Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Departamento de Engenharia de Sistemas

Algoritmos Meméticos Paralelos aplicados a Problemas de Otimização Combinatória

Vinícius Jacques Garcia

Março de 2002

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Engenharia Elétrica.

Área de Concentração: Automação.

Banca Examinadora:

- Paulo Morelato França (Orientador)
- Marco Aurélio Amaral Henriques - DCA - FEEC - UNICAMP
- Felipe Martins Müller - DELC - CT - UFSM
- Pablo Moscato - UNICAMP

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

Garcia, Vinícius Jacques
G165a Algoritmos meméticos paralelos aplicados a problemas de otimização combinatória / Vinícius Jacques Garcia. -- Campinas, SP: [s.n.], 2002.

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

1. Algoritmos genéticos. 2. Programação paralela (Computação). 3. Planejamento da produção. 4. Problema do caixeiro viajante. I. França, Paulo Morelato. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Resumo e Abstract

Resumo

Este trabalho propõe um modelo de algoritmo memético paralelo aplicado à resolução dos problemas de seqüenciamento em máquinas simples e máquinas paralelas e do problema do caixeiro viajante. A estrutura geral dos algoritmos evolutivos, dos algoritmos meméticos, modelos clássicos de algoritmos evolutivos paralelos e aspectos teóricos e práticos da implementação adotada também são abordados. O desempenho do algoritmo proposto foi avaliado através de um conjunto de instâncias para cada problema.

Abstract

This work proposes a model of parallel memetic algorithm applied to solve the single machine scheduling, the parallel machine scheduling and the traveling salesman problem. The main structure of the evolutionary algorithms, memetic algorithms, classical models of parallel evolutionary algorithms as well as the theory and practical issues concern the implementation used are also discussed. The efficiency of the parallel approach was measured using a set of instances for each problem.

Dedicatória

Aos meus pais, meus irmãos e à memória da minha sempre querida Marulina.
A minha noiva Vanessa, meu amor e meu carinho.
A Deus Pai Todo Poderoso.

Agradecimentos

Primeiramente gostaria de agradecer ao professor Paulo Morelato França, pela agradabilíssima convivência, pelo aprendizado, pela amizade e pela confiança que em mim depositou.

Aos amigos e companheiros Alexandre, Pablo e Luciana, pelas sugestões e pelo apoio durante os dois anos que desenvolvi este trabalho.

A todos os amigos do DENSIS e da FEEC pelo aprendizado, pela companhia e pelos momentos de descontração.

Não poderia deixar de manifestar o meu reconhecimento ao amigo e professor Felipe Martins Müller. Obrigado Felipão pela confiança e pelo incentivo.

Também sou muito grato a PET (Programa Especial de Treinamento), que influenciou minha decisão de optar por um curso de pós-graduação.

Quanto aos recursos computacionais necessários para o desenvolvimento deste trabalho, agradeço a Eder Nicoletti Mathias e Celso Maciel da Costa, da Pontifícia Universidade Católica do Rio Grande do Sul, e a Diego Kreutz, Marcelo Pasin e Benhur Stein, da Universidade Federal de Santa Maria.

Ao CTT® (Conselho Tutorial do Trago) pelo companherismo, pela cachaça e pela união.

Aos residentes do chatô-gaudério MC e Gonçalo, a minha profunda gratidão pelos momentos tão agradáveis que passamos juntos.

A toda comunidade gaúcha residente em Campinas que, de alguma forma, contribuiram para diminuir a saudade da Querência: MC, Gonçalo, Luciana, Márcio, Amanda, Célio, Glaidson, Jeferson, Emilene, Mário, Tatiane, Vinícius, César, Ricardo, Geomar e todos os demais que eu tenha esquecido.

A CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pelo apoio financeiro.

Aos meus pais, que sempre me deram o apoio necessário, em todos os sentidos. Serei sempre grato a vocês. Também agradeço a toda minha família, em especial aos meus irmãos Rômulo e Laura, ao meu cunhado Rafael e aos meus queridos sobrinhos Henrique e Victor. Sem o amor de vocês nada seria.

A Vanessa, minha querida noiva, o meu amor, meu carinho e a minha gratidão pelo apoio incondicional em todos os momentos.

A Deus pela inspiração e alegria de viver.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Abreviaturas

Capítulo 1

Introdução

O interesse por algoritmos mais eficientes tem sido uma preocupação da comunidade científica ligada à otimização de sistemas. Embora a evolução na escala de integração dos microprocessadores proporcione a duplicação da velocidade dos mesmos a cada 18 meses[1], há grandes limitações para resolver "grandes" instâncias de problemas de otimização combinatória. Uma alternativa promissora para agregar maior poder computacional e, portanto, resolver mais rápido essas instâncias é empregar técnicas de processamento paralelo.

Vários trabalhos que aplicam técnicas de processamento paralelo podem ser encontrados na literatura. Em algoritmos de programação genética há um grande custo para avaliação dos indivíduos, os trabalhos de (Andre & Koza 1998) e (Tanev, Uozumi & Ono 2001) são bons exemplos de como contornar esse problema empregando técnicas de processamento paralelo. (Andre & Koza 1998) desenvolveram uma implementação paralela de um algoritmo de programação genética aplicado a problemas de regressão simbólica. Eles empregaram uma rede na qual cada nó processador tem uma arquitetura baseada em *transputers*. Suas conclusões indicaram o sucesso da implementação devido ao melhor desempenho obtido com o paralelismo. (Tanev et al. 2001) propuseram uma arquitetura para implentações paralelas e distribuídas de programação genética. Suas conclusões indicaram um desempenho superior do algoritmo que utiliza a arquitetura em relação ao mesmo algoritmo que não faz uso dela.

Em otimização de funções, os trabalhos de (Schlierkamp-Voosen & Muehlenbein 1994) e (Digalakis & Margaritis 2000) são bons exemplos de aplicação de algoritmos evolutivos paralelos. (Schlierkamp-Voosen & Muehlenbein 1994) empregaram um algoritmo genético (AG) com várias subpopulações, cada uma com um comportamento evolutivo distinto. Seus resultados mostraram que, para o conjunto de funções analisadas, a competição entre as subpopulações proporciona melhores resultados em relação ao mesmo algoritmo que emprega várias subpopulações com comportamento evolutivo idêntico. (Digalakis & Margaritis 2000) utilizou uma avaliação de desempenho entre implementações paralelas de AG e algoritmos meméticos (AM), reportando bons resultados em relação a algoritmos seqüenciais.

[1] Segundo a lei de *Moore*.

Um trabalho importante em AG paralelos é o de (Huntley & Brown 1996). Eles propuseram, além da adição de um operador de busca local no AG tradicional, dois modelos paralelos e os aplicaram no problema de designação quadrática. A superioridade do modelo proposto por eles, em relação a outros métodos como *multistart*, fica evidenciada nos resultados. Aplicações específicas em problemas de seqüenciamento são os trabalhos de (Mayer 1999) e (Lee & Kim 1995). (Mayer 1999) apresentou um modelo composto por vários AG independentes, cada um alocado a um nó da rede de computadores. Ele reportou bons resultados para instâncias de 50 e 100 tarefas do problema de seqüenciamento de tarefas em máquina simples. Já o trabalho de (Lee & Kim 1995) propôs um algoritmo semelhante ao anterior aplicado ao mesmo problema. A diferença conceitual é que neste último há várias subpopulações evoluindo independentemente, havendo comunicação entre elas em determinados períodos do processo evolutivo. Foram testados, com relativo êxito, vários operadores de recombinação e algumas políticas de seleção para instâncias de 50, 80 e 100 tarefas do mesmo problema do trabalho anterior.

Uma importante e recente investigação sobre políticas de migração em algoritmos evolutivos paralelos (AEP) foi desenvolvida por (Cantú-Paz 2001). Ele demonstrou, através de estudos práticos e teóricos, que a velocidade de convergência e a escolha dos indivíduos para emigrantes, assim como a escolha dos indivíduos para serem substituídos pelos emigrantes, estão intimamente relacionadas.

Aplicações de técnicas de processamento paralelo para o problema do caixeiro viajante podem ser encontradas em (Gorges-Schleuter 1997) e (Moscato & Norman 1992). No trabalho de (Gorges-Schleuter 1997) é empregado uma modificação da estrutura do algoritmo já consolidado em (Gorges-Schleuter 1989), que utiliza uma estrutura espacial de subpopulações na forma de um anel, aplicado aos casos simétrico e assimétrico. A idéia central do trabalho de (Moscato & Norman 1992) é aplicar um esquema de cooperação-competição entre subpopulações/agentes para realizar otimização global a partir de heurísticas de busca local.

Neste trabalho é proposto um modelo de algoritmo memético paralelo para a resolução dos problemas de seqüenciamento em máquina simples, seqüenciamento em máquinas paralelas e do caixeiro viajante assimétrico. Para os dois primeiros problemas foi realizado uma modificação no código de (Mendes 1999). Para o terceiro empregou-se esta mesma modificação no código de (Buriol 2000).

Esta dissertação está estruturada da seguinte forma: nas demais seções deste capítulo enuncia-se alguns conceitos básicos de computação paralela, medida de desempenho e especificações do *hardware* utilizado nos testes computacionais; no capítulo 2 trata-se dos algoritmos evolutivos e algoritmos meméticos; o algoritmo proposto é enunciado no capítulo 3; os resultados computacionais, para os problemas de seqüenciamento em máquina simples, seqüenciamento em máquinas paralelas e do caixeiro viajante, bem como as especificações dos algoritmos empregados, são apresentados nos capítulos 4, 5 e 6, respectivamente; por fim são apresentadas as conclusões e trabalhos futuros no capítulo 7.

1.1 Conceitos básicos de computação paralela

Nesta seção serão introduzidos alguns conceitos básicos necessários para os capítulos seguintes.

Uma unidade processadora[2] será compreendida como um computador qualquer da rede de computadores utilizada. O conjunto de todas estas unidades processadores disponíveis formará o sistema multiprocessado. Todo e qualquer algoritmo que seja executado em uma ou mais unidades do sistema multiprocessado será tratado como um programa e significará um algoritmo genérico executando em uma ou mais unidades processadoras.

1.1.1 Computação Paralela

Segundo (Gottlieb & Almasi 1989), é possível entender computação paralela como "um conjunto de unidades processadoras que se comunicam e cooperam para resolver grandes problemas rapidamente". Uma característica fundamental de uma aplicação paralela é a concorrência (Foster 1995): habilidade de realizar várias tarefas simultaneamente.

1.1.2 *Hardware*

Há diferentes formas de organização do *hardware* composto de várias CPUs, especialmente quanto a forma de conexão e comunicação entre estas unidades (Tanenbaum 1995). (Flynn 1972) propôs uma classificação para sistemas computacionais com várias CPUs muito citada na literatura. Essa classificação parte de duas características fundamentais: o número de fluxos de instrução (*instruction streams*) e o número de fluxos de dados (*data streams*). As quatro categorias são descritas a seguir.

- **SISD** (*single instruction, single data*): um fluxo de instrução e um fluxo de dados;
- **SIMD** (*single instruction, multiple data*): um fluxo de instrução e vários fluxos de dados;
- **MISD** (*multiple instruction, single data*): vários fluxos de instrução e um fluxo de dados;
- **MIMD** (*multiple instruction, multiple data*): vários fluxos de instrução e vários fluxos de dados.

1.1.3 Processo

Segundo (Tanenbaum 1992), um processo corresponde a um programa em execução, o qual compreende um programa executável, seus dados, um contador de programa, uma pilha, registradores e todas as informações que eventualmente o sistema operacional possa requerer. Em um sistema que empregue a divisão[3] do tempo de CPU entre os vários

[2] Uma *unidade processadora* (ou CPU) pode ter um ou mais processadores.

[3] *Timesharing systems.*

processos, sempre que um destes for interrompido se fará necessário salvar informações a respeito das instruções que estavam sendo executadas no momento em que houve a interrupção. Essas informações serão indispensáveis para que o mesmo seja reiniciado corretamente a partir do ponto em que parou.

Em síntese, um processo é um programa em execução que possui certas informações relevantes para o sistema operacional (Tanenbaum 1992).

1.1.4 *Thread*

Todo fluxo de execução vinculado a um processo corresponde a uma *thread*. Todo processo tem, no mínimo, uma *thread* que representa o programa em execução. Caso ele tenha mais de uma, todas irão compartilhar a área de dados destinada ao processo, contudo terão independência quanto à execução. Essa é a grande vantagem no uso de *threads*: todas elas compartilham a mesma área de dados, facilitando a troca de informações.

Threads podem ser entendidas como mini-processos. Elas compartilham tempo de CPU, tal como os processos, mas não podem ser executadas independentemente[4].

1.1.5 *Sockets*

Um *socket* é um canal de comunicação bidirecional entre dois programas que estão executando em uma rede de computadores, sendo limitado por uma porta que permite a camada TCP (Comer 1998) indentificar para quem os dados devem ser entregues, (Sun-microsystems 2001).

1.1.6 *Speedup*

O critério de desempenho adotado para medir quantas vezes mais rápido[5] um algoritmo paralelo é em relação ao similar seqüencial foi o *speedup* (Neary & Capello 2000). Apesar das limitações, esta medida de desempenho fornece uma boa aproximação para a análise prática necessária a este trabalho.

Tracionalmente, *speedup* pode ser definido como $S = \frac{T_s}{T_p}$, sendo T_s o tempo necessário para executar um algoritmo de forma seqüencial (em um processador) e T_p o tempo para executá-lo de forma paralela em p processadores. Essa definição se aplica para ambientes homogêneos, onde todos os processadores são idênticos[6]. Quando se considera ambientes heterogêneos é possível adaptar o conceito anterior considerando o tempo T_s para cada família de computadores, como descrito a seguir:

- Dados $p_1 + \ldots + p_k$ as quantidades de K tipos distintos de processadores, $T_s(i)$ o tempo para executar o algoritmo seqüencial em um processador do tipo i e $T_p(p_1, \ldots, p_k)$ o tempo para executar o algoritmo paralelo em todos os K tipos processadores.

[4] Uma *thread* precisa estar vinculada a um processo.
[5] Em termos de tempo computacional.
[6] Possuem a mesma velocidade de processamento e a mesma quantidade de memória.

- O tempo médio ponderado para a execução do algoritmo de forma seqüencial e o *speedup* (S), são calculados conforme as equações 1.1 e 1.2, respectivamente.

$$T_s(p_1, \ldots, p_k) = \frac{p_1 T_s(1) + \ldots + p_k T_s(k)}{p_1 + \ldots + p_k} \tag{1.1}$$

$$S = \frac{T_s(p_1, \ldots, p_k)}{T_p(p_1, \ldots, p_k)} \tag{1.2}$$

1.2 Testes Computacionais

A implementação foi feita em linguagem Java^{TM} (*Sun microsystems*), JDK versão 1.3.1. Nos testes computacionais foi utilizada uma rede padrão *Ethernet* 10Mbits, com 8 computadores do tipo PC Intel Celeron 330 MHz, cada um com 64 MB de memória RAM. O sistema operacional utilizado foi o Linux, kernel versão 2.4.

Capítulo 2

Algoritmos Evolutivos

O interesse em técnicas de resolução de problemas inspiradas em processos biológicos tem sido crescente nas últimas décadas. Tal fato se justifica pela grande complexidade associada aos problemas, por exemplo os de natureza combinatória, o que torna proibitiva a aplicação de métodos exatos para a resolução dos mesmos (Lewis & Papadimitriou 1981). Além disso, há casos em que o problema a ser resolvido apresenta características, tais como não-linearidade e não-diferenciabilidade, que impedem a aplicação de métodos tradicionais (método do gradiente, técnicas de programação linear). Neste panorama, o uso de métodos genéricos, como os algoritmos evolutivos (AE), representa uma boa opção de se obter soluções aproximadas para problemas até então intratáveis.

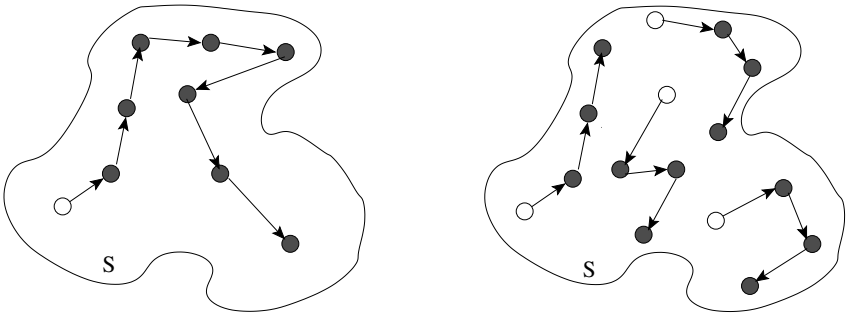
Segundo (Bäck, Fogel & Michalewicz 2000a), é possível descrever AE como procedimentos de simulação da evolução natural, geralmente aplicados em computadores. Eles empregam variação aleatória e seleção em uma população de soluções candidatas. Outros mecanismos como seleção e recombinação também fazem parte da estrutura geral destes algoritmos.

Enquanto outros métodos baseados em busca local, como *busca tabu* e *simulated annealing*, percorrem o espaço de solução (S) utilizando somente uma solução candidata, os AE o fazem com várias, num processo essencialmente paralelo. Isso é o que distingue métodos populacionais (AE) dos não-populacionais (*busca tabu* e *simulated annealing*). As figuras 2.1(a) e 2.1(b) ilustram essa diferença.

Conforme ilustra a figura 2.1, a busca não-populacional percorre S de forma seqüencial, enquanto a populacional o faz de forma distribuída e, por isso, mais abrangente. Isso lhe confere robustez em detrimento do tempo de busca.

Há dois objetivos básicos quando na utilização desses algoritmos. O primeiro é o uso desta técnica altamente flexível para resolver problemas nos quais os métodos tradicionais falham. O segundo se refere a simulação de ambientes naturais altamente complexos. Uma conseqüência desta utilização é a simulação de vida artificial e incorporação de conhecimento (Fogel 1995).

Embora os AE sejam genéricos quanto a sua aplicação, por apresentarem etapas que são facilmente adaptáveis ao problema que se quer tratar, sua eficiência pode ser eventualmente comprometida exatemente por essa generalidade. Existem problemas em que



(a) Busca não-populacional (b) Busca populacional

Figura 2.1: Métodos populacionais e não-populacionais.

o conhecimento acerca do mesmo é fundamental para que boas soluções sejam obtidas. Se o método for especilizado ganha-se em eficiência e perde-se em generalidade. Este compromisso entre eficiência e generalidade corresponde ao grande desafio dos métodos de resolução (Wolpert & Macready 1996).

Um requisito fundamental para aplicação dos AE é a necessidade de recursos computacionais. Por se tratar de uma técnica que envolve grande número de operações[1] computacionalmente custosas, a dimensão do problema a ser resolvido é limitada por esses recursos.

As áreas de aplicação prática desta técnica são as mais diversas. Algumas delas são:

- **Otimização Combinatória**: problemas clássicos como o problema do caixeiro viajante (Goldberg & Lingle 1985), problema de seqüenciamento de tarefas (Davis 1985), problema de empacotamento (Smith 1985).

- **Projeto**: desenvolvimento de sistemas eletrônicos e digitais (Fonseca, Mendes, Fleming & Billings 1993), projeto de redes neurais artificiais, tanto em topologias como em busca pelo conjunto de pesos ótimos, (Zhang & Mühlenbein 1993).

- **Simulação e identificação**: estimação de funções estatísticas (Janikow & Cai 1992), determinação de estruturas tridimensionais de proteínas (Lucasius & Kateman 1992).

- **Controle**: como controladores de sistemas (off-line e on-line), em sistemas de navegação (Krishnakumar & Goldberg 1990).

- **Classificação**: em sistemas capazes de captar e classificar características de um ambiente para dar suporte às decisões tomadas pelo sistema de controle (Fogarty 1994).

Embora existam várias classes de AE, pode-se descrever um algoritmo genérico comum a todos, conforme ilustra a figura 2.2 (Michalewicz 1996).

[1]Essas operações são comparação, avaliação, seleção, geração de números aleatórios, entre outras.

Entrada: População de indivíduos **P**
Saida: População de indivíduos **P**

AlgEvolutivo()
1 Inicialize(P);
2 Avalie(P);
3 $geracoes \leftarrow 0$;
4 **while** $Criteriodeparada = falso$
5 **do** Altere(P);
6 Avalie(P);
7 Selecione(P);
8 $geracoes \leftarrow geracoes + 1$;
9 **return** P;

Figura 2.2: Algoritmo Evolutivo

No primeiro passo a população é inicializada de acordo com a representação adotada, geralmente de forma aleatória. A avaliação dos indivíduos após essa inicialização é feita no passo 2. No passo 5 a população é alterada, o que significa a geração de novos indivíduos através da recombinação de outros como também a aplicação de variação aleatória que provoque pequenas ou grandes modificações nos indivíduos. Esses novos indivíduos, gerados no passo 5, serão avaliados no passo 6 e concorrerão com os demais para permanecerem na população no passo 7. A repetição dos passos 5, 6, 7 e 8 ocorrerá até que o critério de parada seja satisfeito.

Atualmente é possível classificar quatro classes de algoritmos evolutivos:

- **Estratégias Evolutivas**: formalizadas por (Rechenberg 1973). O modelo inicial se propunha otimizar parâmetros discretos e contínuos, empregrando, para isso, somente variação aleatória e seleção. No entanto as técnicas atualmente utilizadas já empregam recombinação.

- **Algoritmos Genéticos**: considerados como a primeira classe de algoritmos evolutivos com ênfase em recombinação, foram propostos por (Holland 1975) como uma tentativa de formalizar e explicar o processo de adaptação em sistemas naturais. Pode-se determinar três características básicas do modelo clássico: método de seleção proporcional, método de recombinação e representação de indivíduos como uma seqüência de *bits*.

- **Sistemas Classificadores**: seus fundamentos foram introduzidos por (Goldberg 1989) e (Holland, Holyoak, Nisbett & Thagard 1986), quando esta técnica se tornou precursora no emprego de Algoritmos Genéticos em aprendizado de máquina. Apresenta grande sofisticação e complexidade nas três camadas que a compõe: inferência, aprendizado e descoberta de conhecimento. É comumente utilizada como controlador de ambientes altamente ruidosos (Twardowski 1993), com grande volume de informações muitas vezes conflitantes.

- **Programação Genética**: proposta por (Koza 1992). Sua estrutura é idêntica a dos Algoritmos Genéticos, exceto por algumas modificações nas estruturas de representação, operadores genéticos e avaliação. Isto porque as estruturas de dados a serem evoluídas constituem programas de computadores executáveis, sendo esta a grande diferença desta técnica em relação as demais.

2.1 Conceitos Básicos

Nesta seção são introduzidos alguns conceitos fundamentais sobre AE.

Em se tratando da teoria evolutiva empregada nestes algoritmos, o paradigma neo-Darwiniano é o mais aceito e utilizado. Esta teoria afirma que a história da vida pode ser explicada por processos físicos operando dentro de populações e espécies. Estes processos são recombinação, mutação, competição e seleção (Bäck et al. 2000a, cap.4). Reprodução é o mecanismo natural para que uma espécie seja perpetuada, o qual compreende a transferência de material genético dos ancestrais para seus descendentes. Mutação representa uma perturbação na informação genética quando a mesma é replicada. Competição é a conseqüência do aumento da população num ambiente com recursos limitados. Seleção representa a competição para preencher o espaço disponível no ambiente. Repetindo estes processos geração a geração é esperado que haja a criação de indivíduos mais bem adaptados.

Genótipo e Fenótipo irão formar a programação genética de cada indivíduo/espécie. O primeiro representa o mecanismo para armazenagem da informação adquirida ao longo do processo evolutivo. O segundo significa a manifestação do genótipo no ambiente, em um determinado instante do processo evolutivo. Quando na avaliação do indivíduo, o fenótipo irá descrever a adaptação do indivíduo ao ambiente.

Assim, o processo evolutivo pode ser utilizado como método de resolução de problemas de otimização. O mecanismo de seleção direciona os fenótipos para o ótimo, a partir das condições iniciais e restrições no ambiente. Contudo, devido às mudanças constantes no ambiente, nenhum organismo pode ser considerado perfeitamente adaptado. Mais especificamente, cada indivíduo será, portanto, uma possível solução para o problema de otimização formulado. No entanto, para que isso ocorra, é necessário encontrar uma representação adequada, ou seja, a partir da especificação de uma solução para o problema encontrar uma codificação que traduza essa especificação em um indivíduo. Esta questão é de suma importância e pode representar o sucesso ou o fracasso do método (Michalewicz 1996). Adicionalmente, o conjunto de todas as possíveis soluções será denominado espaço de busca.

Outro fator importante a ser mencionado é a forma com que são representados os indivíduos em AE. É necessário definir uma terminologia para termos que serão tratados mais adiante. Toda referência a indivíduos remete a um organismo que reside em um ambiente sujeito a determinadas leis de sobrevivência e reprodução. Os cromossomos[2] desses indivíduos formarão a informação genética do mesmo. Por último, como cada

[2]Todos os indivíduos terão um único **cromossomo**.

cromossomo é, em geral, uma seqüência de números, cada um destes terá uma posição relativa (locus) e o valor nela armazenado corresponderá ao alelo.

2.2 Algoritmos Evolutivos Paralelos

Conforme citado anteriormente, algoritmos evolutivos são aplicados em várias áreas, com as mais diversas abordagens. Não há dúvida quanto a sua capacidade em obter boas soluções. No entanto, quando aplicados a "grandes" e "difícieis" problemas há um aumento significativo no tempo para resolução dos mesmos. Por isso existe grande esforço para se projetar AE mais rápidos e eficientes e uma das mais promissoras alternativas é a utilização de implementações paralelas. Além disso, essas implementações também proporcionam um aumento no poder de busca do método, o que é muito desejável em problemas que apresentam mínimos locais.

Evidentemente há um paralelismo implícito nos AE devido a presença de uma população de soluções. Contudo as tarefas como seleção e recombinação necessitam de toda a população para serem aplicadas, o que limita o paralelismo da técnica. Outras, porém, como avaliação e mutação, são totalmente independentes, permitindo uma aplicação particular para cada indivíduo. Exatamente destas fases é que as formas clássicas de implementações paralelas irão se beneficiar.

Dois propósitos permeiam qualquer tentativa de melhorar o desempenho de um AE com técnicas de implementação paralela: reduzir o tempo computacional e aumentar a qualidade da solução encontrada. Como será visto a seguir, algoritmos que objetivam somente reduzir o tempo computacional possuem o mesmo comportamento evolutivo que os correpondentes seqüenciais. Os outros buscam também um compromisso maior entre exploração e explotação, ou intensificação e diversificação, com modelos mais sofisticados que combinam dinâmica populacional e nichos ecológicos.

A seguir descreve-se as três formas clássicas de pararelização dos AE comumente encontradas na literatura. (Cantú-Paz 1997b) fez um estudo amplo descrevendo esses algoritmos clássicos, com as respectivas aplicações práticas.

2.2.1 Algoritmos Evolutivos Paralelos Globais

Os algoritmos evolutivos paralelos globais (AEPG) são assim chamados porque os operadores de seleção, reprodução e mutação são aplicados à população inteira. A implementação é realizada através do paradigma mestre-escravo; uma unidade mestre atribui certas funções do algoritmo a outras unidades chamadas de escravos, que realizam a tarefa e retornam o resultado.

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 uma comunicação ocorre quando a mesma é enviada e recebida. Quando uma unidade mestre espera pelas respostas de todas as unidades escravo para proceder novas etapas, o algoritmo é dito síncrono. Ele preserva todas as características do com-

portamento evolutivo do algoritmo seqüencial, porém com melhor desempenho. Outra possibilidade é fazer com que a unidade mestre não precise esperar por todas as respostas, o que caracteriza os AEPG assíncronos. Neste caso ocorre uma diferenciação em relação às propriedades do algoritmo seqüencial: indivíduos de uma geração passam para as próximas como se houvessem migrado, modificando o comportamento evolutivo do algoritmo. Exatamente nesta definição é que reside a questão entre tempo computacional e qualidade de solução: nos AEPG síncronos o único objetivo é diminuir o tempo computacional necessário para execução do algoritmo; os AEPG assíncronos objetivam não só diminuir o tempo computacional mas também obter melhores soluções.

Os AEPG não requerem uma arquitetura computacional específica, podem ser implementados eficientemente em computadores com memória compartilhada ou distribuída. No primeiro caso, a população é deixada na memória compartilhada e cada unidade do sistema multiprocessado tem acesso à parte que lhe coube. No outro, a população reside em uma unidade que é responsável pelo envio dos indivíduos aos demais e pela coleta dos resultados. Em ambos há um custo associado à comunicação, resultando num compromisso entre tempo de execução e número de unidades escravo. (Cantú-Paz 1997a) analisa detalhadamente este compromisso, fornecendo resultados que indicam a existência de um número ótimo de unidades escravo que minimiza o tempo de execução do algoritmo. A figura 2.3 ilustra o diagrama básico de um AEPG.

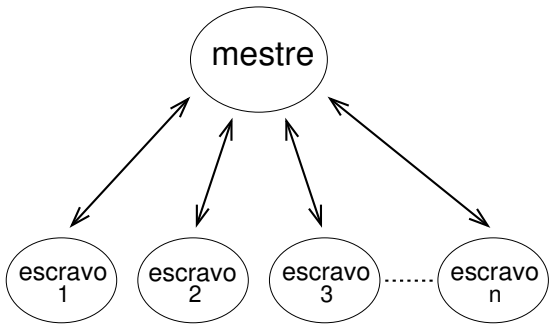


Figura 2.3: Estrutura básica de um **AEPG.**

Um AEPG aplicado ao problema de geração de horário de escolas (*school timetabling*) foi desenvolvido por (Abramson & Abela 1992). Esse algoritmo foi implementado em um computador com memória compartilhada (Encore Multimax com 16 processadores) e os resultados obtidos com um número bem limitado de instâncias pareceu promissor. Uma modificação deste código foi empregada para geração de horários de trens, por (Abramson, Millis & Perkins 1993). Um outro computador paralelo foi empregado neste problema (Fujitsu AP1000 com 128 processadores) e os resultados pareceram animadores para até 16 processadores. Acima disto o custo de comunicação passou a comprometer. (Hauser & Männer 1994) realizaram um bom estudo em relação às dependências entre as fases do AG, implementando um AEPG em vários computadores paralelos, todos do tipo MIMD[3]. Seus melhores resultados foram obtidos naquele computador no qual o programador detém

[3]Seção 1.1.2.

total controle do sistema para decidir quantos processadores usar e que instruções cada um vai executar, apresentando ainda um tempo de comunicação bem baixo. Recetemente (Oussaidène, Chopard, Pictet & Tomassini 1997) empregaram um AEPG para inferir estratégias para investimentos no mercado financeiro. Foi utilizado um computador IBM SP-2 e o *speedup* foi quase linear, no entanto estes resultados apresentaram-se bastante dependentes da dimensão do problema (em geral o *speedup* é maior nos problemas "grandes").

Uma regra muito importante a ser seguida em todo AEPG é observar o tempo de comunicação e o tempo de execução da tarefa a ser distribuída para as unidades escravo. O tempo de comunicação é o tempo que uma dada tarefa leva para ser enviada da unidade mestre para uma unidade escravo. Para que o *speedup* seja maior que 1[4] é necessário que o tempo de execução seja algumas ordens de grandeza maior que o tempo de comunicação. O *speedup* será tão próximo do limitante superior (número de processadores associados) quanto maior for a razão entre o tempo de execução e o tempo de comunicação. Essa questão será tratada com mais detalhes na seção 3.1.

2.2.2 Algoritmos Evolutivos Paralelos Multi-Populacionais

Quando se observa o mundo real constata-se que há somente uma grande população, dividida em continentes, países e regiões. A partir de outro ponto de vista, pode-se entender que há várias populações evoluindo independentemente, isoladas em determindados regiões, países e continentes. Em determinada fase do processo evolutivo ocorre a migração de indivíduos de uma população para outra, compartilhando, portanto, material genético. A idéia é que ambientes isolados seriam mais eficientes, em relação ao poder de busca no espaço de soluções, do que um só que reunisse todos os indivíduos. Os algoritmos evolutivos multi-Populacionais (AEPM) são, portanto, conseqüência da aplicação desta teoria aos métodos de busca para resolução de problemas de otimização.

Em se tratando de AG, há um enorme problema em dimensionar o tamanho da população para que a busca seja eficiente e não incorra em convergência prematura (Bäck, Fogel & Michalewicz 2000b, cap.17). Ou seja, relegar o compromisso entre diversificação e intensificação pode fazer com que o algoritmo explore somente determinada região do espaço de soluções. Uma possível solução é apelar para a diversificação explícita, aceitando os novos indivíduos, advindos do processo de reprodução, se os mesmos forem diferentes de seus pais segundo um determinado fator. Outra possibilidade é o emprego dos AEPM, pois dividindo a população em várias subpopulações e mantendo-as isoladas é possível que cada uma delas explore uma determinada região do espaço de soluções.

Nesta classe, portanto, há não só o interesse em melhorar o desempenho do método em relação ao tempo computacional como também em melhorar a qualidade da solução encontrada. Um aspecto importante e que irá determinar duas novas subclasses é o tamanho das populações, (Cantú-Paz 1997b): a primeira abordagem emprega poucas populações com tamanho "médio/grande"; a segunda adota muitas populações com tamanho "peque-

[4] O que significa que o AEPG é mais rápido que o algoritmo seqüencial.

no". A conseqüência imediata é que no primeiro caso há uma baixa taxa de comunicação e no segundo ela tende a ser alta. Isso diferenciará o tipo de *hardware* empregado em cada um deles.

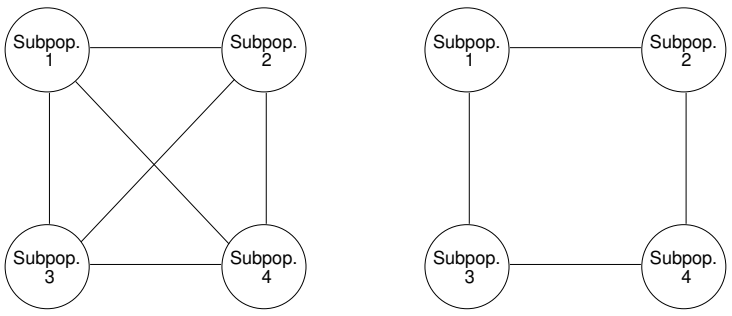
A seguir são descritas essas duas subclasses chamadas de:

- Algoritmos evolutivos paralelos multipopulacionais distribuídos (AEPMD) ;
- Algoritmos evolutivos paralelos multipopulacionais maciçamente Paralelos (AEPMMP).

2.2.2.1 Algoritmos Evolutivos Paralelos Multipopulacionais Distribuídos

Os algoritmos evolutivos paralelos multi-Populacionais distribuídos (AEPMD), ou modelo de ilhas, recebem esta denominação devido principalmente à baixa comunicação e à semelhança com o modelo natural usado para descrever populações isoladas geograficamente. A idéia central é dividir a população principal em várias, sendo cada uma atribuída a uma unidade do sistema multiprocessado. Cada uma dessas unidades, por sua vez, executa um algoritmo evolutivo seqüencial, restrito a sua subpopulação. A troca de material genético é proporcionada pela migração, a qual ocorre em períodos determinados do processo evolutivo de cada subpopulação.

O projeto eficiente deste modelo é bastante penoso porque envolve muitos parâmetros inter-relacionados, como tamanho e número de subpopulações, organização ou topologia entre as subpopulações e critérios de migração. O estudo de (Cantú-Paz 1999) traz uma análise completa desses parâmetros, sinalizando algumas direções promissoras. A figura 2.4(a) ilustra uma topologia para um AEPMD com as subpopulações totalmente conectadas. A outra, 2.4(b), mostra as subpopulações parcialmente conectadas.



(a) Subpopulações totalmente conectadas

(b) Subpopulações parcialmente conectadas

Figura 2.4: Topologias para um **AEPMD**.

Talvez o primeiro experimento com AEPMD tenha sido realizado por (Grosso 1985). Seu objetivo era estudar a dinâmica evolutiva de subpopulações de uma população. Ele dividiu a população em 5 subpopulações, sendo que cada uma delas podia trocar indivíduos

com todas as demais. Sua conclusão em relação ao tempo de convergência, comparando pequenas subpopulações isoladas e uma grande população, foi que no primeiro caso a convergência era mais rápida. Outra observação foi que o valor de *fitness* obtido com subpopulações complementamente isoladas era pior que o obtido com uma grande população, o que significa que dividir a população em várias subpopulações isoladas não apresenta ganhos em relação a qualidade da solução encontrada. Em relação à taxa de migração, seus resultados indicaram que com uma baixa taxa as subpopulações mantêm-se praticamente isoladas, apresentando a mesma qualidade de solução do que quando estão totalmente isoladas. Para taxas intermediárias, a qualidade da solução obtida apresentou-se semelhante àquela encontrada quando empregou-se uma grande população. Dessa forma esses resultados apontam para a existência de uma região crítica abaixo da qual o algoritmo comporta-se como se as subpopulações estivessem completamente isoladas e acima da qual como se houvesse somente uma grande população.

Uma característica importante dos AEPMD é o comportamento evolutivo das subpopulações. É possível que para cada uma delas se designem operadores de recombinação e mutação próprios e diferentes das demais. (Schlierkamp-Voosen & Muehlenbein 1994) empregou esta idéia e usou seu algoritmo para minimização de um conjunto determinado de funções[5]. Seus resultados mostraram que um AEPMD que apresente competição entre as subpopulações é mais efetivo e robusto em relação ao mesmo algoritmo com o mesmo comportamento evolutivo para todas elas. Contudo os parâmetros adicionados, provenientes das estratégias pré-estabelecidas, acabaram tornando as conclusões bastante relativas.

Recentemente (Tanev et al. 2001) propuseram uma arquitetura para implementações paralelas e distribuídas de programação genética. Através de uma rede local ou até mesmo a *Internet* é possível ter subpopulações semi-isoladas, migrando seus indivíduos para as outras através de um barramento comum. Um controle centralizado determina qual será a vizinhança de cada subpopulaçao e a que nó do sistema ela será alocada. Cada uma delas evolui por um número pré-estabelecido de gerações de forma assíncrona, isto é, elas começam e terminam a execução independentemente. Os indivíduos imigrantes recebidos por uma subpopulação aguardam em um *buffer* o momento correto para serem assimilados. A avaliação quanto ao desempenho contemplou tempo computacional e qualidade de solução. Para tanto os autores utilizaram o problema de identificação de funções simbólicas. Suas conclusões indicam a superioridade do algoritmo que utiliza a arquitetura em relação ao mesmo algoritmo com uma única população, para ambos os critérios. Uma observação importante diz respeito ao esforço computacional para execução: para pequenas configurações houve aumento do tempo necessário para executar o algoritmo com uma única população em um único nó processador.

Diante do exposto, algumas razões que tornam esta classe tão popular são:

- Grande semelhança com os AE tradicionais (seqüenciais). Na verdade uma implementação simples pode ser obtida combinando AE seqüenciais, executando cada

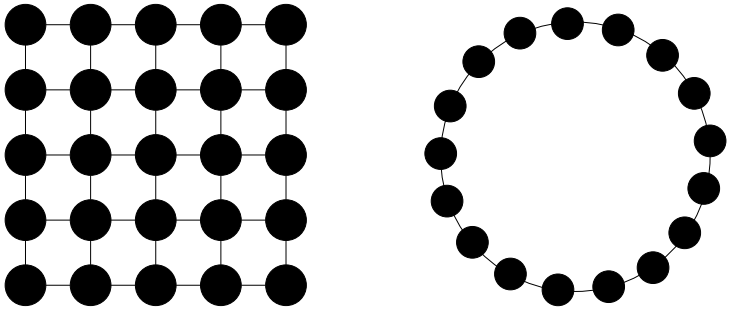
[5]Funções unimodais e multimodais.

um em um nó do computador paralelo e promovendo a migração de indivíduos em certos períodos do processo evolutivo por um número pré-determinado de vezes.

- Baixo requisito de *hardware*. Computadores paralelos com poucos nós processadores e com memória distribuída são comumente encontrados ou facilmente implementáveis em qualquer rede local através de *softwares* livres como PVM (Geist, Beguelin, Dongarra, Jiang, Manchek & Sunderam 1994) ou MPI (Gropp, Lusk & Skjellum 1999).
- Pouco esforço para transformar um AE seqüencial em um AEPMD. Apenas algumas rotinas são necessárias para promover a migração dos indivíduos.

2.2.2.2 Algoritmos Evolutivos Paralelos Multipopulacionais Maciçamente Paralelos

Enquanto que no caso anterior havia baixa comunicação entre as subpopulações, nos Algoritmos Evolutivos Paralelos Multipopulacionais Maciçamente Paralelos (AEPMMP) ocorre uma comunicação bem maior ocasionada pela divisão da população em muitas subpopulações pequenas, de forma a disseminar boas soluções entre elas. Novamente seleção, recombinação e mutação são operações restritas à subpopulação. A figura 2.5(a) ilustra uma topologia em grade para um AEPMMP. Uma topologia em anel é ilustrada pela figura 2.5(b). Observa-se que o grande diferencial entre elas será o número de vizinhos de cada subpopulação (vizinhança). Quanto maior for esse número, maior será a comunicação no algoritmo.



(a) Topologia em grade (b) Topologia em anel

Figura 2.5: Topologias para um **AEPMMP**.

Exatamente pelo número de subpopulações e pelo fato de se destinar cada uma delas a uma unidade do sistema multiprocessado é que nesta classe há um grande requisito de *hardware*. Também pela intensa comunicação entre as unidades, o computador paralelo deve prover um arquitetura na qual o custo de comunicação seja mínimo.

Um dos trabalhos pioneiros nessa classe é o de (Gorges-Schleuter 1989) e (Mühlenbein 1989). Eles propuseram o ASPARAGOS[6], que tem a população na forma de uma estrutu-

[6]Asynchronous Parallel Genetic Optimization Strategy.

ra semelhante a uma escada, com os extremos unidos. Outras características são ausência de controle central sobre as subpopulações, operadores de busca local aplicados aos indivíduos e sobreposição de subpopulações (um indivíduo pode pertencer a mais de uma subpopulação). Esse algoritmo foi aplicado com êxito a vários problemas de otimização (Mühlenbein 1989).

Esse mesmo algoritmo teve a estrutura em escada da sua população modificada para um anel (ASPARAGOS 96) (Gorges-Schleuter 1997), com o objetivo de diferenciar melhor os indivíduos e reduzir o tamanho das subpopulações, diminuindo conseqüentemente o esforço computacional. Os resultados obtidos para o problema do caixeiro viajante simétrico e assimétrico foram promissores em se tratando de redução da complexidade para resolução de ambos.

Outro experimento que utiliza a sobreposição de subpopulações é o trabalho de (Baluja 1992). Sua abordagem objetiva a transferência gradual de material genético em substituição à introdução repentina em determinados intervalos, como ocorre nos modelos da classe anterior. É usada uma estrutura em grade, que também apresenta um mecanismo para garantir a independência entre os comportamentos evolutivos de cada subpopulação, o qual controla exatamente quão sobrepostas elas estarão. A validação desse modelo foi obtida com vários problemas teste.

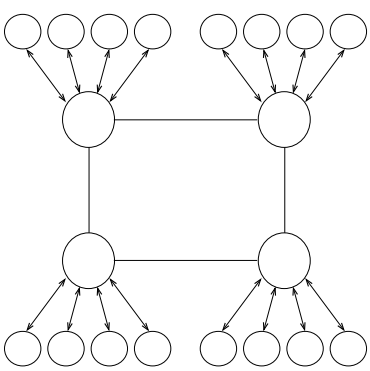
O trabalho de (Schwehm 1993) descreve a implementação de um AEPMMP em um computador MASPAR MP-1, um computador maciçamente paralelo com uma estrutura em grade dos processadores, podendo reunir até 16384 deles. Seus testes apresentaram uma superioridade desta implementação comparada às outras que também utilizam *hardware* paralelo. Uma arquitetura para processamento maciçamente paralelo, baseada na tecnologia Java e na www, é proposta por (Henriques 1999), a qual possibilita que todo computador conectado a *Internet* faça parte desta grande máquina virtual paralela.

2.2.3 Algoritmos Evolutivos Paralelos Híbridos

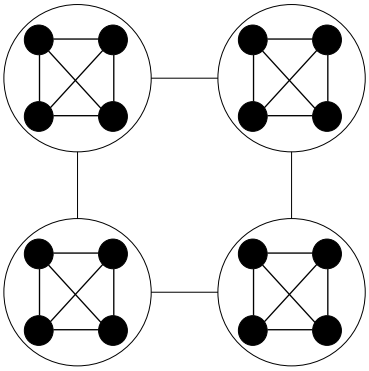
A combinação de classes enunciadas anteriormente tem resultado nos algoritmos evolutivos paralelos híbridos (AEPH) . O grande problema deles é o controle da complexidade resultante da associação de dois ou mais modelos.

Os modelos combinados formam uma hierarquia e em quase todos eles o nível superior é composto por AEPMD. O nível inferior pode ser composto de AEPMMP, AEPG ou até mesmo AEPMD. As figuras 2.6(a), 2.6(b) e 2.6(c) ilustram a combinação de AEPMD com AEPG, AEPMD com AEPMD e AEPMD com AEPMMP, respectivamente.

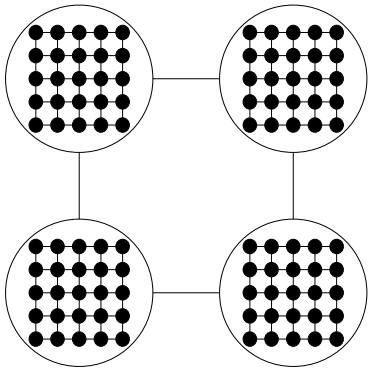
O próprio algoritmo de (Gorges-Schleuter 1997) combina dois níveis de AEPMMP. No nível mais elevado da hierarquia há um AEPMMP com várias subpopulações e em um nível abaixo cada uma delas mantém uma estrutura semelhante ao nível anterior porém com menor número de unidades. Um exemplo de uma hierarquia que combina AEPMD com AEPG é o trabalho de (Oussaidène et al. 1997).



(a) **AEPMD** e **AEPG**.



(b) **AEPMD** e **AEPMD**.



(c) **AEPMD** e **AEPMMP**.

Figura 2.6: Algumas topologias para um **AEPH**.

2.3 Algoritmos Meméticos

Essa classe de AE, também denominada de algoritmos genéticos gíbridos (AGH), caracteriza-se principalmente por adotar uma população de indivíduos autônomos (agentes). Todas as etapas dos AG estão presentes aqui: seleção, recombinação e mutação. A diferenciação advém da inclusão de uma etapa de otimização dos agentes, através da adição de operadores de busca local independentes. Exatemente aqui há outra grande diferença dos Algoritmos Meméticos AM em relação aos AG tradicionais: o compromisso é adicionar todo conhecimento disponível do problema a ser resolvido.

O termo memético denota uma nova unidade de informação, além do gene: o meme. Associado ao contexto da evolução cultural (Moscato 1989), o termo foi introduzido por (Dawkins 1976) para definir um novo conceito em transferência de informação. Tal como (Dawkins 1976) define, enquanto os genes saltam de corpo para corpo, os memes saltam de cérebro para cérebro. Uma observação importante é que esse processo não se limita simplesmente a transmitir a informação inalterada, ela também é processada e aumentada pela comunicação com os outros agentes. Fazendo uma análise simplificada, os memes são adicionados pelos operadores de busca local que particularizam o "aprendizado" para

cada agente. As regras contidas nos operadores não são transmitidas pelo gene, o que o diferencia explicitamente do meme. Contudo outras técnicas, além dos operadores de busca local, podem ser adicionadas na etapa de otimização, tais como algoritmos construtivos de aproximação, heurísticas, entre outros. Como a informação cultural (memes) não está totalmente vinculada ao mecanismo hereditário, ela pode ser transmitida de um indivíduo para toda a população, durante uma única geração.

Os AM foram aplicados com êxito em vários problemas de otimização combinatória (França, Mendes & Moscato 2000), (Lee 1994), (França, Gupta, Mendes, Moscato & Veltink 2000), (Mendes, Müller, França & Moscato 2002), (Min & Cheng 1998), (Merz & Freisleben 1997), (Buriol, França & Moscato 1999), (Holstein & Moscato 1999).

A figura 2.7 ilustra um AM típico. Tal como no AE, os passos 1 e 2 realizam a inicialização e avaliação dos indivíduos, respectivamente. O laço que inicia no passo 4 define o número de gerações do algoritmo, que é comumente utilizado como critério de parada. O outro, passo 6, irá determinar o número de indivíduos gerados[7]. A seleção dos pais para geração de um novo indivíduo é realizada no passo 7. A geração propriamente dita é realiza no passo 8. No passo 9 haverá uma mutação no indivíduo gerado, seguida da otimização, passo 10. A competição entre o novo indivíduo e os demais ocorrerá no passo 11, onde será feita uma tentativa de substituição de algum indivíduo da população por este novo. Ao final será retornada a população de soluções candidatas.

Entrada: População de indivíduos **P**
Saida: População de indivíduos **P**

AlgMemetico()
1 Inicialize(P);
2 Avalie(P);
3 $geracoes \leftarrow 0$;
4 **while** $Criteriodeparada = falso$
5 **do** $individuosGerados \leftarrow 0$;
6 **while** $individuosGerados < limite$
7 **do** $pais \leftarrow$ SelecioneIndividuos(P);
8 $novoIndividuo \leftarrow$ RecombineIndividuos($pais$);
9 RealizeMutacao($novoIndividuo$);
10 Otimize($novoIndividuo$);
11 Adicione($novoIndividuo, P$);
12 $individuosGerados \leftarrow individuosGerados + 1$;
13 $geracoes \leftarrow geracoes + 1$;
14 **return** P;

Figura 2.7: Algoritmo Memético

[7]Esse número determina quantos indivíduos serão gerados no processo de recombinação, durante uma geração.

Capítulo 3

Algoritmos Meméticos Paralelos

A motivação principal para esta dissertação decorre dos trabalhos de (Buriol 2000) e (Mendes 1999), que obtiveram relativo êxito nas abordagens do problema do caixeiro viajante (TSP) e problemas de seqüenciamento de tarefas[1] usando AM. Contudo verificou-se uma limitação quanto ao tempo computacional requerido pelos algoritmos, especialmente no segundo caso. Instâncias de tamanho "médio" do problema de SMS requeriam centenas de segundos. Para esse caso foi realizada uma análise com objetivo de verificar qual era o percentual correspondente a cada uma das fases (figura 2.7) do tempo computacional para executar o algoritmo. Os resultados evidenciaram o que já se suspeitava: a etapa de otimização dos indivíduos (operador de busca local) corresponde, em média, a 90% do tempo total para solucionar uma instância.

Diante dessa realidade, era necessário que se agregasse mais poder computacional para que fosse viável a execução do algoritmo para grandes instâncias. Uma alternativa para contornar esse problema seria o emprego de operadores de busca local mais eficientes, outra seria o emprego de implementações paralelas. O que influenciou a opção pelo segundo caso foi a possibilidade de aplicação da mesma técnica de paralelização para os três problemas (SMS, SMP e TSP), bem como do emprego de outras abordagens que melhoriam também o poder de busca do algoritmo.

Sendo assim, o objetivo inicial era apenas um: aplicar alguma técnica de computação paralela na fase de otimização dos indivíduos para reduzir o tempo de execução, mantendo inalteradas as características evolutivas. Além disso, pretendia-se aplicar esta técnica aos algoritmos já existentes de (Buriol 2000) e (Mendes 1999), que, embora muito semelhantes na concepção, apresentavam diferenças importantes. Com o trabalho de (Mendes, França & Moscato 2001), que apresenta um "*Framework*" para problemas de otimização combinatória, os algoritmos para resolver o problema de SMS e SMP (de (Mendes 1999)) foram reunidos nessa estrutura, tornando ainda mais adequada a aplicação de uma "camada" que proporcionasse a execução em um ambiente paralelo.

Com o intuito de aumentar o desempenho do AM clássico, inerentemente paralelo sob certo aspecto, pensou-se em agregar mais poder de processamento para realizar a tarefa

[1] Os problemas de seqüenciamento de tarefas abordados foram o seqüenciamento de tarefas em máquina simples (SMS) e o seqüenciamento de tarefas em máquinas paralelas (SMP) .

de otimização dos indivíduos (operador de busca local). Como os códigos de (Mendes et al. 2001) e (Buriol 2000) estavam implementados na linguagem Java^{TM} (Sun microsystems), certamente seria mais prático continuar com ela. Quanto ao *hardware* disponível, havia somente uma rede, tecnologia *Ethernet*, com 8 computadores interligados. Sendo assim, era preciso simular um computador paralelo através de funções de *software*, sempre levando em conta o compromisso entre simplicidade e eficiência. Dessa forma não seria possível utilizar sofisticados mecanismos para realizar a comunicação entre os computadores porque ocasionariam um custo computacional inaceitável. Optou-se, então, por efetuar a comunicação através de *sockets*, por se tratar de um mecanismo prático e simples para troca de mensagens.

A seguir descreve-se a estrutura do algoritmo memético paralelo mestre-escravo (AMPME) implementado neste trabalho.

3.1 Algoritmo Memético Paralelo Mestre-Escravo

Como quer-se apenas distribuir o processamento da tarefa de otimização dos indivíduos, ficando as outras fases do algoritmo (figura 2.7) inalteradas, o modelo mais adequado é o AEPG.

Segundo o conceito de AEPG, descrito na seção 2.2.1, há uma unidade mestre que distribui uma determinada tarefa do algoritmo, nesse caso a otimização dos indivíduos, para as unidades escravo. Mais especificamente, a otimização de cada indivíduo constitui-se numa tarefa a ser distribuída para as unidades escravo. Cada uma destas unidades recebe a tarefa, faz o seu processamento e retorna o resultado (indivíduo otimizado) para a unidade mestre.

Conforme citado anteriormente, o *hardware* paralelo disponível é uma rede de computadores, portanto com várias unidades processadoras e cada uma com a sua memória independente. Para manter-se um algoritmo que centralize todas as tarefas e distribua somente a otimização dos indivíduos é preciso simular uma memória compartilhada, figura 3.1, ou seja, todos os indivíduos a serem otimizados devem ser deixados numa memória compartilhada com todas as unidades processadoras.

Essa simulação é realizada com *sockets*. No entanto, devido à estrutura dessa comunicação, cada *socket* só pode ser utilizado para dois pontos, isto é, entre duas unidades processadoras. Assim são necessários tantos *sockets* quantos forem as unidades escravo envolvidas, conforme ilustra a figura 3.2. Outro problema a ser resolvido é a distribuição das tarefas para as unidades escravo (*socket*). Uma característica determinante é que as tarefas são "montadas" uma após a outra, devido à centralização de tais atividades na unidade mestre. Como pode haver heterogeneidade quanto a natureza de cada uma dessas unidades[2], o objetivo é distribuir as tarefas de modo a minimizar o tempo necessário para processar todas elas. Formalmente tem-se m unidades escravo M_i, $(i = 1, ..., m)$, p tarefas T_j, $(j = 1, ..., p)$, cada uma com o tempo de processamento correspondente Tp_{ji}.

[2]É provável que existam computadores mais rápidos que outros, com diferentes configurações de memória.

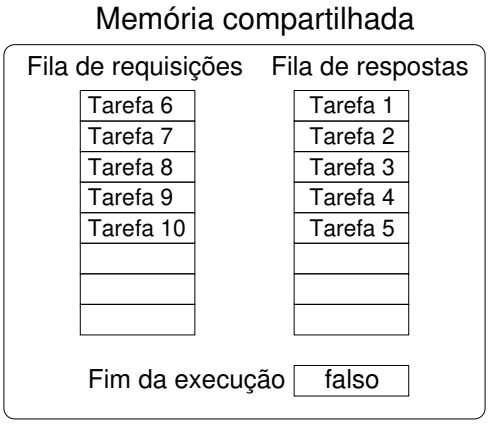


Figura 3.1: Memória compartilhada (**Mc**) para o **AMPME**.

Esse problema se assemelha muito ao SMP com a minimização do *makespan* ($P||C_{max}$). Como o $P||C_{max}$ é NP-hard, torna-se custoso resolvê-lo de forma exata. Além disso, dado que esse problema se repete a cada geração do algoritmo, toda e qualquer economia em se tratando de esforço computacional é valiosa.

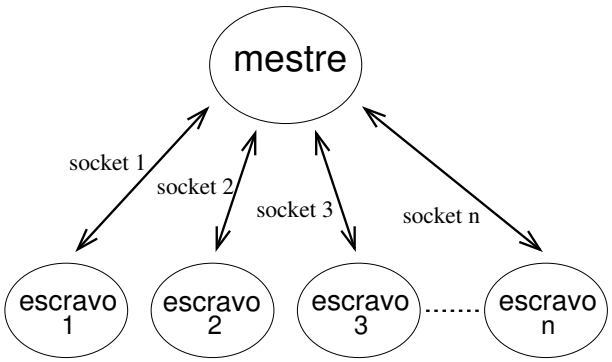


Figura 3.2: *Sockets* no **AMPME**.

A solução proposta é tirar proveito da forma seqüencial com que as tarefas são geradas, alocando-as sob demanda, ou seja, quanto mais rápida for uma unidade escravo, mais tarefas ela irá processar. Contudo esta função não fica a cargo do algoritmo: utilizando *threads* é possível ter na unidade mestre programas concorrentes[3] que compartilham memória. Logo a decisão sobre qual destes estará em execução em um determinado momento será do sistema operacional. O único compromisso é gerenciar essa memória compartilhada (Mc) de modo a evitar inconsistência de dados[4]. Para preservar a ordem de envio e recebimento, essa memória é formada por duas filas[5]: a primeira é reservada para as tarefas que devem ser processadas (fila de requisições) e a segunda armazena as que já o foram (fila de respostas), conforme mostra a figura 3.1.

[3]Cada destes programas concorrentes executa algoritmo descrito na figura 3.4.

[4]Evitar que um determinado programa seja interrompido quando realiza qualquer operação nessa memória compartilhada.

[5]Essas filas são do tipo **FIFO (first in, first out)**.

Um esquema simplificado da arquitetura proposta para o AMPME é ilustrado pela figura 3.3. Verifica-se que cada unidade escravo associada necessita de três componentes: um programa servidor (figura 3.4), residente na unidade mestre, um programa cliente (figura 3.5), residente na unidade escravo, e uma conexão (*socket*). Todos os programas servidores existentes serão *threads* na unidade mestre, portanto programas concorrentes. Assim todos eles disputarão acesso exclusivo à Mc, o qual é garantido pelo uso de uma região de exclusão mútua, com os mecanismos próprios da linguagem de programação adotada.

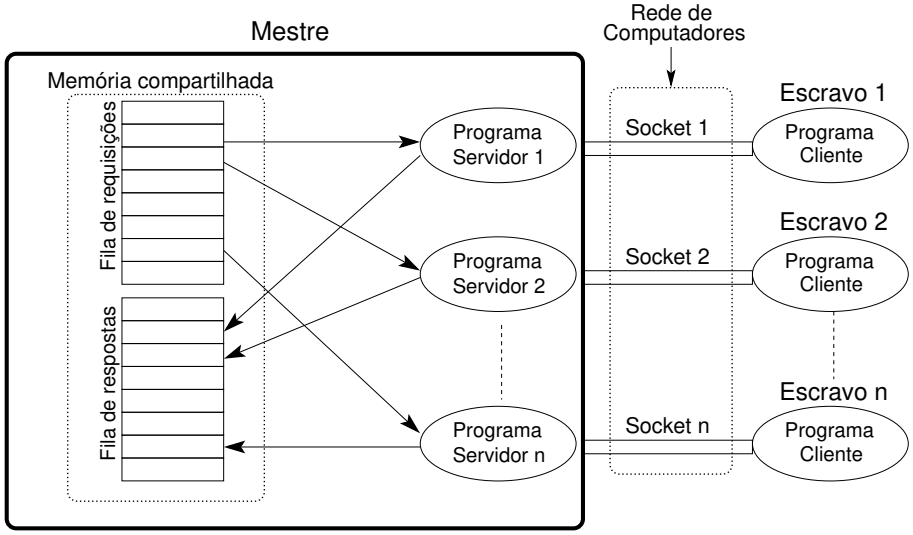


Figura 3.3: Esquema simplificado do **AMPME**.

A partir na inicialização da unidade mestre, um programa servidor, figura 3.4, é inicializado para cada unidade escravo disponível. Nesta unidade é disparado um programa cliente, figura 3.5, que estabelece uma conexão (figura 3.6) e espera por uma tarefa para ser processada. Assim que isso ocorre, o processamento da mesma é realizado, sendo devolvida a seguir para o respectivo programa servidor.

O programa servidor, algoritmo da figura 3.4, verifica permanentemente a Mc a fim de obter uma tarefa para ser processada, enquanto o fim de execução não ocorre, passo 2. O passo 3 é responsável pela leitura da memória compartilhada. Nessa etapa é verificado se há alguma tarefa a ser processada disponível na fila de requisições. Em caso afirmativo, a tarefa é removida de lá e enviada para o respectivo escravo, passo 5. Caso contrário, o algoritmo fica bloqueado no passo 3. O retorno de uma tarefa "vazia" corresponde ao fim da execução, sinalizado pelo algoritmo principal da unidade mestre com o campo "fim da execução" da Mc igual a "verdadeiro". Para que o programa cliente encerre de forma correta, a tarefa "vazia" é enviada para o mesmo, passo 9. O passo 6 é responsável pela recepção da tarefa do respectivo escravo. Assim que a recepção da tarefa ocorre, a mesma é posta na fila de respostas da Mc no passo 7.

Todos esses acessos a Mc são realizados de forma exclusiva, ou seja, somente um servidor está acessando a Mc em um determinado instante.

O programa cliente inicializa a conexão com o respectivo programa servidor no passo 1.

Entrada: Número da porta **Pt** e memória compartilhada (**Mc**)

```
ServidorMe(Pt, Mc)
 1  InicializaServidor(Pt);
 2  while VerificaFimDaExecucao(Mc) = falso
 3  do tarefa ← BuscaTarefa(Mc);
 4     if tarefa! = vazia
 5       then EnviaTarefa(tarefa);
 6            tarefaProcessada ← RecebeTarefa();
 7            InsereTarefa(tarefaProcessada, Mc);
 8       else
 9            EnviaTarefa(tarefa);
10  return ;
```

Figura 3.4: Programa servidor da unidade mestre.

Entrada: Número da porta **Pt**

```
ClienteMe(Pt)
 1  InicializaCliente(Pt);
 2  fimDaExecucao ← falso;
 3  while fimDaExecucao = falso
 4  do tarefa ← RecebeTarefa();
 5     if tarefa! = vazia
 6       then tarefaProcessada ← ProcessaTarefa(tarefa);
 7            EnviaTarefa(tarefaProcessada);
 8       else
 9            fimDaExecucao ← verdadeiro;
10  return ;
```

Figura 3.5: Programa cliente da unidade escravo.

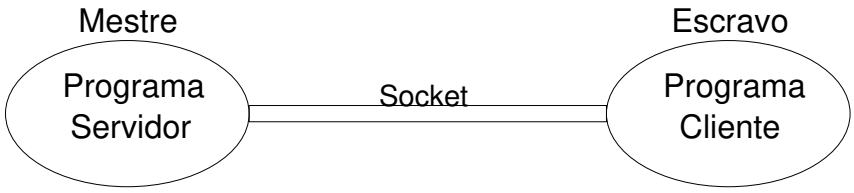


Figura 3.6: Comunicação mestre-escravo.

O laço que inicia no passo 3 é repetido tantas vezes quantas forem as tarefas a serem processadas. A seguir o programa cliente fica bloqueado no passo 4 até que seja possível ler uma tarefa. Assim que isso ocorre, caso a tarefa não seja vazia, a mesma é processada no passo 6 e enviada no passo 7. A recepção de uma tarefa vazia no passo 4 encerra a execução, passos 5 e 9.

Esse processo de formar as tarefas para serem processadas pelos escravos se repete a cada geração. Na fase de otimização dos indivíduos, cada um deles será uma tarefa com a especificação do respectivo algoritmo de busca local a ser executado. Por isso é necessário um certo sincronismo, ou seja, após formadas as tarefas e disponibilizadas na Mc, é necessário esperar[6] pelo resultado (indivíduos otimizados). Dessa forma, quando o mestre termina de formar todas as tarefas, o mesmo deve aguardar (ou não) o resultado para prosseguir com a próxima fase do AM. Com o objetivo de agregar uma funcionalidade e tornar o algoritmo mais flexível, foi estabelecido um parâmetro k para determinar quantas tarefas devem ser aguardadas antes que o algoritmo prossiga com o próximo passo. O valor de k corresponde a percentagem das tarefas enviadas. Num caso em que a população é formada por 13 indivíduos e são gerados 20 novos indivíduos a cada geração, tem-se 20 tarefas a serem executadas pelas unidades escravo. Um valor igual a 0.4 (40%) para k significa que será aguardado o retorno de pelo menos 8 tarefas (indivíduos otimizados) antes de proceder com a próxima fase do algoritmo. Assim, um valor $k = 0$ corresponde ao algoritmo mestre-escravo totalmente assíncrono e $k = 1$ ao totalmente síncrono. Em todos os testes computacionais utilizou-se $k = 1$.

Para tornar o algoritmo mais robusto no que se refere à falhas causadas pela perda de conexão com as unidades escravo durante a execução, foi agregada outra funcionalidade para verificar as conexões ativas a cada geração. Se uma conexão é "perdida", é feita a tentativa de reinicializá-la. Não sendo possível, a mesma é descartada e o algoritmo prossegue com as conexões restantes. Caso a tarefa processada não seja recebida pelo programa servidor da respectiva conexão "perdida", a mesma é recolocada na fila de requisições da Mc para que seja enviada para o processamento em outro escravo disponível.

O pseudo-código completo com todos os passos do AMPME é ilustrado pela figura 3.7. Se for comparado com o AM seqüencial, figura 2.7, verifica-se que são poucas as modificações necessárias para torná-lo paralelo (somente os passos 12, 13 e 14 foram adicionados). No passo 12 é realizada a montagem da tarefa, isso significa incluir o indivíduo gerado e a especificação do algoritmo de busca local a ser executado. Após isso, também no passo 12, essa tarefa é incluída na fila de requisições da Mc (figura 3.1). O passo 13 é responsável por bloquear a execução do algoritmo até que as $k * individuosGerados$ tarefas estejam disponíveis na memória compartilhada[7], ou seja, será esperado um percentual das tarefas geradas (individuosGerados) especificado pelo parâmetro k. A tentativa de inserção dos novos indivíduos na população é realizada no passo 15.

Para uma boa compreensão da paralelização da fase de otimização, exemplifica-se, com a figura 3.8, o transcorrer de uma geração, baseado na unidade mestre. Há três

[6]Pela seção 2.2.1, o algoritmo pode ser síncrono ou assíncrono.

[7]Essas tarefas devem estar disponíveis na fila de respostas.

Entrada: População de indivíduos **P**, lista dos computadores a serem utilizados como **escravos Lc** e taxa de assincronismo (k)
Saida: População de indivíduos **P**

```
AlgMemeticoParaleloMestreEscravo(P, Lc, k)
 1  InicializeMestreEscravo(Lc);
 2  Inicialize(P);
 3  Avalie(P);
 4  geracoes ← 0;
 5  while Criteriodeparada = falso
 6  do individuosGerados ← 0;
 7      while individuosGerados < limite
 8      do pais ← SelecioneIndividuos(P);
 9          novoIndividuo ← RecombineIndividuos(pais);
10          RealizeMutacao(novoIndividuo);
11          individuosGerados ← individuosGerados + 1;
12          MonteTarefaParaEscravos(novoIndividuo);
13      NovosIndividuos ← EspereTarefasProcessadas(k);
14      for i ← 1 to Tamanho(NovosIndividuos)
15      do Adicione(NovosIndividuos[i]);
16      geracoes ← geracoes + 1;
17  return P;
```

Figura 3.7: AM Paralelo Mestre-escravo principal.

unidades escravo idênticas[8] e 6 tarefas a serem processadas. O tempo *To_m* representa o tempo que a unidade mestre fica ociosa e *To_e i* representa o tempo que o escravo *i* fica ocioso. Para facilitar a análise assume-se que o tempo de uma geração corresponde somente à fase de otimização dos indivíduos, desprezando-se os passos 8, 9, 10, 11, 14, 15 e 16 da figura 3.7. Nas equações citadas a seguir, Nt corresponde ao número de tarefas (ou indivíduos) a serem distruídos para os escravos, Ne é o número de escravos a ser utilizado pelo algoritmo paralelo e Tc representa o tempo para transmitir uma tarefa do mestre para o escravo e vice-versa.

Analisando-se a figura 3.8 é possível perceber que o tempo de uma geração para a unidade mestre (Tg_p) só é expresso em função de Tc e To_m, porque a única tarefa a ser realizada é o envio e o recebimento das tarefas. Como toda tarefa necessita de um Tc para ser enviada e um Tc para ser recebida, o tempo total, para todas as tarefas, corresponde a $2.Tc.Nt$, exatamente a primeira parcela da equação 3.1. A outra parcela, que envolve To_m (equação 3.2), corresponde ao número de tempos ociosos da unidade mestre: $Tp - Tc.(Ne - 1) = Tp - Tc.Ne + Tc$. Assim, com as duas parcelas tem-se o tempo total de uma geração para a unidade mestre, Tg_p, expresso pela equação 3.3, que pode ser reescrita conforme a equação 3.4. O tempo de uma geração para o algoritmo seqüencial (Tg_s), sem a distribuição da fase de otimização, é expresso pela equação 3.5.

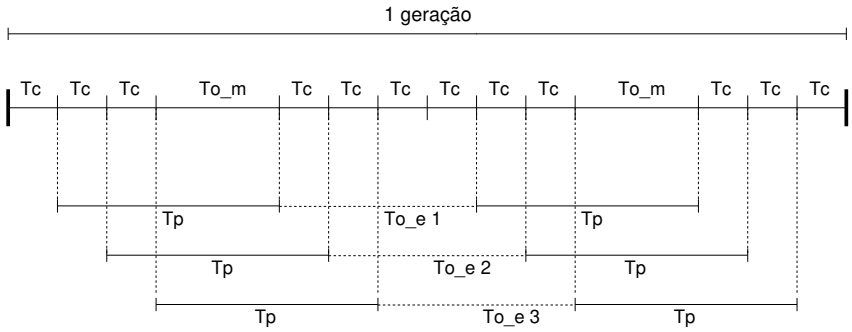


Figura 3.8: Análise da execução do **AMPME**.

$$Tg_p = 2.Tc.Nt + \frac{Nt}{Ne}.To_m \tag{3.1}$$

$$To_m = Tp - Tc.(Ne - 1) = Tp - Tc.Ne + Tc \tag{3.2}$$

$$Tg_p = 2.Nt.Tc + \frac{Nt}{Ne}.(Tp - Tc.Ne + Tc) \tag{3.3}$$

$$Tg_p = \frac{Nt.Tp}{Ne} + \frac{Nt.Tc}{Ne}.(1 + Ne) \tag{3.4}$$

$$Tg_s = Nt.Tp \tag{3.5}$$

[8]O tempo para processar cada tarefa (**Tp**) é o mesmo em todas as unidades escravo.

Para que o algoritmo paralelo tenha melhor desempenho[9] que o algoritmo seqüencial, $\frac{Tg_s}{Tg_p} > 1$. Fazendo algumas operações matemáticas nas equações 3.4 e 3.5 obtém-se a equação 3.6, que demonstra que o tempo de processamento da tarefa Tp deve ser sempre maior que Tc e ainda que quando maior for a razão $\frac{Tp}{Tc}$ melhor será o desempenho.

$$Tp > Tc.\frac{Ne+1}{Ne-1} \tag{3.6}$$

A equação 3.6 demonstra que o ganho em desempenho, com a paralelização de uma determinada fase do algoritmo, ocorrerá somente se Tp for maior que Tc.

Uma importante conclusão sobre a aplicação de AEPG é que só haverá ganho de desempenho se o tempo de processamento da tarefa que se quer distribuir for maior que o tempo de comunicação. Um bom exemplo de uma fase que não apresenta ganho caso seja distribuída é a fase de recombinação. Em geral, o tempo para efetuar uma recombinação é menor que o tempo necessário para enviar os indivíduos-pais e receber o indivíduo-filho.

[9]Em se tratando de tempo computacional.

Capítulo 4

O Problema de Seqüenciamento de Tarefas em Máquina Simples

O problema de seqüenciamento de tarefas em máquina simples (SMS) é um dos mais estudados problemas de otimização combinatória. O interesse provém da freqüência com que é encontrado em problemas práticos de engenharia industrial. Os trabalhos de (Graham, Lawler, Lenstra & Rinooy Kan 1979) e (Graves 1981) são considerados pioneiros e introdutórios para esta classe de problemas na área de Seqüenciamento.

A configuração das máquinas envolvidas no seqüenciamento caracteriza o tipo de problema: uma única máquina ou múltiplas. Em se tratando de múltiplas máquinas elas podem ser idênticas ou não, em paralelo ou em *shops*.

Há diferentes tipos de problemas de SMS em decorrência da natureza dos dados de entrada e da função objetivo. Um dos problemas mais encontrados na literatura é o seqüenciamento de n tarefas, dado um tempo de processamento e uma data de entrega para cada uma delas. A função objetivo minimiza o atraso total, caracterizado pela soma dos atrasos em relação à data de entrega de cada tarefa. Esse problema pode ser estendido com a inclusão de tempos de preparação de cada tarefa, que podem ser dependentes da seqüência, restrições de precedência, entre outros.

O problema de SMS mais simples, sem tempos de preparação, já é NP-hard, como demonstra (Du & Leung 1990). Muitas técnicas foram propostas para sua resolução, com algumas variações em relação ao problema adotado neste trabalho. (Raman, Rachamadugu & Talbot 1989) e (Lee, Bhaskaran & Pinedo 1997) usam regras de despacho baseadas no cálculo de um índice de prioridade para construir uma seqüência inicial, a ser otimizada posteriormente por um procedimento de busca local. (Rubin & Ragatz 1995) criaram um novo operador de recombinação e aplicaram AG. Ainda (Tan & Narasimhan 1997) desenvolveram um método que utiliza *Simulated Annealing*. (França, Mendes & Moscato 2000) propuseram um AM com população estruturada de forma hierárquica.

A seguir é descrito o problema que será tratado neste capítulo.

4.1 Definição do Problema

O problema de SMS a ser tratado neste trabalho pode ser assim descrito:

- **Entrada**: são dadas n tarefas a serem processadas em uma máquina, uma lista de $\{t_1,\ldots,t_n\}$ de tempos de processamento e outra $\{d_1,\ldots,d_n\}$ com as datas de entrega das respectivas tarefas. É dada ainda uma matriz $\{s_{ij}\}$ de tempos de preparação, onde s_{ij} é o tempo de preparação da tarefa j depois da máquina ter processado a tarefa i.

- **Objetivo**: encontrar uma permutação das n tarefas que minimize o atraso total do seqüenciamento. Este atraso é dado pela equação 4.1, onde c_k representa o instante em que a tarefa k terminou de ser processada e d_k a data de entrega da mesma.

$$\sum_{k=1}^{n} max[0, c_k - d_k] \tag{4.1}$$

Os tempos de preparação são dependentes da seqüência em que as tarefas são processadas. Em geral $s_{ij} \neq s_{ji}$.

A figura 4.1 ilustra uma possível seqüência para uma instância com cinco tarefas (*1-4-3-2-5*), assim como a representação gráfica do atraso total.

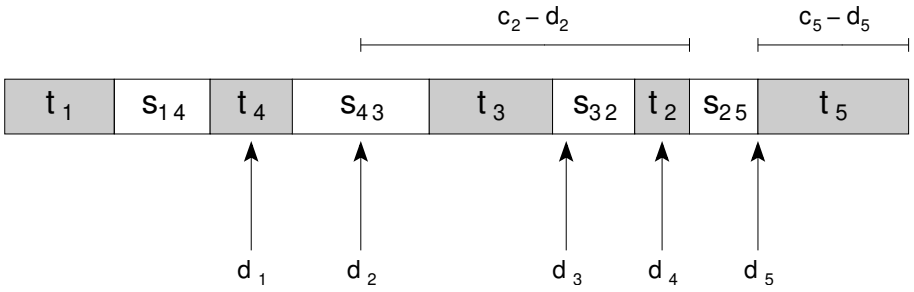


Figura 4.1: Gráfico de Gantt de uma solução para o problema de **SMS**.

4.2 Algoritmo para Resolução

A seguir serão descritas somente as especificações mais importantes do algoritmo proposto, já incluído no *Framework* de (Mendes et al. 2001). A escolha das mesmas foi baseada nos trabalhos de (França, Mendes & Moscato 2000), (Mendes 1999) e (Mendes, França & Moscato 2000).

4.2.1 Representação

Como já foi mencionado no capítulo 2, a representação genética escolhida tem uma grande influência no sucesso do algoritmo evolutivo. São desejáveis representações compactas, completas e estáveis (Mendes 1999).

A representação adotada para o problema de SMS é bem simples e intuitiva: a solução é representada por uma permutação de n (número de tarefas da instância do problema) elementos inteiros e distintos, com valores no intervalo [1,n]. Qualquer solução pode ser mapeada para esta representação e um exemplo para uma instância de 10 tarefas pode ser $< 1, 4, 7, 9, 2, 3, 5, 8, 6, 10 >$.

4.2.2 Estrutura da População

A estrutura populacional escolhida possui uma única população estruturada hierarquicamente, segundo uma árvore ternária completa de três níveis (Mendes 1999). Dessa forma é possível dividí-la em subgrupos de 4 indivíduos, cada um composto por um líder e três seguidores (figura 4.2). O líder é sempre o indivíduo mais adaptado[1] dos quatro. Isso faz com que os níveis superiores possuam os melhores indivíduos, estando o melhor indivíduo localizado no nó raiz da árvore. A árvore utilizada é completa de três níveis, portanto composta de 13 indivíduos.

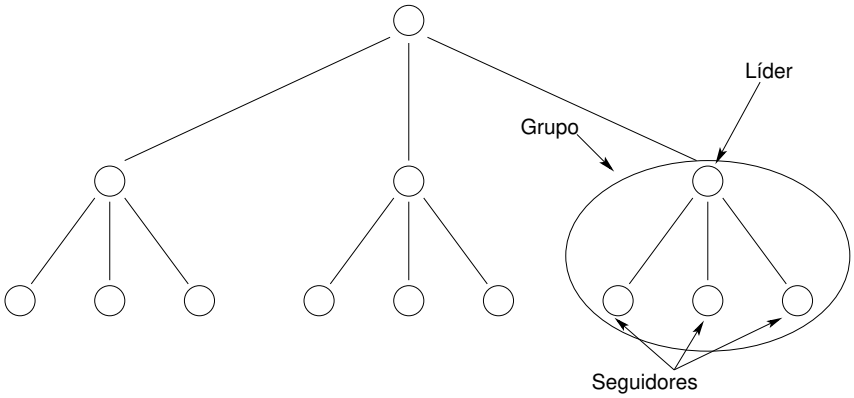


Figura 4.2: Estrutura da população em árvore ternária.

4.2.3 Recombinação

O operador de recombinação adotado (*crossover OX–order crossover* , definido por (Goldberg 1989)) sempre gera um filho a partir de dois pais. O procedimento se inicia com a escolha desses dois pais. Uma vez escolhidos, um fragmento de um deles é selecionado aleatoriamente, com disitribuição uniforme, e copiado para o filho. Esta etapa tende a preservar boas posições absolutas e, pelo fato do fragmento ser contíguo, preserva também as posições relativas ocupadas pelas tarefas na seqüência. As posições não-preenchidas do filho são então completadas com a sequência das tarefas do outro pai, varrendo-se o cromossomo deste no sentido esquerda-direita. Esta segunda etapa procura preservar a posição relativa das tarefas. A figura 4.3 ilustra esse procedimento.

Neste exemplo, o fragmento 7-8-6-9 foi selecionado do pai A e copiado na mesma posição do filho. As posições não completadas foram então preenchidas segundo a sequência

[1]Melhor solução.

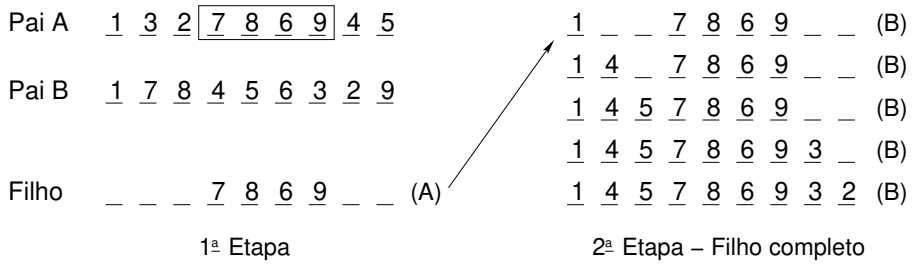
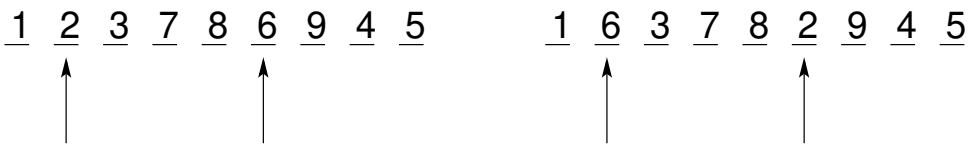


Figura 4.3: Procedimento de recombinação com o *crossover OX*.

das tarefas do pai B.

4.2.4 Mutação

O mecanismo de mutação adotado consiste na troca de posição de dois alelos do cromossomo de um indivíduo. As duas posições são geradas aleatoriamente. Um exemplo de uma possível mutação é dado pela figura 4.4.



(a) Cromossomo antes da mutação.

(b) Cromossomo após a mutação.

Figura 4.4: Exemplo de uma mutação.

4.2.5 Função de Avaliação

A função de avaliação tem por objetivo discriminar os indivíduos da população atribuindo para cada um deles um valor. Este valor irá determinar o quão adaptado ao ambiente um certo indivíduo está. Para isso, essa função deve estar estreitamente relacionada com a função objetivo do problema.

No algoritmo proposto foi adotada uma função bem simples, do trabalho de (Mendes 1999), e muito citada na literatura (Rubin & Ragatz 1995)(Cheng & Gen 1997), dada pela equação 4.2, onde T_i é o atraso total de uma solução i, calculado pela equação 4.1. O número 1 foi somado a T_i, na equação 4.2 para evitar uma divisão por zero, isto porque T_i pode ser nulo (nenhuma tarefa apresenta atraso).

$$f_i = (T_i + 1)^{-1} \tag{4.2}$$

4.2.6 Seleção

O mecanismo de seleção irá selecionar os pais para a recombinação. Como a recombinação adotada necessita de somente dois pais para dar origem a um filho, o primeiro pai é sempre um líder da população. O outro estará restrito ao subgrupo do líder, sendo, portanto, um dos três seguidores. A escolha de um destes três seguidores também é aleatória, com distribuição uniforme.

O novo indivíduo somente será inserido na população se o mesmo tiver um valor de função de avaliação melhor que um de seus dois pais, ocupando, portanto, o lugar do respectivo. Caso contrário o mesmo será descartado.

Devido a essa pressão seletiva, a perda de diversidade será inevitavelmente rápida. Para contornar esse problema, (Mendes 1999) sugere que se mantenha a unicidade dos indivíduos, ou seja, caso um novo indivíduo seja idêntico a um de seus pais, o mesmo é descartado.

4.2.7 Busca Local

Os algoritmos de busca local se baseiam na definição de uma vizinhança que estabelece uma relação entre soluções no espaço de busca. A idéia básica é estabelecer um movimento que irá caracterizar o algoritmo. A vizinhança de uma determinada solução será composta do conjunto de todas as soluções obtidas com a execução desse movimento.

As duas vizinhanças adotadas para esse problema foram "*all-pairs*" e *inserção*. A primeira é baseada na troca ordenada de todos os pares de tarefas de uma dada solução. A vizinhança *all-pairs* de uma dada solução com n tarefas será composta de todos as soluções obtidas com a troca de posição de todos os pares (i, j), com $i \neq j$. Isto irá gerar uma vizinhança de tamanho $\frac{n(n-1)}{2}$. Por exemplo, dado a seqüência $< DEFGHIJK >$, verificam-se todas as trocas possíveis e avalia-se o valor de solução que cada um ocasionaria[2]. Suponha que a troca entre o par (D, E) seja a que ocasione o melhor valor de solução, então a solução resultante será $< EDFGHIJK >$. A partir daí o processo se repete até que, para uma dada solução, nenhuma troca ocasione uma solução melhor que ela. Conclui-se, então, que esta solução é um *mínimo local* relativo a vizinhança *all-pairs*.

A outra vizinhança é a de *inserção*. O movimento desta consiste em, dada uma solução, retirar uma tarefa da sua posição original e inseri-la em outra. Assim, para cada tarefa há $n-1$ possibilidades de inserção, o que ocasiona um tamanho de vizinhança igual ao da anterior, $\frac{n(n-1)}{2}$.

Como as duas vizinhanças possuem complexidade $O(n^2)$, é necessário investir em mecanismos que reduzam a vizinhança, pois o custo[3] para avaliar todas as soluções vizinhas pode vir a comprometer a eficiência do algoritmo.

[2]Na verdade a troca é realizada para que a solução resultante seja avaliada.

[3]Custo de avaliação do indivíduo.

4.2.7.1 Reduções para a vizinhança *all-pairs*

Todas as reduções de vizinhança implementadas utilizam a informação de como os tempos de preparação serão alterados caso a troca seja confirmada.

A figura 4.5 mostra os tempos de preparação e os tempos de processamento que influenciarão o atraso total caso as tarefas i e j sejam trocadas. Os valores V's estão descritos em função das variáveis s e p e o critério que decide se a troca será ou não avaliada baseia-se neles. A figura 4.6 ilustra a configuração das tarefas após a troca.

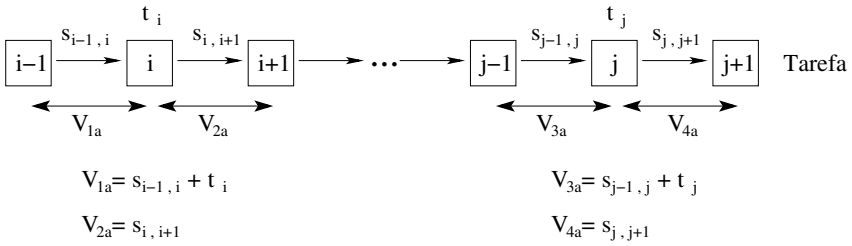


Figura 4.5: Configuração dos tempos de preparação e de processamento antes da troca das tarefas.

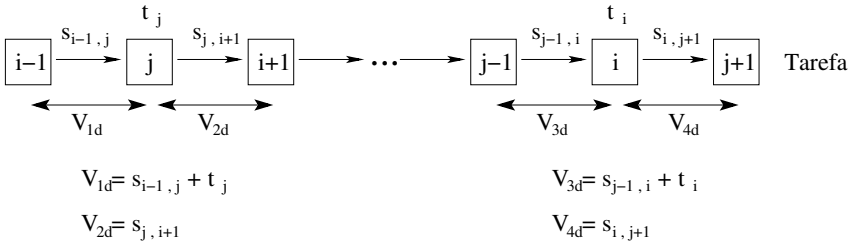


Figura 4.6: Configuração dos tempos de preparação e de processamento após a troca das tarefas.

A decisão se uma troca deverá ou não ser avaliada leva em conta a variação dos V's. Se eles melhoram com a troca, a solução deve ser avaliada. Como existem quatro valores de V envolvidos, há várias combinações possíveis. A tabela 4.1 ilustra as duas políticas de decisão.

Política	*Avaliação*
T_1	$(V_{1d} < V_{1a}) \vee (V_{2d} < V_{2a}) \vee (V_{3d} < V_{3a}) \vee (V_{4d} < V_{4a})$
T_2	$[(V_{1d} < V_{1a}) \vee (V_{2d} < V_{2a})] \wedge [(V_{3d} < V_{3a}) \vee (V_{4d} < V_{4a})]$

Tabela 4.1: Políticas de avaliação para as reduções da vizinhança *all-pairs*.

A política T_1 exige que qualquer um dos valores de V diminua para a troca ser avaliada. Na política T_2, a troca será avaliada se ao menos dos valores de V, para i e j, for reduzido.

A política T_1 é a menos restritiva, reduzindo a vizinhança *all-pairs* em pouco menos de 50%. A outra, T_2, mais restrita, reduz a vizinhança em 90%. Esses valores para a redução são valores médios obtidos empiricamente.

4.2.7.2 Reduções para a vizinhança de *inserção*

As figuras 4.7 e 4.8 descrevem as configurações das tarefas antes e depois da inserção, respectivamente.

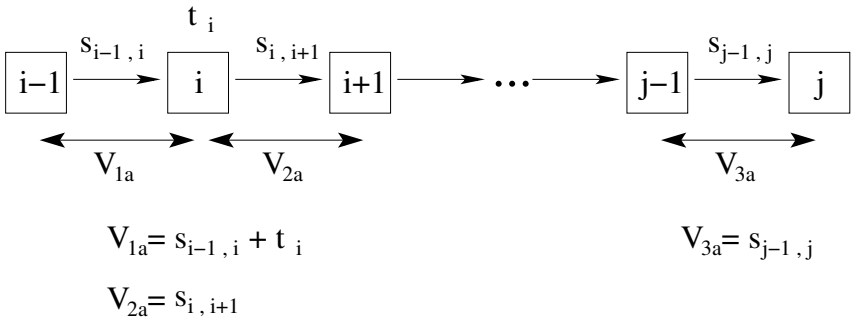


Figura 4.7: Configuração dos tempos de preparação e de processamento antes da inserção.

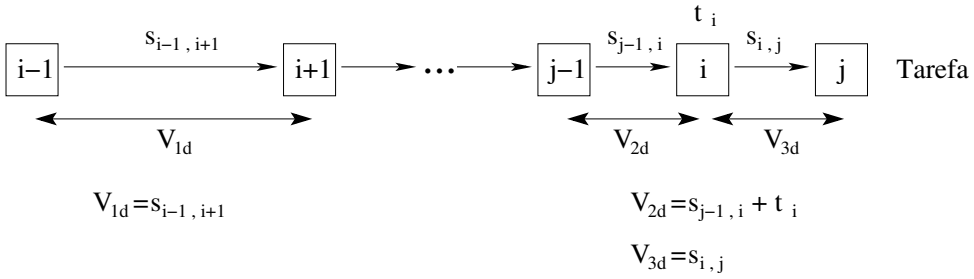


Figura 4.8: Configuração dos tempos de preparação e de processamento após a inserção.

Na figura 4.7 consta a configuração antes da tarefa i ser inserida imediatamente antes da tarefa j. Tem-se apenas três valores para V porque a tarefa $j+1$ não é relevante para a avaliação do movimento. A figura 4.8 ilustra a configuração após a inserção da tarefa i. As políticas de avaliação são ilustradas pela tabela 4.2.

Política	*Avaliação*
I_1	$(V_{1d} < V_{1a} + V_{2a}) \vee (V_{2d} + V_{3d} < V_{3a})$
I_2	$(V_{1d} < V_{1a} + V_{2a}) \wedge (V_{2d} + V_{3d} < V_{3a})$

Tabela 4.2: Políticas de avaliação para as reduções da vizinhança de *inserção*.

As políticas neste caso também dividem os V's em grupos: o grupo da região da tarefa i e o grupo da região da tarefa j. A política I_2 exige que em ambas as regiões o valor dos V's melhore. No caso de serem dois os valores de V, compara-se a soma deles com o valor anterior. A política I_1, por sua vez, exige que a melhora ocorra em pelo menos uma das duas regiões. O esquema I_2 é extremamente restritivo, reduzindo a vizinhança de *inserção* em mais de 90%. Já no I_1 essa redução é de aproximadamente 50%.

4.2.8 Algoritmo Mestre-escravo

O algoritmo mestre-escravo implementado para este problema é idêntico ao descrito na seção 3.1.

4.3 Resultados Computacionais

Conforme citado no capítulo 3, teve-se como objetivo melhorar o desempenho[4] do algoritmo seqüencial de (Mendes 1999) com a implementação do algoritmo mestre-escravo.

Na implementação do algoritmo mestre-escravo para o SMS foi paralelizada apenas a etapa de busca local, onde se concentra o maior esforço computacional para resolver o problema. Como havia duas vizinhanças implementadas e duas reduções para cada uma, investigou-se qual era o efeito que a redução tinha no tempo da busca local e, portanto, no desempenho do algoritmo mestre-escravo. Dessa forma são utilizadas as duas vizinhanças e suas respectivas reduções.

Em todos os testes é fixado um número de gerações, no caso as 20 iniciais, e é executado o algoritmo seqüencial. A seguir executa-se o mesmo algoritmo com a paralelização da etapa de otimização dos indivíduos (pseudo-código da figura 3.7). Com os dois tempos, Ts referente ao tempo para executar o algoritmo seqüencial e Tp correspondente ao tempo para executar o algoritmo paralelo, obtém-se o *speedup* com a razão $\frac{Ts}{Tp}$. Também determinou-se um número de 10 repetições para cada execução.

As instâncias utilizadas foram geradas a partir de instâncias do problema do caixeiro viajante assimétrico (*ATSP*) (Mendes et al. 2000), disponíveis na TSPLIB[5]. As duas letras que sucedem o nome da instância dizem respeito a dificuldade de resolução da mesma. A primeira se refere à forma como são gerados os tempos de processento, com L denotando ser uma instância mais difícil que H. Para as instâncias L os tempos de preparação tornam-se o aspecto mais crítico no seqüenciamento das tarefas, dando uma característica mais semelhante ao ATSP. Para as instâncias H os tempos de processamento tornam-se mais relevantes no seqüenciamento. A segunda letra está relacionada com as datas de entrega: H denota instâncias que possuem tarefas com datas de entrega exatamente iguais ao instante em que elas acabam de ser processadas ($d_k = c_k$), resultado em um atraso total ótimo igual a zero; S denota instâncias com as datas de entrega (d_k) geradas no intervalo $[c_k - p_k, c_k]$, portanto com um atraso total ótimo diferente de zero.

Nos gráficos a seguir plotou-se duas retas adicionais para facilitar a análise e discriminar melhor as instâncias: a primeira é a do "speedup mínimo", discriminando as instâncias que, para um certo número de processadores, tem desempenho inferior ao algoritmo seqüencial se estiverem abaixo e superior se estiverem acima dela; a reta do "speedup ideal" fornece, para um determinado número de processadores, o *speedup* que seria obtido caso não houvesse custo para distribuir o processamento do algoritmo, ou seja, o "speedup ideal".

[4]Reduzir o tempo computacional.

[5]http://www.crpc.rice.edu/softlib/tsplib/

O primeiro teste foi realizado com a vizinhança *all-pairs* e a política de redução T_1. A tabela 4.3 ilustra o valor de *speedup* obtido, assim como o número de tarefas de cada instância (n), o tempo médio para realizar a busca local em um indivíduo[6] (Tp em ms - $10^{-3}s$ segundos) e o tempo de comunicação[7] (Tc em ms - $10^{-3}s$ segundos). No segundo teste foi utilizada a mesma vizinhança com a política T_2, os resultados são ilustrados pela tabela 4.4. As figuras 4.9 e 4.10 ilustram a evolução do *speedup* para um subconjunto de instâncias escolhido, para o primeiro e segundo teste, respectivamente.

Instâncias	n	*Tp (ms)*	*Tc (ms)*	*Speedup* — *Número de processadores*							
				1	2	3	4	5	6	7	8
ftv55HH	56	27	10	0,00	0,00	0,00	0,18	0,26	0,28	0,29	0,30
ftv55LH	56	76	10	0,00	0,00	0,00	0,00	0,40	0,56	0,74	0,81
ftv70HH	71	82	12	0,00	0,00	0,00	0,73	0,71	0,71	0,75	0,74
ftv70LH	71	175	12	0,00	0,00	0,00	0,64	1,25	1,47	1,10	1,06
kro124pHH	100	356	13	0,86	1,43	1,95	2,56	2,69	2,88	3,32	3,33
kro124pLH	100	641	13	0,95	1,60	2,22	2,66	3,04	3,49	3,78	4,22
rbg323HH	323	28866	20	0,99	1,62	2,65	3,28	3,80	4,55	5,21	5,69
rbg323LH	323	23440	20	0,99	1,79	2,63	3,26	3,83	4,43	4,85	5,38

Tabela 4.3: Resultados para o **AMPME**, vizinhança *all-pairs* e redução T_1.

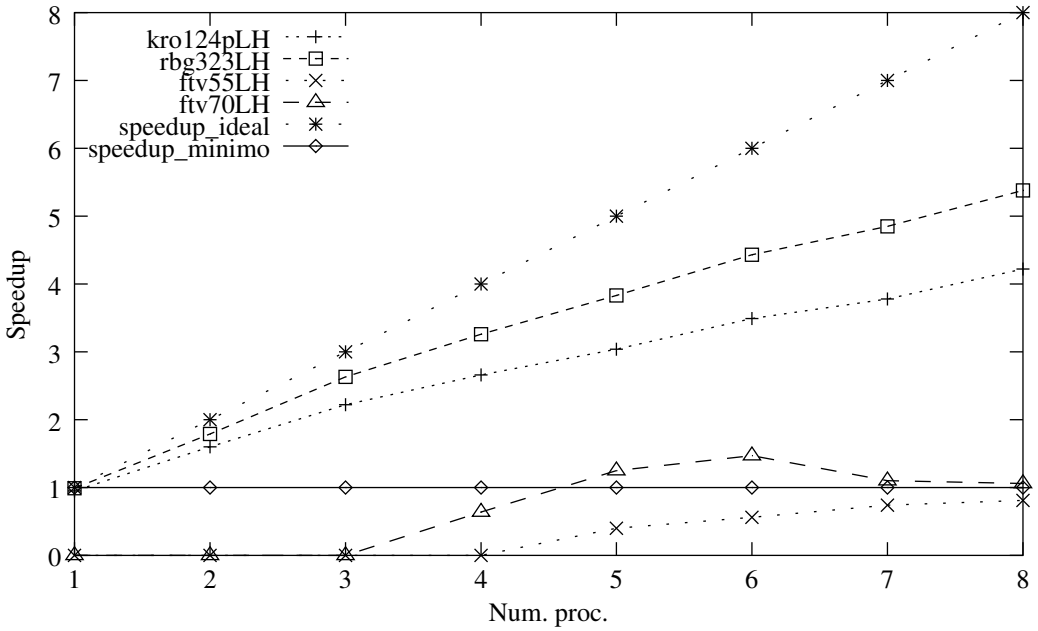


Figura 4.9: Desempenho **AMPME**, vizinhança *all-pairs* e redução T_1.

Para o terceiro e quarto testes faz-se uso da vizinhança de inserção com as políticas de reduções I_1 e I_2, respectivamente. A tabela 4.5 e a figura 4.11 ilustram os resultados

[6]Quando é executado o algoritmo seqüencial.

[7]Tempo para transmitir o indivíduo da unidade *mestre* para a unidade *escravo*.

Instâncias	*n*	*Tp* *(ms)*	*Tc* *(ms)*	*Speedup* *Número de processadores*							
				1	2	3	4	5	6	7	8
ftv55HH	56	62	10	0,74	1,04	1,32	1,48	1,64	1,73	1,77	1,80
ftv55LH	56	22	10	0,51	0,58	0,69	0,78	0,80	0,83	0,80	0,83
ftv70HH	71	145	12	0,84	1,30	1,81	2,05	2,39	2,58	2,79	2,73
ftv70LH	71	45	12	0,63	0,81	1,02	1,17	1,22	1,28	1,33	1,28
kro124pHH	100	355	13	0,92	1,59	2,19	2,84	3,30	3,45	3,80	4,12
kro124pLH	100	118	13	0,81	1,11	1,59	1,89	2,09	2,22	2,47	2,53
rbg323HH	323	26052	20	0,98	1,91	2,68	3,38	3,84	4,23	4,84	5,39
rbg323LH	323	5184	20	0,95	1,72	2,49	3,18	3,64	4,14	4,75	5,34

Tabela 4.4: Resultados para o **AMPME**, vizinhança *all-pairs* e redução T_2.

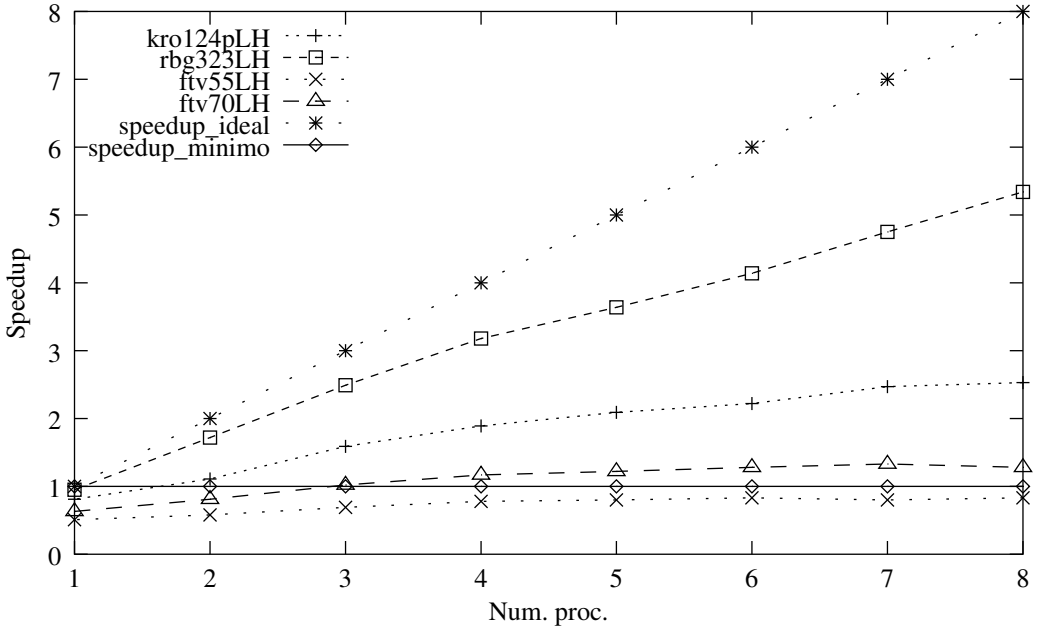


Figura 4.10: Desempenho **AMPME**, vizinhança *all-pairs* e redução T_2.

para o terceiro teste. Os resultados para o quarto teste são ilustrados pela tabela 4.6 e figura 4.12.

Os resultados indicam um comportamento diferenciado para cada uma das vizinhanças, com as respectivas reduções. Na primeira, vizinhança *all-pairs*, a redução mais restrita (T_2) diminui o tempo da busca local, fazendo com que a relação $\frac{Tp}{Tc}$ seja menor que o menos restrito (T_1). Isso ocasionou um *speedup* maior para o caso menos restrito (T_1) em relação ao mais restrito (T_2).

Para a vizinhança de *inserção* ocorreu o contrário. A redução mais restrita aumentou o tempo da busca local, fazendo com que a razão $\frac{Tp}{Tc}$ ficasse maior que o menos restrito (I_1). Conseqüentemente o *speedup* foi maior para a o teste com a segunda redução de vizinhança (I_2).

Esse comportamento diferenciado não é surpresa. Como as vizinhanças têm comportamentos diferenciados, pode ocorrer que uma redução aumente o tempo de busca local

Instâncias	n	Tp *(ms)*	Tc *(ms)*	*Speedup* *Número de processadores* 1	2	3	4	5	6	7	8
ftv55HH	56	289	10	0,82	1,42	1,97	2,31	2,72	2,87	3,20	3,28
ftv55LH	56	189	10	0,81	1,38	1,81	2,12	2,33	2,53	2,73	2,88
ftv70HH	71	576	12	0,93	1,82	2,52	3,03	3,54	3,79	4,20	4,92
ftv70LH	71	365	12	0,90	1,72	2,29	2,76	3,20	3,45	3,84	4,24
kro124pHH	100	2472	13	0,96	1,90	2,83	3,56	3,94	4,49	5,25	5,33
kro124pLH	100	1311	13	0,99	1,97	2,69	3,42	3,92	4,49	4,72	5,52
rbg323HH	323	67759	20	0,99	1,98	2,81	3,62	4,39	5,13	5,92	6,29
rbg323LH	323	16755	20	0,99	1,99	2,91	3,70	4,49	5,12	5,82	6,26

Tabela 4.5: Resultados para o **AMPME**, vizinhança de *inserção* e redução I_1.

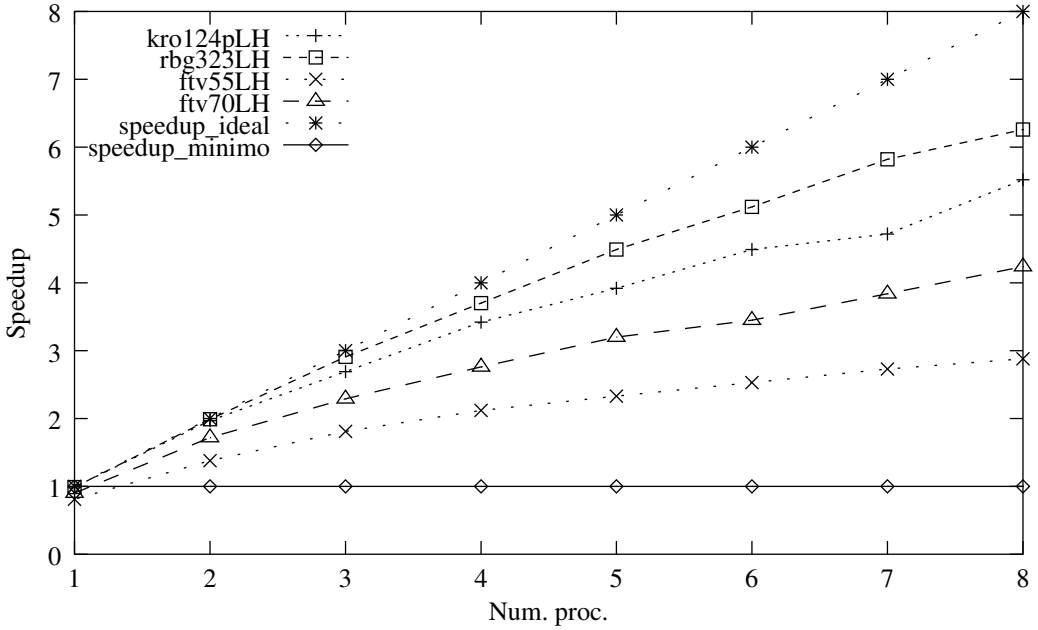


Figura 4.11: Desempenho **AMPME**, vizinhança de *inserção* e redução I_1.

devido a restrição nos movimentos. Isto causa um aumento no tempo e faz com que o AMPME tenha melhor desempenho, conforme mencionado na seção 3.1.

Instâncias	*n*	*Tp* *(ms)*	*Tc* *(ms)*	*Speedup* *Número de processadores*							
				1	2	3	4	5	6	7	8
ftv55HH	56	205	10	0,78	1,49	1,88	2,46	2,78	2,90	3,15	3,30
ftv55LH	56	170	10	0,73	1,34	1,78	2,04	2,20	1,79	2,44	2,02
ftv70HH	71	464	12	0,85	1,65	2,40	2,88	3,38	3,21	3,75	3,92
ftv70LH	71	281	12	0,85	1,63	2,17	2,69	3,09	3,28	3,78	4,15
kro124pHH	100	2229	13	0,94	1,83	2,60	3,38	3,51	3,82	4,88	4,91
kro124pLH	100	1535	13	0,97	1,90	2,61	3,33	3,72	4,11	4,42	5,11
rbg323HH	323	76808	20	0,97	1,92	2,86	3,65	4,43	4,67	5,85	6,92
rbg323LH	323	51255	20	0,98	1,98	2,86	3,61	4,37	5,06	5,35	6,34

Tabela 4.6: Resultados para o **AMPME**, vizinhança de *inserção* e redução I_2.

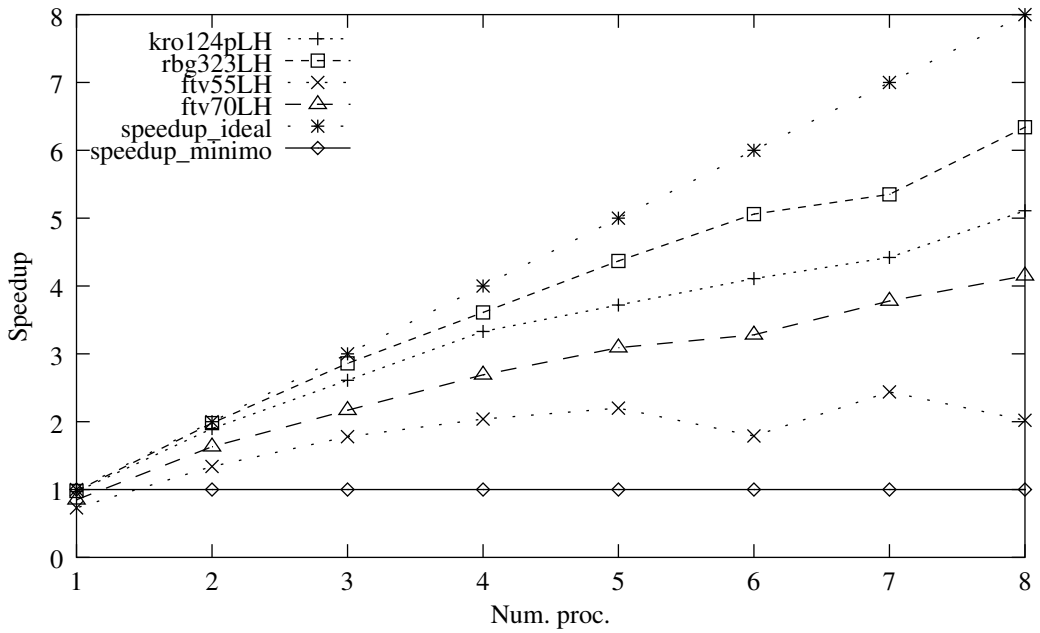


Figura 4.12: Desempenho **AMPME**, vizinhança de *inserção* e redução I_2.

Capítulo 5

O Problema de Seqüenciamento de Tarefas em Máquinas Paralelas

O problema de seqüenciamento de tarefas em máquinas paralelas (SMP) se assemelha muito com o problema de SMS. A diferença se dá no número de máquinas envolvidas no processo. Enquanto naquele havia somente uma máquina para processar as tarefas, no SMP há várias e, no caso específico deste trabalho, idênticas[1]. Seguindo a classificação de (Graham et al. 1979) para problemas de seqüenciamento, o problema tratado neste trabalho pode ser denominado como $P|sds|C_{max}$.

Alguns trabalhos importantes realizados neste problema devem ser mencionados. (Dearing & Henderson 1984) desenvolveram um modelo de programação linear inteira para alocação de máquinas numa indústria têxtil. Eles apresentaram bons resultados obtidos através do arredondamento de soluções obtidas pela relaxação linear do modelo inteiro. (Sumichrast & Baker 1987) propuseram um método heurístico baseado na solução de uma série de subproblemas inteiros 0-1 que melhorou os resultados de (Dearing & Henderson 1984). Um trabalho que apresenta resultados computacionais é o de (Frederickson, Hecht & Kim 1978). Foram empregados algoritmos aproximados derivados da equivalência entre o $P|sds|C_{max}$ e o TSP. (França, Gendreau, Laporte & Müller 1996) fizeram uso de uma heurística baseada em *busca tabu* associada com uma poderoso esquema de vizinhança, o qual emprega o conceito de vizinhos locais e globais. Recentemente (Frangioni, Scutellá & Necciari 1999) aplicaram um algoritmo de busca local com uma vizinhança baseada em múltiplas trocas de tarefas entre as máquinas.

A seguir é descrito o problema a ser tratado, bem como o algoritmo aplicado na sua resolução e os resultados computacionais obtidos.

5.1 Definição do Problema

O problema de seqüenciamento de tarefas em máquinas parelelas tratado neste trabalho pode ser assim descrito:

[1] Uma outra diferença é que agora considera-se que há um tempo de preparação da máquina necessário para executar a próxima tarefa da seqüência.

- **Entrada**: são dadas n tarefas a serem processadas em m máquinas, uma lista $\{t_1,\ldots,\ t_n\}$ de tempos de processamento e uma matriz $\{s_{ij}\}$ de tempos de preparação, onde s_{ij} é o tempo de preparação da tarefa j depois da máquina ter processado a tarefa i. É dada ainda uma lista $\{so_1,\ldots,\ so_n\}$ que informa o tempo necessário para preparar a produção caso a tarefa seja a primeira a ser processada.

- **Objetivo**: encontrar uma permutação das tarefas que minimize o *makespan*, calculado pela equações 5.1 e 5.2, onde *makespan* é o valor máximo do *tempoDeTermino* envolvendo todas as máquinas. A notação $i(k)$ indica a *k-ésima* operação processada na máquina i e $numTarefas_i$ é o número de tarefas processadas na máquina i.

$$makespan = max_i\ [\ tempDeTermino_i\ ]\ ;\quad i = 1\ldots,m \tag{5.1}$$

$$tempoDeTermino_i\ =\ so_{i(1)} + t_{i(1)} + \sum_{k=1}^{numTarefas_i} s_{i(k)i(k+1)} + t_{i(k+1)}\ ;\quad i = 1,\ldots,m \tag{5.2}$$

5.2 Algoritmo para Resolução

O AM proposto para resolver este problema difere somente em três aspectos em relação ao algoritmo aplicado ao problema SMS do capítulo anterior: representação genética, função de avaliação e busca local. Por isso somente estas etapas serão enunciadas nas seções a seguir.

5.2.1 Representação genética

A representação adotada para o problema de SMP é idêntica à proposta por (Cheng & Gen 1997). A solução é representada por um cromossomo cujos alelos assumem valores inteiros e distintos no intervalo $[1, n]$. Além disso, essa seqüência (cromossomo) é divida em m subseqüências, as quais são separadas por um caractere (*) e a *i-ésima* delas representa a seqüência de tarefas alocadas a máquina i.

Desta forma, um exemplo de solução válida para um problema com 10 tarefas e três máquinas seria $< 534 * 891 * 26107 >$. A máquina 1 processa sucessivamente as tarefas 5, 3 e 4; a máquina 2 as tarefas 8, 9 e 1; por último, a máquina 3 é encarregada de processar as tarefas 2, 6, 10 e 7, em seqüência.

5.2.2 Função de Avaliação

A função de avaliação não corresponde exatamente ao *makespan* porque o problema é de minimização. Como é necessário que *makespan* e função de avaliação sejam inversamente proporcionais[2], pois quanto menor o *makespan* melhor será o indivíduo, utiliza-se a função de avaliação dada pela equação 5.3.

[2]Por questão de convenção, o melhor indivíduo é aquele que possui o maior valor de função de avaliação.

$$f = (makespan + 1)^{-1} \tag{5.3}$$

5.2.2.1 Busca Local

Para este problema, a busca local implementada não inclui nenhum tipo de redução de vizinhança. Tal como no problema de SMS, foram implementadas as vizinhanças *all-pairs* e *inserção*. O pseudo-código do algoritmo é dado pela figura 5.1.

Entrada: Indivíduo a ser otimizado **ind**
Saída: Indivíduo otimizado **ind**

```
BuscaLocalPms(ind)
 1  fiAnterior ← fiAtual ← Avalia(ind);
 2  criterioDeParada ← falso;
 3  while criterioDeParada = falso
 4  do ind ← BuscaLocalAllpairs(ind);
 5     ind ← BuscaLocalInsercao(ind);
 6     fiAtual ← Avalia(ind);
 7     if fiAtual > fiAnterior
 8       then fiAnterior ← fiAtual;
 9       else
10              criterioDeParada ← veradeiro;
11  return ind;
```

Figura 5.1: Busca Local para o problema de *SMP*..

Tanto a busca local *all-pairs*, passo 4, como a busca local *inserção*, passo 5, devolvem um indivíduo que corresponde a um mínimo local, pela aplicação de um processo de *hill-climbing*. O laço que inicia no passo 3 só encerra quando nenhum dos procedimentos de busca local (*all-pairs* e *inserção*) gera uma solução melhor que a solução incumbente. Como visto, o procedimento de otimização do indivíduo compreende uma busca local composta de duas vizinhanças, aplicadas de forma consecutiva.

5.3 Resultados Computacionais

O algoritmo paralelo empregado para este problema é o AMPME, já definido anteriormente. O objetivo dos testes é avaliar o desempenho do algoritmo paralelo quanto ao tempo computacional, medindo, portanto, o *speedup*.

Em todos os testes é fixado um número de gerações, no caso 20, e executa-se o algoritmo seqüencial. A seguir é executado o mesmo algoritmo com a paralelização da etapa de otimização dos indivíduos (pseudo-código da figura 3.7). Com os dois tempos, Ts referente ao tempo para executar o algoritmo seqüencial e Tp correspondente ao tempo para executar o algoritmo paralelo, o *speedup* é obtido fazendo-se a razão $\frac{Ts}{Tp}$. Também determinou-se um número de 10 repetições para cada execução.

As instâncias utilizadas foram as mesmas do trabalho de (Mendes et al. 2002). Elas foram geradas randomicamente com um número de 20, 40, 60 e 80 tarefas. O número de máquinas foi fixado em 2, 4, 6 e 8. Os tempos de processamento foram gerados seguindo uma distribuição uniforme no intervalo [1,100]. Os tempos de preparação foram divididos em duas categorias: pequenos[3] e grandes[4]. A seguir trata-se as instâncias com tempos de preparação gerados no intervalo [1,10] como instâncias pequenas e os instâncias com tempos de preparação gerados no intervalo [1,100] como instâncias grandes.

Os tempos de preparação também foram gerados segundo dois outros critérios: estruturados e não-estruturados. Os tempos de preparação estruturados seguem a desigualdade triangular, isto é, $s_{ij} \leq s_{ik} + s_{kj}, \forall\, i,j,k \neq i,\ k \neq j$. Os tempos de preparação não-estruturados não seguem a desigualdade triangular, o que comumente torna a instância mais difícil de ser resolvida.

As tabelas 5.1 e 5.2 apresentam os resultados para instâncias estruturadas pequenas e grandes. Os figuras 5.2 e 5.3 mostram o comportamento do *speedup* para algumas instâncias dos problemas pequenos e grandes, respectivamente. Os resultados das instâncias não-estruturadas pequenas e grandes são apresentados pelas tabelas 5.3 e 5.4 e figuras 5.4 e 5.5, respectivamente.

Em todas as tabelas é apresentado o nome da instância na primeira columa. A segunda contém o número de tarefas de cada instância (n). O número de máquinas (m) de cada instância é apresentado na terceira coluna. O tempo médio (Tp em ms - $10^{-3}s$) para processar a busca local de um indivíduo é apresentado na quarta coluna. A quinta coluna contém uma estimativa de tempo[5] para transmitir um indivíduo entre a unidade mestre e a unidade escravo (Tc em ms - $10^{-3}s$).

Nos gráficos a seguir plotou-se duas retas adicionais para facilitar a análise e discriminar melhor as instâncias: a primeira é a do "speedup mínimo", discriminando as instâncias que, para um certo número de processadores, tem desempenho inferior ao algoritmo seqüencial se estiverem abaixo e superior se estiverem acima dela; a reta do "speedup ideal" fornece, para um determinado número de processadores, o *speedup* que seria obtido caso não houvesse custo para distribuir o processamento do algoritmo, ou seja, o "speedup ideal".

Em todos os tipos e tamanhos de instâncias, o *speedup* foi maior para as maiores instâncias. Em praticamente todos os resultados é possível determinar o quão melhor é o *speedup* de uma instância em relação a outra verificando a razão $\frac{Tp}{Tc}$ de cada uma delas. Há, é claro, exceções devido à imprecisão do tempo de comunicação (dependente de vários fatores como tráfego na rede e colisão) e ao fator não-determinístico da busca local, o que não permite uma medida confiável do tempo Tp.

A comparação entre instâncias grandes e pequenas, estruturadas pequenas com estruturadas grandes e não-estruturadas pequenas com não-estruturadas grandes, não permite

[3]Com valores no intervalo [1,10].

[4]Com valores no intervalo [1,100].

[5]Mediu-se esse tempo calculando o número de *bytes* de cada indivíduo e executando comandos específicos do sistema operacional para transferência entre duas unidades quaisquer da rede de computadores.

Instâncias	n	m	*Tp (ms)*	*Tc (ms)*	*Speedup* *Número de processadores* 1	2	3	4	5	6	7	8
Estrut10D220	20	2	25	3,5	0,10	0,13	0,17	0,25	0,44	0,26	0,37	0,35
Estrut10D240	40	2	199	3,8	0,69	0,80	0,94	1,12	1,48	1,39	1,73	1,54
Estrut10D260	60	2	1151	5,2	0,78	1,06	1,54	1,99	2,15	2,26	2,40	2,54
Estrut10D280	80	2	2922	6,6	0,88	1,47	2,28	2,61	3,14	3,50	3,92	4,37
Estrut10D420	20	4	37	4,1	0,50	0,72	0,89	1,02	1,09	1,17	1,19	1,16
Estrut10D440	40	4	314	6,7	0,90	1,54	2,11	2,59	3,04	3,41	3,72	3,86
Estrut10D460	60	4	1406	9,3	0,95	1,89	2,70	3,49	4,18	4,70	5,34	5,72
Estrut10D480	80	4	2278	11,9	0,98	1,92	2,77	3,57	4,26	4,97	5,54	6,09
Estrut10D620	20	6	36	5,4	0,59	0,79	1,01	1,18	1,27	1,33	1,37	1,41
Estrut10D640	40	6	247	9,4	0,91	1,56	2,16	2,67	3,18	3,46	3,89	3,95
Estrut10D660	60	6	1161	13,4	0,96	1,76	2,52	3,26	3,96	4,61	5,11	5,37
Estrut10D680	80	6	2343	18	0,98	1,82	2,75	3,47	4,13	4,80	5,43	6,01
Estrut10D820	20	8	45	6,8	0,66	0,88	1,13	1,30	1,42	1,51	1,58	1,58
Estrut10D840	40	8	365	12	0,90	1,64	2,27	2,86	3,32	3,77	4,05	4,39
Estrut10D860	60	8	1109	18,8	0,95	1,81	2,57	3,25	3,87	4,53	5,08	5,66
Estrut10D880	80	8	2631	24,2	0,97	1,86	2,74	3,54	4,31	5,12	5,49	6,04

Tabela 5.1: Resultados para instâncias **estruturadas pequenas**.

algo muito conclusivo. Nas instâncias estruturadas percebe-se apenas uma certa vantagem no valor do *speedup* para as instâncias pequenas com maior tamanho (considerando n e m). Quando a mesma comparação é feita para as instâncias não-estruturadas, o resultado é ainda menos conclusivo: embora a maioria das instâncias grandes apresentem um *speedup* melhor que as instâncias pequenas, nada pode ser concluído quanto à relação tamanho/*speedup*.

Quanto aos problemas estruturados e não-estruturados, é possível notar uma ligeira vantagem (no valor do *speedup*) para os problemas estruturados. Embora estes sejam teoricamente mais fáceis que os problemas não-estruturados, a busca local exigiu um esforço computacional maior, ocasionando um Tp maior e, por conseqüência, um *speedup* maior.

Um bom indicativo de sucesso do algoritmo paralelo em relação ao seqüencial é o *speedup* para dois processadores. Naquelas instâncias em que o mesmo é maior que 1 pode-se concluir que é proveitoso adotar o dito paralelismo. Isso é verificado em todas as instâncias com n maior que 20, à exceção das instâncias pequenas, estruturadas e não-estruturadas, com $n = 40$ e $m = 2$.

Outra importante conclusão, a partir dos resultados, é que o valor de *speedup* é tão melhor quanto maior for a instância, para todos os tipos (estruturadas e não-estruturadas, pequenas e grandes).

Instâncias	*n*	*m*	*Tp* (*ms*)	*Tc* (*ms*)	*Speedup* *Número de processadores* 1	2	3	4	5	6	7	8
Estrut100D220	20	2	22	3,5	0,35	0,52	0,64	0,73	0,78	0,81	0,82	0,81
Estrut100D240	40	2	187	3,8	0,82	1,42	1,95	2,37	2,69	3,04	3,26	3,37
Estrut100D260	60	2	749	5,2	0,89	1,75	2,49	3,23	3,82	4,43	4,71	5,13
Estrut100D280	80	2	1646	6,6	0,94	1,82	2,64	3,40	4,11	4,66	5,38	5,83
Estrut100D420	20	4	35	4,1	0,50	0,68	0,87	0,99	1,07	1,11	1,16	1,16
Estrut100D440	40	4	236	6,7	0,83	1,43	2,05	2,45	2,89	3,18	3,46	3,65
Estrut100D460	60	4	757	9,3	0,92	1,76	2,52	3,19	3,89	4,25	4,85	5,19
Estrut100D480	80	4	2014	11,9	0,97	1,79	2,65	3,33	4,03	4,70	5,12	5,65
Estrut100D620	20	6	47	5,4	0,57	0,81	1,06	1,20	1,33	1,42	1,47	1,47
Estrut100D640	40	6	255	9,4	0,85	1,47	2,06	2,56	3,00	3,31	3,59	3,86
Estrut100D660	60	6	879	13,4	0,96	1,74	2,54	3,16	3,77	4,25	4,82	5,01
Estrut100D680	80	6	1591	18	0,97	1,80	2,56	3,39	4,07	4,70	5,31	5,63
Estrut100D820	20	8	58	6,8	0,65	0,91	1,17	1,39	1,46	1,55	1,61	1,65
Estrut100D840	40	8	300	12	0,88	1,61	2,30	2,84	3,37	3,72	4,04	4,21
Estrut100D860	60	8	915	18,8	0,99	1,92	2,74	3,54	4,08	4,76	5,22	5,59
Estrut100D880	80	8	2356	24,2	0,98	1,86	2,67	3,48	4,25	4,86	5,46	5,78

Tabela 5.2: Resultados para instâncias **estruturadas grandes**.

Instâncias	*n*	*m*	*Tp* (*ms*)	*Tc* (*ms*)	*Speedup* *Número de processadores* 1	2	3	4	5	6	7	8
NEstrut10D220	20	2	21	3,5	0,10	0,12	0,13	0,16	0,21	0,36	0,32	0,34
NEstrut10D240	40	2	182	3,8	0,40	0,59	0,71	0,87	0,81	1,06	1,34	1,58
NEstrut10D260	60	2	669	5,2	0,62	1,01	1,28	1,52	1,95	2,41	2,82	3,53
NEstrut10D280	80	2	1558	6,6	0,78	1,39	2,15	2,57	2,89	3,09	3,34	4,67
NEstrut10D420	20	4	26	4,1	0,45	0,69	0,85	0,96	1,05	0,94	1,11	1,11
NEstrut10D440	40	4	228	6,7	0,81	1,47	2,06	2,51	2,90	3,20	3,53	3,60
NEstrut10D460	60	4	778	9,3	0,91	1,77	2,55	3,23	3,97	4,41	4,83	5,33
NEstrut10D480	80	4	1843	11,9	0,97	1,84	2,65	3,38	4,15	4,67	5,41	5,74
NEstrut10D620	20	6	36	5,4	0,57	0,81	1,03	1,22	1,30	1,34	1,38	1,38
NEstrut10D640	40	6	255	9,4	0,82	1,51	2,09	2,62	3,07	3,43	3,72	3,83
NEstrut10D660	60	6	848	13,4	0,93	1,76	2,55	3,28	3,82	4,51	4,97	5,26
NEstrut10D680	80	6	2182	18	0,98	1,86	2,72	3,49	4,28	4,87	5,55	5,84
NEstrut10D820	20	8	47	6,8	0,64	0,83	1,07	1,23	1,36	1,47	1,52	1,49
NEstrut10D840	40	8	289	12	0,85	1,54	2,17	2,79	3,19	3,55	3,85	4,20
NEstrut10D860	60	8	1089	18,8	0,96	1,80	2,64	3,29	4,04	4,54	5,04	5,55
NEstrut10D880	80	8	2546	24,2	0,97	1,82	2,68	3,46	4,18	4,81	5,36	5,92

Tabela 5.3: Resultados para instâncias **não-estruturadas pequenas**.

Instâncias	*n*	*m*	*Tp* *(ms)*	*Tc* *(ms)*	*Speedup* *Número de processadores* 1	2	3	4	5	6	7	8
NEstrut100D220	20	2	18	3,5	0,30	0,51	0,59	0,76	0,80	0,62	0,60	0,62
NEstrut100D240	40	2	170	3,8	0,78	1,37	1,90	2,36	2,66	2,68	2,74	2,89
NEstrut100D260	60	2	630	5,2	0,95	1,77	2,51	3,06	3,82	4,17	4,69	4,39
NEstrut100D280	80	2	1869	6,6	0,98	1,87	2,77	3,47	4,16	4,68	5,38	5,64
NEstrut100D420	20	4	31	4,1	0,49	0,68	0,85	1,00	1,07	1,07	1,15	1,16
NEstrut100D440	40	4	235	6,7	0,80	1,46	2,02	2,46	2,90	3,26	3,46	3,60
NEstrut100D460	60	4	805	9,3	0,94	1,72	2,49	3,24	3,86	4,32	4,72	5,09
NEstrut100D480	80	4	1615	11,9	0,96	1,83	2,60	3,41	4,04	4,63	5,28	5,58
NEstrut100D620	20	6	47	5,4	0,59	0,81	1,02	1,16	1,28	1,33	1,43	1,42
NEstrut100D640	40	6	257	9,4	0,83	1,50	2,08	2,53	3,01	3,30	3,61	3,82
NEstrut100D660	60	6	780	13,4	0,94	1,76	2,55	3,23	3,84	4,35	4,81	5,05
NEstrut100D680	80	6	1965	18	0,98	1,88	2,64	3,45	4,14	4,83	5,42	6,01
NEstrut100D820	20	8	45	6,8	0,64	0,85	1,07	1,26	1,37	1,43	1,50	1,50
NEstrut100D840	40	8	264	12	0,86	1,54	2,17	2,70	3,18	3,59	3,91	4,13
NEstrut100D860	60	8	846	18,8	0,95	1,77	2,52	3,26	3,85	4,49	4,97	5,22
NEstrut100D880	80	8	1771	24,2	0,87	1,50	2,14	2,58	3,19	3,42	3,90	4,03

Tabela 5.4: Resultados para instâncias **não-estruturadas grandes**.

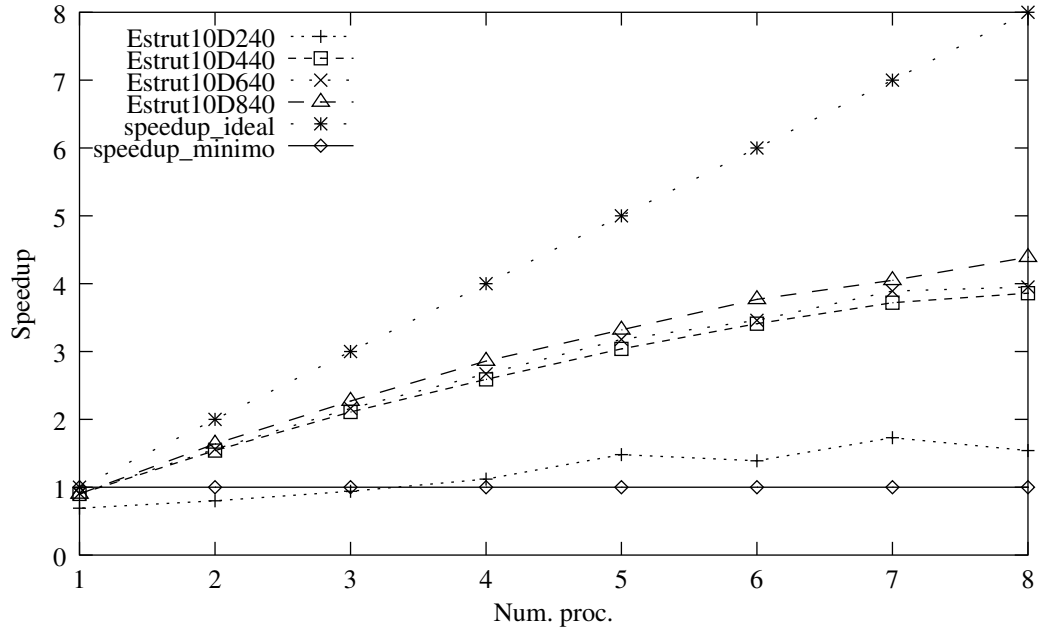


Figura 5.2: Desempenho para algumas instâncias **estruturadas pequenas**.

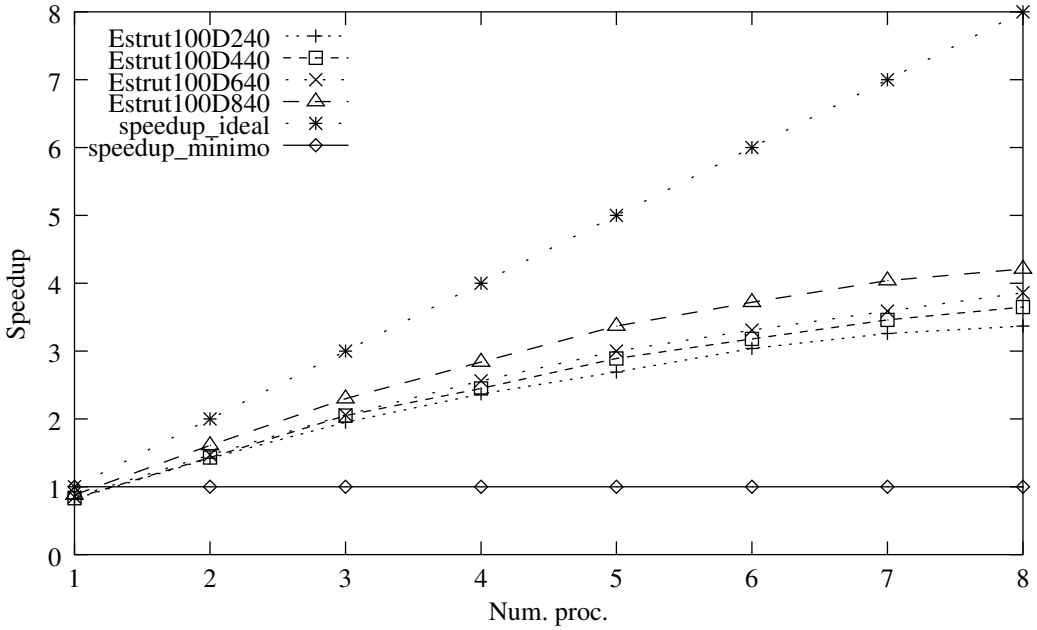


Figura 5.3: Desempenho para algumas instâncias **estruturadas grandes**.

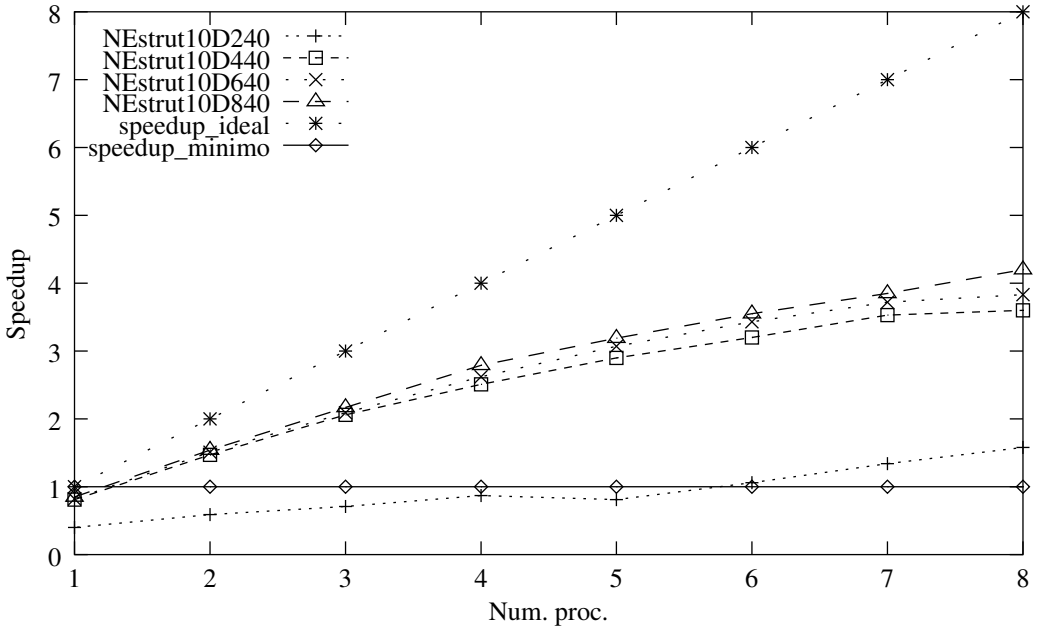


Figura 5.4: Desempenho para algumas instâncias **não-estruturadas pequenas**.

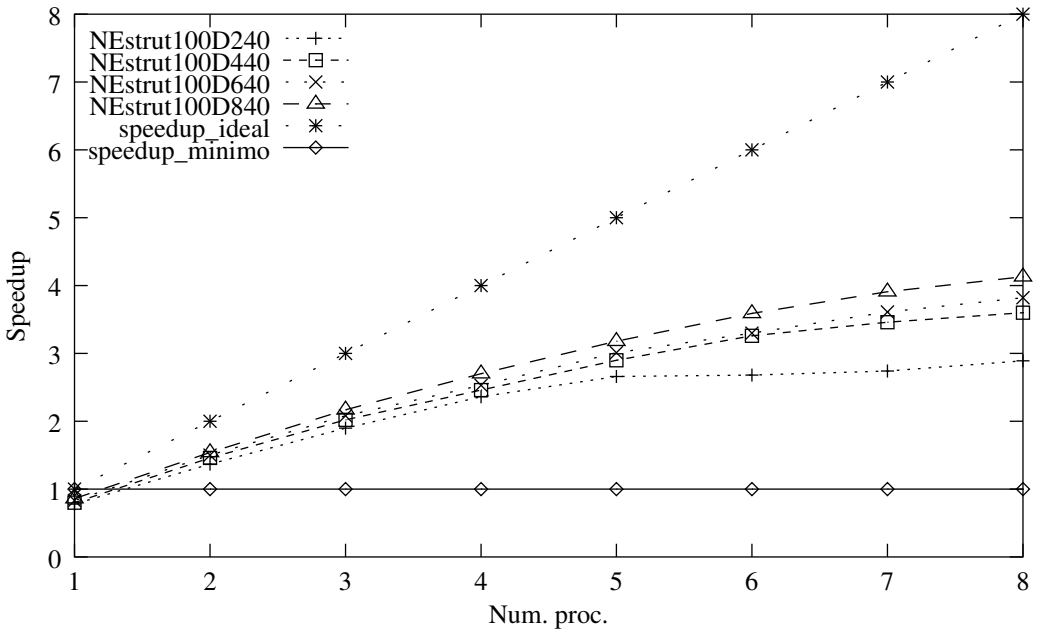


Figura 5.5: Desempenho para algumas instâncias **não-estruturadas grandes**.

Capítulo 6

O Problema do Caixeiro Viajante Assimétrico

O problema do Caixeiro Viajante (TSP) é um dos problemas de otimização mais estudados. Tamanho interesse se deve ao número de aplicações práticas, assim como pela facilidade com que outros problemas podem ser reduzidos a este. Embora haja algoritmos exatos para os casos simétrico (TSP) e assimétrico (ATSP), os algoritmos heurísticos ganharam grande relevância quando (Garey & Johnson 1979) provaram que este problema pertencia à classe NP-hard. Um estudo sobre os métodos exatos de resolução para o TSP pode ser encontrado em (Laporte 1992) e (Jünger, Reinelt & Rinaldi 1995).

Como métodos exatos para o ATSP, destaca-se o trabalho de (Miller & Pekny 1991), que empregou um algoritmo de *branch–and–bound* com uma relaxação do tipo de problema de designação do ATSP. Eles reportaram soluções ótimas para instâncias[1] de até 500 mil cidades. Embora isso pareça surpreendente, é preciso ponderar tais resultados, como observa (Cirasella, Johnson, McGeoch & Zhang 2001): "esse mesmo algoritmo não foi capaz de resolver uma instância de 35 cidades proveniente de um problema real de manufatura".

Um algoritmo heurístico que apresentou relativa eficiência na resolução do ATSP foi realizado por (Freisleben & Merz 1996). Foi utilizado um algoritmo memético associado a um operador de busca local, além de um novo operador de recombinação. Seus resultados indicaram que, para um determinado conjunto de instâncias, é possível encontrar soluções de boa qualidade em tempos razoáveis. (Gorges-Schleuter 1997) empregaram um algoritmo genético paralelo para resolver o ATSP. Este algoritmo emprega uma população de indivíduos distribuídos numa estrutura espacial em anel. Seus resultados, para um conjunto bem limitado de instâncias, foram relativamente bons. Recentemente, (Buriol et al. 1999) relataram o sucesso da aplicação de um algoritmo memético para resolução do ATSP. Os testes computacionais fizeram uso de todas as instâncias disponíveis na TSPLIB e geraram resultados muito bons.

As seções a seguir descrevem o problema que é tratado neste trabalho, o algoritmo

[1] Essas instâncias foram geradas aleatoriamente, com distribuição uniforme em um determinado intervalo.

usado para sua resolução e os resultados obtidos.

6.1 Definição do Problema

O ATSP tratado neste trabalho pode ser descrito da seguinte forma:

- **Entrada**: são dadas n cidades a serem visitadas pelo caixeiro e uma matriz $\{d_{ij}\}$ representando a distância para ir de uma cidade i até uma cidade j.
- **Objetivo**: encontrar uma permutação de cidades que minimize a distância total percorrida. A função objetivo pode ser descrita pela equação 6.1, onde $d(i,j)$ representa a distância (custo) para ir da cidade i até a cidade j.

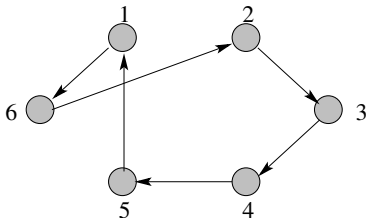
$$\sum_{i=1}^{n-1} d(i,i+1) + d(n,1) \tag{6.1}$$

6.2 Algoritmo para Resolução

O AM empregado na resolução deste problema é baseado nos trabalhos de (Buriol 2000) e (Buriol et al. 1999). A seguir são apresentadas apenas algumas características principais do mesmo.

6.2.1 Representação

A estrutura adotada para representação de uma solução (rota) é um vetor de n posições. Cada cidade i terá suas informações na posição i do vetor. Em cada posição i encontram-se os índices das cidades antecessora (A) e sucessora (S) da cidade i. A figura 6.1 ilustra a representação de uma possível rota para um problema de 6 cidades.



(a) Rota.

	1	2	3	4	5	6
A	5	6	2	3	4	1
S	6	3	4	5	1	2

(b) Representação.

Figura 6.1: Representação de uma solução para o **ATSP**.

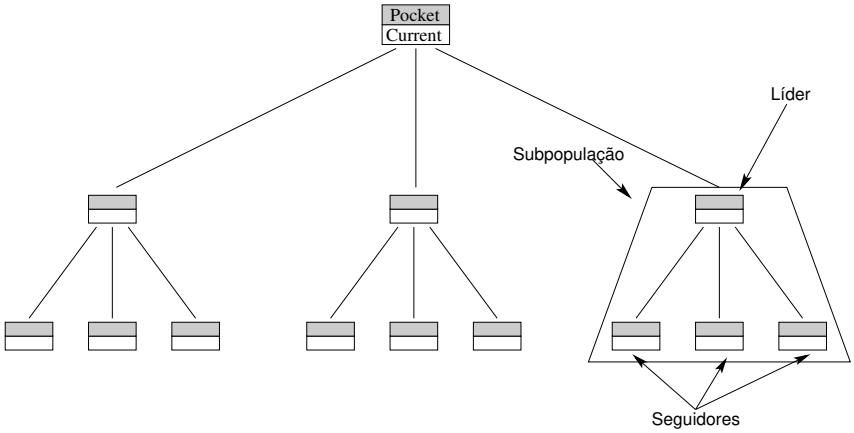


Figura 6.2: Estrutura da população.

6.2.2 Estrutura da População

A população deste AM é composta por 13 agentes organizados em uma árvore ternária com 3 níveis, conforme ilustra a figura 6.2.

Há, portanto, 4 subpopulações, cada uma composta por um líder e três seguidores. O líder é sempre o agente mais bem adaptado dos 4, estando no nó raiz o melhor agente da população.

Cada agente é formado por duas soluções (rotas), denominadas *pocket* e *current*. As operações de mutação, recombinação e busca local são aplicadas nas *rotas currents*, enquanto que as *rotas pockets* funcionam como uma memória, armazenando uma rota de custo reduzido já encontrada por uma *rota current*.

6.2.3 Recombinação

O operador de recombinação utilizado é o SAX (*Strategic Arc Crossover*) (Buriol 2000), uma adaptação do SEX (*Strategic Edge Crossover*) (Moscato & Norman 1992) para o TSP assimétrico. O SEX, por sua vez, é uma modificação do clássico OX (Goldberg 1989).

O SAX é um procedimento relativamente sofisticado, cujo objetivo é gerar um filho com maior número possível de arcos provenientes dos pais.

6.2.4 Mutação

O procedimento de mutação empregado consiste na inserção de uma cidade entre uma cidade escolhida aleatoriamente e a sua cidade sucessora. Selecionada uma cidade i, de forma aleatória, é selecionada uma cidade j, também de forma aleatória, tal que $j \neq i$. Essa cidade j é inserida entre a cidade i e a cidade sucessora de i (Suc(i)), criando o arco (Ant(j),Suc(j)). A figura 6.3 descreve esse procedimento para uma rota hipotética com 10 cidades.

Como mostra a figura 6.3(b), as cidades i e j foram selecionadas, sendo indicadas as cidades Suc(i), Ant(j) e Suc(j). Assim as cidades 1 (j), 2 (Suc(j)), 8 (i), 9 (Ant(j)) e

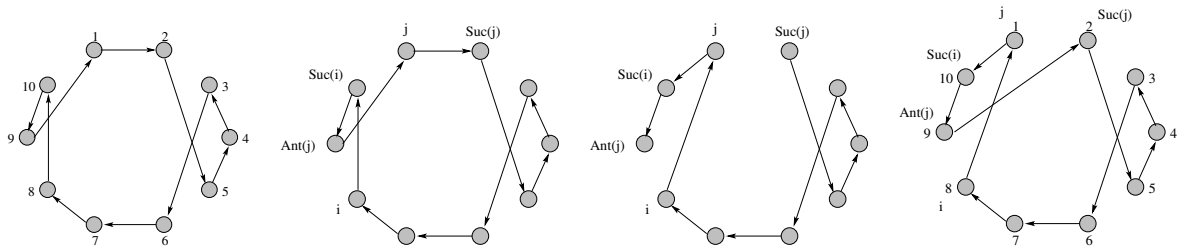


Figura 6.3: Procedimento de mutação.

10 (Suc(i)) participarão do processo de mutação. Ele se inicia com a inserção da cidade j entre as cidades i e Suc(i), removendo os arcos (i,Suc(i)) e (Ant(j),j) e inserindo (i,j) e (j,Suc(i)), conforme ilustra a figura 6.3(c). Finalmente é realizada a inserção do arco (Ant(j),Suc(j)) para fechar a rota, como demonstra a figura 6.3(d).

6.2.5 Seleção

O procedimento de seleção (Buriol 2000) sempre está restrito a uma subpopulação. Seu principal objetivo é a preservação da diversidade da população. Para um bom entendimento desse procedimento, será necessário o uso da seguinte notação:

- **Líder**: o agente líder de uma subpopulação (nó raiz da subárvore-subpopulação);
- **Seguidor1, Seguidor2, Seguidor3**: os três agentes subordinados (nós folha da subárvore);
- **Pocket(i)**: a rota *pocket* do agente i;
- **Current(i)**: a rota *current* do agente i.

A seleção se fará para compor todas as rotas *current* dos agentes da população, ou seja, como há 4 subpopulações e quatro agentes em cada uma, serão geradas 16 novas rotas *current* a cada geração. A política a seguir (Buriol 2000) descreve como são selecionados os pais para compor cada rota *current* de uma subpopulação:

- *Current* do *líder* $\Leftarrow$ Recombine(*Pocket(seguidor2),Pocket(seguidor3)*);
- *Current* do *seguidor1* $\Leftarrow$ Recombine(*Pocket(líder),Current(seguidor2)*);
- *Current* do *seguidor2* $\Leftarrow$ Recombine(*Pocket(seguidor1),Current(seguidor3)*);
- *Current* do *seguidor3* $\Leftarrow$ Recombine(*Pocket(seguidor2),Current(seguidor1)*);

6.2.6 Busca Local

O operador de busca local empregado nesse algoritmo é chamado de RAI (*Recursive Arc Insertion*-inserção recursiva de arcos) e foi desenvolvido por (Buriol 2000). A vizinhança é sempre do tipo *3-opt*, devido a remoção de 3 arcos e a inserção de outros 3, com tamanho de n^3, sendo n o número de cidades da instância a ser resolvida. Logo torna-se necessário restringir os movimentos disponíveis a cada iteração da busca local para que sua aplicação não se torne computacionalmente custosa, ou até mesmo impraticável.

6.2.7 Algoritmo Mestre-escravo

O algoritmo mestre-escravo implementado para este problema é idêntico ao descrito na seção 3.1.

6.3 Resultados Computacionais

Para o ATSP empregou-se a mesma estrutura do algoritmo mestre-escravo descrita na seção 3.1, ou seja, modificou-se o algoritmo seqüencial de (Buriol 2000) para distribuir o processamento do operador de busca local. Novamente buscou-se reduzir o tempo necessário para a execução do algoritmo.

Em todos os testes foi fixado um número de gerações, no caso 20, e executado o algoritmo seqüencial. A seguir, executou-se o mesmo algoritmo com a paralelização da etapa de otimização dos indivíduos (pseudo-código da figura 3.7). Com os dois tempos, Ts referente ao tempo para executar o algoritmo seqüencial e Tp correspondente ao tempo para executar o algoritmo paralelo, o *speedup* é obtido fazendo-se a razão $\frac{Ts}{Tp}$. Também determinou-se um número de 10 repetições para cada execução.

O conjunto de instâncias para teste foi obtido de TSPLIB[2]. Todas elas foram geradas a partir de instâncias do TSP simétrico seguindo um procedimento muito semelhante ao sugerido por (Fischetti & Toth 1997), (Moscato 2001): dada uma distância c_{ij} (matriz de distâncias diagonal superior) tal que $j > i$, a respectiva distância da diagonal inferior é dada por $c_{ji} = c_{ij} + ks$ se o par (i, j) não fizer parte da rota ótima, onde s é o valor da distância média computada entre todo par (i, j) e k é um fator gerado aleatoriamente no intervalo $[0, K_max]$, com distribuição uniforme; caso contrário $c_{ji} = c_{ij}$. Com isso é possível garantir que a rota ótima obtida com a matriz assimétrica tem a mesma distância que a obtida com a matriz simétrica.

O tamanho das instâncias utilizadas nos testes varia de 180 a 1002 cidades. Também adotou-se dois valores para K_{max}: 5% e 50%. Para o caso específico do ATSP foi utilizado outro *hardware*, devido à indisponibilidade daquele descrito na seção 1.2 na fase final deste trabalho: foram 8 computadores do tipo PC Intel Pentium III 1GHz, cada um com dois processadores e 512 MB de memória RAM, interligados por uma rede padrão *Ethernet* 100Mbits.

[2]http://www.crpc.rice.edu/softlib/tsplib/

Em todas as tabelas é apresentado o nome da instância na primeira columa. A segunda contém o número de cidades de cada instância (n). O tempo médio (Tp em ms - $10^{-3}s$) para processar a busca local de um indivíduo é apresentado na terceira coluna. A quarta coluna contém uma estimativa de tempo para transmitir um indivíduo entre a unidade mestre e a unidade escravo (Tc em ms - $10^{-3}s$).

A tabela 6.1 e figura 6.4 ilustram os resultados para as instâncias geradas com $K_{max} = 5\%$. Os resultados para as instâncias geradas com $K_{max} = 50\%$ são ilustrados pela tabela 6.2 e figura 6.5.

Instâncias	n	*Tp (ms)*	*Tc (ms)*	*Speedup* *Número de processadores*							
				1	2	3	4	5	6	7	8
brg180_5	180	30,5	0,98	0,58	0,76	0,99	1,13	1,23	1,29	1,34	1,38
tsp225_5	225	40,25	1,12	0,62	0,93	1,10	1,25	1,37	1,43	1,49	1,52
a280_5	280	28,08	1,34	0,54	0,77	0,94	1,00	1,05	1,13	1,15	1,17
pa561_5	561	275	2,22	0,81	1,32	1,66	1,94	2,15	2,30	2,43	2,55
gr666_5	666	434,5	2,53	0,83	1,45	1,87	2,21	2,48	2,69	2,94	3,06
pr1002_5	1002	1142,91	3,63	0,91	1,54	2,04	2,45	2,78	3,05	3,27	3,41

Tabela 6.1: Resultados para as instâncias geradas a partir de instâncias do **TSP** simétrico, com $K_{max} = 5\%$.

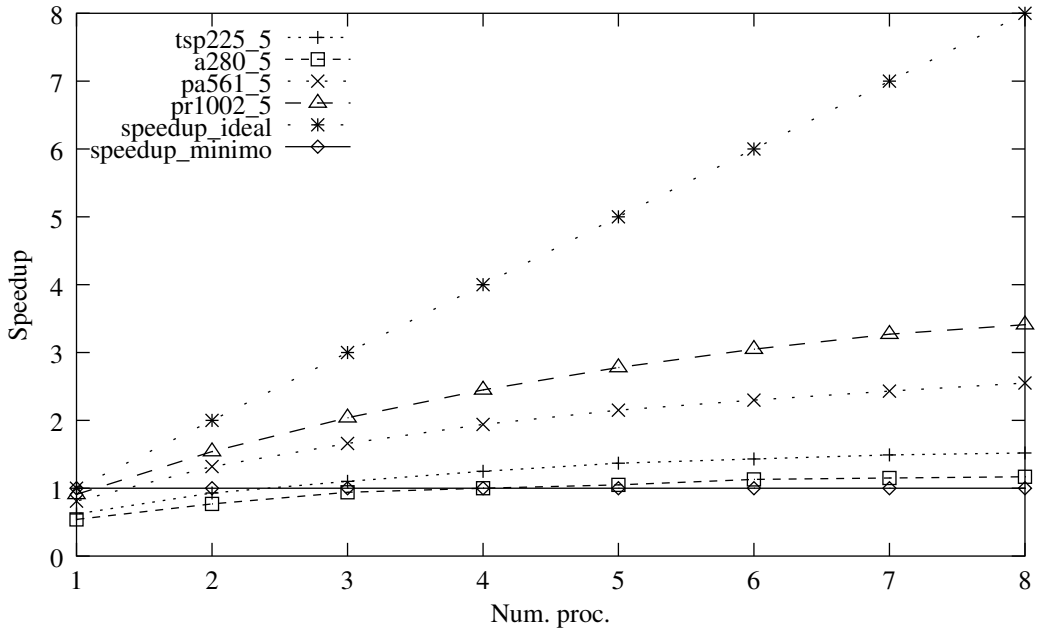


Figura 6.4: Desempenho para algumas instâncias geradas a partir de instâncias do **TSP** simétrico, com $K_{max} = 5\%$.

Analisando-se os resultados para as instâncias geradas com $K_{max} = 5\%$, é possível verificar certo êxito para todas as instâncias, considerando-se o *speedup* e desprezando-se o número de processadores necessários para obtê-lo. Obviamente pode ser desperdício

Instâncias	*n*	*Tp* *(ms)*	*Tc* *(ms)*	*Speedup* *Número de processadores*							
				1	2	3	4	5	6	7	8
brg180_50	180	39,41	0,98	0,60	0,81	1,06	1,23	1,34	1,43	1,48	1,52
tsp225_50	225	18,41	1,12	0,51	0,73	0,87	0,96	1,03	1,07	1,11	1,12
a280_50	280	26,66	1,34	0,51	0,73	0,88	0,98	1,00	1,06	1,09	1,10
pa561_50	561	141,33	2,22	0,74	1,16	1,44	1,65	1,81	1,93	2,01	2,06
gr666_50	666	545,66	2,53	0,88	1,52	2,00	2,42	2,71	2,97	3,19	3,35
pr1002_50	1002	1378,00	3,63	0,92	1,64	2,21	2,68	3,11	3,42	3,70	3,95

Tabela 6.2: Resultados para as instâncias geradas a partir de instâncias do **TSP** simétrico, com $K_{max} = 50\%$.

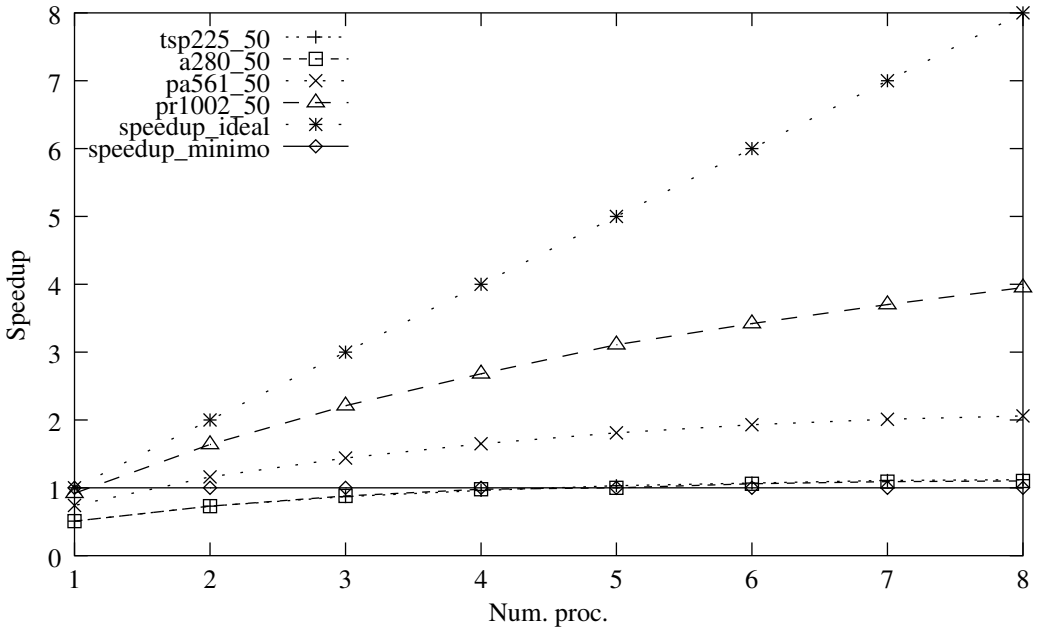


Figura 6.5: Desempenho para algumas instâncias geradas a partir de instâncias do **TSP** simétrico, com $K_{max} = 50\%$.

empregar 8 computadores para se obter um ganho de 17% no tempo de processamento, no caso da instância a280_5. No entanto ela é justamente o pior caso, a que teve o menor Tp e, portanto, ocasionou o pior desempenho. A instância brg180_5 teve um Tp um pouco superior à anterior, obtendo um *speedup* também superior. Um bom indicativo de quão bom é o desempenho de uma instância é o número de processadores necessários para se obter um *speedup* maior que 1: enquanto a instância a280_5 precisou de 5 processadores para obter um tempo de execução do algoritmo paralelo menor que o do algoritmo seqüencial correspondente, a instância brg180_5 necessitou de apenas 4. Caso as instâncias sejam dispostas em ordem crescente dos respectivos Tp (a280_5, brg180_5, tsp225_5, pa561_5, gr666_5, pr1002_5), percebe-se se que o valor do *speedup* acompanha o crescimento do Tp, sendo o menor *speedup* da instância a280_5 e o maior da instância pr1002_5.

Quanto a figura 6.4, para facilitar a análise e discriminar melhor as instâncias, plotou-

se duas retas adicionais: a primeira é a do "speedup mínimo", discriminando as instâncias que, para um certo número de processadores, tem desempenho inferior ao algoritmo seqüencial se estiverem abaixo e superior se estiverem acima dela; a reta do "speedup ideal" fornece, para um determinado número de processadores, o *speedup* que seria obtido caso não houvesse custo para distribuir o processamento do algoritmo, ou seja, o "speedup ideal". A curva da instância pr1002_5 é a que mais se aproxima da reta do "speedup ideal", sendo o menor desvio com 1 processador e o maior com 8. Esse desvio indica o quão vantajoso é o aumento de processadores: quando se passa de 1 para 2 processadores obtém-se um aumento de 69% no *speedup* para instância pr1002_5, no entanto quando se passa de 7 para 8 esse ganho é reduzido para 4%. Isso indica que o aumento para 9 ou mais processadores irá representar um ganho marginal no *speedup*.

As instâncias geradas com $K_{max} = 50\%$ tiveram um desempenho razoável, novamente considerando-se o *speedup* e desprezando-se o número de processadores. O melhor *speedup* obtido foi 3,95, com a instância pr1002_50 e 8 processadores. O pior, 1,10, foi obtido com a instância a280_50. Para esta, o número mínimo de processadores para que se obtenha um tempo de execução do algoritmo menor que o do algoritmo seqüencial correspondente foi 6. Para instâncias com Tp maior que 100 (pa561_50, gr666_50 e pr1002_50), é possível obter um *speedup* maior que 1 com somente 2 processadores.

A figura 6.5 ilustra o desempenho de algumas instâncias geradas com $K_{max} = 50\%$. Para as instâncias tsp225_50 e a280_5, cujos desempenhos foram os piores, só é possível obter *speedups* maiores que 1 com 5 e 6 processadores, respectivamente. As outras, pa561_50 e pr1002_50, necessitaram de apenas 2. Analisando-se o desvio da curva da instância pr1002_50 em relação a reta do "speedup ideal", percebe-se que o menor desvio é justamente com 1 processador e o maior quando se emprega 8 processadores. Tal comportamento pode ser comprovado com o cálculo do ganho percentual no *speedup*: o maior ganho é de 78%, quando se aumenta de 1 para 2 processadores, e o menor é de 6%, quando se aumenta de 7 para 8 processadores.

Comparando-se os resultados obtidos com os dois tipos de instâncias, $K_{max} = 5\%$ e $K_{max} = 50\%$, percebe-se um comportamento particular para cada instância. As instâncias brg180_50, gr666_50, pr1002_50 tiveram Tp superiores se comparados com os Tp das instâncias brg180_5, gr666_5, pr1002_5. Já as instâncias tsp225_50, a280_50 e pa561_50 tiveram Tp inferiores aos Tp das instâncias tsp225_5, a280_5 e pa561_5. O aumento no Tp também representou aumento no *speedup*, assim como a redução no Tp também representou a redução no *speedup*. Isso comprova a dependência entre Tp e *speedup*, discutida na seção 3.1.

Capítulo 7

Conclusões

Neste trabalho propôs-se um AM paralelo para distribuir a tarefa de processamento do operador de busca local. Aplicou-se este algoritmo a 3 problemas de otimização: problema de seqüenciamento de tarefas em máquina simples (SMS), problema de seqüenciamento de tarefas em máquinas paralelas (SMP) e problema do caixeiro viajante assimétrico (ATSP). Para os dois primeiros empregou-se algumas modificações no algoritmo de (Mendes et al. 2001) com o objetivo de distribuir a etapa de otimização dos indivíduos (operador de busca local). Para o terceiro foram utilizadas as mesmas modificações, realizadas no SMS e no SMP, no algoritmo de (Buriol 2000).

A estrutura empregada para realizar a distribuição do processamento da etapa de otimização dos indivíduos, descrita na seção 3.1, foi concebida de forma a obter a maior eficiência possível quanto a utilização do tempo de CPU[1]. Para tanto, fez-se uso de *threads* principalmente na unidade mestre do AMP mestre-escravo. Um fator de grande relevância na estrutura empregada é a forma de comunicação: a opção escolha por *sockets* teve como objetivo minimizar o custo computacional necessário para efetuar a troca de mensagens. Além disso, preocupou-se em proporcionar máxima eficiência[2] na aplicação do algoritmo em ambientes heterogêneos[3].

Para a consolidação do modelo proposto, aplicou-se o AMPME para um conjunto de 8 instâncias do SMS, 64 do SMP e 12 do ATSP. Em todos os problemas foi investigado a performance do AMPME em relação ao algoritmo seqüencial, ambos com o mesmo número de gerações e as mesmas taxas de recombinação e mutação. O critério de desempenho adotado foi o *speedup*, descrito na seção 1.1.6, isto porque teve-se como objetivo apenas quantificar quão rápido um algoritmo é em relação ao outro.

Para o SMS, o algoritmo de (Mendes 1999) já apresentava bons resultados em um tempo bem reduzido para um bom número de instâncias. Por isso foram escolhidas justamente aquelas instâncias para as quais era necessário um tempo relativamente elevado para resolução: instâncias de 56, 71, 100 e 323 tarefas. Para estas foram utilizadas as duas vizinhanças implementadas por (Mendes 1999) com as duas respectivas reduções. O obje-

[1]Isso é proporcionado pelo uso de múltiplas *threads*.

[2]Uso múltiplas *threads*, cada uma correspondendo a uma unidade escravo.

[3]As unidades processadoras podem ter configurações de *hardware* distintas.

tivo foi avaliar a influência das vizinhanças, e suas reduções, no tempo de processamento do operador de busca local (Tp). Os resultados do capítulo 4 indicaram comportamentos diferenciados para as vizinhanças *all-pairs* e *inserção* e suas reduções. Enquanto que a vizinhança *all-pairs* mais restrita teve um Tp menor em relação à vizinhança *all-pairs* menos restrita, a vizinhança de *inserção* apresentou um comportamento oposto. Quanto a relação número de tarefas e Tp, comprovou-se que para instâncias maiores o Tp também é maior. Justamente para a configuração que ocasionou os maiores Tp (vizinhança *all-pairs* menos restrita e vizinhança de *inserção* mais restrita) o *speedup* foi maior. Assim é possível concluir que o AMPME empregado terá resultados tão melhores quanto maior for o Tp, e mais: quanto maior for a razão $\frac{Tp}{Tc}$, melhor será o *speedup* e, portanto, o desempenho do AMPME, comprovando os resultados da seção 3.1.

No SMP foi adotado o mesmo conjunto de instâncias utilizado no trabalho de (Mendes et al. 2002). Foi investigado a influência que as características das instâncias tinham no tempo Tp. Primeiramente quanto ao intervalo e geração: instâncias com tempos de preparação gerados no intervalo [1,10] apresentaram um Tp maior que aquelas que tiveram seus tempos de preparação gerados no intervalo [1,100]. Quanto a outra característica, problemas estruturados apresentaram Tp superiores aos Tp dos problemas não-estruturados. Novamente a configuração que ocasionou maiores Tp foi a que obteve melhores *speedups*, comprovando-se mais uma vez os resultados da seção 3.1.

Finalmente realizou-se os testes para o ATSP com instâncias com tamanhos que variaram de 180 a 1002 cidades. A idéia foi a mesma aplicada aos dois problemas anteriores: comprovar que o desempenho do AMPME está intimamente relacionado com o tempo de processamento do operador de busca local (Tp). Os resultados evidenciaram exatamente isso: o grupo de instâncias com menores Tp (brg180, tsp225 e a280) obteve os menores *speedups*; o outro, com os maiores Tp (pa561, gr666 e pr1002), apresentou *speedups* maiores.

Após todos os estudos de caso, SMS, SMP e ATSP, é possível concluir que houve um bom desempenho do algoritmo proposto, melhor para os dois primeiros problemas. Os resultados práticos obtidos comprovaram os resultados teóricos da seção 3.1 em praticamente todos os casos.

Como trabalhos futuros, sugere-se os seguintes:

- Verificação do comportamento do algoritmo para instâncias que demandem um tempo maior de execução, quando executado em uma rede com mais de 8 unidades processadoras;
- Aplicação de vários operadores de busca local simultaneamente, deixando, senão cada indivíduo, cada grupo de indivíduos com um operador de busca local distinto;
- Estudo da dinâmica populacional e a influência que a migração, o número e tamanho de subpopulações têm na eficiência do algoritmo, no que se refere a qualidade da solução encontrada.

Referências Bibliográficas

Abramson, D. & Abela, J. (1992). A prallel genetic algorithm for solving the school timetabling problem, *Proceedings of the Fifteenth Australian Computer Science Conference*, Vol. 14, pp. 1–11.

Abramson, D., Millis, G. & Perkins, S. (1993). Parallelisation of a genetic algorithm for computation of efficient train schedules, *Proceedings of the 1993 Parallel Computing and Transputers Conference*, pp. 139–149.

Andre, D. & Koza, J. R. (1998). A parallel implementation of genetic programming that achieves super-linear performance, *Information Sciences* (106): 201–218.

Baluja, S. (1992). A massively distributed parallel genetic algorithm, *Technical Report CMU-CS-92-196R*, School of Computer Science, Carnegie Mellon University.

Bäck, T., Fogel, D. B. & Michalewicz, T. (2000a). *Evolutionary Computation 1*, Institute of Physics Publishing.

Bäck, T., Fogel, D. B. & Michalewicz, T. (2000b). *Evolutionary Computation 2*, Institute of Physics Publishing.

Buriol, L. (2000). *Algoritmo memético para o problema do caixeiro viajante assimétrico como parte de um framework para algoritmos evolutivos*, Master's thesis, Faculdade de Engenharia Elétrica e de Computação. Universidade Estadual de Campinas.

Buriol, L., França, P. & Moscato, P. (1999). A new memetic algorithm for the asymmetric traveling salesman problem, *Submited to Journal of Heuristics* .

Cantú-Paz, E. (1997a). Designing efficient master-slave parallel genetic algorithms, *Technical Report 97004*, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign.

Cantú-Paz, E. (1997b). A survey of parallel genetic algorithms, *Technical Report 97003*, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign.

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

Cantú-Paz, E. (2001). Migration policies, selection pressure, and parallel evolutionary algorithms, *Journal of Heuristics* **4**(7): 311–334.

Cheng, R. & Gen, M. (1997). Parallel machine scheduling problems using memetic algorithms, *Computers Ind. Eng.* **33**(3–4): 761–764.

Cirasella, J., Johnson, D. S., McGeoch, L. A. & Zhang, W. (2001). The asymmetric traveling salesman problem: Algorithms, instance generators, and tests, *ALENEX*, pp. 32–59.

Comer, D. E. (1998). *Internetworking with TCPIP, v2̇*, Prentice Hall.

Davis, L. (1985). Job shop scheduling with genetic algorithms, *1st. Int. Conf. on Genetic Algorithms*, pp. 136–140.

Dawkins, R. (1976). *The selfish gene*, Oxford University Press.

Dearing, P. & Henderson, R. (1984). Assigning looms in a textile weaving operation with changeover limitations, *Production and Inventory Management* (25): 23–31.

Digalakis, J. & Margaritis, K. (2000). A performance comparison of parallel genetic and memetic algorithms using mpi, *Submitted to Complexity International* .

Du, J. & Leung, J. (1990). Minimizing total tardiness on one machine is np-hard, *Mathematics of Operations Research* **15**: 483–495.

Fischetti, M. & Toth, P. (1997). A polyhedral approach to the asymmetric traveling salesman problem, *Management Science* **11**(43): 1520–1536.

Flynn, M. (1972). Some computer organizations and their effectiveness, *IEEE Transactions on Computers* **9**(C-21): 948–960.

Fogarty, T. (1994). Co-evolving co-operative populatios of rules in learning control systems, *Evolutionary Computing* pp. 195–209.

Fogel, D. (1995). *Evolutionary Computation: Toward a new Philosophy of Machine Intelligence*, IEEE Press.

Fonseca, C., Mendes, E., Fleming, P. & Billings, S. (1993). Non-linear model term selection with genetic algorithms, *Natural Algorithms in Signal Processing* **2**: 27/1–27/8.

Foster, I. (1995). *Designing and Building Parallel Programs*, Addison-Wesley.

França, P., Gendreau, M., Laporte, G. & Müller, F. (1996). A tabu search heuristic for the multiprocessor scheduling problem with sequences dependent setup times, *International Journal of Production Economics* (43): 79–89.

França, P., Gupta, J., Mendes, A., Moscato, P. & Veltink, K. (2000). Metaheuristic approaches for the pure flowshop manufacturing cell problem, 7^th *International Workshop On Project Management and Scheduling*, Osnabrück, Germany.

França, P., Mendes, A. & Moscato, P. (2000). A memetic algorithm for the total tardiness single machine scheduling problem, *European Journal of Operational Research* **132**–**1**: 224–242.

Frangioni, A., Scutellá, M. G. & Necciari, E. (1999). Multi-exchange algorithms for the minimum makespan machinescheduling problem, *Technical Report TR-99-22*, Università di Pisa.

Frederickson, G., Hecht, M. & Kim, C. (1978). Approximation algorithm for some routing problems, *SIAM Journal on Computing* (7): 178–193.

Freisleben, B. & Merz, P. (1996). A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems, *International Conference on Evolutionary Computation*, pp. 616–621.

Garey, M. & Johnson, D. (1979). *Computers and Intractability: A Guide to the theory of NP-Completeness*, Freeman, San Francisco.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. & Sunderam, V. (1994). Pvm: Parallel virtual machine, a users' guide and tutorial for networked parallel computing,
http://www.netlib.org/pvm3/book/pvm-book.html. Html book.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley.

Goldberg, D. & Lingle, R. (1985). Alleles, loci and the travelling salesman problem, *1st. Int. Conf. on Genetic Algorithms*, pp. 154–159.

Gorges-Schleuter, M. (1989). Asparagos: An asynchronous parallel genetic optimization strategy, *Third International Conference of Genetic Algorithms*, p. 422.

Gorges-Schleuter, M. (1997). Asparagos an asynchronous parallel genetic optimization strategy, *Fourth International Conference on Evolutionary Computation*, pp. 171–174.

Gottlieb, A. & Almasi, G. (1989). *Highly Parallel Computing*, Benjamin Cummings, Redwood City, CA.

Graham, R., Lawler, E., Lenstra, J. & Rinooy Kan, A. (1979). Optimization and approximation in deterministic sequencing and scheduling: A survey, *Annals of Discrete Mathematics* **5**: 287–326.

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

Gropp, W., Lusk, E. & Skjellum, A. (1999). *Using MPI - 2nd Edition, Portable Parallel Programming with the Message Passing Interface*, MIT Press.

Grosso, P. (1985). *Computer simulations of genetic adaptation: Parallel subcomponent interaction in a multilocus model*, PhD thesis, The University of Michigan.

Hauser, R. & Männer, R. (1994). Implementation of standard genetic algorithm on mimd machines, *Parallel Problem Solving from Nature III*, Springer-Verlag, pp. 504–513.

Henriques, M. (1999). A proposal for java based massively parallel processing on the web, *The First Annual Workshop on Java for High-Performance Computing*, ACM International Conference on Supercomputing, Rhodes, Greece, pp. 59–66.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems*, The University of Michigan Press.

Holland, J., Holyoak, K., Nisbett, R. & Thagard, P. (1986). *Induction: Processes of Inference, Learning and Discovery*, MIT Press.

Holstein, D. & Moscato, P. (1999). *Memetic algorithms using guided local search: A case study*, McGraw-Hill, chapter New Ideas in Optimization, pp. 235–244.

Huntley, C. L. & Brown, D. E. (1996). Parallel genetic algorithms with local search, *Computers & Operations Research* **23**(6): 559–571.

Janikow, C. & Cai, H. (1992). A genetic algorithm application in nonparametric functional estimation, *2nd. Int. Conf. on Parallel Problem Solving from Nature*, pp. 249–258.

Jünger, M., Reinelt, G. & Rinaldi, G. (1995). The traveling salesman problem, *in* M. O. Ball, T. Magnanti, C. Monma & G. Nemhauser (eds), *Network Models, Handbook on Operations Research and Management Science*, Vol. 7, Elsevier, North Holland, pp. 225–230.

Koza, J. (1992). *Genetic Programming*, MIT Press.

Krishnakumar, K. & Goldberg, D. (1990). Genetic algorithms in control system optimization, *Proc. AAIA Guidance, Navigation, and Control Conf.* pp. 1568–1577.

Laporte, G. (1992). The traveling salesman problem: An overview of exact and approximate algorithms, *European Journal Of Operational Research* **2**(59): 231–247.

Lee, C. (1994). Genetic algorithms for single machine job scheduling with common due date and symmetric penalties, *Journal of the Operations Research Society of Japan* **2**(37): 83–95.

Lee, C. Y. & Kim, S. J. (1995). Parallel genetic algorithms for the earliness-tardiness job scheduling problem with general penalty weights, *Computers & Industrial Engineering* **28**(2): 231–243.

Lee, Y., Bhaskaran, K. & Pinedo, M. (1997). A heuristic to minimize the total weighted tardiness with sequence-dependent setups, *IIE Transactions* **29**: 45–52.

Lewis, H. & Papadimitriou, C. (1981). *Elements of the Theory of Computation*, Prentice-Hall International.

Lucasius, C. & Kateman, G. (1992). Towards solving subset selection problems with the aid of the genetic slgorithm, *2nd. Int. Conf. on Parallel Problem Solving from Nature*, pp. 239–247.

Mayer, M. (1999). A network parallel genetic algorithm for the one machine sequencing problem, *Computers ans Mathematics with Applications* (37): 71–78.

Mendes, A. (1999). *Algoritmos meméticos aplicados aos problemas de seqüenciamento em máquinas*, Master's thesis, Faculdade de Engenharia Elétrica e de Computação. Universidade Estadual de Campinas.

Mendes, A., França, P. & Moscato, P. (2000). Fitness landscape for the total tardiness single machine scheduling problem, *To appear in Neural Network World* .

Mendes, A., França, P. & Moscato, P. (2001). Npopt: An optimization framework for np problems, *International Conference of the Production and Operations Management Society*. Guarujá, SP, Brazil.

Mendes, A., Müller, F., França, P. & Moscato, P. (2002). Comparing meta-heuristic approaches for parallel machine scheduling problems with sequence-dependent setup times, *Production Planning & Control* **13**(2): 143–154.

Merz, P. & Freisleben, B. (1997). A genetic local search approach to the quadratic assignment problem, *in* T. Bäck (ed.), *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, Morgan Kaufmann, San Francisco, CA.

Mühlenbein, H. (1989). Parallel genetic algorithms, population genetics, and combinatorial optimization, *Evolution and Optimization '89* pp. 79–85.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs*, third edn, Springer.

Miller, D. & Pekny, J. (1991). Exact solution of large asymmetric traveling salesman problem, *Science* (251): 754–761.

Min, L. & Cheng, W. (1998). Identical parallel machine scheduling problem for minimizing the makespan using genetic algorithm combined with simulated annealing, *Chinese Journal of Electronics* **4**(7): 317–321.

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

Moscato, P. (2001). *Problemas de Otimização NP, Aproximabilidade e Computação Evolutiva: Da Prática à Teoria*, PhD thesis, Faculdade de Engenharia Elétrica e de Computação. Universidade Estadual de Campinas.

Moscato, P. & Norman, M. (1992). A memetic approach for the traveling salesman problem. implementation of a computational ecology for combinatorial optimization on message-passing systems, *Parallel Computing and Transputer Applications* pp. 187–194.

Neary, M. & Capello, P. (2000). Internet-based tsp computation with javelin++, *Workshop on Scalable Web Services (ICPP 2000 Workshops)*, Toronto, Canada, pp. 21–24.

Oussaidène, M., Chopard, B., Pictet, O. V. & Tomassini, M. (1997). Parallel genetic programming and its application to trading model induction, *Parallel Computing* (23): 1183–1198.

Raman, N., Rachamadugu, R. & Talbot, F. (1989). Real time scheduling of an automated manufacturing center, *European Journal of Operations Research* **40**: 222–242.

Rechenberg, I. (1973). Evolutionsstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution, Frommann-Holzboog Verlag, Stuttgart.

Rubin, P. & Ragatz, G. (1995). Scheduling in sequence dependent setup enviroment with genetic search, *Computers and Operations Research* **22–1**: 85–99.

Schlierkamp-Voosen, D. & Muehlenbein, H. (1994). Strategy adaptation by competing subpopulations, *Lecture Notes in Computer Science* **866**: 199–208.

Schwehm, M. (1993). A massively parallel genetic algorithm on the maspar mp-1, *in* R. C. . S. N. Albrecht R.F (ed.), *Artificial Neural Nets and Genetic Algorithms*, Springer-Verlag:New York, pp. 503–507.

Smith, D. (1985). Bin packing with adaptative search, *1st. Int. Conf. on Genetic Algorithms*, pp. 202–206.

Sumichrast, R. & Baker, J. (1987). Scheduling parallel processors: an integer linear programming based heuristic for minimizing setup time, *International Journal od Production Research* (25): 761–771.

Sun-microsystems (2001). The $java^{TM}$ tutorial,
http://java.sun.com/docs/books/tutorial/networking/sockets/definition.html.
On-line tutorial.

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

Tanenbaum, A. S. (1992). *Modern operating systems*, Prentice Hall.

Tanenbaum, A. S. (1995). *Distributed Operating Systems*, Prentice Hall.

Tanev, I., Uozumi, T. & Ono, K. (2001). Scalable architecture for parallel distributed implementation of genetic programming on network of workstations, *Journal of Systems Architecture* (47): 557–572.

Twardowski, K. (1993). Credit assignment for pole balancing with learning classifier systems, 5^th *Int.Conf. on Genetic Algorithms*, Morgam Kaufmann, pp. 238–245.

Wolpert, D. & Macready, W. (1996). No free lunch theorems for search, *Technical Report SFI-TR-95-02-010*, Santa Fe Institute.

Zhang, B.-T. & Mühlenbein, H. (1993). Genetic programming of minimal neural nets using occam's razor, *5th Int.Conf. on Genetic Algorithms*, pp. 342–349.