Times Assíncronos para o *Job Shop Scheduling Problem*: heurísticas de construção

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Victor Fernandes Cavalcante e aprovada pela comissão julgadora.

Campinas, 19 de junho de 1995

Prof. Dr. Pedro Sérgio de Souza

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, 19 **de** 06 **de** 1995

Pela Banca Examinadora composta pelos Profs. Drs.

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

Prof(a). Dr(a). ALBERTO ELFES

Prof(a). Dr(a). PEDRO SÉRGIO DE SOUZA

Times Assíncronos para o *Job Shop Scheduling Problem*: heurísticas de construção[1]

Victor Fernandes Cavalcante[2]

Departamento de Ciência da Computação (DCC)
IMECC - UNICAMP

Banca Examinadora:

- Pedro Sérgio de Souza (Orientador)[3]
- Marcus Vinicius Soledade Poggi de Aragão[3]
- Alberto Elfes[4]
- Ricardo Dahab (suplente)[3]

---

1. Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da UNICAMP, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.
2. Bacharel em Ciência da Computação pela Universidade Estadual do Ceará.
3. Professor do Departamento de Ciência da Computação (DCC) - IMECC - UNICAMP.
4. Diretor do Instituto de Automação do Centro Tecnológico para Informática (CTI).

FICHA CATALOGRÁFICA ELEBORADA PELA
BIBLIOTECA DO IMECC

Cavalcante, Victor Fernandes

C314t Times Assíncronos para o Job Shop Scheduling Problem: heurísticas de construção / Victor Fernandes Cavalcante - Campinas, [SP: s.n.], 1995.

Orientador: Pedro Sérgio de Souza
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Matemática Estatística e Ciência da Computação.

1. Otimização combinatória. 2. Heurística. 3. Pesquisa Operacional. I. Souza, Pedro Sérgio. II. Universidade Estadual de Campinas. Instituto de Matemática, Estatística e Ciência da Computação. III. Título.

Dedicatória

Àquele que, paradoxalmente, ensina-me a ignorância do saber e que, por puro desconhecimento, apelidamos de Deus.

Agradecimentos

A meus pais, Darlene e Roberto, pelos seus ombros, que se não suportam o mundo, suportam ideais, inquietudes e saudades de um filho ausente.

A meus irmãos, Roberta, Fábio e Lucas, pelas palavras de carinho e os silêncios de compreensão.

A Adilson, Terezinha, Sandra e Cristina, pelas constantes lembranças do que vem a ser um lar e por não me deixarem esquecer que o calor nordestino brota em noites de frio.

À Cris, presente maior nesta minha jornada, cujos sorrisos e palavras de otimismo me tornaram ausentes as pedras do caminho.

Aos amigos Paulo, Elane, Rita, Dionísio, Marne, Vaneuza e Telmo, e a meus parentes, pelo crédito depositado à distância.

À Tiquinha, cuja fé e força me fazem certo da pequenez do meu conhecimento.

Ao meu orientador, Pedro Sérgio de Souza, pela seriedade com que acompanhou este trabalho e a amizade dispensada.

Aos companheiros de morada, Pedro, Carlos e Ronaldo, pelas discussões técnicas e, sobretudo, as filosóficas.

Aos colegas com os quais tive o prazer de conviver ao longo deste trabalho.

Aos conterrâneos de fato, Inês, Josi, Ronaldo, Marcão e Flávio, e aos de convivência, Pedro Rafael e Carlos Roberto, pelos momentos saudosistas de descontração.

Aos amigos do mestrado, particularmente: Anderson, Hélvio, Elaine, Wesley, Carrilho, Humberto, Kelner e Rosiane, pelos valiosos momentos desfrutados juntos, sem os quais esta caminhada seria bem mais difícil.

Aos casais Marcus/Andréa e Ricardo/Cristina pela amizade ressaltada em momentos importantes.

Aos amigos Pedro Rafael e Ronaldo, pelos momentos alegres compartilhados e, principalmante, pelos momentos difíceis atenuados.

À Inês pelo ombro amigo.

Aos professores que contribuiram direta ou indiretamente com a minha formação durante os dois últimos anos. Particularmente, ao professor e ser humano Cláudio Lucchesi, pelo crucial apoio e confiança em momentos de desencontros, sem os quais este trabalho sequer teria se iniciado.

Aos funcionários desta universidade, em particular ao Luiz, Alda e Solange, pela competência e atenção dispensada.

Às demais pessoas que contribuiram de alguma forma com a realização deste trabalho.

Resumo

Times Assíncronos consistem numa nova técnica para solução aproximada de problemas que tem sido aplicada com sucesso a problemas de Otimização Combinatória. Esta técnica faz uso de diversos algoritmos heurísticos que cooperam entre si e conseguem encontrar soluções que não seriam encontradas pelos mesmos algoritmos quando executados isoladamente.

Este trabalho tem como objetivo averiguar a adequabilidade de Times Assíncronos como metodologia para solução do problema de escalonamento de tarefas conhecido por *Job Shop Scheduling Problem (JSP)*. Este problema é considerado um dos mais complexos dentro da Otimização Combinatória e tem recebido crescente atenção nas últimas décadas devido, principalmante, à sua aplicabilidade a processos industriais.

Especificamente, o cerne do presente trabalho foi a elaboração de Times Assíncronos centrados fundamentalmente em heurísticas de construção para o *Job Shop Scheduling Problem*. Foram concebidas e testadas novas heurísticas para o JSP e novos fluxos de dados que podem ser facilmente acoplados à arquitetura de um Time Assíncrono.

Os Times Assíncronos desenvolvidos foram submetidos a diversas instâncias do JSP. Os bons resultados obtidos, não somente atestaram a viabilidade da nova técnica como ferramenta para solução do JSP, como revelaram a competitividade destes resultados com aqueles produzidos por outros métodos aproximados para o problema.

Abstract

Asynchronous Teams (or A-Teams) are a new problem resolution technique that has been succesfully applied to Combinatorial Optimization problems. This technique uses several heuristic algorithms that cooperate simultaneously with each other and find solutions that would not be found through isolated algorithms.

The objective of this work is to verify the suitability of Asynchronous Teams methodology solving the combinatorial problem known by Job Shop Scheduling Problem (JSP). This problem has been appointed as one of the most complex problem of Combinatorial Optimization and has been received special attention due to your industrial applicability.

Specifically, the kernel of this work was the implementation of A-Teams based on construction heuristics for the Job Shop Scheduling Problem. New heuristics for the JSP were developed and new data flows that can be easily incorporated in an A-Team architecture were elaborated.

Several JSP instances were used to test the A-Teams developed. The good results obtained by these A-Teams not only showed the feasibility of such technique solving the JSP, but also revealed that this results are competitive with others one obtained by good aproximated approachs for the JSP.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1 *Introdução*

A escassez crescente de recursos (sobretudo aqueles não renováveis) tem, paulatinamente, estimulado o comportamento produtivo da sociedade humana. Em acréscimo a este fato, o processo competitivo e, conseqüentemente, a maior agilização em vários setores socio-econômicos têm revelado a necessidade de obtenção de qualidade dentro da sociedade moderna. É dentro deste contexto que nasce a otimização como agente indispensável à modernização das organizações produtivas.

As raízes históricas de otimização, e mais especificamente da Otimização Combinatória, jazem em problemas econômicos: o planejamento e gerenciamento de operações e o uso eficiente de recursos. Gradualmente mais aplicações técnicas foram enfocadas e modeladas como problemas de otimização combinatória, tais como seqüenciamento de máquinas, escalonamento de produção e projeto e *layout* de instalações de produção [GL93].

Em síntese, otimizar significa encontrar o máximo ou mínimo de uma certa função, definida em algum domínio. Deste ângulo, o alvo da Otimização Combinatória - onde o domínio é tipicamente finito - poderia ser trivial caso pensássemos que necessitamos "apenas escolher a melhor dentre um número finito de possibilidades". Entretanto, enumerar todas as possibilidades para encontrar a melhor dentre elas torna-se impraticável, mesmo para instâncias de tamanho moderado de problemas combinatórios.

Problemas de Otimização Combinatória constituem um conjunto diversificado e são caracterizados pela inerente dificuldade de solucionamento. Portanto, é notória a importância de se ter métodos adequados a vários destes problemas, dado que eles aparecem com freqüência no cotidiano do mundo acadêmico e industrial [PS82]. No

escopo de teoria de complexidade, o que se deseja é que estes métodos sejam capazes de encontrar valores ótimos em tempo polinomial.

Dentro de Otimização Combinatória, os problemas de escalonamento de tarefas têm recebido atenção especial nas últimas décadas devido à vasta aplicabilidade destes problemas a processos industriais.

Problemas de escalonamento estão estreitamente associados ao conceito de complexidade dos algoritmos que os resolvem de forma ótima [TT93]. Aqueles para os quais existem algoritmos com solução ótima de complexidade polinomial são ditos problemas polinomiais P, enquanto problemas que atualmente só podem ser resolvidos por algoritmos de complexidade não deterministicamente polinomial são ditos problemas NP-Difíceis [GJ79]. É óbvio que muito freqüentemente o tempo que podemos dispensar à solução de problemas de escalonamento particulares é seriamente limitado, tal que apenas algoritmos de tempo polimonial de baixa ordem podem ser usados.

Especificamente, se um problema é "fácil" (P), ele pode ser resolvido com um algoritmo capaz de encontrar o valor ótimo, sendo que neste caso as pesquisas enfocam a redução da complexidade do algoritmo. Quanto a problemas NP-Difíceis, existem quatro direções a serem seguidas [TT93, BESW93]:

1. Relaxação: podemos relaxar algumas restrições do problema original e resolver o problema relaxado. O resultado pode ser uma boa aproximação para a solução do problema original. No caso de problemas de escalonamento algumas relaxações cabíveis seriam.

   1.1. permitir preempções, mesmo se o problema original trate escalonamentos não preemptivos;

   1.2. supor tarefas de tamanho unitário, quando tarefas arbitrárias são consideradas no problema original;

   1.3. supor certos tipos de grafos de precedência (por exemplo, árvores) mesmo quando grafos arbitrários foram considerados no problema original.
2. Aproximação: o problema é resolvido com a ajuda de algumas heurísticas. Geralmente a qualidade de métodos aproximados é avaliada com base em análises de desempenho no pior caso e no caso médio. Esta abordagem, apesar de não ter compromisso com otimalidade, tende a encontrar soluções ótimas para alguns problemas combinatórios.
3. Enumeração: esta abordagem é seguida apenas se queremos encontrar a solução verdadeiramente ótima. A implicação aqui é a crescente complexidade (usualmente exponencial) dos algoritmos deste tipo. As técnicas usadas neste tipo de otimização inclui programação dinâmica, *branch and bound*, *branch and cut*, entre outros [CP89,Las92,LY92].
4. Sistemas especialistas: têm sido crescentemente empregados na solução de problemas de escalonamento NP-Difíceis [BPN95,LL93]. Usualmente ajudam na combinação efetiva de mais que uma heurística.

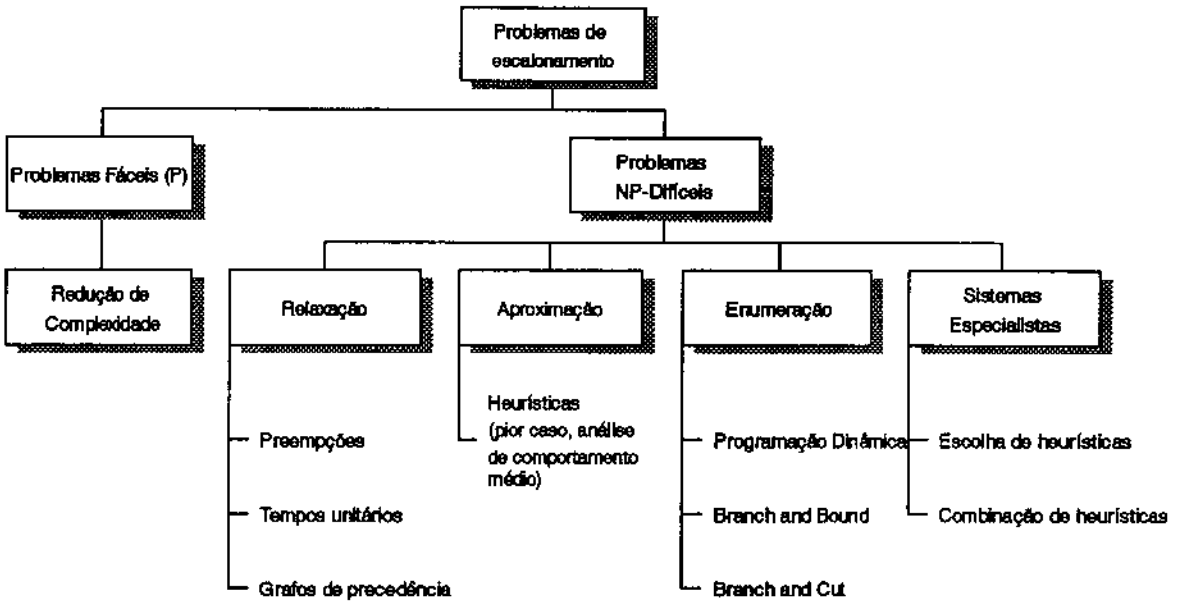


Figura 1.1 Visão esquemática de uma taxonomia para problemas de escalonamento e de metodologias associadas para solução de problemas.

A discussão anterior é sumarizada de forma esquemática na figura 1.1. No esquema são utilizadas soluções para problemas de escalonamento como alvo das diversas metodologias.

Segundo Spachis e King [SK79], recentes pesquisas relacionadas à teoria de complexidade computacional indicam que, exceto para um número limitado de situações especiais, é extremamente improvável que soluções ótimas para problemas de escalonamento serão possíveis, exceto por métodos de enumeração parcial tipo *branch and bound*. Entretanto, os requisitos computacionais neste caso são tais que, mesmo com modernos computadores, o montante de tempo necessário torna-se proibitivo, principalmente em se tratando de problemas de escalonamento de tamanhos tipicamente industriais. Esta colocação reforça a utilização de abordagens aproximadas como a alternativa mais viável na busca de soluções para problemas de porte industrial.

Especificamente, este trabalho aborda o problema de escalonamento conhecido por *Job Shop Scheduling Problem* (JSP), o qual é considerado um dos mais complexos dentro da Otimização Combinatória [BJ93].

Como metodologia para solução do JSP é empregada uma nova técnica para resolução aproximada de problemas, denominada Times Assíncronos, que foi aplicada com sucesso por Souza [Sou93] ao problema do Caixeiro Viajante. Esta técnica faz uso

simultâneo de várias heurísticas que compartilham dados e soluções em memórias compartilhadas, efetuando comunicações de forma assíncrona.

Basicamente, o objetivo desta dissertação é investigar a adequabilidade de Times Assíncronos como metodologia para solução do *Job Shop Scheduling Problem*. Mais precisamente, o cerne deste trabalho consiste na elaboração de Times Assíncronos para o JSP centrados fundamentalmente em heurísticas de construção para o problema.

A opção por um enfoque dirigido ao aspecto construtivo para o JSP é motivada, principalmente, pela dificuldade inerente ao problema de construção de boas soluções para o problema. Além disto, a diversidade de abordagens para o problema centradas em construção constitui-se uma motivação adicional para adoção desse enfoque.

Sumariamente, os demais capítulos estão assim organizados:

No capítulo 2 são apresentadas definições, conceitos e elementos básicos associados a problemas de escalonamento e, mais precisamente, ao *Job Shop Scheduling*. São introduzidas nomenclaturas, formulações, notações e representações relativas ao problema que são adotadas ao longo deste trabalho.

No capítulo 3 são descritas diversas metodologias aproximadas aplicáveis ao JSP. Diversos métodos heurísticos e meta-heurísticos comumente empregados a problemas de escalonamento (e combinatórios em geral) são apresentados de uma forma genérica. Em acréscimo a estes, são descritas algumas heurísticas clássicas para solução do *Job Shop Scheduling*, assim como abordagens mais comumente utilizadas na solução de problemas de escalonamento.

O capítulo 4 introduz conceitos acerca da técnica de Times Assíncronos e descreve os resultados experimentais obtidos com a aplicação desta técnica ao JSP. As heurísticas de construção utilizadas e desenvolvidas são detalhadamente descritas, assim como os algoritmos concebidos especificamente para utilização no times construídos. Finalmente, os resultados obtidos com a implementação são analisados e confrontados com demais resultados disponíveis na literatura.

O capítulo 5 apresenta as conclusões e as contribuições relativas a este trabalho, bem como sugestões de trabalhos futuros.

Capítulo 2

Problemas de Escalonamento: o Job Shop Scheduling

Problemas de escalonamento de tarefas têm recebido uma grande atenção nas últimas décadas. Parte da atenção dispensada provém do interesse teórico de pesquisadores em áreas como otimização combinatória, análise de complexidade e análises de pior caso, entre outras. Em acréscimo a este interesse teórico, o valor prático a eles atribuído (aplicabilidade a sistemas flexíveis de manufatura e sistemas multiprocessadores, por exemplo) tem representado uma motivação adicional ao desenvolvimento de métodos satisfatórios para solução de tais problemas [TT93].

Em geral, pode-se entender problemas de escalonamento como problemas de alocação de recursos ao longo do tempo, cuja finalidade é a execução de um conjunto de tarefas.

Os elementos básicos de problemas de escalonamento são:

- Um conjunto T de n tarefas: $T = \{T_1, T_2, ..., T_n\}$;
- Um conjunto P de m processadores: $P = \{P_1, P_2, ..., P_m\}$.

Além destes, na maioria dos modelos gerais de escalonamento também é considerado um conjunto R de recursos adicionais (outros que não processadores).

Há duas restrições gerais na teoria clássica de escalonamento:

1. cada tarefa pode ser processada por no máximo um processador a cada instante;
2. cada processador é capaz de executar no máximo uma tarefa por vez.

Diferentes critérios que medem a qualidade do desempenho de um conjunto de tarefas são também levados em consideração. Assim, para que problemas de escalonamento de tarefas sejam formulados completamente, são necessárias algumas informações

específicas pertinentes ao conjunto de tarefas T, ao conjunto de processadores P e ao critério de desempenho que se deseja otimizar (conhecido como função objetivo).

2.1 Caracterizações de problemas de escalonamento

São apresentadas a seguir algumas caracterizações dos elementos que compõem problemas de escalonamento de tarefas. Estas caracterizações se fazem necessárias a fim de que a estrutura destes problemas seja bem compreendida.

2.1.1 Conjunto de tarefas

Cada tarefa $T_j \in T$ pode ser caracterizada pelos seguintes dados:

(I) Um vetor de tempo de processamento $p_j = [p_{j1}, p_{j2}, \ldots, p_{jm}]$, onde p_{ji} é o tempo requerido pelo processador P_i, $i = 1,2,\ldots,n$, para executar a tarefa T_j;

(II) Um *ready time* $r_j \geq 0$ que indica o instante de tempo após o qual a tarefa T_j encontra-se disponível para ser executada em algum processador;

(III) Uma *due date* d_j, que declara o instante de tempo desejado como limite para término de execução da tarefa T_j;

(IV) Uma ponderação (prioridade) w_j que retrata a importância relativa da tarefa T_j.

Observe que os dados acima constituem atributos de cada tarefa individualmente. Em acréscimo a estes, para o conjunto de tarefas como um todo são especificadas as seguintes questões:

- O modo de processamento das tarefas pode ser "preemptivo" ou "não preemptivo". Um escalonamento é dito preemptivo se qualquer tarefa pode ser interrompida a qualquer momento, ou seja, pode ter sua execução suspensa por um montante limitado de tempo, e sua continuação ocorrer num instante posterior (no mesmo ou em outro processador) sem custo adicional.
- As tarefas de T são ditas dependentes caso a ordem de execução de pelo menos duas tarefas de T esteja restrita por relações de precedência. A notação $T_i < T_j$ indica que a execução da tarefa T_i deve ser finalizada antes que a tarefa T_j inicie sua execução. Caso inexista a relação de precedência entre tarefas, estas são ditas independentes.

Restrições de precedência são comumente representadas por meio de grafos dirigidos acíclicos (GDA). Os grafos devem ser acíclicos a fim de que impasses (*deadlocks*) na execução das tarefas sejam evitados. A figura 2.1 ilustra um grafo de precedência simples, onde os nós do grafo representam as tarefas e os arcos dirigidos representam as restrições de precedência. Note que, por exemplo, a tarefa T_6 só pode ser iniciada após a conclusão de todas as demais tarefas, excetuando-se a tarefa T_4, dado que esta não guarda nenhuma relação de precedência com aquela.

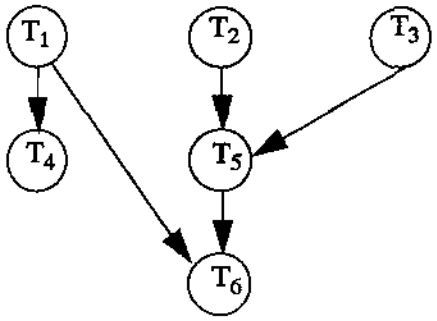


Figura 2.1 Grafo de precedência entre tarefas.

2.1.2 *Conjunto de processadores*

Processadores que podem executar qualquer tarefa são designados "paralelos", enquanto que, se cada processador é especializado na execução de determinadas tarefas, eles são ditos "dedicados".

Os processadores paralelos podem ser distinguidos, de acordo com suas velocidades, em três tipos [BESW93,TT93]:

- Idênticos: todos os processadores de P têm a mesma velocidade de processamento. A velocidade de cada processador é independente da tarefa que está sendo executada.
- Uniformes: os processadores de P têm velocidades distintas e constantes. A velocidade de cada processador é independente da tarefa que está sendo realizada.
- Não Relacionados: a velocidade de cada processador é variável, mas depende da tarefa que está sendo executada.

Assim se para toda tarefa T_j, $j = 1,2,...,n$ é mantida a relação $p_{ij} = p_j$, $i = 1,2,...,m$, então temos processadores idênticos. Enquanto se $p_{ij} = p_j b_i$, $i=1,2,...,m$, onde b_i^{-1} é uma medida de velocidade de P_i independente de j, então temos processadores uniformes [TT93,BESW93].

Processadores dedicados podem operar em um dos seguintes modos [BESW93,Gra81]:

- *Flow Shop*: cada tarefa de T deve ser executada por todos os processadores, na mesma ordem, sendo esta previamente definida.
- *Open Shop*: difere do *FlowShop* quanto à ausência de uma ordem de execução predeterminada das tarefas.
- *Job Shop*: cada tarefa de T deve ser executada por um subconjunto de processadores de P. A ordem de execução da tarefa é arbitrária e, claro, pode ser diferente de tarefa para tarefa. Contudo, esta ordem deve ser dada como entrada para o problema.

2.1.3 *Critérios de desempenho*

Todo escalonamento, seqüência que representa a solução de algum problema de escalonamento, é caracterizado por algum critério ou medida de desempenho. Tal medida consiste em um qualificador do escalonamento baseado na função objetivo associada ao problema. O escalonamento que minimiza (ou maximiza) determinado critério é dito ser ótimo com respeito a este critério. Alguns dos critérios mais utilizados como função objetivo para problemas de escalonamento são [BESW93,WC92]:

(I) Tamanho do escalonamento (*makespan*): $C_{max} = Max_j\{C_j\}$

(II) Tempo médio de fluxo: $\bar{F} = \frac{1}{n}\left(\sum_1^n f_j\right)$

(III) Máximo *Lateness*: $L_{max} = Max_j\{L_j\}$

Conforme [ML93] em [Bak74] é ressaltado que o critério (i) é usado quando se deseja uma utilização eficiente de recursos, (ii) está associado a datas e prazos de prescrições (*deadlines*) e (iii) é um bom critério quando se deseja avaliar demanda por resposta rápida.

2.1.4 *Suposições adicionais*

Algumas suposições adicionais freqüentemente adotadas na teoria de escalonamento são [ML93]:

- Máquinas estão sempre disponíveis e nunca quebram.
- *Buffers* entre processadores têm capacidade ilimitada e uma tarefa concluída em uma máquina pode esperar até que seu processamento seja iniciado na próxima máquina.
- *Ready times*, isto é, os momentos em que as tarefas estão disponíveis para serem processadas, são iguais a zero. Logo, todas as tarefas encontram-se disponíveis no começo do processamento.
- Tempos gastos na configuração de máquinas são independentes do escalonamento e estão incluídos no tempo de processamento.
- Tempos de processamento e restrições tecnológicas são determinísticos e previamente conhecidos.

2.2 *O Job Shop Scheduling Problem (JSP)*

2.2.1 *Definição*

Intuitivamente o *Job Shop Problem* pode ser descrito da seguinte forma: serviços (comumente conhecidos por *jobs*), compostos por um conjunto de tarefas, devem ser

processados em um conjunto de máquinas. O objetivo consiste em otimizar alguma função do tempo de término destes serviços, satisfazendo a diversas restrições.

Entretanto, o JSP pode ser definido formalmente de maneira bem mais precisa que a descrição intuitiva acima apresentada. Para tal, supõe-se que tarefas formam n subconjuntos (ou cadeias) chamados de *jobs*. Assim sendo, o JSP é descrito como segue [BJ93]: são considerados n *jobs* $J_1,J_2,..,J_n$ e m máquinas $M_1,M_2,...,M_m$. O Job_j, $(j=1,2,...,n)$ consiste de $n_j \leq m$ tarefas (ou operações) T_{1j}, T_{2j}..., $T_{n,j}$ que devem ser processadas nesta ordem. A cada T_{ij} está associada uma máquina M_{ij} onde T_{ij} é processada. O tempo que T_{ij} leva para ser processada é denotado por p_{ij}, sendo previamente conhecido para cada tarefa a ser escalonada.

O problema alvo deste trabalho supõe também as seguintes premissas:

- a qualquer instante cada tarefa pode ocupar no máximo um processador e cada processador pode executar no máximo uma tarefa;
- preempções não são permitidas;
- cada job_j é constituído de no máximo m tarefas, isto é, $n_j \leq m$, $j=1,2,...,n$.

2.2.2 *Função objetivo*

A função objetivo escolhida como base para avaliação de qualidade é o tamanho do escalonamento construído. Tal medida consiste no tempo total de processamento de todas as tarefas do sistema e é conhecida como ***makespan***, sendo definida por $C_{max} = Max\{C_j\}$, onde C_j é o tempo de término da última tarefa do job_j.

Minimizar o *makespan* é fundamental, uma vez que leva tanto à maximização do fator de utilização das máquinas, como à minimização do tempo máximo em processo do conjunto de tarefas escalonadas [Cof76, BESW93]. Este critério pode também ser de importância em um sistema de controle de computador onde um conjunto de tarefas chega periodicamente e deve ser processado no tempo mais curto possível. Como critério de custo, o *makespan* é útil quando a ordem na qual *jobs* são finalizados não é importante.

De acordo com [WR90], o JSP com mínimo ***makespan*** tem sido o principal veículo para pesquisas acadêmicas no escopo de problemas de escalonamento determinísticos envolvendo múltiplos recursos.

2.2.3 *Complexidade*

É sabido que o JSP é não somente um modelo com uma ampla área de aplicação, mas também um dos problemas mais difíceis computacionalmente [BESW93]. Dentre os problemas de Otimização Combinatória, muitos (inclusive o *Job Shop Problem*) pertencem à classe dos NP-Difíceis [GJ79,Cof76]. Sendo assim, não se conhece

nenhum algoritmo polinomial capaz de resolver todas as possíveis instâncias deste problema.

Na realidade, mesmo instâncias de tamanhos modestos permanecem computacionalmente intratáveis. Esta intratabilidade é uma das razões do problema ter sido e continuar sendo largamente estudado. Segundo [AC91], alguma da motivação em trabalhar no problema surgiu, sem dúvida, do fato de uma instância específica (composta de 10 máquinas e 10 *jobs*) proposta por Muth e Thompson em 1963, ter permanecido sem solução ótima por aproximadamente vinte anos.

O JSP já é NP-Difícil para apenas duas máquinas. Nenhuma heurística com desempenho garantido foi desenvolvida até agora [BESW93]. Todavia, existem alguns casos especiais para os quais algoritmos de tempo polinomial estão disponíves. São eles:

- Duas máquinas e todas as tarefas tendo tempo de processamento unitário. Pode ser resolvido em tempo linear no número total de tarefas.
- Dois *jobs*. Pode ser resolvido em tempo $O(n_1 n_2 \log n_1 n_2)$, onde n_1 e n_2 são os números de tarefas do primeiro e segundo *jobs*, respectivamente.
- Duas máquinas, onde não há mais que duas tarefas. A solução é obtida em tempo $O(n \log n)$ (algoritmo de Jackson).

2.2.4 *Representação gráfica*

O *Job Shop Scheduling*, assim como os demais modelos de escalonamento, pode ter suas soluções representadas graficamente. Para tal, comumente são utilizados os chamados gráficos de Gantt, que constituem uma ferramenta adequada para expressão de soluções de problemas de escalonamento de tarefas de forma clara e simples.

Antes da apresentação de um gráfico de Gantt como representação de uma solução para um JSP, é ilustrada na figura 2.2 uma representação genérica da estrutura dos dados de definição do problema. Na mesma figura, essa estrutura é exemplificada com uma instância para um JSP constituído de três máquinas e dois *jobs* (compostos de três tarefas cada).

A figura 2.3 apresenta um exemplo de um escalonamento gerado a partir dos dados de definição do JSP da figura 2.2. O eixo das ordenadas indica as máquinas onde as tarefas de cada *job* são executadas e o eixo das abscissas marca os instantes de tempo em que cada tarefa foi iniciada e concluída. Note que no gráfico abaixo os espaços achurados indicam intervalos de tempo em que a máquina ficou ociosa e que o valor do *makespan* da solução é de onze unidades de tempo.

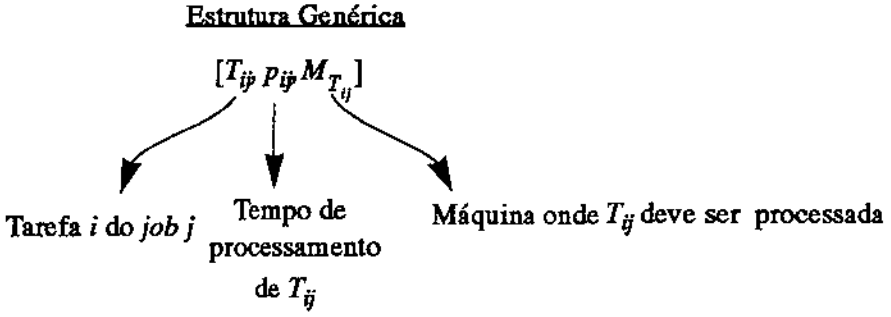


Figura 2.2 Uma representação para os dados de definição de um JSP e uma instância composta de seis tarefas (2 *jobs* x 3 máquinas).

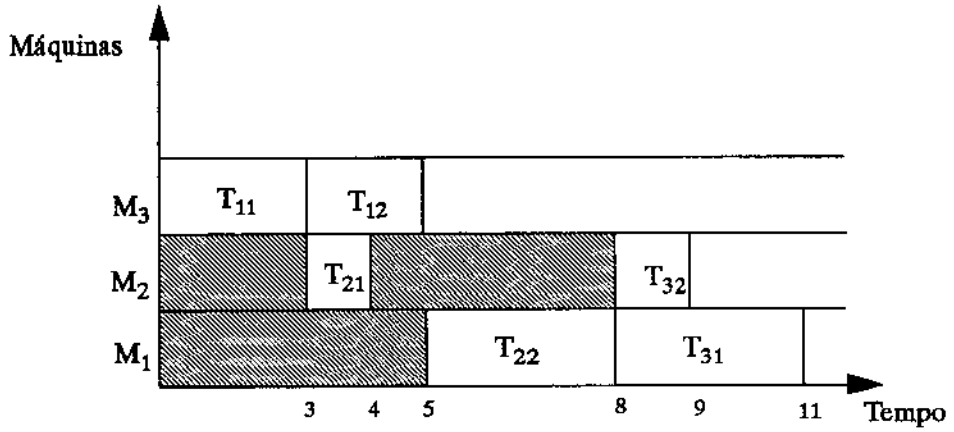


Figura 2.3 Gráfico de Gantt para uma instância 2 *jobs* x 3 máquinas do JSP.

2.3 *Sumário*

Neste capítulo foram apresentadas algumas características gerais de problemas de escalonamento de tarefas. Elementos básicos e representações padrão para este tipo problemas são brevemente descritas e são exemplificados alguns critérios de desempenho freqüentemente usados na avaliação da qualidade de soluções para tais problemas.

Em particular é definido o *Job Shop Scheduling*, problema alvo da presente dissertação. Algumas considerações relativas à dificuldade e à especificidade do problema tratado são efetuadas.

É especificada a medida de qualidade utilizada como função objetivo e são descritas as representações de dados e de soluções comumente utilizadas para o problema em questão.

Capítulo 3

Métodos Heurísticos para o Job Shop Scheduling

O *Job Shop Scheduling Problem* está entre os problemas mais difíceis de Otimização Combinatória. Não somente ele é NP-Difícil, mas mesmo entre os membros desta classe ele aparece como pertencente aos de mais difícil solução [LAL92].

Esta notória intratabilidade do JSP praticamente inviabiliza a aplicação de métodos exatos na solução do problema. Mesmo quando tentando resolver instâncias de tamanhos relativamente pequenos, tais métodos podem requerer um montante proibitivo de esforço computacional [SK79].

Métodos aproximados (ou heurísticos) têm se caracterizado como uma alternativa bastante atrativa a ser empregada na solução de problemas tipicamente combinatórios. Apesar de não estarem compromissados com a obtenção de resultados ótimos, boas estratégias heurísticas tentam produzir soluções com algum grau de proximidade do ótimo, em tempo polinomial.

Dado este panorama, onde abordagens aproximadas surgem como opções razoáveis na solução de problemas difíceis, um crescente esforço tem sido dedicado ao desenvolvimento de boas heurísticas que resolvam tais problemas. Em particular, algoritmos para problemas de escalonamento que sejam rápidos e produzam resultados aceitáveis têm despertado grande interesse, principalmente devido à imediata aplicabilidade destes a processos industriais.

De uma forma ampla, duas abordagens heurísticas têm sido desenvolvidas:

1. Meta-heurísticas: heurísticas de caráter genérico, tipicamente centradas em métodos de busca em vizinhança.

2. Heurísticas especializadas: heurísticas desenvolvidas especificamente para solução de determinado problema. Em geral, heurísticas especializadas são chamadas simplesmente de heurísticas.

A seguir são descritas algumas meta-heurísticas (*Simulated Annealing*, Algoritmos Genéticos e *Tabu Search*) que podem ser aplicadas a problemas de escalonamento. Em seguida apresentaremos algumas heurísticas específicas para o *Job Shop Scheduling Problem* constantes na literatura.

3.1 Meta-heurísticas

3.1.1 Simulated Annealing (SA)

O nome *Simulated Annealing* deve-se à analogia que este método faz entre Otimização Combinatória e a evolução do equilíbrio térmico de sólidos [KGV83,LA87]. A base do método é o chamado algoritmo de Metrópolis [LA87], o qual gera seqüências de estados sólidos da seguinte forma: dado o estado corrente do sólido, uma pequena perturbação gerada aleatoriamente é aplicada. Se a perturbação resultar num estado de energia mais baixa do sólido, o processo prossegue a partir deste novo estado. Em contrapartida, se a energia aumenta, isto é, $\Delta E \geq 0$, então o novo estado é aceito com probabilidade $P = (-\Delta E/(K_B T))$, onde K_B é a constante de Boltzmann e T é a temperatura.

No escopo da Otimização Combinatória, a idéia introduzida com o SA leva a um procedimento iterativo, onde a solução desempenha o papel do estado do sólido e a função custo F (função objetivo) e o parâmetro de controle assumem, respectivamente, os papéis de energia e de temperatura. Dada uma certa solução semente S, o SA aceita com alguma probabilidade a mudança para uma solução inferior (pior conforme a função objetivo considerada). $PA_{SS'}(k)$ representa a probabilidade de aceitação de uma solução S' inferior à solução semente S, estando a solução corrente no estágio k do SA. Assim sendo, o seu valor é calculado como:

$$PA_{SS'}(k) = Min\left[1, exp\left(-\frac{1}{T_k}(F(S) - F(S'))\right)\right]$$

Onde T_k é o valor da temperatura no estágio k.

É definido também um número L_k de transições realizadas neste estágio. Uma vizinhança V_S da solução semente S (isto é, um conjunto de soluções próximas a S) também é considerada.

No decorrer do algoritmo do SA, o parâmetro de controle (temperatura) é diminuido até aproximar-se de zero. Tipicamente o algoritmo é iniciado a uma temperatura T_k

(inicialmente $k = 0$), sendo executado por L_k iterações; a seguir, a temperatura é decrescida por um fator $\alpha < 1$ e $T_{k+1} = \alpha T_k$. O algoritmo termina quando o estado final de equilíbrio térmico é atingido, sendo tomado como solução do problema.

Simulated Annealing

Selecione uma solução S como sendo a corrente e **faça**:
$S^* = S$, $F^* = F(S)$, $k=0$, T_0 = temperatura inicial
Repita até o critério de parada ser satisfeito:
  Repita L_k vezes
    Escolha aleatoriamente uma solução $S' \in V_s$
    Se $F(S') \leq F(S)$
    Então S' torna-se a solução corrente S
      Se $F(S) \leq F^*$
      Então $S^* = S$, $F^* = F(S)$
    Senão obtenha um número aleatório $p \in [0, 1)$
      Se $p < PA_{ss'}(k)$
      Então S' torna-se a solução corrente S
  fim-Repita
  Faça $T_{k+1} = \alpha T_k$ e $k = k + 1$
fim-Repita

3.1.2 Algoritmos Genéticos (AG)

Algoritmos Genéticos foram concebidos com o intuito de imitar alguns dos processos observados na evolução natural [Dav91]. Consistem basicamente em algoritmos de busca baseados no mecanismo natural de reprodução e seleção [Gol89], onde uma população de soluções para um dado problema é melhorada iterativamente.

Assim como *Simulated Annealing* e Redes Neurais, Algoritmos Genéticos podem ser vistos como algoritmos baseados em fortes metáforas do mundo natural [Dav91].

Em síntese, os componentes iniciais para que seja feito uso de um algoritmo genético para produzir uma evolução simulada a partir de uma população de soluções são: (a) um problema; (b) uma forma de codificar soluções para este problema e, (c) uma função que retorne uma medida de qualidade de qualquer codificação efetuada.

Para que AG sejam usados se faz necessário primeiramente codificar as soluções do problema em estruturas chamadas cromossomos - dispositivos orgânicos que contêm a codificação estrutural dos seres vivos.

Cromossomos são constituídos de genes. Os genes usualmente são codificados de forma binária (podem assumir apenas valores 0 e 1). A localização do gene em um cromossomo é denominada *locus*. Cada cromossomo tem um valor associado que

traduz a qualidade da solução por ele representada. Este valor é gerado por uma função que mede a aptidão (*fitness*) do cromossomo.

Operadores genéticos são utilizados sobre os cromossomos. Tradicionalmente são utilizados operadores tais como mutação, que têm o objetivo de manter a diversidade de soluções, e recombinação, responsável pelas novas combinações entre partes de cromossomos a fim de gerar um terceiro (*crossover*).

Algoritmos Genéticos fazem uso de geração de populações. A cada iteração a população é renovada total ou parcialmente, através da inserção de novos elementos avaliados como aptos em lugar de soluções antigas.

Genericamente um Algoritmo Genético pode ser assim descrito [Dav91]:

Algoritmo Genético

Gere uma população inical de soluções (cromossomos).
Avalie cada cromossomo da população.
Repita até o critério de convergência ser satisfeito:
  Gere novos cromossomos a partir da população corrente, utilizando-se de operadores de recombinação e mutação.
  Elimine membros da população corrente, a fim de abrir espaços para novos cromossomos.
  Avalie os novos cromossomos e os insira na população.
fim-Repita
Retorne o melhor cromossomo.

3.1.3 Tabu Search (TS)

Uma das formas mais simples de se entender TS é imaginá-lo como um método descendente com uma função objetivo F de minimização [HW90]. Basicamente, métodos descendentes partem de uma solução inicial S e encontram uma solução S' na sua vizinhança V_S. Se $F(S') < F(S)$, então ocorre um movimento de S para S' e o passo é repetido. Se nenhuma S' que obedeça a inequação é encontrada, o procedimento descendente pára.

Tabu Search parte de uma vizinhança de S e escolhe a melhor solução S^* vizinha (a qual não é necessariamente melhor que a corrente). Entretanto, este movimento de S para S^* pode gerar ciclos entre soluções, isto é, pode permitir que ocorra uma ciclagem entre soluções, possibilitanto retornos, por exemplo, a ótimos locais. Para previnir tais reversões ou, às vezes, repetições de movimentos, é construída uma lista tabu, geralmente implementada como uma estrutura de fila, contendo restrições (movimentos proibidos). Estas restrições não operam isoladamente, sendo contrabalançadas pela aplicação de critérios de aspiração, os quais possibilitam a

aceitação de certos movimentos que constam na lista tabu. Em geral, o algoritmo pára se um número específico de iterações é completado no total ou desde a última mudança da melhor solução encontrada.

Tabu Search [Glo90,HW90,WH89] tem sido empregada na solução de diversos problemas de otimização e o seu algoritmo pode ser descrito simplificadamente como segue :

Tabu Search

Selecione uma solução S, e a adote como melhor conhecida, isto é:

$S^* = S$ e $F^* = F(S)$.

Repita um número fixo de iterações:

Encontre uma solução $S' \in V_S$, tal que $F(S') - F(S)$ seja mínima em V_s.

Se o movimento de S para S' não for tabu ou obedecer a critérios de aspiração.

Então Se $F(S') < F^*$,

$S^* = S'$, $F^* = F(S')$ e

S' torna-se a solução corrente S e a lista tabu é atualizada.

Senão a próxima melhor solução S' em V_s é tomada como corrente.

fim-Repita

3.2 *Heurísticas*

Alguns dos métodos mais utilizados no desenvolvimento de algoritmos heurísticos têm como objeto central de estudo o problema a ser atacado. Estes métodos tentam tirar proveito de características intrínsicas ao problema a ser resolvido (tais como peculiaridades e comportamentos médios do problema).

Segundo Storer *et al.* [SWV92], as heurísticas mais largamente usadas para problemas de escalonamento são aquelas que selecionam uma tarefa a cada passo a fim de construir uma seqüência completa. O procedimento mais simples baseia-se na seleção de tarefas de tal maneira que, uma vez tomada uma decisão, nenhuma reconsideração de seleções alternativas é permitida. Este procedimento é usualmente chamado de método de um passo (ou guloso).

Quanto ao JSP foram identificadas três abordagens heurísticas distintas:

- Baseadas em regras de prioridade: técnica através da qual um número (ou valor) é associado a cada tarefa a ser executada, de acordo com algum método, sendo selecionada a tarefa com menor valor. Esta abordagem tem sido um dos principais recursos empregados na solução de problemas de escalonamento. Ela explora características específicas dos problemas como veículos para obtenção de boas

soluções. Exemplos de regras de prioridade bastante empregadas a problemas de escalonamento são: SPT (*Shortest Processing Time*), MWKR (*Most Work Remaining*), FCFS (*First Come First Served*), RANDOM (aleatória), entre outras;

- Baseadas em classes (ou tipos) de escalonamento: esta abordagem preocupa-se basicamente com escolhas locais de tarefas a serem escalonadas e com a configuração do escalonamento a cada escolha, isto é, com os estados em que o escalonamento se encontra e com a disponibilidade das máquinas e tarefas a serem escalonadas a cada estágio. As escolhas efetuadas devem garantir que o escalonamento a ser produzido seja caracterizado como pertencente à classe alvo;
- Baseadas em "relaxações" do problema original: condições (ou restrições) do problema original são ignoradas e o novo problema gerado é resolvido, sendo sua solução validada para o problema original. Um exemplo clássico desta abordagem é a heurística conhecida como *shifting bottleneck*, desenvolvida por Adams *et al.* [ABZ88].

Descreveremos abaixo algumas das principais heurísticas para o *Job Shop Problem* disponíveis na literatura.

3.2.1 *Heurísticas baseadas em regras de prioridade*

A maioria das pesquisas em heurísticas para escalonamento e seqüenciamento de tarefas têm se concentrado em produzir uma regra específica que obtenha bons resultados quando aplicada pelo menos a uma estreita faixa de circunstâncias [OH85]. O exemplo clássico para este tipo de heurística são as regras de prioridade (também conhecidas por regras de escalonamento ou regras de despacho).

Um vasto trabalho enumerando acima de cem regras de prioridade, suas adequações e formas de aplicações a diversos problemas de escalonamento, foi realizado por Panwalkar e Iskander, e compilado em [PI77]. Este trabalho também cria uma classificação de regras de escalonamento de acordo com diversas categorias adotadas pelos autores.

Regras de prioridade têm sido empregadas de formas bastante distintas, tais como:

- Base única para tomada de decisão local e escolha de tarefas a cada passo de escalonamento.
- Agentes solucionadores de conflitos originários da existência de múltiplas opções de escolha. Ou seja, agem como subsídios na quebra de impasses gerados quando os critérios de seleção disponíveis não são suficientes para determinar qual tarefa deve ser selecionada.
- Estrutura hierárquica de tomada de decisões, onde a cada regra é dada uma prioridade diferente no processo de escolha ao longo do escalonamento.
- Elementos de ponderação ou combinação linear entre diversas regras de prioridade.

Pesquisas têm sido realizadas no intuito de verificar graus de adequabilidade de determinadas regras de prioridade a problemas de escalonamento. Apesar de alguns

trabalhos revelarem experimentalmente a superioridade de certas regras sobre outras (em [HC90] é verificado experimentalmente que, por exemplo, em geral, a utilização da regra MWKR produziu melhores resultados que SPT), existe pouco consenso sobre afirmativas desta natureza.

3.2.2 *Heurísticas baseadas em classes de escalonamento*

Uma abordagem bastante comum na solução de problemas de escalonamento está centrada na elaboração de algoritmos não determinísticos, onde uma amostra de soluções válidas para o problema é gerada e a melhor solução obtida é escolhida. Todavia, duas questões acerca deste tipo de abordagem podem ser rapidamente levantadas: (a) quão abrangente deve ser a amostragem gerada, ou seja, como definir o tamanho ideal do espaço de soluções a serem geradas? e (b) como pode ser garantida certa qualidade das soluções produzidas sem que, contudo, seja demasiadamente restrito o tamanho da amostragem (diminuição do não determinismo do algoritmo gerador) e assim comprometido o espaço de busca do algoritmo?

Dentro deste panorama, têm sido estudadas diversas maneiras de construção de heurísticas para problemas de escalonamento (tais como o JSP) que se enquadrem em determinadas classes de soluções. Além disso, estas classes têm sido objeto de grande interesse, uma vez que a identificação de uma classe que garantidamente contenha soluções de qualidade pode diminuir bastante o esforço computacional gasto nas buscas má direcionadas (ou mesmo aleatórias) por boas soluções.

3.2.2.1 *Classes de escalonamento*

Basicamente, três classes (ou tipos) de escalonamento têm sido discutidos na literatura quando se tratando do JSP [Fre82, HC90, Bak74]:

1. Escalonamentos semi-ativos: garantem que cada tarefa é iniciada tão logo ela possa, desde que obedeça às restrições tecnológicas e à seqüência de escalonamento. O número de escalonamentos semi-ativos embora finito, pode ser muito grande. O número exato geralmente é difícil de ser determinado. Para o problema clássico onde os *jobs* têm exatamente uma tarefa em cada máquina, cada uma delas tem que processar n tarefas. O número de possíveis seqüências é, portanto, $n!$ para cada máquina. Se a seqüência dentro de cada máquina fosse inteiramante independente, existiriam $(n!)^m$ escalonamentos semi-ativos [Bak74,SK79]. Contudo, as restrições de precedência e os roteamentos das máquinas para cada *job* usualmente tornam algumas destas combinações inválidas.
2. Escalonamentos ativos: a seqüência de processamento é tal que nenhuma tarefa pode ser iniciada mais cedo sem atrasar outra tarefa ou sem violar restrições tecnológicas.
3. Escalonamentos sem atraso: nenhuma máquina é mantida ociosa dado que ela poderia começar a processar alguma operação, isto é, máquinas devem processar tarefas uma vez que estas estejam disponíveis.

Observe que escalonamentos ativos constituem um conjunto dominante enquanto escalonamentos sem atraso formam um conjunto não dominante de soluções. Em outras palavras, o escalonamento ótimo para qualquer medida regular de desempenho (o que inclui, obviamente, o *makespan*) será um membro do conjunto de escalonamentos ativos [HC90].

Desde que o conjunto de escalonamentos ativos garante a ocorrência do valor ótimo para o problema, gerar todas as soluções ativas e selecionar a melhor leva a uma solução ótima (ótimo global). A existência freqüente de muito menos escalonamentos ativos que semi-ativos viabiliza o uso deste processo de enumeração completa no caso de problemas pequenos.

Os diagramas de Venn da figura 3.1 ilustram as possíveis relações entre os tipos de escalonamento, onde o asterisco indica a ocorrência de uma escalonamento ótimo. Observe que os conjuntos de escalonamentos semi-ativos e ativos são dominantes. Entretanto, escalonamentos ativos constituem o menor conjunto dominante para o JSP [Bak74].

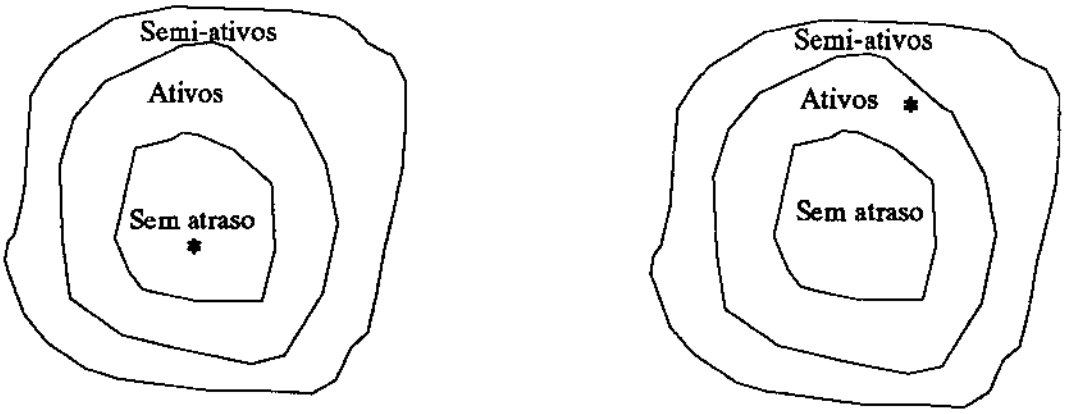


Figura 3.1 Diagrama de Venn para escalonamentos semi-ativos, ativos e sem atraso (* = ótimo).

A figura 3.2 ilustra soluções geradas, respectivamente, por algoritmos para escalonamentos sem atraso e escalonamentos ativos, tendo como base uma instância de um JSP com 2 *jobs*, 2 máquinas e 5 tarefas, onde o *job* 1 é constituído de três tarefas e o *job* 2 de duas tarefas.

O escalonamento (A) é ativo e sem atraso e possui *makespan* igual a 9 unidades de tempo. Note que nenhuma máquina ficou ociosa quando havia trabalho disponível para ser processado. O escalonamento (B) é ativo (mas não sem atraso) e seu *makespan* é igual a 8 (observe que a máquina 2 poderia ter trabalhado no *job* 2, mas permaneceu ociosa por uma unidade de tempo). Este é um problema onde a solução (B) é ótima embora seu escalonamento não seja sem atraso.

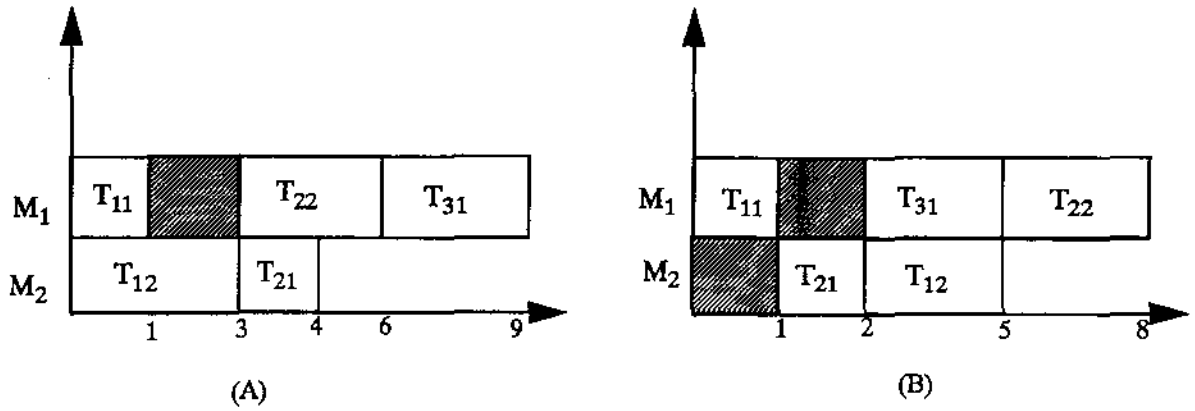


Figura 3.2 Gráficos de Gantt para ilustrar escalonamentos sem atraso e ativo.

Duas heurísticas clássicas (a primeira para geração de escalonamento ativo e a segunda para produção de escalonamento sem atraso) desenvolvidas por Giffler e Thompson (1960) [Fre82,Nas93] são descritas na próxima seção.

Heurísticas para escalonamento ativo e sem atraso

Na descriçao dos algoritmos para construção de escalonamentos sem atraso e ativos, usadas a seguinte notação e terminologia:

- Uma tarefa é dita escalonável se todas as tarefas que devem precedê-la dentro do seu *job* já tenham sido escalonadas;
- Dado que existem nm tarefas, o algoritmo iterage por nm estágios;
- No estágio t temos:

  - P_t: escalonamento das t-1 tarefas;

  - S_t: conjunto de tarefas escalonáveis no estágio t, isto é, todas as tarefas que devem preceder aquelas em S_t estão em P_t;

  - σ_{ij}: instante mais cedo que a tarefa T_{ij} em S_t poderia ser iniciada;

  - ϕ_{ij}: instante mais cedo que T_{ij} em S_t poderia ser finalizada, isto é, $\phi_{ij} = \sigma_{ij} + p_{ij}$, onde p_{ij} = tempo de processamento de T_{ij}.

Em síntese, as heurísticas de Giffler e Thompson para escalonamentos sem atraso e ativos podem ser descritas com segue:

Algoritmo GT - Ativo

1. $t = 1, P_1 = \{\}, S_1 = \{$Todas as primeiras tarefas de cada *job*$\}$.
2. $\phi^* = Min_{T_{ij} \in S_t} \{\phi_{ij}\}$

M^*= máquina onde ϕ^* ocorre (quebre impasses arbitrariamente)

3. Escolha uma tarefa T_{ij} em S_t tal que

   (a) Ela requeira M^* e

   (b) $\sigma_{ij} < \phi^*$

4. $P_{t+1} = P_t + \{T_{ij}\}$

   Se $\exists T_{i+1,j}$, $S_{t+1} = S_t - \{T_{ij}\} + \{T_{i+1,j}\}$. Caso contrário, $S_{t+1} = S_t - \{T_{ij}\}$

   $t = t + 1$

5. Se $t \leq nm$, volte ao passo 2. Caso contrário, pare.

<u>Algoritmo GT - Sem Atraso</u>

1. $t = 1$, $P_1 = \{\}$, $S_1 = \{$Todas as primeiras tarefas de cada *job*$\}$.

2. $\sigma^* = Min_{T_{ij} \in S_t} \{\sigma_{ij}\}$

   M^*= máquina onde σ^* ocorre (quebre impasses arbitrariamente)

3. Escolha uma tarefa T_{ij} em S_t tal que

   (a) Ela requeira M^* e

   (b) $\sigma_{ij} = \sigma^*$

4. $P_{t+1} = P_t + \{T_{ij}\}$

   Se $\exists T_{i+1,j}$, $S_{t+1} = S_t - \{T_{ij}\} + \{T_{i+1,j}\}$. Caso contrário, $S_{t+1} = S_t - \{T_{ij}\}$

   $t = t + 1$

5. Se $t \leq nm$, volte ao passo 2. Caso contrário, pare.

3.2.3 Heurísticas baseadas em relaxações

Dentre todas as heurísticas desenvolvidas para o JSP, o *shifting bottleneck* [ABZ88] é a que desfruta de maior credibilidade, devido aos resultados experimentais obtidos. Dado este fato, dedicamos a presente seção à apresentação da abordagem empregada pela heurística na solução do JSP (abordagem esta que tem como base uma relaxação do problema original).

3.2.3.1 O Shifting Bottleneck

<u>*Modelagem*</u>

O *shifting bottleneck* faz uso de uma modelagem baseada em grafos disjuntivos [ABZ88,Bal69,DL93,WR90] como ferramenta principal para representação e resolução do JSP com mínimo *makespan*.

No contexto de problemas de escalonamento os nós de um grafo disjuntivo correspondem a tarefas e os arcos dirigidos a relações de precedência. Podemos definir um grafo disjuntivo como $G = (N, A, E)$, onde:

- $N = T \cup \{s,t\}$, onde s e t são dois nodos adicionais (respectivamente fonte e sorvedouro) e N representa as tarefas do problema;
- A constitui o conjunto de pares de arcos conjuntivos (arcos dirigidos que representam relações de precedência entre duas tarefas);
- E representa os arcos disjuntivos do grafo, ou seja, o conjunto de tarefas a serem executadas na mesma máquina.

Dada esta representação algumas definições adicionais são construídas em cima do modelo:

(I) A cada arco é associado um valor relativo ao tempo de processamento de uma tarefa.

(II) O conjunto de arcos disjuntivos E decompõe-se em cliques E_k, $E = \bigcup (E_k, k \in M)$, uma para cada máquina de M (conjunto de máquinas).

(III) $D = (N, A)$ é o grafo dirigido obtido a partir de G com a remoção de todos os arcos disjuntivos.

(IV) Uma seleção S_k em E_k contém exatamente um membro de cada par de arcos disjuntivos de E_k. Uma seleção acíclica é uma seleção que não possui ciclos dirigidos. Uma seleção completa S é a união das seleções S_k, uma em cada E_k, $k \in M$. Uma seleção parcial é definida similarmente à seleção completa, excetuando-se que esta é feita sobre um subconjunto $M_0 \subset M$.

(V) $D_s = (N, A \cup S)$ é o grafo formado a partir de D, adicionando-se uma seleção completa a este.

O *makespan* de um escalonamento que é ótimo para S é igual ao tamanho de um caminho mais longo em D_s. Em linguagem de grafos disjuntivos, queremos encontrar uma seleção completa acíclica $S \subset E$ que minimize o tamanho de um caminho mais longo do grafo dirigido D_s.

A figura 3.3 ilustra um grafo disjuntivo, onde: (1) as tarefas são rotuladas de 0 a 14, sendo a tarefa 0 o nó fonte a a tarefa 14 o sorvedouro; (2) os arcos com setas preenchidas representam arcos conjuntivos e (3) os demais arcos representam os pares de arcos disjuntivos. Observe que as tarefas que devem ser executadas na mesma máquina (por exemplo, tarefas 1, 3, 6 e 9) constituem nodos de subgrafos que são cliques e seus arcos são disjuntivos.

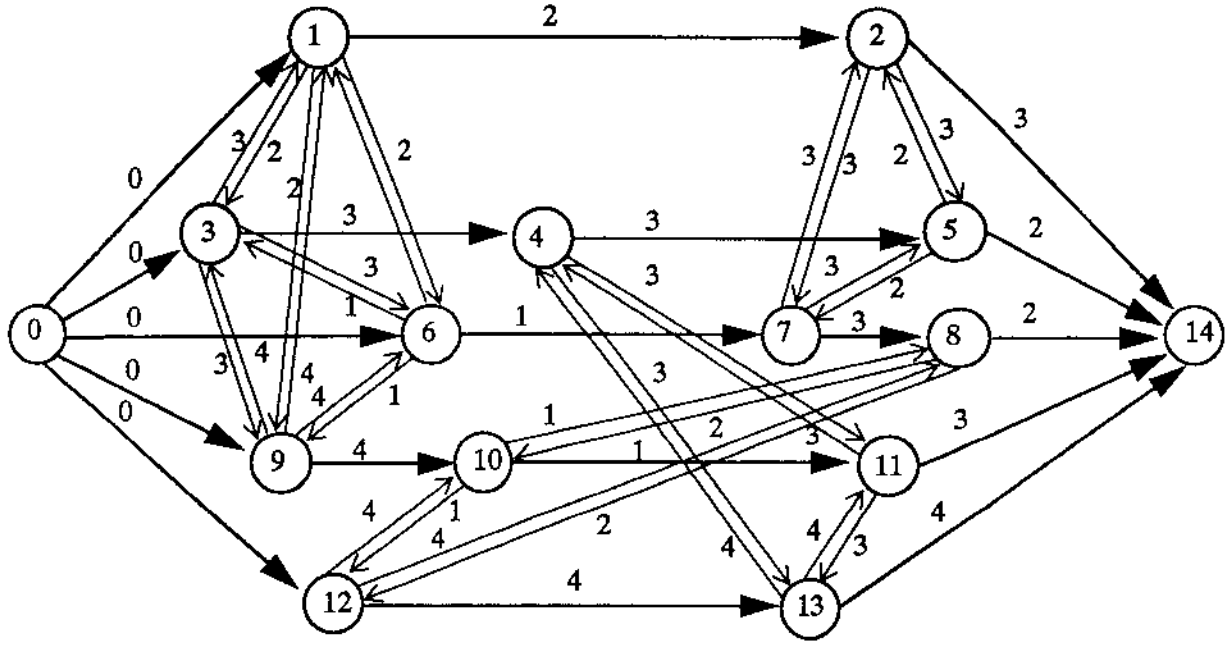


Figura 3.3 Grafo disjuntivo com quinze tarefas (em cinco *jobs*) e quatro máquinas.

O Algoritmo

A filosofia do ***shifting bottleneck*** está baseada em relaxar o problema original e resolver o problema de forma ótima para uma única máquina. É criada uma classificação (*ranking*) entre as máquinas e a cada passo é seqüenciada a máquina com classificação mais alta. Sempre que uma máquina é seqüenciada, é reotimizada a ordem de cada máquina previamente seqüenciada que seja suscetível a melhoria, resolvendo-se novamente o problema para uma máquina.

A principal contribuição desta abordagem é a maneira empregada para decidir a ordem em que as máquinas devem ser seqüenciadas. Esta decisão está centrada na idéia básica de priorizar máquinas-gargalo.

A forma utilizada para verificar que uma máquina k é um gargalo baseia-se no critério de "criticalidade". Uma máquina é dita crítica com respeito a $S = \bigcup (S_k, k \in M_0)$ e D_s, se S_k tem algum arco em um caminho mais longo em D_s.

Todavia, surge aqui outra questão: como quantificar este valor crítico? Para priorizar as máquinas necessita-se de um conceito que expresse a qualidade do gargalo como um problema de grau e não apenas de decisão (sim/não). Para tal, foi usada como medida de qualidade do gargalo da máquina k o valor da solução ótima para um certo problema de escalonamento de uma máquina, a máquina k. Mais precisamente: seja

$M_0 \subset M$ o conjunto de máquinas já seqüenciadas e $k \in M - M_0$, e seja $((P(k, M_0))$ o problema obtido a partir do problema original, tal que:

1. cada conjunto de arcos disjuntivos E_p, $p \in M_0$, é trocado pela seleção S_p;
2. cada conjunto de arcos disjuntivos E_p, $p \in M - M_0$, $p \neq k$ é removido.

A máquina m é dita o gargalo entre as máquinas indexadas por $M - M_0$ se $v(M, M_0) = Max\{v((k, M_0), k \in M - M_0)\}$, onde $v(k, M_0)$ é o valor da solução ótima para $(P(k,M_0))$. Logo, para cada máquina $k \in M - M_0$, é calculado o valor ótimo para o problema de escalonamento da máquina k em D_s, a fim de encontrarmos $m | v(m, M_0) = Max(v(k, M_0))$.

Resumidamente o *shifting bottleneck* pode ser descrito da seguinte forma:

Shifting Bottleneck

Seja M_0 o conjunto de máquinas já seqüenciadas (no início $M_0 = \varnothing$).

Passo1: Identifique uma máquina gargalo m entre as máquinas $k \in M - M_0$ e a seqüencie de forma ótima. Faça $M \leftarrow M_0 \cup \{m\}$.

Passo2: Reotimize a seqüência de cada máquina crítica $k \in M_0$, enquanto as demais seqüências são mantidas fixas. Isto é, faça $M_0 = M_0 - \{k\}$ e resolva $(P(k,M_0))$. Se $M_0 = M$, pare. Senão, volte ao Passo1.

Este algoritmo foi desenvolvido originalmente por Adams *et al.* em duas versões: a primeira (descrita neste capítulo) é denominada pelos autores como versão simplificada; a segunda, mais exaustiva e complexa, utiliza-se de uma enumeração seletiva para resolver o problema.

Observe que mesmo em sua versão simplificada, o *shifting bottleneck* necessita resolver um outro problema NP-Completo (o seqüenciamento ótimo de uma máquina) como passo intermediário para resolver o JSP. Para tal, seus autores utilizaram um algoritmo tipo *branch and bound* desenvolvido por Carlier [Car82]. Somente este fato é suficiente para revelar o grau de dificuldade atrelado à implementação deste algoritmo, mesmo em sua versão mais simples.

3.3 *Sumário*

Este capítulo apresentou métodos aproximados para solução de problemas combinatórios e, mais especificamente, para problemas de escalonamento de tarefas. Foram descritas algumas técnicas aproximadas gerais (meta-heurísticas) para solucionamento de problemas combinatórios e apresentadas algumas heurísticas

clássicas para o problema de escalonamento de tarefas conhecido por *Job Shop Scheduling Problem*.

Foi apresentada uma classificação geral dos tipos (classes) de escalonamentos para o JSP, bem como uma visão geral das diversas abordagens aproximadas empregadas na solução do problema. Os algoritmos de Giffler e Thompson para escalonamentos ativo e sem atraso e o algoritmo *shifting bottleneck* (proposto por Adams *et al.*[ABZ88]) foram descritos.

Capítulo 4

Times Assíncronos e o Job Shop Scheduling Problem

Devido à necessidade de agilização do processo produtivo, causada principalmente pelo surgimento de um mercado crescentemente competitivo, têm-se buscado técnicas novas e eficientes para solução dos mais diversos problemas presentes no cotidiano de atividades produtivas.

Muito esforço tem sido dispensado no desenvolvimento de novos métodos de solução de problemas de otimização. Destacadamente, problemas de escalonamento de produção, tal qual o JSP, têm despertado grande interesse, sobretudo nos meios industrial e acadêmico. Tanto métodos exatos, quanto aproximados, vêm sendo objetos de pesquisa na procura por formas convenientes à solução de problemas combinatórios (particularmente o *Job Shop Scheduling*). De fato, diversos são os trabalhos publicados que propõem novos algoritmos para o JSP, sendo boa parte deles pertencentes à classe de heurísticas de construção.

Times Assíncronos (ou *A-Teams*, do inglês *Asynchronous Teams*) é um método de resolução de problemas que faz uso simultâneo de diversos algoritmos. Estes continuamente trocam dados entre si, através de memórias compartilhadas, podendo gerar soluções que não seriam capazes de fazê-lo quando executados isoladamente. Este método foi apresentado por Souza [Sou93], tendo sido empregado com sucesso na solução do problema do Caixeiro Viajante [NW88].

Do cenário acima descrito, vislumbra-se a seguinte situação: de um lado, um problema combinatório, reconhecidamente de difícil solução, para o qual é desconhecido um algoritmo rápido que sempre gere soluções garantidamente boas; em contrapartida, dispõe-se de uma nova técnica para solução de problemas aplicada com sucesso ao, também difícil, problema do Caixeiro Viajante. Surge, deste quadro, a motivação

mestra para a elaboração da presente investigação, onde é averiguada a aplicabilidade de *A-Teams* ao JSP.

Neste capítulo são apresentados: (a) uma conceituação teórica básica relativa a *A-Teams* (características, estruturas, representação, etc); (b) uma descrição detalhada da estrutura dos Times Assíncronos baseados em heurísticas de construção para o JSP desenvolvidos no transcorrer deste trabalho e (c) os resultados experimentalmente obtidos a partir da implementação realizada.

4.1 *Times Assíncronos*

O conceito de *A-Teams* é um passo inovador no domínio da Teoria de Organização [Sou93], principalmente quanto à descentralização do controle dentro da estrutura organizacional onde os agentes interagem. Segundo observado em Souza [Sou93], percebe-se que, apesar de algum esforço, tem-se alcançado pouco sucesso com organizações de software que não seguem o modelo organizacional tradicional e hierárquico.

Um Time Assíncrono é uma técnica que utiliza uma organização de agentes que se comunicam através de memórias compartilhadas. Tais agentes são algoritmos associados a protocolos específicos para manipulação de informações dentro do *A-Team*.

4.1.1 *Uma descrição gráfica*

Graficamente, pode-se representar agentes como arcos e memórias compartilhadas como retângulos. Agentes podem ler ou escrever em memórias compartilhadas. Uma memória pode ser composta de várias outras. Na figura 4.1 é apresentado um exemplo de estrutura de um *A-Team* que pode ser assim descrito: agentes A e B lêem da memória 1 e escrevem, respectivamente, nas memórias 1 e 3. Agente C lê da memória 2 e escreve na memória 1. Agente D lê da memória 3 e pode escrever tanto na memória 1 quanto na memória 2 (note o retângulo externo a estas memórias). Agente E lê da memória 2 e escreve na memória 3. Agente F inicia as memórias 1 e 2, enquanto o agente G é responsável por remover soluções destas memórias

4.1.2 *Propriedades básicas*

As propriedades que caracterizam um *A-Team* são [Sou93,TS90,TS92]:

- agentes autônomos: agentes que efetuam suas próprias escolhas relativas aos dados de entrada, à estratégia de escalonamento e à política de alocação de recursos disponíveis;
- comunicações assíncronas: agentes podem ler e escrever informações em memórias compartilhadas sem qualquer sincronização entre eles;

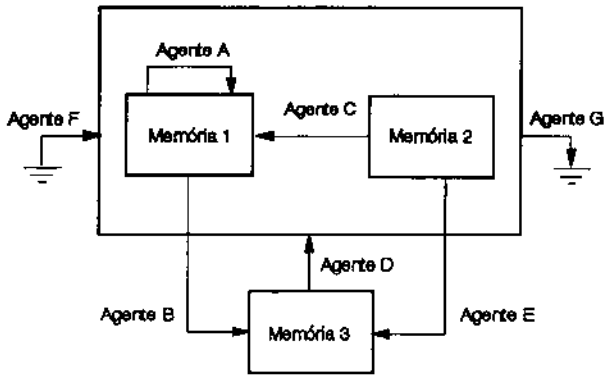


Figura 4.1 Exemplo de um *A-Team*.

- fluxo cíclico de dados: agentes recuperam, modificam e armazenam informações nas memórias compartilhadas. O fluxo cíclico de dados propicia um fluxo contínuo de modificações executadas pelos agentes, permitindo interação e refinamento de dados entre eles.

Observe que a autonomia dos agentes em conjunto com comunicações assíncronas permite o trabalho paralelo dos agentes. Além disso, devido apresentarem uma estrutura não hierárquica, *A-Teams* não precisam de processos gerenciadores para controlar a cooperação entre os algoritmos; portanto, agentes podem ser facilmente incluídos ou excluídos de um *A-Team* a qualquer momento, garantindo a robustez da organização.

4.1.3 Sinergia e Eficiência em Escala

Além das propriedades básicas que caracterizam a estrutura de um *A-Team*, faz-se necessário ressaltar dois aspectos importantes comumente revelados com a dinâmica da organização: sinergia e eficiência em escala.

Quando a efetividade de uma cooperação é tal que, em algum sentido, "o todo é maior que a soma das partes", diz-se que esta cooperação é sinérgica [Tal93,TS90]. No contexto de *A-Team* observa-se que o uso de mais que um algoritmo na solução do problema, permite que a cooperação entre os algoritmos aumente as chances de obtenção de melhores soluções que aquelas geradas por cada algoritmo separadamente [Sou93].

Segundo Talukdar e Souza [ST91,TS92], uma organização é eficiente em escala se ela é aberta (cresce facilmente) e seus membros, antigos e novos, cooperam efetivamente. Para melhor compreensão deste conceito, uma organização pode ser vista como uma estrutura composta de *slots* (aberturas) para agentes. Supõe-se que c é uma capacidade qualquer, ou seja, qualquer medida de interesse relativa aos resultados produzidos por uma coleção de agentes. Seja A um conjunto de agentes que compõem uma organização, então esta organização é dita aberta para A se tem (ou pode

automaticamente construir) um *slot* para cada agente em A. A organização é efetiva sobre A e c se faz com que os membros de A cooperem entre si, melhorando os valores de c. Ou ainda, se existe pelo menos uma ordem de acréscimo de membros de A na organização que produz melhoria monotônica em c. Portanto, a organização é dita eficiente em escala sobre A e c se é aberta para A e efetiva sobre A e c [Sou93].

4.1.4 *Potencialidade ao paralelismo*

Observe que *A-Teams* constituem um sistema completamente descentralizado e, portanto, altamente adequado ao processamento distribuído numa rede de computadores. Dois fatos principais atestam esta adequação: (a) a freqüente granularidade grossa dos seus algoritmos quando resolvendo problemas complexos e de grande porte (o tempo de comunicação entre os algoritmos é praticamente desprezível em relação ao tempo total de processamento) e (b) a independência existente entre os algoritmos, permitindo execução simultânea em diversos processadores.

4.2 *A-Teams para o JSP*

O fato do JSP ser considerado um Problema Multi-Algoritmo[1] (PMA) [ST93,Sou93] reforça a expectativa de que *A-Teams* se constituam uma ferramenta bastante adequada para solucionar o *Job Shop Problem*.

Particularmente, o presente trabalho concentra-se na elaboração e experimentação de *A-Teams* baseados em algoritmos de construção para o JSP. A seguir são descritos alguns algoritmos desenvolvidos e utilizados como componentes (agentes) dos *A-Teams* implementados.

4.2.1 *Fluxo de dados*

Um dos parâmetros de projeto quando se deseja construir um *A-Team* é o seu fluxo de dados. Este define a estrutura funcional de um *A-Team*, explicitando os tipos de acesso (leitura/escrita) dos agentes às memórias compartilhadas.

Dado que uma das características de *A-Teams* é o fluxo cíclico de dados, algoritmos são especialmente construídos objetivando promover esta ciclagem dentro da organização, propiciando a interação entre os membros da população de soluções disponíveis nas memórias.

A figura 4.2 apresenta, de forma simplificada, um exemplo típico do processo de ciclagem em um *A-Team* centrado em construção. Basicamente o ciclo é dividido em

1. Um problema *P* é dito PMA se existem algoritmos A_j, $j>1$, que provenham soluções para *P*.

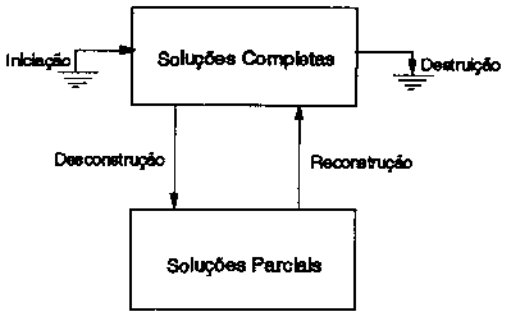


Figura 4.2 Exemplo típico de um ciclo em um *A-Team* centrado em agentes de construção.

duas fases: (1) fase de desconstrução, onde é produzida uma solução parcial (incompleta) para o problema, sendo esta armazenada na memória de soluções parciais e, (2) fase de reconstrução, onde a partir de uma solução parcial é produzida uma solução completa para o problema.

Nas próximas seções serão detalhados os algoritmos específicos para *A-Teams* desenvolvidos e discutidos no transcorrer deste trabalho.

4.2.2 *Algoritmos baseados em consenso*

Dado que *A-Teams* usam uma população de soluções distintas para resolver o problema que está sendo atacado, é possível tirar vantagem das diferenças e semelhanças entre estas soluções, a fim de aumentar a eficiência do processo de busca por soluções melhores que as já disponíveis.

Algoritmos de consenso são aqueles que usam duas ou mais soluções para gerar uma outra, dispondo, para tal, de informações das soluções de entrada [Sou93]. Souza [Sou93] identificou pelo menos duas classes de algoritmos de consenso: consenso baseado em intersecção[1] e consenso baseado em união[2]. Além destas, foi identificada uma terceira classe de algoritmos de consenso, a qual recebeu a denominação de consenso baseado em diferença.

Nesta seção são apresentados os algoritmos de consenso baseados em intersecção, união e diferença desenvolvidos para o JSP.

4.2.2.1 *Consenso baseado em intersecção*

Denominou-se consenso baseado em intersecção aquele que extrai o que é comum entre várias soluções, gerando uma solução parcial (ou incompleta) para o problema

1. Chamados de *common-agreement algorithms* em [Sou93].
2. Chamados de *suggestion-based algorithms* em [Sou93].

[CS94,CS95,Pei95,Sou93]. Esta solução parcial é reconstruída através da introdução das demais partes do problema. A idéia por trás deste tipo de algoritmo é que intersecções entre soluções constituem boas sementes na obtenção de outras soluções para o problema.

Em termos de JSP, podem ser imaginados diversos mecanismos heurísticos (algoritmos) para obtenção de consenso baseados em intersecção entre soluções. Três destes algoritmos desenvolvidos e avaliados no decorrer deste trabalho são descritos a seguir.

Intersecção baseada em ordem de escalonamento (IOE)

Em geral, a representação padrão de uma solução para o JSP consiste em, para cada máquina m, ter-se uma seqüência de tarefas e os respectivos tempos de escalonamento de cada tarefa em m [Bak74]. Isto é, tem-se atrelados a cada solução uma ordem de escalonamento de tarefas e os momentos em que as tarefas foram escalonadas nas máquinas. Considerando a ordem de escalonamento das tarefas um elemento importante no JSP, elaborou-se um algoritmo de consenso centrado nesta ordem.

Para facilitar o entendimento do algoritmo desenvolvido, considerou-se como estrutura representativa de uma solução para o JSP a mesma adotada por Giffler e Thompson (segundo French [Fre82]) quando da especificação de seus algoritmos. Basicamente, uma solução para o problema consiste em uma lista L ($L = \langle T_1, T_2, ..., T_l \rangle$, com $l \leq nm$) das tarefas escalonadas. Para cada par (T_{t-1}, T_t) de tarefas adjacentes na lista ($t = 2, ..., l$), T_t é escalonada em seguida a T_{t-1}, logo que possível, em sua respectiva máquina, respeitando as restrições do problema.

A figura 4.3 ilustra o funcionamento deste algoritmo de intersecção, onde as partes escuras nas duas primeiras listas indicam as coincidências na ordem de escalonamento das tarefas em cada solução. Na terceira lista é representada a solução parcial obtida com o consenso das duas primeiras soluções. Na última lista está representada uma possível solução, construída a partir da solução parcial. No processo de reconstrução, as posições relativas entre as tarefas que fazem parte da intersecção são conservadas. Assim, se na solução parcial uma tarefa T_a precede uma tarefa T_b, esta relação de precedência é mantida na fase de reconstrução de uma solução completa.

De forma sucinta, o algoritmo pode ser assim descrito:

1. Escolha aleatoriamente duas soluções distintas L_a e L_b dentre as disponíveis na população de soluções completas.
2. Selecione as tarefas comuns a L_a e L_b que ocupam as mesmas posições em cada uma das listas, gerando a solução parcial L_p. Ou seja, para todo $L_a[k] = L_b[k]$, $k = 1, 2, ..., l$, insira $L_a[k]$ (ou $L_b[k]$) em $L_p[k]$.

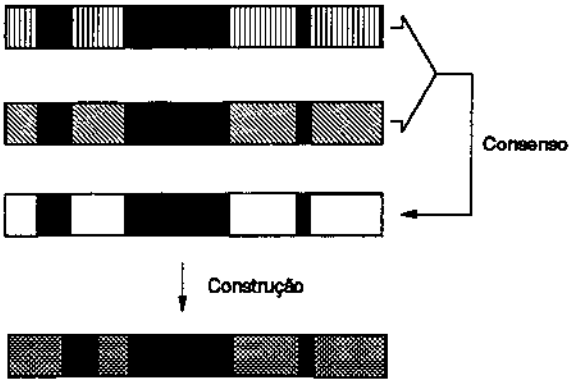


Figura 4.3 Intersecção baseada na ordem de escalonamento das tarefas.

Intersecção baseada em tempos de escalonamento (ITE)

Este tipo de consenso procura explorar um outro atributo importante para construção de uma solução, que não a ordem em que as tarefas foram escalonadas. Aqui são levados em consideração os momentos (instantes de tempo) em que as tarefas foram escalonadas dentro das máquinas.

Intersecções baseadas em tempos de escalonamento traduzem de forma mais fiel as coincidências entre soluções do JSP, que intersecções simplesmente centradas na ordem de escalonamento.

Coincidência na posição de escalonamento de uma mesma tarefa de acordo com duas soluções distintas representadas por listas, não implica, obrigatoriamente, que ambas as soluções concordam quanto ao início do processamento da tarefa. Note que o elemento que determina perfeitamente a igualdade (ou diferença) entre soluções de um JSP é o momento de escalonamento de suas tarefas dentro das máquinas.

Desta forma, não é difícil observar que, no caso de problemas de escalonamento como o JSP (problema com múltiplas máquinas), a concepção de consensos parametrizados segundo tempos de escalonamento das tarefas parece mais adequada que o uso da ordem em que as tarefas foram escalonadas como parâmetro de comparação de soluções. Provavelmente, consensos baseados em ordem de escalonamentos são mais adequados a problemas de seqüenciamento de tarefa, onde apenas uma única máquina é considerada.

A figura 4.4 apresenta, com auxílio de gráficos de Gantt, um exemplo de intersecção baseada em tempos de escalonamento. Na fase de reconstrução as posições relativas dentro das máquinas são mantidas.

O algoritmo de intersecção baseada em tempos de escalonamento pode ser assim descrito:

1. Escolha aleatoriamente duas soluções distintas S_a e S_b dentre as disponíveis na população de soluções completas.
2. Para cada $T_{ij} \in S_a, S_b$, se o instante de escalonamento (início) de T_{ij} em S_a coincidir com seu início em S_b, então insira T_{ij} na solução parcial S_p.

Note que no passo 2 do algoritmo acima (fase de desconstrução), os tempos de escalonamento das tarefas de S_a e S_b são utilizados como instrumentos de comparação entre as soluções e, por conseguinte, como elementos base na obtenção da intersecção entre as soluções. Na fase de reconstrução, em contrapartida, o importante é a manutenção da ordem relativa entre as tarefas de cada máquinas, segundo S_p.

Assim, a diferença básica entre os algoritmos baseados em ordem de escalonamento e aqueles centrados nos tempos de escalonamento das tarefas consiste no elemento base (instantes de tempo ou posição na solução) para obtenção do consenso (fase de desconstrução). Na fase de reconstrução a preocupação é comum: preservar ordens relativas entre as tarefas dentro da estrutura que representa uma solução.

Intersecção baseada em máquinas-gargalo (IMG)

Uma forma interessante de se atacar problemas de escalonamento de tarefas consiste na identificação de elementos que sejam gargalos na solução do problema [ABZ88,CP89,DL93], isto é, elementos que dificultam (ou mesmo impedem) a obtenção de boas soluções. O presente algoritmo de intersecção faz uso dessa estratégia na busca de soluções satisfatórias.

Dado que o JSP com mínimo *makespan* é um problema de escalonamento de tarefas em máquinas, onde a existência de máquinas tardiamente utilizadas pode comprometer a qualidade das soluções, toma-se máquinas assim utilizadas como elementos-gargalo a serem evitados (ou modificados). Mais precisamente, são adotadas como máquinas-gargalo de uma solução, aquelas que por último finalizaram seus processamentos.

A identificação de uma máquina como sendo gargalo implica a admissão da necessidade de um reescalonamento dentro da máquina. Em termos de *A-Teams*, esta identificação pode significar que o seqüenciamento dentro da máquina não é um bom ponto de partida para obtenção de uma nova solução e, conseqüentemente, a presença de tal seqüenciamento dentro de uma solução parcial pode não ser conveniente. Assim, partindo de um prisma de reconstrução de soluções, este algoritmo de intersecção procura evitar dentro das soluções parciais a presença de seqüenciamentos de máquinas consideradas gargalos.

A idéia básica deste consenso é possibilitar a geração de soluções completas melhores que as soluções utilizadas no consenso. Para tal, são utilizadas na elaboração do

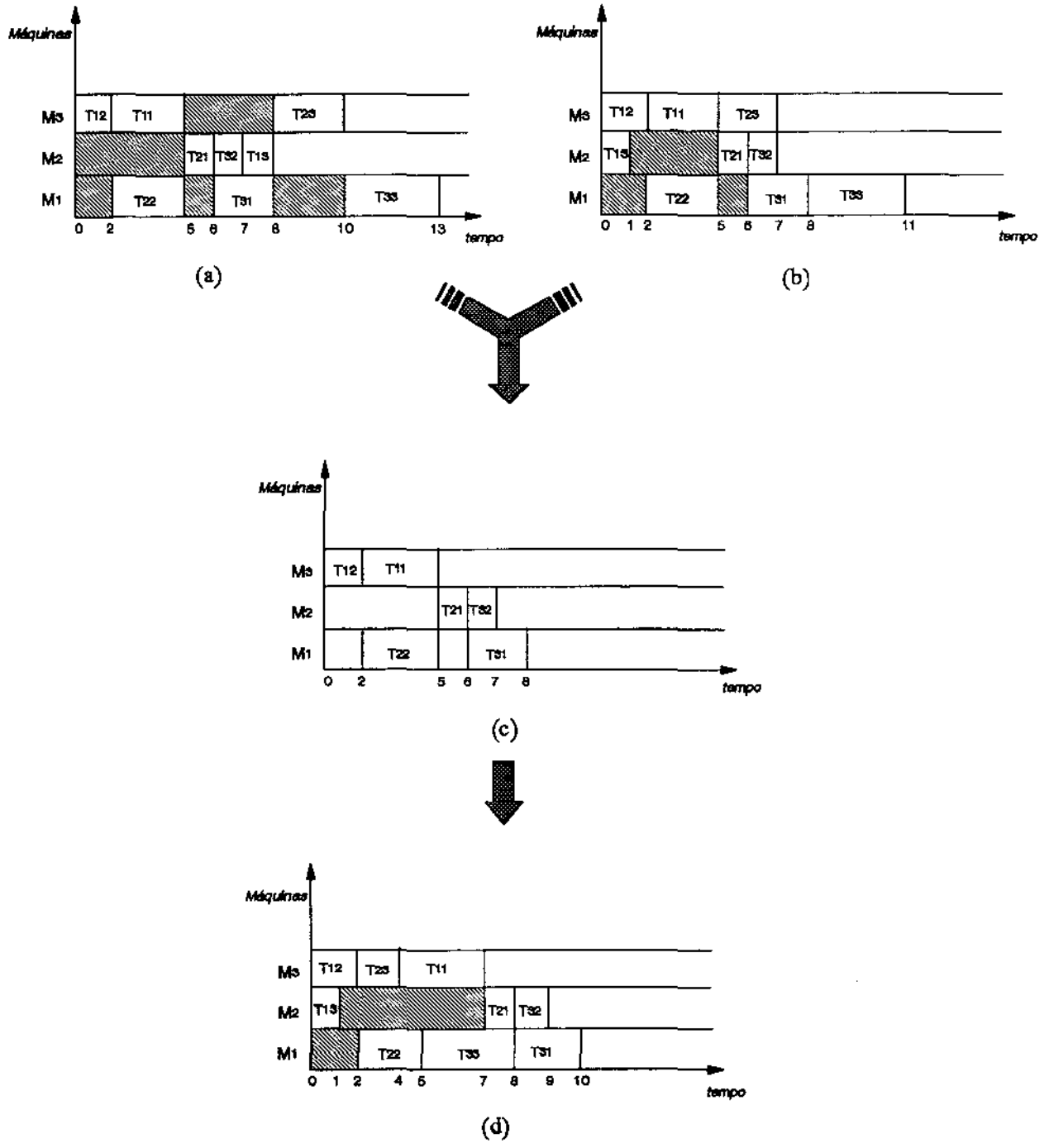


Figura 4.4 Intersecção baseada em tempos de escalonamento. Dadas duas soluções (a) e (b), uma solução parcial (c) é gerada com base na intersecção das duas primeiras. A partir de (c) é obtida uma nova solução completa (d) para o problema.

consenso as máquinas-gargalo comuns a duas soluções distintas quaisquer, e uma destas soluções é arbitrariamente escolhida como base para composição de uma solução parcial. A solução parcial gerada será constituída das tarefas da solução base, da forma que foram escalonadas, extraindo-se as tarefas que são executadas nas máquinas consideradas gargalos quando do consenso.

Note que um parâmetro necessário à construção desta intersecção é um número k (fixo ou variável) de máquinas-gargalo de cada solução. Nos experimentos realizados foram testadas várias configurações do algoritmo de intersecção com valores distintos de k.

As preocupações em encontrar valores convenientes do parâmetro k versaram basicamente sobre: (a) garantir que quando feito o consenso, os seqüenciamentos de pelo menos duas máquinas-gargalo não seriam incluídos na solução parcial, tornando possível, na fase de reconstrução, a obtenção de uma solução diferente das originais; (b) evitar que o consenso produzisse soluções parciais muito vazias (tipicamente com valores de k muito grandes), tornando muito cara a fase de composição de uma nova solução, e (c) possibilitar um nível razoável de diversidade dentro da memória de soluções parciais.

Como versão final, foram adotados valores aleatórios de k dependentes do tamanho da instância do problema sendo resolvido. Mais especificamente, adotou-se valores de k tais que $\left\lceil \frac{m}{2} \right\rceil + 1 \leq k < m$ (foram utilizados como testes problemas onde $m \geq 5$). Adicionalmente, como controle do nível de esvaziamento das soluções parciais, limitou-se o número de máquinas-gargalo consensuais entre as soluções em $\left\lceil \frac{m}{2} \right\rceil + 1$.

De forma mais clara, este algoritmo pode ser descrito como segue:

1. Escolha aleatoriamente duas soluções distintas S_a e S_b dentre as disponíveis na população de soluções completas.
2. Identifique as k máquinas-gargalo de S_a e S_b, onde k é um número aleatório e $k \in \left[\left\lceil \frac{m}{2} \right\rceil + 1, m - 1 \right]$.
3. Seja LMG_a e LMG_b, respectivamente, as listas de máquinas-gargalo das soluções S_a e S_b, gere a lista $LMG_{a \cap b}$ de até $\left\lceil \frac{m}{2} \right\rceil + 1$ máquinas-gargalo comuns a ambas as soluções.
4. Tome arbitrariamente umas das duas soluções sorteadas (S_a ou S_b), adotando-a como solução base.

5. Gere uma solução parcial a partir da solução base, tal que nenhuma tarefa que seja executada em alguma máquina de $LMG_{a \cap b}$ seja inserida como componente da solução parcial.

A figura 4.5 mostra um exemplo deste algoritmo para uma instância 3 *jobs* x 3 máquinas (com seis tarefas). Note que este algoritmo de consenso pode gerar uma solução melhor que as soluções utilizadas na intersecção.

4.2.2.2 *Consenso baseado em união*

O consenso baseado em união (ou algoritmo de sugestão, como também é referenciado ao longo do texto), ao contrário do que acontece com algoritmos baseados em intersecção, constrói uma nova solução unindo partes das soluções originais. Assim sendo, algoritmos desta natureza aceitam sugestões (não necessariamente iguais) de várias soluções na composição de uma nova solução (seja esta parcial ou completa).

Sugestão baseada em intercalação (SI)

Algoritmos para *A-Teams* são, preferencialmente, simples e de fácil codificação. O algoritmo de intercalação desenvolvido é um exemplo claro dessa simplicidade.

Este algoritmo está centrado na idéia de intercalação de duas soluções, segundo a ordem em que as tarefas foram escalonadas. Uma vez obtida a solução intercalada (a solução dupla gerada, já que possui redundância de cada tarefa), uma nova solução simples é gerada através da eliminação das redundâncias existentes na solução intercalada.

Aqui, novamente, considerou-se uma solução como sendo uma lista de tarefas escalonadas de acordo com a ordem que aparecem na lista. No processo de eliminação de redundâncias é realizada uma varredura na solução duplicada, sendo eliminadas desta as segundas ocorrências de cada tarefa da solução. Como resultado da varredura tem-se uma nova ordem de escalonamento, a qual é obrigatoriamente válida (dado que as soluções originais são válidas e que a intercalação e a eliminação não infringem restrições do problema) e determina uma nova solução para o problema.

Apesar de aparentemente ingênuo, este algoritmo se mostrou bastante eficaz, principalmente no processo de convergência dentro da população de soluções existentes nas memórias compartilhadas.

Simplificadamente o algoritmo é assim descrito:

1. Escolha aleatoriamente duas soluções distintas L_a e L_b dentre as disponíveis na população de soluções completas.
2. Gere um solução dupla DL_{ab} a partir da intercalação de L_a e L_b, na seqüência em que suas tarefas foram escalonadas. A intercalação é feita tarefa a tarefa conforme ilustrado na figura 4.6.

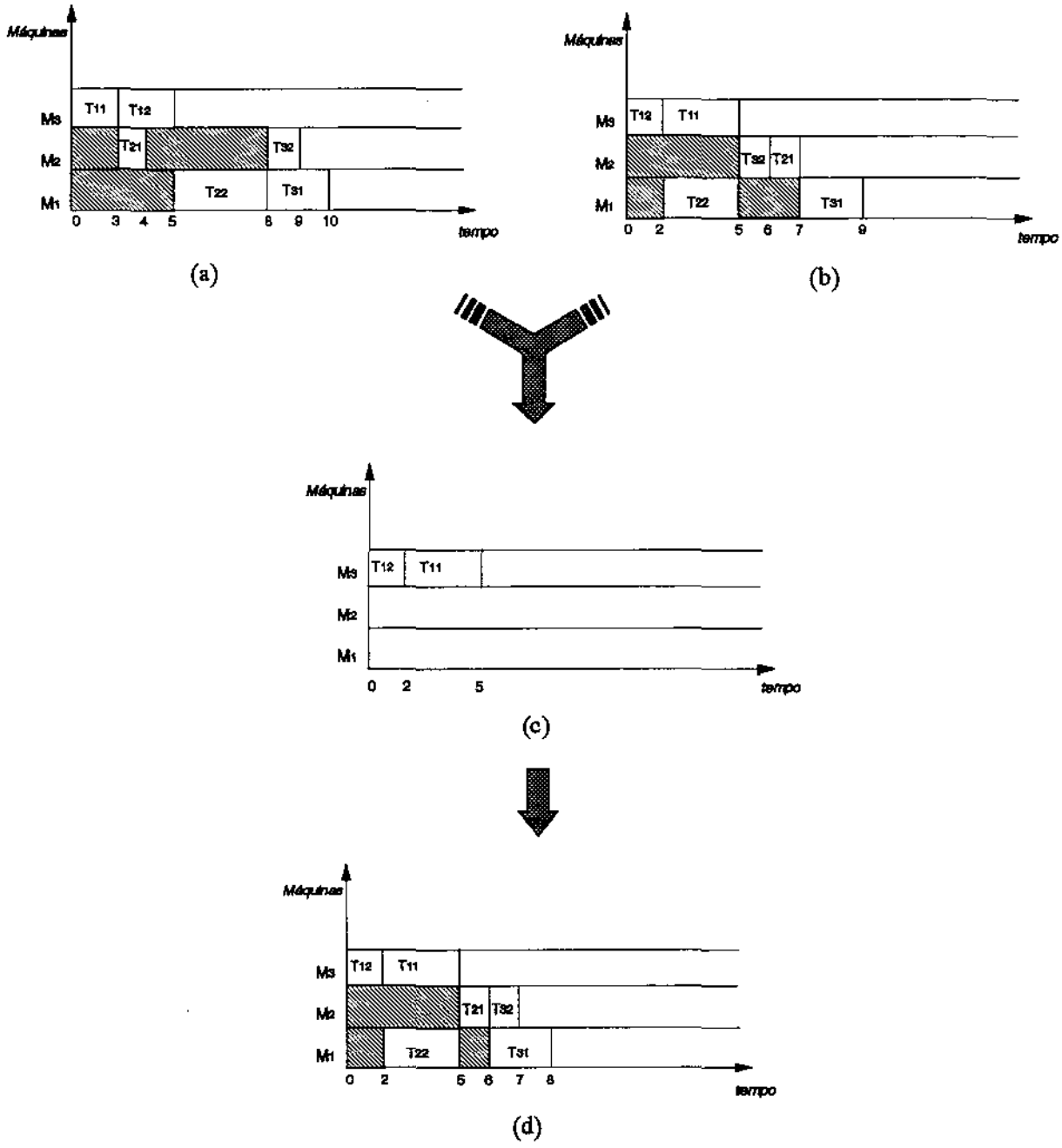


Figura 4.5 Intersecção baseada em máquinas-gargalo, onde k=2. É efetuada a intersecção entre as máquinas-gargalo das soluções representadas em (a) e (b). A solução parcial obtida (solução base = (b)) é representada em (c) e, uma possível nova solução é ilustrada em (d).

3. Elimine de DL_{ab} as redundâncias de tarefas, percorrendo-a, para tal, da esquerda para a direita (ou seja, varrendo DL_{ab} a partir da primeira tarefa escalonada).
4. Recalcule os tempos de escalonamento de cada tarefa remanescente em DL_{ab}, obtendo assim uma nova solução válida.

Este algoritmo é ilustrado na figura 4.6, onde duas soluções, compostas de cinco tarefas cada, são intercaladas gerando uma nova solução.

Sugestão baseada em máquinas-gargalo (SMG)

Semelhante ao algoritmo de intersecção baseado em máquinas-gargalo, o presente algoritmo utiliza-se do mesmo conceito de gargalo. Neste caso, entretanto, não são tomadas as máquinas-gargalo comuns, e sim a união destas máquinas. Apesar de uma máquina-gargalo para determinada solução não ser, necessariamente, gargalo para outra solução, o reescalonamento de máquinas não gargalo pode, devido às restrições de precedência do JSP, propiciar uma alteração nos tempos de escalonamento dentro das máquinas-gargalo comuns.

Note que o processo de desconstrução empregado por este algoritmo é muito mais acentuado que o efetuado pelo algoritmo de intersecção de máquinas-gargalo. Mesmo valores pequenos k (k = número de máquinas gargalo de cada solução) podem acarretar

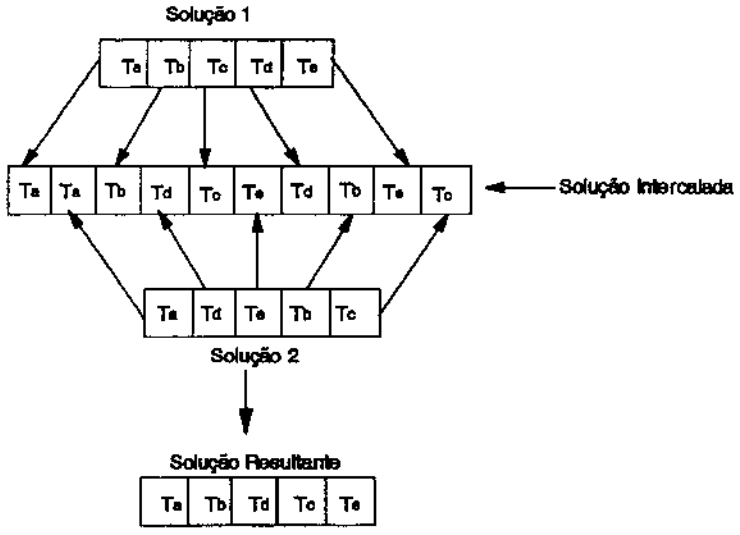


Figura 4.6 Algoritmo baseado na intercalação de duas soluções (aqui representadas por duas listas de tarefas, na ordem em que estas foram escalonadas).

um esvaziamento muito acentuado das soluções parciais geradas, transferindo para a fase de reconstrução muito da responsabilidade de produzir uma nova solução boa.

A fim de evitar tal esvaziamento, foi introduzida uma heurística complemetar de seleção das tarefas a serem excluídas das soluções parciais. Tarefas que são executadas em máquinas-gargalo não são descartadas, indiscriminadamente, quando da composição de soluções parciais. Empregou-se como heurística complementar, a eliminação apenas das menores tarefas (tarefas com menor tempo de processamento associado). Considerou-se que uma tarefa está entre as menores, quando seu tempo de processamento é menor que o tempo médio de processamento das tarefas do problema que está sendo resolvido.

A opção por serem desconsideradas apenas as tarefas menores (conservando tarefas relativamente grandes) teve como intuito possibilitar a reconstrução de novas soluções com valores de *makespan* razoavelmente previsíveis (não muito distantes dos *makespans* das soluções originais). A eliminação indiscriminada de tarefas de qualquer duração, por exemplo, poderia comprometer a qualidade da nova solução completa a ser obtida.

Quanto aos valores do parâmetro k, foram usados valores não menores que dois (o que garante pelo menos duas máquinas no conjunto união) e não maiores que $\lceil m/3 \rceil$ nos casos em que $k > 3$.

Resumidamente, segue o algoritmo:

1. Escolha aleatoriamente duas soluções distintas S_a e S_b dentre as disponíveis na população de soluções completas.
2. Identifique as k máquinas-gargalo de S_a e S_b, onde k é um número aleatório e $k \in [2, max(2, \lceil m/3 \rceil)]$.
3. Sejam LMG_a e LMG_b, respectivamente, as listas de máquinas-gargalo de S_a e S_b, gere a lista $LMG_{a \cup b} = (LMG_a \cup LMG_b)$.
4. Adote, arbitrariamente, uma das duas soluções sorteadas (S_a ou S_b) como solução base.
5. Gere uma solução parcial a partir da solução base, tal que uma tarefa que execute em alguma máquina de $LMG_{a \cup b}$ só seja inserida como componente da solução parcial, caso seu tempo de processamento seja superior ou igual ao tempo médio de processamento das tarefas do problema.

A figura 4.7 ilustra o comportamento deste algoritmo com o auxílio de uma instância composta de três *jobs* e três máquinas (totalizando seis tarefas). É apresentada também uma possível nova solução obtida com uma reconstrução efetuada com base na solução parcial gerada.

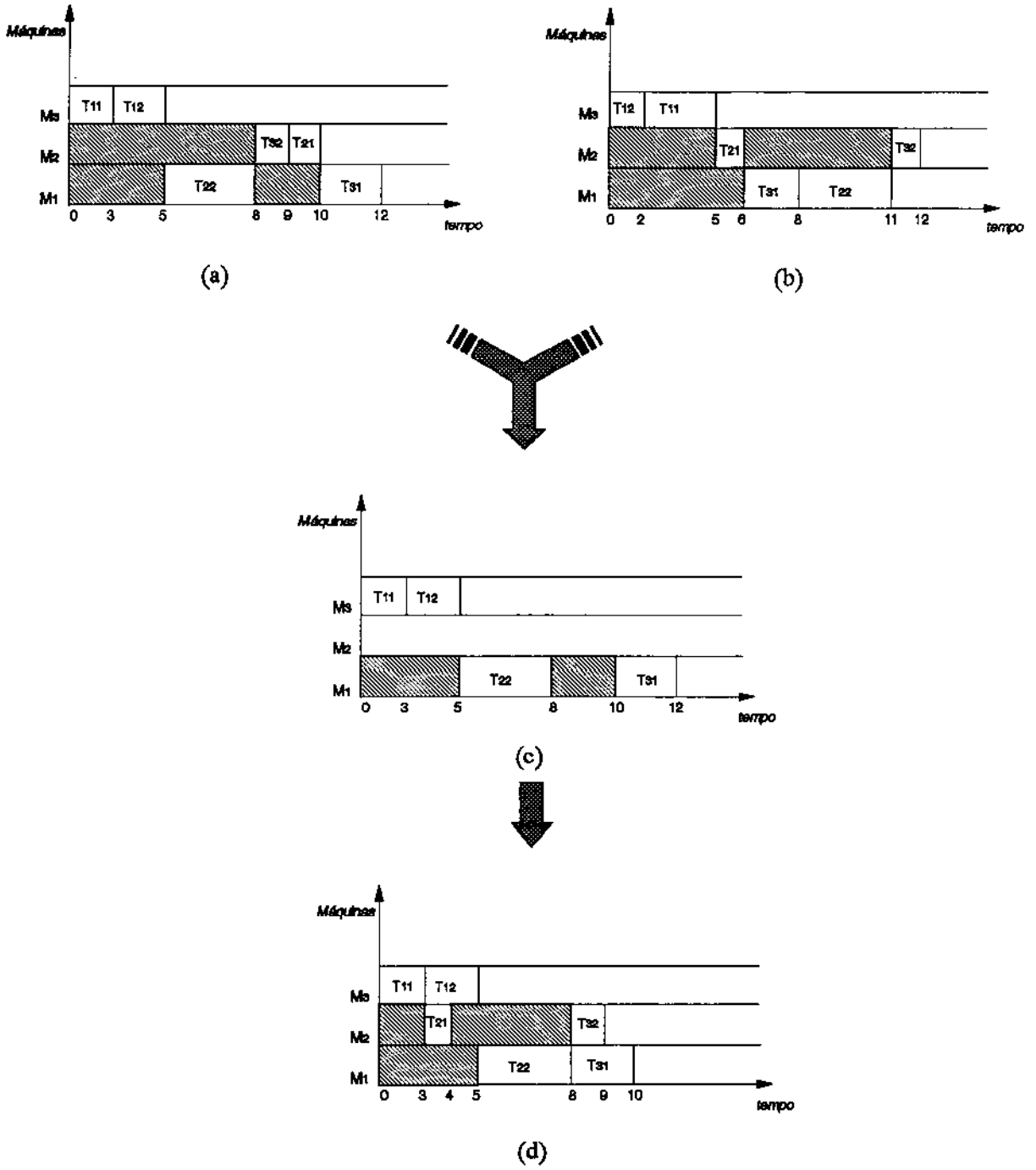


Figura 4.7 Sugestão baseada em máquinas-gargalo, onde o número de máquinas gargalo (k) é igual a três e as soluções utilizadas na sugestão estão representadas em (a) e (b). Observe que o processo de desconstrução afetou somente as tarefas da máquina dois (c). Isto se deu devido à média dos tempos de processamento das seis tarefas ser igual a 2, e, assim sendo, apenas as tarefas que são executadas na máquina dois não obedecem à restrição do passo 5 do algoritmo. O diagrama representado em (d) ilustra uma possível reconstrução a partir da solução parcial (c).

4.2.2.3 *Consensos baseados em diferenças (DOE e DTE)*

Os algoritmos centrados em diferença desenvolvidos estão centrados no seguinte princípio: dadas duas soluções A e B, distintas, se, por exemplo, A é de alguma forma melhor que B, então é porque há algo de bom em A que inexiste em B. Partindo-se desta premissa, foram desenvolvidos dois algoritmos baseados em diferenças entre duas soluções de um JSP. Apesar destes algoritmos usarem a mesma heurística, trabalham em cima de estruturas distintas quando da comparação das soluções.

O primeiro algoritmo (DOE), assim como o algoritmo IOE, faz uso da ordem de escalonamento das tarefas. O outro algoritmo (DTE), em contrapartida, utiliza-se dos tempos de escalonamento das tarefas (semelhante ao que ocorre com ITE).

Algoritmos de diferença, ao contrário do que acontece com algoritmos de intersecção (os quais buscam coincidências como base para composição de uma nova solução), partem das diferenças identificadas e geram novas soluções.

Na realidade estes algoritmos poderiam ser vistos como não consensos (ou mesmo discordância). Entretanto, pelo fato deste tipo de algoritmo fazer uso de duas soluções na geração de uma terceira, optou-se, por definição, no enquadramento de algoritmos de diferença como sendo algoritmos de consenso.

Com base na experimentação realizada constatou-se que a interação de algoritmos baseados em diferenças com algoritmos de consenso mostrou-se benéfica, sobretudo do ponto de vista de geração de diversidade dentro de um Time Assíncrono.

Basicamente estes algoritmos podem ser descritos fazendo-se algumas alterações nos algoritmos de intersecção IOE e ITE. Assim, o algoritmo DOE segue:

1. Escolha aleatoriamente duas soluções distintas L_a e L_b dentre as disponíveis na população de soluções completas.
2. Tome como solução base a melhor solução entre L_a e L_b (quebre empates arbitrariamente).
3. Realize uma intersecção baseada na ordem de escalonamento de L_a e L_b. Isto é, selecione as tarefas comuns a L_a e L_b que ocupam as mesmas posições em cada uma das listas.
4. Construa uma solução parcial composta de todas as tarefas da solução base que não pertencem à intersecção gerada (conservando a ordem em que as tarefas estão dispostas na solução base).

O algoritmo centrado em tempos de escalonamento é praticamente idêntico ao baseado na ordem de escalonamento. A única diferença está no fato de que a intersecção deste (passo 3) baseia-se em tempos de escalonamento.

Para uma melhor percepção destas duas variações dos algoritmos de diferença concebidos, observe as figuras 4.8 e 4.9. A primeira toma como base para obtenção da

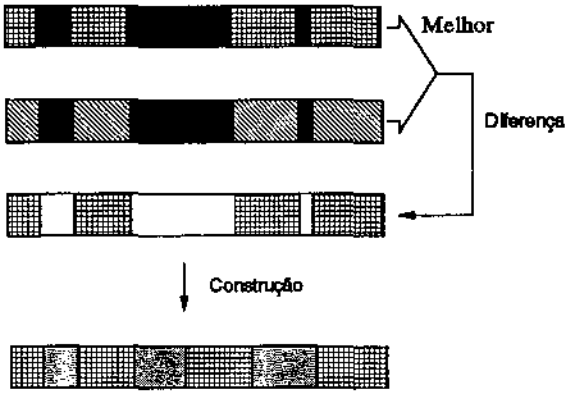


Figura 4.8 Algoritmo baseado em diferença segundo a ordem de escalonamento das tarefas (DOE). As partes mais escuras (intersecções) são desconsideradas na construção da solução parcial. Observe que apenas as tarefas relativas à melhor solução (menor *makespan*) são mantidas como integrantes da solução parcial.

diferença a ordem de escalonamento das tarefas. A última apresenta o algoritmo baseado em tempos de escalonamento.

4.2.3 *Políticas de destruição*

Políticas de destruição consistem em mecanismos para eliminação de soluções das memórias compartilhadas. Tais políticas definem as estratégias que os chamados algoritmos de destruição adotam para trocar soluções existentes nas memórias por novas soluções geradas. Este processo de reciclagem dentro das memórias é de grande importância para manutenção da diversidade e no refinamento da população de soluções.

Memórias simples, todavia, não necessitam de agentes destruidores para efetuarem a eliminação de soluções. Os próprios agentes que fazem acesso a estas memórias (escritas e leituras) podem se encarregar desta tarefa. Assim, estas memórias se assemelham a *buffers* onde os agentes depositam e retiram dados. Neste caso pode-se seguir uma política tipo fila - *FIFO* (*First In First Out*), onde, uma vez lida por um agente, a solução é automaticamente retirada da memória. Tipicamente, o tamanho destas memórias é irrelevante ao desempenho do *A-Team* construído, pois os dados nelas depositados são processados em um tempo finito. Na implementação realizada foram utilizadas duas memórias compartilhadas: a memória "Soluções Completas" e a memória "Soluções Parciais". Esta consiste de uma memória simples que segue uma política tipo *FIFO*. Quanto à primeira, consiste numa memória de soluções crescentemente ordenadas pelo valor de seus *makespans*, onde quatro políticas de destruição foram testadas como mecanismo de eliminação de soluções.

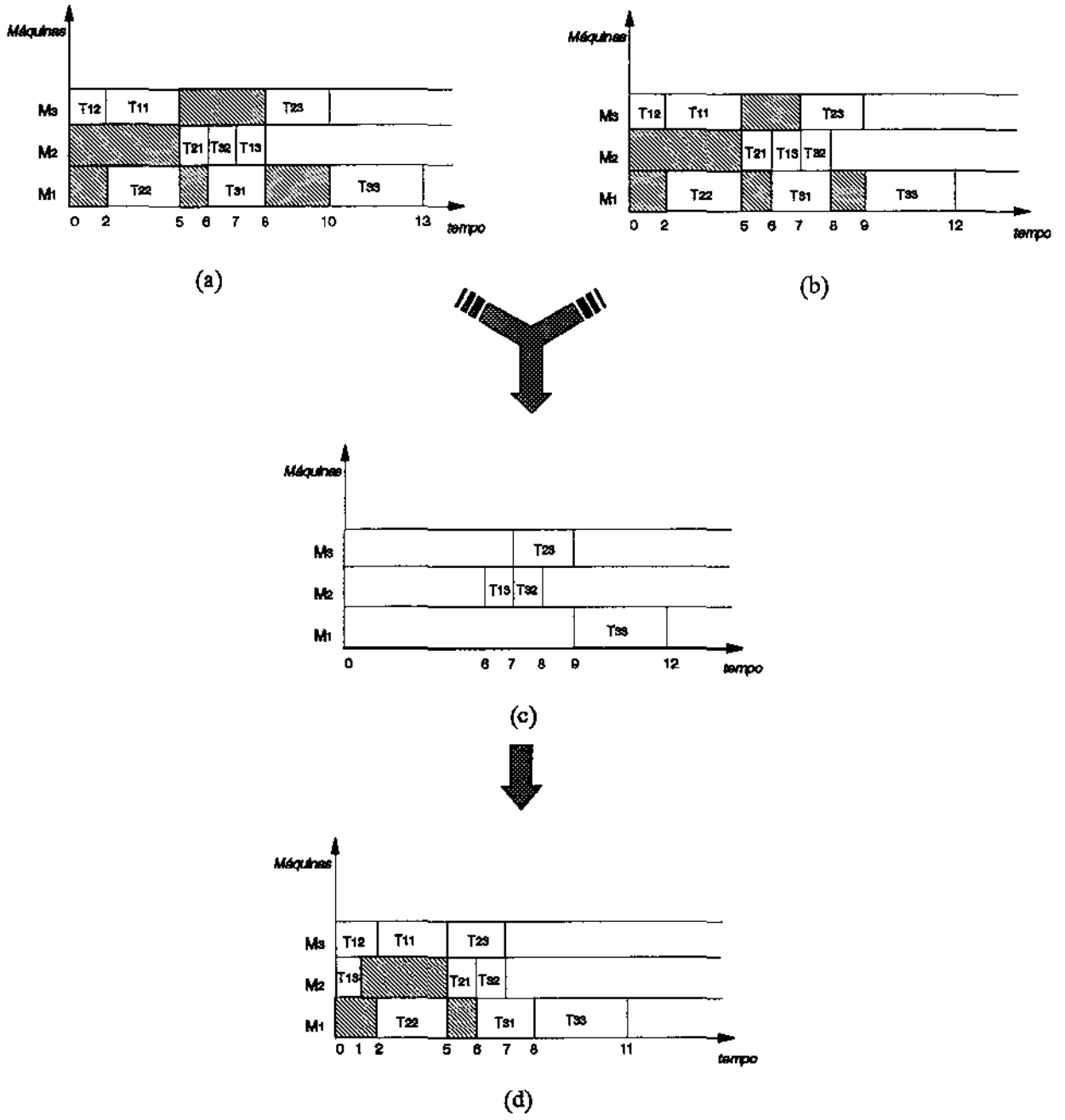


Figura 4.9 Algoritmo de diferença (DTE) centrado nos tempos de escalonamento de cada tarefa. Apenas as tarefas originárias da solução representada em (a) são mantidas quando da composição de (c). Observe que na fase de reconstrução (geração de (d)) é mantida a posição relativa entre as tarefas dentro de cada máquina.

A seguir, são detalhadas as políticas de destruição implementadas e ilustrados graficamente seus comportamentos através de experimentações feitas com uma instância para o JSP. O problema empregado (Law20) nos experimentos foi proposto por Lawrence [Law84] e é composto de 10 *jobs* e 10 máquinas (100 tarefas). As curvas esboçadas retratam, para cada política implementada, os valores dos *makespans* da pior e melhor soluções existentes na memória de soluções completas a cada melhora ocorrida.

1. Eliminação da pior solução (D1): sempre que uma nova solução é gerada, caso esta seja melhor que a pior solução da memória, esta pior é eliminada, liberando espaço para que a nova solução possa ser inserida. Em síntese, esta política procura sempre manter em "Soluções Completas" as melhores soluções disponíveis a cada instante (veja figura 4.10).

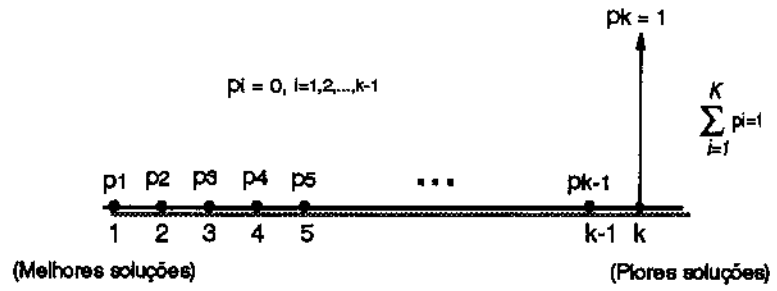


Figura 4.10 Política de destruição D1: a pior solução presente na memória é eliminada com probabilidade igual a um e as demais soluções nunca são eliminadas. Novas soluções são aceitas somente se forem qualitativamente melhores que a pior solução existente na memória de soluções completas.

Como resultados dos testes realizados com esta política, observou-se que, semelhante ao que ocorreu com o problema do Caixeiro Viajante em [Sou93], houve uma convergência abrupta das soluções. Estas rapidamente alcançaram configurações e valores de *makespans* similares.

A convergência muito rápida para soluções semelhantes ocasionou, em muitos casos, uma perda igualmente rápida de diversidade de soluções na memória. Este fato foi uma conseqüência razoavelmente previsível dentro do *A-Team*, dado que a maioria dos algoritmos faz uso das diferenças entre as soluções na geração de novas soluções. Em boa parte dos problemas testados, esta perda de diversidade pareceu prejudicar o *A-Team* no processo de fuga de ótimos locais, impedindo a ocorrência de grandes saltos qualitativos.

A queda abrupta na diversidade ocasionada pela rápida convergência propiciada pelo emprego de D1 pode ser observada no gráfico da figura 4.13.

2. Eliminação de soluções segundo uma distribuição de probabilidade uniforme (D2): neste caso, uma nova solução gerada é aceita, indiscriminadamente, e a solução da memória a ser eliminada é selecionada segundo uma distribuição de probabilidade

uniforme (exceção feita à melhor solução na memória, cuja probabilidade de ser eliminada é igual a zero - veja figura 4.11).

Esta política, apesar de evitar a rápida convergência ocorrida em D1, mantendo um excelente nível de diversidade, aceita que muita solução ruim seja introduzida na memória. Na implementação realizada verificou-se que este excesso de soluções "ruins" inseridas requereu um esforço computacional muito grande na obtenção de bons resultados. Isto se deu devido ao número de pontos do espaço de busca ter ficado demasiadamente grande.

Os resultados gerados com o emprego desta política foram mais fracos que aqueles obtidos com a política anterior (D1) devido, principalmente, à dificuldade de convergência dentro do *A-Team* com o uso de D2.

Figura 4.11 Política de destruição D2: a escolha da solução a ser destruída é feita de acordo com uma distribuição de probabilidade uniforme. A única exceção é a melhor solução na memória, cuja probabilidade de ser eliminada é igual a zero. Qualquer solução, independente de sua qualidade (valor de *makespan*), pode ser inserida na memória.

O gráfico da figura 4.14 revela o comportamento desta política.

3. Eliminação de soluções segundo uma distribuição de probabilidade linear (D3): esta política seleciona uma solução segundo uma distribuição de probabilidade linear, a qual tem probabilidade zero de selecionar a melhor solução na memória e cresce linearmente até a pior solução. A figura 4.12 ilustra esta distribuição de probabilidade.

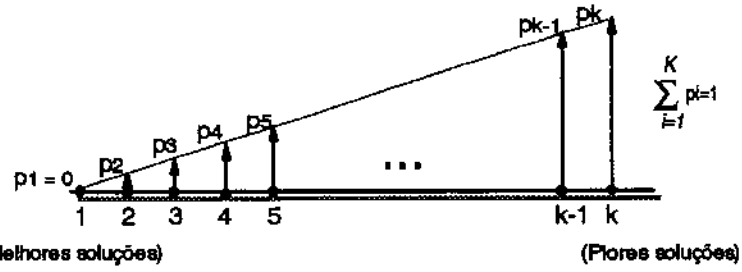


Figura 4.12 Política de destruição D3: as soluções são eliminadas segundo uma distribuição linear de probabilidade. Observe que a probabilidade da melhor solução ser eliminada é nula. Apenas soluções melhores que a pior existente na memória estão aptas a serem aceitas.

Na aceitação de uma nova solução gerada é utilizado o mesmo critério de D1, ou seja, aceitar somente soluções com *makespans* menores que o da pior solução existente na memória de soluções completas.

Esta política mantém um nível intermediário de diversidade entre D1 e D2. Ela mostrou um bom nível de compromisso entre a proporção de boas soluções (menores *makespans*) eliminadas em relação à eliminação de más soluções.

Verificou-se, entretanto, com o emprego desta política, dois comportamentos pouco desejáveis dentro da memória: (a) a restrição de não aceitar soluções piores que as existentes na população de soluções completas pareceu comprometer o nível de diversidade na memória, propiciando uma convergência relativamente rápida e pouco desejável e (b) a permanência de boas soluções na memória, apesar de acontecer com uma probabilidade maior que as demais, pareceu não contribuir significativamente na geração de novas soluções.

O gráfico esboçado na figura 4.15 ilustra o comportamento da memória quando aplicada a política de destruição D3 à instância escolhida.

4. Eliminação de soluções segundo uma distribuição de probabilidade linear combinada a um critério de tolerância (D4): esta política trabalha de forma semelhante à política D3. A diferença básica entre elas consiste em que, neste caso, foi acrescido um fator de tolerância à aceitação de novas soluções completas.

   Resultados experimentais mostraram que, apesar dos bons resultados obtidos com o emprego de D3, o problema de decréscimo rápido de diversidade dentro da memória de soluções completas permaneceu.

   Na busca do equilíbrio entre qualidade vs. diversidade, adicionou-se ao processo de aceitação de novas soluções um fator (taxa) de tolerância α. Assim sendo, soluções piores que a pior presente na memória de soluções completas poderiam ser aceitas. Como base para aplicação de α, utilizou-se o valor do *makespan* da melhor solução existente na memória (único valor disponível na memória garantidamente não crescente com o decorrer do tempo). Logo, caso uma nova solução produzida não fosse melhor que a pior solução da memória, esta seria aceita caso seu valor de *makespan* fosse estritamente menor que o *makespan* da melhor solução até então obtida acrescido de ($\alpha \times$ melhor *makespan*).

   Foram testados valores de α iguais a 0.1, 0.2 e 0.5. De acordo com este experimento, α igual a 0.1 (ou seja 10% do valor do *makespan* da melhor solução diponível) apresentou o melhor compromisso entre qualidade vs. diversidade dentro da memória. Os resultados das experimentações com variações de α podem ser vistos na tabela 4.1. Utilizou-se para testes três instâncias de Lawrence [Law84], compostas de 10 *jobs* e 5 máquinas. Estes três problemas foram submetidos à configuração de *A-Teams* considerada final (veja seção 4.3.2).

   A introdução da taxa de tolerância α à política de destruição teve pelo menos duas conseqüências fortes: (1) a eliminação do processo abrupto de convergência na população de soluções e (2) a possibilidade de se trabalhar com populações de dimensões relativamente pequenas, devido à garantia de conservação de um certo grau de diversidade.

Tabela 4.1 Resultados obtidos com variações do parâmetro de tolerância α para três instâncias do JSP. A tabela retrata a média de três testes com cada problema escolhido.

Instâncias	α = 10%	α = 20%	α = 50%
Law02	659,00	669,33	675,67
Law03	609,67	616,00	618,67
Law04	596,67	599,33	600,00

Na maioria das instâncias testadas a adoção desta política (com taxa de tolerância igual a 10%) gerou ganhos qualitativos consideráveis, atestando a importância de um ajuste adequado da política de destruição dentro de um Time Assíncrono. O comportamento desta política é exemplificado no gráfico da figura 4.16.

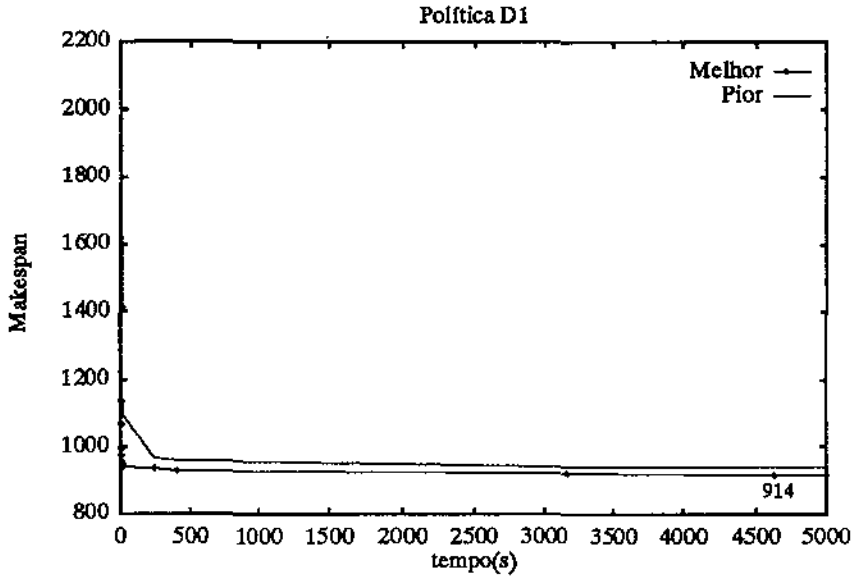


Figura 4.13 Comportamento da melhor e pior soluções na memória de soluções completas quando empregada a política D1. Observe a rápida convergência ocorrida na memória.

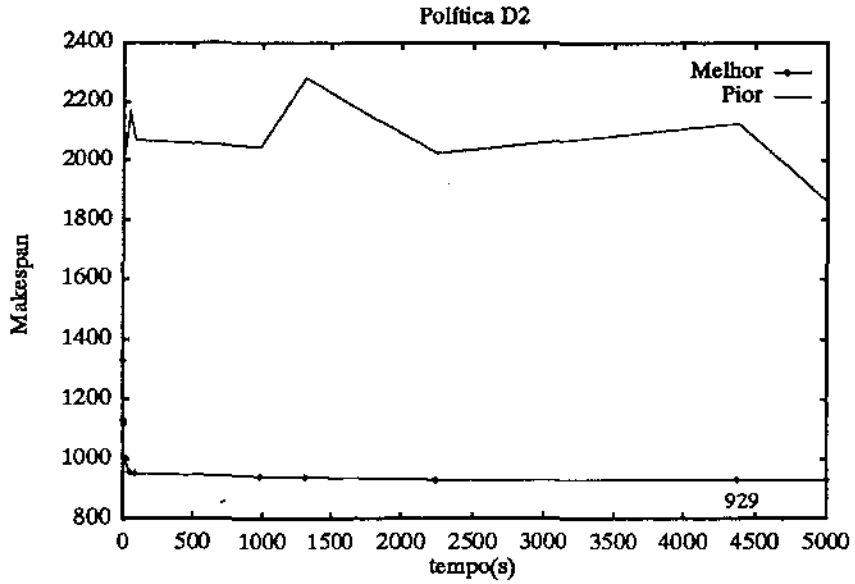


Figura 4.14 Comportamento da melhor e pior soluções na memória de soluções completas. Note que não há restrições quanto à presença de soluções ruins dentro da memória ao longo do tempo.

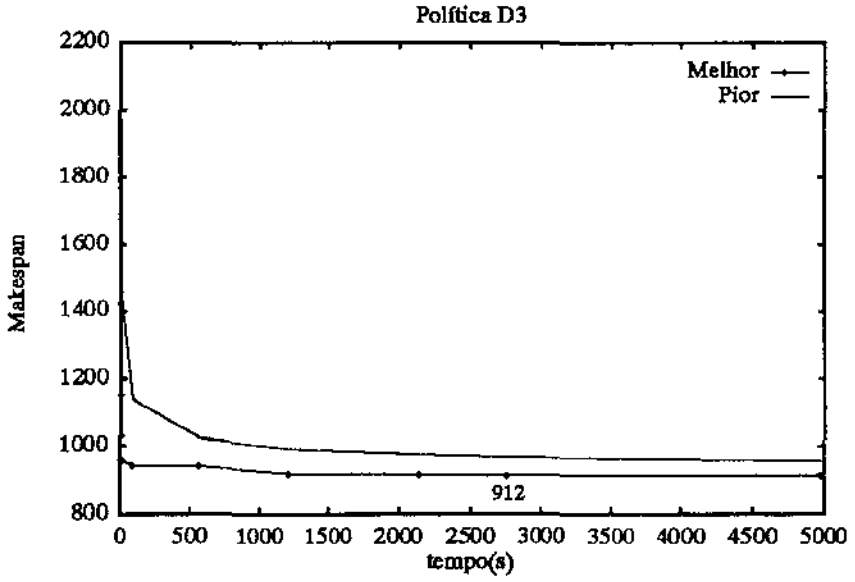


Figura 4.15 Comportamento da melhor e pior soluções na memória de soluções completas. Observe que a convergência foi postergada um pouco mais, neste caso, quando comparado a D1.

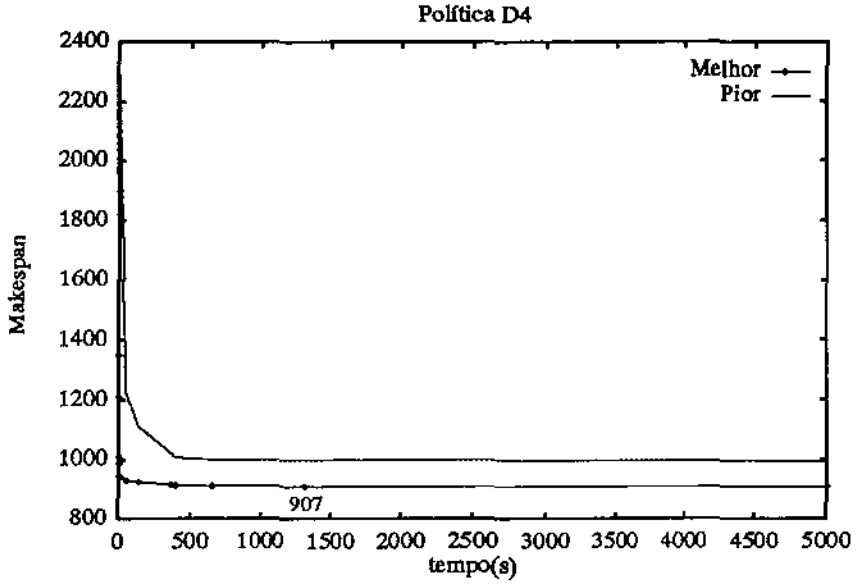


Figura 4.16 Comportamento da melhor e pior soluções na memória de soluções completas. Note que, apesar do princípio de convergência rápida, a intervenção do fator de tolerância α (aproximadamente a partir de t=500) descontinuou este processo, provendo um nível razoável de diversidade e possibilitando saltos qualitativos.

4.2.4 Algoritmos de construção

Diversidade é um elemento essencial e indispensável em se tratando de Times Assíncronos. A interação entre soluções distintas é o motor central na dinâmica de uma organização baseada em *A-Teams*. Assim, não fica difícil perceber que é altamente recomendável o não determinismo como característica intrínseca aos algoritmos de construção utilizados em *A-Teams*.

Face a esta necessidade, a elaboração de algoritmos de construção de comportamento não determinístico constitui-se em um primeiro passo na construção de *A-Teams* para solução do problema desejado.

No caso de problemas de escalonamento, e mais especificamente do JSP, a disponibilidade de inúmeras regras de prioridade [ABZ88,Fre82,OH85,PI77] como heurísticas de tomada de decisão, constitui-se no mecanismo mais acessível (barato e de fácil implementação) na geração de não determinismo.

Em termos de algoritmos de construção para o JSP, foram implementadas diversas heurísticas disponíveis na literatura e a elas foram acoplados mecanismos variados de obtenção de não determinismo. Além destas, outras heurísticas foram elaboradas e utilizadas como agentes de construção nos *A-Teams* concebidos.

Serão apresentados aqui dois mecanismos de geração de não determinismo implementados e, em seguida, serão descritos os algoritmos utilizados na construção de soluções para o JSP.

4.2.4.1 Janelas de Prioridade

Espaços de busca podem ser gerados definindo-se espaços de heurísticas. A chave do sucesso deste método é a habilidade de desenvolver versões parametrizadas de heurísticas, $h(\pi, p)$, que resolvem problemas específicos, onde p é o problema dado e π parametriza a heurística [SWV92].

Storer *et al.* [SWV92] definem um número de parametrizações de heurísticas baseadas em regras de despacho (ou prioridade) para o JSP. Uma das propostas apresentadas consiste em dividir as decisões de escalonamento em grupos (janelas) de acordo com a seqüência em que as decisões ocorrem no algoritmo. Portanto, dentre várias regras de despacho disponíveis, uma é associada a cada janela. Por exemplo, dados um problema composto de $\Im$ tarefas e a disponibilidade das regras SPT, MWKR e RANDOM (detalhadas mais adiante), a regra SPT pode ser usada para despachar as $\Im_0$ primeiras tarefas, MWKR para as $\Im_1$ seguintes e RANDOM para as $\Im_2$ últimas tarefas ($\Im_0 + \Im_1 + \Im_2 = \Im$). Genericamente, com **W** janelas a heurística é parametrizada por $(\pi_1, \pi_2 \ldots, \pi_w)$, onde cada π_j parametriza um conjunto de regras de prioridade já definidas (no exemplo, $\pi_j \in$ [SPT,MWKR,RANDOM]).

Esta proposta de manuseio do espaço de busca foi amplamente utilizada no presente trabalho. Basicamente, janelas de prioridade foram empregadas como meio para obtenção de algoritmos não determinísticos, de forma a direcionar as buscas sem, contudo, restringir em demasia o espaço a ser explorado pelos algoritmos.

Observe que a obtenção de algum sucesso utilizando-se este esquema de janelas está vinculada a boas escolhas relativas às regras de prioridade a serem associadas às janelas.

Na implementação efetuada foram utilizadas dez janelas de prioridade e seis regras de despacho. Como base para escolha das regras implementadas foram consideradas citações e classificações de regras de prioridade para o *Job Shop Problem* com mínimo *makespan*, descritas em [Bak74,Fre82,PI77]. Foram implementadas as seguintes regras de prioridade:

- SPT (*Shortest Processing Time*): seleciona uma tarefa com o tempo de processamento mais curto;
- MWKR (*Most Work Remaining*): seleciona uma tarefa que pertença ao *job* com maior tempo de processamento remanescente;
- LWKR (*Least Work Remaining*): seleciona uma tarefa que pertença ao *job* com menor tempo de processamento remanescente;

- MOPNR (***Most Operations Number Remaining***): seleciona uma tarefa que pertença ao *job* com o maior número de tarefas remanescentes;
- LOPNR (***Least Operations Number Remaining***): seleciona uma tarefa que pertença ao *job* com o menor número de tarefas remanescentes;
- RANDOM (***Random***): seleciona uma tarefa aleatoriamente.

Logo, tem-se que, nesta implementação, para cada π_j de $(\pi_1,\pi_2\ldots,\pi_w)$, $\pi_j \in$ [SPT,MWKR,LWKR,MOPNR,LOPNR,RANDOM].

4.2.4.2 Perturbações

Um segundo espaço de busca definido para problemas de escalonamento em [SWV92] e incorporado no presente trabalho está centrado na perturbação dos dados do problema. Em particular, sendo i,j os índices referentes à tarefa i do *job* j, os tempos de processamento das tarefas T_{ij} são perturbados aleatoriamente em um montante de tempo U_{ij}. Os novos tempos de processamento obtidos a partir da perturbação são usados em cada ponto de decisão em que é aplicada uma regra de despacho. Uma vez realmente escalonadas as tarefas, os tempos de processamento originais são usados para construir o escalonamento, isto é, associar às tarefas os momentos em que foram escalonadas nas respectivas máquinas.

Observe que a construção do espaço de busca está baseada no próprio problema (instância do problema) e não no espaço definido pela heurística que constrói as soluções, como acontece com as janelas de prioridade. A idéia central deste mecanismo é que boas soluções para um problema P' "vizinho" de P (problema original), podem constituir boas soluções para P.

Um ponto importante a ser ressaltado (e que será quantificado quando da descrição do algoritmo com base em perturbações implementado) é o parâmetro de controle do tamanho da perturbação. Perturbações muito grandes podem distorcer o conceito de problemas vizinhos, enquanto perturbações muito pequenas podem restringir demais o tamanho da vizinhança a ser explorada.

4.2.4.3 Notação Básica

Na descrição dos algoritmos de construção desenvolvidos e implementados, será feito uso de uma notação básica semelhante à adotada por French [Fre82] e Storer *et al.* [SWV92] na apresentação das heurísticas para o JSP por eles descritas.

Basicamente, a notação aqui utilizada é uma formalização um pouco mais detalhada da notação empregada na descrição dos algoritmos de Giffler e Thompson no capítulo 3. Assim, a seguinte notação e terminologia são usadas:

- T_{ij}: tarefa i do *job* j;
- M_{ij}: máquina onde T_{ij} deve ser executada;
- p_{ij}: tempo de processamento de T_{ij} em M_{ij};
- n e m: respectivamente, número de *jobs* e número de máquinas do problema;
- uma tarefa é dita escalonável se todas as tarefas que devem precedê-la dentro do seu *job* já foram escalonadas até o estágio t-1;
- dado que existem nm tarefas, o algoritmo iterage por nm estágios;
- no estágio t tem-se:
  - P_t: escalonamento das t-1 tarefas;
  - S_t: conjunto de tarefas escalonáveis no estágio t;
  - σ_{ij}: instante mais cedo que a tarefa T_{ij} em S_t pode ser iniciada, dado que suas predecessoras já foram escalonadas;
  - ϕ_{ij}: instante mais cedo que T_{ij} em S_t pode ser finalizada, isto é, $\phi_{ij} = \sigma_{ij} + p_{ij}$;
  - σ_{min}: $min_{T_{ij} \in S_t} \{\sigma_{ij}\}$;
  - σ_{max}: $max_{T_{ij} \in S_t} \{\sigma_{ij}\}$;
  - ϕ_{min}: $min_{T_{ij} \in S_t} \{\phi_{ij}\}$;
  - ϕ_{max}: $max_{T_{ij} \in S_t} \{\phi_{ij}\}$.

4.2.4.4 *A heurística GTWA*

Esta heurística está centrada no algoritmo para gerar escalonamentos ativos proposto por Giffler e Thompson em meados dos anos sessenta. Como mecanismo adicional de definição do espaço de busca desta heurística foi utilizado o esquema de janelas proposto por Storer *et al.* e descrito em 4.2.4.1. As janelas de prioridade foram usadas como elementos de decisão e empregadas nos momentos de escolhas indefinidas (quebra de impasses) do algoritmo original de Giffler e Thompson.

A principal motivação para implementar um algoritmo de construção que gere escalonamentos ativos provém do fato (como mencionado no capítulo 3) de que a classe de escalonamentos ativos contém pelo menos um escalonamento ótimo.

Como pode ser observado, a diferença fundamental entre GTWA e o algoritmo original de Giffler e Thompson para escalonamentos ativos está na escolha a ser efetuada no passo 4 de GTWA. Aqui, tal escolha não mais é efetuada de forma arbitrária ou baseada em uma única regra de prioridade, como ocorria no algoritmo original, mas sim de acordo com a regra de despacho associada à janela correntemente ativa no momento da escolha.

Algoritmo GTWA

Passo1: Faça $t = 1$, $P_1 = \emptyset$, S_1 = {Todas as primeiras tarefas de cada *job*},

n_w = número de janelas (W) a serem utilizadas e $W_{corrente} = W_1$

Passo2: Associe a cada janela W_k ($k = 1, ..., n_w$):

(i) uma regra de prioridade escolhida aleatoriamente dentre as disponíveis

(ii) um início para vigência de W_k (momento em que W_k passa a ser a regra corrente), tal que W_k.início = $(k-1) \cdot \lfloor n \cdot m / n_w \rfloor + 1$

Passo3: Calcule ϕ_{min} e faça M^* = Máquina onde ϕ_{min} ocorre (quebre impasses arbitrariamente)

Passo4: Escolha uma tarefa T_{ij} em S_t tal que

(a) T_{ij} requeira M^* e

(b) $\sigma_{ij} < \phi_{min}$. No caso de ocorrência de impasses (escolhas a serem efetuadas), resolva-os com base na regra de despacho associada à $W_{corrente}$. Caso persistam impasses, resolva-os aleatoriamente.

Passo5: Faça $P_{t+1} = P_t + \{T_{ij}\}$

Se $\exists T_{i+1,j}$, $S_{t+1} = S_t - \{T_{ij}\} + \{T_{i+1,j}\}$.

Caso contrário, $S_{t+1} = S_t - \{T_{ij}\}$.

t = t + 1

Passo6: Se $t \geq W_{k+1}$.início,

(i) k = k + 1

(ii) $W_{corrente} = W_k$

Passo7: Se $t \leq nm$, volte ao passo 3. Caso contrário, pare.

4.2.4.5 A heurística GTWSA

Uma outra abordagem bastante empregada como heurística para solução do *Job Shop Scheduling* baseia-se na construção de escalonamentos sem atraso [Bak74,CTS93,Fre82,HC90]. Apesar de ser um subconjunto dos escalonamentos ativos que não contém necessariamente um ótimo global, a classe de escalonamentos sem atraso tem sido ressaltada como, em média, detentora de melhores soluções que a classe de escalonamentos ativos [Bak74,CTS93]. Além disso, no caso de escalonamentos sem atraso, dispõe-se de um espaço de enumeração bem menor que aquele relativo a escalonamentos ativos. Este fato tem se constituído num atrativo adicional para a exploração da classe de escalonamentos sem atraso [HC90].

O algoritmo apresentado a seguir consiste numa versão modificada do algoritmo clássico de Giffler e Thompson para obtenção de escalonamentos sem atraso, descrito no capítulo 3. A diferença básica do algoritmo GTWSA para sua versão original está na adoção da proposta de Storer *et al.* [SWV92], da mesma forma que em 4.2.4.4, como mecanismo de definição de espaços de busca para o algoritmo.

Algoritmo GTWSA

Passo1: Faça $t = 1$, $P_1 = \varnothing$, S_1 = {Todas as primeiras tarefas de cada *job*},
n_w = número de janelas (W) a serem utilizadas e $W_{corrente} = W_1$

Passo2: Associe a cada janela W_k ($k = 1, \ldots, n_w$):
(i) uma regra de prioridade escolhida aleatoriamente dentre as disponíveis
(ii) um início para vigência de W_k (momento em que W_k passa a ser a regra corrente), tal que W_k.início = $(k-1) \cdot \lfloor n \cdot m / n_w \rfloor + 1$

Passo3: Calcule σ_{min} e faça M^* = Máquina onde σ_{min} ocorre (quebre impasses arbitrariamente)

Passo4: Escolha uma tarefa T_{ij} em S_t tal que
(a) T_{ij} requeira M^* e
(b) $\sigma_{ij} = \sigma_{min}$. No caso de ocorrência de impasses (escolhas a serem efetuadas), resolva-os com base na regra de despacho associada à $W_{corrente}$. Caso persistam impasses, resolva-os aleatoriamente

Passo5: Faça $P_{t+1} = P_t + \{T_{ij}\}$
Se $\exists T_{i+1,j}$, $S_{t+1} = S_t - \{T_{ij}\} + \{T_{i+1,j}\}$.
Caso contrário, $S_{t+1} = S_t - \{T_{ij}\}$.
t = t + 1

Passo6: Se $t \geq W_{k+1}$.início,
(i) k = k + 1
(ii) $W_{corrente} = W_k$

Passo7: Se $t \leq nm$, volte ao passo 3. Caso contrário, pare.

4.2.4.6 A heurística GTWH

Esta heurística está baseada numa proposta híbrida apresentada por Storer *et al.* [SWV92]. Basicamente, esta proposta consiste numa modificação dos algoritmos para geração de escalonamentos ativos e sem atraso desenvolvidos por Giffler e Thompson (1960). Como elemento adicional aos algoritmos originais de Giffler e Thompson, o algoritmo híbrido utiliza-se de um parâmetro $\delta \in [0,1]$. Este parâmetro determina o grau de flexibilidade associado à seleção de tarefas a serem escalonadas. Em termos práticos δ representa uma tolerância, quando da escolha de uma tarefa a escalonar, acrescida ao algoritmo para construção de escalonamentos sem atraso proposto por Giffler e Thompson. Como pode ser observado no algoritmo descrito mais adiante, quando $\delta = 0$, o algoritmo produz um escalonamento sem atraso; quando $\delta = 1$, o algoritmo gera um escalonamento ativo.

Tabela 4.2 Intervalos de variação testados para o parâmetro δ da heurística GTWH. Foi utilizado para testes o problema composto de 10 *jobs* e 10 máquinas proposto por Muth e Thompson.

Intervalo para δ	GTWH	
	Melhor *Makespan*	Média *Makespans*
[0,0.1]	993	1204,31
[0,0.25]	993	1206,77
[0,0.5]	987	1214,52
[0,1.0]	1003	1256,10

A idéia central do algoritmo híbrido é ampliar os pontos do espaço de busca em relação a escalonamentos sem atraso, sem ter que se confrontar com um espectro tão vasto quanto o de escalonamentos ativos. Esta abordagem apresenta como proposta principal a geração de soluções nas fronteiras de escalonamentos sem atraso e ativos como mecanismo alternativo para obter boas soluções que não seriam produzidas por algoritmos "puros".

Foram testados (neste trabalho e por Storer *et al.*) vários valores limites para flutuação de δ. Com base nestes testes, adotou-se [0,0.1] como intervalo de variação de δ, dado que esta faixa de valores produziu os melhores resultados, em média, na experimentação realizada (veja tabela 4.2). Note que foram usados valores de δ bem mais próximos de δ = 0 (escalonamentos sem atraso) que de δ = 1 (escalonamentos ativos).

A implementação desta heurística difere da proposta inicial de Storer *et al.* Enquanto nesta incorporou-se, a exemplo de GTWA e GTWSA, o esquema de janelas implementado (daqui provém o nome GTWH - Giffler e Thompson com janelas para escalomentos híbridos), Storer *et al.* fizeram uso de apenas uma única regra de prioridade no algoritmo híbrido original.

4.2.4.7 A heurística PERTURBA (PRT)

Como apresentado em 4.2.4.2, uma outra abordagem interessante, que pode ser empregada na solução do JSP está centrada na perturbação dos dados que definem o problema. Ou seja, variações (perturbações) em determinados dados de um problema podem ser efetuadas e utilizadas na composição de uma nova solução para o problema original. Este algoritmo, cujo espaço de busca está baseado no problema, foi implementado seguindo as idéias gerais para perturbação do problema apresentadas, acopladas ao algoritmo para gerar escalonamentos sem atraso descrito no capítulo 3.

Sendo P o problema a ser resolvido, a heurística efetua uma perturbação θ nos tempos de processamento das tarefas de P, construindo assim um novo problema P' (o qual é

Algoritmo GTWH

Passo1: Faça $t = 1$, $P_1 = \varnothing$, S_1 = {Todas as primeiras tarefas de cada *job*},
n_w = número de janelas (W) a serem utilizadas e $W_{corrente} = W_1$

Passo2: Associe a cada janela W_k ($k = 1, ..., n_w$):
(i) uma regra de prioridade escolhida aleatoriamente dentre as disponíveis
(ii) um início para vigência de W_k (momento em que W_k passa a ser a regra corrente), tal que W_k.início = $(k-1) \cdot \lfloor n \cdot m / n_w \rfloor + 1$

Passo3: Calcule σ_{min} e faça M^* = Máquina onde σ_{min} ocorre (quebre impasses arbitrariamente)

Passo4: Escolha uma tarefa T_{ij} em S_t tal que
(a) T_{ij} requeira M^* e
(b) $\sigma_{ij} \leq \sigma_{min} + \delta(\phi_{min} - \sigma_{min})$, $\delta \in [0,0.1]$. No caso de ocorrência de impasses (escolhas a serem efetuadas), resolva-os com base na regra de despacho associada à $W_{corrente}$. Caso persistam impasses, resolva-os aleatoriamente.

Passo5: Faça $P_{t+1} = P_t + \{T_{ij}\}$
Se $\exists T_{i+1,j}$, $S_{t+1} = S_t - \{T_{ij}\} + \{T_{i+1,j}\}$.
Caso contrário, $S_{t+1} = S_t - \{T_{ij}\}$.
t = t + 1

Passo6: Se $t \geq W_{k+1}$.início,
(i) k = k + 1
(ii) $W_{corrente} = W_k$

Passo7: Se $t \leq nm$, volte ao passo 3. Caso contrário, pare.

chamado de problema perturbado). P' é, então, resolvido, e a ordem em que suas tarefas foram escalonadas é usada como solução base para P.

Observe que, como exposto em 4.2.4.2, um fator importante nesta heurística é o nível θ de perturbação dos dados do problema original. Na presente implementação utilizou-se $\theta \in [-\overline{TP}, +\overline{TP}]$, onde $\overline{TP}$ é o tempo médio de processamento de uma tarefa de P e θ está distribuído uniformemente no intervalo descrito.

Esta heurística pode ser assim descrita:

Algoritmo PERTURBA

Passo1: Para cada tarefa T_{ij} de P, gere $T'_{ij} \in P'$, onde o tempo de processamento de T'_{ij} é igual ao tempo de processamento de T_{ij} acrescido de θ, $\theta \in [-\overline{TP}, +\overline{TP}]$.

Passo2: Use o algoritmo de Giffler e Thompson para escalonamento sem atraso na solução de P' (quebre impasses aleatoriamente).

Passo3: Adote a mesma seqüência de escalonamento das tarefas obtida quando da solução de P' como ordem de escalonamento para solução P.

Passo4: Tome os tempos de processamento de P e a ordem obtida a partir da solução de P' e calcule os instantes de escalonamento das tarefas para P, obtendo uma solução válida para P.

4.2.4.8 A heurística MTPR

Em geral, regras de prioridade têm sido empregadas na solução de problemas de escalonamento de diversas maneiras: como únicas heurísticas; combinadas a outras regras; na quebra de impasses, entre outras.

A presente heurística foi desenvolvida a partir de uma nova regra de despacho. Esta regra foi combinada, de uma nova forma (que não as usualmente disponíveis na literatura) a um algoritmo para construir escalonamentos sem atraso.

A nova regra (MTPR - Maior Tempo de Processamento Remanescente) é uma regra de prioridade simples que seleciona a tarefa que é executada numa máquina que possui o maior tempo de processamento remanescente. Diferente das regras de prioridade comumente encontradas, onde no processo de seleção basicamente enfocam as características das tarefas remanescentes dos *jobs* (tempos de processamento, quantidades, etc) ou o tempo de processamento já realizado pelas máquinas, MTPR atem-se ao trabalho por ser realizado em cada máquina. O intuito desta nova regra é propiciar um escalonamento mais uniforme no tocante à quantidade de processamento a ser efetuado pelas máquinas.

Sob um prisma amplo, o JSP com mínimo *makespan* pode ser visto como um problema de escalonamento de tarefas onde um montante de processamento deve ser realizado em um conjunto de máquinas, no mínimo tempo possível. Pode-se alternativamente, portanto, visualizar duas estruturas centrais no transcorrer da solução de um JSP :

1. Os dados de definição do problema ainda não selecionados no escalonamento do problema até aquele momento. Isto é, as tarefas remanescentes, uma vez que ainda não foram escolhidas para composição do solução correntemente sendo construída.
2. A solução parcialmente gerada, ou seja, as tarefas já escalonadas até aquele instante do escalonamento.

Assim, com base na figura 4.17, pode-se observar que, numa visão simplificada, o processo de escalonamento nada mais é que a transferência de dados (tarefas) da estrutura A para a estrutura B. Note que, no tempo $t = 0$ (antes, pois, de realizada qualquer transferência), a estrutura A fornece, de imediato, um limite inferior para o problema a ser resolvido.

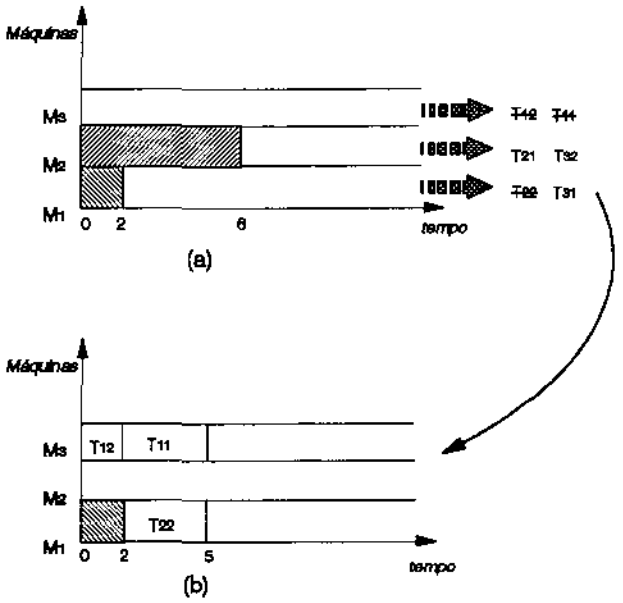


Figura 4.17 Exemplo de um estado intermediário do processo de transferência entre as estruturas (a) e (b) para um problema composto de seis tarefas. Na figura, as tarefas grifadas já foram escalonadas (tranferidas da estrutura (a) para a estrutura (b)). Os montantes de processamento por realizar nas máquinas um e dois equivalem, respectivamente, a duas e seis unidades de tempo.

A regra MTPR está, portanto, a cada escolha, preocupada com o estado da estrutura A. Esta abordagem pode parecer estranha, uma vez que o problema que se está resolvendo e cuja qualidade será avaliada tem sua configuração representada através da estrutura B. Além disso, não há restrições tecnológicas atreladas à estrutura A, tal como acontece com B. Este fato inviabiliza conclusões tipo: " boas seleções para A implicam escalonamentos adequados para B". Escolhas efetuadas a partir do maior tempo de processamento remanescente em A, poderiam ser extremamente inadequadas para composição de B, propiciando, por exemplo, um grande aumento de ociosidade das máquinas do sistema.

Cientes do problema causado pelas diferenças estruturais de A (em cima da qual MTPR está centrada) e de B, concatenou-se à MTPR algumas restrições de seleção e um esquema de escolha alternativa baseada em B.

Como restrições foram adotados dois limites considerados indesejáveis. Caso atributos da tarefa selecionada por MTPR assumissem valores limites, esta seleção seria ignorada, e um novo processo de seleção baseado em condições para gerar escalonamentos sem atraso seria ativado.

O algorimo é descrito resumidamente logo abaixo. Em seguida o mesmo algoritmo é apresentado mais detalhadamente, conforme a nomenclatura e notação previamente adotadas.

1. Obtenha um conjunto de tarefas "Escalonáveis" (S).
2. Escolha uma tarefa de S segundo MTPR.
3. Sejam os limites:

   3.1. MaxFim = maior tempo que uma tarefa "escalonável" levaria para ser concluída.

   3.2. MaxInício = tempo mais tardio em que uma tarefa "escalonável" poderia ser iniciada.
4. Se a tarefa escolhida em 2 não possui seus prováveis início e fim, respectivamente, iguais a MaxInício ou MaxFim, escalone-a e volte para 1. Caso contrário, escolha uma tarefa de S segundo condições de seleção de uma heurística para construir escalonamentos sem atraso.

Os limites MaxFim e MaxInício foram considerados indesejáveis devido à ociosidade que poderia ser introduzida nas máquinas com contínuas seleções de tarefas de início ou término tardios. Tal ociosidade ocasionaria atrasos no término do processamento das máquinas, o que é claramente indesejável para o JSP com mínimo *makespan*.

Antes de uma descrição detalhada do algoritmo baseado na regra MTPR implementado, faz-se necessária a incorporação da seguinte notação adicional:

- $L\mu$: lista de todas as máquinas M_{ij}, tais que $T_{ij} \in S_i$;
- μ^* : máquina m, tal que $TPR_m = max\{TPR_i\}$, $i \in L\mu$;
- $C(m)$: momento em que a última tarefa escalonada na máquina m foi completada.

Três observações importantes relativas à concepção da heurística MTPR merecem ser feitas:

1. A opção pelo uso de condições de seleção alternativa baseadas em escalonamentos sem atraso deveu-se, sobretudo, aos bons resultados obtidos. Condições para obtenção de escalonamentos ativos também foram testadas. Entretanto, em média, estas geraram soluções bem inferiores às obtidas a partir de condições de escalonamentos sem atraso.
2. A implementação desta heurística, a qual segue uma abordagem não usual quanto ao emprego da regra de prioridade MTPR, teve como principal estímulo o eventual benefício que um novo agente de construção, de natureza distinta dos já

Algoritmo MTPR

Passo1: Faça $t = 1$, $P_1 = \varnothing$, S_1 = {Todas as primeiras tarefas de cada *job*} e

$L\mu$ = {Todas as máquinas $m_1, m_2, \ldots, m_k$, $k \leq m \mid \exists T_{ij} \in S_t$ que é executada em $m_i (i = 1, 2, \ldots, k)$ }

Passo2: Calcule σ_{max}, ϕ_{max} e μ^* .

Passo3: Se $\exists T_{ij} \in S_t$ que satisfaça às seguintes condições:

1. $\sigma_{ij} = \sigma_{max} \vee i = 1$
2. $\phi_{ij} = \phi_{max}$
3. $M_{ij} = \mu^*$

Escalone T_{ij}. Quebre impasses aleatoriamente.

Senão

Calcule σ_{min} e M^* = Máquina onde σ_{min} ocorre (quebre impasses arbitrariamente)

Escalone $T_{ij} \in S_t \mid \sigma_{ij} = \sigma_{min} \wedge M_{ij} = M^*$

Quebre impasses aleatoriamente

$P_t = P_t + \{T_{ij}\}$

Passo4: Faça $P_{t+1} = P_t + \{T_{ij}\}$

Se $\exists T_{i+1,j}$, $S_{t+1} = S_t - \{T_{ij}\} + \{T_{i+1,j}\}$.

Caso contrário, $S_{t+1} = S_t - \{T_{ij}\}$.

t = t + 1

Passo5:Atualize $L\mu$ da seguinte forma:

$L\mu = L\mu - \{M_{ij}\}$

Se $\exists T_{i+1,j}$, $L\mu = L\mu + \{M_{i+1,j}\}$

Passo6: Se $t \leq nm$, volte ao passo 2. Caso contrário, pare.

implementados, poderia trazer quando introduzido numa organização de Times Assíncronos.

3. A heurística, apesar de fazer uso alternativo de critérios de seleção para gerar escalonamentos sem atraso, não necessariamente gera escalonamentos dessa classe. Este fato se dá devido a tais critérios serem levados em consideração apenas nos casos em que a regra MTPR é considerada não satisfatória. Ou seja, na construção de uma solução segundo a heurística MTPR pode-se ter tarefas selecionadas de acordo com a regra MTPR e, alternativamente, tarefas selecionadas de acordo com critérios para obtenção de escalonamentos sem atraso. Esta afirmativa vem reforçar o que foi dito em 2, no tocante ao diferente espaço de soluções geradas por esta heurística em relação às demais já apresentadas.

Um ciclo de otimização

Adicionalmente a este novo algoritmo desenvolvido, foi concebido um mecanismo heurístico de busca (que foi chamado de ciclo de otimização), cujo objetivo inicial era efetuar uma varredura em determinada vizinhança de uma solução (parcial ou completa), a procura de outra solução melhor.

Este ciclo de otimização pode ser aplicado basicamente em dois momentos do processo de construção de uma solução para o JSP, a saber:

1. Logo após o escalonamento de cada tarefa (ou de um conjunto de tarefas) de um problema. Neste caso o ciclo é incorporado como passo intermediário de um algoritmo de construção e age sobre uma vizinhança da solução parcialmente construída.
2. Logo em seguida ao escalonamento da última tarefa quando da solução de um problema. Particularmente nesta ocasião o ciclo pode ser interpretado como sendo um algoritmo de melhoria, que atua na vizinhança de uma solução completamente construída. Todavia, assim não foi considerado, devido a este caso se constituir apenas numa ocasião particular do processo que acontece em 1, ou seja, a última aplicação do ciclo de otimização.

Devido ao grande custo associado à incorporação do ciclo de otimização como passo intermediário e repetitivo no processo de construção de soluções, optou-se por utilizá-lo da forma bem menos exaustiva exposta no item 2 logo acima.

Resumidamente, no ciclo de otimização é tomada uma solução (parcial ou completa) e um conjunto de máquinas dentro das quais poderão ocorrer modificações no escalonamento. Uma vez determinado o conjunto de máquinas a serem "otimizadas", é construída uma hierarquia de máquinas consideradas gargalo (neste caso o grau de gargalo de uma máquina é definido pela medida do *makespan* da máquina).

A busca por soluções melhores é realizada tarefa a tarefa. O algoritmo de otimização procura efetuar trocas entre tarefas de uma mesma máquina. Esta trocas são realizadas segundo uma varredura, efetuada do começo para o fim na lista de tarefas já escalonadas. A cada passo de otimização procura-se antecipar uma tarefa dentro da máquina (trocá-la por outra anteriormente inserida na mesma máquina), respeitando-se as restrições do problema de escalonamento e objetivando-se obter uma solução melhor que a correntemente disponível.

A busca nas máquinas não é feita de forma exaustiva, dado que, conseguida alguma melhora dentro de uma máquina (redução do seu *makespan*), a tarefa seguinte na máquina assume o papel de tarefa base para próxima busca. Logo, uma única melhora dentro de uma máquina (a primeira, caso ocorra) determina uma mudança na busca por uma solução melhor.

Para descrição do ciclo de otimização proposto se faz necessária a seguinte notação complementar:

- $Ant(T_{ij}, M_{ij})$: retorna a tarefa antecessora à tarefa T_{ij} na máquina M_{ij};
- *UT*: conjunto das *m* últimas tarefas escalonadas e *P*;
- *LMG*: lista das máquinas eleitas como sendo gargalos;
- $Transfere(T_{ij}, T_{kl})$: transfere T_{ij} para a posição logo anterior a T_{kl} em *P*.

Ciclo de Otimização(*P*: solução; *Mk*: makespan de *P*)

Passo1: Calcule *UT* e faça *LMG* = { todas as máquinas $M_{ij} | T_{ij} \in UT$}

Passo2: Se $\forall (M \in LMG) | C[M] \neq Mk$

Então tome arbitrariamente uma máquina $M | M \notin LMG$ e $C[M] = Mk$

e faça $LMG = LMG + M$

Passo3: Ordene decrescentemente *LMG* pelos valores de *makespans* das máquinas

Passo4: (i) Para cada $M_{ij} \in LMG$

(ii) Para cada $T_{kl} \in P$ onde

(a) $M_{kl} = M_{ij}$

(b) T_{kl}.Início > 0

(c) $T'_{kl} = Ant(T_{kl}, M_{kl})$

(iii) Se $\exists T_{kl}$ tal que T'_{kl}.Início $\geq T_{k-1,l}$.Término

$Tranfere(T_{kl}, T'_{kl})$ em *P*, gerando *P'*

Faça *Mk'* = *Makespan* de *P'*

Se *Mk'* < *Mk*

Mk = *Mk'* e *P* = *P'*

Senão

$T'_{kl} = Ant(T'_{kl}, M_{kl})$ e vá para (iii)

Passo5: Retorne *P*, *Mk*

Como teste para observação do refinamento associado ao ciclo de otimização proposto foram utilizadas seis configurações do algoritmo MTPR. Cada configuração utilizava-se, respectivamente, de zero, um, dois, três, cinco e dez ciclos de otimização.

Com cada configuração foram testadas cem soluções para três problemas clássicos:

- Law02, composto de 5 *jobs* e 10 máquinas (totalizando 50 tarefas), proposto por Lawrence [Law84];
- MT10x10, composto de 10 *jobs* e 10 máquinas (totalizando 100 tarefas), proposto por Muth e Thompson [FT63];
- ABZ9, composto de 20 *jobs* e 15 máquinas (totalizando 300 tarefas), proposto por Adams *et al.* [ABZ88].

Com a escolha destes três problemas procurou-se obter uma visão panorâmica do comportamento do ciclo de otimização quando empregado na solução de problemas de dimensões distintas.

Para cada cem soluções geradas (as mesmas para todos os testes com cada problema), foram coletados os seguintes dados (veja tabela 4.3):

1. valor do *makespan* da melhor dentre as cem soluções construídas;
2. média dos *makespan* das cem soluções geradas;
3. percentual médio das melhoras obtidas, calculado sobre o número de soluções que conseguiram ser refinadas com a utilização dos ciclos;
4. tempo gasto para construção das cem soluções para cada configuração da heurística MTPR testada.

Com base na tabela 4.3, pode-se verificar que, para os três problemas, o emprego de mais que três ciclos de otimização não conseguiu incorporar qualquer melhoramento às soluções geradas. Além disso, o esforço computacional requerido em cada configuração pareceu ser, aproximadamente, proporcional ao número de ciclos utilizados pelo algoritmo.

Observe também que as melhoras conseguidas (em média) foram mais significativas para problemas menores, o que caracterizaria o ciclo de otimização proposto como sendo mais adequado à solução de problemas pequenos.

Outro fato importante, claramente perceptível com os resultados tabelados, é o descompromisso deste ciclo de otimização com garantia de melhorias. Observe que para a maior instância testada, por exemplo, o emprego de ciclos de otimização não trouxe modificações à melhor solução obtida. Todavia, o desempenho médio da heurística MTPR, mesmo com o uso de poucos ciclos de otimização (dois ou três), revelou-se bastante interessante, principalmente quando se objetiva produzir um conjunto de soluções de qualidade razoável sem comprometer em demasia o desempenho do sistema. Logo, o ciclo de otimização da forma não exaustiva como foi implementado, tem compromisso bem maior com o refinamento de soluções fracas que com a melhoria de boas soluções.

4.2.5 *Estratégias de iniciação*

Para que os algoritmos de um *A-Team* comecem a executar é necessária uma população inicial de soluções em alguma memória compartilhada (tipicamente na memória "Soluções Completas").

Diversas estratégias podem ser empregadas na obtenção da população inicial. Neste trabalho optou-se por iniciar a memória integralmente (100% do tamanho da memória), antes que os agentes começassem a trabalhar dentro dos *A-Teams*.

Tabela 4.3 Resultados da utilização de zero, um, dois, três, cinco e dez ciclos de otimização acoplados à heurística MTPR. São apresentados resultados relativos a cem soluções geradas para três problemas (Law02, MT10x10 e ABZ9) compostos, respectivamente, de 50, 100 e 300 tarefas. A tabela mostra, da esquerda para a direita, para cada configuração testada: o valor da melhor solução obtida; a média dos resultados obtidos; a melhora percentual alcançada com as soluções que conseguiram ser refinadas, e, o tempo total necessário para gerar as cem soluções. O sinal de desigualdade (<) indica que o tempo necessário requerido foi inferior ao valor especificado.

Instâncias	Número de Ciclos	Melhor Solução	Média das Soluções	% Média de Melhora	Tempo Total(s)
Law02	0	741	832,09	-	<1
	1	720	808,30	4,62	2
	2	720	806,25	5,01	4
	3	720	805,94	5,08	5
	5	720	805,94	5,08	9
	10	720	805,94	5,08	17
MT10x10	0	1092	1251,85	-	1
	1	1092	1233,27	2,34	6
	2	1081	1228,16	2,98	12
	3	1081	1227,55	3,05	17
	5	1081	1227,55	3,05	27
	10	1081	1227,55	3,05	55
ABZ9	0	814	908,19	-	10
	1	814	903,56	1,10	57
	2	814	902,70	1,30	90
	3	814	902,54	1,34	151
	4	814	902,54	1,34	244
	10	814	902,54	1,34	479

A princípio, não há regras bem definidas referentes à iniciação de memórias em *A-Teams*. Construtores de quaisquer natureza podem ser utilizados para gerar populações iniciais. Tipicamente, geradores aleatórios de soluções têm sido utilizados como agentes de iniciação [CS94,Had94,Pei95,PS94,Sou93]. Duas razões principais podem ser identificadas como justificativas para tal procedimento:

- a diversidade garantida a partir do uso de geradores aleatórios;
- a oportunidade de verificação do grau de refinamento (convergência) das soluções que ocorre em um *A-Team* no decorrer de sua execução.

Alternativamente às soluções geradas aleatoriamente, algoritmos menos ingênuos podem ser empregados na iniciação de memórias. Tais algoritmos podem ser utilizados como únicos iniciadores ou como um dos iniciadores responsáveis pela geração da população inicial. No transcorrer do presente trabalho foram desenvolvidos e testados três algoritmos como agentes exclusivos de iniciação, os quais são descritos a seguir.

4.2.5.1 Algoritmo aleatório

Este algoritmo gera escalonamentos baseados em escolhas aleatórias. Ou seja, a cada passo do algoritmo, a tarefa é aleatoriamente selecionada dentre as disponíveis (escalonáveis) naquele estágio de execução do algoritmo. Este algoritmo mostrou-se ser muito rápido e capaz de gerar uma enorme diversidade de soluções.

4.2.5.2 Algoritmo de escalonamento de blocos aleatórios

Este algoritmo procura escalonar, a cada passo, blocos de tarefas de um mesmo *job*. Assim sendo, as escolhas efetuadas por ele são baseadas em *jobs* e não em tarefas. A cada passo é escolhido um *job* J, aleatoriamente, dentre os *jobs* que ainda possuem tarefas a serem escalonadas (tarefas remanescentes nos *jobs*). Uma vez selecionado J, é sorteado um número n_t aleatório de tarefas de J a serem escalonadas ($0 < n_t \leq$ número de tarefas remanescentes).

O objetivo deste algoritmo é gerar soluções que apresentem grupos de tarefas de um mesmo *job* escalonadas de forma contígua. Este iniciador mostrou-se rápido e capaz de gerar soluções qualitativamente extremas (muito boas e muito ruins).

4.2.5.3 Algoritmo de escalonamento intercalado de tarefas

Este algoritmo atua sobre um espaço de soluções bem mais restrito que os dois algoritmos acima apresentados. Na verdade apenas 2n (onde n = número de *jobs* do problema) soluções distintas podem ser geradas por este iniciador.

Em síntese, este algoritmo escolhe aleatoriamente um *job* (ponto de partida) e uma ordem de escalonameto (crescente/decrescente) - por exemplo, $J_2,J_3\ldots,J_n,J_1$ - e, a cada n tarefas escalonáveis, o escalonamento é construído seguindo esta ordem. No exemplo, as n primeiras tarefas escalonadas seriam as seguintes: $T_{12},T_{13}\cdots,T_{1n},T_{11}$.

Note que, para instâncias com o mesmo número de tarefas em cada *job*, entre os escalonamentos de duas tarefas de um mesmo *job* , pelo menos n-1 tarefas de outros *jobs* foram escalonadas. Portanto, não é difícil perceber naturezas completamente distintas entre este algoritmo e o algoritmo de blocos aleatórios. Enquanto a idéia do escalonamento em blocos tem por meta propiciar algum escalonamento conjunto (contíguo) entre tarefas de um mesmo *job*, o escalonamento intercalado procura oferecer uma menor proximidade entre tarefas de um mesmo *job* e um escalonamento mais uniforme dos *jobs* no tocante ao número de tarefas escalonadas no decorrer da construção de uma solução.

A diferença clara de comportamento entre este algoritmo e o algoritmo baseado em blocos aleatórios constituiu-se na principal motivação para implementação destas heurísticas de construção. A intenção com a elaboração e emprego destes algortimos era gerar uma população inicial nitidamente heterogênea.

Este algoritmo, apesar de um pouco mais elaborado que os outros dois iniciadores, também é de fácil implementação e bastante rápido. As soluções geradas por ele apresentaram também uma grande diversidade e, qualitativamente, o algoritmo, mostrou-se capaz de produzir soluções superiores (em média) às geradas pelos outros dois iniciadores implementados.

4.3 *Resultados Computacionais*

Os testes computacionais realizados com as implementações efetuadas ao longo deste trabalho foram feitos em estações de trabalho SUN SPARCstation 10 (os tempos de processamento obtidos referem-se a tempos de CPU).

4.3.1 *Instâncias utilizadas*

Como testes para os *A-Teams* construídos, foi utilizada uma grande quantidade de instâncias do JSP. Estas cobriram um vasto espectro de problemas, variando desde problemas de tamanhos modestos compostos de dez *jobs* e cinco máquinas (cinqüenta tarefas), até instâncias de grandes dimensões compostas de cem *jobs* e vinte máquinas (duas mil tarefas).

Especificamente, foram utilizadas trinta e uma instâncias do JSP como testes para as experimentações efetuadas: vinte e cinco instâncias geradas por Lawrence [Law84], duas instâncias propostas por Muth e Thompson [FT63], três instâncias criadas por Adams *et al.* [ABZ88] e uma instância de Taillard [Tai93]. Estas instâncias foram empregadas em diversas fases do presente trabalho, seja em experimentos com heurísticas ou configurações específicas de *A-Teams*, seja nos testes com a implementação final de *A-Teams*.

As vinte e cinco instâncias de Lawrence utilizadas estão divididas em cinco grupos compostos de cinco instâncias cada, conforme as seguintes configurações:

- 10 *jobs* x 5 máquinas (50 tarefas);
- 15 *jobs* x 5 máquinas (75 tarefas);
- 20 *jobs* x 5 máquinas (100 tarefas);
- 10 *jobs* x 10 máquinas (100 tarefas);
- 15 *jobs* x 10 máquinas (150 tarefas).

As instâncias de Muth e Thompson submetidas à implementação realizada foram as clássicas 10 *jobs* x 10 máquinas e 20 *jobs* x 5 máquinas (ambas compostas de 100 tarefas). As instâncias geradas por Adams *et al.* escolhidas, compostas de 20 *jobs* e 15

máquinas (300 tarefas) são alguns dos mais difíceis problemas encontrados na literatura (são problemas ainda em aberto). Estes problemas, assim como os propostos por Lawrence [Law84], têm sido vastamente mencionados na literatura [AC91,ABZ88,BPN95,LAL92], sendo, pois, de grande valia na avaliação dos resultados obtidos.

A instância de Taillard [Tai93] utilizada reflete dimensões reais de problemas de escalonamento industrial. Tal instância foi considerada por Taillard uma das mais difíceis de serem resolvidas dentre aquelas por ele geradas. Particularmente, dentre os problemas de sua dimensão (duas mil tarefas), este foi o único para o qual Taillard não conseguiu alcançar o valor ótimo.

Para obter suas instâncias do JSP, Taillard usa um algoritmo gerador onde as sementes de tempo de processamento e máquina utilizadas como entrada pelo gerador definem cada instância. A configuração da instância selecionada e a semente extraída de [Tai93] utilizadas como entradas para o gerador implementado foram as seguintes:

- configuração: 100 *jobs* x 20 máquinas (2000 tarefas);
- $sementes_{(tempo,\ máquina)}$ = (682761130, 262763021).

4.3.2 *Configurações de A-Teams*

Várias configurações de *A-Teams* foram testadas e seus resultados analisados. Ajustes nos parâmetros dos *A-Teams* e dos algoritmos concebidos foram, paulatinamente, efetuados, buscando a obtenção de generalizações que promovessem um nível de compromisso desejável, senão, considerado satisfatório.

Os *A-Teams* concebidos foram implementados através de simulação de concorrência. Por simulação de concorrência entende-se um mecanismo seqüencial onde a ordem na execução dos algoritmos é determinada aleatoriamente, através de sorteio.

Nesta seção são ressaltadas algumas características verificadas com as configurações de *A-Teams* elaboradas e discriminadas a configuração de *A-Team* considerada a mais adequada desenvolvida durante o este trabalho, (por simplicidade, esta configuração será chamada de configuração final). Os resultados obtidos com estas configurações são apresentados, discutidos e confrontados com outros resultados conhecidos.

4.3.2.1 *Tamanho da Memória*

No caso do *Job Shop Problem* não observou-se relações precisas entre número de máquinas e/ou tarefas dos problemas em questão e o tamanho da memória mais adequado para resolvê-lo. Os tamanhos da memória de soluções completas testados para as diversas instâncias e os melhores resultados obtidos para cada configuração são apresentados na tabela 4.4. Estes tamanhos foram empiricamente encontrados e procuraram, sobretudo, estabelecer um compromisso satisfatório entre o tamanho do problema em solução (principalmente número de máquinas e tarefas) e o custo associado ao manuseio de memórias muito extensas. Os tamanhos tabelados foram

experimentados para um problema de cada dimensão. No caso das instâncias de Lawrence [Law84], onde cinco instâncias de problemas de cada dimensão foram utilizadas, procurou-se efetuar os testes com os problemas considerados os "mais difíceis" de resolver. Considerou-se como "mais difíceis" aqueles problemas cuja versão simplificada do *shifting bottleneck* não conseguiu resolver de forma ótima ou aqueles que requereram mais tempo para efetuá-lo.

Especificamente, foram utilizadas nos testes as seguintes instâncias: Law02, Law08, Law15, Law18, Law24, ABZ7 e Tai80, que são apresentadas na próxima seção. Nos casos em que, para duas configurações, os mesmos resultados foram alcançados, optou-se por utilizar aquela que o fez em menor tempo. Assim, na implementação testada, foram utilizados os tamanhos de memórias relativos à configuração 2, dado que estes revelaram um melhor compromisso entre qualidade dos resultados e tempo requerido para produzi-los.

Tabela 4.4 Tamanhos das memórias de soluções completas testados para os diversos problemas selecionados. Foram adotados como tamanhos das memórias os valores correspondentes à configuração 2. Os ajustes deste parâmetro procuraram estabelecer um compromisso entre qualidade obtida e esforço computacional dispensado satisfatórios.

Dimensão dos Problemas (n x m)	Configuração 1		Configuração 2		Configuração 3	
	Memórias (tamanho)	Melhor *Makespan*	Memórias (tamanho)	Melhor *Makespan*	Memórias (tamanho)	Melhor *Makespan*
10 x 5	25	677	50	655*	100	658
15 x 5	25	863*	50	863*	100	863*
20 x 5	25	1207*	50	1207*	100	1207*
10 x 10	30	848*	60	848*	100	861
15 x 10	30	992	60	990	100	992
20 x 15	100	729	150	718	200	718
100 x 20	100	5416	200	5383	500	5443

4.3.2.2 Configuração final

A configuração final, à qual foi submetida a bateria de testes já mencionados, é apresentada na figura 4.18.

Observe que são indicados como iniciadores da memória de soluções completas os algoritmos para geração aleatória de escalonamentos e o algoritmo de construção GTWH. Todavia, a utilização destes dois algoritmos não se deu de forma conjunta. O emprego de GTWH como iniciador só foi efetuado para a instância de Taillard. Além disso, o parâmetro δ deste algoritmo quando empregado como iniciador da memória obedeceu uma distribuição uniforme no intervalo [0,1]. A justificativa para o emprego de GTWH[0,1] como iniciador para a instância citada está nos seguintes fatos:

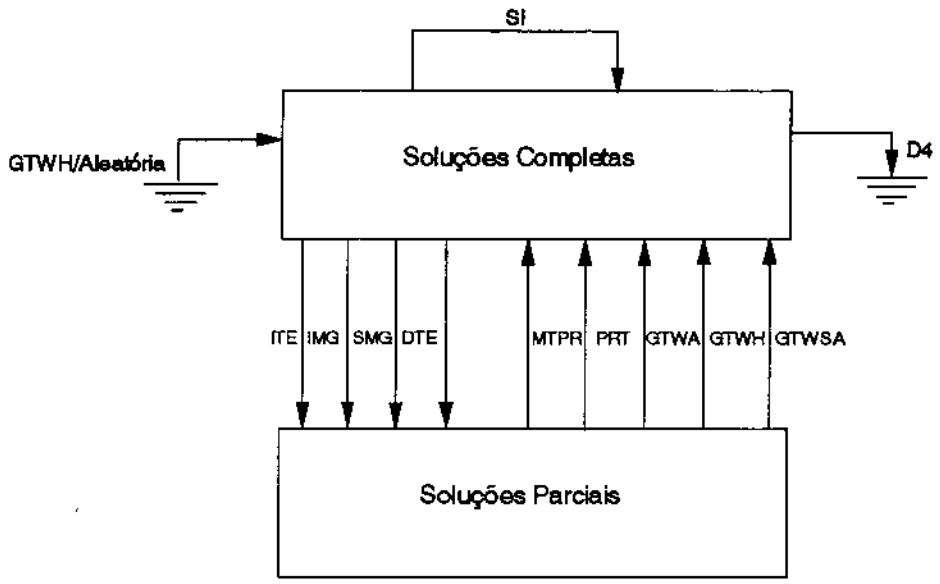


Figura 4.18 Configuração final implementada: fluxo de dados, memórias e agentes.

1. GTWH revelou-se experimentalmente o melhor algoritmo de construção implementado;
2. a variedade de soluções geradas por GTWH com δ distribuído no intervalo [0,1] mostrou-se grande o bastante para, no caso de instâncias de grandes dimensões, não comprometer a diversidade dentro da memória;
3. resultados experimentais mostraram que, para instâncias grandes, bons iniciadores provocavam uma diminuição no tempo necessário para alcançar boas soluções (quando comparados com iniciações aleatórias) sem, contudo, comprometer a qualidade dos resultados finais.

Além destes dois algoritmos, os demais iniciadores implementados também foram testados. Entretanto, os resultados obtidos não foram significativos o suficiente a fim de justificar a utilização destes outros algoritmos (seja simultânea ou isoladamente).

Quanto aos demais agentes da versão final, duas observações merecem ser feitas:

- Apesar de ter-se implementado os algoritmos de desconstrução baseados na ordem de escalonamento das tarefas (IOE e DOE), optou-se por utilizar os algoritmos baseados em tempos de escalonamento (ITE e IDE). Esta escolha se deu devido aos melhores resultados alcançados com estes últimos e à maior adequabilidade destes ao JSP (como já mencionado quando da descrição dos algoritmos).
- A heurística MTPR foi empregada sem ciclos de otimização, uma vez que o nível de refinamento propiciado às soluções geradas por ela dentro do *A-Team* permitiu dispensar o uso dos ciclos implementados.

Os resultados obtidos com a versão final são apresentados nas tabelas 4.5, 4.6 e 4.7. Para todas as três tabelas, os valores médios foram obtidos a partir da média aritmética de três execuções do *A-Team* para cada instância testada. A terceira tabela contém os resultados relativos às vinte e cinco instâncias geradas por Lawrence [Law84] empregadas neste trabalho. Esta tabela apresenta os resultados obtidos com a implementação dita final, e os confronta com os resultados conseguidos a partir da versão simplificada (SB-S) e da versão enumerativa do *shifting bottleneck* (SB-E) [ABZ88], e com os ótimos conhecidos para cada problema. A ausência de resultados tabelados para algumas das instâncias deve-se ao fato do algoritmo SB-E só ter sido testado para problemas não resolvidos por SB-S. As implementações do *shifting bottleneck* foram testadas em um VAX 780/11 e os tempos de CPU concedidos às versões SB-S e SB-E para solução dos problemas da tabela não ultrapassaram, respectivamente, trinta segundos e oito minutos.

Note que o *A-Team* produziu, para todos os problemas tabelados (veja tabela 4.7), resultados iguais ou superiores àqueles alcançados por SB-S. Para os problemas não resolvidos por SB-S, na metade dos casos, o *A-Team* conseguiu um desempenho não inferior ao apresentado pelo algoritmo SB-E, sendo que para três problemas (Law02, Law04 e Law18) apenas o *A-Team* conseguiu atingir a otimalidade. Observe também que, conforme a tabela 4.7, SB-S alcançou valores ótimos para onze problemas (44%), enquando o *A-Team* o fez para dezesseis problemas (64% dos testes).

É importante ressaltar que SB-E utiliza-se de uma árvore de enumeração parcial, realizando uma enumeração seletiva para obtenção de soluções melhores que as geradas pela versão mais simples do algoritmo. Assim, apesar da comparação dos resultados alcançados pelo *A-Team* aos obtidos por um algoritmo de enumeração parecer inadequada, optou-se por aqui fazê-lo como forma de ilustrar a qualidade alcançada pelo *A-Team* implementado.

Também mecerem ser destacados dois fatores importantes que levaram à não implementação do *shifting bottleneck* no presente trabalho: (1) a complexidade atribuída à programação deste algoritmo (mesmo em sua versão simplificada) [AC91,DL93], o que poderia requerer um montante de tempo não disponível; (2) os bons resultados obtidos a partir de heurísticas simples para o JSP (algumas vezes superiores àqueles produzidos pelo *shifting bottleneck*).

A tabela 4.5 apresenta os resultados obtidos pela mesma configuração do *A-Team* utilizado para os problemas de Lawrence [Law84]. Esta tabela mostra resultados obtidos para problemas propostos por Adams *et al.* [ABZ88]. Os problemas de Adams *et al.* são problemas razoavelmente grandes (trezentas tarefas), notoriamente de difícil solução (estando, ambos, ainda em aberto).

A tabela 4.6 apresenta os resultados obtidos para a instância de Taillard [Tai93] selecionada. O problema proposto recentemente por Taillard é extremamente grande (duas mil tarefas). Segundo Taillard [Tai93] foram empregadas técnicas de busca baseadas em *tabu search* para gerar soluções para os problemas por ele propostos.

Nos testes efetuados, os problemas de Adams *et al.* foram submetidos ao *A-Team* construído por aproximadamente quinze horas. Quanto ao problema gerado por Taillard, optou-se por testá-lo durante um período de tempo estimado (com base em [Tai93]) ser próximo àquele gasto durante os testes de Taillard.

Com estes dois últimos testes, intencionou-se averiguar o comportamento do *A-Team* concebido quando resolvendo problemas grandes e comprovadamente difíceis.

Com base na tabela 4.5 pode-se verificar que, apesar dos resultados conseguidos com *A-Teams* estarem relativamente distantes do melhor valor conhecido para cada problema (acima de 8% distantes), eles foram consideravelmente superiores àqueles alcançados pela versão heurística do *shifting bottleneck* (em média, acima de 12% distantes dos melhores conhecidos). Merece ser ressaltado aqui que, embora conhecidos os valores relativos aos limites superiores para tais problemas, os métodos usados para encontrar tais valores são desconhecidos, assim como os tempos e equipamentos requeridos para atingi-los. Estes valores foram conseguidos via recente comunicação pessoal [Nui95].

Quanto aos testes com o problema proposto por Taillard [Tai93], em um tempo aproximadamente igual ao que Taillard dispensou a testes com o problema, conseguiu-se um resultado distante apenas 3,26% do obtido por Taillard e 3,86% do ótimo para o problema.

Uma comparação mais clara entre a qualidade das soluções geradas pela versão heurística do *shifting bottleneck* com aquelas geradas pelo *A-Team* construído é ilustrada na tabela 4.8. Nesta tabela pode-se perceber que, para aquelas instâncias onde o ótimo não foi encontrado pelas duas heurísticas, as soluções geradas pelo *A-Team* estavam percentualmente, em média, duas vezes mais próximas do ótimo/melhor conhecido que as soluções gerada pela heurística de Adams *et al.*

Tabela 4.5 Resultados obtidos com as difíceis instâncias de Adams *et al.*

Instâncias	*A-Team*			SB - S	Melhores Valores Conhecidos
	Makespan (Melhor)	*Makespan* (Média)	Tempo (s) (Média)		
ABZ7	718	719,33	31.621,67	730	665
ABZ8	734	738,00	52.535,67	774	670

Tabela 4.6 Resultados obtidos para o problema de Taillard escolhido.

Instância	*A-Team*			Taillard	Ótimo
	Makespan (Melhor)	*Makespan* (Média)	Tempo (s) (Média)		
Tai80	5383	5396,67	33.703,33	5213	5183

Tabela 4.7 Resultados obtidos com o *A-Team* final, SB-S (shifting bottleneck simplificado) e SB- E (shifting bottleneck enumerativo) para as vinte e cinco instâncias de Lawrence utilizadas. O asterisco (*) indica que a solução é ótima, o hífen (-) indica que a instância não foi submetida ao algoritmo e a desigualdade (<) indica que o tempo requerido foi inferior ao indicado.

Instâncias	*A-Team*			SB- S	SB - E	Ótimo
	Makespan (Melhor)	*Makespan* (Média)	Tempo (s) (Médio)			
Law01	666*	666*	11,00	666*	-	666
Law02	655*	659,00	612,33	720	669	655
Law03	604	609,67	623,33	623	605	597
Law04	590*	596,67	740,33	597	593	590
Law05	593*	593*	<1,00	593*	-	593
Law06	926*	926*	2,33	926*	-	926
Law07	890*	890*	3,48	890*	-	890
Law08	863*	863*	11,00	866	863*	863
Law09	951*	951*	3,00	951*	-	951
Law10	958*	958*	2,33	958*	-	958
Law11	1222*	1222*	8,67	1222*	-	1222
Law12	1039*	1039*	11,00	1039*	-	1039
Law13	1150*	1150*	8,67	1150*	-	1150
Law14	1292*	1292*	<1,00	1292*	-	1292
Law15	1207*	1207,00	2021,00	1207*	-	1207
Law16	979	979,33	2919,33	1021	978	945
Law17	792	792,67	3302,67	796	787	784
Law18	848*	855,00	1874,67	891	859	848
Law19	866	867,33	3735,00	875	860	842
Law20	907	912,33	2045,33	924	914	902
Law21	1106	1120,33	4124,67	1172	1084	1046
Law22	967	968,33	5679,33	1040	944	927
Law23	1032*	1034,00	4274,00	1061	1032*	1032
Law24	990	993,67	5580,00	1000	976	935
Law25	1044	1046,00	6612,00	1048	1017	977

4.3.2.3 *Eficiência em Escala*

A conjectura levantada por Souza [Sou93] de que *A-Teams* são eficientes em escala é reforçada com experimentos realizados ao longo deste trabalho. As figuras 4.19 e 4.20 apresentam uma seqüência de introdução de agentes que resultou em uma crescente melhoria dos resultados obtidos.

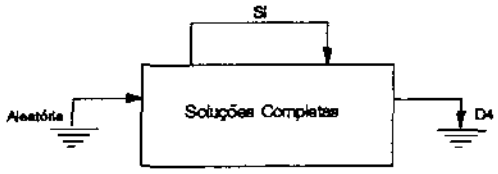
Tabela 4.8 Comparação dos resultados obtidos com *A-Teams* (versão final) confrontados com a versão simplificada do *shifting bottleneck*.

Instâncias	Distância do ótimo/ Limite Superior (%)	
	A-Team	SB - S
Law02	0,00	9,92
Law03	1,17	4,36
Law04	0,00	1,19
Law08	0,00	0,35
Law16	3,60	8,04
Law17	1,02	1,53
Law18	0,00	5,07
Law19	2,85	3,92
Law20	0,55	2,44
Law21	5,74	12,05
Law22	4,31	12,19
Law23	0,00	2,81
Law24	5,88	6,95
Law25	6,86	7,27
ABZ07	7,97	9,77
ABZ08	9,55	15,52
% médio	**3,09**	**6,46**

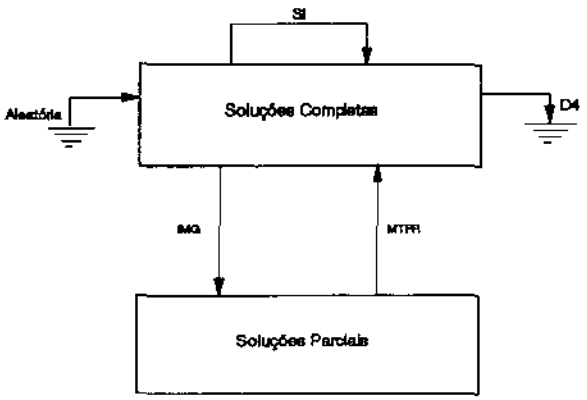
Observe que as configurações (a) e (b) diferem de dois agentes devido à necessidade de fechamento de uma ciclo dentro do *A-Team*.

Como ilustração do comportamento eficiente em escala obtido, testes computacionais foram feitos com uma instância proposta por Muth e Thompson [FT63], composta de vinte *jobs* e cinco máquinas (totalizando cem tarefas). Para cada configuração foram realizados três testes e a média das três execuções é explicitada na tabela 4.9. A configuração (i) refere-se àquela adotada como final (veja figura 4.18).

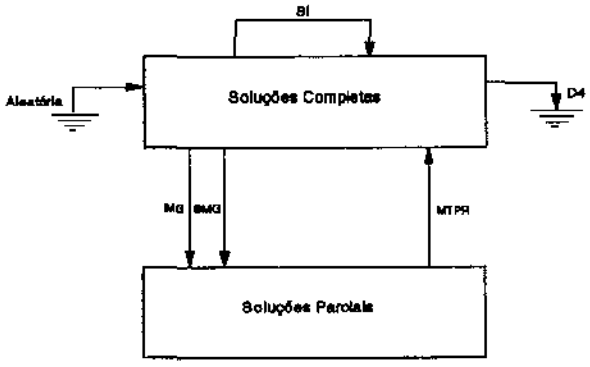
A melhoria monotônica ocasionada pela inserção gradativa dos agentes pode ser mais claramente percebida através do gráfico da figura 4.21. Este gráfico mostra os resultados alcançados a cada algoritmo inserido a partir da configuração (a). Note o ganho substancial obtido com a incorporação do segundo agente de construção (configuração (f)).



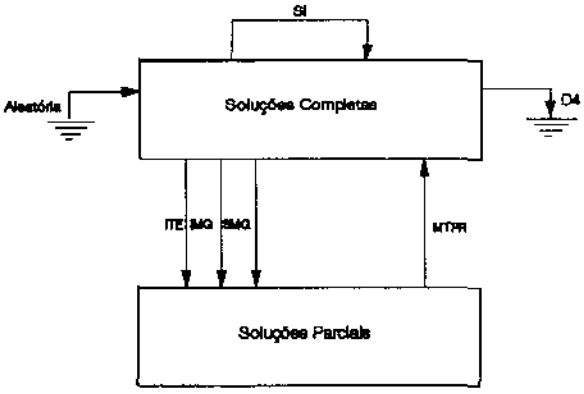
(a)



(b)

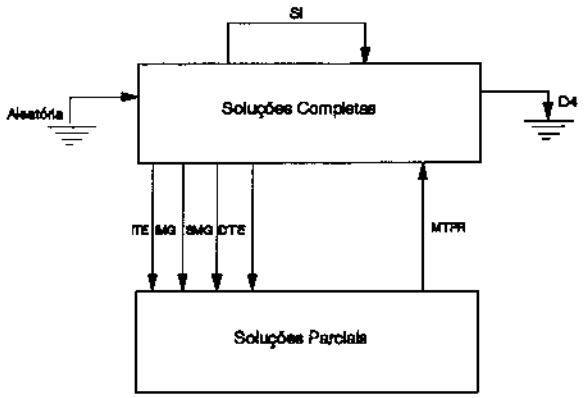


(c)

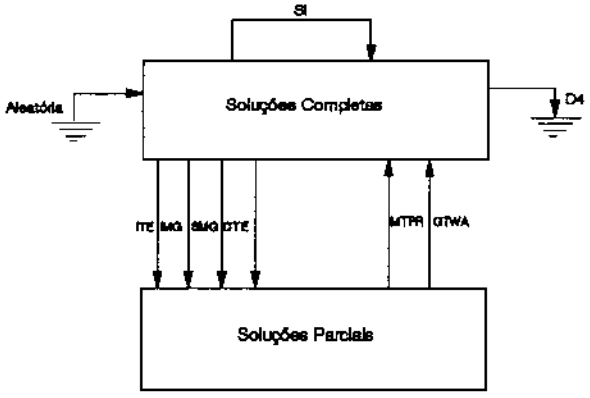


(d)

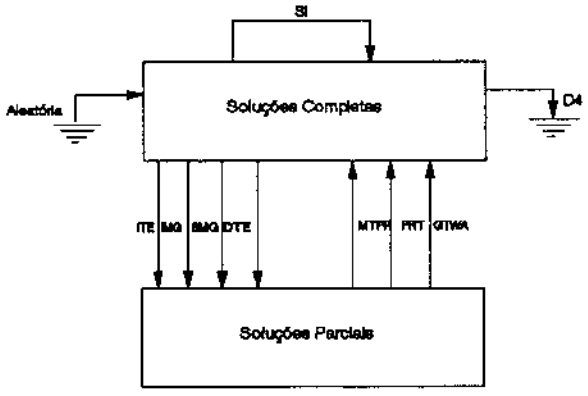
Figura 4.19 Seqüência (a), (b), (c) e (d) dos fluxos de dados utilizados para verificação de Eficiência em Escala.



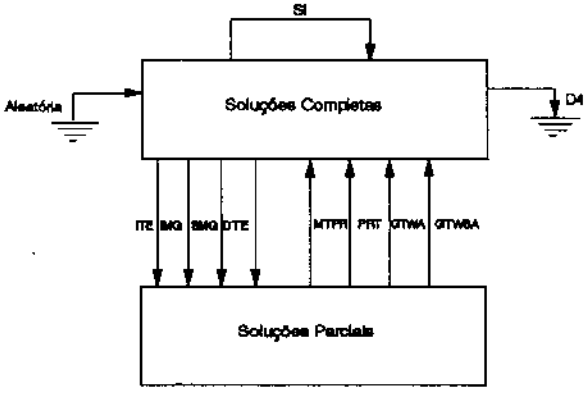
(e)



(f)



(g)



(h)

Figura 4.20 Seqüência (e), (f), (g) e (h) dos fluxos de dados utilizados na verificação de Eficiência em Escala.

Tabela 4.9 Valores médios obtidos com cada configuração para uma instância de cem tarefas.

MT - 20x5		
Configurações	**Número de Algoritmos**	***Makespans* (Médias)**
(a)	1	1593,67
(b)	3	1480,00
(c)	4	1476,33
(d)	5	1468,33
(e)	6	1457,00
(f)	7	1264,67
(g)	8	1237,33
(h)	9	1206,00
(i)	10	1193,00

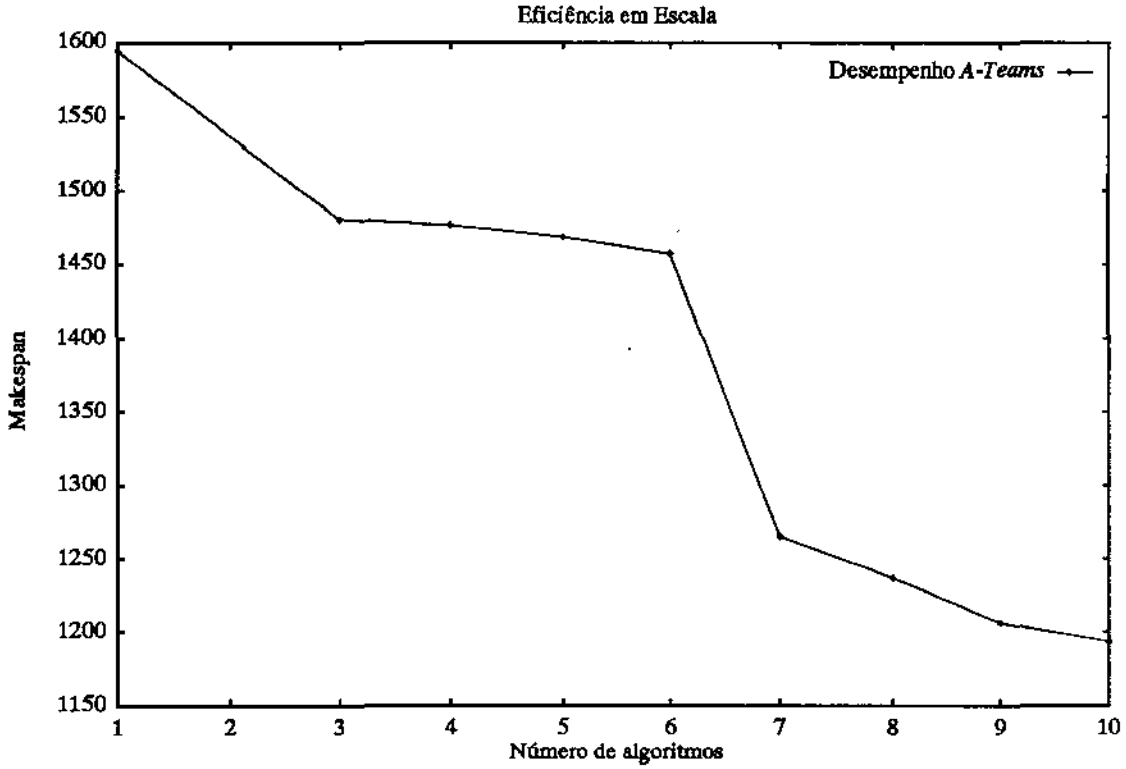


Figura 4.21 Curva ilustrativa da Eficiência em Escala obtida para um problema de Muth e Thompson. Note a melhoria monotônica dos resultados a cada novo agente introduzido.

Também, com base no gráfico ilustrativo de eficiência em escala (figura 4.21), pode-se identificar que esta ocorre de forma categorizada, isto é, dentro de um certo tipo (ou categoria) de agentes. Note que tanto com a introdução do quarto, quinto e sexto agentes (configurações (c), (d) e (e)), como quando da inserção do oitavo, nono e décimo agentes (configurações (g), (h) e (i)) verificou-se eficiência em escala com a introdução de um mesmo tipo de agente: agentes de desconstrução (os três primeiros) e agentes de construção (os três seguintes). Observe que esta eficiência em escala categorizada pode, entre outras, servir de auxílio na decisão de incorporação (ou não) de agentes de um mesmo tipo dentro de um *A-Team*.

4.3.2.4 Sinergia

Com a finalidade de ilustrar o caráter sinérgico dos *A-Teams*, realizaram-se dois testes básicos:

1. Foram submetidas instâncias de Lawrence [Law84] aos agentes de construção utilizados na implementação efetuada, durante o mesmo tempo que o *A-Team* final (em média) necessitou para alcançar seus melhores resultados. Dentre as vinte e cinco instâncias de Lawrence foram escolhidas para este teste aquelas para as quais a versão simplificada do *shifitting bottleneck* [ABZ88] não conseguiu alcançar o ótimo, ou aquelas que, quando submetidas à configuração final, não tiveram sua otimalidade atingida em todas as três execuções. Desta forma, puderam ser verificados os seguintes aspectos:

   1.1. O nível de qualidade das heurísticas de construção implementadas frente aos problemas que se revelaram de mais difícil solução dentre os vinte e cinco testados. Desta forma é possível, adicionalmente, observar se tais heurísticas constituem-se em algoritmos adequados [Sou93, Pei95] para solução dos problemas testados, ou seja, se estão aptas a encontrar as soluções desejadas no tempo disponível.

   1.2. O comportamento destes algoritmos isoladamente comparado àquele revelado quando da inserção dos algoritmos no *A-Team* construído.

2. Foi adotada uma estrutura de *A-Team* composta de um único algoritmo de construção (tomou-se arbitrariamente GTWH) conjuntamente com todos os algoritmos de desconstrução empregados na configuração final, e confrontou-se seus resultados com os deste algoritmo executado isoladamente. Neste teste, utilizou-se GTWH também na geração da população inicial, a fim de se evitar a presença (interferência) de qualquer outro algoritmo de construção neste experimento. Tomou-se para esta experiência a instância clássica 10 *jobs* x 10 máquinas de Muth e Thompson [FT63], a qual foi submetida a três execuções do *A-Team*.

Os resultados obtidos com estes experimentos revelaram nítido benefício causado pela cooperação dentro do *A-Team*. Com base no experimento 1 (veja tabela 4.10), pode-se observar que a manifestação de um comportamento sinérgico em *A-Teams* parece ter uma caráter genérico, ou seja, pouco dependente de especificidades do problema que está sendo tratado. Note que, para quase a totalidade dos problemas, o *A-Team* conseguiu atingir resultados melhores que os alcançados pelos algoritmos isoladamente e que, em nenhum caso, obteve resultado inferior.

Tabela 4.10 Resultados obtidos com a execução isolada de cada algoritmo de construção implementado e da melhor versão de *A-Teams* concebida.

Instâncias	GTWA	MTPR	PRT	GTWSA	GTWH	*A-Team*
Law02	699	684	682	677	677	655*
Law03	650	636	638	619	623	604
Law04	611	611	611	611	611	590*
Law08	866	863*	863*	863*	863*	863*
Law15	1256	1241	1230	1210	1209	1207*
Law16	1001	993	998	990	983	979
Law17	830	822	796	793	792	792
Law18	923	861	864	863	860	848*
Law19	925	887	874	875	875	866
Law20	955	941	914	938	917	907
Law21	1213	1131	1128	1120	1124	1106
Law22	1076	1000	1011	967	975	967
Law23	1160	1049	1038	1032*	1037	1032*
Law24	1082	1025	1019	1003	993	990
Law25	1090	1078	1104	1052	1048	1044

O experimento 2, além de servir como crédito adicional ao experimento 1, ressalta o ganho que pode ser atribuído à execução de um algoritmo quando da inserção deste em um *A-Team*. Tal observação evidencia ainda mais o aspecto de refinamento comumente verificado dentro de Times Assíncronos. Observe, com base na tabela 4.11, que tanto no melhor caso como em média o *A-Team* conseguiu melhores resultados que GTWH isoladamente (melhor heurística implementada). Este fato deixa nítido o caráter de refinamento atribuído à técnica de *A-Teams*.

Tabela 4.11 Resultados da execução de GTWH isoladamente e dentro de uma organização A-Teams.

MT - 10x10		
A-Team		GTWH
Melhor	Média	
967	969	972

4.4 *Sumário*

Neste capítulo foram apresentadas detalhadamente a experimentação atrelada à presente dissertação, bem como a fundamentação teórica na qual estão centradas as implementações realizadas.

Inicialmente, introduziram-se conceitos básicos e características gerais relativas à teoria de Times Assíncronos. Foi também definido o enfoque adotado pelo presente trabalho, isto é, o aspecto construtivo para o JSP.

Em seguida foram apresentados os algoritmos para *A-Teams* desenvolvidos e as idéias associadas a cada heurística concebida. Algumas heurísticas para *A-Teams* disponíveis na literatura [Sou93] foram adaptadas ao problema atacado, assim como novas formas de manipulação de soluções foram propostas e implementadas.

Em acréscimo aos algoritmos específicos para *A-Teams*, foram descritos os algoritmos de construção empregados neste trabalho. Algumas heurísticas clássicas para o JSP foram acopladas a novas propostas para manipulação do problema. Além disso, foi elaborada e testada uma nova heurística de construção para o problema.

Foram descritos os *A-Teams* desenvolvidos para o JSP. Explicitaram-se os fluxos de dados e a estrutura e forma de manipulação (políticas de destruição) das memórias utilizados.

Finalmente, foram apresentados os resultados computacionais alcançados com os *A-Teams* concebidos. Características típicas de *A-Teams* foram verificadas e os resultados obtidos a partir da implementação concebida foram confrontados com aqueles disponíveis na literatura. Como base para comparação do desempenho dos *A-Teams* implementados com aqueles relativos a outros métodos aproximados, utilizaram-se diversas instâncias de problemas de dimensões distintas, as quais são bastante citadas na literatura existente para o JSP.

Capítulo 5 *Conclusão*

Esta dissertação descreve a utilização da técnica de Times Assíncronos na solução do problema de escalonamento de tarefas conhecido por *Job Shop Scheduling Problem*. Especificamente, foram desenvolvidos *A-Teams* para o JSP enfocando o aspecto de construção heurística de soluções para o problema.

Foram elaborados e implementados diversos algoritmos de construção para o JSP, bem como heurísticas específicas para *A-Teams*. Um vasto espectro de instâncias do problema (de diversos tamanhos e graus de dificuldade) foram empregadas como testes para a implementação concebida, objetivando a formação de um padrão comportamental dos *A-Teams* desenvolvidos quando resolvendo um JSP.

Características desejáveis, tais como eficiência em escala e cooperação sinérgica, se mostraram presentes nos *A-Teams* implementados. Além disso, os bons resultados obtidos com o emprego de *A-Teams* para o problema tratado comprovaram a adequabilidade desta técnica à solução do *Job Shop Scheduling Problem*.

Qualitativamente, os *A-Teams* implementados mostraram-se capazes de produzir resultados bastante competitivos quando confrontados com aqueles produzidos por outros métodos aproximados e com os melhores resultados disponíveis na literatura.

É válido ressaltar que, apesar dos *A-Teams* construídos se utilizarem apenas de algoritmos simples para solução do JSP, testes realizados com instâncias clássicas do problema revelaram resultados nunca inferiores àqueles produzidos pela melhor heurística conhecida para o problema, sendo que esta heurística é sabidamente bastante intrincada e de difícil codificação.

5.1 Contribuições

- concepção e implementação de uma nova heurística de construção para o JSP (MTPR);
- comprovação da adequabilidade de *A-Teams* ao domínio de problemas de escalonamento de tarefas e, mais especificamente, ao *Job Shop Problem*;
- comprovação de que, para o JSP, algoritmos simples trabalhando conjuntamente são capazes de produzir melhores resultados que quando executados isoladamente;
- implementação de algoritmos de construção clássicos para o *Job Shop Scheduling Problem* acoplados a heurísticas de busca recentemente propostas;
- desenvolvimento de novas heurísticas para elaboração de fluxos de dados dentro de um *A-Team*;
- proposta de uma heurística para refinamento de soluções do JSP;
- verificação de que mesmo *A-Teams* baseados exclusivamente em algoritmos de construção apresentam um caráter de melhoria, isto é, conseguem partir de soluções completas para um problema e gerar soluções melhores;
- constatação de que eficiência em escala em *A-Teams* é uma conjectura que também se verifica para o JSP.

5.2 Possíveis extensões e trabalhos futuros

- estudo da viabilidade de incorporação de algoritmos de construção complexos para o JSP à estrutura de *A-Teams* desenvolvida (por exemplo, o *shifting bottleneck*);
- desenvolvimento e incorporação de agentes de melhoria para o JSP à estrutura já proposta;
- implementação de uma versão paralela (ou distribuída) de *A-Teams* para o JSP, objetivando mostrar a potencialidade da técnica ao paralelismo e possibilitando testes mais exaustivos com problemas grandes;
- desenvolvimento de novos fluxos de dados para o JSP;

Referências

[ABZ88] J. Adams, E. Balas e D.Zawack, *The shifting bottleneck procedure for the job shop scheduling*, Management Science, Vol. 34, 1988.

[AC91] D. Applegate e W. Cook, *A Computacional Study of the Job Shop Scheduling Problem*, ORSA Journal on Computing, Vol. 3, No 2, 1991.

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

[Bal69] E. Balas, *Machine Sequencing Via Disjunctive Graphs: An Implicit Enumeration Algorithm*, Operation Research, 1969.

[BESW93] J. Blazewicz, K. Ecker, G. Schimidt e J. Werglarz, *Scheduling in Computer and Manufacturing Systems*, Spring-Verlag, Berlin, 1993.

[BJ93] P. Brucker e B. Jurish, *A New Lower Bound for the Job-Shop Scheduling Problem*, European Journal of Operational Research, 156-167, 1993.

[BPN95] P. Baptist, C. L. Pape, W. Nuijten, *Incorporating Efficient Operations Research Algorithms in Constraint-Based Scheduling*, to appear in: Proceedings of the Fisrt International Joint Workshop on Artificial Intelligence and Operations Research, Timberline Lodge, Oregon, 1995.

[Car82] J. Carlier, *The one-machine sequencing problem*, European Journal of Operational Research 11 (1982) 42-47.

[Cof76] E. G. Coffman, Jr. *et al.*, *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, New York, 1976.

[CP89] J. Carlier, E. Pinson, *An Algorithm for solving the job-shop problem*, Management Science, Vol. 35, 1989.

[CS94] V. F. Cavalcante e P. S. de Souza, *Times Assíncronos na Solução do Job Shop Scheduling Problem*, Anais do XXVI Simpósio Brasileiro de Pesquisa Operacional, pp 791-796, Florianópolis - SC, 1994.

[CS95] V. F. Cavalcante e P. S. de Souza, *Solving the Job Shop Scheduling Problem by Asynchronous Teams*, to appear in: Proceedings of International Symposium on Operations Research with Applications in Engineering, Technology, and Management (ISORA'95).

[CTS93] S. Y. Chen, S. N. Talukdar e N. M. Sadeh, *Job Shop Scheduling by Asynchronous Teams of Optimization Agents*, Workshop Notes, IJCAI-93 Workshop on Knowledge-Based Production Planning, Scheduling and Control, Chambery, France, 1993.

[Dav91] L. Davis, *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.

[DL93] S. Dauzere-Peres and J, -B., Lawserre, *A modified shifting bottleneck procedure for job-shop scheduling*, Int. J. Prod. Res., Vol. 31, No. 4, 923-932, 1993.

[Fre82] S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, John Willey, New York, 1982.

[FT63] M. J. Fischer e G. L. Thompson, *Industrial Scheduling*, Prentice Hall, Englewood Cliffs, NJ, 1963.

[GJ79] M. R. Garey e D. S. Johnson, *Computer and Intractability: a guide to the theory of NP-Completeness*, Freeman, San Francisco, 1979.

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

[Glo90] F. Glover, *Tabu Search: A Tutorial*, Interfaces 20: 4 - Jul/Ago, 1990 (pp. 74-94).

[Gra81] S. C. Graves, *A Review of Production Scheduling*, Operations Research, Vol. 29, No 4, Jul/Ago, 1981.

[Gol89] D. E. Goldberg, *Genetic Algorithms in search*, optimization and machine learning, Addison Wesley, Reading, MA, 89.

[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 (UNICAMP), 1994.

[HC90] J. Hutchison e Y,. -L, Chang [HC90], *Optimal nondelay job shop schedules*, Int. J. Pord. Res. 1990, Vol. 28, No 2, 245-257.

[HW90] A. Hertz, D. de Werra, *The Tabu Search Metaheuristic: how we used It*, Annals of Mathematics and Artificial Intelligence, 1 (1990), 111-121.

[KGV83] S. Kirkpatrick, C. D. Gelatt, Jr., M. P. Vecchi, *Optimization by Simulated Annealing*, Science 13, Maio 1983, Vol. 220, No. 4598.

[LA87] P. J. M. V. Laarhoven e E. H. L. Aarts, *Simulated Annealing Theory and Applications*, D. Reidel Publishing Company, 1987.

[LAL92] P. J. M. V. Laarhoven, E. H. L. Aarts e J. K. Lenstra, *Job Shop Scheduling by Simulated Annealing*, Operations Research Society of America, 1992.

[Las92] J. -B., Lawsserre, *An Integrated Model for Job-Shop Planning and Scheduling*, Management Science, Vol. 38, No. 8, Ago. 1992.

[Law84] S. Lawrence, *Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques*, 1984.

[LL93] S. Labide e W. Lejouad, *De L'Intelligence Artificielle Distribuée aux Systemes Multi-Agents*, Rapport de Recherche No. 2004 - Ago. 1993, INRIA.

[LY92] C. J. Liao, C. T. You, *An Improved Formulation for the Job-Shop Scheduling Problem*, J. Opl. Res. Soc., Vol. 43, No. 11, 1992.

[ML93] B. L. Maccarthy e J. Liu, *Addressing the gap in scheduling research: a review of optimization and heuristic methods in production scheduling*, Int. Journal of Prod. Reserach, 1993.

[Nas93] M. A. Nascimento, *Giffler and Thompson's Algorithm for Job Shop Scheduling is Still Good for Flexible Manufacturing Systems*, J. Opl. Res. Soc., Vol. 44, No. 5, pp. 521-524, 1993.

[Nui95] W. Nuijten, *Personal Comunication*, Abr. 1995.

[NW88] G. L. Nemhauser e L. A. Wolsey, *Integer and Combinatorial Optimization*, Jonh Willey, Chichester,UK,1988.

[OH85] P. J. O'Gradye e G. Harrison, *A general search sequencing rule for job shop sequencing*, Int. J. Prod. Res., Vol. 17, 1979.

[Pei95] H. P. Peixoto, *Metodologia de Especificação de Times Assíncronos para Problemas de Otimização Combinatória*, Dissertação de Mestrado, Departamento de Ciência da Computação, Universidade Estadual dc Campinas, UNICAMP, Mar. 1995.

[PI77] S. S. Panwalker e W. Iskander. *A Survey of Scheduling Rules*. Operations Research, Vol. 25, No. 1, 1977.

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

[PS94] H. P. Peixoto e P. S. de Souza, *Uma metodologia de Especificação de Times Assíncronos*, Anais do XXVI Simpósio Brasileiro de Pesquisa Operacional, Florianópolis - SC, 1994.

[SK79] A. S. Spachis e J. R. King, *Job-shop scheduling heuristics with local neighborhood search*, Int. J. Prod. Res., 1979, Vol. 17, No. 6, 507-526.

[SL93] D. Sun e L. Lin, *A dynamic job shop schaduling framework: a backward approach*, Int. J. Prod. Res., 1994, Vol. 32, No. 4, 967-985.

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

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

[ST93] P. S. Souza e S. N. Talukdar, *Asynchronous Organizations for Multi-Algorithm Problemas*, ACM Symposium on Applied Computing, Indianápolis, Feb. 1993.

[SWV92] R. H. Storer, S. D. Wu e R. Vaccari, *New Search Spaces for Sequencing Problems with Applications to Job Shop Scheduling*, Management Science, Vol. 38, 1992.

[Tai93] S. N. Taillard, *Benchmarks for basic scheduling problems*, European Journal of Operational Research, 1993.

[Tal93] S. N. Talukdar, *Asynchronous Teams*, Fourth Symposium on Expert Systems Application to Power Systems, Melbourn, Austrália, 1993.

[TS90] S. N. Talukdar e P. S. Souza, *Asynchronous Teams*, Second SIAM Conf. on Linear Algebra: Signals, Systems and Control, San Francisco, CA, Nov. 1990.

[TS92] S. N. Talukdar e P. S. Souza, *Scale Efficient Organizations*, IEE Int. Conf on Systems, Man and Cybernetics, Chicago, IL, 1992.

[TT93] S. Tzafestas e A. Triantafyllakis, *Deterministic Scheduling in Computing and Manufacturing Systems: a survey of models and algorithms*, Mathematics and Computers in Simulation 35 (1993) 397- 434.

[WC92] L. M. Wein e P. B. Chevalier, *A Broader View of the Job-Shop Scheduling Problem*, Management Science, Vol. 38, No. 7, Jul. 1992.

[WH89] D. de Werra e A. Hertz, *Tabu Search Techniques: A Tutorial and an Application to Neural Networks*, OR Spektrum (1989) 11:131-141.

[WR90] K. P. White, Jr e R. V. Rogers, *Job-Shop Scheduling: limits of the binary disjunctive formulation*, Int. J. Prod., 1990.