Uma Estratégia de Duas Fases

para o Problema de Seqüenciamento

em Células Flexíveis de Manufatura

Mário Antonio do Nascimento

Este exemplar corresponde à redação final da tese defendida por Mário Antonio do Nascimento e aprovada pela Comissão Julgadora em 23 / 11 / 1990.

Vinícius A. Armentano
Orientador

Orientador: Prof. Dr. Vinícius A. Armentano
DENSIS - FEE - UNICAMP

Tese apresentada à Faculdade de Engenharia Elétrica da Universidade Estadual de Campinas como parte dos requisitos necessários para obtenção do título de "Mestre em Engenharia Elétrica - Modalidade Automação".

Agradeço sinceramente:

Aos meus pais pelo simples motivo de terem tornado tudo isto possível e, que não têm interesse pelo texto em si, mas é como se tivessem;

À Anna, pelo amor, carinho, compreensão e paciência, e pela ajuda, sem a qual este texto não existiria;

Ao Vinícius, pela orientação e convivência;

Aos amigos e professores (também amigos) de graduação (IMECC, 1984) e pós-graduação (FEE, 1988), que estiveram, estão e/ou estarão comigo;

À EMBRAPA, em especial ao NTIA, por ter possibilitado a realização deste trabalho;

Ao Fernando, pela amizade e pelo primeiro "a" deste texto (que ficou sendo de agradecimento);

Ao Mitur, por auxiliar com as figuras;

À CAPES e ao CNPq, pelo auxílio durante o período em que fui "apenas" estudante;

Ao CTI (apesar dos pesares), pelo uso dos recursos computacionais;

À todos aqueles que não cito nominalmente, para não correr o risco de, involutáriamente, ser injusto, mas que estiveram de uma maneira ou outra ligados a mim e a este trabalho e

Aos milhões de brasileiros (e brasileiras), que sem saber financiaram e apoiaram este trabalho;

Dedico este trabalho à meus pais, meus irmãos e a Anna, como uma minúscula retribuição a tudo que eles têm sido e feito.

Resumo

Neste trabalho abordamos o problema de seqüenciamento ("scheduling") de peças em um célula flexível de manufatura. Propomos, como contribuição principal, um algoritmo heurístico de duas fases:

- Decompor o problema de N peças e M máquinas em N subproblemas de 1 peça e M máquinas; resolver cada um destes subproblemas otimizando algum critério.

- Combinar as soluções obtidas de todos o subproblemas, de modo a obter uma solução fatível para o problema original e que contemple a otimização de algum critério.

Propomos ainda um algoritmo exato do tipo "Branch-and-Bound" que servirá de referência para avaliar as soluções obtidas pela estratégia de duas fases. Um modelo de programação matemática também é apresentado.

Abstract

In this thesis we treat the problem of scheduling jobs in a flexible manufacturing cell. We propose, as the main contribution, a two phase heuristic algorithm:

- To decompose the N jobs and M machines problem into N sub-problems of 1 job and M machines; to solve each of them minizing some criterion.

- To mix the solution obtained from the sub-problems into a single feasible solution, in which some optimization criterion is considered.

Furthermore we propose an exact Branch-and-Bound algorithm which will provide solutions to be compared with those obtained by the two phase approach. A mathematical programming model is also presented.

CONTEÚDO

Capítulo 1

Introdução

1.1. Evolução dos Sistemas de Manufatura

Nas duas últimas décadas a indústria de manufatura tem passado por transformações muito grandes. Transformações estas ligadas a exigências de consumo, mais precisamente à necessidade de diversificação dos produtos a serem consumidos. Isto afeta, de um modo ou de outro, a duração da vida útil dos produtos. Com a diminuição dessa vida útil, e pressionada pelo mercado, a indústria se encontrou numa situação inédita e difícil. Nesse contexto, novas técnicas tiveram que ser criadas para adaptar as indústrias às novas necessidades. Vejamos a seguir um resumo da evolução da situação apresentada aqui (Agostinho, 1989).

- Situação Anterior:

  - Produção em série, grandes lotes;
  - Pequena necessidade de diversificação;

- Nova situação:

  - Diminuição dos lotes de produção;
  - Aumento da necessidade de diversificação;

- Diagnóstico:

  - Diminuir estoques
  - Aumentar a diversificação, mantendo a produtividade;
  - Buscar técnicas que possibilitem não apenas tornar os sistemas de manufatura flexíveis mas também aproveitar esta flexibilidade.

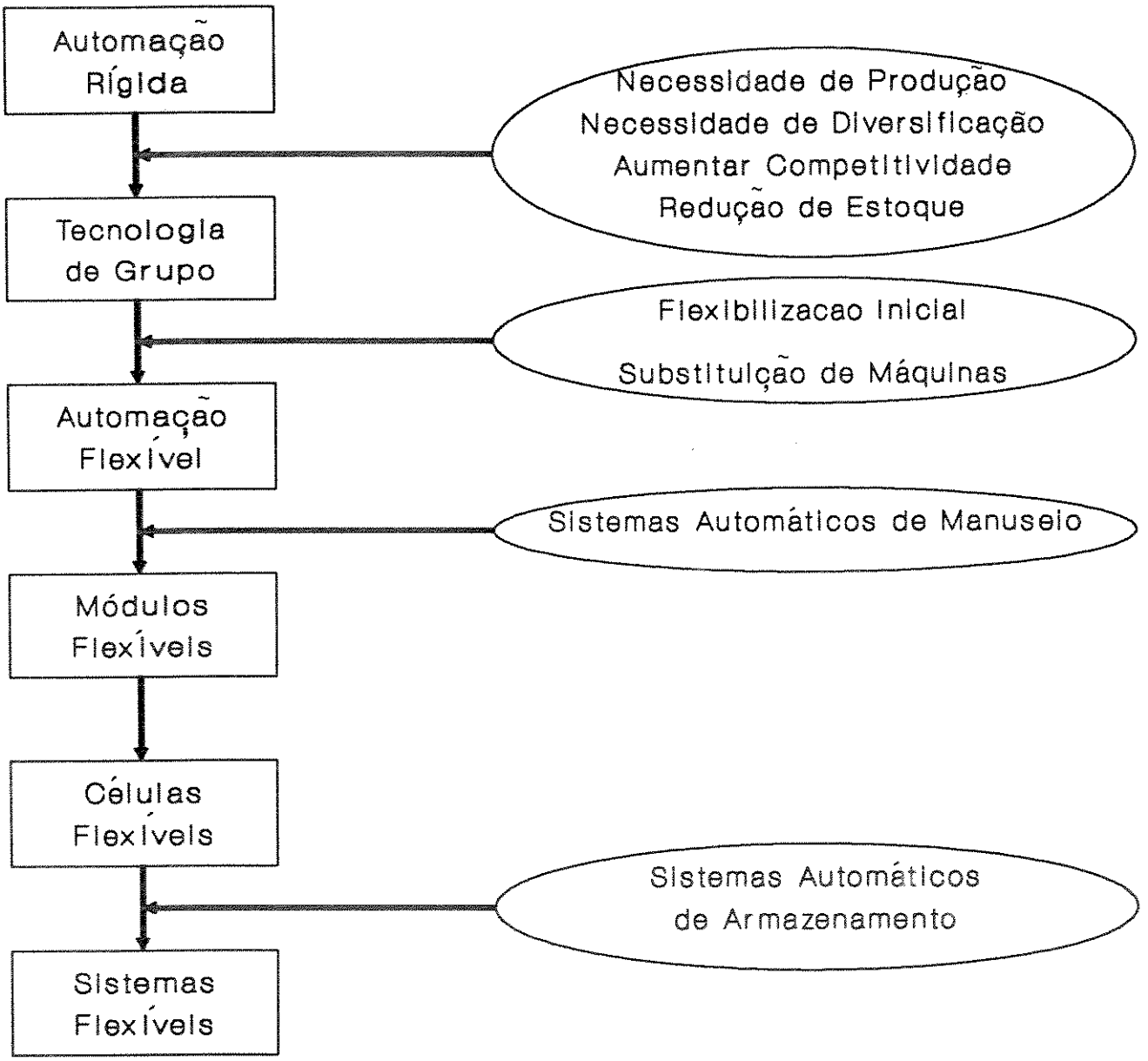

Automação Rígida
Necessidade de Produção
Necessidade de Diversificação
Aumentar Competitividade
Redução de Estoque
Tecnologia de Grupo
Flexibilizacao Inicial
Substitulção de Máquinas
Automação Flexível
Sistemas Automáticos de Manuseio
Módulos Flexíveis
Células Flexíveis
Sistemas Automáticos de Armazenamento
Sistemas Flexíveis

FIGURA 1.1

Muitos estudos foram feitos para que se alcançasse o objetivo colocado acima e os sistemas de manufatura evoluíram (talvez seja melhor falar que estão evoluindo) de acordo com a figura 1.1, onde podemos definir:

Automação Rígida: A seqüencia das operações do roteiro de fabricação é definida pela configuração do equipamento;

Tecnologia de Grupo: Técnicas que consiste em agrupar máquinas capazes de realizar operações semelhantes em peças pertencentes a uma mesma família (Sério, 1990);

Automação Flexível: O equipamento é projetado para que não seja o fator determinante no ambiente da manufatura, isto é, uma mesma máquina pode assumir o papel de várias, guardadas algumas restrições;

Módulos Flexíveis: Uma máquina de Controle Numérico, carga e descarga por robô programavel; troca automática de ferramentas; controle computadorizado local, por módulo;

Células Flexíveis: Dois ou mais Módulos flexíveis; interligados por carros guiados ou robôs; controle local por célula;

Sistemas Flexíveis: Duas ou mais Células Flexíveis; interligadas por carros guiados ou robôs; controle local porém interligado, a nível global, com o resto da produção; armazenamento automático; controle de tráfego.

A medida que o nível de flexibilidade dos sistemas de manufatura aumentava, alguns fatores além daqueles relativos a máquinas passaram a se tornar limitantes. Assim, a diversidade dos itens produzidos proporcionou o desenvolvimento de técnicas, como "Just-in-Time" e "Kan-Ban" (Schonberger, 1984), orientadas para a produção em pequenos lotes. Porém estas técnicas isoladamente não são suficientes para que um ambiente de manufatura seja eficiente, torna-se necessário um esforço em estágios mais básicos da manufatura, desde o planejamento da produção até a execução

dos planos a nível de chão de fábrica.

1.2. Organização do texto

No caso desta tese exploramos uma pequena parte desse problema de manufatura. Vamos nos ater especificamente ao ambiente de uma Célula Flexível de Manufatura (**CFM**), que tem a característica de poder mudar rapidamente de configuração adaptando-se ao produto produzido. Abordamos o Problema de Seqüenciamento[1] em uma Célula Flexível de Manufatura (**PSCFM**), isto é, em que ordem, em qual máquina e quando processar as operacões devidas numa série de peças dadas.

Apresentamos aqui uma estratégia para a determinação de um seqüenciamento para peças que devem ser processadas numa **CFM**, que se baseia na união de sub-planos ótimos encontrados através de buscas guiadas e solucionando conflitos com o uso de regras heurísticas, resultando num planejamento sub-ótimo.

Este trabalho está dividido da seguinte maneira: No capítulo 2 apresentamos um resumo sobre o problema de seqüenciamento em "job-shop" e incluimos o modelo de Sistema Flexível de Manufatura (**SFM**) de maneira a poder discutir as estratégias para o **PSCFM** utilizadas por outros pesquisadores. Continuando, no capítulo 3, apresentamos algoritmos utilizados em Inteligência Artificial que serão necessários para o entendimento da estratégia de resolução do **PSCFM** que será apresentada no capítulo 4, onde detalhamos a estratégia proposta para a resolução do **PSCFM**. A seguir, no capítulo 5, discute-se o modelamento do **PSCFM** como um problema de programação inteira mista não-linear, assim como a implementação de um método de solução do tipo "Branch-and-Bound" que garante resultados ótimos segundo um certo critério. Finalmente, no capítulo 6, apresentamos conclusões a respeito da estratégia usada e indicamos direções para pesquisas futuras na área. Apresentamos ainda o apêndice A onde fazemos uma breve introdução aos algoritmos do tipo "Branch-and-Bound" e o apêndice B onde introduzimos algumas idéias relativas a Sistemas Especialistas.

---

[1] Usamos o termo seqüenciamento, assim como poderíamos ter usado programação, no lugar do termo inglês "scheduling", usado correntemente na literatura.

Capítulo 2

Problema de Seqüenciamento em Manufatura

2.1. Introdução

O problema de seqüenciamento num ambiente de manufatura pode ser colocado como: "uma coleção de ações a serem realizadas com algumas restrições tecnológicas e temporais" (LeCocq, 1988). Vale ressaltar que este conceito pode ser estendido para outros ambientes que não de manufatura. Podemos citar como exemplo: o planejamento da construção de usinas hidroelétricas (Ribeiro, 1980), a programação de reparos em aeronaves (Grant, 1986) e o escalonamento de médicos em hospitais (LaFortune & outros, 1981) entre muitos outros.

Existem na literatura muitos artigos que apresentam revisões ("reviews") de trabalhos na área. Como uma revisão geral do tópico de seqüenciamento podemos citar os trabalhos de Graves (1981) e Lawler & outros (1982) e, para um nível de detalhamento maior, porém mais básico, indicamos os trabalhos de Baker (1974) e French (1982). A seguir apresentaremos algumas notações e definições que usaremos para fazer uma revisão de alguns problemas de seqüenciamento. Note que a notação será dirigida para o ambiente de manufatura, sem perda de generalidade.

Suponha a existência de um conjunto $\alpha = \{ j_1, j_2,..., j_N \}$ de N peças ("j" deriva do inglês "job") cada uma composta composta por um conjunto $\gamma = \{ o_1, o_2,..., o_O \}$ de operações a serem realizadas e um ambiente com M máquinas definidas pelo conjunto $\beta = \{ m_1, m_2,..., m_M \}$. A hipótese de que cada peça é composta de exatamente O operações é apenas simplificadora. No caso geral cada peça poderia ser composta por qualquer número finito de operações. Em qualquer caso, porém, a seqüência das operações define a peça.

É interessante colocar, a partir desse momento, uma definição mais formal de seqüenciamento. Resolver o problema de seqüenciamento significa: determinar em que momento cada peça terá cada uma das operações que a compõe executada (na ordem

pré-determinada) e em que máquina; ou de outra maneira, em que ordem cada peça percorrerá as máquinas, com o objetivo de otimizar algum critério mensurável quantitavamente. Porém torna-se necessária agora uma hipótese básica, seja qual for o modelo estudado:

H.B.0) Cada peça pode ser processada por apenas uma máquina de cada vez e cada máquina pode processar apenas uma peça de cada vez.

Cabe agora enunciar as hipóteses básicas classicamente usadas para podermos trabalhar com os modelos a serem estudados neste capítulo (Baker, 1974).

H.B.1) O conjunto α de peças está disponível a partir do instante inicial de trabalho;

H.B.2) Os tempos de configuração ("set-up") das máquinas para processamento das operações não dependem da seqüência das mesmas e estão incluidos no tempo de processamento;

H.B.3) As peças, num sentido completo, isto é, juntamente com as operações necessárias são conhecidas "a priori";

H.B.4) Uma máquina está sempre disponível e nunca fica ociosa se existe trabalho a ser executado pela mesma;

H.B.5) Nenhuma operação uma vez iniciada pode ser interrompida, diremos que são "não-interruptíveis";

H.B.6) Cada máquina é capaz de realizar uma e apenas uma operação.

Vejamos agora algumas informações sobre estas entidades (peças, operações e máquinas) que podem ser, e na maioria das vezes são, fornecidas a priori, além de medidas que servirão para avaliar o seqüenciamento.

Considere:

t_{ij}: tempo de processamento da operação i na máquina j , $t_{ij} \in [0,+\infty)$;

r_j: data a partir da qual a peça j está disponível; se considerarmos **(H.B.1)** temos $r_j = 0$, em geral $r_j \geq 0$ ("ready-time");

d_j: data limite para que a peça j esteja acabada, isto é, todas as suas operações já tenham sido realizadas ("due-date").

De posse das informações acima (podem haver outras) podemos coletar uma série de informações sobre um dado seqüenciamento. Por exemplo:

C_j: data de finalização da peça j ("completion time");

L_j: folga da peça, $L_j = C_j - d_j$ ("lateness") e

T_j: atraso da peça, $T_j = \max\{0, L_j\}$ ("tardiness").

Se o interessante é uma medida do efeito do seqüenciamento sobre todas as peças podemos definir, entre outras medidas:

$\bar{T}$: atraso médio, $\bar{T} = \frac{1}{N}\sum_{i=1}^{N} T_i$;

M: tempo total de processamento ou "makespan" como mais usado na literatura, $M = \max_i \{C_i\}$.

Para o modelo que apresentaremos a seguir existem várias metodologias e algoritmos para atacar o problema com relação a várias medidas. Concentrar-nos-emos, porém, num critério específico bastante intuitivo num ambiente de manufatura, o de minimizar o "makespan", que será denotado por M^*:

$$M^* = minimizar\,\{\max_j \{C_j\}\}$$

Informalmente obter M^* significa buscar a seqüência que permite finalizar todas as peças de maneira mais rápida.

2.2. O modelo de "job-shop"

No modelo de "job-shop" cada peça é composta de várias operações ordenadas, e cada máquina é capaz de realizar apenas uma operação. Em geral faz-se a hipótese de que $M = O$. O fluxo das peças entre as máquinas não é unidirecional, isto é, a seqüência de máquinas a serem percorridas pode ser distinta para cada peça. No modelo de "job-shop" torna-se necessário a tripla (i,j,k) para denotar a realização da i*-ésima* operação da peça j na máquina k.

Trataremos aqui do critério de desempenho "makespan" que tem sido bastante investigado em relação a outros, usando como base o trabalho de Baker (1974).

O "job-shop" é composto de várias peças e um seqüenciamento factível é a união dos roteamentos de cada peça sem que nenhuma das hipóteses básicas seja violada. É claro que existe apenas um número finito, porém provavelmente muito grande, de seqüenciamentos possíveis. Se considerarmos a inserção de tempos ociosos, isto é, períodos de tempos onde as máquinas ficam paradas (repare que relaxamos desta forma (**H.B.4**)), passamos a ter um número infinito de seqüenciamentos possíveis, e mais, nunca melhores que aqueles que os originaram. Logo os seqüenciamentos com inserção de tempo ocioso são dominados por aqueles sem inserção, e não devemos considerá-los.

As propriedades de dominância são muito importantes na medida em que podem contribuir para diminuir o esforço de busca por uma solução ótima. Essa redução se dá da seguinte maneira: verifica-se que para uma seqüência ser ótima ela tem que ter, obrigatoriamente, uma certa característica; ora, torna-se então necessário buscar por soluções que tenham aquela característica, considerar outras que não as tenham é inútil já que podem levar a resultados não ótimos.

O resultado anterior pode ajudar na seguinte conclusão: Se num dado seqüenciamento podemos antecipar o início de uma operação sem que a ordem das operações em qualquer máquina seja alterada então o período de tempo ganho era supérfluo.

O ajuste que mencionamos acima é chamado de "deslocamento à esquerda" e o conjunto de seqüenciamentos onde "deslocamentos à esquerda" não são possíveis é o conjunto de seqüenciamentos semi-ativos. Num modelo de "job-shop" clássico o número de seqüenciamentos semi-ativos é da ordem de $O((n!)^m)$. Podemos agora enunciar:

P.1) O conjunto de seqüenciamentos semi-ativos domina o conjunto de todos os seqüenciamentos possíveis (Baker, 1974).

Podemos ainda refinar esta propriedade de uma maneira até que bastante intuitiva. Suponha que entre duas operações i e j consecutivas em uma máquina haja, por algum motivo, um período de tempo ocioso onde um deslocamento à esquerda não é possível, isto é, um seqüenciamento semi-ativo. Porém se há uma operação k posterior a operação j que pode ser transladada de modo a passar a ser anterior a j sem atrasar nenhuma daquelas que a sucedem então o valor da solução pode melhorar e com certeza não será pior. Denominaremos esta translação sem causar atrasos em nenhuma outra operação de "translado à esquerda" e os seqüenciamentos onde nenhum translado à esquerda pode ser executado de seqüenciamentos ativos. Assim concluimos:

P.2) O conjunto de seqüenciamentos ativos é um subconjunto do conjunto de seqüenciamento semi-ativos, e claramente o domina (Baker, 1974).

A título de exemplo suponha o seguinte seqüenciamento semi-ativo[1]:

[1] Cada quintúpla (i,j,k,e,s) denota: operação i, peça j, máquina k, inicio do processamento no tempo e e fim no tempo s.

Máquina 1:

|(1,2,1,0,2)|.....|(2,1,1,3,5)|

Máquina 2:

| (1,1,2,0,3) |.....|(3,1,2,5,7)|(2,2,2,7,9)|

e o correspondente seqüenciamento ativo:

Máquina 1:

|(1,2,1,0,2)|.....|(2,1,1,3,5)|

Máquina 2:

| (1,1,2,0,3) |(2,2,2,3,5)|(3,1,2,5,7)|

Alguns outros estudos têm sido feitos com respeito a um conjunto de seqüenciamentos denominados sem-demora (do inglês "non-delay"), porém este não constitui um conjunto dominante e não o consideramos aqui.

Como vimos, as propriedades de dominância são extremamente importantes e, nesse contexto muito esforço foi feito no sentido de se conseguir um algoritmo que permita a geração de seqüenciamentos onde as propriedades de dominância fossem observadas e que ao mesmo tempo não gerasse seqüenciamentos que não pertencessem ao conjunto dominante. Dessa forma o esforço de busca seria otimizado no sentido em que o número de seqüenciamentos investigados seria no máximo igual a cardinalidade do conjunto dominante. Como o conjunto de seqüenciamentos ativos constitui um conjunto dominante, é razoável que se pense em algoritmos que construa apenas seqüenciamentos pertinentes a este conjunto. De fato, Giffler & Thompson (Baker, 1974, p:189) desenvolveram um algoritmo com este propósito. O algoritmo baseia-se na construção de seqüenciamentos parciais e de conjuntos que contém as operações já seqüenciadas e aquelas ainda por serem colocadas no seqüenciamento de cada peça. Na sua forma original o algoritmo parte da seqüência vazia e constrói uma árvore onde cada nó descendente contém uma operação seqüenciada a mais que o nó anterior, tornando possível construir todos os seqüenciamentos ativos possíveis.

Podemos perceber a utilidade do algoritmo comentado acima, porém na maioria dos casos o porte do modelo implica no fato de que o número total de seqüenciamentos ativos é grande o suficiente para que uma enumeração total seja feita

na busca de uma solução ótima.

Assim a primeira idéia que surge é a de se aproveitar a estrutura de árvore do algoritmo de Giffler & Thompson usando uma estratégia do tipo "Branch-and-Bound". Brook & White (Baker, 1974, p:193) desenvolveram um algoritmo baseado na idéia acima, onde em cada nó da árvore calculavam-se dois limitantes inferiores para a solução:

L.I.1) Relaxando a hipótese (**H.B.0**) no sentido de que as máquinas poderiam processar mais de uma peça simultâneamente e

L.I.2) Relaxando a hipótese de que as operações devem ser executadas numa ordem pré-determinada.

É claro que as duas soluções que seriam encontradas com as relaxações acima seriam muito provavelmente infactíveis mas de qualquer maneira fornecem limites inferiores úteis no processo de enumeração implícita. Vale ressaltar que em cada nó podemos obter os limites (**L.I.1**) e (**L.I.2**). Escolhemos como limite inferior daquele nó o maior deles.

Atualmente o melhor limitante inferior encontrado na literatura é aquele determinado por Lenstra & outros (1977). Para determinar este limite inferior os autores modelam o problema de "job-shop" como um problema de fluxo em grafos com arcos disjuntos que estabelecem as condições de conflito a serem resolvidas.

Outros enfoques do tipo: programação inteira-mista (que levam a modelos do porte apreciável mesmo em casos onde o número de máquinas e peças é pequeno), "Branch-and-Bound" associado a regras de despacho e, é claro, procedimentos heurísticos têm sido bastante discutidos na literatura relativa ao problema.

2.3. Modelos de sistemas flexíveis de manufatura

Como já dissemos anteriormente um Sistema Flexível de Manufatura (**SFM**) é composto de várias Células Flexíveis de Manufatura (**CFM**), ou seja, são várias

máquinas de controle numérico com carga e descarga de peças feitas por robôs, troca automática de ferramentas e interligadas por um sistema automatizado para transporte de peças entre as máquinas, além de um sistema automático de armazenamento. Num caso ideal, todas as máquinas seriam capazes de realizar todas as operações em todas as peças. Porém, uma desvantagem que pode ser vista neste caso seria a taxa, provavelmente alta, de troca de ferramentas em cada máquina, implicando num período razoável, gasto apenas com troca de ferramentas (também chamado tempo de configuração). Uma idéia para se evitar isso seria a de usar estratégias de tecnologia de grupo onde algumas máquinas seriam dedicadas ao processamento a uma família de peças. Isto posto, podemos tecer comentários comparando este ambiente aos anteriormente estudados. Das hipóteses básicas, anteriormente colocadas, (**H.B.0**) e (**H.B.5**) são as únicas que continuarão sendo observadas, embora possa haver o caso onde elas também pudessem ser relaxadas. Relaxando (**H.B.1**) e (**H.B.3**) permite-se um ambiente extremamente dinâmico, (**H.B.2**) deixa de ser válida porque num caso geral o tempo de configuração pode variar muito. Sobre (**H.B.4**) podemos agora dizer que a inserção de tempo ocioso pode agora ser benéfico para o sequenciamento, já que há possibilidade da escolha de uma máquina mais eficiente. O motivo pelo qual (**H.B.6**) não é mais considerada dispensa comentários.

Com a relaxação de tantas hipóteses, é facil ver que um ambiente deste tipo é mais livre que o ambiente de "job-shop" (e este já era bastante complexo dado o seu grau de "liberdade"). Assim este problema tornou-se bastante interessante para a aplicação de técnicas de Inteligência Artificial (**IA**). A idéia de se usar as técnicas de **IA** é a de poder aproveitar o conhecimento dos especialistas da área, adquirida durante muito tempo, de maneira a explorar o problema de maneira mais direta e inteligente.

No campo de **IA**, a técnica mais explorada para a resolução deste problema é a de construção de Sistemas Especialistas (**SE**, apêndice B). Este tipo de enfoque tem, no entanto, uma limitação: como a idéia é a de se explorar inteligentemente o problema num dado ambiente, o **SE** é projetado de maneira a ser acoplado àquele ambiente e dificilmente poderá ser usado em outro.

Antes de apresentarmos alguns estudos que tem sido feitos nesta área, se torna necessário comentar o trabalho de Fox (1983) bastante citado na literatura da área como um dos primeiro a usar técnicas de **IA** para resolver o problema de sequenciamento.

2.3.1. Fox (1983).

O sistema desenvolvido por Fox, denominado **ISIS**, e projetado para o ambiente de "job-shop" teve como objetivo, segundo suas próprias palavras:

> "concentrar-se, a nível de sequenciamento para "job-shop" na representação e utilização de todas as restrições relevantes no processo de sequenciamento, projetar e construir um sistema iterativo para o modelamento e sequenciamento de modelos gerais de "job-shop" e diminuir o elo entre sistemas de sequenciamento que simplesmente guiam um operador humano criando um sistema que pode controlar as operações em tempo real".

O sistema **ISIS** é baseado numa estratégia guiada por restrições, isto é com base em restrições que vão sendo encontradas e/ou criadas, vai se tornando o ambiente cada vez mais "rígido" até determinar um sequenciamento. Para isso é necessário representar:

- conflito de restrições;
- importância das restrições;
- interação entre as restrições;
- obrigação de satisfação de restrições e
- geração de restrições

sem esquecer no entanto, da factibilidade do processo de geração do sequenciamento e da qualidade de solução gerada. Porém, Ow (1986) aponta uma deficiência do **ISIS**: o sistema consegue resolver muito bem os conflitos de procedência de operações por máquina mas não resolve satisfatoriamente os problemas de competição, entre as peças, por um mesmo recurso. Apesar disto, o **ISIS** foi muito importante dado o seu pioneirismo no uso de técnicas de **IA** no problema de sequenciamento.

A seguir apresentamos um resumo dos principais trabalhos na área. Note que todos se referem a **SFM**, porém, o que diferencia um **SFM** de uma **CFM** é a existência de armazenamento automático e nenhum dos trabalhos que apresentaremos contempla este aspecto o que leva a crer que a denominação de **SFM** tem sido atribuida a ambientes flexíveis que não necessariamente se enquadram na classificação de **SFM** dada no capítulo 1. Assim podemos usá-los como referência para o tópico do **PSCFM**.

2.3.2. Subramanyam & Askim (1986)

O principal mérito do trabalho desenvolvido pelos autores é que ele permite um seqüenciamento em tempo real. Esse seqüenciamento é baseado em decisões do tipo:

- Qual a próxima peça a ser processada;
- Qual máquina será designada para uma dada operação;
- Qual a próxima operação a ser executada em cada peça (suponha que não há restrições de precedência) entre outras.

O trabalho apresentado pelos autores concentra-se no entanto no seguinte tipo de decisão: "que peça processar, dentre aquelas na fila para entrarem no sistema".

Para que o **SE** possa decidir algo é preciso alimentá-lo com algumas informações. No caso há três parâmetros: os estados do sistema, da máquina e da peça. Cada um deles pode estar num nível crítico, normal ou abaixo do normal. A partir destes parâmetros, o sistema é "reconhecido" e a partir desse reconhecimento escolhe-se um critério a ser otimizado localmente. Os possíveis critérios investigados são: custo, atraso e "makespan". Após isso, temos posse do "estado" do sistema e do critério a ser otimizado, e a partir de regras absorvidas de um especialista na área, ações são desencadeadas. Repare que a "inteligência" do **SE** vem justamente do conhecimento adquirido anteriormente e da capacidade de aquisição de novos conhecimentos. Estratégias análogas podem ser desenvolvidas para outros tipos de decisão como por exemplo "solucionar qual máquina será usada".

Uma crítica pode ser feita a esse trabalho. É claro que as decisões feitas em tempo real tendem a explorar bem a flexibilidade do sistema, possibilitando uma estratégia de seqüenciamento para ambientes dinâmicos, mas ao se otimizar sempre somente a nível local, nunca observamos o seqüenciamento como um todo, segundo um critério específico, e em muitos casos talvez a otimização global de apenas um critério fosse mais recomendável, econômica ou técnicamente, do que a de vários critérios localmente.

O **SE** construído pelos autores usa regras de produção para representar o conhecimento. Para a estratégia de controle desse conhecimento tanto buscas dirigidas por dados como por objetivo poderiam ser utilizados, embora nós acreditemos que a especialização para o caso de buscas dirigidas por dados fosse mais interessante.

2.3.3. Bruno, Elia & Laface (1986)

Aqui os autores usam dois enfoques bastante distintos, o algoritmico e o simbólico (aquele usado pelos **SEs**) acoplados para a resolução do problema de seqüenciamento. A idéia utilizada pelos autores é a seguinte: para cada lote (os autores não trabalham com o seqüenciamento a nível de peças mas de lotes de peças, é claro que poderíamos recair no caso de seqüenciamento de peças bastando considerar lotes unitários) calcula-se uma medida de prioridade para o mesmo. De posse das prioridades dos lotes, estes podem ser introduzidos a medida que suas pré-condições se tornem reais (cada lote tem uma pré-condição para ser processada, por exemplo, uma máquina estar disponível e uma certa operação já ter sido executada neste lote), de acordo com as suas prioridades. Antes de se seqüenciar efetivamente o lote, o efeito da sua inclusão é avaliado e o lote só será seqüenciado se essa avaliação for positiva.

Podemos agora identificar na idéia acima estratégias de processamento simbólico, na busca e identificação de lotes cujas pré-condições são satisfeitas e processamento algoritmico, na avaliação do efeito da introdução dos lotes no sistema, através de algoritmos de redes de filas fechadas. O **SE** implementado pelos autores explora bem esse acoplamento através do uso de estrutura de dados separadas. Em termos de tempo de computação o sistema implementado, quando comparado com um anterior totalmente algoritmico, foi ligeiramente inferior o que não pode ser considerado como demérito do trabalho, já que a área de programação em **IA** ainda é restrita.

2.3.4. LeCocq & Guiot (1988)

Neste artigo os autores apresentam um **SE** voltado para a resolução dos problemas do planejamento e seqüenciamento em um **SMF** (no sentido relaxado, isto é, um conjunto de **CMF** interligadas sem armazenamento automático). O sistema proposto se concentra em dois pontos: absorver e aproveitar o máximo da flexibilidade do ambiente e levar em conta eventuais perturbações (tal como quebra de máquinas) no sistema. Assim o **SE** pode realizar o planejamento e seqüenciamento dinâmicamente para um horizonte finito. A principal característica do **SE** construído é a sua capacidade de reduzir o problema original, normalmente de médio prazo [2], em subproblemas para serem resolvidos em curto prazo. Desse modo é requerido menos esforço computacional e menos dados para processamento.

O projeto do **SE** foi feito de modo que o mesmo possa utilizar seu conhecimento para resolver possíveis conflitos e efetuar o balanceamento de tarefas (distribuição equitativa de trabalho para as máquinas) evitando o uso de algoritmos de otimização, computacionalmente muito "pesados". Uma vez feito o seqüenciamento o próprio **SE**, fazendo uso de diagnósticos obtidos em tempo-real, determina quando realizar as operações planejadas e quando fazer o replanejamento (e conseqüente re-seqüenciamento) devido a alguma perturbação no sistema.

O **SE** toma como entrada um programa de produção e o transforma em objetivos de curto prazo, para isso usa um enfoque baseado em três pontos:

1) Faz-se um planejamento "grosseiro" da produção respeitando restrições como data prometida e balanceamento da produção;

2) Define-se os limites de uso dos recursos garantindo que os mesmos não são sobrecarregados e resolve os conflitos que podem surgir dada a capacidade de cada recurso e

[2] considere curto, médio e longo prazo como sendo respectivamente horas, dias e semanas, por exemplo.

3) Por último atribui-se a cada tarefa um recurso (aqui podemos considerar como recursos máquinas, ferramentas e dispositivos de transporte), determinando o seqüenciamento propriamente dito.

No trabalho apresentado pode-se notar uma grande preocupação quanto a representação do conhecimento e entidades relacionadas a este. Os autores usam um **SE** baseado em Regras de Produção. Esta representação se mostrou conveniente no caso de monitoramento e diagnose do sistema. Num nível mais baixo o sistema se concentra no uso de "roteamentos hierárquicos", composto de "agregação de dados" e uso dinâmico de recursos. A "agregação de dados" é a tentativa de se formar macro-operações composta de operações singulares que usam o mesmo recurso. Essa agregação é feita recursivamente até obtermos blocos tão compactos quanto possível. Dessa maneira conseguimos obter um dado importante sobre a dinâmica dos recursos, ou seja, recursos com um grande número de operações alocadas tem uma dinâmica bem maior que aqueles com poucas operações e por conseguinte não podem ser planejadas a longo prazo ao contrário daqueles recursos, por assim dizer, pouco exigidos.

Assim podemos verificar que a flexibilidade dada pelo **SE** aliado ao monitoramento em tempo real confere ao sistema a flexibilidade desejada no caso dos **SMF**s.

2.3.5. Ben-Arieh & Moodie (1987)

A intenção dos autores era a de construir um sistema computacional que utilize uma base de conhecimento para a determinação de um seqüenciamento factível para um **SFM**. Não há nenhuma garantia de que o seqüenciamento encontrado seja ótimo.

Este trabalho, como os anteriores, tem como forte pré-requisito a possibilidade de se efetuar o monitoramento em tempo real do sistema de manufatura, esse monitoramento seria feito usando como base uma rede local (**LAN**, do inglês "Local Area Network") que permite o conhecimento de parâmetros dinâmicos do sistema tais como: quebra de máquinas, filas em recursos e disponibilidade de estoque. Consideramos parâmetros fixos como sendo exemplo: seqüência de operações por peça, data de entrega (prometida), tempos de processamento, etc. Além do mais, os parâmetros dinâmicos, podem mudar de maneira tal que um seqüenciamento ótimo feito "a priori" se torne muito ruim. Suponha por exemplo um dado seqüenciamento onde se faz uso

intensivo de uma certa máquina, e que num dado instante a máquina quebre, o seqüenciamento determinado a priori se torna infactível. No caso de termos um monitoramento do sistema um re-seqüenciamento poderia ser feito sem maiores prejuízos.

Os autores defendem a tese de que um sistema computacional de seqüenciamento interagindo com um especialista humano dotado de uma política simples porém "conhecedor" do estado do sistema pode ser mais eficiente que um seqüenciador "off-line" ("a priori") dotado de uma política complexa, porém, fixa.

No trabalho apresentado, os autores apresentam um **SE** de auxílio para um seqüenciamento humano. A idéia é a de que um **SE** pode tomar decisões de seqüenciamento baseado em medidas qualitativas e ainda avaliar o efeito dessas decisões através de uma ferramenta de simulação. Esse sistema, denominado **KBRS** (do inglês "Knowledge Based Routing System"), tem dois níveis de conhecimento:

- o primeiro decide que decisões tomar considerando todo o conhecimento requerido e

- o nível de meta-conhecimento, isto é, identifica, com base no estado do sistema, que regras são selecionáveis para uma decisão futura,

e três bases de conhecimento:

- Uma base para dados dinâmicos, para monitoramento em tempo real;

- Uma base para dados estáticos, relativa aos parâmetros fixos do sistema e

- Uma base de regras de comportamento do sistema.

Todas essas bases são interligadas por um sistema de consulta que por sua vez interage com os componentes responsáveis pelas decisões do sistema.

O processo de seqüenciamento é composto de três algoritmos. Considere que o objetivo do **KBRS** é maximizar a produção de peças montadas (isto é, não são peças simples, uma peça montada é normalmente composta por duas ou mais peças simples):

1) Identificar, dada uma "árvore de montagem", todas as datas esperadas para cada componente na estação de montagem;

2) Um algoritmo que busca as datas prometidas para cada componente e as possíveis rotas, calculando assim uma estimativa do tempo de espera em fila para a peça candidata e

3) Um algoritmo para a introdução de novos componentes que só ocorre se não houver nenhum outro similar no sistema. A intenção deste algoritmo seria a de diversificar a produção.

A seguir apresentamos um resumo da experiência computacional realizada pelos autores.

De maneira a avaliar a desempenho do **KBRS** foram utilizadas quatro medidas:

1) Utilização de máquina;

2) Comprimento médio da fila;

3) Taxa de produção de cada máquina e

4) Taxa de produção de cada peça e do produto final montado.

Alguns parâmetros foram ajustados para se obter uma "análise de sensibilidade" do sistema testado, esses parâmetros foram:

1) Taxa de quebra de máquinas;

2) Horizonte de simulação e

3) Estrutura da árvore de montagem.

Para se comparar o seqüenciamento fornecido pelo **KBRS** foram usadas 7 outras estratégias, baseadas numa heurística gulosa, algumas combinações desta e escolha aleatória. O melhor seqüenciamento foi aquele encontrado pelo **KBRS** e o segundo o encontrado pelo humano.

O **KBRS** se comportou muito bem em relação as máquinas quebradas (verificando se espera conserto ou muda de máquina) e mantendo a média de utilização das máquinas. A sensitividade do horizonte de simulação do **KBRS** é afetada pela taxa de quebras. No caso geral um horizonte médio de simulação fornece bons resultados.

A principal diferença entre as estratégias usadas pelo **KBRS** e o humano reside no fato de que o humano tende a sacrificar o estado futuro do sistema para obter melhores resultados a curto prazo, gerando assim uma maior fila de espera nas estações de montagem, enquanto que o **KBRS** tem um menor tempo médio de espera.

2.3.6. Shaw & Whinston (1989)

O enfoque usado pelos autores é apresentado a seguir, usando suas próprias palavras:

> "O sequenciamento de **SFMs**, feito por um Sistema Baseado em Conhecimento (**SBC**), é executado por um algoritmo de planejamento não-linear que decompõe um problema de N peças e M máquinas em N subproblemas, onde cada subproblema é definido como o roteamento de uma única peça. Uma inferência dirigida por objetivos é aplicada para gerar um "plano" para os N subproblemas; a interação primárias entre estes subproblemas é o compartilhamento das M máquinas. O objetivo do seqüenciamento é minimizar o "makespan", evitando conflitos e pode ser visto como o critério do problema de geração de planos em **IA** (Sacerdori, 1977): maximizar o paralelismo mantendo factíveis as interações entre os sub-planos."

O sistema desenvolvido tem 3 componentes básicos:

1) Modelo do ambiente:

Contém uma descrição simbólica do ambiente real. Esse modelo está armazenado na base de dados do **SBC** e uma instância deste é um estado do ambiente.

2) Modelo de ação:

Capaz de descrever os efeitos de ações que levam um estado a outro. Cada ação, ou operador, é composto por uma lista de atributos descritos a seguir:

Pré-condição:	Estado necessário do modelo do ambiente para que o operador seja acionado;
Lista de Adição:	Fatos que serão adicionados ao estado do sistema se o operador for acionado;
Lista de Deleção:	Similar a lista de adição, porém com respeito a fatos atualmente verdadeiros e que o deixarão de ser.
Recurso:	Recurso necessário para o operador
Duração:	Intervalo de tempo durante o qual o operador está sendo executado.

O modelo de ação está armazenado na base de conhecimento do **SBC**.

3) Máquina de inferência:

Dirige o processo de geração de planos, selecionando uma seqüência de operadores para levar um dado estado inicial ao estado objetivo, isto é, completar o seqüenciamento respeitando as restrições de conflito.

Esse sistema poderia usar um método de busca dirigida por dados (retroativa), porém como não há a independência dos subplanos (eles competem no uso dos mesmos recursos) usa-se a idéia de adição de restrições de precedência entre

operadores conflitantes e daí o seqüenciamento final pode ser visto como uma rede parcialmente ordenada de operadores.

Os autores usam os componentes acima descritos para a construção de um algoritmo para o **PSCFM**, descrito a seguir:

Algoritmo de Planejamento Não-Linear

1) Geração de um plano seqüencial e linear para cada peça;

2) Identificações de interações conflitantes entre os operadores planejados e estabelecimento de restrições de precedência para evitar conflitos de seqüenciamento;

3) Uso de uma estratégia de revisão do plano para melhorar o "makespan".

Podemos agora detalhar brevemente cada um destes passos:

Determinações dos subplanos:

A máquina de inferência usa uma busca dirigida por dados para gerar os planos de cada peça. Essa busca é dirigida por uma heurística que avalia o custo de cada solução disponível. Na verdade essa avaliação fornece um limitante inferior do valor da solução ótima para as operações restantes num dado sub-plano. Neste contexto a busca pode ser visualizada como uma busca em um grafo dirigido e os autores utilizam o algoritmo A^*, que descreveremos em detalhe no capítulo 3 desta tese, para determinar os sub-planos que são, com certeza, ótimos segundo o critério de minimização do tempo necessário para o processamento total da peça.

Identificação dos Conflitos:

São discutidas duas estratégias para esta etapa:

1) Tabela de Múltiplos Efeitos: Cada operador tem associado a ele duas listas, uma do conjunto de operadores que produzem suas pré-condições (lista de adicionadores) e outra daqueles que deletam suas pré-condições (lista de inibidores). Esta estratégia, porém, não se mostra vantajosa por ser computacionalmnente "pesada" e "míope", isto é, não consegue usar as informações de flexibilidade do sistema.

2) Decisão sobre recursos: Determina-se nos planos lineares "seções críticas" (uma seção crítica é definida como um conjunto de operadores consecutivos que usam o mesmo recurso) e impõe-se restrições de modo que não haja conflitos no uso de recursos comuns a "seções críticas" distintas. Essa estratégia se mostra de fato mais forte porque podemos usar a flexibilidade do sistema na determinação de como estabelecer a restrição de precedência. A seguir veremos este ponto com mais detalhes.

Revisão de plano:

Dado um conflito entre duas seções críticas busca-se um recurso alternativo ocioso naquele instante que possa diminuir o tempo de espera numa eventual fila de espera para uso daquele recurso-conflito. Se existe tal recurso e se a realocação é melhor que o tempo de espera então faz-se um replanejamento do plano-linear da peça realocada do ponto de realocação para a frente de maneira a encontrar um novo plano linear que, apesar de ser gerado como anteriormente, pode não ser ótimo já que uma parte dele já havia sido fixa.

No artigo apresentado pelos autores são realizados vários experimentos usando diferentes funções de avaliação para a heurística que guia a busca na determinação dos planos lineares. Uma observação que cabe ser feita em relação ao trabalho apresentado por Shaw & Whinston é: não há um estudo de quão boa é a solução em termos de "otimalidade" e também não se apresentam os tempos de computação necessários para a resolução do problema.

No final do artigo os autores delineiam algumas idéias interessantes sobre o processo de aprendizagem de máquinas onde informações adquiridas durante a realização de seqüenciamentos anteriores poderiam ser usadas como meta-heurísticas na

determinação de seqüenciamentos "on-line" em tempo real, fazendo uso de toda a flexibilidade e dinamismo do ambiente.

Capítulo 3

Algoritmos Heurísticos de Busca

3.1. Introdução

Podemos encontrar no nosso universo muitos problemas que podem ser colocados da seguinte forma: Partindo de um estado inicial encontre uma seqüência de operações que leve este estado a um estado final (objetivo). A título de exemplo citar o Problema do Caixeiro Viajante (**PCV**) onde um estado pode ser visto como a localização do caixeiro, e, o que é de nosso interesse, o **PSCFM** onde o estado inicial é dado pelas peças "brutas" e o final pelas mesmas totalmente manufaturadas e as operações transformadoras de estados são aquelas necessárias em cada peça.

Naturalmente surge um algoritmo capaz de resolver este problema, vejamos:

Algoritmo H1

ESTADO ← estado_inicial

ENQUANTO ESTADO ≠ estado_final

OPER ← alguma operação aplicável em ESTADO

SE OPER = { }

RETORNE ao ESTADO anterior e escolha uma operação distinta daquelas já escolhidas

ESTADO ← resultado da aplicação de OPER em ESTADO

É muito fácil ver que este algoritmo sempre encontra uma solução (supondo que sempre exista pelo menos uma, e que não há ciclagem) mas a mesma tanto pode ser ótima como não. Além disso, como o número de soluções "boas" pode ser bastante reduzido em relação ao número total, este algoritmo não se mostra muito eficiente.

Adotemos agora a visão de que quando uma operação é executada um estado-pai gera um estado-filho e estabelece-se um arco dirigido do pai para o filho. Note que a cada estado pode haver mais de uma operação aplicável, nesse caso o nó-pai tem tantos filhos (e respectivas ligações) quantas operações factíveis existirem. Sob esse prisma, a obtenção de uma solução é uma busca num grafo orientado implícito. Vejamos agora alguns algoritmos, que podem ser vistos como refinações daquele previamente apresentado (H1), mas dotados de alguma "inteligência".

3.2 Busca com Retrocesso.

Estratégia de busca com retrocesso (ou "backtracking") são adequados para problemas que exigem pouco esforço de busca. As idéias principais dessa estratégia são (considere que já se decidiu explorar um caminho que passa por um dado filho):

- Limitar a profundidade do grafo de busca (se o objetivo demora para ser encontrado, então o caminho atual provavelmente não é bom) e

- evitar ciclagem (isto é, retornar a um estado já investigado e infrutífero).

Se um dos casos anteriores ocorre o algoritmo retorna ao estado-pai do estado atual e decide-se por outro estado-filho.

O procedimento pode ser descrito recursivamente como a seguir:

```
Procedimento RETROCESSO (CONJ-ESTADO)

INICIO

    ESTADO-ATUAL ← último estado produzido

    SE ESTADO-ATUAL ∈ CONJ-ESTADO-ATUAL
```

RETORNE (FALHA) [1]

SE ESTADO-ATUAL = ESTADO-OBJETIVO

RETORNE (ESTADO-NULO)

SE ESTADO-ATUAL = ESTADO-NULO

RETORNE (FALHA) [2]

SE #(CONJ-ESTADO) > LIMITE

RETORNE (FALHA) [3]

CONJ-OPER ← operadores aplicáveis ao ESTADO-ATUAL

LOOPING

SE CONJ-OPER = { }

RETORNE (FALHA) [4]

OPER ← algum operador de CONJ-OPER

CONJ-OPER ← CONJ-OPER - { OPER }

ESTADO-RESULT ← resultado da aplicação de OPER em ESTADO_ATUAL

---

[1] Ocorrência de ciclagem
[2] Não existe solução através deste nó
[3] Profundidade da árvore excedeu o limite pré-definido
[4] Não existe nenhuma regra aplicável

CONJ-ESTADO ← CONJ-ESTADO ∪ ESTADO-RESULT

CAMINHO ← RETROCESSO (CONJ-ESTADO)

SE CAMINHO = FALHA

LOOPING

RETORNE (OPER ∪ CAMINHO)

FIM

3.3 Busca por Profundidade e Largura

Continuamos agora trabalhando com o mesmo problema: partindo de um estado inicial vamos aplicar operadores que o levam a um estado objetivo; porém veremos agora duas maneiras de efetuar a busca por uma solução de maneira a otimizar, de algum modo, o esforço de busca. Continuamos usando a notação de grafo orientado.

No caso onde nenhuma informação quantitativa nem qualitativa a respeito do problema está disponível, podemos apresentar um algoritmo de busca como a seguir:

Algoritmo BUSCA-EM-GRAFO

INICIO

GRAFO-BUSCA ← nó inicial

EXPANDIDOS ← { }

ABERTOS ← nó inicial

LOOPING

SE *ABERTOS* = { }

RETORNE (FALHA) [5]

ORDENE os nós de *ABERTOS* usando algum critério.

N ← primeiro nó de *ABERTOS*

ABERTOS ← *ABERTOS* - { N }

SE N = nó objetivo

RETORNE (SUCESSO) [6]

EXPANDA o nó N, aplicando a ele todos os operadores possíveis, gerando o conjunto M de nós sucessores de N. Coloque os elementos de M como sucessores de N no GRAFO-BUSCA

EXPANDIDOS ← *EXPANDIDOS* $\cup$ { N }

PARA CADA nó m $\in$ M

SE m $\in$ M e m não $\in$ *ABERTOS* $\cup$ *EXPANDIDOS*

ESTABELECA APONTADORES de m para N

ADICIONE m a *ABERTOS*

---

[5] Não há nenhum nó a ser pesquisado.

[6] O caminho solução pode ser encontrado partindo do nó objetivo até o inicial através dos apontadores pré-estabelecidos.

SE m ∈ M e m ∈ *ABERTOS*

DECIDA sobre o redirecionamento (ou não) de m para N

SE m ∈ M e m ∈ *EXPANDIDOS*

DECIDA sobre o redirecionamento (ou não) de m para N assim como o redirecionamento (ou não) de cada um de seus sucessores

LOOPING

FIM

Vale a pena comentar os dois últimos passos. A utilidade dos mesmos se dá quando o grafo de busca não é uma árvore, ou seja, um nó pode ter dois pais diferentes. Nesse momento pode haver a situação de se descobrir um novo caminho de um dado nó ao inicial mais "econômico" que um anteriormente determinado.

Existe no "LOOPING" do algoritmo um passo definido de maneira bastante vaga:

ORDENE os nós de *ABERTOS*, usando algum critério

Como fazer esta ordenação? Existem duas maneiras clássicas de se fazê-lo. São estratégias, num certo sentido complementares que podem levar a resultados qualitativa e quantitativamente bastante diferentes. Vejamos:

1) Ordene os nós de *ABERTOS* em ordem decrescente de profundidade.[7]

Exploramos desta maneira sempre o nó mais profundo da árvore até que ela não seja mais um candidato em potencial (ou seja, levou a um estado final distinto do objetivo).

[7] Definimos profundidade como sendo o número de nós antecessores ao atual caminhando retroativamente até o nó raiz da árvore.

2) Ordene os nós de *ABERTOS* em ordem crescente de profundidade.

Assim só passamos a pesquisar um candidato pertencente a um dado nível quando todos os nós do nível anterior já foram explorados, isto é, pertencem a *EXPANDIDOS*.

O Algoritmo BUSCA-EM-GRAFO com a estratégia (1) e conhecido como busca-em-profundidade ("depth-first") e com a estratégia (2) busca-em-largura ("breadth-first").

Vejamos um exemplo da solução obtida pelas duas estratégias. Considere que o nó objetivo da árvore na figura 3.1 é o nó 5. A seqüência de nós explorados pela busca-em-largura seria {1, 2, 3, 4, 5} enquanto que pela busca-em-profundidade seria {1, 2, 5}. Suponha agora que o nó objetivo seja o nó 4. A busca-em-largura exploraria os nós {1, 2, 3, 4} (nessa ordem) enquanto que a busca-em-profundidade tomaria os nós {1, 2, 5, 6, 3, 7, 8, 4}. Não há como saber, sem um conhecimento do problema qual estratégia é mais adequada. Podemos, porém, fazer uso de uma regra intuitiva: Se "acreditarmos" que o processo de busca deve levar a um número grande de mudança de estado então a busca-em-profundidade "pode" ser apropriada, caso contrário "é possível" que a busca-em-largura tenha mais sucesso.

Esse procedimento recursivo sempre retorna uma solução (podendo inclusive ser a indicação de que não existe nenhuma solução factível), porém além de ser conveniente apenas para problemas de pequeno porte (dado o número de passos que pode gerar) não há nenhuma garantia quanto a qualidade da solução obtida.

3.3 A* - Uma busca ótima

As estratégias apresentadas anteriormente tem uma deficiência: não há nenhuma garantia quanto a qualidade da solução obtida. É fácil ver que na maioria dos casos onde se aplica uma estratégia de busca, justifica-se a aplicação pela existência de um número considerável de soluções possíveis e a obtenção de uma solução boa é de fundamental importância. É fato também que os algoritmos anteriores não fazem uso de nenhuma informação quanto ao "custo" de uma solução. Quando falamos "custo" devemos abstrair o seu conceito. Por exemplo "custo" pode ser visto como: a

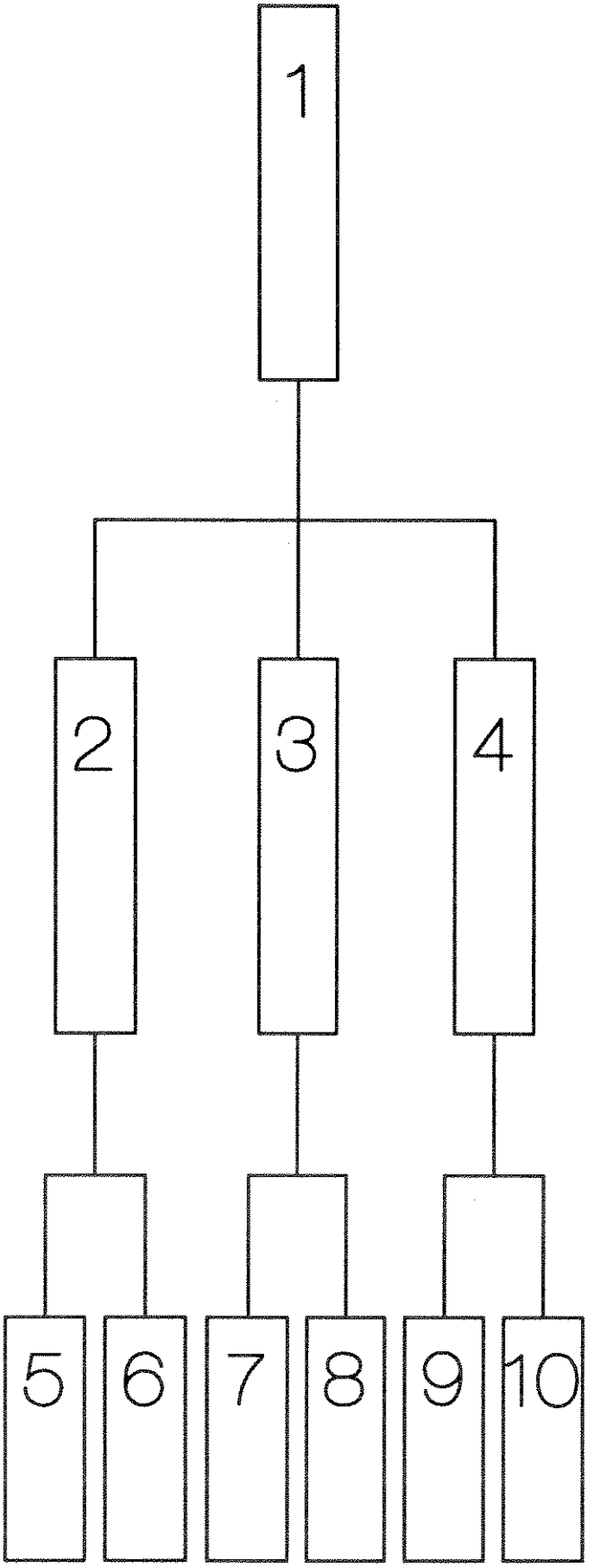


FIGURA 3.1.

profundidade do grafo de busca, número de operadores aplicados, custo econômico de uma dada operação de manufatura numa peça, etc. Nesse contexto é altamente desejável que além de se obter uma solução factível para o problema, essa solução seja ótima ou aproximadamente ótima segundo algum critério de custo. Novamente aqui invocamos o **PSCFM** como exemplo de aplicação, onde usamos como medida de custo o tempo total de processamento de uma peça. Assim quanto mais rápido o processamento da peça, melhor a solução.

Note que esse conceito de "custo" pode não ser equivalente àquele de custo econômico real, mas ainda assim será aquele utilizado por nós.

O algoritmo A^* descrito a seguir foi apresentado originalmente por Hart e outros (1968) como uma alternativa aos algoritmos de otimização combinatória tradicionais, onde a grande idéia seria a de se explorar o conhecimento do problema a ser resolvido dotando o algoritmo de "inteligência" e, além disso, garantir a obtenção de soluções ótimas segundo algum critério de custo.

Faremos uso das seguintes hipóteses:

- Existe um grafo de estados (Π) onde cada arco tem associado um custo $c_{ij} \geq \delta > 0$ necessário para se levar um estado i a um estado "descendente" j.

- Existe o nó raiz do grafo Π e conhece-se o estado-objetivo (algumas vezes este estado pode ser representado por um conjunto de nós e não apenas um).

- É possível determinar uma função de avaliação $f(.)$ cujo argumento é um nó e que mede quão promissor é a busca através daquele nó.

Neste trabalho usamos a função de avaliação descrita por Hart e outros no artigo original, ou seja:

$$f(.) = g(.) + h(.)$$

onde:

$g(.)$: é uma estimativa do custo acumulado para se chegar do estado-origem ao estado atual. Em geral podemos usar $g(.)$ como sendo o custo real acumulado.

h(.): é uma estimativa do custo restante para chegar do estado-atual até um estado-objetivo, logo

$f(.)$: é uma estimativa do custo da solução restrito a conter o nó atual.

O algoritmo A^* tem a mesma estrutura do algoritmo BUSCA-EM-GRAFO onde substitui-se o passo:

ORDENE os nós de *ABERTOS* usando algum critério.

por:

ORDENE os nós de *ABERTOS* em ordem não decrescente no valor de f(.)

e o passo:

EXPANDA o nó N, aplicando a ele todos os operadores possíveis, gerando o conjunto M de nós sucessores de N. Coloque os elementos de M como sucessores de N no GRAFO-BUSCA

tem adicionada a seguinte "instrucão":

Calcule para cada sucessor m $\in$ M o valor da função f(m).

Anteriormente citamos o fato de que o algoritmo A^* pode obter soluções ótimas segundo algum critério, mas sob que hipóteses?

Vejamos agora algumas propriedades formais do algoritmo A^*, bem como as hipóteses necessárias para a obtenção das mesmas. As demonstrações foram omitidas por poderem ser encontradas em muitos textos correlatos. Dentre estes indicamos os de Nilsson (1982) e Pearl (1984).

D.1) um algoritmo é dito **completo** se o mesmo encontra uma solução, se esta existir.

P.1) O algoritmo A^* é **completo**, mesmo para grafos infinitos (Nilsson, 1982 e Pearl, 1984).

A argumentação é a seguinte: havendo uma solução a mesma tem que se encontrar a uma profundidade finita na árvore de busca. Assim se o A^* escolhe um caminho vitualmente infinito, em algum momento o custo acumulado, $g(.)$, se tornará grande o suficiente para que o mesmo seja abandonado. Este raciocinio pode ser repetido até que reste somente o caminho solução. No caso de grafos finitos podemos afirmar que o A^* também é finito, isto é, pára tendo encontrado ou não uma solução.

D.2) Considere $h^*(.)$ como sendo o custo ótimo para se chegar do nó atual até um nó objetivo. Se a função $h(.)$ é tal que $h(.) \leq h^*(.)$ para todo nó do grafo, então a função $h(.)$ é dita **admissível**.

D.3) Um algoritmo é considerado **admissível** se o mesmo retorna a solução ótima dado que pelo menos uma solução existe.

P.3) Se o algoritmo A^* usa no cálculo de $f(.)$ uma função $h(.)$ admissível, então o algoritmo é **admissível** (Nilsson, 1982 e Pearl, 1984).

Imediatamente podemos concluir que:

R.1) O algoritmo busca-por-largura é admissível.

Dados os resultados anteriores e considerando que o algoritmo Busca-por-Largura é um caso especial do A^* onde $h(.) \equiv 0$ e $g(.) =$ profundidade do nó atual, a comprovação da asserção é imediata.

D.4) Se dispomos de duas funções heurísticas: $h_1(.)$ e $h_2(.)$ e podemos afirmar que: $h_1(.) < h_2(.)$ para todo nó do grafo, então dizemos que a função $h_2(.)$ **é mais informada que a função** $h_1(.)$.

P.4.) Suponha dois algoritmos do tipo A^*, A_1^* e A_2^* usando respectivamente $h_1(.)$ e $h_2(.)$, suponha ainda que $h_1(m) < h_2(m)$ para todo nó m não-objetivo. Então podemos afirmar que A_2^* **domina**[8] A_1^*.

Deve-se notar que um algoritmo dominar outro não implica no fato de que o mesmo seja computacionalmente mais eficiente. A idéia no caso do A^* é que se uma função $h(.)$ sempre é superior à outra então esta "enxerga" mais longe. Repare que propriedade enunciada não exige a admissibilidade da funções $h(.)$ comparadas.

D.5) Se temos: $h(n_i) \leq h(n_j) + c(n_i,n_j)$, $h(t) \equiv 0$ onde o nó n_j é descendente direto do nó n_i, e o nó t é um nó-objetivo, então a função $h(.)$ é dita **monótona**.

P.5.) No caso da função $h(.)$ ser monótona, o algoritmo A^* só expande um nó quando encontra um caminho ótimo de do nó inicial ao atual, ou seja, $g(n) = g^*(n)$.

3.4 Algoritmos de Busca Sub-ótimos.

O algoritmo A^* visto anteriormente tem muitos méritos, ainda mais se usarmos o argumento de que a idéia contida no mesmo é razoavelmente simples e, sujeita uma hipótese (de admissibilidade de $h(.)$) relativamente fraca, proporciona resultados ótimos. Mas, e se tivermos um dos seguintes casos (Pearl, 1984):

- Minimizar o custo de busca ao invés do custo da solução;

[8] No sentido de ser mais eficiente em relação ao número de nós expandidos.

- O porte do problema pode ser tal que a resolução ótima seja excessivamente demorada ou;

- A única função $h(.)$ disponível é fraca (fornecendo limites ruins) ou excessivamente difícil de ser computada.

al-go-rit-mo

Nesses casos o A^* revela um ponto fraco. Muito do esforço do algoritmo pode ser dispendido na escolha de um caminho, entre vários promissores, que se mostre infrutífero no futuro. Assim, uma função $h(.)$ não muito "inteligente" pode acabar com a "elegância" do algoritmo, forçando-o a retomar caminhos anteriormente abandonados.

Pode haver ainda o caso onde a obtenção de soluçes tenha que ser feita rápidamente e dada esta prioridade pode-se haver alguma perda na qualidade da solução.

Usando este enfoque desenvolveram-se pesquisas que usassem o A^*, um algoritmo definitivamente elegante, como base para a construção de algoritmos mais rápidos, sendo possível, porém, que se perca na qualidade da solução obtida, mantendo verdadeira a idéia da relação custo/benefício.

Uma das primeiras idéias usadas foi a de se alterar o peso das funções $g(.)$ e $h(.)$ na avaliação do nó, passando a se usar:

$$f_w(.) = (1 - w)g(.) + wh(.)$$

onde $w \in [0,1]$.

Essa idéia foi muito estudada, mas recaiu num problema: o valor de w era dependente não só do problema mas de quão apurada era $h(.)$, requerendo, dessa maneira, estudos empíricos que, de qualquer modo, não poderiam garantir a desempenho do algoritmo.

Outras variações surgiram e as mais frutíferas foram as que podiam garantir o pior caso dentro de um certo limite. A seguir veremos duas dessa variações.

3.5.1 WA^*

Este algoritmo apresentado originalmente por Pohl (Pearl, 1984, p.: 88), tem a mesma estrutura do A^*, com a exceção de como a função f(.) é calculada. Ao invés de usarmos, como no A^*, $f(.) = g(.) + h(.)$, usamos:

$$f(.) = g(.) + h(.)[1 + \varepsilon(1 - d(.)/D)]$$

onde: $d(.)$ é a profundidade do nó atual e D é a profundidade do nó objetivo (ou um limitante superior para este).

Assim a medida que a busca avança diminui-se o peso dado a $h(.)$, porém não tornando-o menor que aquele dado a $g(.)$. O nome WA^* usado aqui elucida o fato da ponderação ("weighting A^*") que é dinâmica durante o processo de busca.

Mostremos que se $h(.)$ é admissível então o algoritmo WA^* é ε-admissível, isto é, a solução encontrada é no máximo $(1 + \varepsilon)$ vezes pior que a ótima. Usando a forma como $f(n)$ é calculada, para um dado nó n pertencente ao caminho ótimo, teremos:

$f(n) = g(n) + h(n)[1 + \varepsilon(1 - d(n)/D)]$; $g(n) = g^*(n)$ dado que n pertence ao caminho ótimo

$$f(n) \leq g^*(n) + h^*(n)[1 + \varepsilon(1 - d(n)/D)]$$

$$f(n) \leq g^*(n) + h^*(n) + \varepsilon(1 - d(n)/D)h^*(n) \leq f^*(n) + \varepsilon h^*(n)$$

$$f(n) \leq f^*(s)(1 + \varepsilon)^9$$

para qualquer n, inclusive n pertencente ao conjunto de nós objetivos.

3.5.2. A_ε^*

[9] f*(.) = g*(.) + h*(.), e s é o nó raiz da árvore de busca

Trata-se de outro algoritmo ε-admissível, porém mais elaborado que o WA^*. No A^*_ε considera-se, além do conjunto de nós *ABERTOS*, o conjunto *FOCAL* onde cada nó tem atribuido a si uma segunda função de avaliação, $h_f(.)$. Esta segunda função tem como objetivo avaliar o custo "computacional", ou o esforço a ser empregado na busca, em se escolhendo um dado nó para expansão. Porém nem todos os nós fazem parte de *FOCAL* que é construído da seguinte forma:

$$FOCAL = \{n \mid f(n) \leq (1 + \varepsilon)\min_{m \in ABERTOS}\{f(m)\}\}$$

A modificação no algoritmo A* é simples, basta mudar o passo:

ORDENE os nós de *ABERTOS* em ordem crescente de $f(.)$

pelos seguintes:

CONSTRUA $FOCAL = \{n \mid f(n) \leq (1 + \varepsilon)\min_{m \in ABERTOS}\{f(m)\}\}$

ORDENE os nós de *FOCAL* em ordem crescente de $h_f(.)$

e o passo:

REMOVA de *ABERTOS* o primeiro nó, chame-o de n

por:

REMOVA de *ABERTOS* o nó equivalente ao primeiro de *FOCAL*, chame-o de n

A idéia contida aqui é razoavelmente intuitiva: entre vários nós que indicam uma solução aproximadamente equivalente escolhemos aquela que aparentemente indica o caminho "mais rápido". A função $h_f(.)$ pode eventualmente ser a mesma que $h(.)$, mas deve aproveitar o conhecimento do problema para avaliar se a busca se encontra próxima ou não de uma solução final.

Provemos agora que o algoritmo A^*_ε é ε-admissível. Considere: n_0, n_1 e t como sendo o primeiro nó de *FOCAL*, o primeiro nó de *ABERTOS* num caminho ótimo e um nó objetivo respectivamente.

Temos que para t sendo nó objetivo, $f(t) = g(t)$. Então:

$f(n_1) \leq f^*(s)$ (Veja Lema 1, Pearl (1984), p.: 77)

$f(n_0) \leq f(n_1)$

$f(t) \leq f(n_0)(1 + \varepsilon) \leq f(n_1)(1 + \varepsilon) \leq f^*(s)(1 + \varepsilon)$

Capítulo 4

Resolvendo o PSCFM

4.1. Introdução

A estratégia que propomos nesta tese é baseada naquela apresentada por Shaw & Whinston (1989). No nosso caso, porém, não utilizaremos um sistema baseado em conhecimento e sim um enfoque baseado em duas fases. Mais especificamente:

- Fase Linear:

  Consiste em seqüenciar cada peça individualmente sem levar em consideração as outras eventualmente existentes, elaboradando para cada uma um plano linear.

- Fase Não-Linear:

  São resolvidos os conflitos que venham a existir, encontrando o que chamaremos de plano não-linear.

Ao final da Fase Não-linear temos o seqüenciamento final que deve ser executado.

Quando afirmamos que não usaremos um enfoque do tipo **SE** queremos dizer que usaremos uma estratégia algoritmica, portanto determinística, onde o conhecimento do problema terá uma participação razoável. Um ponto importante a ser ressaltado nesta estratégia é que não usamos nenhuma técnica de programação matemática, do tipo programação linear ou dinâmica, apenas o enfoque heurístico empregado pelo A^* e algumas regras de decisão quanto aos conflitos a serem realocados.

Neste capítulo nos concentraremos em discutir a implementação da estratégia usada mostrando como serão utilizadas as propriedades enunciadas anteriormente.

Para tanto devemos reforçar as hipóteses que usamos para modelar, e resolver, o **PSCFM**

Suponha doravante:

- Existem N peças a serem manufaturadas;
- Cada peça necessita de O operações, designadas por "nomes" distintos, e ordenadas;
- Existem M máquinas disponíveis, possivelmente distintas;
- Existe um sistema de transporte automático, um robô, por exemplo, e o tempo de transferência de uma peça entre duas máquinas quaisquer é constante[1] (t_t);
- Existe um dispositivo para carga e descarga de uma peça na célula e que realiza estas operações num tempo constante;
- Cada peça tem atribuída a si uma data de entrega, ou seja, uma data prevista como limite para sua finalização;
- Cada operação requer um tempo t_{ik} para ser processada, onde i e k denotam respectivamente a operaçào e uma máquina. Se a máquina não é capaz de realizar a operação i então $t_{ik} \rightarrow +\infty$;
- A seqüência de operações não contém repetições;
- Cada máquina dispõe de um armazém para contenção temporária de peças, que eventualmente tenham que esperar pela "liberação" da mesma, cuja capacidade é grande o suficiente.

[1] Esta hipótese é apenas simplificadora, podendo ser fácilmente generalizada.

Isto posto, podemos seguir demostrando a estratégia que propomos.

4.2 Fase Linear.

Um seqüenciamento para uma peça é dado por uma seqüência linear de quíntuplas da forma (j, i, k, t_i^j, t_f^j) onde j, i, k, t_i^j, t_f^j denotam respectivamente: a peça, a operação, a máquina, a data inicial e a final relativa ao processamento. Um seqüenciamento ótimo é aquele onde a data final (t_f) da última quíntupla é a menor possível, mantendo, é claro, a factibilidade.

Suponha por exemplo a sequinte matriz T_{mo} onde cada t_{ki} representa o tempo necessário para que a máquina k processe a operação i:

$$T_{mo} = \begin{bmatrix} 3 & 2 & +\infty \\ 7 & 1 & 4 \\ +\infty & 5 & 3 \end{bmatrix}$$

e uma peça que necessite as seguintes operações:

$$S_{op} = [\ 2\ 3\ 1\ ]$$

Então um seqüenciamento factível seria (considere que o tempo de transporte entre peças é $t_t = 1$):

S_1: (., 2, 3, 0, 5), (., 3, 2, 6, 10), (., 1, 1, 11, 14)

e outro também factível seria:

S_2: (., 2, 2, 0, 1), (., 3, 2, 1, 5), (., 1, 1, 6, 9)

Como dissemos anteriormente estamos interessados em seqüenciamento que minimizem o "makespan" e nesse caso o seqüenciamento S_2 é melhor que o S_1, porém no caso simples acima quantos são os seqüenciamentos possíveis? Apenas 12, mas o exemplo apresentado é de dimensão muito pequena. Para um problema de maior

dimensão teríamos um esforço enorme se resolvessemos encontrar a solução por inspeção, ainda mais no caso que estudamos nesta tese, o **PSCFM**, onde N problemas de seqüenciamento de uma peça em M máquinas devem ser resolvidos. Nesses casos a matriz T_{mo} é da ordem de M x O onde O é o número de operações necessárias em cada peça que pode ser grande, independentemente de M e N.

Surge então a questão de como resolver este problema.

Repare que podemos propor uma formulação de "caminho mínimo" para o problema do roteamento (plano linear) de cada peça.

De fato, cada peça deve ter processada O operações, ordenadas linearmente. Para cada operação i existem M máquinas disponíveis (algumas possivelmente infactíveis).

Assim, suponha um nó E_0 representando o estado de carregamento da peça na célula. Suponha ainda um conjunto de MO estados E_{ik} representando a operação i na máquina k e um nó ficticio E_f simbolizando o descarregamento da peça. Estes estados estão ligados (e orientados) da seguinte maneira ($c(A, B)$ representa o custo de se levar o estado A ao estado B):

$c(E_0, E_{1k}) = t_t + t_{1k}$; para k = 1, 2, ..., M;

$c(E_{nl}, E_{(n+1)k}) = \alpha t_t + t_{(n+1)k}$; para n = 2, 3, ..., (O-1); k, l = 1, 2, ..., M; e α = 1 se $k \neq l$ ou 0 caso contrário e

$c(E_{(O-1)k}, E_f) = t_t$; para k= 1, 2, ..., M.

A solução ótima para o roteamento é o caminho mais curto (mínimo) de E_0 à E_f. O mesmo poderia ser encontrado usando-se algoritmos tradicionais como o de Dijkstra (Minieka, 1978, p.:43). Porém resolvemos abordar este problema usando o algoritmo A^* recém-apresentado. Nossa decisão foi também encorajada pelo relato de Ribeiro (1980) que sugere a superioridade do A^* em relação ao algoritmo de Dijsktra.

Computacionalmente a implementação algoritmo A^* não é complexa. Podemos resumi-la no seguinte procedimento, escrito em pseudo-código. (Suponha que

adicionemos ao início e fim do conjunto ordenado O, os índices 0 e $+\infty$, relacionados a carga e descarga da peça respectivamente).

Procedimento PLANO-LINEAR;

INICIO

OBTENHA: T_{mo}, S_{op}

LS $\leftarrow$ um limitante superior para o valor da solução

$n \leftarrow (j, 0, 0, 0, 0)$

CALCULE $f(n) = g(n) + h(n)$

LOOPING

SE $n = (j, +\infty, ., ., .)$

FIM

VERIFIQUE qual a próxima operação necessária em n, chame-a de p_o.

PARA cada máquina m_i tal que $t_{m_i p_o} < +\infty$

CRIE um nó n'

ESTABELEÇA um apontador de n' para n e vice-versa

SE $m_i \neq m_n$ (m_n é a máquina à qual a operação anterior está alocada)

$$n' \leftarrow (j, p_o, m_i, t_f^j + t_t, t_f^j + t_t + t_{m_i p_o})$$

SENÃO

$n' \leftarrow (j, p_o, m_i, t_f^j, t_f^j + t_{m_i, p_o})$

CALCULE $f(n') = g(n') + h(n')$

VERIFIQUE se o estado correspondente a n' já existe. Redirecionando, se for o caso, o seu antecessor (Veja o procedimento TROCA-DE-NOS, a seguir).

SE $f(n') \leq LS$

ARMAZENE n' em *ABERTOS*

SENÃO

DESCARTE-O

ORDENE *ABERTOS* em ordem crescente de $f(.)$

$n \leftarrow$ primeiro nó de *ABERTOS*

ABERTOS $\leftarrow$ *ABERTOS* - { n }

LOOPING

FIM

OBTENHA a solução caminhando de n "para trás" (através dos apontadores) até chegar ao nó raiz

VALOR da solução = t_f^n

Discutiremos agora alguns detalhes desta implementação. Como realizar o cálculo de $f(.) = g(.) + h(.)$? Como vimos anteriormente $g(.)$ e $h(.)$ são estimativas do custo até um nó e do custo restante respectivamente. Porém, como nossa noção de custo está associada ao tempo de processamento podemos fazer, para todo nó n criado na busca:

$$g(n) = t_f^n$$

isto é, a estimativa de custo para se chegar até esta operação é exatamente o gasto já computado para fazê-lo.

Com respeito a $h(.)$ a idéia é bastante simples. Queremos, obviamente, $h(.)$ tal que possamos provar a condição $h(.) \leq h^*(.)$ e obter vantagem da admissibilidade do A^*. Suponha que dada uma certa instância da peça em processamento ainda restem operações a serem realizadas e essas operações estejam representadas pelo conjunto O_r. Podemos escolher as seguintes funções $h(.)$:

- $h_1(.) \equiv 0$;
- $h_2(.) = \min_{m \in M}\{t_{m,o+1}\}$ onde $(o+1)$ denota a próxima operação necessária, ou
- $h_3(.) = \sum_{o \in O_r} \min_{m \in M}\{t_{m,o}\}$

As funções $h(.)$ apresentadas acima são admissíveis. Vejamos:

- $h_1(.)$ é a clássica Busca-Por-Largura já provada admissível.
- $h_2(.)$ é classicamente chamada de heurística míope já que dá uma estimativa muito limitada a respeito do resto do sequenciamento e obviamente $h_2(.) \leq h^*(.)$
- $h_3(.)$, a qual chamaremos de heurística míope composta é admissível. De fato:

$$h^*(.) = \sum_{o \in O_r} (t_{o^*,o}) + ut_t$$

onde "o*" $\in M$ denota a máquina usada para a realização da operação o na seqüencia ótima e $u \in [0, \#(O_r)]$ indica o número de transportes, entre máquinas, que se fizeram necessários. Claramente temos:

$$\min_{m \in M}\{t_{m,o}\} \leq t_{*,o} \text{ para } o \in O_r$$

então $h_3(.) \leq h^*(.)$.

Outro ponto importante a se ressaltar é o de como podemos determinar o limitante superior LS. As buscas ε-admissíveis nos fornecem, sempre, limitantes superiores. A idéia é usar uma busca ε-admissível que tenha um bom desempenho, no sentido de ser rápida, para determinar LS. Usando o resultado de Ghallab & Allard (1983) de que uma busca ε-admissível com $\varepsilon \rightarrow +\infty$ teria uma complexidade linear, resolvemos usar uma busca desta classe para a obtenção de LS. Optamos por usar WA^* por que como poderemos ver na próxima seção ("Resultados Computacionais") esta apresentou resultados muito melhores que A^*_ε. Na verdade o LS determinado pelo WA^* se mostrou muito bom, se considerarmos que o valor de $\varepsilon \rightarrow +\infty$ não nos garante absolutamente nada, e extremamente rápido de ser obtido.

É fácil ver que $h_3(.) \geq h_2(.) \geq h_1(.)$, então pela propriedade (**P.4**) da seção (**3.4**) o A^* com $h_3(.)$ terá melhor desempenho que usando as funções $h_2(.)$ e muito melhor que $h_1(.)$. Um exemplo prático do poder de dominância de funções $h(.)$ pode ser encontrado no trabalho de Nascimento (1989).

Concentrando-nos então em $h_3(.)$, afirmamos que $h_3(.)$ é monótona segundo a definição (**D.5**) da seção (**3.4**).

Considere o como sendo a próxima operação do conjunto O_r, a ser realizada e denotemos: $\bar{t}_{m,o} = \min_{m \in M}\{ t_{m,o} \}$. Então:

$$h_3(n_i) = \sum_{o \in O_r} \bar{t}_{m,o} \quad \text{e}$$

$h_3(n_j) = \sum_{o \in O_{r-o}} \bar{t}_{m,o}$ ainda podemos reescrever

$h_3(n_i) = \sum_{o \in O_{r-o}} \bar{t}_{m,o} + \bar{t}_{m,o}$ logo:

$h_3(n_i) = h_3(n_j) + \bar{t}_{m,o}$ e como $\bar{t}_{m,o} \leq t_{o^*,o}$

$h_3(n_i) \leq h_3(n_j) + t_{*,o}$ donde finalmente:

$h_3(n_i) \leq h_3(n_j) + c(n_i, n_j)$

Agora que sabemos que $h_3(.)$ é monótona (na verdade $h_1(.)$ e $h_2(.)$ também o são) usamos a propriedade (**P.5**) da seção (**3.3**) para afirmar que se o A^* expande o nó n então $g(n) = g^*(n)$. Assim o A^* não redirecionará filhos de nós já expandidos durante a busca.

Agora comentaremos o que podemos usar do que já foi discutido para gerar uma aplicação que use o algoritmo WA^* assim como o A^*_ε.

O algoritmo WA^* usa a mesma estrutura do A^*, diferindo deste apenas no cálculo da função de avaliação f(.) que é a seguinte:

$$f(.) = g(.) + h(.)[1 + \varepsilon(1 - d(.)/D)]$$

Nesse caso podemos usar uma informação, muito simples, do **PSCFM** para determinar o valor exato de D ao invés de usarmos um limitnate superior para o mesmo. Recorde que D representa a profundidade na árvore de busca onde se encontra o nó objetivo. Vejamos: cada peça deve ter processada um conjunto O de operações, exatamente, então um nó objetivo, isto é, um nó onde não haja operações a serem realizadas, estará a uma profundidade igual a cardinalidade do conjunto O, ou seja, $D = \#(O)$. Na construção, em pseudo-código, feita anteriormente do A^* não consideremos a operação "+ ∞" de maneira a aumentar $\#(O)$. Esta operação é adicionada a O apenas de modo a ilustrar melhor o funcionamento do algoritmo, enquanto que a operação "0" deve ser considerada por simbolizar o nó raiz (carregamento da peça) da

árvore de busca.

Agora sobre o A^*_ε nos resta comentar a forma como construiremos o conjunto *FOCAL*. Duas idéias nos surgem imediatamente:

$h_{f1}(.) = h(.)$ e

$h_{f2}(.) = \#(O_r)$

No caso de $h_{f1}(.)$ podemos escolher $h_3(.)$ que se mostrou mais forte que $h_1(.)$ e $h_2(.)$ e assim valores pequenos/(grandes) de $h_3(.)$ significariam que estamos perto/(longe) de um nó-objetivo, enquanto que $h_1(.)$ e $h_2(.)$ seriam pouco sensíveis nesse aspecto. Uma função do tipo $h_{f2}(.)$ parece mais adequada à função que deve cumprir, que é a de estimar o esforço computacional necessário para encontrar uma solução. Assim escolhemos $h_f(.) = h_{f2}(.)$.

4.2.1.Um exemplo

Suponha o seguinte conjunto de dados:

$$T_{mo} = \begin{bmatrix} +\infty & 9 & 2 \\ 7 & 3 & 2 \\ 8 & +\infty & 5 \end{bmatrix}, S_{op} = [\,2\ 3\ 1\,] \text{ e } t_t = 2$$

onde T_{mo} denota a matriz de tempos de processamento, ou seja $t_{ik} = (T_{mo})_{ik}$ e $(S_{op})_r$ denota a r-ésima operação necessária na peça.

Qual é o roteamento ótimo, isto é, qual é a seqüência de máquinas de implica no processamento mais rápido da peça em questão ?

Na figura 3.1 apresentada a seguir temos a árvore de busca produzida pelo A^* durante a resolução do problema acima. A seguir detalharemos alguns passos do algoritmo[2].

[2] A dupla (i,k) denota um estado onde a operação i é processada na máquina k.

INICIO

ABERTOS = { (.,0,0,0,0) $_{f=14}$ }

EXPANDIDOS = { }

LOOPING #1

N ← { (.,0,0,0,0) }

M ← { (.,2,2,2,5) $_{f=14}$, (.,2,1,2,11) $_{f=20}$ }

ABERTOS = *M*

EXPANDIDOS = { (.,0,0,0,0) }

LOOPING #2

N ← (.,2,2,2,5)

M ← { (.,3,2,5,7) $_{f=14}$, (.,3,1,7,9) $_{f=16}$, (.,3,3,7,12) $_{f=19}$ }

EXPANDIDOS ← { (.,0,0,0,0), (.,2,2,2,5) }

ABERTOS ← { (.,3,2,5,7) $_{f=14}$, (.,3,1,7,9) $_{f=16}$, (.,3,3,7,12) $_{f=19}$, (.,2,1,2,11) $_{f=20}$ }

e assim sucessivamente até

LOOPING #4

ABERTOS ← { (.,1,2,7,14) $_{f=16}$, (.,3,1,7,9) $_{f=16}$, (.,1,3,9,17) $_{f=17}$, (.,3,3,7,12) $_{f=19}$, (.,2,1,2,11) $_{f=20}$ }

N ← (.,1,2,7,14) ≡ nó-objetivo

FIM

SOLUCAO ← { (.,0,0,0,0), (.,2,2,2,5), (.,3,2,5,7), (.,1,2,7,14) }

A seguir veremos alguns detalhes sobre a estrutura de dados e os resultados computacionais obtidos usando-se o que foi apresentado anteriomente.

4.2.2 Estrutura de dados

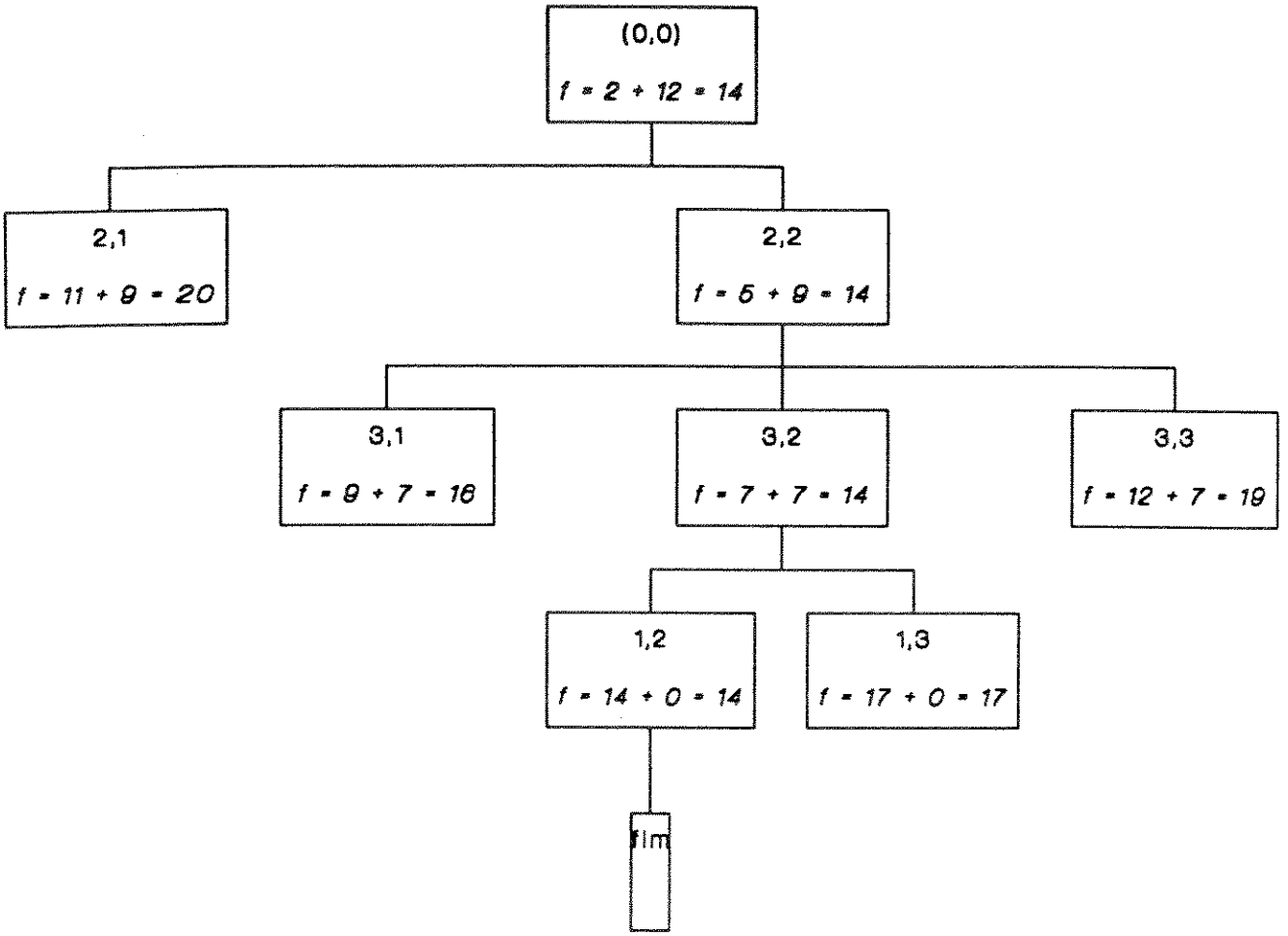

(0,0)
f = 2 + 12 = 14
2,1
f = 11 + 9 = 20
2,2
f = 5 + 9 = 14
3,1
f = 9 + 7 = 16
3,2
f = 7 + 7 = 14
3,3
f = 12 + 7 = 19
1,2
f = 14 + 0 = 14
1,3
f = 17 + 0 = 17
fim

FIGURA 4.1.

A estrutura de dados implementadas tem destaque em duas partes da resolução dos sub-planos: grafo de busca e ordenação do conjunto *ABERTOS*. Ambos os casos têm em em comum a caracterîstica de fazerem uso de alocação dinâmica de memória. Usamos, deste modo, apenas a memória realmente necessária, otimizando o seu uso. Sabemos de antemão que o número total de estados existentes é $OM + 1$, onde inclui-se o nó de carregamento da peça. Assim poderîamos usar uma estrutura matricial de ordem (OM) para armazenar todos os estados, porém resolvemos usar alocação dinâmica, em detrimento do tempo de processamento, de modo a economizar espaço de memória. Convém ressaltar que quanto melhor for a heurîstica mais dirigida é a busca, logo, menor o número de nós expandidos e gerados.

- Grafo de Busca

O principal ponto na construção do grafo de busca é a não repetição de estados já gerados, ou seja, quando expande-se um nó (estado) geram-se outros nós, possivelmente já existentes no grafo de busca. Um nó no grafo caracteriza-se pela operação (ou seu ordinal, relativo a seqüencia da peça) e a máquina a qual a mesma está alocada.

Quando da ocorrência da geração de um nó já existente procede-se aos seguintes passos:

Procedimento TROCA-DE-NOS

INICIO

$n \leftarrow$ novo nó e $v \leftarrow$ nó anterior

SE $g(n) < g(v)$

COPIA as informações de n para v

MUDE o nó antecessor de v para o nó antecesseor de n

SE n pertence *EXPANDIDOS*

ATUALIZA os CUSTOS de seus sucessores de acordo com o novo custo $g(n)$

REORDENA a lista *ABERTOS*

APAGA o nó v

SENAO

APAGA o nó n

FIM

- Ordenando *ABERTOS*

O número máximo de nós a serem ordenados no conjunto *ABERTOS* é $O(M-1)+1$, ou seja, é de ordem de $O(OM)$. Nesta etapa da resolução poderíamos usar o algoritmo HEAP (ou lista de prioridades segundo Szwarcfiter & Markenzon (1989)). Trata-se de um algoritmo simples e eficiente para o seguinte caso: dada uma lista de prioridades, INSERIR um novo elemento qualquer e REMOVER aquele de maior prioridade, provida é claro uma função de avaliação. As operações de INSERIR e REMOVER tem tempo de execução $O(log(n))$, onde n é o número de elementos da lista. Porém, há um problema. Como vimos anteriormente no caso de encontrarmos um caminho mais barato para um nó já existente, devemos reordenar o conjunto *ABERTOS*. Nesse caso qualquer elemento de *ABERTOS* está sujeito a reavaliação e consequentemente pode mudar de posição (prioridade) dentro da lista montada pelo algoritmo HEAP. Esse procedimento de reavaliação e reordenação da lista não é trivial (Imai & Iri, 1984) e torna-se preferível utilizar uma outra estrutura, mais simples, que dentro de uma certa faixa (de número de elementos na lista) ainda seja competitiva com o algoritmo HEAP.

Segundo comunicação verbal de Bortolon & Garcia (1989) se o número de elementos na lista de prioridades (*ABERTOS*) for de até 150 elementos então uma

estrutura do tipo lista ligada ordenada é mais vantajosa em relação à lista de prioridades. Repare que 150 é um número razoável, já que corresponde a um seqüenciamento de peças com, por exemplo, 15 operações e 10 máquinas que já representa um problema de porte considerável.

Essa estrutura da lista ordenada, também conhecida como "fio" é de implementação extremamente simples e suas operações de INSERIR e RETIRAR são executadas respectivamente em $O(n)$ e $O(1)$.

4.2.3 Resultados Computacionais

Para obtermos problemas a serem resolvidos usamos dados gerados da seguinte maneira[3]:

$t_{ij} \in \{[0, 30], G\}$ e $P(t_{ij} = G) = 0.5$

Cada peça terá processada O operações distintas cuja ordem será sorteada aleatóriamente. Usaremos, para estudo, $O = 5, 10, 15$ operações;

O número de máquinas disponíveis, M, que tem influência direta na tendência "de alargamento" da árvore de busca assumirá os possíveis valores: 3, 5, e 7 máquinas.

Resolveremos o plano linear usando as seguintes estratégias:

- A^* com $h(.) = h_3(.)$;
- WA^* com $h(.) = h_3(.)$ e $\varepsilon \in \{0.1, 0.2, G\}$ e
- A^*_ε com $h(.) = h_3(.)$, $h_f(.) = \#(O_r)$ e $\varepsilon \in \{0.1, 0.2, G\}$.

[3] Considere $G \rightarrow +\infty$

Teremos assim 9 classes (dimensão O x M) de problemas, para cada classe serão gerados 10 conjuntos de dados distintos, ou seja 90 problemas. Cada um destes será resolvido usando as estratégias colocadas acima, ou seja, de 7 modos diferentes. Cabe agora dizer como faremos as comparações entre as estratégias, de modo a tentarmos eleger uma delas como mais promissora. Usaremos como base os seguintes dados gerados por cada estratégia (média de 10 casos):

1) número de nós expandidos;

2) número de direções tomadas;

3) tempo de CPU exigido e

4) perda no valor da função objetivo.

As medidas (1) e (2) são comumente usadas nos estudos de buscas heurísticas. A medida (3) normalmente não é apresentada nos estudos encontrados na literatura e, no nosso modo de ver, é bastante importante para se decidir sobre a viabilidade das buscas; por exemplo: uma busca inteligente, isto é, com um número de nós expandidos baixo pode ser extremamente elaborada e de cálculo complexo implicando um grande uso de CPU (medida (3)), podendo ser desvantajosa em relação a uma outra busca de cálculos rápidos, as vezes até redundantes. A medida (4) ilustra o desempenho das estratégias ε-admissíveis com vários valores de ε em relação ao A^*.

A seguir apresentamos as tabelas com os resultados obtidos num computador VAX 11/785 com sistema operacional VMS 5.3.. Os programas foram escritos em PASCAL e usamos a versão 3.8 do compilador disponível no ambiente. Chamamos atenção para o fato de que para $O = 15$ o A_ε^* não foi testado devido aos resultados ruins apresentados para os casos com O igual a 5 e 10 operações. As tabelas serão apresentadas a seguir.

	Número de Máquinas		
5 Operações	3	5	7
A^*	6.60	7.00	8.20
WA^* (10%)	6.60	6.60	7.80
WA^* (20%)	6.40	6.40	7.60
WA^* (G %)	6.00	6.00	6.00
A^*_ε (10%)	6.40	6.60	8.20
A^*_ε (20%)	6.00	7.80	8.20
A^*_ε (G %)	6.00	6.00	6.00
10 Operações			
A^*	12.60	16.00	19.00
WA^* (10%)	11.80	14.60	17.60
WA^* (20%)	11.20	12.40	15.00
WA^* (G %)	11.00	11.00	11.00
A^*_ε (10%)	12.20	16.00	16.20
A^*_ε (20%)	11.60	15.60	17.20
A^*_ε (G %)	11.00	11.00	11.00
15 Operações			
A^*	20.20	26.00	34.80
WA^* (10%)	18.80	23.80	30.00
WA^* (20%)	16.80	19.20	25.80
WA^* (G %)	16.00	16.00	16.00

Tabela 4.2.1: Número de nós expandidos.

	Número de Máquinas		
5 Operações	3	5	7
A^*	2.20	2.60	4.60
WA^* (10%)	2.20	2.00	4.20
WA^* (20%)	1.80	1.80	3.40
WA^* (G %)	1.00	1.00	1.00
A^*_ε (10%)	1.60	1.80	4.20
A^*_ε (20%)	1.00	3.20	4.60
A^*_ε (G %)	1.00	1.00	1.00
10 Operações			
A^*	3.40	9.00	13.40
WA^* (10%)	2.00	6.60	11.20
WA^* (20%)	1.40	3.40	7.80
WA^* (G %)	1.00	1.00	1.00
A^*_ε (10%)	3.00	7.60	8.20
A^*_ε (20%)	2.00	8.40	9.20
A^*_ε (G %)	1.00	1.00	1.00
15 Operações			
A^*	8.00	17.00	28.40
WA^* (10%)	5.40	13.00	22.00
WA^* (20%)	2.40	6.20	15.40
WA^* (G %)	1.00	1.00	1.00

Tabela 4.2.2: Número de direções tomadas.

	Número de Máquinas		
5 Operações	3	5	7
A^*			
WA^* (10%)	0.00	0.00	0.00
WA^* (20%)	0.00	0.00	0.00
WA^* (G %)	0.00	0.27	0.50
A^*_ε (10%)	0.95	2.08	2.65
A^*_ε (20%)	12.68	7.73	7.10
A^*_ε (G %)	49.11	87.75	113.01
10 Operações			
A^*			
WA^* (10%)	0.00	0.00	0.00
WA^* (20%)	0.62	0.00	0.00
WA^* (G %)	0.85	0.00	0.85
A^*_ε (10%)	2.95	2.28	1.82
A^*_ε (20%)	11.64	8.32	7.80
A^*_ε (G %)	38.19	105.26	171.95
15 Operações			
A^*			
WA^* (10%)	0.00	0.00	0.00
WA^* (20%)	0.16	0.90	0.00
WA^* (G %)	1.56	0.82	1.73

Tabela 4.2.3: Perda no valor da solução [%].

	Número de Máquinas		
5 Operações	3	5	7
A^*	28.00	42.00	58.00
WA^* (10%)	28.00	42.00	56.00
WA^* (20%)	28.00	36.00	62.00
WA^* (G %)	14.00	16.00	26.00
A^*_ε (10%)	32.00	58.00	76.00
A^*_ε (20%)	32.00	64.00	84.00
A^*_ε (G %)	20.00	34.00	48.00
10 Operações			
A^*	54.00	102.00	176.00
WA^* (10%)	54.00	100.00	164.00
WA^* (20%)	50.00	90.00	148.00
WA^* (G %)	30.00	42.00	60.00
A^*_ε (10%)	86.00	160.00	226.00
A^*_ε (20%)	84.00	170.00	272.00
A^*_ε (G %)	64.00	112.00	182.00
15 Operações			
A^*	94.00	176.00	360.00
WA^* (10%)	94.00	170.00	324.00
WA^* (20%)	88.00	154.00	292.00
WA^* (G %)	44.00	70.00	108.00

Tabela 4.2.4: Tempo de CPU [1/100 seg.].

Observando a tabela (4.2.1) vemos que o número de nós expandidos pelo WA^* é melhor que aqueles gerados pelo A^*_ε (para o mesmo valor de ε). O número máximo de nós a serem expandidos é M(O - 1) + 2 e o número mínimo = O + 1[1]. Repare que $\varepsilon = G$ implicou no número mínimo de nós expandidos para as buscas ε- admissíveis, ou

[1] Incluimos aqui o nó de carregamento da peça.

seja uma busca com complexidade linear. A medida que ε cresce o número de nós expandidos diminui, mostrando o relaxamento da busca, tanto para WA^* como no caso do A_{ε}. No A^* o número de nós expandidos girou em torno de 40 % do número máximo.

De acordo com tabela (4.2.2) para $\varepsilon = G$ o número de direções tomadas pelas buscas ε-admissíveis é igual a 1, isso mostra que a busca é feita por profundidade ("depth-first") apenas uma vez expandindo o número mínimo de nós (tabela 4.2.1.). Para um mesmo valor de ε e $O = 10$ o número de direções tomadas pelo WA^* é menor que aquele de A_{ε}^*, donde concluimos que o WA^* é mais dirigido (informado) que o A_{ε}^*. Num problema de menor porte ($O = 5$)) o A_{ε}^* se mostrou um pouco mais dirigido. A tendência é de que conforme cresça O o WA^* vá se tornando superior. O motivo é que a função $h_F(.) = \#(O_r)$ avalia igualmente todos os nós de uma mesma profundidade. Assim, torna-se importante o critério de desempate na escolha do nó a ser expandido. No presente estudo essa escolha foi feita aleatóriamente.

Um resultado claro da tabela (4.2.3) é o de que a perda no algoritmo WA^* foi muito pequena em todos os casos. O A_{ε}^* piora o desempenho com $\varepsilon = G$ a medida que M cresce, ou seja, que a árvore de busca se torna mais larga, enquanto que o WA^* não é afetado por esta mudança. Na verdade, o WA^* nunca perdeu mais que 5% em relação ao valor ótimo.

Finalmente, da tabela (4.2.4) temos que o A_{ε}^* se mostra muito mais lento que o WA^*. Uma razão para isto é a necessidade de se montar e desmontar o conjunto *FOCAL* toda vez que se buscava um nó para se expandir. Ainda com relação ao A_{ε}^*, quanto maior o ε, mais nós constituiam o conjunto *FOCAL* o que explica o crescimento do tempo de CPU proporcionalmente ao crescimento do ε. Para $\varepsilon = G$ o WA^* se mostrou muito rápido, com o A_{ε}^* não ocorre o mesmo (chegando mesmo a ser mais lento que o A^*) porque, embora a busca seja linear, o algoritmo sofre da deficiência (de implementação) comentada acima.

Assim sendo, podemos chegar a seguinte conclusão. O A_{ε}^* tem desempenho inferior em relação a todas as medidas investigadas quando comparado com o WA^*. As buscas com WA^* foram bastante rápidas e, o que merece maior atenção, apresentaram

resultados muito próximos do valor ótimo da solução. Na verdade, todas as buscas ε-admissíveis (com $\varepsilon < G$), e o A^*, usaram como limitante superior, LS, o valor fornecido pelo WA^* com $\varepsilon = G$. Poderia-se argumentar que essa procura pelo LS torna a busca mais lenta, porém, com isso o A^* conseguiu podar, ou seja, eliminar da lista de candidatos a expansão, de 20% (com O = 15) até 40% (com O = 5) nós reduzindo sobremaneira o esforço de busca, e expansão, de nós. Além disso, usar simplesmente o valor fornecido por uma busca ε-admissível implica no risco de se estar usando um valor perto do pior caso e o A^* "bem podado" se torna mais rápido garantindo resultados ótimos. Contudo no caso de "urgência" na obtenção da solução o uso do WA^* deve, em geral, implicar em bom resultados.

4.2 Fase Não-Linear.

Neste ponto da resolução do **PSCFM** já temos posse dos N planos lineares correspondentes ao seqüenciamento de todas as peças individualmente, que foram obtidos de modo a termos certeza de que são os melhores possíveis segundo o critério de minimização do "makespan". Resta então "mesclar" estes N sub-planos de maneira a obtermos um único, que é a solução, ou melhor, uma solução para o **PSCFM**. Na busca do plano não linear teremos que resolver conflitos de recursos que definimos a seguir:

conflito: Duas peças distintas usam simultâneamente, de acordo com seus planos lineares, um mesmo recurso (máquina).

Agora, como resolver os conflitos? Como vimos anteriormente um conflito é definido pela co-existência simultânea de varias peças numa mesma máquina. Há basicamente duas maneiras de se desfazer um conflito:

1) Realocando peças para outras máquinas (ociosas e factíveis pois não há interesse em apenas "mudar o conflito de lugar") ou

2) Retardar o inicio da operação necessária de algumas peças para após o término do uso daquela máquina por alguma outra peça.

Estes conflitos têm que ser encontrados e resolvidos, ou seja, tem que deixar de existir. O racíocinio empregado é o seguinte: determinar um intervalo de conflito; identificar todas as máquinas e as peças que as usam (formando assim os conjuntos de conflito); resolver estes conflitos (avançando para a próxima operação corrente de cada peça que, no intervalo de conflito, esteja sozinha). Este passos devem ser repetidos até que se chegue a última operação de cada peça. Esta última operação pode ser representada por uma operação fictícia G. Vejamos como encontrar os conflitos e resolvê-los usando o seguinte algoritmo:

Procedimento ENCONTRA-CONFLITOS

INICIO

OPER-CORR(j) = 1 para $j = 1, 2, ..., N$[2]

LOOPING

$T_i \leftarrow$ inicio mais cedo das operações correntes.

$T_f \leftarrow$ final mais cedo das operações correntes.

PARA cada peça n faça

SE $t_i^n < T_f$ e $t_f^n > T_i$[3]

COLOQUE n em CONJUNTO-PEÇAS

COLOQUE m em CONJUNTO-MAQUINA

PARA cada $m \in$ CONJUNTO-MAQUINA

---

[2] OPER-CORR(j) denota o ordinal da operação corrente da peça j.

[3] Repare que as operações correntes (candidatas a serem conflitantes) não devem necessáriamente estar totalmente contidas no intervalo $[T_i, T_f]$.

PARA cada n ∈ CONJUNTO-PEÇAS

SE a operação corrente da peça *n* está atribuída a *m*

COLOQUE n EM CONJUNTO-CONFLITO

SE CONJUNTO-CONFLITO = { 1, 2, ..., N } e *m* = *G*

FIM

SE #(CONJUNTO-CONFLITO) ≠ 1

EXECUTE o algoritmo RESOLVA-CONFLITO

SENAO

OPER-CORR(*u*) = OPER-CORR(*u*) + 1[4]

LOOPING

FIM

A SOLUÇAO pode ser obtida inspecionando-se todos os planos lineares finais.

Resta ainda a questão de como resolver os conflitos identificados. Para isso propomos o algoritmo que se segue.

procedimento RESOLVA-CONFLITO

INICIO

[4] *u* é a única peça pertencente a CONJUNTO-CONFLITO.

LOOPING

USE alguma REGRA de seleção para identificar uma peça P a ser realocada.

SE há máquinas disponíveis factíveis

MUDE a P da máquina atualmente usada para a mais rápida entre as disponíveis

ENCONTRE um novo plano linear ótimo para P, a partir da operação corrente (use o programa PLANO-LINEAR apresentado anteriormente.

SENÃO

ADIE O inicio desta operação de modo que P não conflite com nenhuma outra peça. Faça: $t_i^P \leftarrow$ fim mais tarde das operações correntes

RETIRE P de CONJUNTO-CONFLITO

SE #(CONJUNTO-CONFLITO) = 1

OPER-CORR(u) = OPER-CORR(u) + 1

FIM

SENÃO

LOOPING

FIM

Agora repare que no passo:

USE alguma REGRA de seleção para identificar uma peça a ser realocada.

temos que determinar qual peça, dentre as conflitantes, deve ser realocada. Neste trabalho nos preocupamos básicamente com o aspecto de minimização do "makespan" do seqüenciamento. Então é natural que as estratégias de realocação a serem usadas contemplem este aspecto. Um outro critério que contemplaremos, com respeito a realocação de peças, é o de minimizar o atraso de cada peça, supondo que cada uma tem uma data de entrega finita. Neste contexto a escolha de qual peça realocar é de grande importância, o que nos faz descartar, de imediato, a possibilidade de se efetuar esta escolha aleatoriamente.

A seguir apresentamos as regras de seleção que implementamos:

1) Mínimo Tempo de Processamento (**MTP**)

Dentre as peças pertencentes a um dado CONJUNTO-CONFLITO escolha para realocação aquela com menor tempo total de processamento.

2) Mínimo Incremento da Data de Finalização (**MIDF**)

Considere α o valor de maior data de finalização, em relação ao sequenciamento parcial atual, onde, possivelmente, ainda há conflitos a serem resolvidos. Seja ainda γ_k o tempo total de processamento da peça k, já levando-se em conta o uso de uma máquina alternativa e eventual tempo de transporte que se faça necessário. É importante observar que quando nos referimos a uma máquina alternativa queremos dizer "a mais rápida entre as factíveis e disponíveis", e se não houver nenhuma então simplesmente atrasa-se o início da operação conflitante da peça em questão. Então, dentre as peças conflitantes realoca-se a peça δ tal que:

$$\delta = arg\,\min_k \{\gamma_k - \alpha\}$$

3) Mínimo Tempo de Atraso (**MTA**)

Dentre as peças pertencentes a CONJUNTO-CONFLITOS realoque aquele cuja diferença entre a data de entrega a ela atribuída e o tempo de finalização atual (onde, possivelmente, ainda existem conflitos a serem resolvidos) é a menor dentre todas.

É interessante discutir brevemente as regras que acabamos de apresentar.

A regra **MTP** é de avaliação imediata e tenta minimizar a solução a nível local. Já que a peça a ser realocada possivelmente terá uma data de finalização maior do que tinha anteriormente, a peça que menos deveria influir no incremento do "makespan" (na verdade ainda se trata de um limitante inferior do mesmo, pois podem haver conflitos a serem resolvidos) é aquela que é finalizada mais cedo.

A regra **MIDF** é um pouco mais elaborada e verifica o incremento do "makespan" (veja a observação feita no parágrafo anterior) que seria verificado se cada uma das peças fosse realocada, escolhendo então aquela de efeito menos prejudicial (no sentido de "alongar" ainda mais o seqüenciamento atual). É uma regra de escolha mais complexa do que a **MTP** porque tenta ver "um pouco" mais adiante a mudança que ocorre no sequenciamento, em relação ao "makespan" apenas, e essa complexidade tem, óbviamente, seu custo computacional.

Finalmente a regra **MTA** tenta fazer com que a peça realocada seja a menos prejudicada em relação a data de entrega e não se preocupa com o "makespan" do sequenciamento. Se observarmos bem, a regra **MTP** tenta de um certo modo contemplar o aspecto "data de entrega" já que, em geral, as peças que são finalizadas mais cedo são as menos atrasadas.

A seguir ilustraremos com um exemplo cada uma das regras aqui apresentadas.

Considere os seguintes subplanos (planos lineares) exibidos parcialmente:

Peça A:

|...| (A,d,Φ,12,29) |T_j| (A,b,Ψ,31,50) |

Peça B:

|.....|(B,c,Φ,15,26)|(B,b,Φ,26,39)|

Ainda temos as datas de entrega para as peças A e B iguais a 55 e 40 respectivamente.

Podemos verificar um conflito no período [15,26] entre as peças A e B na disputa pela máquina Φ. Suponha que existe uma máquina Γ, ociosa neste periódo, e que $t_{d\Gamma} = 18$ e $t_{c\Gamma} = 13$. Vejamos qual seria o comportamento de cada uma das regras de escolha que apresentamos anteriormente [5].

1) **MTP**

Peça A:

|..| (A,d,Φ,12,29) |T_t| (A,b,Ψ,31,50)	

Peça B:

	.....	(B,c,Γ,15,28)	T_t	(B,b,Φ,30,43)	

2) **MIDF**

Neste caso temos $\alpha = 50$ e k = A, B e

$\gamma_A = 51$, $\gamma_B = 43$ e

$\delta_k = \{1_{(k=A)}, -7_{(k=A)}\}$

logo realocamos a peça B, obtendo:

Peça A:

|..| (A,d,Φ,12,29) |T_t| (A,b,Ψ,31,50)	

Peça B:

	.....	(B,c,Γ,15,28)	T_t	(B,b,Φ,30,43)	

[5] Manteremos a quíntupla seguinte ao conflito, porém a mesma poderia ser realocada j' que após a realocação da operação conflitante procede-se a busca de um novo plano linear ótimo, a partir daquele ponto de conflito (resolvido).

3) **MTA**

A peças A e B estão adiantadas de 5 e 1 unidades de tempo respectivamente. Realocamos então a peça A.

Peça A:

|..| (A,d,Γ,12,30) |T_t| (A,b,Ψ,32,51) |

Peça B:

|.....|(B,c,Φ,15,26)|(B,b,Φ,26,39)|

Suponha agora que não há máquina ociosa disponivel. A única alternativa passa a ser é o adiamento do inicio da operação realocada e considerar este período de ociosidade mais a duração atual da operação como a duração desta operação numa máquina "fantasma".

1) **MTP**

Peça A:

|..| (A,d,Φ,12,29) |T_t| (A,b,Ψ,31,50) |

Peça B:

|.....|.....| (B,c,Φ,29,40) |(B,b,Φ,40,53)|

2) **MIDF**

Neste caso temos $\alpha = 50$ e k = A, B e

$\gamma_A = 64$, $\gamma_B = 53$ e

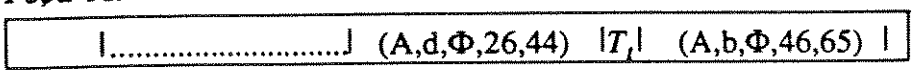
$\delta_k = \{14_{(k=A)}, 3_{(k=A)}\}$

logo realocamos a peça B, obtemos a mesma solução apresentada acima.

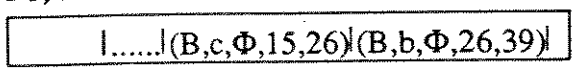
3) **MTA**

A peças A e B estão adiantadas de 5 e 1 unidades de tempo respectivamente. Realocamos então a peça A.

Peça A:



Peça B:

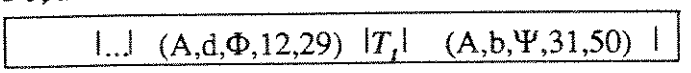


No caso de adiamento óbviamente não há necessidade de se obter um novo plano linear. O caso de haver uma máquina alternativa capaz de realizar apenas uma das operações, restando a outra peça conflitante a única alternativa de adiamento, é uma composição dos casos apresentados acima.

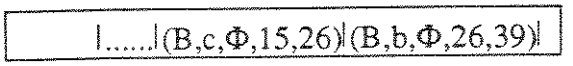
4.3.1.Um exemplo

Considere a situação apresentada acima. Representada pelos seguintes planos lineares:

Peça A:



Peça B:



O algoritmo procederia da seguinte maneira:

INICIO ...

Suponha que as operações correntes das peças A e B são, respectivamente, d e c exibidas acima.

LOOPING #1

T_i = 12 e T_f = 26

CONJUNTO-PECAS = { A, B }

CONJUNTO-MAQUINAS = { Φ }

CONJUNTO-CONFLITO = { A, B }

LOOPING #1 (RESOLVE-CONFLITO)

P = B (regra MTP)

Mudamos (B,c,Φ,15,26) para (B,c,Γ,15,28) e encontramos o novo plano linear a partir deste ponto (representado abaixo)

CONJUNTO-CONFLITO = { 1 }

OPER-CORR(1) = (ordinal da operação d na peça A) + 1

Peça A:

| |..| (A,d,Φ,12,29) |T_i| (A,b,Ψ,31,50) |

Peça B:

| |.....| (B,c,Γ,15,28) |T_i|(B,b,Ψ,30,43)|

LOOPING #1 + 1

T_i = 12 e T_f = 28

CONJUNTO-PECAS = { B }

CONJUNTO-MAQUINAS = { Γ }

CONJUNTO-CONFLITO = { B }

OPER-CORR(2) = (ordinal da operação c na peça B) + 1

e assim sucessivamente até

LOOPING #j

CONJUNTO-CONFLITO = { A, B } e m = G

FIM

4.3.2.Estrutura de dados

Nesta fase da resolução o problema de estrutura de dado a ser implementada é um problema quase que totalmente resolvido. O passo onde a estrutura de dados é mais importante é justamente na elaboração dos planos lineares. Aqui a nossa preocupação foi com a ocupação de memória de computador.

Repare que a cada plano linear encontrado uma árvore de busca (embutida dentro de um grafo) teve que ser construída. Dependendo do tamanho do problema (ordem de M e O) essa árvore pode ocupar uma espaço de memória considerável. Ainda mais, temos N peças logo o espaço de memória usada é da ordem de NMO estados distintos.

Como trabalhamos com espaço de memória restrito usamos a seguinte estratégia. Dado um grafo de busca onde a solução já foi encontrada:

- Copiar o plano linear ótimo (caminhando do nó final ao nó raiz através de seus antecessores) numa lista duplamente ligada e

- Liberar o espaço de memória utilizado pelo grafo de busca atual.

Convém notar que, desta maneira, deixaremos de aproveitar a árvore de busca existente no caso de um replanejamento (ocorrido durante alguma realocação). Nessa situação, poderia ocorrer o caso onde o estado (operação, peça, máquina) a partir do qual deve-se encontrar o novo plano ótimo já tivesse sido explorado, quando necessitaríamos explorar apenas as folhas (nós fronteira) descendentes deste estado. Se levarmos em conta, porém, que esperamos que a função $h(.)$ dirija bem a busca, o compromisso da perda de "agilidade" pode ser compensada com o ganho de memória.

4.3.3.Resultados Computacionais

A seguir apresentamos resultados computacionais obtidos usando-se a estratégia de duas fases para o **PSCFM**. Para a obtenção destes resultados obviamente a fase 1, explanada nas seções anteriores, teve que ser resolvida. Para tanto usamos a função heurística $h(.) = h_3(.)$, ou seja, a heurística míope composta. Poderíamos ter usado uma das funções ε-admissíveis, sem dúvida, porém optamos por $h_3(.)$ (mais eficiente que $h_1(.)$ e $h_2(.)$) por uma razão de simples segurança na obtenção do plano linear ótimo e, assim podemos mensurar devidamente a estratégia que propomos.

Os dados computacionais relativos a tempo de processamento são os mesmos que usamos na seção (**4.1.1**). A data de entrega para cada peça foi gerada como sendo entre uma e duas vezes o valor da data de finalização da mesma. Faremos estudos com um número de operações fixo, $O = 10$; o número de máquinas, $M = 3$, 5 e número de peças, $N = 5$ e 10. Resultando, portanto, em 4 configurações diferentes do **PSCFM**. Cada configuração será resolvida com 5 conjuntos de dados gerados aleatóriamente, o que nos permitirá analisar os resultados de 40 problemas distintos. Obtidos estes resultados analisaremos os seguintes dados, para cada problema resolvido:

1) Média do atraso na finalização peças (tomando como base sua data de entrega),

2) Valor do "makespan" final e

3) Tempo total de CPU consumido.

Cada caso será resolvido usando as três estratégias de realocação apresentadas (**MTP**, **MIDF** e **MTA**), esperamos assim poder concluir qual delas seria mais indicada para uma certa dimensão do problema. Para isso, além de tabelas mostrando os resultados para cada caso resolvido, uma outra tabela comparando o desempenho das regras entre si será apresentada e analisada.

	MTP	MIDF	MTA
"Makespan" [u.t.]			
Conjunto 1	241.00	206.00	256.00
Conjunto 2	199.00	191.00	243.00
Conjunto 3	235.00	203.00	222.00
Conjunto 4	175.00	132.00	160.00
Conjunto 5	253.00	242.00	284.00
Atraso Médio [%]			
Conjunto 1	26.00	18.00	14.00
Conjunto 2	30.00	43.00	78.00
Conjunto 3	32.00	18.00	8.00
Conjunto 4	25.00	18.00	17.00
Conjunto 5	54.00	18.00	21.00
Tempo CPU [seg.]			
Conjunto 1	1.16	0.99	1.01
Conjunto 2	0.78	1.01	1.03
Conjunto 3	1.93	1.39	1.46
Conjunto 4	1.14	0.66	0.56
Conjunto 5	1.12	1.01	1.34

Tabela 4.2.1. 5 peças, 10 operações e 5 máquinas.

	MTP	MIDF	MTA
"Makespan" [u.t.]			
Conjunto 1	371.00	338.00	379.00
Conjunto 2	483.00	399.00	564.00
Conjunto 3	413.00	376.00	507.00
Conjunto 4	478.00	391.00	475.00
Conjunto 5	608.00	526.00	544.00
Atraso Médio [%]			
Conjunto 1	66.00	49.00	25.00
Conjunto 2	54.00	18.00	39.00
Conjunto 3	18.00	51.00	74.00
Conjunto 4	52.00	42.00	41.00
Conjunto 5	122.00	75.00	67.00
Tempo CPU [seg.]			
Conjunto 1	0.77	0.64	0.59
Conjunto 2	0.96	0.94	0.70
Conjunto 3	0.60	0.65	0.85
Conjunto 4	0.73	0.66	0.56
Conjunto 5	0.81	0.68	0.84

Tabela 4.2.1. 5 peças, 10 operações e 3 máquinas.

	MTP	MIDF	MTA
"Makespan" [u.t.]			
Conjunto 1	466.00	364.00	440.00
Conjunto 2	373.00	350.00	473.00
Conjunto 3	390.00	294.00	363.00
Conjunto 4	821.00	572.00	878.00
Conjunto 5	430.00	388.00	402.00
Atraso Médio [%]			
Conjunto 1	85.00	58.00	85.00
Conjunto 2	264.00	134.00	174.00
Conjunto 3	102.00	72.00	50.00
Conjunto 4	231.00	109.00	188.00
Conjunto 5	120.00	81.00	60.00
Tempo CPU [seg.]			
Conjunto 1	4.21	3.71	5.16
Conjunto 2	3.73	2.96	2.82
Conjunto 3	8.58	5.31	5.16
Conjunto 4	2.69	2.37	2.20
Conjunto 5	5.61	4.17	3.23

Tabela 4.2.1. 10 peças, 10 operações e 5 máquinas.

	MTP	MIDF	MTA
"Makespan" [u.t.]			
Conjunto 1	964.00	678.00	911.00
Conjunto 2	625.00	442.00	529.00
Conjunto 3	1709.00	986.00	1571.00
Conjunto 4	1382.00	1211.00	160.00
Conjunto 5	1104.00	713.00	955.00
Atraso Médio [%]			
Conjunto 1	242.00	134.00	177.00
Conjunto 2	234.00	144.00	154.00
Conjunto 3	370.00	181.00	355.00
Conjunto 4	289.00	152.00	197.00
Conjunto 5	296.00	171.00	211.00
Tempo CPU [seg.]			
Conjunto 1	2.92	2.44	2.62
Conjunto 2	2.67	3.05	2.77
Conjunto 3	3.36	3.40	2.75
Conjunto 4	2.80	2.78	2.71
Conjunto 5	3.00	2.72	3.07

Tabela 4.2.1. 10 peças, 10 operações e 3 máquinas.

	MIDF / MTP	MIDF / MTA	MTA / MTP
Makespan			
(5, 3)	13.00	17.00	-6.00
(5, 5)	12.00	16.00	-5.00
(10, 5)	19.00	20.00	-3.00
(10, 3)	34.00	26.00	11.00
Atraso médio			
(5, 3)	-7.00†	-5.00	-31.00†
(5, 5)	25.00	-20.00	11.00
(10, 5)	40.00	4.00	31.00
(10, 3)	45.00	24.00	25.00
Tempo CPU			
(5, 3)	7.00	-4.00	6.00
(5, 5)	13.00	3.00	7.00
(10, 5)	22.00	-3.00	20.00
(10, 3)	2.00	-4.00	5.00

Tabela 4.2.5: Comparando as relações entre as regras. [6]

[5] † Média muito afetada por uma medida.

[6] (i, j) denota o número de peças e máquinas respctivamente e a coluna "A / B" mostra o quanto a regra A foi melhor que a regra B, em termos percentuais.

Usando a tabela (4.2.5) como um resumo das tabelas anteriores podemos tirar as seguintes conclusões.

Com relação ao "makespan" a regra **MIDF** tem desempenho superior à **MTP** e **MTA** sendo que essa superioridade é proporcional a relação (número de peças)/(número de máquinas). Quanto maior a razão peças/máquinas, mais crítica a configuração e maior a probabilidade de existência de conflitos. Ainda observamos que a regra **MTA** perde para a regra **MTP**, em quase todos os casos, tendo vantagem porém, na configuração mais crítica, levando a crer que regras simples perdem em qualidade a medida que a complexidade da configuração aumenta.

Na medida do atraso médio a regra **MIDF** teve bons resultados para configurações mais críticas, enquanto que para configurações mais relaxadas a regra **MTA** teve melhores resultados. A regra **MTP** obteve os piores resultados.

Quando consideramos o tempo de CPU utilizado verificamos que a regra **MTA** é. em geral, ligeiramente superior a **MIDF** e claramente mais rápida que a **MTP**. Em poucos casos a regra **MTP** teve bons resultados.

Assim podemos concluir que no caso geral a regra **MIDF** leva vantagem sobre a **MTA** especialmente em configurações mais críticas. A vantagem da **MTA** sobre a **MIDF** com relação ao tempo de processamento leva a crer que a escolha de realocação foi boa no que concerne a re-decisões e não na minimização do "makespan", já que a regra **MIDF** obtém melhores resultados com respeito a este critério. É interessante observar que a regra **MTA**, cuja intenção era contemplar o critério de minimização do atraso médio, só conseguir seu intuito em configurações pouco críticas. No caso geral a **MIDF** é superior neste aspecto. O modo de geração das datas de entrega pode ter influencia sobre esse fato.

Finalmente a regra **MTP** se mostra vantajosa apenas quando comparada a **MTA** e com o critério de "makespan" implicando ainda assim um maior uso da "CPU" (o que indica que o número de conflitos que tiveram que ser resolvidos foi grande).

Capítulo 5

Modelos de Programação Matemática para o PSCFM.

5.1 Introdução

A estratégia que apresentamos no capítulo anterior, e que é a contribuição principal deste trabalho, pode, sem dúvida alguma, ser usado para o que se propõe, isto é, resolver o **PSCFM**. Porém, surge naturalmente uma pergunta: Quão boa é a estratégia proposta?

Conforme dissemos anteriormente, muitas pesquisas feitas sobre este problema usaram abordagens baseadas em **SBCs** e **SEs**, enfim, em técnicas de **I.A.**. São estratégias interessantes no que concerne à aplicação de uma metodologia em si qualitativa para a resolução de problemas quantitativos, porém elas sofrem de um mesmo "problema", por assim dizer. Não há garantia de quão boas são as soluções obtidas, em termos de medidas de desempenho do sistema, por exemplo: valor do "makespan" e tempo de atraso no acabamento da peça. A única garantia que temos é a de que podemos dispor de uma solução factível ao final da resolução do problema, mas se esta solução é ótima ou está próxima da solução ótima é pergunta que não encontramos respondida nos trabalhos que investigamos.

Dado este quadro, tentamos desenvolver modelos de programação matemática que pudessem resolver o **PSCFM**. A seguir apresentaremos dois modelos, um formulando o **PSCFM** como um problema de programação inteira mista não-linear e um do tipo "branch-and-bound". O primeiro terá apenas sua formulação apresentada e comentada, dado que sua resolução seria extremamente complexa e foge do escopo deste trabalho. Quanto ao segundo serão apresentados alguns resultados computacionais obtidos ao se aplicá-lo para a resolução do **PSCFM**. Ainda usaremos alguns resultados do algoritmo "branch-and-bound" para avaliar o desempenho da estratégia de duas fases.

5.2 Um modelo de programação inteira mista não-linear.

Suponha que:

- Existem N peças, cada uma necessitando n_i (i = 1, 2, ..., N) operações, tendo para isso disponíveis M máquinas;

- As operações da peça i são denotadas pela seqüência ordenada $\{ i_1, i_2, ..., i_{n_i-1}, i_{n_i} \}$ e cada operação pode ser executada em um conjunto S de máquinas, onde $S \neq \{ \}$ e $S \in \{ 1, 2, ..., M \}$

- $P_{i_j,k}$ = tempo de processamento da j-ésima operação da peça i na máquina k. Se a operação não é possivel então $P_{i_j,k} \rightarrow +\infty$;

- t_t = tempo médio necessário para o transporte entre duas máquinas para a realização de duas operações consecutivas.

Suponha ainda que entre as operações e máquinas consideradas, encontram-se aquelas correspondentes ao carregamento e descarregamento das peças.

Considere agora as seguintes variáveis:

- $x_{i_j,r}$ = tempo de finalização da j-ésima operação da peça i na máquina r;

- $\gamma_{i_j,k}$ = 1 se fixamos i_j na máquina k, ou 0 caso contrário.

- $\beta_{i_j,r_q,k}$ = 1 se i_j precede r_q na máquina k, ou 0 caso contrário.

Vejamos agora o modelo:

Queremos minimizar o valor do "makespan". Considere então uma variável real y que seja um limitante superior para o mesmo, ou seja:

R.1) $y \geq x_{i_{n_i},k}$;

para i=1, 2,...,N e k = 1, 2, ..., M

Dada essa restrição a função objetivo fica sendo: *minimizar* y que implica que o valor ótimo de y é o valor ótimo do "makespan".

Sabemos que:

R.2) $\gamma_{i_j,k} = 1$ se e só se $x_{i_j,k} > 0$

R.3) $\gamma_{i_j,k} \equiv 0$ se e só se $x_{i_j,k} \equiv 0$

para i=1, 2,...,N e k = 1, 2, ..., M

O que pode ser traduzido na seguinte dupla de restrições (considere $G \rightarrow +\infty$)

R.4) $x_{i_j,k} \geq 0$;

R.5) $x_{i_j,k} \leq G\gamma_{i_j,k}$;

para i = , 1, 2, $\cdots$, N; $j = 1, \cdots, n_i$; $k = 1, 2, \cdots, M$

Precisamos assegurar que cada par i_j tenha atribuído a si apenas uma máquina. Para isso exigimos que:

R.6) $\sum_{k=1}^{M} \gamma_{i_j,k} = 1$;

para $i = 1, 2, \cdots, N$; $j = 1, 2, \cdots, n_i$

Agora temos que verificar que uma operação i_j só possa ser feita após o término de $i_{(j-1)}$, levando em conta o tempo de transporte necessário caso $i_{(j-1)}$ tenha sido

processada numa máquina distinta daquela designada para i_j. Essa restrição pode ser escrita na seguinte inequação:

R.7) $x_{i_j,l} \geq [\, x_{i_{(j-1)},k} + P_{i_j,l} + t_l(1 - \gamma_{i_{(j-1)},l})]\gamma_{i_j,l}$;

R.8) $x_{i_1,l} \geq P_{i_1,l}\gamma_{i_1,l}$;

para $i = 1, 2, \cdots, N$; $j = i_2, i_3, \cdots, i_{n_i}$; $l, k = 1, 2, \cdots, M$

Há a necessidade de se multiplicar todo o lado direito da inequacão anterior por $\gamma_{i_j,l}$ para se manter a factibilidade do modelo, com relação às restrições (**R.4**)/(**R.5**) e (**R.7**) e (**R.8**). Surge assim a não linearidade do modelo.

Finalmente precisamos ter certeza que dois pares distintos i_j e r_q não sejam designadas simultaneâmente para a mesma máquina k. Repare que não há necessidade de se verificar o caso de duas operações não podem ser realizadas simultâneamente numa mesma peça já que a precedência de operações já foi levada em conta. Necessitamos então das seguintes restrições:

R.9) $x_{i_j,k} - P_{i_j,k} \geq x_{r_q,k} - \beta_{i_j r_q,k} G$

R.10) $x_{r_q,k} - P_{r_q,k} \geq x_{i_j,k} - (1 - \beta_{i_j r_q,k})G$

para $i, r = 1, 2, \cdots, N$; $i \neq r$; $j = i_1, i_2, \cdots, i_{n_i}$; $q = i_1, i_2, \cdots, i_{n_r}$

Assim o modelo proposto fica sendo:

minimizar y

sujeito a:

(R.1), **(R.4)**, **(R.5)**, **(R.6)**, **(R.7)**, **(R.8)**, **(R.9)**, **(R.10)**.

Se considerarmos que $n_i = O$ para $i = 1, 2, \ldots, N$ então a dimensão do modelo apresentado é a detalhada a seguir:

NOM variáveis do tipo $\gamma_{i_j,k}$

NOM variáveis do tipo $x_{i_j,k}$

$(N(N-1)O^2M)/2$ variáveis do tipo $\beta_{i_j,r_q,k}$

repare que se $\beta_{i_j,r_q,k}$ é definida então $\beta_{r_q,i_j,k} = (1 - \beta_{i_j,r_q,k})$ e portanto não precisa ser definida. O número total de variáveis é de $1/2(NOM(4+(N-1)O))$ e o número total de restrições é de $N(1+O(1+N+M(2+O(N-1))))$.

Como dissemos anteriormente este modelo foi apenas desenvolvido já que um estudo computacional do mesmo estaria fora da proposta deste trabalho.

Chamamos ainda a atenção para o fato de que o modelo de programação inteira apresentado por Baker (1974, pp: 207) para o modelo clássico de "job-shop", pode ser considerado como caso especial do modelo apresentado acima. Nesse caso não há necessidade de se usar a variável binária $\gamma_{i_j,k}$, já que cada operação tem apenas uma máquina associda a ela.

5.3 Resolvendo o PSCFM através de um algoritmo do tipo "Branch-and-Bound"

Antes de discutirmos o algoritmo desenvolvido aqui recomendamos ao leitor que leia o Apêndice 1 para uma introdução à filosofia dos algoritmos tipo "Branch-and-Bound", independentemente da aplicação.

Apresentamos nesta seção um algoritmo do tipo "Branch-and-Bound" para a obtenção da solução ótima do **PSCFM**. Este algoritmo usa como base o algoritmo de Chang & Sullivan (1990) que por sua vez é uma generalização do algoritmo de Giffler & Thompson (seção **2.2**).

De modo a diminuir a complexidade do modelo deixaremos de considerar os conflitos entre os recursos de transporte (também incluem-se aqui os de carregamento e descarregamento). No entanto, os tempos necessários para o transporte de peças entre máquinas ainda serão computados.

O interesse de Chang e Sullivan (1990) era o de prover um algoritmo que fornecesse todos os seqüenciamentos ativos para o **PSCFM**. Entre estes encontra-se o ótimo. O nosso interesse nesta etapa é encontrar a solução ótima do **PSCFM**. Assim, acoplamos ao algoritmo de Chang e Sullivan (1990) a técnica de "branch-and-bound". Desse modo obtemos limitantes inferiores para o valor final de cada solução parcial. Diminuindo assim o esforço de geração de seqüenciamentos, já que muitos poderão ser podados durante a busca. Ainda usaremos um limitante superior inicial para o valor da solução do **PSCFM** para auxiliar na eliminação de ramos da árvore geradora de seqüenciamentos ativos. Este limitante inicial será obtido através da estratégia de duas fases.

A seguir, após algumas definições, apresentamos o algoritmo de Chang e Sullivan (1990) acoplado ao "branch-and-bound".

Considere:

PS: conjunto de quíntuplas (i, j, k, t_i^j, t_i^f) já seqüenciadas.

S: conjunto de pares (operação, peça) a serem seqüenciados

σ_{ijk}: início mais cedo para $(i, j, k) \in S$ ser processado

ϕ_{ijk}: fim mais cedo para a finalização de $(i, j, k) \in S$

Algoritmo B & B - **PSCFM**

INICIO

$PS \leftarrow \{\ \}$

$S \leftarrow \{\ (1, j) \mid j = 1, ..., N\ \}$

$LS \leftarrow$ limitante superior para o valor da solução

EXECUTE GERACAO-RECURSIVA(PS_0, S_0)

FIM

Procedimento GERACAO-RECURSIVA(PS, S)

INICIO

$\phi^* \leftarrow \min\{\phi_{ijk} \mid (i,j,k) \in S_0\}$

PARA CADA $(i, j) \in S$

SE $\sigma_{ijk} < \phi^*$ e $t_{ki} < G$

$PS' \leftarrow PS \cup \{(i, j, k, \sigma_{ijk}, \sigma_{ijk}+t_{ik})\}$

$S' \leftarrow S - \{(i, j)\} \cup \{((i + 1, j)\}$

SE $S' = \{\ \}$

SE LI(PS', S') $< LS$

GUARDE PS'

$LS \leftarrow$ LI(PS', S')

SENAO

DESCARTE PS'

SENAO

SE LI(PS', S') < LS

EXECUTE GERACAO-RECURSIVA(PS',S')

SENAO

DESCARTE PS'

FIM

Função LI(PS, S)

INICIO

PARA CADA peça $j \in PS$ e S

$T_{PS}^{j} \leftarrow \phi_{ijk}$, onde i e a última operação executada na peça j e k é a máquina associada

$T_{S}^{j} \leftarrow \sum_{l \in O_r} \min_{m \in M} \{t_{lm}\}$, onde O_r denota o conjunto de operações ainda necessárias na peça j

$LI \leftarrow \max_j \{T_{PS}^{j} + T_{S}^{j}\}$

RETORNE (LI)

FIM

O algoritmo de Chang & Sullivan, difere do de Giffler e Thompson basicamente em um passo do algoritmo. A saber:

SE $\sigma_{ijk} < \phi^*$ e $t_{ki} < G$

que seria, originalmente,

SE $\sigma_{ijk} < \phi^*$ e k = máquina associada a ϕ^*

O teste de "factibilidade" $t_{ik} < G$ não é necessário porque cada operação do conjunto S tem associado apenas uma máquina ("job-shop" tradicional). No caso da máquina associada a ϕ^*, denotada por m^*, não for única escolhemos uma aleatóriamente.

A idéia apresentada por Giffler & Thompson é a de que, pela definição de ϕ^*, nenhuma operação pode ser acabada antes da data ϕ^*. Assim, se a máquina m^* ficasse ociosa até a data ϕ^* então alguma operação poderia ser transladada à esquerda para dentro deste intervalo ocioso caracterizando um seqüenciamento não-ativo.

Um resultado importante para o algoritmo de Giffler & Thompson é o seguinte: "todos os seqüenciamentos ativos são gerados uma única vez" (Lageweg & Outros, 1977).

A generalização de Chang & Sullivan relaxa esta condição, não considerando apenas a máquina m^* associada a ϕ^*, mas todas as máquinas onde as operações que possam ser iniciadas antes de ϕ^* sejam factíveis. É claro que esta relaxação torna a árvore de geração de seqüenciamentos ativos muito mais ramificada. Porém, o algoritmo de Chang & Sullivan não goza da propriedade de não repetição de seqüenciamentos ativos. De fato o algoritmo proposto pode gerar seqüenciamentos ativos repetidos (Apêndice 3).

Assim nos surgiu a idéia de usar o algoritmo original de Giffler & Thompson, generalizando a possibilidade de várias máquinas processarem uma mesma operação e mantendo a observância com relação a máquina m^*, como uma heurística para resolver o **PSCFM**.

O conjunto de seqüenciamentos gerados será bastante menor, porém possivelmente não conterá a solução ótima. Investigaremos então o compromisso (velocidade de processamento)/(qualidade da solução).

A seguir demonstramos resultados computacionais obtidos com o uso dos algoritmos acima comentados.

5.3.1.Resultados Computacionais

Dado o caráter fortemente exponencial e o limitante inferior pouco poderoso, no que concerne à eliminação de ramos da árvore de busca, nos limitaremos a investigar problemas de pequeno porte.

Fixamos o número de operações , O, em 2 e usaremos $M = 2$ e 3, e $N = 3$ e 4. Para cada configuração resolveremos 3 casos gerados aleatóriamente. Cada caso será resolvido pelos algoritmos de Chang & Sullivan (**ACS**), Giffler & Thompson "generalizado" (**AGT**) e a estratégia de duas fases (**EDF**). Na estratégia de duas fases usaremos a regra **MIDF** para a escolha da peça a ser realocada.

Usaremos como medida de comparação os valores de "makespan" gerado e o tempo de CPU necessário (limitado em no máximo 60 segundos).

As tabelas apresentadas a seguir estão em ordem crescente com relação à razão (número de peças)/(número de màquinas). Na tabela (5.2.5) a coluna **A / B** mostra quanto **A** foi pior que **B** em termos de "makespan" e quanto **B** foi mais lento que **A** comparando os tempos de CPU.

	EDF	ACS	AGT
"Makespan" [u.t.]			
Conjunto 1	128.00	66.00	68.00
Conjunto 2	94.00	72.00	72.00
Conjunto 3	91.00	39.00	41.00
Conjunto 4	70.00	46.00	46.00
CPU [seg.]			
Conjunto 1	0.17	1.12	0.27
Conjunto 2	0.15	1.82	0.16
Conjunto 3	0.13	1.06	0.28
Conjunto 4	0.13	0.37	0.12

Tabela 5.2.1.: 3 peças e 3 maáquinas.

	EDF	ACS	AGT
"Makespan" [u.t.]			
Conjunto 1	131.00	--	93.00
Conjunto 2	119.00	94.00	96.00
Conjunto 3	129.00	53.00	60.00
Conjunto 4	118.00	51.00	56.00
CPU [seg.]			
Conjunto 1	0.24	--	4.06
Conjunto 2	0.24	17.32	0.86
Conjunto 3	0.18	45.16	2.49
Conjunto 4	0.18	4.47	1.18

Tabela 5.2.2: 4 peças e 3 maáquinas.

	EDF	ACS	AGT
"Makespan" [u.t.]			
Conjunto 1	95.00	50.00	50.00
Conjunto 2	58.00	34.00	35.00
Conjunto 3	37.00	32.00	32.00
Conjunto 4	66.00	65.00	65.00
CPU [seg.]			
Conjunto 1	0.05	0.14	0.04
Conjunto 2	0.06	0.10	0.04
Conjunto 3	0.05	0.07	0.03
Conjunto 4	0.04	0.30	0.03

Tabela 5.2.3: 3 peças e 2 maáquinas.

	EDF	ACS	AGT
"Makespan" [u.t.]			
Conjunto 1	109.00	82.00	87.00
Conjunto 2	95.00	93.00	93.00
Conjunto 3	105.00	103.00	103.00
Conjunto 4	81.00	74.00	78.00
CPU [seg.]			
Conjunto 1	0.15	9.81	3.87
Conjunto 2	0.20	1.19	0.38
Conjunto 3	0.14	0.48	0.39
Conjunto 4	0.14	50.01	7.02

Tabela 5.2.4: 4 peças e 2 maáquinas.

Observando a tabela seguir como um sumário das anteriores vemos que a perda no valor da solução obtida pelo **AGT** foi muito pequena em todos os problemas resolvidos. Recorde que o **ACS** fornece a solução ótima. Se comparamos quanto o **ACS** foi mais lento que o **AGT** concluimos que, pelo menos para este porte de problema, fica mais apropriado o uso do **AGT**. O uso de **ACS** só se justificaria se a obtenção da solução ótima fosse realmente necessária.

A **EDF** não obteve resultados muito bons se compararmos a média dos resultados, além disso a variação do desempenho foi muito grande, variando de 1% a 130% de perda. Ainda nos parece que , assim como no caso do **AGT**, a **EDF** melhora seu desempenho a medida que o fator (número de peças)/(número de máquinas) cresce. Os tempos de CPU necessários pela **EDF** são muito menores que aqueles exigidos pelo **ACS**. Essa diferença era esperada, dada a diferença de estratégia usada nos dois casos.

	EDF / ACS	**AGT / ACS**
"Makespan" [%]		
(3, 3)	78.00	2.00
(4, 3)	100.00	0.08
(3, 2)	44.00	0.00
(4, 2)	12.00	0.03
CPU [%]		
(3, 3)	643.00	460.00
(4, 3)	11496.00	1306.00
(3, 2)	234.00	358.00
(4, 2)	10700.00	251.00

Tabela 5.2.5.: Comparando os algoritmos.

Capítulo 6

Conclusões e comentários

6.1. Sobre o plano linear

Na resolução do plano linear ficou clara a superioridade do WA^* sobre o A_ε^*. Porém, deve-se ressaltar que uma função $h_f(.)$ diferente do que usamos pode mudar este quadro; se esta for "melhor informado" que a usada aqui a busca se torna mais "inteligente" portanto mais eficiente. O processamento mais lento do A_ε^* deve-se também a implementação feita. Uma especialização desta contribuiria para melhorar os resultados obtidos. Note que somente aumentar a velocidade do A_ε^* não é suficiente, já que com relação ao valor da solução encontrada, apresentou em média resultados bastante inferiores aqueles do WA^*, é necessário torná-la mais "informada".

Ainda nesse quadro, o uso do WA^* se mostra promissor para a resolução de problemas de maior porte, já que nos casos estudados houve ganho de velocidade com pequena perda (muitas vezes nenhuma) de "otimalidade".

Sugerimos como pesquisa futura o estudo e implementação de outras estratégias baseadas no A^*, por exemplo: os algoritmos IDA^* (Korf, 1985) e A_ε (Ghallab & Allard, 1983) onde a preocupação principal é propiciar economia na memória necessária. No caso do IDA^* o algoritmo é admissível e ótimo segundo o critério de memória necessária que é da ordem de $O(s)$ onde s é o número de mudanças necessárias para se chegar do estado inicial ao final. O A_ε é um algoritmo ε-admissível cuja estratégia é de tentar sempre que possível permanecer no caminho atual enquanto este for "aceitável"; o ponto principal é quando um estado é aceitável ou quando torná-lo (num certo sentido) aceitável.

6.2. Sobre o plano não linear

O principal ponto no plano não linear é a possibilidade de utilizá-lo para a obtenção de uma solução factível para o **PSCFM** e que contemple o critério de minimização do "makespan". Uma outra possível função da estratégia de duas fases (**EDF**) apresentada aqui seria a resolução do **PSCFM** em tempo "quase"-real. Dadas as necessidades iniciais de seqüenciamento determina-se um plano "a priori", e a medida que novas peças chegam à célula estas poderiam ser introduzidas no seqüenciamento paralelamente a sua execução. No caso de quebra de máquinas um re-seqüenciamento é possível de ser feito rápidamente, basta interpretar esta quebra como o surgimento de um conflito onde uma peça fictícia permanece alocada à máquina defeituosa até que esta seja consertada. Tanto no caso anterior, como no de ser necessário rapidamente um seqüenciamento factível, o plano não linear pode ser encontrado usando-se uma busca ε-admissível durante a busca dos planos lineares, quanto maior a "urgência" mais relaxada a otimilidade (maior o valor de ε).

Uma extensão deste trabalho poderia ser feita no refinamento da regra **MIDF** apresentada aqui seria a de não apenas calcular o efeito da realocação isoladamente, mas sim o efeito da realocação através do novo plano linear a ser encontrado para cada peça candidata à realocação. Teríamos assim uma informação mais precisa sobre o efeito da realocação; em contrapartida, a obtenção desta informação tornaria a solução mais lenta de ser obtida.

6.3. Sobre o modelo de "branch-and-bound

A relaxação do algoritmo de Giffler & Thompson (Baker, 1974, p:189) que implementamos mostrou bons resultados, tornando possível tratar melhor (em termos computacionais) problemas que o algoritmo de Chang & Sullivan (1990). Não pudemos usar os resultados do "branch-and-bound" para avaliar a estratégia de duas fases de uma maneira mais eficiente, já que a aplicação da **EDF** se torna mais indicada a medida em que a configuração do **PSCFM** cresce e nesses casos o "branch-and-bound" se torna computacionalmente cada vez mais difícil de se resolver. Um ponto negativo a ser observado no algoritmo de Chang & Sullivan é a geração de seqüenciamentos ativos repetidos que torna a busca redundante. Uma linha de pesquisa a se seguir seria a de se investigar uma maneira de se evitar a duplicação de

seqüenciamentos parciais já gerados.

Seria altamente desejável que limitantes mais fortes pudessem ser usados durante a busca. Neste contexto sugerimos o uso da EDF para o fornecemento de limitantes superiores factíveis para cada nó da busca. Nesse caso a EDF seria empregada de modo a "completar" a solução parcial disponível em cada nó, fornecendo à busca mais informações, tornando-a portanto, mais "informada".

Apêndice 1

A Técnica de "Branch-and-Bound"

Muitos problemas de otimização combinatória, como o problema do caixeiro viajante, o problema de seqüenciamento de peças job-shop, problemas de programação inteira, inteira-mista e inteira-zero-um poderiam ser resolvidos trivialmente, bastando avaliar todas as soluções possíveis e escolher a melhor delas, considerando o caso de minimização da função objetivo, a melhor seria aquela de menor valor. A rigor todos os problemas poderiam ser resolvidos por enumeração explícita, como dito acima, mas alguns modelos, de porte apenas médio, dos problemas exemplificados acima, exigiriam um tempo de computação impossível de se conceber no nosso mundo real.

Fica claro assim que se tornam necessárias técnicas que reduzam a cardinalidade do conjunto de soluções factíveis, onde se efetua a busca pela solução ótima. Essas técnicas tornam possíveis a enumeração implícita do conjunto de soluções factíveis. A técnica que descrevemos aqui é conhecida como "Branch-and-Bound" e é bastante empregada principalmente em conjunto com técnicas analíticas de programação matemática, como, por exemplo, no caso de programação inteira (Garfinkel & Nemhauser, 1972).

A estratégia do algorítmo é bastante intuitiva e consiste de dois procedimentos principais:

1) Ramificar : particionar o problema atual em subproblemas de resolução mais simples;

2) Sondar : eliminar subproblemas candidatos infrutíferos

Fica claro que um algorítmo que se baseie nestes dois procedimentos requer que sua resolução possa ser baseada numa estrutura de árvore, onde cada nó-pai tem nos nós-filhos subproblemas associados a ele.

A idéia de se particionar ("branching") o problema pai em subproblemas é a de se obter subproblemas mutuamente exclusivos; parcialmente resolvidos em relação ao original e principalmente mais fáceis de solucionar que o original.

Quanto a sondagem de subproblemas, isto é, eliminação dos mesmos, podemos comentar o seguinte. Suponha que exista uma solução factível disponível. Só vale a pena persistir na resolução de um subproblema se há pelo menos alguma possibilidade de se obter uma solução melhor que a atualmente disponível, se esta existir. Suponha que há uma solução factível. Suponha ainda que é possível obter limitantes inferiores ("bounds") para o valor da solução ótima de cada subproblema. Daí podemos verificar o limitante inferior do subproblema atual, se este for maior que o valor da solução disponível significa que, com certeza, o subproblema atual não levará a uma solução melhor que a já disponível e podemos descartá-lo. Caso contrário, não podemos descartar este subproblema porque há a possibilidade de poder encontrar uma solução melhor que a disponível e, quando isto ocorrer, a nova solução substituirá aquela disponível no momento. Uma outra possibilidade de sondagem de subproblemas se dá quando os subproblemas são infactíves ou tem o conjunto solução vazio.

A seguir apresentaremos uma primeira versão do algorítmo "branch-and-bound":

Usamos as seguintes definições:

S, P :subproblema viável a ser investigado;.

L: lista ordenada de subproblemas viáveis, isto é, que ainda podem ser ramificados;

Z: solução incumbente, ou seja, o valor da melhor solução encontrada até o momento;

Algorítmo "BRANCH-AND-BOUND"

INICIO

P ← problema original

L ← {P}

Z ← $+\infty$

LOOPING

SE L = { }

FIM

SENAO

RETIRAR um subproblema S da lista L de acordo com algum critério

CRIAR subproblemas a partir de S de acordo com algum critério e adicioná-los a L

ELIMINAR da lista L todos os subproblemas que possam ser sondados

RESOLVER S

SE a solução de S é solução de P

SE o valor da solução de $S < Z$

GUARDE a solução encontrada de S

Z ← valor da solução de S

SENAO

DESCARTE o subproblema S

LOOPING

SENAO

LOOPING

FIM

Vale a pena comentar o passo:

RETIRAR um subproblema S da lista L de acordo com algum critério.

Alguns critérios de ordenamento[1] da lista L poderiam ser, entre outros:

- Em ordem crescente de limitante inferior. Se o limitante inferior for idêntico ao valor ótimo da solução então esta regra implica no menor número de ramificações possíveis;

- Em ordem decrescente de profundidade, definimos aqui profundidade como sendo o número de subproblemas antecessores ao atual. Nesse caso quanto maior a profundidade de um subproblema mais próximos estamos de uma solução factível além da solução ser mais fácil de ser encontrada;

- Em ordem crescente da diferença entre os valores dos limitantes superior (se este puder ser encontrado) e inferior do subproblema atual. Dessa forma investigamos primeiramente aqueles problemas cujos limites levam a acreditar, dependendo da proximidade entre os limitantes, a disponibilidade da solução ótima do subproblema (veja o comentário acima), entre outras.

Agora veremos alguns melhoramentos que podem ser feitos a partir dessa primeira versão do algoritmo apresentado:

---

[1] Consequentemente o critério de retirada seria o de retirar o primeiro subproblema da lista.

- Iniciar Z com o valor de alguma solução factível, que possa ser fácil e rapidamente obtida (isto evita a necessidade de ter que se primeiro obter uma solução pelo algorítmo para somente após isso podermos descartar subproblemas);

- Obter para cada subproblema um limitante superior para o valor da solução através de um procedimento heurístico computacionalmente rápido. Assim um subproblema com limitante superior melhor que Z poderia substituí-lo e passar a representar a melhor solução disponível até o momento;

- Acoplar ao algorítmo a verificação de propriedades de dominância (quando estas forem disponíveis) para descarte de subproblemas, e, entre outros

- Usar estruturas de dados eficientes e/ou especializadas para a resolução computacional do problema.

Comentando rápidamente uma aplicação da técnica aqui descrita podemos usar como exemplo o caso da resolução de problemas de programação inteira. Nesse caso o limitante inferior poderia ser calculado resolvendo o subproblema atual, com as restrições de integralidade relaxadas através de um algoritmo de programação linear. Os subproblemas seriam gerados a partir do "problema-pai", adicionando-se restrições que assegurassem a integralidade de pelo menos uma variável do problema original.

Apêndice 2

Introdução aos Sistemas Especialistas

Um Sistema Especilista (**SE**) é um programa de computador que se comporta como um especialista em algum domínio, geralmente bastante restrito (Bratko, 1987). Um **SE** é capaz de solucionar problemas, desde que colocados de uma maneira coerente com o seu desenho, podendo ainda explicar o raciocínio utilizado para chegar a conclusão fornecida e ser capaz de lidar com incertezas, porém aqui não trataremos deste último caso.

Um **SE** é composto basicamente por dois módulos que apresentamos a seguir:

Base de Conhecimento: (**BC**), contém o conhecimento específico do domínio de aplicação, e por conhecimento pode-se entender: regras que descrevem relações entre fatos e/ou estados do domínio e fatos fornecidos ou inferidos (através das regras citadas) pelo próprio **SE**.

Estratégia de Controle: (**EC**), é responsável pela administração do conhecimento armazenando na **BC**. Como esse conhecimento é manipulado depende da forma usada para representá-lo, isto é, do Sistema de Representação do Conhecimento (**SRC**).

Existe uma série de formas de se representar o conhecimento e até hoje muita pesquisa tem sido feita nesta área. Algumas formas que podemos citar são: "Frames", Redes Semânticas e Regras de Produção. Vamos nos concentrar aqui nas Regras de Produção. Outras formas de representação podem ser encontradas no trabalho de Nilsson (1982). Regras de Produção são regras da forma:

SE (antecedente) ENTAO (consequente)

Esse tipo de representação tem se mostrado muito vantajoso pelas seguintes características:

- modularidade (independência de uma regra em relação a outras);
- transparência (facilidade de compreenção pelo usuário) e
- expansibilidade (novas regras podem ser adicionadas a qualquer momento).

A **EC** tem por objetivo comparar o estado atual do sistema, usando a **BC**, com os antecedentes das regras do **SRC**, supondo o uso de regras de produção, e, dentre aquelas cujo antecedente é satisfeito, selecionar uma para ser "disparada". O efeito deste "disparo" é o de mudar o estado atual do sistema para alcançar um determinado objetivo. São usados basicamente dois tipos de estratégia de controle, são elas:

Dirigida pelo objetivo: Também conhecido como encadeamento para trás ("backward chaining"). A partir do objetivo procura-se uma regra que o tenha como consequente. Toma-se o antecedente desta regra como sub-objetivo e repete-se o raciocínio até que se chegue a uma regra cujo antecedente é igual ao estado inicial do sistema. Para se recuperar a sequência de regras que compõe a solução basta aplicar sucessivamente as regras encontradas na ordem inversa, isto é, partindo da última regra encontrada (que é aplicável ao estado inicial) até a primeira (que leva ao estado objetivo);

Dirigida por dados: Ou encadeamento a frente ("forward chaining"). A partir do estado atual do sistema tenta-se encontrar regras aplicáveis que levem o sistema a satisfazer a condição objetivo. A solução é obtida à medida em que se avança na busca.

A partir disso podemos concluir que é de extrema importância a escolha de como se representar o conhecimento e como procurar por uma solução, sendo que na maioria

dos casos uma solução factível não é suficiente e há a necessidade de se otimizar algum critério. Nesse sentido alguns pesquisadores têm estudado a possibilidade de se aumentar a "inteligência" dessas estratégias, que podem ser vistas como uma busca em um grafo dirigido. Um enfoque já bastante investigado e promissor é o uso de funções heurísticas que avaliem quão promissor é um caminho do grafo que esta sendo explorado.

Podemos citar os seguintes casos como exemplos de **SE** aplicados em ambientes de manufatura:

- A administração dos investimentos nos vários setores de uma fábrica (Sullivan & LeClair, 1985);

- **CAPP** (Computer Aided Process Planning), agindo através da interação **CAD** (Computer Aided Design), que pode "reconhecer" a peça e **CAM** (Computer Aided Manufacturing), que pode executar a sequência de operações determinada pelo **SBC** (Matsushima & outros, 1982 e Phillips & Monleeswaran, 1985);

- Sequënciamento da produção, que consiste em decidir quando (em que instante) e em que máquina (quando houver possibilidade de escolha) processar as operações devidas em cada peça (determinadas pelo **CAPP**), de maneira a otimizar algum critério (Subramanyan & Askin, 1986).

Apêndice 3

Um Exemplo para o Algoritmo de Chang & Sullivan

A notação que usaremos aqui será a mesma apresentada no seção (**5.2**). A exceção será feita aos seqüenciamentos parciais a serem adicionados ao já existente. Estes serão denotados por (i, j, k, t_i^j, t_i^j) que representam respectivamente: a operação, a peça, a máquina utilizada, o tempo de início e fim de processamento. Para representarmos todas as máquinas factíveis para o par (i, j) usaremos $(i, j, .)$. A figura (A.3.1) ilustra a busca abaixo explicitada.

Suponha os seguintes dados:

$$T_{mo} = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \quad \text{e} \quad S_{po} = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$$

Executando o algoritmo teríamos:

(0) $PS_0 = \{\ \}$

$S_0 = \{ (1,1,.) , (1,2,.) \}$

$\phi^* = \min \{ \phi_{1,1,1}, \phi_{1,1,2}, \phi_{1,2,1}, \phi_{1,2,2} \} = \min \{1, 2, 2, 4\} = 1$

$\sigma_{1,1,1} = \sigma_{1,1,2} = \sigma_{1,2,1} = \sigma_{1,1,2} = 0 < \phi^*$

Assim todos os sequenciamentos candidatos podem compor novas seqüências parciais. Porém, concentraremos nossa atenção nas triplas (1,2,1) e (1,1,2), explorando assim os nós (1) e (2) da busca.

(1) $PS_1 = \{ (1,2,1,0,2) \}$

$S_1 = \{ (1,1,.) , (2,2,.) \}$

$\phi^* = \min \{ 3, 2, 3, 4 \}$

$\sigma_{1,1,2} = 0 < \phi^*$ gerando o nó (3)

(2) PS_2 = { (1,1,2,0,2) }

S_2 = (2,1,.) , (1,2,.)

ϕ^* = min { 4,6,2,3 } = 2

$\sigma_{1,2,2} = 0 < \phi^*$ gerando o nó (4)

(3) PS_3 = { (1,2,1,0,2), (1,1,2,0,2) }

S_3 = (2,1,.) , (2,2,.)

ϕ^* = min { 4,6,3,4 } = 3

$\sigma_{2,1,2} = 2 < \phi^*$ gerando o nó (5)

(4) PS_4 = { (1,2,1,0,2), (1,1,2,0,2) }

S_2 = (2,1,.) , (2,2,.)

Note que o nó (4) é uma duplicação do nó (3).

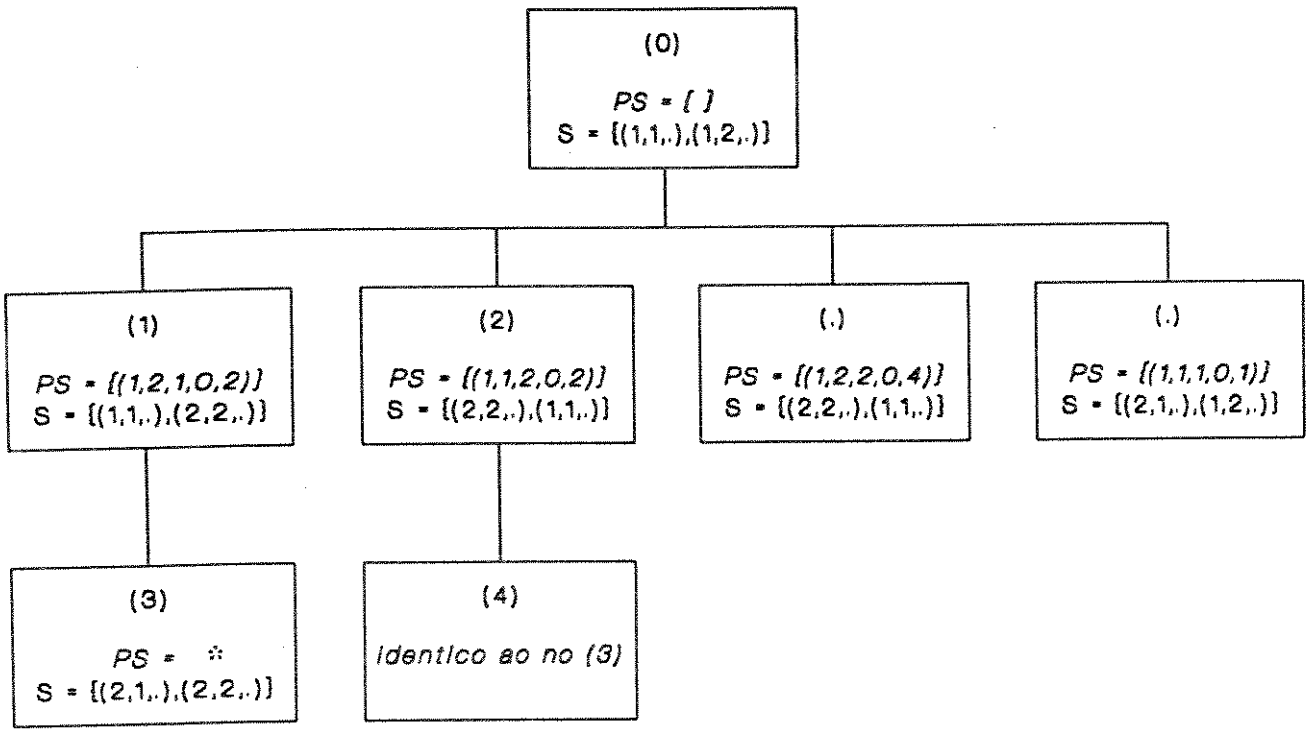
Assim todo o esforço de se explorar o nó (3), e seus descendentes, será repetido ao se explorar o nó (4), e seus descendentes. Isto nos leva a gerar seqüenciamentos parciais ativos redundantes. Neste ponto o algoritmo deveria podar um dos nós redundantes, contudo, isto não ocorre.

Se continuássemos a explorar um dos nós ((3) ou (4)) chegaríamos, dentre outros, ao seguinte seqüenciamento:

(*) $PS_{(*)}$ = { (1,2,1,0,2), (1,1,2,0,2), (2,1,1,2,4), (2,2,2,2,4) }

$S_{(*)}$ = { }

que é a solução ótima do sequenciamento.



* = {(1,2,1,0,2),(1,1,2,0,2)}

FIGURA A.3.1.

Referências

Agostinho, L.O., 1989, *Curso: manufatura integrada por computador - Notas de Aulas*, não publicado, Campinas.

Baker, K.R., 1974, *Introduction to sequencing and scheduling*, John Wiley, New York.

Ben-Arieh, D. & Moodie, C.L., 1987, Knowledge based routing and sequencing for discrete part production, *J. of Manufact. Systems*, 6 (4), pp: 287-297.

Bortolon, S. & Garcia, A.S., 1989, Algoritmos de caminhos mínimos aplicados ao roteamento telefônico, *Anais do XXII Simpósio Brasileiro de Pesquisa Operacional*, pp: 443-450, Fortaleza.

Bratko, I., 1987, *PROLOG programming for artificial intelligence*, Addison-Wesley, Reading.

Bruno, G., Elia, A. & Laface, P., 1986, A rule-based system to schedule production, *Computer*, 19 (7), pp: 32-39.

Chang, Y. & Sullivan, R.S., 1990, Schedule generation in a dynamic job-shop, *Int. J. Prod. Res.*, 28 (1), pp: 65-74.

French, S., 1982, *Sequencing and scheduling - an introduction to the mathematics of the job-shop*, Ellis Horwood, Chichester.

Fox. M.S., 1983, Constraint-directed search: a case study of job-shop scheduling, *Tese de doutoramento*, Pittsburgh.

Garfinkel R.S. & Nemhauser, G.L., 1972, *Integer programming*, John Wiley, New York.

Ghallab, M. & Allard, D.G., 1983, A_ε - an efficient near admissible heuristic search algorithm, *Proceedings of the IJCAI-83 vol.2*, pp: 789-791, Karlsruhe.

Grant, T.J., 1986, Lessons from O.R. to A.I. - a scheduling case study, *J. Opl. Res. Soc.*, 37 (1), pp: 41-57.

Graves, S.C., 1981, A review of production scheduling, *Op. Res.*, 2 (4), pp: 646-675.

Hart, P.E., Nilsson, N.J. & Raphael, B., 1968, A formal basis for the heuristic determination of minimum cost paths, *IEEE Trans. of Sys. Sc. and Cyb.*, 4 (2), pp: 100-107.

Imai, H. & Iri, M., 1984, Pratical efficiencies of existing shortest-path algorithms and a new bucket algorithm, *J. of Opns Res. Soc. of Japan", 27 (1), pp: 43-57*

Korf, R.E., Depth-first iterative deepening: an optimal admissible search algorithm, *Artif. Intell.*, 27 (2), pp: 97-109.

LaFortune, M.F., Hosios, A.J & Rousseau, J.M., 1981, Scheduling medical students to teaching hospitals, *Euro. J. Opn. Res.*, 8 (1), pp: 24-30.

Lageweg, B.L., Lenstra, J.K., & Rinnoy Kan, A.H.G., 1977, Job-shop scheduling by implicit enumeration, *Manag. Sc.*, 24 (4), pp: 441-450.

Lawler, E.L., Lenstra, J.K. & Rinnoy Kan, A.H.G., 1982, Recent development in deterministic sequencing and scheduling - a survey, *Deterministic and Stochastic Scheduling*, Demps, M.A.H. et al (eds.), pp: 35-73.

LeCocq, P. et al, 1988, An expert systems application to increase the flexiblity and the efficiency of real-time FMS controllers, *Preprint FMS21*, Bruxelas.

LeCocq, P. & Guiot, T., 1988, Expert systems for production planning and scheduling, *Preprint FMS23*, Bruxelas.

Matsushima, K., Okada K. & Sata, T., 1982, The integration of CAD and CAM by application of artificial intelligence techniques, *Annals of the CIRP*, vol 31/1/1982, pp: 329-332.

Minieka, E., 1978, *Optimization algorithms for networks and graphs*, Marcel Dekker, New York.

Nascimento, M.A., 1989, Using the A^* algorithm to solve small sequence dependent scheduling problems, *Anais do XXII Simpósio Brasileiro de Pesquisa Operacional*, pp: 43-47, Fortaleza.

Nascimento, M.A. & Armentano, V.A., Seqüenciamento de peças em uma célula flexível de manufatura através de busca em grafos, *Anais do VIII Congresso Brasileiro de Automática*, pp: 1148-1151, Belém.

Nilsson, N., 1982, *Principles of artificial intelligence*, Tioga, Palo Alto.

Ow, P., 1986, Experiments in knowledge-based scheduling, *Carnegie-Mellon Technical Report*, Pittsburgh.

Pearl, J., 1984, *Heuristics: intelligent search strategies for computer problem solving*, Addison-Wesley, Reading.

Philips, R.H. & Mouleeswaran, C.B., 1985, A knowledge-based approach to generative process planning, *Autofact'85 Conference Proceedings*, pp: 10.1-10.15, Detroit.

Ribeiro, C.C., 1980, Planejamento da expansão de sistemas de geração através de algoritimos de busca em grafos, *Anais do XIII Simpósio Brasileiro de Pesquisa Operacional*, pp: 222-235. Rio de Janeiro.

Schonberger, R.J., 1984, *Técnicas industriais japonesas*, Livraria Pioneira, São Paulo.

Sério, L.C., 1990, *Tecnologia de grupo no planejamento de um sistema produtivo*, Ícone, São Paulo.

Subramanyan, S. & Askin, R.G., 1986, An expert system approach to scheduling in flexible manufacturing systems, *Flexible manufacturing systems: methods and studies*, Kusiak, A. (ed.), Elsevier, New York.

Sullivan, W.G. & LeClair, S.R., 1985, Justification of flexible manufacturing systems using expert system technology, *Autofact'85 Conference Proceedings*, pp: 7.1-7.13, Detroit.

Szwarcfiter, J.L. & Markenzon, L., 1989, Estrutura de Dados e Algoritmos, *I Encontro de algorítmos e otimização*, Belo Horizonte.