O *Framework* NP-Opt e suas Aplicações a Problemas de Otimização

Alexandre de Sousa Mendes

Departamento de Engenharia de Sistemas - DENSIS
Faculdade de Engenharia Elétrica e de Computação - FEEC
Universidade Estadual de Campinas - UNICAMP
Brasil

Tese de Doutorado

Banca Examinadora

Paulo Morelato França - FEEC - UNICAMP (Presidente, orientador)
Alexandre Linhares - EBAPE - FGV/RJ
Alistair Clark - University of the West of England - Inglaterra
Christiano Lyra Filho - FEEC - UNICAMP
Fernando José Von Zuben - FEEC - UNICAMP
Vinícius Armentano - FEEC - UNICAMP

Tese defendida na Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas, SP, Brasil.

11 de Agosto de 2003

© Alexandre de Sousa Mendes 2003

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

M522f

Mendes, Alexandre de Sousa
O framework NP-Opt e suas aplicações a problemas de otimização / Alexandre de Sousa Mendes. -- Campinas, SP: [s.n.], 2003.

Orientador: Paulo Morelato França
Tese (doutorado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Otimização combinatória. 2. Algoritmos genéticos. 3. Pesquisa operacional. 4. Programação orientada a objetos (Computação). I. França, Paulo Morelato. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Resumo

Uma das maiores dificuldades em se trabalhar com otimização combinatória é que os métodos utilizados requerem procedimentos específicos para cada problema tratado. Mesmo métodos de uso geral, como os algoritmos evolutivos, precisam de operadores desenvolvidos especificamente para os problemas. Suponha, por exemplo, que alguém tenha desenvolvido um código que resolva problemas de *Sequenciamento em Máquina Simples*. Caso essa pessoa queira agora mudar e resolver um problema de *Flowshop*, terá que reformular grande parte de seu código, perdendo assim um tempo precioso devido a uma baixa reutilização das partes de seu programa. Para reduzir esse problema, desenvolvemos um ambiente de otimização, batizado de *NP-Opt*, que permite um alto grau de reutilização do código, facilitando a migração de um tipo de problema para outro. No centro desse ambiente, estão dois *algoritmos evolutivos*: um *Genético* e um *Memético*.

Nesta tese examinaremos de forma detalhada os aspectos principais do framework e ainda serão apresentados resultados de desempenho para quatro tipos de problemas NP que fazem parte do NP-Opt: *Sequenciamento em Máquina Simples*, *Flowshop Permutacional com Famílias de Tarefas*, *Gate Matrix Layout* e *Ordenamento de Genes*. Há ainda um problema não incorporado ao framework devido à necessidade de comunicação constante com pacotes externos, mas cuja implementação utilizou classes do NP-Opt: *Localização de Capacitores em Redes de Energia Elétrica*. Este problema está descrito como uma aplicação externa ao NP-Opt.

Abstract

One of the greatest difficulties in working with combinatorial optimization is that the methods require specific procedures for each problem. Even general purpose methods, like Evolutionary Algorithms, require operators developed specifically for the problems. Suppose, for example, that someone has developed a code to solve *Single Machine Scheduling* problems. If such person decides to change and address a *Flowshop* problem, most of this code will have to be changed, wasting precious time due to a low reutilization of the software components. In order to overcome this problem, we developed an optimization framework, named *Np-Opt*, which allows a high degree of code reutilization, facilitating the migration from any type of problem to another. The core of this framework is composed of two *evolutionary algorithms*: a *Genetic* and a *Memetic*.

In this thesis we will examine thoroughly all the main characteristics of the framework and present results for four different NP problems that are incorporated to the NP-Opt: *Single Machine Scheduling*, *Permutational Flowshop with families of jobs*, *Gate Matrix Layout* and *Gene Ordering*. A fifth problem, which was not included in the framework due to the need of accessing external packages, but whose implementation uses NP-Opt classes is also commented. It is named *Capacitor Allocation in Distribution Networks*, and is described as an external application of the NP-Opt.

Agradecimentos

Aos meus pais, que sempre me apoiaram em todas as decisões, especialmente as mais importantes. Além disso, a eles devo a minha formação intelectual e como ser humano em um sentido mais amplo. As inúmeras enciclopédias e visitas a museus e planetários renderam muito mais do que eu poderia imaginar. Assim, me considero em dívida eterna com ambos. Ao meu irmão, um lutador, dono de uma mente brilhante, e meu companheiro de risadas e discussões filosóficas.

À Michelle, meu amor e minha dupla inseparável.

Ao meu orientador, Paulo França, que sempre depositou uma grande confiança em meu trabalho. Sua objetividade, aliada a uma vontade muito grande de pesquisar e experimentar, foram fundamentais para que não perdesse o rumo ao longo dessa jornada.

Ao meu grande amigo, Pablo Moscato, companheiro de muitos trabalhos, cujo entusiasmo contagiante pela pesquisa foi uma inspiração para mim. Sua colaboração constante e suas idéias revolucionárias não têm preço.

Ao meu colega Vinícius Jacques Garcia, que desenvolveu ao longo de seu trabalho de mestrado as rotinas de paralelização, cedendo-as em seguida para serem incorporadas ao NP-Opt.

Ao professor Carlos Cotta, por ter me recebido em Málaga e me introduzir na área de bioinformática, um campo até então completamente novo para mim.

A todos os meus companheiros de trabalho, alunos e professores, que colaboraram de forma efetiva para a elaboração desta tese.

Aos meus colegas de laboratório do DENSIS, que fizeram das longas jornadas em frente ao computador uma rotina mais humana e prazerosa.

À Companhia Paulista de Força e Luz S.A. (CPFL), pelo auxílio na execução do capítulo referente ao problema de Localização de Capacitores em Redes de Distribuição de Energia Elétrica.

À Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), pelo apoio financeiro, fundamental para todas as atividades executadas ao longo destes quatro anos. É sem dúvida um motivo de orgulho para São Paulo e para o Brasil.

Conteúdo

1 Introdução **1**
1.1 Visão geral 1
1.2 Algoritmos evolutivos 2
1.3 Np-Opt 4

2 Algoritmos Meméticos **7**
2.1 Introdução 7
2.2 Estrutura dos algoritmos 8
2.3 Inicialização da população 11
2.4 Estrutura populacional 11
2.5 Seleção para recombinação 13
2.6 Representação e recombinação 14
2.7 Mutação 15
2.8 Busca local 16
2.9 Inserção de novos indivíduos 17
2.10 Abordagem multipopulacional 18
2.11 Resumo 20

3 O NP-Opt **23**
3.1 Introdução 23
3.2 Rodando um problema 25
3.3 Abrindo e gravando um *workspace* 30
3.4 Criando *batches* 30
3.5 Processamento distribuído 31
3.5.1 Tipos de arquitetura 31
3.5.2 Detalhes da implementação 33
3.6 Estrutura de classes 34
3.7 Incluindo um novo problema 36
3.7.1 Modificações na classe *Framework* 36
3.7.2 Modificações na classe *ProblemWindow* 37
3.7.3 Especialização da classe *InstanceWindow* 37
3.7.4 Especialização da classe *MethodWindow* 38
3.7.5 Especialização da classe *PopulationWindow* 39
3.7.6 Especialização da classe *Workspace* 40
3.7.7 Especialização da classe *ParametersWindow* 41
3.7.8 Modificações na classe *BatchMethod* 41

3.7.9 Especialização da classe *Instance* 42
3.7.10 Especialização da classe *Individual* 42
3.7.11 Especialização da classe *Crossover* 43
3.7.12 Especialização da classe *LocalSearch* 44
3.7.13 Modificações nas classes *Memetic/Genetic/MultipleStart* 45
3.7.14 Modificações na classe *Community* 45
3.7.15 Modificações na classe *Population* 45
3.7.16 Modificações na classe *Slave* 46
3.8 Adaptação do NP-Opt 46
3.8.1 Ajuste da população inicial 46
3.8.2 Ajuste da política de seleção dos pais 47
3.8.3 Ajuste da política de aceitação dos novos indivíduos . . 47
3.8.4 Ajuste do critério de convergência da população 47
3.8.5 Ajuste das políticas de migração 48
3.8.6 Ajuste das buscas locais 48
3.8.7 Ajuste da estrutura da população 49
3.9 Resumo 50

4 Localização de Capacitores **51**
4.1 Introdução 51
4.2 Descrição do problema 52
4.3 Representação utilizada 53
4.4 População inicial 54
4.5 Operador de recombinação 54
4.6 Mutação 55
4.7 Busca local 55
4.7.1 Busca local *Add/Drop* 56
4.7.2 Busca local de capacidade 56
4.7.3 Busca local *Swap* 56
4.7.4 Características especiais 56
4.8 Função de *fitness* 58
4.9 Testes computacionais 59
4.9.1 Análise da variação do preço do MWh. 61
4.9.2 Análise da variação do orçamento máximo. 61
4.9.3 Análise da variação do prazo de amortização 62
4.10 Resumo 63

5 Gate Matrix Layout **65**
5.1 Introdução 65
5.2 Descrição do problema 66
5.3 Representação, população inicial, recombinação e mutação . . . 67
5.4 Busca local 68
5.5 Testes computacionais 70
5.5.1 Abordagem unipopulacional x multipopulacional 70
5.5.2 Eficiência do algoritmo memético 73
5.6 Resumo 75

6 Sequenciamento em Flowshop **77**
6.1 Introdução 77
6.2 Descrição do problema 78
6.3 Problemas testados 80
6.4 Representação e operadores de recombinação, mutação e busca local 81
6.5 Testes computacionais 83
6.6 Resumo 86

7 Sequenciamento em Máquina Simples **87**
7.1 Introdução 87
7.2 Descrição do problema 88
7.3 Criação das instâncias 89
7.3.1 Parâmetros variáveis 91
7.4 Representação, população inicial, recombinação, mutação e busca local. 92
7.5 Testes computacionais 93
7.5.1 Versão unipopulacional 93
7.5.2 Versão multipopulacional 94
7.6 Resumo 95

8 Ordenamento de Genes **97**
8.1 Introdução 97
8.2 Descrição do problema 98
8.3 Representação 99
8.4 População inicial, recombinação e mutação 100
8.5 Busca local 102
8.6 Migração e processamento distribuído 103
8.7 Testes computacionais 103
8.7.1 Instâncias utilizadas 103
8.7.2 Tamanho das janelas móveis 103
8.7.3 Aplicação das buscas locais 105
8.7.4 Número de populações 106
8.7.5 Testes com processamento distribuído 107
8.7.6 Comparação visual das soluções 111
8.8 Resumo 112

9 Conclusão **115**

A Artigos publicados ou submetidos **119**

Bibliografia **121**

Lista de Figuras

1.1 Adaptação do bico do tentilhão nas Ilhas Galápagos. 4

2.1 Estrutura populacional adotada no NP-Opt. 12
2.2 Diagramas das políticas de migração. 20

3.1 Tela do NP-Opt rodando sob o sistema operacional *Windows*. . 24
3.2 Janela inicial do NP-Opt. 25
3.3 Janela de seleção do tipo de problema. 25
3.4 Janela de seleção da instância. 26
3.5 Janela geral de seleção de arquivo. 26
3.6 Janela de seleção dos métodos. 26
3.7 Janelas de ajuste das populações e da política de migração. . . 27
3.8 Janela de parâmetros do *multiple start*. 28
3.9 Janela de execução. 29
3.10 Janela de processamento paralelo. 29
3.11 Janelas de saída. 30
3.12 Janela de ajuste dos *batches*. 31
3.13 Arquitetura mestre-escravo. 32
3.14 Implementação da arquitetura mestre-escravo. 34
3.15 Estrutura de classes do NP-Opt. 35
3.16 Especialização de classes. 35

4.1 Codificação do cromossomo para o PLC. 53
4.2 Operador de recombinação. 55
4.3 Soluções para a instância de 9 seções. 60

5.1 Instância do problema de Gate Matrix Layout. 66
5.2 Block Order Crossover (BOX). 68
5.3 Identificação de uma porta crítica no problema GML. 69
5.4 Dados de saída do GML. 71

6.1 Codificação do cromossomo para o problema de *Flowshop*. . . . 81
6.2 Recombinação para o problema de *Flowshop*. 82

7.1 Caso errôneo da transformação PCVA/SMS. 90
7.2 Operadores de recombinação para o problema de SMS. 92

8.1 Codificação do cromossomo para o problema de Ordenamento de Genes. 100
8.2 Operador de recombinação para o Ordenamento de Genes. . . . 101
8.3 Influência dos diferentes tamanhos de janelas na instância *Fibroblast*. 104
8.4 *Speedup* do algoritmo memético para diferentes números de populações. 107
8.5 a. *Speedups* resultantes; b. Melhora do *fitness* em comparação com a abordagem sequencial. 110
8.6 Soluções da instância *Herpes*. 112
8.7 Soluções da instância *Linfoma*. 113
8.8 Soluções da instância *Fibroblast*. 114
8.9 Soluções da instância *Yeast*. 114

Lista de Tabelas

4.1 Dados dos capacitores utilizados. 53
4.2 Resultados das redes pequenas de Gallego et al. (2001). 59
4.3 Dados da rede de distribuição. 60
4.4 Análise da variação do preço do MWh. 61
4.5 Análise da variação do orçamento anual máximo. 62
4.6 Análise da variação do prazo de amortização. 62

5.1 Dados das instâncias de GML. 71
5.2 Resultados da instância V4470. 71
5.3 Resultados da instância X0. 72
5.4 Resultados da instância W3. 72
5.5 Resultados da instância W4. 72
5.6 Número de avaliações do algoritmo memético. 73
5.7 Número de avaliações do *microcanonical optimization*. 74
5.8 Estatísticas do algoritmo memético. 75

6.1 Resultado dos algoritmos para os problemas LSU. 83
6.2 Resultado dos algoritmos para os problemas MSU. 84
6.3 Resultado dos algoritmos para os problemas SSU. 85

7.1 Parâmetros das instâncias originais do PCVA. 92
7.2 Resultados do algoritmo memético unipopulacional. 94
7.3 Resultados do algoritmo memético multipopulacional. 95

8.1 Instâncias utilizadas nos testes computacionais. 104
8.2 Resultados médios para diferentes intensidades de busca local. . 106
8.3 Dados das instâncias utilizadas no processamento distribuído. . 109
8.4 Descrição da carga de trabalho dos *escravos*. 109
8.5 *Speedups* máximos teóricos para as instâncias. 110

Capítulo 1

Introdução

1.1 Visão geral

Um dos principais problemas das metaheurísticas é que elas são especializadas para cada tipo de problema. Tome por exemplo um algoritmo genético que esteja sendo utilizado para resolver um problema de sequenciamento do tipo *Flowshop*. Provavelmente, caso o pesquisador deseje utilizar o mesmo código para resolver um problema de Sequenciamento em Máquina Simples, terá que refazer boa parte do código, especialmente se não estiver utilizando recursos de programação orientada a objetos.

Este trabalho tem por objeto central o desenvolvimento de um ambiente de otimização capaz de lidar com uma ampla gama de problemas com o máximo de reutilização de códigos. Para tanto, a programação é inteiramente baseada na orientação a objetos e no uso da linguagem Java, que permite uma maior portabilidade. Com isso, o usuário tem disponível para si uma ferramenta de otimização com características bastante complexas, e cuja utilização é relativamente fácil para um programador Java de bom nível. A adição de novos problemas e/ou métodos de otimização requer poucas mudanças no código, fato esse que se refletiu na quantidade de problemas presentes no software. Além disso, a qualidade dos resultados indica que os recursos utilizados são bastante adequados, tendo sido igual ou superior a métodos pré-existentes.

No presente momento há cinco tipos de problemas disponíveis no NP-Opt. Eles são Sequenciamento em Máquina Simples, Máquinas Paralelas e Flowshop; Gate Matrix Layout e Ordenamento de Genes. Os três primeiros problemas pertencem à área de Sequenciamento da Produção. O problema de Gate Matrix Layout tem relação com o desenho de circuitos elétricos e o último já pertence à área de Bioinformática.

Além dos cinco problemas presentes no software, esta tese faz referência a outro problema tratado utilizando as classes do NP-Opt, mas que não foi incorporado ao mesmo porque era necessária a utilização de pacotes externos e a comunicação entre a linguagem Java e os pacotes era consideravelmente complexa. Este problema diz respeito à alocação de capacitores em uma rede de distribuição, e pertence à área de Engenharia Elétrica, ou mais especificamente,

Planejamento de Redes Elétricas.

Sobre as características do software, ressaltamos que no momento estão disponíveis três métodos para todos os problemas. Eles são um algoritmo genético, um memético e uma estratégia de *multiple start*. Para o problema de Sequenciamento em Máquina Simples há ainda uma heurística construtiva chamada *ATCS* que é utilizada como base para comparação de desempenho. Os algoritmos evolutivos utilizam recursos importantes como população hierarquicamente estruturada, múltiplas populações com migração segundo um modelo de ilhas, múltiplas buscas locais e reduções de vizinhanças.

O software conta também com o uso de processamento distribuído. Essa etapa do estudo foi feita com a colaboração do estudante de doutorado *Vinícius Jacques Garcia*, do DENSIS-UNICAMP, que já havia elaborado as classes para efetuar o processamento distribuído, reduzindo o trabalho à adaptação das mesmas à estrutura do NP-Opt, que faz uso intenso de recursos gráficos. O resultado foi excelente e permitiu que o processo mais custoso do ponto de vista computacional - a busca local - fosse distribuído por um número indefinido de máquinas dentro de qualquer rede *Unix/Linux* rodando sob o protocolo de comunicação *TCP/IP*. Isto permitiu um salto quantitativo na qualidade das soluções obtidas e da eficiência do software.

Esta tese está estruturada da seguinte forma. Neste capítulo introdutório faremos, nas seções seguintes, alguns breves comentários sobre os fundamentos dos algoritmos evolutivos, em especial dos algoritmos genéticos/meméticos e o NP-Opt. No capítulo 2, serão descritos em detalhe os aspectos principais dos algoritmos evolutivos utilizados neste trabalho. No capítulo 3 trataremos da codificação do software NP-Opt. Serão analisadas as estruturas das classes e os porquês das escolhas de determinadas codificações em detrimento de outras. A maneira como o processamento distribuído atua será também abordada nesse capítulo. Terminados esses três capítulos iniciais, entraremos na parte das aplicações, onde cada capítulo tratará de um problema abordado, em um total de cinco capítulos. Por fim, faremos uma discussão dos resultados obtidos e possibilidades de trabalhos futuros. No Apêndice, selecionamos alguns trabalhos publicados que dizem respeito aos assuntos tratados na tese.

1.2 Algoritmos evolutivos

Os Algoritmos Evolutivos fazem parte da grande área do conhecimento chamada Inteligência Artificial. Eles buscam na natureza a inspiração para a criação de mecanismos de busca de soluções. Entre esses mecanismos podemos citar recombinação, mutação, seleção natural, competição, adaptabilidade, aprendizado, entre outros. Nos algoritmos genéticos/meméticos temos esses operadores atuando na tentativa de encontrar boas soluções para o problema abordado. A base dos métodos são as analogias solução/indivíduo e problema/ambiente. Quando o algoritmo começa, temos soluções em geral ruins para o problema, que seriam equivalentes a indivíduos pouco adaptados ao ambiente. À medida em que os mecanismos evolutivos atuam, as soluções se tornam melhores e ao fim de um número suficiente de gerações, espera-se ter uma população constituída por soluções de alta qualidade para o problema. Ou seja, os indivíduos

que inicialmente eram pouco adaptados ao meio, evoluíram e se tornaram bem adaptados, segundo a linha filosófica da Teoria da Evolução.

Os algoritmos evolutivos, em especial os algoritmos genéticos, tiveram seu primeiro impulso com a publicação do livro intitulado 'Adaptation in Natural and Artificial Systems', por John Holland em 1975 [37]. No entanto, somente com a proliferação dos computadores em centros de pesquisas é que a pesquisa científica nessas áreas teve seu crescimento acelerado, que permanece até os dias atuais. A necessidade de sistemas computacionais poderosos é fundamental, pois todos os estudos se baseiam na simulação de sistemas naturais, o que requer esforço computacional proporcional à tarefa sendo conduzida. Por volta dos anos 80, uma nova classe de algoritmos genéticos, mais 'inteligentes', começaram a aparecer na literatura especializada. A principal idéia por trás desses novos algoritmos era fazer uso de outros tipos de conhecimento, ou outros métodos de solução já disponíveis para o problema sendo tratado. Como consequência direta, boa parte das analogias evolutivas presentes nos algoritmos genéticos originais se perderam nesses novos métodos. O uso de outros tipos de operadores, que não derivados de analogias genéticas, passou a ser encarado como a incorporação da noção de aprendizado nos métodos, criando o conceito de *evolução cultural*. É notório que as noções de aprendizado, conhecimento, cultura e instinto são fundamentais em qualquer população cujos membros se relacionam uns com os outros. Em culturas mais avançadas, o papel do aprendizado e do conhecimento é bem mais importante do que o puramente genético na questão de definir a adaptabilidade de um indivíduo. Em 1976, Richard Dawkins publicou o livro 'The selfish gene' [15], onde definiu o termo *meme* como sendo a menor unidade de conhecimento que pode ser transmitida entre indivíduos. Em 1989, por sua vez, Moscato [58] definiu o termo *algoritmos meméticos* para métodos que empregam a noção de conhecimento e aprendizado em adição - ou não - aos métodos puramente genéticos.

Neste trabalho, além dos operadores evolutivos, utilizamos ainda conceitos de migração e de múltiplas populações. Em ambientes reais, as espécies evoluem agrupadas em populações, cujas fronteiras são demarcadas em geral por características especiais, como distância ou barreiras geográficas. O papel de populações isoladas (ou quase isoladas) é muito importante. Tomemos por exemplo as Ilhas Galápagos, objetos de inspiração e estudo para o biólogo inglês Charles Darwin, quando a bordo do HMS Beagle em sua viagem ao redor do mundo [14]. O arquipélago das Galápagos é composto por um conjunto de ilhas separadas por vários quilometros de oceano e que foram inicialmente colonizadas por uma espécie de pássaro chamada tentilhão. No começo todos os pássaros possuíam características fenotípicas semelhantes, e compartilhavam o mesmo *genetic pool* - ou conjunto de informações genéticas. Mas à medida em que o processo evolutivo acontecia, os grupos presentes em cada ilha começaram a se diferenciar, adaptando-se às características particulares de cada uma delas [73]. De fato, a maior mudança se deu no formato dos bicos, pois os alimentos presentes em cada ilha eram diferentes. Na Figura 1.1 estão os desenhos originais feitos por Darwin de quatro espécies de tentilhão, de um total de 13 identificadas e originárias de um mesmo descendente comum. A seleção natural e a adaptação contínua levaram, ao final de um número suficiente de gerações,

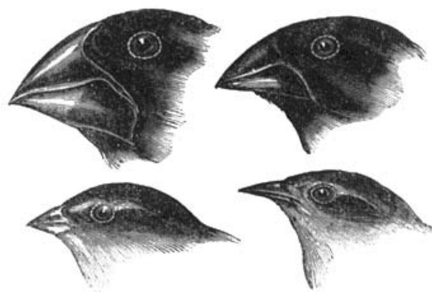


Figura 1.1: *Adaptação do bico do tentilhão nas Ilhas Galápagos.*

a diferentes espécies. De fato, mesmo que as ilhas fossem idênticas, possuindo o mesmo tipo de alimento, espécies diferentes ainda poderiam surgir devido ao relativo isolamento e ao conceito de *genetic drift* - ou deriva genética [73].

O conceito de *genetic drift* diz que se duas populações idênticas são separadas e submetidas às mesmas condições de ambiente, devido à natureza aleatória dos processos evolutivos, elas deverão seguir caminhos diferentes e se tornar espécies diferentes após uma grande quantidade de gerações. Seguindo este raciocínio, é muito comum nos algoritmos evolutivos que, devido à natureza aleatória dos métodos, sejam obtidas soluções finais diferentes quando o mesmo algoritmo é executado seguidamente. Apesar de ser visto como um problema, ou uma falta de consistência, esse fenômeno pode ser muito útil quando múltiplas populações são utilizadas. Com várias populações evoluindo em paralelo e seguindo caminhos evolutivos diferentes, porções maiores do espaço de soluções podem ser avaliadas. Além disso, qualquer informação genética importante incorporada a uma certa população pode ser espalhada para as outras através da migração de indivíduos, criando uma sinergia bastante positiva. Esse mecanismo torna as buscas paralelas potencialmente mais poderosas que abordagens unipopulacionais [9, 10].

1.3 Np-Opt

O ambiente de otimização NP-Opt segue a filosofia da programação orientada a objetos. Ele busca reduzir o trabalho de programação, tornando a expansão do sistema mais fácil, com novos componentes sendo adicionados de maneira modular. Assim, quando o usuário decide incluir um novo problema, seu trabalho será o de criar algumas poucas novas classes de objetos, que serão utilizadas somente para o seu problema em particular. As mudanças nas partes de uso geral do NP-Opt são mínimas. Da mesma forma, a adição de novos métodos ou recursos é igualmente simples.

A linguagem utilizada é o Java, criado pela Sun Microsystems [41]. A escolha foi influenciada principalmente pelo alto grau de portabilidade da mesma. Além disso, a linguagem Java tem disponível uma ampla gama de recursos para

uso distribuído em redes de computadores. Isso foi fundamental pois uma das idéias originais era a de se utilizar processamento distribuído. Muitas vezes a complexidade dos problemas tornava o processamento sequencial proibitivo, fato esse que constantemente alimentava a vontade de utilizar uma abordagem alternativa, onde fosse possível lançar mão de recursos computacionais mais poderosos.

A estrutura do NP-Opt é composta por quatro grandes blocos. O primeiro é o de interface com o usuário. Optamos por um sistema inteiramente gráfico, que tornasse a comunicação usuário/software mais simples e ao mesmo tempo eficiente. Existem diversas janelas de diálogo que definem o tipo de problema, os métodos, parâmetros, arquivos de dados, entre outros. O uso é bastante iterativo e um usuário mediano se sentirá à vontade após poucas sessões de uso. O segundo bloco é composto pelos métodos. É o 'motor' do Np-Opt e possui diversas características interessantes do ponto de vista evolutivo. O fato do Np-Opt ter obtido resultados fortes em todos os problemas tratados até o momento, utilizando sempre o mesmo algoritmo memético como base é bastante significativo. Nesse segundo bloco estão os operadores genéticos e as rotinas e definições de cada problema tratado, como instância, indivíduo, operadores de mutação, recombinação, busca local, entre outros. O terceiro bloco é o de processamento distribuído, que possui as rotinas necessárias para a distribuição das tarefas. Por fim, o quarto bloco é composto pela interface de saída, onde são exibidos os resultados encontrados pelos métodos.

Sobre o processamento distribuído, a paralelização de tarefas só é vantajosa se o trabalho a ser distribuído for responsável pela maior parte do esforço computacional do método. No caso dos algoritmos meméticos, a escolha logicamente recaiu sobre a busca local, que em grande parte das vezes chega a ser responsável por 95% de todo o tempo computacional consumido pelo método. A distribuição segue um modelo *mestre-escravo*, onde o computador mestre fica responsável pelo processamento geral do NP-Opt e distribui as buscas locais para os computadores escravos. À medida em que os computadores escravos vão terminando suas tarefas eles enviam o resultado de volta para o computador mestre e aguardam o recebimento de uma nova tarefa. Esses aspectos serão discutidos no Capítulo 2 em detalhe.

Esta página foi deixada intencionalmente em branco.

Capítulo 2

Algoritmos Meméticos

2.1 Introdução

Os algoritmos meméticos podem ser vistos como uma extensão dos algoritmos genéticos, que levam em conta, além da evolução do ponto de vista genético, a evolução cultural. O termo memético deriva de *meme*, que é a menor unidade de conhecimento que pode ser transmitida de um indivíduo a outro. No caso do algoritmo utilizado no NP-Opt, a evolução cultural é representada por operadores de busca local associados a uma estratégia de *hill-climbing*. O *hill-climbing* determina que somente modificações que melhorem a adaptabilidade do indivíduo devem ser aceitas no processo de busca local. Em todos os problemas tratados, a busca local teve um papel fundamental na obtenção de soluções de boa qualidade, porém com diferentes graus de importância. Como será mostrado nos capítulos seguintes, esta é a etapa mais crítica na formulação de um algoritmo memético e a que mais influencia o desempenho global do algoritmo. Como a busca local é de longe a parte computacionalmente mais pesada do algoritmo, no caso dela ser mal elaborada, o desempenho global será severamente prejudicado. Devido a isso, em todas as aplicações do NP-Opt, prestamos especial atenção a esse tema. Além do mais, buscas locais bem desenvolvidas podem ser úteis mesmo para outros tipos de algoritmos que se valem da definição de vizinhanças, como *Busca Tabu* [28], *GRASP* [22], *Scatter Search* [27], entre outros.

O desenvolvimento de buscas locais é um assunto relativamente complexo e deve levar em conta o *trade-off* entre poder de busca e tempo computacional. Em vários casos foi necessária a utilização de políticas de redução de vizinhança, pois a busca local se tornara demasiadamente pesada. Os casos mais interessantes de buscas locais foram gerados nos problemas de Ordenamento de Genes, Localização de Capacitores e Gate Matrix Layout. Eles foram justamente os que mais demandaram estudos a respeito e consequentemente os melhores resultados em comparação com outros métodos.

Outro operador em que nos concentramos foi o de recombinação. A principal lição que tiramos após aplicações de vários operadores em uma ampla gama de problemas é que os melhores sempre possuem um certo grau de aleato-

riedade. Recombinações determinísticas levam a uma rápida perda de diversidade, com uma consequente redução no desempenho global. Por incrível que pareça, operadores clássicos como o OX [29] e suas variantes foram os que obtiveram melhores resultados. A experiência também nos mostrou que não importa quão boa seja a estratégia de busca local, a recombinação sempre terá um papel fundamental na dinâmica do algoritmo. Isso ficou claro nas comparações feitas entre o algoritmo memético e o método de *multiple start*.

O operador de mutação também foi tratado, mas com uma importância menor que os dois anteriores. A principal função da mutação é manter a diversidade em um nível satisfatório entre os indivíduos. Porém, no nosso caso utilizamos um critério de verificação de diversidade que ao notar qualquer sinal de homogeneidade efetuava um *restart* - uma reinicialização - na população como um todo. Talvez esse fato tenha reduzido a importância de tal operador no algoritmo memético.

Outros aspectos importantes do algoritmo são abordados, como o uso de populações hierarquicamente estruturadas, migração de indivíduos, além de processamento paralelo. Neste trabalho são ainda discutidos vários aspectos que influenciam a dinâmica do algoritmo memético, entre eles o critério de aceitação de novos indivíduos; elitismo; a intensidade da recombinação e da aplicação da busca local nos indivíduos; quando e como efetuar a reinicialização da população; tipos de políticas de migração e número de populações. Esses temas serão abordados neste e nos próximos capítulos. A seguir analisaremos um pseudo-código do algoritmo memético utilizado nos problemas. Alguns problemas utilizaram pequenas variações do mesmo, que serão devidamente explicadas.

2.2 Estrutura dos algoritmos

Nesta seção vamos explicar a estrutura dos algoritmos memético e genético presentes no NP-Opt. O algoritmo genético básico é composto por três fases principais: inicialização da população; recombinação/mutação e inserção dos novos indivíduos. Essas etapas se subdividem em outras, entre elas a avaliação e reestruturação da população, seleção de indivíduos para recombinação, verificação de convergência/reinicialização dos indivíduos e migração. Os pseudo-códigos mostrados a seguir representam os algoritmos genéticos uni e multipopulacionais, além de duas variantes do algoritmo memético. A diferença entre os algoritmos genéticos e meméticos reside na aplicação da busca local apenas, e como a estratégia de aplicação da busca local varia de um problema para outro, ela será descrita mais profundamente nas próximas seções.

Os pseudo-códigos dos algoritmos meméticos são muito similares aos dos algoritmos genéticos. As únicas diferenças estão nas linhas que fazem referência ao comando **buscaLocal**. Nos dois primeiros pseudo-códigos, chamados *algGeneticoUniPopulacional* e *algGeneticoMultiPopulacional*, não há nenhuma chamada para o operador de busca local. A pressão evolutiva se concentra desta forma apenas nos operadores de recombinação e de mutação. Nesses dois pseudo-códigos podemos identificar claramente três partes distintas. Primeiramente, a parte de inicialização, com a inicialização propriamente dita dos in-

divíduos, a avaliação do *fitness* e a estruturação da população. Em seguida, o laço da geração dos novos indivíduos, com a seleção dos pais, a recombinação, a mutação, a avaliação e a inserção. A terceira parte engloba a estruturação da população e a verificação da convergência - e a migração, no caso do algoritmo genético multipopulacional. A diferença entre o primeiro algoritmo e o segundo está no uso ou não de múltiplas populações. Do ponto de vista computacional, essa mudança consiste em rodar o primeiro método *numberOfPopulations* vezes em sequência, armazenando as populações finais. Quando todas elas tiverem convergido, efetua-se a migração e a reinicialização, repetindo então o processo caso o critério de parada ainda não tenha sido satisfeito.

```
Method algGeneticoUniPopulacional;
begin
   inicializaPopulação(pop);
   avaliaFitnessPopulação(pop);
   estruturaPopulação(pop);
   repeat
      for i = 1 to numberOfRecombinations do
         selecionaPais(individuoA, individuoB) ⊆ pop;
         novoInd = recombina(individuoA, individuoB);
         if (efetuaMutação novoInd) then novoInd = mutação(novoInd);
         avaliaFitnessIndivíduo(novoInd);
         inserePopulação(novoInd, pop);
      end
      estruturaPopulação(pop);
      if (populaçãoConvergiu pop) then inicializaPopulação(pop);
   until (condiçãoParada);
end
```

```
Method algGeneticoMultiPopulacional;
begin
   repeat
      for i = 1 to numberOfPopulations do
         inicializaPopulação(pop(i));
         avaliaFitnessPopulação(pop(i));
         estruturaPopulação(pop(i));
         repeat
            for j = 1 to numberOfRecombinations do
               selecionaPais(individuoA, individuoB) ⊆ pop(i);
               novoInd = recombina(individuoA, individuoB);
               if (efetuaMutação novoInd) then novoInd = mutação(novoInd);
               avaliaFitnessIndivíduo(novoInd);
               inserePopulação(novoInd, pop(i));
            end
            estruturaPopulação(pop(i));
         until (populaçãoConvergiu pop(i));
      end
      for i = 1 to numberOfPopulations do
         efetuaMigração pop(i);
      end
   until (condiçãoParada);
end
```

Os dois pseudo-códigos a seguir, *algMemeticoBuscaLocalGeral* e *algMemeticoBuscaLocalRestrita*, se referem a duas variantes do algoritmo memético unipopulacional. A extensão multipopulacional deles é feita de forma análoga

ao algoritmo genético multipopulacional. No primeiro método, a chamada da busca local está localizada dentro do laço da geração dos novos indivíduos. Isso faz com que todos os novos indivíduos criados passem pelo processo de busca local. Essa versão foi utilizada nos problemas em que as vizinhanças são relativamente pequenas, permitindo a aplicação do operador em um grande número de indivíduos sem prejudicar a dinâmica do algoritmo.

No segundo pseudo-código, a chamada da busca local está localizada depois da convergência da população, o que faz com que ela só seja aplicada a poucos indivíduos. Essa versão é mais apropriada para buscas locais muito pesadas, que consomem um tempo computacional excessivo. Conforme dito anteriormente, as ocasiões em que são prefevíveis utilizar uma ou outra estratégia serão discutidas nos capítulos seguintes, assim como o porquê de cada escolha.

```
Method algMemeticoBuscaLocalGeral;
begin
   inicializaPopulação(pop);
   avaliaFitnessPopulação(pop);
   estruturaPopulação(pop);
   repeat
      for i = 1 to numberOfRecombinations do
         selecionaPais(individuoA, individuoB) ⊆ pop;
         novoInd = recombina(individuoA, individuoB);
         if (efetuaMutação novoInd) then novoInd = mutação(novoInd);
         novoInd = buscaLocal(novoInd);
         avaliaFitnessIndivíduo(novoInd);
         inserePopulação(novoInd, pop);
      end
      estruturaPopulação(pop);
      if (populaçãoConvergiu pop) then inicializaPopulação(pop);
   until (condiçãoParada);
end
```

```
Method algMemeticoBuscaLocalRestrita;
begin
   inicializaPopulação(pop);
   avaliaFitnessPopulação(pop);
   estruturaPopulação(pop);
   repeat
      for i = 1 to numberOfRecombinations do
         selecionaPais(individuoA, individuoB) ⊆ pop;
         novoInd = recombina(individuoA, individuoB);
         if (efetuaMutação novoInd) then novoInd = mutação(novoInd);
         avaliaFitnessIndivíduo(novoInd);
         inserePopulação(novoInd, pop);
      end
      estruturaPopulação(pop);
      if (populaçãoConvergiu pop) then
         buscaLocal(setOfIndividuals ⊆ pop);
         avaliaFitnessPopulação(pop);
         estruturaPopulação(pop);
         inicializaPopulação(pop);
      end;
   until (condiçãoParada);
end
```

2.3 Inicialização da população

Nesta seção, vamos analisar a primeira parte do algoritmo, que trata da inicialização da população. Nos problemas abordados, os indivíduos iniciais são todos aleatórios. Em alguns casos, como nos problemas de sequenciamento, a aleatoriedade é total, sem nenhum mecanismo especial na geração dos valores. Em outros casos, a aleatoriedade é tendenciosa, como para o problema de Localização de Capacitores. Nesse problema, o tamanho inicial dos capacitores é aleatório, mas está mais concentrado em uma região que torna a solução mais realista. Além disso, a quantidade de capacitores inicialmente alocados é condizente com o que empiricamente se espera de uma rede de distribuição real. Isso faz com que o algoritmo consiga obter soluções de melhor qualidade mais rapidamente. No caso do problema de Ordenamento de Genes, a população inicial possui três elementos que são gerados através de heurísticas construtivas de agrupamento. Em problemas pequenos - contendo até pouco menos de 400 genes - o algoritmo memético conseguiu obter soluções de alta qualidade e em pouco tempo computacional diretamente a partir de populações iniciais formadas por indivíduos inteiramente aleatórios. No entanto, para problemas maiores, soluções iniciais heurísticas se mostraram fundamentais para o sucesso do método.

Apesar de não termos adotado soluções iniciais heurísticas em todos os algoritmos meméticos, acreditamos que seu uso é em geral benéfico, especialmente quando a busca local utilizada é pouco poderosa em relação à complexidade do espaço de busca. Caso o algoritmo usado seja um genético puro, a adoção de soluções iniciais não-aleatórias é primordial para a obtenção de bons resultados, pois a evolução obtida apenas através da atuação dos operadores de recombinação e de mutação é bastante lenta.

Nos pseudo-códigos mostrados anteriormente, a chamada para o método **inicializaPopulação** ocorre também quando a população converge pois seu efeito é equivalente a um *restart*. Como optamos por adotar uma política elitista, onde o melhor indivíduo da população é sempre preservado, as chamadas do método **inicializaPopulação** com o objetivo de *restart* carregam um parâmetro que poupa esse melhor indivíduo. Dessa forma garantimos que a melhor solução obtida para a população em questão nunca se perca ao longo do processo evolutivo.

2.4 Estrutura populacional

Uma das características do algoritmo memético utilizado é que para todos os problemas adotamos uma população hierarquicamente estruturada, onde os indivíduos são classificados de acordo com sua qualidade. O uso de populações hierarquicamente estruturadas em algoritmos evolutivos é ainda raro. No entanto, testes comparativos entre populações estruturadas e não-estruturadas [23] mostram que essa abordagem pode gerar um grande salto de desempenho em algoritmos tanto genéticos quanto meméticos. Outro ponto interessante é que o número de indivíduos utilizados pelo método pode ser reduzido consideravelmente sem que haja perda de desempenho causada pelo efeito da con-

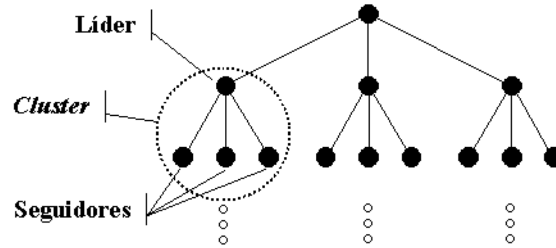


Figura 2.1: *Estrutura populacional adotada no NP-Opt.*

vergência prematura. Isto leva a uma redução do esforço computacional e a um aumento no número de gerações executadas em um mesmo intervalo de tempo, com uma melhora notável no desempenho dos algoritmos [53].

A estrutura escolhida em todos os problemas foi de árvore ternária. Essa escolha tem como motivação o fato da relação de hierarquia ser mais intuitiva. Estruturas do tipo anel e malhas do tipo *grid* não remetem a uma hierarquia própria, mas somente a uma relação de comunicação - ou topologia - entre indivíduos próximos. Estruturas em forma de árvore podem ser encaradas como organizações sociais divididas em castas, onde os indivíduos situados na parte mais alta da árvore estão em vantagem em relação aos indivíduos de partes mais inferiores. Na Figura 2.1 mostramos um diagrama representando a estrutura populacional adotada em todos os problemas.

A Figura 2.1 apresenta algumas características interessantes. A primeira é que a estrutura de árvore ternária é naturalmente dividida em grupos - que chamamos de *clusters* - de quatro indivíduos. Cada cluster é composto por um indivíduo líder e três seguidores, e a hierarquia presente estabelece que a solução líder deve ser a melhor das quatro. Quando extrapolamos esse raciocínio para a população como um todo, fica claro que a melhor solução deverá estar localizada no indivíduo líder do *cluster* mais alto. As piores soluções, por sua vez, tendem a se posicionar nas folhas da árvore ternária.

A segunda característica é a dinâmica evolutiva dentro da árvore. A recombinação dos indivíduos na estrutura de árvore é sempre feita entre dois indivíduos pertencentes ao mesmo *cluster* - o líder e um dos três seguidores - e o filho toma o lugar de um dos pais - isso será detalhado nas seções 2.5 e 2.9, respectivamente. Isso faz com que os *clusters* se comportem como subpopulações, com dinâmicas independentes de recombinação, seleção e inserção. A comunicação entre *clusters* está restrita à fase de reestruturação da população, quando pode haver um tipo de 'migração' de indivíduos entre os mesmos. Essa característica faz com que uma população de apenas 13 indivíduos, por exemplo, tenha uma dinâmica semelhante a quatro populações de quatro indivíduos cada. Esta pode ser uma das razões pelas quais a abordagem estruturada é tão superior à não-estruturada. A estrutura hierárquica, em conjunto com a seletividade na escolha dos pais na hora de efetuar uma recombinação, permite dar um caráter multipopulacional a um método que na realidade é unipopulacional.

A escolha da estrutura de árvore ternária em detrimento da binária, quater-

nária, entre outras, foi regida por considerações empíricas. Uma árvore binária teria um 'efeito subpopulacional' muito pequeno, uma vez que cada *cluster* seria formado por apenas três indivíduos e haveria apenas duas possibilidades de recombinação em cada um deles. Árvores quaternárias ou de grau mais elevado aumentam o caráter subpopulacional, porém testes iniciais indicaram que a qualidade das soluções obtidas não é muito melhor e além disso, a quantidade de indivíduos cresce demasiadamente com o aumento do número de níveis da árvore, o que aumenta o esforço computacional. Desta forma, o melhor *trade-off* nos remete à estrutura ternária.

Do ponto de vista computacional, a implementação de uma população organizada segundo um árvore ternária é bastante simples. A estrutura de dados continua sendo uma matriz, porém a posição de cada indivíduo nela é relevante e deve ser coerente com o que estabelece a hierarquia adotada. Suponha uma população representada por uma árvore ternária de n níveis. A matriz de dados associada deve conter $\frac{3^n-1}{2}$ linhas, cada linha contendo a informação de um indivíduo. Levando-se em conta que tal estrutura conterá $\frac{3^{n-1}-1}{2}$ *clusters*, o líder do k-ésimo *cluster* estará localizado na linha k. Por sua vez, os seguidores pertencentes a este k-ésimo *cluster* estarão localizados nas linhas $[3.k - 1]$, $[3.k]$ e $[3.k + 1]$ da matriz. Esses índices devem ser utilizados tanto na recombinação quanto na reestruturação da população visando manter a consistência hierárquica.

A reestruturação da árvore ternária é um procedimento simples. Um laço percorre todos os *clusters* e em cada um deles o líder é comparado com seus seguidores. Sempre que se verificar que um seguidor é melhor que o líder, eles trocam de posição. Esse procedimento é repetido seguidamente até que após varrer todos os *clusters*, nenhum indivíduo tenha trocado de posição. Isso significa que a consistência hierárquica está mantida.

2.5 Seleção para recombinação

A seleção para recombinação é uma das fases mais críticas na dinâmica do algoritmo. Uma seleção mal feita tem grande chance de dar origem a indivíduos ruins, independente da estratégica de recombinação utilizada. Entre as seleções mais usuais estão a uniformemente aleatória, a aleatória tendenciosa e operadores de seleção restritivos, utilizados em geral em populações estruturadas ou separadas em castas. Nos algoritmos genético e memético utilizamos este terceiro tipo de seleção.

A seleção uniformemente aleatória foi utilizada originalmente por Holland [37]. Ela é facilmente implementável, porém não possui muita correlação com o que ocorre na natureza. Em populações reais, os indivíduos mais bem adaptados em geral têm chances maiores de procriarem. A razão é simplesmente que esses indivíduos tendem a viver mais e por isso acabam gerando mais descendentes. Seleções tendenciosas - por exemplo, *roulette tournament* - conseguem repetir esse efeito com um bom grau de confiabilidade, sendo utilizadas na grande maioria dos trabalhos de algoritmos genéticos e meméticos. Já a seleção restritiva simula situações especiais ou pouco comuns. Em algumas

espécies, os indivíduos são separados, por exemplo, em castas, onde não é permitido o cruzamento entre indivíduos pertencentes a castas distintas. Ao utilizarmos uma população hierarquicamente estruturada e dividida em *clusters*, na realidade estamos simulando este tipo de comportamento. A política de permitir cruzamentos somente entre indivíduos pertencentes ao mesmo *cluster* restringe de forma contundente as possibilidades de escolha dos pares de pais. No entanto, o ambiente utilizado é relativamente flexível pois permite que um indivíduo passe de uma casta para outra na etapa de reestruturação da população. Isso adiciona um certo grau de 'mobilidade social' ao modelo.

Na população estruturada, a recombinação só pode ocorrer entre um líder e um de seus seguidores, dentro do mesmo *cluster*. O método seleciona um líder de maneira uniformemente aleatória e em seguida um de seus três seguidores, também de maneira uniformemente aleatória. Inicia-se então a recombinação em si, que obviamente varia de problema para problema. Há no entanto uma característica que é igual em todos os métodos de recombinação utilizados. Eles sempre geram um único filho. Essa característica nos parece intuitivamente ser mais apropriada pois permite que, dado um tempo limitado, mais pares distintos de pais sejam selecionados para recombinação, o que deve aumentar o poder de exploração da etapa de recombinação. Em outras palavras, acreditamos que seja preferível selecionar 10 casais distintos e fazê-los gerar um filho cada, a selecionar, por exemplo, apenas cinco casais fazendo-os gerar dois filhos cada.

Após um indivíduo ser criado ele passa pelo processo de inserção na população. Nessa etapa, o algoritmo decide se o indivíduo será aproveitado ou descartado. A política adotada nos problemas abordados é bastante restritiva e será explicada em detalhes na seção 2.9.

2.6 Representação e recombinação

O sucesso de um algoritmo evolutivo depende diretamente da representação genética utilizada para descrever as soluções na forma de um cromossomo. O mais recomendado é utilizar representações que sejam compactas, completas e estáveis. Uma representação compacta deve se valer do menor número possível de variáveis para representar de forma unívoca uma solução. Representações completas devem ser capazes de representar todas as possíveis soluções do problema, inclusive a ótima, é claro. Uma representação estável tem como característica que pequenas mudanças no cromossomo levem a alterações também pequenas da adaptabilidade. Caso essas mudanças resultem em grandes alterações de adaptabilidade, poderá haver problemas na evolução da população. Pode-se chegar a um quadro onde mesmo após muitas gerações, a população como um todo seja muito pouco adaptada pois informações importantes aprendidas durante o processo evolutivo estão sendo continuamente perdidas.

Cada problema abordado tem uma representação própria para os indivíduos da população. Foram utilizadas representações na forma de valores inteiros, reais e binários, além de mesclas destes tipos, com estruturas que variaram desde listas simples de inteiros até listas múltiplas e árvores. As representações serão analisadas uma a uma à medida em que os problemas forem sendo apresentados.

A recombinação é talvez o principal mecanismo da evolução. Nos problemas aqui tratados, ela se baseia na criação de um novo indivíduo através da combinação das informações presentes em dois outros; combinação essa que pode ser total ou parcial. A combinação total indica que toda a informação presente no filho é herdada dos pais, ao passo que na parcial parte da informação genética contida no filho é nova ou aleatória. Nos dois casos, têm-se como meta que o filho herde as características principais de cada um dos pais, pois se eles forem bem adaptados, provavelmente seu filho também o será. Todos os *crossovers* implementados têm um certo grau de aleatoriedade, pois conforme dito antes, operadores puramente determinísticos em geral levam a uma perda de diversidade acelerada. Boa parte deles está baseada no operador Order Crossover(OX) [29]. No entanto, alguns problemas necessitaram de operadores especiais, como o de Ordenamento de Genes, onde foi necessário implementar um *crossover* especial para estrutura de árvore. No problema de Localização de Capacitores, por sua vez, necessitamos de um operador que em parte do cromossomo implementa um *Crossover* Uniforme (UX) [68] e em outra parte um *crossover* especial para valores inteiros. Devido a essas particularidades, assim como a representação, a recombinação utilizada será explicada caso a caso.

2.7 Mutação

A mutação tem um papel muito importante na evolução [55]. Ela consiste em uma mudança aleatória de parte, ou partes, do código genético do indivíduo. Essa natureza puramente aleatória faz a mutação ter um caráter bem mais destrutivo do que construtivo. De fato, a grande maioria das mutações são destrutivas, criando indivíduos piores ou mesmo não-viáveis. Indivíduos assim são eliminados através dos mecanismos de seleção natural e os resultados de mutações ditas 'desastrosas' não são perpetuados. O lado bom é que, quando bem sucedida, a mutação pode dar origem a características excelentes ou mesmo fazer o indivíduo saltar de mínimos locais ao melhorar sua adaptabilidade. De uma forma simples, pode-se dizer que uma população está presa em um mínimo local quando ela está enfrentando dificuldades para obter soluções melhores que a incumbente. Um bom movimento de mutação pode pôr fim a essa situação, criando uma nova solução incumbente que seria virtualmente inacessível via operações de recombinação apenas.

No NP-Opt a mutação tem um papel secundário, pois uma de suas tarefas - a de manter a diversidade da população - é exercida pela estratégia de verificação de convergência. Assim, seu objetivo principal fica reduzido a auxiliar o algoritmo na busca de soluções quando a população está presa em um mínimo local - algo que lembra uma busca local. No entanto, nos algoritmos meméticos, essa função também pode ser melhor exercida através da reinicialização da população atuando em conjunto com operadores de busca local. Assim, pelo menos para os algoritmos meméticos, a mutação tem pouca importância. Já para os algoritmos genéticos puros, que não implementam operadores próprios de busca local, a mutação tem um papel fundamental, especialmente quando a população está com dificuldades para obter novas soluções incumbentes. Ressaltamos por

fim que as mutações são altamente dependentes do problema sendo tratado e por esse motivo a forma como foram implementadas será descrita nos capítulos referentes às aplicações do NP-Opt.

2.8 Busca local

A busca local é a principal diferença entre os algoritmos meméticos implementados e os algoritmos genéticos. Nos algoritmos genéticos tradicionais, nenhum indivíduo passa por qualquer processo de otimização via aplicação de operadores de busca local. No entanto, a sua influência sobre o desempenho do algoritmo é notável, e após todos os testes com os vários problemas abordados, é difícil imaginar um método eficiente que não utilize uma busca baseada em uma definição de vizinhança. Métodos de busca local para otimização combinatória geralmente se baseiam em uma definição de vizinhança que estabelece uma relação entre soluções no espaço de configuração do problema.

Para problemas em que o cromossomo é composto por alelos não-repetidos, por exemplo, buscas locais do tipo *troca* ou *inserção* de alelos em geral têm bons resultados. Para problemas em que o cromossomo é composto por valores binários, uma vizinhança do tipo *bit-swap* pode ser um bom ponto de partida. Há ainda outras possibilidades, menos gerais, que podem levar em conta características intrínsecas do problema sendo tratado. De um modo geral, quanto mais conhecimento do problema se utiliza na elaboração da busca local, melhores serão os resultados obtidos, como foi o caso do problema de Ordenamento de Genes. Isso ocorre especialmente porque em vizinhanças grandes, muitas das possibilidades que são avaliadas são absolutamente desnecessárias. Regras empíricas baseadas nas características esperadas das boas soluções permitem uma redução do tamanho da vizinhança. Outra possibilidade consiste em não testar movimentos que não afetem a qualidade da solução. No problema de *Gate Matrix Layout* utilizamos uma abordagem nesse sentido com sucesso.

A criação de buscas locais é um trabalho bem complicado. Por um lado, tenta-se conseguir o maior poder de busca possível, e por outro é necessário manter a complexidade computacional sob controle. Em muitos casos, foi difícil conciliar esses dois fatores, sendo necessário lançar mão de estratégias de redução de vizinhança, algumas vezes simples, outras vezes bem complicadas. Nesse campo, contribuições importantes foram feitas, em especial nos problemas de *Gate Matrix Layout* e de Localização de Capacitores.

Outro ponto importante abordado é a frequência de aplicação da busca local. É preferível ter uma busca local mais 'leve' e aplicá-la a todos os indivíduos ou ter uma busca local mais 'pesada' computacionalmente e aplicá-la somente a uma parte da população, ou ao melhor indivíduo apenas? Perguntas como essas são importantes, pois as escolhas feitas influenciam consideravelmente o desempenho geral do algoritmo. Após todos os testes, verificamos que cada caso deve ser analisado separadamente. Fatores como complexidade da busca local, comportamento da função de *fitness* e mesmo o tamanho da instância afetam essa escolha.

Neste trabalho tratamos de problemas intrinsecamente diferentes, com representações das mais diversas, o que faz com que as buscas locais sejam

também muito diversas. Os capítulos que tratam dos problemas trarão descrições detalhadas dos métodos de busca local utilizados; pontos fortes e fracos serão analisados, bem como o caminho traçado até se chegar às estratégias finais.

2.9 Inserção de novos indivíduos

A inserção de novos indivíduos na população é outra etapa crítica na dinâmica do algoritmo. Em geral, encontramos duas estratégias na literatura. A primeira é a de aceitar uma porcentagem fixa dos novos indivíduos criados, independente de seu *fitness*. O ponto forte dessa política é que ela mantém a diversidade da população por mais tempo. Por outro lado, dado que a população em geral tem um tamanho fixo, a situação em que um bom indivíduo pré-existente é substituído por um novo indivíduo muito pior é possível de acontecer, chegando às vezes a ser algo comum. Isso pode ser parcialmente resolvido ajustando a porcentagem de indivíduos aceitos para valores baixos e/ou substituindo os indivíduos na sequência do pior para o melhor, o que caracterizaria um política elitista, em que o melhor indivíduo é sempre preservado.

A segunda política também estabelece que se deve aceitar uma porcentagem fixa dos novos indivíduos. Porém, a escolha de quais serão aceitos e quais serão descartados segue uma probabilidade que é proporcional ao *fitness* de cada indivíduo criado. Essa é uma representação mais realista do que ocorre na natureza. Assim que nascem, os filhos já têm que enfrentar a competição por alimento, além dos seus predadores naturais. Desta forma, sua inserção depende em grande parte de sua adaptabilidade ao meio, bem como de uma certa dose de sorte. Uma seleção para inserção efetuada mediante um *roulette tournament* simula bem essa realidade.

Uma terceira política é sugerida neste trabalho e foi utilizada em todas as aplicações do NP-Opt. Ela estabelece que um novo indivíduo só será aceito na população caso ele seja melhor que um dos pais. Além disso, a sua inserção se dará no lugar do pai que possuir o pior *fitness*. Obviamente, essa política de aceitação não tem nenhuma inspiração natural. Ela gera uma enorme quantidade de filhos que são descartados e uma perda acelerada de diversidade. No entanto, do ponto de vista algorítmico, os resultados são excelentes. Parte desse bom desempenho se deve a uma rotina de verificação de convergência que é constantemente aplicada. Alguns testes indicaram que essa política é especialmente benéfica no caso dos algoritmos genéticos, fazendo com que a etapa de recombinação funcione como uma busca local. A analogia é clara. Da mesma forma que na busca local, somente os passos que melhoram a solução são aceitos, na inserção dos novos indivíduos somente filhos que melhoram os pais são aceitos. No caso dos algoritmos meméticos, foi também verificada uma melhora ocasionada pelo uso desta política de aceitação, porém em menor grau.

Para lidar com a perda de diversidade acelerada, adotamos um critério de verificação bastante sensível e ao mesmo tempo simples de ser implementado. É bastante comum a convergência da população ser avaliada pelo grau de similaridade entre seus indivíduos, mas neste trabalho optou-se por uma abordagem alternativa. O critério utilizado estabelece que, se ao longo de uma geração

inteira nenhum indivíduo gerado foi melhor que seus pais, ou seja, todos foram descartados, considera-se que a população convergiu. Isso significa que todos os indivíduos que compõem a população são de boa qualidade e que o algoritmo está com dificuldades para gerar indivíduos ainda melhores. Quando isso ocorre, a população é reinicializada. Esse critério só faz sentido quando se adota a política de inserção de novos indivíduos proposta. O ponto forte é que não são necessários cálculos complexos para verificação de semelhança entre os indivíduos. O ponto fraco, por sua vez, é que a taxa de recombinação cresce bastante, o que aumenta a complexidade computacional de cada geração. Uma taxa de recombinação mais alta é absolutamente necessária pois ela eleva a pressão evolutiva e reduz o risco de se efetuar reinicializações em excesso. Só para ilustrar essa característica, em trabalhos utilizando políticas de aceitação baseadas em uma porcentagem dos novos indivíduos, a taxa de recombinação gira entre 40% e 60% da população. Por outro lado, a política de aceitação dos novos indivíduos baseada na comparação com os pais eleva esse valor para algo entre 150% e 250%.

2.10 Abordagem multipopulacional

A opção pelo uso da abordagem multipopulacional tem como principal objetivo uma melhor exploração do espaço de soluções. Para entender melhor o tema é necessário primeiro definir dois termos normalmente utilizados em processos de busca em geral: *intensificação* e *diversificação*. Intensificação - *exploitation* - é o termo utilizado para uma exploração intensa em uma pequena região do espaço de soluções, geralmente promissora, visando encontrar uma solução de boa qualidade. Em um algoritmo memético, por exemplo, a busca local tem o papel de agente intensificador. A diversificação por sua vez, é uma exploração de caráter mais amplo - *exploration* -, utilizada na busca de regiões promissoras do espaço de soluções, onde processos de intensificação serão executados. De novo, em um algoritmo memético, esse seria o papel exercido pela recombinação.

Algoritmos unipopulacionais tendem a ser bons 'intensificadores' e maus 'diversificadores'. A razão é que por mais reinicializações que sejam efetuadas, o espaço de busca permanece muito restrito por quase todo o tempo de execução. Algoritmos multipopulacionais conseguem balancear melhor intensificação e diversificação, especialmente se adotarem políticas de migração coerentes. Uma das razões do sucesso dos algoritmos multipopulacionais reside no chamado *genetic drift* [73]. Esse conceito estabelece que se duas populações isoladas e idênticas forem postas sob as mesmas condições, seus indivíduos seguirão caminhos evolutivos diferentes e ao fim de um número suficiente de gerações, essas populações serão completamente distintas. Esse fenômeno se deve ao caráter aleatório de grande parte dos processos evolutivos.

O *genetic drift* possui um potencial exploratório latente, que pode ser utilizado quando lançamos mão de várias populações. Suponha inicialmente que tenhamos várias populações isoladas e que elas evoluam separadamente até suas respectivas convergências. Pode-se crer que já nesse estágio seus indivíduos serão consideravelmente distintos. Nesse momento, se efetuamos uma migração de alguns poucos indivíduos, estaremos na realidade mesclando informações das

duas populações, explorando assim o espaço de soluções contido *entre* elas. Isso gera um aumento considerável no caráter diversificador do algoritmo [30, 47].

As versões multipopulacionais dos algoritmos implementados seguem um *modelo de ilhas*, ou *island-model*, para a dinâmica evolutiva das várias populações [29]. Segundo esse modelo, as populações estão evoluindo separadamente em paralelo, com pouca migração de indivíduos entre elas. Em um ambiente com essas características, pode-se tirar maior proveito do *genetic drift*, onde a migração cria uma sinergia entre as populações, levando o algoritmo a soluções inatingíveis de outra forma.

O modelo de ilhas necessita de uma topologia para a comunicação entre as populações. Essa topologia é utilizada para definir as rotas de migração admissíveis, sendo uma das mais utilizadas a do tipo *anel*. Essa estrutura coloca as populações dispostas sobre um anel, restringindo a migração apenas às que estão em posições adjacentes. Essa abordagem aparentemente funciona melhor do que o caso em que todas as populações estão conectadas. Essa segunda escolha reduz o efeito diversificador do algoritmo, possivelmente piorando o desempenho global.

Depois de definida a topologia, é necessário determinar a taxa de migração. No nosso caso, a etapa de migração somente ocorre depois que todas as populações convergiram. Apenas isso já faz com que a taxa de migração seja relativamente baixa quando se comparam o número de indivíduos criados com o número de indivíduos que migram. Porém, no que se refere à taxa de migração em si, optamos por estudar três níveis de intensidade.

- *0-Migrate*: Nenhuma migração é feita e todas as populações evoluem em paralelo, sem nenhum tipo de comunicação entre elas.

- *1-Migrate*: A migração ocorre em todas as populações e uma cópia do melhor indivíduo de cada uma delas migra para a população adjacente, tomando o lugar de um indivíduo escolhido aleatoriamente - com exceção do melhor. Assim, cada população recebe apenas um novo indivíduo em cada etapa de migração.

- *2-Migrate*: A migração também ocorre em todas as populações, mas duas cópias do melhor indivíduo migram para as duas populações adjacentes, tomando também o lugar de indivíduos quaisquer, exceto os melhores. Desta forma, cada população recebe dois novos indivíduos em cada etapa de migração.

Os testes indicaram que a política *1-Migrate* é a que leva aos melhores resultados, mas isso será discutido caso a caso nos problemas em que aplicamos os algoritmos multipopulacionais. A Figura 2.2 mostra dois diagramas representando as políticas de migração. Em cada um deles, quatro populações estão posicionadas formando um anel. No caso da *1-Migrate* apenas um único indivíduo externo é recebido em cada população, representado pelas setas unidirecionais. Na política *2-Migrate*, esse número sobe para dois indivíduos, representados pelas setas duplas. Na realidade, as três políticas são uma comparação entre a falta de migração, uma migração fraca e uma forte, dadas as diferenças de intensidade de comunicação entre as populações. As flechas, além

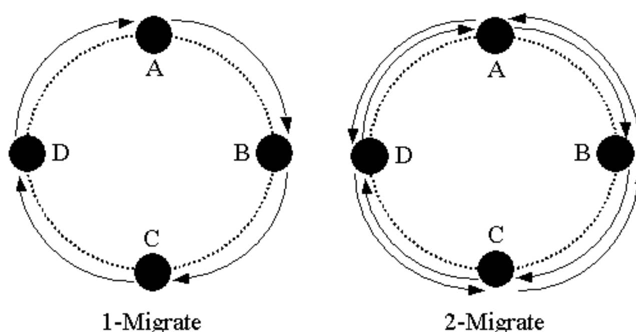


Figura 2.2: *Diagramas das políticas de migração.*

de indicarem o número de indivíduos migrando, indicam também as rotas de migração, que nesse caso só ocorre entre populações adjacentes.

A opção pela migração de uma cópia do melhor indivíduo foi tomada após testes iniciais terem indicado que migrar um indivíduo qualquer era de pouca valia. Na verdade, o melhor indivíduo da população possui a melhor carga genética disponível e por isso deve causar um maior impacto em futuras recombinações com indivíduos de uma outra população.

Depois que a etapa de migração ocorre, todas as populações passam por um processo de reinicialização, onde todos os indivíduos são inicializados, com exceção dos melhores e dos indivíduos recebidos via migração.

2.11 Resumo

Neste capítulo abordamos as características principais dos algoritmos meméticos implementados. Entre elas podemos citar a estrutura hierárquica da população, que segue um estilo em árvore ternária, com a população dividida em *clusters* de quatro indivíduos cada um. Os motivos do uso desse tipo de estrutura em detrimento de outros foram discutidos e, dados os resultados obtidos, acreditamos que esse foi um dos fatores cruciais para o bom desempenho geral do método. Outro aspecto abordado é a seleção de indivíduos para recombinação. Essa seleção segue uma dinâmica própria, estreitamente relacionada com a estrutura populacional e com a existência dos *clusters*. A inserção de novos indivíduos, por sua vez, é altamente restritiva e possui um alto grau de rejeição. A grande quantidade de novos indivíduos descartados torna necessário adotar uma taxa de recombinação alta para obtermos uma dinâmica populacional equilibrada. Na etapa de inserção dos novos indivíduos também há uma relação com a estrutura hierárquica dividida em *clusters*. A inicialização da população é tratada de forma breve, uma vez que em boa parte dos problemas ela é puramente aleatória. De fato, em apenas um dos problemas - o de Ordenamento de Genes - foi necessário utilizar uma heurística construtiva para a criação das populações iniciais. Para todos os outros, o algoritmo memético foi capaz de obter soluções de boa qualidade a partir das soluções aleatórias

iniciais.

Nesse capítulo também foram discutidos a representação e os operadores de busca local, recombinação e mutação. Como esses aspectos são altamente dependentes do problema tratado, eles serão continuamente abordados ao longo deste trabalho, sempre que for necessário ressaltar algum aspecto particular.

Apresentamos também o algoritmo memético multipopulacional, bem como o modelo de ilhas utilizado na migração de indivíduos. A motivação do uso de múltiplas populações se concentra no fenômeno do *genetic drift* que é brevemente comentado ao longo deste capítulo.

Por fim, apresentamos quatro pseudo-códigos. Dois referentes a algoritmos genéticos e dois a algoritmos meméticos. Eles servem para descrever as semelhanças e diferenças entre as duas abordagens de uma forma mais simples e direta. Nos pseudo-códigos também estão explícitas as alterações no processamento quando passamos da abordagem unipopulacional para a multipopulacional.

Esta página foi deixada intencionalmente em branco.

Capítulo 3

O NP-Opt

3.1 Introdução

[1] O objetivo principal deste trabalho foi o desenvolvimento de um ambiente de otimização onde o usuário pudesse inserir novos métodos e problemas de uma forma fácil e rápida. A filosofia principal era fazer uso de um código simples, de fácil entendimento e que pudesse ser reutilizado em grande parte. Para tanto, todo o software foi criado segundo o paradigma da orientação a objetos, com a criação de classes que pudessem ser especializadas para atender às necessidades dos problemas. A estrutura do NP-Opt também deveria ser modular, de forma a facilitar a inclusão de novos recursos, e esses recursos deveriam ser incluídos de forma a ficarem disponíveis para todos os problemas, sempre que possível. Outro ponto foi o uso de uma interface gráfica para facilitar a comunicação usuário/máquina, além de ferramentas que pudessem facilitar seu uso prático. Por fim, o NP-Opt deveria ser capaz de resolver os problemas de forma eficiente. Isso ficou sob a responsabilidade de um algoritmo memético que utiliza recursos relativamente complexos, como população hierarquicamente estruturada, migração e processamento paralelo. O modelo de algoritmo memético utilizado em todos os problemas é basicamente o mesmo, e isso tem como objetivo validar o NP-Opt como uma plataforma de otimização de uso geral. De fato, se fossem necessárias mudanças radicais no código ou na estrutura do algoritmo memético para se atingir um desempenho superior, não haveria como justificar a existência do software. O NP-Opt conta atualmente com cinco problemas disponíveis, além do conjunto de todas instâncias utilizadas nos testes presentes nesses trabalho. São eles:

[1]Este capítulo é baseado nos artigos:

A. Mendes, P. França e P. Moscato. **NPOpt: An Optimization Framework for NP Problems**, Proceedings do *POM2001 - International Conference of the Production and Operations Management Society*, pág. 82-89, Guarujá, Brasil, Agosto, 2001.

V. Garcia, A. Mendes, P. França e P. Moscato. **Algoritmo Memético Paralelo Aplicado a Problemas de Sequenciamento em Máquina Simples**, Proceedings do *XXXIII SOBRAPO - Simpósio Brasileiro de Pesquisa Operacional*, pág. 971-981, Campos do Jordão, Brasil, Novembro, 2001.

- Sequenciamento em Máquina Simples
- Sequenciamento em Máquinas Paralelas
- Sequenciamento em *Flowshop*
- *Gate Matrix Layout*
- Ordenamento de Genes

O NP-Opt foi inteiramente desenvolvido utilizando a linguagem *Java* [41] e com componentes *Java Swing* para a parte gráfica. Isso permite uma maior portabilidade, além de um aspecto visual mais agradável. O NP-Opt pode ser executado em qualquer sistema operacional que suporte um JVM (*Java Virtual Machine*), além de recursos gráficos. O programa utiliza poucos recursos computacionais. Como seu tamanho é de pouco mais de 300 KB, a configuração recomendada depende basicamente da complexidade do problema e do tamanho das instâncias a serem testadas. A parte gráfica é bastante leve, sendo composta por janelas de diálogo simples e sem recursos de animação, como pode ser visto na Figura 3.1.

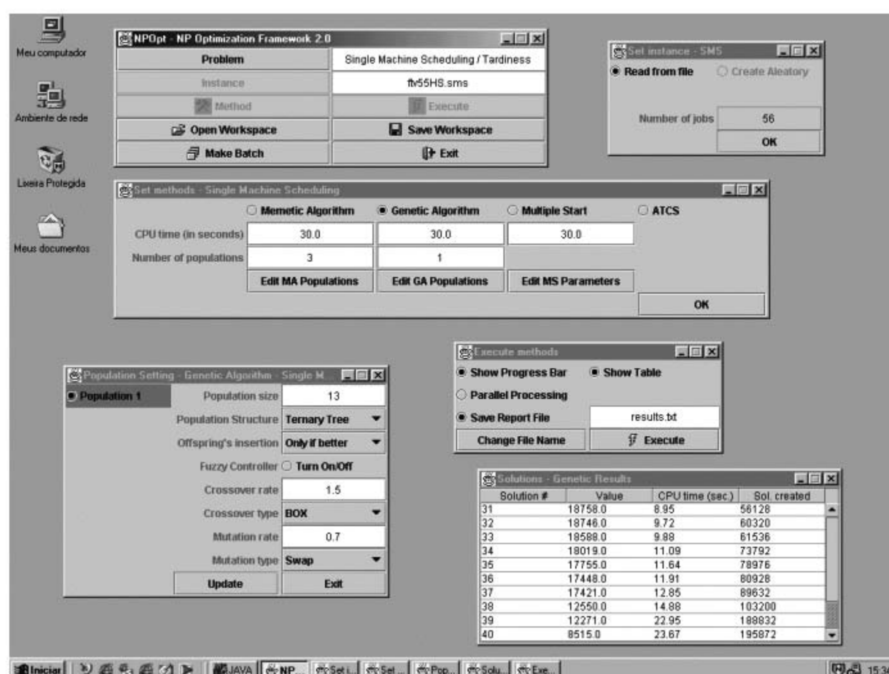


Figura 3.1: *Tela do NP-Opt rodando sob o sistema operacional Windows.*

O NP-Opt já foi testado em vários sistemas operacionais - *Sun Solaris*, *Windows 98/NT/2000* e *Linux* - sendo que em todos eles o desempenho foi normal, sem que fosse notado nenhum tipo de incompatibilidade ou falha. Recomenda-se utilizar uma versão recente do Java e compilar o código sempre na máquina

que se vai efetuar os testes usando código nativo. A versão mais recente do *Java* faz isso automaticamente, compilando o código fonte de forma otimizada e fazendo o desempenho melhorar consideravelmente. Ressaltamos, por fim, que o uso de processamento paralelo só é possível em ambientes *Unix/Linux* pois o *Windows* não permite o disparo remoto de processos devido a restrições de segurança.

3.2 Rodando um problema

O NP-Opt foi desenvolvido de forma que o usuário pudesse se comunicar com o programa utilizando apenas janelas gráficas. A seguir, vamos explicar a função de cada uma delas, começando pela janela inicial mostrada na Figura 3.2.

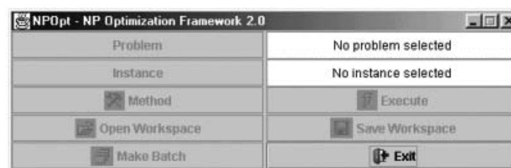


Figura 3.2: *Janela inicial do NP-Opt.*

A janela inicial é composta por vários botões de seleção, além de dois campos de texto. No lado esquerdo temos os botões *Problem*, *Instance*, e *Method*. Ao lado desse último temos o botão *Execution*, que executa os métodos selecionados. Abaixo deles temos os botões de abertura e gravação de *workspaces* (ou seja, a configuração atual do *framework*). Na parte superior direita há dois campos de texto que mostram o tipo de problema e a instância sendo resolvida. O botão denominado *Make Batch*, localizado na parte inferior da janela, é utilizado para efetuar testes sequenciais automáticos e, por fim, temos o botão *Exit*, que sai do programa. Inicialmente, os únicos botões habilitados são o *Problem* e o *Exit*. Clicando sobre o botão *Problem* abre-se a janela de seleção do tipo de problema, representada na Figura 3.3.

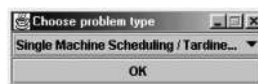


Figura 3.3: *Janela de seleção do tipo de problema.*

A janela de seleção do tipo de problema é composta por um menu, onde o usuário pode escolher um problema percentence a uma lista. Suponha que se selecione a opção *Single Machine Scheduling / Tardiness*, clicando em seguida no botão *OK*. A janela vai desaparecer e o tipo do problema será escrito no campo de texto ao lado do botão *Problem*. Depois disso, vários botões serão habilitados. No entanto, vamos seguir com a ordem lógica e selecionar a instância. Para tanto, basta clicar no botão *Instance* e esperar que se abra a janela de seleção de instância, mostrada na Figura 3.4.

Figura 3.4: *Janela de seleção da instância.*

A janela de seleção da instância permite ao usuário ler uma instância a partir de um arquivo gravado em disco. Há espaço para a possibilidade de se criar uma instância aleatória, mas essa opção está desabilitada no momento devido à falta de necessidade do uso desse recurso. Após selecionar a opção *Read from file* e clicar sobre o botão *OK*, abre-se uma janela que permite ao usuário selecionar um arquivo de disco (ver Figura 3.5). A janela de seleção de instância é específica do problema. Nela, o usuário pode incluir os campos que considerar necessários para caracterizar a instância. No caso do problema de Sequenciamento em Máquina Simples (SMS), o único campo presente é o de número de tarefas.

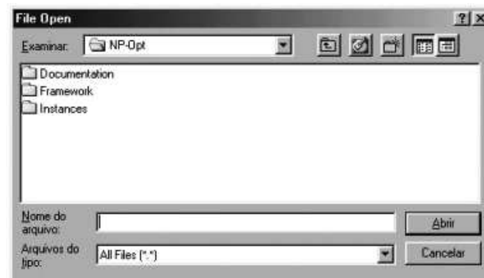


Figura 3.5: *Janela geral de seleção de arquivo.*

Na janela de seleção de arquivo, selecione um arquivo válido e clique no botão *Abrir*. As duas janelas abertas serão fechadas, o nome do arquivo será escrito ao lado do botão *Instance* e o botão *Method* se habilitará. Caso se queira verificar as propriedades da instância, em qualquer momento pode-se clicar de novo no botão *Instance* e verificar os dados, fechando a janela em seguida. O próximo passo é ajustar os métodos para se resolver o problema selecionado. Após clicar sobre o botão *Method*, abre-se a janela de seleção dos métodos (ver Figura 3.6).

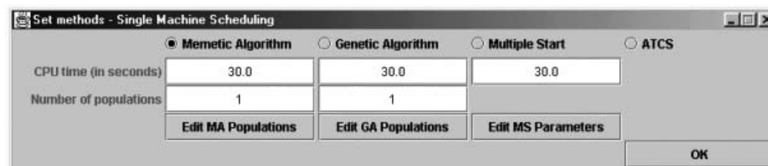


Figura 3.6: *Janela de seleção dos métodos.*

Existem 4 métodos disponíveis para o problema de SMS. Eles podem ser selecionados clicando-se nos botões na parte superior da janela. Abaixo dos métodos temos o campo de texto que define o tempo de CPU máximo para cada um deles. Este é o único critério de parada disponível no momento. O método ATCS é na realidade uma regra de despacho descrita na referência [45], e que por ser muito rápida dispensa a definição do tempo de CPU máximo. Abaixo dos tempos de CPU temos o número de populações para os algoritmos memético e genético. Nesse campo o usuário faz a escolha por uma abordagem unipopulacional ou multipopulacional. Por fim, na parte de baixo da janela, estão localizados três botões para ajuste dos parâmetros dos métodos. Ao se clicar em um dos botões de ajuste das populações, duas novas janelas vão se abrir (ver Figura 3.7).

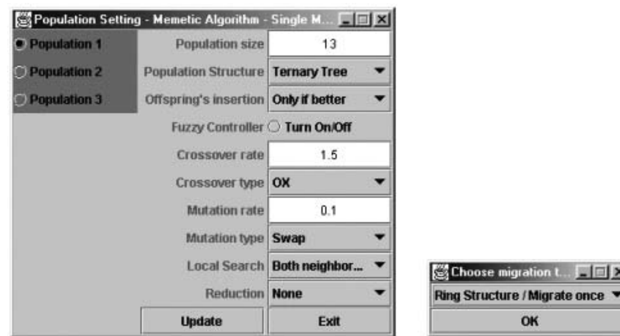


Figura 3.7: *Janelas de ajuste das populações e da política de migração.*

A janela maior permite ao usuário editar as características das populações, uma por vez. Primeiramente, selecione a população clicando em um dos botões do lado esquerdo. Em seguida, faça as modificações nos campos situados no lado direito:

- **Population size**: O número de indivíduos da população.
- **Population Structure**: Seleciona se a população terá uma estrutura hierárquica ou não. O número de indivíduos para estrutura de árvore ternária deve ser 1, 4, 13, 40, 121, etc. Já para árvore binária, os valores admissíveis são 1, 3, 7, 15, 31, 63, etc. Se a população não utilizar nenhum tipo de estrutura, qualquer número é válido.
- **Fuzzy Controller**: Liga/desliga um controlador *fuzzy* para as taxas de mutação e de recombinação. Este controlador está disponível somente para o problema de Sequenciamento em Máquina Simples.
- **Crossover rate**: A taxa de recombinação da população, que está relacionada à quantidade de novos indivíduos criados a cada geração.
- **Crossover type**: O operador de recombinação que a população vai utilizar para criar os novos indivíduos.

- **Mutation rate**: A taxa de mutação da população, que está relacionada à porcentagem dos novos indivíduos que irão sofrer mutação.
- **Mutation type**: O operador de mutação a ser aplicado nos novos indivíduos.
- **Local Search**: O operador de busca local. Disponível somente para o algoritmo memético, uma vez que os algoritmos genéticos puros não utilizam esse recurso.
- **Reduction**: Estratégia de redução de vizinhança, quando disponível.

Depois de todos os parâmetros terem sido definidos, clique no botão *Update* para guardar as alterações e somente então avance para a próxima população. A janela menor, na direita da Figura 3.7, permite ao usuário selecionar a política de migração, definindo como as populações deverão trocar indivíduos entre si. Os parâmetros são os seguintes:

- **Ring structure**: As populações estão conectadas em uma estrutura de anel, podendo trocar indivíduos apenas com as duas populações vizinhas.
- **No structure**: Cada população está conectada com todas as outras, e a troca de indivíduos é livre entre quaisquer duas populações.
- **Migrate once**: Uma cópia do melhor indivíduo é enviada para outra população.
- **Migrate twice**: Duas cópias do melhor indivíduo são migradas para duas populações diferentes.
- **No migration**: Todas as populações evoluem em paralelo, sem nenhuma troca de indivíduos.

Depois de definidos todos os parâmetros, clique nos botões *OK*, na janela de migração, e *Exit* na janela das populações. Caso se queira editar os parâmetros do *multiple start* basta clicar no botão *Edit MS parameters* e uma nova janela será aberta (ver Figura 3.8).

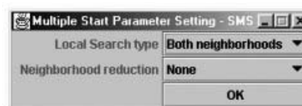


Figura 3.8: *Janela de parâmetros do multiple start.*

Na janela de parâmetros do *multiple start*, pode-se selecionar o tipo de busca local e a redução de vizinhança desejadas. Depois de ajustar as duas opções, basta clicar no botão *OK*. Ajustados todos os parâmetros da janela de métodos, pode-se fechá-la e passar para a etapa de execução. Ao clicar no botão *Execution*, a janela de execução se abrirá (ver Figura 3.9).

Figura 3.9: *Janela de execução.*

A janela de execução possui quatro botões de seleção. O botão do alto à esquerda liga e desliga a opção de uma barra de progresso que monitora o tempo de CPU. Ao seu lado, na direita, pode-se acionar a opção de visualizar em uma janela à parte cada melhor solução encontrada enquanto o algoritmo efetua o processo de busca. Essas duas opções não afetam o comportamento do algoritmo, porém o seu uso cria um gasto computacional extra no controle de recursos gráficos.

O botão *Parallel Processing* permite a distribuição do esforço da busca local através de uma rede de computadores, e o botão *Edit Properties* abre uma nova janela (ver Figura 3.10) para edição dos parâmetros da rede. O usuário também pode criar um arquivo de saída de dados, que será composto por todas as melhores soluções geradas ao longo do processo de busca, com as respectivas estruturas dos cromossomos. Com isso, o usuário tem acesso não apenas ao valor da função objetivo, mas à solução propriamente dita. O nome padrão para o arquivo é *result.txt*, mas ele pode ser mudado clicando-se no botão *Change File Name*. A seguir, na Figura 3.10, mostramos a janela de processamento paralelo.

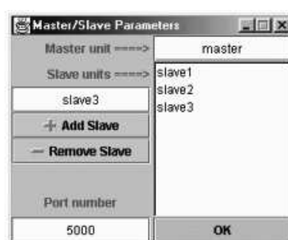


Figura 3.10: *Janela de processamento paralelo.*

A janela de processamento paralelo permite ajustar os parâmetros da rede. Deve-se especificar o nome do computador *master* e dos computadores *slave*. Além disso, o usuário também deve ajustar o número da porta de comunicação. O valor padrão é 5000, mas em algumas redes, portas altas são protegidas por *firewalls*, o que pode criar problemas de segurança. Neste caso, o usuário deve ajustar o valor para uma porta não protegida. A arquitetura do processamento distribuído será discutida mais à frente, ainda neste capítulo. A seguir, na Figura 3.11, mostramos as últimas duas janelas que faltam ser descritas: a tabela das soluções e a barra de progresso.

A janela maior mostra uma tabela com as soluções encontradas até o momento. Cada vez que o algoritmo encontra uma melhor solução, ela é incluída

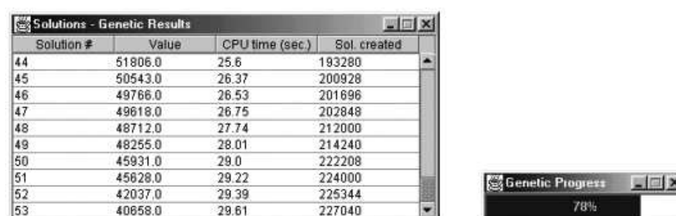


Figura 3.11: *Janelas de saída.*

na tabela, bem como o tempo de CPU e o número de soluções criadas. A janela menor mostra a porcentagem do tempo de CPU gasto, na forma de uma barra de progresso. Assim, o usuário pode avaliar o quanto já foi executado e o que falta para terminar a rodada.

3.3 Abrindo e gravando um *workspace*

Consideramos como *workspace* o conjunto das informações referentes aos vários parâmetros do NP-Opt. Esta é uma ferramenta criada que visa simplificar e dinamizar a execução de testes. Como se pôde ver na seção anterior, a quantidade de parâmetros a serem ajustados é bastante grande e se cada vez que se desejar fazer um novo teste for necessário efetuar todos esses ajustes, o trabalho se torna um tanto cansativo. Com o uso dos *workspaces*, é possível, a qualquer momento, gravar o estado atual de todos os parâmetros do NP-Opt em um arquivo de dados e resgatá-lo no futuro. O botão *Save Workspace* grava todos os parâmetros, como nome da instância, configurações das populações, políticas de migração, dados da rede para processamento paralelo, entre outros. Esses dados podem ser resgatados clicando-se no botão *Load Workspace*. Uma janela de seleção de arquivos (ver Figura 3.5) é utilizada como *interface* com o usuário.

Como os algoritmos disponíveis e os parâmetros são dependentes do problema, é necessário selecionar o tipo de problema antes de abrir um *workspace*. Devemos ressaltar que os *workspaces* são gravados em um arquivo de texto, que pode ser editado facilmente.

3.4 Criando *batches*

Batches podem ser descritos como conjuntos de testes, a serem executados em sequência. São especialmente necessários quando se deseja efetuar testes exaustivos, com dezenas de configurações de parâmetros, ou envolvendo uma grande quantidade de instâncias. De novo, esta opção foi criada também visando facilitar o uso em situações reais. Na Figura 3.12 pode-se ver a janela de ajuste dos *batches*, que é aberta quando se clica no botão *Make Batch*.

Os componentes da janela de ajuste dos *batches* estão dispostos em colunas. A primeira coluna é o índice do *batch*. Na segunda coluna o usuário deve especificar quantas vezes deseja que o *workspace* seja executado. Por fim, na

Figura 3.12: *Janela de ajuste dos batches.*

terceira coluna é mostrado o nome do *workspace* que será executado. Para passar de um *batch* para outro, basta clicar nos botões do lado esquerdo da janela.

Como a execução dos *batches* se resume a rodadas sequenciais de diferentes *workspaces*, o usuário deve antes criar todos os *workspaces* necessários, salvando-os em arquivos distintos. Esses arquivos são então selecionados clicando-se nos botões *Select Workspace*. Depois de todos os parâmetros terem sido ajustados, basta clicar no botão *Run*. Uma barra de progresso irá aparecer e o processamento terá início. Os resultados de cada rodada são gravados em arquivos de texto. Seus nomes seguem o fomato *batch_i_k.txt*, onde i é o índice do *batch*, e j indica qual é a rodada do *workspace*. Todos os arquivos são gravados no diretório raiz do NP-Opt.

3.5 Processamento distribuído

A maior eficiência dos algoritmos meméticos em relação aos algoritmos genéticos puros se deve a bons procedimentos de busca local, geralmente específicos para o problema que se quer resolver. Neste trabalho são relatadas várias aplicações em que a simples inclusão da busca local cria um salto de desempenho considerável. No entanto, o desafio neste caso é que para instâncias grandes, a busca local gera um esforço computacional muitas vezes proibitivo. Uma das formas de contornar esse problema é o emprego de técnicas para redução de vizinhança. Há, contudo, outros casos em que, mesmo com uma boa redução, a exploração da vizinhança resultante continua sendo bastante custosa. Neste caso, torna-se necessário o desenvolvimento de algoritmos que explorem técnicas paralelas de execução.

3.5.1 Tipos de arquitetura

Várias formas clássicas de algoritmos evolutivos paralelos são encontradas na literatura, como demonstram os trabalhos de Cantú-Paz [9, 11]. Em geral, podemos classificar os algoritmos evolutivos paralelos em três classes principais:

- Algoritmos Evolutivos Paralelos Globais (AEPG)
- Algoritmos Evolutivos Paralelos Multi-Populacionais (AEPMP)
- Algoritmos Evolutivos Paralelos Híbridos (AEPH)

Os AEPG são assim chamados porque os operadores de seleção, reprodução e mutação são aplicados em toda a população. A implementação geralmente é realizada utilizando a arquitetura *mestre-escravo*. Uma unidade mestre atribui certas funções do algoritmo a outras unidades escravas, que realizam a tarefa e retornam o resultado. Nos algoritmos genéticos, a tarefa comumente distribuída para os escravos é a avaliação dos indivíduos, por ser independente para cada um. Dessa forma, uma fração da população é atribuída a cada escravo e a comunicação ocorre quando a mesma é enviada ou recebida. Quando a unidade mestre espera pelas respostas de todas as unidades escravas para então dar prosseguimento ao algoritmo, o método é classificado como síncrono, e preserva todas as características do comportamento evolutivo do algoritmo sequencial, porém com melhor desempenho. Outra possibilidade é fazer com que a unidade mestre não mais espere por todas as respostas, o que caracteriza um método assíncrono. Neste caso ocorre uma diferenciação em relação às propriedades do algoritmo sequencial. Indivíduos de uma geração passam para as próximas como se houvessem migrado, o que modifica o comportamento evolutivo do algoritmo. A Figura 3.13 ilustra o diagrama básico de um AEPG.

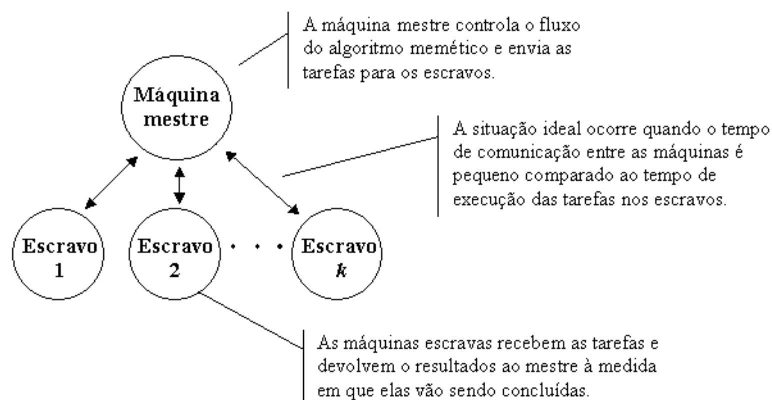


Figura 3.13: *Arquitetura mestre-escravo.*

Nos AEPG há um custo associado à comunicação, resultando num compromisso entre número de unidades escravas e eficiência do método. Cantú-Paz [9] analisa detalhadamente este compromisso, fornecendo resultados que indicam a existência de um número ótimo de unidades escravas que minimiza o tempo de execução do algoritmo.

A classe dos AEPMP se caracteriza principalmente pela divisão da população principal em várias populações, sendo cada uma atribuída a uma unidade

do sistema multiprocessado. Cada uma dessas unidades, por sua vez, executa um algoritmo evolutivo sequencial, restrito à sua população. Em certos períodos ocorre uma migração, caracterizando assim um modelo de ilhas como o descrito no Capítulo 1.

Já nos AEPH, a maior dificuldade está relacionada com a complexidade resultante da associação de dois ou mais modelos de algoritmos evolutivos paralelos. Os modelos mais comuns usam uma hierarquia, combinando os modelos anteriores, sendo o nível mais elevado composto pelo AEPMP distribuído. Em um nível mais baixo pode haver um AEPMP massivamente paralelo, um AEPG mestre-escravo ou até mesmo o próprio AEPMP distribuído.

Estudos extensos envolvendo diferentes arquiteturas de sistemas distribuídos e suas aplicações a algoritmos evolutivos e problemas NP podem ser encontrados nas referências [5, 10, 31]. No momento, o NP-Opt utiliza a arquitetura AEPG, onde a tarefa distribuída é a busca local. No entanto, tendo em vista o uso de múltiplas populações, o próximo passo lógico é a implementação de um AEPMP.

3.5.2 Detalhes da implementação

Como já foi dito antes, a necessidade do uso de técnicas paralelas de execução de algoritmos depende basicamente de três fatores: o tamanho da vizinhança, o tamanho da instância e a quantidade de indivíduos que passarão pela busca local. Em todos os problemas tratados nesta tese, o tempo gasto na busca local é significativamente maior que o gasto nas outras etapas do algoritmo memético. Assim, a busca local é a candidata natural para ser paralelizada.

Outro fator importante é que as buscas locais são independentes. A única comunicação que ocorre é quando a máquina escrava recebe o indivíduo a ser otimizado e quando ela devolve o indivíduo final para a máquina mestre. Nesse meio tempo, a comunicação é nula.

Pode-se imaginar o algoritmo memético como um conjunto de duas camadas, sendo a superior incumbida de processar todas as tarefas, com exceção da otimização dos indivíduos, e a inferior somente desta última. Somente quatro passos são necessários para que essa funcionalidade seja posta em prática: a primeira seria a inicialização da camada mestre-escravo; a segunda a chamada propriamente dita para a execução da otimização de um determinado indivíduo gerado; a terceira faz com que o algoritmo não prossiga enquanto a otimização de todos os indivíduos não estiver terminada e a última é a finalização. A Figura 3.14 mostra a implementação da arquitetura mestre-escravo no NP-Opt.

Dado que o tempo necessário para a busca local é bem maior que o gasto no restante do algoritmo, pode-se imaginar que à medida em que os indivíduos vão sendo criados, uma fila de espera é formada, aguardando que alguma máquina escrava seja liberada. Assim, os indivíduos a serem otimizados vão sendo posicionados em uma *fila de requisições*. Cada escravo, que é um computador da rede, portanto uma unidade da máquina multiprocessada virtual, irá processar os pedidos desta fila e depositar a resposta - o indivíduo otimizado - em outra, chamada de *fila de respostas*. Os indivíduos são então retirados dessa fila de respostas e inseridos na população de novo. A fila de resposta em geral

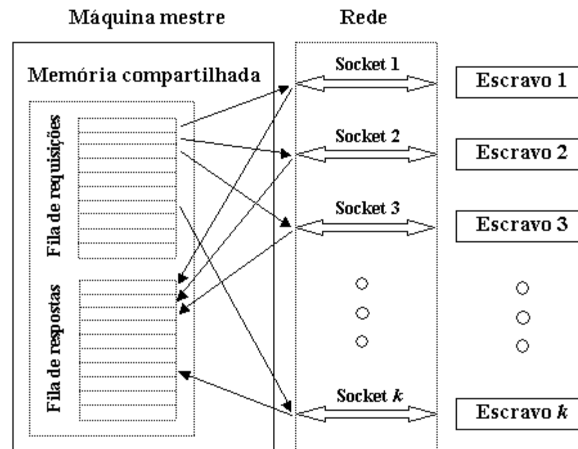


Figura 3.14: *Implementação da arquitetura mestre-escravo.*

tem sempre tamanho reduzido, pois a reinserção dos indivíduos na população é uma operação bastante rápida. Uma característica positiva desta abordagem é que a não alocação pré-determinada de tarefas a nenhum escravo específico cria um melhor balanceamento de carga de acordo com a demanda, ou seja, os escravos mais rápidos processarão mais tarefas. O ponto negativo é que o processamento síncrono pode causar uma parada total do algoritmo enquanto se aguarda a última busca local ser concluída, por exemplo. Se as buscas locais têm sempre a mesma complexidade, indiferentemente do indivíduo, esse problema é inexistente, mas quando isso não é verdade, podem ocorrer problemas na dinâmica do processamento. O processamento síncrono é também mais sensível a falhas de comunicação da rede, que quando demoram a ser detectadas atrasam o algoritmo como um todo.

3.6 Estrutura de classes

A estrutura de classes utilizada no NP-Opt é apresentada na Figura 3.15. Ela se divide em quatro grandes grupos. O primeiro, mais ao alto, é composto pela classe principal - *Framework* - e pelas classes de entrada de dados - todas com final '*Window*'. Não estão incluídas nesse grupo as duas classes de saída de dados - *ProgressFrame* e *OutputTable*.

O grupo maior está localizado na parte esquerda. Ele é composto pelas classes que implementam os métodos de solução. Desse grupo, a classe principal é a *ExecutionMethod*, que controla a chamada e o fluxo dos métodos. Na parte central, à direita, temos as classes responsáveis pelo processamento paralelo. Elas se comunicam com a classe *Memetic* através de chamadas para variáveis da classe principal *Framework*. Por fim, no canto inferior direito, tempos as duas classes de saída de dados. Pode-se notar que algumas classes estão marcadas com a cor cinza. Essas classes são dependentes do problema - de-

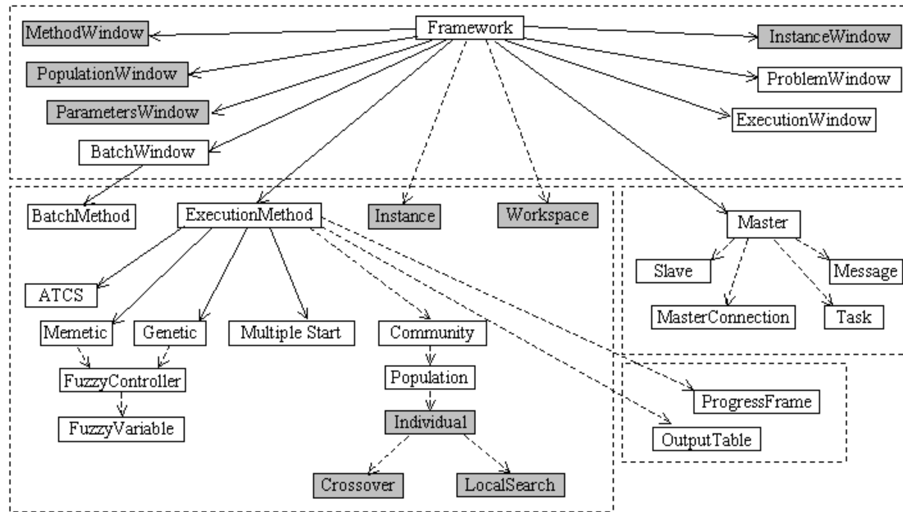


Figura 3.15: *Estrutura de classes do NP-Opt.*

nominadas *abstratas* - e por isso necessitam ser especializadas cada vez que um novo problema é somado ao NP-Opt, como mostra a Figura 3.16. As outras não necessitam de especialização, mas sim de pequenas mudanças no código para fazê-las lidarem com novos problemas. As setas também são de dois tipos: contínuas ou tracejadas. Setas contínuas indicam herança e as tracejadas o uso de uma classe por outra.

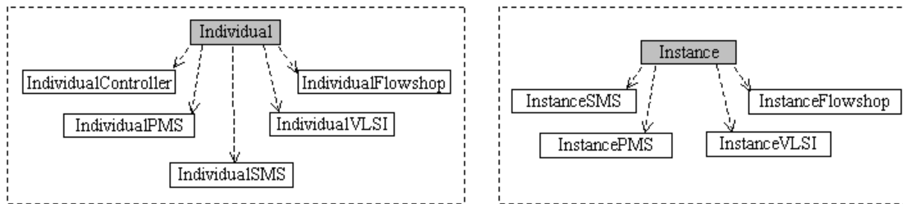


Figura 3.16: *Especialização de classes.*

Na Figura 3.16 temos dois exemplos de especialização, nas classes *Individual* e *Instance*. A classe abstrata contém apenas as variáveis comuns e os cabeçalhos dos métodos, que são implementados nas classes especializadas, de acordo com o problema. Assim, a chamada para os métodos pode sempre ser a mesma, independente do problema sendo resolvido. Essa abordagem permite uma redução considerável no esforço de programação quando se deseja incluir um novo problema no NP-Opt.

A especialização das classes é uma tarefa bem rápida. No caso das várias janelas, como todas possuem a mesma aparência, independentemente do problema, a tarefa de programação segue um estilo *cut-and-paste*. O problema maior reside na especialização das classes não-gráficas, como *Instance*, *Cross-*

over, entre outras. No entanto, caso já se disponha dos códigos ou mesmo das definições de instância, indivíduo ou qual operador de recombinação será utilizado, este trabalho também não é dos mais extenuantes. De fato, um exame rápido revelará muitas semelhanças entre classes especializadas pertencentes a problemas distintos, com boa parte da programação ainda seguindo o estilo *cut-and-paste*. A parte final é a modificação do código de algumas poucas classes para a inclusão do novo problema. Essas modificações envolvem, por exemplo, a inclusão do nome do problema em uma lista para que o usuário possa escolhê-lo na janela de seleção do tipo de problema, entre outras tarefas pouco complexas. A seguir, mostraremos passo a passo como o usuário deve fazer para incluir um problema novo no NP-Opt.

3.7 Incluindo um novo problema

A abordagem via orientação a objetos permite a inclusão de novos problemas de maneira bem rápida. Nesta seção será mostrado como e quais arquivos devem ser modificados para que isso seja feito.

3.7.1 Modificações na classe *Framework*

A classe *Framework* deve passar por algumas mudanças. Inicialmente, atualize o índice dos problemas, no cabeçalho da classe, adicionando uma linha para o novo problema:

```
public static final int FUZZY = -1;
public static final int FUNCTION = 0;
public static final int VLSI = 1;
public static final int PMS = 2;
public static final int FLOWSHOP = 3;
public static final int SMS = 4;
public static final int NEWPROBLEM = 5;
```

Estabelecemos a denominação NEWPROBLEM para o novo problema que se deseja incluir no NP-Opt. Obviamente, a nomenclatura realmente utilizada deve ser mais mnemônica, guardando uma maior relação com o problema, e deve substituir todas as ocorrências da palavra NEWPROBLEM. A linha adiciona uma nova constante que é utilizada sempre que o NP-Opt deseja verificar qual problema está sendo tratado para então efetuar chamadas de classes ou métodos especializados. No método **actionPerformed**, adicione ao bloco:

`if (e.getActionCommand().equals("Instance"))`, a linha:

```
if (problemType == NEWPROBLEM)
                  {instanceTypeWindow = new InstanceWindowNewProblem();}
```

E ao bloco:

`if (e.getActionCommand().equals("Method"))`, a linha:

```
if (problemType == NEWPROBLEM)
                  {methodTypeWindow = new MethodWindowNewProblem();}
```

Caso o usuário deseje implementar os botões *Load Workspace* e *Save Workspace*, adicione ao bloco:

`if (e.getActionCommand().equals("Open Workspace"))`, as linhas:

```
if (problemType == NEWPROBLEM)
                  {workspace = new WorkspaceNewProblem();}
if (problemType == NEWPROBLEM)
                  {instanceToSolve = new InstanceNewProblem();}
```

E ao bloco:

`if (e.getActionCommand().equals("Save Workspace"))`, a linha:

```
if (problemType == NEWPROBLEM)
                  {workspace = new WorkspaceNewProblem();}
```

Estas são as modificações necessárias na classe *Framework*. Como se pode ver, é um procedimento pouco complexo, e essa complexidade se repete em todas as outras classes que precisam ser apenas modificadas.

3.7.2 Modificações na classe *ProblemWindow*

A classe *ProblemWindow* é onde o usuário seleciona o problema que deseja resolver. Assim, ela deve ser modificada para que o novo problema seja incluído. No método **openWindow** modifique a linha:

`String[] items = new String[5];`, para:

```
String[] items = new String[6];
```

E adicione a linha:

```
items[5] = "New Problem";
```

Faça a seguinte modificação na linha:

`for (i = 0; i < 5; i++) choice.addItem(items[i]);`, para:

```
for (i = 0; i < 6; i++) choice.addItem(items[i]);
```

E no método **upDateValues**, adicione a linha:

```
if (problemType == NEWPROBLEM) choice.setSelectedIndex(NEWPROBLEM);
```

Isso finaliza a atualização da classe *ProblemWindow*.

3.7.3 Especialização da classe *InstanceWindow*

O processo de especialização de uma classe sempre requer um trabalho maior que o processo de simples modificação. No entanto, como todas as janelas possuem a mesma aparência no NP-Opt, o usuário verá que esse processo é também relativamente simples. A classe *InstanceWindow* permite que se leia uma instância de um arquivo e também que se visualize as características principais dela. Inicialmente, deve-se abrir uma classe especializada *InstanceWindow*, como por exemplo, **InstanceWindowPMS.java** e gravá-la como **InstanceWindowNewProblem.java**. As modificações são as seguintes:

`public class InstanceWindowPMS extends InstanceWindow`, para:
`public class InstanceWindowNewProblem extends InstanceWindow`

`public static InstanceWindowPMS dialog;`, para:
`public static InstanceWindowNewProblem dialog;`

`public InstanceWindowPMS()`, para:
`public InstanceWindowNewProblem()`

`dialog = new InstanceWindowPMS();`, para:
`dialog = new InstanceWindowNewProblem();`

No método **openWindow**, o usuário implementa os parâmetros da instância que irão aparecer na janela de seleção da instância. Para o problema de Sequenciamento em Máquinas Paralelas (SMP), ficou decidido que os parâmetros *número de tarefas* e *número de máquinas* eram relevantes para caracterizar uma dada instância e por isso deveriam ser mostrados. Quando um novo problema é adicionado, o usuário deve escolher as informações relevantes e criar seus campos correspondentes, colocando-os na janela. A sugestão é que se utilize o SMP como exemplo, implementando as modificações de forma mais fácil e mantendo o aspecto visual original do NP-Opt. De volta às modificações, no método **actionPerformed**, é necessário alterar a linha:

`instanceToSolve = new InstancePMS();`, para:
`instanceToSolve = new InstanceNewProblem();`

Ainda neste método, o usuário deve prestar atenção ao bloco `if (e.getActionCommand().equals("Radio Button"))`, ajustando-o para os novos campos criados. O mesmo se aplica ao método **upDateWindow**. Isso conclui a especialização da classe.

3.7.4 Especialização da classe *MethodWindow*

Na classe *MethodWindow*, o usuário indica quais são os métodos disponíveis para o novo problema. A sugestão óbvia é que o usuário aproveite todos os recursos já disponíveis, implementando assim um algoritmo genético/memético para seu problema. A disponibilidade de uma busca local também abre espaço para a definição de um método de *multiple start*. Inicialmente, sugerimos que se abra o arquivo **MethodWindowPMS.java**, gravando-o em seguida como **MethodWindowNewProblem.java**. As modificações são as seguintes:

`public class MethodWindowPMS extends MethodWindow`, para:
`public class MethodWindowNewProblem extends MethodWindow`

`public static MethodWindowPMS dialog;`, para:
`public static MethodWindowNewProblem dialog;`

`public MethodWindowPMS()`, para:
`public MethodWindowNewProblem()`

`dialog = new MethodWindowPMS();`, para:
`dialog = new MethodWindowNewProblem();`

No método **actionPerformed** mude as linhas:

`populationEditWindowMA = new PopulationWindowPMS();`, para:
`populationEditWindowMA = new PopulationWindowNewProblem();`

`populationEditWindowGA = new PopulationWindowPMS();`, para:
`populationEditWindowGA = new PopulationWindowNewProblem();`

Após modificar essas linhas, o usuário deve decidir se ele deseja implementar um algoritmo genético/memético e um *multiple start* para seu problema. Caso queira, o restante da classe permanece o mesmo. Caso contrário, o usuário precisará eliminar qualquer referência a todos métodos que não serão implementados. Os locais em que tais referências estão localizadas são o cabeçalho da classe e os métodos **actionPerformed**, **getValues**, **showValues**, **createObjects** e **addObjects**. De novo, sugerimos que o usuário implemente ao menos um algoritmo genético, de forma a tirar vantagem de todos os operadores que atuam em nível populacional no NP-Opt. Além disso, se o programador decidir implementar uma busca local, o algoritmo memético e o *multiple start* resultarão com um mínimo de esforço.

3.7.5 Especialização da classe *PopulationWindow*

A classe *PopulationWindow* é onde o usuário ajusta os parâmetros das populações do algoritmo genético/memético. Ela é aberta quando os botões *Edit MA Population* ou *Edit GA Population* são selecionados na janela dos métodos. De novo, comecemos abrindo uma classe pré-existente, por exemplo a **PopulationWindowPMS.java**, e renomeando-a para **PopulationWindowNewProblem.java**. As modificações são as seguintes:

`public class PopulationWindowPMS extends PopulationWindow`, para:
`public class PopulationWindowNewProblem extends PopulationWindow`

`public static PopulationWindowPMS dialog;`, para:
`public static PopulationWindowNewProblem dialog;`

`public PopulationWindowPMS()`, para:
`public PopulationWindowNewProblem()`

`dialog = new PopulationWindowPMS();`, para:
`dialog = new PopulationWindowNewProblem();`

Essas são as modificações iniciais. Agora, o usuário precisa decidir quais operadores serão implementados: tipos de *crossover*, operadores de mutação e de busca local. Índices de referência para esses operadores precisam ser listados no método **createLists**, de forma a que o usuário possa selecioná-los a partir dos menus da janela. O usuário também deve tomar cuidado com a correspondência que existe entre os tipos de operadores e os valores das constantes no cabeçalho da classe *Framework*. Por exemplo, veja o código presente no cabeçalho da referida classe:

```
public static final int UNIFORM = 0;
public static final int OX = 0;
public static final int PMX = 1;
public static final int ALTEDGEX = 2;
```

```
public static final int EDGEX = 3;
public static final int BOX = 4;
```

Os valores das constantes estão relacionados com a ordem em que os operadores de recombinação aparecem nos menus e essa ordem deve ser respeitada. Suponha, por exemplo, que o usuário implemente três tipos de *crossover* para o novo problema e que seus nomes sejam *FirstX*, *SecondX* e *ThirdX*. Neste caso, ele deverá adicionar três linhas ao código anterior:

```
public static final int FirstX = 0;
public static final int SecondX = 1;
public static final int ThirdX = 2
```

O menu de seleção do tipo de recombinação deverá ser composto pelos operadores existentes, sendo eles introduzidos na ordem correta. Assim, os índices designados pelo compilador para a variável de controle do menu estarão em harmonia com os valores das constantes.

Para a mutação, por outro lado, nenhuma modificação é necessária na classe *Framework*, pois a implementação é feita dentro da classe *Individual*. Assim, o usuário deve apenas tomar cuidado com o índice que o operador de mutação receberá no menu de seleção e utilizar o mesmo índice para indentificá-lo na classe *Individual*.

Os operadores de busca local seguem os mesmos procedimentos dos de mutação. Suas chamadas são feitas dentro da classe *Individual*, e por isso não são necessárias quaisquer mudanças em outras classes. É necessário apenas fazer a correspondência correta entre o índice recebido no menu de seleção e o método de chamada dentro da classe *Individual*. Feitas essas observações, terminamos a especialização da classe *PopulationWindow*.

3.7.6 Especialização da classe *Workspace*

A classe *Workspace* é responsável pela abertura e gravação da configuração dos parâmetros do NP-Opt para utilização futura. Como o *workspace* depende do problema que está sendo resolvido, a classe deve ser especializada. Inicialmente, abrimos o arquivo **WorkspacePMS.java**, salvando-o em seguida como **WorkspaceNewProblem.java**. As modificações começam pela mudança:

`public class WorkspacePMS extends Workspace`, para:
`public class WorkspaceNewProblem extends Workspace`

Em seguida, o usuário deve adaptar o código de leitura e de gravação para os métodos e operadores implementados. No método **loadWorkspace**, deve-se prestar atenção com o bloco `if (input.equals('<Methods>'))`. Dentro dele, o usuário vai adicionar (ou eliminar) os '*blocos-if*' de acordo com os métodos implementados para o novo problema. Se algum outro método além dos algoritmos genético/memético ou *multiple start* for introduzido, o usuário deve adicionar um novo '*bloco-if*' para esse novo método e listar todos os parâmetros necessários, de forma análoga aos outros métodos anteriormente presentes.

No método **saveWorkspace**, o usuário deve também realizar as mudanças para adaptar o código aos novos parâmetros/métodos introduzidos. Mantenha-se atento ao fato de que tudo o que for gravado pelo método **saveWorkspace** deverá ser lido no **loadWorkspace**. Desta forma, ambos os métodos estão intimamente relacionados.

3.7.7 Especialização da classe *ParametersWindow*

Nesta parte, o usuário irá especializar a classe utilizada para ajustar os parâmetros do *multiple start*. Se algum operador de busca local estiver disponível para o novo problema essa janela deve ser criada. Como ponto inicial, vamos abrir o arquivo **ParametersWindowPMS.java** e salvá-lo como **ParametersWindowNewProblem.java**. As modificações começam com as seguintes mudanças nas linhas:

`public class ParametersWindowPMS extends ParametersWindow`, para:
`public class ParametersWindowNewProblem extends ParametersWindow`

`public static ParametersWindowPMS dialog;`, para:
`public static ParametersWindowNewProblem dialog;`

`public ParametersWindowPMS()`, para:
`public ParametersWindowNewProblem()`

`dialog = new ParametersWindowPMS();`, para:
`dialog = new ParametersWindowNewProblem();`

Depois de modificar essas linhas, o usuário decide quais parâmetros serão ajustáveis. Se há apenas a busca local, então nenhuma outra mudança é necessária. Se houver alguma estratégia de redução de vizinhança, ela também deverá ser incluída nessa janela. Nesse caso, a melhor opção é utilizar o arquivo **ParametersWindowSMS.java** como ponto de partida, pois ele implementa reduções de vizinhança para as buscas locais.

A modificação do restante do arquivo é feita de forma análoga à da classe *PopulationWindow* e deve incluir alterações nos métodos **openWindow**, **getValues**, **showValues**, **createLists** e **createFrame**, de forma que os operadores possam ser selecionados a partir dos menus que compõem a janela.

3.7.8 Modificações na classe *BatchMethod*

A modificação dessa classe é bem simples. A mudança se resume à adição de um bloco no método **readData**:

```
if (problemType == NEWPROBLEM)
{
  workspace = new WorkspaceNewProblem();
  instanceToSolve = new InstanceNewProblem();
}
```

Esta é a única modificação necessária.

3.7.9 Especialização da classe *Instance*

A classe *Instance* é responsável pela declaração das variáveis que representam o problema. Analogamente às modificações anteriores, comecemos abrindo a classe **InstancePMS.java** e gravando-a como **InstanceNewProblem.java**. Essa classe possui um método principal, chamado **readInstanceFile**, que é responsável pela leitura da instância a partir de um arquivo de dados. O usuário deve modificar esse método para torná-lo capaz de ler toda a informação contida no arquivo que define uma instância do novo problema. Essa classe pode ainda conter outros métodos, utilizados por exemplo para processar informação enquanto o arquivo é lido, mas isso depende exclusivamente do formato da instância e do problema. Assim, o usuário deve verificar se algum processamento pós-leitura é necessário e assim criar outros métodos além do **readInstanceFile**.

Para fazer o NP-Opt lidar com o novo problema, é preciso ainda informar quais são as novas variáveis, o que é feito modificando-se a classe *Instance*. As modificações devem se restringir à inserção das variáveis do novo problema. Como exemplo, vamos ver como isso foi feito para o problema de SMP. Na classe *Instance*, está escrito na forma de comentário que o SMP utiliza as mesmas variáveis do problema de SMS. São elas *sij*, *soj*, *tp*, *numberOfMachines*, *numberOfJobs* e *sizeOfIndividual*. As duas primeiras estão relacionadas aos tempos de *setup* entre as tarefas. A variável *tp* é o tempo de processamento de cada tarefa. *NumberOfMachines* e *numberOfJobs* se referem ao número de máquinas e ao número de tarefas, respectivamente. Finalmente, *sizeOfIndividual* representa o tamanho do cromossomo, que nesse caso é *numberOfJobs* + *numberOfMachines*. Como todas as variáveis necessárias para descrever o SMP já estavam presentes, nenhuma modificação foi necessária. O usuário portanto deve checar quais variáveis já presentes podem ser aproveitadas pelo seu problema. Incentivamos a reutilização das variáveis pois as modificações no código do NP-Opt se tornam ainda menores.

3.7.10 Especialização da classe *Individual*

A classe *Individual* relaciona todas as características do indivíduo, e vários dos métodos que atuam nesse nível. Estão incluídos nessa classe a representação, cálculo do *fitness*, operadores de mutação e procedimentos de inicialização do indivíduo. Inicialmente, deve-se abrir um arquivo-base, por exemplo **IndividualPMS.java** e salvá-lo como **IndividualNewProblem.java**. As modificações começam com as alterações nas linhas:

`public class IndividualPMS extends Individual`, para:
`public class IndividualNewProblem extends Individual`

`public IndividualPMS(int sizeOfChromosome)`, para:
`public IndividualNewProblem(int sizeOfChromosome)`

*`ls = new LocalSearchPMS();`, para:
*`ls = new LocalSearchNewProblem();`

*Esta linha só deve ser modificada se for implementada uma busca local para o novo problema. Caso contrário, ela deverá ser apagada.

`dialog = new PopulationWindowPMS();`, para:
`dialog = newPopulationWindowNewProblem();`

No método **fitness** o usuário deve implementar o cálculo da função objetivo. O NP-Opt está preparado para trabalhar diretamente com a função objetivo, buscando sempre a sua minimização. Assim, cada vez que o *fitness* de dois indivíduos é comparado, na realidade o NP-Opt estará comparando o valor de suas funções objetivos. Vale ressaltar que se o problema original for de maximização, deve-se utilizar algum recurso matemático para transformá-lo em um problema de minimização, como por exemplo tomar o inverso ou fazer uma multiplicação por -1.

Uma rápida verificação irá revelar que existem dois métodos **fitness**. O segundo deles possui três parâmetros de entrada, sendo um chamado *fitnessTolerance*. Esse método é utilizado quando há um limitante superior disponível e a função objetivo é calculada cumulativamente. Assim, o cálculo pode ser interrompido no meio do processo, caso o valor parcial ultrapasse esse limitante superior. Essa segunda versão do método **fitness** é bastante útil quando se está efetuando a busca local em problemas em que o cálculo da função objetivo é custoso. Nesse caso, o limitante superior pode ser fixado como sendo o valor da solução antes do movimento da busca local. Caso esse movimento gere uma piora da qualidade do indivíduo, pode-se economizar tempo computacional se essa piora for detectada antes do cálculo completo da função objetivo.

No método **mutate** são implementadas as chamadas para os operadores de mutação implementados. Como só há uma mutação para o problema de SMP, há apenas uma proposição '*if*' e também uma única chamada para o método **swapMutation**. Se for implementada mais de uma mutação para o novo problema, o usuário deverá listar neste método todas as chamadas para os diversos operadores. Finalmente, temos o método **initializeIndividual**, que é utilizado para criar os indivíduos iniciais. Aqui o usuário define como os cromossomos serão inicialmente preenchidos.

Como a maior parte desses métodos é chamado a partir de outros métodos que atuam no nível das populações, o usuário não deve mudar quaisquer especificações, como tipo de variável de retorno ou variáveis de entrada, pois nesse caso certamente ocorrerão diversos erros de compilação.

3.7.11 Especialização da classe *Crossover*

Na classe *Crossover* são implementados os operadores de recombinação para o novo problema. Começamos abrindo a classe **CrossoverOXPMS.java** e gravando-a em seguida como **CrossoverNewProblem.java**. As modificações começam na linha:

`public class CrossoverOXPMS extends Crossover`, para:
`public class CrossoverNewProblem extends Crossover`

Existem dois tipos de chamada para os métodos de recombinação. O primeiro possui três parâmetros de entrada - dois indivíduos e a instância - e é especialmente indicado quando o *crossover* necessita de informação a respeito da instância para efetuar a recombinação genética. O segundo método

de recombinação possui apenas os dois indivíduos-pais como parâmetros de entrada. Ambos os métodos retornam o mesmo tipo de variável: um novo indivíduo.

Caso o usuário queira implementar um tipo mais complexo de recombinação, com uso de múltiplos pais ou outras características sofisticadas, ele deverá criar um método específico. Nesse caso, ainda será necessário incluir uma declaração no arquivo **Crossover.java**, para que o novo *crossover* possa ser chamado a partir de métodos localizados em outras classes. A declaração seguiria um modelo parecido com:

```
public abstract Individual executeCrossover(Individual ind1,
                    Individual ind2, Individual ind3, float A, float B);
```

Como a classe *Crossover* é abstrata, todos os outros arquivos *crossover* que estendem a classe abstrata devem implementar esse mesmo método, mas sem nenhum código 'útil', por assim dizer. Assim, a inclusão desse método nas outras classes derivadas da *Crossover* deve seguir um modelo semelhante ao do método com três parâmetros de entrada do arquivo **CrossoverOXPMS.java**. A implementação poderia ser algo simples, como:

```
public Individual executeCrossover(Individual ind1, Individual ind2,
                       Individual ind3, float A, float B) {return ind1;}
```

3.7.12 Especialização da classe *LocalSearch*

A especialização da classe *LocalSearch* só é necessária se o usuário for implementar uma busca local para o novo problema. De início, vamos abrir o arquivo **LocalSearchPMS.java** e gravá-lo como **LocalSearchNewProblem.java**. As modificações começam com a mudança da linha:

`public class LocalSearchPMS extends LocalSearch`, para:
`public class LocalSearchNewProblem extends LocalSearch`

No método **localSearch**, o usuário deve implementar as chamadas para os métodos de busca local. No caso do problema de SMP, o menu de seleção irá assinalar os índices 0, 1 e 2 para as buscas locais *Both*, *Swap* e *Insertion*, respectivamente. Essa correspondência deve ser observada nas proposições '*if*' dentro do método **localSearch**. O procedimento é análogo à correspondência feita entre os tipos de recombinação e o índice assinalado pelo menu de seleção de tipo de recombinação.

O próximo passo consiste da implementação das buscas locais, onde cada uma é implementada em um método diferente. A busca local em geral recebe um indivíduo e a instância como parâmetros de entrada, porém mais informação pode ser utilizada. Neste caso, o usuário deverá criar um novo método e fazer modificações no arquivo **LocalSearch.java**. Também serão necessárias mudanças em todas as outras classes de busca local, pois a *LocalSearch* é uma classe abstrata. O procedimento completo é análogo ao do *crossover*.

3.7.13 Modificações nas classes *Memetic/Genetic/MultipleStart*

As classes *Memetic*, *Genetic* e *MultipleStart* implementam os algoritmos memético, genético e de *multiple start*, respectivamente. É necessária uma única modificação. Em cada uma das três classes, dentro dos métodos homônimos, adicione a linha:

```
if (problemType == NEWPROBLEM)
         {newIndividual = new IndividualNewProblem (sizeOfIndividual);}
```

3.7.14 Modificações na classe *Community*

A classe *Community* implementa os métodos necessários para o uso de múltiplas populações nos algoritmos memético/genético . É necessária uma modificação. No método construtor, adicione a linha:

```
if (problemType == NEWPROBLEM)
{
   this.bestIndividual = new IndividualNewProblem(sizeOfIndividual);
   this.bestIndividual.fitness = UPPERBOUND;
}
```

3.7.15 Modificações na classe *Population*

A classe *Population* possui toda a informação necessária para fazer os algoritmos memético/genético trabalharem com populações de indivíduos. Os métodos incluem seleção de indivíduos para recombinação, inserção dos novos indivíduos, chamadas para os operadores de recombinação, entre outros. As modificações são as seguintes. No método **Population**, adicione o bloco:

```
if (problemType == Framework.NEWPROBLEM)
{
  this.ind = new IndividualNewProblem[sizeOfPopulation];
  bestIndividual = new IndividualNewProblem(sizeOfIndividual);
}
```

Ainda no método **Population**, dentro do laço '*for*', adicione a linha:

```
if (problemType == Framework.NEWPROBLEM)
            {this.ind[i] = new IndividualNewProblem(sizeOfIndividual);}
```

No método **Recombine**, o usuário deverá listar todos os operadores de recombinação implementados para o novo problema. Considere os três operadores introduzidos anteriormente, cujos nomes eram *FirstX*, *SecondX* e *ThirdX*. O bloco a ser adicionado é da forma:

```
if (problemType == Framework.NEWPROBLEM)
{
  newIndividual = new
           IndividualNewProblem(this.ind[parents[0]].sizeOfIndividual);
```

```
  if (crossoverType == Framework.FirstX)
                                 {crossover = new CrossoverFirstX();}
  if (crossoverType == Framework.SecondX)
                                {crossover = new CrossoverSecondX();}
  if (crossoverType == Framework.ThirdX)
                                 {crossover = new CrossoverThirdX();}
  newIndividual =
          crossover.executeCrossover(this.ind[parents[0]],
                                         this.ind[parents[1]]);
}
```

Estas são as modificações necessárias para a classe *Population*.

3.7.16 Modificações na classe *Slave*

A classe *Slave* pertence ao conjunto de classes responsáveis pela distribuição dos processos de busca local. São necessárias apenas duas mudanças. No método **setInstanceToSolve**, dentro do comando '*switch*', adicione uma nova opção '*case*':

```
case Framework.NEWPROBLEM:
                      this.setInstanceToSolve(new InstanceNewProblem());
```

No método **setIndividual**, também dentro do comando '*switch*', adicione uma nova opção '*case*':

```
case Framework.NEWPROBLEM: this.ind.ls = new LocalSearchNewProblem();
```

Com a classe *Slave* atualizada, terminamos as modificações do NP-Opt para a inclusão do novo problema. Apesar de não serem poucas, as mudanças são em geral bastante simples. Agora, vamos analisar alguns pontos-chave do NP-Opt, onde o usuário pode ajustar o *software* para suas necessidades específicas.

3.8 Adaptação do NP-Opt

Nesta seção vamos fornecer informações sobre alguns pontos-chave do NP-Opt. Com essas informações, o usuário poderá efetuar modificações em propriedades específicas do *software*, adaptando-o para uma maior conveniência de uso.

3.8.1 Ajuste da população inicial

A população inicial é criada dentro da classe *Population*, no método **generatePopulation**. Caso se deseje uma população inicial com características especiais, como por exemplo um dos indivíduos sendo composto por uma solução heurística, há duas possibilidades. A primeira consiste em criar uma chamada dentro do método **generatePopulation** para a heurística em questão. A segunda é atuando no método **initializeIndividual**, pertencente à classe *Individual*. Este método possui entre os parâmetros de entrada, um valor inteiro que pode ser utilizado para selecionar um procedimento especial de criação de indivíduos.

3.8.2 Ajuste da política de seleção dos pais

A seleção dos pais é definida também na classe *Population*, no método **selectIndividuals**. Os tipos de seleção estão relacionados ao uso ou não de populações estruturadas. Assim, caso se queira que a população não tenha hierarquia alguma, o método utilizado é o **chooseParentsFree**. Por outro lado, se se desejar que a população seja hierarquicamente estruturada, o método utilizado é o **chooseParentsClusters**, que recebe o tipo de hierarquia como parâmetro. Qualquer outro tipo de seleção dos pais deve ser introduzido nesta parte do *software*.

3.8.3 Ajuste da política de aceitação dos novos indivíduos

O NP-Opt utiliza duas políticas de inserção dos novos indivíduos. A regra '*Only if better*' estabelece que um novo indivíduo só será aceito na população se ele for melhor que um dos pais. Caso contrário ele é descartado. A segunda opção é a '*Always accept*', onde o novo indivíduo é sempre aceito, tomando o lugar de um dos pais. Caso o usuário decida utilizar uma outra política, ele pode especificá-la no método **insertIndividual**, dentro da classe *Population*. Deve-se ressaltar que quaisquer mudanças devem ser acompanhadas também, caso necessário, por mudanças na classe *PopulationWindow* para introduzir as novas políticas de inserção no menu correspondente localizado na janela de ajuste das populações.

3.8.4 Ajuste do critério de convergência da população

A dinâmica dos algoritmos genéticos/meméticos fazem a população perder diversidade continuamente, e após um certo número de gerações se torna muito difícil obter qualquer melhora na solução incumbente. Quando isso ocorre, dizemos que a população convergiu. Para aumentar a eficiência do processo de busca, é necessário adicionar diversidade, o que pode ser feito reinicializando-se a população. Utilizando o critério mais restritivo de aceitação de novos indivíduos, a população é reinicializada somente se ao longo de uma geração inteira, nenhum novo indivíduo for aceito para inserção. Essa é uma forte indicação de perda de diversidade. No entanto, o usuário pode querer utilizar outro critério, e nesse caso, as modificações devem se concentrar nos métodos **memeticMainLoop** e **geneticMainLoop**, localizados nas classes *Memetic* e *Genetic*, respectivamente. Há uma variável *booleana* de controle da convergência, chamada *populationHasConverged*. Toda vez que ela assume um valor *true*, a população é reinializada. Qualquer critério de convergência deve fazer referência a essa variável, atribuindo-lhe o valor *true* quando o novo critério de convergência for satisfeito.

A forma como a reinicialização é feita também pode ser modificada pelo usuário. O método responsável por essa parte é o **restartCommunity**, e está localizado na classe *Community*. A linha que chama o procedimento de reinicialização é:

```
this.pop[i].generatePopulation(instanceToSolve, startIndividual);
```

Ao se examinar o método **mutatePopulation**, localizado na classe *Population*, vê-se que todos os indivíduos da população são reinicializados, começando no indivíduo *startIndividual*. Esta variável de controle é necessária quando se está utilizando a população estruturada e se deseja preservar o melhor indivíduo após a reinicialização. Nesse caso, como o melhor indivíduo está sempre na posição 0, ela não deve ser incluída no processo de mutação, sendo assinalado o valor 1 para a variável de controle *startIndividual*.

A reinicialização pode ser encarada como um processo de randomização, pois cada indivíduo passa por $3.n$ mutações do tipo *troca de alelos*, onde n é o tamanho do cromossomo. Após tantas trocas, o indivíduo resultante terá as mesmas características de outro qualquer gerado aleatoriamente.

3.8.5 Ajuste das políticas de migração

Ao se utilizar múltiplas populações, o usuário pode desejar modificar as políticas de migração. A decisão de quais populações irão trocar indivíduos é feita no método **blendPopulations**, localizado na classe *Community*. Como o NP-Opt possui cinco políticas disponíveis, este método possui também cinco opções dentro do bloco '*switch*'. Cada opção implementa uma política de migração diferente, indexadas na mesma ordem em que aparecem no menu de seleção da janela de migração. Qualquer mudança na política de migração deve ser feita nesse método.

Depois de definir quais populações vão trocar indivíduos, pode-se ainda modificar a forma como fazem isso. O método **migrateIndividuals**, localizado na classe *Community*, recebe como parâmetro duas populações. Atualmente esse método seleciona o melhor indivíduo de uma delas e migra uma cópia dele para a outra, fazendo-o ocupar uma posição aleatória. A variável de retorno é a população que recebeu o indivíduo copiado. Essa estratégia pode ser facilmente modificada para melhor se adequar às necessidades do usuário.

3.8.6 Ajuste das buscas locais

Por *default*, a busca local é aplicada em todo novo indivíduo criado através de recombinação, logo após a etapa de mutação. No entanto, algumas vezes essa escolha pode tornar a complexidade computacional muito grande. Nesse caso, pode-se optar por critérios mais 'leves', como por exemplo aplicar a busca local:

- Ao fim de cada geração, apenas ao melhor indivíduo da população.
- Ao fim de cada geração, a uma porcentagem da população.
- Somente após a população convergir e apenas ao melhor indivíduo da população.
- Somente após a população convergir e a uma porcentagem da população.

Para implementar essas opções, o usuário deve modificar o método **memeticMainLoop**, localizado na classe *Memetic*. A linha responsável por chamar a busca local é:

```
newIndividual.ls.localSearch(newIndividual,
              localSearchType, neighborReductionType, instanceToSolve);
```

Essa linha pode ter sua posição modificada para melhor atender às necessidades do usuário. Além da sua posição, pode-se modificar a variável referente ao indivíduo que passará pela busca local. Por exemplo, para efetuar a busca local sobre o i-ésimo indivíduo da população, deve-se modificar a linha anterior para:

```
pop.ind[i].ls.localSearch(pop.ind[i],
              localSearchType, neighborReductionType, instanceToSolve);
```

Para fazer a busca local ao final de uma geração completa, deve-se posicionar a linha de chamada após o fechamento do laço '*for*' das recombinações:

```
for (i = 0; i < noImprovementLimit; i++)
```

Para fazer a busca local somente após a população convergir, por sua vez, deve-se posicionar a linha de chamada após o fechamento do laço '*while*' de verificação de convergência:

```
while (!populationHasConverged)
```

Deve-se ainda tomar cuidado para, ao se utilizar população estruturada, efetuar a busca local sempre depois da reestruturação da mesma. Ou seja, colocar a chamada somente depois da execução da linha:

```
pop.arrangePopulation(populationStructure);
```

3.8.7 Ajuste da estrutura da população

Há duas estruturas populacionais disponíveis no NP-Opt: árvores binária e ternária. Caso o usuário deseje experimentar estruturas diferentes, serão necessárias algumas modificações, começando pela classe *Population*. No método **selectIndividuals** deve-se incluir uma chamada para a estratégia de seleção dos indivíduos que se adapte à nova estrutura populacional. Crie um novo método, análogo ao **chooseParentsClusters** e **chooseParentsFree**, descrevendo como os pais são selecionados no novo ambiente estruturado.

Em seguida, modifique o método **arrangePopulation**, caso a estrutura seja hierárquica, para efetuar o rearranjo da população de acordo com algum critério específico. Por fim, insira a nova estrutura disponível no menu da janela das populações para que ela possa ser selecionada pelo usuário. Isso é feito adicionando-se uma linha ao método **createLists**, em todas as classes do tipo *PopulationWindow* existentes:

```
populationStructureType.add("New structure name");
```

Vale ressaltar que, como as modificações no nível populacional são herdadas por todos os problemas, o usuário poderá testar a nova estrutura populacional em qualquer um dos problemas disponíveis. É importante ainda que se faça a correspondência correta entre o índice designado para a nova estrutura no menu de seleção da janela das populações e os identificadores de tipo de estrutura utilizados no restante do framework.

3.9 Resumo

Este capítulo descreve as principais características do software NP-Opt. Inicialmente avaliamos as motivações do uso de programação orientada a objetos, assim como as características necessárias para um software desse tipo. O NP-Opt atualmente aborda cinco problemas da classe NP-hard. O nível de reutilização dos códigos é bastante alto - acima de 75% - e o programa apresenta uma interface gráfica de comunicação bem agradável e de manejo simples.

São apresentadas todas as 14 janelas gráficas, assim como seus componentes. Em seguida, fazemos uma descrição do modelo de processamento distribuído adotado no NP-Opt - a arquitetura mestre-escravo e alguns detalhes da implementação.

A estrutura das classes do software também é abordada, com a apresentação de três diagramas mostrando a hierarquia de classes e a divisão das mesmas em grupos. A especialização de algumas das classes, necessária para fazer o software operar problemas distintos, também é discutida.

Mostramos em seguida o processo para a inclusão de um novo problema no NP-Opt. Esse é um ponto muito importante, pois fornece um guia das modificações necessárias para que o software passe a abordar um novo problema a critério do usuário. Nessa parte são indicadas todas as modificações necessárias no código do NP-Opt, entre elas as alterações no código pré-existente e as novas classes que devem ser criadas.

Por fim, são feitas algumas observações sobre a adaptação do NP-Opt, ou seja, o que o usuário deve alterar no código do software para torná-lo mais adaptado às suas necessidades. Mais especificamente abordamos possíveis modificações na população inicial, seleção dos pais, inserção de novos indivíduos, entre outros, totalizando sete pontos importantes onde o usuário pode ter interesse em modificar o código original.

Capítulo 4

Localização de Capacitores

4.1 Introdução

[1] Capacitores são fontes de energia reativa. Os objetivos de sua aplicação em sistemas de potência é a compensação de energias reativas produzidas por cargas indutivas ou reatâncias de linhas. Quando adequadamente utilizados, permitem a obtenção de um conjunto de benefícios correlatos que incluem a redução de perdas de energia, correção dos perfis de tensões, controle dos fluxos de potência, melhoria do fator de potência e aumento da capacidade dos sistemas. No contexto desse trabalho, a instalação de capacitores é avaliada conjuntamente sob a ótica de redução de perdas e do consequente aumento do lucro na distribuição de energia. Aspectos operacionais também são levados em conta, pois geram restrições que não podem ser desprezadas quando se planeja instalar capacitores em uma rede elétrica real.

As perdas técnicas podem ser reduzidas pela instalação de capacitores em pontos adequados da rede, proporcionando 'geração' de energia reativa nas proximidades das cargas. Dessa forma, diminui-se (ou, no limite, elimina-se) o componente associado ao fluxo de corrente reativa nas linhas.

Os benefícios reais obtidos com a instalação de capacitores em sistemas de distribuição dependem das características dos equipamentos e da forma como é feita essa instalação. Especificamente, dependem do número e tamanho dos capacitores, de sua localização, do tipo (fixos ou chaveados) e do esquema de controle utilizado. Neste trabalho, o *Problema de Localização de Capacitores* (PLC) tratado restringe-se ao problema de encontrar a localização, o número e a dimensão dos capacitores a serem instalados.

Antes da década de 50 os capacitores para redução de perdas eram colocados nas subestações, no início dos alimentadores. Com a constatação da vantagem de instalá-los em pontos mais próximos às cargas e do aparecimento de equipa-

[1] Este capítulo é baseado no artigo:

A. Mendes, P. França, C. Lyra, C. Pissarra e C. Cavelucci. **An Evolutionary Approach for Capacitor Placement in Distribution Networks**. Proceedings (CD-ROM apenas - 6 páginas) do *EIS2002 - 3rd International NAISO Symposium on Engineering of Intelligent Systems*, Málaga, Espanha, Setembro, 2002.

mentos de menor porte, que podiam ser instalados em postes de distribuição, o problema de encontrar sua melhor localização se tornou mais complexo. A complexidade também se deve ao fato das redes reais serem muito grandes - normalmente compostas por milhares de seções - e aos efeitos da instalação dos capacitores em uma parte da rede se progragarem por toda ela.

Entre os trabalhos sobre o assunto, convém citar um *survey* das técnicas de solução já propostas, que pode ser encontrado em Ng *et al.* [60]. Dentre os muitos enfoques possíveis, destacam-se as metaheurísticas, já que métodos exatos não são adequados para tratar redes de tamanho real. Na categoria das metaheurísticas, destacam-se o trabalho de Chiang *et al.* [35] que utiliza a técnica de *simulated annealing*, o artigo de Gallego *et al.* [24], que aplica uma busca tabu e o artigo de Huang *et al.* [39], que propõe um enfoque imunológico. Entretanto, a maior parte das propostas recentes utiliza os algoritmos genéticos [26, 48, 57, 67].

Neste capítulo é apresentada uma nova abordagem via algoritmos meméticos para a solução do PLC. A contribuição mais importante está no emprego de uma estratégia de busca local desenvolvida utilizando informações próprias do problema. Essa busca local leva em conta aspectos tanto operacionais quanto de detalhes de codificação que influenciam fortemente o desempenho do método. Outras características incluem o uso de uma estrutura populacional hierárquica e a consideração de restrições adicionais como limite anual de orçamento para investimento em capacitores. A eficiência do método permite sua utilização em redes reais com milhares de barras, obtendo soluções de alta qualidade em poucos minutos de processamento.

Este problema não faz parte do NP-Opt pois havia a necessidade constante de comunicação externa com rotinas desenvolvidas em C++. Essas rotinas são responsáveis pelos cálculos do fluxo de potência e das perdas elétricas. Assim, optamos por resgatar as classes do NP-Opt e traduzi-las para C++, visando o seu uso em conjunto com as rotinas já existentes.

4.2 Descrição do problema

Considere uma rede de distribuição, de estrutura radial, constituída por diversas subestações e alimentadores, além de cargas distribuídas de maneira não uniforme ao longo da mesma. O objetivo é determinar a localização e o tamanho dos bancos de capacitores a serem instalados de forma a reduzir as perdas elétricas, aumentando assim o lucro financeiro oriundo da distribuição. Para tanto, devem ser consideradas variáveis como:

- Número máximo de capacitores a serem instalados (restrição operacional determinada pela equipe de manutenção da companhia).
- Orçamento máximo disponível anual para instalação de bancos capacitores (restrição imposta pelo departamento de finanças da companhia).
- Custo de compra e instalação de cada banco capacitor.
- Prazo de amortização do investimento.

A formulação matemática do problema é bastante complexa. O PLC pode ser formulado como um problema de otimização não-linear inteira mista, onde a função objetivo consiste em minimizar as perdas de energia e o investimento necessário. As principais restrições englobam a satisfação da carga em cada seção, restrições operacionais, como perfil de voltagem, além da magnitude da corrente em cada ramo da rede referente a cada perfil de carga. O caráter não-linear é devido ao cálculo das perdas na função objetivo, enquanto a integralidade deve ser forçada por causa das variáveis de decisão binárias utilizadas para alocar os capacitores na rede. Detalhes do modelo matemático utilizado podem ser encontrados nos trabalhos de Baran e Wu [6, 7].

4.3 Representação utilizada

Nessa seção descreveremos o tipo de representação utilizado, ou seja, como uma configuração de capacitores pode ser codificada em termos de um cromossomo. A representação escolhida para o PLC é a de um cromossomo cujos alelos assumem valores binários em uma parte dele e inteiros em outra. A primeira parte do cromossomo codifica os locais candidatos (as seções dos alimentadores) que serão usados para possível instalação dos capacitores. Se o alelo correspondente à posição i tem valor 1, isso significa que deverá ser instalado um capacitor na seção i, caso contrário a seção não receberá capacitores. Desta forma, a dimensão dessa *string* é igual ao número de seções. A segunda parte do cromossomo codifica o valor, em kVar, da capacidade dos capacitores instalados. Na Figura 4.1 mostramos um exemplo de uma solução para um problema com seis seções de alimentador.

[0 1 1 0 0 1 / 3 5 2 1 2 4]
seções **capacidades dos capacitores**

Figura 4.1: *Codificação do cromossomo para o PLC.*

Índice	**Capacidade em kVar**	**Custo em Reais**
1	150	3.595,00
2	300	3.850,00
3	450	3.889,00
4	600	4.375,00
5	900	6.120,00
6	1200	7.092,00

Tabela 4.1: *Dados dos capacitores utilizados.*

Analisando a Figura 4.1 e a Tabela 4.1 em conjunto, vê-se que as seções 2, 3 e 6 receberiam capacitores de capacidade 900, 300 e 600 kVar, respectivamente, e o custo total seria de R$ 14.345,00. As outras seções permaneceriam

sem capacitores, pois seus alelos correspondentes na 1ª parte do cromossomo assumiram valor 0. Nestes casos, os valores de kVar da 2ª parte do cromossomo são ignorados quando se calcula as perdas elétricas correspondentes à essa solução.

4.4 População inicial

A população inicial é criada de forma totalmente aleatória, tanto na localização dos bancos capacitores quanto nos seus respectivos tamanhos. Em nossa implementação, aproximadamente 20% dos locais candidatos recebem inicialmente um capacitor. A capacidade atribuída a esses capacitores varia entre os valores de 150 kVar e de 1200 kVar, com uma concentração maior na região entre 300 e 600 kVar. Esses parâmetros são apenas aproximados e visam dar um ponto de partida para o algoritmo que seja mais próximo ao que se espera de uma solução de boa qualidade. Apesar do aparente exagero em se fazer 20% das seções receberem capacitores inicialmente, do ponto de vista da dinâmica do algoritmo é bom começar com mais capacitores que o necessário e ir aos poucos limpando a rede. Isso é ainda mais importante quando se está utilizando um algoritmo genético puro, sem buscas locais. Para os algoritmos meméticos, no entanto, testes com outras configurações de população inicial foram realizados, sempre gerando resultados próximos. A conclusão é que a configuração inicial, apesar de ajudar, não é fundamental neste caso. Acreditamos que as buscas locais, por atuarem nas duas partes do cromossomo e por agirem de maneira complementar, corrigem qualquer distorção presente na população inicial.

4.5 Operador de recombinação

O fato do cromossomo ser composto por duas partes distintas exige que o operador mantenha-as separadas ao longo do processo de recombinação. Assim, são duas as estratégias de recombinação: uma para a parte binária do cromossomo e outra para a parte inteira. Na parte binária foi adotado um *crossover* uniforme, onde o alelo do filho é determinado escolhendo-se aleatoriamente um dos dois pais e copiando o valor presente nesse pai. Como consequência, se os pais possuem o mesmo alelo em uma determinada posição, o filho herdará esse valor. Se os valores forem distintos, o filho poderá herdar tanto o valor 0 quanto o valor 1, com a mesma probabilidade para ambos. Já na parte inteira, faz-se uma média dos valores encontrados nos pais, ou seja, somam-se os valores inteiros presentes na mesmo posição dos dois pais e divide-se por dois. Este será o valor passado para o filho. Se a soma der um valor ímpar e a divisão por dois não for inteira, efetua-se um arredondamento, para cima ou para baixo, aleatoriamente.

No exemplo da Figura 4.2, temos na 1ª posição da 2ª parte do cromossomo valores 3 e 5 para os pais. Assim, o filho recebeu um 4 nessa posição. Na posição 5, os valores dos pais são 4 e 3. A média de 3,5 foi arredondada para cima por uma escolha puramente aleatória.

	1ª parte						2ª parte					
Posição	1	2	3	4	5	6	1	2	3	4	5	6
Pai A	0	1	0	1	1	0	3	2	1	1	4	2
Pai B	1	1	1	1	0	1	5	1	5	3	3	2
Filho	1	1	0	1	0	1	4	1	3	2	4	2

Figura 4.2: *Operador de recombinação.*

Como pode-se notar, a característica mais marcante desse procedimento é a preservação por completo das características comuns dos pais. Caracteres conflitantes são escolhidos de forma aleatória, na parte binária, e através de uma média, na parte inteira. Quanto à taxa de recombinação, foram testados valores no intervalo $[1, 3]$. No final, optou-se pela criação de 20 indivíduos por geração, o que equivale a uma taxa de crossover de 1,5, levando-se em conta que a população tem um tamanho de 13 indivíduos (uma árvore ternária de três níveis). À primeira vista, este valor parece ser muito alto, mas devido à política de inserção de novos indivíduos ser muito restritiva, a dinâmica do algoritmo fica equilibrada, conforme explicado no Capítulo 2.

4.6 Mutação

O operador de mutação visa agregar diversidade à população de indivíduos. Apesar de possuir apenas um papel secundário no algoritmo memético implementado, optamos por sua inclusão no mesmo. O operador escolhido possui duas partes: a primeira altera a porção binária do cromossomo, escolhendo aleatoriamente uma única posição do indivíduo e trocando o valor de seu alelo (*bit-swap*) pelo valor complementar. Ou seja, se o valor era 0, passa a ser 1, e vice-versa. A segunda parte age nos valores inteiros do cromossomo, escolhendo aleatoriamente uma única posição do indivíduo e somando ou subtraindo uma unidade de seu valor. A escolha de somar ou subtrair é também determinada aleatoriamente. A mutação é aplicada em 10% dos novos indivíduos gerados. Em geral, valores altos na taxa de mutação devem ser evitados pois acabam por adicionar ruído ao processo evolutivo, ou pior, eliminar boas características já presentes nos cromossomos.

4.7 Busca local

A busca local adotada para o PLC engloba três buscas locais aplicadas em sequência. Elas são complementares e atuam em características distintas do problema. Foi necessário um estudo bastante prolongado, com testes abrangendo diversas estratégias, até que se chegasse à configuração final. Todos os passos, erros e acertos serão descritos detalhadamente. A seguir descrevemos as três buscas locais utilizadas.

4.7.1 Busca local *Add/Drop*

Nessa estratégia, somente a primeira parte do cromossomo é alterada, ou seja, a busca local é feita apenas na localização dos capacitores. Cada *bit* do cromossomo é alterado, de forma sequencial, para seu valor complementar e verifica-se se tal mudança ocasionou uma melhora da função objetivo. Assim, um local candidato escolhido para receber capacitores é desativado (*Drop*) ou um local vazio é suprido com capacitores (*Add*). Se houve uma melhora, a mudança é mantida e passa-se ao *bit* seguinte. Se não houve melhora, o *bit* mudado retorna ao valor original e de novo passa-se ao seguinte. Essa busca local é aplicada uma única vez, independentemente se houve melhora ou não da função objetivo. Essa escolha visa reduzir a complexidade computacional do processo, que foi um problema complicado desde o início devido ao tamanho da instância tratada.

4.7.2 Busca local de capacidade

Essa busca local atua sobre o tamanho dos capacitores, ou seja, na segunda parte do cromossomo. O processo tenta encontrar o tamanho ideal para cada posição onde se pretende instalar um capacitor. Essa busca local testa os tamanhos imediatamente inferior e superior do capacitor atualmente instalado. Por exemplo, se o capacitor instalado em uma certa posição é de 600 kVar, testa-se o de 450 kVar e o de 900 kVar. Se há alguma melhora, o valor é modificado. Essas tentativas são, a exemplo do *Add/Drop*, feitas capacitor a capacitor, um por vez. Ressaltamos que somente são testadas as posições cujo correspondente na primeira parte do cromossomo tenha valor 1. Isso faz a complexidade desta busca ser bem menor que a *Add/Drop*. No entanto, apesar da menor complexidade, mantivemos a política de aplicá-la uma única vez, sempre visando reduzir o tempo gasto no processo.

4.7.3 Busca local *Swap*

Essa estratégia atua de novo na primeira parte do cromossomo, tentando retirar um capacitor de uma posição e colocá-lo em outra. Com isso, mantém-se inalterado o número de capacitores instalados. A busca é feita percorrendo-se a primeira parte do cromossomo por inteiro. Cada vez que um valor 1 é encontrado, troca-se ele por 0 e testa-se em sequência a troca de todos os valores 0 por 1, um por vez. Ou seja, é como se o capacitor fosse retirado da sua posição original e testado em todas as outras posições até então vazias. Salientamos ainda que são trocados apenas os valores da primeira parte do cromossomo, permanecendo a segunda parte inalterada. Assim, apesar do número de capacitores permanecer igual, a capacidade total instalada pode variar. Esta busca é complementar à *Add/Drop* e, dadas algumas condições, muito necessária.

4.7.4 Características especiais

O desempenho das buscas locais foi excelente sempre que não houve restrições quanto ao número de capacitores a serem instalados. Porém, quando foi testada

a possibilidade de se estabelecer um orçamento máximo a ser gasto com capacitores, ocorreram perturbações indesejáveis que comprometeram fortemente o desempenho do algoritmo.

A busca local atua sequencialmente em cada alimentador da rede, e por isso, a ordem em que eles são processados tem uma influência direta no resultado. Por exemplo, suponha um caso em que o primeiro alimentador a ser otimizado é pequeno e pouco desequilibrado em relação aos outros, possuindo uma potência reativa baixa. O algoritmo memético inicialmente coloca os capacitores que julga necessários para reduzir as perdas reativas nesse alimentador, sem saber como é a situação real dos outros. Se o orçamento é muito baixo, corre-se o risco de boa parte dele ser gasta já nesse alimentador inicial. Assim, outros alimentadores maiores ou com fatores de potência piores, que dariam maiores retornos em termos de redução de perdas, acabam recebendo menos capacitores do que seria desejado. A busca local é um procedimento 'guloso', e por isso, escolhas iniciais erradas têm sua influência propagada em todo o restante do processo.

Este efeito foi parcialmente contornado com a estratégia de ordenar os alimentadores a serem otimizados pelo resultado da razão da potência reativa pela potência ativa. Esse valor fornece uma boa noção de quão desequilibrado o alimentador está e permite que a otimização seja feita do mais para o menos desequilibrado. Apesar de obtermos uma melhora considerável, notamos que o problema continuava no fato de se gastar até o limite do orçamento, qualquer que fosse o alimentador em questão. É razoável imaginar que os primeiros capacitores colocados em um dado alimentador irão gerar uma redução muito maior nas perdas do que os últimos. Assim, existe um ponto em que em vez de se colocar mais capacitores, torna-se mais conveniente pular para o alimentador seguinte.

Para implementar essa política, uma possibilidade seria dividir o orçamento, liberando-o pouco a pouco. A estratégia adotada divide inicialmente o orçamento em partes iguais (neste trabalho adotamos uma divisão em duas vezes o número de alimentadores). Ele é então liberado parte a parte, sempre para o alimentador mais desequilibrado no momento.

Outro problema surgiu ao examinarmos o método de alocação dos capacitores. Originalmente, as posições estavam sendo testadas seguindo a sequência da rede elétrica real, onde as mais próximas às subestações são as primeiras. Isso criou uma concentração excessivamente grande de capacitores nessa parte da rede. Apesar da colocação de capacitores próximo às subestações não ser algo de todo ruim, essa tendência, por ser exagerada, estava prejudicando os resultados. Assim, passamos a ordenar os testes das seções do alimentador não mais pela sequência da rede real, mas sim de forma aleatória, o que gerou uma melhor distribuição dos capacitores ao longo da rede como um todo. Além disso, cada vez que a busca local é executada, a sequência aleatória muda, o que elimina qualquer tendência prejudicial.

Sobre a aplicação da busca local, já no início dos testes ficou claro que não poderíamos aplicá-la a todos os novos indivíduos gerados. O tempo computacional seria demasiadamente alto. Tentamos então aplicar a todos os indivíduos da população depois de sua convergência. No entanto, tal procedi-

mento também se mostrou muito custoso, sendo necessária mais de uma hora de tempo de CPU - no caso da instância maior - para que o método convergisse. A solução então foi aplicar a busca local somente ao melhor indivíduo da população, sempre após a convergência da mesma. Com esta opção, o tempo computacional caiu para níveis mais aceitáveis - alguns poucos minutos - e a qualidade das soluções se manteve em um nível elevado.

4.8 Função de *fitness*

A função de *fitness* tem por finalidade avaliar a qualidade dos indivíduos gerados. Para tanto ela deve guardar uma relação estreita com a função objetivo do problema em questão. Mantendo a tradição dos algoritmos evolutivos que manda valorizar o indivíduo com maior valor de *fitness*, ou adaptabilidade, uma escolha adequada para essa função no caso do PLC precisa considerar diversos fatores. O primeiro é o custo das perdas na rede. Para o cálculo das perdas faz-se necessário rodar um algoritmo de fluxo de carga. Elas são comparadas com as perdas antes da colocação dos capacitores, e o ganho energético obtido é computado na forma de lucro anualizado. Para obter o ganho em reais a partir da redução das perdas em kW utiliza-se a Equação 4.1.

$$Ganho_{reais} = \underbrace{Custo_{MWh}}_{R\$}.8,75.\underbrace{Reducao_{perdas}}_{kW} \tag{4.1}$$

Onde $Custo_{MWh}$ é o custo em reais do MWh no mercado. O valor 8,75 é uma constante que transforma kW em MWh anualizado. Essa constante traduz o número de horas em um ano, dividido por 1.000, pois a redução das perdas é dada em kW e o custo da energia em MWh.

Do ganho energético obtido pela instalação dos capacitores deve ser abatido o custo de compra e instalação dos mesmos. Essa parcela é composta por um somatório dos custos de todos os capacitores a serem instalados.

O custo dos capacitores é anualizado levando-se em conta um prazo de amortização do equipamento, em conjunto com a taxa de juros anual (fixa). Para tanto, utiliza-se a fórmula que, dado um valor presente, calcula o valor da prestação anual para amortizar aquele valor, dado um horizonte de n anos e uma taxa de juros i. Assim, o custo anual dos capacitores é calculado pela Equação 4.2.

$$CustoCap_{anual} = \frac{i.Custo_{cap}}{1 - \frac{1}{(1+i)^n}} \tag{4.2}$$

Adotamos ainda uma opção de escolha do montante a ser investido anualmente na instalação dos capacitores, uma vez que em situações reais há restrições de orçamento. Essa restrição é controlada na função objetivo através da Equação 4.3.

$$Penaliz_{orcam} = \Big(\max\big[0, CustoCap_{anual} - Orcam_{anual}\big]\Big)^2 \tag{4.3}$$

Onde $Orcam_{anual}$ é o gasto máximo anual permitido na instalação dos capacitores. A penalização quadrática apresentou um comportamento excelente, gerando indivíduos factíveis já nas primeiras gerações do algoritmo memético, porém outros tipos de penalização também são possíveis.

Por fim, há ainda a opção de limitar o número de bancos de capacitores a serem instalados. Essa restrição é operacional e guarda estreita relação com a capacidade da equipe de manutenção da companhia. Essa restrição é controlada na função objetivo através da Equação 4.4.

$$Penaliz_{cap} = \Big(\max\left[0, Num_{cap} - NumMax_{cap}\right]\Big)^2 \tag{4.4}$$

Onde $NumMax_{cap}$ é o número máximo permitido de bancos de capacitores e Num_{cap} é o número de capacitores presentes na solução em questão. O *fitness* do indivíduo é então determinado pela Equação 4.5. Ela traz o ganho anual resultante das perdas elétricas, subtraído do custo também anual da instalação dos capacitores e das duas penalizações. Quando a solução é factível, os dois últimos termos valem zero, e a qualidade da solução é representada pelo lucro líquido obtido com a instalação dos capacitores.

$$fitness = Ganho_{reais} - CustoCap_{anual} - Penaliz_{orcam} - Penaliz_{cap} \tag{4.5}$$

4.9 Testes computacionais

Nesta seção apresentamos os resultados computacionais obtidos com o uso do algoritmo memético proposto. Inicialmente testamos o método em duas redes pequenas - compostas por 9 e 135 seções - apresentadas no trabalho de Gallego *et al.* [24]. Nesse trabalho, os autores utilizam uma *busca tabu* para encontrar a melhor configuração de localização e tamanho dos capacitores. Os resultados são apresentados na Tabela 4.2.

Instância	**Gallego et al. (2001)**	**Algoritmo Memético**
Rede de 9 seções	308.909	307.158
Rede de 135 seções	192.339	190.446

Tabela 4.2: *Resultados das redes pequenas de Gallego et al. (2001).*

Os valores representam o custo das perdas anuais somado ao custo dos capacitores (dados em US$). Assim quanto menor o valor, melhor é o resultado. Este teste era necessário para comparar o algoritmo memético com o melhor método anterior disponível na literatura. A função objetivo utilizada para essas duas instâncias é a mesma apresentada na referência [24], e difere um pouco da que é descrita na seção 4.8. Os resultados indicam que o algoritmo memético melhorou os resultados anteriores para ambos os problemas. Apesar da diferença ser mínima, uma característica interessante é notada. O AM tende a colocar os capacitores maiores mais próximos às subestações, enquanto a busca tabu faz justamente o oposto. Talvez a diferença de desempenho mais

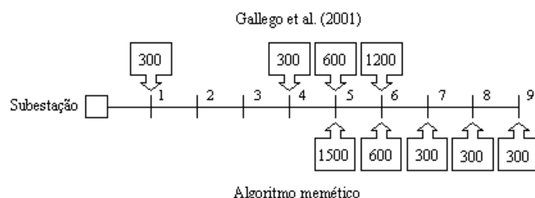


Figura 4.3: *Soluções para a instância de 9 seções.*

clara esteja refletida nos tempos computacionais. O AM é bem mais rápido que o método utilizado por Gallego *et al.* [24]. Para as instâncias de 9 e de 135 seções, o método de Gallego necessitou de 60 e 300 segundos, ao passo que o algoritmo memético utilizou apenas 2 e 5 segundos, respectivamente. Isso representa um decréscimo considerável no esforço computacional. Na Figura 4.3 mostramos os tamanhos e as localizações dos capacitores sugeridos por ambos os métodos.

O próximo passo lógico foi aumentar a dimensão do problema. Para verificar o potencial do AM em problemas grandes, obtidos a partir de cenários reais, utilizamos uma rede de distribuição correspondente a uma cidade brasileira de médio porte da área de concessão da *Companhia Paulista de Força e Luz S.A.* (CPFL). As características da rede são descritas na Tabela 4.3.

Número de nós (seções)	2.274
Número de alimentadores	3
Perdas iniciais	663 kW
Carga total	59.433 kW

Tabela 4.3: *Dados da rede de distribuição.*

Os testes para o problema de localização de capacitores levam em conta um conjunto de cenários possíveis, onde variamos os parâmetros que influenciam diretamente o comportamento do método. Esses testes visam verificar como o algoritmo memético se adapta às diferentes situações que podem ser encontradas em uma aplicação real. Os parâmetros incluídos na análise de sensibilidade são: preço do MWh, orçamento máximo e prazo de amortização do investimento. As tabelas de resultados apresentam os seguintes dados:

- **Capacidade Total**: Soma das capacidades (em kVar) dos capacitores instalados. Entre parênteses é apresentada a quantidade de capacitores.
- **Perdas em kW**: É a perda técnica após a instalação dos capacitores. Deve-se comparar esse valor com as perdas iniciais, na Tabela 4.3.
- **Lucro líquido anual**: É o lucro final obtido pela economia de energia, já descontado o custo dos capacitores.

Os tempos computacionais foram de cerca de três minutos em todos os testes. Esses tempos não variam muito pois estão diretamente relacionados

com o tamanho da instância, sendo pouco influenciados pelos parâmetros. O algoritmo foi implementado em C++ e utilizamos um computador Pentium II Celeron de 366 MHz.

4.9.1 Análise da variação do preço do MWh.

Os parâmetros foram ajustados da seguinte forma. Preço do MWh variando entre R$ 50 e R$ 300; número máximo de capacitores ilimitado; orçamento máximo ilimitado; prazo de amortização de 5 anos com juros de 12% ao ano.

Preço do MWh em R$	**Capacidade Total (num. de capacit.)**	**Perdas em kW**	**Lucro líquido anual em R$**
50	6.450 (15)	585	17.033
100	10.800 (21)	564	62.056
200	14.850 (41)	553	148.041
300	17.100 (46)	550	246.551

Tabela 4.4: *Análise da variação do preço do MWh.*

A Tabela 4.4 mostra que à medida em que o preço do MWh aumenta, se torna mais vantajoso colocar capacitores na rede. Neste caso, o custo dos mesmos é facilmente compensado pela economia resultante da redução das perdas. Ao preço de R$ 300/MWh, é sugerida a instalação de 46 capacitores, uma quantidade considerável, mesmo para uma cidade de médio porte. A inclusão no algoritmo da restrição que limita a quantidade de capacitores instalados evita esse tipo de resultado. Soluções com um número muito elevado de capacitores são operacionalmente impraticáveis, pois necessitam de manutenção em larga escala. As perdas elétricas se reduzem continuamente com o aumento no número de capacitores; fato esse que, em conjunto com o alto custo da energia que é economizada, faz com que o lucro líquido anual se multiplique rapidamente.

4.9.2 Análise da variação do orçamento máximo.

Os parâmetros foram ajustados da seguinte forma. Preço do MWh em R$ 100; número máximo de capacitores ilimitado; orçamento máximo variando entre R$ 5.000 e R$ 30.000; prazo de amortização de 5 anos com juros de 12% ao ano.

A Tabela 4.5 mostra que o algoritmo se adapta facilmente à variação do orçamento disponível. Se há mais capital à disposição, o método sugere a compra de mais capacitores, visando o aumento do lucro líquido. Na Tabela 4.4, vimos que o lucro líquido era proporcional ao custo da energia. Na Tabela 4.5, nota-se que, com o preço de R$ 100/MWh, o lucro líquido varia entre 2 e 5 vezes o capital investido, sendo que essa proporção varia seguindo a oscilação do preço do MWh. Esses resultados justificam o investimento, pois tal nível de retorno do investimento pode ser classificado como sendo muito bom. Outro fato verificado é que a influência da instalação dos capacitores vai diminuindo

Orçamento R$	**Capacidade Total (num. de capacit.)**	**Perdas em kW**	**Lucro líquido anual em R$**
5.000	2.100 (4)	627	26.670
10.000	4.800 (8)	603	42.542
20.000	7.200 (18)	577	54.926
30.000	10.800 (21)	564	62.056

Tabela 4.5: *Análise da variação do orçamento anual máximo.*

à medida em que a rede se torna mais equilibrada. Tome por exemplo o caso quando o orçamento é de R$ 5.000, e o retorno é de cinco vezes o capital investido. Isso ocorre porque o ganho elétrico proporcionado pela instalação de apenas quatro capacitores é alto, dado que a rede inicialmente está bastante desequilibrada. O aumento do número de capacitores passa então a ter um impacto cada vez menor na redução das perdas, até que finalmente, quando há 21 capacitores instalados, o retorno sobre o capital investido é reduzido para pouco mais de duas vezes.

4.9.3 Análise da variação do prazo de amortização

Os parâmetros foram ajustados da seguinte forma. Preço do MWh em R$ 100; número máximo de capacitores ilimitado; orçamento máximo ilimitado e prazo de amortização variando entre 1 e 10 anos com juros de 12% ao ano.

Prazo de amortização em anos	**Capacidade Total (num. de capacit.)**	**Perdas em kW**	**Lucro líquido anual em R$**
1	3.450 (7)	603	20.362
3	9.000 (15)	569	53.109
5	10.800 (21)	564	62.056
10	13.800 (38)	555	68.396

Tabela 4.6: *Análise da variação do prazo de amortização.*

O prazo de amortização do investimento e a taxa de juros influenciam diretamente o custo anualizado dos capacitores. Na Tabela 4.6 é possível ver como esse custo é reduzido quando o prazo de amortização é maior - a alta taxa de juros colabora para esse comportamento. Com um custo anualizado menor, o algoritmo sugere, nesse caso, a instalação de uma quantidade maior de equipamentos. Para prazos de amortização menores, menos capacitores são instalados. De um modo geral, o prazo de amortização deve guardar uma relação estreita com o tempo necessário para a substituição dos equipamentos. A instalação de um banco de capacitores pode ser encarada como um investimento cujo custo será diluído ao longo da sua vida útil, pois somente quando o equipamento necessitar de substituição é que teremos de fazer um novo investimento. O prazo de amortização de 5 anos utilizado nas Tabelas 4.4 e 4.5 pode

ser considerado um valor bastante conservador, tendo em vista que o tempo de vida desse tipo de equipamento é em geral bem maior. Assim, acreditamos que seja plenamente justificável o uso de prazos ainda maiores, o que geraria um crescimento dos lucros líquidos obtidos.

4.10 Resumo

Neste capítulo tratamos do Problema de Localização de Capacitores (PLC) em um rede de distribuição de energia. Para tanto, utilizamos os recursos normais disponíveis no NP-Opt aliados às estratégias de recombinação e de busca local especialmente desenvolvidas para o problema.

Inicialmente, optamos por uma representação que divide o cromossomo em duas partes: uma relacionada à localização dos capacitores e outra ao tamanho dos mesmos. O crossover utilizado atua nessas duas partes separadamente e apresenta um alto grau de aleatoriedade na sua dinâmica.

A busca local é resultado da soma de três vizinhanças e age tanto na localização quanto no tamanho dos capacitores. Tendo em vista os fracos resultados iniciais, fomos pouco a pouco descobrindo e eliminando os pontos ruins. O resultado foi uma estratégia de alocação diferenciada, onde os capacitores vão sendo alocados pouco a pouco, sempre nos alimentadores mais desbalanceados no momento. Essa abordagem obteve uma taxa de sucesso bastante alta. Os acertos e erros cometidos ao longo do processo de desenvolvimento da busca local são descritos em detalhes, bem como suas possíveis razões.

Os testes computacionais foram realizados utilizando dois conjuntos de dados. O primeiro é composto por duas instâncias de pequeno porte, utilizadas no trabalho anterior de Gallego *et al.* [24]. O segundo é composto por uma rede real, referente a uma cidade de médio porte do Estado de São Paulo composta por três alimentadores e mais de 2000 nós. Efetuamos ainda uma análise de sensibilidade referente a variações do preço do MWh, do orçamento máximo disponível e do prazo de amortização do investimento.

Os resultados mostraram que o método se adaptou bem ao problema, conseguindo efetuar a alocação dos capacitores de forma equilibrada e respeitando as várias restrições do problema. O lucro gerado pela redução das perdas elétricas é bastante alto e facilmente compensa o gasto de investimento. Ressaltamos que provavelmente este é a primeira vez que um problema de grande porte - composto por milhares de nós - é abordado com sucesso. Trabalhos prévios se restringem a problemas de pequeno porte, compostos por dezenas, ou no máximo algumas centenas de nós.

Esta página foi deixada intencionalmente em branco.

Capítulo 5

Gate Matrix Layout

5.1 Introdução

[1] O problema de Gate Matrix Layout pertence à área de desenho de circuitos, e é também conhecido pelo nome de *VLSI Design*. O objetivo é organizar um conjunto de portas, ou *gates*, de forma que a área total do circuito seja minimizada. Este desenho de circuito foi desenvolvido nos Laboratórios Bell por Lopez e Law em 1980 [52]. A abordagem proposta traz bons resultados para circuitos transistorizados de larga escala que seguem a tecnologia CMOS (*Complementary Metal-Oxide Semiconductor*), onde a regularidade da estrutura e a alta densidade dos elementos são as características principais. A estrutura de Gate Matrix consiste de linhas e colunas que se cruzam em certos pontos, determinando assim a localização dos transistores e suas interconexões. Cada linha do *layout* é chamada de trilha, ao passo que cada coluna representa uma porta específica. O objetivo é obter uma disposição das colunas que resulte na utilização da menor quantidade de recursos, que na implementação são representados pelo número de trilhas utilizadas. Como o foco deste trabalho é a parte algorítmica, esclarecemos que o leitor pode obter maiores informações sobre essa arquitetura nas referências [38, 56, 61, 74].

São duas as principais contribuições deste trabalho. A primeira reside na criação de uma busca local extremamente eficiente e que utiliza uma redução de vizinhança especial. A outra contribuição é uma análise comparativa entre as abordagens multipopulacional e unipopulacional. A melhora de desempenho em relação aos trabalhos anteriormente disponíveis foi considerável. Este problema é parte constituinte do Np-Opt e todos os resultados podem ser reproduzidos diretamente. As instâncias utilizadas também estão disponíveis e

[1] Este capítulo é baseado nos artigos:

A. Mendes, P. França, P. Moscato e V. Garcia. **Population Studies for the Gate Matrix Layout Problem**. Proceedings do *IBERAMIA2002* - 8th *IberoAmerican Conference on Artificial Intelligence*, *Lecture Notes in Artificial Intelligence*, 2527:319-328, Springer-Verlag, 2002.

A. Mendes e A. Linhares. **Gate Matrix Layout: a multiple population evolutionary approach**. Submetido ao *International Journal of Systems Science* em Julho de 2003.

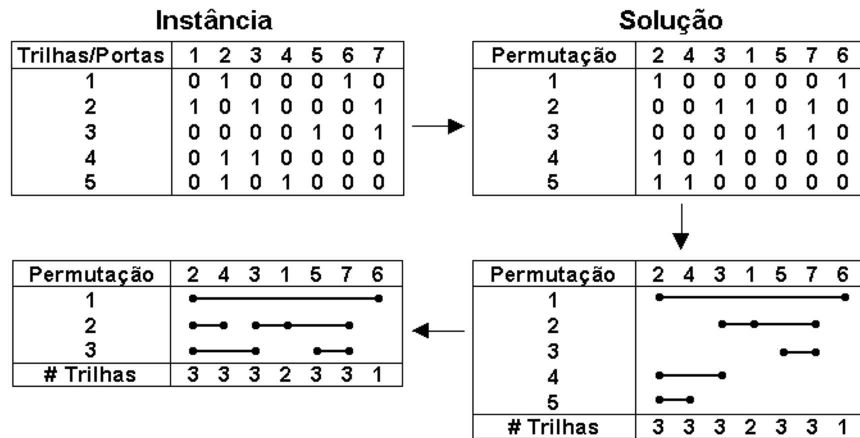
acompanham o software.

5.2 Descrição do problema

O problema de Gate Matrix Layout (GML) pertence à classe *NP-hard* [46, 50, 59]. Ele aparece na fase de desenho de sistemas VLSI (*Very Large Scale Integration*). Suponha que existam p portas (fios verticais) e t trilhas (fios horizontais) em um circuito tipo Gate Matrix. Uma vez mais, as portas podem ser descritas como fios verticais com transistores posicionados em locais específicos. A arquitetura do circuito exige que haja trilhas interconectando horizontalmente todas as portas distintas que possuem transistores nas mesmas posições.

Uma instância pode ser representada como uma matriz 0-1, denotada por $\{M_{t,p}\}$, com t linhas e p colunas. Um valor 1 na posição $(i,\ j)$ da matriz M significa que um transistor deve ser implantado na i-ésima trilha e na j-ésima porta. Um valor 0 indica que a conexão não deve ser feita. Lembramos que a arquitetura exige que todos os transistores pertencentes a uma mesma trilha estejam conectados.

Dada um sequência específica de portas, toda vez que duas trilhas se sobrepõem, a sua implementação exige duas trilhas fisicamente separadas. A superposição de todas as trilhas define a quantidade mínima necessária para construir o circuito. O objetivo passa a ser encontrar a permutação das portas cuja superposição de interconexões seja mínima, minimizando assim o número de trilhas e a área total do circuito. A Figura 5.1 mostra um exemplo e como criar um circuito a partir da representação matricial de uma solução.



Instância

Trilhas/Portas	1	2	3	4	5	6	7
1	0	1	0	0	0	1	0
2	1	0	1	0	0	0	1
3	0	0	0	0	1	0	1
4	0	1	1	0	0	0	0
5	0	1	0	1	0	0	0

Solução

Permutação	2	4	3	1	5	7	6
1	1	0	0	0	0	0	1
2	0	0	1	1	0	1	0
3	0	0	0	0	1	1	0
4	1	0	1	0	0	0	0
5	1	1	0	0	0	0	0

Permutação	2	4	3	1	5	7	6
1							
2							
3							
4							
5							
# Trilhas	3	3	3	2	3	3	1

Permutação	2	4	3	1	5	7	6
1							
2							
3							
# Trilhas	3	3	3	2	3	3	1

Figura 5.1: *Instância do problema de Gate Matrix Layout.*

No exemplo da Figura 5.1, a permutação das portas é [2-4-3-1-5-7-6]. Depois de definidas as interconexões de todos os transistores, representadas pelas linhas horizontais, calculamos o número de trilhas necessário para construir cada porta. Este número é dado pela soma das posições utilizadas em cada coluna, e o número de trilhas necessário para construir o circuito é o valor máximo.

No exemplo da Figura 5.1, esse número é três. Na parte inferior esquerda do diagrama, é mostrado o circuito resultante após o grupamento das conexões, e o uso de apenas três trilhas, ao invés das seis originais. Mais informações sobre esse problema, incluindo outros ambientes industriais onde ele aparece podem ser encontrados nas referências [49, 51, 75]. A título de curiosidade, ressaltamos que este não é um problema *NP-hard* 'comum'. Ele foi de fato o primeiro problema identificado como sendo tratável a parâmetros fixos, e esse resultado levou à criação de uma nova classe de problemas, sob o rótulo de *FTP-tractable* [17, 21].

5.3 Representação, população inicial, recombinação e mutação

A representação do problema é bastante intuitiva. Como o objetivo é encontrar uma permutação das p portas que minimize o número de trilhas utilizado, optamos por utilizar um cromossomo cujos alelos são valores inteiros distintos no intervalo $[1, p]$.

A população inicial é puramente aleatória, e não utiliza nenhuma heurística especial para a geração dos indivíduos. De fato, o algoritmo memético não teve dificuldade em encontrar soluções de boa qualidade, partindo de populações aleatórias em um tempo computacional bastante reduzido.

O operador de recombinação utilizado é uma variante do *Order Crossover* (OX) [29], chamada BOX. Ele é muito semelhante à 2ª variante do *crossover* OX, apresentada no trabalho de Syswerda [69]. O operador OX é muito simples e fácil de ser implementado. Ele começa copiando uma parte do cromossomo de um dos pais para o filho. A escolha dessa parte é feita de forma aleatória. Dois pontos do cromossomo são escolhidos e toda a informação contida entre eles é copiada. Esta pode ser uma limitação importante no caso das boas características presentes no indivíduo não estarem agrupadas, mas sim separadas em diversos blocos. Neste caso, uma recombinação mais ajustada ao problema deveria ser capaz de copiar várias partes do cromossomo do pai para o filho. Esta é a motivação do BOX, onde, em vez de um único pedaço, várias partes do cromossomo são copiadas.

O restante do operador BOX é idêntico ao OX. As posições do filho que permaneceram vazias recebem os alelos do outro pai, sendo sequencialmente preenchidas a partir da extremidade esquerda do cromossomo. Ambos os operadores - OX e BOX - tendem a perpetuar a ordem relativa das portas, mas o BOX consegue transmitir as configurações dos conjuntos de portas de maneira mais eficiente para os filhos.

Na Figura 5.2 é mostrado o funcionamento do BOX através de um exemplo simples. Nele, o *Pai A* contribui com duas partes do cromossomo para o filho; uma composta por apenas um alelo e a outra por três. Essas partes são copiadas no filho nas mesmas posições que ocupam no pai. Os espaços vazios são então preenchidos com a informação proveniente do *Pai B*, no sentido esquerda-direita. Os valores do *Pai B* que já estão presentes no filho são desconsiderados, sendo copiados apenas os novos. A contribuição de cada pai

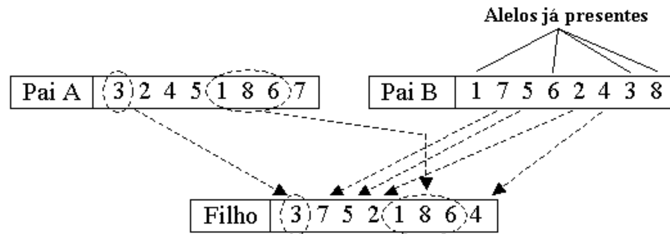


Figura 5.2: *Block Order Crossover (BOX).*

para o filho foi ajustada para ser igual, o que faz com que cada pai transmita aproximadamente 50% de seus alelos. Como em todos os testes foram utilizadas populações estruturadas, optou-se por designar como *Pai A* sempre o líder do subgrupo, cabendo ao indivíduo seguidor o papel do *Pai B*.

A mutação utilizada se baseia na troca de posição das portas. Duas posições do cromossomo são selecionadas de maneira uniformemente aleatória e seus valores são trocados. Este procedimento é aplicado a 10% de todos os novos indivíduos criados.

5.4 Busca local

A busca local adotada é composta na realidade por duas estratégias distintas. A primeira, denominada *all-pairs* efetua uma troca de posição entre colunas. A busca local de inserção, por sua vez, retira a coluna de sua posição e a insere entre outras duas. Para instâncias pequenas, é possível efetuar todas as trocas e inserções possíveis, o que resulta em uma complexidade total de $O(p^2)$. No entanto como aqui tratamos de problemas considerados grandes (com mais de 100 portas), foi necessário reduzir essa complexidade, sob a pena do tempo computacional se tornar proibitivo. Assim, implementamos inicialmente uma estratégia de redução de vizinhança bem simples, onde apenas as k-ésimas posições mais próximas eram testadas para movimentos de troca ou inserção de colunas. Por exemplo, se $k = 20$, então cada porta será testada para troca e inserção com seus 10 vizinhos mais próximos à direita e à esquerda.

No entanto, como as instâncias tratadas possuíam até 141 portas, a complexidade ainda permanecia muito alta e a redução dos valores de k já havia atingido o seu limite prático, antes de começar a ter um impacto negativo forte no desempenho do algoritmo. Os valores mínimos de k necessários para que se chegasse às soluções presumidamente ótimas ainda eram muito grandes e tornavam o processo de busca muito lento. Desta forma, uma nova estratégia de redução de vizinhança foi criada. Essa redução descarta boa parte das tentativas de trocas e inserções pois é capaz de verificar de antemão quais delas terão influência ou não sobre o *fitness* do indivíduo.

A segunda política de redução de vizinhança se baseia na localização das chamadas 'portas críticas'. Essas portas são as que definem o número de trilhas da solução; em outras palavras, elas definem o *fitness* do indivíduo. A Figura

5.3 mostra um exemplo que identifica tais colunas.

Permutação	3	6	5	2	1	8	7	4	9
1	•				•	•			
2						•			•
3	•		•	•					
4				•	•	•			
5								•	•
6			•	•					
7		•			•				
8				•			•		•
# Trilhas	2	3	4	6	4	4	2	3	3

Figura 5.3: *Identificação de uma porta crítica no problema GML.*

Na Figura 5.3, a porta crítica é a de número 2, que requer 6 trilhas para ser implementada. A redução de vizinhança age proibindo qualquer tentativa de troca ou inserção que não possa afetar o valor dessa porta crítica. No exemplo, movimentos envolvendo quaisquer pares de portas do conjunto [1, 8, 7, 4, 9] não são testados, pois eles não podem reduzir o valor do número de trilhas requerido pela porta 2. Analogamente, movimentos compostos por pares de portas do conjunto [3, 6, 5] também são descartados de antemão.

Se há mais de uma porta crítica, os movimentos proibidos são aqueles entre portas pertencentes à região mais à esquerda da primeira porta crítica, e entre as portas situadas à direita da última porta crítica. Como a complexidade computacional tanto da vizinhança de troca quanto da de inserção é $O(p^2)$, a redução se torna mais relevante nas instâncias maiores.

A motivação dessa estratégia é que qualquer modificação da posição das portas que não afete as colunas críticas poderá gerar apenas melhorias locais - isto é, nas colunas situadas entre as portas envolvidas no movimento - e não melhorias globais. Como o critério utilizado para descrever o *fitness* é o valor máximo do número de trilhas necessárias, melhorias locais não causam nenhum impacto na determinação do grau de adaptabilidade do indivíduo. Essa segunda estratégia de redução melhorou consideravelmente o desempenho do algoritmo memético, fazendo-o atingir soluções de alta qualidade com um número bem menor de avaliações de indivíduos. A redução da complexidade da busca local ocasionada pelo uso de informações das portas críticas permitiu o aumento do número k de posições vizinhas testadas, o que logicamente gerou um aumento proporcional no poder de busca do método. O resultado foi uma melhora geral no desempenho do algoritmo memético, em termos tanto de tempo computacional quanto de número de indivíduos testados.

O modo como a busca local é aplicada também foi objeto de estudo. A aplicação em todos os indivíduos novos é proibitiva do ponto de vista computacional. Testes considerando a aplicação da busca local somente após cada geração também produziram resultados decepcionantes. Para este problema, a melhor opção foi aplicá-la somente ao melhor indivíduo da população, após a sua convergência. Essa política altamente restritiva reflete a alta complexidade

da busca local mesmo com o uso das duas reduções de vizinhança. Em geral, quando a busca local é muito custosa computacionalmente, opta-se por aplicá-la em uma parcela pequena da população para que haja um equilíbrio maior entre o tempo total gasto nessa etapa e no restante do algoritmo memético. Caso contrário, corre-se o risco de deixar pouco tempo disponível para os processos de recombinação, mutação, e mesmo para o processo evolutivo que naturalmente ocorre ao longo de sucessivas gerações, o que pode prejudicar de forma contundente o desempenho geral do algoritmo.

5.5 Testes computacionais

5.5.1 Abordagem unipopulacional x multipopulacional

Os testes comparativos entre as abordagens avaliam a influência de dois parâmetros no desempenho do algoritmo memético multipopulacional. O primeiro é o número de populações, cujo valor variou de dois a cinco. O segundo parâmetro é a política de migração adotada. Como existem três políticas de migração disponíveis - *0-Migrate*, *1-Migrate* e *2-Migrate* - temos uma combinação de 12 testes possíveis. Somando-se a este número o teste do algoritmo unipopulacional, totalizamos 13 configurações para cada instância. Os testes foram realizados com cinco instâncias cujos tamanhos variam de 33 a 141 portas, onde cada configuração foi executada 10 vezes. O número total de rodadas, dessa forma, atingiu o valor de 650. Neste problema utilizamos o algoritmo memético sequencial. O recurso da distribuição das buscas locais através de uma rede de computadores ainda não estava disponível no software NP-Opt.

O critério de parada foi o tempo de CPU, que variou de acordo com o tamanho do problema tratado e o tempo necessário para encontrar soluções de boa qualidade. Este tempo foi fixado de forma que fosse suficientemente longo para que boas soluções começassem a aparecer. Da mesma forma, esse tempo não poderia ser muito longo, pois assim a maior parte das configurações retornaria resultados semelhantes, enfraquecendo quaisquer comparações. Na Tabela 5.1 são apresentadas informações sobre as instâncias utilizadas, bem como o tempo de CPU adotado. O conjunto de testes é composto por um problema pequeno, três médios e um grande. No trabalho de Linhares *et al.* [50] encontramos os testes computacionais mais extensos para o problema de GML, com 25 problemas no total, mas há poucos de tamanho razoável. A maioria deles é muito pequena - possui menos de 30 portas - sendo muito fácil para o algoritmo memético atingir a solução ótima ou a melhor conhecida nesses casos. Desta forma, optamos por nos concentrar nos problemas maiores - V4470, X0, W2, W3 e W4; todos com mais de 30 portas.

A seguir, são mostrados os resultados experimentais. Para cada configuração, quatro números são utilizados para caracterizar o desempenho do algoritmo memético. Na Figura 5.4 vemos, à esquerda no alto e em negrito, o melhor resultado obtido para o número de trilhas nas 10 tentativas. Seguindo em sentido horário temos o número de vezes que essa melhor solução foi encontrada; o pior valor encontrado; e finalmente, na parte inferior esquerda, o valor médio para o número de trilhas.

Instância	Núm. de portas	Núm. de trilhas	Melhor solução	Tempo de CPU (seg.)	Parâm. k
W2	33	48	14	10	10
V4470	47	37	9	30	20
X0	48	40	11	30	20
W3	70	84	18	90	30
W4	141	202	27	2400	60

Tabela 5.1: *Dados das instâncias de GML.*

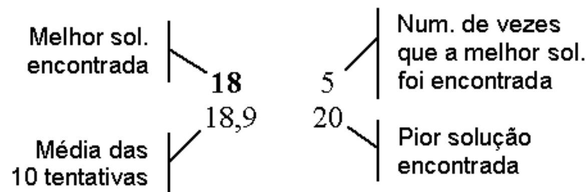


Figura 5.4: *Dados de saída do GML.*

Os testes foram realizados em um computador Pentium II Celeron de 366 MHz. O algoritmo memético atingiu o ótimo da instância W2 em todas as 10 tentativas para cada uma das configurações de política de migração e número de populações. Por esse motivo, resolvemos omitir a respectiva tabela de resultados. Ressaltamos porém, que a melhor configuração foi a da estratégia *1-Migrate* com o uso de quatro populações, pois ela necessitou da menor quantidade de avaliações de indivíduos no cômputo geral. As instâncias maiores mostraram maiores diferenças no desempenho e por isso os resultados estão explicitados nas Tabelas 5.2 a 5.5.

Como os testes levam em conta dois parâmetros - política de migração e número de populações - vamos avaliá-los separadamente, começando pela política de migração. A estratégia *0-Migrate* equivale ao algoritmo sem nenhuma migração. Quando utilizamos essa estratégia com n populações, na realidade estamos criando o mesmo efeito de executar n vezes o algoritmo memético

V4470	**Número de populações**									
	1		**2**		**3**		**4**		**5**	
0-Migrate	**9**	2	**9**	1	**9**	4	**9**	1	**9**	5
	10,1	11	10,1	11	9,6	10	10,0	11	9,5	10
1-Migrate			**9**	2	**9**	6	**9**	5	**9**	4
			9,9	11	9,5	11	9,6	11	9,6	10
2-Migrate			**9**	5	**9**	4	**9**	2	**9**	5
			9,5	10	9,7	11	9,8	10	9,5	10

Tabela 5.2: *Resultados da instância V4470.*

X0	Número de populações									
	1		2		3		4		5	
0-Migrate	**11**	6	**11**	9	**11**	8	**11**	9	**11**	9
	11,6	13	11,1	12	11,2	12	11,1	12	11,1	12
1-Migrate			**11**	10	**11**	10	**11**	10	**11**	9
			11,0	11	11,0	11	11,0	11	11,1	12
2-Migrate			**11**	8	**11**	9	**11**	8	**11**	9
			11,2	12	11,1	12	11,2	12	11,1	12

Tabela 5.3: *Resultados da instância X0.*

W3	Número de populações									
	1		2		3		4		5	
0-Migrate	**18**	3	**18**	3	**18**	5	**18**	7	**18**	5
	20,0	23	20,0	23	19,1	20	18,4	20	18,8	20
1-Migrate			**18**	3	**18**	5	**18**	6	**18**	9
			20,0	22	18,9	20	18,6	21	18,1	19
2-Migrate			**18**	4	**18**	7	**18**	6	**18**	4
			20,1	23	18,6	21	18,5	20	18,9	22

Tabela 5.4: *Resultados da instância W3.*

W4	Número de populações									
	1		2		3		4		5	
0-Migrate	**29**	4	**28**	1	**28**	1	**28**	2	**28**	2
	30,5	34	30,9	36	30,6	34	30,4	35	30,3	34
1-Migrate			**29**	2	**28**	2	**27**	2	**27**	1
			30,9	34	30,6	35	29,4	34	30,2	34
2-Migrate			**28**	1	**28**	4	**28**	2	**28**	3
			31,2	36	30,0	34	29,7	32	29,4	30

Tabela 5.5: *Resultados da instância W4.*

unipopulacional. O resultado é uma maior instabilidade nas respostas, com valores de pior solução e solução média um pouco mais fracos para as instâncias W3 e W4. Em especial, na instância W4, o valor de 27 trilhas é alcançado pouquíssimas vezes, e somente com o uso de múltiplas populações. Nos circuitos menores essa diferença é ainda menos clara, talvez pela maior facilidade do algoritmo em resolvê-las. É possível que a falta de migração sobrecarregue o caráter intensificador do algoritmo memético, em detrimento do caráter diversificador. Isso faz com que o algoritmo se torne muito dependente da região onde as populações se iniciam, pois ocorre pouca diversificação ao longo do processo de busca. Se o algoritmo tiver sorte e seus indivíduos iniciais forem gerados em uma região promissora do espaço de soluções, não há problema. No entanto, se eles estiverem em regiões ruins, é bem provável que o algoritmo

fique preso a essas regiões, criando uma instabilidade maior no conjunto dos resultados.

Por outro lado, a política *1-Migrate* mostrou uma melhor estabilidade, retornando bons valores médios e com melhores soluções sendo encontradas com uma boa frequência. Um equilíbrio maior entre intensidade e diversidade deve ter sido responsável por esse comportamento. A política *2-Migrate* não teve um desempenho tão bom, resultando em uma piora dos valores médios e da pior solução. Esse efeito pode ter sido causado já por uma sobrecarga da diversificação, em detrimento da intensificação. Desta forma, analisando à parte a política de migração, podemos dizer que a política de *1-Migrate* foi a de melhor desempenho geral.

O segundo aspecto a ser analisado é o número de populações. Apesar de não ficar claro qual configuração é a melhor, certamente o uso de apenas uma não é a melhor escolha pois diversas configurações multipopulacionais tiveram desempenhos superiores. À primeira vista, a conclusão a que se chega é que quando múltiplas populações são utilizadas, pelo menos três delas devem ser empregadas. Com apenas duas, o algoritmo parece não conseguir tirar proveito do efeito de *genetic drift*.

5.5.2 Eficiência do algoritmo memético

Uma característica verificada no algoritmo memético é a sua alta eficiência, traduzida pela pequena quantidade de avaliações de indivíduos, especialmente quando comparamos com o método anterior de melhor desempenho, descrito em Linhares *et al.* [50]. O método de *microcanonical optimization* apresentado naquele trabalho é uma variação mais eficiente de um *simulated annealing* tradicional. Esse método em questão foi capaz de melhorar os resultados - ou seja, obteve uma redução no número de trilhas - para cinco instâncias do total de 25 apresentadas, sendo que nas outras 20 conseguiu se igualar aos resultados anteriores. O algoritmo memético, por sua vez, conseguiu se igualar ao método de *microcanonical optimization* em todas as instâncias, porém utilizando muito menos recursos computacionais. Para ilustrar isso, nas Tabelas 5.6 e 5.7 comparamos o número de soluções avaliadas em cada método para as cinco instâncias tratadas aqui.

	Algoritmo memético		
Instância	**Mínimo**	**Médio**	**Máximo**
W2	3.125	3.523	5.398
V4470	32.509	176.631	451.377
X0	18.136	43.033	117.384
W3	79.089	203.892	495.306
W4	3.213.532	9.428.591	15.643.651

Tabela 5.6: *Número de avaliações do algoritmo memético.*

Os valores mostrados nas Tabelas 5.6 e 5.7 refletem a maior eficiência do algoritmo memético em relação à abordagem de *microcanonical optimization.*

	Microcanonical Optimization		
Instância	**Mínimo**	**Médio**	**Máximo**
W2	12.892	19.839	26.541
V4470	102.976	1.109.036	2.714.220
X0	52.126	95.253	187.335
W3	1.700.667	7.143.872	21.289.846
W4	24.192.291	167.986.282	405.324.093

Tabela 5.7: *Número de avaliações do microcanonical optimization.*

Devemos enfatizar que a redução, que varia entre 80% e 90%, foi impressionante e certamente superior à esperada. Após uma consulta com os autores do trabalho anterior, verificamos que o tempo de CPU deles para a instância W4, por exemplo, era da ordem de várias horas. Isto nos deu maior confiança nos resultados pois o algoritmo memético raramente ultrapassava a marca de uma hora de tempo de CPU até atingir uma solução com 27 trilhas. Levando-se em conta que os equipamentos utilizados tinham poder de processamento semelhante, a diferença na eficiência dos métodos ficou evidente.

Podemos citar ao menos três razões para a maior eficiência do algoritmo memético. Primeiramente, o uso da redução de vizinhança baseada na informação das portas críticas faz com que uma grande parte dos movimentos de troca e inserção de portas sejam rejeitados antes de serem avaliados. Isso se traduz em um ganho considerável de tempo, sem uma perda proporcional na capacidade de busca do algoritmo memético. Em segundo lugar, a restrição do número de portas vizinhas testadas, caracterizada pelo parâmetro k, ajuda a reduzir o número de possíveis testes. De fato, não convém testar trocas ou inserções de portas em posições distantes durante a busca local, pois esse papel quem deve cumprir é o operador de recombinação. A busca local em geral tem um papel mais de ajuste fino, deixando as mudanças estruturais de caráter mais amplo a cargo da recombinação. Por fim, a aplicação da busca local apenas aos melhores indivíduos concentra os esforços nos indivíduos mais promissores, aumentando as chances de se encontrar soluções de melhor qualidade em um tempo menor. O bom desempenho resultante do uso de uma política restritiva na aplicação da busca local é específico para o problema de GML. Ele foi altamente influenciado pela estrutura do problema, bem como pelas características da função de *fitness*. Enfatizamos que não se deve tentar estabelecer uma correspondência direta entre a complexidade pura e simples da busca local e o seu grau de aplicação nos indivíduos. Esse é um ponto bem mais complexo, sendo necessários sempre testes extensivos considerando várias configurações.

Está claro que o bom desempenho do algoritmo memético deve-se ao conjunto de suas características - busca local, operador de recombinação, mutação, migração, política de inserção de novos indivíduos, entre outros fatores. Mas certamente a busca local é um dos fatores que mais influenciaram o desempenho do algoritmo.

Ainda sobre as tabelas anteriores, essa comparação de esforço computacional lançou alguma luz sobre os resultados da instância V4470. Apesar de

Instância	**Parâmetro** k	**Avaliações por busca local**	**Avaliações por segundo de CPU**
W2	10	379	6.906
V4470	20	548	5.702
X0	20	585	5.557
W3	30	749	3.374
W4	60	3.242	662

Tabela 5.8: *Estatísticas do algoritmo memético.*

ela ser menor que a instância X0, o algoritmo memético teve dificuldades em encontrar a solução ótima. Essa dificuldade está refletida no número de avaliações feitas, bem maior que na instância X0. Inicialmente, pensamos que se tratava de um problema com o algoritmo em si, mas depois de comparar com os resultados do *microcanonical optimization*, concluímos que essa instância é intrinsicamente difícil, e sua estrutura merece um estudo mais aprofundado para esclarecer o porquê desse comportamento. Agora, para fornecer maiores informações a respeito da velocidade do algoritmo memético, apresentamos a Tabela 5.8, com algumas dados estatísticos do seu desempenho.

Os valores de k apresentados na Tabela 5.8 são os mesmos da Tabela 5.1 e que foram utilizados nos testes computacionais. Na coluna ao lado temos o número médio de avaliações por busca local. Esse valor leva em conta as duas reduções de vizinhança - a baseada no valor k e a que utiliza informação das portas críticas. A última coluna traz o número médio de avaliações feitas por segundo. Como cada avaliação necessita que toda a matriz 0-1 seja percorrida, a complexidade resultante é $O(t.p)$. Como há pouca margem para otimização desse cálculo, acreditamos que ele pode ser um critério bastante confiável para comparação de algoritmos distintos, executados em equipamentos também distintos.

5.6 Resumo

Neste capítulo tratamos do problema de Gate Matrix Layout, que pertence à área de desenho de circuitos. Aqui utilizamos a versão multipopulacional do algoritmo memético presente no NP-Opt. As principais contribuições desta abordagem se concentram no uso de múltiplas populações, no uso do operador de recombinação BOX e no desenvolvimento de uma busca local altamente eficiente para o problema.

As instâncias tratadas variam de 33 até 141 portas, e para todas elas igualamos os resultados do melhor método existente, porém com um uso bem menor dos recursos computacionais. A busca local foi a grande responsável por esse resultado. Ela se baseia em duas vizinhanças - troca e inserção - e na utilização de duas reduções de vizinhança. A primeira é uma redução bastante simples, que restringe os testes a posições localizadas a uma distância máxima uma da outra. A segunda redução, mais complexa, restringe os testes aos movi-

mentos que possam influenciar as chamadas *portas críticas*, em outras palavras, às portas que definem o *fitness* do indivíduo. Com isso, evita-se perder tempo ao testar movimentos que não possam melhorar a adaptabilidade do indivíduo.

Os resultados computacionais foram muito bons. O uso de múltiplas populações se justifica nas instâncias maiores. Na instância W4, por exemplo, o algoritmo memético só conseguiu se igualar ao método de *microcanonical optimization* em duas configurações, ambas multipopulacionais. Para instâncias menores, a necessidade do uso de múltiplas populações não é tão clara. Outro aspecto importante foi a forte redução do esforço computacional, de até 90% em comparação com o método anterior mais eficiente, o de *microcanonical optimization*.

Capítulo 6

Sequenciamento em Flowshop

6.1 Introdução

[1] Este capítulo trata do problema de Sequenciamento em *Flowshop*, cuja característica principal é a separação das tarefas em famílias. Esta é uma situação bastante comum uma vez que empresas sempre buscam tirar vantagem do que se convencionou chamar de *group technology* (GT) [66]. Neste problema, uma família é composta por várias tarefas que possuem necessidades semelhantes em termos de ferramentas, *setup* e sequências de operações. Normalmente, as famílias são designadas para uma determinada célula de manufatura com base nas sequências de operações de forma que o fluxo de materiais e o sequenciamento em si sejam simplificados. Este processo pode resultar em uma situação em que cada família seja processada por um conjunto de máquinas e todas as tarefas sejam processadas seguindo a mesma rota tecnológica.

Neste ambiente de produção, as células de manufatura lembram os tradicionais *flowshops*, a menos da existência de famílias de tarefas. Como as tarefas pertencentes à mesma família possuem necessidades semelhantes de recursos, pode-se deduzir que o tempo gasto em *setup* entre elas é mínimo, sendo nesse caso incluído no próprio tempo de processamento da tarefa. Por outro lado, um tempo de *setup* considerável deve ser necessário para mudar a produção de uma família para outra, e assim ele precisa ser tratado separadamente.

O *flowshop* com famílias de tarefas é um problema combinatorial pertencente à classe *NP-hard*. De fato, quando cada família é composta por uma

[1] Este capítulo é baseado nos artigos:

P. França, J. Gupta, A. Mendes, P. Moscato e K. Veltink. **Metaheuristic Approaches for the Pure Flowshop Manufacturing Cell Problem**. Proceedings do *PMS2000* - 7th *International Workshop On Project Management and Scheduling*, pág. 128-130, Osnabrück, Alemanha, Abril, 2000.

P. França, J. Gupta, A. Mendes, P. Moscato e K. Veltink. **Evolutionary Algorithms for Flowshop Scheduling with Family Setups**. Aceito para publicação na *Computers and Industrial Engineering*, 2003.

única tarefa, o problema se torna um *flowshop* tradicional com tempos de preparação dependentes da sequência. Problema este que já foi provado ser *NP-hard* quando o número de máquinas é maior que um [34].

Devido à natureza *NP-hard* do problema geral de *flowshop*, grande parte dos pesquisadores se concentrou no desenvolvimento de procedimentos heurísticos que fornecessem boas sequências permutacionais (onde a ordem de processamento das tarefas é a mesma em todas as máquinas) em um tempo computacional razoável. Porém, não há nenhuma garantia de que uma sequência permutacional seja ótima em um ambiente composto por várias máquinas. Na realidade é muito provável que o sequenciamento ótimo seja composto por distintas permutações de tarefas para cada máquina. Porém, considerar permutações distintas para cada máquina aumenta tremendamente a complexidade computacional tornando o problema intratável mesmo para instâncias muito pequenas. A solução então se torna considerar a mesma sequência de tarefas para todas as máquinas, ao custo da inclusão de uma restrição desnecessária, mas que reduz a complexidade do problema a níveis mais razoáveis.

Revisões bibliográficas recentes [3, 12] mostram que até o momento, as pesquisas se concentraram mais no caso de tempos de *setup* independentes da sequência. Para o problema considerando tempos dependentes da sequência, temos o trabalho de Hitomi *et al.* [36], onde um modelo de simulação é descrito mostrando que regras de sequenciamento que consideravam *setups* dependentes da sequência explicitamente tiveram melhores resultados que regras que não faziam essa consideração. Tendo isso em vista, Schaller *et al.* [66] desenvolveram e testaram várias heurísticas para minimizar o *makespan* em um ambiente de *flowshop* com tempos de *setup* entre famílias de tarefas dependentes da sequência.

6.2 Descrição do problema

A descrição do problema de Sequenciamento em *Flowshop* será dividida em duas partes. A primeira estabelece como se caracteriza uma instância do mesmo. A segunda por sua vez descreve a função objetivo a ser minimizada - no caso o *makespan* - que é traduzido como o tempo decorrido entre o início do processamento da primeira tarefa na primeira máquina e o final do processamento da última tarefa na última máquina.

Entrada: Seja n o número de tarefas a serem processadas e m o número de máquinas utilizadas na produção. Todas as tarefas são processadas seguindo a mesma rota tecnológica, criando assim a estrutura do *flowshop*. Seja f o número de famílias de tarefas. Considere também o tempo de *setup* para mudar a produção de uma família para outra, representado por $\{S^l_{ij}\}$, onde o elemento $s^l_{i,j}$ é o tempo de *setup* da família j depois que a família i foi processada, na máquina l. Finalmente, seja $\{P_{ij}\}$ a matriz de tempos de processamento, onde o elemento $p_{i,j}$ é o tempo de processamento da tarefa i na máquina j.

Objetivo: Encontrar a permutação das famílias, e das tarefas dentro de cada família, que minimize o *makespan*.

O cálculo do *makespan* para esse problema não é uma tarefa simples. Assim, vamos descrevê-lo pouco a pouco, começando com a nomenclatura utilizada. Suponha que em uma dada solução, as famílias estejam sequenciadas na ordem $\{\pi(1), \pi(2), \ldots, \pi(f)\}$, e que a ordem das tarefas dentro de cada família seja dada pela sequência $\{\sigma_f(1), \sigma_f(2), \ldots, \sigma_f(n_f)\}$, onde n_f é o número de tarefas na família f. Além disso, seja $tt^m_{\sigma_f(i)}$ o tempo total de processamento dentro da família f até a tarefa $\sigma_f(i)$, na máquina m; ou seja, o intervalo de tempo entre o momento em que a máquina terminou o seu *setup*, ficando assim pronta para processar a primeira tarefa da família f, e o instante em que a tarefa $\sigma_f(i)$ é completada. Este valor $tt^m_{\sigma_f(i)}$ pode ser calculado seguindo a Equação 6.1.

$$tt^m_{\sigma_f(i)} = to^m_{\sigma_f(i)} + \sum_{z=1}^{i} p_{\sigma_f(z),m}, \tag{6.1}$$

onde $to^m_{\sigma_f(i)}$ é o tempo ocioso - *idle time* - dentro da família f, na máquina m, acumulado até a tarefa $\sigma_f(i)$. O tempo ocioso ocorre sempre que a máquina está parada, sem operar, aguardando a próxima tarefa ficar disponível. Na primeira máquina não há tempo ocioso, e a sua produção segue sem interrupções. Isso simplifica a Equação 6.1, fazendo-a ser apenas a soma dos tempos de processamento dentro da família f, representada pelo segundo termo da soma. Mas nas outras máquinas, podem ocorrer tempos ociosos, dependendo da sequência adotada. O tempo de término da i-ésima tarefa da f-ésima família na máquina m, representado por $c^m_{\sigma_{\pi(f)}(i)}$ pode assim ser calculado pela Equação 6.2.

$$c^m_{\sigma_{\pi(f)}(i)} = \underbrace{\sum_{z=1}^{f-1} \left(tt^m_{\sigma_{\pi(z)}(n_{\pi(z)})} + s^m_{\pi(z),\pi(z+1)} \right)}_{tempo\ total\ antes\ da\ familia\ \pi(f)} + \underbrace{tt^m_{\sigma_{\pi(f)}(i)}}_{tempo\ total\ dentro\ da\ familia\ \pi(f)} \tag{6.2}$$

A primeira parte da Equação 6.2 calcula o tempo total de processamento antes da f-ésima família, levando em consideração todos os tempos de *setup*, de processamento e ociosos antes dela. O segundo termo calcula o tempo total de processamento dentro da família $\pi(f)$ (tempos de processamento + tempos ociosos) até a tarefa $\sigma_{\pi(f)}(i)$. Agora, falta apenas explicitar o cálculo dos tempos ociosos. Os tempos ociosos ocorrem sempre que uma máquina termina de processar uma tarefa, ou completa o *setup* de uma família, e a próxima tarefa ainda está sendo processada pela máquina anterior. Isso cria um *gap* no sequenciamento, forçando a máquina a esperar até que a próxima tarefa fique disponível. O tempo ocioso dentro da família $\pi(f)$, acumulado até a tarefa $\sigma_{\pi(f)}(i)$ pode ser calculado pela Equação 6.3.

$$to^{m}_{\sigma_{\pi(f)}(i)} = \underbrace{\max(0, c^{m-1}_{\sigma_{\pi(f)}(1)} - c^{m}_{\sigma_{\pi(f-1)}(n_{\pi(f-1)})} + s^{m}_{\pi(f-1),\pi(f)})}_{tempo\ ocioso\ logo\ antes\ da\ 1^a\ tarefa\ da\ f-esima\ familia} + \underbrace{+\sum_{z=1}^{i} \max(0, c^{m-1}_{\sigma_{\pi(f)}(z)} - c^{m}_{\sigma_{\pi(f)}(z-1)})}_{tempos\ ociosos\ entre\ as\ tarefas\ anteriores\ a\ \sigma_{\pi(f)}(i)} \quad (6.3)$$

A primeira parte da Equação 6.3 calcula o tempo ocioso imediatamente anterior à 1ª tarefa da f-ésima família. Para tanto, ela usa informação sobre o tempo de término da última tarefa da família anterior, além do tempo de *setup* entre a f-ésima família e a sua predecessora. A segunda parte adiciona os tempos ociosos entre cada duas tarefas consecutivas da f-ésima família, até a tarefa $\sigma_{\pi(f)}(i)$. Para esse cálculo, a equação utiliza informação do tempo de término da tarefa anterior na máquina atual (m) e da tarefa atual na máquina anterior ($m-1$). O *makespan* é então calculado iterativamente, tarefa a tarefa, máquina a máquina, sendo representado pelo tempo de término da última tarefa na última máquina.

6.3 Problemas testados

As instâncias testadas nesse capítulo são as mesmas apresentadas na referência [66], estando divididas em três classes. Em todas elas, os tempos de processamento são valores inteiros, gerados segundo uma distribuição discreta uniforme DU[1, 10]. Uma das características do problema de *flowshop*, e dos problemas de sequenciamento em geral, é que a dificuldade para se encontrar a sequência ótima é influenciada pela relação entre a dimensão dos tempos de *setup* e dos tempos de processamento. Em geral, quanto maiores forem os tempos de *setup* em relação aos de processamento, mais difícil se torna o processo de busca. A divisão dos problemas em três classes distintas visa avaliar essa característica e sua influência no algoritmo memético. As classes são:

- Tempos de *setup* pequenos (SSU): DU[1, 20]
- Tempos de *setup* médios (MSU): DU[1, 50]
- Tempos de *setup* grandes (LSU): DU[1, 100]

De acordo com as definições SSU, MSU e LSU, a primeira classe possui uma razão média (tempo de *setup*)/(tempo de processamento) de 2:1; para a classe MSU, a razão é de 5:1 e na classe LSU a razão salta para 10:1.

O número de famílias em cada problema varia de 3 a 10, assim como o número de máquinas. Esses dois parâmetros, em conjunto com o tipo de *setup*, definem uma configuração. O número de tarefas por família é sempre um valor aleatório entre 1 e 10. Para cada configuração de parâmetros, 30 instâncias foram testadas. Apenas para exemplificar a notação utilizada nos conjuntos de instâncias, tomemos por exemplo o conjunto *LSU108*. Ele consiste de 30 instâncias com tempos de *setup* no intervalo [1, 100], compostas por 10 famílias

de tarefas e sendo processadas em oito máquinas. Considerando-se todas as combinações adotadas para os parâmetros, foram testados 900 problemas no total.

6.4 Representação e operadores de recombinação, mutação e busca local

O problema de *flowshop* com famílias de tarefas possui uma estrutura que permite sua divisão em duas partes; o sequenciamento das famílias e das tarefas dentro de cada família. A representação utilizada aproveita essa característica para tornar os operadores mais simples e eficientes. Na Figura 6.1 mostramos o exemplo de um cromossomo com 12 tarefas distribuídas em 4 famílias.

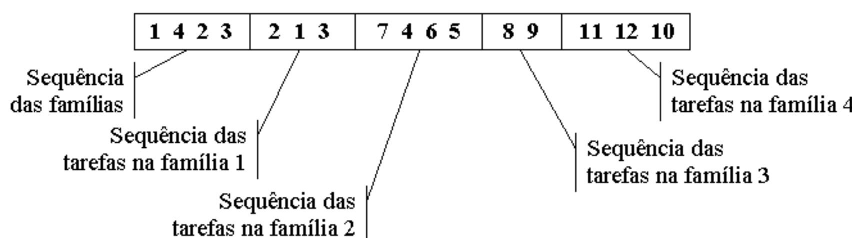


Figura 6.1: *Codificação do cromossomo para o problema de Flowshop.*

Na Figura 6.1 nota-se a presença de quatro famílias de tarefas. A família 1 é composta pelas tarefas [1, 2, 3]; a família 2 pelas tarefas [4, 5, 6, 7]; a família 3 pelas tarefas [8, 9] e a família 4 pelas tarefas [10, 11, 12]. A solução representada pelo cromossomo da Figura 6.1 tem as famílias ordenadas na sequência [1, 4, 2, 3]. As tarefas serão processadas na sequência [2, 1, 3] para a família 1; [7, 4, 6, 5] para a família 2; [8, 9] para a família 3 e finalmente [11, 12, 10] para a família 4.

Essa divisão do cromossomo em $f + 1$ partes independentes faz com que os operadores de busca local, recombinação e mutação sejam aplicados separadamente em cada parte, sem afetar o restante do cromossomo - o que lembra vagamente uma estratégia de *divisão-e-conquista*. De qualquer forma, o resultado é uma redução considerável no esforço computacional, especialmente da busca local.

O operador de recombinação utilizado é o Order Crossover (OX) [29], com a diferença que ele é aplicado sequencialmente em cada parte do cromossomo. Na Figura 6.2, temos um exemplo de como o operador funciona. Inicialmente, o OX seleciona partes do cromossomo do *Pai A* e as copia para o filho nas mesmas posições. Estas partes estão circundadas na figura. Note que apenas um pedaço de cada parte do cromossomo é copiada. Na fase final, o *Pai B* completa o material genético do filho com seus alelos, com cada parte sendo preenchida no sentido esquerda-direita, seguindo a sequência dos alelos ainda não copiados. O número de filhos criados em cada geração é de 26, considerando uma população de tamanho 13. Esse valor é reflexo da adoção de uma taxa

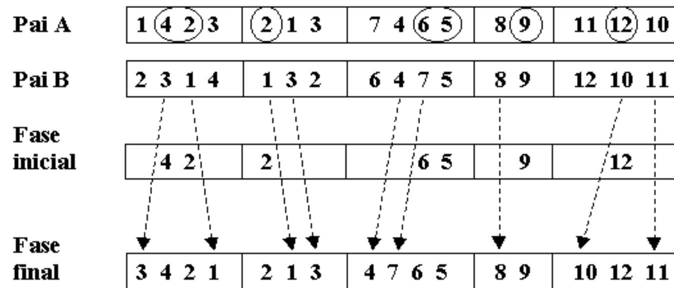


Figura 6.2: *Recombinação para o problema de Flowshop.*

de recombinação alta, que visa contrabalançar o alto número de rejeições na etapa de inserção dos novos indivíduos.

A mutação, por sua vez, é baseada na troca de alelos. Ela inicialmente escolhe uma das partes do cromossomo aleatoriamente. Dentro desta parte será efetuado um único movimento de troca de alelos, sendo este operador aplicado a apenas 10% dos novos indivíduos. Aqui, uma outra vez, a mutação desempenha um papel de pouca importância, pois o método para verificação de convergência é muito estrito e não permite que o algoritmo perca tempo operando com uma população sem diversidade. Desta forma, o papel de manutenção da diversidade, geralmente desempenhado pela mutação não é tão crucial nesta implementação.

A busca local adotada é baseada nas vizinhanças de troca e inserção, muito utilizadas nos problemas apresentados até o momento. O filho inicialmente passa pela busca local de troca e em seguida pela de inserção. Sobre a política de aplicação da busca local, optamos por aplicá-la a todos os novos indivíduos criados, em parte devido à sua baixa complexidade computacional e em parte devido às características do problema de *flowshop* em si. Testes feitos com a aplicação da busca local somente sobre o melhor indivíduo da população se mostraram desastrosos. Observou-se que nesse caso a população evolui de maneira bem mais lenta e a qualidade das soluções finais é consideravelmente pior.

Decidimos não utilizar nenhum tipo de redução de vizinhança, pois mesmos nos problemas de maior dimensão, as buscas locais completas demonstraram ser suficientemente rápidas. A separação do cromossomo em partes distintas reduziu consideravelmente o esforço computacional. A título de ilustração, suponha por exemplo que todas as famílias sejam compostas por t tarefas. O cromossomo inteiro deve possuir então $(f.t + f)$ alelos, ou seja, f famílias de t tarefas cada uma, mais a parte do cromossomo que define a sequência das famílias - mais f alelos - o que resulta no total citado. A ordem de complexidade de uma busca local de troca tipo *all-pairs*, onde são testadas as trocas de todos os possíveis pares de alelos, seria de $O((f.t + f)^2)$. Com a separação do cromossomo em partes, a complexidade cai para $O(f.t^2 + f^2)$, o que torna problemas relativamente grandes perfeitamente tratáveis com o algoritmo memético, mesmo utilizando vizinhanças completas.

6.5 Testes computacionais

Os melhores resultados para o problema de *flowshop* tratado neste capítulo estão no artigo de Schaller *et al.* [66]. Naquele trabalho são fornecidos limitantes inferiores para o *makespan*, além de uma heurística construtiva baseada em busca local chamada CMD.

O algoritmo memético implementado emprega todas as técnicas descritas no Capítulo 2, com exceção da abordagem multipopulacional. Neste trabalho foi utilizada uma única população. O problema de *flowshop* tratado aqui está disponível no NP-Opt, assim como as instâncias testadas. O equipamento utilizado nos testes foi um Pentium II de 266 MHz. Como o CMP é uma heurística relativamente simples, que utiliza em seu final uma única busca local, ela tem um tempo de execução muito pequeno, sempre inferior a três segundos. Para o algoritmo memético, depois de alguns testes preliminares, verificamos que um tempo de CPU da ordem de 30 segundos era mais que suficiente, dado o tamanho das instâncias. Assim, o algoritmo pára somente quando o limitante inferior é atingido - o que significa que a solução ótima foi encontrada - ou ao final de 30 segundos de CPU. As tabelas 6.1, 6.2 e 6.3 comparam os resultados do algoritmo memético com os da heurística CMD, para as instâncias LSU, MSU e SSU, respectivamente.

Os resultados da Tabela 6.1 revelam um desempenho muito bom do algoritmo memético. Os valores das colunas 'Mín', 'Méd' e 'Máx' representam desvios percentuais a partir dos limitantes inferiores. Na coluna dos valores mínimos, os números subscritos entre parênteses são o número de vezes em que o algoritmo atingiu o limitante inferior, encontrando assim a solução ótima. Já na linha das médias, na parte de baixo da tabela, o subscrito representa o total das soluções ótimas encontradas. O algoritmo memético ultrapassou o desempenho na heurística CMD em todas as configurações de instâncias. Um fato que deve ser destacado é que o grande número de soluções ótimas encontradas é um indicativo da alta qualidade dos limitantes inferiores apresentados no trabalho de Schaller *et al.* [66]. Ressaltamos ainda que cada configuração

	Algoritmo memético				**CMD**			
Instân.	**Mín**	**Méd**	**Máx**	**CPU**	**Mín**	**Méd**	**Máx**	**CPU**
LSU33	$0,00_{(27)}$	0,07	1,12	3,1	$0,00_{(21)}$	0,91	8,41	0,04
LSU34	$0,00_{(20)}$	0,32	2,43	10,1	$0,00_{(12)}$	1,08	16,39	0,06
LSU44	$0,00_{(20)}$	0,20	1,09	10,1	$0,00_{(8)}$	1,95	10,27	0,12
LSU55	$0,00_{(18)}$	0,28	1,86	12,1	$0,00_{(4)}$	2,49	9,57	0,21
LSU56	$0,00_{(9)}$	0,51	2,42	21,1	$0,00_{(4)}$	3,37	17,07	0,26
LSU65	$0,00_{(15)}$	0,31	2,42	15,1	$0,00_{(3)}$	3,29	10,02	0,30
LSU66	$0,00_{(15)}$	0,19	1,36	15,3	$0,00_{(3)}$	3,03	10,47	0,44
LSU88	$0,00_{(6)}$	0,58	1,86	24,7	$0,28_{(0)}$	6,25	18,17	0,95
LSU108	$0,00_{(1)}$	0,47	1,19	29,8	$0,12_{(0)}$	6,22	11,25	1,82
LSU1010	$0,00_{(1)}$	0,77	2,27	29,5	$0,53_{(0)}$	6,30	11,42	2,37
Média	$0,00_{(132)}$	0,37	1,80	17,1	$0,09_{(55)}$	3,49	12,30	0,66

Tabela 6.1: *Resultado dos algoritmos para os problemas LSU.*

de instâncias é composta por 30 problemas, o que resulta em um total de 300 problemas por tabela.

Analizando o algoritmo memético, ele encontrou 132 soluções ótimas (44% do total), ou mais que o dobro encontrado pela heurística CMD. Todas as médias foram superiores, com exceção do tempo de CPU, como esperado. Apesar de muito bons, esses resultados eram previsíveis, pois o algoritmo memético possui características bem mais complexas que o CMD, e que tornam seu mecanismo de busca bem mais poderoso. Outra característica interessante é que o desvio percentual médio dos limitantes inferiores sempre se mantém em níveis baixos, independente do tamanho dos problemas. Normalmente, os métodos de busca perdem desempenho à medida em que as instâncias ficam maiores, o que é um sinal que elas estão se tornando muito complexas para a capacidade de busca do algoritmo. Esta característica ficou evidente nos resultados da heurística CMD, cujo desvio médio começa em 0,91% e termina com um desvio considerável de 6,30%. No entanto, isso não foi observado para o algoritmo memético, o que nos leva a crer que o algoritmo ainda está longe do seu limite e que teria sido capaz de lidar com instâncias ainda maiores. Apesar de haver um declínio das médias do algoritmo memético quando as instâncias se tornam maiores, ele provavelmente se deve mais a uma perda de qualidade dos limitantes inferiores do que a uma fraqueza do método de busca.

Analisando a Tabela 6.2, que traz os resultados para os problemas da classe MSU, temos um desempenho similar ao da Tabela 6.1, com o algoritmo memético obtendo resultados superiores aos do CMD em todos os critérios, menos o tempo de CPU. O número de soluções ótimas atingidas pelos dois métodos foi inferior, mas o algoritmo memético manteve a proporção de 2:1 em relação ao CMD, encontrando 98 soluções ótimas (33% do total). As médias foram também um pouco piores que as obtidas para o conjunto de instâncias LSU, mas acreditamos que isso se deva à piora da qualidade dos limitantes inferiores. Ambos os conjuntos de problemas - LSU e MSU - possuem características similares no que diz respeito ao número de máquinas e de famílias. Além disso, o algoritmo memético não utiliza informações dos tempos de *setup*

	Algoritmo memético				**CMD**			
Instân.	**Mín**	**Méd**	**Máx**	**CPU**	**Mín**	**Méd**	**Máx**	**CPU**
MSU33	$0,00_{(23)}$	0,37	3,37	7,1	$0,00_{(21)}$	0,92	11,46	0,04
MSU34	$0,00_{(17)}$	0,56	2,29	13,1	$0,00_{(11)}$	2,00	16,28	0,05
MSU44	$0,00_{(11)}$	0,50	2,32	19,1	$0,00_{(4)}$	1,96	11,11	0,13
MSU55	$0,00_{(15)}$	0,45	2,09	15,1	$0,00_{(4)}$	3,10	8,48	0,19
MSU56	$0,00_{(6)}$	0,87	3,12	24,1	$0,00_{(1)}$	3,58	13,13	0,27
MSU65	$0,00_{(14)}$	0,36	1,22	16,2	$0,00_{(3)}$	3,68	8,88	0,30
MSU66	$0,00_{(8)}$	0,50	1,63	22,7	$0,00_{(1)}$	4,59	15,77	0,40
MSU88	$0,00_{(3)}$	0,99	2,98	27,3	$0,63_{(0)}$	5,68	12,68	0,97
MSU108	$0,00_{(1)}$	0,86	1,80	30,1	$2,89_{(0)}$	6,11	10,83	1,84
MSU1010	$0,15_{(0)}$	1,15	2,53	30,8	$2,29_{(0)}$	5,73	9,92	2,37
Média	$0,01_{(98)}$	0,66	2,33	20,6	$0,58_{(45)}$	3,73	11,85	0,65

Tabela 6.2: *Resultado dos algoritmos para os problemas MSU.*

em seus operadores, que assim não deveriam ter seus comportamentos afetados pela diferença na forma com que foram gerados os mesmos. Em vista disso, pelo menos no que se refere ao algoritmo memético, não deveriam ocorrer variações relevantes do posto de vista estatístico no número de soluções ótimas encontradas. A redução de 132 para 98 - aproximadamente 25% - nas soluções ótimas atingidas nos leva a crer que o método para geração de limitantes inferiores apresentado por Schaller *et al.* [66] deve funcionar melhor para problemas com tempos de *setup* maiores. Esse ponto de vista é reforçado também pela redução no número de soluções ótimas encontradas pelo método CMD.

	Algoritmo memético				**CMD**			
Instân.	**Mín**	**Méd**	**Máx**	**CPU**	**Mín**	**Méd**	**Máx**	**CPU**
SSU33	$0,00_{(23)}$	0,31	2,47	7,1	$0,00_{(18)}$	0,67	4,13	0,04
SSU34	$0,00_{(15)}$	0,83	2,94	15,1	$0,00_{(8)}$	1,85	8,46	0,05
SSU44	$0,00_{(15)}$	0,57	2,90	15,0	$0,00_{(8)}$	1,94	8,33	0,12
SSU55	$0,00_{(4)}$	0,92	2,34	26,0	$0,00_{(1)}$	3,15	6,61	0,20
SSU56	$0,00_{(3)}$	1,56	3,08	27,4	$0,00_{(1)}$	4,02	8,93	0,25
SSU65	$0,00_{(6)}$	0,99	3,44	24,0	$0,00_{(2)}$	3,00	6,90	0,32
SSU66	$0,00_{(1)}$	1,28	2,72	29,1	$1,24_{(0)}$	4,06	9,16	0,44
SSU88	$0,28_{(0)}$	1,85	3,31	30,3	$3,20_{(0)}$	5,62	8,59	0,91
SSU108	$0,72_{(0)}$	1,77	2,90	30,7	$2,58_{(0)}$	5,63	8,96	1,82
SSU1010	$0,59_{(0)}$	2,33	3,65	30,6	$4,07_{(0)}$	6,86	9,11	2,21
Média	$0,16_{(67)}$	1,24	2,98	23,5	$1,11_{(38)}$	3,68	7,92	0,64

Tabela 6.3: *Resultado dos algoritmos para os problemas SSU.*

Na Tabela 6.3, referente aos problemas da classe SSU, pode-se ver que o algoritmo memético manteve seu desempenho. Foram observados bons valores médios, além do esperado decréscimo no número de soluções ótimas encontradas. Este valor se reduziu para 22% do total dos problemas testados. A perda de desempenho que vem junto com o aumento do tamanho da instância se torna mais evidente. Em vista disso temos o algoritmo memético falhando em atingir qualquer solução ótima para instâncias com 8 ou mais famílias de tarefas.

Dada as experiências anteriores com buscas locais, e como elas se comportam com diferentes tamanhos de instâncias, é muito provável que o número de soluções ótimas efetivamente encontrado pelo algoritmo memético seja bem maior que o relatado. É difícil acreditar que o algoritmo tenha encontrado grandes dificuldades em quaisquer dos problemas testados, pois a representação utilizada reduziu consideravelmente a complexidade dos mesmos. De fato, efetuar buscas locais de troca e inserção em conjuntos de 10 alelos é uma tarefa das mais simples e o poder de busca das duas vizinhanças $O(n^2)$ associadas nesse caso é enorme. Apesar dos problemas necessitarem de várias buscas locais sequenciais, a complexidade total ainda permanece baixa, do nosso ponto de vista.

6.6 Resumo

Neste capítulo tratamos do problema de sequenciamento em *Flowshop*. O problema possui algumas características especiais, como a divisão das tarefas em famílias e o uso de tempos de *setup* entre as famílias de tarefas. Por uma questão de simplificação do modelo, consideramos que a sequência de processamento das tarefas não muda de uma máquina para outra, o que caracteriza um *flowshop* permutacional. O objetivo é a minimização do *makespan*.

O algoritmo memético utilizado segue a versão unipopulacional e utiliza os recursos comuns do NP-Opt - população hierarquicamente estruturada e operadores de seleção e inserção especiais - e não apresentou maiores dificuldades para lidar com instâncias de até 10 famílias, 10 tarefas por família e 10 máquinas. Boa parte do bom desempenho se deve à representação utilizada, que divide o cromossomo em várias partes distintas, reduzindo bastante o esforço computacional do processo de busca local. Assim, não foi necessário implementar políticas de redução de vizinhança. Vale ressaltar que ao final dos testes ainda ficamos com a impressão de que o algoritmo teria capacidade de busca suficiente para tratar problemas bem maiores.

As instâncias tratadas variaram entre 3 famílias de tarefas e 3 máquinas até 10 famílias e 10 máquinas. Os grupos de problemas foram divididos em três classes: SSU, MSU e LSU. A diferença entre essas classificações está no intervalo em que os tempos de *setup* foram gerados.

Os resultados computacionais foram excelentes, com o algoritmo memético atingindo os valores ótimos em boa parte dos casos e sempre superando a heurística CMD - o melhor método para o problema até então.

Capítulo 7

Sequenciamento em Máquina Simples

7.1 Introdução

[1]O problema de Sequenciamento em Máquina Simples (SMS) foi um dos primeiros problemas estudados na área de sequenciamento da produção [32, 33]. Existem vários tipos de problemas de SMS - geralmente devido aos diferentes tipos de dados de entrada e de função objetivo. Um dos mais simples de se definir, mas não tão simples de se resolver, é o problema de sequenciar n tarefas, dados os seus tempos de processamento e datas de entrega - diferentes para cada tarefa; sendo o objetivo minimizar o atraso total. Como todos os outros problemas tratados no conjunto deste trabalho, este também pertence à classe *NP-hard* [44].

O atraso total é uma medida de desempenho *regular*, o que significa que no sequenciamento ótimo não há tempos ociosos entre as tarefas. Além disso, o atraso aumenta somente se o tempo de término de pelo menos uma das tarefas também aumentar. Sob essas condições, uma solução válida pode ser representada como uma permutação das tarefas, o que nos remete a um espaço de busca cujo tamanho é de $n!$ possíveis soluções.

O problema de SMS tratado neste capítulo possui, além das características citadas anteriormente, tempos de *setup* entre as tarefas. Os tempos de *setup* são em geral necessários para efetuar limpeza, trocas de ferramentas e ajustes na máquina quando mudamos a produção de uma peça para outra. Para outros problemas comuns na área de sequenciamento, sugerimos consultar o livro de Gen e Cheng [25], que traz uma grande quantidade de variantes do problema tratado aqui.

Mesmo quando nos deparamos com um ambiente de manufatura industrial complexo, não é difícil identificar situações onde um problema de SMS tenha de

[1]Este capítulo é baseado no artigo:

A. Mendes, P. França e P. Moscato. **Fitness Landscapes for the Total Tardiness Single Machine Scheduling Problem**, *Neural Network World - International Journal on Neural and Mass-Parallel Computing and Information Systems*, 2(2):165-180, 2002.

ser resolvido [62]. Em processos que necessitam de múltiplas operações, uma dada peça passa por diversas máquinas, sendo que cada máquina desempenha uma tarefa específica. Dependendo dos processos envolvidos, podem se formar 'gargalos' na produção, que poderiam ser modelados como problemas de SMS. Tarefas complexas, realizadas por máquinas dedicadas se encaixam nessa categoria. Essas máquinas são em geral muito caras e inúmeras vezes há apenas uma no ambiente de produção, que é utilizada *full-time*. Essa situação representa um caso típico de SMS, onde a produção da fábrica como um todo será prejudicada no caso de um sequenciamento mal planejado.

7.2 Descrição do problema

O problema de SMS tratado considera que as tarefas são independentes, compostas por uma única operação e estão disponíveis a partir do instante zero - ou seja, o *ready-time* é zero para todas as tarefas. A máquina está sempre disponível e não há restrição de recursos. A disponibilidade de ferramentas não é um fator limitante para a produção. Não são consideradas quebras de máquina ou outras contingências. E finalmente, todos os parâmetros que definem o ambiente de produção - tempos de processamento, tempos de *setup* e datas de entrega - são previamente conhecidos. A seguir temos a definição do problema em questão.

Entrada: Seja n o número de tarefas a serem processadas em uma máquina. Seja $P = \{p_1, p_2, \ldots, p_n\}$ a lista de tempos de processamento das tarefas. Seja $D = \{d_1, d_2, \ldots, d_n\}$ a lista de datas de entrega e $S_0 = \{s_{01}, s_{02}, \ldots, s_{0n}\}$ a lista tempos de *setup* iniciais. Seja ainda $\{S_{ij}\}$ a matriz dos tempos de *setup*, onde $s_{i,j}$ é o tempo necessário para preparar a produção da tarefa j depois que a máquina terminou o processamento da tarefa i.

Objetivo: Encontrar a permutação que minimize o atraso total da produção. Suponha que as tarefas estejam ordenadas na sequência $\{\pi(1), \pi(2), \ldots, \pi(n)\}$. Então, o atraso total pode ser calculado pela Equação 7.1.

$$Atraso_{total} = \sum_{k=1}^{n} max\Big[0, c_{\pi(k)} - d_{\pi(k)}\Big] \tag{7.1}$$

Onde $c_{\pi(k)}$ representa o tempo de término da tarefa $\pi(k)$. Os valores $c_{\pi(k)}$ podem ser calculados pela Equação 7.2.

$$c_{\pi(k)} = s_{0\pi(1)} + p_{\pi(1)} + \sum_{i=2}^{k} s_{\pi(i-1),\pi(i)} + p_{\pi(i)} \tag{7.2}$$

O problema de sequenciar tarefas em uma máquina, sem tempos de *setup*, já pertence à classe *NP-hard* [18]. Apesar de sua grande aplicabilidade em ambientes de produção reais, o problema tratado aqui recebeu pouca atenção na literatura de sequenciamento. Os trabalhos prévios incluem um *branch-and-bound* proposto por Ragatz [63], onde apenas instâncias pequenas foram resolvidas à otimalidade. Os trabalhos de Raman *et al.* [64] e de Lee *et al.* [45]

apresentam regras de despacho baseadas no cálculo de um índice de prioridade para construir uma sequência inicial, que é então otimizada pela aplicação de um procedimento de busca local. Convém ressaltar que a regra *ATCS* [45] teve um desempenho impressionante, considerando a sua simplicidade. Três trabalhos apresentam metaheurísticas para o problema de SMS. Rubin e Ragatz [65] desenvolveram um novo operador de recombinação e aplicou um algoritmo genético para um conjunto de problemas de tamanho médio. Os resultados foram então comparados com um *branch-and-bound* e uma estratégia de *multiple start*. No trabalho de Tan e Narasimhan [70] também é utilizado um método de *multiple start* como base de comparação, porém o algoritmo desenvolvido é um *simulated annealing*. Mais recentemente, temos o trabalho de França *et al.* [23], onde um novo algoritmo memético é apresentado e os resultados se mostram superiores aos das abordagens anteriores. Os experimentos nesse capítulo são uma extensão desse trabalho e apresentam melhoras significativas.

7.3 Criação das instâncias

O conjunto de teste utilizado para a análise experimental consiste de 20 problemas com tamanhos variando entre 17 e 100 tarefas. Ele foi criado a partir de instâncias do *Problema do Caixeiro Viajante Assimétrico* (PCVA), cujas respectivas soluções ótimas estão disponíveis na literatura [72]. Com algumas modificações estruturais, foi possível transformar as instâncias do PCVA em instâncias de SMS, mantendo o *tour* ótimo do PCVA como a sequência ótima do SMS. Todas as instâncias utilizadas nesse trabalho estão disponíveis e acompanham o software NP-Opt.

É evidente que instâncias de PCVA e de SMS possuem algumas semelhanças. A mais importante delas, no nosso caso, é a relação entre os tempos de *setup* e a matriz de distâncias entre cidades. Essa correspondência nos remeteu a um procedimento de transformação para criar instâncias de SMS a partir de instâncias de PCVA, que é dividido em três etapas. Na transformação, a matriz de tempos de *setup* deve ser igual, ou um múltiplo da matriz de distância entre cidades. Então, a criação dos parâmetros que faltam segue procedimentos simples. Inicialmente, suponha que $\{\pi(1), \pi(2), \ldots, \pi(n)\}$ represente a sequência das cidades vizitadas no *tour* ótimo do problema de PCVA.

Passo 1 - Criação da lista S_0: Seja $[\pi(k), \pi(k+1)]$ o par de cidades vizinhas do *tour* do PCVA com a maior distância entre cidades - representada pela variável $dist$. Seja $s_{0,\pi(k+1)} = dist_{\pi(k),\pi(k+1)}$. Sejam todos os outros s_{0j}, $j \neq \pi(k+1)$ maiores que $s_{0,\pi(k+1)}$. Se $dist_{\pi(k+1),\pi(k)} \leq dist_{\pi(k),\pi(k+1)}$, então a tarefa inicial será a de índice $\pi(k+1)$.

Passo 2 - Criação da lista P: Inicialmente, renumere as cidades de forma que $\pi(1) \Leftarrow \pi(k+1)$; $\pi(2) \Leftarrow \pi(k+2)$; $\ldots$; $\pi(n) \Leftarrow \pi(k)$. Em seguida, construa a lista $\{p_{\pi(1)}, p_{\pi(2)}, \ldots, p_{\pi(n)}\}$ de forma que $\{p_{\pi(1)} < p_{\pi(2)} < \ldots < p_{\pi(n)}\}$.

Passo 3 - Criação da lista D: Defina as datas de entrega de forma que $\{d_{\pi(1)} < d_{\pi(2)} < \ldots < d_{\pi(n)}\}$ e $d_{\pi(i)} \in [c_{\pi(i)} - p_{\pi(i)}, c_{\pi(i)}]\ \forall\ \pi(i) : i = 1, \ldots, n$, onde $c_{\pi(i)}$ é o tempo de término da tarefa $\pi(i)$ segundo o *tour* ótimo do PCVA.

No momento ainda não há prova matemática de que a transformação des-

crita preserve o *tour* ótimo do PCVA como a sequência ótima do SMS e por isso os valores dos atrasos totais ótimos apresentados para as instâncias ainda não estão provados serem verdadeiros. Ressaltamos, no entanto, que em todos os testes executados, nunca foi encontrado um valor de atraso menor que os valores supostamente ótimos e em vista disso, esses valores ainda podem ser considerados *limitantes superiores* de alta qualidade. Agora vamos descrever o porquê da transformação sugerida.

A manutenção da matriz de distâncias entre cidades como os tempos de *setup* é lógica. Imagine se fizermos todos os tempos de processamento iguais a zero no problema de SMS. Neste caso, reduzimos o SMS a um problema de PCVA com prazos limites para se chegar em cada cidade.

O fato de se começar a sequência de produção pelo *setup* equivalente à maior distância pertencente ao *tour* ótimo tem por objetivo evitar que uma troca na tarefa inicial gere uma redução em qualquer um dos tempos de término, o que faria com que a sequência ótima fosse perdida. Outro ponto importante é a exigência $dist_{[\pi(k+1),\pi(k)]} \leq dist_{[\pi(k),\pi(k+1)]}$. Vamos exemplificar com um caso onde a transformação não funciona, descrito na Figura 7.1.

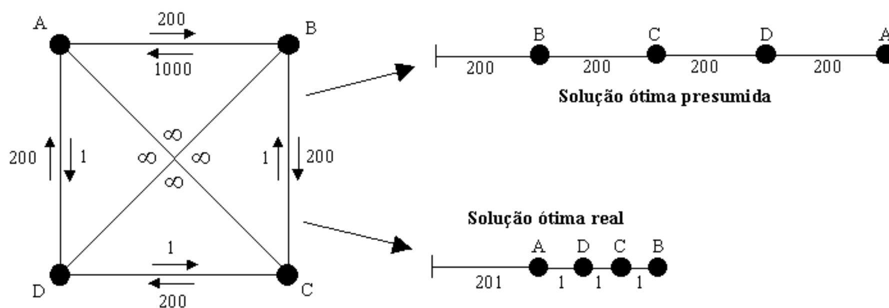


Figura 7.1: *Caso errôneo da transformação PCVA/SMS.*

Na Figura 7.1, temos na parte esquerda uma instância de PCVA composta por quatro cidades. Como os arcos internos valem infinito, duas rotas com custo menor que infinito são, por exemplo, [B C D A], com tamanho 800 e [A D C B], com tamanho 1003. Assim, um possível *tour* ótimo seria [B C D A]. Quando passamos para o SMS, na parte direita, optamos, sem perda de generalidade, por ignorar tanto os tempos de processamento quanto as datas de entrega. Nesse caso, estamos apenas interessados no efeito causado pelos tempos de *setup*. Na parte direita da Figura 7.1 mostramos a sequência ótima presumida do SMS, cujo *makespan* de 800 é igual à distância total do *tour* ótimo do PCVA. Porém, pelo *Passo 1* da transformação, se não houver a restrição $dist_{[\pi(k+1),\pi(k)]} \leq dist_{[\pi(k),\pi(k+1)]}$, e o arco inicial ao ser percorrido no sentido oposto tiver um valor muito alto, ele poderá ser substituído na transformação por um valor pequeno, porém maior que $s_{0,\pi(k+1)}$. No exemplo mostrado, ele foi substituído por 201, que é bem menor que 1000. Com isso, o ótimo real passa a ser menor que o presumido e a transformação falha. Assim, uma condição necessária é que o arco inicial tenha sempre um valor maior quando é percorrido no sentido da solução ótima presumida do que quando é percorrido

no sentido oposto.

Os passos 2 e 3, por sua vez, têm a mesma função. Ao ordenar os tempos de processamento e as datas de entrega em ordem crescente segundo a sequência ótima, evitamos que a troca de uma tarefa de posição gere uma redução no tempo de término, ou uma redução no atraso de uma parte da sequência. A geração das listas P e D produz instâncias que são resolvidas na otimalidade utilizando-se as heurísticas *Shortest Processing Time* (SPT) e *Earliest Due Date* (EDD). Métodos que utilizem essas heurísticas, ou suas variantes, irão resolver as instâncias instantâneamente, produzindo resultados tendenciosos. No entanto, para avaliar o algoritmo memético implementado, que não utiliza nenhuma das duas heurísticas em seus operadores, as instâncias são absolutamente adequadas.

7.3.1 Parâmetros variáveis

Como a matriz de tempos de *setup* é fixa, os únicos parâmetros livres são os tempos de processamento e as datas de entrega das tarefas. Para cada parâmetro especificamos duas regras de criação, o que gerou quatro tipos distintos de instâncias.

Tempos de processamento:

$$LOW \Rightarrow p_k \in \left[0, \frac{1}{4} \cdot \max(s_{i,j})\right] \quad \forall k: \ k = 1, \dots, n$$

$$HIGH \Rightarrow p_k \in \left[0, 2 \cdot \max(s_{i,j})\right] \quad \forall k: \ k = 1, \dots, n$$

Datas de entrega:

$$HARD \Rightarrow d_k = c_k \quad \forall k: \ k = 1, \dots, n$$

$$SOFT \Rightarrow d_k \in \left[c_k - p_k, c_k\right] \quad \forall k: \ k = 1, \dots, n$$

A regra LOW faz do tempo de *setup* uma característica mais crítica, enfatizando o lado PCVA das instâncias. Nelas, os tempos de processamento são pequenos quando comparados aos tempos de *setup*. Por outro lado, nas instâncias *HIGH*, o lado do sequenciamento é mais relevante pois os tempos de processamento são até duas vezes maiores que os tempos de *setup*.

A regra $HARD$ cria instâncias cujo atraso ótimo é zero. Esse valor é garantido pois as datas de entrega estão localizadas justamente no instante em que cada tarefa é completada seguindo a sequência ótima. Já nas instâncias $SOFT$, o atraso total é maior que zero e seu uso é recomendado para fazer comparações relativas, onde desvios percentuais são numericamente necessários. Na Tabela 7.1, são mostradas as características das instâncias originais do PCVA, como número de cidades e distâncias mínima e máxima entre cidades.

Instância	n	**Mín** $dist_{i,j}$	**Máx** $dist_{i,j}$
br17	17	0	74
ftv33	34	7	332
ftv55	56	6	324
ftv70	71	5	348
kro124p	100	81	4.545

Tabela 7.1: *Parâmetros das instâncias originais do PCVA.*

7.4 Representação, população inicial, recombinação, mutação e busca local.

A representação utilizada para o problema de SMS é bastante intuitiva. Como uma solução pode ser representada por uma permutação simples das tarefas, optamos por um cromossomo cujos alelos assumem valores inteiros e distintos no intervalo [1, n]. A população inicial é inteiramente composta de indivíduos gerados aleatoriamente. Mesmo para as instâncias maiores, foi possível atingir soluções de boa qualidade rapidamente com o uso da busca local, mesmo partindo de populações inteiramente aleatórias. Em testes preliminares com o algoritmo genético puro, o desempenho foi bem pior. Nesse caso sugerimos o uso de uma heurística para a criação da população inicial, nos moldes da regra ATCS por exemplo, para se obter uma maior eficiência do método.

Para a recombinação, utilizamos dois operadores - OX [29] e BOX. Este último guarda semelhanças com a 2ª variante do crossover OX que é apresentada no trabalho de Syswerda [69]. Ambos atuam de forma semelhante, mas enquanto o primeiro copia apenas uma parte do cromossomo do *Pai A* - constituído pelo líder do *cluster* - para o filho, o segundo copia várias partes dele. Em seguida, a informação que falta para completar o filho é obtida a partir do cromossomo do *Pai B* - constituído por um dos seguidores. A Figura 7.2 ilustra os dois operadores de recombinação.

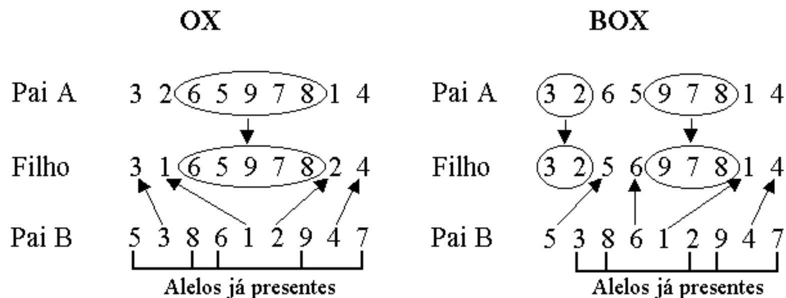


Figura 7.2: *Operadores de recombinação para o problema de SMS.*

Na Figura 7.2 são mostrados os dois operadores de recombinação utilizados nos testes computacionais. Os resultados mostraram que o operador BOX é superior ao OX. Aparentemente, a possibilidade de transmitir blocos separados

do cromossomo para os filhos é vantajosa para o problema. O *crossover* BOX implementado neste trabalho é o mesmo utilizado no capítulo que trata do problema de Gate Matrix Layout, uma vez que ambos os problemas utilizam a mesma representação. O número de indivíduos criados em cada geração é bem alto - duas vezes o número de indivíduos na população. Esse valor é consequência da política de aceitação dos novos indivíduos ser muito restrita. Assim, para balancear o alto número de indivíduos não aproveitados, deve-se gerar uma grande quantidade deles a cada geração.

A busca local segue a mesma linha descrita no trabalho de França *et al.* [23]. Ela é baseada nas vizinhanças de troca *all-pairs* e *inserção*. No entanto, dado que ambas possuem complexidade de $O(n^2)$, foi necessário adotar uma regra para redução de vizinhança. Um estudo extenso que compreende diversas regras pode ser encontrado também no trabalho de França *et al.* [23]. Uma vez que o foco principal desse capítulo é a avaliação do desempenho do modelo unipopulacional contra o do modelo multipopulacional, utilizamos em todos os experimentos apenas a melhor redução de vizinhança. A regra adotada se baseia na variação dos tempos de *setup* da solução quando as tarefas mudam de posição. Se for verificada alguma redução nos tempos de *setup*, o movimento é testado, gastando-se assim um cálculo de função de *fitness*. Caso contrário, a tentativa é descartada, economizando-se tempo de CPU. De fato, é muito provável que uma modificação na sequência só reduza o atraso total caso ao menos um dos tempos de *setup* que a compõe também se reduza. Para informações mais completas a respeito da busca local, sugerimos a leitura da referência citada anteriormente nesse parágrafo.

7.5 Testes computacionais

Os testes computacionais foram divididos em duas partes. A primeira se refere ao algoritmo memético unipopulacional, utilizando os dois operadores de recombinação. Na segunda parte, avaliamos o mesmo algoritmo memético, porém com múltiplas populações. A diferença de desempenho entre as quatro configurações testadas é notável.

7.5.1 Versão unipopulacional

A Tabela 7.2 mostra os resultados obtidos para o algoritmo memético unipopulacional, utilizando os operadores de recombinação OX e BOX. O tempo computacional foi fixado em quatro minutos, sendo que nas vezes em que o ótimo presumido é atingido, contabilizamos esse valor como o tempo de CPU. O equipamento utilizado foi um Pentium II Celeron de 366 MHz. Estes testes utilizaram algoritmo memético sequencial disponível no NP-Opt e as instâncias fazem parte do pacote do software.

Os resultados da Tabela 7.2 mostram que o algoritmo memético unipopulacional conseguiu resolver a maior parte do conjunto de instâncias. O crossover OX obteve um desempenho geral pior que o BOX, falhando em boa parte das instâncias de 100 tarefas. O BOX, por sua vez, obteve sucesso na maioria dos problemas. Nas colunas intituladas 'Atraso médio', temos entre parênteses o

Instância	Atraso ótimo presum.	Atraso médio OX	Atraso médio BOX	Tempo CPU OX	Tempo CPU BOX
br17LH	0	$0_{(10)}$	$0_{(10)}$	0,1	0,1
br17LS	52	$52_{(10)}$	$52_{(10)}$	0,1	0,1
br17HH	0	$0_{(10)}$	$0_{(10)}$	0,1	0,1
br17HS	547	$547_{(10)}$	$547_{(10)}$	0,1	0,1
ftv33LH	0	$0_{(10)}$	$0_{(10)}$	2,1	2,1
ftv33LS	664	$664_{(10)}$	$664_{(10)}$	2,8	2,6
ftv33HH	0	$0_{(10)}$	$0_{(10)}$	1,3	1,3
ftv33HS	5.324	$5.324_{(10)}$	$5.324_{(10)}$	1,5	1,4
ftv55LH	0	$0_{(10)}$	$0_{(10)}$	17,4	13,8
ftv55LS	1.170	$1.170_{(10)}$	$1.170_{(10)}$	50,9	19,8
ftv55HH	0	$0_{(10)}$	$0_{(10)}$	6,9	8,0
ftv55HS	8.515	$8.515_{(10)}$	$8.515_{(10)}$	12,9	14,5
ftv70LH	0	$918,6_{(7)}$	$0_{(10)}$	148,5	46,0
ftv70LS	1.506	$1.557,4_{(8)}$	$1.506_{(10)}$	165,0	70,1
ftv70HH	0	$0_{(10)}$	$0_{(10)}$	41,8	23,6
ftv70HS	12.368	$12.368_{(10)}$	$12.368_{(10)}$	40,3	35,3
kro124pLH	0	$23.872,0_{(2)}$	$22.717,1_{(3)}$	229,5	205,5
kro124pLS	26.111	$74.636,3_{(0)}$	$68.618,8_{(1)}$	240,0	224,3
kro124pHH	0	$4.710,6_{(7)}$	$0_{(10)}$	157,3	107,1
kro124pHS	223.890	$238.114,2_{(3)}$	$231.997,5_{(6)}$	220,4	185,5
* Total de soluções ótimas		*(167)	*(180)		

Tabela 7.2: *Resultados do algoritmo memético unipopulacional.*

número de vezes que o ótimo presumido foi atingido - de um total de 10 tentativas. Com respeito a esse critério, a obtenção de 167 e 180 soluções ótimas - de 200 possíveis - para os dois *crossovers* pode ser considerada uma taxa bastante razoável, tendo em vista a variedade de tamanho das mesmas. Com base nos resultados, também pode-se dizer que as instâncias de 100 tarefas estão no limiar da capacidade de busca dos algoritmos, pois o número de ótimos encontrados reduziu-se de forma drástica. Para a instância *kro124pLS*, por exemplo, o ótimo foi encontrado uma única vez, em 20 tentativas. Outro ponto importante é o tempo de CPU gasto por cada configuração. O BOX necessita de menos tempo de CPU que o OX para chegar nas soluções ótimas. Essa característica fica ainda mais evidente nas instâncias *ftv70* e *kro124p*.

7.5.2 Versão multipopulacional

Os testes da versão multipopulacional visam validar o uso de múltiplas populações para o problema de SMS. O número de populações foi fixado em quatro, e a política de migração adotada foi a *1-Migrate*, em consonância com os resultados obtidos em outros problemas onde foi adotada essa abordagem. O tempo de CPU máximo permaneceu fixo em 240 segundos e a Tabela 7.3 apresenta os resultados detalhados dos testes.

O algoritmo multipopulacional obteve um desempenho pouco melhor que o

Instância	Atraso ótimo presum.	Atraso médio OX	Atraso médio BOX	Tempo CPU OX	Tempo CPU BOX
br17LH	0	$0_{(10)}$	$0_{(10)}$	0,1	0,1
br17LS	52	$52_{(10)}$	$52_{(10)}$	0,1	0,1
br17HH	0	$0_{(10)}$	$0_{(10)}$	0,1	0,1
br17HS	547	$547_{(10)}$	$547_{(10)}$	0,1	0,1
ftv33LH	0	$0_{(10)}$	$0_{(10)}$	1,7	2,2
ftv33LS	664	$664_{(10)}$	$664_{(10)}$	3,4	2,6
ftv33HH	0	$0_{(10)}$	$0_{(10)}$	1,2	1,6
ftv33HS	5.324	$5.324_{(10)}$	$5.324_{(10)}$	1,6	2,0
ftv55LH	0	$0_{(10)}$	$0_{(10)}$	43,8	13,6
ftv55LS	1.170	$1.170_{(10)}$	$1.170_{(10)}$	40,1	22,5
ftv55HH	0	$0_{(10)}$	$0_{(10)}$	8,4	8,1
ftv55HS	8.515	$8.515_{(10)}$	$8.515_{(10)}$	12,8	8,7
ftv70LH	0	$0_{(10)}$	$0_{(10)}$	122,7	48,3
ftv70LS	1.506	$1.684,7_{(8)}$	$1.506_{(10)}$	172,9	53,6
ftv70HH	0	$0_{(10)}$	$0_{(10)}$	39,2	18,0
ftv70HS	12.368	$12.368_{(10)}$	$12.368_{(10)}$	50,8	35,1
kro124pLH	0	$48.536,5_{(1)}$	$26.175,9_{(6)}$	231,7	206,6
kro124pLS	26.111	$72.553,7_{(0)}$	$53.807,4_{(4)}$	240,0	227,7
kro124pHH	0	$3.337,5_{(8)}$	$0_{(10)}$	159,3	118,9
kro124pHS	223.890	$240.213,1_{(3)}$	$233.074,3_{(9)}$	228,1	141,3
* Total de soluções ótimas		*(170)	*(189)		

Tabela 7.3: *Resultados do algoritmo memético multipopulacional.*

unipopulacional. O número de ótimos encontrados foi maior - 170 e 189 contra 167 e 180 da versão unipopulacional. Com o uso do *crossover* BOX conseguiu-se excelentes resultados inclusive para as instâncias de 100 tarefas, o que nos leva a acreditar que o método ainda conseguiria resolver problemas maiores. Aqui também se repete a relação entre os tempos de CPU utilizados pelo OX e pelo BOX. O segundo é bem mais eficiente e chega nas soluções ótimas mais rapidamente.

7.6 Resumo

Neste capítulo abordamos o problema de Sequenciamento em Máquina Simples (SMS) com tempos de *setup* e datas de entrega. O objetivo é minimizar o atraso total do sequenciamento. A principal contribuição apresentada foi a formulação de um procedimento para transformação de instâncias do problema do Caixeiro Viajante Assimétrico (PCVA) em instâncias de Sequenciamento em Máquina Simples. Apesar de ainda não ter sido demonstrado matematicamente que o procedimento funciona, são apresentadas algumas condições que, quando não satisfeitas, invalidam a transformação. De uma forma estrita, pode-se dizer que o procedimento é capaz de gerar simplesmente instâncias com limitantes superiores conhecidos de boa qualidade. A motivação dessa transformação é

bastante simples. Existem instâncias de PCVA resolvidas na otimalidade para centenas e até milhares de cidades, disponíveis na literatura e em *websites*. Já para o problema de SMS, é muito difícil obter instâncias grandes para testes de novos métodos - atualmente estão disponíveis na literatura poucas instâncias ótimas, sendo todas elas pequenas, de no máximo 50 tarefas. Assim, a possibilidade de criar instâncias de SMS de uma maneira simples, e de dimensões até então inatingíveis, nos pareceu uma opção tentadora.

Os testes computacionais foram realizados sobre um conjunto de 20 instâncias transformadas do ATSP, que variam entre 17 e 100 tarefas. Os valores ótimos presumidos para os atrasos totais são frequentemente atingidos pelo algoritmo memético, em tempos computacionais relativamente curtos - de até quatro minutos. Foram testadas as versões uni e multipopulacionais do algoritmo memético presente no NP-Opt e também dois tipos de recombinação - OX e BOX. A estratégia BOX obteve resultados superiores, assim como a versão multipopulacional.

As buscas locais utilizadas são as de troca e inserção, em conjunto com políticas de redução de vizinhanças. Essas reduções são imprescindíveis para os problemas maiores, de mais de 70 tarefas, onde a complexidade computacional começa a se tornar um ponto crítico.

O número de ótimos presumidos atingidos pelo algoritmo memético reforça a robustez do método - em mais de 90% dos testes o algoritmo obteve sucesso. Além disso, devemos ressaltar que durante os testes, não foi obtida nenhuma solução melhor que o ótimo presumido, para nenhuma das 20 instâncias. Isso também reforça a idéia de que o procedimento de transformação, mesmo não tendo sido ainda provado matematicamente correto, apresenta um alto grau de sucesso no seu intento.

Capítulo 8

Ordenamento de Genes

8.1 Introdução

[1] Devido à grande quantidade de dados gerada depois do *Projeto Genoma Humano*, assim como as informações compiladas a partir de outras iniciativas da área, a tarefa de interpretar as relações entre genes aparece como um dos grandes desafios para a comunidade científica [43]. A abordagem tradicional da Biologia Molecular, baseada no cenário 'um gene, um experimento', limita sensivelmente o conhecimento do 'quadro geral' e torna difícil rastrear as interações das funções genéticas. Por essa razão, novas tecnologias têm sido desenvolvidas nos últimos anos. Nesse sentido, a técnica de *DNA Microarray* [8, 16] tem atraído muito interesse pois ela permite monitorar a atividade de um genoma completo em apenas um experimento, simplificando e agilizando o trabalho de mapeamento genético.

Tendo em vista a análise da enorme quantidade de dados que continuamente são disponibilizados, o uso de técnicas de redução se torna uma prática absolutamente necessária. De fato, acredita-se que a atividade de cada gene seja influenciada por não mais que 10 outros genes [4]. Para atingir este grau de redução, e consequentemente permitir que biólogos moleculares se concentrem nos subconjuntos sensíveis de genes, técnicas de agrupamento (*clustering*) como k-medianas, ou métodos aglomerativos são geralmente utilizados [19, 20]. No entanto, ressaltamos que ainda há muito espaço para melhorias nas soluções propostas por esses trabalhos.

Os algoritmos meméticos foram recentemente propostos como uma técnica para auxiliar este processo [54]. Neste capítulo relatamos outra aplicação do NP-Opt, desta vez na variante do problema de *clusterização* chamada Ordena-

[1] Este capítulo é baseado nos artigos:

C. Cotta, A. Mendes, V. Garcia, P. França e P. Moscato. **Applying Memetic Algorithms to the Analysis of Microarray Data**. Proceedings do *EvoBIO2003* - 1st *European Workshop on Evolutionary Bioinformatics*, *Lecture Notes in Computer Science* 2611:22-32, Springer-Verlag, 2003.

A. Mendes, C. Cotta, V. Garcia, P. França e P. Moscato. **Parallel Memetic Algorithms for Gene Ordering in Microarray Data**. Submetido ao *INFORMS Journal on Computing*.

mento de Genes. Ele consiste em fazer um ordenamento de alta qualidade das informações da atividade genética, de forma que genes relacionados (do ponto de vista dos seus níveis de atividade), sejam posicionados em locais próximos na sequência em questão. Este problema é motivado pela representação bidimensional normalmente utilizada pelos *microarrays*, e assim ele combina características de clusterização e de visualização. Neste trabalho, concentramos os esforços na análise da influência de várias características do algoritmo memético - como a intensidade da busca local e a configuração da população - no desempenho do método. O impacto de diferentes funções objetivo na qualidade das soluções obtidas é também analisado. Este por sinal, é um item crítico, especialmente porque a qualidade do ordenamento é um atributo relativo, fortemente conectado com o aspecto visual da solução.

8.2 Descrição do problema

Nesta seção, faremos alguns comentários sobre a natureza das informações que definem o problema. Os dados contidos em um *microarray* revelam o nível de atividade de um grupo de genes ao longo de um intervalo de tempo. A representação normalmente utilizada para esses dados é a de uma matriz de números reais. O nível de atividade é representado por tonalidades de vermelho e verde, onde o vermelho indica que a atividade genética está induzida e o verde, reprimida. O nível de atividade é proporcional à intensidade da coloração, que é traduzida então para uma escala de valores reais. Considerando apenas o problema tratado aqui, a saída de um experimento de *microarray* pode ser caracterizado como uma matriz $M = \{m_{ij}\}$, $1 \leq i \leq n$, $1 \leq j \leq m$, onde n é o número de genes sendo estudados e m é o número de experimentos por gene (correspondente às medidas sob diferentes condições ou em instantes de tempo diferentes). Desta forma, o elemento m_{ij}, um valor real, fornece uma indicação do nível de atividade do gene i na condição j.

Conforme mencionado na seção 8.1, uma boa solução para o problema de ordenamento de genes - ou seja, uma boa permutação dos genes - deve ter genes com comportamentos similares agrupados próximos, em *clusters*. Assim, uma noção de distância deve ser definida para medir o grau de similaridade entre genes. Para este trabalho, consideramos a *distância Euclidiana*, mas ela não é a única opção possível [71]. Estabelecida a medida de distância, pode-se construir uma matriz de distância entre genes, que será utilizada para medir a qualidade dos ordenamentos.

A maneira mais simples de se tentar agrupar genes semelhantes é adotando um cálculo que minimize a distância total entre genes adjacentes, de forma análoga à que se faz no *problema do caixeiro viajante* (PCV). O lado negativo desse tipo de função objetivo é que, como ela utiliza apenas informação dos genes adjacentes, acaba tendo uma visão muito míope da solução. Para um melhor agrupamento dos genes, o uso de *janelas móveis* nos parece ser mais recomendável. A distância total passa a ser calculada através de uma somatória dupla, ao invés de uma somatória simples como no caso do PCV. Na somatória dupla, o primeiro termo efetua a soma das distâncias entre o gene central da janela e todos os outros contidos na janela. A segunda somatória efetua a

soma das distâncias parciais enquanto a janela se move ao longo de toda a sequência. Para complementar a função objetivo, um termo multiplicador $w_{i,l}$ foi adicionado visando dar um peso maior para a distância entre genes mais próximos na sequência. Assim, seja $\pi = (\pi_1, \pi_2, \cdots, \pi_n)$ a ordem dos n genes em uma dada solução. A função objetivo pode ser representada pela Equação 8.1.

$$fitness(\pi) = \sum_{l=1}^{n} \sum_{i=\min(l-s_w,1)}^{\max(l+s_w,n)} w_{i,l} D[\pi_l, \pi_i] \qquad (8.1)$$

Onde $2s_w + 1$ é o tamanho da janela - composta pelos s_w genes anteriores, o gene central e os s_w genes posteriores - e $w_{i,l}$ é um valor inteiro que mede a influência do gene localizado na posição l no gene localizado na posição i. Os pesos utilizados são proporcionais a $s_w - |l - i| + 1$, ou seja, eles são lineares na distância entre os genes e normalizados para que a soma de todos os pesos em cada soma interna da Equação 8.1 seja igual a 1. Os operadores *max* e *min* são relevantes quando o gene central está localizado próximo ao início ou ao fim da sequência, onde o número de genes pertencentes à janela é menor que o normal. O uso dessa função objetivo resulta em melhores valores para soluções onde os genes estão agrupados em conjuntos grandes, com transições suaves entre eles. Na realidade, essa função de certa forma assume que todos os genes poderiam ser o centro de um *cluster*, e subsequentemente mede a similaridade dos genes próximos. A capacidade de visão mais ampla da abordagem via janelas móveis leva a crer que soluções mais robustas serão encontradas.

8.3 Representação

A representação utilizada neste problema adota algumas idéias originárias da clusterização hierárquica. As soluções são representadas por uma árvore binária em cujas folhas estão localizados os genes. Desta forma, incorpora-se às soluções uma informação extra referente ao nível de relação entre os genes. Outros tipos de representação, focados apenas na sequência em si, como por exemplo permutações simples, não permitem tal efeito. O uso da informação do nível de relação entre os genes permite criar operadores de recombinação e de mutação que geram menos ruído, evitando romper a sequência em pontos 'ruins' e mantendo genes relacionados mais próximos uns dos outros.

A forma de armazenar a árvore binária segue uma ordem inversa, também conhecida como *pre-order traversal*. Mais precisamente, o cromossomo é composto por uma *string* de inteiros no intervalo [-1, n-1], onde n é o número de genes a serem ordenados. Cada um dos genes é identificado com um número inteiro do intervalo [0, n-1], e os nós internos da árvore com o valor -1, não havendo diferença entre eles. Assim, o comprimento total de cada cromossomo será de $2n - 1$ alelos. Na Figura 8.1 é mostrado o exemplo de um cromossomo formado por seis genes e a árvore binária correspondente.

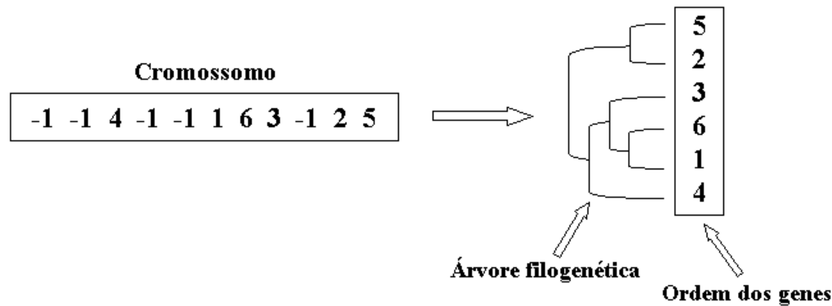


Figura 8.1: *Codificação do cromossomo para o problema de Ordenamento de Genes.*

8.4 População inicial, recombinação e mutação

A população inicial é composta por 10 indivíduos aleatórios, mais três gerados através de heurísticas de agrupamento especiais. O uso dessas heurísticas para criar soluções iniciais não é muito importante para instâncias pequenas, mas como neste trabalho trabalhamos com instâncias de mais de 500 genes, o uso desse tipo de estratégia é fundamental para facilitar a dinâmica do algoritmo memético nas gerações iniciais. Sem o uso delas, o método perde muito tempo eliminando a grande quantidade de ruído presente na população aleatória inicial. De forma simples, as heurísticas fazem uma clusterização dos genes de acordo com os seus níveis de ativação. Todas são heurísticas construtivas e por isso geram soluções de qualidade apenas mediana, mas ainda assim bem superiores às aleatórias. Os algoritmos de clusterização são o *complete-linkage*, *average-linkage*, e *single-linkage*. Para maiores informações a respeito dessas heurísticas, sugerimos verificar o trabalho de Faluso [20], que traz descrições detalhadas de todas elas.

Estando definidas a representação e a população inicial, operadores evolutivos adequados devem ser desenvolvidos para manipular a estrutura especial do cromossomo de forma a tirar proveito da informação que ela carrega. Considerando inicialmente o operador de recombinação, a abordagem é similar à utilizada no contexto de inferência filogenética apresentado no trabalho de Cotta e Moscato [13]. A seleção é igual à utilizada em todas as aplicações apresentadas neste trabalho. Inicialmente, faz-se a seleção de dois pais - um líder e um seguidor - pertencentes ao mesmo *cluster*. Em seguida, toda a informação do pai seguidor é copiada para o filho. Seleciona-se aleatoriamente uma subárvore T do indivíduo líder do *cluster* e removem-se todas as folhas da subárvore T que estejam presentes no filho, de forma a evitar a duplicação de genes. Finalmente, insere-se T em uma posição aleatória do filho. A subárvore T pode ser inserida no filho à esquerda ou à direita da ramificação escolhida. Essa escolha do tipo de inserção é feita aleatoriamente, com 50% de chance para as duas opções. Assim, o operador de *crossover* atua sempre utilizando informação das subárvores, que são estruturas bem organizadas, visando criar a menor quantidade de ruído, ou perturbação indesejada, no filho resultante. A Figura 8.2 mostra como funciona o operador.

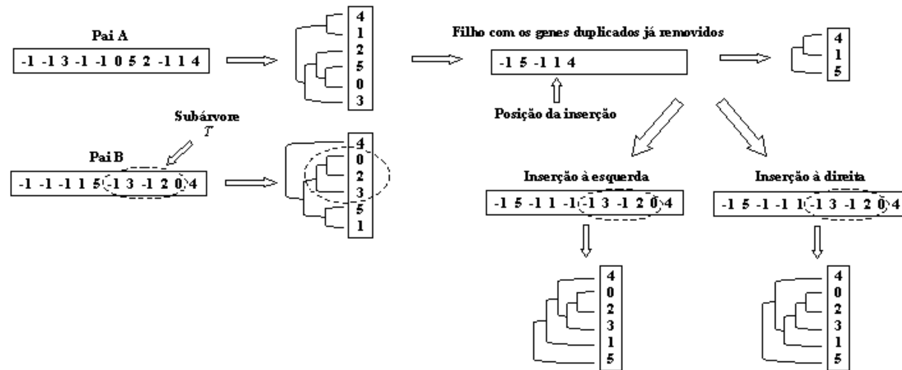


Figura 8.2: *Operador de recombinação para o Ordenamento de Genes.*

O número de indivíduos criados a cada geração foi fixado em 26, o que equivale a duas vezes o tamanho das populações - cada uma composta por 13 indivíduos. No entanto, para este problema verificamos um comportamento bastante indesejável. Após algumas gerações, notávamos sempre que o algoritmo ficava preso a uma mesma solução incumbente, realizando seguidas reinicializações nas populações, sem no entanto conseguir melhorá-la. A saída encontrada foi efetuar recombinações seguidas até que uma nova solução incumbente fosse encontrada pelos processos de recombinação e mutação apenas - ou seja, sem busca local. É claro que às vezes isso demanda muito tempo, o que nos fez também adicionar um critério de parada baseado no número de gerações, que foi fixado em 200 - isso equivale a um máximo de 5200 indivíduos criados tentando melhorar a solução incumbente. Assim, a busca local, que nos outros problemas era em geral aplicada após a convergência da população, passou a ser aplicada toda vez que uma nova solução incumbente era encontrada ou após 200 gerações terem sido concluídas. A melhora de desempenho foi notável, com o algoritmo atingindo soluções de qualidade superior às anteriores em bem menos tempo.

A adoção desse critério fez com que a definição prévia de convergência da população, adotada em todos os problemas anteriores fosse abandonada. Como o problema de Ordenamento de Genes foi o último a ser tratado, não houve tempo de se realizar testes com os problemas anteriores para ver se esse novo critério apresentaria resultados melhores também para eles. No entanto, acreditamos que não, pois os operadores de recombinação utilizados nos outros problemas são bem mais disruptivos que o utilizado aqui, gerando muito mais ruído nos filhos. Assim, é provável que seja bem mais difícil encontrar soluções incumbentes puramente através de recombinação e mutação para os problemas de Sequenciamento, Localização de Capacitores, entre outros, do que para o problema de Ordenamento de Genes. Esse fato desqualificaria, a princípio, a adoção desta nova abordagem nos problemas anteriormente tratados. No entanto, reconhecemos que esse assunto merece um estudo mais aprofundado no futuro.

A mutação, por sua vez, também utiliza informação das subárvores e se baseia na inversão das mesmas. O procedimento é bastante simples, com uma subárvore T sendo aleatoriamente selecionada e sua estrutura invertida por completo. Desta forma, tanto a sequência dos genes pertencentes a T, quanto a estrutura que indica o nível de relação entre os genes são invertidas. Essa mutação preserva o agrupamento dos genes dentro da subárvore e teremos ruído apenas nas fronteiras dela com o restante do cromossomo.

8.5 Busca local

A busca local tem um papel muito importante na implementação do algoritmo memético para este problema. O fato da complexidade do cálculo da função de *fitness* ser de $O(n.s_w)$ torna muito importante que se realize o menor número possível de avaliações nesta etapa do algoritmo. Desta forma, eliminamos de antemão quaisquer vizinhanças de busca local com complexidade $O(n^2)$. Vizinhanças do tipo troca *all-pairs* ou inserção normalmente utilizadas, inclusive em algumas aplicações do NP-Opt em outros problemas, foram descartadas. O mais lógico era tentar tirar proveito da estrutura de árvore binária utilizada na representação.

Uma possibilidade imediata era a vizinhança de *inversão de subárvores*. A sua complexidade é de $O(n)$, e ela claramente tira proveito da estrutura da solução. Na realidade, o procedimento é uma extensão do que é feito na mutação descrita anteriormente. Na mutação, uma subárvore é escolhida ao acaso e invertida. Na busca local de inversão, todas as subárvores da solução são invertidas, em sequência, e cada vez que uma inversão resulta em uma melhora da função de *fitness* ela é confirmada. Analogamente, se uma inversão piora o resultado, a subárvore volta a sua conformação anterior. Essa estratégia de busca local é muito apropriada pois permite fazer mudanças radicais no cromossomo sem que sejam perdidas informações sobre o agrupamento dos genes. Ela também é muito bem sucedida em aproximar grupos distantes que possuem genes com comportamento parecidos.

Após testes iniciais utilizando apenas essa busca local notamos que havia uma falta de suavidade nas soluções como um todo. Os genes estavam agrupados, mas o aspecto visual da solução ainda não era satisfatório. Era possível notar vários grupos de genes bem definidos, mas dentro desses grupos, o ordenamento era pobre. A idéia então foi implementar uma segunda busca local, que agisse de forma mais local, efetuando um ajuste fino na solução. A escolha recaiu na vizinhança de troca de pares adjacentes de genes, cuja complexidade, de $O(n)$, é baixa. Com isso, as soluções passaram a apresentar um aspecto mais suave e os valores de *fitness* melhoraram.

As duas buscas locais são complementares. Enquanto a busca de inversão de subárvores atua em um nível mais global da solução, a vizinhança de troca de genes adjacentes atua mais localmente, procurando ajustar pequenas falhas no ordenamento. De fato, verificamos que os grandes saltos de qualidade nas soluções ocorrem invariavelmente com aplicação da primeira busca local, enquanto que a segunda cumpre o papel a que se propunha - o de ajuste fino apenas.

As duas buscas são aplicadas em sequência. A inversão de subárvores é aplicada na solução inicial e o indivíduo resultante passa pelo ajuste de troca de genes adjacentes. Outro item importante é a determinação de quais indivíduos devem passar pela busca local. Conforme dito anteriormente, a complexidade dessa etapa é bem grande e assim, tivemos de restringir o número de indivíduos que fazem busca local. A opção de efetuar a busca em todos os novos indivíduos criados é simplesmente proibitiva do ponto de vista computacional. A opção mais lógica recaiu sobre a aplicação da busca apenas nos indivíduos que compõem a população após a sua convergência. Isso reduz o esforço computacional e garante que a busca será aplicada apenas em indivíduos de boa qualidade, pois a população terá cumprido várias gerações antes de convergir. A influência do número de indivíduos que estão sujeitos à busca local no desempenho global do algoritmo memético será estudada na seção 8.7.

8.6 Migração e processamento distribuído

O uso de migração neste problema seguiu os parâmetros definidos no Capítulo 2. Neste caso, adotamos a política de *1-Migrate* e a topologia de anel para a comunicação entre populações. Testes referentes ao número de populações que melhor se adapta ao problema são apresentados na seção 8.7, onde estão descritas as várias alternativas testadas.

Este problema foi tratado em conjunto com o professor Carlos Cotta, da Universidade de Málaga, que colocou à disposição uma rede de estações de trabalho para testarmos o algoritmo. Assim, para tirarmos o máximo proveito dessa abordagem, utilizamos uma estratégia de processamento distribuído, onde o esforço da etapa mais pesada do algoritmo memético - a busca local - foi distribuído entre as várias máquinas que compunham a rede. Essa distribuição seguiu os mesmos parâmetros também descritos no capítulo referente ao NP-Opt.

8.7 Testes computacionais

8.7.1 Instâncias utilizadas

As características das três instâncias utilizadas para avaliar o algoritmo memético estão listadas na Tabela 8.1. Elas foram extraídas de sequências reais de genes disponíveis na literatura. As dimensões variam de 106 até um máximo de 517 genes.

Os tempos computacionais adotados para as instâncias *Herpes*, *Linfoma* e *Fibroblast* foram de 30, 300 e 500 segundos, respectivamente. Os tempos de CPU definidos são aproximadamente proporcionais a n^2, devido ao custo computacional crescente das buscas locais.

8.7.2 Tamanho das janelas móveis

Os testes apresentados nessa subseção têm por objetivo avaliar a influência do tamanho da janela móvel - mais especificamente o parâmetro s_w - na quali-

Instância	Núm. de genes	Núm. de experim.	Descrição da instância
Herpes	106	21	Vírus da herpes associado ao sarcoma de Kaposi [42]
Linfoma	380	19	Linfoma difuso da célula-B [2]
Fibroblast	517	18	Resposta dos fibroblastos humanos ao serume [40]

Tabela 8.1: *Instâncias utilizadas nos testes computacionais.*

dade das soluções obtidas. Conforme dito antes, a avaliação das soluções segue critérios altamente subjetivos, pois o problema tem forte relação com sua representação gráfica. Assim, o aspecto visual deve ser utilizado como guia para medir o desempenho do método. A seguir, na Figura 8.3, mostramos testes com diferentes tamanho de janelas para a instância *Fibroblast*, onde se pode ver como eles variam consideravelmente. Para este teste, foi utilizado um computador Pentium IV de 1.7 GHz.

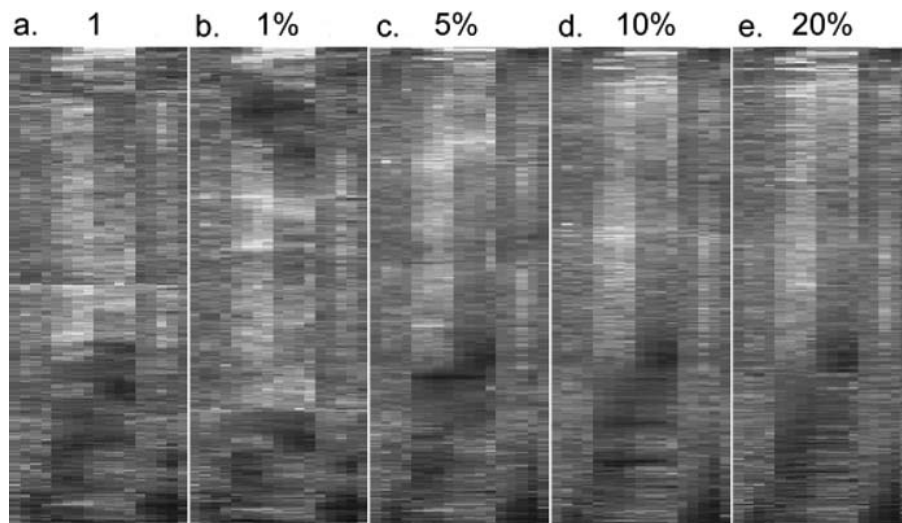


Figura 8.3: *Influência dos diferentes tamanhos de janelas na instância Fibroblast.*

Na Figura 8.3, são mostrados cinco experimentos para diferentes valores do parâmetro s_w. Da esquerda para a direita, temos 1, 1%, 5%, 10% e 20% do tamanho da instância. As diferentes tonalidades de cinza representam os vários graus de ativação dos genes. Os tons mais claros equivalem ao graus mais altos de *indução* de atividade. Os tons mais escuros, por sua vez, representam altos níveis de *repressão* da atividade genética. O exame visual dos resultados revela que uma função de *fitness* do tipo *caixeiro viajante*, vista no diagrama a, gera uma solução bastante fraca. Há uma falta geral de suavidade na figura, com excessivas alternâncias entre regiões claras e escuras. Em especial, na parte

superior da figura, temos muita mistura entre os tons, assim como na parte central. O ponto com melhor configuração é a parte inferior, onde temos uma boa concentração de genes com atividade reprimida.

No diagrama *b*, com uma janela de 1% do tamanho da instância, apesar da pouca diferença, há uma pequena melhora no aspecto geral. O ponto negativo é que há duas regiões escuras disjuntas bem definidas. A janela de 1% aparentemente foi pequena demais para evitar esse tipo de efeito. O diagrama *c* apresentou um comportamento muito bom, onde se vê uma suavidade excelente nas transições entre as tonalidades. As partes superior e inferior do diagrama apresentam as maiores perturbações. A região central está bem definida.

O diagrama *d* representa a função de *fitness* com um tamanho de janela de 10% do tamanho da instância. O aspecto é parecido com o do diagrama *c*, com a mesma instabilidade no alto e em baixo. Porém, após um exame mais cauteloso, vê-se que a parte central tem uma conformação pior que a do diagrama *c*. Por fim, no diagrama *e* começa-se a notar uma degradação da qualidade da solução. O aspecto geral é bem menos suave, o que o aproxima mais dos diagramas *a* e *b*. Tendo em vista os resultados, optamos pelo uso de um tamanho de janela de 5%, pois além de apresentar resultados gerais melhores, mantém a complexidade computacional do cálculo do *fitness* em um nível tolerável. Ressaltamos que os resultados foram semelhantes para as outras duas instâncias tratadas, mas optamos por exibir somente os resultados da maior delas.

8.7.3 Aplicação das buscas locais

Os testes seguintes foram realizados para verificar qual é a melhor estratégia para a aplicação da busca local. Dado que a população tem uma estrutura de árvore ternária composta por três níveis, inicialmente testamos em quais desses níveis deveríamos aplicar a busca local. Em segundo lugar, era necessário decidir o número de vezes que cada busca local seria aplicada em cada indivíduo - representado pelo número de passagens n_p. Uma única passagem não garante que o indivíduo resultante seja um mínimo local, porém, é justamente ela que tem o maior impacto na melhora da função objetivo. As passagens seguintes em geral continuam ocasionando melhoras, mas com uma intensidade que se reduz a cada nova passagem. Esse comportamento torna conveniente uma avaliação do grau de intensificação da busca local. As configurações testadas foram as seguintes.

- Somente da raíz (ou seja, somente no melhor indivíduo), n_p igual a 1, 4, e 13.
- Somente nos dois primeiros níveis (ou seja, nos quatro melhores indivíduos), n_p igual a 1 e 3.
- Em todos os 3 níveis (ou seja, na população inteira), n_p igual a 1.

A busca local foi testada em um total de seis configurações, sendo aplicada somente na população após a sua convergência. A convergência nesse caso seguiu os novos critérios citados na seção 8.4. O número de passagens

indica o número de vezes que as buscas locais de inversão de subárvores e de troca de genes adjacentes devem ser aplicadas sequencialmente no indivíduo, no máximo. Logicamente, se após uma passagem completa não houver melhoria do indivíduo, concluímos que ele está em um mínimo local de ambas as vizinhanças e a busca pára imediatamente. Como foi utilizado o mesmo tempo de CPU para todas as configurações, o número máximo de passagens (n_p) foi fixado de forma que, cada vez que a população converge, são executadas no máximo 13 passagens completas de busca local. Os resultados médios para 10 execuções são mostrados na Tabela 8.2.

	1 nível			2 níveis		3 níveis
Instância	$n_p = 1$	$n_p = 4$	$n_p = 13$	$n_p = 1$	$n_p = 3$	$n_p = 1$
Herpes	600,1	603,9	604,3	600,0	599,9	598,8
Linfoma	2.609,3	2.607,7	2.609,6	2.610,6	2.620,3	2.622,6
Fibroblast	1.376,8	1.386,6	1.390,3	1.382,2	1.398,1	1.407,4

Tabela 8.2: *Resultados médios para diferentes intensidades de busca local.*

A configuração com o melhor *tradeoff* entre qualidade de solução e esforço computacional foi a que aplica a busca local somente ao melhor indivíduo, fazendo uma única passagem ($n_p = 1$). Esta configuração é a que concentra o menor esforço na busca local, dando condições para que os outros operadores evolutivos desempenhem seu papel na dinâmica da população. Isso é um forte indicativo de que os outros operadores genéticos também são importantes para o problema e contribuem de maneira efetiva na busca de soluções de boa qualidade.

8.7.4 Número de populações

O último parâmetro testado foi o número de populações. Os testes incluíram ensaios com 2, 4, 6, 8 e 10 populações. Eles foram efetuados em um ambiente sequencial, e o foco foi estudar o *speedup* do algoritmo memético. Como *speedup* nos referimos a um limitante superior do *speedup* computacional teórico que poderia ser alcançado em um ambiente fisicamente distribuído. Este limitante é calculado da seguinte forma: primeiramente, um nível de *fitness* desejado é determinado. Este é o ponto que queremos que o algoritmo atinja. Em seguida todas as configurações são testadas e verifica-se quanto tempo cada uma levou para atingir este nível de *fitness* desejado. Os tempos correspondentes são então comparados normalizando-os através de sua divisão pelo número de populações utilizadas. Este valor reflete o tempo ideal que o algoritmo memético iria levar para chegar à solução desejada em um ambiente distribuído ideal, onde os tempos de comunicação entre as máquinas fossem desprezíveis. Conforme dito antes, esse método gera um limitante superior para o *speedup* atingível, pois o tempo de comunicação na maioria das vezes não é desprezível. Os resultados são mostrados na Figura 8.4. Nela, temos no eixo horizontal o número de populações, e no eixo vertical o *speedup* do algoritmo em um ambiente distribuído ideal.

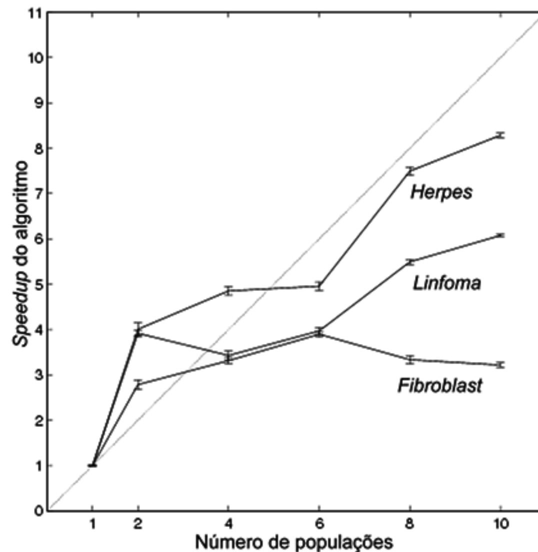


Figura 8.4: *Speedup do algoritmo memético para diferentes números de populações.*

Como pode ser visto na Figura 8.4, excelentes valores de *speedup* podem ser atingidos para até quatro populações. Depois deste ponto, o desempenho começa a deteriorar, especialmente para as instâncias maiores, como a *Fibroblast*. No entanto, valores bons continuam sendo possíveis para *Herpes*, e moderados para *Linfoma*. Esse resultado sugere que a migração possui um papel importante no processo de busca. Deve-se notar também que em alguns casos há um *speedup* super-linear, representados pelos pontos posicionados acima da diagonal. Esse é um efeito que ocorre devido ao comportamento distinto de um algoritmo multipopulacional comparado ao de um unipopulacional [1]. De fato, esse tipo de comportamento é precisamente uma das razões que dá suporte ao uso desse tipo de algoritmo.

Na próxima etapa, testamos o algoritmo memético em um novo ambiente, de processamento distribuído utilizando apenas uma população. Enquanto que os testes anteriores validaram o uso de múltiplas populações em um ambiente de processamento sequencial, os testes a seguir, por sua vez, validarão o uso do processamento distribuído em uma abordagem unipopulacional. Ele é especialmente recomendado quando tratamos com instâncias grandes, cujo custo computacional é alto demais para o uso sequencial de múltiplas populações.

8.7.5 Testes com processamento distribuído

Os testes a seguir têm por objetivo verificar o desempenho do NP-Opt em um ambiente de processamento distribuído. Para tornar o processo válido, a abordagem anterior, de efetuar apenas uma única busca local após a população ter convergido teve que ser logicamente abandonada. Para os testes em paralelo, optamos por aplicá-la em toda a população após a sua convergência, o que gera

um total de 13 buscas. Este cenário torna viável a aplicação de procedimentos de distribuição de tarefas, pois a busca local passou a consumir algumas vezes mais de 90% do tempo de CPU total, tornando-a a candidata mais apropriada para paralelização. Esta foi a abordagem adotada aqui; efetuar várias buscas locais em paralelo, distribuindo-as por uma rede de computadores.

O procedimento é bastante simples. Ele emprega uma arquitetura do tipo *mestre-escravo*, com o computador mestre controlando o algoritmo memético, designando as tarefas de busca local para as máquinas escravas e recebendo de volta os indivíduos otimizados. O procedimento é o mesmo descrito no capítulo referente ao NP-Opt.

Os testes computacionais desta fase avaliam tanto o desempenho da arquitetura mestre-escravo quanto a qualidade das soluções obtidas. Os experimentos foram realizados em uma rede de *SUN Workstations UltraSparc Iii* de 440 MHz, 256Mb de RAM e 48 Gb de HDD, comunicando-se por uma rede *Ethernet* de 100-Mbit.

Os testes incluíram, além das três instâncias anteriores, uma quarta denominada *Yeast*, composta por 979 genes - com 79 experimentos por gene - cuja função está relacionada com alguns processos complexos do ciclo celular da levedura [19]. O uso dessa instância, já com um tamanho considerável, visa tirar o máximo proveito da distribuição da busca local, que reduz o tempo de CPU a níveis mais razoáveis. O desempenho foi medido de duas formas. A primeira é o incremento da velocidade em si, em termos de *speedup*. O segundo é a qualidade da solução final obtida em termos de melhora sobre a abordagem sequencial utilizando o mesmo tempo de CPU.

Inicialmente mostramos na Tabela 8.3 as características das instâncias testadas, o tempo de CPU utilizado e a porcentagem do tempo total gasto na busca local considerando o processamento sequencial. Esse último dado é importante pois quanto maior é essa porcentagem, maior é o *speedup* teórico alcançável quando utilizamos o processamento distribuído. Esse valor pode ser calculado usando a Equação 8.2, que assume que o tempo de comunicação é desprezível e que há infinitos processadores disponíveis.

$$speedup_{max} = \frac{1}{1-p} \tag{8.2}$$

Na Equação 8.2, o parâmetro p representa a porcentagem do tempo de CPU que pode ser paralelizável, assumindo-se que as tarefas possam ser infinitamente divididas com o propósito de otimizar a distribuição. Obviamente, essa é uma estimativa fraca, mas ela fornece um primeiro limitante para os resultados alcançáveis.

O tempo de comunicação entre as máquinas é muito baixo. Para a instância maior, de 979 genes, é de menos de um segundo. Isso se deve à característica da rede e também à quantidade de dados que navega por ela. Como apenas os indivíduos são transmitidos, o maior tamanho de um pacote enviado de uma máquina para outra é de 8Kb - um indivíduo tipo *Yeast*, com 979 genes, tem um cromossomo de 1.957 posições, cada uma delas sendo um valor inteiro (32-*bit*). A seguir, na Tabela 8.4 apresentamos a relação entre o número de máquinas e o número de indivíduos que teoricamente são otimizados por cada uma delas.

Instância	**Número de genes**	**Tempo de CPU (seg.)**	**Porcentagem de busca local**	***Speedup* máx. teórico**
Herpes	106	60	66,2%	2,9
Linfoma	380	300	88,6%	8,8
Fibroblast	517	1.000	91,4%	11,7
Yeast	979	3.600	95,7%	23,4

Tabela 8.3: *Dados das instâncias utilizadas no processamento distribuído.*

O *speedup* teórico resultante é também mostrado e pode ser calculado pela Equação 8.3.

$$speedup_m = \frac{1}{1 - p + \frac{p}{w_m}} \tag{8.3}$$

Onde p representa a porcentagem do tempo computacional gasto na busca local, m é o número de máquinas, e $1/w_m$ é a máxima porcentagem de buscas locais designada para uma máquina. Este último fator deve ser levado em conta, pois a etapa de busca local não é livremente divisível. Ela só pode ser dividida indivíduo a indivíduo. Os valores reais de w_m podem ser vistos na Tabela 8.4.

Número de máquinas	**Núm. de indivíduos por máquina**	w_m
1	{13}	1,00
2	{6, 7}	1,86
3	{4, 4, 5}	2,60
4	{3, 3, 3, 4}	3,25
5	{2, 2, 3, 3, 3}	4,33
7	{1, 2, 2, 2, 2, 2, 2}	6,50
13	{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}	13,00

Tabela 8.4: *Descrição da carga de trabalho dos escravos.*

Cada configuração testada possui um número máximo diferente de indivíduos avaliados por máquina. O teste destes casos apenas, visa verificar como o algoritmo tirou vantagem da redução da carga de trabalho decorrente do aumento do número de máquinas. Os valores w_m são calculados dividindo o número máximo teórico de indivíduos avaliados em cada máquina por 13, que é o número total de buscas locais efetuadas. Deve-se considerar que o *speedup* teórico é calculado assumindo tempos de comunicação desprezíveis e, mais importante, que todas as buscas locais têm o mesmo custo computacional. Esta última consideração não é nem um pouco realista para o caso tratado aqui, e assim esperamos um desvio considerável dos valores teóricos. A Tabela 8.5 mostra os resultados finais dos *speedups* teóricos esperados para as quatro instâncias avaliadas após todas as considerações anteriores. Os valores foram obtidos pela aplicação da Equação 8.3.

A Tabela 8.5 mostra que os valores de $speedup_m$ estão bem abaixo dos valores teóricos que seriam obtidos pela Equação 8.2. Esses valores devem ser

Número de máquinas	**Instâncias**			
	Herpes	**Linfoma**	**Fibroblast**	**Yeast**
1	1,00	1,00	1,00	1,00
2	1,44	1,69	1,73	1,79
3	1,69	2,20	2,29	2,43
4	1,85	2,59	2,72	2,96
5	2,04	3,14	3,37	3,79
7	2,27	3,99	4,41	5,26
13	2,75	5,49	6,40	8,58

Tabela 8.5: *Speedups máximos teóricos para as instâncias.*

comparados com os obtidos nos experimentos reais para que se possa ter uma idéia do desempenho do método. Para tanto, observando a Figura 8.5 (Esq.) pode-se tirar algumas conclusões importantes. Inicialmente, vemos que as instâncias *Linfoma* e *Yeast* tiveram os melhores desempenhos, com resultados sempre muito próximos aos máximos teóricos. As outras duas obtiveram bons *speedups* até o limite de sete máquinas e em seguida se distanciaram demasiadamente do máximo teórico.

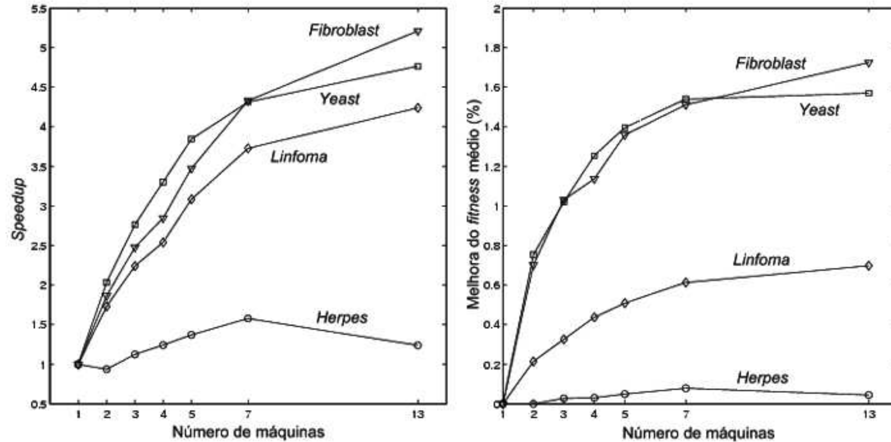


Figura 8.5: *a. Speedups resultantes; b. Melhora do fitness em comparação com a abordagem sequencial.*

Os resultados apresentados na Figura 8.5 (Esq.) são bastante ilustrativos para mostrar algumas características do funcionamento do processamento paralelo. A instância menor tem um *speedup* muito baixo por duas razões: primeiramente, a busca local é responsável apenas por uma fração do tempo de CPU total, e assim o *speedup* não pode ser mesmo muito alto (ver Tabela 8.3); em segundo lugar, o tempo gasto em cada busca local é quase o mesmo gasto na comunicação entre as máquinas. De fato, quando 13 computadores são utilizados, o *speedup* até diminui devido ao excesso de comunicação na rede.

As duas instâncias seguintes retornaram valores esperados, com o *speedup*

crescendo continuamente até a configuração final de 13 máquinas. A instância *Yeast*, por sua vez, teve um comportamento complexo. Seus resultados foram piores que os da instância *Fibroblast* para as configurações de 7 e 13 máquinas, apesar da busca local exigir uma maior porcentagem do tempo total de CPU. A razão disso foi a diferença entre os tempos gastos na aplicação da busca local nos vários indivíduos, como mencionado anteriormente. Foi verificado que em alguns casos, para a instância de 979 genes, a diferença de tempos entre duas buscas locais pode variar em até 400%, dependendo das características dos dois indivíduos iniciais. Para 7 e 13 máquinas essa diferença é muito prejudicial pois a máquina *mestre* espera até o último processo ser completado para então continuar com o algoritmo, fato esse característico do processamento 100% *síncrono*.

Essa situação é análoga à encontrada em problemas de *bin-packing*. Quando os *bins* (o tempo total utilizado pelo escravo para completar a tarefa) são grandes quando comparados aos tamanhos dos *itens* (tempos individuais para cada tarefa), soluções balanceadas são facilmente obtidas. No entanto, quando essas duas grandezas estão próximas, as soluções tendem a ser mais desbalanceadas. Mesmo assim, pode-se considerar o *speedup* como sendo bom para até sete máquinas, com uma perda de desempenho importante somente na última configuração.

O segundo critério de desempenho foi a melhora obtida ao final dos testes distribuídos em relação aos testes sequenciais. Os resultados estão na Figura 8.5 (Dir.). Apesar dos valores percentuais serem pequenos, a melhora é facilmente notada quando se analisa o aspecto visual das soluções. Além disso, os resultados finais são em torno de 10% melhores que os obtidos pelas heurísticas construtivas que geram as soluções iniciais.

Ainda sobre a Figura 8.5 (Dir.), a melhoria cresce à medida em que se aumenta o número de máquinas, exceto para o caso patológico da instância *Herpes*, que já foi explicado. O comportamento das curvas é suavemente assintótico, com poucos altos e baixos, o que é um sinal de robustez do algoritmo memético. Além disso, é um indicativo de que a rede é confiável e possui uma comunicação eficiente e rápida entre as máquinas.

8.7.6 Comparação visual das soluções

Nesta parte, mostramos o aspecto visual das soluções obtidas com o algoritmo memético. Nas Figuras 8.6, 8.7, 8.8 e 8.9 temos mais à esquerda as respectivas soluções aleatórias (marcadas com a letra a). As figuras centrais (letra b) representam os dados originais do problema como são apresentados. Esses dados originais já passaram por um processo de clusterização através de uma heurística construtiva simples. As figuras mais à direita (letra c) são as soluções obtidas pelo algoritmo memético.

A conformação visual é a forma mais intuitiva de se verificar a qualidade de uma solução. As soluções aleatórias mostradas são apenas ruído, sem nenhum tipo de agrupamento dos genes. O procedimento heurístico, por sua vez, produz soluções medianas, com vários grupos bem definidos. No entanto essas soluções tendem a ter genes semelhantes colocados em grupos distintos, ou então dois ou

mais grupos muito parecidos, que poderiam estar juntos em apenas um. Além disso, a suavidade como um todo é menor.

O algoritmo memético não teve dificuldades em alocar os genes, e além disso manteve uma transição suave entre os grupos. Não há saltos abruptos nos níveis de atividade de genes próximos. Essa característica provavelmente é resultado da função de *fitness* utilizada, que penaliza transições violentas.

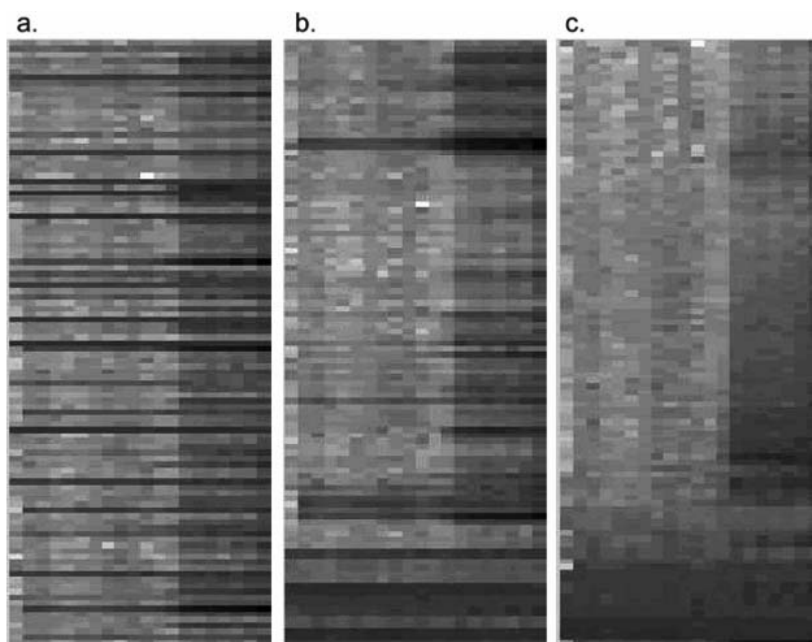


Figura 8.6: *Soluções da instância Herpes.*

8.8 Resumo

Neste capítulo abordamos o problema de Ordenamento de Genes. O objetivo é agrupar genes com comportamentos semelhantes de forma que se possam identificar grupos que sejam responsáveis por determinada característica relevante. A principal contribuição nessa parte do trabalho se concentra no uso, pela primeira vez, de processamento distribuído no NP-Opt.

O algoritmo memético utilizado é o multipopulacional padrão do NP-Opt, porém com a distribuição dos procedimentos de busca local para diversas máquinas pertencentes a uma rede local de computadores. A opção pela distribuição da tarefa de busca local se deve ao fato de que essa etapa é responsável por mais de 80% do tempo computacional total gasto nas instâncias de tamanhos médio e grande.

Para tratar do problema, utilizou-se uma representação em forma de árvore para o ordenamento, procurando tirar proveito das relações de proximidade entre os vários genes. Essa abordagem guarda estreita relação com o conceito

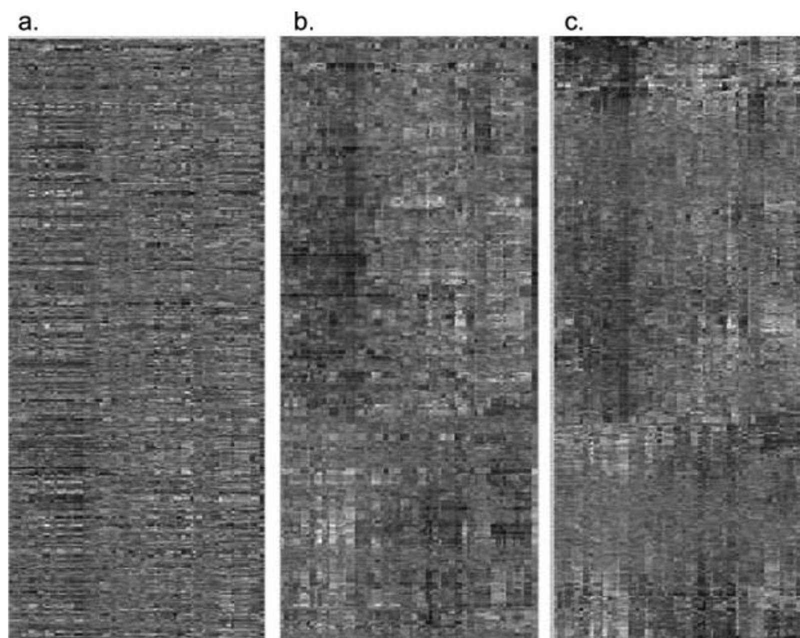


Figura 8.7: *Soluções da instância Linfoma.*

de árvore filogenética, que fornece informação a respeito de hierarquia entre espécies. Devido a essa representação especial, operadores de recombinação e de busca local *ad hoc* foram desenvolvidos. A recombinação utiliza uma estratégia de transferência de ramos, ao passo que a busca local utiliza vizinhanças de inversão de ramos e de troca de genes adjacentes.

Os testes foram realizados em quatro instâncias criadas a partir de genomas reais, disponíveis em *websites* da área. Os tamanhos variam entre 106 e 979 genes. Os estudos se concentraram na influência do número de populações e do número de máquinas no desempenho do algoritmo, tanto em termos de *speedup* quanto de melhora do fitness do melhor indivíduo.

Foram realizados também alguns testes quanto à melhor função de *fitness* a ser adotada. Na tarefa de agrupar genes, existem várias maneiras de se medir o grau de similaridade entre os mesmos. Para tanto, definimos uma 'janela' de medição, cujo tamanho varia de acordo com o tamanho da instância. Essa abordagem foi a que obteve melhores resultados. Essa parte do trabalho exige um certo grau de abstração do leitor, pois a comparação da qualidade das soluções leva em conta tão somente o aspecto visual delas.

Na parte final, os melhores resultados do algoritmo memético foram comparados com uma solução aleatória e uma solução obtida por uma heurística construtiva de grupamento. Essa comparação é de novo puramente visual, mas a diferença de qualidade entre as soluções é facilmente percebida.

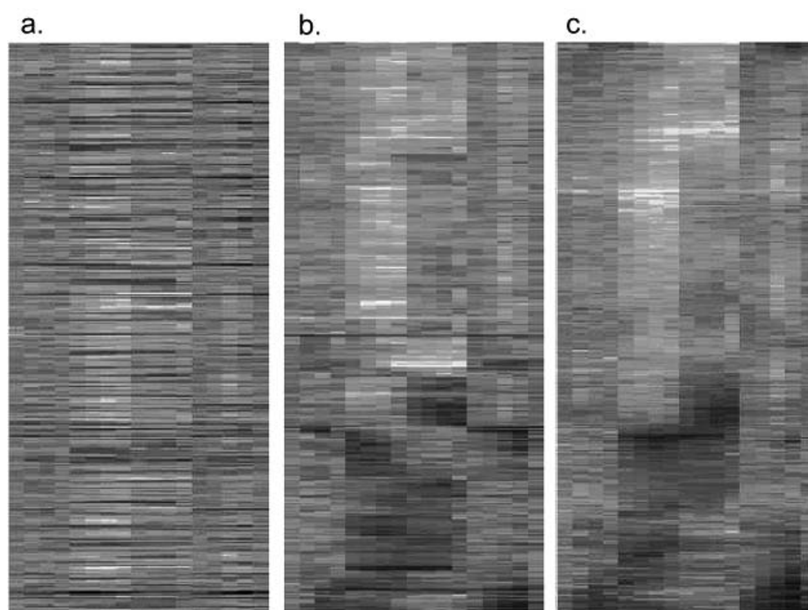


Figura 8.8: *Soluções da instância Fibroblast.*

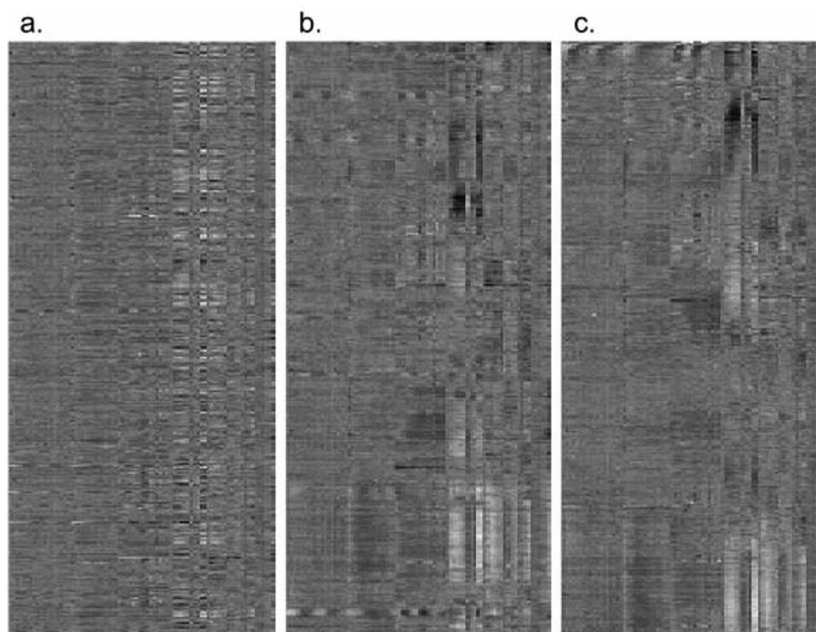


Figura 8.9: *Soluções da instância Yeast.*

Capítulo 9

Conclusão

Este trabalho se concentrou no desenvolvimento de um ambiente de otimização chamado NP-Opt, cuja ferramenta de busca é baseada em um algoritmo memético. Os algoritmos meméticos são uma classe de metaheurística que pertence à classe maior dos algoritmos evolutivos, e podem ser considerados uma extensão dos conhecidos algoritmos genéticos. Muitas vezes chamados algoritmos genéticos híbridos, os algoritmos meméticos foram caracterizados como tal no fim dos anos 80. No software NP-Opt, a diferença entre um algoritmo genético e a sua 'versão memética' reside na utilização ou não de estratégias de busca local especializadas. De fato, boa parte das contribuições deste trabalho está justamente nessas buscas locais, ou então em estratégias de redução de vizinhança, que reduzem o esforço computacional concentrando a busca apenas nos movimentos ditos promissores.

O algoritmo memético que move o NP-Opt conta com uma série de características que tornam o seu desempenho bastante superior ao de outros métodos. Entre elas podemos citar a adoção de populações hierarquicamente estruturadas. Essa abordagem cria, em uma única população, uma dinâmica que em geral só é encontrada quando várias populações são utilizadas. A estrutura populacional divide a população em subgrupos - denominados *clusters* - e restringe a seleção dos indivíduos para recombinação e a inserção dos novos indivíduos. Outra característica importante é a política altamente restritiva de aceitação dos novos indivíduos que, em conjunto com a estratégia de reinicialização da população, acelera o processo evolutivo, gerando rapidamente soluções de alta qualidade sem cair na armadilha da convergência prematura.

O NP-Opt conta também com a possibilidade de evoluir várias populações simultaneamente, efetuando trocas de indivíduos entre elas esporadicamente, segundo um *modelo de ilhas* tradicional. Com isso, tira-se vantagem do efeito da 'deriva genética', ou *genetic drift*, que reduz a possibilidade do método ficar preso de forma prematura a uma só região do espaço de soluções e deixar assim escapar a solução ótima. Comparações entre abordagens unipopulacionais e multipopulacionais mostraram uma clara vantagem para esta última, que sistematicamente obtém soluções melhores utilizando pouco tempo de CPU.

Por fim, temos ainda alguns testes com processamento distribuído, onde a

tarefa de busca local - de longe a mais custosa do ponto de vista computacional - foi distribuída por uma rede de computadores. Esses testes só puderam ser realizados com o problema de Ordenamento de Genes, pois essa foi uma das últimas alterações feitas no NP-Opt e assim os problemas anteriores não puderam usufruir dela. A distribuição da busca local segue um modelo *mestre-escravo* tradicional, com uma máquina mestre controlando o algoritmo memético e enviando as tarefas para as máquinas escravas, que reenviam os indivíduos otimizados de volta.

Os problemas tratados nesta tese foram cinco: Localização de Capacitores, *Gate Matrix Layout*, Sequenciamento em *Flowshop*, Sequenciamento em Máquina Simples e Ordenamento de Genes. A adaptação do algoritmo memético geral para o problema de Localização de Capacitores foi bem sucedida, sendo que as principais contribuições aconteceram na forma de representação das soluções e na estratégia de busca local, que se mostrou eficiente o bastante para tratar de problemas de grande porte - compostos por milhares de seções. Até então, só era possível encontrar na literatura problemas pequenos ou médios, de no máximo algumas centenas de seções. Há diversas possibilidades de pesquisas futuras nesse campo, como por exemplo a inclusão de capacitores automáticos, com capacidade variável e ainda analisar o comportamento da rede com vários perfis de carga, o que aproximaria o modelo das situações reais normalmente encontradas.

O problema de *Gate Matrix Layout* consiste em determinar a configuração em que as portas que compõem um circuito elétrico devem ser ordenadas de forma a minimizar o tamanho do mesmo. Para esse problema aplicamos o algoritmo memético multipopulacional, uma vez que os resultados dos métodos anteriores já eram bastante fortes e a abordagem unipopulacional se mostrou pouco eficiente frente a eles. A principal contribuição aqui foi de novo a busca local, mais especificamente a redução de vizinhança que utiliza informação a respeito das chamadas *portas críticas*. Com a sua utilização, o esforço computacional foi significativamente reduzido, fazendo o algoritmo memético atingir valores idênticos aos previamente disponíveis em um tempo de CPU bem menor. A redução média no tempo computacional é superior a 80%. Como desenvolvimentos futuros, uma escolha promissora talvez seja testar novas vizinhanças de busca local, uma vez que a influência dessas é muito grande. Novas políticas de redução de vizinhança também parecem ser um campo interessante, pois dado que a avaliação de uma solução é um procedimento muito custoso, deve-se ter sempre em mente evitar testes desnecessários.

O problema de Sequenciamento em *Flowshop* foi talvez o mais simples de ser tratado. Grande parte dessa facilidade se deve à representação adotada. A divisão do cromossomo em partes distintas, não relacionadas diretamente, reduziu em muito a complexidade da etapa de busca local. O algoritmo memético conseguiu tratar de forma eficiente problemas de até 10 famílias de tarefas e 10 máquinas, sendo que ao final dos testes, ficamos com a impressão que instâncias bem maiores ainda poderiam ter sido testadas com sucesso. Desenvolvimentos futuros para esse problema só seriam necessários se primeiro fossem disponibilizadas instâncias bem maiores, onde cada parte do cromossomo fosse composta por talvez 30 ou 40 alelos. Isso caracterizaria problemas de porte acima do nor-

mal, com centenas de tarefas e com dezenas de famílias e de máquinas. Nesse caso, reduções de vizinhança seriam imprescindíveis e abririam campo para pesquisas futuras.

No problema de Sequenciamento em Máquina Simples a contribuição mais forte foi a elaboração de um procedimento para gerar instâncias a partir do problema do Caixeiro Viajante Assimétrico. Esse procedimento permite que partindo de uma instância do PCVA cuja solução ótima seja conhecida, obtenha-se uma instância para o problema de SMS também com solução ótima conhecida. Como ainda não existe prova matemática de que essa transformação esteja correta, podemos considerar os ótimos presumidos apenas como limitantes superiores. No entanto, ressaltamos que para todas as instâncias criadas dessa forma, não foi encontrada durante os testes nenhuma solução melhor que as ótimas presumidas. Isso nos leva a crer que, caso o procedimento não seja correto, ao menos ele é capaz de gerar limitantes superiores de alta qualidade. A outra contribuição forte reside na aplicação do método multipopulacional, que levou a resultados muito bons. De fato, a taxa de acerto do método beira os 90%, sendo que mesmo para instâncias consideradas grandes, de 100 tarefas, foi possível atingir várias soluções ótimas.

O último problema tratado foi o de Ordenamento de Genes. Esse problema pertence à área de Bioinformática, tendo aplicações em importantes ramos do conhecimento como Biologia, Bioquímica e Medicina. O objetivo é agrupar genes de acordo com seus comportamentos, de forma que genes que atuam de forma similar fiquem próximos uns dos outros. Para lidar com esse problema utilizamos todo o potencial do NP-Opt; múltiplas populações, buscas locais especializadas, processamento paralelo, entre outros. A representação em forma de árvore necessitou de um conjunto de operadores especiais de recombinação, busca local e mutação, que pudessem tirar proveito das características presentes nessa estrutura. As instâncias, criadas a partir de dados extraídos de genomas reais, variaram desde 106 até 979 genes, o que caracteriza instâncias de tamanho pequeno a médio. Para genomas maiores, da ordem de milhares de genes, novas estruturas de dados seriam necessárias, tendo em vista principalmente algumas limitações de memória da maioria dos sistemas computacionais e também da própria linguagem Java. Os resultados do algoritmo memético foram muito bons, tendo sido possível agrupar os genes de forma satisfatória, quando comparados a outros métodos mais simples. Devido à pouca idade dessa área como um todo, acreditamos que aqui se concentrem as melhores possibilidades de trabalhos futuros, tanto em termos de resultados computacionais quanto de impacto na comunidade científica. O fato do campo da bioinformática contar com projetos como o Genoma Humano e de sequenciamento de diferentes tipos de vírus, pragas agrícolas, entre outros, aumenta a quantidade de dados reais ditos 'crus', de onde se deseja extrair a maior quantidade possível de informação. O campo de aplicação para as metaheurísticas é simplesmente enorme.

Ao fim deste trabalho temos como produto final um software de código livre, capaz de lidar com uma ampla gama de problemas e que apresenta um bom desempenho geral. A modularidade do programa permite que funções específicas sejam extraídas e incorporadas a outros softwares, transformando o NP-Opt

também em um biblioteca de procedimentos. Algumas características importantes foram validadas, como o uso de pequenas populações hierarquicamente estruturadas, com métodos de seleção e inserção próprios. O uso de buscas locais também se mostrou primordial. Acreditamos que absolutamente nenhum dos problemas aqui tratados teria tido sucesso caso tivéssemos adotado um algoritmo genético puro. E muitas vezes buscas locais consideravelmente complexas tiveram que ser desenvolvidas para atingirmos um nível de desempenho satisfatório.

Outra característica importante foi o uso de operadores de recombinação relativamente simples, e com alto grau de aleatoriedade. Com isso, aumenta-se a capacidade de diversificação do operador, deixando a intensificação a cargo da busca local. Muitas vezes deixou-se a população evoluir por longos períodos, restrita à fase de recombinação, antes de lançar mão da busca local, já sobre uma população de alta qualidade, composta por indivíduos promissores. Isso muitas vezes aumentou a eficiência do método, reduzindo os efeitos da convergência prematura.

Da forma como está, o NP-Opt pode ser utilizado como ferramenta de testes tendo em vista a facilidade para se realizar comparações entre diversos procedimentos simultaneamente. Testar uma nova recombinação, busca local, mutação, estratégia de seleção etc., é muito simples. A inclusão de novos problemas é uma tarefa bem mais árdua, porém bem menos do que começar a programar um algoritmo memético partindo do zero. Usos pedagógicos do NP-Opt também são claros. Efeitos como convergência prematura, aumento ou decréscimo de taxas de recombinação e/ou mutação podem ser facilmente notados e, apesar de dispensável, a interface gráfica torna a visualização de tais efeitos mais agradável.

Trabalhos futuros em termos do software NP-Opt em si podem se concentrar na parte de distribuição de tarefas. Em especial, vimos que a distribuição somente da busca local possui algumas restrições e em alguns casos pode se revelar pouco vantajosa. Uma abordagem com maior chance de sucesso seria a distribuição de populações inteiras. Com isso, o uso de múltiplas populações seria algo automático, o que por si só é um avanço. Ao enviar cada população para uma máquina distinta, teríamos um alto grau de independência na rede, associado a uma assincronicidade natural, o que deverá aumentar a eficiência de todo o sistema. Eventuais perdas de conexão entre máquinas teriam pouca relevância, dado que haveria sempre populações inteiras evoluindo continuamente e trocando indivíduos entre si, de forma descentralizada. Este é a principal melhoria que podemos vislumbrar para o NP-Opt no curto prazo. É claro que melhorias pontuais em um procedimento específico de um problema também são possíveis, mas do ponto de vista geral, a distribuição de populações inteiras nos parece ser a escolha mais promissora.

Apêndice A

Artigos publicados ou submetidos

1. P. França, J. Gupta, A. Mendes, P. Moscato e K. Veltink. **Metaheuristic Approaches for the Pure Flowshop Manufacturing Cell Problem**. Proceedings do *PMS2000 - 7th International Workshop On Project Management and Scheduling*, pág. 128-130, Osnabrück, Alemanha, Abril, 2000.

2. A. Mendes, P. França e P. Moscato. **Fuzzy-Evolutionary Algorithms Applied to Scheduling Problems**. Proceedings do *POM2000 - 1st World Conference on Production and Operations Management*, pág. 1-10, Sevilha, Espanha, Agosto, 2000.

3. A. Silva, T. Ohishi, A. Mendes, F. França e E. Delgado. **Using Genetic Algorithm and Simplex Method to Stabilize an Oil Treatment Plant Inlet Flow**. Proceedings do *IPC2000 - International Pipeline Conference*, pág. 1459-1466, Calgary, Canadá, Outubro, 2000.

4. A. Mendes, P. França e P. Moscato. **NPOpt: An Optimization Framework for NP Problems**. Proceedings do *POM2001 - International Conference of the Production and Operations Management Society*, pág. 82-89, Guarujá, Brasil, Agosto, 2001.

5. V. Garcia, A. Mendes, P. França e P. Moscato. **Algoritmo Memético Paralelo Aplicado a Problemas de Sequenciamento em Máquina Simples**. Proceedings do *XXXIII SOBRAPO - Simpósio Brasileiro de Pesquisa Operacional*, pág. 971-981, Campos do Jordão, Brasil, Novembro, 2001.

6. A. Mendes, P. França e P. Moscato. **Fitness Landscapes for the Total Tardiness Single Machine Scheduling Problem**. *Neural Network World - International Journal on Neural and Mass-Parallel Computing and Information Systems*, 2(2):165-180, 2002.

7. A. Mendes, P. França, P. Moscato e V. Garcia. **Population Studies for the Gate Matrix Layout Problem**. Proceedings do *IBERAMIA2002 - 8th IberoAmerican Conference on Artificial Intelligence*, *Lecture Notes in Artificial Intelligence* 2527:319-328, Springer-Verlag, Málaga, Espanha, 2002.

8. A. Mendes, P. França, C. Lyra, C. Pissarra e C. Cavelucci. **An Evolutionary Approach for Capacitor Placement in Distribution Networks**. Proceedings (CD-ROM apenas - 6 páginas) do *EIS2002 - 3rd International NAISO Symposium on Engineering of Intelligent Systems*, Málaga, Espanha, Setembro, 2002.

9. A. Mendes, J. Amorim e R. Miskulin. **Connecting Mathematics and Biology in the Information Society Schools: A Brazilian Perspective on Technology Usage**. Proceedings do *MEC2002 - International Conference on The Humanistic Renaissance in Mathematics Education*, pág. 258-262, Palermo, Itália, Setembro, 2002.

10. A. Mendes, C. Cotta, V. Garcia, P. França e P. Moscato. **Parallel Memetic Algorithms for Gene Ordering in Microarray Data**. Submetido ao *INFORMS Journal on Computing* em Junho de 2003.

11. C. Cotta, A. Mendes, V. Garcia, P. França e P. Moscato. **Applying Memetic Algorithms to the Analysis of Microarray Data**. Proceedings do *EvoBIO2003 - 1st European Workshop on Evolutionary Bioinformatics*, *Lecture Notes in Computer Science* 2611:22-32, Springer-Verlag, Colchester, Inglaterra, Abril, 2003.

12. A. Mendes, P. França, C. Lyra, C. Pissarra e C. Cavelucci. **A New Method for Capacitor Placement in Large-Sized Distribution Networks**. Submetido ao *IEEE Transactions on Power Systems* em Junho de 2003.

13. A. Mendes e A. Linhares. **Gate Matrix Layout: a Multiple Population Evolutionary Approach**. Submetido ao *International Journal of Systems Science* em Julho de 2003.

14. P. França, J. Gupta, A. Mendes, P. Moscato e K. Veltink. **Evolutionary Algorithms for Flowshop Scheduling with Family Setups**. Aceito para publicação na *Computers and Industrial Engineering*, 2003.

Bibliografia

[1] E. Alba. Parallel evolutionary algorithms can achieve super-linear performance. *Information Processing Letters*, 82(1):7–13, 2002.

[2] A.A. Alizadeh et al. Distinct types of diffuse large b-cell lymphoma identified by gene expression profiling. *Nature*, 403:503–511, 2001.

[3] A. Allahverdi, J.N.D. Gupta, and T. Aldowaisan. A survey of scheduling research involving setup considerations. *OMEGA - International Journal of Management Science*, 27(2):219–239, 1999.

[4] A. Arnone and B. Davidson. The hardwiring of development: Organization and function of genomic regulatory systems. *Development*, 124:1851–1864, 1997.

[5] S. Baluja. A massively distributed parallel genetic algorithm. Technical Report CMU-CS-92-196R, Carnegie Mellon University, 1992.

[6] M.E. Baran and F.F. Wu. Optimal capacitor placement on radial distribution systems. *IEEE Transactions on Power Delivery*, 4(1):725–734, 1989.

[7] M.E. Baran and F.F. Wu. Optimal sizing of capacitors placed on a radial distribution systems. *IEEE Transactions on Power Delivery*, 4(1):735–743, 1989.

[8] P.O. Brown and D. Botstein. Exploring the new world of the genome with DNA microarrays. *Nature Genetics*, 21:33–37, 1999.

[9] E. Cantú-Paz. A survey of parallel genetic algorithms. Technical Report 97003, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, 1997.

[10] E. Cantú-Paz. Topologies, migration rates, and multi–population parallel genetic algorithms. Technical Report 97007, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, 1999.

[11] E. Cantú-Paz and D.E. Goldberg. Efficient parallel genetic algorithms: theory and practice. *Computer Methods in Applied Mechanics and Engineering*, 186:221–238, 2000.

[12] T.C.E. Cheng, J.N.D. Gupta, and G. Wang. A review of flowshop scheduling research with setup times. *Production and Operations Management*, 9(3):283–302, 2000.

[13] C. Cotta and P. Moscato. Inferring phylogenetic trees using evolutionary algorithms. In *Proceedings of the PPSN'02 - VII Parallel Problem Solving From Nature, Lecture Notes in Computer Science n.2439*, pages 720–729. Springer-Verlag, 2002.

[14] C.R. Darwin. *The origin of the species*. New York: Random House, 1993.

[15] R. Dawkins. *The selfish gene*. Oxford University Press, 1976.

[16] J.L. DeRisi, V.R. Lyer, and P.O. Brown. Exploring the metabolic and genetic control of gene expression on a genomic scale. *Science*, 278:680–686, 1997.

[17] R.G. Downey and M.R. Fellows. Fixed parameter tractability and completeness 1. basic results. *SIAM Journal on Computing*, 24(4):873–921, 1995.

[18] J. Du and J.Y.T. Leung. Minimizing total tardiness on one machine is NP–hard. *Mathematics of Operations Research*, 15:483–495, 1990.

[19] M.B. Eisen, P.T. Spellman, P.O. Brown, and D. Botstein. Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Sciences of the USA*, 95:14863–14868, 1998.

[20] D. Fasulo. An analysis of recent work on clustering algorithms. Technical Report UW-CSEO1-03-02, University of Washington, 1999.

[21] M.R. Fellows and M.A. Langston. Non-constructive advances in polynomial-time complexity. *Information Processing Letters*, 26:157–162, 1987.

[22] T.A. Feo and M.C.G. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[23] P.M. França, A.S. Mendes, and P. Moscato. A memetic algorithm for the total tardiness single machine scheduling problem. *European Journal of Operational Research*, 132(1):224–242, 2001.

[24] R.A. Gallego, A.J. Monticelli, and R. Romero. Optimal capacitor placement in radial distribution networks. *IEEE Transactions on Power Systems*, 16(4):630–637, 2001.

[25] M. Gen and R. Cheng. *Genetic Algorithms and Engineering Design*. New York: John Wiley & Sons, 1997.

[26] T. Ghose, S.K. Goswami, and S.K. Basu. Solving capacitor placement problems in distribution systems using genetic algorithms. *Electric Machines and Power Systems*, 27:429–441, 1999.

[27] F. Glover. Scatter search and star-paths - beyond the genetic metaphor. *OR Spektrum*, 17(2–3):125–137, 1995.

[28] F. Glover and M. Laguna. *Tabu Search*. Kluwer, 1997.

[29] D.E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley, 1989.

[30] V.S. Gordon and D. Whitley. Serial and parallel genetic algorithms as function optimizers. In *Proceedings of the ICGA'93 - 5th International Conference on Genetic Algorithms*, pages 177–183, 1993.

[31] M. Gorges-Schleuter. Asparagos96 and the traveling salesman problem. In *Proceedings of the 4th International Conference on Evolutionary Computation*, pages 171–174, 1997.

[32] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G Rinooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[33] S.C. Graves. A review of production scheduling. *Operations Research*, 29(4):646–675, 1981.

[34] J.N.D Gupta and W.P. Darrow. The two–machine sequence dependent flowshop scheduling problem. *European Journal of Operational Research*, 24(3):439–446, 1986.

[35] Chiang H.D., J.C. Wang, O. Cokings, and H.D. Shin. Optimal capacitor placement in distribution systems - part II: solution algorithms and numerical results. *IEEE Transactions on Power Delivery*, 5(2):643–649, 1990.

[36] K. Hitomi, N. Nakamura, T. Yoshida, and K. Okuda. An experimental investigation of group production scheduling. In *Proceedings of the 4th International Conference on Production Research*, pages 608–617, 1977.

[37] J. Holland. *Adaptation in natural and artificial systems*. The University of Michigan Press, 1975.

[38] Y.S. Hong, K.H. Park, and M. Kim. A heuristic for ordering the columns in one-dimensional logic array. *IEEE Transactions on Computer-Aided Design*, 8:547–562, 1989.

[39] Y.C. Huang, H.T. Yang, and C.L. Huang. Solving the capacitor placement problem in a radial distribution system using tabu search approach. *IEEE Transactions on Power Systems*, 11(4):1868–1873, 1996.

[40] V.R. Iyer et al. The transcriptional program in the response of human fibroblasts to serum. *Science*, 283:83–87, 1999.

[41] Java Sun. Web site. http://java.sun.com.

[42] R.G. Jenner, M.M. Alba, C. Boshoff, and P. Kellam. Kaposi's sarcoma-associated herpesvirus latent and lytic gene expression as revealed by DNA arrays. *Journal of Virology*, 75(2):891–902, 2001.

[43] E.V. Koonin. The emerging paradigm and open problems in comparative genomics. *Bioinformatics*, 15:265–266, 1999.

[44] E.L. Lawler, J.K Lenstra, A.H.G. Rinooy Kan, and D.B. Shmoys. *Handbooks in Operations Research and Management Science - Vol. 4*, chapter Sequencing and Scheduling: Algorithms and Complexity, pages 445–522. North-Holland, 1993.

[45] Y.H. Lee, K. Bhaskaran, and M. Pinedo. A heuristic to minimize the total weighted tardiness with sequence–dependent setups. *IIE Transactions*, 29:45–52, 1997.

[46] T. Lengauer. *Combinatorial algorithms for integrated circuit layout*. New York: John Wiley & Sons, 1990.

[47] D. Levine. A parallel genetic algorithm for the set partitioning problem. Technical Report ANL-94/23, Illinois Institute of Technology, 1994.

[48] G. Levitin, Kalyuzhny A., A. Shenkman, and M. Chertkov. Optimal capacitor allocation in distribution systems using a genetic algorithm and a fast energy loss computation technique. *IEEE Transactions on Power Delivery*, 15(2):623–628, 2000.

[49] A. Linhares. Synthesizing a predatory search strategy for VLSI layouts. *IEEE Transactions on Evolutionary Computation*, 3(2):147–152, 1999.

[50] A. Linhares, H. Yanasse, and J. Torreão. Linear gate assignment: a fast statistical mechanics approach. *IEEE Transactions on Computer–Aided Design on Integrated Circuits and Systems*, 18(12):1750–1758, 1999.

[51] A. Linhares and H.H. Yanasse. Connections between cutting-pattern sequencing, VLSI design, and flexible machines. *Computers & Operations Research*, 29(12):1759–1772, 2002.

[52] A.D. Lopez and H. S. Law. A dense gate matrix layout method for MOS VLSI. *IEEE Transactions on Electron Devices*, 27(8):1671–1675, 1980.

[53] A.S. Mendes. Algoritmos meméticos aplicados aos problemas de sequenciamento em máquinas. *Tese de Mestrado, Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas - UNICAMP - Brasil*, 1999.

[54] P. Merz. Clustering gene expression profiles with memetic algorithms. In *Proceedings of the PPSN'02 - VII Parallel Problem Solving From Nature, Lecture Notes in Computer Science n.2439*, pages 811–820. Springer-Verlag, 2002.

[55] P. Merz and B. Freisleben. *New Ideas in Optimization*, chapter Fitness landscapes and memetic algorithm design, pages 245–260. McGraw-Hill, 1999.

[56] R. Möhring. *Computational Graph Theory - Computing Supplement 7*, chapter Graph problems related to gate matrix layout and PLA folding, pages 17–51. Springer-Verlag, 1990.

[57] K.N. Miu, H.D. Chiang, and G. Darling. Capacitor placement, replacement and control in large-scale distribution systems by a ga-based two-stage algorithm. *IEEE Transactions on Power Systems*, 12(3):1160–1166, 1997.

[58] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: towards memetic algorithms. Technical Report C3P 826, Caltech Concurrent Computation Program, 1989.

[59] K. Nakatani, T. Fujii, T. Kikuno, and N. Yoshida. A heuristic algorithm for gate matrix layout. In *Proceedings of the International Conference of Computer-Aided Design*, pages 324–327, 1986.

[60] H.N. Ng, M.M.A. Salama, and A.Y. Chikhami. Classification of capacitor allocation techniques. *IEEE Transactions on Power Delivery*, 15(1):387–392, 2000.

[61] T. Ohtsuki, H. Mori, E.S. Kuh, T. Kashiwabara, and T. Fujisawa. One-dimensional logic gate assignment and interval graphs. *IEEE Transactions on Circuits and Systems*, 26(9):675–683, 1979.

[62] P.S. Ow and T.E. Morton. The single machine early/tardy problem. *Management Science*, 35:177–191, 1989.

[63] G.L. Ragatz. A branch–and–bound method for minimum tardiness sequencing on a single processor with sequence dependent setup times. In *Proceedings of the 24th Annual Meeting of the Decision Sciences Institute*, pages 1375–1377, 1993.

[64] N. Raman, R.V. Rachamadugu, and F.B. Talbot. Real time scheduling of an automated manufacturing center. *European Journal of Operational Research*, 40:222–242, 1989.

[65] P.A. Rubin and G.L. Ragatz. Scheduling in a sequence dependent setup environment with genetic search. *Computers & Operations Research*, 22(1):85–99, 1995.

[66] J.E. Schaller, J.N.D Gupta, and A.J. Vakharia. Scheduling a flowline manufacturing cell with sequence dependent family setup times. *European Journal of Operational Research*, 125(1):324–339, 2000.

[67] S. Sundhararajan and A. Pahwa. Optimal selection of capacitors for radial distribution systems using a genetic algorithm. *IEEE Transactions on Power Systems*, 9(3):1499–1507, 1994.

[68] G. Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9, San Mateo, CA, 1989. Morgan Kaufmann.

[69] G. Syswerda. *Handbook of Genetic Algorithms*, chapter Schedule optimization using genetic algorithms, pages 332–349. New York: Van Nostrand Reinhold, 1991.

[70] K.C. Tan and R. Narasimhan. Minimizing tardiness on a single processor with sequence–dependent setup times: a simulated annealing approach. *OMEGA - International Journal of Management Science*, 25(6):619–634, 1997.

[71] H.-K. Tsai, J.-M. Yang, and C.-Y. Kao. Applying genetic algorithms to finding the optimal gene order in displaying the microarray data. In *Proceedings of the GECCO2002 - Genetic and Evolutionary Computation Conference*, 2002.

[72] TSPLIB. Web site. http://www.crpc.rice.edu/softlib/tsplib/.

[73] J. Weiner. *The beak of the finch.* New York: Vintage Books, 1995.

[74] O. Wing, S. Huang, and R. Wang. Gate matrix layout. *IEEE Transactions on Computer-Aided Design*, 4:220–231, 1985.

[75] H.H. Yanasse. On a pattern–sequencing problem to minimize the number of open stacks. *European Journal of Operational Research*, 100(3):454–463, 1997.