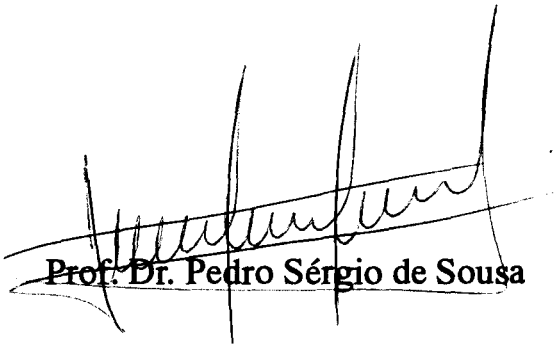
Metodologia de Especificação de Times Assíncronos para Problemas de Otimização Combinatória

Hélvio Pereira Peixoto

Metodologia de Especificação de Times Assíncronos para Problemas de Otimização Combinatória

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Hélvio Pereira Peixoto e aprovada pela comissão julgadora.

Campinas, 24 de março de 1995

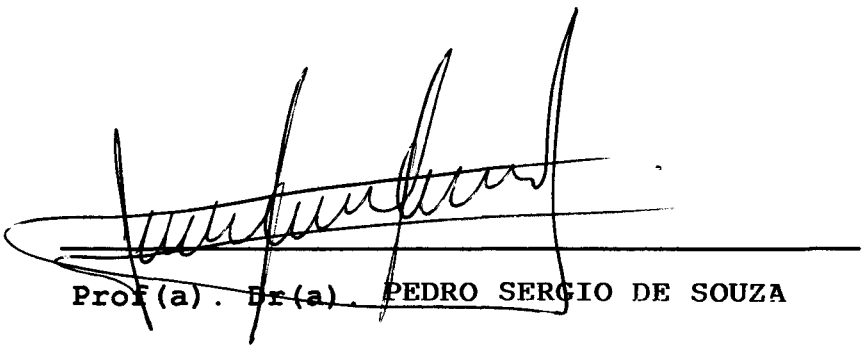


Prof. Dr. Pedro Sérgio de Sousa

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para obtenção do Título de MESTRE em Ciência da Computação.

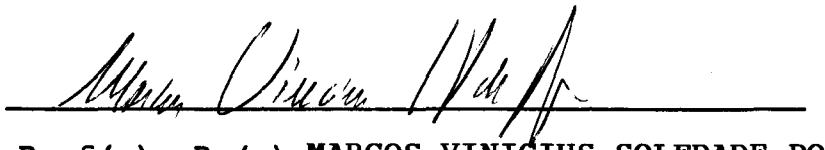
Tese defendida e aprovada em, 24 de março de 1995

Pela Banca Examinadora composta pelos Profs. Drs.



Prof(a). Dr(a). PEDRO SERGIO DE SOUZA

Prof(a). Dr(a). ELERI CARDOZO



Prof(a). Dr(a). MARCOS VINICIUS SOLEDADE POGGI DE ARAGÃO

Metodologia de Especificação de Times Assíncronos para Problemas de Otimização Combinatória[1]

Hélvio Pereira Peixoto[2]

Departamento de Ciência da Computação
IMECC - UNICAMP

Banca Examinadora:

- Pedro Sérgio de Souza (Orientador)[3]
- Marcus V. S. Poggi de Aragão[3]
- Cid Carvalho de Souza[3]
- Eleri Cardozo[4]

---

1. Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.
2. O autor é Bacharel em Ciência da Computação pela Universidade Federal de Uberlândia.
3. Professor do Departamento de Ciência da Computação - IMECC - UNICAMP.
4. Professor do Departamento de Engenharia da Computação e Automação Industrial - FEE - UNICAMP.

Agradecimentos

A Deus, responsável maior pela minha presença na terra e pelas dádivas concedidas nestes últimos dois anos.

À minha eterna namorada Daniella e minhas adoráveis filhas Lianna e Isabella, pela compreensão durante a minha ausência e pelo imenso amor e carinho dedicados na constante participação desta conquista.

A meus pais, Isabel e Jurandir, que muitas vezes renunciaram aos seus sonhos para que eu pudesse realizar os meus.

À minha irmã, Ângela Cristina, que sempre me deu apoio e incentivo para ir até o final desta jornada.

Ao meu orientador, Prof. Pedro Sérgio de Sousa, por ter sido um guia, um exemplo e sobretudo um amigo.

Aos amigos que tive o prazer de conhecer. Em particular, agradeço aos amigos Victor, Andersom, Elaine, Humberto, que se mostraram incansáveis frente aos problemas e adoráveis nos tantos momentos vividos juntos. Sei que sem eles esta caminhada teria maiores disabores.

Aos funcionários desta universidade, em particular os funcionários da secretaria do DCC, pela competência e prestatividade demonstrada nos serviços prestados.

A todas as demais pessoas que direta ou indiretamente contribuíram para o término deste trabalho.

Não poderia deixar de agradecer à fatia da população brasileira que tem contribuido com impostos, mesmo sem saber, para o desenvolvimento científico, tecnológico e cultural desta nação.

Resumo

Times Assíncronos perfazem uma nova técnica de resolução aproximada de problemas, baseada na utilização simultânea de diversos algoritmos heurísticos, que cooperam entre si de maneira sinérgica, encontrando soluções ótimas ou quase ótimas, as quais não seriam encontradas pelos mesmos algoritmos quando executados isoladamente. Esta nova técnica tem sido aplicada com sucesso a problemas combinatórios de grande porte.

O tema central deste trabalho é o desenvolvimento de uma metodologia de especificação de Times Assíncronos, em particular para os problemas de Otimização Combinatória de uma única função objetivo, tendo em vista a não existência de trabalhos neste sentido. O objetivo é fornecer uma seqüência de passos e sugestões que venham a facilitar e agilizar a concepção e implementação de Times Assíncronos.

Como um exemplo de aplicação da metodologia proposta, abordou-se o problema clássico de escalonamento de tarefas *Flow Shop Problem* de permutação, para o qual foram especificados e implementados Times Assíncronos. Os resultados obtidos por esses Times Assíncronos sobre as instâncias testadas foram tão bons quanto ou superiores às melhores soluções conhecidas. Esses testes foram efetuados de forma paralela, mostrando uma aceleração linear na obtenção dos resultados, conforme o número de processadores utilizados. Além dos Times Assíncronos para o *Flow Shop Problem*, desenvolveram-se novas fórmulas para cálculo de limites inferiores, demonstrando, assim, a otimalidade de algumas soluções obtidas e melhorando os limites inferiores conhecidos de dezenas de outras instâncias.

Abstract

Asynchronous Teams (A-Teams) are a new problem resolution technique that uses simultaneously various heuristic algorithms. These algorithms cooperate synergically one with the other to find optimal or nearly optimal solutions that would not be found through isolated algorithms. This technique has been successfully applied to large combinatorial problems.

The main objective of this work is the development of a methodology to specify Asynchronous Teams to Combinatorial Optimization Problems with one objective function, since there is no literature about that. The purpose is to generate a sequence of steps and suggestions making the conception and implementation of Asynchronous Teams easy and quick.

As an example of the proposed methodology, Asynchronous Teams were specified and implemented to the classical permutation Flow Shop Problem. The results obtained by these A-Teams over the tested instances were equivalent or better than those published as the best known values. These A-Teams were executed in a parallel computer, showing linear speed up in the number of processors. Not only were the A-Teams to FSP developed, but do were two new formulas to calculate lower bounds. These lower bounds proved the optimally of two instances and improved the lower bounds of many others.

Conteúdo

Agradecimentos vi

Resumo viii

Abstract ix

Capítulo 1 *Introdução* 1

1.1 Objetivos do trabalho 2

1.2 Caracterização dos Problemas de Otimização Combinatória 3
1.2.1 Definições 3
1.2.2 Aplicações 3
1.2.3 Complexidade 4

1.3 Organização da Dissertação 5

Capítulo 2 *Métodos de Resolução de Problemas de Otimização Combinatória* 7

2.1 Introdução 8

2.2 Métodos Exatos 8

2.2.1 Branch-and-Bound 8
2.2.2 Programação Dinâmica 9
2.2.3 Branch-and-cut 9
2.3 Métodos Aproximados 10
2.3.1 Heurísticas 10
2.3.2 Meta-Heurísticas 10
2.4 Times Assíncronos - um novo método aproximado 14
2.5 Notas Bibliográficas 14

Capítulo 3 *Times Assíncronos* 15

3.1 Introdução 16
3.2 Modelos Organizacionais 17
3.2.1 O cérebro humano e os neurônios 18
3.2.2 Insetos 18
3.2.3 Organizações Humanas 18
3.2.4 Um modelo organizacional 20
3.3 Histórico 22
3.3.1 O Precursor 23
3.4 Definições e Características 23
3.5 Exemplo Natural de A-Teams 25
3.5.1 Algumas Colônias de Insetos 25
3.6 Exemplo Artificial de A-Teams 26
3.6.1 A-Teams para o Problema do Caixeiro Viajante 26
3.7 Sumário 28
3.8 Notas Bibliográficas 28

Capítulo 4 *Metodologia de Especificação de Times Assíncronos* 29

4.1 Introdução 30
4.2 Especificação do Problema 30
4.3 Definição da Representação e Padrão de Qualidade das Soluções 30
4.4 Adequabilidade de Problemas ao uso da Metodologia de Especificação de A-Teams 33
4.4.1 Espaço dos Problemas 33
4.4.2 Espaços de Variáveis 34
4.4.3 Problemas Multi-Algoritmos 37
4.5 Classificação dos Algoritmos Disponíveis 37
4.6 Uma Estrutura Geral de A-Teams 41
4.7 As Memórias 45

4.7.1 Tamanho das Memórias 45
4.7.2 Avaliação das informações armazenadas pelas memórias 45
4.7.3 O Armazenamento de Soluções 46
4.7.4 Iniciação das Memórias 46
4.8 Os Agentes 47
4.8.1 A escolha dos algoritmos e heurísticas 49
4.9 Métodos de Destruição 50
4.9.1 Destruição da Pior Solução 51
4.9.2 Destruição de qualquer solução com distribuição uniforme de probabilidade 52
4.9.3 Destruição de soluções com distribuição linear de probabilidade 53
4.9.4 Destruição Dinâmica, conforme a diversidade da memória 53
4.10 Implementação 56
4.11 Validação e Modificações 56
4.12 Sobre a Representação Gráfica 58
4.13 Sumário 60
4.14 Notas Bibliográficas 62

Capítulo 5 *A-Teams para o Flow Shop Problem* 63

5.1 Definição do Flow Shop Problem 64
5.2 Complexidade do FSP 66
5.3 Adequabilidade do FSP a A-Teams 67
5.4 Algoritmos para o FSP 67
5.4.1 Heurísticas baseadas em métodos de ordenação 67
5.4.2 Heurística de Nawas, Enscore e Ham (NEH) 69
5.5 Estruturas de A-Teams para o FSP 71
5.5.1 Estrutura de Fluxo de Dados utilizada para o FSP 73
5.6 As Memórias 74
5.7 Os Agentes 74
5.8 Métodos de Destruição 75
5.9 Implementação 75
5.10 Instâncias de Testes 76
5.11 Testes Computacionais 77
5.11.1 O tamanho das memórias 77
5.11.2 As políticas de destruição 77
5.11.3 Eficiência em Escala 80
5.11.4 Resultados 84
5.11.5 Processamento Distribuído 88
5.12 Conclusões 89

5.13 Sumário 89
5.14 Notas Bibliográficas 89

Capítulo 6 *Novos Limites Inferiores para o Flow Shop Problem* 91

6.1 Introdução 92
6.2 LB Simples calculado em máquinas (LB1 e LB2) 92
6.3 LB Completo calculado em máquinas (LB3) 93
6.4 LB Sem Interferência (LB4) 94
6.5 LB Calculado com base em jobs (LB5) 95
6.6 LB com base na relaxação das capacidades das máquinas (LB6) 96
6.7 Outro LB Calculado com base em jobs (LB7) 97
6.8 Primeiro LB proposto (*LB8*) 97
6.9 Segundo LB proposto (*LB9*) 98
6.10 Resultados 98
6.11 Conclusões 99
6.12 Notas Bibliográficas 99

Capítulo 7 *Conclusões* 101

7.1 Contribuições 102
7.2 Possíveis extensões 102

Apêndice A *Novos Limites Infeirores para instâncias do FSP* 103

Apêndice B *Lista de Resultados* 109

Referências 111

Lista de Figuras

Figura 3.1 Seqüências possíveis para as heurísticas A, B e C de melhoria e D de construção... 17

Figura 3.2 Taxonomia organizacional. A primeira coluna representa o tipo de fluxo de dados (FD) e controle (FC), representado por [FD, FC], em que FD e FC assumem os valores Nulo (N), Cíclico (C) e Acíclico (A). A segunda coluna exemplifica um elemento típico desta estrutura organizacional. A última coluna descreve, sucintamente, a respectiva organização de software. Os grafos com retângulos e arcos contínuos, círculos e arcos pontilhados indicam, respectivamente, fluxo de dado e de controle..... 21

Figura 3.3 Um "time" composto por um *Blackbord* e dois algoritmos que representam dois métodos distintos de resolução de equações algébricas, SD e NR. 22

Figura 3.4 Exemplo da representação gráfica de um *A-Team* de duas memórias e oito agentes. 24

Figura 3.5 *A-Team* proposto por Souza para o problema do Caixeiro Viajante..... 27

Figura 4.1 Classificação dos problemas segundo os algoritmos disponíveis para resolvê-los .. 33

Figura 4.2 Diagrama de decomposição de problemas. Setas representam algoritmos disponíveis para resolver um problema, mapeando elementos de um espaço para outro..... 35

Figura 4.3 Um exemplo de algoritmo de Consenso por Intersecção de soluções para o problema de escalonamento *Flow Shop Problem* de Permutação de 5 jobs. Uma nova solução parcial é construída através da intersecção de duas soluções originais A e B 38

Figura 4.4 Exemplo de um algoritmo de Consenso por União, o algoritmo *Mixer*, proposto por Souza para o PCV. Dadas duas soluções A e B, o *Mixer* constrói uma nova solução completa utilizando apenas arestas presentes nas soluções A e B 39

Figura 4.5 Estrutura geral proposta de um *A-Team* envolvendo todos os Agentes compostos por algoritmos que seguem a classificação dada. Os retângulos representam as memórias compartilhadas e as setas as classes de Agentes. 42

Figura 4.6 Alguns tipos de Soluções para o PCV. 43

Figura 4.7 Estrutura dos Agentes. O protocolo de Leitura e Escrita se refere a maneira pela qual o Agente se comunica com a memória de onde retira e deposita os dados, respectivamente. 47

Figura 4.8 Exemplo do uso de algoritmos que não pertencem à classificação da seção 4.5. O algoritmo A lê de uma solução parcial, constrói uma nova solução, a qual pode ser armazenada tanto na memória de soluções Factíveis como na de soluções Infactíveis, dependendo ou não da violação de restrições. 49

Figura 4.9 Política de destruição: destruição da pior solução. Sempre que não houver espaço na memória para alocar novas soluções, esta política seleciona a pior solução com probabilidade 1 de destruí-la. Novas soluções que sejam inferior em qualidade à pior solução da memória não são aceitas. 51

Figura 4.10 Política de destruição: destruição de qualquer solução com distribuição uniforme de probabilidade. Sempre que não houver espaço na memória para alocar novas soluções, esta política seleciona qualquer solução presente na memória. Novas soluções que sejam inferiores em qualidade à pior solução da memória não são aceitas. 52

Figura 4.11 Política de destruição: destruição de soluções com distribuição linear de probabilidade. Sempre que não houver espaço na memória para alocar novas soluções, esta política seleciona com probabilidade linear uma solução presente na memória (exceto a melhor). Novas soluções que sejam inferiores em qualidade à pior solução da memória não são aceitas. 53

Figura 4.12 Exemplo de uma política de destruição dinâmica. Num primeiro instante, sempre que não houver espaço na memória para alocar novas soluções, esta política seleciona com probabilidade linear uma solução presente na memória (exceto a melhor). Novas soluções que sejam inferiores em qualidade à pior solução da memória não são aceitas. Quando é considerado um baixo nível de diversidade, esta política seleciona qualquer uma das x piores soluções presentes na memória e novas soluções com qualquer qualidade são aceitas. Num último instante, após a inserção de z soluções e conseqüente aumento na diversidade, esta política volta a operar como inicialmente. 55

Figura 4.13 Exemplo do decrescimo da Taxa de aceitação de soluções de dois Agentes A e B, em instantes diferentes de tempo. 57

Figura 4.14 A equivalência entre a representação descrita no capítulo 3 e a representação proposta nesta seção. Ambas representam um Agente A que lê seus dados de entrada tanto de M1 como de M2. 59

Figura 4.15 Exemplo de um *A-Team* com 5 Agentes e 4 memórias, segundo a representação proposta. 60

Figura 4.16 Algoritmo de construção de *A-Teams*, conforme a metodologia proposta..... 61

Figura 5.1 Relação entre o *Flow Shop Problem* de permutação (FSP) e os demais problemas clássicos de escalonamento..... 65

Figura 5.2 Exemplo de uma instância do FSP de três jobs e duas máquinas. O gráfico de Gantt mostrado na figura corresponde à representação gráfica da solução (1,2,3)..... 66

Figura 5.3 Exemplo de uma iteração do algoritmo M sobre as seqüências S1 e S2. A primeira posição da seqüência S3 é preenchida com o job 3 pois, neste exemplo hipotético, o job 3 na primeira posição causa um menor *Makespan* que o job 2. 68

Figura 5.4 Exemplo de duas iterações do algoritmo IE. Na primeira iteração verifica-se que a permutação entre os jobs da posição um e dois não reduz o *Makespan*. Neste caso a ordem permanece a mesma. Na segunda iteração permutam-se com sucesso os jobs das posições 2 e 3, obtendo uma nova seqüência de menor *Makespan*. 69

Figura 5.5 Estrutura geral de *A-Teams* para o FSP composta pelas heurísticas IE, GE, M, MNEH e Cheap-NEH. As setas pontilhadas indicam a ausência de algoritmos para a formação de ciclos..... 71

Figura 5.6 Exemplo do algoritmo de Junção sobre um FSP de seis jobs. A partir de duas sub-seqüências, constrói-se uma nova seqüência através da juncão de seus jobs. 72

Figura 5.7 Estrutura de um *A-Team* para o FSP. Os agentes GE e MNEH leêm e escrevem na Memória de Soluções Completas e Refinadas. Partibilidade e Consenso leêm de Soluções Completas e Refinadas e escrevem, respectivamente, em Sub-Soluções e Soluções Parciais. M e J leêm de Sub-Soluções e escrevem em Soluções Não Refinadas, assim como Cheap-NEH que lê de Soluções Parciais. O agente IE lê de Soluções Não Refinadas e escreve em Soluções Completas e Refinadas. As setas pontilhadas indicam os agentes Destruidor e Iniciador, cujos algoritmos são dependentes das políticas adotadas..... 73

Figura 5.8 Comportamento da melhor e pior soluções na memória de Soluções Completas e Refinadas do *A-Team* resolvendo FSP de 100x10, utilizando o fluxo de dados apresentado na figura 5.7 e as políticas de destruição D1 (a) e D2 (b). Após uma e quatro horas de procesamento, as curvas não apresentaram nenhuma alteração..... 79

Figura 5.9 Comportamento da melhor e pior soluções na memória de Soluções Completas e Refinadas do *A-Team* resolvendo FSP de 100x10, utilizando o fluxo de dados apresentado na figura 5.7 e a política de destruição D3. Observe que não há restrição no valor da pior solução..... 80

Figura 5.10 Comportamento da melhor e pior soluções na memória de Soluções Completas e Refinadas do *A-Team* resolvendo FSP de 100x10, utilizando o fluxo de dados apresentado na figura 5.7 e a política de destruição D4. A figura (a) e (b) correspondem, respectivamente, às janelas de tempo de quatro e quarenta horas. .. 81

Figura 5.11 Comportamento da melhor e pior soluções na memória de Soluções Completas e Refinadas do *A-Team* resolvendo FSP de 100x10, utilizando o fluxo de dados apresentado na figura 5.7 e a política de destruição D5. 82

Figura 5.12 Fluxos de Dados utilizados para mostrar a performance de cada um dos quatro cíclos possíveis do *A-Team* para o FSP (veja figura 5.7). Cíclos são utilizados uma vez que apenas os algoritmos GE e MNEH são capazes de produzirem soluções válidas por si só. 83

Figura 5.13 Performance individual dos algoritmos num *A-Team*, conforme as estruturas da figura 5.12. Nenhum algoritmo (ou ciclo) individual foi capaz de alcançar o *Makespan* da melhor solução conhecida. 85

Figura 5.14 Fluxo de dados usados para mostrar eficiência em escala. A partir de um ciclo (Partibilidade, J, M e IE), introduzimos o algoritmo de mehoria GE, um segundo ciclo (Consenso, Cheap-NEH e IE) e finalmente o algoritmo MNEH. 86

Figura 5.15 Performance dos *A-Teams* após a adição sucessiva de agentes. Cada curva mostra a performance de um *A-Team* da figura 5.14. A curva (a) corresponde ao ciclo formado pelos agentes Partibilidade, J, M e IE. As curvas (b), (c) e (d) correspondem à adição dos agentes GE, ciclo Consenso, Cheap-NEH e, finalmente, ao agente MNEH. À medida que novos algoritmos são adicionados, a performance do time é monotonicamente melhoriada (tempo e qualidade). O *A-Team* (d) encontrou melhores soluções que as publicadas recentemente [Tai93]. 87

Figura 5.16 Execução paralela do *A-Team* para a instância 100x10 utilizando um, dois, quatro e oito procesadores. A linha mostra a redução do tempo após adições sucessivas de processadores e os pontos as médias dos *Makespans* das três execuções. 88

Figura 5.17 Execução paralela do *A-Team* para a instância 100x10 utilizando um, dois, quatro e oito procesadores. A linha indica os valores esperados para um *speed up* linear. Os pontos mostram as médias dos resultados obtidos pelos *A-Teams*. 89

Capítulo 1 *Introdução*

A realidade no setor de serviços e de produção está marcada pela competitividade e qualidade total [JG91]. Mais do que nunca os clientes insistem em maior funcionalidade e qualidade a preços menores. Para sobreviver, os prestadores de serviços e indústrias têm investido montantes consideráveis na modernização e automação de suas organizações. Por modernização entende-se a busca pela qualidade, a criação da imagem desejada da organização, a valorização do cliente e do funcionário, o incentivo a líderes facilitadores e partidários, a busca de métodos eficientes na execução das tarefas, etc [JG91]. A automação consiste na introdução de mecanismos computadorizados com o objetivo de melhorar, facilitar e agilizar as tarefas desempenhadas na organização. Esses motivos, aliados aos baixos custos atuais dos computadores, justificam a crescente automação.

Nas primeiras três décadas da era da computação (1950-1970), a principal preocupação era reduzir os custos de processamento e armazenamento de dados. O resultado foi o avanço tecnológico da micro-eletrônica, levando a computadores menores, mais baratos, confiáveis e rápidos. Hoje o cenário é outro. Tanto o hardware como o software são projetados visando à melhoria da qualidade das soluções de problemas cada vez mais complexos, bem como à redução do tempo necessário para obtê-las.

A *Otimização Combinatória* surgiu a partir destes problemas complexos e do desejo de gerenciar e utilizar eficientemente os recursos disponíveis. Desde então, vários métodos surgiram com o intuito de resolvê-los da melhor maneira possível. Porém, como foi observado no final dos anos 60, parecem existir duas classes de problemas, conforme o esforço necessário na obtenção das melhores soluções (ótimas): a classe dos problemas "fáceis" (P) e a classe dos problemas "difíceis" (NP-Difícil) [GJ79]. Para os problemas da primeira classe, existem algoritmos que produzem as soluções ótimas

em tempo polinomial em relação ao tamanho da instância que se resolve. Já os problemas da segunda classe, apresentam uma complexidade inerente que, até o momento, nos obriga a um esforço super-polinomial para a obtenção das soluções ótimas. O resultado desta conjectura é que, para alguns problemas de tamanho moderado, mesmo de posse do mais rápido dos computadores, encontrar a solução ótima pode demandar séculos de processamento. Nesse contexto, e devido à necessidade de resolver esses problemas, surgiram as Heurísticas, que são métodos aproximados de resolução de problemas, que encontram soluções de boa qualidade (não necessariamente as ótimas) num tempo geralmente pequeno.

Recentemente, Souza [Sou93] introduziu uma nova técnica de resolução aproximada de problemas que se vale da execução de várias heurísticas simultaneamente. O uso concomitante de várias heurísticas, através do compartilhamento de dados e soluções, permite que soluções de qualidade ótima ou quase ótima sejam encontradas mais efetivamente. Esta nova técnica, intitulada Times Assíncronos (do inglês *Asynchronous Teams* ou *A-Teams*), tem sido aplicada com sucesso a problemas de grande porte.

1.1 Objetivos do trabalho

O tema central deste trabalho é a descrição de uma Metodologia de Especificação de Times Assíncronos para Problemas de Otimização Combinatória, visto que não existe nenhum trabalho neste sentido. O objetivo é fornecer uma seqüência de passos e sugestões, colocando-os sob um mesmo enfoque (os problemas de Otimização Combinatória de uma única função objetivo) de modo a facilitar e agilizar a concepção e implementação de Times Assíncronos.

O presente texto é destinado, principalmente, às pessoas interessadas no projeto e funcionamento dos Times Assíncronos. Pretende-se, ainda, oferecer um compêndio da literatura disponível de *A-Teams*, permitindo, assim, que estudos sobre casos específicos possam ser encontrados.

O segundo objetivo deste trabalho é o desenvolvimento de *A-Teams* para o problema de escalonamento de tarefas conhecido por *Flow Shop Problem*. Este é um problema clássico de escalonamento de tarefas de grande utilidade prática e pertencente à classe dos problemas NP-Difíceis [GJ79].

O restante deste capítulo descreve as principais características dos problemas de Otimização Combinatória e a organização desta dissertação.

1.2 Caracterização dos Problemas de Otimização Combinatória

1.2.1 Definições

A escassez de recursos acompanha a história humana há muito tempo. Nas últimas décadas, porém, a explosão populacional, as crises mundiais (energéticas, econômicas, guerras, catástrofes, etc) a universalização da economia e a competitividade em todos os setores não deixaram outra saída senão o uso comedido (otimizado) dos poucos e/ou caros recursos disponíveis.

Assim, a Otimização Combinatória ganhou importância entre os administradores e interesse por parte dos pesquisadores e, desde então, vem sofrendo rápida evolução.

Muitos problemas consistem em encontrar a melhor solução ou a solução ótima dentre um número finito ou infinitamente contável de soluções alternativas. A esses problemas dá-se o nome de *Problemas de Otimização* [PS82]. Antes de definir formalmente os problemas de Otimização Combinatória, definiremos as instâncias e os problemas de otimização.

> Uma *Instância de um problema de otimização* é um par (F, C), onde F é um conjunto de pontos factíveis e C é a função de custo que se deseja maximizar ou minimizar, tal que:
>
> $$C: F \rightarrow \Re \quad . \qquad \text{(Eq 1.1)}$$
>
> Um *Problema de Otimização* consiste em encontrar um $x \in F$ tal que:
>
> $$C(x) \leq C(y), \forall y \in F \qquad \text{(Eq 1.2)}$$

Se o conjunto F for discreto, diz-se então que o problema é de Otimização Combinatória. Nemhauser e Wolsey [NW88] propõem a seguinte definição:

> *Otimização Combinatória* trata problemas de maximização ou minimização de uma função de uma ou mais variáveis, sujeitos a: restrições de igualdade ou desigualdade e restrições de integralidade de algumas ou de todas as variáveis.

1.2.2 Aplicações

A definição de Nemhauser e Wolsey é genérica o suficiente para englobar um grande número de problemas que podem ser modelados como problemas de Otimização Combinatória. Como exemplo, temos:

Caminhos Mínimos em Grafos [CLR90]: Dado um conjunto de n cidades (vértices) e m estradas (arestas) que as interligam, qual o caminho mais curto da cidade a para todas as demais $(n-1)$ cidades?

Fluxo Máximo em Redes [BJS90]: Dado um conjunto de *n* refinarias de petróleo (vértices) e *m* dutos (arestas) que as interligam, como bombear petróleo de uma refinaria *a* para outra *b* minimizando os custos?

Árvore de Espalhamento Mínimo [CLR90]: Dado um conjunto de *n* cidades (vértices) e as distâncias entre todas elas (todas as arestas com custos associados), como conectar todas as cidades através de linhas telefônicas, minimizando a quantidade de fios necessários?

Coloração de Vértices [BM76]: Dado um conjunto de *n* elementos químicos (vértices) e um conjunto de pares de incompatibilidades entre os elementos (arestas), como agrupá-los compativelmente de forma a minimizar o número de containers (cores) necessários para armazená-los?

Caixeiro Viajante [LLKS85]: Dado um conjunto de *n* pontos (vértices) de uma placa (chip) na qual um canhão de solda deve atuar, qual deve ser a rota do canhão de solda tal que o espaço total por ele percorrido seja minimizado ?

Knapsack [MT90]: Dado um conjunto de *n* volumes, cada volume com um valor associado, como agrupar um subconjunto destes numa mochila sem exceder a capacidade da mochila e maximizando o valor dos volumes armazenados?

Flow Shop Problem [Bak74]: Dado um conjunto de *n* serviços (jobs) e *m* máquinas (processadores), qual a seqüência de serviços que minimiza o tempo total de processamento de todos eles?

Roteamento de Veículos [Lap92]: Dado um conjunto de clientes, uma frota de caminhões e um conjunto de depósitos, como satisfazer os pedidos dos clientes minimizando os custos de transporte?

Inúmeros outros problemas podem ser modelados como problemas de Otimização Combinatória. Problemas como planejamento de produção, planejamento de investimentos, seqüenciamento de genes, desenvolvimento de novas moléculas, projeto de redes de telecomunicações, posicionamento de satélites, projeto e produção de circuitos VLSI e escalonamento de trens são apenas alguns outros exemplos. A lista ainda se estende por áreas como arqueologia, esportes e psicologia, dentre outras[GL93].

1.2.3 Complexidade

Todos os problemas de Otimização Combinatória podem ser convertidos em sua versão de decisão, ou seja, a sua solução consiste em uma resposta positiva ou negativa [GJ79]. A transformação de um problema de otimização em sua versão de decisão pode ser assim entendida: use a versão de otimização para obter a solução ótima e então responda à pergunta da versão de decisão.

Os problemas de Otimização Combinatória podem ser classificados pela Teoria da Complexidade [GJ79]. Nessa Teoria, todo problema é tratado como sendo um

conjunto de parâmetros (definição das instâncias) e um conjunto de propriedades (restrições do problema) às quais as soluções devem satisfazer. Sendo assim, entre instâncias de um mesmo problema, as únicas variações estão nos conjuntos de parâmetros. A Teoria da Complexidade considera o tamanho das instâncias como sendo a quantidade de *bits* necessária para representá-las em computadores digitais. A questão é: qual o número de computações (operações em *bits*) necessárias para obter a solução ótima? Conforme o número de computações necessárias para obter a solução ótima, os problemas podem ser classificados em três classes principais:

P Problemas onde o número de computações do melhor algoritmo conhecido cresce polinomialmente em função do tamanho da instância.

Intratáveis Problemas onde o número de computações do melhor algoritmo conhecido cresce super-polinomialmente em função do tamanho da instância e existe a garantia da não existência de algoritmos melhores.

NP Problemas onde o número de computações do melhor algoritmo conhecido cresce super-polinomialmente em função do tamanho da instância, não existe a garantia da não existência de algoritmos melhores e, dada uma solução para o problema, pode-se verificar, em tempo polinomial, se a solução dada satisfaz o problema de decisão positivamente ou não.

Cook, em 1971, provou que todos os problemas da classe NP podem ser reduzidos polinomialmente ao problema da Satisfatibilidade (SAT) [Coo71]. Isto significa que se alguém propuser um algoritmo para o SAT, que resolva qualquer uma de suas instâncias em tempo polinomial, então todos os problemas da classe NP também podem ser resolvidos em tempo polinomial em relação ao tamanho da entrada (P=NP). Os demais problemas que também possuem esta mesma propriedade do SAT são considerados pertencentes à classe dos problemas NP-Difíceis. A intersecção das classes dos problemas NP-Difíceis e NP define a classe dos problemas NP-Completos. Desde então, vários problemas foram identificados como sendo NP-Difíceis. Como por exemplo: o problema do Caixeiro Viajante, Coloração de Vértices, *Knapsack*, *Flow Shop Problem* e Roteamento de Veículos [GJ79].

Embora parte significativa dos problemas práticos sejam NP-Difíceis, muitos outros não o são. O problema de Caminhos Mínimos, o problema de Fluxo em Redes e o problema da Árvore de Espalhamento Mínimo são exemplos para os quais existem algoritmos polinomiais eficientes capazes de encontrar soluções ótimas.

1.3 Organização da Dissertação

Esta dissertação está organizada como segue: o segundo capítulo oferece uma resenha dos principais métodos atualmente utilizados na resolução dos problemas de Otimização Combinatória. Em seguida fornecemos a descrição dos Times Assíncronos, conforme proposto por Souza [Sou93]. O capítulo 4 apresenta a metodologia proposta

para especificação de Times Assíncronos, e o capítulo 5 a aplicação desta metodologia ao problema de escalonamento de tarefas *Flow Shop Problem* de permutação e os resultados decorrentes. O capítulo 6 descreve dois Limites Inferiores propostos por este autor para o *Flow Shop Problem* de permutação. O último capítulo apresenta as conclusões e contribuições deste trabalho, seguidas das sugestões para trabalhos futuros.

Capítulo 2

Métodos de Resolução de Problemas de Otimização Combinatória

Com o objetivo de fornecer ao leitor uma visão geral das técnicas geralmente utilizazas na resolução de problemas de Otimização Combinatória, este capítulo aborda, brevemente, diversos métodos encontrados na literatura. Esses métodos podem ser classificados em duas classes gerais: Métodos Exatos e Métodos Aproximados, conforme a garantia de otimalidade das soluções por eles obtidas. As seções que seguem descrevem os principais representantes dessas duas classes, seguidos de uma introdução a um novo método aproximado, os *A-Teams*.

2.1 Introdução

Otimizar, como foi visto, consiste em encontrar um máximo ou mínimo de uma dada função, sobre um domínio previamente definido. Algumas teorias de otimização tratam de problemas em que o domínio é infinito, como por exemplo Calculo Diferencial. Sob esta ótica, pode parecer que problemas de Otimização Combinatória em que o domínio da função objetivo é finíto sejam mais "fáceis" que os primeiros: basta encontrar, dentre todas as possíveis soluções de um conjunto finito, a melhor delas. Porém, todas as possíveis soluções podem ser um número finito muito grande, como por exemplo, o número de circuitos Hamiltoniano num grafo completo de n vértices, mesmo para valores moderados de n [BM76].

Conforme ressaltado pela teoria da complexidade, muitos problemas de Otimização Combinatória parecem não possuir algoritmos eficientes que encontrem soluções ótimas em tempo polinomial [GJ79]. Neste caso, como provar que uma dada solução é ótima? Como não existe nenhuma teoria de Dualidade[1] para os problemas NP-Difíceis, uma opção é a força bruta: enumerar todas as possíveis soluções e escolher a melhor. Porém, pode-se utilizar de alguns artifícios na tentativa de minorar as computações necessárias na obtenção de tais soluções.

A próxima seção descreve três estratégias comumente usadas para a obtenção de soluções garantidamente ótimas.

2.2 Métodos Exatos

2.2.1 Branch-and-Bound

A idéia deste método é encontrar limites superiores e inferiores bastante próximos ao valor ótimo, para assim reduzir o número de passos enumerativos. Para ser mais específico, assumimos que o problema é de minimização e que Ω é o conjunto de todas as soluções factíveis.

Para implementar um *Branch-and-Bound* é preciso de um algoritmo para calcular limites inferiores. Esse algoritmo será utilizado pelo *Branch-and-Bound* sempre que novos sub-problemas forem criados.

A cada passo o *Branch-and-Bound* divide recursivamente o problema original em novos (dois ou mais) sub-problemas menores e verifica, em relação aos limites superiores e inferiores, a viabilidade de cada um desses sub-problemas. Inicialmente, pode-se calcular um limite superior para a instância que se resolve através do uso de heurísticas. Ao dividir o problema original em n sub-problemas, verifica-se quais destes sub-

1. A teoria da Dualidade para problemas de programação linear pode ser usada, por exemplo, para provar a otimalidade de uma dada solução factível [NW88].

problemas possuem limites inferiores maiores que o limite superior disponível, ou seja, qual dos sub-problemas levará garantidamente a soluções piores que a melhor disponível. Neste ponto ocorre a eliminação de parte do conjunto Ω que não será enumerado. Por esse motivo, é importante a existência de bons algoritmos para calcular os limites inferiores e superiores. A divisão do problema corresponde ao *Branching* e a verificação da viabilidade ou não de um sub-problema corresponde ao *Bounding*.

O *Branch-and-Bound* termina quando não existir mais nenhum sub-problema para ser resolvido. Desse modo, esse método só termina se o número de soluções factíveis em Ω for finito.

Embora esse método seja matematicamente trivial, na prática é trabalhoso torná-lo computacionalmente eficiente. Mesmo que se consigam bons algoritmos para o cálculo dos limites inferiores, no pior caso, esses métodos enumeram todas as soluções factíveis.

2.2.2 *Programação Dinâmica*

A programação dinâmica surgiu através da modelagem de processos seqüenciais de decisão, em instantes discreto do tempo. A idéia utilizada é a seguinte: suponha que para encontrar uma solução de um problema seja necessário efetuar n decisões, digamos D_1, D_2, ..., D_n. Se a solução obtida for ótima, então diz-se que a seqüência de decisões associada é ótima. Conseqüentemente, as últimas k decisões também são ótimas, ou seja, D_{n-k+1}, D_{n-k+2}, ..., D_n. Este princípio é denominado *princípio da otimalidade* [PS82].

Em termos de problemas de Otimização Combinatória, a essência da Programação Dinâmica está em dividir o problema original em estágios onde são efetuadas as decisões e em encontrar uma relação de recorrência que permita voltar aos estágios anteriores. Portanto, a Programação Dinâmica constrói uma solução ascendentemente (*botton-up*) e, sempre que uma solução não for ótima, a Programação Dinâmica retorna a estágios anteriores e efetua outras decisões [AHU74]. Virtualmente, todo problema de otimização pode ser modelado por Programação Dinâmica. Porém, geralmente, as soluções ótimas não são encontradas eficientemente, pois o número de estados pode ser exponencial.

2.2.3 *Branch-and-cut*

Esse método é baseado na resolução sucessiva de programas lineares, através da adição de restrições e variáveis. O problema original é relaxado e, então, considerado como problema de programação linear. Resolve-se o problema relaxado e, sempre que a solução obtida pelo programa linear violar alguma das restrições de integralidade do problema original, adiciona-se mais uma restrição, com a esperança de que a solução ótima seja obtida sem a introdução de todas as restrições necessárias do problema original (que pode ser um número exponencial) [GP85]. Esse método requer um

estudo teórico profundo do problema que se pretende resolver e um grande esforço de implementação para torná-lo eficiente.

2.3 Métodos Aproximados

2.3.1 Heurísticas

As heurísticas podem ser definidas como sendo procedimentos que encontram soluções aceitáveis (eventualmente ótimas) em tempo geralmente pequeno. Embora as soluções encontradas não sejam garantidamente ótimas, as grandes virtudes das heurísticas consistem na simplicidade de implementação e na rapidez para encontrar uma solução.

2.3.2 Meta-Heurísticas

São heurísticas que podem ser aplicadas a problemas Combinatórios em geral, bastando pequenas alterações para ajustá-las ao problema em foco. As Meta-Heurísticas mais utilizadas são: Busca Local, *Simulated Annealing*, *Tabu Search* e Algoritmos Genéticos.

2.3.2.1 Busca Local ou Método Descendente

Dada uma solução, a Busca Local procura encontrar outras soluções que sejam melhores que a original, em um curto intervalo de tempo. Mais explicitamente, essas Meta-Heurísticas consistem em melhorar uma solução através de sucessivas trocas e/ou rearranjos de seus elementos, até que não seja mais possível melhorar o valor da função objetivo. Nesse instante a Busca Local atingiu um mínimo local, que pode não ser o mínimo global. À solução S' que é obtida através de uma troca ou rearranjo de elementos (movimento) a partir de uma solução S, dá-se o nome de vizinha de S. O conjunto de todos os vizinhos de uma solução S, denotado por $V(S)$, é denominado vizinhança de S. Nesse contexto, e definindo $F(S)$ como sendo o valor da função objetivo para a solução S, o algoritmo de Busca Local pode ser assim formulado:

Algoritmo Busca Local

Selecione uma solução S como sendo a solução corrente.
Continua = Verdade
Enquanto *Continua* **faça**:
  Encontre uma solução $S' \in V(S)$ tal que $F(S') - F(S)$ seja minimal
  Se $F(S') - F(S)$ for negativo
    Então S' se torna a solução corrente S
    Senão *Continua* = Falso
Fim

Nesse algoritmo, a última solução corrente consiste na melhor solução encontrada.

Na prática, o tamanho do conjunto V(S) pode ser muito grande e a tarefa de escolher o melhor vizinho pode não ser simples. Nesses casos, constuma-se usar um subconjunto de V(S) como sendo a vizinhança restrita de uma solução S.

2.3.2.2 *Simulated Annealing (SA)*

SA é um algoritmo não determinístico, derivado dos processos físicos de resfriamento de fluidos, onde os parâmetros de otimização da função objetivo correspondem aos átomos do fluido e o valor da função objetivo corresponde ao nível de energia do mesmo. Assim sendo, um procedimento para resfriar um fluido a um baixo estado de energia servirá como ferramenta para encontrar uma solução de boa qualidade para um problema de Otimização [KGV83].

O SA pode ser visto, também, como um aprimoramento do algoritmo de Busca Local, da seguinte maneira: ao invés de escolher a melhor solução de uma vizinhança V(S) da solução corrente S, escolhe-se uma solução S' aleatoriamente de V(S); se $F(S') - F(S)$ for menor que zero (ou seja, S' é melhor que S), então S' se torna a solução corrente, caso contrário S' se torna a solução corrente conforme uma certa probabilidade. Esta probabilidade é dada pela distribuição de Boltzmann [LA87]:

$$exp\left[-\frac{1}{T}(F(S') - F(S))\right] \qquad \text{(Eq 2.1)}$$

em que T é denominado "temperatura" devido à analogia com o processo físico. A temperatura deve decrescer ao longo do tempo, e uma das maneiras mais usadas é o decréscimo geométrico. O algoritmo começa numa temperatura T_0, permanece por L iterações sobre uma mesma temperatura; em seguida, a temperatura é decrescida por um fator $\alpha < 1$ e $T_1 = \alpha T_0$. Após L iterações na temperatura T_1, a temperatura é decrescida para $T_2 = \alpha T_1$, e assim sucessivamente. Na verdade, o algoritmo original do *Simulated Annealing* permanece numa dada temperatura até que o equilíbrio térmico seja alcançado. O algoritmo pára, após um certo número de passos, sem que haja alterações significativas no valor da função objetivo.

Algoritmo *Simulated Annealing*

Selecione uma solução S como sendo a solução corrente.
$S^* = S, F^* = F(S)$
Continua = Verdade
Enquanto *Continua* **faça**:
  Repita L **vezes**
    Escolha aleatoriamente uma solução $S' \in V(S)$
    Se $F(S') - F(S)$ for negativo
      Então S' se torna a solução corrente S
        Se $F(S) < F^*$
          Então $S^* = S, F^* = F(S)$
      Senão gere um número aleatório P entre [0,1]

Se $P \leq exp\left[-\frac{1}{T}(F(S') - F(S))\right]$

Então S' se torna a solução corrente S

Faça $T = \alpha T$

Se F decresceu menos que $\varepsilon\%$ após um número fixo de L iterações sucessivas

Então Continua = Falso.

Fim

S^* armazena a melhor solução encontrada durante todo o processo. O algoritmo SA geralmente é simples de se adaptar a um problema específico, bastando, para isso, definir a vizinhança, a temperatura inicial, α, L e ε.

2.3.2.3 Tabu Search (TS)

TS também é uma extensão do método descendente, em que soluções ruins são aceitas como "pontes" para novas soluções, preferencialmente melhores. Esse método utiliza uma memória, denominada lista Tabu, para armazenar a "história" da evolução das soluções.

No algoritmo do TS, a melhor solução $S' \in V(S)$ é escolhida independentemente de ser melhor ou não que S. Dizemos, nesse caso, que o algoritmo se moveu de S para S'. Porém, ao atingir um mínimo local, essa decisão de mover sempre para o melhor vizinho pode criar uma ciclagem entre soluções. Para evitar que isso ocorra, mantém-se uma lista Tabu que contém um número fixo dos últimos movimentos efetuados. Sendo assim, só é permitido mover-se nas direções não presentes na lista Tabu. Geralmente, a lista Tabu é atualizada pela política FIFO[1]. Em verdade, a lista Tabu não armazena a seqüência de soluções obtidas num movimento, mas sim alguma característica desse movimento que seja rápida de ser verificada e conveniente para armazenar. O TS pode ser descrito de forma simplificada como segue:

Algoritmo Tabu Search

Selecione uma solução S como sendo a solução corrente.
$S^* = S, F^* = F(S)$

Repita por um número fixo de iterações
  Encontre uma solução $S' \in V(S)$ tal que $F(S') - F(S)$ seja minimal
  Se $F(S') - F(S)$ for negativo e o movimento não pertencer à lista Tabu
    Então S' se torna a solução corrente S e atualize a lista Tabu
      Se $F(S) < F^*$
        Então $S^* = S, F^* = F(S)$
    Senão a segunda melhor solução de $V(S)$ torna-se a solução corrente
Fim

1. Do inglês, *First In First Out*.

Outras características podem ser associadas ao algoritmo TS, como por exemplo, critério de aspiração, listas de candidatos e estratégias de intensificação e diversificação, [Glo89b, Lag95].

2.3.2.4 *Algoritmos Genéticos*

Algoritmos Genéticos são algoritmos de busca baseados no mecanismo natural de reprodução e seleção, em que uma população de soluções é melhorada após sucessivas gerações (iterações) [Gol89].

Para utilizar um Algoritmo Genético, é necessário, primeiramente, codificar as soluções do problema numa estrutura denominada cromossomos. Os cromossomos são constituídos de genes. Os valores que os genes podem assumir são denominados alenes. Geralmente, genes são codificados como binários, ou seja, eles podem assumir apenas os valores 0 e 1. A posição de um gene no cromossomo é denominada locus. Portanto, codificar um problema consiste em representar todas as suas variáveis na forma binária e colocá-las juntas num único cromossomo.

Cada cromossomo possui um valor associado que representa o nível de qualidade da solução. Este valor é gerado por uma função que mede a aptidão do cromossomo.

Os operadores genéticos comumente utilizados sobre os cromossomos são Recombinação e Mutação. O primeiro consiste em recombinar partes de dois cromossomos para criar um terceiro. O segundo seleciona aleatoriamente um gene no cromossomo e altera seu valor.

A cada iteração a população de cromossomos é total ou parcialmente renovada, conforme os operadores adotados. Os novos elementos são avaliados pela função de aptidão, e os menos aptos possuem maior probabilidade de serem eliminados. A idéia fundamental é garantir a *sobrevivência* dos cromossomos (soluções) mais adequados. Simplificadamente, um algoritmo Genético pode ser assim descrito:

Algoritmo Genético

Inicie a população de cromossomos (soluções).
Avalie os cromossos da população.

Enquanto o critério de convergência não for atingido **faça**:
  Selecione cromossomos da população para Recombinação ou Mutação
  Elimine com maior probabilidade os cromossomos de pouca aptidão para abrir espaço aos novos.
  Avalie os cromossomos produzidos através da função de aptidão.
Fim

Observe que, para um mesmo problema, podem existir diversas representações para uma única solução. Num Algoritmo Genético, é preciso definir quais os operadores a serem utilizados. Muitas vezes, esses operadores não utilizam nenhuma informação

específica do problema, e um operador que seja bom para um problema pode não o ser para outro. Esse é, geralmente, o caso de operadores do tipo Mutação, que podem criar cromossomos correspondentes a soluções infactívies.

2.4 *Times Assíncronos - um novo método aproximado*

Recentemente, Pedro S. de Souza [TS90, ST91, Sou93] introduziu uma nova técnica de resolução de problemas denominada Times Assíncronos (ou do inglês, *Asynchronous Teams - A-Teams*). Essa técnica baseia-se na utilização simultânea de diversos algoritmos heurísticos destinados a resolver um certo problema. Tais algoritmos interagem e ajudam-se reciprocamente (sobre um conjunto de soluções válidas ou inválidas) com o objetivo de propiciarem soluções ótimas ou quase ótimas. Desta forma, os *A-Teams* geram soluções que não seriam obtidas pelos algoritmos quando executados isoladamente. Essa nova técnica é abordada no capítulo 3.

2.5 *Notas Bibliográficas*

Existe uma grande variedade de publicações e trabalhos versando sobre Métodos Exatos. A título de ilustração, podem-se ressaltar os seguintes trabalhos: [PS82, BT85, NW88] para *Branch-and-Bound*; [BD62, KH67, AHU72, KK88] para Programação Dinâmica e finalmente [DFJ54, GP79, GP79b, GP85, PR91, JRR94] para *Branch-and-Cut*.

O algoritmo de SA surgiu através do trabalho de Metropolis *et al.* [MRRTT53], sendo adaptado para problemas de otimização por Kirkpatrick *et al.* [KGV83] em 1983. SA tem sido utilizado em diferentes problemas de Otimização Combinatória, como pode ser verificado, por exemplo, em [KGV83, Rut89]. Para outras aplicações de SA veja [BR84] e sobre as propriedades de convergência [FS88, RR92].

Tabu Search foi desenvolvido independentemente por Glover [Glo89] e Hansen *et al.* [HJ87]. Maiores detalhes e aplicações podem ser encontrados em [Glo86, Glo89, Glo89b, Glo90, Glo90b, HW87, WH89, WH89b, Lag95].

Os primeiros trabalhos relacionados com algoritmos Genéticos surgiram em meados de 1950, quando vários pesquisadores começaram a simular sistemas biológicos [Gol89]. Atualmente, algoritmos genéticos têm sido aplicados com sucesso em diversas áreas, tais como: resolução de equações algébricas não lineares [ST91], problemas de escalonamento [CS89], reconhecimento de padrões [Cal91], problemas combinatórios [Muh89], dentre outros.

Capítulo 3

Times Assíncronos

Este capítulo descreve uma nova técnica aproximada de resolução de problemas. Essa técnica, denominada Times Assíncronos, é baseada na utilização simultânea de diversos algoritmos heurísticos, que cooperam entre si de maneira sinérgica, encontrando soluções ótimas ou quase ótimas, as quais não seriam encontradas isoladamente pelos algoritmos.

A primeira parte deste capítulo descreve um modelo organizacional e a história do desenvolvimento dos Times Assíncronos. São abordados, em seguida, as definições e suas principais características. Finalmente, são apresentados alguns exemplos de Times Assíncronos.

3.1 Introdução

Após o advento da teoria que trata os problemas NP-Completos e NP-Difíceis, muitos problemas que pertencem a essas classes foram contemplados por algoritmos heurísticos que se propõem a resolvê-los aproximadamente, uma vez que a existência de algoritmos exatos polinomiais é improvável [GJ79]. Para ratificar tal fato, basta verificar o número de trabalhos publicados onde os problemas Combinatórios são resolvidos heuristicamente. Como exemplo, considere o Problema do Caixeiro Viajante [LLKS85], o problema do Roteamento de Veículos [Lap92], o problema de escalonamento de tarefas *Flow Shop Problem* e *Job Shop Problem* [ML93], entre outros.

Uma vez identificado o problema que se deseja resolver e as respectivas heurísticas disponíveis, qual delas é a mais adequada? Essa pergunta pode levar a uma análise exaustiva de todas elas, tomando como parâmetro de comparação, a complexidade de implementação, a qualidade das soluções produzidas, a eficiência computacional, a robustez, e muitas outras características [ZE81, BM81]. Um estudo detalhado pode respaldar a escolha de uma delas, porém, ao utilizar apenas uma, desprezam-se as eventuais qualidades das demais. Não haveria, então, uma maneira de organizar um "time"de heurísticas de modo a tirar proveito das virtudes de cada uma e obter soluções melhores que as obtidas isoladamente? Já em 1975, Weiner sugeria a utilização de uma heurística de melhoria[1] sobre a solução gerada por uma heurística de construção[2] [Wei75].

Como foi observado por Weiner, ao dispormos de uma coleção de heurísticas que se propõem a resolver um problema, pode-se combiná-las com o objetivo de obter soluções melhores que as geradas por elas isoladamente. Por exemplo: Sejam **A**, **B**, **C**, e **D** quatro heurísticas determinísticas distintas de um dado problema **P** tal que **A**, **B**, e **C** são heurísticas de melhoria e **D** uma heurística de construção. Pode-se combiná-las basicamente de seis maneiras distintas, como ilustrado na figura 3.1.

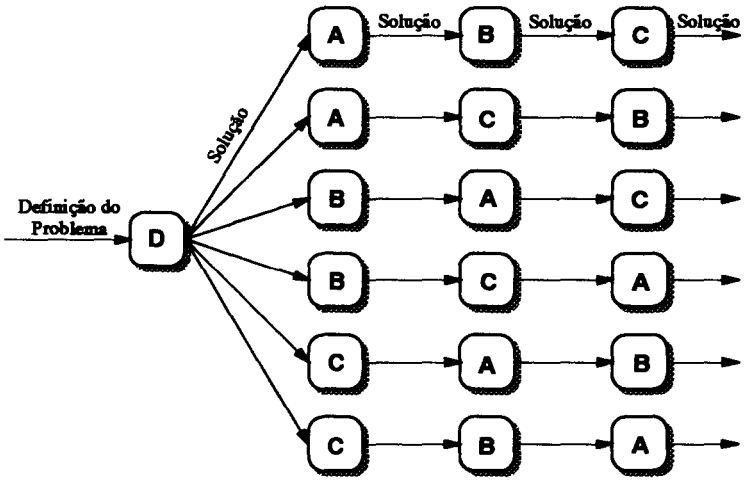
Qual dessas seqüências produz os melhores resultados? Seja qual for a resposta, têm-se dois inconvenientes principais:

1. A seqüência escolhida pode gerar soluções de boa qualidade apenas para um subconjunto das instâncias de **P**;
2. Mesmo não se levando em consideração o item anterior, o número de combinações possíveis é infinito, tornando proibitiva a tentativa de analisar qual seqüência seria a "melhor". Na verdade, a figura 3.1 ilustra apenas as seqüências de três heurísticas distintas. As combinações possíveis são todas aquelas em que um mesmo algoritmo não aparece consecutivamente na seqüência, independentemente do seu tamanho.

1. Heurísticas que partem de uma solução válida e a modificam à procura de soluções melhores.

2. Heurísticas que a partir da descrição da instância do problema constroem uma solução elemento a elemento.

Figura 3.1 Seqüências possíveis para as heurísticas **A**, **B** e **C** de melhoria e **D** de construção.



Dadas tais dificuldades, seria interessante a existência de uma *organização de software* que explorasse a interação entre os algoritmos heurísticos inteligentemente. A próxima seção descreve alguns modelos organizacionais, sendo um deles correspondente a uma nova técnica para agrupar heurísticas cooperativamente: o modelo dos Times Assíncronos. Em seguida apresenta-se o histórico, definições e características dos Times Assíncronos, bem como dois exemplos de tais organizações: um natural e um artificial.

3.2 *Modelos Organizacionais*

O conceito de Times Assíncronos é um passo inovador no domínio da teoria Organizacional[1], onde pouco esforço tem sido dispendido com sucesso na aplicação de tais modelos às organizações de software. Antes que seja apresentado o modelo organizacional que represente os *A-Teams*[2], as sub-seções que seguem exemplificam alguns outros modelos encontrados na literatura [Sou93].

1. Por organização entende-se um grupo de indivíduos ou elementos que cooperam entre si na obtenção de um objetivo comum.

2. Do inglês, *Asynchronous Teams* ou *A-Teams*.

3.2.1 O cérebro humano e os neurônios

O cérebro humano é formado por cinco partes: córtex, cerebelo, mesencéfalo, bulbo e medula. Juntas, essas partes controlam as funções motora, visual, auditiva e de raciocínio. Em média, o cérebro humano contém 10^{10} neurônios de diversos tipos, cada um capaz de receber milhares de informações (através de sinais elétricos) [Lau92]. Normalmente, os neurônios possuem os dendritos, que recebem as informações de entrada, e o axônio, que transmite as informações de saída [McR86].

Embora seja grande a complexidade das redes de neurônios e das tarefas desempenhadas pelo cérebro, não existe nenhum controle central sobre os neurônios.

Com o intuito de resolver problemas complexos, a organização do cérebro tem sido objeto de estudo de vários pesquisadores e aproximadamente reproduzida em modelos matemáticos. Redes neurais artificiais, por exemplo, têm sido aplicadas com sucesso em problemas de reconhecimento de imagens [IM92].

3.2.2 Insetos

As organizações das colônias de insetos possuem duas características fundamentais:

1. existência de classes sociais, e
2. divisão do trabalho

Entre as milhares de espécies de colônias de insetos, existem três características comuns a todas elas, como foi observado por Oster *et al.* [OW78]:

1. cooperação no cuidado dos elementos novatos da sociedade;
2. existência concomitante de pelo menos duas gerações capazes de ajudar no trabalho da colônia;
3. atribuição da reprodução a poucos indivíduos e o restante do trabalho aos demais elementos da colônia. A divisão do trabalho é uma qualidade essencial à existência das colônias.

A divisão do trabalho restringe o repertório de comportamento e aumenta a especialização dos indivíduos da colônia. Além da especialização, a capacidade de executar tarefas paralelamente torna uma colônia de vários indivíduos mais eficiente que um único indivíduo unipotente. Uma vantagen da existência de vários indivíduos na colônia, capazes de executar as mesmas tarefas, é que o erro de um deles pode ser reparado com sucesso pelos demais.

3.2.3 Organizações Humanas

As organizações humanas podem ser entendidas como sendo um grupo de pessoas que possuem objetivos comuns e que se relacionam sob certas regras e estruturas pré definidas [Dun78].

Com o advento das grandes corporações, o estudo das organizações tornou-se importante e, desde então, alguns modelos oranizacionais humanos têm sido aplicados a outras áreas. Como exemplo, Fox [Fox79] define um paralelo entre alguns modelos básicos organizacionais humanos e de softwares:

Uma única pessoa

Uma única pessoa executa todas as tarefas necessárias para se alcançarem os objetivos da organização. Equivale a uma organização de software onde um único programa é executado num único processador.

Grupo de pessoas

Um grupo de pessoas executam as tarefas necessárias para se alcançarem os objetivos, que tendem a ser mais complexos e trabalhosos. Os indivíduos da organização compartilham todas as informações e a cada um é associada uma tarefa. As decisões tomadas são aquelas que satisfazem a todos os elementos do grupo. Uma organização de software que represente esse modelo é aquela em que os módulos de um mesmo programa estão alocados em diferentes processadores, mas todos os módulos possuem conhecimento total do processamento.

Hierarquias simples

Em grupos grandes, a tomada de decisões que agrade a todos os indivíduos é praticamente impossível. Sendo assim, uma pessoa é designada para a tarefa de tomar decisões e apenas ela possui acesso a todas as informações da organização. Numa organização de software, este modelo equivale a vários módulos independentes controlados por um único módulo, sendo cada um executado num processador.

Hierarquia Uniforme

Quando o número de pessoas de uma organização cresce muito, uma única pessoa não é suficiente para tomar todas as decisões. Nessas organizações são criados níveis de gerenciamento denominados hierarquias uniformes. Cada nível da organização decide apenas sobre o que é habilitado, passando para os níveis mais altos as decisões que não são capazes de tomar. Essas organizações são representadas por softwares que possuem uma estrutura em árvore em que, geralmente, o fluxo de controle é direcionalmente contrário ao fluxo de informações.

Hierarquia de Multiplas Divisões

Em algumas organizações, devido ao grande número de indivíduos e complexidade dos objetivos, torna-se inapropriadas até mesmo uma única hierarquia uniforme. Nesse caso, várias hierarquias uniformes são criadas (divisões), conforme seus respectivos objetivos. Comum a todas elas, existe apenas um grupo de tomadores de decisões que ditam seus objetivos gerais. Uma organização de software que representa esse modelo é aquela de um grande sistema computacional distribuído, sendo cada módulo responsável por uma função e capaz de trocar informações com outros módulos, mesmo que distantes.

Fox [Fox79] sugere que a complexidade de uma organização de software é proporcional ao tamanho e complexidade dos problemas que se propõe a resolver, reproduzindo o que acontece com as grandes organizações humanas.

Embora os *A-Teams* sejam adequados à resolução de problemas grandes e complexos [Che92, Mur92, Sou93, PS94], este trabalho ressaltará que o modelo organizacional que representa os *A-Teams* não se encaixam perfeitamente nos modelos propostos por Fox.

3.2.4 *Um modelo organizacional*

Talukdar e Souza [TS90, TS92, Tal93] definem uma organização como sendo um esquema através do qual agentes (neurônios, insetos e homens) se agrupam para formar super-agentes (sistema nervoso, colônias de insetos e corporações). Noutras palavras, uma organização é um conjunto de políticas pelo qual agentes são agregados para formarem super-agentes [TSM93].

Com o objetivo de modelar os vários tipos de interação entre os agentes, Talukdar e Souza [TS90, Tal93, TS93] definem um modelo estrutural para classificar as organizações, em particular as de softwares. Esse modelo considera que:

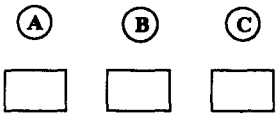
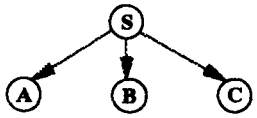
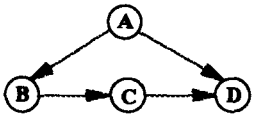
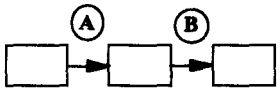
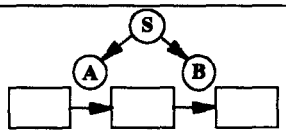
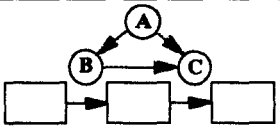
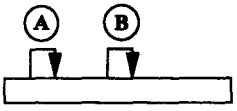
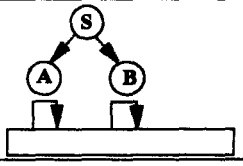
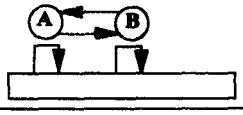
1. apenas super-agentes cujos sistemas de comunicação possam ser modelados por um conjunto finito de memórias compartilhadas serão considerados (por sistemas de comunicação entende-se a maneira pela qual os agentes se comunicam). Os agentes podem se comunicar através de memórias comparilhadas ou através de trocas de mensagens, uma vez que estas duas maneiras são equivalentes;
2. os estados internos de um agente não são de nenhum interesse, pois agentes podem ser recursivamente decompostos em sub-agentes e sistemas de comunicação, até que o nível de detalhamento desejado seja alcançado.

Nesse contexto, super-agentes são considerados como sendo uma coleção de memórias que são compartilhadas entre seus agentes. Um agente pode ler e escrever nas memórias compartilhadas, transformando um objeto em outro.

A estrutura de um super-agente é representada por dois fluxos: de dados e de controle. O primeiro determina quais memórias cada agente pode acessar. O último define qual dado (objeto) nas memórias os agentes podem ler (políticas de seleção), quando o agente pode processar seus dados (escalonamento dos agentes) e quais recursos serão alocados.

Dado estes dois tipos de fluxos, Talukdar e Souza definem uma taxonomia para apresentar as atuais organizações de softwares, ilustradas na figura 3.2. A primeira coluna representa a tupla *fluxo de dados* e *fluxo de controle*, representados nos gráficos da segunda coluna por setas contínuas e pontilhadas, respectivamente. Ambos os tipos de fluxos podem ser *nulos*, *cíclicos* ou *acíclicos*. A última coluna descreve um exemplo de uma organização de software correspondente a cada uma das estruturas.

Figura 3.2 Taxonomia organizacional. A primeira coluna representa o tipo de fluxo de dados (FD) e controle (FC), representado por [FD, FC], em que FD e FC assumem os valores Nulo (N), Cíclico (C) e Acíclico (A). A segunda coluna exemplifica um elemento típico desta estrutura organizacional. A última coluna descreve, sucintamente, a respectiva organização de software. Os grafos com retângulos e arcos contínuos, círculos e arcos pontilhados indicam, respectivamente, fluxo de dado e de controle.

[FD, FC]	Membro Típico	Apl. Software
[N, N]		*Bibliotecas de programas e dados.*
[N, A]		*Agentes que compartilham computadores, mas não dados. O fluxo de controle é para a alocação de processador.*
[N, C]		*Não usado comumente.*
[A, N]		*Um conjunto de agentes autônomos ligados em série.*
[A, A]		*Configuração mais utilizada.*
[A, C]		*Não usado comumente.*
[C, N]		*A-Teams.*
[C, A]		*Blackboards.*
[C, C]		*Não usado comumente.*

O restante deste capítulo descreve as organizações de softwares que possuem fluxo de dados cíclicos e fluxo de controle nulo, representados na terceira linha de baixo para cima na figura 3.2 e denominados *A-Teams*.

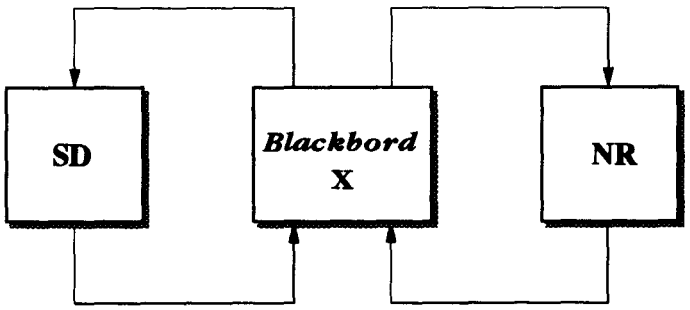
3.3 Histórico

Embora os *A-Teams* tenham surgido recentemente, a idéia de resolver um problema através da interação de agentes não é novidade, pelo menos na área de Resolução Distribuída de Problema [Sou93].

Em 1985, Pyo [Pyo85] apresentou um trabalho que posteriormente inspirou a criação dos *A-Teams*. Pyo denominou *Team Approach* a idéia de combinar algoritmos já existentes de forma distribuída e síncrona. Para concretizar suas idéias, Pyo resolveu equações algébricas usando o conceito de *BlackBoards* [Pyo85]. O *Blackbord* servia de mediador entre dois métodos aproximados, Newton-Raphson (NR) e Steepest Descent (SD) que, dado um ponto inicial, aproximam, sucessivamente, as soluções. Inicialmente o *Blackbord* fornece um mesmo ponto de partida aos dois métodos. Cada método efetua seus processamentos independentemente um do outro e, ao término, ambos retornam seus resultados ao *Blackbord*. O *Blackbord* analisa os resultados fornecidos e então decide que o método que teve menor progresso deve ser executado novamente sobre os resultados fornecidos pelo método de maior progresso. Este "time" encontrou resultados melhores que os encontrados por ambos os métodos (NR e SD) quando executados isoladamente, mesmo quando nenhum deles era capaz de encontrar alguma solução. A figura 3.3 ilustra o procedimento utilizado por Pyo.

Neste modelo existe um controle supervisor por parte do *Blackbord*, uma vez que ele decide qual procedimento a ser executado e sobre quais dados.

Figura 3.3 Um "time" composto por um *Blackbord* e dois algoritmos que representam dois métodos distintos de resolução de equações algébricas, SD e NR.



3.3.1 O Precursor

O termo Times Assíncronos foi introduzido, em 1990, por Talukdar e Souza [TS90], através de um trabalho para resolver equações algébricas não lineares. Estes autores aprimoraram o modelo proposto por Pyo, introduzindo alguns conceitos dos *A-Teams*, que serão mencionados na próxima seção. Nesse trabalho, Talukdar e Souza combinaram o método de Newton-Raphson e algoritmos genéticos num mesmo time, porém sem a existência de um mediador. Estes métodos possuem características bastantes distintas. Os algoritmos genéticos são métodos globais de busca e foram utilizados para evitar mínimos locais. O NR, devido à sua rápida convergência, pode refinar as soluções geradas por outros algoritmos, em particular pelo algoritmo genético. Esta combinação, num pequeno tempo, obteve soluções de qualidade muito boa, quando comparadas com as produzidas pelos métodos isoladamente.

Nesse exemplo, as virtudes de cada método foram exploradas na obtenção de soluções melhores.

3.4 Definições e Características

Usando o modelo organizacional da seção 3.2 e segundo a definição de Talukdar e Souza [TS92], um *A-Team* é qualquer super-agente cujos agentes são autônomos, a comunicação entre eles é assíncrona e o fluxo de dados é cíclico. Em outras palavras, um *A-Team* é uma maneira recente de organizar diversos algoritmos (agentes), cada um trabalhando de maneira assíncrona, interativa e cíclica, sobre um conjunto de dados depositados em memórias compartilhadas.

As memórias armazenam soluções ou informações relevantes à resolução do problema e, o fato de serem compartilhadas garante que os resultados produzidos por um agente estarão disponívies aos demais. Os agentes podem ler e escrever nas memórias compartilhadas sempre que desejarem, conforme suas respectivas políticas de acesso às mesmas. Nos *A-Teams*, cada agente tem a oportunidade de escrever sua melhor solução nas memórias, que pode então ser usada como entrada por outro agente do time. Esta interação entre os agentes leva à necessidade da existência de fluxo cíclico das soluções. Estes ciclos permitirão *feedback* e a possibilidade de um agente operar sobre uma solução criada previamente por ele e modificada pelos demais. Esta cooperação entre os agentes aumenta as chances de o time gerar soluções melhores, que não seriam geradas pelos agentes isoladamente.

Um agente consiste num algoritmo que se propõe a resolver o problema (ou parte dele) e um protocolo de comunicação com a memória. Geralmente, devido ao porte e complexidade dos problemas tratados, os agentes despendem muito mais tempo processando dados que efetuando comunicações com as memórias. Esta característica, aliada à autonomia dos agentes e ao assincronismo das comunicações, torna os *A-Teams* adequados ao processamento paralelo e distribuído.

Diferentemente dos *Blackbords*, nos *A-Teams* não existe a figura de agentes supervisores. Cada agente é livre para escolher quando e qual dado processar. Conseqüentemente, agentes podem entrar e sair do time a qualquer instante. A saída de um agente, via de regra, causa apenas uma degradação suave no time, não comprometendo de todo a geração de novas soluções.

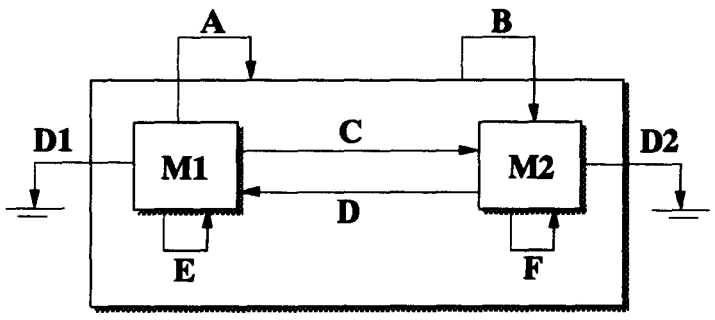
Uma vez que as memórias compartilhadas não podem incorporar soluções indefinidamente (elas são finitas), a cada uma estão associados um ou mais agentes denominados *Destruidores*. As únicas funções de um Destruidor são julgar e eliminar, baseado em uma política de destruição, que solução eliminar a fim de abrir espaço a uma nova. Essa política de destruição é, geralmente, relacionada à qualidade dos dados. Nos *A-Teams*, tão importante quanto a tarefa de produzir boas soluções é a de eliminar soluções não promissoras.

Os *A-Teams* podem ser representados graficamente através de setas, indicando os agentes, e retângulos, indicando as memórias compartilhadas. O sentido das setas mostra de onde os agentes retiram os dados e aonde os agentes depositam os seus resultados. A figura 3.4 ilustra um *A-Team* composto por oito agentes e duas memórias.

A seta que representa o agente A indica que este agente efetua suas leituras na memória **M1** e escreve seus resultados em **M1**, **M2** ou em ambas. O agente **D** lê da memória **M2** e escreve na memória **M1**, e o agente **F** lê e escreve apenas em **M2**. Os agentes **D1** e **D2** são os Destruidores de soluções das memórias **M1** e **M2**, respectivamente. embora não esteja representado na figura 3.4, uma mesma mémoria pode possuir mais de um agente Destruidor, visto que cada um deles pode operar sob políticas distintas de eliminação.

Em resumo, as propriedades de uma organização que caracterizam um *A-Team* são [TS92]:

Figura 3.4 Exemplo da representação gráfica de um *A-Team* de duas memórias e oito agentes.



Agentes Autônomos: Um agente é considerado autônomo se nenhum outro agente interfere na escolha de seus dados de entrada, quando e como processá-los. Uma vez que os agentes são autônomos, novos agentes podem entrar e sair da organização sem qualquer notificação prévia ou supervisão dos demais.

Fluxo de Dados Cíclico: O fluxo de dados cíclico permite que as mudanças realizadas por um agente sobre um dado sejam acessíveis aos demais, permitindo assim *feedback* e interação entre eles. Isto gera a possibilidade de se ter modificações e alterações nos dados de forma a criar outros novos e melhores para o problema em questão.

Comunicação Assíncrona: Agentes podem ler e escrever nas memórias sempre que lhes aprouver, livres de qualquer sincronismo, o que permite que sejam executados concorrente ou paralelamente.

Como supra mencionado, o número de agentes num *A-Team* pode variar no decorrer do tempo, visto que a inserção e remoção ocorrem independentemente do conscentimento e notificação dos demais. Esta característica torna os *A-Teams* organizações abertas, como foi observado por Talukdar e Souza [TS92].

Uma outra característica dessas organizações é que a introdução de novos agentes no time tende a "facilitar" a obtenção dos objetivos da organização. Geralmente, à medida que novos agentes são incorporados às organizações, ocorre uma melhora monotônica em algumas de suas medidas de performance. Esta característica foi intitulada, por Talukdar e Souza [TS92], *Eficiência em Escala*. Via de regra, pode-se observar que a interação entre os agentes dos *A-Teams* é sinérgica, ou seja, o resultado obtido da interação entre eles é maior que a soma dos resultados produzidos individualmente.

As próximas seções ilustram dois exemplos de *A-Teams*, um natural e um artificial.

3.5 Exemplo Natural de A-Teams

3.5.1 Algumas Colônias de Insetos

Algumas espécies de insetos formam organizações que realizam tarefas complexas tais como construção, manutenção e defesa do ninho, localização, transporte e armazenamento de alimento. Nessas organizações, qualquer elemento que possua suas caracteríticas (aparência, odor, etc.) pode fazer parte do grupo. Em verdade, a inteligência dessas organizações, como um todo, parece ser muito maior que a soma de suas partes. Os insetos possuem capacidades muito limitadas de racioncínio e memória, não sendo capazes de efetuar generalizações. Mesmo assim, essas organizações são Eficientes em Escala e se comportam de maneira Sinérgica [TS92, Tal93].

Nessas organizações, embora exista a presença da rainha, não existe supervisão por parte de nenhum membro. A única função da rainha é a reprodução, e por isso não lidera nem ordena tarefas aos demais membros da organização.

Os membros são livres para tramitarem pela colônia quando e onde desejarem, sem qualquer sincronismo e supervisão. As comunicações, quando necessárias, são feitas diretamente (inseto a inseto) ou através de recursos locais temporários (como sinais químicos). Como observado por Wilson [Wil71], os insetos não capazes de uma visão global da colônia, nem que por uma fração de minutos. Além disso, os insetos são altamente especializados e trabalham em paralelo.

Fazendo uma analogia às organizações que caracterizam os *A-Teams*, os insetos de uma colônia correspondem aos agentes desta organização que, como mencionado, são autônomos e trabalham interativamente. Durante a migração de uma colônia, por exemplo, enquanto alguns insetos estão retirando dos ninhos ovos, larvas, etc , outros estão continuamente retornando os mesmos objetos para seus locais costumeiros [Tal93]. Como pode ser observado a partir da comunicação dos insetos, o fluxo de dados nestas organizações é cíclico. Os insetos (agentes) se comunicam diretamente através de troca de mensagens ou de sinais químicos depositados em regiões limitadas do espaço, que neste caso funcionam como memórias compartilhadas. Essas organizações são abertas, visto que qualquer inseto que possua as mesmas características podem fazer parte da organização.

3.6 Exemplo Artificial de A-Teams

3.6.1 A-Teams para o Problema do Caixeiro Viajante

O problema do Caixeiro Viajante tem sido um dos problemas clássicos da literatura mais estudados devido a sua simplicidade de formulação, dificuldade em se obterem as soluções ótimas e inúmeras aplicações práticas.

O problema do Caixeiro Viajante consiste em encontrar o menor circuito Hamiltoniano, dado um conjunto de cidades e as distâncias entre as mesmas [LLKS85]. Este problema é NP-Difícil e sua resolução para grandes instâncias só se torna viável através do uso de métodos aproximados, em particular as heurísticas. As heurísticas, embora não garantam soluções ótimas, possuem, geralmente, complexidade polinomial e há dezenas de heurísticas disponíveis na literatura [LLKS85].

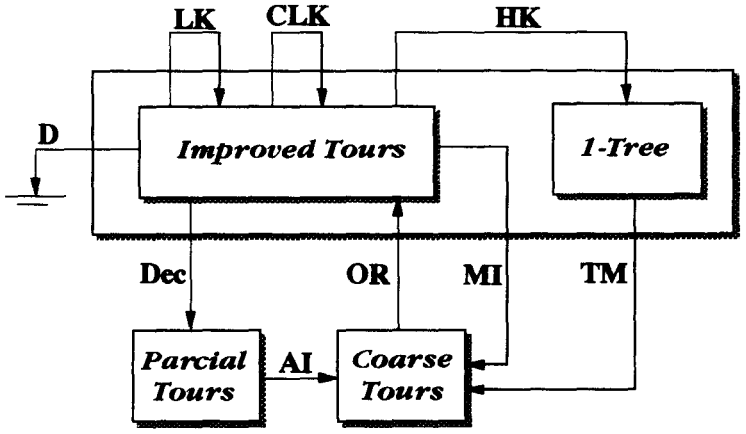
Souza [Sou93] construiu um *A-Team* para o problema do Caixeiro Viajante usando algumas heurísticas da literatura, dentre elas:

- *Arbitrary Insertion* (AI)
- *OR algorithm* (OR)
- *Lin-Kernighan* (CLK e LK)
- *Held-Karp algorithm* (HK)

Além desses algoritmos, Souza criou dois novos algoritmos, Mixer(M) e Tree-Mixer (TM), um agente Destruidor (D), um agente de Desconstrução (Dec) e usou quatro memórias compartilhadas para armazenar tipos distintos de dados. A representação gráfica do *A-Team* proposto por Souza está ilustrada na figura 3.5. A memória *Improved Tours* armazena soluções válidas, enquanto que a memória denominada *1-Tree* armazena soluções infactíveis (neste caso, essas soluções correspondem a Limites Inferiores para o problema). A memória *Partial Tours* armazena soluções incompletas (nem todas as cidades foram visitadas) produzidas pelo agente Dec. O agente Dec retira duas soluções completas da memória *Improved Tours,* verifica o que essas soluções possuem em comum e cria uma nova solução parcial apenas com as coincidências. O agente MI cria uma nova solução completa utilizando apenas as arestas presentes em duas soluções quaisquer da memória *Improved Tours*. Essas soluções incompletas são posteriormente concluídas e armazenadas na memória *Coarse Tours* e, após serem aprimoradas pela heurística de melhoria OR, são depositadas na memória *Improved Tours*. O algoritmo HK constrói uma solução inválida, denominada *1-Tree*, que serve de Limite Inferior para o problema. O agente TM é similar ao agente MI, exceto pelo fato de que TM retira uma solução de *Improved Tours* e uma solução infactível da memória *1-Tree*.

Este *A-Team* mostrou Eficiência em Escala e mostrou, ainda, que estes algoritmos juntos são capazes de encontrar soluções ótimas para todas as instâncias testadas, enquanto que os mesmos algoritmos executados isoladamente só foram capazes de encontrar soluções ótimas para a menor das instâncias.

Figura 3.5 *A-Team* proposto por Souza para o problema do Caixeiro Viajante.



3.7 Sumário

Este capítulo abordou uma nova técnica de resolução aproximada de problemas, os *A-Teams*. Primeiramente apresentou-se uma taxonomia na qual as organizações podem ser classificadas, particularmente as de software. Em seguida o histórico do surgimento dos *A-Teams*, bem como suas respectivas definições e principais características. Finalmente, foram descritos dois exemplos de *A-Teams*, um natural e um artificial.

3.8 Notas Bibliográficas

Por se tratar de técnica recente, pouco pode ser encontrado na literatura. O texto introdutório desta técnica pode ser encontrado em [TS90, ST91].

Com respeito a outros exemplos naturais de *A-Teams*, podem-se citar os trabalhos de Talukdar, Souza e Murthy [Tal93, TS92, Sou93, Mur92].

Alguns trabalhos têm sido publicados sobre a aplicação de *A-Teams* a problemas reais complexos e de grande porte. O problema do Caixeiro Viajante, como foi reportado por Souza [Sou93], problemas de Design por Murthy [Mur92], problemas de diagnóstico de falhas em redes de transmissão por Chen [Che92], problemas de escalonamento de tarefas *Job Shop Problem* por Cavalcante *et al.* [CS94] e *Flow Shop Problem* por Peixoto *et al* [PS94].

Capítulo 4

Metodologia de Especificação de Times Assíncronos

O sucesso obtido por *A-Teams* na resolução de problemas de grande porte [Che92, Mur92, Sou93] e suas características (descritas no capítulo anterior) justificam a conjectura de que *A-Teams* são úteis na resolução de um amplo espectro de problemas de Otimização Combinatória.

Este capítulo apresenta uma Metodologia de Especificação de Times Assíncronos para Problemas de Otimização Combinatória, cujo objetivo é facilitar e agilizar a especificação de *A-Teams*. Primeiramente, é necessário especificar o problema que se deseja resolver por *A-Teams*, o formato das soluções e o padrão de qualidade desejado. Em seguida, os problemas são classificados em relação aos diversos algoritmos disponíveis, sugerindo, assim, a adequabilidade ou não do uso de *A-Teams*. Os demais itens da metodologia consistem em: classificação dos algoritmos disponíveis e a sugestão de uma estrutura geral para agrupá-los, especificação das memórias, escolha e criação de algoritmos para compor Agentes, políticas de destruição, implementação e, por fim, sugestões de modificações.

4.1 Introdução

Embora a idéia que fundamente os *A-Teams* seja simples, pergunta-se: Quais as características de um problema que indicam a adequabilidade ou não do uso de *A-Teams*? Como conceber e criar um *A-Team*? Haveria estruturas prévias de *A-Teams* para algumas classes de problemas? Quais seriam essas classes e as respectivas estruturas?

O presente capítulo descreve uma Metodologia para a resolução aproximada de Problemas de Otimização Combinatória, particularmente através da especificação e construção de *A-Teams*. As seções que seguem descrevem os passos da metodologia proposta.

4.2 Especificação do Problema

A primeira tarefa na resolução de qualquer problema é sua correta identificação e especificação. Todas as variáveis, seus domínios e restrições devem ser estabelecidos nesta fase.

4.3 Definição da Representação e Padrão de Qualidade das Soluções

A eficiência dos métodos aproximados (inclusive os *A-Teams*) depende de vários fatores, dentre eles, a exploração dos dados do problema e da estrutura das soluções desejadas.

Uma vez que os *A-Teams* operam sobre soluções depositadas em memórias, é preciso definir claramente qual é a estrutura de uma solução válida. Essa estrutura não é única e depende do problema que está sendo resolvido. Como exemplo, considere-se o Problema do Caixeiro Viajante.

O *Problema do Caixeiro Viajante* (PCV) é definido como segue. Um Caixeiro Viajante deseja visitar todo um conjunto de n cidades exatamente uma única vez, terminando a visita na mesma cidade em que começou. O objetivo é encontrar o caminho que minimize a distância total percorrida.

O PCV pode ser representado como um problema de programação inteira, como foi formalizado por Dantzig, Fulkerson e Johnson [DFJ54]. As variáveis x_{ij} indicam quando a cidade j é visitada após a cidade i ($x_{ij} = 1$), ou não ($x_{ij} = 0$). A distância da cidade i para a cidade j é dada por c_{ij}. A formulação correspondente é:

$$\max \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \qquad \text{(Eq 4.1)}$$

$$\text{sujeito a} \quad \sum_{i=1}^{n} x_{ij} = 1, j = 1, \ldots, n \qquad \text{(Eq 4.2)}$$

$$\sum_{j=1}^{n} x_{ij} = 1, i = 1, \ldots, n \qquad \text{(Eq 4.3)}$$

$$\sum_{i\in S}\sum_{j\in S} x_{ij} \leq |S| - 1 \qquad \text{(Eq 4.4)}$$

onde n é o número de cidades, S é um subconjunto próprio não vazio de $\{1, 2, ..., n\}$ e $|S|$ denota a cardinalidade de S. Nesta representação uma solução corresponde a uma matriz binária nxn, onde o elemento ij corresponde à variável x_{ij}.

Uma outra maneira de representar uma solução válida para o PCV consiste numa seqüência de n números, onde o primeiro número corresponde à cidade origem (primeira cidade visitada), o segundo número corresponde à segunda cidade visitada, e assim sucessivamente. O último número indica a última cidade visitada antes de o Caixeiro Viajante retornar à cidade de origem. Uma solução nessa representação corresponde a um vetor de n posições, onde a posição i indica a i-ésima cidade visitada, $1 \leq i \leq n$.

Embora esta segunda representação seja mais intelegível e compacta sob o prisma humano, a escolha da representação adequada ao *A-Team* depende também dos agentes que farão uso das soluções. Se todos os agentes trabalham sob uma dada representação, qualquer outra utilizada para armazenar as soluções nas memórias dos *A-Teams* pode significar a necessidade de conversões constantes, não justificando a escolha.

Quanto ao nível de qualidade desejado, embora esteja-se sempre em busca do ótimo, para muitos problemas, as soluções esperadas não são necessariamente as soluções ótimas. Por exemplo, se os dados de entrada do problema a ser resolvido são imprecisos, a busca por alguma solução ótima significa esforço desnecessário. Nesse e noutros exemplos, pode-se estabelecer que soluções que distam $x\%$ do ótimo sejam as soluções ideais, $x \geq 0$.

Em outros casos, não é possível avaliar precisamente a função objetivo. Como exemplo, temos o Problema de Desenho Automático de Grafos no Plano, quando o objetivo é facilitar a um ser humano a legibilidade da informação representada. Indiscutivel-

mente, a legibilidade (ou estética) é uma característica subjetiva, depende da informação a ser representada e do gosto particular de cada pessoa. Um usuário pode relacionar a legibilidade aos seguintes critérios:

- minimização do número de cruzamentos entre arestas;
- minimização do tamanho global das arestas;
- distribuição uniforme dos vértices dentro da área de desenho;

e, mesmo assim, uma solução que o agrade pode possuir um número maior de cruzamentos e tamanho global das arestas que uma outra que seja "quantitativamente" melhor.

O conceito de soluções ótimas não se aplica a outros tipos de problemas. Pode ser o caso, por exemplo, de problemas que buscam minimizar vários objetivos simultaneamente - *Problemas de Funções Multi-Objetivos* [HPY80, SGD86]. Em tais problemas, os objetivos são geralmente conflitantes entre si, de maneira que para satisfazer as restrições, a minimização de um deles implica acréscimo de um outro. Os problemas de funções multi-objetivos, portanto, consistem em encontrar as soluções que satisfaçam as restrições e *minimizem todos os objetivos* da melhor maneira possível. Essas soluções não são "dominadas" por nenhuma outra, ou seja, não existe uma outra solução que seja melhor em todos os objetivos (podem ser melhores em alguns objetivos, porém são, necessariamente, piores em outros). O conjunto dessas "soluções boas" que não são dominadas por nenhuma outra é conhecido como conjunto de soluções pertencentes ao *Pareto Ótimo* [PS88].

A abordagem normalmente utilizada para resolver os problemas de funções multi-objetivos é a redução de todas as funções que se deseja minimizar a uma única função, através da atribuição de pesos a cada um dos objetivos. Mesmo que todos os pesos sejam iguais, isto não significa que todos os objetivos serão tratados igualmente, uma vez que as restrições sobre cada um dos objetivos podem ser distintas. Portanto, encontrar uma única função objetivo para os problemas multi-objetivos não é uma tarefa fácil. Murthy [Mur92], usando a abordagem de *A-Teams*, resolveu o problema de projeto de manipuladores mecânicos a partir de especificação de tarefas, minimizando certos objetivos (peso, deflexão, etc), mas respeitando todas as restrições do problema, onde o objetivo é encontrar boas soluções factíveis e não uma solução ótima.

Para os problemas onde não existe o conceito de solução ótima, o ideal a um tomador de decisões é a possibilidade de escolha, dentre um conjunto de soluções boas, a solução mais adequada ou de melhor compromisso entre os objetivos envolvidos. Vale ressaltar que os *A-Teams* proporcionam ao usuário ou tomador de decisões a oportunidade de escolha, uma vez que operam sobre conjuntos de soluções depositadas em memórias compartilhadas.

4.4 Adequabilidade de Problemas ao uso da Metodologia de Especificação de A-Teams

Nesta seção classificamos os problemas face aos algoritmos existentes que se propõem a resolvê-los. Esta classificação tem o único objetivo de delinear as características dos problemas para os quais acreditamos que *A-Teams* sejam adequados, não significando sobremodo a não existência de contra-exemplos.

4.4.1 Espaço dos Problemas

Uma vez definido o problema que se deseja resolver, a representação de uma solução válida e o padrão de qualidade mínimo associado, o passo seguinte é a coleta dos diversos algoritmos disponíveis na literatura para resolvê-lo. Ao identificar tais algoritmos e suas respectivas características, podem-se classificar os problemas em três classes gerais [Sou93]: a classe dos problemas que possuem ao menos um *Algoritmo Adequado*, a classe dos problemas que possuem apenas *Algoritmos Inadequados* e classe dos problemas que possuem apenas *Algoritmos Inaptos*, como mostrado na figura 4.1.

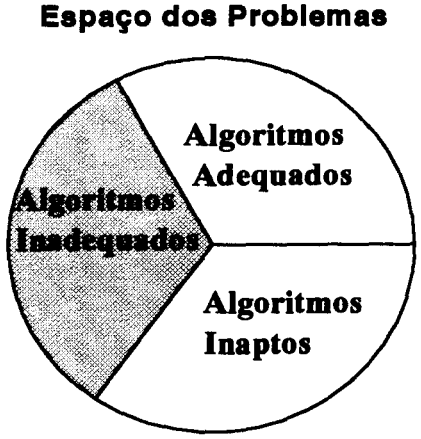
Essa classificação não se refere à complexidade dos algoritmos, mas sim à habilidade dos algoritmos em prover as soluções desejadas, respeitando todas as restrições do problema, inclusive restrições em relação ao tempo disponível para resolvê-lo.

Os *Algoritmos Adequados* correspondem aos algoritmos que encontram as soluções desejadas no tempo disponível. Exemplos de Algoritmos Adequados:

- QuickSort de ordenação de vetores [AHU83];
- Simplex para resolução de problemas de programação linear [BSJ92];
- Branch&Bound para resolução de problemas de programação inteira [PR88];

considerando que todos tenham tempo suficiente para encontrar as soluções desejadas.

Figura 4.1 Classificação dos problemas segundo os algoritmos disponíveis para resolvê-los



Os *Algoritmos Inadequados* são aqueles que, na maioria das vezes, não satisfazem o padrão de qualidade desejado (gerando soluções aproximadas) ou violam sensivelmente as restrições do problema. Este é geralmente o caso de heurísticas (que não conseguem atingir o padrão de qualidade desejado) e algoritmos de Relaxação (que geram soluções infactíveis que podem ser facilmente ajustadas a soluções factíveis). Como exemplo de Algoritmos Inadequados:

- Heurísticas de construção de forma geral;
- Algoritmos que desconsideram alguma restrição do problema com o intuito de facilitar a busca de soluções melhores;

considerando que nos dois casos eles encontram as soluções citadas no tempo disponível.

A classe dos problemas de *Algoritmos Inaptos* corresponde aos problemas para os quais os algoritmos disponíveis são capazes de encontrar apenas soluções profundamente infactíveis. Esses problemas são muito difíceis ou possuem restrições severas, inclusive em relação ao tempo. Um exemplo é o problema de resolver em poucos minutos, um conjunto de equações diferenciais não lineares de milhões de variáveis em aplicações de previsão do tempo, onde as variáveis são modificadas rápida e periodicamente [Bar92].

Entre as três classes mencionadas, estamos interessados na classe dos problemas de Algoritmos Inadequados. Esse interesse se deve a dois fatores principais:

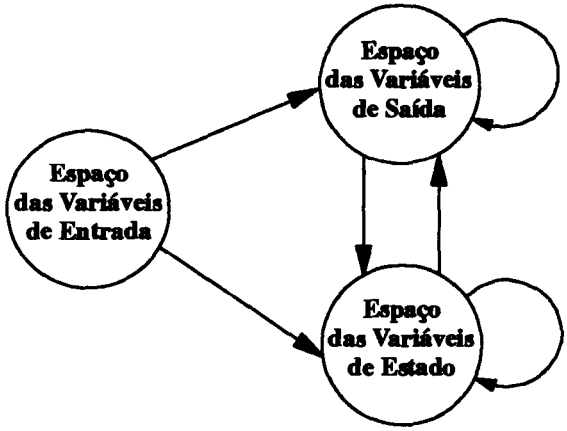
1. Muitos problemas de aplicação prática se enquadram nessa classe, em particular a grande maioria dos problemas de Otimização Combinatória. Isso decorre do tempo limitado (geralmente pequeno) de que dispomos para resolvê-los;
2. Existe um conhecimento disponível sobre esses problemas, uma vez que existem algoritmos que são capazes de encontrar soluções factíveis ou que sejam "ligeiramente" infactíveis.

4.4.2 Espaços de Variáveis

Um problema pode ser decomposto em três espaços de variáveis [Sou93] (veja figura 4.2):

1. **Espaço das Variáveis de Entrada**: contém todas as variáveis que definem o problema.
2. **Espaço das Variáveis de Estado**: contém todas as variáveis que são relevantes durante o processo de resolução do problema, podendo representar soluções infactíveis.
3. **Espaço das Variáveis de Saída**: contém todas as variáveis que especificam uma solução para o problema.

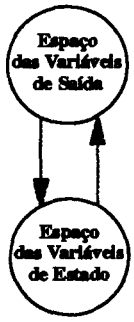
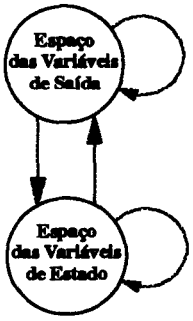
Figura 4.2 Diagrama de decomposição de problemas. Setas representam algoritmos disponíveis para resolver um problema, mapeando elementos de um espaço para outro.



Os espaços de variáveis são conectados por setas, representando os algoritmos disponíveis que efetuam o mapeamento dos elementos de um espaço para outro. O espaço de variáveis de entrada é fixo, uma vez que a definição da instância de um problema não pode ser modificada. Como exemplo de decomposição, considere o PCV. O espaço de variáveis de entrada do PCV é composto pelas cidades a serem visitadas pelo Caixeiro Viajante e a matriz de distância entre as mesmas. O espaço de variáveis de estados contém, por exemplo, as soluções parciais que, por sua vez, podem ser infactíveis (não contém todas as cidades ou possui mais de um ciclo). O espaço de variáveis de saída contém seqüências de cidades que indicam a ordem a ser visitada pelo Caixeiro Viajante.

Esta decomposição tem por objetivo atestar a existência de algoritmos que trabalham interativamente sobre as variáveis do problema. Estes algoritmos são representados por ciclos na figura 4.2 e estão descritos na tabela 4.1.

Tabela 4.1 Algoritmos que formam ciclos no Diagrama de decomposição de problemas.

Ciclos	Características
	Algoritmos que constroem diretamente uma solução válida a paritr de outra solução válida. Exemplo: heurística 2-opt para o PCV [LLKS85].
	Algoritmos que trabalham sobre dados intermediários do problema. Exemplo: algumas heurísticas para o problema de Roteamento de Veículos utilizam heurísticas para o PCV na construção de rotas parciais [GM74].
	Algoritmos que mapeam uma solução válida em outra representação equivalente no espaço das variáveis de estado, e vice-versa.
	Estes algoritmos, formados da composição de vários ciclos, decompõem uma solução válida em alguma representação intermediária, efetuam processamentos nos dados dessa representação, compõem novamente uma solução válida e, ainda, podem tentar melhorá-la. Exemplo: heurística Lin-Kernighan para o PCV [JRR94].

A existência de ciclos garante-nos a possibilidade de criar novas soluções a partir de soluções já existentes ou de suas representações equivalentes. Essa é uma característica crucial aos *A-Teams*, uma vez que os Agentes são modificadores de dados e soluções.

4.4.3 Problemas Multi-Algoritmos

Souza [Sou93] define *Problema Multi-Algoritmo* (PMA) como sendo um problema que pertença à classe dos Algoritmos Inadequados e que possua ao menos um ciclo no diagrama de decomposição de problemas. A presença de ciclos garante que um ou mais algoritmos possam ser executados interativamente.

De acordo com a definição de Souza, qualquer problema que pertença à classe dos problemas de Algoritmos Inadequados pode ser transformado em um PMA, bastando, para isso, que se criem algoritmos que formem ciclos no diagrama de decomposição de problemas (caso não os possua). Evidentemente, essa definição é abrangente o suficiente para incorporar grande parte dos problemas práticos e, por esse motivo, a metodologia proposta se destina aos problemas de Otimização Combinatória de uma única função objetivo que sejam PMA. Portanto, para efeito de uso dessa metodologia, consideramos um problema adequado a ser resolvido por *A-Teams* se esse for de Otimização Combinatória e PMA.

Como ilustração de um problema adequado à metodologia no presente capítulo, escolhemos o problema de escalonamento de tarefas *Flow Shop Problem* (para definição e maiores detalhes veja o capítulo 5). Este é um problema clássico de Otimização Combinatória que também é PMA, pois não está disponível nenhum algoritmo que resolva grandes instâncias deste problema num tempo razoável [GJ79], e existem várias heurísticas disponíveis na literatura que encontram soluções aproximadas.

4.5 Classificação dos Algoritmos Disponíveis

A maioria dos problemas NP-Difíceis possuem várias heurísticas que se propõem a resolvê-los aproximadamente, uma vez que é improvável que existam algoritmos exatos polinomiais [GJ79]. De acordo com a definição, muitos dos problemas da classe dos Algoritmos Inadequados também possuem várias heurísticas, pois , no tempo disponível, só são resolvidos aproximadamente ou com pequenas violações nas restrições. Essa disponibilidade de heurísticas para o problema que se pretende resolver por *A-Teams* é um fator coadjuvante na construção dos agentes, pois são, quase sempre, simples e fáceis de serem implementadas.

Com o objetivo de fazer uso de uma estrutura geral de *A-Teams* que será apresentada na próxima seção e, conforme a abordagem utilizada na obtenção de uma solução, as heurísticas e os algoritmos que se propõem a resolver um problema podem ser assim classificados:

Heurísticas de Construção

São heurísticas ou algoritmos que a partir de uma solução parcial (incompleta ou nula) constroem uma solução completa através de adições sucesivas de componentes individuais do problema. Geralmente, em algoritmos de construção, uma solução factível é obtida apenas no final do procedimento. Exemplos: O algoritmo

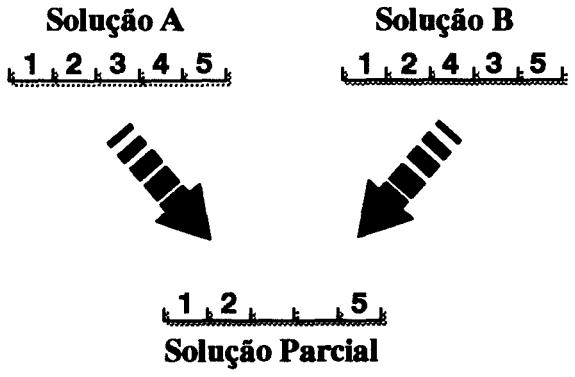
de Kruskal para construção de Árvores de Espalhamento Mínimo [CLR90], a heurística de Inserção Arbitrária para o PCV [GS85] e heurísticas gulosas em geral.

Algoritmos de Consenso

São algoritmos que geram uma nova solução (completa ou não) a partir de duas ou mais soluções. Esses algoritmos tentam capturar um "consenso de qualidade", gerando soluções potencialmente boas. Para tanto, os algoritmos de Consenso podem extrair informações das soluções de entradas de duas maneiras: através da Intersecção ou da União de soluções. Na Intersecção de soluções, o princípio utilizado é o seguinte: se duas ou mais soluções consideradas boas e possivelmente obtidas por algoritmos distintos possuem fragmentos[1] comuns, então esses fragmentos possuem grandes chances de estarem em alguma solução ótima. Na União de soluções, os algoritmos de Consenso "reduzem" o problema, considerando, na construção de uma nova solução, apenas os elementos do problema constantes na união das soluções originais. Como exemplo de algoritmos de Consenso por Intersecção, considere-se o problema de escalonamento de tarefas *Flow Shop Problem* de Permutação [Bak74]. Nesse problema, um conjunto de n jobs devem ser escalonados em m máquinas, e uma solução válida é qualquer permutação dos n jobs. Um possível algoritmo de Consenso por Intersecção, dadas duas soluções válidas, pode construir uma terceira solução parcial, onde a posição i é não vazia se e somente se nas duas soluções originais, um mesmo job ocupa a prosição i. Esse algoritmo é ilustrado na figura 4.3.

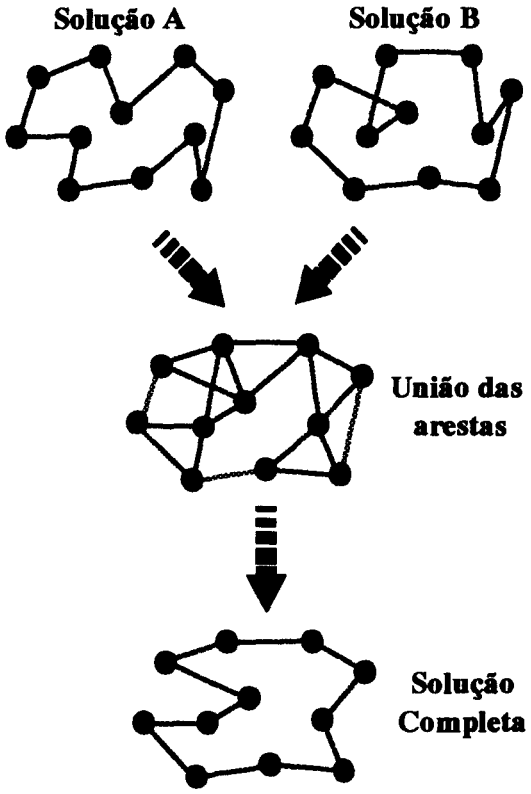
Como exemplo de algoritmo de Consenso por União, considere o algoritmo *Mixer* descrito por Souza [Sou93] para o PCV. O algoritmo *Mixer* constrói uma solução completa, utilizando apenas as arestas presentes nas duas soluções fornecidas como entrada. A figura 4.4 ilustra os passos do algoritmo *Mixer*.

Figura 4.3 Um exemplo de algoritmo de Consenso por Intersecção de soluções para o problema de escalonamento *Flow Shop Problem* de Permutação de 5 jobs. Uma nova solução parcial é construída através da intersecção de duas soluções originais A e B



1. Denominamos fragmento qualquer subconjunto dos elementos de uma solução.

Figura 4.4 Exemplo de um algoritmo de Consenso por União, o algoritmo ***Mixer***, proposto por Souza para o PCV. Dadas duas soluções A e B, o ***Mixer*** constrói uma nova solução completa utilizando apenas arestas presentes nas soluções A e B



Heurísticas de Relaxação

São heurísticas que expandem o espaço de soluções do problema original com o objetivo de obter um novo problema tratável, onde seja mais fácil a obtenção de soluções, mesmo que infactíveis. Essas heurísticas são úteis para gerarem Limites Inferiores. Como exemplo, considere-se o problema da Mochila (*Knapsack*), onde um conjunto de n objetos está disponível, cada objeto i possui um valor c_i e um peso a_i, e o objetivo é encontrar um sub-conjunto desses objetos que não exceda a capacidade b da mochila e maximize o valor total dos objetos armazenados. De outra maneira:

$$\max \sum_{j=1}^{n} c_j x_j \qquad \text{(Eq 4.5)}$$

$$\text{sujeito a} \quad \sum_{j=1}^{n} a_j x_j \leq b \qquad \text{(Eq 4.6)}$$

onde $x_i = 1$ se objeto i está na mochila e $x_i = 0$, no caso contrário. Uma heurística de relaxamento para este problema consiste em substituir a restrição de integralidade das variáveis x_i pela restrição $0 \leq x_i \leq 1$ e resolver o problema resultante por programação linear. Se no resultado obtido alguma variável x_i não for inteira, a solução não é factível, mas pode ser facilmente convertida a uma solução válida.

Algoritmos de Factibilidade

Estes algoritmos ajustam as soluções de problemas relaxados para tornarem-nas válidas em relação ao problema original. No exemplo anterior, o problema da Mochila, para construir uma solução factível a partir de uma que não o seja, basta fazer $x_i = 0$, sempre que valor de x_i não for inteiro.

Heurísticas de Melhoria

São heurísticas que a partir de uma solução factível tentam melhorá-la através de sucessivas trocas e rearranjo de seus elementos. São, geralmente, algoritmos de Busca Local e uma solução válida está sempre disponível durante a execução da heurística. Como exemplo, as heurísticas que utilizam o princípio *k*-opt (onde uma solução é fracionada em *k* partes que são posteriormente reorganizadas com o objetivo de minimizar a função objetivo) ou que exploram sucessivas vizinhanças a partir de uma solução inicial.

Algoritmos de Partibilidade

Estes algoritmos particionam o problema original em problemas menores, onde cada um é resolvido independentemente do outro, sendo uma solução para o problema original o conjunto das soluções para os diversos problemas resolvidos. Exemplo: Algumas heurísticas para o Problema de Roteamento de Veículos tratam este problema como sendo uma combinação do problema de alocação de recursos e o PCV [GM74].

Algoritmos de Composição

Dado que os algoritmos de Partibilidade geram sub-soluções para diversos sub-problemas, os algoritmos de Composição agrupam estas sub-soluções a fim de gerar soluções válidas para o problema original. No exemplo anterior, após resolver o problema de alocação de recursos e o PCV, as suas soluções são combinadas, gerando uma solução para o Problema de Roteamento de Veículos.

Algoritmos Interativos

Dependendo do problema, informações fornecidas por um elemento humano podem ser de grande valia no processo de resolução e avaliação. Esses algoritmos permitem, de alguma forma, a introdução de tais informações na geração e avaliação das soluções. Um exemplo seriam algoritmos Interativos para avaliação de soluções que envolvem conceitos subjetivos, tais como *Design* e estética.

Algoritmos Exatos

Esses algoritmos resolvem o problema de forma exata, ou seja, encontram sempre as soluções ótimas. Porém, para os problemas NP-Difíceis, tais algoritmos levam, no pior caso, tempo exponencial em relação ao tamanho da entrada. Mesmo assim, para alguns problemas e sob certas circunstâncias, os algoritmos exatos podem ser usados convenientemente. Por exemplo, quando estão disponíveis soluções muito próximas da ótima. bons limites inferiores e quando a instância a ser resolvida é de tamanho moderado, algoritmos exatos como *Branch&Bound* podem ser convenientes para garantir a otimalidade.

Algoritmos que convertem soluções Primais em Duais

No caso de problemas para os quais existem modelagens por programação inteira, pode-se convertê-los numa versão de programação linear e, em seguida, obter problemas duais. Nesse caso, o problema dual fornece limites inferiores para o problema original (de programação inteira). Dessa forma, uma solução para o problema original é heuristicamente convertida numa solução para o problema de programação linear, sendo finalmente convertida na correspondente dual. As soluções duais podem ser utilizadas por um algoritmo (por exemplo o de melhoria) e, assim, melhorar os limites inferiores, que por sua vez podem ser utilizados por outros algoritmos. Em seu estudo, Longo [Lon94] vêm aplicando com sucesso o uso de limites inferiores e superiores na resolução do problema de recobrimento de conjuntos.

Algoritmos que convertem soluções Duais em Primais

Esses algoritmos trabalham de maneira inversa aos algoritmos Primais-Duais.

4.6 Uma Estrutura Geral de A-Teams

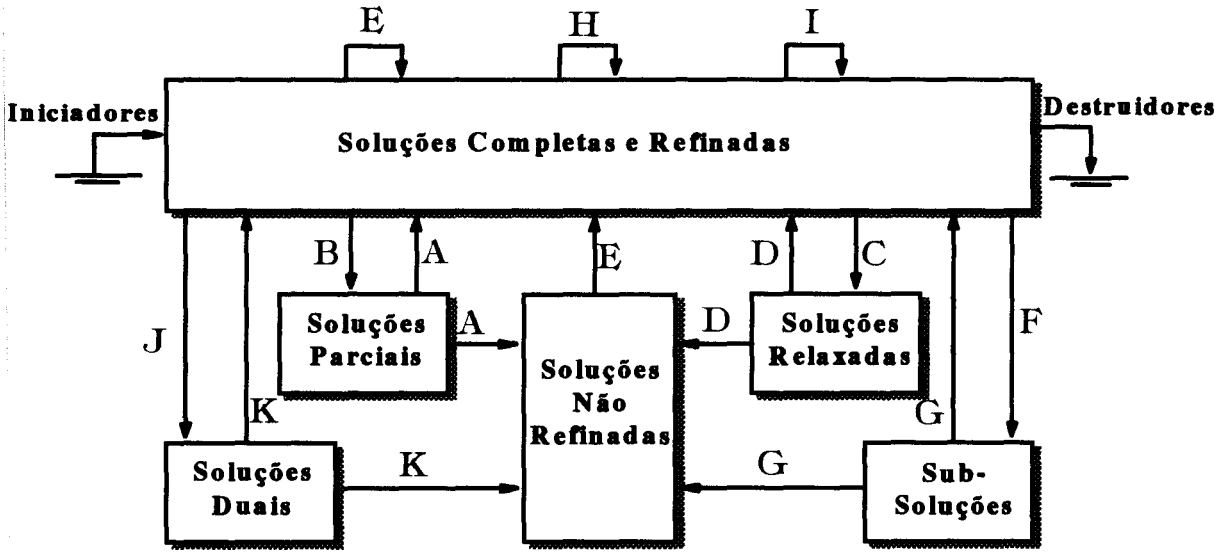
A classificação dos algoritmos e heurísticas descrita anteriormente permitiu desenvolver uma estrutura geral de *A-Teams*. Essa estrutura é denominada *geral* por contemplar a classificação anterior e deve ser considerada como ponto de partida e não como estrutura única e final.

A estrutura proposta na figura 4.5 ilustra como os diversos tipos de Agentes podem interagir através de memórias compartilhadas. Nessa figura, os retângulos especificam as memórias e as setas, as classes de Agentes que utilizam algoritmos conforme a classificação dada.

Essa estrutura utiliza seis memórias compartilhadas, sendo que cinco delas armazenam tipos distintos de soluções. Os tipos de soluções aos quais nos referimos são:

- **Soluções Completas** - São soluções válidas e completas que servem como soluções para o problema que está sendo resolvido. Essas soluções podem ser produzidas por Agentes de Construção, Agentes de Factibilidade, Agentes de

Figura 4.5 Estrutura geral proposta de um *A-Team* envolvendo todos os Agentes compostos por algoritmos que seguem a classificação dada. Os retângulos representam as memórias compartilhadas e as setas as classes de Agentes.



A Agentes de Construção
B Agentes de Consenso
C Agentes de Relaxamento
D Agentes de Factibilidade
E Agentes de Melhoria
F Agentes de Partibilidade
G Agentes de Composição
H Agentes Interativos
I Agentes Exatos
J Agentes Dual-Primal
K Agentes Primal-Dual

Melhoria, Agentes de Composição, Agentes Interativos, Agentes Exatos e Agentes que convertem soluções Duais em Primais.

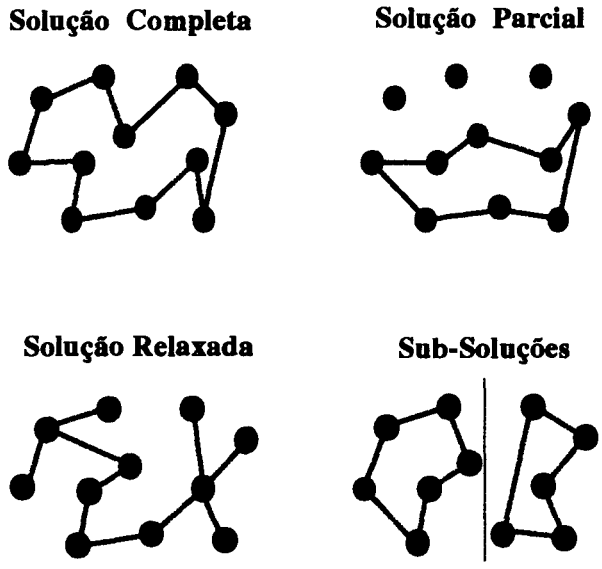
- **Soluções Parciais** - São soluções incompletas, ou seja, não possuem o número mínimo ou necessário de elementos (arestas, vértices, jobs, etc...). São produzidas por Agentes de Consenso ou Agentes que utilizam alguma filosofia de desconstrução de soluções. Essas soluções são geralmente encontradas nos passos intermediários das heurísticas de construção e, por esse motivo, são comumente concluídas por Agentes de Construção.

- **Soluções Relaxadas** - São soluções que violam alguma restrição do problema. São geradas por Agentes de Relaxamento.
- **Sub-Soluções** - São soluções para sub-problemas do problema original, geradas por Agentes de Partibilidade.
- **Soluções Duais** - São soluções para o problema original na sua versão dual. Geralmente, referem-se a soluções duais do problema modelado por programação Linear. São geradas por Agentes que transformam soluções Primais em Duais.

Para ilustrar os tipos de soluções mencionados, considere o PCV. Uma solução completa é qualquer ciclo Hamiltoniano no grafo do problema. Uma solução parcial pode ser considerada como uma solução que não inclui no seu ciclo todas as cidades. Soluções relaxadas podem ser exemplificadas por Árvores de Espalhamento Mínimo, que não constituem soluções válidas. Para exemplificar as Sub-Soluções, é preciso supor que o problema original foi dividido em dois novos PCV menores, obtendo-se, em cada, um ciclo Hamiltoniano. Esses dois ciclos, que juntos passam por todas as cidades, podem ser considerados como sub-soluções. As soluções duais, geralmente, não expressam com clareza o significado de uma solução. Veja figura 4.6.

A memória de soluções Completas e Refinadas armazena soluções válidas e é essencial a qualquer *A-Team*, visto que dela serão retiradas as melhores soluções do problema.

Figura 4.6 Alguns tipos de Soluções para o PCV.



Durante a execução de um *A-Team*, podem ser geradas soluções válidas e completas que sejam potencialmente boas, porém inferiores em qualidade quando comparadas às soluções presentes na memória de Soluções Completas e Refinadas. Dependendo da política de destruição adotada, essas soluções podem ser eliminadas antes mesmo de serem utilizadas por algum outro Agente. Nesse caso, essas soluções foram produzidas inutilmente. Para evitar que isso ocorra, foi introduzida a memória de Soluções Não Refinadas, cujo objetivo é permitir o aperfeiçoamento de tais soluções por alguma heurística de melhoria. Evidentemente, se alguns dos Agentes da estrutura proposta produz soluções muito boas, não há necessidade de depositá-las na memória de Soluções Não Refinadas. Por esse motivo, existem duas setas representando a mesma classe de Agentes que sai das memórias de Soluções Parciais, Duais, Relaxadas e Sub-soluções, para as memórias de Soluções Completas e Refinadas e Não Refinadas.

Essa estrutura geral tem o propósito de servir como ponto de partida para a especificação de um *A-Team*, visto que nem todos os problemas possuem algoritmos em todas essas classes e cada um possui características próprias. Podem existir algoritmos que não se enquadrem na classificação dada. Tais algoritmos podem pertencer a uma combinação das classes mencionadas e não devem ser desconsiderados. A estrutura ilustrada acima na figura 4.5, foi adotada pela simplicidade de compreensão e não representa todas as possíveis combinações de fluxos do Agentes.

Dependendo do problema, pode ser necessário armazenar outros tipos de dados, e não apenas soluções. Por exemplo, em problemas dinâmicos, os recursos, as restrições e até mesmo os objetivos podem ser alterados durante a execução do *A-Team*. Nesse caso, é essencial que as informações estejam depositadas em memórias, permitindo que os Agentes possam se inteirar constantemente do contexto corrente. Como as restrições podem ser alteradas a qualquer momento, as memórias também são "dinâmicas" no que se refere aos tipos de soluções armazenadas. Após a alteração de uma restrição, por exemplo, a memória de Soluções Completas pode estar armazenando soluções factíveis e infactíveis. A tal memória pode estar associado um Agente Destruidor que elimina todas as soluções que, naquele instante, não satisfazem as restrições. Eventualmente, o Agente Destruidor pode eliminar todas as soluções da memória, sendo necessário re-iniciar o *A-Team*. Existem, porém, pelo menos dois motivos para que as soluções recentemente denominadas infactíves sejam processadas por algum outro Agente, que não o Destruidor:

1. caso em que as alterações nas restrições não são bruscas, sendo necessários pequenos ajustes para tornar factível uma solução infactível, e
2. a possibilidade de explorar os esforços despendidos na obtenção de soluções antes das alterações nas restrições.

Uma possível maneira de restaurar tais soluções é criar uma memória de soluções infactíves, para onde outro Agente realoca as soluções que violem poucas restrições, e destruas as demais (soluções profundamente infactíveis). A essa memória de soluções infactíves estão associados Agentes de Factibilidade, que fechariam o ciclo com a memória de Soluções Completas e Refinadas.

4.7 As Memórias

Num *A-Team*, as memórias possuem duas funções principais: armazenar e permitir o compartilhamento de dados entre os Agentes. A especificação de memórias merece atenção especial, pois seu tamanho, bem como sua estrutura, podem influir na performance de um *A-Team*.

4.7.1 Tamanho das Memórias

O tamanho de uma memória é dependente do tipo de dados que ela armazena, podendo ser encontrado diretamente a partir da definição do problema, ou calculado empiricamente através de testes. No primeiro caso, estamos nos referindo às memórias que armazenam dados de uma instância, como exemplo, uma matriz de custo que é alterada dinâmicamente ou a disponibilidade de recursos. No segundo caso, referimo-nos às memórias que armazenam as soluções do problema, cujo tamanho pode variar entre uma única solução e um número máximo determinado pela capacidade do sistema em que o *A-Team* será implementado. Poucas soluções numa memória pode significar uma amostragem muito reduzida do espaço de soluções e, conseqüentemente, uma possível estagnação dos Agentes em mínimos locais. Isso pode ser explicado do seguinte modo: após um tempo suficientemente grande, as diversas maneiras de combinar as soluções através dos Agentes disponíveis podem não ser suficientes para produzir uma nova população de soluções que representem outros mínimos locais e, eventualmente, algum mínimo global. Por outro lado, uma memória muito grande pode representar esforço inútil com soluções que estejam próximas a mínimos locais, previamente identificados como ruins. Além disso, os Agentes de Consenso, devido à grande diferença entre a pior e a melhor solução, podem trabalhar inutilmente. Isso pode ocorrer caso não seja possível extrair de soluções muito distintas o que de bom elas tenham em comum.

O tamanho ideal de uma memória é aquele que permite a busca em várias regiões distintas do espaço de soluções e, ao mesmo tempo, permite uma exploração local das soluções potencialmente boas. O tamanho depende do problema em questão e é determinado empiricamente através de sucessivas execuções dos *A-Teams*. Tipicamente, para os problemas Combinatórios encontrados na prática, o tamanho das memórias é de uma a várias centenas de soluções. Uma característica das memórias muito pequenas é que a diversidade das soluções é rapidamente reduzida após um "pequeno" tempo de execução, ao passo que nas memórias muito grandes, a redução na diversidade é muito pequena, mesmo após um "longo" tempo de execução. Neste caso, há um desperdício de esforço computacional em tratar soluções muito ruins em relação às melhores já existentes na memória.

4.7.2 Avaliação das informações armazenadas pelas memórias

Dentre as memórias da estrutura proposta na figura 4.5, apenas a memória de Soluções Completas e Refinadas possui Agentes denominados Destruidores. Esses Agentes são utilizados como os únicos capazes de eliminar soluções da memória, poupando os demais agentes de tal tarefa e unificando o modo pelo qual as soluções são eliminadas.

As demais memórias, se permanecerem na forma simplificada da figura 4.5, são memórias que funcionam como *buffers* entre os Agentes que depositam e retiram soluções. Nesse caso está implícito um Agente Destruidor que elimina uma solução toda vez que ela for consultada. Tal mecanismo é sugerido sempre que as soluções armazenadas pela memória não sejam mensuráveis qualitativamente (não é possível deduzir quão boa ou ruim uma solução possa ser) ou sempre que se desejar reprocessar dados ou soluções. Por exemplo, as memórias Parciais funcionam como *buffers* devido à impossibilidade de calcular a qualidade de uma solução Parcial, já as memórias Não Refinadas funcionam como *buffers* para permitir o refinamento de suas soluções. As memórias de soluções Duais, Sub-soluções ou soluções Relaxadas podem funcionar como *buffers* ou não, dependendo do problema e dos algoritmos disponíveis. Essas memórias podem alcançar a complexidade das memórias de Soluções Completas e Refinadas, possuindo Agentes de Destruição mais sofisticados e todos os demais tipos de Agentes associados à memória de Soluções Completas e Refinadas.

4.7.3 O Armazenamento de Soluções

Para facilidade de acesso e eliminação das soluções, a memória de Soluções Completas e Refinadas deve armazenar as soluções numa seqüência não (de)crescente de qualidade. Dessa maneira, as melhores, intermediárias e piores soluções podem ser rapidamente identificadas, agilizando também a identificação de soluções redundantes. As demais memórias, se forem consideradas como *buffers*, podem ser implementadas segundo a política FIFO.

4.7.4 Iniciação das Memórias

Antes que os Agentes dos *A-Teams* comecem a processar, é preciso iniciar as memórias com soluções. Num *A-Team* que possua uma estrutura semelhante à estrutura proposta na seção 4.6, basta iniciar a memória de Soluções Completas e Refinadas, como indicado pelo Agente Iniciador da figura 4.5. As demais memórias dispensam iniciação, pois podem ser preenchidas pelos seus respectivos Agentes que lhes depositam dados.

As memórias podem ser iniciadas de diversas maneiras. Porém, algumas delas apresentam-se com maior relevância. São elas:

1. **Aleatória** - A memória é preenchida por soluções geradas aleatoriamente. Esta é uma maneira rápida de iniciar a memória e garante grande diversidade entre as soluções.
2. **Construção** - A memória é iniciada com soluções produzidas por algoritmos de construção. Claramente, estes algoritmos devem ser capazes de produzir soluções distintas, a fim de garantir diversidade entre elas.
3. **Aleatória e Construção** - Parte da memória é preenchida por algoritmos que produzem soluções aleatoriamente e o restante por algoritmos de construção. Desta maneira se garante um mínimo de diversidade.

Embora iniciar a memória com soluções geradas aleatoriamente garanta diversidade, na prática isso pode significar um número grande de soluções muito ruins que, por não induzirem a boas soluções, são inutilmente processadas pelos Agentes.

Os algoritmos de construção, por sua vez, podem, produzir soluções muito semelhantes. Nesse caso, pode-se usar mais de um algoritmo de construção ou então aumentar o não determinismo de tais algoritmos. Esta abordagem poupa dos Agentes a necessidade de um pré-refinamento das soluções, pois essas são relativamente boas quando comparadas às geradas aleatoriamente.

A última abordagem mencionada pode ser utilizada quando não se consegue, através de algoritmos de construção, produzir muitas soluções distintas.

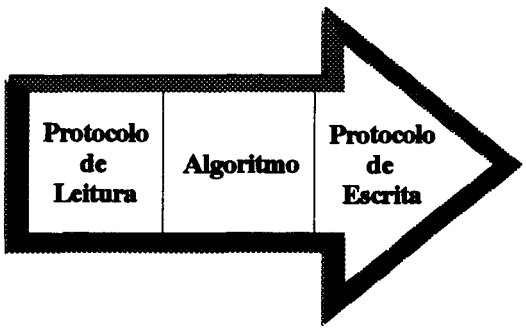
Inicialmente, preencher completamente a memória com soluções pode ser desnecessário, pois algumas soluções que não foram utilizadas por nenhum Agente serão eliminadas para abrir espaço às recem produzidas. Para minimizar este incidente, a memória pode ser preenchida, por exemplo, em apenas 80% de sua capacidade.

4.8 Os Agentes

Os Agentes de um *A-Team* são os elementos ativos da organização e, cooperativamente, através da combinação e modificação de dados e soluções, produzem melhores resultados que nos casos em que são utilizados isoladamente.

Os Agentes são compostos de um algoritmo que se propõe a resolver o problema e de protocolos de comunicação para com as memórias, como ilustrado na figura 4.7.

Figura 4.7 Estrutura dos Agentes. O protocolo de Leitura e Escrita se refere a maneira pela qual o Agente se comunica com a memória de onde retira e deposita os dados, respectivamente.



Os Protocolos de Leitura descrevem as memórias das quais os Agentes obtêm seus dados e como esses dados são acessados - política de seleção. Para as memórias que funcionam como *buffers* é indiferente a política de seleção utilizada. Para as demais memórias, existem cinco políticas gerais de seleção:

Da melhor para a pior

O Agente seleciona a melhor solução presente na memória que não tenha sido por ele previamente selecionada ou gerada. Esta política é utilizada, geralmente, em Agentes de Melhoria determinísticos, pois, dessa maneira às melhores soluções da memória é dada a posibilidade de conduzirem a soluções ainda melhores. Entretanto, não se aplica a Agentes não determinísticos, já que uma mesma solução de entrada pode produzir resultados diferentes em execuções sucessivas. Para armazenar as novas soluções produzidas, é necessário eliminar outras, geralmente as de pior qualidade que ainda não foram utilizadas por estes Agentes. Desse modo, esta é uma política que tende a diminuir a diversidade das memórias.

Da pior para a melhor

O Agente seleciona a pior solução presente na memória que não tenha sido por ele previamente selecionada ou gerada. Esta política também não se aplica à Agentes não determinísticos. Ao contrário da anterior, esta política permite que a pontencialidade das piores soluções seja explorada, mantendo uma maior diversidade na memória. Encontrar as soluções ótimas ou quase ótimas, porém, pode ser um processo mais lento, pois, normalmente, soluções muito ruins não induzem os Agentes à produzirem soluções muito boas.

Aleatória

O Agente escolhe aleatoriamente uma solução presente na memória. Se o Agente for Determinístico, uma nova solução é sorteada sempre que a solução escolhida tenha sido por ele previamente selecionada ou gerada. Se o Agente não for Determinístico, qualquer solução pode ser escolhida. Esta política é geralmente utilizada por Agentes de Consenso, Relaxamento e Partibilidade.

Da melhor para a pior com distribuição linear de probabilidade de escolha

O Agente seleciona uma solução com uma distribuição linear de probabilidade, onde a probabilidade de escolher a pior solução é próxima de zero, e cresce à medida que se aproxima das melhores soluções da memória. Esta política pode ser usada por Agentes de Melhoria não Determinístico, uma vez que todas as soluções podem ser contempladas: as melhores, porém, com maior probabilidade. Pode ser usada também pelos Agentes que transformam soluções Primais em Duais, permitindo, assim, que a memória de soluções Duais tenha maior diversidade.

Da pior para a melhor com distribuição linear de probabilidade de escolha

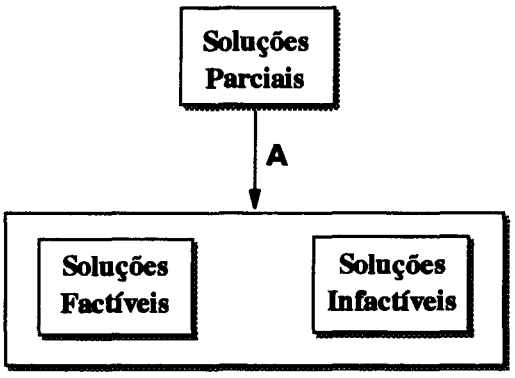
A seleção de soluções acontece como na política anterior, à exceção da distribuição linear de probabilidade que cresce da melhor para a pior.

4.8.1 A escolha dos algoritmos e heurísticas

A classificação dos algoritmos da seção 4.5 e a estrutura geral da seção 4.6 foram apresentadas exclusivamente para guiar a especificação de *A-Teams*, visto que nem todos problemas possuem algoritmos disponíveis em todas as classes e, conseqüentemente, nem todos os ciclos da estrutura geral podem ser implementados. Alguns algoritmos podem não se enquadrar na classificação dada, mas isso não significa que devam ser descartados. Os algoritmos podem pertencer a uma combinação das classes, e estariam representados na estrutura geral da figura 4.5 por setas que partem de ou chegam a retângulos que incorporam mais de uma memória. Como exemplo, considere-se um algoritmo **A** que dada uma solução parcial, gera uma nova solução completa factível ou não. Esse algoritmo, que não se enquadra na classificação dada, pode ser representado num *A-Team* como mostrado na figura 4.8.

A fim de permitir *Feedback* e interação entre os Agentes, é necessário, conforme mencionado no capítulo 3, que os *A-Teams* possuam fluxos de dados cíclicos. Desta forma, um dos primeiros passos na especificação de Agentes é a identificação de quais dos algoritmos disponíveis formam ou viabilizam ciclos, através da criação de outros algoritmos. Dependendo do problema, a criação de algoritmos para compor novos ciclos pode ser uma tarefa fácil. Por exemplo, se o problema já possui algoritmos de Construção, podem-se especificar algoritmos de Consenso, a fim de formar novos ciclos.

Figura 4.8 Exemplo do uso de algoritmos que não pertencem à classificação da seção 4.5. O algoritmo **A** lê de uma solução parcial, constrói uma nova solução, a qual pode ser armazenada tanto na memória de soluções Factíveis como na de soluções Infactíveis, dependendo ou não da violação de restrições.



Dentre todos os algoritmos selecionados que formam ciclos, alguns podem não ser adequados aos *A-Teams*. Este pode ser o caso dos algoritmos de granularidade exageradamente grossa, ou seja, algoritmos que levam um tempo muito grande para retornarem uma solução. Os Agentes que utilizam esses algoritmos não se valem do trabalho em grupo, permanecendo a maior parte do tempo processando suas soluções, independentemente do que é produzido pelos demais Agentes do time. Como exemplo, as metas heurísticas de melhoria Buca Tabu [Glo90] e *Simulated Annealing* [KGV83]. Uma maneira de afinar a granularidade destes algoritmos, com o objetivo de utilizá-los nos *A-Teams*, seria através do uso de um número pequeno de iterações no Tabu e decréscimo rápido de temperatura no *Simulated Annealing*. Porém, na maioria dos casos, estas meta-heurísticas não produzem resultados significativos nessas condições. Uma possível fronte de pesquisa seria a implantação conveniente dessas meta-heurísticas num *A-Team*.

Uma vez que os *A-Teams* permitem que uma solução seja processada por mais de um Agente, não faz sentido utilizar dois ou mais Agentes que empregam algoritmos de uma mesma abordagem na resolução do problema. Nos *A-Teams* é interessante que haja diversidade no modo de tratar as soluções, permitindo que vários "pontos de vista" sejam utilizados na obtenção de soluções, desejadamente melhores.

De um modo geral, os Agentes de um *A-Team* devem ser capazes de produzir, simultaneamente, diversidade e convergência na memória de soluções Completas e Refinadas. A diversidade tem a função de livrar a memória de mínimos locais, enquanto a convergência tem o objetivo de explorar a potencialidade de cada solução. A relação entre a diversidade e convergência é estabelecida, principalmente, pela política de destruição utilizada pelos Agentes Destruidores, pelos Agentes de Melhoria, Consenso e Construção. Os Agentes Destruidores eliminam dados definindo, desse modo, o perfil da memória de soluções Completas e Refinadas. Os Agentes de Melhoria, normalmente, tentam produzir soluções não muito distintas da original, mas que sejam de melhor qualidade. Quanto aos Agentes de Consenso, estes também são induzidos a produzir soluções semelhantes às soluções presentes na memória. Resta aos Agentes de Construção e aos demais Agentes do time a tarefa de produzirem diversidade.

4.9 Métodos de Destruição

As memórias que funcionam como *buffers* eliminam automaticamente os dados que são lidos e, por esse motivo, não possuem Agentes Destruidores.

Os Agentes Destruidores são essenciais à memória de Soluções Completas e Refinadas, pois são eles quem julgam e eliminam, baseados em políticas de destruição, qual dado deve ser eliminado para abrir espaço a um novo. Normalmente, as políticas de destruição associam a cada solução presente na memória uma probabilidade de destruição, relativa a algumas de suas características, tais como: qualidade, número de restrições violadas, número de vezes que a solução foi consultada por outros Agentes, etc.

A função dos Agentes Destruidores não é apenas garantir espaço a novas soluções. Num *A-Team*, tão importante quanto produzir bons dados é a tarefa de eliminar dados não promissores. As políticas de destruição influem diretamente na diversidade e convergência das soluções da memória, podendo direcionar a produção de soluções do *A-Team* em uma dada direção.

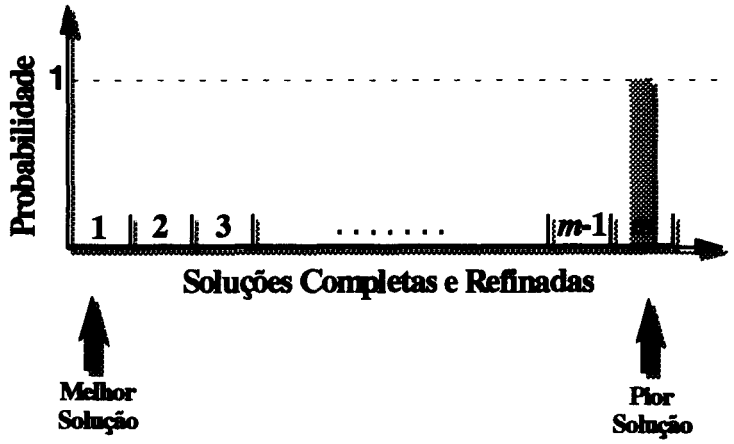
Sendo a memória de Soluções Completas e Refinadas composta de no máximo m soluções, as políticas de destruição, geralmente associadas à qualidade das soluções, são descritas nas sub-seções que seguem.

4.9.1 Destruição da Pior Solução

A solução de pior qualidade é destruída com probabilidade 1, para que novas soluções, melhores que a destruída, possam ser alocadas. A figura 4.9 ilustra esta política.

Um Agente Destruidor que utilize esta política eleva, monotonicamente, ao longo do tempo, a média de qualidade das soluções na memória. Em geral, as soluções geradas

Figura 4.9 Política de destruição: destruição da pior solução. Sempre que não houver espaço na memória para alocar novas soluções, esta política seleciona a pior solução com probabilidade 1 de destruí-la. Novas soluções que sejam inferior em qualidade à pior solução da memória não são aceitas.



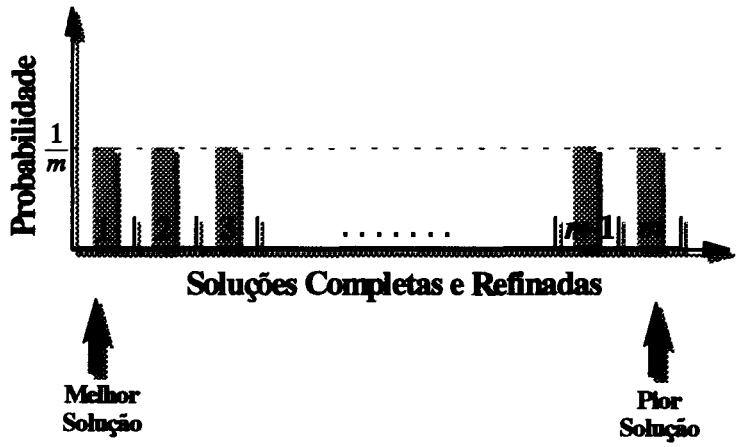
pelos Agentes de Consenso e Melhoria[1], não são siginificativamente diferentes das soluções originias e, quando são melhores, ocupam os espaços desocupados pelas piores. Dessa forma, o *A-Team* é induzido a uma rápida convergência, através da redução da diversidade entre as soluções. Essa rápida convergência pode preencher a memória com soluções vizinhas de algum mínimo local, não permitindo que o *A-Team* encontre, através da modificação e combinação entre soluções, outros mínimos locais.

4.9.2 Destruição de qualquer solução com distribuição uniforme de probabilidade

Nesta política todas as soluções da memória possuem a mesma probabilidade de serem eliminadas. A figura 4.10 mostra a distribuição uniforme de probabilidade para as soluções da memória.

Esta política reduz a taxa de convergência dos *A-Teams*, visto que a memória permanece por mais tempo com grande diversidade. Entretanto, muitas soluções boas podem ser eliminadas antes mesmo de serem utilizadas por outros Agentes. Nesse caso estas soluções foram produzidas inutilmente, podendo degradar a performance do *A-Team*.

Figura 4.10 Política de destruição: destruição de qualquer solução com distribuição uniforme de probabilidade. Sempre que não houver espaço na memória para alocar novas soluções, esta política seleciona qualquer solução presente na memória. Novas soluções que sejam inferiores em qualidade à pior solução da memória não são aceitas.



1. Em verdade, quase todos os Agentes preservam alguns fragmentos das soluções originais nas soluções recem criadas, visto que utilizam as originais como ponto de partida.

4.9.3 *Destruição de soluções com distribuição linear de probabilidade*

Esta política de destruição associa à melhor solução presente na memória uma probabilidade de ser eliminada igual a zero, e uma probabilidade linearmente crescente às demais soluções, da segunda melhor para a pior. A distribuição linear de probabilidade que representa esta política está ilustrada na figura 4.11.

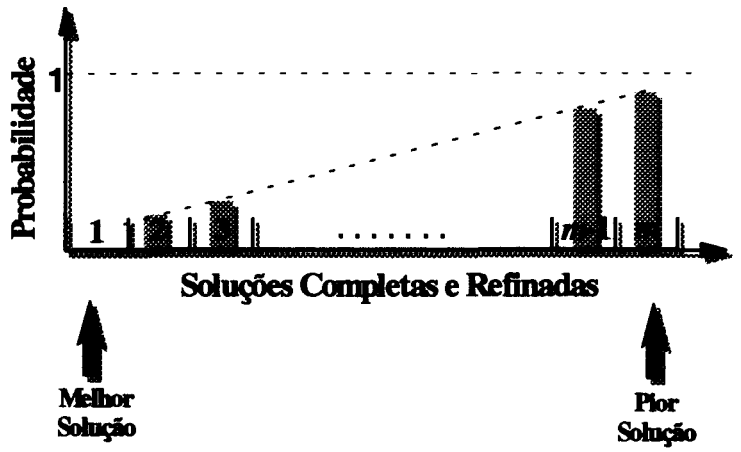
Diferentemente da primeira e segunda políticas apresentadas, esta política representa um bom compromisso entre a diversidade e a velocidade de convergência da memória. Eliminando as piores soluções com maior probabilidade, esta política permite uma convergência mais lenta que a segunda política apresentada. Eliminando com menor probabilidade as soluções de melhor qualidade, evita que muitas soluções "semelhantes" e de boa qualidade permaneçam na memória, aumentando a diversidade entre as melhores e, ao mesmo tempo, evitando que boas soluções sejam perdidas.

De forma geral, esta política tende a apresentar melhores resultados que as anteriores, como foi observado por Souza nos *A-Teams* para o PCV [Sou93].

4.9.4 *Destruição Dinâmica, conforme a diversidade da memória*

Utilizando qualquer uma das políticas mencionadas anterioremente, a partir de algum instante do tempo, as memórias estarão repletas de soluções que pertençam à vizinhança de alguns mínimos locais. Isto ocorre devido aos Agentes de Melhoria e Consenso, que forçam à convergência das soluções na memória.

Figura 4.11 Política de destruição: destruição de soluções com distribuição linear de probabilidade. Sempre que não houver espaço na memória para alocar novas soluções, esta política seleciona com probabilidade linear uma solução presente na memória (exceto a melhor). Novas soluções que sejam inferiores em qualidade à pior solução da memória não são aceitas.



A partir do momento em que a memória atingir uma baixa diversidade, os Agentes podem não ser capazes de encontrar outros mínimos locais através da reutilização e combinação das soluções, mesmo que bastante tempo seja dispendido neste sentido. Esta política de destruição dinâmica permite, temporariamente, que a memória aceite soluções de qualquer qualidade, sempre que um baixo nível de diversidade for atingido. Assim sendo, a diversidade aumenta e as chances de se produzirem soluções que pertençam a outros mínimos locais também.

Para que a memória não destrua completamente as boas soluções produzidas anteriormente, esta política atribui a apenas uma parte da memória a possibilidade de substituir soluções boas por soluções piores. Para que essas soluções tenham a chance de proliferarem, após a renovação da fração da memória onde é permitida a substituição por quaisquer soluções, é preciso retornar à antiga política de destruição adotada e restringir a inserção de soluções piores. Permite-se, desta maneira, que a convergência da memória volte a ocorrer.

Portanto, um Agente que utilize uma política de destruição dinâmica, deve ser capaz de medir a diversidade das soluções da memória e usar duas outras políticas de destruição: uma para operar quando a memória aceita apenas soluções melhores que a pior solução corrente e outra, no caso contrário.

Não adiantaria adotar esta política de destruição dinâmica, se nenhum dos Agentes que escrevem na memória é capaz de produzir soluções diversificadas. Pode-se, em última instância e com esta finalidade, utilizar Agentes que produzam soluções aleatórias.

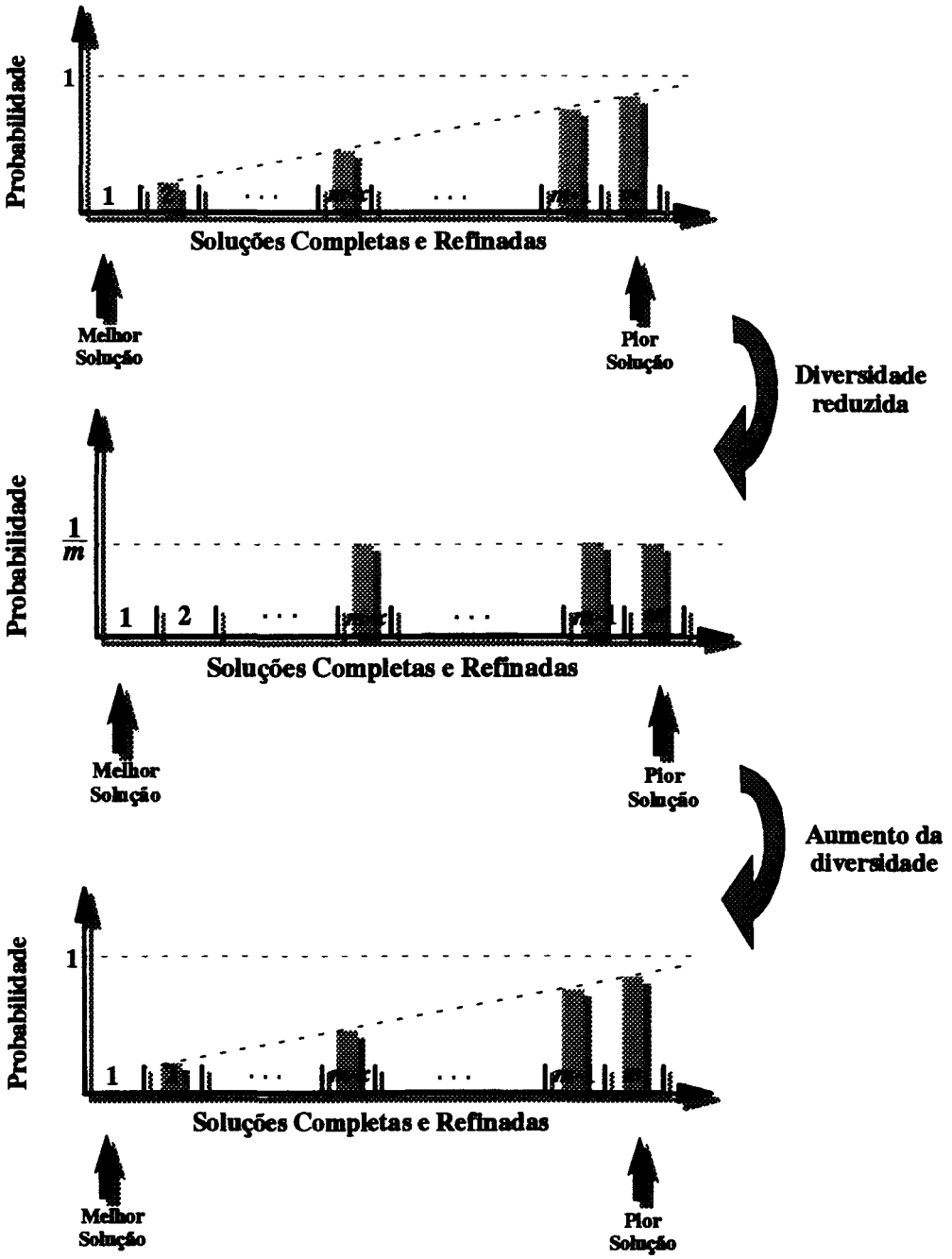
Como exemplo de uma possível política de destruição dinâmica, considere-se que:

- a memória do *A-Team* seja composta por m soluções;
- a política de destruição adotada, quando a memória não aceita soluções piores que a pior presente, seja a política de destruição de soluções com distribuição linear de probabilidade;
- a política de destruição adotada, quando a memória aceita qualquer solução, seja a política de destruição de soluções com distribuição uniforme de probabilidade para as x piores soluções;
- a diversidade das soluções na memória possa ser considerada baixa quando, após a produção consecutiva de y soluções, nenhum Agente tenha sido capaz de inserir soluções na memória;
- após inseridas z soluções na memória, quando esta aceita soluções de qualquer qualidade, volte-se a restringir a inserção de soluções piores e à respectiva política de destruição.

A figura 4.12 ilustra uma memória e as probabilidades de destruição das soluções em três instantes consecutivos do tempo. Primeiro, a memória não aceita soluções de qualidade inferior à pior solução. Segundo, após fracassadas as inserções consecutivas de y soluções, o Agente Destruidor altera sua política de destruição. Finalmente, após

Figura 4.12

Exemplo de uma política de destruição dinâmica. Num primeiro instante, sempre que não houver espaço na memória para alocar novas soluções, esta política seleciona com probabilidade linear uma solução presente na memória (exceto a melhor). Novas soluções que sejam inferiores em qualidade à pior solução da memória não são aceitas. Quando é considerado um baixo nível de diversidade, esta política seleciona qualquer uma das x piores soluções presentes na memória e novas soluções com qualquer qualidade são aceitas. Num último instante, após a inserção de z soluções e conseqüente aumento na diversidade, esta política volta a operar como inicialmente.



a inserção de outras z soluções de qualquer qualidade, o Agente Destruidor volta a utilizar a política de destruir soluções com distribuição linear de probabilidade.

4.10 Implementação

Devido à total independência dos Agentes nos *A-Teams*, sua implementação pode ser modularizada. Módulos funcionalmente independentes são mais fáceis de implementar (é preciso menos interface e nenhum conhecimento dos demais agentes), mais fáceis de serem modificados e testados (efeitos colaterais são minimizados e a propagação de erros não existe) e possibilita a reutilização dos módulos (agentes). Além dessas vantagens, a modularização permite uma análise evolutiva do *A-Team*, ou seja, após a implementação e inserção de um Agente no time, pode-se observar quais são os resultados produzidos por essa expansão.

Deve-se priorizar o uso de linguagens e ferramentas que permitam processamento distribuído e/ou paralelo.

4.11 Validação e Modificações

Após a implementação de todos os Agentes especificados anteriormente, os resultados obtidos podem ser avaliados. Dois fatores, dentre outros, são relevantes na análise dos resultados: o nível de qualidade das soluções obtidas e o tempo dispendido para produzi-las.

Caso as soluções desejadas sejam obtidas no intervalo de tempo disponível para resolver o problema, então os *A-Teams* alcançaram os objetivos especificados anteriormente. Porém, se as soluções obtidas satisfazem o nível de qualidade, mas o tempo necessário de processamento é muito longo, é sinal de que modificações são necessárias. Nesse caso, pode-se tentar melhorar a performance do *A-Team* de pelo menos três maneiras:

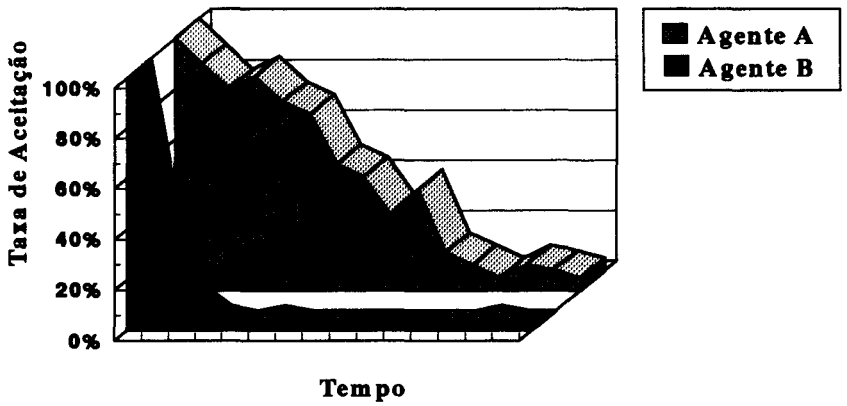
1. otimizando a implementação de cada um dos Agentes, por exemplo, através do uso de outros algoritmos e estruturas para manipulação de dados que sejam mais eficientes;
2. paralelizando ou distribuindo o processamento dos Agentes;
3. modificando os fluxos de dados e Agentes utilizados no *A-Team*. Esta tarefa necessita de uma análise mais profunda do comportamento dos Agentes e da dinâmica das soluções na memória. Nos próximos parágrafos são detalhadas algumas sugestões para modificações de *A-Teams*.

Pode-se ter ainda um terceiro caso em que os *A-Teams*, mesmo após muito tempo de processamento, não produzam as soluções esperadas. Pode-se então observar o comportamento das memórias e dos Agentes ao longo do tempo, identificando como cada Agente atua na produção das soluções e como estas passam pelas memórias.

No início do processamento, quase todas as soluções produzidas pelos Agentes são aceitas pela memória de Soluções Completas e Refinadas. Entretanto, ao passo que mais soluções são aceitas pela memória, o número de soluções produzidas iguais e piores tende a crescer, pois soluções "mais refinadas" já se fazem presentes. Porém, nem todos os Agentes conseguem escrever na memória com a mesma freqüência no decorrer do tempo. Por exemplo, Agentes de Melhoria que empregam boas heurísticas tendem a ter suas soluções aceitas pela memória por maior tempo que Agentes que produzem soluções aleatórias. O gráfico da figura 4.13 exemplifica as taxas de aceitação das soluções produzidas por dois Agentes: **A**, que emprega boa heurística de melhoria e **B**, que produz soluções aleatoriamente. A taxa de soluções aceitas de um Agente pode ser definida como sendo a razão entre o número de soluções que o Agente teve aceitas pela memória num dado intervalo de tempo e o número de soluções que este Agente produziu no mesmo intervalo. Neste exemplo, durante um mesmo intervalo de tempo, o Agente **A** possui uma maior taxa de soluções aceitas que **B**.

Idealmente, a cada instante, ao menos um Agente deveria ser capaz de inserir suas soluções na memória e, dessa forma, melhorar continuamente a qualidade das soluções. No gráfico da taxa de aceitação em função do tempo isso significa um valor sempre maior que zero, não importando quanto. Porém, na prática, a memória pode convergir rapidamente para alguns mínimos locais e os Agentes não serem capazes de produzir soluções melhores. Nesse instante, todos os Agentes terão a sua taxa de

Figura 4.13 Exemplo do decrescimo da Taxa de aceitação de soluções de dois Agentes **A** e **B**, em instantes diferentes de tempo.



aceitação iguais a zero. Se os Agentes permanecerem indefinidamente com a taxa de aceitação igual a zero e a melhor solução não satisfaz a qualidade desejada, é sinal de que são necessárias modificações. Pode-se aumentar o tamanho da memória, substituir a política de destruição, ou ainda, modificar os Agentes (por exemplo, introduzindo não determinismo), a fim de que outros mínimos locais sejam encontrados.

Os Agentes podem, ainda, possuir taxa de aceitação maior que zero e, mesmo assim, a memória não conter boas soluções. Nesse caso a convergência da memória está muito baixa indicando, por exemplo, a necessidade da troca da política de destruição, a ausência de Agentes de Consenso, Melhoria ou que empregam algoritmos com filosofia de Busca Local.

Mais importante do que ter um grande número de Agentes, é agrupar Agentes que se beneficiam da execução mútua. Uma característica destes Agentes é a diferença nas abordagems usadas para resolver o problema. Mesmo que muitos dos algoritmos que utilizam abordagens distintas não sejam disponíveis, a oportunidade de estudá-los ao mesmo tempo em que se busca um modo de agrupá-los cooperativamente, pode sugerir a concepção de novos algoritmos que sejam adequados ao *A-Team*.

Se, após exaustivas modificações, os resultados obtidos pelo *A-Team* não satisfizerem a qualidade desejada, fica a certeza de que nenhum dos métodos aproximados utilizados nos Agentes seriam capazes de produzir soluções melhores.

Por serem organizações "abertas", os *A-Teams* garantem-nos, no pior caso, soluções tão boas quanto as soluções fornecidas por métodos aproximados de construção. Para que isso seja verdade, basta introduzir, no início da execução do *A-Team* as soluções fornecidas por tais métodos.

4.12 Sobre a Representação Gráfica

A representação gráfica de *A-Teams* apresentada no capítulo 3 é muito simples e pode, em alguns casos, prejudicar a legibilidade da informação representada. Nesta seção acrescenta-se à representação gráfica de *A-Teams* uma nova representação para os Agentes, cujo propósito é facilitar o desenho e incrementar a legibilidade.

Alguns *A-Teams* podem possuir memórias que funcionam como fontes de dados, ou seja, memórias que são atualizadas apenas por Agentes Externos[1]. Essas memórias, geralmente, armazenam dados que representam o contexto corrente e, por esse motivo, podem ser acessadas pela maioria dos Agentes do time. A representação gráfica que se vale apenas de setas e retângulos pode tolher a legibilidade e expressividade, uma vez que vários retângulos estarão formando intersecções e as setas saindo sempre de retângulos mais externos.

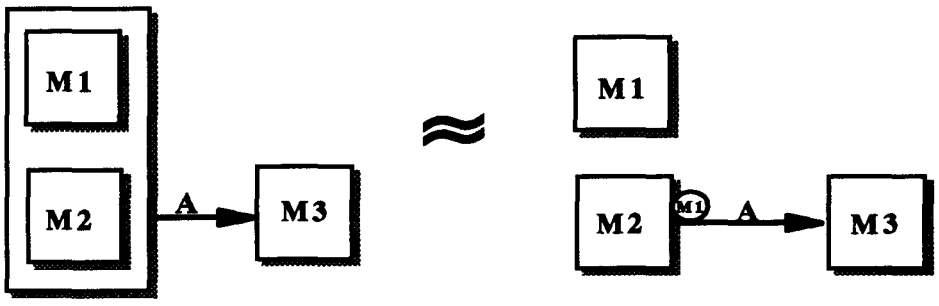
1. Em problemas dinâmicos, por exemplo, estes Agentes equivalem a eventos externos.

Para contornar este problema, podem-se dotar as setas que representam um Agente da capacidade de representarem mais de uma memória de leitura, mesmo sendo iniciadas em uma única memória. Em tal representação, um Agente é representado por setas, como anteriormente, e círculos sobre as setas, que indicam quais as demais memórias de leitura. A figura 4.14 ilustra, através de um exemplo de equivalência com a representação anterior, esta nova representação.

O exemplo da figura 4.14 é muito simples e foi utilizado apenas para ilustrar a equivalência entre as representações. Como exemplo mais complexo, considere-se um problema cujo *A-Team* possui 4 memórias e 5 agentes. As memórias e os Agentes são definidos como segue:

- M1 - memória de soluções completas e factíveis;
- M2 - memória de soluções infactíveis;
- M3 - memória de soluções parciais;
- M4 - memória das restrições correntes do problema;
- A1 - Agente de Consenso que lê soluções de M1 e escreve as parciais em M3;
- A2 - Agente de Construção que lê uma solução parcial em M3 e as restrições em M4 para construir uma solução completa factível ou não;
- A3 - Agente de Melhoria que lê uma solução de M1 e as restrições em M4 e gera uma nova solução factível.
- A4 - Agente de Factibilidade que transforma uma solução infactível de M2, conforme as restrições em M4, numa solução factível para M1.
- A5 - Agente Externo que atualiza as restrições de M4;

Figura 4.14 A equivalência entre a representação descrita no capítulo 3 e a representação proposta nesta seção. Ambas representam um Agente A que lê seus dados de entrada tanto de M1 como de M2.

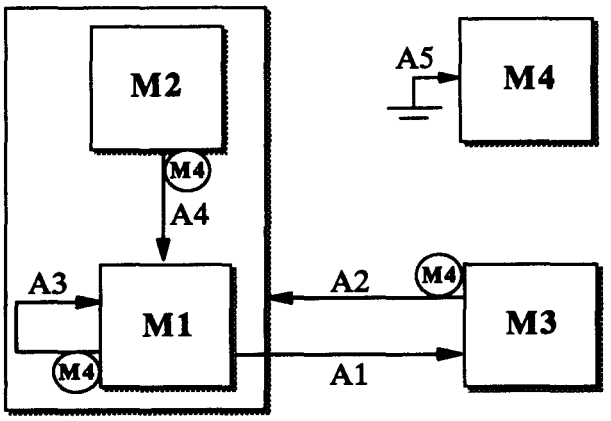


Qualquer tentativa de desenhar o *A-Team* descrito acima terá ou um grande número de retângulos para indicar exatamente de onde cada Agente lê, ou simplificadamente, um único retângulo de onde todos Agentes lêem e onde todos podem escrever. A representação proposta para este *A-Team* está ilustrada na figura 4.15.

4.13 Sumário

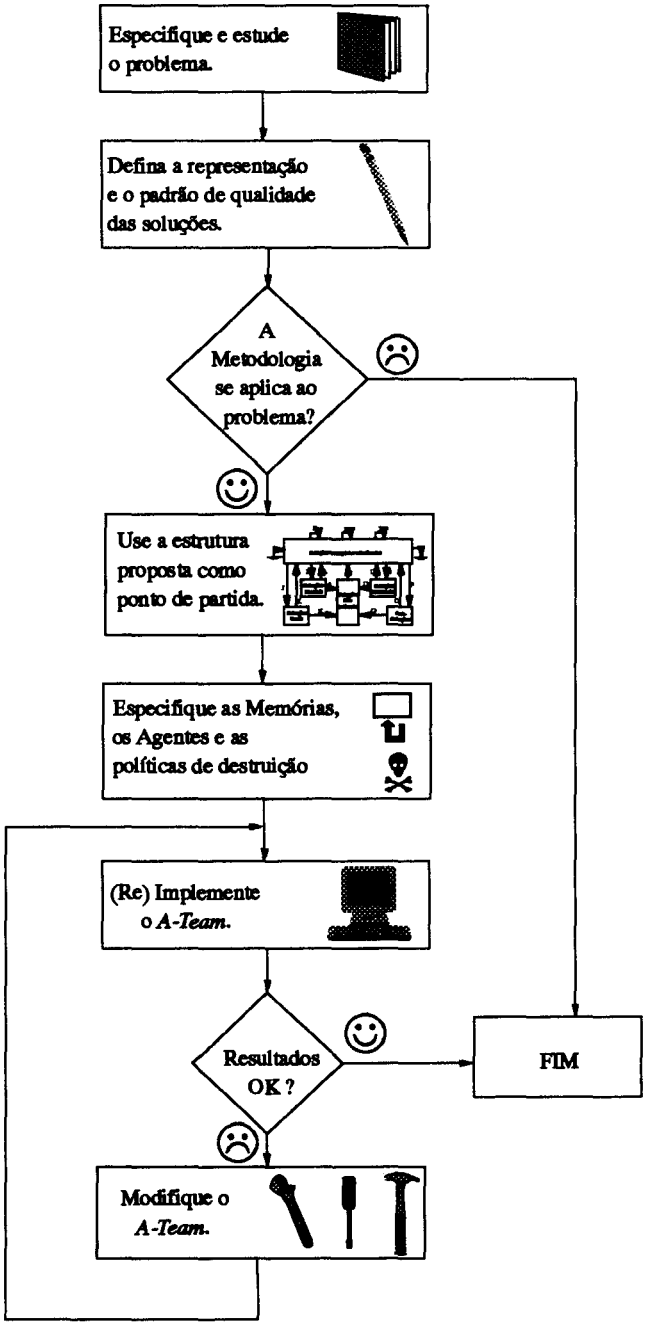
Este capítulo abordou uma Metodologia para especificação de Times Assíncronos para problemas de Otimização Combinatória, onde todos os parâmetros de projeto de um *A-Team* são detalhados. O primeiro passo na resolução de um problema por *A-Teams* é sua identificação, a definição do tipo de solução desejada como resposta e o nível de qualidade associado. Para fazer uso da Metodologia proposta, os diversos algoritmos existentes para resolver os problemas são classificados e posteriormente relacionados através de uma estrutura geral de *A-Teams*. Em seguida, a especificação das memórias compartilhadas e dos Agentes é apresentada detalhadamente. Embora os métodos de destruição sejam intimamente relacionados ao problema que se resolve, sugerem-se quatro políticas gerais de destruição. Finalmente, são dadas algumas sugestões para as modificações de *A-Teams*, sempre que os resultados produzidos não corresponderem aos esperados.

Figura 4.15 Exemplo de um *A-Team* com 5 Agentes e 4 memórias, segundo a representação proposta.



Esta Metodologia de especificação de Times Assíncronos para problemas de Otimização Combinatória pode ser resumida no algoritmo ilustrado na figura abaixo:

Figura 4.16 Algoritmo de construção de *A-Teams*, conforme a metodologia proposta.



4.14 Notas Bibliográficas

A classificação dos problemas mencionada na seção 4.4 também pode ser encontrada em [Sou93]. Parte da classificação dos algoritmos e heurísticas apresentada na seção 4.5 foi extraida dos trabalhos de Zanakis *et al.* [ZEV89, ZE81], Müller-Merbach [Mül81], Ball e Magazine [BM81], e Weiner [Wei75].

Certamente, os exemplos de trabalhos que utilizam *A-Teams* como técnica de resolução de problemas podem contribuir para a especificação de novos *A-Teams*. Os trabalhos de Souza [Sou93] e Murthy [Mur92] são exemplos de *A-Teams* que obtêm bons resultados para problemas complexos e de grande porte. Souza aplicou a técnica de *A-Teams* para resolver o PCV e Murthy utiliza *A-Teams* para projetar braços mecânicos de robôs, dada uma tarefa a executar.

Dentre outros trabalhos que utilizam esta técnica, podem ser citados os *A-Teams* para os problemas de escalonamento de tarefas *Job Shop Problem*, desenvolvidos por Cavalcante [Cav94] Haddad [Had94], *Flow Shop Problem* desenvolvido por este autor [PS94], o problema de Roteamento de Veículos por Glienke [Gli94] e o problema de Recobrimento de Conjuntos, por Longo [Lon94]. A abordagem de *A-Teams* para problemas dinâmicos e multi-objetivos está em estudo nos trabalhos de Camponogara [Cam94] e Rodrigues [Rod94], respectivamente.

Parte deste capítulo foi previamente publicada por este autor em [PS94b].

Capítulo 5

A-Teams para o Flow Shop Problem

Este capítulo apresenta a aplicação da metodologia proposta no capítulo anterior ao problema clássico de escalonamento de tarefas *Flow Shop Problem*. Primeiramente, como sugerido pela metodologia, especifica-se o problema em questão. Em seguida, define-se qual a representação de uma solução válida e o padrão de qualidade desejado. O próximo passo consiste no levantamento bibliográfico dos algoritmos e sua respectiva classificação. Alguns novos algoritmos são propostos para compor os *A-Teams*. São analisados também os parâmetros de projeto como: estrutura dos fluxos de dados, tamanho das memórias e políticas de destruição. Os resultados obtidos em suas versões concorrente e paralela são apresentados e comparados aos métodos responsáveis pelos melhores valores conhecidos.

5.1 Definição do Flow Shop Problem

O problema de alocar recursos em instantes determinados do tempo para que tarefas distintas possam ser executadas apresenta-se numa grande variedade de situações. Na maior parte dos casos, entretanto, *escalonar*[1] não é possível enquanto questões fundamentais de planejamento não sejam resolvidas. O planejamento é definido, geralmente, a partir de decisões gerenciais. Por exemplo, em aplicações de manufatura as questões fundamentais de planejamento consistem na escolha do produto a produzir e na respectiva escala de produção. Após estudos de marketing e análise econômica necessários para resolver essas questões, o passo seguinte é de cunho tecnológico: como estes produtos devem ser produzidos.

Em resumo, o planejamento está relacionado às seguintes perguntas:

- Que produto e/ou serviço será fornecido?
- Qual a demanda de mercado para curto, médio e longo prazo?
- Qual o montante de investimentos necessários?
- Que recursos estarão disponíveis?

enquanto escalonar responde às seguintes perguntas:

- Que recurso será alocado para cada tarefa? (Alocação de Recursos)
- Quando cada tarefa será executada? (Decisões de Seqüenciamento)

Neste contexto, vários problemas de escalonamento podem ser encontrados na literatura, conforme as restrições tecnológicas e os objetivos a serem alcançados. As restrições tecnológicas são determinadas, principalmente, pelo sentido de fluxo dos itens (produtos ou jobs) nas máquinas (recursos).

O problema de escalonamento escolhido para exemplificar o uso da metodologia proposta é caracterizado por dois elementos principais:

1. um sistema de produção composto por m máquinas;
2. um conjunto de n itens ou jobs, similares em sua natureza tecnológica, tal que todos possuem a mesma ordem de processamento nas máquinas, podendo o sistema de produção ser visto como um *pipeline*.

Este problema é conhecido como *Flow Shop Problem de Permutação* (FSP) e pode ser definido como segue: um conjunto de n jobs, cada job composto por m operações, deve ser processado na mesma seqüência nas m máquinas disponíveis, dado que o tempo de processamento t_{ij} do job i na máquina j é fixo, não negativo, e pode ser zero se o respectivo job não for processado na referida máquina. O objetivo é encontrar uma

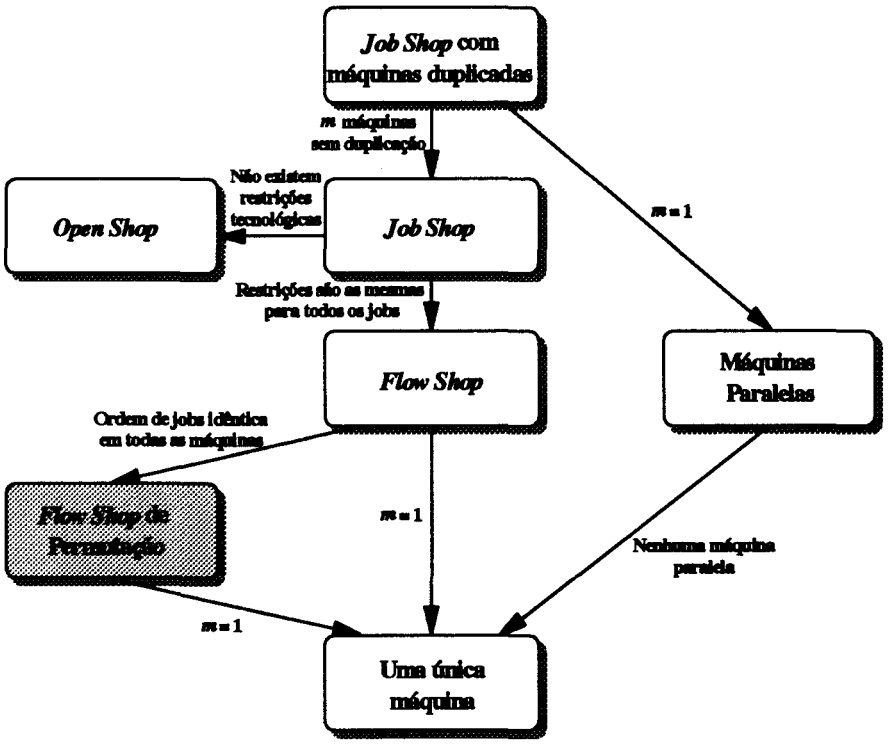
1. do inglês, *scheduling*, que significa alocar recursos no tempo para que sejam executadas tarefas.

seqüência de jobs em todas as máquinas que minimize o ***Makespan***, ou seja, o tempo entre o início do processamento do primeiro job, na primeira máquina, e o término do último job, na última máquina. Esse critério de otimização, além de minimizar o tempo máximo de processamento do conjunto de jobs, maximiza, também, a utilização das máquinas [Bak74]. A figura 5.1 ilustra a relação do FSP frente aos demais problemas clássicos de escalonamento, conforme suas respectivas restrições tecnológicas.

Considera-se, ainda, que:

- as máquinas estão sempre disponíveis e nunca quebram;
- cada máquina pode processar no máximo um job a cada instante;
- cada job pode ser processado no máximo em uma máquina a cada instante;
- todos os jobs estão disponíveis no começo do processamento;
- não é permitida preempção - uma vez que um job começe numa máquina ele lá permanece até ser completado;
- tempos de *setup* são independentes do escalonamento e são inclusos nos tempos de processamento;
- o tempo de processamento e as restrições tecnológicas são determinísticas e conhecidas a priori.

Figura 5.1 Relação entre o *Flow Shop Problem* de permutação (FSP) e os demais problemas clássicos de escalonamento.



O número de soluções válidas para o FSP é $n!$, uma vez que qualquer permutação de jobs constitui uma solução válida.

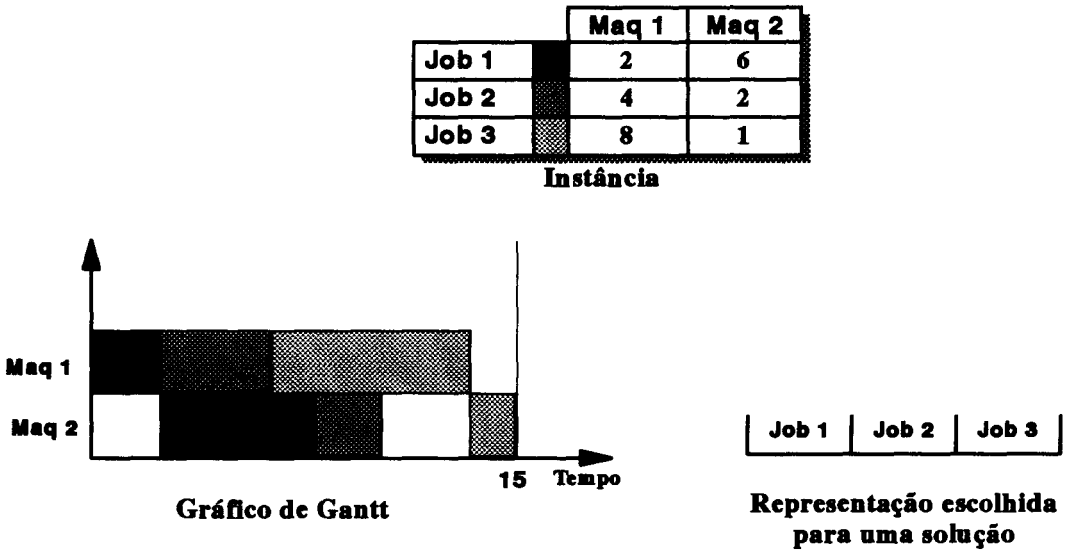
Devido às características mencionadas, uma seqüência de n números, cada número representando um job, é perfeitamente adequada e compacta para representar uma solução válida. A figura 5.2 exemplifica uma instância de um FSP de 2 máquinas e 3 jobs, bem como a representação de uma solução pelo gráfico de Gantt e uma seqüência de jobs.

5.2 Complexidade do FSP

A teoria da complexidade objetiva estabelecer diferenças entre os problemas tratáveis e intratáveis [GJ79]. Esta teoria classifica os problemas conforme as computações necessárias para resolvê-lo em computadores digitais. Parte desta teoria foi apresentada no capítulo 2.

O FSP, como muitos outros problemas de otimização combinatória, pertence à classe dos problemas NP-Difícieis, como foi demonstrado em [GJS76]. Desse modo, é pouco provável a existência de algoritmos polinomiais que resolvam quaisquer de suas instâncias. Entretanto, para instâncias onde o número de máquinas se limita a apenas duas, e um caso particular de três, o problema pode ser resolvido em $O(n\log n)$ pelo algoritmo proposto por Johnson [Bak74].

Figura 5.2 Exemplo de uma instância do FSP de três jobs e duas máquinas. O gráfico de Gantt mostrado na figura corresponde à representação gráfica da solução (1,2,3).



5.3 Adequabilidade do FSP a A-Teams

Neste trabalho estamos interessados em obter as melhores soluções possíveis, independente da instância que se deseja resolver. A revisão bibliográfica deste problema apontou a não existência de algoritmos que produzam rapidamente as soluções ótimas. Sendo assim, apenas Algoritmos Inadequados encontram-se disponíveis (veja seção 4.4), e são, em sua maioria, heurísticas. Dentre as heurísticas encontradas, algumas são de melhoria [Pag61, WH89, OS90, Tai90], o que indica que o FSP possui ciclos no diagrama de decomposição de problemas. Desse modo, podemos concluir que o FSP é um PMA (veja seção 4.4) e potencialmente adequado a ser resolvido por *A-Teams*.

5.4 Algoritmos para o FSP

Os algoritmos para FSP também podem ser classificados em duas classes majoritárias: algoritmos exatos e algoritmos aproximados. Os algoritmos exatos, embora garantam soluções ótimas, levam tempo exponencial em relação ao tamanho da entrada. Os aproximados, embora sejam rápidos, não garantem a otimalidade das soluções produzidas.

Os métodos exatos encontrados na literatura são, geralmente, algoritmos de *Branch-and-Bound*, métodos de eliminação e formulação inteira [ML93]. Neste trabalho estamos interessados apenas em algoritmos aproximados, no intuito de usá-los na especificação de Times Assíncronos. As próximas duas sub-seções descrevem as heurísticas que serão utilizadas na especificação dos Times Assíncronos para o FSP.

5.4.1 Heurísticas baseadas em métodos de ordenação

E. S. Page, em 1961, sugeria que problemas de escalonamento fossem tratados como problemas de ordenação de dados [Pag61]. Page desenvolveu algumas heurísticas de construção baseando-se em procedimentos de ordenação como o *MergSort* e *Buble-Sort* [CLR90].

Merging (M)

Como no método de ordenação *MergSort*, este algoritmo parte de uma seqüência qualquer de jobs criando, nos passos intermediários, várias seqüências parciais que são consideradas "ordenadas". Primeiramente são "ordenados" pares de jobs adjacentes. Em seguida, são "ordenados" quartetos, octetos, e assim sucessivamente, até que uma nova seqüência de jobs seja obtida. A cada estágio, é necessário considerar apenas a ordem relativa dos jobs que lideram cada uma das seqüências envolvidas. Um job i é considerado predescessor de outro job j se sua inserção na seqüência parcial causa um menor *Makespan*. Em verdade, este é um algoritmo de melhoria. Porém, neste trabalho, simplificamos o algoritmo proposto por Page para desempenhar apenas o último estágio, ou seja, a partir de duas seqüências de $n/2$ jobs, construímos uma terceira (e final) que contém todos os jobs. Desse modo esse

algoritmo desempenha uma função de composição entre duas seqüências parciais. Sua complexidade é $O(n^2m)$. A figura 5.3 ilustra uma iteração desse algoritmo.

Individual Exchange (IE)

Esta heurística de melhoria é semelhante ao algoritmo de ordenação *BubleSort* [CLR90]. Partindo de um escalonamento qualquer e do primeiro para o último job, este algoritmo efetua trocas de jobs adjacentes na busca de soluções com menores *Makespans*. As trocas continuam até que todos os pares adjacentes da solução corrente permaneçam na mesma posição por um ciclo inteiro. A complexidade deste algoritmo não é clara, uma vez que não se sabe a priori quantas trocas serão efetuadas. Na prática, porém, poucas trocas são efetuadas. A figura 5.4 mostra uma seqüência de duas iterações do algoritmo IE.

Group Exchange (GE)

Esta também é uma heurística de melhoria que permuta grupos de jobs em vez de jobs isolados, tal como o *Individual Exchange*. Inicialmente permutam-se dois grupos, o grupo formado pelos $n/2$ primeiros elementos de uma seqüência e o grupo dos $n/2$ jobs restantes. Em seguida, grupos adjacentes de tamanho $n/4$, $n/8$, ... e, finalmente, grupos unitários são permutados. O tamanho dos grupos só é reduzido quando durante todo um ciclo não for possível melhorar a solução corrente. Dessa forma, o último estágio (trocas de grupos unitários) do algoritmo GE corresponde ao algoritmo IE. A complexidade desse algoritmo também não pode ser expressa precisamente.

Figura 5.3 Exemplo de uma iteração do algoritmo M sobre as seqüências S1 e S2. A primeira posição da seqüência S3 é preenchida com o job 3 pois, neste exemplo hipotético, o job 3 na primeira posição causa um menor *Makespan* que o job 2.

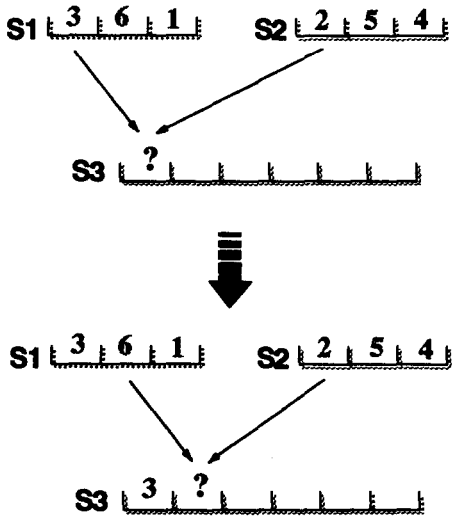
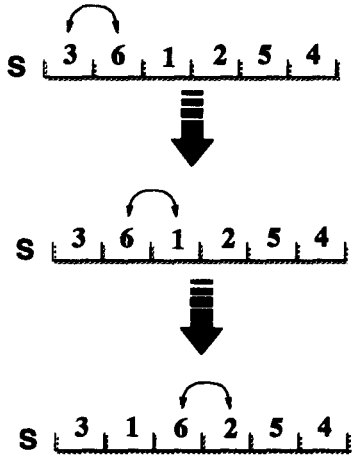


Figura 5.4 Exemplo de duas iterações do algoritmo IE. Na primeira iteração verifica-se que a permutação entre os jobs da posição um e dois não reduz o *Makespan*. Neste caso a ordem permanece a mesma. Na segunda iteração permutam-se com sucesso os jobs das posições 2 e 3, obtendo uma nova seqüência de menor *Makespan*.



5.4.2 Heurística de Nawas, Enscore e Ham (NEH)

Em 1983, Nawaz, Enscore e Ham propuseram uma heurística determinística de construção para o *Flow Shop Problem* de permutação [NEH83] . Eles consideram que o job que possui o maior tempo total de processamento deva receber maior atenção que um job com menor tempo de processamento. Sendo assim, inicialmente, apenas os dois jobs de maior tempo total de processamento são selecionados e, por busca exaustiva, a melhor seqüência parcial com estes dois jobs é encontrada (ou seja, tentando as duas combinações possíveis). Em seguida, o terceiro job com maior tempo total de processamento é escolhido e inserido em todas as possíveis posições da seqüência parcial. A seqüência em que este último job inserido produzir o menor *Makespan* é escolhida como sendo a nova seqüência parcial. Esse processo se repete até que não reste nenhum job a escalonar.

Esta heurística tem sido aplicada com sucesso ao *Flow Shop Problem* na otimização de outros critérios que não o *Makespan*. Particularmente, sobre a minimização do *Makespan*, NEH é considerada a melhor heurística polinomial determinística, como foi observado empiricamente por Park *et al.* [PPE84], Widmer *et al.* [WH89] e Taillard [Tai90].

A heurística de construção NEH inspirou a criação de duas novas heurísticas, uma de melhoria determinística (MNEH) e outra de construção não determinística (Cheap-NEH). Dado uma seqüência, a heurística MNEH produz uma nova solução em que os jobs são re-escalonados a partir do que possui maior tempo total de processamento

para o menor. Basicamente, retira-se cada job da posição que ele ocupa e verifica-se se sua inserção em alguma das outras (n-1) posições reduz o *Makespan*. Este algoritmo, de complexidade $O(n^3m)$[1], pode ser especificado como segue:

Algoritmo MNEH

Ordene os n jobs em V[n] em ordem decrescente de tempo total de processamento
Para k=1 até n **faça**
  Retire o job V[k] da seqüência atual
  Insira o job V[k], entre as n possibilidades, na posição que minimize
    o *Makespan*
FIM

Embora a literatura para o FSP seja farta em algoritmos de construção, verificamos a ausência de heurísticas de construção não determinística que partam de uma seqüência parcial não vazia. Geralmente, as heurísticas utilizam os tempos de processamento dos jobs para gerar prioridades de escalonamento, sendo, portanto, determinísticas. Deste modo, desenvolvemos uma heurística de construção que, dada uma seqüência parcial vazia ou não, constrói uma seqüência completa. A Cheap-NEH constrói, primeiramente, um conjunto C de jobs não escalonados (não presentes na seqüência parcial). Em seguida, verifica-se se a primeira posição da seqüência parcial já possui algum job escalonado. Caso de não o possuir, a heurística escolhe ao acaso um job do conjunto C e o atribui à posição que produzir o menor *Makespan* (inicialmente só existe uma posição). Caso contrário, a heurística procura a próxima posição vazia da seqüência para reiniciar este processo. Como os jobs são escolhidos ao acaso, esta heurística é não determinística. Se a seqüência inicial for vazia, esta heurística efetua o mesmo número de computações que NEH. Porém, se a posição k da seqüência parcial estiver preenchida com algum job, $O(knm)$ operações são poupadas. O próximo algoritmo descreve esta heurística.

Algoritmo Cheap-NEH

Construa um conjunto C com os jobs não escalonados na seqüência parcial
Para k=1 até n **faça**
  Se posição k da seqüência parcial não possui job
    Então Retire ao acaso um job i em C
      Insira o job i, entre as k possibilidades, na posição que minimize
        o *Makespan*
FIM

---

1. Taillard [Tai90] menciona um algoritmo não publicado de Th. Mohr que permite a redução da complexidade da heurística NEH para $O(n^2m)$.

5.5 Estruturas de A-Teams para o FSP

As heurísticas apresentadas na seção anterior podem ser combinadas de diversas maneiras a fim de compor um *A-Team*. Essas heurísticas foram classificadas como segue:

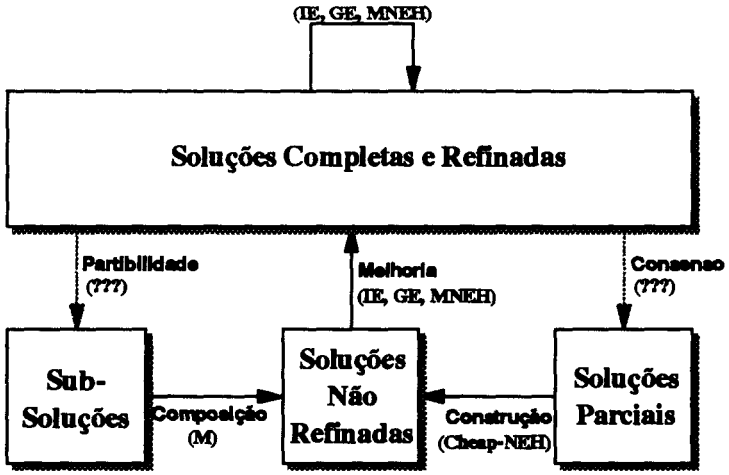
- Três heurísticas de melhoria (IE, GE, MNEH);
- Uma heurística de construção (Cheap-NEH);
- Uma heurística de composição (M).

De posse dessa classificação e da estrutura geral apresentada no capítulo 4, observa-se que estruturas gerais de *A-Teams* como a ilustrada na figura 5.5 podem ser geradas para o FSP. Entretanto, para usar as heurísticas M e Cheap-NEH, é necessário desenvolver algoritmos de Partibilidade e Consenso, visto que apenas as heurísticas de melhoria formam ciclos.

Algoritmo de Partibilidade (P)

Ao analisar as soluções fornecidas por algumas heurísticas da literatura [Pal65, CDS70, Gup71, NEH83], observa-se que vários jobs ocupam as mesmas posições relativas em cada solução. Portanto, alguns jobs são escalonados antes de outros, independentemente da heurística utilizada. Tendo em vista esse fato, propôs-se um algoritmo para particionar soluções em duas sub-seqüências (ou sub-soluções): uma sub-seqüência em que os jobs são escalonados nas ($n/2$) primeiras posições (jobs de maior prioridade) e uma outra em que os jobs permanecem nas posições

Figura 5.5 Estrutura geral de *A-Teams* para o FSP composta pelas heurísticas IE, GE, M, MNEH e Cheap-NEH. As setas pontilhadas indicam a ausência de algoritmos para a formação de ciclos.



finais (jobs de menor prioridade). Em cada uma dessas sub-seqüências, se o job *i* precede o job *j*, então o job *i*, nas soluções consultadas, foi considerado prioritário no escalonamento em relação ao job *j*. Consequentemente, o último job da primeira sub-seqüência possui "precedência" maior ou igual ao primeiro job da segunda sub-seqüência. Este algoritmo consulta quatro soluções quaisquer da memória de Soluções Completas e Refinadas para produzir um par de sub-seqüências.

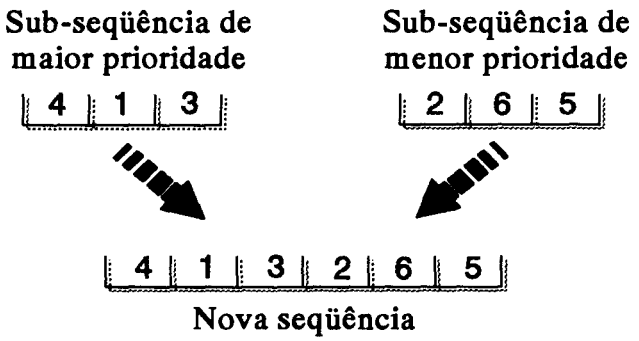
Algoritmo de Consenso (C)

Assim como o algoritmo de Partibilidade tira proveito das soluções produzidas por outros algoritmos, este algoritmo explora o consenso por intersecção de soluções. O algoritmo Consenso consulta duas soluções da Memória de Soluções Completas e Refinadas, verifica quais as posições que possuem jobs em comum e então cria uma nova solução com esses jobs nas respectivas posições (veja figura 4.3 no capítulo 4). A idéia de que faz uso este algoritmo é a de tentar capturar o que as boas soluções possuem em comum. As soluções criadas por este algoritmo servirão de ponto de partida para o algoritmo de construção Cheap-NEH.

Algoritmo Junção (J)

Embora já exista um algoritmo de Composição, o algoritmo M, construiu-se um outro algoritmo muito simples que visa à construção de uma nova seqüência a partir de duas sub-seqüências parciais, as quais podem ser consideradas como sugestões de prioridades de escalonamento. O algoritmo Junção simplesmente considera a sub-seqüência de menor prioridade como sendo continuação da sub-seqüência de maior prioridade. As seqüências obtidas por este algoritmo refletem como os jobs estão escalonados nas soluções da Memoria de Soluções Completas e Refinadas. A figura 5.6 ilustra de que maneira o algoritmo J constrói uma nova seqüência a partir de duas sub-seqüências.

Figura 5.6 Exemplo do algoritmo de Junção sobre um FSP de seis jobs. A partir de duas sub-seqüências, constrói-se uma nova seqüência através da juncão de seus jobs.



5.5.1 Estrutura de Fluxo de Dados utilizada para o FSP

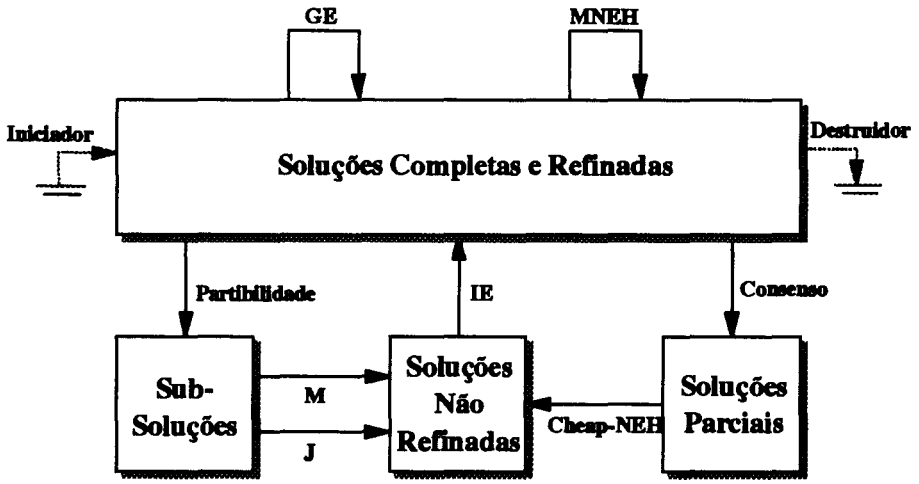
Dos algoritmos descritos até o momento, apenas os de melhoria possuem a opção de ocupar mais de uma das setas da estrutura da figura 5.5. Nesse caso, a questão que se coloca é: que algoritmo alocar e em qual posição? A definição do fluxo de dados constitui uma tarefa importante, visto que a combinação dos algoritmos pode influenciar tanto na exploração das soluções quanto na obtenção de bons resultados.

O algoritmo que vier a ocupar a seta que liga a Memória de Soluções Não Refinadas à Memória de Solucões Completas e Refinadas precisa ser um algoritmo rápido, pois, caso contrário, cria-se um gargalo no refinamento dessas soluções. Das três heurísticas de melhoria disponíveis (IE, GE e MNEH), IE é a mais rápida e simples. Sendo assim, decidimos alocar esta heurística nessa posição. Além do mais, uma solução melhorada pela heurística GE não pode ser melhorada pela IE, uma vez que GE, no último passo, efetua as mesmas operações que IE. A heurística MNEH é muito lenta, o que inviabiliza sua escolha.

As outras duas heurísticas de melhoria, GE e MNEH, são alocadas na melhoria de soluções da Memória de Soluções Completas e Refinadas. Deste modo, o *A-Team* obtido para o FSP possui a estrutura de fluxo de dados mostrada na figura 5.7.

Neste *A-Team* preliminar ainda não estão especificados os agentes Destruidor e Iniciador, cujos algoritmos dependem das políticas adotadas de iniciação e destruição.

Figura 5.7 Estrutura de um *A-Team* para o FSP. Os agentes GE e MNEH leêm e escrevem na Memória de Soluções Completas e Refinadas. Partibilidade e Consenso leêm de Soluções Completas e Refinadas e escrevem, respectivamente, em Sub-Soluções e Soluções Parciais. M e J leêm de Sub-Soluções e escrevem em Soluções Não Refinadas, assim como Cheap-NEH que lê de Soluções Parciais. O agente IE lê de Soluções Não Refinadas e escreve em Soluções Completas e Refinadas. As setas pontilhadas indicam os agentes Destruidor e Iniciador, cujos algoritmos são dependentes das políticas adotadas.



5.6 As Memórias

Conforme sugerido pela metodologia apresentada no capítulo anterior, apenas a Memória de Soluções Completas e Refinadas possui um agente Destruidor. Isto decorre do fato de as demais memórias serem simples o bastante, a ponto de funcionarem como *buffers* entre os algoritmos que depositam e retiram dados. Nesse caso, o tamanho destas memórias não é relevante à performance dos *A-Teams*, pois todos os dados nelas depositados são processados num tempo finito.

O tamanho da memória de Soluções Completas e Refinadas, como veremos posteriormente, foi determinado empiricamente. Nos testes desempenhados, $2n$ soluções foram suficientes para produzirem ótimos resultados.

Antes dos agentes entrarem em operação, é necessário iniciar a Memória de Soluções Completas e Refinadas. No caso particular do FSP, não encontramos na literatura algoritmos rápidos capazes de produzirem centenas de soluções distintas de boa qualidade. Desse modo, optou-se por iniciar toda a memória de Soluções Completas e Refinadas com soluções geradas aleatoriamente. Embora não seja um método que produza boas soluções, a simplicidade de implementação, associada à rapidez e à grande diversidade, justificaram a escolha.

5.7 Os Agentes

Para cada algoritmo alocado no *A-Team* da figura 5.7, é preciso especificar como o agente por ele representado efetua a leitura nas memórias. Para os agentes que leêm seus dados das Memórias de Sub-Soluções, Soluções Não Refinadas ou Soluções Parciais, é indiferente qual dado é escolhido a cada instante. Sendo assim, utilizamos a política de leitura FIFO. Entretanto, para os agentes que leêm da memória de Soluções Completas e Refinadas, a política de escolha é um fator importante, visto que pode evitar esforço desnecessário.

Nos testes desempenhados isoladamente, o agente de melhoria MNEH apresentou melhores resultados frente às demais heurísticas de melhoria. Entretanto, a sua granularidade é a mais grossa. Para que o *A-Team* produza rapidamente bons resultados, a partir das soluções presentes na memória, a política de leitura utilizada para o MNEH é a "Da melhor para a pior", ou seja, dá-se primeiramente às melhores soluções da memória a chance de serem melhores ainda. Já o agente GE escolhe uma solução aleatoriamente, visto que ele é de granularidade mais fina que MNEH e pode, num mesmo intervalo de tempo, "melhorar" mais soluções que MNEH. Esta política foi escolhida para que todas as soluções tenham a oportunidade de serem aprimoradas, e não apenas as melhores.

Os algoritmos Partibilidade e Consenso escolhem aleatória e respectivamente, quatro e duas soluções da memória de Soluções Completas e Refinadas.

5.8 Métodos de Destruição

A escolha da política de destruição também é um fator empírico. Neste trabalho decidiu-se experimentar seis políticas distintas de destruição, todas elas relacionadas ao *Makespan*, para que a mais adequada ao FSP fosse identificada. As políticas são:

D1 - Destruição da pior solução e aceitação de novas apenas no caso de serem de melhor qualidade que a pior presente na memória;

D2 - Destruição de soluções com distribuição linear de probabilidade e aceitação de novas apenas se forem de melhor qualidade que a pior presente na memória;

D3 - Destruição de soluções com distribuição linear de probabilidade e aceitação de qualquer nova solução, independente de sua qualidade;

D4 - Destruição de qualquer solução com distribuição uniforme de probabilidade e aceitação de novas apenas se forem de melhor qualidade que a pior presente na memória;

D5 - Destruição de qualquer solução com distribuição uniforme de probabilidade e aceitação de qualquer nova solução, independente de sua qualidade;

A seção dos resultados computacionais apresenta comparativamente os desempenhos de cada uma delas.

5.9 Implementação

Devido ao fato de os *A-Teams* serem adequados ao processamento paralelo e distribuído, decidiu-se implementá-los numa estrutura que permita a exploração de tais virtudes. Utilizou-se uma estrutura cliente/servidor, onde os agentes desempenham o papel de clientes e as memórias o de servidores de dados. Essa escolha decorreu das seguintes vantagens:

1. Chamadas de procedimentos remotos são adequados ao processamento distribuído;
2. Não existe limite ou restrição no número de clientes ou servidores usados para implementar um *A-Team*;
3. Os servidores processam as requisições dos clientes segundo a política FIFO, serializando as requisições e, conseqüentemente, evitando colisão de acesso aos dados depositados nas memórias compartilhadas (garantia de integridade de dados por serialização de requisições).
4. Devido ao fato de os clientes e servidores serem implementados como processos individuais, eles podem ser executados em qualquer máquina, independentemente de qualquer alteração de código;
5. Fácil implementação através da ferramenta DPSK+P [Car87, CS92].

DPSK+P é uma ferramenta desenvolvida por Cardozo [Car87] para oferecer facilidades de programação e suporte a aplicações de Sistemas Multi-Agentes (MAS, do inglês, *Multi-Agent Systems*). Esta ferramenta possui três componentes [CS92]:

1. um *kernel* responsável pelas funções de processamento distribuído (inter-comunicação de máquinas, manutenção de dados compartilhados, gerenciamento de processos etc..);
2. uma interface para conectar a aplicação do usuário à ferramenta (conjunto de classes e métodos em C++ e CLOS);
3. e um conjunto de facilidades para gerenciar e inspecionar os objetos ativos.

Embora este mecanismo de comunicação possa ter alguma sobrecarga de comunicação, na prática, o tempo efetivamente gasto em comunicação é insignificante, frente à granularidade grossa dos algoritmos.

5.10 Instâncias de Testes

Taillard publicou recentemente várias instâncias de *benchmarks* para problemas de seqüenciamento de tarefas, em particular para o FSP [Tai93]. As instâncias geradas por Taillard foram obtidas a partir de um gerador de números aleatórios, para o qual é necessário um número semente (ou inicial). Taillard produziu centenas de instâncias variando de poucos jobs (20 jobs e 5 máquinas, ou 20x5) até várias centenas deles (500 jobs e 20 máquinas, ou 500x20). Para cada tamanho de instância, Taillard selecionou as 10 mais "difíceis" e publicou o número semente que as caracteriza, juntamente com um limite inferior e superior. O limite superior (melhor solução conhecida) apresentado foi obtido através da resolução do problema por *Tabu Search* com um grande número de iterações. As instâncias consideradas difíceis eram as que apresentavam maior distância entre os limites superior e inferior, bem como o fato de que várias tentativas de resolvê-las a partir de soluções iniciais diferentes conduziam a soluções com *Makespan* distintos. Em seu trabalho, Taillard não apresenta nenhuma solução ótima para o FSP. Entretanto, durante o curso deste trabalho desenvolveu-se novas e melhores fórmulas de calcular limites inferiores (veja capítulo 6), demostrando, assim, a otimalidade de duas das instâncias publicadas. Nestes dois casos, o valor obtido pela nova fórmula de cálculo dos limites inferiores coincide com os apresentados para os superiores.

Para efeito de avaliação e comparação dos resultados produzidos pelos *A-Teams*, escolheu-se, aleatorimente, quatro instâncias de tamanhos variados (pequeno, médio e grande) e as duas que agora, sabidamente, possuem soluções ótimas. As seis instâncias escolhidas estão descritas na tabela abaixo conforme o seu tamanho, número semente e limite superior. Na próxima seção nos referiremos às instâncias apenas pelo seu tamanho, ficando subtendido o número semente a ela associado.

Tabela 5.2 Instâncias utilizadas para avaliação e comparação dos resultados obtidos pelos *A-Teams*. A primeira coluna indica o tamanho da instância. A segunda apresenta a semente utilizada para gerá-las, e a terceira o valor do melhor *Makespan* conhecido.

Tamanho	Semente	Melhor *Makespan* Conhecido
20x5	873654221	1278
20x10	587595453	1582
50x5(a)	1328042058	2724[a]
50x5(b)	248421594	2782[a]
100x10	1539989115	5776
500x20	1368624604	26316

a. ótimo

5.11 Testes Computacionais

Nesta seção descreveremos diversos testes realizados nos *A-Teams* para o FSP. Primeiramente, mencionaremos o tamanho adotado para a memória de Soluções Completas e Refinadas em conseqüência dos testes preliminares. Em seguida, apresentamos os resultados obtidos com as políticas de destruição (D1, D2, D3, D4 e D5), as diversas configurações de fluxo de dados testadas e os resultados obtidos do processamento distribuído. Exceto quando especificado em contrário, a instância de teste mencionada nas próximas sub-seções se refere à 100x10.

5.11.1 O tamanho das memórias

Após a adoção do fluxo de dados apresentado na figura 5.7, implementou-se, primeiramente, um *A-Team* através de simulação de concorrência. Por simulação de concorrência entende-se uma versão seqüencial em que a ordem de execução dos agentes é determinada por sorteio. Nos testes preliminares desenvolvidos nesta versão, os melhores resultados foram obtidos com a memória de Soluções Completas e Refinadas contendo $2n$ soluções, onde n é o número de jobs da instância. Assim, adotamos este tamanho como sendo suficiente para os demais testes, inclusive para a versão distribuída.

5.11.2 As políticas de destruição

Das políticas de destruição experimentadas, D1 induziu a uma rápida e brusca redução da diversidade entre as soluções presentes na memória. Eliminando sempre a pior, observamos que a partir de um determinado instante apenas as soluções produzidas pelo agente MNEH eram aceitas. Desse modo, todos os demais agentes produziam inutilmente centenas de soluções, algumas potencialmente boas, mas não suficientemente boas para entrarem na memória. A figura 5.8 (a) mostra uma curva típica da

rápida convergência da pior solução para a melhor. Observe que após duas horas de processamento a memória estacionou definitivamente em alguns mínimos locais, permanecendo assim por todo o tempo de processamento permitido (mais de 12 horas). Os testes com esta política não atingiram o melhor valor conhecido (para as instâncias 100x10 e 500x20)

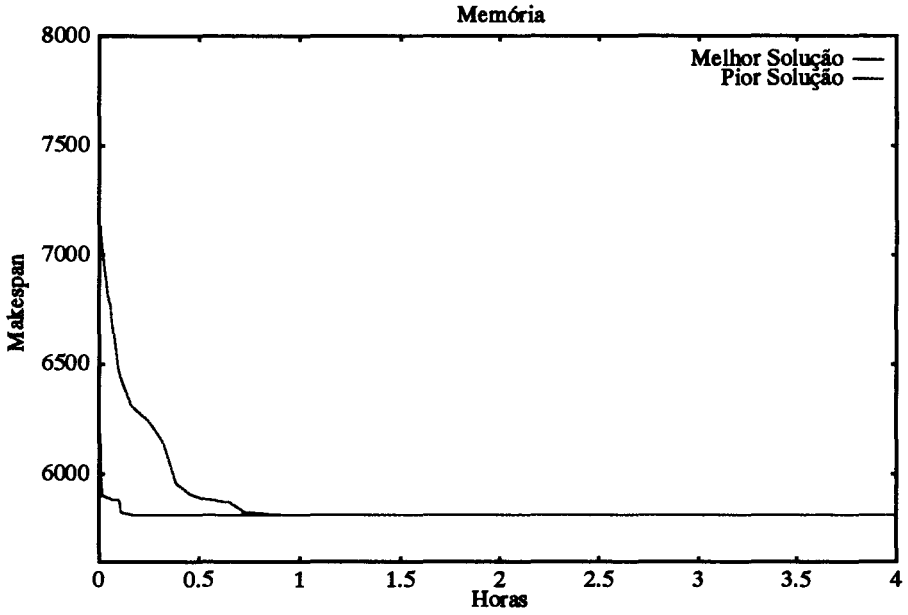
Para o FSP, a política de destruição D2 posterga a convergência da memória, mas não evita o estacionamento em mínimos locais. Embora esta política evite que várias soluções similares estejam na memória, em todas as execuções não se conseguiu atingir o melhor valor conhecido (instâncias 100x10 e 500x10). Acredita-se que ainda assim não se atingiu a diversidade necessária para produzir soluções de outros mínimos locais. A figura 5.8 (b) descreve o comportamento da pior e melhor soluções da memória sob a política D2. Veja a semelhança entre os gráficos da figura 5.8 (a) e (b).

Para aumentar ainda mais a diversidade da memória de Soluções Conpletas e Refinadas, decidiu-se alterar D2 de modo que qualquer solução fosse aceita, mesmo que ela viesse a ser pior que todas as demais da memória. Esta política denominada D3 não apresenta a convergência entre a pior e a melhor, uma vez que qualquer solução é aceita. Entretanto, esta política permitiu ao *A-Team* encontrar soluções de melhores *Makespan* que as encontradas pelos *A-Teams* usando D1 e D2. A relação entre a melhor e pior soluções da memória para o *A-Team* com a política de destruição D3 está ilustrada na figura 5.9. O fato de D3 permitir a entrada de qualquer solução e, ainda assim, viabilizar o *A-Team* encontrar soluções de melhores *Makespan*, sugeriu que o *Flow Shop Problem* necessita de grande diversidade e que, soluções aparentemente ruins, podem ser combinadas para produzirem soluções de alta qualidade. Experimentou-se, então, utilizar a política D4, em que todas as soluções possuem a mesma probabilidde de serem eliminadas, exceto a primeira (melhor). Embora a taxa de convergência seja mais lenta que a de D2, esta política de destruição não foi capaz de produzir soluções tão boas quanto as melhores conhecidas. A figura 5.10 (a) e (b) mostra o comportamento da melhor e da pior solução ao longo do tempo em duas janelas de tempo, quatro e quarenta horas de processamento. Observe que mesmo após muito tempo de processamento, não houve a convergência total como em D1 e D2. Entretanto, o valor da pior solução limitou a entrada de soluções que permitissem a exploração de outros mínimos locais. D4 conduziu a *Makespans* semelhantes aos obtidos por D1 e D2. Entretanto, D3 produziu melhores resultados que D4.

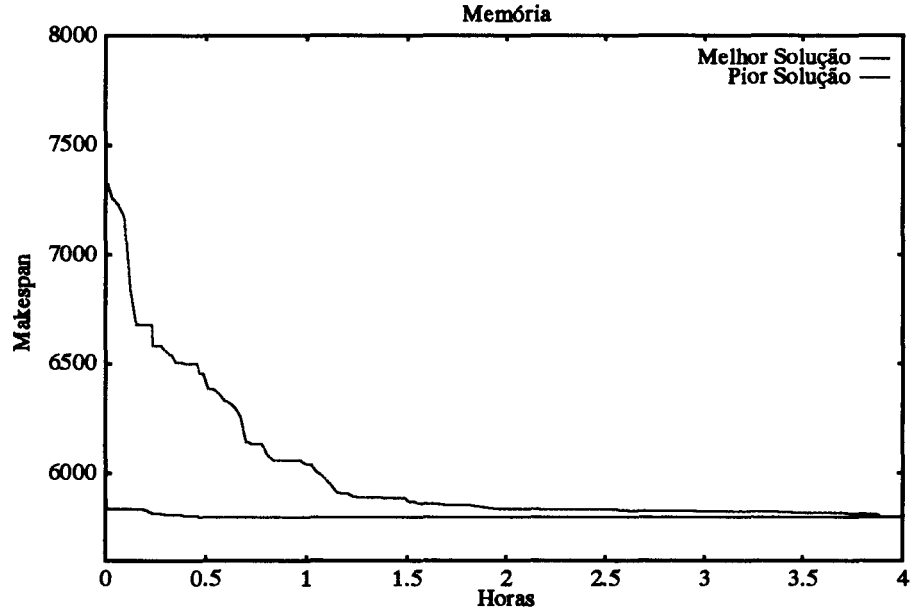
Assim como em D3 a aceitação de qualquer solução na memória, independentemente de seu *Makespan*, produziu boas soluções, a política D5 apresentou os melhores resultados (para todas as instâncias). Conjectura-se, que tal fato deve-se à necessidade de grande diversidade na memória. Os *A-Teams* com a política de destruição D5 encontraram soluções tão boas e até mesmo melhores que as publicadas como as melhores soluções conhecidas. As figuras 5.11 (a) e (b) mostram o compartamento da melhor e pior soluções ao longo do tempo em duas janelas de tempo, quatro e trinta e cinco horas de processamento. Observe pelos gráficos que D1 converge mais rápido que D2, que por sua vez converge mais rápido que D3, enquanto que D4 e D5 não apresentam convergência.

Figura 5.8 Comportamento da melhor e pior soluções na memória de Soluções Completas e Refinadas do *A-Team* resolvendo FSP de 100x10, utilizando o fluxo de dados apresentado na figura 5.7 e as políticas de destruição D1 (a) e D2 (b). Após uma e quatro horas de procesamento, as curvas não apresentaram nenhuma alteração.

(a)



(b)



5.11.3 *Eficiência em Escala*

O conceito de Eficiência em Escala foi introduzido recentemente por Talukdar e Souza [TS92] e pode ser compreendido da seguinte maneira. Considere-se uma organização como uma estrutura que contém *slots* (aberturas) para agentes. Suponha que p seja qualquer medida de performance dessa organização relacionada com os resultados produzidos por seus agentes. A organização compõe, assim, um super-agente. Seja A um conjunto de agentes que podem compor uma organização (super-agente). Então, uma organização é dita *Aberta* se ela possui, ou pode automaticamente produzir, um *slot* para cada agente de A. Uma organização é dita *Efetiva* sobre A e p se ela permite, através da adição e cooperação dos membros de A, melhora nos valores de p. Mais especificamente, existe pelo menos uma ordem na adição dos membros de A à organização que produz uma melhora monotônica no valor de p. Deste modo, uma organização é dita *Eficiente em Escala* se ela for aberta a A e efetiva sobre A e p [TS92, Sou93].

Antes de verificarmos se os *A-Teams* para o FSP são Eficientes em Escala, vamos apresentar os resultados por eles obtidos através do uso de apenas um ciclo, uma vez que sem ciclos não se produzem soluções válidas. Os quatro *A-Teams* possíveis são os ilustrados na figura 5.12. Devido aos testes efetuados até este momento, escolhemos D5 como a política de destruição e $2n$ como o tamanho máximo da memória de Soluções Completas e Refinadas. A execução destes algoritmos isoladamente não foi

Figura 5.9 Comportamento da melhor e pior soluções na memória de Soluções Completas e Refinadas do *A-Team* resolvendo FSP de 100x10, utilizando o fluxo de dados apresentado na figura 5.7 e a política de destruição D3. Observe que não há restrição no valor da pior solução.

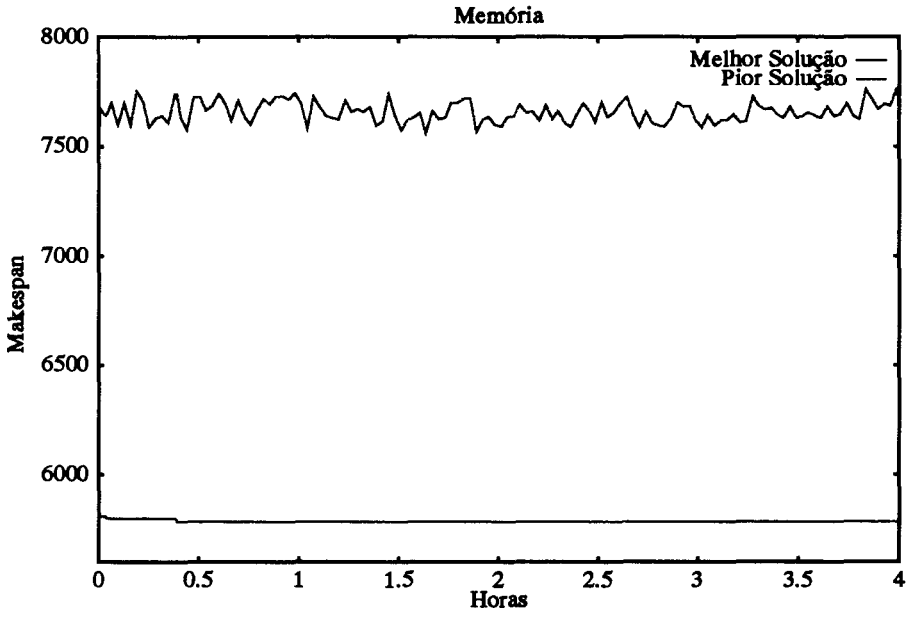
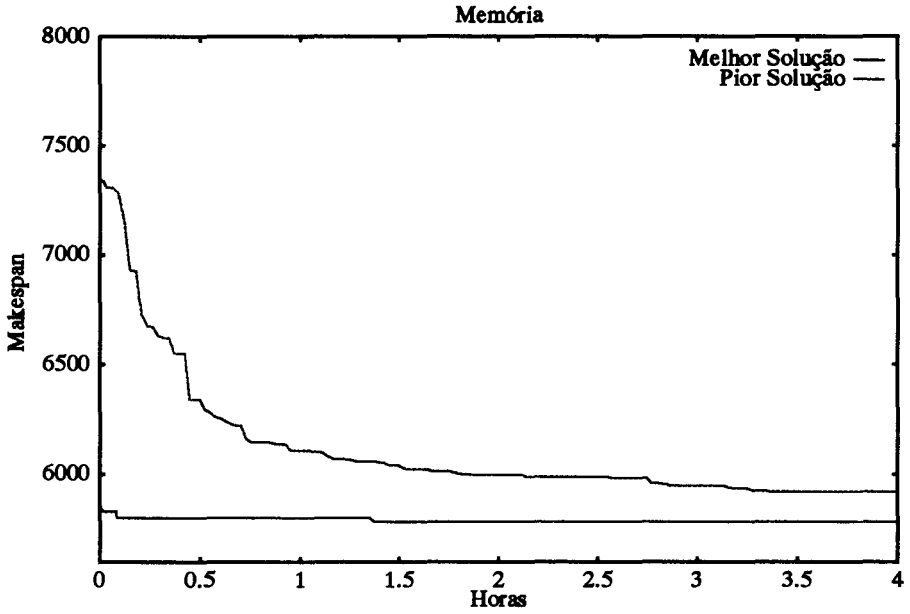


Figura 5.10 Comportamento da melhor e pior soluções na memória de Soluções Completas e Refinadas do *A-Team* resolvendo FSP de 100x10, utilizando o fluxo de dados apresentado na figura 5.7 e a política de destruição D4. A figura (a) e (b) correspondem, respectivamente, às janelas de tempo de quatro e quarenta horas.

(a)



(b)

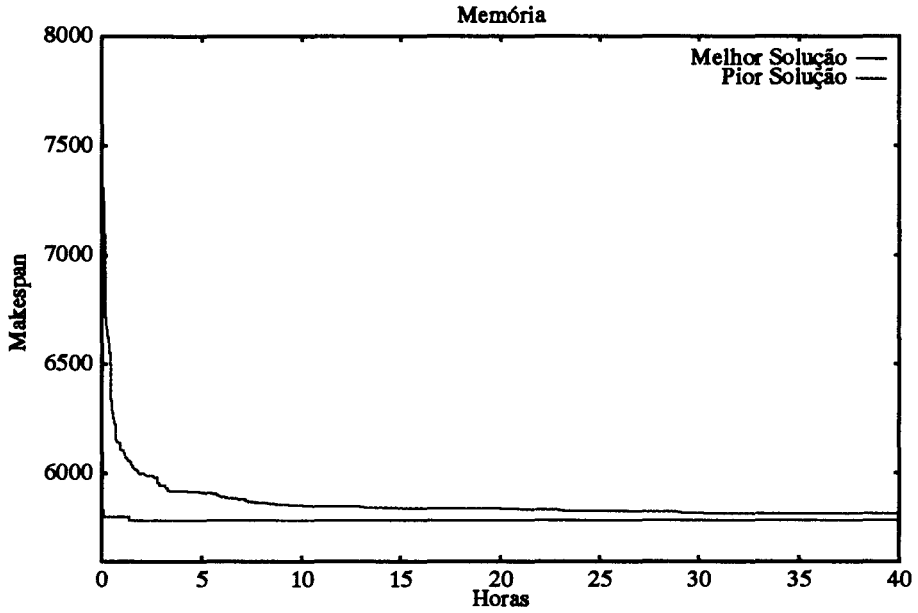
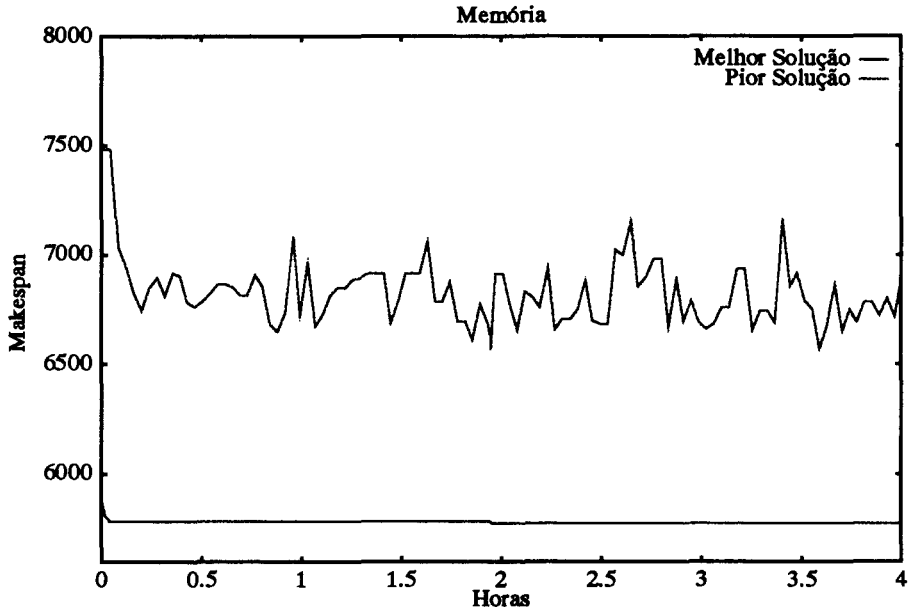


Figura 5.11 Comportamento da melhor e pior soluções na memória de Soluções Completas e Refinadas do *A-Team* resolvendo FSP de 100x10, utilizando o fluxo de dados apresentado na figura 5.7 e a política de destruição D5.

(a)



(b)

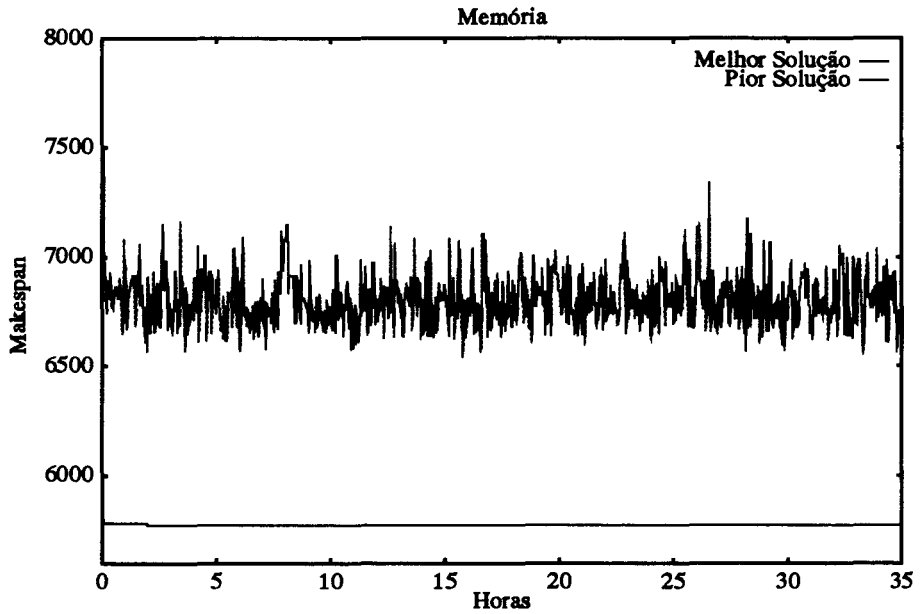
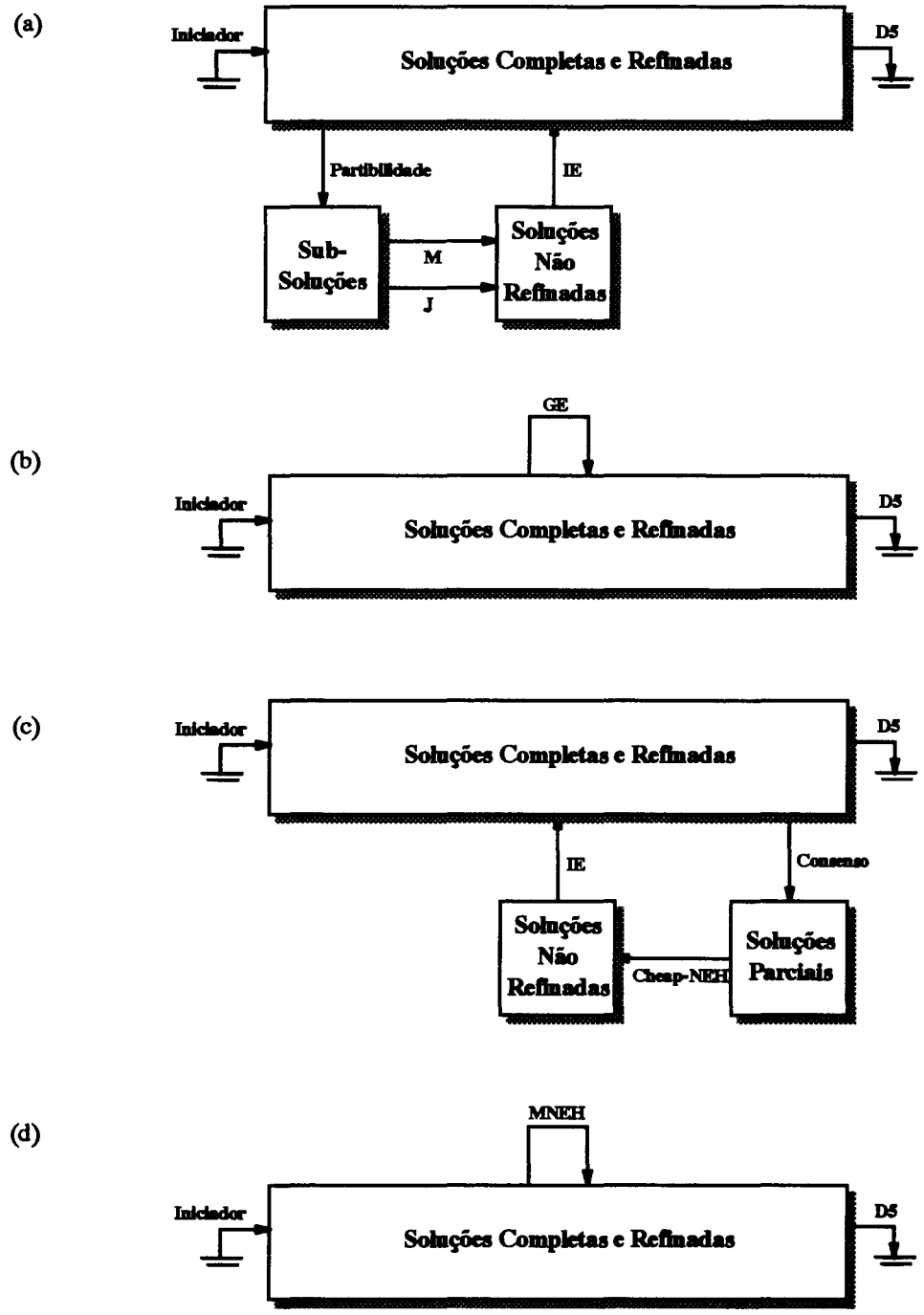


Figura 5.12 Fluxos de Dados utilizados para mostrar a performance de cada um dos quatro cíclos possíveis do *A-Team* para o FSP (veja figura 5.7). Cíclos são utilizados uma vez que apenas os algoritmos GE e MNEH são capazes de produzirem soluções válidas por si só.



suficente para encontrar as melhores soluções conhecidas, como pode ser observado pelo gráfico da figura 5.13.

Para mostrar que os *A-Teams* para o FSP são Eficientes em Escala sobre o conjunto dos agentes mencionados anteriormente (figura 5.12) e o *Makespan*, executou-se quatro configurações diferentes de *A-Teams* para a instância 100x10. A primeira configuração consiste apenas do ciclo formado pelos algoritmos de Partibilidade, J, M e IE. As próximas três configurações incorporaram, respectivamente, o algoritmo de melhoria GE, o ciclo formado pelos algoritmos Consenso e Cheap-NEH e, finalmente, o algoritmo de melhoria MNEH. Esta seqüência está ilustrada na figura 5.14. Observamos que à medida que os algoritmos eram introduzidos, a performance do time era gradativamente melhorada, como pode ser verificado no gráfico da figura 5.15. Cada ponto deste gráfico corresponde ao *Makespan* da configuração utilizada em função do tempo.

Embora possa existir uma ordem em que a adição dos algoritmos não cause uma melhora monotônica na performance do time, a definição de eficiência em escala requer a existência de ao menos uma ordem. Portanto, pode-se conluir que os *A-Teams* para o FSP são eficientes em escala.

5.11.4 *Resultados*

Todos os testes computacionais foram realizados em estações de trabalho SUN SPARCstation 10 ligadas em rede. Os tempos de processamento apresentados são tempo de CPU. Para cada uma das seis instâncias utilizadas, apresentamos os resultados obtidos em três execuções. Esses resultados estão apresentados na tabela 5.3.

Tabela 5.3 Resumo dos resultados produzidos pelo *A-Team* (d) da figura 5.14.

Instância	Limite Inferior Proposto (Cap. 6)	Melhor *Makespan* Conhecido	*A-Team*				
			Melhor *Makespan*	Média *Makespan*	Taxa Sucesso	Melhor tempo (s)	Média tempo (s)
20x5	1232	1278	1278	1278,0	3/3	1	1
20x10	1454	1582	1582	1582,0	3/3	27	56
50x5(a)	2724	2724[a]	2724	2724,0	3/3	1	1
50x5(b)	2776	2782[a]	2782	2782,0	3/3	1	1
100x10	5761	5776	5771	5776,7	2/3	7013	13868,7
500x20	25922	26316	26253	26379,0	1/3	89797	405313,3

a. ótimo.

Em duas das três execuções para a instância 100x10, encontrou-se melhores *Makespan* (5774 e 5771) que os publicados recentemente por Taillard [Tai93]. Isto também ocorreu para a instância 500x20, porém em apenas uma das execuções. Desse modo, a taxa de sucesso descrita na tabela corresponde à habilidade do *A-Team* em encontrar

soluções pelo menos tão boas quanto aquelas publicadas anteriormente. A coluna do Melhor Tempo mostra em que instante os *A-Teams* produziram a melhor solução daquela execução, enquanto a última coluna apresenta a média destes tempos das três execuções.

Figura 5.13 Performance individual dos algoritmos num *A-Team*, conforme as estruturas da figura 5.12. Nenhum algoritmo (ou ciclo) individual foi capaz de alcançar o *Makespan* da melhor solução conhecida.

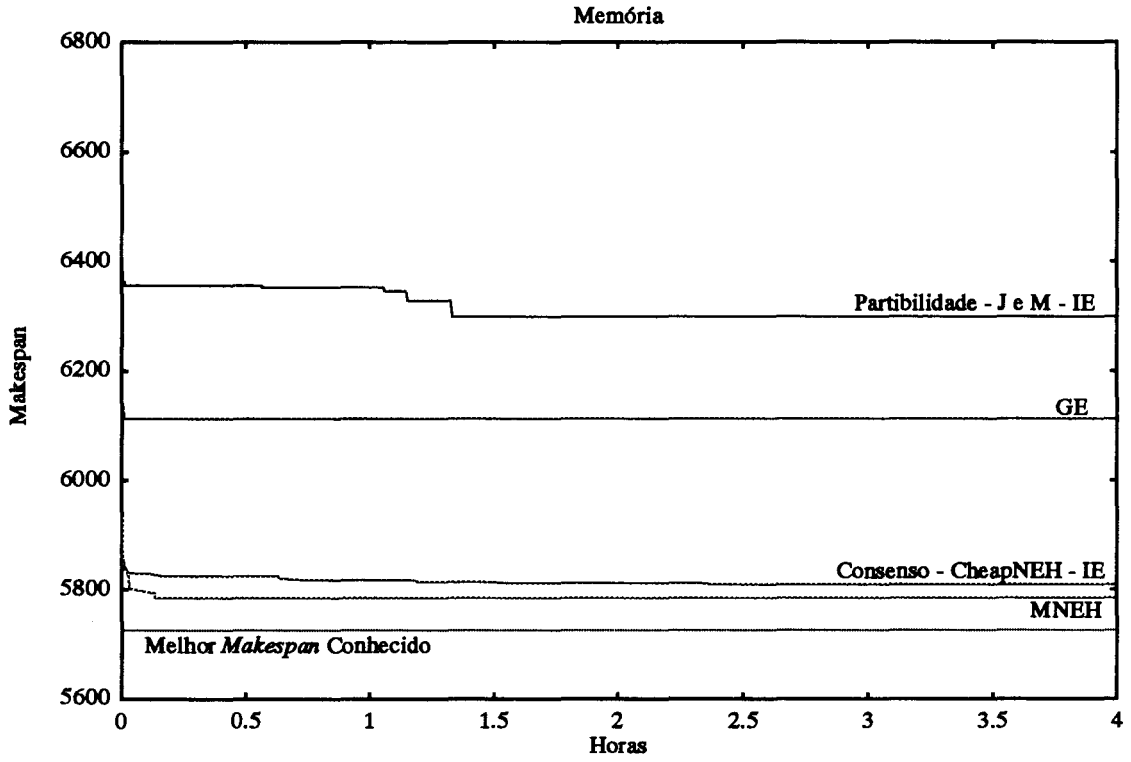
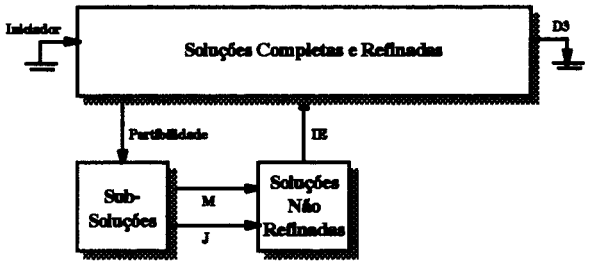
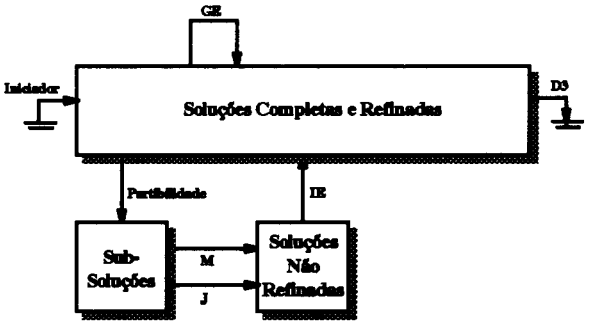


Figura 5.14 Fluxo de dados usados para mostrar eficiência em escala. A partir de um ciclo (Partibilidade, J, M e IE), introduzimos o algoritmo de melhoria GE, um segundo ciclo (Consenso, Cheap-NEH e IE) e finalmente o algoritmo MNEH.

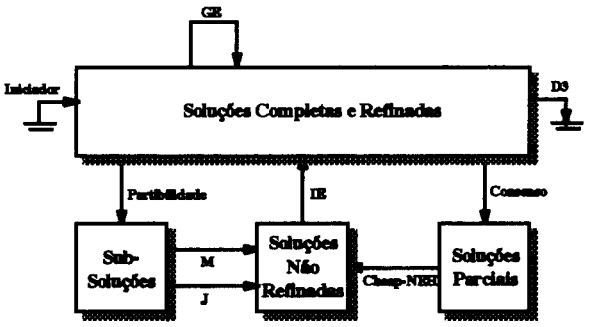
(a)



(b)



(c)



(d)

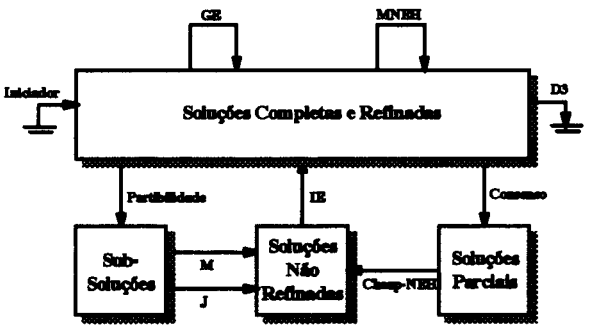
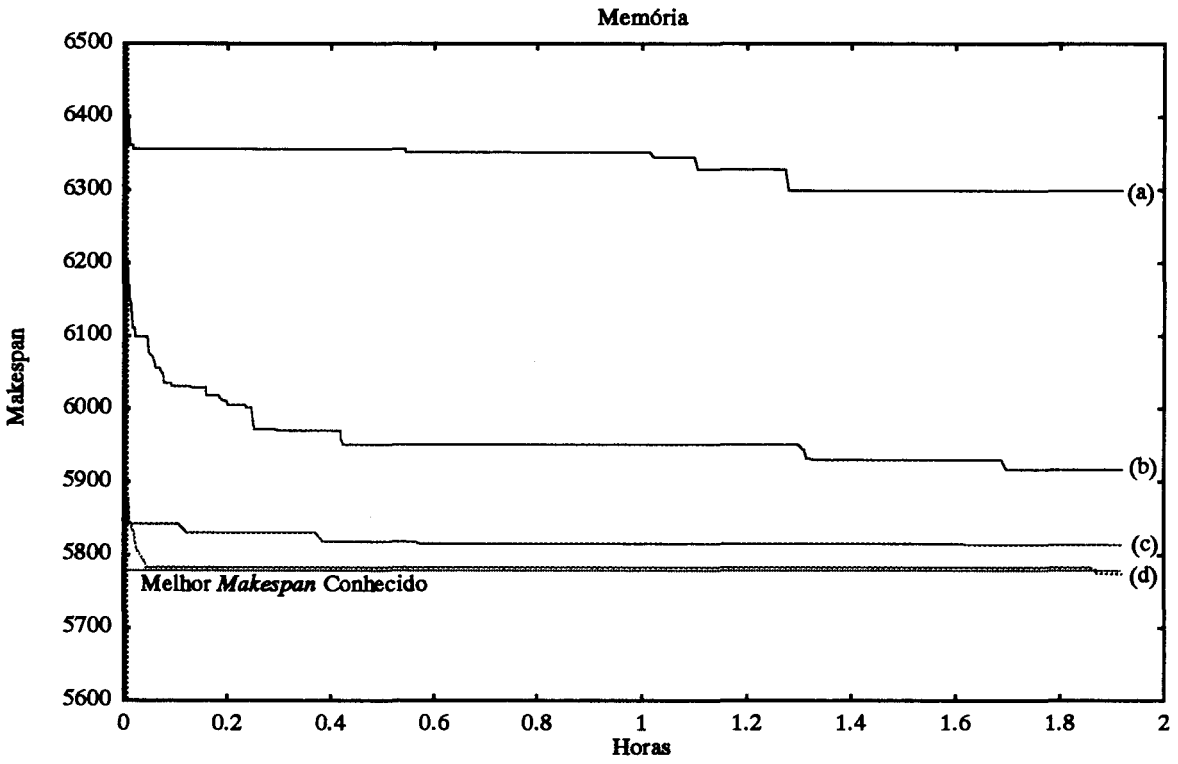


Figura 5.15 Performance dos *A-Teams* após a adição sucessiva de agentes. Cada curva mostra a performance de um *A-Team* da figura 5.14. A curva (a) corresponde ao ciclo formado pelos agentes Partibilidade, J, M e IE. As curvas (b), (c) e (d) correspondem à adição dos agentes GE, ciclo Consenso, Cheap-NEH e, finalmente, ao agente MNEH. À medida que novos algoritmos são adicionados, a performance do time é monotonicamente melhoriada (tempo e qualidade). O *A-Team* (d) encontrou melhores soluções que as publicadas recentemente [Tai93].



5.11.5 *Processamento Distribuído*

Devido à total independência dos agentes, ao fato de as comunicações ocorrerem assincronamente e devido, ainda, à granularidade grossa dos algoritmos, os *A-Teams* se mostram adequados ao processamento distribuído e/ou paralelo. Neste trabalho, testamos a execução do *A-Team* (d) da figura 5.14 para a instância 100x10 em apenas um, dois, quatro e oito processadores. Os resultados detalhados para um processador foram apresentados na tabela anterior. Os resultados para dois, quatro e oito estão apresentados comparativamente na tabela abaixo.

Tabela 5.4 Comparação do *Makespan* e tempos obtidos usando um, dois, quatro e oito processadores. Os valores abaixo são as médias de três execuções.

		Processadores			
		1	2	4	8
100x10	Média Makespan	5776,6	5779,0	5778,3	5778,7
	Média Tempo	13868,7	7967,3	4640,0	1839,3

É importante mencionar que a taxa de sucesso no processamento distribuído foi de 1/3 (ou seja, foi encontrado ao menos uma solução de *Makespan* melhor ou igual a 5776). Observe também que foi obtido um *speed up* linear, como pode ser verificado nos gráficos que seguem.

Figura 5.16 Execução paralela do *A-Team* para a instância 100x10 utilizando um, dois, quatro e oito procesadores. A linha mostra a redução do tempo após adições sucessivas de processadores e os pontos as médias dos *Makespans* das três execuções.

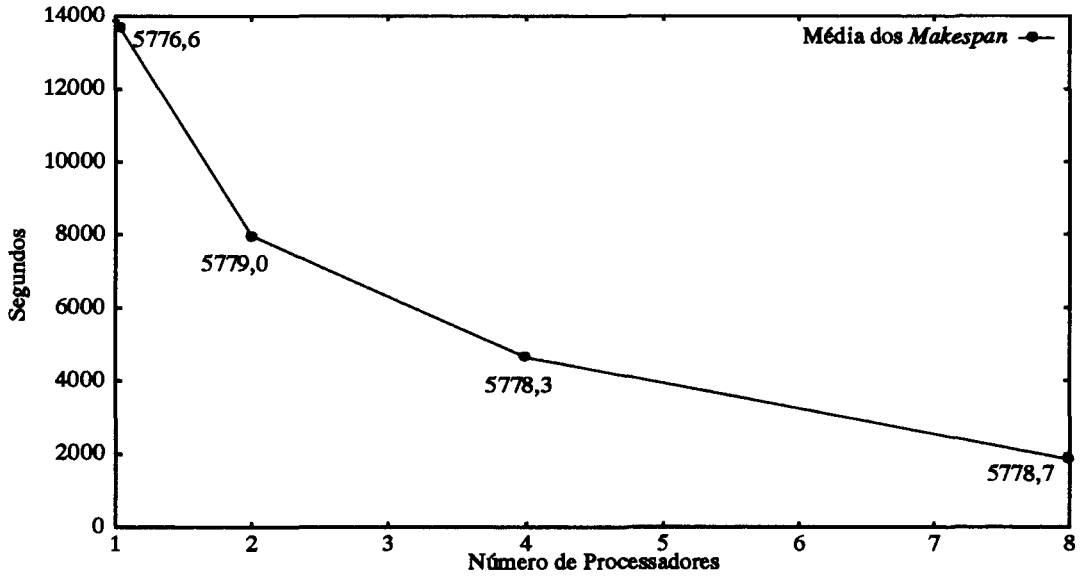
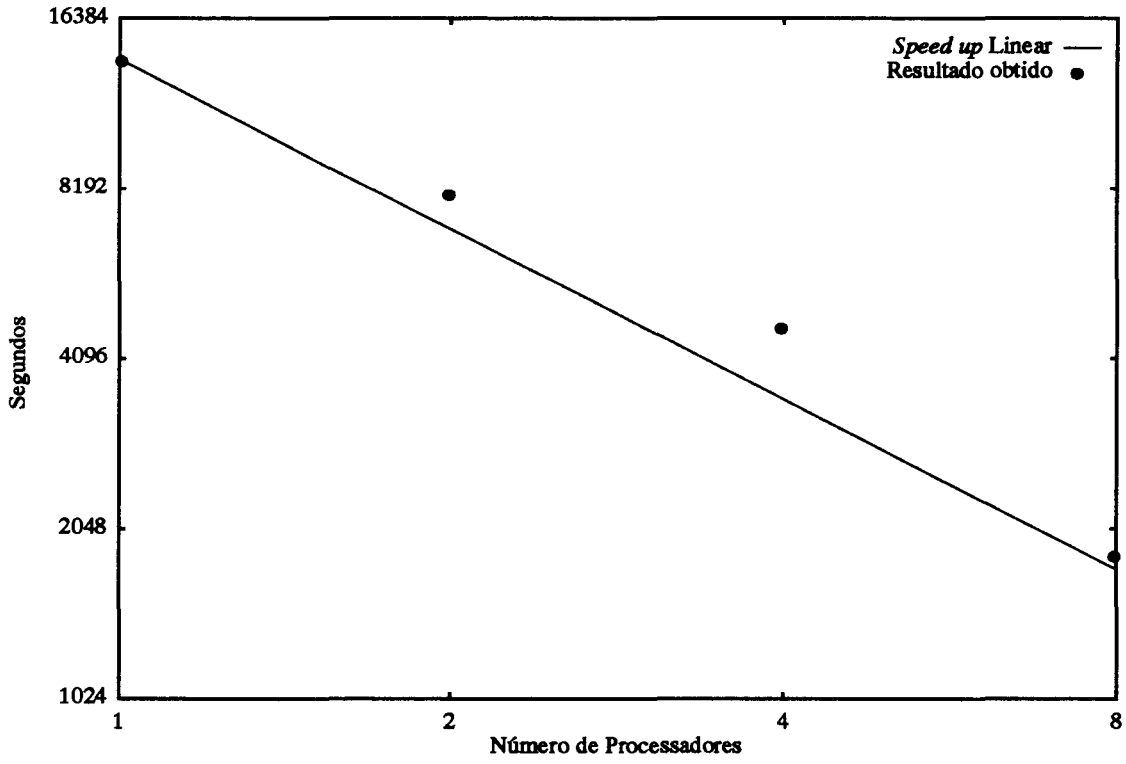


Figura 5.17 Execução paralela do *A-Team* para a instância 100x10 utilizando um, dois, quatro e oito procesadores. A linha indica os valores esperados para um *speed up* linear. Os pontos mostram as médias dos resultados obtidos pelos *A-Teams*.



5.12 Conclusões

Dos testes desempenhados neste capítulo pode-se concluir que:

- a metodologia apresentada no capítulo 4 foi aplicada com sucesso ao FSP de forma simples e quase direta;
- poucos algoritmos (simples) foram criados para compor os *A-Teams* apresentados;
- os resultados obtidos para as instâncias testadas foram iguais ou melhores que os publicados recentemente;
- os *A-Teams* apresentaram eficiência em escala, ou seja, à medida que novos algoritmos eram inseridos no time, a performance melhorava monotonicamente;
- os *A-Teams* apresentaram um *speed up* extremamente próximo ao linear, ou seja, o tempo de processamento decresce proporcionalmente ao número de processadores utilizados.

5.13 Sumário

Este capítulo tratou da especificação e implementação de *A-Teams* para o problema clássico de escalonamento de tarefas *Flow Shop Problem* de permutação. A metodologia proposta no capítulo 4 mostrou poder ser facilmente empregada na solução de problemas de otimização combinatória, através da especificação das memórias, dos fluxos de dados, das políticas de destruição e iniciação. A maioria dos agentes utilizados foram obtidos quase diretamente da literatura, sendo necessárias poucas modificações para adaptá-los aos *A-Teams*. Seis instâncias publicadas recentemente foram utilizadas para efeito de comparação. Os resultados obtidos pelos *A-Teams* foram iguais ou superiores em qualidade aos melhores *Makespans* conhecidos. Os *A-Teams* desenvolvidos também apresentaram eficiência em escala e *speed up* linear no número de processadores utilizados.

5.14 Notas Bibliográficas

Desde a publicação do artigo de Johnson para o *Flow Shop Problem* de duas máquinas em 1954, a literatura de problemas de seqüenciamento cresceu significativamente. Johnson propôs um algoritmo de complexidade $O(n\log n)$ para escalonar n jobs em apenas duas máquinas com o objetivo de minimizar o *Makespan*. Além de provar que esse algoritmo encontra sempre a solução ótima, Johnson provou também a existência de uma solução ótima em que uma mesma permutação de jobs pode ser usada nas duas máquinas.

Em relação ao *Flow Shop Problem* de *nxm*, existem, na literatura, diversas resenhas. Em particular, podem-se citar os seguintes trabalhos [PPE84, KSSTT89, DPS92, ML93].

Diversas heurísticas foram publicadas para o *Flow Shop Problem* de permutação onde o objetivo é minimizar o *Makespan*. Muitas dessas heurísticas são baseadas em regras de prioridades, como pode ser observado nos seguintes trabalhos [Pal65, CDS70, Gup71, Bak74, Dan77, KS80, HR88]. Várias outras abordagens como heurísticas de melhoria, *Tabu Search*, Algoritmos Genéticos, *Simulated Annealing*, etc.., encontram-se na literatura [BG76, SS82, NEH83, WH89b, Tai90, OS90, BP91, HC91, Ree93].

Parte deste capítulo pode ser encontrada em [PS94], onde os resultados preliminares obtidos para o FSP foram publicados.

Capítulo 6

Novos Limites Inferiores para o Flow Shop Problem

O presente capítulo descreve novos limites inferiores para o problema clássico de escalonamento de tarefas *Flow Shop Problem* de permutação. Esta nova fórmula de calcular limites inferiores é aplicada a 120 instâncias da literatura, provando a otimalidade de algumas e obtendo melhores limites inferiores para as demais. Outros limites inferiores significativos da literatura são também mencionados para efeito de introdução e comparação.

6.1 Introdução

Bons Limites Inferiores (do inglês *Lower Bounds* - LB) são importantes tanto para os Métodos Aproximados como para os Métodos Exatos. Para alguns métodos exatos, por exemplo o *Branch & Bound*, bons limites inferiores significam a possibilidade de eliminação de parte do espaço de soluções e, conseqüentemente, melhor performance. Para os métodos aproximados, bons limites inferiores indicam o quanto as soluções obtidas podem distar da solução ótima.

Várias fórmulas para calcular limites inferiores podem ser encontradas na literatura. Por motivo de clareza, as fórmulas de LB apresentadas neste capítulo estarão na sua forma mais simples, ou seja, as fórmulas fornecerão LB para instâncias e não para soluções parciais das instâncias (como é feito nos algoritmos de *Branch & Bound*). A extensão das fórmulas para a inclusão das soluções parciais pode ser efetuada sem nenhuma dificuldade, bastando que se considere, no instante do cálculo do LB, que as máquinas estarão disponíveis para processamento após o término dos jobs previamente escalonados. Seja σ uma seqüência parcial de jobs já escalonados e σi uma seqüência parcial acrescida de um job i. Para uma dada seqüência parcial σi, $q(\sigma i, k)$ indica o instante de término de processamento dessa seqüência parcial na máquina k, e é geralmente calculado recursivamente pela fórmula abaixo:

$$q(\sigma i, k) = max\{q(\sigma, k), q(\sigma i, k-1)\} + t_{ik} \qquad k = 1, 2, \ldots, m \qquad \text{(Eq 6.1)}$$

em que $q(\sigma, 0) = q(\varnothing, k) = 0$. Portanto, para modificar as fórumlas apresentadas neste capítulo, basta considerar que a máquina k só estará disponível para processamento após $q(\sigma, k)$, em que σ indica a seqüência dos jobs já escalonados.

As próximas seções deste capítulo descrevem, respectivamente, algumas fórmulas significativas da literatura para o cálculo de LBs, as novas fórmulas propostas e os resultados obtidos sobre 120 instâncias publicadas recentemente para o FSP. O apêndice A fornece a tabela das instâncias utilizadas e os novos LBs encontrados.

6.2 LB Simples calculado em máquinas (LB1 e LB2)

Este LB é calculado a partir da definição do *Makespan*. O *Makespan* é definido como sendo o instante de término do último job na última máquina (considerando que o instante inicial de processamento foi zero). Portanto, este LB consiste no tempo total de processamento da última máquina para a instância I, ou seja:

$$LB1(I) = \sum_{i=1}^{n} t_{im} \qquad \text{(Eq 6.2)}$$

Embora o cálculo de *LB1* seja da ordem de $O(n)$, seu resultado não leva em consideração as demais máquinas. Uma vez que a máquina m pode possuir tempo total de processamento inferior a alguma outra máquina, pode-se melhorar *LB1* calculando o maior tempo total de processamento dentre todas as máquinas. Seja *TTPM(j)* o tempo total de processamento da máquina *j*:

$$TTPM(j) = \sum_{i=1}^{n} t_{ij} \qquad \text{(Eq 6.3)}$$

O novo *LB2* consiste no maior *TTPM(j)* para todo *j*.Claramente o *LB2* é da ordem de $O(nm)$. *LB1* pode ser encontrado em [Bak75].

6.3 LB Completo calculado em máquinas (LB3)

No *LB2* mencionado anteriormente, o LB é calculado em cada máquina, selecionando-se, então, o maior deles. Um fato não explorado por *LB2* é que, ao término do processamento em uma máquina *j*, o último job processado em *j*, digamos *k*, ainda será processado nas máquinas subseqüentes. Portanto, existe um tempo restante de processamento nas máquinas subseqüentes a *j*, no qual o job *k* estará sendo processado. Como não sabemos qual será o último job a ser escalonado, podemos escolher o job que produz o menor tempo de processamento entre as máquinas *k+1* e *m*, garantindo um novo LB. Denominaremos *Cauda(i, j)* o tempo restante de processamento do job *i* entre as máquinas *j* e *m* (inclusive):

$$\begin{aligned} Cauda(i, m+1) &= 0 \qquad && i = 1 \ldots n \\ Cauda(i,j) &= \sum_{l=j}^{m} t_{il} && \begin{matrix} i = 1 \ldots n \\ j = 1 \ldots m \end{matrix} \end{aligned} \qquad \text{(Eq 6.4)}$$

Portanto, *LB3* pode ser calculado pela sequinte fórmula:

$$\begin{aligned} a_j &= min_{1 \le i \le n} Cauda(i, j+1) \\ LB3(I) &= max_{1 \le j \le m} \{ a_j + TTPM(j) \} \end{aligned} \qquad \text{(Eq 6.5)}$$

Note que $LB1(I) \le LB2(I) \le LB3(I)$, qualquer que seja a instância *I*. *LB3* pode ser calculado em $O(nm^2)$ e pode ser encontrado em [IS65].

6.4 LB Sem Interferência (LB4)

Esta é uma outra maneira de calcular o LB com base no *Makespan* da última máquina. Diferentemente de *LB1*, neste caso estamos interessados nos tempos de processamento dos jobs nas demais máquinas.

A máquina m só poderá iniciar seu processamento no instante zero se, qualquer que seja o job i, $t_{ij} = 0$, para $1 \le j \le m\text{-}1$. Essa observação, embora muito realista, não é considerada em *LB1*. Vamos definir $c(i,m)$ como sendo um limite inferior para o instante de término do job i na máquina m-1 (antes do instante $c(i,m)$, o job i não está disponível para processamento na máquina m):

$$c(i,1) = 0 \qquad i = 1, \ldots n$$
$$c(i,j) = \sum_{k=1}^{j-1} t_{ik} \qquad \begin{matrix} i = 1, \ldots n \\ j = 2, \ldots m \end{matrix} \qquad \text{(Eq 6.6)}$$

Para efeito de cálculo do LB, pode-se dizer que a máquina m só poderá iniciar seu processamento a partir do instante determinado por $Tm = min_{1 \le i \le n}\{ c(i,m) \}$. Poderíamos considerar $LB4'(I) = Tm + TTPM(m)$. Porém, podemos tirar proveito dos jobs e seus respectivos $c(i,m)$. Sem perda de generalidade, considere que os jobs estão rotulados tal que $c(1,m) \le c(2,m) \le \ldots \le c(n,m)$. A seqüência de jobs 1, 2, ... n define uma ordem de escalonamento na máquina m segundo a política FIFO, ou seja, os jobs são escalonados à medida que estejam disponíveis.

Seja $Td_{1m} = Tm$ o instante do tempo em que o job 1 estará disponível para ser escalonado na máquina m. Claramente, o job 2 só poderá ser escalonado na máquina m no instante $Td_{2m} = max\{ Td_{1m}+t_{1m}, c(2,m) \}$. Se o job 2 estiver disponível para processamento antes do job 1 terminar, o job 2 deve esperar ($Td_{2m} = Td_{1m}+t_{1m} \ge c(2,m)$). Caso contrário, o job 1 terminou antes de o job 2 estar disponível ($Td_{2m} = c(2,m)$). Nesse caso, a máquina m permanecerá ociosa por $c(2,m)-(Td_{1m}+t_{1m})$ unidades de tempo, até que o job 2 esteja disponível. Este processo é continuado até que todos os jobs tenham sido escalonados. Generalizando,

$$Td_{0m} = 0 \qquad t_{0m} = 0$$
$$Td_{im} = max\{Td_{(i-1)m} + t_{(i-1)m}, c(i,m)\} \qquad \text{(Eq 6.7)}$$

Portanto, o instante de término do último job a ser processado na máquina m é dado por:

$$LB4(I) = Td_{nm} + t_{nm} \qquad \text{(Eq 6.8)}$$

O cálculo de *LB4* pode ser feito em $O(nm+n\log n)$ e pode-se dizer apenas que $LB1(I) \leq LB4(I)$. *LB2* e *LB3* dependendo da instância, podem ser melhores ou piores que *LB4*. *LB4* foi originalmente proposto por Ashour [Ash70].

6.5 LB Calculado com base em jobs (LB5)

Qualquer que seja a solução ótima para o FSP, o seu respectivo *Makespan* inclui o maior tempo total de processamento entre todos os jobs. Este maior tempo total de processamento também pode ser considerado como um LB, ao qual denominaremos *LB5'*. Usando a função *Cauda* definida anteriormente, $LB5'(I) = max_{1 \leq i \leq n}\{Cauda(i,1)\}$. *LB5'* considera apenas o maior tempo total de processamento e por isso pode permanecer distante do *Makespan* da solução ótima. Na verdade, além do valor fornecido por *LB5'*, temos que processar os demais jobs. Como não sabemos qual a ordem dos demais jobs, podemos somar a *LB5'* o tempo da menor operação de cada um dos outros jobs na máquina 1 e *m*, obetendo o novo LB:

$$LB5''(I) = max_{1 \leq i \leq n}\{Cauda(i,1)\} + \sum_{k=1, k \neq i}^{n} min\{t_{k1}, t_{km}\} \qquad \text{(Eq 6.9)}$$

LB5'' pode fornecer resultados muito decepcionantes se, por exemplo, o desvio padrão entre os tempos de processamento das operações de um mesmo job for significativo (pois estaremos sempre considerando a menor das operações e desconsiderando todas as demais).

Considere uma instância *I* qualquer para o FSP e seja $M^*(I)$ o *Makespan* da solução ótima para *I*. Ao retirarmos a máquina 1 da instância *I*, obtemos uma nova instância *I'*, onde $M^*(I) \geq M^*(I')$ e qualquer LB para *I'* vale também como LB para *I* . Por bizarro que pareça, ao retirarmos a primeira máquina de uma instância *I*, obtendo uma nova instância *I'*, podemos ter $LB5''(I') \geq LB5''(I)$. Isso pode ocorrer, por exemplo, se as menores operações de todos os jobs estiverem na primeira máquina. Sendo assim, podemos incrementar o *LB5''* através de sucessivas retiradas da primeira máquina, até que reste uma instância de apenas duas máquinas. O maior *LB5''* obtido em todas as instâncias geradas compõe o *LB5*, que pode ser formulado como segue:

$$LB5(I) = max_{1 \leq j \leq m-1}\left\{max_{1 \leq i \leq n}\{Cauda(i,j)\} + \sum_{k=1, k \neq i}^{n} min\{t_{kj}, t_{km}\}\right\} \qquad \text{(Eq 6.10)}$$

LB5 pode ser encontrado em [McB67] e os cálculos necessários para obtê-lo são da ordem de $O(mn^2)$. *LB5* pode ser melhor ou pior que *LB2*, *LB3* e *LB4*, dependendo da instância.

LB1, *LB3*, *LB4* e *LB5* são comparativamente abordados em [Bak75].

6.6 LB com base na relaxação das capacidades das máquinas (LB6)

A idéia básica é obter um LB através da relaxação das capacidades das máquinas. ***Máquinas gargalo*** (representadas por MG_i) com capacidade de processar apenas um job a cada instante são tratadas como ***Máquinas não gargalo*** (representadas por $\overline{MG}_i$), capazes de processar infinitos jobs a cada instante.

Problemas de escalonamento que possuem mais de duas máquinas gargalo são classificados como problemas NP-Difíceis [Bak74]. Portanto, para obtermos os LB, pode-se relaxar as capacidades das máquinas a fim de reduzir o número de máquinas gargalo.

Dada uma instância I para o FSP de m máquinas gargalo, vamos convertê-la em uma nova instância I' de, no máximo, duas máquinas gargalo. Dentre todas as máquinas não gargalo de I', sejam MG_u e MG_v as máquinas gargalo, onde $1 \leq u \leq v \leq m$. Duas máquinas não gargalo consecutivas, $\overline{MG}_1$ e $\overline{MG}_2$, podem ser tratadas como uma única máquina não gargalo $\overline{MG}_3$, bastando, para isso, considerar o tempo de processamento do job j em $\overline{MG}_3$ como sendo a soma dos tempos de processamento do job j nas máquinas $\overline{MG}_1$ e $\overline{MG}_2$. Portanto, qualquer seqüência de máquinas não gargalo pode ser considerada como apenas uma máquina não gargalo. Como em I' só são permitidas, no máximo, duas máquinas gargalo, decorre que, na conversão de I para I', obtemos uma instância de no máximo 5 máquinas. Formalmente, dadas as máquinas u e v escolhidas como gargalo, obtemos uma nova instância I', onde desejamos escalonar os n jobs de I nas máquinas $\overline{MG}_{<u}$, MG_u, $\overline{MG}_{uv}$, MG_v e $\overline{MG}_{v>}$, minimizando o *Makespan*. Esse *Makespan* obtido de I' é um LB para o problema original. Embora seja uma relaxação, resolver este problema de 5 máquinas para quaisquer u e v continua sendo NP-Difícil [LLK78]. Observe que, a partir da instância I, podem-se criar $m(m+1)/2$ novas instâncias I', conforme a escolha de u e v.

As máquinas não gargalo $\overline{MG}_i$ podem ser retiradas da instância I' e serem compensadas através da adição de $r_i = min_{1 \leq k \leq n}\{t_{ki}\}$ ao *Makespan* , ou zero se u=1, v=u+1 ou v=m. Este artifício é útil para simplificar ainda mais I', obtendo uma instância que possa ser resolvida polinomialmente.

Eliminando apenas as máquinas $\overline{MG}_{<u}$ e $\overline{MG}_{v>}$ de I' através do artifício supra-mencionado, obtemos uma instância I'' de 3 máquinas, que pode ser resolvida pelo algoritmo de Johnson [Bak74], usando os tempos de processamento $t_{iMGu}+t_{i\overline{MG}uv}$ e $t_{i\overline{MG}uv}+t_{iMGv}$. Ao *Makespan* obtido pelo algoritmo de Johnson sobre I'' é adicionado $r_{<u}$ e $r_{v>}$, dando origem a um LB para I' e I, que denominaremos *LB6'*.

Como *LB6'* é calculado sobre instâncias geradas através da escolha de u e v, pode existir um u_1 e v_1 que produza um *LB6'* melhor que o produzido por u_2 e v_2. Portanto, *LB6* é definido como sendo o maior *LB6'* obtido para todo u e v tal que $1 \leq u \leq v \leq m$. A descrição completa desse LB pode ser encontrara em [LLK78], onde se demostra que *LB6(I)* é dominante sobre *LB5(I)*, *LB4(I)* e *LB3(I)*, qualquer que seja a instância I.

LB6 pode ser calculado em $O(m^2 n \log n)$.

6.7 Outro LB Calculado com base em jobs (LB7)

Taillard publicou, recentemente, instâncias de *benchmarks* para o FSP, acompanhadas de um LB e o *Makespan* da melhor solução conhecida [Tai93]. O LB utilizado por Taillard pode ser considerado um aprimoramento de *LB3*.

No cálculo de *LB3* considera-se que os jobs de qualquer máquina estão disponíveis para serem processados no instante zero. Na verdade, a máquina i só pode processar jobs a partir do instante $b_i = min_{1 \le j \le n}\{ c(j,i) \}$. Sendo assim, Taillard acrescenta b_i ao LB da máquina i, que é calculado como em *LB3*. *LB7* é definido como sendo o máximo entre *LB3* acrescido de b_i e o maior dos tempos de processamento dos jobs, ou seja:

$$b_i = min_{1 \le j \le n}\{c(j,i)\}$$
$$LB7(I) = max\{max_{1 \le i \le m}[a_i + b_i + TTPM(i)], max_{1 \le j \le n} Cauda(j,1)\} \quad \text{(Eq 6.11)}$$

Observe as semelhancas de *LB7* e *LB3*. LB7 também pode ser calculado em $O(nm^2)$.

6.8 Primeiro LB proposto (LB8)

O autor desta dissertação propôs um novo LB com base no *LB7* proposto por Taillard. No *LB7*, em cada máquina, são calculados a_i, b_i e o tempo total de processamento da respectiva máquina. Veja que a_i e b_i da máquina i são independentes do a_j e b_j da máquina j. Isso significa, por exemplo, que $b_i = c(k,i)$ e $b_j = c(h,j)$, ou seja, na máquina i escolhe-se o job k para ser processado primeiro e em j o job h. Porém, como o problema que se trata é FSP de permutação, na seqüência ótima, um mesmo job será processado primeiramente em todas as máquinas. Sendo assim, propomos que se b_i na máquina i utiliza o job k, então todas as demais máquinas utilizam k. Para garantirmos um LB, é evidente que a escolha de k não pode ser aleatória. O job k escolhido como sendo o primeiro job em todas as máquinas, será aquele que, após ser processado em i e somado o tempo de processamento restante da máquina i, produzir o menor *Makespan*. Podemos determinar o job k como aquele que produzir o menor *Makespan* pela seguinte fórmula:

$$min_{1 \le k \le n}\{max_{1 \le j \le m}[c(k,j) + TTPM(j)]\} \quad \text{(Eq 6.12)}$$

Este mesmo raciocínio usado sobre b_i pode ser aplicado a a_i, encontrando um outro job h que seja considerado o último a ser processado em todas as máquinas. O LB proposto pode ser calculado pela fórmula abaixo, dado que o job k está determinado.

$$LB8(I) = min_{1 \le h \le n}\{max_{1 \le j \le m}[c(k,j) + TTPM(j) + Cauda(h, j+1)]\} \quad \text{(Eq 6.13)}$$

Claramente $LB7(I) \leq LB8(I)$. *LB8* também pode ser calculado em $O(nm^2)$.

6.9 Segundo LB proposto (LB9)

No cálculo do *LB4* considerava-se que um job i na máquina m só poderia iniciar seu processamento após $c(i, m)$. Esses tempos serviam como regra de escalonamento (FIFO) que, em outras palavras, só permite a um job ser escalonado, quando esse escalonamento for potencialmente viável.

Pode-se incorporar também ao cálculo de *LB8* a regra de escalonamento FIFO, obtendo, assim, um LB ainda mais realístico. Seja *TTFIFO(j,k)* o tempo total de processamento na máquina j, considerando que o primeiro job escalonado é o job k e que os demais jobs são escalonados segundo a política FIFO. Sem perda de generalidade, pode-se considerar que na máquina j os jobs estão dispostos na ordem (1, 2, ..., n) tal que $c(2,j) \leq c(3,j) \leq \ldots \leq c(n,j)$ e $k=1$. Vamos expandir Td_{im} definido para *LB4* de forma a incorporar outras máquinas. Seja Td_{ij} o instante em que o job i pode ser processado na máquina j, respeitando $c(i,j)$ e os eventuais jobs escalonados anteriormente, temos:

$$Td_{0j} = 0 \qquad t_{0j} = 0 \qquad j = 1, \ldots m$$

$$Td_{ij} = max\{Td_{(i-1)j} + t_{(i-1)j}, c(i,j)\} \qquad \begin{matrix} i = 1, \ldots n \\ j = 1, \ldots m \end{matrix} \qquad \text{(Eq 6.14)}$$

Portanto, *TTFIFO(j,k)* consiste em calcular Td_{nj}, dado que a ordem estabelecida para os jobs apenas na máquina j é (1, 2, ..., n), onde o job $k=1$ e $c(2,j) \leq c(3,j) \leq \ldots \leq c(n,j)$.

$$TTFIFO(j,k) = Td_{nj} + t_{nj} \qquad \text{(Eq 6.15)}$$

Como em *LB8*, o primeiro passo no cálculo de *LB9* consiste em determinar o job k. Em seguida, basta utilizar a fórmula abaixo:

$$LB9(I) = min_{1 \leq h \leq n}\{max_{1 \leq j \leq m}[TTFIFO(j,k) + Cauda(h, j+1)]\} \qquad \text{(Eq 6.16)}$$

LB9 é dominante sobre *LB8* e pode ser calculado em $O(mn^2+nm^2)$.

6.10 Resultados

Taillard publicou, recentemente, um algoritmo para gerar instâncias do FSP de permutação [Tai93]. O objetivo do artigo de Taillard é fornecer um conjunto de instâncias de *Benchmarks* para alguns problemas de seqüenciamento de tarefas (FSP, JSP OSP). A cada uma das 120 instâncias sugeridas, está associado um LB e o valor da melhor solução conhecida (*Upper Bound* - UB). Para obtenção do LB, Taillard propôs a fórmula descrita em *LB7* e o UB foi obtido através de Busca Tabu com um número muito

grande de iterações. Embora em muitas das vezes o LB estivesse próximo ao UB, Taillard não menciona a otimalidade de nenhuma das 120 instâncias.

O *LB9* proposto nesta dissertação obteve resultados superiores a *LB7* em 57.5% das 120 instâncias, sendo que a melhoria sobre o valor de *LB7* chegou até 3.76%.

Embora na maioria das instâncias a melhoria sobre o *LB7* tenha sido inferior a 1%, em duas instâncias de 50 jobs e 5 máquinas a melhoria de 0.44% e 0.21% foi suficiente para mostrar a otimalidade do UB fornecido. A tabela A.1, no apêndice A, indica os valores obtidos por *LB7*, *LB8* e *LB9*.

A complexidade computacional atribuida a LB8 e LB9 é reduzida para $O(nm)$ sempre que os mesmos forem recalculados. Isto decorre do fato de as funções *TTFIFO*, *c*, e *Cauda* já terem sido computadas e seus respectivos resultados armazenados. Esta é uma vantagem importante, pois caso sejam utilizados em algoritmos de *Branch & Bound*, um esforço maior é despendido apenas num pré-processamento inicial.

A comparação de *LB6* com *LB9* é um ponto em aberto que poderá ser atacado futuramente.

6.11 Conclusões

Neste capítulo abordamos o cálculo de Limites Inferiores para o *Flow Shop Problem*. Os principais métodos encontrados na literatura foram reescritos com o objetivo de facilitar a compreensão dos dois novos Limites Inferiores propostos nesta dissertação. Esses limites se mostraram superiores aqueles publicado recentemente por Taillard [Tai93] em 57.5% das instâncias testadas.

6.12 Notas Bibliográficas

Os primeiros cinco LB apresentados podem ser encontrados em Baker [Bak75], onde um estudo comparativo entre eles é detalhado no uso em algoritmos de *Branch & Bound*. *LB6* foi apresentado por Lenstra *et al.* [LLK78] em 1978, onde diversos LB são classificados sob a ótica de relaxação das capacidades das máquinas. Um estudo da aplicabilidade destes LBs aos algoritmos de *Branch & Bound* também são apresentados em [LLK78].

Em 1992, Karabati *et al.* [KKK92] apresentou uma nova maneira de calcular LB para o FSP através da resolução de problemas lineares de atribuição. Porém, para conseguir bons LB é necessária a resolução de $\binom{n+m-2}{n-1}$ problemas lineares de atribuição, sendo justificado seu uso em apenas alguns casos particulares de FSP.

O LB proposto por Taillard bem como o algoritmo para construção de instâncias do FSP podem ser encontrados em [Tail93].

Capítulo 7 *Conclusões*

Esta dissertação apresenta uma Metodologia de Especificação de Times Assíncronos, desenvolvida com o propósito de auxiliar a especificação e implementação de *A-Teams* para problemas de Otimização Combinatória em geral. Para tanto, uma seqüência de passos e sugestões são apresentadas para que cada um dos parâmetros de *design* sejam abordados e detalhados (representação de uma solução, algoritmos disponíveis e sua classificação, fluxos de dados a partir de uma estrutura geral, políticas de acesso, políticas de destruição, iniciação, agentes, implementação e modificações).

Esta Metodologia foi aplicada com sucesso ao problema clássico de escalonamento de tarefas *Flow Shop Problem* de permutação, cujos *A-Teams* desenvolvidos resolveram grandes instâncias, encontrando soluções de igual ou melhor qualidade que aquelas publicadas recentemente (1993). Vale resaltar que os *A-Teams* desenvolvidos utilizaram uns poucos algoritmos simples. Constatou-se também que algoritmos podem ser combinados interativa e assincronamente para resolver problemas grandes e complexos, tendo em vista, particularmente, os problemas de escalonamento. Os *A-Teams* desenvolvidos para o *Flow Shop Problem* de permutação foram implementados e testados de maneira distribuída, mostrando o potencial de paralelismo inerente a esta nova técnica.

Desenvolveram-se também duas novas fórmulas para cálculo de limites inferiores para o *Flow Shop Problem* de permutação. Essas fórmulas, quando aplicadas a pouco mais de uma centena de instâncias, produziram limitantes de melhor qualidade para dezenas delas (melhora em mais de 57% das instâncias). Esses novos limitantes demonstraram a otimalidade de duas das instâncias publicadas, até então desconlhecidas como tal.

7.1 Contribuições

Dentre as contribuições deste trabalho podemos citar:

- o desenvolvimento de uma metodologia simples para auxiliar a especificação de Times Assíncronos para problemas de Otimização Combinatória;
- uma seqüência de passos e sugestões para especificar *A-Teams*;
- uma classificação dos algoritmos (heurísticas) e a estrutura geral de fluxos de dados decorrente;
- a comprovação de que algoritmos (simples) juntos são capazes de produzirem melhores resultados do que quando executados isoladamente;
- a comprovação da adequabilidade de *A-Teams* ao domínio de aplicação de problemas de escalonamento de tarefas;
- a determinação de novos limitantes superiores para instâncias de *Benchmark* publicadas recentemente para o *Flow Shop Problem* de permutação;
- o desenvolvimento de duas novas fórmulas para o cálculo de limites inferiores para o *Flow Shop Problem* de permutação;
- a determinação de novos limitantes inferiores para dezenas das instâncias de *Benchmark* publicadas recentemente para o *Flow Shop Problem* de permutação, provando, para duas delas, a otimalidade do limitante superior conhecido.

7.2 Possíveis extensões

Por se tratar de uma nova técnica, *A-Teams* constituem um vasto campo de pesquisa. Relacionados à esta dissertação, algumas das possíveis extensões são:

- expansão da metodologia proposta para incorporar problemas combinatórios mais complexos, como por exemplo problemas dinâmicos e multi-objetivos;
- introdução de aprendizado nos agentes, permitindo que a busca por soluções de boa qualidade seja guiada através da análise da sua performance e dos dados produzidos pelos demais agentes;
- desenvolvimento de um *Framework* que agilize e facilite a implentação de *A-Teams*;
- desenvolvimento de outros fluxos de dados para a estrutura geral proposta;
- desenvolvimento de técnicas para obter a "harmonia" entre os fluxos de dados. Por harmonia entende-se igualar a quantidade de soluções que são depositadas e retiradas das memórias, evitando assim que "gargalos" de dados sejam formados;
- desenvolvimento de outros fluxos de dados e agentes para o FSP.

Apêndice A

Novos Limites Infeirores para instâncias do FSP

Este apêndice contém os resultados obtidos sobre 120 instâncias de *Bechmark* para os novos limites inferiores apresentados no capítulo 6.

As instâncias estão dispostas na tabela abaixo na mesma seqüência apresentada por Taillard [Tai93]. Esta disposição dispensa a replicação das sementes utilizadas para gerar cada uma das instância e por este motivo foi adotada. Os espaços em branco nas colunas de *LB8* e *LB9* indicam que o LB obtido é igual ao fornecido por *LB7*. A última coluna indica a distância de LB9 à LB7 (em percentagens).

Tabela A.1 Novos limites inferiores para as instâncias publicadas por Taillard [Tai93]. As linhas sombreadas indicam as instâncias utilizadas para testes no capítulo 6.

Instâncias	LB7	LB8	LB9	%
20x5	1232			0.00
	1290			0.00
	1073			0.00
	1268			0.00
	1198			0.00
	1180	1192	1192	1.00
	1226			0.00
	1170	1181	1181	0.93
	1206			0.00
	1082	1085	1085	0.27

Instâncias	LB7	LB8	LB9	%
20x10	1448		1454	0.41
	1479	1515	1515	2.37
	1407	1419	1419	0.84
	1308			0.00
	1325			0.00
	1290	1309	1309	1.45
	1388	1397	1397	0.64
	1363	1365	1365	0.14
	1472			0.00
	1356	1409	1409	3.76
20x20	1911	1970	1970	2.99
	1711	1732	1732	1.21
	1844	1870	1870	1.39
	1810	1839	1839	1.57
	1899	1947	1964	3.31
	1875	1921	1921	2.39
	1875	1905	1905	1.57
	1880			0.00
	1840	1909	1909	3.61
	1900			0.00
50x5	2712	2724	2724	0.44
	2808		2812	0.14
	2596	2600	2600	0.15
	2740			0.00
	2837	2849	2849	0.42
	2793	2803	2803	0.35
	2689			0.00
	2667			0.00
	2527	2531	2531	0.15
	2776	2782	2782	0.21

Instâncias	LB7	LB8	LB9	%
50x10	2907	2929	2929	0.75
	2821			0.00
	2801			0.00
	2968	2972	2972	0.13
	2908	2916	2916	0.27
	2941			0.00
	3062			0.00
	2959			0.00
	2795	2801	2801	0.21
	3046			0.00
50x20	3480	3563	3563	2.32
	3424		3433	0.26
	3351	3376	3376	0.74
	3336	3383	3383	1.38
	3313			0.00
	3460	3510	3510	1.42
	3427	3455	3455	0.81
	3383			0.00
	3457			0.00
	3438	3503	3503	1.85
100x5	5437			0.00
	5208	5221	5221	0.24
	5130	5136	5136	0.11
	4963	4988	4988	0.50
	5195	5224	5224	0.55
	5063		5072	0.17
	5198	5215	5215	0.35
	5038	5041	5041	0.05
	5385		5394	0.16
	5272			0.00

Instâncias	LB7	LB8	LB9	%
100x10	5759	5761	5761	0.03
	5345			0.00
	5623	5654	5654	0.54
	5732	5759	5759	0.46
	5431	5438	5438	0.12
	5246	5270	5270	0.45
	5523			0.00
	5556	5567	5567	0.19
	5779			0.00
	5830			0.00
100x20	5851			0.00
	6099	6120	6120	0.34
	6099	6125	6125	0.42
	6072	6103	6103	0.50
	6009			0.00
	6144			0.00
	5991	6002	6002	0.18
	6084			0.00
	5979	5995	5995	0.26
	6298			0.00
200x10	10816			0.00
	10422	10440	10440	0.17
	10886			0.00
	10794			0.00
	10437	10471	10471	0.32
	10255			0.00
	10761		10796	0.32
	10663			0.00
	10348			0.00
	10616			0.00

Instâncias	LB7	LB8	LB9	%
200x20	10979	11022	11022	0.39
	10947	10969	10969	0.20
	11150	11212	11212	0.55
	11127	11146	11146	0.17
	11132			0.00
	11085	11107	11107	0.19
	11194	11221	11221	0.24
	11126	11195	11195	0.61
	10965			0.00
	11122	11188	11188	0.58
500x20	25922			0.00
	26353	26374	26374	0.00
	26320			0.00
	26424			0.00
	26181	26243	26243	0.00
	26401			0.00
	26300	26304	26304	0.00
	26429	26492	26492	0.00
	25891			0.00
	26315	26376	26376	0.00

Apêndice B

Lista de Resultados

Este apêndice apresenta os resultados obtidos pelo *A-Team* (d) da figura 5.14 para as instâncias testadas.

Tabela B.1 Resultados para testes efetuados em um processador.

	A-Team	
	Makespan	**Tempo(s)**
20x5	1578	1
	1578	1
	1578	1
20x10	1582	112
	1582	27
	1582	29
50x5(a)	2724	1
	2724	1
	2724	1
50x5(b)	2782	1
	2782	1
	2782	1

Tabela B.1 Resultados para testes efetuados em um processador.

	A-Team	
	Makespan	**Tempo(s)**
100x10	5771	23673
	5774	7013
	5785	10920
500x20	26253	638957
	26439	89797
	26445	487186

Tabela B.2 Resultados obtidos para os testes paralelos para a instância 100x10.

	A-Team		
	Processadores	***Makespan***	**Tempo(s)**
100x10	2	5771	22153
		5781	1190
		5785	559
	4	5771	8162
		5781	3361
		5782	2397
	8	5776	2962
		5779	1211
		5781	1345

Referências

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[Ash70] S. Ashour, "A Branch-and-Bound Algorithm for Flow-Shop Scheduling Problems.", *AIIE Trans.*, 2:172-176, 1970.

[Bak74] K. R. Baker, *Introduction to Sequencing and Scheduling*. John Wiley, New York, 1974.

[Bar92] R. G. Barry, *Atmosphere, weather and climate*. Routledge, London, 1992.

[BD62] R. Bellman and S. Dreyfus, *Applied Dynamic Programming*. Princeton university Press, Princeton, NJ, 1962.

[BG76] M. C. Bonney and S. W. Gundry, "Solutions to the Constrained Flowshop Sequencing Problem.", *Opl. Res. Q.*, 27(4):869-883, 1976.

[BJS92] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali, *Linear Programming and Network Flows*. John Wiley & Sons, 1992.

[BM76] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*. American Elsevier Publishing Co. New York, 1976.

[BM81] M. Ball and M. Magazine, "The Design and Analysis of Heuristics.", *Networks*, 11(2):215-219, 1981.

[BP91] V. S. Bandami and C. M. Parks, "A Classifier Based Approach to Flow Shop Scheduling.", *Computers ind. Engng.*, 21(1-4):329-333, 1991.

[BR84] R. E. Burkard and F. Rendl, "A termodynamically motivated simulation procedure for combinatorial optimization problems.", *European Journal of Operational Research*, 17(2):169-174, 1984.

[BT85] E. Balas and P. Toth, "Branch and bound methods." In: *The Travelling Salesman Problem.* E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shimoys, editors. John Wiley & Sons, Chichester, 1985.

[Cal91] D. L. Calloway, "Using a Genetic Algorithm to Design Binary Phase-Only Filters for Pattern Recognition.", *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, Los Altos, CA, 1991.

[Cam94] E. Camponogara, "Times Assíncronos para problemas de Escalonamento Dinâmico.", Proposta de dissertação de mestrado, Departamento de Ciência da Computação, Universidade Estadual de Campinas - Campinas - SP, 1994.

[Car87] E. Cardozo, "DPSK: a kernel for distributed problem solving.", Ph. D. dissertation, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, 1987.

[Cav94] V. F. Cavalcante, "Times Assíncronos na resolução do Job Shop Scheduling Problem - Heurísticas de Construção", Proposta de dissertação de mestrado, Departamento de Ciência da Computação, Universidade Estadual de Campinas - Campinas - SP, 1994.

[CDS70] H. G. Campbell, R. A. Dudek and M. L. Smith, "A heuristic algorithm for the *n* job *m* machine sequencing problem.", *Management Science*, 16(10):B-630 - B-637

[Che92] C. L. Chen, "Bayesian Nets and A-Teams for Power System Fault Diagnosis.", Ph. D. dissertation, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, 1992.

[CLR90] T. H. Cormem, C. E. Leiserson, and L. R. Rivest, *Introduction to algorithms*. McGraw-Hill, 1990.

[Coo71] S. A. Cook, "On the complexity of theorem-proving procedures." *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971.

[CS89] G. A. Cleveland and S. F. Smith, "Using Genetic Algorithms to Schedule Flow Shop Releases.", *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, Los Altos, CA, 1989.

[CS92] E. Cardozo and J. S. Sichman, "DPSK+P user's manual - C++ Interface, version 1.0", FEE/UNICAMP Internal Report, October 1992.

[Dan77] D. G. Dannenbrig, "An evaluation of flow shop sequencing heuristics.", *Management Science*, 23(11):1174-1182, 1977.

[DFJ54] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson, "Solution of a large-scale travelling salesman problem.", *Operations Resarch*, 2:393-410, 1954.

[DPS92] R. A. Dudek, S. S. Panwalkar and M. L. Smith, "The lessons of Flowshop Scheduling Research.", *Operations Research*, 40(1):7-13, 1992.

[Dun78] W. J. Ducan, *Organizational Behavior*. Houghton Mifflin, Boston, MA, 1978.

[Fox79] M. S. Fox, "Organizational Structuring: Designing Large, Complex Software." Technical Report CMU-CS-79-155, Carnegie Mellon University, 1979.

[FS88] U. Faigle, R. Schrader, "On the convergence of stationary distributions in simulated annealing algorithms." *Inf. Process Letters*, 27:189-194, 1988.

[GJ79] R. M. Garey and D. S. Johnon, *Computers and intractability, a guide to the theory of NP-Completeness*. W. H. Freeman and co., 1979.

[GL93] M Grötschel and L. Lovász, "Combinatorial Optimization: A Survey." DIMACS Technical Report 93-29, May 1993.

[Gli94] M. V. Glienke, "Métodos Multi-Algoritmicos Aplicados a Problemas de Roteamento de Veículos"Proposta de dissertação de mestrado, Departamento de Ciência da Computação, Universidade Estadual de Campinas - Campinas - SP, 1994.

[Glo86] F. Glover, "Future paths for integer programming and links to artificial intelligence." *Computers and Operations Reserch*, 13(5):533-549, 1986.

[Glo89] F. Glover, "Tabu Search I", *Orsa Journal on Computing*, 1(3):190-206, 1989.

[Glo89b] F. Glover, "Candidate list strategies and Tabu Search" CAAI Reseach Report, University of Colorado, Boulder, July 1989.

[Glo90] F. Glover, "Tabu search: a Tutorial." *Interfaces*, 20:74-94, 1990.

[Glo90b] F. Glover, "Tabu Search II", *Orsa Journal on Computing*, 2(1):4-32, 1990.

[Gol89] D. E. Goldberg, *Genetic Algorithms in search, optimization, and machine learning*. Addison Wesley, Reading, MA, 1989.

[GM74] B. E. Gillet and L. R. Miller, "A heuristic algorithm for the Vehicle-Dispatch Problem", *Opns. Res.*, 22(3):340-349, 1974.

[GP79] M. Grötschel and M. W. Padberg, "On the symmetric travelling salesman problem I: inequalities." *Math. Programming*, 16:265-280, 1979.

[GP79b] M. Grötschel and M. W. Padberg, "On the symmetric travelling salesman problem II: liftings theorems and facets." *Math. Programming*, 16:281-302, 1979.

[GP85] M. Grötschel and M. W. Padberg, "Polyedral Theory.", In: *The Travelling Salesman Problem*. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shimoys, editors. John Wiley & Sons, Chichester, 1985.

[GS85] B. L. Golden and W. R. Stewart, "Empirical analysis of heuristics." in *The Travelling-Salesman Problem*, E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shimoys, editors, John Wiley & Sons, Chichester, 1985.

[GSJ76] M. R. Garey, D. S. Johnson, and R. Sethi, "The complexity of flow shop scheduling problem", *Mathematics of Operations Research*, 22(1):117-129, 1976.

[Gup71] J. N. D. Gupta, "A Functional Heuristic Algorithm for the Flowshop Scheduling Problem", *Operational Research Quarterly*, 22(1):39-47, 1971.

[Had94] E. G. Haddad, "Times Assíncronos na resolução do Job Shop Scheduling Problem - Heurísticas de Melhoria" Proposta de dissertação de mestrado, Departamento de Ciência da Computação, Universidade Estadual de Campinas - Campinas - SP, 1994.

[HC91] J. C. Ho and Yih-Long Chang, "A new heuristic for the *n*-job, M-machine flow-shop problem." *European Journal of Operational Research*, 52(2):194-202, 1991.

[HPY80] C. L. Hwang, S. R. Paidy, and K. Yoon, "Mathematical Programming with Multiple Objectives; A Tutorial.", *Computer & Operacional Research*, Great Britain: Pergamon Press, 7:5-31, 1980.

[HR88] T. S. Hundal and J. Rajgopal, "An extension of Palmer's heuristic for the flow shop scheduling problem.", *Int. J. Prod. Res.*, 26(6):1119-1124, 1988.

[HW87] A. Hertz and L. D. de Werra, "Using tabu Search techniques for Graph Coloring.", *Computing*, 39:345-351, 1987.

[IM92] H. Inouchi and N McLoughlin, *Parallel techniques for image processing and artificial neural network simulations*. Springer-Verlag, London, 1992.

[IS65] E. Ignal and L. Schrange, "Aplication of the Branch-and-Bound Technique to Some Flow-Shop Scheduling Problems." *Opns. Res.*, 13:400-412, 1965.

[JG91] J. M. Juram and F. M. Gryna, *Controle da Qualidade*. MAKRON, McGraw-Hill, 1991.

[Joh54] S. M. Johnson, "Optimal Two and Three-Stage Production Schedules With Setup Times Included.", *Naval Res. Logist. Quart.*, 1(1):61-68, 1954.

[JRR94] M. Jünger, G. Reinelt, and G. Rinaldi, *The Travelling Salesman Problem*. Heidelberg:IWR, march 1994, (preprint).

[KGV83] S. Kirkpatrick, Jr. C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing.", *Science*, 220(4598):671-680, 1983.

[KH67] R. M. Karp and M. H. Held, "Finite state processes and dynamic programming.", *SIAM J. Appl. Math*, 15:693-718, 1967.

[KK88] V. Kumar and L. Kanal, "The cdp: A unifying formulation for heuristic search, dynamic programming, and branch and bound.", In: *Search in Artificial Intelligence*. L. Kanal and V. Kumar, editors, Springer-Verlag, New York, 1988.

[KKK92] S. Karabati, P. Kouvelis and A. S. Kiran, "Games, Critical Paths and Assignment Problems in Permutation Flow Shop and Cyclic Scheduling Flow Line Environments.", *Journal of Opl. Res. Soc.*, 43(3):241-258, 1992.

[KS80] J. R. King and A. S. Spachis, "Heuristics for flow-shop scheduling.", *Int. J. Prod. Res.*, 1980, 18(3):345-357, 1980.

[KSSTT89] M. Ya. Kovalev, Ya. M. Shafranskij, V. A. Strusevich, V. S. Tanaev and A. V. Tuzikov, "Approximation Scheduling Algorithms: A Survey", *Optimization*, 20(6):859-878, 1989.

[LA87] P. J. M. van Laarhoven and E. H. L. Aarts, *Simulated Annealing: Theory and Applications*. Reidel, 1987.

[Lag95] M. Laguna, *Tabu Search Tutorial*, II Escuela de Verano Latino-Americana de Investigacion Operativa, Mendes, RJ, Janeiro, 1995.

[Lap92] G. Laporte, "The vehicle routing problem: An overview of exact and aproximate algorithms.", *European Journal of Operational Research*, 59(3):345-358, 1992.

[Lau92] C. Lau, editor, *Neural Networks*, IEEE Press, New York, NY, 1992.

[LLK78] E. L. Lawler, J. K. Lenstra and A. H. G. Rinnooy Kan, "A General Bounding Scheme for the Permutation Flow-Shop Problem.", *Opns. Res.*, 26(1):53-67.

[LLKS85] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shimoys, editors, *The Travelling Salesman Problem*. John Wiley & Sons, Chichester, 1985.

[Lon94] H. J. Longo, "Resolvendo problemas de Recobrimento/Particionamento de um Conjunto por A-Teams." Proposta de dissertação de mestrado, Departamento de Ciência da Computação, Universidade Estadual de Campinas - Campinas - SP, 1994.

[McB67] G. B. McMahon and P. G. Burton, "Flow-Shop Scheduling with the Branch-and-Bound Method.", *Opns. Res.*, 15:473-481, 1967.

[McR86] J. L. McClelland and D. E. Rumelhart, editors, *Parallel Distributed Processing*. MIT Press, Cambridge, MA, Vol. 2, 1986.

[ML93] B. L. Maccarthy and J. Liu, "Addressing the gap in scheduling resarch: a review of optimization and heuristic methods in production scheduling", *Int. Journal of Prod. Research*, 31(1):59-79, 1993.

[MRRTT53] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller and E. Teller, "Equation of State Calculations by Fast Computing Machines.", *Journal of Chem. Physics*, 21, 1953.

[MT90] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Chichester, 1990

[Muh90] H. Mühlenbein, "Parallel genetic algorithms, Population Genetics and Combinatorial Optimization.", *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 1989.

[Mül81] H. Müller-Merbach, "Heuristics and their design: a survey.", *European Journal of Operational Research*, 8(1):1-13, 1981.

[Mur92] S. Murthy, "Synergy in cooperation agents: designing manipulators from task specifications", Ph. D. dissertation, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, 1992.

[NEH83] M. Nawaz, E. E. Enscore and I. Ham, "A heuristic algorithm for the m-machine, n-job flow shop sequencing problem", *OMEGA, Int. J. of Management Science*, 11(1):91-95, 1983.

[NW88] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. John Wiley, Chichester, UK, 1988.

[OS90] F. A. Ogbu and D. K. Smith, "The application of the simulated annealing algorithm to the solution of the $n/m/C_{max}$ flowshop problem.", *Computers Opns. Res.*, 17(3):243-253, 1990.

[OW78] G. F. Oster and E. O. Wilson, *Caste and Ecology in the Social Insects*. Princenton University Press, Princenton, NJ, 1978.

[Pag61] E. S. Page, "An Approach to Scheduling of Jobs on Machines", *Journal of the Royal Statistical Society, Series B*, 23:484-493, 1961.

[Pal65] D. S. Palmer, "Sequencing jobs through a multi-stage process in the minimum total time - a quick method to obtaining a near optimum.", *Operations Research Quarterly*, 16(1):101-107, 1965.

[PPE84] Y. B. Park, C. D. Pegden and E. E. Enscore. "A survey and evaluation of static flow-shop scheduling heuristics.", *International Journal of Production Research*, 22(1):127-141, 1984.

[PR88] R. G. Parker and R. L. Rardin, *Discrete Optimization*. Academic Press, Boston, MA, 1988.

[PR91] R. G. Parker and R. L. Rardin, "A Branch-and-Cut algorithm for the resolution of the large-scale symmetric travelling salesman problem.", *SIAM Review*, 33(1):60-100, 1991.

[PS82] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[PS88] F. P. Preparata and M. I. Shamos, *Computacional Geometry: an Introduction*. Springer-Verlag, New York, NY, 1988.

[PS94] H. P. Peixoto and P. S. de Souza, "Times Assíncronos: uma nova técnica para o *Flow Shop Problem.*" In: *Anais do XXI Seminário Integrado de Sotware e Hardware - SEMISH*, Caxambu - MG, 1994.

[PS94b] H. P. Peixoto and P. S. de Souza, "Uma Metodologia de Especificação de Times Assíncronos." In: *Anais do XXVI Simpósio Brasileiro de Pesquisa Operacional*, Florianópolis - SC, 1994.

[Pyo85] S. S. Pyo, "Asynchronous Algorithms for Distributed Processing." Ph. D. dissertation, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, 1985.

[Ree93] C. R. Reeves, "Improving the efficiency of Tabu Search for Machine Sequencing Problems.", *J. Opl. Res. Soc.*, 44(4):375-382, 1993.

[Rod94] R. de F. Rodrigues, "Times Assíncronos para problemas de Otimização Combinatória com Múltiplas Funções Objetivo", Proposta de dissertação de mestrado, Departamento de Ciência da Computação, Universidade Estadual de Campinas - Campinas - SP, 1994.

[RR92] S. Rajasekaran and J. H. Reif, "Nested annealing: a provable improvement to simulated annealing." *Theorical Computer Science*, 99:157-176, 1992.

[Rut89] R. A. Rutenbar, "Simulated annealing algorithms: an overview.", *IEEE Circuits and Devices Magazine*, 5(1), January 1989.

[SGD86] F. Szidarovszky, M. E. Gershon, and L. Duckstein, "Techniques for Multiobjective Decision Making in Systems Management", Advances in Industrial Engineering 2, Netherlands: Elsevier Science, 1986.

[Sou93] P. S. de Souza, "Asynchronous Organizations for Multi-Algorithm Problems.", Ph. D. dissertation, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, 1993.

[SS82] J. P. Stinson and A. W. Smith. "A heuristic programming procedure for sequencing the static flowshop.", *Int. J. Prod. Res.*, 20(6):753-764, 1982.

[ST91] P. S. de Souza and S. N. Talukdar, "Genetic Algorithms in Asynchronous Teams." *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, Los Altos, CA, 1991.

[ST93] P. S. de Souza and S. N. Talukdar, "Asyncronous Organizations for Multi-Algorithm Problems.", *ACM Symposiun on Applied Computing*, Indianapolis, In, February, 1993.

[Tai90] E. Taillard, "Some efficient heuristic methods for the flow shop sequencing problem", *European Journal of Operational Research*, 47(1):65-74, 1990.

[Tai93] E. Taillard, "Benchmarks for basic scheduking problems.", *European Journal of Operational Research*, 64(2):278-285, 1993.

[Tal93] S. N. Talukdar, "Asynchronous Teams." In: *Fourth Synposium on Expert Systems Application to Power Systems*, January 1993.

[TS90] S. N. Talukdar and P. S. de Souza, "Asyncrhonous Teams." In: *Second SIAM Conf. on Linear Algebra: Signals, Systems and Control*, November 1990.

[TS92] S. N. Talukdar and P. S. de Souza, "Scale efficienty organizations." In: *IEEE Int. Conference on Systems, Man and Cybernetics*, October 1992.

[TS93] S. N. Talukdar and P. S. de Souza, *Objects organizations and super-objects*. In: Engineering Design: the creation of products and processes. MacGraw Hill, 1993.

[Wei75] P. Weiner, "Heuristics.", *Networks*, 5(1):101-103, 1975.

[WH89] D. de Werra and A. Hertz, "Tabu Search Techiniques: A tutorial and an Aplication to Neural Networs." *OR Spektrum*, 11:131-141, 1989.

[WH89b] M. Widmer and A. Hertz, "A new heuristic method for the flow shop sequencing problem.", *European Journal of Operational Research*, 41(2):186-193, 1989.

[ZE81] S. H. Zanakis and J. R. Evans, "Heuristic "optimization": why, when and how to use it.", *Interfaces*, 11(5), 1981.

[ZE81] S. H. Zanakis, J. R. Evans, and A. A. Vazacopoulos, "Heuristic methods and applications: A categorized survey.", *European Journal of Operational Research*, 43(1):88-110, 1989.