# CAPÍTULO III
# Programação da Produção

Este capítulo pretende fazer uma pequena abordagem ao enorme mundo da programação da produção. Esta actividade, que é normalmente designada em inglês por *Production Shceduling* também é muitas vezes designada em português por "escalonamento" da produção e por "calendarização" da produção. Pode-se dizer, de uma forma simplista, que programar a produção é definir numa escala de tempo os instantes de início e conclusão do processamento dos lotes/peças nos diversos recursos do sistema produtivo. A programação da produção é das tarefas de mais baixo nível no quadro do planeamento e controlo da produção de um sistema produtivo pois está directamente ligada ao espaço fabril e com um carácter de curto prazo. Para melhor situar este tipo de tarefas, elas enquadra-se no nível quatro do planeamento e controlo da produção denominado controlo fabril (ver capítulo II). Neste capítulo ênfase é dada aos problemas de programação da produção e/ou programação sequencial desde o problema de máquina única às linhas e oficinas de produção.

## 3.1 Introdução

Em sistemas produtivos de bens, as ordens de produção são envidas para o espaço fabril com as respectivas datas de entrega e planos de processo (inclui informação como as sequências de fabrico, que máquinas utilizar, tempos de processamento, etc.). Essas ordens tem de ser processadas nos postos de trabalho nas sequências previstas. O processamento das ordens é muitas vezes atrasado porque os postos de trabalho estão ocupados ou porque entretanto chegaram ordens de fabrico com maior prioridade. Há também o problema das avarias inesperadas de máquinas que podem por em causa o cumprimento dos programas. Outros problemas como tempos de processamento maiores do que o previsto podem produzir atrasos. Como se isto não chegasse, ainda temos a interacção com outras funções de gestão como por exemplo o planeamento das necessidades de materiais (MRP). Se os materiais necessários não estiverem disponíveis nos momentos previstos também tem como consequências atrasos e programação da produção ineficiente.

A programação da produção tem tido por parte da comunidade académica e cientifica uma grande popularidade nas últimas décadas. É das áreas com mais publicações em conferências e em revistas da especialidade mas é no entanto das áreas onde há talvez mais distância entre aquilo que se investiga e aquilo que se aplica na prática industrial. Esta classe de problemas, embora de relativa simplicidade em termos de formulação e de visualização do que é requerido, apresenta-se de extrema dificuldade em termos de encontrar a solução óptima para os problemas mais comuns. A esmagadora maioria dos problemas reais de programação da produção são por natureza muito complexos e com resolução muito difícil, em termos da solução óptima. Não se quer aqui dizer que os problemas de programação da produção do dia a dia das nossas empresas não seja resolvidos. O que se quer dizer é que as soluções encontradas por elas não são as soluções óptimas, são apenas soluções que vão resolvendo os problemas com a eficiência possível.

### *3.1.1 Notação adoptada*

Convém estabelecer aqui alguns atributos das entidades. Alguns deles existem antes de serem processados e outros só passam a ser conhecidos depois de serem processados. Para os primeiros vamos usar uma notação baseada em letra minúscula e para os segundos letra maiúscula. Para a informação existente antes do processamento das entidades temos:

$t_i$ - Tempo de processamento para a entidade *i*.

$r_i$ – O instante a partir do qual a entidade *i* está disponível para processamento.
$d_i$ - Data de entrega da entidade *i* - Ponto no tempo a que a entidade *i* deve estar concluída.

Quanto à informação que só é obtida depois de processada a programação temos:

$C_i$ – Instante em que a entidade *i* acabou de ser processada (*completion time*).
$F_i$ - Tempo de percurso (*Flowtime*) - Tempo que a entidade i demora no sistema: Fi = Ci - ri
Ai - Atraso - Tempo em que o fim do processamento da entidade *i* excedeu a data de entrega: $A_i = C_i - d_i$

O tempo de percurso de uma entidade é o tempo que esse entidade espera entre a sua chegada ao sistema e a sua partida do sistema. O atraso representa a conformidade do programa de produção com uma dada data de entrega, é igual à diferença entre a data de conclusão da entidade e a data de entrega prevista. É importante notar que o valor do atraso toma valores negativos se o processamento da entidade for concluído antes da data de entrega. Muitas vezes existe custo associado aos atrasos positivos mas normalmente não há nenhum beneficio associado ao atraso negativo.

Além das notações referidas até aqui convém incluir as notações usadas na classificação dos problemas de programação da produção. Vários tipos de notações e classificações aparecem na literatura mas algumas parecem recolher mais consenso e como tal serão aqui adoptadas. Um sistema geral de programação da produção pode ser classificado por:

*A/B/C/D*

onde: *A* representa o número de entidades/lotes/peças,
*B* representa o número de máquinas,
*C* representa o tipo de fluxo (F - linha e G - oficina),
D representa a medida de desempenho a optimizar.

Assim, a título de exemplo, um problema de programação da produção definido por n/4/L/Fmed diz respeito a uma linha de fabrico com 4 máquinas e qualquer número de lotes havendo a necessidade de encontrar a solução que conduz ao menor tempo de percurso médio.

O termo **entidade** será sempre usado para se referir ao que flui no sistema, quer se trate de lotes, peças isoladas, componentes ou produtos. O termo **máquina** será usado para se referir a qualquer tipo de recurso.

### *3.1.2 Medidas de desempenho*

Os programas de produção são normalmente avaliados por medidas de desempenho. De seguidas são apresentadas algumas medidas de desempenho mais típicas e que são as que serão normalmente usadas neste capítulo para avaliar as soluções:

1. *Tempo de percurso médio:* $F_{med} = \overline{F} = \frac{1}{n}\sum_{i=1}^{n} F_i$

2. *Atraso médio:* $A_{med} = \overline{A} = \frac{1}{n}\sum_{i=1}^{n} A_i$

3. *Tempo de percurso máximo:* $F_{\max} = \max_{1 \le i \le n}\left\{F_i\right\}$

4. *Atraso máximo:* $A_{\max} = \max_{1 \le i \le n}\left\{A_i\right\}$

5. *Numero de entidades atrasadas:* $N_a = \sum_{i=1}^{n} \partial(A_i)$

*onde* $\partial(x)=1$ *se* $x>o$

*e* $\partial(x)=0$ *caso contrário*

### *3.1.1 Complexidade do Problema*

O caso mais simples deste tipo de problemas é o problema da ordenação de n entidades (lotes, peças, produtos ou componentes) em máquina ou processador único (n/1//) para a maioria das medidas de desempenho apresentadas neste capítulo. O número de soluções possíveis é neste caso de n!. Por outro lado o caso mais complexo é o caso de um oficina de fabrico com m máquinas e *n* entidades *(n/m/G/ )*. Neste último caso o número de soluções possíveis é de $(n!)^m$. A tabela seguinte mostra de que forma varia o número de sequências possíveis com a variação do número de entidades e número de máquinas numa oficina de fabrico.

| Número de entidades (n) | Número de máquinas (m) | número de soluções |
|---|---|---|
| 5 | 1 | 120 |
| 5 | 3 | 1.7 milhões |
| 5 | 5 | 25.000 milhões |
| 10 | 10 | $3.96*10^{65}$ |

Como se pode observar pela tabela acima, o número de sequências cresce para números extremamente elevados mesmo para problemas relativamente pequenos de programação da produção. Como não existe relação matemática entre as diferentes soluções do problema, a única forma de se encontrar a solução óptima passa pela enumeração completa de todas as soluções. A partir da enumeração completa das soluções, basta comparar o desempenho de cada uma e seleccionar a melhor. Apenas para dar a ideia da complexidade de tais problemas, se um computador demorar 0,000001 segundos para avaliar cada solução, seriam necessários $3.96*10^{59}$ segundos, ou seja $1,26*10^{35}$ milhões de biliões de séculos para avaliar todas as soluções.

Alguns problemas de programação da produção são relativamente fáceis de resolver, podem ser formulados segundo programação linear e a solução óptima é obtida usando algoritmos existentes. Outros problemas simples podem ser resolvidos com algoritmos que também encontram a solução óptima sendo normalmente chamados algoritmos de tempo polinomial. Como para este tipo problemas há algoritmos de tempo polinomial, então mesmo nos casos em que estes problemas apresentam grandes dimensões (milhares de máquinas e milhares de lotes) podem igualmente ser resolvidos por computador em relativamente pouco tempo.

Contudo, a grande maioria dos problemas de programação da produção são intrinsecamente muito difíceis. Esse tipo de problemas é denominado por *NP hard problems* cuja complexidade cresce exponencialmente com a dimensão do problema e encontrar a solução óptima torna-se praticamente impossível para problemas de razoáveis dimensões. Este tipo de problemas não pode ser formulado usando programação linear nem existem regras ou algoritmos simples que garantam encontrar a solução óptima num espaço de tempo limitado de computador (horas ou poucos dias). Assim, para este tipo de problemas não se consegue garantir a solução óptima e apenas se conseguem relativamente boas soluções.

## 3.2 Procedimentos gerais de programação da produção

Embora algumas técnicas sejam apropriadas para este ou aquele problema de programação da produção específico, convém esclarecer que algumas técnicas, além do seu uso geral na PP também são usadas em muitas outras áreas. Uma grande parte dos problemas PP são problemas de optimização combinatória e como tal podem ser resolvidos com as técnicas que

tem sido desenvolvidas para resolver essa classe de problemas. Assim, antes de estudarmos as técnicas de uso específico vamos tentar dar uma visão global das técnicas, heurísticas e algoritmos de uso global.

De uma forma simplista poderemos dizer que há três grandes tipos de problemas de PP: (1) Em primeiro lugar temos os problemas que podem ser formulados através da programação linear e para esses existem algoritmos que encontram a solução óptima. (2) Num outro grupo temos problemas simples para os quais existem algoritmos que garantem a solução óptima em tempo polinomial, ou seja, mesmo para grandes dimensões do problema, a solução é encontrada em tempo útil. (3) Finalmente temos o grupo de problemas, onde infelizmente se encontram os mais comuns, que são intrinsecamente difíceis, conhecidos por problemas NP difíceis. Para este tipo de problemas, não são conhecidos métodos que permitam encontrar a solução óptima em tempo "aceitável".

As técnicas para resolver problemas e programação da produção podem ser do tipo **construtivo** ou do tipo **melhoramento**. No primeiro caso a solução vai sendo construída à medida que o método vai sendo aplicado e no segundo tipo (melhoramento) parte-se de uma solução inicial que vai sendo sucessivamente transformada em melhores soluções à medida que o método ou técnica vai sendo aplicado.

### *3.2.1 Regras de prioridade*

Uma regra de prioridade ou regra de despacho é uma regra que estabelece a prioridade com que serão processadas as entidades que esperam para ser processadas numa máquina. O esquema de prioridade pode estar relacionado com atributos das entidades, atributos das máquinas, ou também relacionado com o tempo actual. Quando uma máquina completa o processamento de uma entidade, é a regra de prioridade que selecciona, da lista de espera, a entidade com a prioridade mais alta.

O número de regras de prioridades existentes é enorme e é de pelo menos várias centenas e não nos interessa cobrir a maior parte delas. É no entanto interessante falar dos diferentes modos como elas podem ser classificadas e focar as regras mais tipicamente usadas.

Uma possível classificação tem como base o facto que umas regras são **estáticas** e outras são **dinâmicas**: As regras de prioridade estáticas são aquelas cujo valor da prioridade é independente do instante de tempo actual. A regra SPT (Shortest Processing Time) em que a entidade com menor tempo de processamento é a que tem maior prioridade é uma regra estática pois o tempo de processamento não varia com o tempo. As regras dinâmicas são regras cujo valor de prioridade varia com o tempo. Vejamos a regra "Folga para processamento" que traduz o tempo que ainda há para processamento até à data de entrega. O valor desta regra varia continuamente com o tempo, por isso é uma regra dinâmica.

Outra forma de classificação tem a ver com o tipo de informação em que se baseia a regra em questão. Por uma lado temos as regras **locais** que apenas levam em linha de conta a informação relativa à fila de espera onde se encontra a entidade respectiva. SPT é um exemplo de uma regra local. Por outro lado temos as regras **globais**. Estas regras levam em linha de conta informação de todo o sistema produtivo. Exemplos desta classe e regras é "Most Work Remaining " onde tem maior prioridade a entidade cujo somatório dos tempos de processamento das operações que ainda falta realizar até ficar completamente pronta.

Muitas regra podem ser encontradas na literatura e a sua apresentação exaustiva não é necessária nestes textos. Apenas com o intuito de dar alguns exemplos mais típicos aqui vão algumas regras de prioridade:

- RANDOM (*Random*): Quando uma máquina fica livre, de todas as entidades na fila de espera respectiva, é seleccionada para processamento uma entidade aleatoriamente. Não há objectivo nenhum em optimizar qualquer medida de desempenho.

- EDD (*Earliest Due Date*): Quando uma máquina fica livre, de todas as entidades na fila de espera respectiva, é seleccionada para processamento a entidade com a data de entrega mais cedo. Esta regra tende em minimizar o atraso máximo entre as entidades da fila de espera. Nos casos de máquina única, esta regra garante encontrar o menor atraso máximo.
- SPT (*Shortest Processing Time*): Quando uma máquina fica livre, de todas as entidades na fila de espera respectiva, é seleccionada para processamento a entidade com menor tempo de processamento nessa máquina. Em problemas de máquina única esta regra garante o menor tempo de percurso médio.
- LPT (*Longest Processing Time*): Quando uma máquina fica livre, de todas as entidades na fila de espera respectiva, é seleccionada para processamento a entidade com maior tempo de processamento nessa máquina.
- FCFS (*First Come First Served*): Quando uma máquina fica livre, de todas as entidades na fila de espera respectiva, é seleccionada para processamento a entidade que mais cedo chegou à fila de espera respectiva. Esta regra garante que nenhuma entidade fica eternamente na fila de espera, ao contrário das duas regras anteriores que em casos extremos não dão esta garantia.
- MWKR (Most Work Remaining): Quando uma máquina fica livre, de todas as entidades na fila de espera respectiva, é seleccionada para processamento a entidade cujo somatório dos tempos de processamentos nas operações a efectuar (nessa e nas próximas máquinas) é maior.
- LWKR (Least Work Remaining): Quando uma máquina fica livre, de todas as entidades na fila de espera respectiva, é seleccionada para processamento a entidade cujo somatório dos tempos de processamentos nas operações por efectuar, é menor.
- MOPNR (Most Operations Remaining): Quando uma máquina fica livre, de todas as entidades na fila de espera respectiva, é seleccionada para processamento a entidade com maior numero de operações por efectuar.

Das muitas outras regras existentes em publicações, vamos incluir algumas, diferentes das anteriores, que são sugeridas por Pinedo e Chao (1999):

- ERD (*Earliest Release Date*): Quando uma máquina fica livre, de todas as entidades na fila de espera respectiva, é seleccionada para processamento a entidade com data mais cedo de entrada no sistema. Esta regra tenta, até certo ponto, minimizar a variação nos tempos de espera.
- MS (*Minimum Slack*): Esta regra é uma variação da regra EDD. Quando a máquina fica livre, de todas as entidades na fila de espera respectiva, é seleccionada para processamento a entidade cujo tempo de folga até à data de entrega é menor. O tempo de folga até à data de entrega é obtido subtraindo, ao tempo que ainda falta até à data de entrega, os tempos que serão ainda necessários para processamento.
- SST (*Shortest Setup Time*): Quando uma máquina fica livre, de todas as entidades na fila de espera respectiva, é seleccionada para processamento a entidade com o menor tempo de preparação da máquina.
- SQNO (*Shortest Queue at the Next Operation*): Esta regra é usada em oficinas. Quando uma máquina fica livre, de todas as entidades na fila de espera respectiva, é seleccionada para processamento a entidade que em seguida irá para a máquina com a menor fila de espera. A medida do tamanho da fila de espera poder ser feita em termos do número de entidades em espera ou em termos do tempo de processamento total em espera.

Na tabela 3.1 encontra-se um resumo de algumas das regras mais típicas classificadas por tipos e com os objectivos normalmente esperados com o seu uso (Pinedo e Chao 1999). Em alguns ambientes industriais algumas destas regras garantem a solução óptima e noutros

casos, infelizmente a maioria, são apenas heurísticas razoáveis que não garantindo a solução óptima garantem soluções rápidas e razoáveis.

**Tabela 3.1 - Pequeno resumo de regras de prioridade.**

| | Regra | Dados | Objectivos |
|---|---|---|---|
| Regras que dependem da data de entrega ou da data de entrada no sistema. | ERD | $r_i$ | Variância dos tempos de percurso |
| | EDD | $d_i$ | Atraso máximo |
| | MS | $d_i$ | Atraso máximo |
| Regras que dependem dos tempos de processamento | LPT | $t_i$ | Carga equilibrada em processadores paralelos |
| | SPT | $t_i$ | Tempo de percurso médio e WIP |
| Diversas | Random | - | Fácil de implementar |
| | SST | $s_{ij}$ | Makespan e tempo de percurso |
| | SQNO | - | Utilização das máquinas |

Em diversos estudos que tem sido feitos ao uso de regras de prioridade em ambientes industriais não se pode dizer que determinada regra de prioridade domina as outras em termos de *makespan*. Contudo a regra com mais sucesso tem sido mais vezes a regra MWKR e algumas das suas variações. Apesar disso, a regra SPT, produz melhores resultados num pequeno número de situações. Em termos de tempo de percurso médio (Fmed) como medida de desempenho, também não se chegou à conclusão de que uma regra dominava as outras, embora SPT e LWKR sejam normalmente mais eficazes do que as outras.

As experiências demonstraram que a programação da produção em oficinas de fabrico baseadas em regras de prioridade é um método praticável para a obtenção de soluções sub-óptimas. Quando o *makespan* é a medida de desempenho a considerar, as regras com mais sucesso são as regras que exprimem a prioridade em relação ao trabalho que ainda é necessário levar a cabo em operações a realizar. Para os problemas de Fmed parece que resultam melhor as regras que dão prioridade às entidades com menor carga de trabalho em operações a efectuar.

### *3.2.2 Técnicas heurísticas*

As heurísticas são técnicas que embora não garantindo a solução óptima, encontram soluções razoáveis de uma forma simples e rápida. Todos nós usamos técnicas heurísticas no dia a dia para encontrar as soluções para os problemas que nos aparecem. Se tivermos de ir às compras a três lojas, podemos decidir por exemplo ir em primeiro lugar à loja que está mais perto, ou àquela cujo volume da compra é menor. A solução óptima pode não ter sido conseguida porque, por exemplo, a primeira loja que decidimos ir estava, naquele momento, cheia de gente. Acabamos por desistir e ir a outra em primeiro lugar. Neste caso utilizamos heurísticas pois a decisão foi rápida e simples mas não conseguimos garantir a solução óptima.

Vamos ver um exemplo de uma heurística aplicada a um problema clássico que é "**o problema do caixeiro viajante**". Na sua formulação clássica, o viajante tem de visitar clientes em cada uma das *n* cidades do negócio. O viajante procura encontrar o trajecto mais curto que o leve a cada uma, e apenas uma, de todas as cidades e voltar ao ponto de partida. Dadas as distâncias entre pares de cidades, a tarefa do viajante é encontrar a trajectória que lhe minimiza a distância percorrida. Vejamos o exemplo da tabela 3.2. em que o viajante tem de as visitar quatro cidades conhecendo as distâncias entre elas.

**Tabela 3.2 – Um exemplo do problema do caixeiro viajante.**

| | Cidade 1 | Cidade 2 | Cidade 3 | Cidade 4 | Cidade 5 |
|---|---|---|---|---|---|
| **Cidade 1** | -- | 4 | 8 | 6 | 8 |
| **Cidade 2** | 5 | -- | 7 | 11 | 13 |
| **Cidade 3** | 4 | 6 | -- | 8 | 11 |
| **Cidade 4** | 5 | 7 | 2 | -- | 3 |
| **Cidade 5** | 10 | 9 | 7 | 5 | -- |

O número de soluções para este problema é de ***n!***, ou seja, 120 soluções possíveis. Para este tamanho do problema não é difícil encontrar a solução óptima, bastando para tal avaliar todas as soluções e escolher a melhor. É claro que para problemas com 1000 cidades não será viável avaliar todas as 1000! soluções possíveis, pelo que outros métodos terão de ser usados. Estamos claramente em presença de um problema da classe NP difícil.

Uma heurística muito simples para o problema do caixeiro viajante é a regra da "**cidade mais próxima ainda não visitada**" (*Nearest Unvisited City*). Este algoritmo segue os seguintes passos:

1. Seleccionar aleatoriamente uma cidade para cidade de origem.
2. Determinar qual a cidade, das ainda não visitadas, que lhe fica mais próxima.
3. Das cidades não visitadas determinar a que fica mais próxima da seleccionada anteriormente.
4. Repetir o ponto 3 até que todas as cidades sejam visitadas.

Assim, para o problema da tabela 3.2, se seleccionarmos a cidade 2 como sendo a cidade de início, temos como cidade mais próxima a cidade 1, seguidamente a cidade 4, depois a cidade 3 e finalmente só nos resta para visitar a cidade 5. A solução resultante é então a sequência 2 1 4 3 5 que traduz um distância percorrida de 5+6+2+11+9 = 33. Embora não haja garantias que esta seja a solução óptima, sabemos que é uma solução razoável.

Esta capacidade de gerar soluções boas de forma rápida torna esta regra atraente para problemas cuja determinação da solução óptima implica custos proibitivos. Claro que a heurística pode ser aplicado várias vezes seleccionando qualquer das cidades para origem obtendo-se assim tantas soluções quantas as cidades do problema. Naturalmente que a melhor de todas essas soluções deverá então ser seleccionada.

O algoritmo foi testado considerando também uma variação com capacidade de ver não só a próxima cidade mas também a próxima da próxima. Isto é, determinar as duas cidades cuja distância total á cidade de referência é mínima. Os resultados indicaram que as soluções estavam dentro de um erro máximo de 10% sobre a solução óptima quando o número de cidades é inferior ou igual a 20. No entanto isto o desempenho do algoritmo deteriora-se se houver grande variabilidade entre as distâncias.

A eficiência da aplicação de métodos heurísticos para a solução de problemas de produção, poderá ser melhorada pelo seu uso repetitivo, quando possível, no mesmo problema, de forma a encontrar diferentes soluções das quais se poderá escolher a melhor.

O problema do caixeiro viajante é importante porque alguns problemas de programação da produção podem ser formulados como um problema desse tipo. Para resolvermos esse tipo de problemas de programação da produção basta formulá-los como problemas do caixeiro viajante e usar os métodos que resolvem o problema do caixeiro viajante para obter a solução.

### *3.2.3 Técnicas de "Branch and Bound"*

As técnicas de *Branch and Bound* são esquemas inteligentes de enumeração completa, ou seja, conseguem mostrar desnecessário a exploração de grupos de soluções provando que não incluem a solução óptima. Este tipo de técnicas garante encontrar a solução óptima mas

para problemas de grandes dimensões o tempo necessário pode ser proibitivo. Para ilustrar os princípios deste tipo de técnicas vamos usar um exemplo de um método desta classe.

Um dos primeiros estudos de investigação abordando a técnica de *branch and bound* foram levados a cabo por **Little et al (1963)**. Este método tem como objectivo encontrar a solução óptima do problema do caixeiro viajante que vimos anteriormente. Trata-se de um método construtivo que vai construindo a solução final pela selecção de pares de cidades. A melhor solução é o trajecto cuja distância percorrida é menor.

Este método começa por calcular o limite mínimo teórico (*lower bound*). Esse *lower bound* dá-nos a certeza que não é possível encontrar nenhuma solução cuja distância percorrida seja inferior a esse valor. Além disso, este método vai gerando limites mínimos para todos os ramos da árvore indicando os ramos mais "promissores". Embora não haja garantia que seja possível encontrar soluções com os desempenhos dos tais *lower bound*s, esses limites mínimos são vitais para a lógica do método.

Para entendermos o método vamos considerar o exemplo da tabela 3.2 que se encontra novamente apresentado na tabela 3.3a. Os passos do método são os seguintes:

a) colocar na coluna **u** os menores valores de cada linha.

b) subtrair esses menores valores a todos os elementos das linhas correspondentes (ver tabela 3.3b com o resultado). Nesta tabela temos os valores que representam a distância adicional que é necessário percorrer se não se escolher o para cuja distância é a menor. A título de exemplo: escolher o par 1-3 custa mais 4 unidades do que escolher o par 1-2.

c) partindo da tabela 3.3b colocar na linha **v** os menores valores de cada coluna.

d) subtrair esses menores valores (**v**) a todos os elementos das colunas correspondentes. Como resultado obtemos a tabela 3.3c.

e) o somatório dos **u**s e dos **v**s corresponde ao limite inferior (*lower bound*) que alguma solução poderia atingir. Onde existem zeros correspondem a pares com as menores distâncias mas pode não ser possível uma solução com combinação desses pares.

f) as casas com zeros, correspondem a pares que, ao ser considerados na solução final, não trazem nenhum acréscimo à distância percorrida. Assim, vamos seleccionar um desse pares como sendo um dos pares possíveis para fazer parte da solução final.

g) dos pares que tem zero na matriz é seleccionado o par que não sendo escolhido incorre no maior custo. Assim, cada zero é etiquetado com a soma do menor da linha e o menor da coluna correspondes. O par escolhido pode ser o par (1-2) ou o par (5-4) visto serem estes os que tem o maior valor na etiqueta. Vamos escolher o par (1-2), neste caso o par (2-1) nunca poderá existir pois se existisse nunca o ciclo seria completo. Agora entra a técnica de ramificação. Dois ramos são considerados: Escolha do par (1-2) $P_{12}$ e a proibição do mesmo par $P_{\overline{12}}$ como indica a figura 3.1. No ramo do par (1-2) ainda não sabemos o valor para o *Lower Bound* (LB), mas no caso da negação do par (1-2) temos um LB de 21 + 4 (etiqueta maior) = 25. O valor de LB indica que, para o ramo em questão, nunca será possível encontrar uma solução com um valor de tempo de percurso total (*Makespan*) inferior a esse valor.

**Tabela 3.3a. P**

| | (1) | (2) | (3) | (4) | (5) | **u** |
|---|---|---|---|---|---|---|
| (1) | -- | 4 | 8 | 6 | 8 | 4 |
| (2) | 5 | -- | 7 | 11 | 13 | 5 |
| (3) | 4 | 6 | -- | 8 | 11 | 4 |
| (4) | 5 | 7 | 2 | -- | 3 | 2 |
| (5) | 10 | 9 | 7 | 5 | -- | 5 |

**Tabela 3.3b. P (reduzida)(linhas)**

| | (1) | (2) | (3) | (4) | (5) |
|---|---|---|---|---|---|
| (1) | -- | 0 | 4 | 2 | 4 |
| (2) | 0 | -- | 2 | 6 | 8 |
| (3) | 0 | 2 | -- | 4 | 7 |
| (4) | 3 | 5 | 0 | -- | 1 |
| (5) | 5 | 4 | 2 | 0 | -- |
| **v** | | | | | 1 |

**Tabela 3.3c. P (reduzida)(colunas)**

| | (1) | (2) | (3) | (4) | (5) |
|---|---|---|---|---|---|
| (1) | -- | $0^4$ | 4 | 2 | *3* |
| (2) | $0^2$ | -- | 2 | 6 | *7* |
| (3) | $0^2$ | 2 | -- | 4 | *6* |
| (4) | 3 | 5 | $0^2$ | -- | *$0^3$* |
| (5) | 5 | 4 | 2 | $0^4$ | -- |

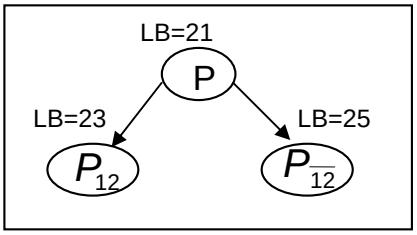



Figura 3.1 Ramificação de opções.

h) O procedimento de redução da matriz pode ser agora levado a cabo. Em $P_{12}$ dado que o par (1-2) faz parte da solução, o par (2-1) tem de ser proibido, mais todos os outros pares que comecem com 1 e todos os outros pares que acabem com 2. Assim temos como resultado a tabela 3.3d para o ramo $P_{12}$ e a tabela 3.3e para o ramo $P_{\overline{12}}$.

i) A cada uma destas tabelas usamos o mesmo procedimento que usamos até aqui. Cada um dos ramos é agora subdividido em 2 ramos e assim por diante.

**Tabela 3.3d.** $P_{12}$

| | (1) | (2) | (3) | (4) | (5) |
|---|---|---|---|---|---|
| (1) | -- | -- | -- | -- | -- |
| (2) | -- | -- | 2 | 6 | 7 |
| (3) | 0 | -- | -- | 4 | 6 |
| (4) | 3 | -- | 0 | -- | 0 |
| (5) | 5 | -- | 2 | 0 | -- |

**Tabela 3.3e.** $P_{\overline{12}}$

| | (1) | (2) | (3) | (4) | (5) |
|---|---|---|---|---|---|
| (1) | -- | -- | 4 | 2 | 3 |
| (2) | 0 | -- | 2 | 6 | 7 |
| (3) | 0 | 2 | -- | 4 | 6 |
| (4) | 3 | 5 | 0 | -- | 0 |
| (5) | 5 | 4 | 2 | 0 | -- |

O cálculo do LB para a ramificação $P_{12}$ é feito a partir da tabela 3.3d. depois de lhe ser feita a redução dos menores valores em cada linha e em cada coluna. Como podem então ver na referida tabela, o único valor a subtrair é o valor 2 correspondente ao menor da segunda linha, dado que todas as outras linhas e todas as colunas terem pelo menos um zero. Desta forma, o valor do LB para a ramificação $P_{12}$ é obtido somando 2 ao valor inicial de 21 (ver figura 3.1).

Neste momento, o ramo correspondente às sequências que incluem o par (1,2), é o que promete melhores soluções pelo que deverá ser o primeiro a ser explorado, não querendo isto dizer que a melhor solução está neste ramificação.

### *3.2.4 Técnicas de procura local*

O esforço computacional requerido para resolver problemas combinatórios cresce enormemente com o tamanho do problema. Embora seja difícil dizer quando é que um problema tem o tamanho típico dos problemas práticos é necessário ter sempre em atenção que as técnicas de optimização são sempre "gulosas" em tempo computacional. Problemas de 5 ou 10 entidades pode ser resolvido rapidamente, mas um problema com 20 entidades pode não ser praticável onde as capacidades computacionais são escassas ou caras ou ainda onde as decisões de programação tem de ser tomadas em minutos. Métodos de sub-optimização ou procedimento heurísticos são adequados para a obtenção rápida de boas soluções para a ordenação de entidades, mas que não são necessariamente óptimas. As técnicas de procura local (NST - *Neighborhood Search Techniques*) permite obter rapidamente, de forma simples e flexível, boas soluções.

Os elementos básicos, na aplicação desta abordagem à programação da produção, são os conceitos de *semente*, de *vizinhança* e de *mecanismo* para geração de vizinhanças. O mecanismo de geração é um método que cria sistematicamente soluções relacionadas com um solução *semente*. Um exemplo de um mecanismo pode ser o da troca entre pares adjacentes. Se a solução semente for: 1,2,3,4,5,6,7,..n-2,n-1,n; então as soluções geradas por troca de pares adjacentes seriam:

**2, 1,** 3, 4, 5, 6, 7, .. n-2, n-1, n ;
1, **3, 2**, 4, 5, 6, 7, .. n-2, n-1, n ;
1, 2, **4, 3**, 5, 6, 7, .. n-2, n-1, n ;
……. ;
……. ;
1, 2, 3, 5, 4, 6, 7, .. **n-1**, **n-2**, n ;
1, 2, 3, 4, 6, 5, 7, .. n-2, **n**, **n-1**

Isto é uma lista de (n-1) soluções distintas, que é chamada a *vizinhança* da semente, para este particular mecanismo de geração. Não será difícil de imaginar outros métodos para criar vizinhanças. Outro mecanismo seria o de introduzir o última entidade n, em diferentes posições da mesma solução semente:

**n,** 1, 2, 3, 4, 5, 6, 7, .. n-2, n-1 ;
1, **n,** 2, 3, 4, 5, 6, 7, .. n-2, n-1 ;
1, 2, **n,** 3, 4, 5, 6, 7, .. n-2, n-1 ;
……. ;
……. ;
1, 2, 3, 5, 4, 6, 7, .. **n**, n-1 ;

O resultado é de novo uma lista de (n-1) soluções. O mecanismo pode afectar o número de soluções da vizinhança. Por exemplo, a vizinhança pode ser criada por um mecanismo que faz a troca entre pares adjacentes ou não, o que será o número de combinações possíveis de pares:

$$Np = \binom{n}{2} = \frac{n!}{(n-2)!*2!} = \frac{n(n-1)}{2}$$

Isto quer dizer que neste caso teríamos n(n-1)/2 soluções na vizinhança. Em geral, dada uma semente e um mecanismo de geração, qualquer solução que pode ser formada da semente, numa única aplicação do mecanismo, é definida como pertencendo à vizinhança da semente.

**Procedimento de NST**

1. Seleccionar uma solução semente avaliando o seu desempenho.
2. Gerar e avaliar as soluções da vizinhança da semente. Se nenhuma das soluções produz melhor desempenho do que a semente, então termina a busca. Doutra forma continuar.
3. Seleccionar a solução da vizinhança com melhor desempenho para ser a nova semente.

Além disto é necessário especificar as seguintes opções:

1. Um método para obter a semente.
2. Um particular mecanismo de geração de vizinhança.
3. Um método para seleccionar a solução que irá ser a próxima semente.

Para clarificar este tipo de técnicas vamos ver o seguinte exemplo. Considere a minimização de Na (número de trabalhos atrasados), no seguinte problema:

| Entidade | Tempo de processamento | Data de entrega |
|---|---|---|
| 1 | 1 | 2 |
| 2 | 5 | 7 |
| 3 | 3 | 8 |
| 4 | 9 | 13 |
| 5 | 7 | 11 |

Um outro dado do problema é a forma para seleccionar a primeira semente, usar a regra SPT, e o mecanismo de geração de vizinhança, a troca entre pares adjacentes. Considerar também que a primeira sequência que melhorar o desempenho será a nova semente.

Resolução:

Primeira semente, obtida pela regra SPT:

| **Sequência** | 1 | 3 | 2 | 5 | 4 |
|---|---|---|---|---|---|
| **Data de entrega** | 2 | 8 | 7 | 11 | 13 |
| **Tempo de percurso** | 1 | 4 | 9 | 16 | 25 |
| **Entidades atrasadas** | | | 1 | 1 | 1 |

Na = 3

Geração da primeira vizinhança:

| **3** | **1** | 2 | 5 | 4 | Na=4 |
|---|---|---|---|---|---|
| 1 | **2** | **3** | 5 | 4 | Na=3 |
| 1 | 3 | **5** | **2** | 4 | Na=2 * |
| 1 | 3 | 2 | **4** | **5** | Na=3 |

Nova semente: 1, 3, 5, 2, 4

Geração de vizinhança:

| **3** | **1** | 5 | 2 | 4 | Na=3 |
|---|---|---|---|---|---|
| 1 | **5** | **3** | 2 | 4 | Na=3 |
| 1 | 3 | **2** | **5** | 4 | Na=3 |
| 1 | 3 | 5 | **4** | **2** | Na=2 |

Como nenhuma sequência melhora o Na=2 da última semente, a pesquisa termina com Na=2.

A sequência gerada pelo procedimento NST é um óptimo local (em relação à estrutura da vizinhança). Não há, em geral, forma de saber se o óptimo local é também o óptimo global (a melhor solução de todas). Pode-se melhorar o procedimento NST, com vista a encontrar o óptimo global, sem no entanto garantir que foi encontrado, através de:

1. Geração de várias sementes para começar a busca, aplicando o procedimento de busca a cada semente e retirando a solução óptima gerada.
2. Para cada vizinhança, guardar as soluções melhores que a semente. Usar cada uma destas soluções como sementes de novas vizinhanças. A sequência óptima assim gerada terá maior probabilidade de ser o óptimo global que a gerada pelo procedimento NST.
3. Escolha de mecanismos de geração de vizinhanças que gere maiores vizinhanças.
4. Etc..

Qualquer destes mecanismos, incluindo o procedimento NSP, é razoavelmente eficaz na obtenção de boas soluções.

### *3.2.4 Algoritmos genéticos*

Os algoritmos genéticos tem como base a lógica da evolução genética. Os indivíduos com melhor desempenho são os que sobrevivem e que se reproduzem para gerar nova geração e assim sucessivamente. Assim, cada nova geração tem apenas as combinações genéticas dos melhores indivíduos da geração anterior. Esta lei natural assegura que cada espécie tenda naturalmente a melhorar o seu desempenho médio ao longo das gerações.

Os algoritmos genéticos podem ser considerados como pertencendo à classe das técnicas de procura local com algumas particularidades: A principal característica é que em vez de se iniciar o processo com uma solução semente, o processo é iniciado com um conjunto de

soluções semente. Os algoritmos genéticos quando aplicados à programação da produção vêem as soluções como *indivíduos* ou membros de uma *população*. Cada indivíduo é avaliado quanto ao seu desempenho que dita a sua sobrevivência na população. Trata-se de um processo iterativo em que cada iteração é uma geração. É necessário definir qual a população inicial, qual o método de cruzamento e/ou mutação, qual a medida de desempenho e qual o critério de paragem.

Os algoritmos genéticos tem genericamente os seguintes passos:

a) Definição da população inicial. Esta população é um conjunto de X soluções (programas) que são geradas usando normalmente uma heurística.

b) Definição do mecanismo de cruzamento e mutação dos indivíduos (soluções). Estes mecanismos nem sempre são simples de criar pois poderá haver muitas restrições.

c) Depois disto há que fazer cruzamentos e/ou mutações entre as soluções iniciais (progenitores) por forma a gerar os filhos (soluções da 2ª geração). O tamanho da 2ª geração poderá ser igual ao número de progenitores (X).

d) Do conjunto das duas gerações (2X indivíduos) seleccionam-se os X melhores como sendo a população sobrevivente. Os melhores são os que apresentação melhor desempenho de acordo com o objectivo.

e) Novos cruzamentos e/ou mutações são levadas a cabo sobre esta nova população e nova selecção será feita dos melhores indivíduos.

f) Este processo repete-se até que deixe de haver melhoria significativa no desempenho das populações.

## 3.3 Programação em máquina única

A programação da produção em máquina única ou processador único é um problema de programação especial no qual a ordenação (pôr em sequência) das entidades, determina completamente o programa de produção. Podemos dizer que se trata de um problema sequencial puro e apesar de ser o mais simples dos problemas de programação da produção, é contudo muito importante. Por um lado ilustra a variedade de tópicos nos problemas de programação da produção e por outro, fornece um contexto no qual se investiga várias medidas de desempenho e várias técnicas. Além disso, o desempenho de um sistema produtivo passa muitas vezes pelo uso que é dado ao estrangulamento[1] desse mesmo sistema. Assim, a ordenação das entidades no estrangulamento pode ser levada a cabo de uma forma isolada caindo claramente num problema de máquina única. Todo o resto do sistema é depois programado em função das decisões tomadas no estrangulamento. Além disso, para se poder entender o comportamento de um sistema complexo, é vital que se entenda o comportamento dos seus componentes. Assim, o problema de programação da produção em máquina única aparece muitas vezes como um componente elementar de um problema de programação complexo.

O problema básico em máquina única é caracterizado pelas seguintes condições:

1. Um grupo de *n* entidades (lotes, peças, produtos, etc.) independentes, com apenas uma operação está disponível ao instante zero.
2. Os tempos de preparação da máquina são independentes da sequência e podem ser incluídos nos tempos de processamento.
3. A máquina está continuamente disponível.

[1] Estrangulamento: Posto de trabalho crítico do sistema, o que limita a capacidade de todo o sistema. Este assunto será abordado noutros capítulos.

4. Uma vez começado o processamento, nunca será interrompido até que todas as entidades sejam processadas.

Nestas condições temos o valor de **n!** como número total de combinações possíveis, que é o número de diferentes permutações de **n** elementos. A título de exemplo, para 5 entidades temos 120 sequências possíveis e para 10 entidades temos 3 milhões, 628 mil e 800 sequências possíveis. Se a avaliação de uma medida de desempenho ou eficiência para cada sequência demorasse 0,1 segundo, seriam necessários, para o caso dos 10 entidades, 12.6 dias de 8 horas para se poder determinar, por enumeração completa das sequências possíveis, a sequência que permitiria optimizar a medida de desempenho em questão.

### *3.3.1 Minimização de algumas medidas de desempenho*

Para certas medidas de desempenho, existem algoritmos de optimização, de aplicação expedita pelo que não se torna necessário fazer enumeração completa e avaliação de todas as sequências possíveis. Assim, em máquina única, quando se pretende minimizar o atraso médio (Amed) relativamente às datas de entrega das entidades, podemos aplicar o seguinte teorema:

**Minimização do atraso médio.** O atraso médio, Amed, das entidades a processar em máquina única é minimizado ordenando as entidades por ordem crescente dos seus tempos de processamento, ou seja, pelo uso da regra SPT (Shortest Processing Time).

**Minimização do tempo de percurso médio.** O tempo de percurso médio Fmed, das entidades a processar em máquina única também é minimizado ordenando as entidades por ordem crescente dos tempos de processamento, ou seja, pelo uso da mesma regra SPT.

**Minimização do atraso máximo.** O atraso máximo, Amax, é minimizado através da ordenação das entidades pela ordem crescente das datas de entrega. Uso da regra EDD (Earliest due date)

**Minimização do número de entidades com atraso.** O algoritmo de Hodgson minimiza o número de entidades em atraso, Na. Tem os seguintes passos:

1. Colocar todas as entidades no conjunto *E* ordenando-as pela ordem crescente das suas datas de entrega (regra EDD). Criar um conjunto *L* vazio.
2. Se nenhuma entidade em *E* tem atraso, termina aqui: *E* é óptimo. Caso contrário, identificar a primeira entidade de *E* que tem atraso. Vamos supor que é a entidade *k*.
3. Identificar a entidade com tempo de processamento maior nas primeiras *k* entidades do conjunto *E*. Tirar essa entidade do conjunto *E* colocando-a no conjunto *L*. Recalcular os tempos de conclusão das entidades que restam no conjunto *E*. Voltar ao passo 2.

Com o objectivo de clarificar o algoritmos vejamos um exemplo:

| Entidade i | $t_i$ | $d_i$ |
|---|---|---|
| 1 | 1 | 2 |
| 2 | 5 | 7 |
| 3 | 3 | 8 |
| 4 | 9 | 13 |
| 5 | 7 | 11 |

Primeiro estágio

| | | |
|---|---|---|
| Passo 1. Inicialização. | E={1-2-3-5-4} | L={} |
| Passo 2. Entidade 3 é a primeira entidade com atraso. | | |
| Passo3. Entidade 2 é retirada de E. | E={1-3-5-4} | L={2} |

Segundo estágio

Passo 2. Entidade 4 é a primeira entidade com atraso.
Passo 3. Entidade 4 é retirada de E. E={1-3-5} L={2,4}

Terceiro estágio

Passo 2. Não há nenhuma entidade com atraso no conjunto E. A sequência óptima é: 1-3-5-2-4

As entidades com atraso, conjunto L, podem ser colocadas na sequência em qualquer ordem pois isso não altera o número de entidades com atraso.

As técnicas de procura local referidas anteriormente neste mesmo capítulo também podem ser usadas para resolver problemas de programação da produção em máquina única.

## 3.4 Máquina única com tempos de preparação variáveis

Em muitos problemas reais, os tempos de preparação da máquina variam com o tipo de entidade que acabou de ser processada e do tipo de entidade a ser processada a seguir. A título de exemplo, suponhamos que uma linha produz 4 tipos de gasolina: de competição, super, normal e sem chumbo. A matriz dos tempos de preparação, $s_{ij}$, pode ser representada pela seguinte tabela:

| | **De corrida (1)** | **Aditivada (2)** | **S/ chumbo 95 (3)** | **S/ Chumbo 98 (4)** |
|---|---|---|---|---|
| **De corrida (1)** | -- | 30 | 50 | 90 |
| **Aditivada (2)** | 40 | -- | 20 | 80 |
| **S/ chumbo 95 (3)** | 30 | 30 | -- | 60 |
| **S/ chumbo 98 (4)** | 20 | 15 | 10 | -- |

Num ciclo completo de produção, a quantidade de tempo improdutivo (tempo de preparação) depende da sequência pela qual as gasolinas são produzidas, como se pode ver na tabela acima. Em particular, o tempo total de preparação em cada uma das 6 possíveis sequências que inclui todos os 4 tipos é diferente.

s(1-2-3-4-1) 30+20+60+20 = 130
s(1-2-4-3-1) 30+80+10+30 = 150
s(1-3-2-4-1) 50+30+80+20 = 180
s(1-3-4-2-1) 50+60+15+40 = 165
s(1-4-2-3-1) 90+15+20+30 = 155
s(1-4-3-2-1) 90+10+30+40 = 170

Assume-se implicitamente que a produção é contínua e que o ciclo é sempre mantido. A quantidade de tempo necessária para completar todas as entidades é chamada “Makespan” e será denotada de M. Assim, vem que:

$$F_1 = s_{0,1} + t_1$$
$$F_2 = s_{1,2} + t_2 + F_1$$
$$F_{n-1} = s_{n-2,n-1} + t_{n-1} + F_{n-2}$$
$$F_n = s_{n-1,n} + t_n + F_{n-1}$$

Onde o estado 0 corresponde ao estado inicial.

Se o estado *n+1* representa o estado terminal (neste caso idêntico ao estado inicial) então o *Makespan* é de:

$$M = F_n + s_{n,n+1} = \sum_{i=1}^{n+1} s_{i-1,i} + \sum_{i}^{n} t_i$$

Sabendo que o segundo somatório é constante, o problema de minimização do *Makespan* é equivalente ao da minimização do primeiro somatório. Esta soma representa o tempo improdutivo total em cada ciclo de produção. O tipo de estrutura representada por este problema de *Makespan* é equivalente ao "problema do caixeiro viajante" e como tal pode ser resolvido com as técnicas aplicadas a esse problema clássico.

## 3.5 Programação da produção em processadores paralelos

O processo de programação da produção requer por um lado decisões quanto à sequenciação e por outro lado, decisões quanto à atribuição de recursos. Quando há apenas um recurso, o problema de programação da produção limita-se à sequenciação enquanto que se houver mais do que um recurso, é necessário tomar decisão também no respeitante à atribuição de recursos.

### *3.5.1 Processadores paralelos idênticos e entidades independentes*

Em programação da produção é muitas vezes possível tirar partido do paralelismo na estrutura dos recursos. Um contexto simples para investigar os efeitos dos recursos paralelos é o problema da sequenciação de estágio simples com várias máquinas. No modelo básico, supõem-se que há *n* entidades de apenas uma operação simultaneamente disponíveis no tempo zero. Supõe-se também que há *m* máquinas disponíveis para processamento e que uma entidade só pode ser processada por uma máquina de cada vez.

**Minimizar "Makespan" (Tempo de percurso total)**

Uma resolução ao problema de minimização do *makespan* em processadores paralelos foi proposto por McNaughton, 1959 onde as entidades são independentes e a interrupção da entidade é permitida. O processamento de uma entidade pode ser interrompido para ser concluído noutra máquina. A propriedade central está no facto de que o *makespan* mínimo, M*, é dado por:

$$M^* = \max\left\{\frac{1}{m}\sum_{i=1}^{n} t_i, \max\left[t_i\right]\right\}$$

Não deve ser difícil compreender a razão pela qual esta equação é válida. Esta equação diz que, ou as máquinas são utilizadas completamente através de uma programação óptima, ou a duração da entidade com maior tempo de processamento determinará o *makespan*.

O método de construção do programa óptimo é o seguinte.

**Algoritmo de McNaughton:**

1. Seleccionar uma entidade para começar na máquina 1 ao tempo zero.
2. Escolher qualquer entidade ainda não seleccionada e coloca-la o mais cedo possível na mesma máquina. Repetir este passo até que a máquina ficar ocupada até ao tempo M* (ou até todas as entidades ficarem atribuídas).
3. Pegar na parte da entidade que ficou por completar na máquina anterior e atribuí-la à próxima máquina. Volta ao passo 2.

Se a interrupção das entidades é proibida, o problema de minimização do *makespan* é algo mais complicado. Não há conhecimento de algum algoritmo que encontre a solução óptima embora haja um procedimento heurístico para a construção de um programa envolvendo o uso da regra LPT (Longest Processing Time) como um mecanismo de prioridade.

**Procedimento heurístico para minimização do M**

1. Ordenar as entidades pelo seu tempo de processamento mais longo (LPT).
2. Programar essas entidades por ordem, atribuir a entidade à máquina que fica livre mais cedo.

Este heurístico não garante um *makespan* óptimo.

**Procedimento para minimização do Fmed**

1. Ordenar as entidades pelo seu tempo de processamento mais curto (SPT).
3. Atribuir a próxima entidade à máquina que fique livre mais cedo. Repetir até que todas as entidades estejam atribuídas.

## 3.6 Programação da produção em linhas de fabrico

No processamento de entidades em linha de fabrico (linha de produção, linha de montagem) cada entidade é sujeita a mais do que uma operação que terá de acontecer numa sequência pré-definida, ou seja, cada entidade requer uma sequência específica de operações a ser processadas por forma a que a entidade seja concluída. Este tipo de estrutura é, às vezes, chamada estrutura de precedência linear.

Uma linha contém *m* máquinas e cada entidade consiste de *m* operações, cada uma das quais requerendo máquinas diferentes. As máquinas numa linha são numeradas: *1, 2, 3……,m*: e as operações na entidade *i* são numeradas de forma correspondente: *(i,1), (i,2), (i,3), …., (i.m)*. Na linha de fabrico pura, cada entidade requer uma operação em cada máquina. Numa linha de fabrico vista de uma forma mais geral, entidades podem requerer menos operações do que o número de máquinas. Neste último caso algumas entidades podem prescindir de alguma ou algumas máquinas. Apesar disso, é suficiente estudar apenas a linha pura para ilustrar os principais aspectos do modelo.

Condições que caracterizam um problema de programação da produção em linhas de fabrico:

1. Todas as *n* entidades estão disponíveis ao tempo zero. (Cada entidade requer *m* operações e cada operação requer uma máquina diferente).
2. Os tempos de preparação são independentes da sequência e estão incluídos nos tempos de processamento.
3. As descrições das entidades são previamente conhecidas.
4. Todas as máquinas estão continuamente disponíveis.
5. Operações individuais não podem ser interrompidas uma vez iniciadas.

No problema da programação da produção em linhas de fabrico, há *n!* sequências diferentes possíveis para cada máquina e consequentemente $(n!)^m$ programas de produção (soluções) diferente para ser examinados.

### *3.6.1 Programas ordenados de produção*

Do universo de programas possíveis numa linha de fabrico há um subgrupo importante de programas que são os programas ordenados. Os programas ordenadas são apenas aqueles em a sequência pela qual as entidades entram na primeira máquina é mantida em todas as máquinas da linha (ver figura 3.2a). Desta forma, a linha pode ser vista como apenas uma máquina com muitas operações, o que resulta num número de programas igual a *n!*. Na procura da solução óptima, não basta considerar apenas os programas ordenados pois a solução óptima pode muito bem ser um programa não ordenado. No entanto, também nem sempre é necessário considerar os $(n!)^m$ programas para se encontrar a solução óptima. As duas propriedades seguintes indicam o quanto é possível reduzir o número de programas neste tipo de problemas.

**Propriedade 1.** No que diz respeito a qualquer medida de desempenho, é suficiente considerar, para as máquinas 1 e 2, apenas os programas ordenados.

É possível compreender esta propriedade considerando um programa em que as sequências nas máquinas 1 e 2 são diferentes. Vamos considerar que algures no programa temos duas entidades *i* e *j* em que a operação *(i,1)* precede a operação adjacente *(j,1)* mas a operação

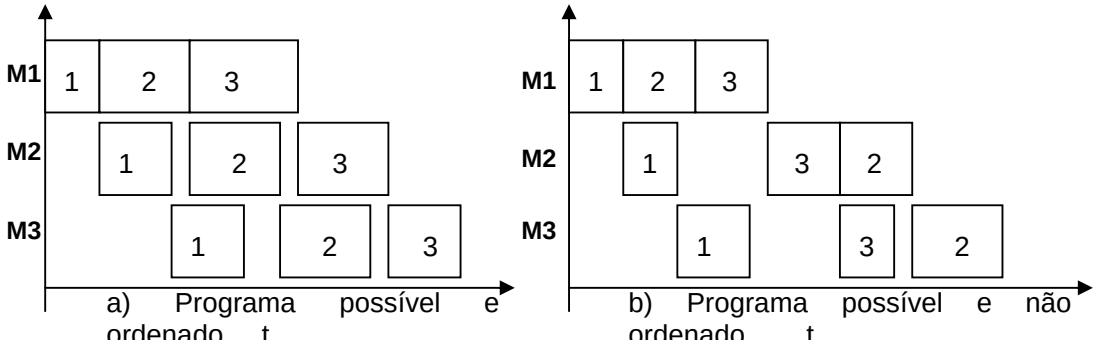



**Figura 3.2. Programas ordenados e possíveis.**

*(j,2)* precede *(i,2)*. Esta situação está representada na figura 3.3.

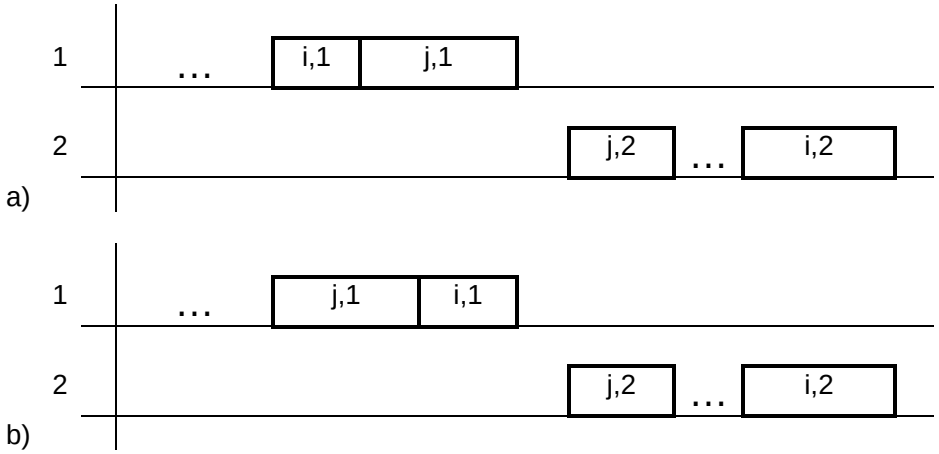



**Figura 3.3 Troca de pares entre duas operações na máquina 1.**

A ideia é mostrar que a ordem das entidades na máquina 2 (j antes de i) pode ser também aplicada na máquina 1 sem afectar negativamente a medida de desempenho. Se as operações (i,1) e (j,1) forem trocadas resulta no programa apresentado na figura 3.3b), então:

1. Com a excepção da operação (i,1), nenhuma operação será atrasada.
2. A operação (i,2) não será atrasada pela troca.

3. O processamento de (j,2) bem como outras operações pode ser levado a cabo mais cedo como resultado da troca.

Assim a conclusão das entidades não seria atrasada pela troca das operações. Isto quer dizer que nenhuma entidade seria concluída mais tarde o que mostra que não iria piorar nenhuma medida regular de desempenho.

**Propriedade 2.** No que diz respeito ao *makespan* como medida de desempenho, é suficiente considerar apenas os programas ordenados nas máquinas *m-1* e *m*.

A demonstração desta propriedade é baseada num argumento parecido do que foi usado na propriedade 1. Considerar um programa em que as sequências nas máquinas m e m-1 são diferentes.

Da figura 3.4 podem-se tirar as seguintes conclusões:

1. Com a excepção de (j,m) nenhuma operação é atrasada pela troca.
2. A operação (j,m) não é concluída mais tarde do que (i,m) do programa inicial.
3. Pode ser possível que como resultado da troca que as operações (i,m) e (j,m) sejam processadas mais cedo.

Assim a troca não leva ao aumento do *makespan*. Este tipo de argumento aplica-se a qualquer outro programa nas quais as sequências diferem entre as máquinas m-1 e m.

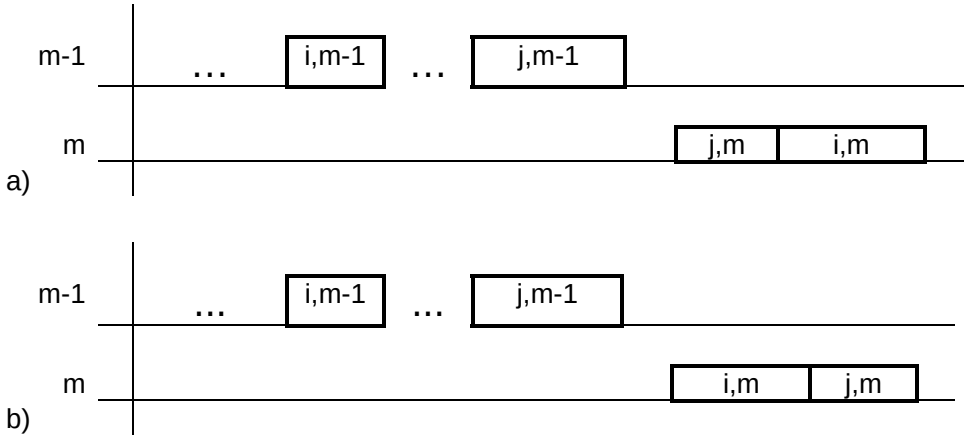



**Figura 3.4 Troca de pares entre duas operações na máquina m.**

As implicações destas duas propriedades são que quando se procura pela solução óptima não é necessário considerar todos os programas possíveis. O número de programas necessários são:

1. Para qualquer medida regular de desempenho temos $(n!)^{m-1}$ programas.
2. Para o *makespan* temos $(n!)^{m-2}$ programas desde que m>2.

### *3.6.2 Regra de Johnson*

É conhecido pelo problema de Johnson, o problema de linha de fabrico com duas máquinas sendo o *makespan* o a medida de desempenho a minimizar o. Os resultados originalmente obtidos por Johnson, 1954, são agora fundamentos normalizados na teoria de programação da produção. A regra de Johnson diz que a entidade i precede a entidade j numa sequência óptima se:

$$\min\{t_{i1},t_{j2}\} \leq \min\{t_{i2},t_{j1}\}$$

A implementação desta regra segue os seguintes passos:

*Passo 1*. Encontrar $\min\{t_{i1}, t_{i2}\}$

*Passo 2a*. Se o menor tempo de processamento requer a 1ª máquina, colocar a entidade respectiva na primeira posição disponível. Ir para o passo 3.

*Passo 2b*. Se o menor tempo de processamento requer a 2ª máquina, colocar a entidade respectiva na última posição disponível. Ir para o passo 3.

*Passo 3*. Retirar a entidade atribuída e voltar ao passo 1 até que todos as entidades sejam atribuídas.

### *3.6.3 Algoritmo de Ignall-Schrage*

Este algoritmo aplica-se a problemas de programação da produção em linha de fabrico e encontra o programa com menor *makespan* de entre todos os programas ordenados.

Para ilustrar o procedimento para $m$=3 considere-se o seguinte:

$m$ - número de máquinas.
$t_{ij}$ - tempo de processamento da entidade $j$ na máquina $i$.
$\sigma$ - Conjunto de entidades consideradas na sequência parcial.
$\sigma'$ - Conjunto de entidades ainda não consideradas na sequência parcial.

Para uma dada sequência parcial $\sigma$ considere que:

$q_1$ - o tempo de conclusão mais tardio na máquina 1 entre as entidades em $\sigma$ (este o tempo mais cedo a partir do qual alguma entidade de $\sigma'$ pode ser processada).
$q_2$ - o tempo de conclusão mais tardio na máquina 2 entre as entidades em $\sigma$.
$q_3$ - o tempo de conclusão mais tardio na máquina 3 entre as entidades em $\sigma$.

A quantidade de tempo de processamento ainda requerido na máquina 1 é de: $\sum_{j \in \sigma'} t_{j1}$

Para além disso, suponham que uma dada entidade $k$ é a ultima da sequência. Depois da entidade $k$ ter sido concluída na máquina 1, um intervalo de pelo menos ($t_{k2} + t_{k3}$) é necessário antes toda o programa possa ser concluído, como mostra a figura 3.5a. Na situação mais favorável, a ultima entidade

(1) não tem de esperar entre a conclusão de uma operação e o começo da próxima, e
(2) tem a soma mínima ($t_{j2} + t_{j3}$) entre as entidades de $\sigma'$.

O cálculo de $q_1$, $q_2$ e $q_3$ quando o conjunto $\sigma$ é apenas composto de uma entidade, é feito da seguinte forma:

$$q_1 = t_{11}$$
$$q_2 = t_{11} + t_{12}$$
$$q_3 = t_{11} + t_{12} + t_{13}$$

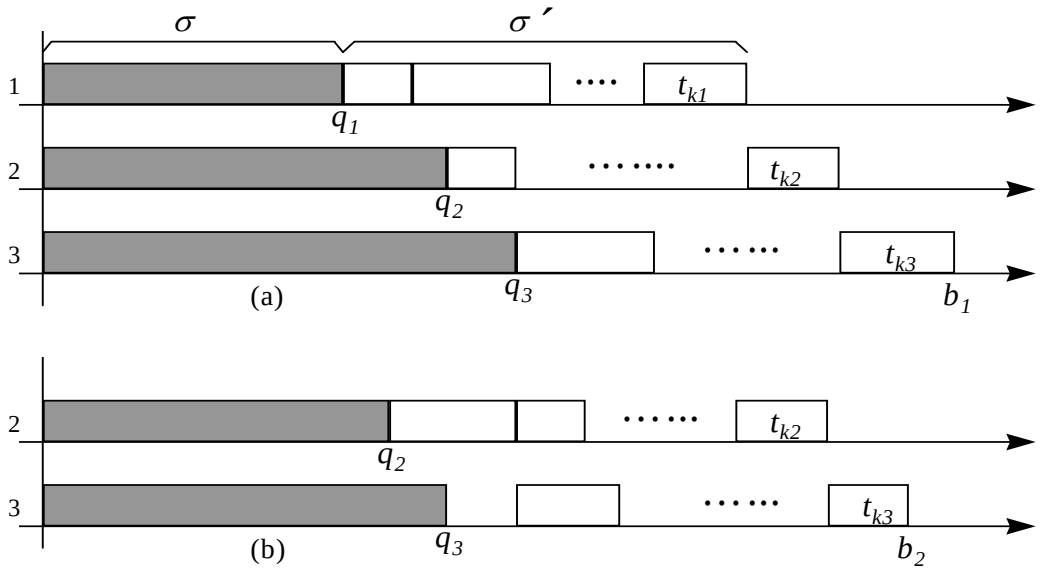



Figura 3.5 Estrutura de um programa de produção para linha de fabrico com o intuito de calculo dos limites inferiores de *makespan* para uma dada sequência parcial. As áreas a cinzento representam os tempos de processamento das entidades já consideradas para a sequência parcial.

O cálculo de $q_1$, $q_2$ e $q_3$ fica um pouco complexo quando o conjunto $\sigma$ tem mais do que uma entidade. Neste caso o cálculo é feito da seguinte forma:

$q_1 = q_{1(anterior)} + t_{i1}$
$q_2 = max\{q_1 , q_{2(anterior)}\} + t_{i2}$
$q_3 = max\{q_2 , q_{3(anterior)}\} + t_{i3}$

O índice $_{(anterior)}$ refere-se à entidade seleccionada na iteração anterior.

Assim, um dos limites inferiores (*lower bound*) do *makespan* é: $b_1 = q_1 + \sum_{j\in\sigma'} t_{j1} + \min_{j\in\sigma'}\left\{t_{j2} + t_{j3}\right\}$

Um raciocínio similar é também aplicado ao processamento requerido na máquina 2 (figura 3.5b) resultando no segundo limite inferior do *makespan*:

$$b_2 = q_2 + \sum_{j\in\sigma'} t_{j2} + \min_{j\in\sigma'}\left\{t_{j3}\right\}$$

Finalmente, um limite inferior baseado no processamento ainda requerido pela máquina 3 é:

$$b_3 = q_3 + \sum_{j\in\sigma'} t_{j3}$$

O limite inferior da cada alternativa é dado por: $B=\max\{b_1, b_2, b_3\}$

Para clarificar o método, vejamos o seguinte exemplo:

| **Entidade i** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|
| $t_{i1}$ | 3 | 11 | 7 | 10 |
| $t_{i2}$ | 4 | 1 | 9 | 12 |
| $t_{i3}$ | 10 | 5 | 13 | 2 |

Para as sequências que incluam a entidade **1** temos:

$\sigma = \{1\}$ e $\sigma' = \{2, 3, 4\}$

Assim, $q_1 = t_{11} = 3$, $q_2 = t_{11} + t_{12} = 7$ e $q_3 = t_{11} + t_{12} + t_{13} = 17$

Os limites inferiores são:

$b_1 = 3 + 28 + 6 = 37$
$b_2 = 7 + 22 + 2 = 31$
$b_3 = 17 + 20 = 37$

Os cálculos requeridos para encontrar o óptimo são:

| Sequência parcial | $(q_1, q_2, q_3)$ | $(b_1, b_2, b_3)$ | B |
|---|---|---|---|
| 1 | (3, 7, 17) | (37, 31, 37) | 37 |
| 2 | (11, 12, 17) | (45, 39, 42) | 45 |
| 3 | (7, 16, 29) | (37, 35, 46) | 46 |
| 4 | (10, 22, 24) | (37, 41, 52) | 52 |
| 12 | (14, 15, 22) | (45, 38, 37) | 45 |
| 13 | (10, 19, 32) | (37, 34, 39) | 39 |
| 14 | (13, 25, 27) | (37, 40, 45) | 45 |
| 132 | (21, 22, 37) | (45, 36, 39) | 45 |
| 134 | (20, 32, 34 | (37, 38, 39) | 39 |

Para as sequências que incluam as entidades **1** e **2** temos:

$\sigma = \{1, 2\}$ e $\sigma' = \{3, 4\}$
$q_1 = q_{1(anterior)} + t_{i1} = 3 + 11 = 14$
$q_2 = max\{q_1 , q_{2(anterior)}\} + t_{i2} = \max\{14, 7\} + 1 = 15$
$q_3 = max\{q_2 , q_{3(anterior)}\} + t_{i3} = \max\{15, 17\} + 5 = 22$

A árvore de "Branch and Bound" correspondente à tabela anterior está representada na figura 3.6.

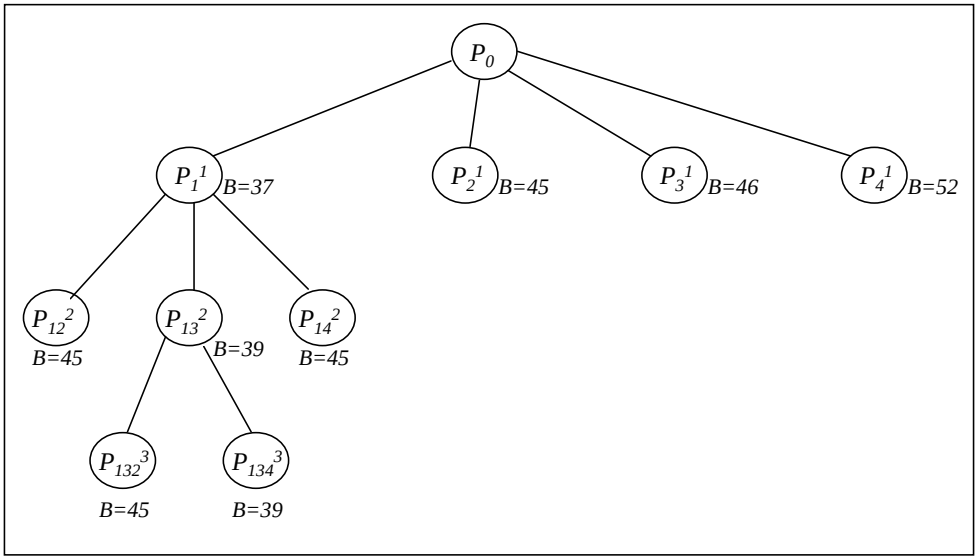



**Figura 3.6 A árvore de "Branch and Bound" para o exemplo anterior.**

## 3.7 Programação da produção em oficinas de fabrico

O problema clássico de programação de oficinas difere do problema das linhas num aspecto importante: o fluxo dos materiais não é unidireccional. Cada entidade apresenta uma sequência própria de máquinas a visitar. Apesar que cada entidade possa ter um número qualquer de operações, a formulação mais comum assume que todas as entidades tem *m* operações, uma

em cada máquina. Não é conceptualmente mais difícil lidar com o caso geral em que uma entidade pode requerer mais do que uma vez o processamento na mesma máquina. A figura 3.7 mostra todas as possibilidades que uma máquina numa oficina de fabrico pode ter em termos de fluxo. Cada máquina pode ser a primeira, a última ou ter qualquer outra ordem na sequência de operações requeridas para o processamento de uma entidade.

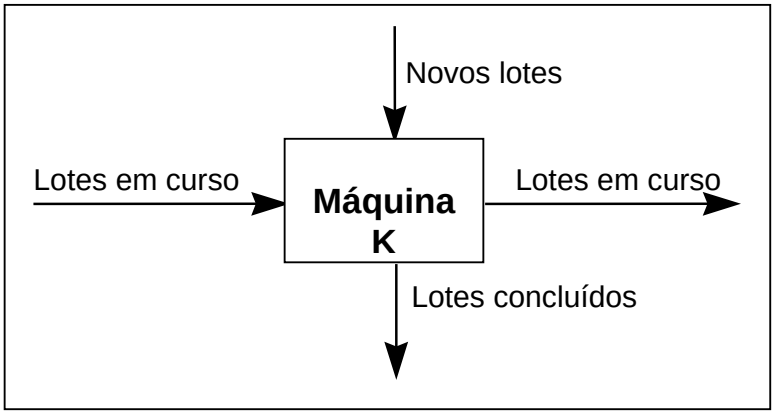



**Figura 3.7 Fluxo numa máquina típica numa oficina.**

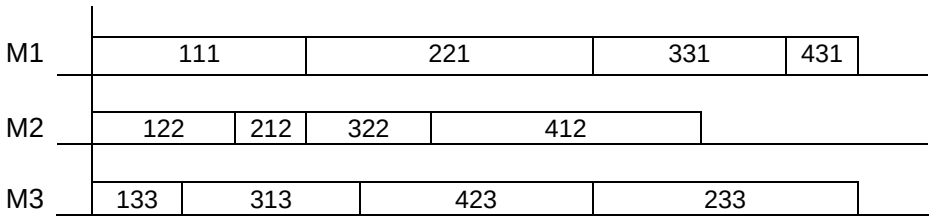
Em linhas de fabrico, a operação *k* de qualquer entidade é levada a cabo na máquina *k,* e não há de facto necessidade de distinguir entre número da máquina e número da operação. No caso das oficinas é mais apropriado descrever uma operação com (*i, j, k*) por forma a definir claramente que a operação *j* da entidade *i* requer a máquina *k*.

A descrição gráfica das entidades a serem processadas em cada máquina, no problema da oficinas, está representado na figura 3.8a por um gráfico de Gantt. A numeração sequencial das operações para uma dada entidade é uma forma de indicar a sequência de operações. Se as operações forem colocadas da forma mais compacta possível no gráfico de Gantt de uma forma arbitrária como mostra a figura 3.8a, o gráfico representa uma forma de carregar as máquinas que mais provavelmente não será exequível. Um programa exequível para o mesmo problema é apresentado na figura 3.8b, que obedece aos constrangimentos dos recursos quando duas operações não podem ser levadas a cabo simultaneamente na mesma máquina.

O grupo de máquinas que uma dada entidade tem de visitar constitui a trajectória (*routing*) dessa entidade e faz parte do plano de processo da mesma. Por exemplo, a entidade 2 tem uma trajectória de 2-1-3.

O problema fica completamente formulado quando também definimos a medida de desempenho a optimizar.

Sempre que numa solução (programa de produção) exista a possibilidade de começar mais cedo uma operação numa máquina sem alterar a sequência das operações em nenhuma das máquinas então estamos em presença de tempo supérfluo de inactividade. O conjunto de programas em que não existe nenhum tempo supérfluo de inactividade chamam-se programas *semi-activos.* Só é necessário considerar os programas semi-activos por forma a encontrar o programa que optimiza uma dada medida de desempenho.



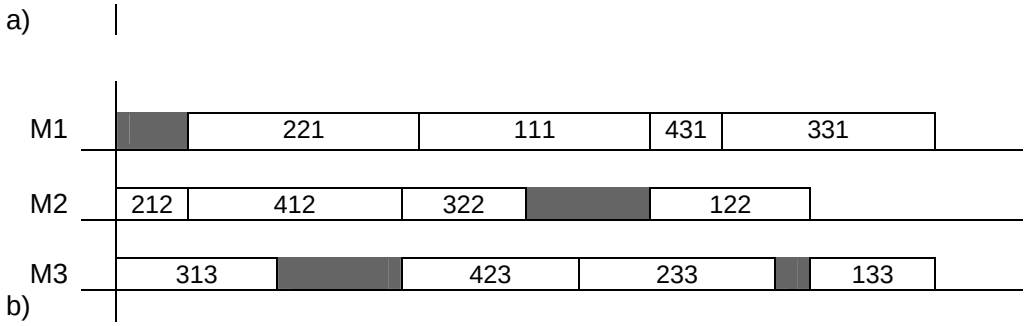



**Figura 3.8 Representação de programas em diagrama de Gantt. a) inexequível, b) exequível.**

O numero de programas semi-activos é finito embora possa ser extremamente grande. O numero exacto é normalmente difícil de determinar. Se considerarmos que todas as entidades visitam todas as máquinas da oficina então temos $(n!)^m$ programas semi-activos.

### *3.7.1 Oficina com 2 máquinas - Método de Jackson*

Este método tem como objectivo a minimização do *makespan* em problemas de programação da produção em oficinas de fabrico com apenas 2 máquinas e qualquer número de entidades.

Para explicar o método vamos usar um exemplo: Considere que se pretende executar/processar 7 entidades de L1 a L7, numa oficina com duas máquinas, A e B, cuja sequência e tempos de processamento são indicados na tabela seguinte. Os valores entre parêntesis dizem respeito aos tempos de processamento.

| Entidades | L1 | L2 | L3 | L4 | L5 | L6 | L7 |
|---|---|---|---|---|---|---|---|
| 1ª operação | B(5) | A(7) | B(2) | A(4) | B(6) | A(3) | B(5) |
| 2ª operação | -- | -- | -- | B(5) | A(2) | B(7) | A(3) |

Aplicação do método:

{A} = {L2} => Conjunto de entidades processadas só na máquina A.
{B} = {L1, L3} => Conjunto de entidades processadas só na máquina B.
{A,B} = {L4, L6} => Conjunto de entidades processadas primeiro em A e depois em B.
{B,A} = {L5, L7} => Conjunto de entidades processadas primeiro em B e depois em A.

Aplicar o método de Johnson aos conjuntos {A,B} e {B,A} assumindo tratar-se de dois problemas independentes de linhas de fabrico. Assim, resulta para cada um dos conjuntos as seguintes sequências:

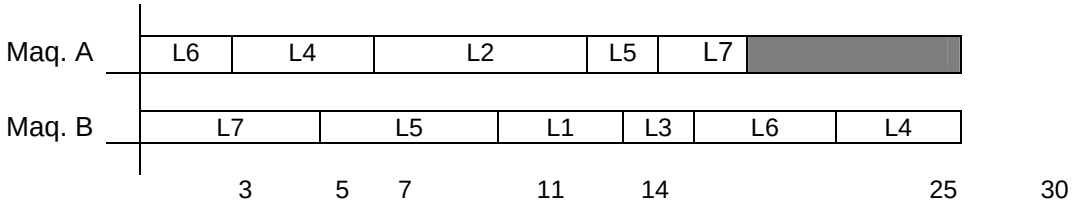
{A,B} = {L4,L6} => (L6) → (L4)
{B,A} = {L5,L7} => (L7) → (L5)

De acordo com o método de Jackson, o programa de produção resultante para o problema será o seguinte:

Na máquina A: (L6) → (L4) → {A} → {B,A}
Na máquina B: (L7) → (L5) → {B} → {A,B}

*Nota: A ordem para as entidades dos conjuntos {A} e {B} é arbitrária, não influenciando o valor .*

A visualização do programa pode fazer-se através de um diagrama de Gantt, de onde podemos também obter o valor de *makespan,* que para o exemplo é:




### *3.7.2 Oficina com m máquinas*

Este é um problema NP difícil e não pode ser formulado usando programação linear, no entanto, podem ser formulados com programação inteira e programação disjuntiva (Pinedo e Chau, 1999). Os mesmos autores apresentam um método de "Branch and Bound" para resolver este problema em termos de *makespan.* Como sabem este tipo de métodos garante encontrar a solução óptima mas, para problemas com grandes dimensões, o enorme esforço computacional requerido torna proibitiva a enumeração completa de todos programas semi-activos, mesmo que essa enumeração seja truncada por um esquema de *Branch and Bound*. Ainda mesmos autores apresentam a heurística "Shifting Bottleneck" para resolver este tipo de problemas. É possível fazer o *download* grátis de uma aplicação informática para resolver problemas de programação da produção onde esta heurística pode ser experimentada (http://www.ieor.columbia.edu/~andrew/scheduling/Lekin.html).

O mecanismo mais comum para resolver estes problemas na industria em geral, passa pelo uso de regras de prioridade. A tomada de decisão acontece em tempo real em cada uma das máquina da oficina. Várias são as regras que são usadas e delas já foi feita referência neste mesmo capítulo. Nas empresas menos organizadas a tomada de decisão é feita numa base pouco sistemática mas pressupõem sempre algum tipo de regra de prioridade, como por exemplo, as entidades do cliente mais importante são as primeiras a ser processadas, ou, se um cliente fizer muita pressão, o seu lote poderá ser o próximo a ser processado, etc..

**Regras de prioridade**

As regras de prioridade são normalmente usadas em ambientes dinâmicos, ou seja, onde novas entidades estão continuamente a entrar no sistema, e são usadas em tempo real, ou seja, as decisões vão sendo tomadas à medida que haja alterações no sistema. Quando uma entidade é conluiada numa máquina é necessário decidir qual a próxima entidade a ser processada, ou, se uma máquina avaria, é necessário decidir o que fazer às entidades da fila de espera respectiva.

A única coisa que podemos fazer aqui, é prever o comportamento do sistema se uma determinada regra for usada num sistema produtivo. Para ilustrar o que pode acontecer no uso de regras de prioridade suponham que a regra MWKR (most work remaining) é aplicada ao seguinte exemplo:

**Tempos de processamento**

| | | Operação | | |
|---|---|---|---|---|
| | | **1** | **2** | **3** |
| **Entidades** | **E1** | 4 | 3 | 3 |
| | **E2** | 1 | 4 | 4 |
| | **E3** | 3 | 2 | 3 |
| | **E4** | 3 | 3 | 1 |

**Trajectórias**

| | | Operação | | |
|---|---|---|---|---|
| | | **1** | **2** | **3** |
| **Entidades** | **E1** | M1 | M2 | M3 |
| | **E2** | M2 | M1 | M3 |
| | **E3** | M3 | M2 | M1 |
| | **E4** | M2 | M3 | M1 |

Para se prever o que vai acontecer com o uso da regra MWKR é necessário usar algum método que vá registando a evolução do tempo e as alterações que vão acontecendo nas filas de espera e nas máquinas. Podemos utilizar um gráfico de Gantt onde se vai construindo o programa ou então uma tabela.

Vamos começar com o instante zero. Neste instante todas as entidades estão disponíveis para processamento. Neste instante as máquinas M1 e M3 são apenas requeridas por uma entidade cada uma, logo, não há nenhum problema de decisão, essas entidades podem ser imediatamente enviadas para processamento. Assim a entidade E1 é enviada para a máquina M1 e a entidade E3 para a máquina M3. Os instantes de conclusão previstos são respectivamente de 4 e 3. O problema de decisão acontece apenas com as entidades E2 e E4 pois ambas requerem a máquina M2. Das duas entidades a que tem maior valor de MWKR é a entidade E2 com 1+4+4 = 9 enquanto que o valor de MWKR de E4 é de 7. Assim, a E2 vai para M2 e E4 fica em fila de espera.

| T actual | Fila M1 | M1 | T conclusão | Fila M2 | M2 | T conclusão | Fila M3 | M3 | T conclusão |
|---|---|---|---|---|---|---|---|---|---|
| 0 | -- | E1 | 4 | E4 | E2 | 1 | -- | E3 | 3 |
| 1 | E2 | E1 | 4 | -- | E4 | 4 | -- | E3 | 3 |
| 3 | E2 | E1 | 4 | E3 | E4 | 4 | -- | -- | -- |
| 4 | -- | E2 | 8 | E3 | E1 | 7 | -- | E4 | 7 |
| 7 | E4 | E2 | 8 | -- | E3 | 9 | | E1 | 10 |
| 8 | -- | E4 | 9 | -- | E3 | 9 | E2 | E1 | 10 |
| 9[2] | -- | E3 | 12 | -- | -- | -- | E2 | E1 | 10 |
| 10[3] | -- | E3 | 12 | -- | -- | -- | -- | E2 | 14 |
| 12 | -- | -- | -- | -- | -- | -- | -- | E2 | 14 |
| 14 | -- | -- | -- | -- | -- | -- | -- | -- | -- |

**Figura 3.9 Representação do programa em forma de tabela.**

O próximo instante em que haverá um evento é o instante 1, instante de conclusão de E2 em M2. Nesse instante E2 será transportada para a fila de espera da máquina M1. O processamento de E4 é iniciado. O processo continua nesta lógica até que todas as entidades estejam concluídas (ver figura 3.9). O *makespan* obtido pelo uso desta regra será de 14.

Para o mesmo problema, usando a aplicação LEKIN®, com a regra Minimum Slack, obtemos um programa com um *makespan* também de 14. Esse programa, em diagrama de Gantt e tal qual aparece na referida aplicação é apresentado na figura 3.10.

[2] Instante em que E4 conclui todas as operações

[3] Instante em que E1 conclui todas as operações

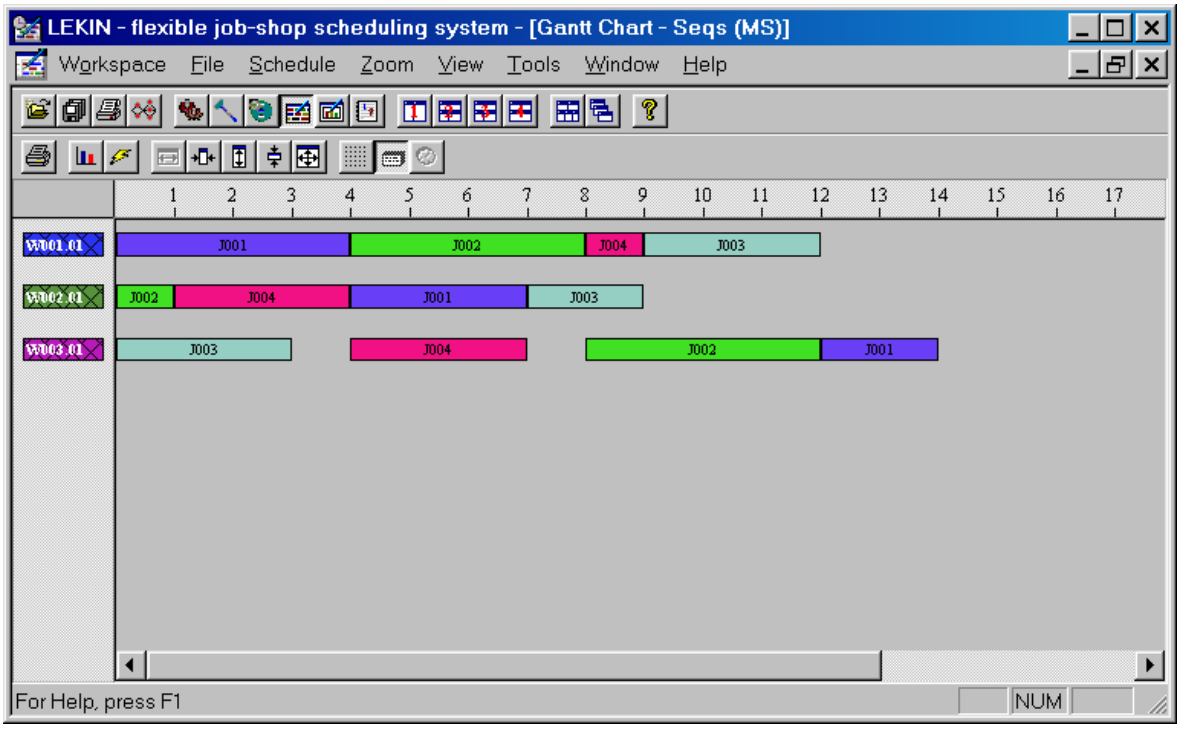



**Figura 3.10 – Aspecto do programa obtido pela aplicação LEKIN. Uso da regra MS.**

O mesmo problema, mas desta vez resolvido pelo uso da heurística "Shifting Bottleneck" com a mesma aplicação, foi obtido um programa com um makespan ligeiramente menor (13). Esse programa é apresentado na figura 3.11.

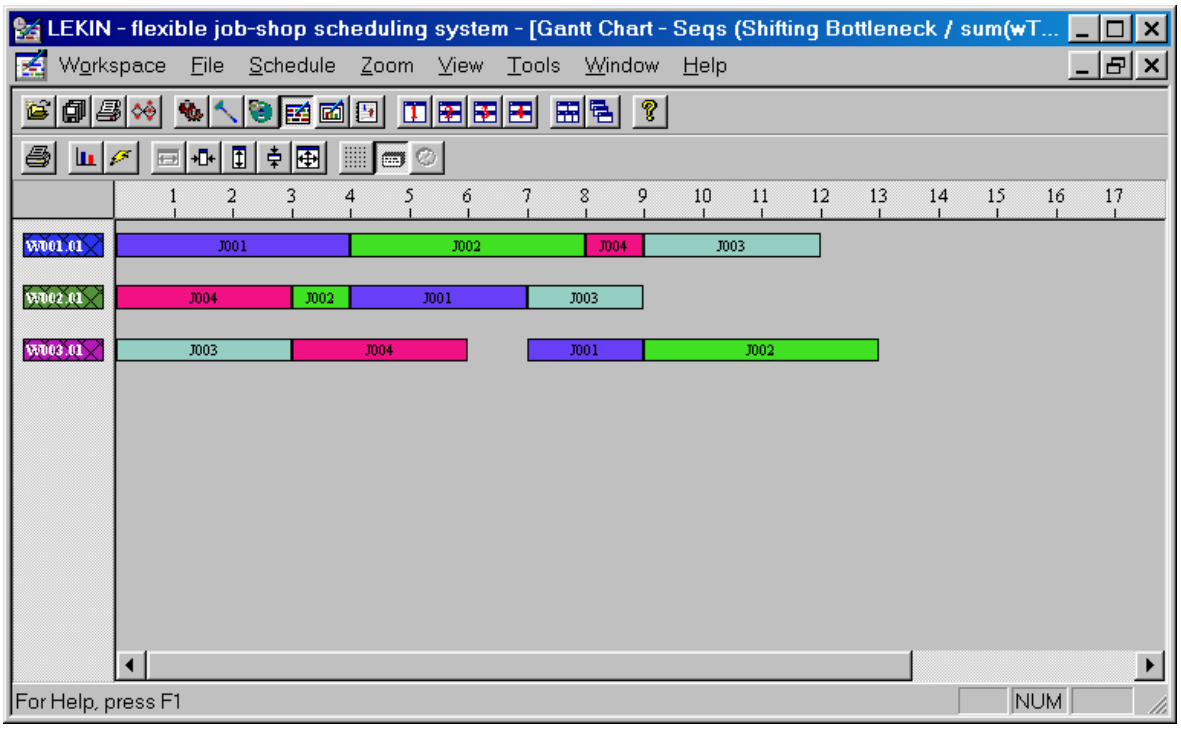



Figura 3.11 – Aspecto do programa obtido pela aplicação LEKIN. Uso da heurística "Shifting Bottleneck".

**Procedimentos probabilísticos de prioridade**

Tendo como objectivo a selecção de um programa que apresenta uma boa medida de desempenho, poder-se-ia gerar aleatoriamente uma série de programas, avaliá-los e finalmente escolher o que apresentar a melhor medida de desempenho. Outra forma seria a de repetir a criação de programas usando regras de prioridade diferentes. Por exemplo, no problema do *makespan* poder-se-ia obter um programa usando a regra MWKR, depois outro programa usando a regra MOPNR e depois outro com a regra SPT e assim por diante. No final escolher-se-ia o programa que apresentasse o melhor valor do *makespan*.

Uma abordagem similar é a de usar uma família de regras combinando as filosofias de regras de prioridade e amostragem aleatória. Mesmo com boas regras de prioridade

## Referências


Baker K R, 1974, *Introduction to sequencing and scheduling*, Jonh wiley & sons, 1974

Johnson S M, 1954, “Optimal two- and three-stage production schedules with setup times included”, Naval research logistics quarterly, vol.1, no. 1, (Março, 1954).

Karg R and Thompson G L, 1964, “A Heuristic approach to solving traveling salesman problems”, Management science, vol 10, no2 (Janeiro, 1964).

LEKIN®, http://www.ieor.columbia.edu/~andrew/scheduling/Lekin.html

Little J D C, Murty K G, Sweeny D W and Karel C, 1963, “An algorithm for the traveling salesman problem”, Operations research, vol 11, no 6 (Novembro, 1963).

McNaughton R, 1959, “Scheduling with deadlines and loss functions”, Management science, vol.6, no.1 (Outubro, 1959).

Pinedo M e Chao X, 1999, *Operations Scheduling: With applications in manufacturing and services*, McGraw-Hill International Editions, NSBN 0-07-116675-0