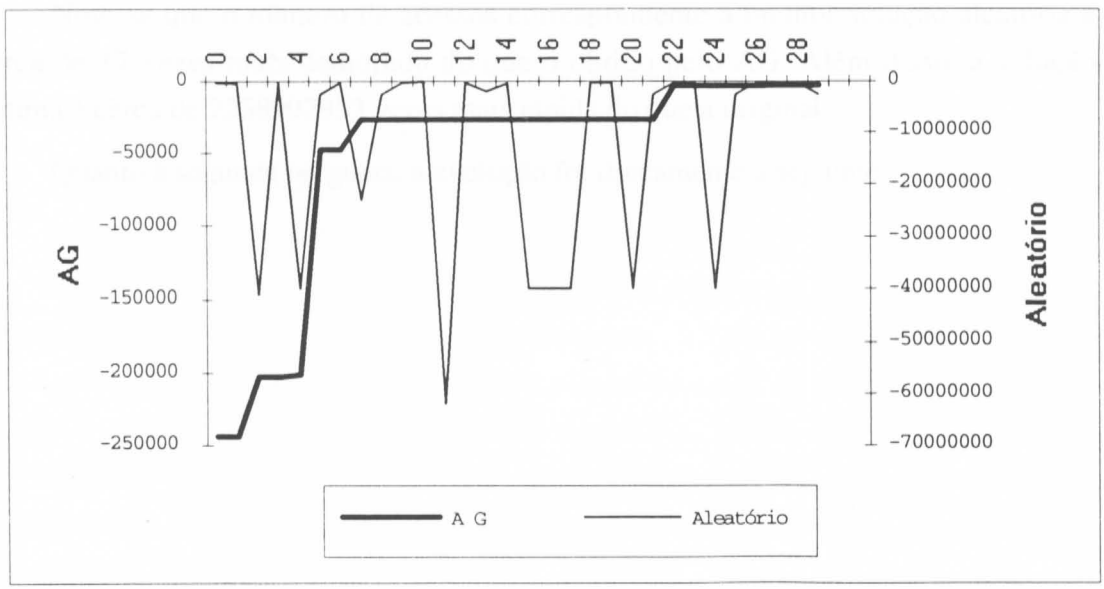

Figura 29. Evolução do AG na resolução do problema 1

Vale a pena fazer alguns comentários a esta figura. Em primeiro lugar, é necessário notar a diferença abismal entre a qualidade dos resultados obtidos pelo AG e

pelo aleatório. De facto, foi necessário colocar duas escalas diferentes para dar uma ideia da evolução. Em segundo lugar, a evolução e a sua convergência é bem nítida, sinal de que os operadores genéticos estão a cumprir o seu papel. Por fim, é de notar que o AG convergiu para o óptimo, que nesta pergunta corresponde ao indivíduo:

```
(3 1 5 6 4 5 3 2 6 5 4 7 11 4 6 10 1 8 3 15)
```

Este indivíduo, descodificado, dá origem à seguinte expressão:

```
(proj (p_nome v_marca cp_desc) (sel (== c_cpostal 4200) (join
(btree p_casa c_endereco) (sel (== p_carro v_matricula) (prod
(prod (sel (== p_bi 3) pessoas) (sel (== v_matricula 1)
viaturas)) cpostais)) casas)))
```

A adaptação respectiva é de -2834.911340, que corresponde ao óptimo para esta pergunta.

Por outro lado, a adaptação da melhor solução encontrada pelo algoritmo aleatório foi de -47519.207685, correspondendo ao cromossoma

```
(5 7 8 7 12 9 0 4 7 0 10 3 3 3 6 5 9 13 2 4)
```

que descodificado dá a seguinte expressão:

```
(proj (p_nome v_marca cp_desc) (sel (== p_casa c_endereco)
(sel (== p_carro v_matricula) (join (ord c_cpostal cp_codigo)
casas (prod (sel (== p_bi 3) pessoas) (prod (sel (==
v_matricula 1) viaturas) cpostais))))))
```

Note-se que o número de acessos correspondente à melhor solução aleatória é cerca de 17 vezes mais demorado do que o obtido pelo AG. Além disso, a solução óptima é cerca de 2258997953 vezes mais rápida do que a original.

Quanto à segunda pergunta, a evolução foi tipicamente a seguinte:

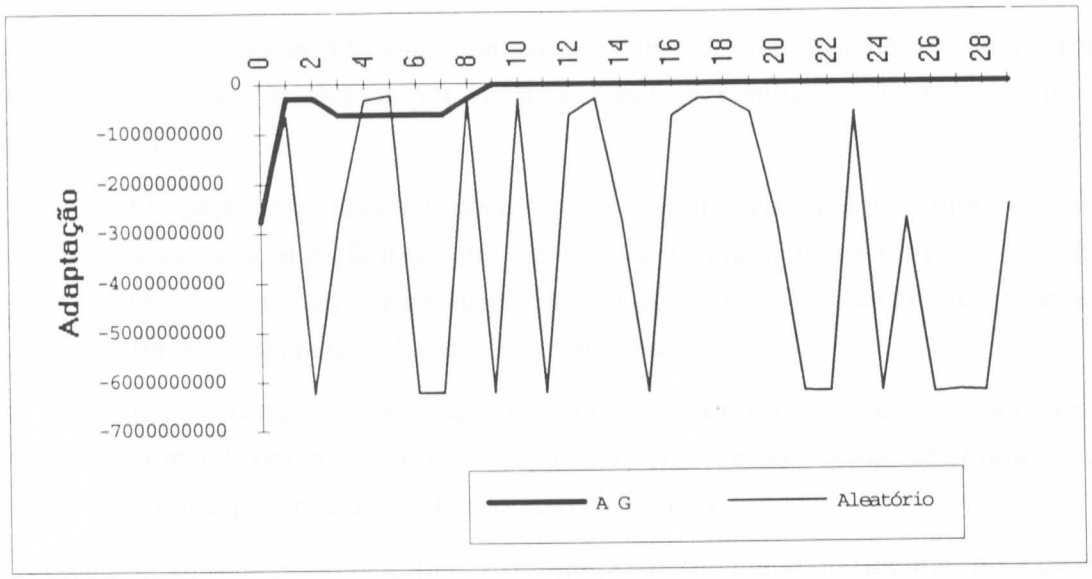

Figura 30. Evolução do AG na resolução do problema 2

Neste caso, que até parece mais simples, há muito menos hipóteses de optimização, o que dá laguma vantagem ao aleatório. No entanto, nota-se perfeitamente que o AG evolui e converge para uma solução, enquanto que o aleatório oscila entre valores muito díspares. A melhor solução encontrada pelo AG foi de -25804861.400298, a que corresponde o cromossoma (8 5 11 4 4 2 12 9 7 7 11 14 4 11 6 5 10 8 15 6) e uma expressão:

```
(proj (p_nome v_marca cp_desc) (sel (== c_cpostal 4200) (join
(ord c_endereco p_casa) casas (join (btree p_carro
v_matricula) (sel (== p_nome 3) pessoas) viaturas))))
```

Embora pareça um resultado fraco, a verdade é que esta é uma solução quase óptima (tendo em conta a função de avaliação usada...). Para ser melhor, seria preciso que a selecção (== c_cpostal 4200) fosse efectuada antes da junção. O algoritmo aleatório também não conseguiu atingir o óptimo, tendo obtido o melhor de -246965500.137893, ainda assim cerca de uma ordem de grandeza pior do que o obtido geneticamente. O custo inicial da pergunta era de 64020000000, ou seja, foi conseguida uma melhoria de cerca de 2481 vezes.

É interessante verificar que duas perguntas, muito semelhantes, acabam por ter custos muito diferentes e, ainda por cima, a que parecia mais complicada acabou por ser, de longe, a menos custosa. Esta é uma das facetas que torna a optimização relacional um problema difícil e interssante.

12.6. Comparação com métodos clássicos

Embora não tenham sido feitas comparações directas entre as soluções obtidas por via genética e soluções obtidas por via tradicional, os resultados obtidos permitem algumas notas:

- O optimizador genético pode ser interrompido em qualquer altura e obter uma solução melhor do que a expressão original, ao contrário do método clássico, cuja primeira solução é a solução final. Esta característica pode ser fundamental para ambientes em tempo-real
- O optimizador genético aglutina a fase de optimização algébrica (ou lógica) com a de optimização física, fugindo assim a alguns "vícios" de optimização em que podem cair os optimizadores clássicos
- O optimizador genético é temporalmente mais ineficiente do que o optimizador clássico, uma vez que considera maior número de soluções possíveis, por usar uma pesquisa baseada em populações. No entanto, essa desvantagem pode ser anulada se for usada uma máquina paralela.
- No optimizador genético é fácil acrescentar regras para tratar, por exemplo, informação semântica adicional proveniente de modelos como OMT [Rumbaugh91], uma vez que estas regras não correspondem a heurísticas mas sim a regras de reescrita sintáticas. No caso do optimizador clássico, seria preciso rever as heurísticas e o próprio algoritmo
- No optimizador genético, o custo dos operadores relacionais está totalmente contido na função de custo. Isto faz com que seja possível alterar esse custo sem mexer no algoritmo. Ao contrtário, no optimizador clássico, o custo dos operadores relacionais está espalhado nas heurísticas que definem o algoritmo
- As soluções encontradas pelo algoritmo genético são semelhantes às encontradas pels algoritmos clássicos, pelo menos para os exemplos ensaiados durante este trabalho, uma vez que convergiram quase sempre para óptimos e por vezes para quasi-óptimos

Por isto tudo, podemos dizer que os algoritmos genéticos constituem uma alternativa a explorar, no domínio da optimização relacional, em relação à abordagem clássica.

13. Conclusões

Neste capítulo é feita a análise do trabalho realizado, com destaque para aquele que é inovador ou pelo menos diferente do tradicional. Em seguida, tenta dar-se uma perspectiva do que poderão ser evoluções do trabalho. Por fim, fazem-se algumas considerações finais.

13.1. Trabalho realizado

O objectivo inicial de aplicar AG à Optimização Relacional depressa se transformou no objectivo de conhecer mais a fundo o campo dos AG. Daí que tenha sido investido muito tempo em experiências em outros campos que não a Optimização Relacional. Mas a verdade é que essas experiências foram fundamentais para a criação de alguma sensibilidade aos parâmetros e comportamento dos AG, indispensável para atingir o objectivo inicial.

Os resultados obtidos, em qualquer das áreas referidas em capítulos anteriores, revelaram-se melhores do que o esperado[18], inclusivamente na Optimização Relacional. Isto significa que os AG são uma técnica em que valerá a pena apostar, a médio-longo prazo.

Um dos factores que tornou esta dissertação aliciante foi a dificuldade de arranjar documentação. De facto, os AG ainda são um campo novo e a documentação não abunda, além de estar espalhada por campos tão díspares como as Redes Neuronais, a Investigação Operacional ou mesmo os Sistemas de Energia, havendo muito mais publicações sobre aplicações de AG do que sobre temas teóricos de AG. Esta é mais uma prova de que os AG são uma técnica de aplicação genérica. A escassez de documentação fez com que algumas coisas tivessem que ser "reinventadas" pelo autor, o que por vezes contribuiu para uma mais profunda compreensão dos problemas.

---

[18] Embora o autor não seja de forma alguma um pessimista

Por outro lado, o campo dos AG é tão vasto que se torna impossível, numa dissertação de Mestrado, cobrir em profundidade razoável toda a área disponível. Isto significa que foi dada mais importância aos aspectos fundamentais, até porque esta dissertação não era simplesmente um trabalho de síntese, tendo também alguma criatividade.

De todo o trabalho realizado, há que destacar as novas representações para permutações e para problemas de manipulação simbólica baseados em reescrita, que nos parecem técnicas inovadoras de uso genérico e no espírito "minimalista" dos AG tradicionais.

13.2. Trabalhos futuros e possibilidades de extensão

Para começar, o próximo passo será a completa integração do optimizador no interpretador de SQL referido no capítulo 11. Esta integração não se prevê como difícil e poderá constituir eventualmente um trabalho para uma cadeira de Licenciatura.

Outra vertente importante é a exploração de AG no âmbito de sistemas em tempo real e em sistemas em ambientes paralelos e/ou ruidosos. Parece-nos que uma das grandes forças dos AG reside precisamente na sua capacidade em se adaptarem a ambientes exigentes[19]. Este tópico poderá dar origem a teses de mestrado, inclusivé na área de *hardware*.

Outro dos campos que não foram devidamente explorados nesta tese é a aquisição de regras através da experiência, ou sistemas de conhecimento baseados em AG. Este tópico parece-nos indicado para uma tese de mestrado, pelo grande âmbito que abrangem.

Além disso, há alguns pontos interessantes que foram tocados mas que mereciam mais atenção. Os mais importantes serão talvez as representações concebidas para permutações e para manipulação simbólica. Esperamos que haja oportunidade futura para desenvolver melhor estes tópicos e entender se são mesmo coisas interessantes ou se não passam de meros "truques" para resolver um tipo de problema.

Por fim, seria interessante potenciar as ferramentas concebidas e desenvolvidas para esta dissertação, em futuras aplicações e trabalhos.

---

[19]Ou não fossem uma técnica inspirada na evolução natural

Anexo A: "GÉNIO", uma ferramenta genética genérica

Para ajudar às a realizar as experiências referidas atrás, foi criada uma ferramenta genérica para ensaio de AG. Esta ferramenta, o GÉNIO, foi desenvolvida em C, em ambiente UNIX. O seu interface em NextStep foi realizado por um aluno [Neto93], ambora exista uma versão para ser usada a partir da linha de comando. A parametrização é feita através da linha de comando (para alguns parâmetros) e/ou através de variáveis externas. Este tipo de abordagem revelou-se extremamente flexível, permitindo a ligação de módulos especializados nos problemas, sem alterar o núcleo fundamental do programa.

Esta ferramenta não irá ser descrita muito em detalhe, pois cremos que não é muito relevante para o fim desta tese, além de que uma descrição mais completa poderá ser encontrada nos anexos e nas referências.

A.1. Arquitectura do simulador

O simulador está dividido em diversos módulos independentes e redefiníveis, por forma a fomentar o reuso de soluções:

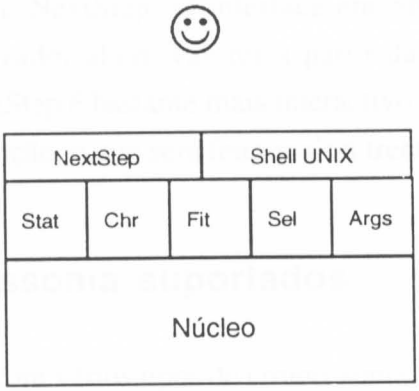

Figura 31. Arquitectura do GÉNIO

O módulo principal, o **núcleo**, é responsável pelo ciclo genético e utiliza seriços dos outros módulos, redefiníveis e parametrizáveis pelo utilizador. O utilizador não pode redefinir nem alterar funções pertencentes a este módulo, podendo apenas parametrizá-lo através do módulo **args**, que trata os argumentos da linha de comando.

O módulo **stat** é responsável pelas estatísticas e pelo apontamento de resultados em ficheiros. Possui algumas estatísticas por omissão (média, mediana, desvio-padrão, melhor, etc) e pode ser aumentada pelo utilizador através da utilização "ganchos", ou funções do utilizador chamadas em momentos críticos. Pode ser também alterado pelos argumentos da linha de comando.

O módulo **chr** concentra o conhecimento sobre os tipos de cromossomas e seus operadores. O utilizador pode configurá-lo por variáveis externas, argumentos da linha de comando e por "ganchos". O **núcleo** recorre a este módulo para determinar quais os operadores genéticos a chamar.

A selecção é controlada pelo mósulo **sel**. O utilizador pode redefinir valores para parâmetros e acrescentar funções. Pode fazê-lo através da linha de comando ou por variáveis externas.

O módulo **fit** trata da manutenção das adaptações dos indivíduos. O utilizador é forçado a redefinir a função de adaptação, através da atribuição a variáveis externas que contêm apontadores para as funções a chamar. Haverá uma função de adaptação para cada tipo de problema a resolver. Este é o módulo mais radicalmente redefinido (normalmente...) para atacar um novo problema.

Por fim, temos os módulos que asseguram a interacção com o utilizador, em UNIX simples e em ambiente NextStep. O interface em Shell de UNIX é o mais simples, deixando que o utilizador altere valores a partir da linha de invocação do comando. O interface em NextStep é bastante mais interactivo e elaborado, merecendo uma secção só para a sua descrição, o que será feito mais à frente.

A.2. Tipos de cromossoma suportados

O GÉNIO permite lidar com vários tipos de cromossoma, nomeadamente:

- Binários
- Inteiros
- Reais
- Lista

Destes tipos, o único que merece comentários é o tipo "Lista", que permite implementar programação genética. Os restantes são os tipos normais de cromossoma, que já referimos. A escolha do tipo de cromossoma determina, entre outras coisas, os operadores de mutação e cruzamento que podemos usar.

Tirando o tipo "Lista", todos os outros são cromossomas de tamanho fixo, i.e., cujo comprimento não varia ao longo de uma simulação, embora possa ser ajustado dinamicamante. Os cromossomas "Lista" podem mudar de tamanho durante a evolução. Está prevista uma extensão para acomodar cromossomas planos de tamanho variável ("messy" [Goldberg 90]).

A.3. Operadores relacionais

O GÉNIO suporta os principais operadores fundamentais, com versões para os diferentes tipos de cromossoma. As secções seguintes descrevem a sua implementação.

Selecção

Foram implementados os dois operadores mais simples para a selecção, a Roleta Viciada e o Torneio Estocástico. Este último é parametrizável no número de elementos. É extraordinariamente simples implementar (em C) novos operadores de selecção, o que

significa que poderão, em breve, ser adicionados mais. No entanto, a partir do interface em NextStep apenas é possível escolher um dos operadores de selecção já existentes.

Cruzamento e mutação

O cruzamento e a mutação dependem muito do formato do cromossoma. Assim, há versões destes operadores para cada um dos tipos de cromossoma. A mutação é mais simples e coincide, em tudo, com o que foi explicado no texto. O cruzamento tem várias opções, podendo tomar as seguintes formas:

- Cruzamento multi-ponto
- Cruzamento uniforme
- Cruzamento para listas

Evidentemente que as probabilidades de cruzamento e mutação são ajustáveis.

Adaptação

A função de adaptação é o principal ponto de contacto entre o núcleo do AG e aplicação. Assim, a definição da função de adaptação é feita através do C, atribuíndo valores a variáveis externas. No entanto, numa futura versão, está prevista a definição dinâmica, em LISP ou C, de novas funções de adaptação, recorrendo às facilidades de *linking* dinâmico orefecidas pelo NextStep.

A função de adaptação definida pelo utilizador é chamada sempre que o AG necessita de avaliar um indivíduo e deve retornar um **double**. É passado, para a função, um apontador para o indivíduo a avaliar, indivíduo esse representado por uma estrutura de C.

Anexo B: "GAS", um simulador simplificado

Apesar das facilidades oferecidas pelo GÉNIO, a optimização de Perguntas Relacionais foi desenvolvida numa outra ferramenta, menos poderosa mas mais simples e eficiente, o GAS. Esta ferramenta não possui um interface tão sofisticado mas está mais adaptada ao problema da optimização simbólica em particular.

Trabalha apenas com cromossomas binários, embora possua um módulo para conversão para cromossomas de inteiros, o que permite à aplicação que o usa ver os cromossomas como vectores de números inteiros (tipo **int** do C).

A função de adaptação tem que ser carregada juntamente com o arquivo do simulador, o que faz com que seja necessário recarregar o programa de cada vez que se troca a função de adaptação, ao contrário do que acontecia com o GÉNIO.

O interface do GAS com outra aplicação pode ser feita a dois níveis de sofisticação. Aqui será apenas descrito o nível mais alto, que foi o que foi usado para a realização do trabalho. Para uma descrição mais detalhada consultar [Ranito93] e [Neto93].

No nível mais elevado, existe apenas um arquivo, **gas.a**, que serve como ponto de comunicação entre o simulador e o programa de aplicação. Este deve ser apenas constituído por funções e nem sequer deve possuir um **main**, uma vez que o arquivo **gas.a** fornece um.

As funções a implementar pela aplicação são as seguintes:

appinit() - chamada depois da inicalização dos *defaults* e antes do processamento das opções da linha de comando, serve para a aplicação mudar os parâmetros que entender necessários para a realização da optimização.

appprint(char *c) - esta função é invocada quando o AG entende que deve imprimir um inidivíduo. É normalmente utilizada para imprimir uma descodificação do cromossoma que seja mais facilmente identificável no domínio em questão.

append() - esta função é invocada quando o AG chega ao fim, permitindo à aplicação a libertação de memória alocada ou impressão de resultados.

double appfit(char *c) - invocada quando o AG necessita do valor de um indivíduo, cujo cromossoma é **c**. Deve retornar um **double**.

Destas funções, a única que é verdadeiramente indispensável é a função **appinit**, sem a qual o AG não realiza nenhum trabalho. As restantes podem ser funções vazias (embora tenham que existir para não acontecerem erros durante a fase de *link*).

Além disso, há uma série de variáveis externas que podem ser usadas para parametrizar o funcionamento do AG. A tabela seguinte mostra as variáveis, o seu tipo e para que servem:

Variável	**Significado**
int gpsize	Numero de elementos da população
int gisize	Tamanho do cromossoma
int gngers	Numero máximo de geracoes
int gnger	Geracao actual (*)
double gpmut	Probabilidade de mutacao
double gpcross	Probabilidade de cruzamento
double gpwin	Probabilidade de vitoria do melhor no troneio
int grand	Geração de populações aleatórias?
int gtsize	Ordem do troneio
int gdiftrys	Tentativas para encontrar um elemento diferente no torneio
int gcrosspt	No. de pontos de crossover (0=uniforme)

double gpalcross	Probabilidade de trocar um alelo em cruzamento uniforme
FILE *glogfp	Apontador para o ficheiro de registo
ind gbest	Melhor de sempre (*)
ind gbestger	Melhor da geração (*)
ind gmed	Mediana da geração (*)
double gtotfit	Fitness total da população
double gavg	Fitness médio da geração
double gstdv	Desvio padrão do fitness da geração
double gintq	Intervalo inter-quartis do fitness da geração
int gverb	Verboso?
FILE *gseedf	Ficheiro com semente para a primeira população
int ghead	Coloca header no ficheiro de registo?

Tabela 5. Variáveis que a aplicação pode alterar para parametrizar o AG

As variáveis assinaladas com * não devem ter os seus valores alterados pelo utilizador, sendo apenas de consulta. A sua alteração pode provocar alterações inesperadas ao funcionamento normal do simulador.

Além destas variáveis, o utilizador pode mudar, a partir da linha de comando, uma série de opções, algumas delas sobreponíveis às variáveis. As opções, e respectivos argumentos, são as seguintes:

Opção	**Função**
-u	Crossover uniforme
-v	Verboso
-s *key*	Mostra os valores estatísticos indicados por *key*
-o *n*	Torneio de ordem *n*
-i *size*	Tamanho do cromossoma passa a ser *size*
-r	Geração de populações aleatórias em vez de usar operadores
-m *prob*	Probabilidade de mutacao passa a ser *prob*
-c *prob*	Probabilidade de crossover passa a ser *prob*
-g *n*	Numero máximo de geracoes passa a ser *n*
-h	Imprime *header* no ficheiro de registo
-p *size*	Tamanho da populacao passa a ser *size*
-f *file*	Usar o ficheiro *file* como população inicial

Tabela 6. Opções para o simulador GAS

A única opção que merece mais algum comentário é a opção **-s**, que permite a visulaização de alguns valores estatísticos no ficheiro de registo. Esta opção deve ser seguida de uma chave composta pela concatenação dos seguintes códigos:

Código	Mnemónica	Valor a mostrar no ficheiro de registo
g	ger	Número da geração
c	champ	Adaptação do melhor de sempre
b	bestg	Adaptação do melhor da geração
a	avg	Média das adaptações da geração
s	stdv	Desvio-padrão das adaptações da geração
m	med	Mediana das adaptações da geração
i	intq	Intervalo interquartis das adaptações da geração
f	fenbest	Fenótipo do melhor indivíduo da geração

Tabela 7. Chave para visualização de valores estatísticos

Vamos ver um exemplo, muito simples, em que o AG é usado para calcular o crmossoma com maior número de bits, com um cromossoma de 20 bits, uma população de 20 indivíduos, uma probabilidade de mutação de 0.01 e uma probabilidade de cruzamento de 0.8. O utilizador teria que escrever a seguinte para inicialização dos parâmetros genéticos:

```
/* Inicializacao da aplicacao */
appinit()
{
     extern int gpsize,gisize;
     extern double gpmut,gpcross;

     gpsize = 20; /* No de individuos */
     gisize = 20; /* Numero de bits do cromossoma */
     gpmut = 0.01; /* Probabilidade de mutacao */
     gpcross = 0.8; /* Probabilidade de cruzamento */
}
```

A função de finalização e de impressão poderiam ser vazias:

```
appprint(){}
append(){}
```

A função de cálculo da adaptação poderia ser algo como:

```
double appfit(c)
char *c;
{
      int i; /* Para iterar pelo cromossoma */
      double res = 0.0; /* Acumulador para o resultado */
      extern int gisize; /* Tamanho do cromossoma */

      for(i = 0; i < gisize; i++)
            res *= (double) c[i];

      return(res);
}
```

Assumindo que estas funções estavam no ficheiro **aplic.c**, para criar um executável e experimentar, bastaria dar o comando:

```
$ cc -o aplic aplic.c gas.a
$ _
```

Após a criação do executável, bastaria invocá-lo, possivelmente alterando alguns parâmetros. No nosso caso, vamos impôr uma população inicial lida do ficheiro **seed** e visualizar a média, o melhor da geração e o número da geração:

```
$ aplic -sgab -f seed
...
```

A aplicação mandaria para o écran (o ficheiro de registo por omissão) uma linha por cada geração, em que imprimiria os valores pedidos. Pararia ao fim do número de gerações que está especificado por omissão.

BIBLIOTECA F.E. PORTO

Bibliografia

[Aho87] *Compilers: Principles, Techniques and Tools*
Aho, Ullman, Sethi
1987 Addison/Wesley

[Angeline92] *Coeveolving High-level Representations*
Peter J. Angeline, Jordan B. Pollack
LAIR Technical Report
Submitted to the Proceedings of Artificial Life III

[Bach86] *The Design of the UNIX Operating System*
Maurice J. Bach
1986 Prentice/Hall International

[Bayer91] *SPLICER: a Genetic Algorithm Tool for Search and Optimization, Vol I e II*
Steven E. Bayer
1991 Stanford

[Belew90] *Evolving Networks: Using the Genetic Algorithm with Connectionist Learning*
Richard K. Belew, John McInerney, Nicol N. Schrudolph
1990 CSE Technical Report, Univ. California at San Diego

[Bessière92] *A Parallel Genetic Algorithm for the Graph Partitioning Problem*
P. Bessière, E. G. Talbi
SuperNode2 - Internal Report
1992 Grenoble

[Bonnet92] *Predição de Séries Temporais Usando Redes Neuronais*
José Maria Bonnet
Dissertação de Mestrado (em preparação)
1992 DEEC/FEUP

[Chalmers90] *The Evolution of Learning: An Experiment in Genetic Connectionism*
David J. Chalmers
1990 Morgan Kaufmann

[Cobb91] *Learning the Persistence of Actions in Reactive Control Rules*
Helen G. Cobb, John J. Grefenstette
Naval Research Laboratory - Internal Report
1991 Washington

[Cobb91a] *Is the Genetic Algorithm a Cooperative Learner?*
Helen G. Cobb
Naval Research Laboratory - Internal Report
1991 Washington

[Cunha92] *O Problema do Escalonamento na Têxtil Manuel Gonçalves*
Paulo Cunha
1992 Comunicação pessoal

[Davis91] *Handbook of Genetic Algorithms*
Lawrence Davis *et al.* (ed.)
1991 Van Nostrand Reinhold

[De Jong91] *A Formal Analysis of the Role of Multi-Point Crossover in Genetic Algorithms*
Kenneth De Jong, William M. Spears
Naval Research Laboratory - Internal Report
1991 Washington

[DeJong91a] *Learning Concept Classification Rules Using Genetic Algorithms*
Kenneth De Jong, William M. Spears
Naval Research Laboratory - Internal Report
1991 Washington

[DeJong91b] *An Analysis of the Interacting Roles of Population Size and Crossover in Genetic Algorithms*
Kenneth De Jong, William M. Spears
Naval Research Laboratory - Internal Report
1991 Washington

[Gascuel92] *A digital CMOS fully connected neural network with in-circuit learning capability and automatic identification of spurious attractors*
Jean Dominique Gascuel, Michel Weinfeld, Sami Chakroun
MIT Internal report
1992 Massachussets

[Goldberg89] *Genetic Algorithms in Search, Optimization, and Machine Learning*
David E. Goldberg
1989 Addison-Wesley

[Goldberg90] *Messy Genetic Algorithms: Motivation, Analysis, and First Results*
David E. Goldberg, Bradley Korb, Kalyanmoy Deb
1990 Complex Systems Publications

[Goldberg90] *Messy Genetic Algorithms Revisited: Studies in Mixed Size and Scale*
David E. Goldberg, Bradley Korb, Kalyanmoy Deb
1990 Complex Systems Publications

[Goldberg91] *A Note on Boltzman Tournament Selection for Genetic Algorithms and Population-Oriented Simulated Annealing*
David E. Goldberg
1990 Complex Systems Publications

[Goldberg91a] *Sizing Populations for Serial and Parallel Genetic Algorithms*
David E. Goldberg
Genetic Algorithms in Search, Optimization and Machine Learning
Summer School
1991 Stanford

[Goldberg91b] *An Investigation of Niche and Species Formation in Genetic Fucntion Optimization*
David E. Goldberg, Kalyanmoy Deb
Genetic Algorithms in Search, Optimization and Machine Learning
Summer School
1991 Stanford

[Goldberg91c] *A Comparative Analysis of Selection Schemes Used in Genetic Algorithms*
David E. Goldberg, Kalyanmoy Deb
Genetic Algorithms in Search, Optimization and Machine Learning
Summer School
1991 Stanford

[Goldberg91d] *Genetic Algorithms and Walsh Functions: Part I a Gentle Introduction*
David E. Goldberg
Genetic Algorithms in Search, Optimization and Machine Learning
Summer School
1991 Stanford

[Goldberg91d] *Genetic Algorithms and Walsh Functions: Part II, Deception and Its Analysis*
David E. Goldberg
Genetic Algorithms in Search, Optimization and Machine Learning
Summer School
1991 Stanford

[Goldberg91d] *A Note on the Non-Uniform Walsh-Schema Transform*
David E. Goldberg
Genetic Algorithms in Search, Optimization and Machine Learning
Summer School
1991 Stanford

[Goldberg91d] *Construction of High-order Deceptive Function Using Low-order Walsh Coefficients*
David E. Goldberg
Genetic Algorithms in Search, Optimization and Machine Learning
Summer School
1991 Stanford

[Goldberg91e] *Genetic Algorithms and the Variance of Fitness*
David E. Goldberg, Mike Rudnick
Genetic Algorithms in Search, Optimization and Machine Learning
Summer School
1991 Stanford

[Goldberg91f] *mGA 1.0: A Common LISP Implementation of a Messy Genetic Algorithm*
David E. Goldberg, Travis Kerzic
Genetic Algorithms in Search, Optimization and Machine Learning
Summer School
1991 Stanford

[Goldberg91g] *Real-coded Genetic Algorithms, Virtual Alphabets and Blocking*
David E. Goldberg
Genetic Algorithms in Search, Optimization and Machine Learning
Summer School
1991 Stanford

[Goldberg91h] *Genetic Algorithms as a Computational Theory of a Conceptual Design*
David E. Goldberg
Genetic Algorithms in Search, Optimization and Machine Learning Summer School
1991 Stanford

[Goldberg91j] *Zen and the Art of Genetic Algorithms*
David E. Goldberg
Genetic Algorithms in Search, Optimization and Machine Learning Summer School
1991 Stanford

[Grefenstette89] *Learning Sequential Decision Rules Using Simulation Models and Competition*
John J. Grefenstette, Connie Loggia Ramsey, Alan C. Shultz
1989 Washington

[Grefenstette91] *Conditions for Implicit Parallelism*
John J. Grefenstette
Naval Research Laboratory - Internal Report
1991 Washington

[Grefenstette91a] *Lamarckian Learning in Multi-Agent Environments*
John J. Grefenstette
Naval Research Laboratory - Internal Report
1991 Washington

[Grefenstette91b] *Deception Considered Harmful*
John J. Grefenstette
Naval Research Laboratory - Internal Report
1991 Washington

[Hofstadter79] *Godel, Escher, Bach: an Eternal Golden Braid*
Douglas R. Hofstadter
Penguin Books

[Holland75] *Adaptation in Natural and Artificial Systems*
John H. Holland
The University of Michigan Press
1975 Ann Arbor

[Hu90] *Combinatorial Algorithms*
T. C. Hu
1990 Addison-Wesley

[Iverson92] *Metabolizing Music*
Eric Iverson, Roger Hartley
1992 New Mexico

[Kirkpatrick83] *Optimization by Simulated Annealing*
S. Kirkpatrick, C. D. Gelatt, Jr., M. P. Vecchi
1983 Science

[Knuth73] *The Art of Computer Programming, Vol I - Fundamental Algorithms*
Donald E. Knuth
1973 Addison-Wesley

[Knuth73a] *The Art of Computer Programming, Vol III - Sorting and Searching*
Donald E. Knuth
1973 Addison-Wesley

[Knuth81] *The Art of Computer Programming, Vol II - Seminumerical Algorithms*
Donald E. Knuth
1981 Addison-Wesley

[Koch90] *Towards a Neurobiological Theory of Consciouness*
Francis Crick, Christof Koch
Seminars in The Neurosciences 2: 263-275(1990)

[Koza91a] *The Genetic Programming Paradigm: Genetically Breeding Populations of Computer Programas to Solve Problems*
John R. Koza
Genetic Algorithms in Search, Optimization and Machine Learning Summer School
1991 Stanford

[Koza91b] *Evolving Emergent Wall Following Robotic Behaviour Using the Genetic Programming Paradigm*
John R. Koza
Genetic Algorithms in Search, Optimization and Machine Learning Summer School
1991 Stanford

[Martí92a] *Genetically Represented Neural Network I: Representational Effects*
Leonardo Martí
Center for Adaptive Systems - Internal report
1992 Boston University

[Martí92b] *Genetically Represented Neural Network II: Searching for an Optimal Representation*
Leonardo Martí
Center for Adaptive Systems - Internal report
1992 Boston University

[Mauduit92] *Lneuro 1.0: a piece of hardware LEGO for building neural network systems*
Nicolas Mauduit
MIT Internal report
1992 Massachussets

[Neta92] *Um interpretador modular de SQL*
João Neta
Relatório de Projecto
1992 Porto

[Neto93] *Uma implementação distribuída de Algoritmos Genéticos sobre NextStep*
João Neto
Relatório de Mestrado
1993 Porto

[Oliveira88] *Apontamentos Sobre Métodos de Pesquisa para a Cadeira de Informática Aplicada II*
Eugénio Oliveira
1988 FEUP

[Pereira91] *Apontamentos Sobre Optimização Não Linear - Cadeira de Programação Matemática do Mestrado do DEEC/FEUP*
Fernando Lobo Pereira
1991 DEEC/FEUP

[Ranito,Simões92] *Um Simulador e Arquitecturas para o Reconhecimento Óptico de Caracteres - Relatório do Trabalho de Redes Neuronais do Mestrado do DEEC/FEUP*
João Ranito, Jorge Simões
1992 FEUP

[Ranito91] *Apontamentos Sobre Linguagens de Manipulação de Dados*
João Ranito
1991 FUNDETEC

[Ranito92a] *CLISP, um mini-interpretador de LISP para usar em C*
João Ranito
Relatório Interno
1992 INESC

[Ranito93] *GAS, uma biblioteca simplificada para AG*
João Ranito
Relatório Interno
1993 INESC

[Rumbaugh91] *Object-Oriented Modelling and Design*
J. Rumbaugh *et al.*
1991 Prentice-Hall

[Rumelhart88] *Parallel Distributed Processing, Vol I, II e III*
David E. Rumelhart, James L. McClelland
1988 The MIT Press

[Silva90] *Speeding Up Backpropagation*
F. M. Silva, L. B. Almeida
R. Eckmiller ed.
Advanced Neural Computers
1990 North-Holland

[Soeiro88] *Apontamentos para cadeira de Investigação Operacional*
J. Soeiro Ferreira
1988 FEUP

[Spears91] *On the Virtues of Parametrized Uniform Crossover*
William M. Spears, Keneth A De Jong
Naval Research Laboratory - Internal Report
1991 Washington

[Spears91a] *Adaptive Strategy Selection for Concept Learning*
William M. Spears, Diana F. Gordon
Naval Research Laboratory - Internal Report
1991 Washington

[Spears91b] *A Study of Crossover Operators in Genetic Programming*
William M. Spears, Vic Anand
Naval Research Laboratory - Internal Report
1991 Washington

[Spears91c] *An Analysis of Multi-Point Crossover*
William M. Spears, Kenneth A. De Jong
Naval Research Laboratory - Internal Report
1991 Washington

[Spears91d] *Using Genetic Algorithms for Supervised Concept Leraning*
William M. Spears, Kenneth A. De Jong
Naval Research Laboratory - Internal Report
1991 Washington

[Ullman81] *Algorithms and Data Structures*
Ullman, Hopcroft
1981 Addison/Wesley

[Ullman89] *Principles of Database and Knowledge-Base Systems, Vol I e II,*
Jeffrey D. Ullman
1989 Computer Science Press

[Yin91] Investigations on Solving the Load Flow Problem by Genetic Algorithms
Xiadong Yin, Noel Germany
Electric Power Systems Reasearch, 22 (1991), 151-163
1991 Elsevier

FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO
BIBLIOTECA
0000067122