UNIVAP - Universidade do Vale do Paraíba

FCC - Faculdade de Ciência da Computação

Curso de Ciência da Computação

Aplicação De Métodos Heurísticos No Problema De Escalonamento Da Produção Em Ambientes De Manufatura Do Tipo Job Shop

por

Jefferson Alessandro Rodriguês Alves

Prof. MSc. Jackson Paul Matsuura

Orientador Acadêmico

Prof. Dr. Márcio Magini

Coordenador da Disciplina de TG

São José dos Campos - SP, Junho de 2004

Relatório do Trabalho de Graduação Apresentado à Faculdade de Ciência da Computação da Universidade do Vale do Paraíba como parte dos requisitos para obtenção do Título de Bacharel em Ciência da Computação.

Aplicação De Métodos Heurísticos No Problema De Escalonamento Da Produção Em Ambientes De Manufatura Do Tipo Job Shop

por
Jefferson Alessandro Rodriguês Alves

Relatório aprovado em versão final pelos abaixo assinados:

_____

Prof. MSc. Jackson Paul Matsuura
Orientador Acadêmico

_____

Prof. Dr. Márcio Magini
Coordenador da Disciplina de TG

São José dos Campos - SP, Junho de 2004

Aplicação De Métodos Heurísticos No Problema De Escalonamento Da Produção Em Ambientes De Manufatura Do Tipo Job Shop

Jefferson Alessandro Rodriguês Alves

Banca Examinadora:

Prof. MSc. Jackson Paul Matsuura (Orientador Acadêmico)

Prof. Dr. Márcio Magini (Coordenador da Disciplina de TG)

Prof. MSc. Moacir de Sousa Prado

UNIVAP, Junho de 2004

Dedicatória

Dedico este trabalho à minha noiva Alessandra que sempre serviu-me de exemplo pela sua garra, coragem, perseverança e, principalmente, foco nos objetivos.

Agradecimentos

Agradeço à todas aqueles que,
de alguma forma, contribuíram para a
realização deste trabalho, particularmente aos
meus orientadores, seja pela confiança irrestrita depositada
seja pela elucidação das dúvidas surgida e apoio nos momentos difíceis .

Resumo

Os Sistemas de Manufatura são complexos e sensíveis às alterações tanto externas como internas, tendo influência direta em atividades de Planejamento, Programação e Controle da Produção (PPCP) e, conseqüentemente, na empresa como um todo.Uma das principais atividades do PPCP é o Escalonamento da Produção que objetiva uma eficaz designação dos recursos produtivos (máquinas, homens...) visando a execução das atividades que compõe um projeto.

O problema de “Job Shop Scheduling” (JSP) é considerado um problema combinatorial do tipo NP-completo.(enumeração explícita de todas as alternativas possíveis para garantir a solução ótima).

Por se tratar de um problema de grande porte costuma-se sacrificar a obtenção de uma solução ótima mediante a aplicação de métodos heurísticos resultando numa solução “subótima”, com um tempo computacional aceitável.

Neste trabalho é proposto e utilizado um método heurístico para solucionar este problema objetivando um resultado com as características citadas acima.

Por ser considerado um problema clássico e também considerando o enfoque acadêmico, não houve necessidade de se utilizar um caso prático para aplicação do método, no entanto, sua aplicabilidade poderia ser feita com relativa facilidade, apenas adequando-se a alguns aspectos.

A implementação do método procurou considerar não somente aspectos inerentes ao problema, do ponto de vista computacional, mas também aspectos relacionados ao domínio da aplicação, no caso, mais especificamente os chamados recursos gargalos que possuem um papel importante num ambiente fabril.

Para se aferir a qualidade dos resultados, foi realizada uma comparação entre os resultados do método proposto (Busca Heurística), e os resultados obtidos a partir de uma análise de todas as possibilidades de solução para o problema (Busca Exaustiva), comprovando-se a eficiência do método para os problemas desta natureza.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

IA	:	**I**nteligência **A**rtificial é um ramo da ciência da computação que a partir da exploração do conhecimento humano e através do emprego de técnicas visa solucionar problemas diversos de forma otimizante.
(n/m) JSP	:	**J**ob **S**hop Scheduling **P**roblem corresponde a um problema de alocação de um conjunto de **n** tarefas em **m** recursos.
Job	:	Denominação dada a uma tarefa, podendo ser uma ordem de fabricação de um produto ou mesmo uma atividade de um projeto.
Makespan	:	Corresponde a uma métrica de qualidade da alocação de tarefas num caso (n,m) JSP, sendo o total de tempo necessário a execução de todas as tarefas.

1. Introdução

1.1. Motivação

O emprego de técnicas heurísticas na solução de problemas dos mais variados tipos, cresce a cada dia nas mais diversas áreas.

Tais técnicas ganharam espaço no mercado na mesma proporção em que novas tecnologias foram sendo criadas e à medida que empresas e universidades precisaram de sistemas inteligentes de solução de problemas que envolviam um número intratável de variáveis de forma analítica. Por se tratar de uma área relativamente nova e muitas vezes de grande complexidade, ainda há muito ainda a ser explorado e desenvolvido, no entanto, sua aceitabilidade vem crescendo em face da contribuição na solução de problemas considerados de difícil solução.

A principal motivação para a escolha deste tema foi a possibilidade de aplicar efetivamente alguns dos conceitos apresentados no ambiente acadêmico, em particular relacionados à área de IA. Mais especificamente buscou-se uma aplicação, próxima da real, onde pode-se testar a complexidade e eficiência de uma técnica heurística.

1.2. Escopo

Propor um método heurístico para tratar o Problema JSP fundamentado em conceitos de IA com enfoque puramente acadêmico, ou seja, sem considerar todos os aspectos que, na vida prática, estão envolvidos em problemas desta natureza. Não obstante, entende-se por modelo um sistema simplificado que tenta reproduzir os principais aspectos de um problema.

Sem perdas de generalidades, a palavra modelo não será utilizada em momento algum no presente texto, tem-se aqui a intenção de mostrar um modelamento, abordando os conhecimentos necessários e as dificuldades para sua criação assim como, um teste simplificado que mostra como este funciona.

1.3. Organização

Visando um melhor entendimento do método proposto, no Capítulo 2 são apresentados os fundamentos teóricos usados neste trabalho tais como: Busca em Profundidade, Busca em Amplitude, Busca Heurística e Estratégias de Controle.

No Capítulo 3 é apresentado o Problema Job Shop Scheduling abrangendo desde o conceito até uma exemplificação do mesmo.

No Capítulo 4 é apresentada uma proposta de solução para o problema através de um método fundamentado nos conceitos dos capítulos 2 e 3.

No Capítulo 5 é apresentado um exemplo de aplicação do método proposto, sendo discutidos alguns aspectos de implementação e procedimentos e analisados os resultados obtidos.

No Capítulo 6 são apresentadas as conclusões sobre o trabalho desenvolvido e logo a seguir as referências bibliográficas utilizadas para o desenvolvimento do mesmo.

2. Fundamentação Teórica [6]

2.1. O que é IA ?

A Inteligência Artificial (IA) é um ramo da computação bastante recente, aproximadamente 30 anos de existência. Este ramo era largamente utilizado em problemas de ciência básica, ficando restrita aos ambientes de laboratório em centros acadêmicos para estudos de alguns fenômenos e tarefas, tais como jogos e demonstração de teoremas.

Tendo em vista a sua história recente ainda em evolução, não há uma definição absoluta para a IA, contudo podemos expor diversos conceitos e definições. Em caráter informativo, segue abaixo uma das definições possíveis de IA:

"Inteligência Artificial (IA) é o estudo de como fazer os computadores realizarem tarefas que, no momento as pessoas fazem melhor."

Com o avanço das pesquisas em IA, foram desenvolvidas técnicas para a manipulação de uma maior quantidade de conhecimento, expandindo-se de forma considerável e transformando-se num sucesso tecnológico e industrial, extrapolando assim o seu uso, que antes era exclusivamente feito através de incursões em ciência básica.

Uma aplicação de particular interesse é no controle no processo produtivo em grandes empresas, além de outros exemplos tais como: diagnósticos de falhas em computadores e doenças em seres humanos, projeto de computadores, apólices de seguros, jogos xadrez etc. A figura abaixo ilustra alguns dos domínios de tarefas atualmente que a IA abrange:

Tarefas Comuns	Percepçao	
		Visão
		Fala
	Linguagem Natural	
		Compreensão
		Geração
		Tradução
	Raciocínio do Senso Comum	
		Controle de Robôs
Tarefas Formais	Jogos	
		Xadrez
		Gamão
		Damas
		Go
	Matemática	
		Geometria
		Lógica
		Cálculo Integral
		Demonstração de Propriedades de Programas
Tarefas de Especia	Engenharia	
		Projeto
		Descoberta de Erros
		Planejamento de Manufatura
	Análise Científica	
	Diagnóstico Médico	
	Análise Financeira	

Tabela 1 – Alguns dos domínios de tarefas da IA

2.2. O que é uma Técnica de IA ?

Existem inúmeras técnicas apropriadas para uma variedade destes problemas. Antes de se examinar algumas destas técnicas, faz-se necessário alguns esclarecimentos para uma melhor compreensão das mesmas.

Um dos poucos resultados rápidos e difíceis a surgir nas três primeiras décadas de pesquisa em IA é o que a *inteligência requer conhecimento*. Para compensar sua principal característica, a indispensabilidade, o conhecimento possui algumas propriedades menos desejáveis, incluindo:

- Ele é volumoso,
- É difícil caracterizá-lo com precisão,
- Ele está mudando constantemente,
- Ele difere de simples dados por organizar-se de uma maneira que corresponde ao modo como será usado.

Portanto, pode-se concluir que uma técnica de IA é um método que explora o conhecimento, devendo ser representado de tal forma que:

- O conhecimento capture generalizações. Noutras palavras, não que seja necessário representar separadamente cada situação individual. Ao invés disso, as situações que compartilham propriedades são agrupadas. Se não houver essa propriedade, seria necessário uma grande quantidade de memória e atualização. É por isso que normalmente chama-se algo sem essa propriedade de "dados" ao invés de conhecimento,

- Ele precisa ser compreendido pelas pessoas que o fornecem. Embora para muitos programas o "grosso dos dados" possa ser adquirido automaticamente (por exemplo, através de medições de instrumentos), em muitos domínios da IA, grande parte do conhecimento que um programa precisa é fornecido basicamente pelas pessoas através de sua compreensão,

- Ele pode ser facilmente modificado para corrigir erros e refletir mudanças do mundo e da nossa visão do mundo,

- Ele pode ser usado em inúmeras situações, mesmo que não seja totalmente preciso nem esteja completo,

- Ele pode ser usado para superar o seu próprio volume, auxiliando a limitar as várias possibilidades que, em geral, tem de ser consideradas.

As técnicas de IA precisam ser projetadas levando-se em conta estas restrições impostas pelos problemas de IA; entretanto, existe um certo grau de independência entre os problemas e as técnicas de solução dos mesmos. É possível solucionar problemas de IA sem que se usem técnicas de IA. E é possível aplicar técnicas de IA à solução de problemas não relacionados à inteligência artificial.

2.3. Definindo o problema como uma busca num espaço de estados

Supondo como definição inicial para um problema "jogar xadrez". Embora haja várias pessoas para as quais se possa propor um jogo de xadrez e esperar com razoável certeza o comportamento pretendido, a forma atual é uma definição bastante incompleta do problema a ser solucionado.

Para se criar um programa de “jogar xadrez”, deve-se primeiramente especificar a posição inicial no tabuleiro, as regras que definem os movimentos legais e as posições que representam vitória para um lado ou outro. Além disso, é preciso explicitar o objetivo implícito anteriormente de não apenas jogar um jogo lícito de xadrez, mas também de ganhar o jogo, se possível.

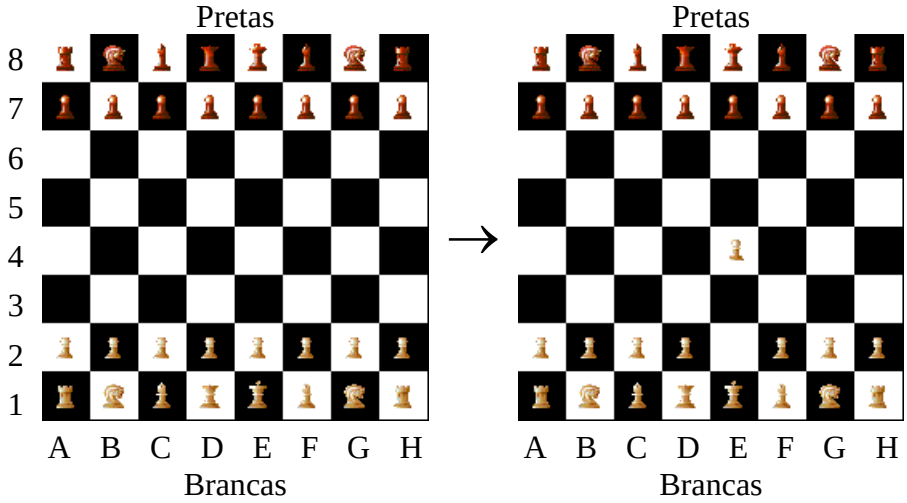


Figura 1 – Movimento legal no xadrez

No caso do problema “jogar xadrez”, é relativamente fácil proporcionar uma definição formal e completa para o mesmo. A posição inicial pode ser descrita na forma de uma matriz 8 x 8, onde cada posição contém um símbolo que representa a peça apropriada na posição oficial de abertura do jogo. Pode se definir como meta qualquer posição do tabuleiro na qual o oponente não tenha um movimento legal, e seu rei esteja sendo atacado.

Os movimentos legais são a maneira de sair do estado inicial e chegar a um estado-meta. Eles podem facilmente ser descritos como um conjunto de regras que consistem em duas partes: um lado esquerdo que serve como padrão a ser usado como base, e um lado direito que descreve a mudança a ser feita na posição do tabuleiro, afim de refletir o movimento.

Existem vários modos pelos quais estas regras podem ser escritas. Por exemplo, pode-se escrever uma regra como aquela ilustrada na figura acima. Entretanto, regras como esta teriam que ser escritas em número muito grande, já que deve haver uma regra separada para cada uma das aproximadamente 10^{120} possíveis posições no tabuleiro. O uso de tantas regras resulta em sérias dificuldades práticas:

- Nenhuma pessoa seria capaz de fornecer um conjunto completo dessas regras. Este processo demoraria muito e certamente não poderia ser realizado sem erros,

- Nenhum programa conseguiria lidar facilmente com essas regras. Apesar de poder ser usado um esquema para encontrar com relativa rapidez as regras relevantes de cada movimento, o simples armazenamento de tantas regras implica em sérias dificuldades.

Afim de minimizar esses problemas, deve-se procurar uma maneira mais genérica de escrever as regras para descrever os movimentos legais. Para tanto, é útil introduzir algumas notações convenientes na descrição de padrões e substituições. Por exemplo, a regra descrita na figura anterior e também muitas outras parecidas com a mesma, poderiam ser escritas conforme mostra a figura a seguir. Em geral, quanto mais sucintas forem as regras necessárias, menor será o trabalho para fornecê-las e mais eficiente será o programa ao utilizá-las.

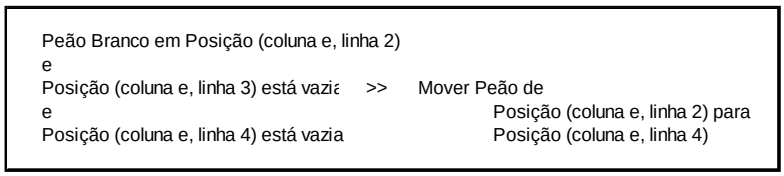


Figura 2 – Uma outra maneira de descrever movimentos no jogo de xadrez

Acaba-se de descrever o problema de "jogar xadrez" como um problema de movimentar-se num espaço de estados, onde cada espaço corresponde a uma posição legal no tabuleiro. Pode-se então jogar xadrez começando num estado inicial, usando um conjunto de regras para ir de um estado a outro, e tentar terminar num dos conjuntos de estados finais.

Esta representação do espaço de estados parece natural no caso do xadrez, porque o conjunto de estados, que corresponde ao conjunto de posições no tabuleiro, é artificial e bem organizada. A representação de espaço de estados forma a base da maioria dos métodos de IA. Sua estrutura corresponde a estrutura de solução de problemas em dois aspectos importantes:

- Ela permite a definição formal de um problema com sendo a necessidade de converter uma dada situação desejada usando um conjunto de operações permitidas,

- Ela permite definir o processo de solução de um determinado problema como uma combinação de técnicas conhecidas (cada uma representada por uma regra que define uma única etapa no espaço) e busca, sendo que a técnica geral é explorar o espaço para tentar descobrir o percurso do estado corrente para o estado-meta. A busca é um processo muito importante na solução de problemas difíceis para os quais não há técnicas mais diretas disponíveis.

2.4. Estratégias de Controle

Uma questão importante é a decisão de quais regras devem ser aplicadas durante o processo de busca de uma solução para o problema. Mesmo sem uma grande análise, fica claro que o modo como estas decisões serão tomadas terá impacto crucial sobre a velocidade com que o problema será solucionado, e mesmo se ele virá ou não a ser solucionado.

- A primeira exigência para uma boa estratégia de controle é que ela cause movimento, caso contrário, elas nunca levarão a uma solução,

- A segunda exigência para uma boa estratégia de controle é que ela seja sistemática, proporcionando assim movimento e, conseqüentemente, chegando-se a uma solução. Normalmente se opta por escolher aleatoriamente uma regra e aplica-la , no entanto, há chances de se avaliar um estado mais de uma vez durante o processo e de se usar muito mais etapas além do necessário. A exigência de que a estratégia de controle seja sistemática corresponde à necessidade de movimento global (ao longo do curso de várias etapas) e também de movimento local (ao longo do curso de uma única etapa). Uma estratégia de controle sistemática para um problema é a seguinte: Construa uma árvore com o estado inicial na raiz. Gere todas as ramificações da raiz aplicando ao estado inicial cada uma das regras pertinentes. A figura 3 mostra como a árvore estará neste ponto. Agora, em cada nó-folha, gere todos o sucessores aplicando todas as regras apropriadas. A árvore neste ponto é mostrada na figura 4. Continue o processo até que uma das regras produza o estado-meta. Este processo, chamado de busca em amplitude (breadth-first-search), pode ser descrito precisamente da seguinte maneira.

Algoritmo: Busca em Amplitude

1. Crie uma variável chamada Lista-de-Nós e ajuste-a para o estado inicial.
2. Até ser encontrado um estado-meta ou Lista-de-Nós ficar vazia, faça o seguinte:
   (a) Remova o primeiro elemento de Lista-de-Nós e chame-o de E. Se Lista-de-Nós estiver vazia, saia.
   (b) Para cada maneira como cada regra pode ser casada com o estado descrito em E, faça o seguinte:
      I. Aplique a regra para gerar um novo estado
      II. Se o novo estado for um estado-meta, saia e retorne este estado
      III. Caso contrário, acrescente o novo estado ao final da Lista-de-Nós

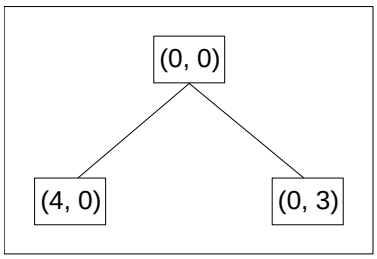


Figura 3 Um nível de uma árvore de busca em amplitude

Existem também outras estratégias de controle disponíveis. Por exemplo, pode-se perseguir uma única ramificação da árvore até ela produzir uma solução ou até ser tomada uma decisão de terminar o caminho. Faz sentido em encerrar-se um caminho quando este encontra um beco sem saída, produz um estado anterior ou fica maior que um imite pré-especificado de "futilidade". Neste caso, ocorrerá um retrocesso (backtracking) O estado mais recentemente criado pelos movimentos alternativos que estiverem disponíveis será revisitado e um novo estado será criado. Esta forma é chamada de *retrocesso cronológico* porque a ordem em que as etapas são desfeitas depende apenas da seqüência temporal em que as etapas foram originalmente seguidas. Especificamente, a etapa mais recente é sempre a primeira a ser desfeita.

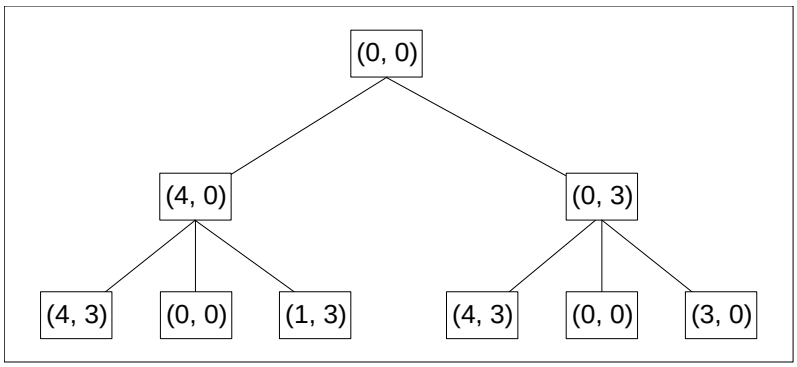


Figura 4 Dois níveis de uma árvore de busca em amplitude

O procedimento de pesquisa que acabamos de descrever é também chamado de pesquisa depth-first. O algoritmo a seguir descreve isto em termos precisos.

Algoritmo: Busca em Profundidade

1. Se o estado inicial é um estado-meta, saia e retorne sucesso,
2. Caso contrário, faça o seguinte até a sinalização de sucesso ou fracasso:
   (a) Gere um sucessor E, do estado inicial. Se não houver mais sucessores, sinalize fracasso,
   (b) Chame Busca em Profundidade com E como estado inicial,
   (c) Se for retornado sucesso, sinalize sucesso. Caso contrário, continue nesse laço (loop).

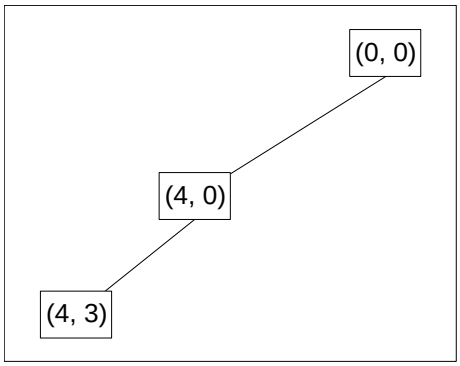


Figura 5 Uma árvore de busca em profundidade

A figura 5 mostra uma ilustração de uma busca em profundidade para um determinado problema .Uma comparação entre estes dois métodos simples produz as seguintes observações:

Vantagens da Busca em Profundidade

- A busca em profundidade requer menos memória, já que apenas os nós do caminho corrente são armazenados. Isto contrasta com a busca em amplitude em que toda a árvore gerada até o momento precisa ser armazenada.

- Por acaso (ou se houver cuidado na ordenação dos estados sucessores alternativos), a busca em profundidade pode encontrar a solução sem examinar grande parte do espaço de busca. Isto contrasta com a busca em amplitude na qual todas as partes da árvore precisam ser examinadas no nível n antes de serem examinados os nós do nível n + 1. Isto é particularmente significativo se existirem muitas soluções aceitáveis. A busca em profundidade pode parar quando uma delas for encontrada.

Vantagens da Busca em Amplitude

- A busca em amplitude não cairá na armadilha de explorar um beco sem saída. Isto contrasta com a busca em profundidade que pode seguir um caminho infrutífero durante um longo tempo, talvez para sempre, antes que o caminho termine num estado que não tenham mais sucessores. Este é um problema para a busca em profundidade particularmente no caso de ciclos (isto é, um estado tem um sucessor que é também um de seus antecessores); haverá necessidade de cuidados especiais para testar este tipo de situação. No exemplo da figura 4, se continuar a escolher sempre a primeira regra (na seqüência numérica) aplicável, ocorrerá exatamente este problema.

- Se houver uma solução, então a busca em amplitude certamente a encontrará. Além disso, se houver várias soluções, então uma solução mínima (isto é, uma que exija o menor número de etapas) será encontrada. Isto é garantido pelo fato de que os caminhos mais longos só são explorados depois que os mais curtos tiverem sido examinados. Este procedimento contrasta com a busca em profundidade, que pode encontrar um caminho longo para uma solução em parte da árvore, quando existe um caminho mais curto, em alguma outra parte inexplorada da árvore.

Uma combinação destas vantagens seria algo desejável. Considere-se o seguinte problema:

O Problema do Caixeiro Viajante: Um vendedor tem uma lista de cidades que precisa visitar exatamente uma vez. Há estradas entre cada par de cidades da lista. Encontre a rota que o vendedor deverá seguir para que a viagem seja a menor possível, e que termine numa mesma cidade, que poderá ser qualquer uma da lista.

Uma estrutura de controle simples, sistemática e que cause movimento poderá, em princípio, solucionar o problema. Ela simplesmente exploraria todos os caminhos possíveis na árvore e retornaria aquele de menor comprimento. Esta abordagem até funciona na prática quando a lista de cidades é pequena. Mas ela se torna inadequada quando o número de cidades começa a aumentar. Se há N cidades, então o número de diferentes percursos entre elas é $1*2*...*(N-1)$ ou $(N-1)!$. O tempo necessário para que se examine um único caminho é proporcional a $N!$. Supondo-se que haja 10 cidades, 10! é igual a 3.628.800, um número bastante grande. É muito fácil para um vendedor ter em sua lista 25 cidades a visitar. A solução deste problema poderia demorar mais tempo do que ele estaria disposto a esperar. Este fenômeno é chamado de *explosão combinatória*. Para combatê-lo, precisa-se de uma nova estratégia de controle.

Pode-se ludibriar a estratégia descrita usando uma técnica chamada ramificar-e-podar (branch-and-bound). Comece gerando caminhos completos, tomando nota do menor caminho encontrado até o momento. Desista de explorar qualquer caminho tão logo seu comprimento parcial fique menor que o menor caminho encontrado até então. Usando esta técnica, ainda pode-se garantir o encontro do menor caminho. Infelizmente, embora este algoritmo seja mais eficiente do que o primeiro, ele mesmo assim requer tempo exponencial. O tempo exato que ele economiza para um determinado problema depende da ordem em que os caminhos são explorados. Mas persiste a inadequação no caso da solução de problemas grandes.

2.5. Busca Heurística

Para resolver eficientemente muitos problemas difíceis, geralmente é necessário comprometer as exigências de mobilidade e sistematicidade, construindo uma estrutura de controle que não mais garanta encontrar a melhor resposta, mas que quase sempre encontre uma resposta muito boa.

Deste modo, apresenta-se a idéia de heurística[1]. A *heurística* é uma técnica que melhora a eficiência de um processo de busca, possivelmente sacrificando pretensões de completeza, algo parecido com um guia de turismo

Ela é válida no sentido de que aponta para direções geralmente interessantes; é imprópria no sentido de que pode deixar de fora pontos de interesse para determinados indivíduos. Certas heurísticas podem ajudar a guiar um processo de busca sem sacrificar a completeza de que o

[1] A palavra heurística vem do grego *heuriskein* que significa “descobrir”, que também é a origem da palavra *eureka*, derivada da famosa exclamação de Arquimedes, *heurika* (de "Descobrir"), preferida quando ele descobriu um método para determinar a pureza do ouro.

processo possa ter tido anteriormente. Outras podem também ocasionalmente fazer com que um excelente caminho seja negligenciado (na verdade é o que ocorre com muitas das melhores idéias).

No entanto, na média, elas melhoram a qualidade dos caminhos que são explorados. Usando uma boa heurística, pode-se obter boas soluções (embora possivelmente não ótimas) para problemas difíceis, como o do caixeiro-viajante, num tempo menor que o exponencial. Há boas heurísticas de propósito geral úteis numa ampla variedade de domínios. Além disso, é possível construir heurísticas de propósito específico que exploram conhecimento específico do domínio da aplicação para resolver determinados problemas.

Um exemplo de uma boa heurística de propósito geral útil numa variedade de problemas combinatórios é a *heurística do vizinho mais próximo*, que funciona através da seleção de alternativa localmente superior em cada etapa. Aplicando-a ao problema do caixeiro-viajante, produzir-se-á o seguinte procedimento:

1. Selecione arbitrariamente uma cidade inicial.
2. Para selecionar a próxima cidade, examine todas as cidades que ainda não foram visitadas e selecione a que estiver mais perto da cidade atual. Vá para lá a seguir.
3. Repita a etapa 2 até todas as cidades terem sido visitadas.

Este procedimento é executado em tempo proporcional a N^2, uma melhoria significativa sobre $N!$, e é possível provar um limite superior no erro em que ela incorre Quando se trabalha com heurísticas de propósito geral, como a do vizinho mais próximo, normalmente é possível provar tais limites de erro, o que garante que não está pagando um preço tão alto em precisão, a favor da velocidade.

Em muitos problemas de IA, porém, não é possível produzir esses limites tranqüilizadores. Isto é verdadeiro por dois motivos:

- Quando se lida com problemas reais, geralmente é difícil medir precisamente o valor de determinada solução. Embora a extensão de uma viagem a várias cidades seja uma noção precisa, o mesmo não ocorre com respostas e perguntas do tipo “Por que a inflação subiu ?”.

- Quando se lida com problemas reais, geralmente é bom usar heurísticas baseadas em conhecimentos relativamente não estruturados. É freqüentemente impossível definir este conhecimento de forma a permitir que seja realizada uma análise matemática de seu efeito no processo de busca.

Há dois modos importantes pelos quais os conhecimentos heurísticos, específicos de cada domínio, podem ser incorporados num procedimento de busca baseado em regras:

- Nas regras propriamente ditas. Por exemplo, as regras de um sistema de jogo de xadrez podem descrever não simplesmente o conjunto de movimentos legais, mas sim um conjunto de movimentos “sensatos”, determinados pelo autor das regras.

- Na forma de uma função heurística que avalia estados de problemas isolados e determina se eles são desejáveis ou não.

Uma função heurística é uma função que mapeia de descrições de estados de problema até medidas de conveniência, normalmente representadas como números. Os aspectos do estado do problema que são considerados, o modo como esses aspectos são avaliados e os pesos atribuídos a aspectos isolados de forma que o valor da função heurística num determinado nó no processo de busca forneça a melhor estimativa possível sobre se o nó está ou não no caminho desejado para uma solução.

A busca heurística é um método bastante genérico, aplicável a uma grande classe de problemas. Ela engloba uma série de técnicas específicas, sendo que cada uma delas é particularmente eficaz para uma pequena classe de problemas. A fim de escolher o método mais apropriado (ou uma combinação de métodos) para um determinado problema, é necessário analisá-lo em várias dimensões-chave:

- O problema pode ser decomposto num conjunto (ou quase isto) de subproblemas independentes, menores ou mais fáceis ?
- Certos passos em direção à solução podem ser ignorados ou pelo menos desfeitos caso fique provado que são imprudentes ?
- O universo do problema é previsível ?
- Uma boa solução para o problema pode ser considerada óbvia sem haver comparação com todas as outras soluções possíveis ?
- A solução desejada é um estado do mundo ou um caminho para um estado ?
- Há necessidade absoluta de grande quantidade de conhecimento para resolver o problema, ou o conhecimento é importante apenas para limitar a busca ?
- Um computador que simplesmente receba o problema tem condições de retornar a solução, ou esta exige interação entre ele e uma pessoa ?

Quando se examinam problemas reais do ponto de vista de todas essas questões, fica claro que existem várias classes nas quais os problemas podem ser enquadrados. Cada uma dessas classes pode ser associada a uma estratégia de controle genérica apropriada à solução do problema. Por exemplo, o problema genérico de classificação. A tarefa aqui é examinar uma informação recebida e decidir, dentre um conjunto de classes conhecidas, de qual classe ela é instância. A maioria das tarefas de diagnóstico, incluindo o diagnóstico médico e também o diagnóstico de defeitos em dispositivos mecânicos, são exemplos de classificação. Um outro exemplo de estratégia genérica é propor e refinar. Muitos problemas de projeto e planejamento podem ser abordados com essa estratégia.

Dependendo da granulosidade com que se tenta classificar problemas e estratégias de controle, pode-se obter diferentes listas de tarefas e procedimentos genéricos. Entretanto, é importante que não existe uma única maneira de resolver todos os problemas. Ao contrário, se forem analisados e classificados cuidadosamente os problemas e métodos de solução de acordo com o tipo de problema aos quais se adaptam, poder-se-á trazer para cada novo problema muito do que se aprendeu com a solução de outros problemas semelhantes.

2.6. Algumas Técnicas de Busca Heurística

As técnicas apresentadas resumidamente a seguir constituem-se numa variedades da busca heurística, discutida no tópico anterior, sendo independentes de qualquer tarefa ou domínio de problema em particular. No entanto, quando aplicadas a estes, sua eficácia torna-se altamente dependente do modo como se explora o conhecimento de cada domínio da aplicação, já que sozinhas não são capazes de superar a explosão combinatória à qual os processos de busca são tão vulneráveis. Por este motivo, estas técnicas são chamadas de métodos fracos.

Um resultado importante que emergiu nas últimas três décadas de busca em IA foi a percepção de que estes métodos fracos têm eficácia limitada para resolver sozinhos problemas difíceis; entretanto, essas técnicas continuam a fornecer a estrutura na qual pode ser colocado o conhecimento específico a cada domínio, manualmente ou como resultado do aprendizado automático. Assim, eles continuam a formar o núcleo da maioria dos sistemas de IA.

2.6.1. Gerar e Testar

Esta estratégia é a mais simples de todas, sendo um procedimento de busca em profundidade, já que soluções completas precisam ser geradas antes de serem testadas. Em sua forma mais sistemática, ela é simplesmente uma busca exaustiva do espaço do problema. Logo abaixo está o seu algoritmo:

Algoritmo: Gerar e Testar

1. Gerar uma solução possível. Para alguns problemas, isto significa gerar um ponto em particular no espaço do problema, para outros, significa gerar um caminho, a partir de um estado inicial.
2. Testar para ver se a solução gerada é realmente uma solução, comparando o ponto escolhido ou o ponto final do caminho escolhido com o conjunto de estados-meta aceitáveis.
3. Se uma solução tiver sido encontrada, saia. Caso contrário, volte à etapa 1.

2.6.2. Subida da Encosta Simples

Subida da encosta (Hill Climbing) é uma variação do método gerar e testar, na qual a realimentação dos testes é usada para ajudar o gerador a decidir-se para onde deslocar-se no espaço de busca..

Num procedimento gerar e testar puro, a função de teste responde apenas com um sim ou não. Mas, se a função de teste for incrementada com uma função heurística que forneça uma estimativa de proximidade que um determinado estado está do estado-meta, o procedimento de geração poderá explorá-lo. Logo abaixo está o seu algoritmo:

Algoritmo: Subida da Encosta Simples

1. Avalie o estado inicial. Se ele for um estado-meta, retorne-o e encerre. Caso contrário, continue com o estado inicial como estado corrente.
2. Repita a operação (“faça um loop”) até que uma solução seja encontrada ou até não haver nenhum operador a ser aplicado ao estado corrente.

(a) Escolha um operador que ainda não tenha sido aplicado ao estado corrente e aplique-o para produzir um novo estado.

(b) Avalie o novo estado

I. Se for um estado-meta, então retorne-o e encerre.

II. Se não for um estado-meta, mas for melhor que o estado corrente, então transforme-o num estado corrente.

III. Se não for melhor que o estado corrente, então continue no laço (“loop”).

A diferença chave entre este algoritmo e aquele fornecido para o procedimento gerar e testar é o uso da função de avaliação como meio de injetar no processo de controle conhecimentos específicos a respeito da tarefa, conferindo assim poder, tanto a este como outros métodos de busca heurística, para solucionar problemas de outro modo intratáveis.

2.6.3. Subida da Encosta pela Trilha mais Íngreme

Uma variação útil da subida da encosta simples considera todos os movimentos feitos a partir do estado corrente e seleciona o melhor deles como próximo estado. Este método é chamado de subida da encosta pela trilha mais íngreme (steepest-ascent-hill climbing) ou busca pelo gradiente. Nota-se que este método contrasta com o método básico pelo qual se seleciona o primeiro estado que for melhor que o estado corrente. Logo abaixo está o seu algoritmo:

Algoritmo: Subida da Encosta pela Trilha mais Ìngreme

1. Avalie o estado inicial. Se ele for também for um estado-meta, retorne-o e encerre. Caso contrário, continue com o estado inicial como estado corrente.
2. Repita o processo até encontrar uma solução ou até que uma iteração completa não produza nenhum mudança no estado corrente.
   (a) Atribua a *SUCESSOR* seja um estado onde qualquer possível sucessor do estado corrente seja melhor do que *SUCESSOR*.
   (b) Para cada operador que se aplique ao estado corrente, faça o seguinte:
      I. Aplique o operador e gere um estado novo.
      II. Avalie o novo estado. Se for um estado-meta, retorne-o e encerre, senão, compare-o a *SUCESSOR*. Se for melhor então atribua-o a *SUCESSOR*., senão for melhor, deixe *SUCESSOR* como está.
   (c) Se *SUCESSOR* for melhor que o estado corrente, então transforme-o em estado corrente.

2.6.4. Busca pela Melhor Escolha

A busca pela melhor escolha é um método que combina as vantagens das buscas em profundidade e em amplitude.

A operação do algoritmo é relativamente simples. Ele progride em passos, expandindo um nó a cada passo, até gerar um nó que corresponda a um estado-meta. A cada passo, ele pega os nós mais promissores gerados até o momento que ainda não foram expandidos. Ele gera os sucessores do nó escolhido, aplica a função heurística a eles, e acrescentá-os à lista de nós abertos, depois de verificar se algum deles já foi gerado antes. Com esta verificação, garante-se que cada nó só aparecerá uma vez, embora muitos nós possam apontar para ele como sucessor. Começa-se então o passo seguinte.

Algoritmo: Busca pela Melhor Escolha

1. Comece com *ABERTOS* contendo apenas o estado inicial.
2. Até ser encontrado um estado-meta ou até não haver mais nenhum nó em *ABERTOS*, faça o seguinte:
   (a) Pegue o melhor nó de *ABERTOS*.
   (b) Gere seus sucessores.
   (c) Para cada sucessor, faça:
      I. Se ele ainda não tiver sido gerado antes, avalie-o, acrescente-o à *ABERTOS* e registre seu pai.
      II. Se ele já tiver sido gerado antes, mude o pai se este novo caminho for melhor que o anterior. Neste caso, atualize o custo de chegar a este nó e a qualquer um dos sucessores que este nó possa ter.

A figura abaixo mostra o início de um procedimento de busca pela melhor escolha. Inicialmente, há apenas um nó, portanto ele será expandido. Esta expansão gera três novos nós. A função heurística, que, neste exemplo, é uma estimativa do custo de se chegar a uma solução a partir de um dado nó, é aplicada a cada um destes novos nós. Como o nó D é o mais promissor, portanto ele é analisado e gera os sucessores, E e F. Depois, a função heurística é aplicada a eles. Agora, um outro caminho, que passa pelo nó B, parece mais promissor, portanto ele é analisado e gera os nós G e H. Mas, novamente, quando estes novos nós são avaliados, eles parecem menos promissores do que o outro caminho, portanto, a atenção é voltada ao caminho que vai de D a E. O nó E é então expandido, produzindo os nós I e J. Na etapa seguinte, o nó J será expandido por ser o mais promissor. Este processo pode continuar até que uma solução seja encontrada.

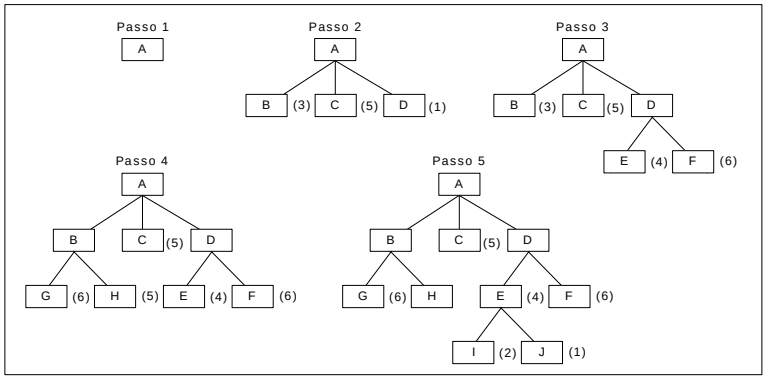


Figura 6 Uma busca pela melhor escolha

2.6.5. Satisfação de Restrições

Muitos problemas de IA podem ser vistos como problemas de satisfação de restrições, em que o objetivo é descobrir algum estado do problema que satisfaça a um determinado conjunto de restrições.

Ao classificar um problema como sendo deste tipo, normalmente é possível reduzir substancialmente a quantidade de busca necessária, em comparação com um método que tenta formar soluções parciais diretamente através da escolha de valores específicos para componentes da solução eventual.

A satisfação de restrições é um procedimento de busca que opera num espaço de conjunto de restrições. O estado inicial contém as restrições originalmente fornecidas na descrição do problema. Um estado-meta é qualquer estado que tenha sido “suficientemente” limitado, onde o termo “suficientemente” precisa ser definido para cada problema.

Este processo consiste de duas etapas Primeiro, as restrições são descobertas e propagadas o mais distante possível em todo o sistema. Depois, se ainda não houver uma solução, a busca começa. Uma suposição sobre alguma coisa é feita, e esta é acrescentada à lista como nova restrição. A propagação pode então ocorrer com esta nova restrição, e assim por diante.

A primeira etapa, propagação, surge do fato de que em geral há dependências entre as restrições. Estas dependências ocorrem porque muitas restrições envolvem mais de um objeto, e muitos objetos participam de mais de uma restrição. Portanto, por exemplo, assuma-se que se comece com uma restrição N = E + 1. depois, se acrescentar a restrição N = 3, poder-se-á propagá-la para obter uma restrição mais forte sobre E, especificamente, E = 2.

A propagação de restrições também surge da presença de regras de inferências já fornecidas. A propagação termina por um destes dois motivos. Primeiro, pode sr detectada uma contradição. Se isto acontecer, então não há solução consistente com todas as restrições conhecidas. Se a contradição envolver apenas as restrições que foram definidas com parte da especificação do problema (ao contrário daquelas que foram supostas durante a solução do problema), então não existe nenhuma solução.

O segundo motivo para o término é que a propagação perdeu a força e não há nenhuma alteração a ser feita com base no conhecimento atual. Se isto acontecer, e nenhuma solução tiver sido ainda adequadamente especificada, então a busca é necessária para que o processo entre outra vez em movimento.

Neste ponto, começa a segunda etapa. Há necessidade de que certas hipóteses sobre uma maneira de fortalecer as restrições sejam feitas. No caso de um problema de criptoaritmética, por exemplo, isto em geral significa supor um determinado valor para alguma letra. Uma vez cumprida esta operação, a propagação de restrições pode começar novamente a partir deste novo estado. Se for encontrada uma solução, ela poderá ser reportada. Se ainda mais suposições forem necessárias, elas poderão ser feitas. Se for detectada uma contradição, então o retrocesso pode ser usado para tentar uma suposição diferente e prosseguir com ela. Este procedimento pode ser definido mais precisamente da seguinte maneira:

Algoritmo: Satisfação de Restrições

1. Propague as restrições disponíveis. Para tanto, primeiro atribua a *ABERTOS* o conjunto de todos os objetos que precisam ter valores atribuídos a eles numa solução completa. Depois, fará até ser detectada uma inconsistência ou até *ABERTOS* ficar vazio.
   (a) Selecione um objeto *OB* de *ABERTOS*. Fortaleça o máximo possível o conjunto de restrições que se aplicam a *OB*.
   (b) Se este conjunto for diferente do conjunto assinalado quando *OB* foi examinado pela última vez, ou seja, se esta for a primeira vez que *OB* é examinado, então acrescente a *ABERTOS* todos os objetos que compartilham com *OB* quaisquer restrições.
   (c) Remove *OB* de *ABERTOS*.
2. Se a união de restrições descobertas definir uma solução, então saia e reporte a solução encontrada.
3. Se a união de restrições descobertas definir uma contradição, então retorne fracasso.
4. Se não ocorrer nenhuma das hipóteses citadas, então será necessário fazer suposições sobre algo afim de prosseguir com o processo. Para tanto, repita a operação até que uma solução tenha sido encontrada ou até que todas as soluções possíveis tenham sido eliminadas.
   (a) Selecione um objeto cujo valor ainda não tenha sido determinado e selecione um modo de fortalecer as restrições sobre aquele objeto.
   (b) Recursivamente, invoque a satisfação de restrições com o atual conjunto de restrições aumentado pela restrição de fortalecimento que acabou de ser solucionada.

Este algoritmo foi definido da maneira mais genérica possível. Sua aplicação a um domínio de problema em particular requer o uso de dois tipos de regras: regras que definem o modo como as restrições podem ser propagadas de forma válida e regras que sugerem suposições quando estas se fizerem necessárias.

3. O Problema Job Shop Scheduling

3.1. Introdução

Como dito anteriormente os processos de IA, atualmente são utilizados com sucesso em grandes empresas. O problema do escalonamento ou sequenciamento da produção é uma forma de tomada de decisão que possui um papel crucial nas empresas, tanto de manufatura como de serviços. Note que em uma linha de produção interrupções em um dado processo podem acarretar em perdas de capital, atrasos ou até mesmo falhas no processo de fabricação de um dado produto. No atual ambiente competitivo, o efetivo escalonamento tornou-se uma necessidade para a sobrevivência no mercado. As companhias devem esforçar-se ao máximo para cumprir as datas

acordadas com seus clientes [1] sob pena de comprometer significativamente sua imagem perante os mesmos.

O escalonamento das atividades ou scheduling é uma das atividades que compõe o Planejamento da Produção. Nele são levadas em consideração uma série de elementos que disputam vários recursos por um período de tempo, recursos estes que possuem uma capacidade limitada. Os elementos a serem processados são chamados de ordens de fabricação ou *jobs* e são compostos de partes elementares chamadas atividades ou operações. Os principais objetivos tratados no problema de escalonamento podem ser resumidos a: atendimento de prazos (ou datas de entrega), minimização do tempo de fluxo dos estoques intermediários, identificação dos gargalos e maximização da utilização da capacidade disponível, ou mesmo na combinação destes objetivos [2].

A maioria dos problemas de programação estudados aplica-se ao ambiente conhecido como *Job Shop.* O *Job Shop* tradicional é caracterizado por permitir diferentes fluxos das ordens entre as máquinas e diferentes números de operações por ordem, que são processadas apenas uma vez em cada máquina. Também podemos dizer que é uma forma de produção onde as diversas peças a serem produzidas atravessam o sistema de manufatura com ultrapassagem, ou seja, não possuem uma mesma seqüência. Por outro lado, nos *Flow Shops* todas as peças atravessam o sistema de manufatura na mesma ordem, ou seja, as ordens compõem-se de seqüências de operações estritamente ordenadas, onde todos os movimentos devem ter uma direção uniforme. Este tipo de produção é caracterizado por produzir quantidades elevadas de um tipo de produto de cada vez, de forma repetitiva [3].

3.2. Modelos de Solução

O problema Job-Shop Scheduling é classificado na literatura como NP-Completo, isto é, um problema para o qual não existem algoritmos que o resolva em tempo polinomial. Trata-se de um "problema de otimização combinatória". Utiliza-se de uma enumeração explícita ou implícita de todas as alternativas possíveis a procura de soluções satisfatórias ou a procura da solução ótima, dependendo da classe do algoritmo de otimização utilizado.[4]

Os algoritmos de otimização podem ser classificados em 2 classes: os modelos matemáticos (procura pela solução ótima) e os heurísticos (procura por aproximação).

- **Modelos Matemáticos**: Modelo que enfatiza a obtenção de resultados ótimos em função de algum parâmetro de desempenho. Este pode ser, por exemplo, a minimização dos tempos de produção ou a maximização do uso dos recursos. Dependendo da complexidade do problema tratado, pode consumir muito tempo para obter a solução ótima [8]

- **Modelos Heurísticos**: Modelo baseado em regras práticas (indutivas) de escalonamento que enfatiza a obtenção de "boas" soluções, próxima da solução ótima, sem, no entanto, garantir a solução ótima. Os modelos híbridos são caracterizados pela obtenção de boas soluções para um problema em tempos de computação viáveis.

A escolha do modelo mais adequado à solução de problemas reais do tipo Job-Shop deve levar em consideração alguns aspectos como [4]:

- O número de máquinas envolvidas.
- Os roteiros das ordens de fabricação (jobs).
- O regime de chegada das ordens.
- A variabilidade dos tempos de processamento entre outras características.

Devido ao grande número de soluções possíveis e à complexidade do problema de sequenciamento *(scheduling*), torna-se difícil e até impossível modelar todas as variáveis envolvidas no processo utilizando modelos matemáticos. Além disso, o tempo de resposta para os modelos matemáticos eleva-se consideravelmente, tornando impraticável a obtenção de soluções ótimas em tempos satisfatórios. Dessa forma, algoritmos otimizantes matemáticos são computacionalmente viáveis quando aplicados a problemas reais pequenos, com objetivos limitados.[4]

Para problemas de porte similar aos encontrados no ambiente real, costuma-se sacrificar a obtenção de uma solução ótima por métodos heurísticos, que resultem em uma solução subótima ("boa" solução), com tempo computacional aceitável.[4]

Nos modelos heurísticos, ditos satisfatorizantes, pois não garantem que a solução encontrada seja a ótima, mas sim muito próxima desta, quando há mais de uma atividade disputando um recurso, cabe a uma regra selecionar a atividade a atender primeiro, de forma a otimizar o programa de acordo com algum critério.[4]

Em virtude das dificuldades intrínsecas aos problemas de otimização combinatória, pesquisadores têm concentrado esforços na utilização de heurísticas para solucionar problemas desse nível de complexidade.

3.3. Conceituação

Descreve-se, a seguir, um problema da programação da produção do tipo Job-Sho*p*, em sua forma básica.

Um conjunto de *n jobs J = { J1, J2, ... , Jn }* deve ser processado em um conjunto de *m* máquinas *M = { M1, M2, ... , Mm }* disponíveis. Cada job possui uma ordem de execução específica entre as máquinas, ou seja, um job é composto de uma lista ordenada de operações, cada uma das quais definida pela máquina requerida e pelo tempo de processamento na mesma. As restrições que devem ser respeitadas são:

- Operações não podem ser interrompidas e cada máquina pode processar apenas uma operação de cada vez
- Cada job só pode estar sendo processado em uma única máquina de cada vez.
- Cada job é fabricado por uma seqüência conhecida de operações.
- Não existe restrição de precedência entre operações de diferentes jobs.
- Não existe qualquer relação de precedência entre as operações executadas por uma mesma máquina.

Uma vez que as seqüências de máquinas de cada job são fixas, o problema a ser resolvido consiste em determinar as seqüências dos jobs em cada máquina, de forma que o tempo de execução transcorrido, desde o início do primeiro job até o término do último, seja mínimo. Essa medida de qualidade de programa, conhecida por *makespa*n, não é a única existente, porém é o critério mais simples e o mais largamente utilizado. O objetivo é encontrar a solução com menor *makespan* (tempo de finalização / fluxo*)*, ou seja, a melhor solução para o problema.[5]

Normalmente o número de restrições é muito grande, o que torna o Job-Shop um dos mais difíceis problemas. Testes para problemas maiores são mais difíceis de obter o sequenciamento ótimo conhecido, ou seja, mais difícil é obter o arranjo de todas as tarefas nas máquinas que satisfaça as restrições de precedência cujo processamento se faz em um menor tempo. É por isso que é tão difícil avaliar a performance de algoritmos de sequenciamento.[5]

3.4. Regras de Priorização [7]

O processo de determinação de qual tarefa deve ser iniciada numa linha de produção, é conhecido como de prioridades. São utilizadas para se obter uma seqüência de tarefas. As mais comuns são:

1. **PEPS** ➔ Primeiro a Entrar, Primeiro a Sair ➔ Os pedidos são executados segundo a ordem de chegada na produção.

2. **METP** ➔Menor Tempo de processamento ➔ É executado primeiro a tarefa com o menor tempo de conclusão.

3. **DE** ➔Data de Entrega ➔ Executar a tarefa com data de entrega mais próxima da primeira.

4. **FE** ➔ Folga até Entrega ➔ É calculado como a diferença entre o TE (tempo de entrega) e o TP (tempo de processo). Os pedidos com a menor folga são executados primeiro.

5. **RC** ➔ Reação Crítica ➔ É calculada como a diferença entre a data de entrega e a data atual dividida pelo nº de operações restantes.
   (Data de entrega – Data atual) / Nº operações restantes

Também podem ser utilizadas regras não otimizantes, tais como: importância dos clientes, valor do pedido, Curva ABC, tipo de pedido (à vista, à prazo, etc..) etc

3.5. Exemplo

Um Problema Job-Shop é normalmente referenciado como um *n/m* JSP, onde *n* é número de trabalhos/tarefas e *m* o número de máquinas/recursos. Abaixo está um exemplo ilustrativo de um problema de programação do tipo Job-Shop problema *3/10* JSP.

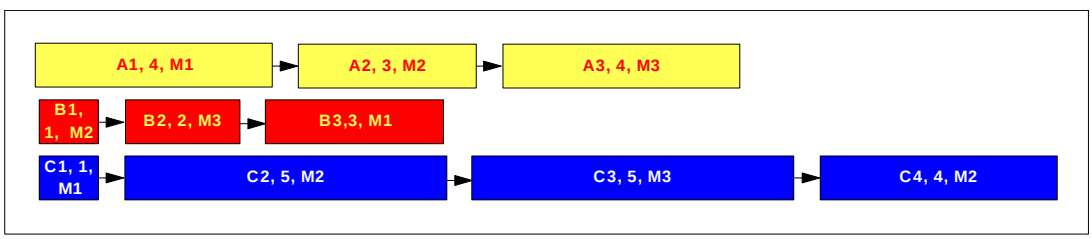


Figura 7 – Tarefas do caso (3,10) JSP

Cada bloco representa uma tarefa e o sua inscrição (X, Y, Z) possui o seguinte significado:

- X – nome da tarefa e sua operação.
- Y – tempo de processamento.
- Z – recurso em que a operação deverá ser executada.

Tarefa	Tempo de Process.(u)	Recurso	Legenda
A1	4	M1	A1, 4, M1
A2	3	M2	A2, 3, M2
A3	4	M3	A3, 4, M3
B1	1	M2	B1, 1, M2
B2	2	M3	B2, 2, M3
B3	3	M1	B3, 3, M1
C1	1	M1	C1, 1, M1
C2	5	M2	C2, 5, M2
C3	5	M3	C3, 5, M3
C4	4	M2	C4, 4, M2

Tabela 2 –Tarefas do caso (3, 10) JSP

Logo abaixo está a distribuição das tarefas nos recursos, desconsiderando qualquer regra de priorização, conforme a especificação acima, obtendo-se assim a Fila de Prioridades de Execução das Tarefas dos Recursos.

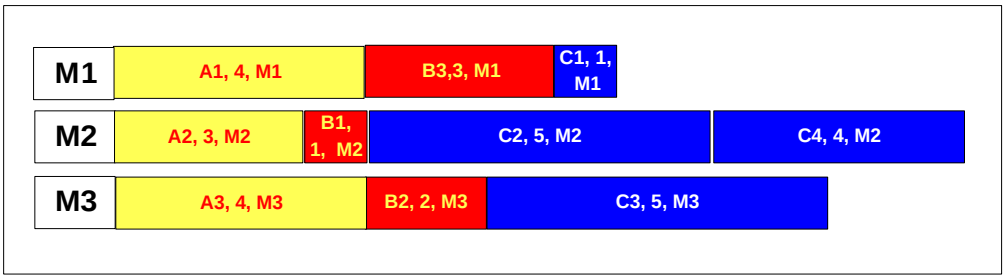


Figura 8 – Fila de Prioridades de Execução das Tarefas do caso (3,10) JSP

A partir de sucessivas trocas de posições é possível obter-se uma distribuição de tarefas diferentes e um makespan variável. A figura abaixo ilustra o processo de troca de uma tarefa na fila de prioridade de um recurso:

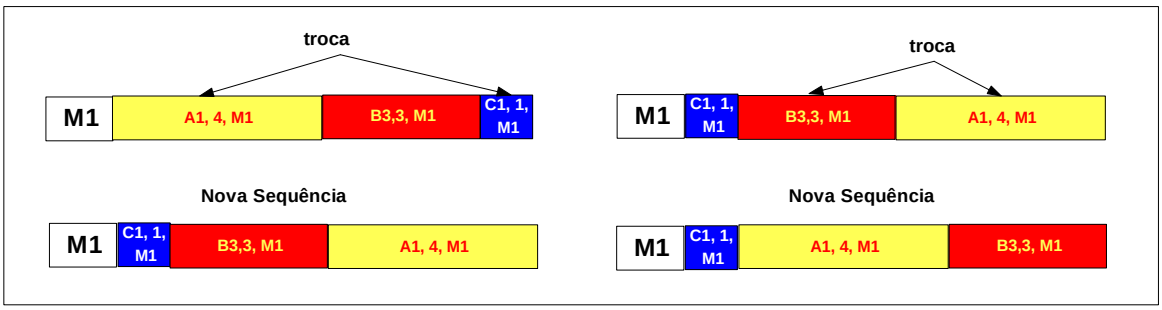


Figura 9 – Troca de posições entre as tarefas de um recurso

Abaixo está o sequenciamento inicial das tarefas considerando os aspectos descritos na seção Descrição do Problema:

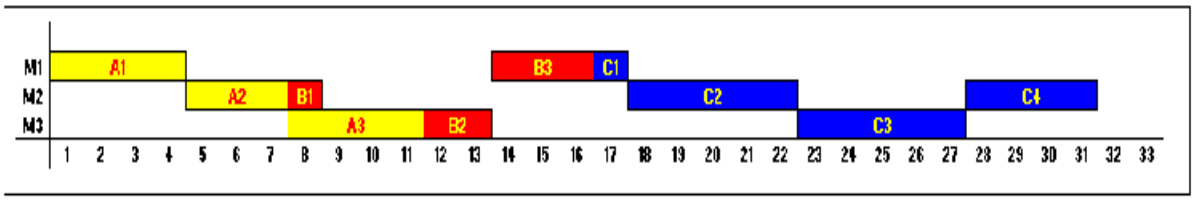


Figura 10 – Sequenciamento Inicial das Tarefas do caso (3,10) JSP

O makespan obtido no sequenciamento acima foi de 31 unidades de tempo. Após várias trocas de posições entre as tarefas de cada fila é possível chegar-se ao melhor makespan que corresponde ao valor 15 conforme mostra a figura abaixo:

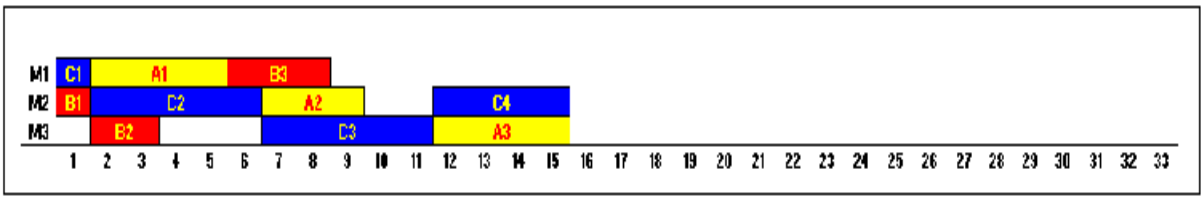


Figura 11 – Melhor Sequenciamento das Tarefas do caso (3,10) JSP

Apenas para este conjunto de tarefas e considerando a sua distribuição, tem-se um total de 112 combinações possíveis de sequenciamento. Maiores detalhes sobre o cálculo do nº de combinações bem como os mecanismos de troca e cálculo do makespan, serão discutidos nos capítulos 4 e 5.

O gráfico logo abaixo ilustra o espaço de soluções para o caso citado acima, obtido a partir da execução de uma Busca Exaustiva ao problema. A linha em azul representa todas as possibilidades de solução para o problema e a linha em vermelho representa todas as boas soluções encontradas ao longo desta busca.

Nota-se que além de existirem poucas boas soluções, foi necessário percorrer-se praticamente todo o espaço de solução para encontrar-se a “melhor boa solução” ou mais comumente conhecida como “subótima”

A Busca Exaustiva é um importante mecanismo de análise pois pode fornecer indicações quanto a padrões de comportamentos da solução possibilitando assim traçar estratégias para se aproximar do estado-meta, distanciar-se das soluções insatisfatórias...

Figura 12 – Gráfico de Sequenciamento das Tarefas do caso (3,10) JSP

4. Proposta de Solução

4.1. Introdução

Conforme exposto no capítulo Fundamentação Teórica o objetivo é percorrer o máximo de regiões possíveis (amplitude) no espaço de soluções avaliando um nº determinado de soluções por região (profundidade), obtendo-se assim a melhor solução dentre o conjunto de soluções avaliadas.

A figura abaixo ilustra o funcionamento da lógica empregada no mecanismo de Busca Heurística implementado:

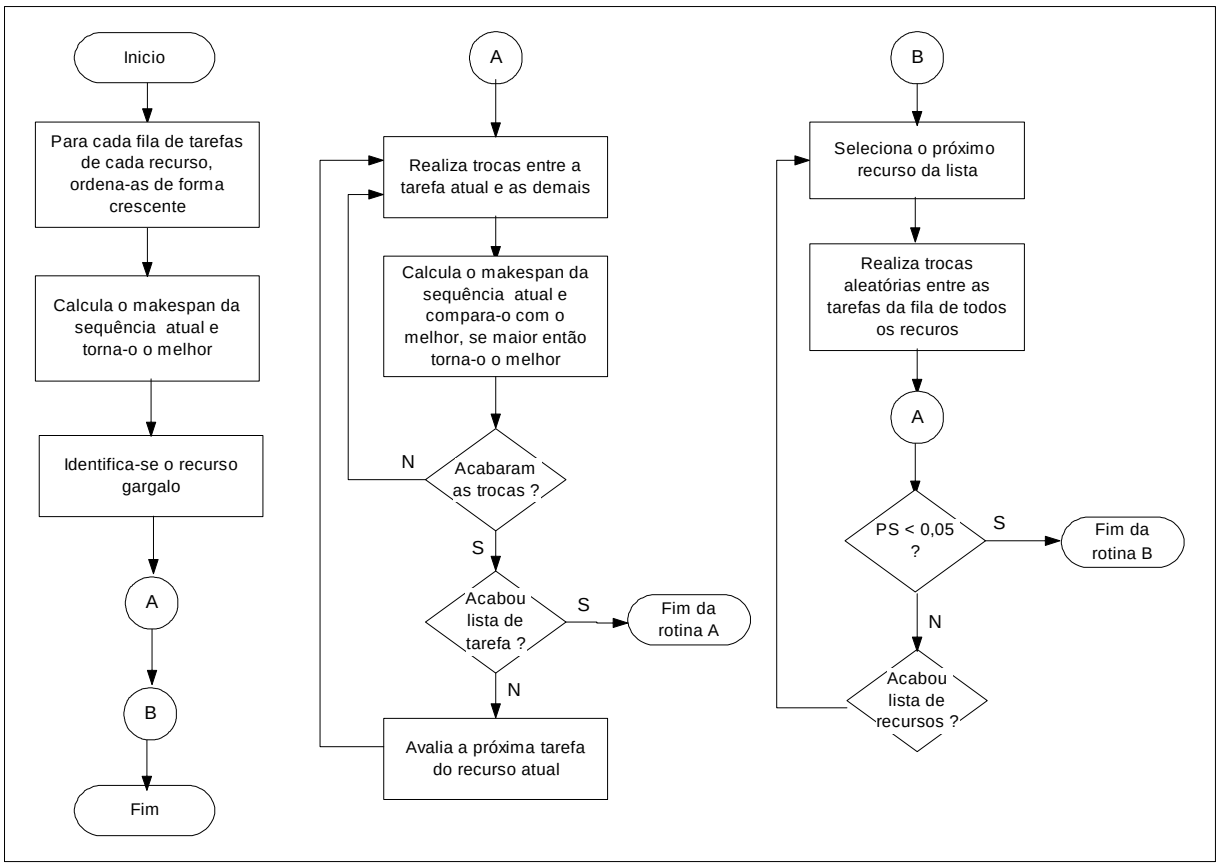


Figura 13 – Lógica empregada na Busca Heurística

4.2. Mecanismo de Controle da Busca Heurística

Para que o tempo de busca não seja muito longo e, ao mesmo tempo, satisfaça aos critérios tanto qualitativos, melhor valor encontrado ou o mais próximo deste, e quantitativos, maior nº de regiões possíveis analisadas, foi acrescentado um controle, chamado Porcentagem de Sucesso (PS), explicada logo a seguir:

PS = (Nº Seq Mksp * 100) / Nº Seq Med ,

onde:

Nº Seq Mksp - quant.de boas seqüências obtidas (menor makespan medido)

Nº Seq Med - quant.de seqüências medidas, exceto as já avaliadas

Este mecanismo de controle, além de estabelecer o ponto de parada da busca, fornece uma importante métrica de desempenho da mesma por ter uma relação direta com a amplitude e profundidade alcançadas.

Após sucessivos testes constatou-se que um índice de valor 0,05 possibilita atingir uma boa amplitude para a maioria dos casos.

4.3. Mecanismo de Controle da Troca de Tarefas

Considerando cada Fila de Prioridades de Tarefas do Recurso como sendo uma lista de elementos distintos, tem-se um problema de permutação de nº elementos dois a dois. A figura abaixo ilustra o processo de obtenção das possibilidades de combinações das tarefas:

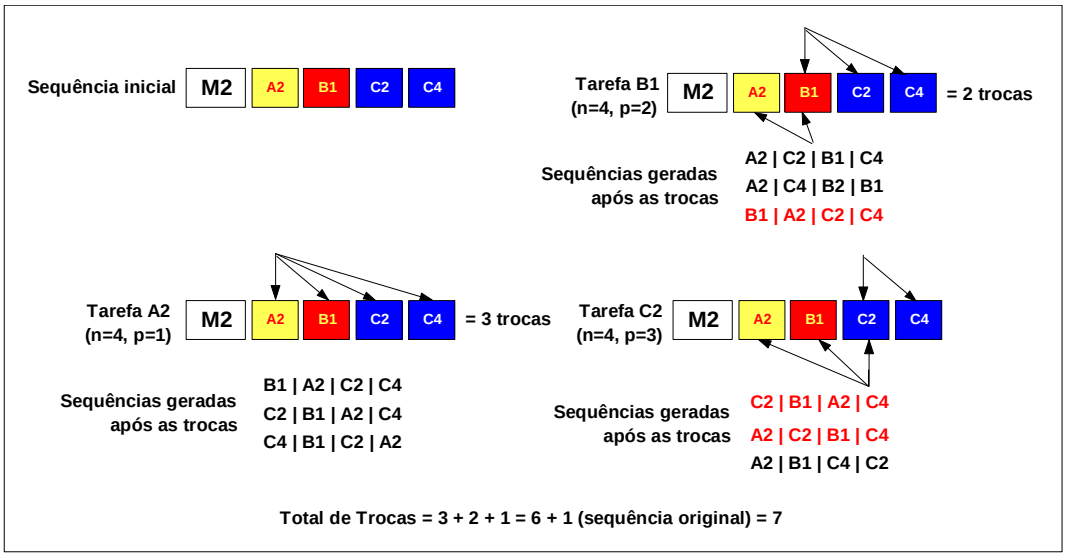


Figura 14 – Mecanismo de Troca de Tarefas

A representação matemática para o cálculo acima seria a descrita abaixo:

$Tr = 1 + \sum_{p=1}^{n} (n - p)$, onde:

Tr Total de trocas na fila do recurso
n nº de tarefas da fila
p posição da tarefa atual

Tt = Tr1 * Tr2 * **Trn** onde **Tr 1..n** - total de trocas de cada recurso.

O chamado salto de uma região para outra no espaço de soluções seria obtido mediante um nº determinado de trocas válidas em todos os recursos e, a partir daí, aplicando-se a lógica exposta acima.

A figura 8 ilustra alguns passos de uma busca no espaço de soluções de um problema hipotético qualquer. A cada salto uma nova região é definida e alguns elementos desta são analisados.

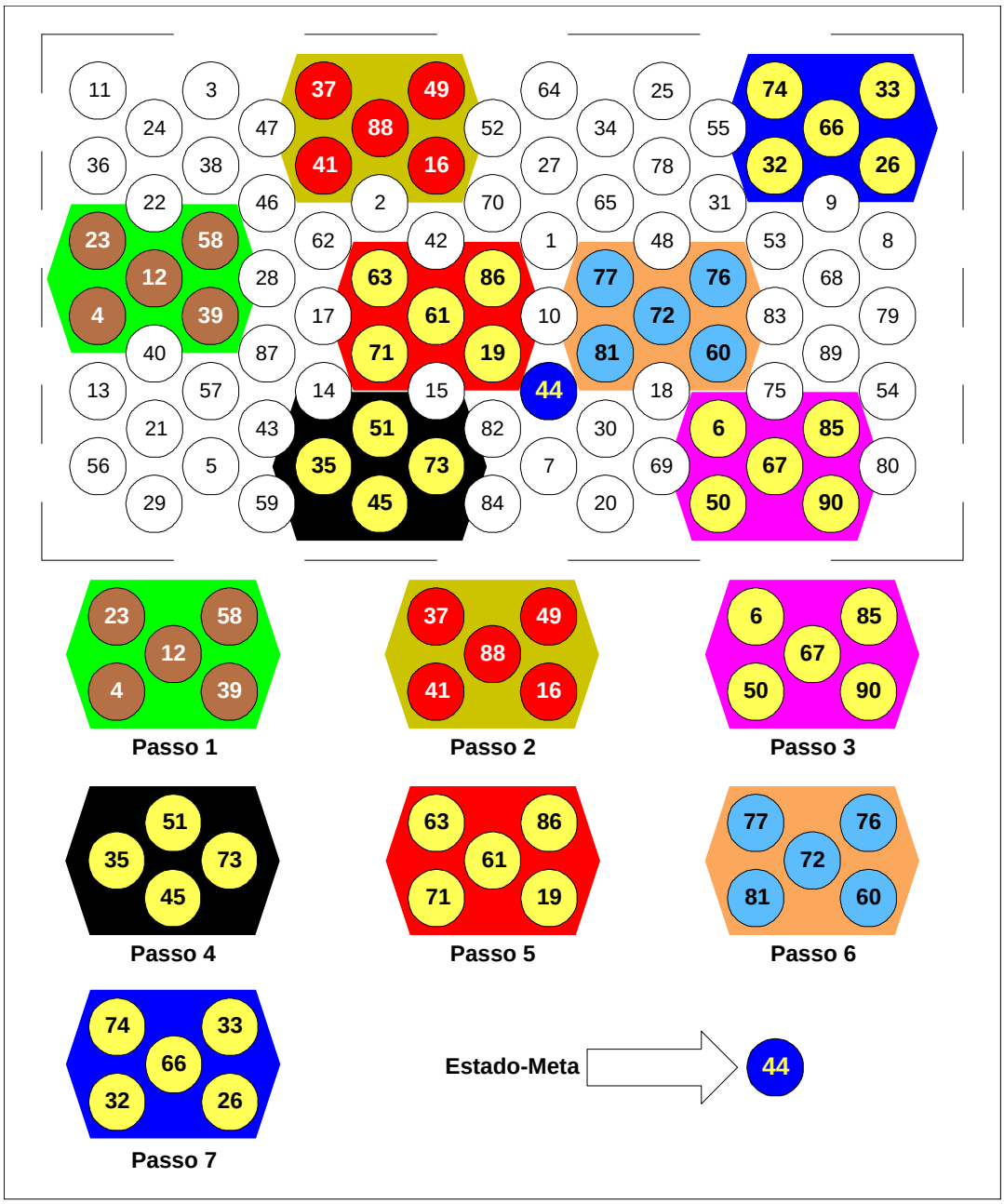


Figura 15 – Etapas do processo de busca no espaço de soluções de um problema

4.4. Mecanismo de Cálculo do Makespan

O cálculo do makespan consiste em medir o total de tempo necessário para se executar todas as tarefas alocadas nos recursos. O algoritmo abaixo descreve os passos para a obtenção do mesmo:

```
Algoritmo CalculaMakeSpan (Recursos: ListaDeRecursos): inteira
Variável  TudoPronto: lógica
Início
    TudoPronto = falso

    //Inicializa os valores e flags dos recursos e tarefas
    Para I = 1 Até Recursos.Quantidade Faça
        Recursos[i].TarefaAtual = 0
        Recursos[i].Pronto = falso
        Recursos[i].Valor = 0
        Para j = 1 Até Recursos[i].Tarefas.Quantidade Faça
                Tarefa.Pronto:= false
                Tarefa.Valor = 0
            Fim_Para
    Fim_Para

    //repete até que todos os recursos tenham sido processados
    Enquanto Não ( TudoPronto ) Faça
        TudoPronto = verdadeiro
        Para I = 1 Até Recursos.Quantidade Faça
            Recurso:=Recursos[i]
            Se Não ( Recurso.Pronto ) Então
                Para j = ( Recurso.TarefaAtual + 1 ) Até Recurso.Tarefas.Quantidade Faça
                    Tarefa = Recurso.Tarefa[j]
                Se Não Existe( Tarefa.Antecessor )  Então
                        Tarefa.Valor = ( Recurso.Valor +Tarefa.Duração )
                        Tarefa.Pronto = verdadeiro
                        Recurso.Valor = Tarefa.Valor
                        Recurso.TarefaAtual = ( Recurso.TarefaAtual + 1 )
                Senão Se Tarefa.Antecessor.Pronto Então
                        Tarefa.Valor =ValorMáximo ( Recurso.Valor, Tarefa.Antecessor.Valor ) +Tarefa.Duracao
                        Tarefa.Pronto = verdadeiro
                        Recurso.Valor =Tarefa.Valor
                        Recurso.TarefaAtual:= (Recurso.TarefaAtual + 1 )
                Senão
                    Fim_Para
                    Fim_Se
                    Se ( Recurso.TarefaAtual = Recurso.Tarefas.Quantidade ) Então
                        Recurso.Pronto = verdadeiro
                    Fim_Se
                Fim_Para
                TudoPronto = TudoPronto E Recurso.Pronto
            Fim_Se
        Fim_Para
    Fim_Enquanto
    Para j = 1 Até Recursos[i].Tarefas.Quantidade Faça
        Resultado = ValorMáximo (Resultado, Recurso[i].Valor)
    Fim_Para
Fim
```

Tabela 3 – Pseudo-Código do Cálculo do Makespan

Logo abaixo são apresentadas a melhor solução encontrada para o problema (3, 10) JSP apresentado nos capítulos anteriores, a execução do algoritmo passo-a-passo tanto através de um teste de mesa como graficamente:

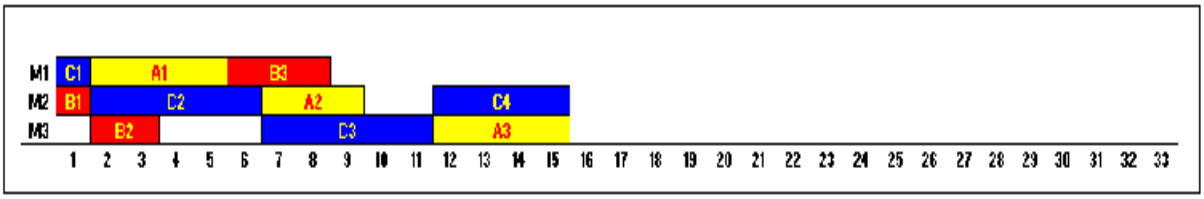


Figura 16 – Melhor Sequenciamento das Tarefas do caso (3,10) JSP

Iteração	Recurso (r)							
	Nome	Avaliação Completa	MkSp	Tarefa Atual (n)				
				Nome	Avaliação Completa	MkSp	Duração	Tarefa Anterior (n-1)
1	M1	N	1	C1	S	1	1	-
2	M1	N	5	A1	S	2	4	-
3	M1	N	5	B3	N		3	B2
4	M2	N	1	B1	S	1	1	-
5	M2	N	6	C2	S	1	5	C1
6	M2	N	8	A2	S	8	3	A1
7	M2	N	8	C4	N		4	C3
8	M3	N	3	B2	S	3	2	B1
9	M3	S	11	C3	S	11	5	C2
10	M1	S	8	B3	N	8	3	B2
11	M2	S	15	C4	N	15	4	C3

Makespan da Sequência		
Recurso	Valor Medido	Maior Valor
M1	8	8
M2	15	15
M3	11	15

Tabela 4 – Cálculo do Makespan para a melhor seqüência do caso (3, 10) JSP

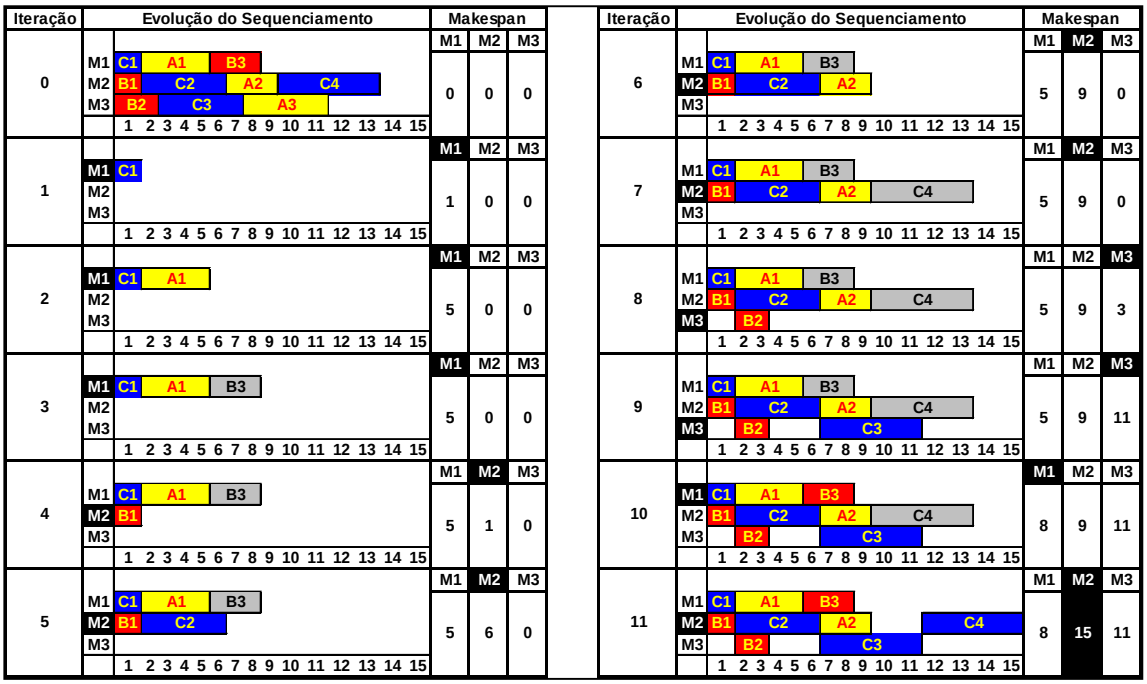


Figura 17 – Etapas do Cálculo do Makespan do Caso (3,10) JSP

A representação matemática para o cálculo acima seria a descrita abaixo e sua execução foi descrita nos exemplos anteriores:

$$MkSp \Rightarrow \sum_{r=1}^{r} \text{Max} \left[\sum_{n=1}^{n} \left(\text{TmpTar n, r} + \text{Max}(\text{TmpTar a , TmpTar n -1, r}) \right) \right]$$

onde:

Mksp maior valor de makespan obtido para a seqüência de tarefas
Σr somatória de valores dos recursos
Σn somatória de valores de tempos das tarefas do recurso
TmpTar n,r - tempo da tarefa n do recurso r
TmpTar a - tempo da tarefa anterior a tarefa n
TmpTar n - 1,r - tempo da tarefa precedente a tarefa n do recurso r

5. Exemplo de Aplicação

Conforme o exposto nos capítulos anteriores, o problema JSP caracteriza-se por uma explosão combinatória do nº de possibilidades de soluções para um problema, tornando-se inviável a resolução do mesmo pelos métodos tradicionais.

Normalmente a área de planejamento da produção de empresas de manufatura, trabalha com cenários bastante complexos como, por exemplo, 400 ou mais tarefas a serem alocadas em 40 recursos bem como um conjunto considerável de restrições envolvidas.

Apenas a título ilustrativo, a figura abaixo mostra uma estrutura de produto hipotética e seu respectivo processo produtivo refletindo as dependências entre seus elementos (tarefas).

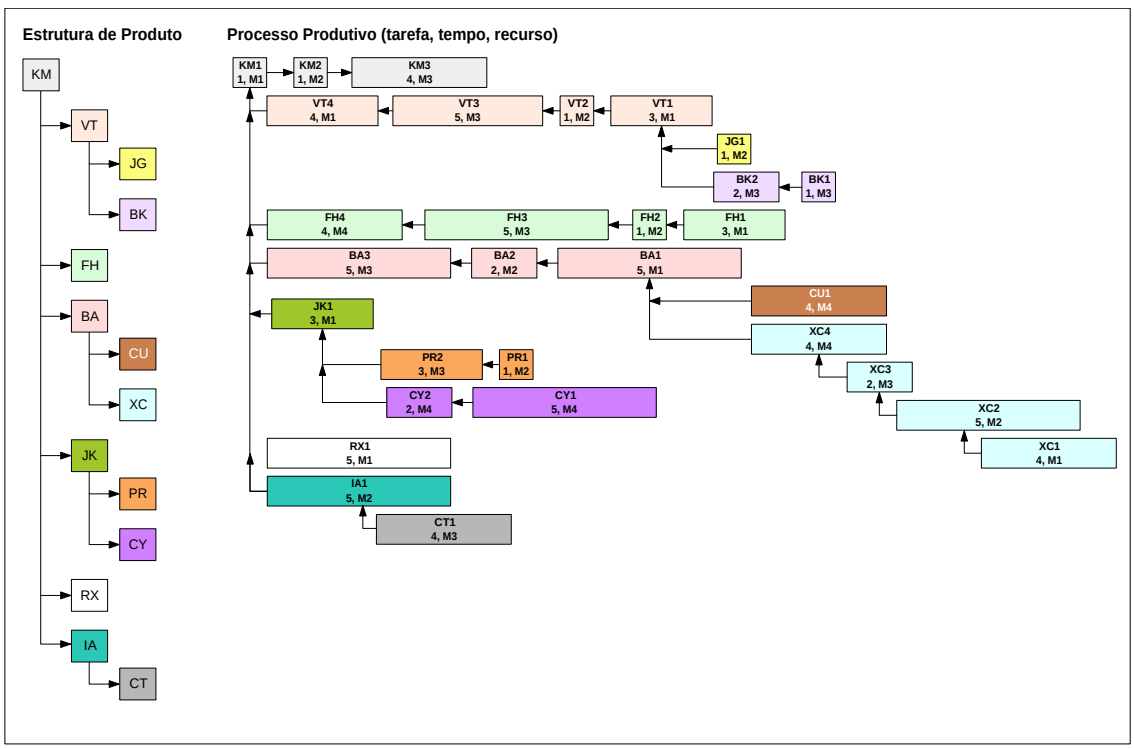


Figura 18 – Exemplo de Estrutura de um produto e seu processo produtivo

O caso de estudo que será apresentado no próximo capítulo não levará em conta todas as restrições e nuances envolvidas no tema, dada a sua complexidade adquirida, no entanto, apesar do

porte do problema ser considerado relativamente pequeno, acredita-se que este caso tenha cumprido o seu papel no sentido de possibilitar a aplicação de todos os conceitos expostos até o momento.

5.1. Caso de Estudo

Logo abaixo está a relação das tarefas e seus respectivos recursos definidos e logo a seguir uma explicação a respeito:

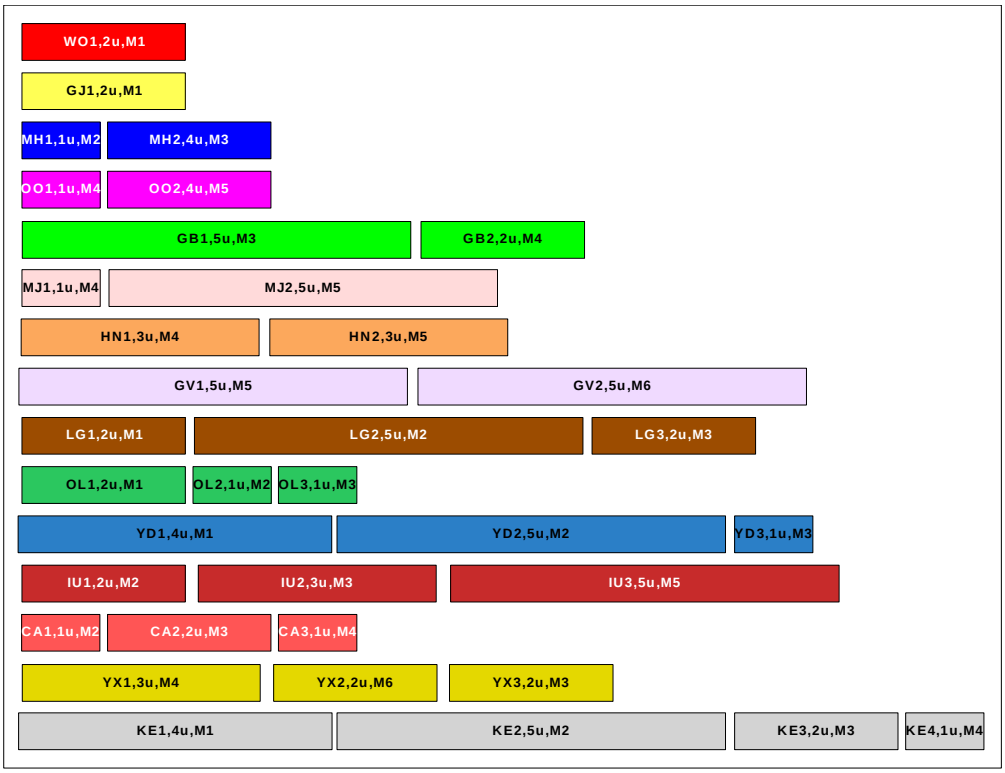
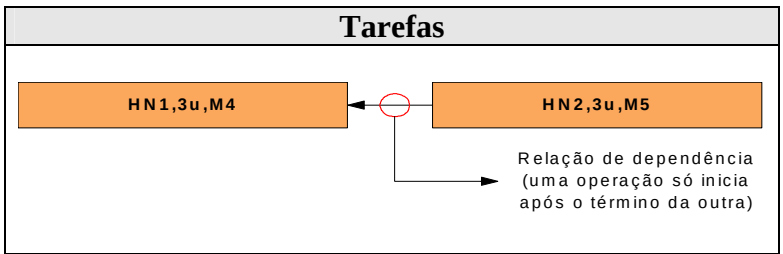


Figura 19 – Tarefas do Caso (6, 36) JSP

Cada tarefa acima possui uma identificação, tempo e local de execução, sendo que algumas possuem dependências com outras tarefas. Tome-se como exemplo a tarefa HN que possui 2 operações, sendo que a tarefa HN1 só iniciará após o término da tarefa HN2. Abaixo uma exemplificação desta tarefa:



Nome	Tempo	Recurso
HN1	3	M4
HN2	3	M5

Figura 20 – Tarefa Exemplo do Caso (6, 36) JSP

Ao analisarem-se todas as tarefas, pode-se notar que muitas irão competir pelos mesmos recursos, além da própria restrição de precedência de operações entre algumas delas.

Com base nesta lista de tarefas, é possível montar-se a Fila de Prioridade de Tarefas dos Recursos, desconsiderando inicialmente qualquer critério de avaliação da ordem de uma tarefa em seu recurso. A figura abaixo ilustra o exposto acima:

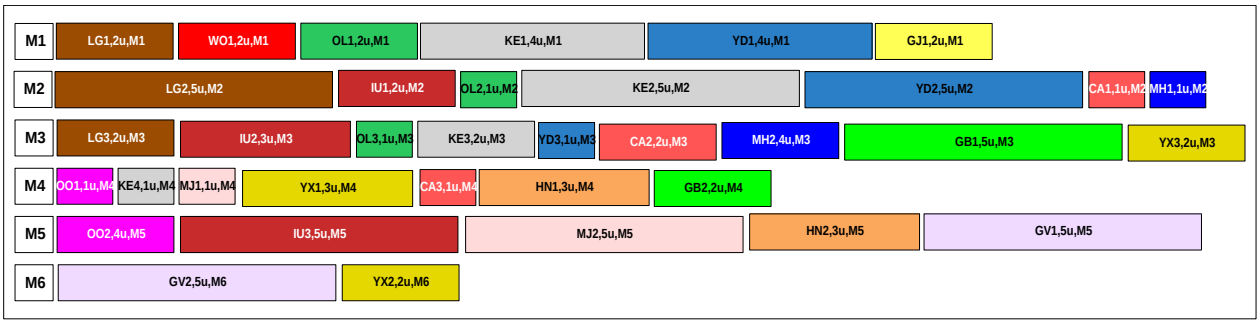


Figura 21 – Fila de Tarefas dos recursos do Caso (6, 36) JSP

Para se atingir o objetivo final que é um sequenciamento de tarefas com o menor tempo de execução possível, proporcionando assim um máximo de aproveitamento dos recursos produtivos, são necessários os seguintes requisitos:

- Uma ou mais técnicas de busca
- Uma boa estratégia de controle
- Uso do conhecimento do domínio do problema

Os dois primeiros itens já foram discutidos nos capítulos anteriores. Quanto ao último será explicado logo a seguir.

Quanto às tarefas, percebe-se que algumas possuem uma dependência entre si, sendo iniciadas somente após o término das quais dependem. Sendo assim, pode se afirmar que:

- Quanto mais cedo uma tarefa que possua dependentes iniciar melhor será. Se esta tarefa for a última da fila do recurso ao qual deve ser alocada, conseqüentemente as demais serão executadas após esta, resultando num aumento do makespan.

- Quanto mais próximas estiverem as tarefas que possuam dependentes, menor será o makespan.

Quanto aos recursos, as seguintes considerações devem ser levadas em conta:

- Existem 2 tipos de recursos num ambiente fabril: os gargalos e os não gargalos. [7]

- Um recurso gargalo é aquele que possui uma demanda (somatório do tempo das tarefas nele alocadas) muito próxima ou igual a sua disponibilidade (total de horas disponíveis num período). [7]

- Uma hora ganha num recurso corresponde a uma hora ganha em todo o sistema (conjunto de recursos do ambiente de manufatura).[7]

Portanto, fica claro que uma boa estratégia aplicada ao(s) recurso(s) gargalo(s) resultará num incremento da eficiência de todo o sistema ou, analisando sob outra ótica, haverá uma grande possibilidade de uma aproximação com o estado-meta.

Os mecanismos e as estratégias foram descritos no capítulo 4 e suas implementações são discutidas no capítulo 6.

5.2. Implementação e Procedimentos

Como foi utilizado um exemplo puramente conceitual para o desenvolvimento do método, não houve a necessidade de coleta de dados numa empresa, pois o caso escolhido atendia aos propósitos gerais do trabalho. Note-se que mesmo sendo uma simulação, é possível com pouco conhecimento sobre o funcionamento de um sistema de manufatura gerar um procedimento que reproduz um sistema real de forma simplificada porém lógica.

Para a implementação do problema foi utilizado o sistema Borland Delphi 7 Enterprise Edition por se tratar de uma ferramenta RAD bastante poderosa e amigável, além de haver uma experiência considerável no uso da mesma.

Algumas ferramentas específicas também foram utilizadas para a elaboração tanto dos diagramas e fluxos como as ilustrações presentes neste documento, tais como: Flow Chart e Model Maker etc.

Apesar de existirem diversos algoritmos para aplicação a casos deste gênero, optou-se por desenvolver um método próprio fundamentado nos conceitos expostos nos capítulos 2 e 3, sendo este um dos objetivos de um trabalho de graduação, gerar o aprendizado de novas técnicas utilizando conhecimentos adquiridos durante o curso.

Para a fase de modelagem do problema foi utilizada, quando conveniente a UML para a geração do diagrama de classes [10, 11] e para as estruturas de dados utilizou-se as classes disponíveis pela Borland, apesar de existirem inúmeras boas bibliotecas de terceiros disponíveis que abrangeriam as necessidades [9] no entanto, o foco do problema não se faz nas ferramentas e sim no conhecimento gerado.

Este método poderia ser implementado em qualquer ferramenta de programação já que o mesmo está fundamentado em conceitos. Sendo assim, apenas a título ilustrativo, segue abaixo o diagrama de classes da aplicação desenvolvida e logo a seguir a tela da aplicação.

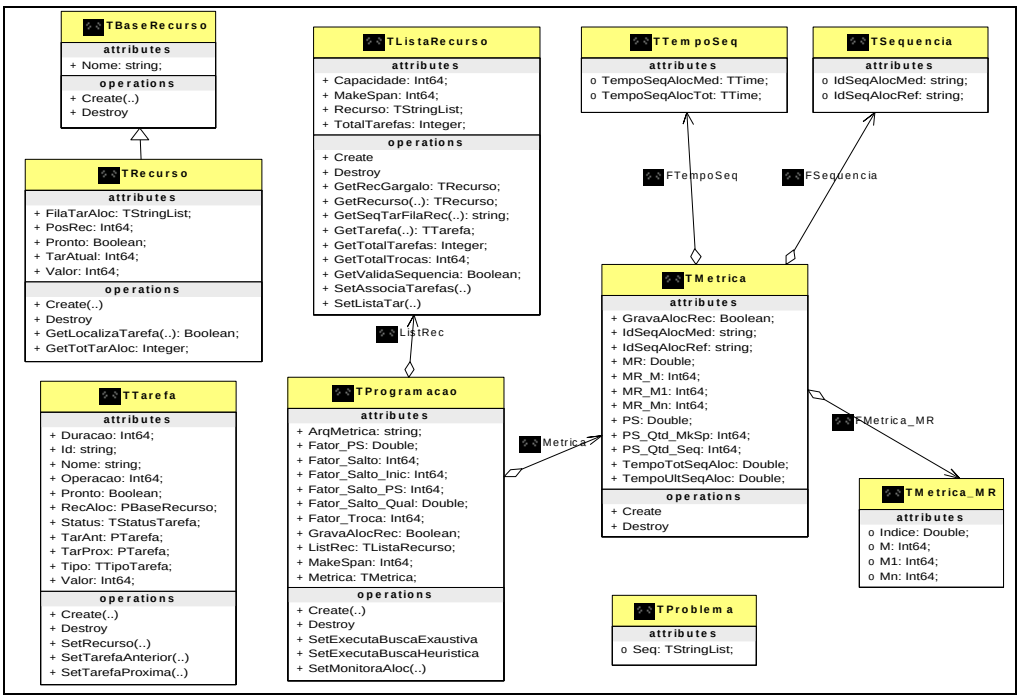


Figura 22 – Diagrama de Classes da Aplicação

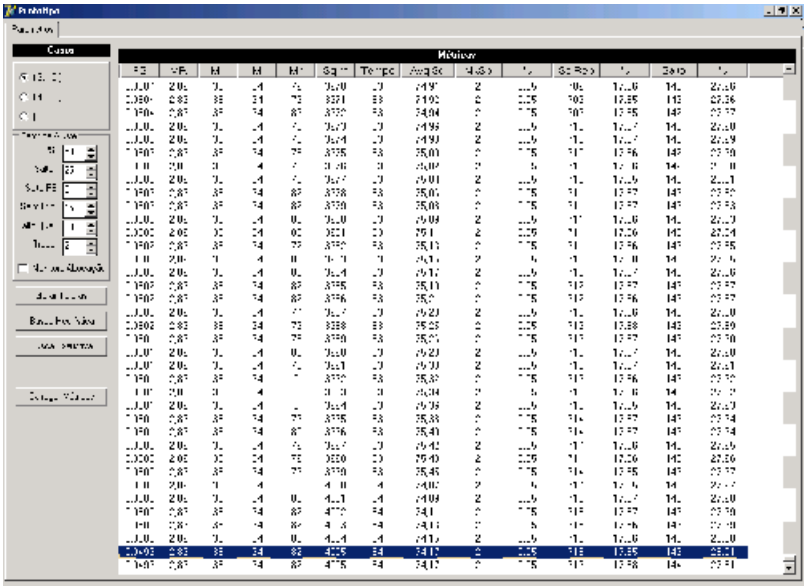


Figura 23 – Tela da Aplicação desenvolvida

Como todos os principais mecanismos presentes no método já foram descritos no capítulo 4, seja através de pseudo-código ou mesmo fluxogramas não haverá a necessidade de um detalhamento tendo em vista a relativa simplicidade no seu desenvolvimento.

5.3. Resultados Obtidos

Para que se pudesse aferir com precisão a qualidade dos resultados obtidos pelo método proposto, através da execução de uma Busca Heurística, foi aplicada uma Busca Exaustiva ao problema, permitindo assim uma análise de todo o conjunto de soluções possíveis para o mesmo.

Abaixo está a tabela comparativa dos resultados de ambas as buscas mencionadas e logo a seguir dois momentos da Busca Heurística:

Métricas / Buscas	Exaustiva	Heurística
Nº de soluções avaliadas	2.160.000	≈ 4000
Tempo de análise	≈ 20Hs	54s
Melhor Makespan	32	34

Tabela 5 – Comparativo entre as Buscas Exaustiva e Heurística

Início da busca

PS	MR	M1	M	Mn	Sq nº	Tempo
100,...	0,00	35	35	35	1	0
50,0...	0,00	35	35	35	2	0
33,3...	0,00	35	35	35	3	0
50,0...	2,86	35	34	34	4	0
40,0...	2,86	35	34	39	5	0
33,3...	2,86	35	34	41	6	0
28,5...	2,86	35	34	34	7	0

Fim da busca

PS	MR	M1	M	Mn	Sq nº	Tempo
0,0500	2,86	35	34	76	3997	53
0,0500	2,86	35	34	76	3998	53
0,0500	2,86	35	34	76	3999	53
0,0500	2,86	35	34	0	4000	54
0,0500	2,86	35	34	82	4001	54
0,0500	2,86	35	34	82	4002	54
0,0500	2,86	35	34	82	4003	54

Figura 24 – Dois momentos da Busca Heurística para o Caso (6, 36) JSP

Antes de serem comentados os resultados, segue abaixo uma breve explicação sobre o significado das colunas mais relevantes contidas na lista da figura acima:

- PS : porcentagem de sucesso
- MR : melhoria relativa é um indicador do grau de evolução entre o makespan da 1ª solução e o makespan da solução considerada melhor até o momento
- M1 : makespan da primeira solução analisada
- M : melhor makespan medido entre todas as soluções até o momento
- Mn : makespan medido da solução avaliada no momento
- Sq nº : indica a quantidade de seqüências avaliadas
- Tempo : indica o tempo (em segundos) decorrido até o momento

A estratégia adotada no início obteve um makespan muito próximo de uma ótima solução para o problema. Em apenas quatro etapas foi possível chegar-se ao "valor definitivo" para o caso em questão, demonstrando que a segunda estratégia, aplicada ao gargalo, foi bastante eficiente (maiores detalhes sobre as estratégias, consultar os capítulos 4 e 5) e o tempo computacional necessário foi bastante satisfatório.

O gráfico logo abaixo ilustra o espaço de soluções para o caso (6, 36) JSP. A linha em azul representa todas as possibilidades de solução para o problema, obtidas através de uma busca exaustiva, e a linha em vermelho representa todas as boas soluções encontradas ao longo desta busca.

Nota-se que além de existirem poucas boas soluções, foi necessário percorrer-se praticamente todo o espaço de solução para encontrar-se a "melhor boa solução" ou mais comumente conhecida como "subótima".

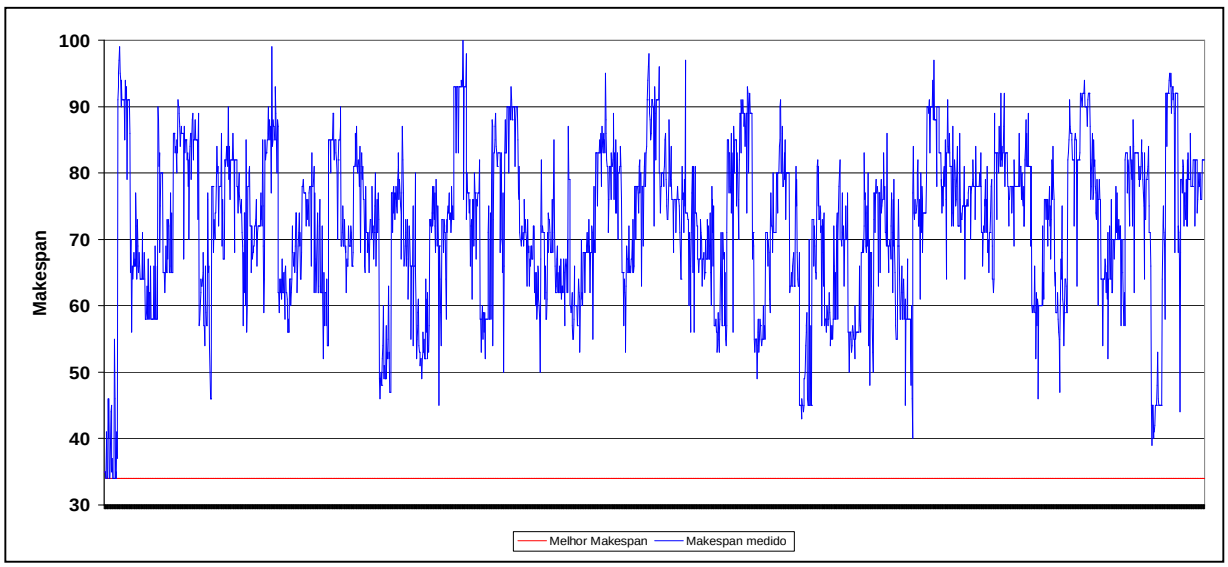


Figura 25 – Gráfico da Busca Heurística do caso (6,36) JSP

Devido ao elevado nº de análises geradas, não será possível ilustrar graficamente os resultados obtidos pela Busca Exaustiva, por comprometer a sua legibilidade e também pela dificuldade em acessar o conteúdo do arquivo, já que este ultrapassou 2GB de tamanho.

6. Conclusões

Tão importante quanto encontrar uma solução satisfatória para um problema é adotar uma abordagem sistemática e científica para tratá-lo, de forma a assegurar que nenhum aspecto relevante seja esquecido.

Não existem soluções definitivas para tratar um problema, qualquer que seja a sua natureza, no entanto, a partir do emprego de conceitos e técnicas cientificamente comprovados, muitas vezes sendo uma combinação de uma ou mais técnicas, é possível chegar-se a uma solução desde que considere-se não apenas aspectos inerentes a computação ou técnicas empregadas mas, principalmente, em relação ao domínio do problema em questão.

Pelos resultados obtidos ficou comprovada a eficiência do método proposto e implementado, abrindo perspectivas para um aperfeiçoamento do mesmo através da incorporação de outras técnicas /conceitos disponíveis. É possível observar que mesmo problemas de maior complexidade dependem não de uma lógica melhor elaborada mas de heurísticas mais eficiente ou mesmo de sistema computacionais mais potentes. O problema exemplo aqui mostrado já incorpora de forma objetiva alguns dos principais aspectos do problema JSP.

De uma maneira mais geral e acadêmica pode-se concluir que o uso de técnicas heurísticas na resolução de problemas constitui-se num vasto campo de possibilidades e desafios, desde que acompanhadas de aspectos do domínio do problema para as quais estão sendo empregadas.

8. Referências Bibliográficas

[1] PINEDO, M.; CHAO, X.: ***Operations Scheduling With Applications in Manufacturing and Services***; Irwin McGraw-Hill, 1999

[2] WALTER, C; ***Planejamento e Controle da Produção – PCP***. [S.l.:s.n.], Apostila de aula – Mestrado em Engenharia de Produção – UFRGS, 1999

[3] HAX, A.; CANDEA, D.; ***Production and Inventory Management***, Englewood Cliffs, N.J., Prentice-Hall, 1984

[4] PACHECO, R.F. e Santoro, M.C. – ***Proposta de Classificação Hierarquizada dos Modelos de Solução para o Problema de Job Shop Shceduling*** – GESTÃO E PRODUÇÃO – Revista do Deptº de Engenharia de Produção – Universidade São Carlos – Abril de 1999

[5] Garey, M.R.; Johnson, D.S.; Sethi R. ***Complexity of Flow-Shop and Job-Shop Scheduling***, Mathematics of Operations Research 1976

[6] Rich, E.; Knight, K. - ***Inteligência Artificial***, Makron Books / 1994,

[7] Corrêa, H.L.; Gianesi, I.G. N.; Caon, Mauro ***Planejamento, Programação e Controle da Produção,*** Atlas, 2001

[8] Luna , H.P.L.; Goldberg, Marco César - ***Otimização Combinatória e Programação Linear - Modelos e Algoritmos***, Campus / 2000,

[9] Bucknall, J. - ***Algoritmos e Estruturas de Dados com Delphi***, Berkeley / 2002,

[10] Furlan, J.D.- ***Modelagem de Objetos através da UML***, Makron Books / 1998,

[11] Boratti , I.C. - ***Programação Orientada a Objetos usando Delphi***, Visual Books / 2002,

This document was created with Win2PDF available at http://www.daneprairie.com.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.