MAC 5758/MAC 436 - Introdução ao Escalonamento e Aplicações (à Computação Paralela)

Professor: Alfredo Goldman (gold@ime.usp.br)
Página web: http://www.ime.usp.br/~gold/cursos/mac5758/mac5758.html Inscreva-se na lista de discussão !

Estou colocando abaixo os tópicos que pretendo abordar no decorrer do curso. Como o programa é interdisciplinar (por exemplo, existem cursos inteiros que são voltados para heurísticas), talvez a apresentação de alguns dos tópicos abaixo tenha que ser bem resumida, ou mesmo estendida.

Roteiro do Curso

1. Introdução
   - exemplos de problemas - "insights" de soluções;
   - Notação $\alpha|\beta|\gamma$, definição e nomenclatura;
   - Noções de complexidade
     - algoritmos polinomiais
     - programação dinâmica
     - branch and bound
     - heurísticas
   - Problemas clássicos
     - Escalonamento em uma máquina
     - Escalonamento em máquinas paralelas
     - Problemas "shop"
     - Sistemas de manufatura
2. Aplicações
   - Escalonamento em computação paralela
     - Modelos
     - Atraso de comunicação
     - Escalonamento dinâmico
   - Estudo de problemas reais

Critério de Aprovação

Média aritmética entre uma prova (a princípio no dia 6 de dezembro) e de um trabalho prático. As opções de escolha para o trabalho prático são: seminário, estudo teórico ou implementação (sugiro que esta escolha seja feita no máximo até o começo de outubro).

Bibliografia da introdução:
P. Brucker, *Scheduling Algorithms, 2nd edition*, Springer-Verlag, 1998.
M. Pinedo, *Scheduling: Theory, Algorithms and Systems*, Prentice-Hall, 1995.

Capítulo 1

Motivação

Apesar da definição de escalonamento (*scheduling* em inglês) não ser de difícil compreensão, optamos por começar com uma definição informal para em seguida introduzirmos alguns exemplos.

Definição informal 1 *Um escalonamento é uma alocação, no tempo, de recursos a tarefas, considerando algum critério de otimização.*

1.1 Fábrica de tintas

No problema a seguir poderemos ver vários exemplos de escalonamento, conforme as restrições que serão introduzidas.

Problema 1 *Fábrica de tintas: Em uma fábrica hipotética existem m máquinas capazes de fabricar tintas. Conforme uma lista de pedidos, devem ser produzidas n latas de tinta com tamanhos respectivos $l_1, \ldots, l_n$. Para simplificar o problema supomos as latas e as matérias primas necessárias para a produção de tinta estejam disponíveis em quantidades e tamanhos suficientes.*

Em um primeiro passo podemos relacionar as latas às tarefas e as máquinas aos recursos. Além disto, também podemos impor alguns critérios, intrínsecos ao problema:

- Cada máquina só pode fabricar um tipo de tinta por vez.
- Vamos supor inicialmente que as máquinas são idênticas e podem produzir um litro de tinta por minuto.
- Um pedido só pode ser entregue quando todas as tintas pedidas estejam prontas. A partir desta restrição, o critério de otimização "natural" é minimizar o tempo total de fabricação das tintas, isto é, procurar uma solução na qual a última lata seja produzida no menor tempo possível.

A seguir vamos analisar várias possibilidades para modelar (e talvez resolver) o problema.

1.1.1 Um problema simples?

Para começar, vamos supor que exista apenas um tipo de tinta, isto é, todos as latas vão ser preenchidas com a mesma tinta. Vejamos um exemplo:

Exemplo 1 *Fábrica com duas máquinas, m_1 e m_2 e um pedido para fabricar 5 latas com tamanhos 10, 9, 8, 4 e 3.*

A representação de um escalonamento possível pode ser dada pelo seguinte diagrama de espaço-tempo (diagrama de Gantt). O tempo está representado no eixo horizontal (no caso em minutos) e as diferentes máquinas estão representadas no eixo vertical.

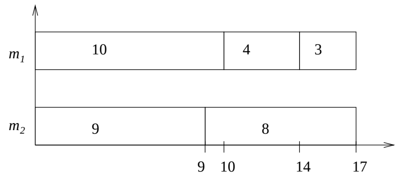


Figura 1.1: Diagrama de Gantt para o exemplo 1.

Da representação da figura 1.1 podemos deduzir que a máquina m_1 fabrica as latas de tinta com 10, 4 e 3 litros, nesta ordem. A lata de 10 litros é preenchida a partir do começo (instante zero) durante 10 minutos, isto é, até o instante 10. A máquina m_2 fabrica as latas com 9 e 8 litros. O tempo total deste escalonamento é de 17 minutos.

Definição 1 *Denomina-se* makespan *o instante de tempo em que a última tarefa termina de ser executada.*

Um alternativa para a representação espaço-tempo é a representação tarefa-tempo (ver figura 1.2). Existem problemas de escalonamento onde este segundo tipo de diagrama pode ser mais útil do que o primeiro.

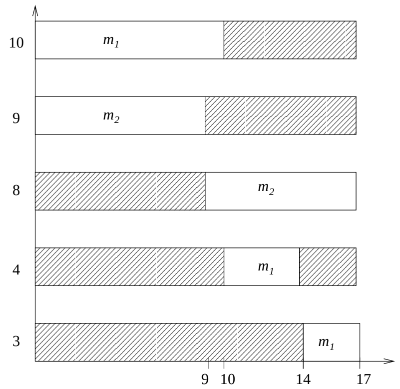


Figura 1.2: Diagrama de tarefa-tempo para o exemplo 1.

Os retângulos hachurados correspondem aos instantes em que as respectivas tarefas não estão alocadas a nenhuma das máquinas disponíveis.

Observando o exemplo 1 e a solução da figura 1.1 não é difícil de deduzir que esta é ótima. Intuitivamente, as máquinas estão ocupadas do início ao fim do escalonamento. Mais exatamente, o tempo mínimo para o problema com m máquinas e n latas com tamanhos $l_1, \ldots, l_n$ é o máximo entre o trabalho total dividido pelo número de máquinas e o tamanho da maior lata. Se denotarmos este tempo por LB, temos:

$$LB = \max\{\max_{i=1}^{n} l_i, \sum_{i=1}^{n} l_i/m\}.$$

Logo para o exemplo 1, LB=17, que é o makespan do escalonamento da figura 1.1.

Observando que na busca por um escalonamento ótimo, é mais "difícil" encontrar a alocação para latas maiores podemos deduzir a seguinte idéia: alocar primeiro as maiores latas. Como a alocação consiste também da escolha do recurso e do intervalo de tempo, é interessante alocar cada tarefa no primeiro intervalo de tempo disponível. A partir destas idéias podemos propor o seguinte algoritmo:

1. Ordene as latas em ordem decrescente de tamanho $(l_1, \ldots, l_n)$;
2. Para i =1 até n
3. aloque l_i na máquina que possa executá-la mais cedo.

Este tipo de algoritmo onde as tarefas são primeiro ordenadas para em seguida serem alocadas segundo um critério dado são conhecidos como algoritmos de lista. No caso, o algoritmo que ordena as tarefas em ordem decrescente e aloca as tarefas o quanto antes nas máquinas disponíveis é conhecido por LPT (*Longest Processing Time First*).

Vejamos agora o seguinte exemplo:

Exemplo 2 *Fábrica com duas máquinas, m_1 e m_2 e um pedido para fabricar 5 latas com tamanhos 9, 9, 6, 6 e 6.*

Aplicando o algoritmo LPT obtemos o seguinte diagrama de Gantt da figura 1.3.

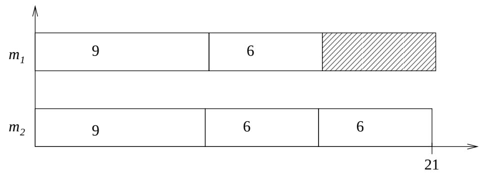


Figura 1.3: Diagrama de Gantt do algoritmo LPT para o exemplo 2.

Entretanto, observando a figura 1.4 podemos constatar que o escalonamento fornecido pelo algoritmo LPT não foi ótimo. O escalonamento da figura 1.4 tem makespan 3 minutos menor.

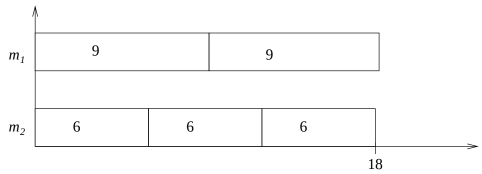


Figura 1.4: Diagrama de Gantt de uma solução ótima para o exemplo 2.

É interessante observar que nem sempre a existência de tempos de inatividade caracteriza soluções que não são ótimas. Observe o exemplo seguinte:

Exemplo 3 *Fábrica com duas máquinas, m_1 e m_2 e um pedido para fabricar 3 latas com tamanhos 4, 4 e 4.*

A solução fornecida pelo algoritmo LPT, que no caso também é uma solução ótima, está representada na figura 1.5.

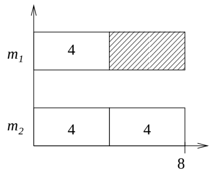


Figura 1.5: Diagrama de Gantt de uma solução ótima para o exemplo 3.

Encontrar a solução ótima para este problema não é fácil, isto é, este problema é considerado computacionalmente difícil. Quando o número de máquinas é igual a 2, o problema é equivalente ao problema de particionar um conjunto de números inteiros em dois conjuntos cujas somas tenham o mesmo valor [1].

Também é interessante notar que o algoritmo LPT, apesar de nem sempre fornecer a solução ótima, garante que o makespan do escalonamento encontrado é no máximo $\frac{4}{3} - \frac{1}{3m}$ vezes pior que o makespan de um escalonamento ótimo. Nós também veremos como este resultado foi encontrado no decorrer do curso.

Exercício 1 *No exemplo 2 o algoritmo LPT é $\frac{7}{6}$ maior que o ótimo. Caracterize situações para $m \geq 3$ em que o limite $\frac{4}{3} - \frac{1}{3m}$ é alcançado.*

Dica: para m=2 uma solução é $(3, 3, 2, 2, 2)$, para $m = 3$ uma solução é $5, 5, 4, 4, 3, 3, 3$.

1.1.2 Interrupção da fabricação de uma lata - Preemption

Uma das formas de simplificar o problema anterior é permitir que uma lata de tinta seja preenchida por diferentes máquinas. Na figura 1.6 vemos um escalonamento que preenche uma das latas do exemplo 3 em duas máquinas diferentes.

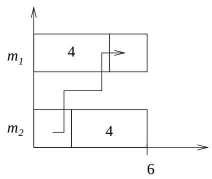


Figura 1.6: Diagrama de Gantt de uma solução ótima para o exemplo 3.

Neste caso, uma das latas começa a ser preenchida em uma máquina, para em seguida ser completada em outra máquina. Para este caso, o desempenho do algoritmo LPT não é bom, pois ele não aproveita a possibilidade de se alocar uma mesma lata em máquina diferentes.

Como temos que considerar o tamanho da maior lata, pois apesar de podermos encher uma lata em duas máquinas diferentes não podemos fazer isto simultaneamente, o tempo mínimo para resolver este problema ainda é igual a $LB = \max\{\max_{i=1}^{n} l_i, \sum_{i=1}^{n} l_i\}$. Uma idéia para um algoritmo é de ocupar as máquinas, uma a uma até o limite LB. Caso alguma lata ultrapasse este limite, resolvemos o problema fazendo com que esta lata seja pré preenchida por outra máquina.

1. Ordene as latas em ordem decrescente de tamanho $(l_1, \ldots, l_n)$;

[1] Formalmente este problema é $\mathcal{NP}-$completo, estudaremos este assunto mais para a frente.

2. $LB = \max\{\ \max_{i=1}^{n} l_i, \sum_{i=1}^{n} l_i\}$;
3. $j = 1$;
4. Para i =1 até n
5. aloque l_i na máquina m_j;
6. se a quantidade de tinta produzida por $m_j \geq LB$
7. aloque a quantidade de tinta em $m_j - LB$ em m_{j+1};
8. $j = j + 1$;

O algoritmo dado é linear no número de latas da instância. Entretanto o tempo necessário para transportar uma lata de uma máquina a outra não é considerado.

Exemplo 4 *Três máquinas e quatro latas de tamanho 6.*

Na figura 1.7 podemos ver o resultado do algoritmo de escalonamento com interrupção. Por exemplo, na máquina m_1 quando da alocação da segunda lata, o tempo $LB = 8$ foi ultrapassado e a lata passou a ser pré preenchida pela máquina m_2.

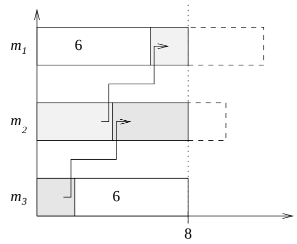


Figura 1.7: Escalonamento com interrupção para o exemplo 4.

Quando o tempo de transporte é considerado, nem sempre o algoritmo anterior fornece uma solução ótima.

Exemplo 5 *Três máquinas, sendo o tempo de transporte entre duas máquinas igual a 2 minutos e o tamanho das latas* 6, 6, 4, 3, 2.

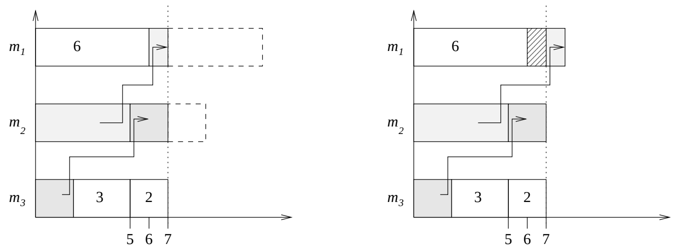


Figura 1.8: Escalonamentos do exemplo 5. O da esquerda não considera o custo do transporte, logo não é válido. O da direita considera um custo de transporte 2.

1.1.3 Cada máquina produz uma cor básica

Neste caso, as latas que devem ser preenchidas com cores compostas devem passar por mais de uma máquina diferente.

Exemplo 6 *Dadas três máquinas onde cada uma produz uma cor básica (magenta, azul e amarelo) produzir 5 latas de tinta sendo: 2 litros de tinta azul, 1 litro de verde (amarelo+azul), 3 litros de marrom (magenta+azul), 2 litros de laranja (magenta+amarelo) e 1 litro de preto (mistura de todas).*

1.1.4 Cada máquina produz todos os tipos de cores

Apesar deste problema parecer inicialmente mais simples, com hipóteses realistas, este problema é na verdade muito complicado (na verdade muito mais difícil que o problema da seção 1.1.1. Se por exemplo, considerarmos que para mudar a produção de uma cor c_i para uma cor c_j o custo é de c_{ij}, veremos que este problema em uma só máquina é equivalente ao TSP (*Traveling Salesman Problem*).

Exemplo 7 *Dado um pedido de* 4 *latas de* 5 *litros de tinta diferentes, com cores* $c_1, \ldots, c_4$*, e uma máquina que produz todos os tipos de cores, onde o tempo para mudar da produção de uma cor* c_i *para uma cor* c_j *é* c_{ij}*, queremos minimizar o custo de produção de uma série destes pedidos.*

No exemplo anterior, como só existe uma máquina e o tempo de produção de cada lata é o mesmo, o importante é determinar a ordem em que as cores serão produzidas. O diagrama de Gantt de um escalonamento será parecido com o da figura 1.9, sendo que a cada vez o ciclo de produção destas quatro tintas se repete.

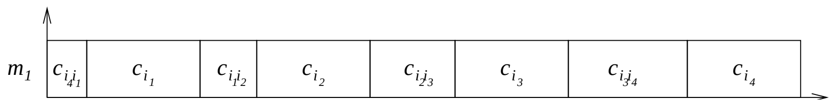


Figura 1.9: Formato dos escalonamentos para o exemplo 7.

Se as tintas são produzidas na seguinte ordem de cor $c_{i_1}, c_{i_2}, c_{i_3}, c_{i_4}$, o makespan deste escalonamento será igual a $20 + c_{i_4 i_1} + c_{i_1 i_2} + c_{i_2 i_3} + c_{i_3 i_4}$. Logo o problema de minimização se resume a encontrar uma ordem de fabricação de tintas tal que a soma dos custos de transição entre uma cor e a cor seguinte seja minimizada.

Observe agora o grafo da figura 1.10. Neste grafo temos 4 vértices, numerados de 1 a 4, assim como arcos com peso c_{ij} entre cada vértice i e cada vértice j, $i \neq j$.

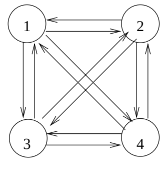


Figura 1.10: Grafo dirigido, onde o peso do arco do vértice i ao vértice j é c_{ij}.

Um circuito $C = (i_1, i_2, i_3, i_4)$ no grafo da figura 1.10 corresponde exatamente à produção das cores na ordem $c_{i_1}, c_{i_2}, c_{i_3}, c_{i_4}$. Logo minimizar o peso do circuito é equivalente a minimizar o tempo de escalonamento de um pedido. Quando os pesos das arestas têm como significado a distância entre duas cidades o problema é conhecido por TSP (Traveling Salesman Problem).

1.2 Escalonamento em computação paralela

Uma aplicação bem interessante do escalonamento é a elaboração de programas paralelos, a partir de algoritmos sequenciais. Dada uma aplicação seqüencial, a mesma pode ser decomposta em tarefas, as quais obedecem relações de precedência. Qualquer alocação em um único processador que respeite as relações de precedência, sem inatividade do processador entre as tarefas, terá tempo de execução seqüencial ótimo.

Por outro lado, quando dispomos de vários processadores, devemos encontrar uma atribuição de processadores e intervalos de tempo para cada uma das tarefas. Um dos objetivos naturais é o de minimizar o tempo de execução paralela, isto é, minimizar o makespan. Para isto devemos encontrar entre as alocações válidas, a com o menor tempo de execução.

Entre as várias dificuldades, antes mesmo de tratar o problema de escalonamento, podemos citar: escolher uma aplicação seqüencial, encontrar uma decomposição conveniente da aplicação em tarefas e encontrar um compromisso no modelo para a máquina paralela entre o realismo do mesmo e a dificuldade de trabalhar com o mesmo.

Exemplo 8 *Multiplicação matriz vetor.*

Dadas uma matriz A, $n \times n$ onde n é par, e um vetor x de tamanho n, calcular $b = A \times x$. Particionando a matriz e o vetor em blocos temos:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}.$$

O grafo de precedência que representa as tarefas está desenhado na figura 1.11. Não é difícil de ver que cada um dos quatro nós mais a esquerda, podem eles mesmos, serem divididos em sub-árvores com o mesmo formato. Conforme o tamanho de cada tarefa no grafo, define-se a granularidade do grafo de precedência.

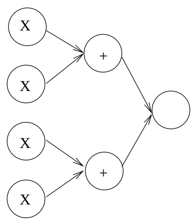


Figura 1.11: Grafo de precedência do algoritmo de multiplicação matriz vetor. Os nós mais a esquerda correspondem as multiplicações $A_{ij} \times b_k$, os nós do meio a somas parciais para a obtenção de b_1 e b_2 e o nó da direita a junção de b_1 e b_2 para obter b.

Na figura seguinte temos um exemplo de escalonamento para o grafo da figura 1.11. As tarefas que efetuam as multiplicações das sub matrizes são maiores que as que fazem a soma, que são maiores do que a que une b_1 e b_2 em um único vetor.

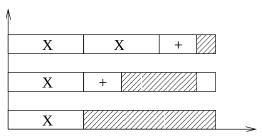


Figura 1.12: Escalonamento do grafo da figura 1.11 em três processadores.

1.3 Campeonato de Futebol

Nos dois próximos exemplos, veremos dois problemas de escalonamento difíceis que não possuem formas de modelagem padrão através de técnicas de escalonamento. Nos dois casos, um algoritmo que encontra uma solução ótima para os problemas deve verificar todas as possibilidades, que são em número exponencial no número de entradas.

Dados vários campeonatos de futebol, cada um com o seu calendário de jogos pré-determinado (ex: Copa do Brasil, Libertadores e Campeonatos estaduais), determinar uma atribuição de equipes a estas vagas de tal forma que a chance de uma equipe jogar duas partidas em menos de 48 horas seja minimizada.

Por exemplo, no primeiro semestre de 2000 vários campeonatos e copas foram realizados, alguns times só participaram de campeonatos estaduais, outros participaram dos campeonatos estaduais e da Copa do Brasil, poucas equipes participaram de duas copas e do campeonato estadual respectivo. Sabendo que as copas são disputadas no sistema eliminatório com duas partidas, é fácil conhecer todas as datas possíveis de jogos segundo a chave em que a equipe é alocada. O mesmo é válido para os campeonatos, onde após o turno e returno, um sistema eliminatório é colocado em prática para as quatro equipes com mais pontos.

Vários jornais noticiaram no começo do campeonato que equipes como o Palmeiras e o Corinthias teriam partidas em datas conflitantes caso chegassem ao final das três competições. Sabendo-se que existem equipes com mais chance de chegar à final das competições (por exemplo: a chance do Palmeiras chegar à final de uma competição é maior do que a do Atlético Paranaense), pode-se criar calendários com pouca chance de conflitos?

Exemplo 9 *Consideremos o seguinte exemplo: no decorrer de um mês 8 equipes participam de uma copa, sendo que quatro delas também participam de um campeonato. Na figura 1.13 vemos as fases da copa, as quartas de final devem ocorrer até o dia 10, as semi-finais do dia 11 ao dia 20, e a final a partir do dia 21.*

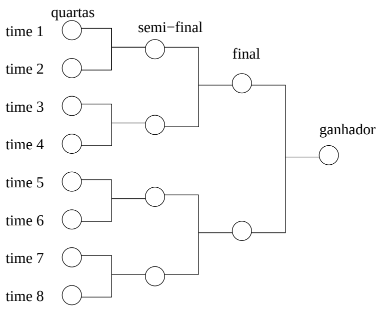


Figura 1.13: Copa com 8 equipes.

Por outro lado, para o campeonato da figura 1.14, a rodada i deve ocorrer entre os dias $5*(i-1)+1$ e $5*i$. Como determinar as datas em que cada equipe deve jogar?

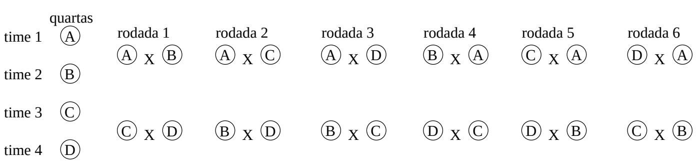


Figura 1.14: Campeonato com 8 equipes.

Para resolver este problema podemos associar os jogos a tarefas e os times aos recursos. Para o campeonato, cada tarefa é composta de 6 partes, cada uma correspondendo a uma rodada. Como não podemos conhecer de antemão quem estará na final da copa, todos as tarefas serão compostas de 3 partes, sendo que algumas delas, conforme o resultados dos jogos não serão executadas.

Uma unidade de tempo natural para resolver o problema são os dias, mais especificamente os dias do mês. Dado um escalonamento, a sua pontuação pode ser dada pelo número de máquinas (no caso equipes) que são utilizadas mais do que uma vez por período de 3 dias. Finalmente, a execução de uma tarefa fora das datas pré-fixadas pode ser penalizada com $+\infty$. Claramente, um escalonamento com custo zero respeita as 48 horas de descanso necessárias para os jogadores.

Uma forma de resolver este problema, sem enumerar todas as soluções possíveis, é a partir de várias soluções iniciais, algumas talvez com custo ∞, procurar soluções melhores. No caso esta procura é feita localmente, observando as soluções vizinhas. Sendo estas caracterizadas ou por uma mudança de data de entre jogos de uma mesma equipe, ou através da mudança de posição de duas equipes na tabela de uma das competições. Veremos mais adiante em detalhes a aplicação deste método.

Um exemplo de escalonamento do exemplo 9 pode ser visto na figura 1.15. Além de respeitar os critérios originais do problema, este escalonamento também maximiza o número de dias com jogos de futebol no mês. Os quadrados com C representam jogos da copa e os quadrados com R jogos do campeonato, para saber a fase de cada uma das competições basta contar o número de jogos da competição a partir do início do mês. É interessante ressaltar que apesar de apenas duas equipes disputarem a final da copa, foi necessário "alocar" a data no calendário de todas.

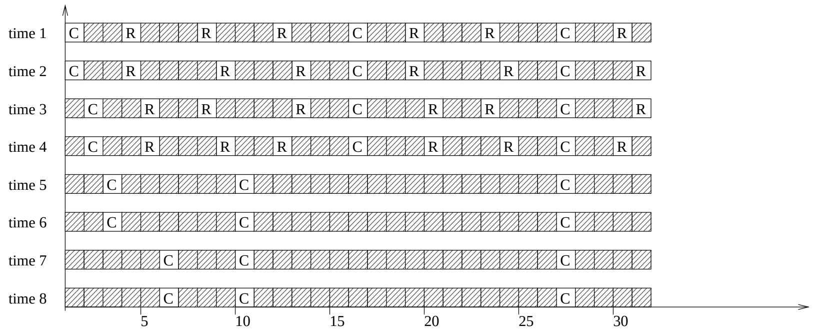


Figura 1.15: Calendário com as datas dos jogos.

1.4 Atribuição de carga didática

Como, neste exemplo, as modelagens podem ser bem variadas, vamos apenas apresentar o problema e sugerir alguns fatores que devem ser levados em conta. Dados um elenco de disciplinas e um quadro de docentes, encontrar uma alocação de docentes aos cursos, que também devem ser alocados a uma grade horária que maximize as prioridades iniciais de escolha das disciplinas pelos docentes.

Naturalmente para o problema acima algumas configurações devem ser penalizadas com $+\infty$, entre elas podemos citar: alocar o mesmo docente para disciplinas diferentes com intersecção de horário e alocar disciplinas para os alunos de uma mesma turma em horários conflitantes. Outras configurações também podem ser penalizadas, como alocar mais do que um máximo, ou menos do que um mínimo de carga horária por docente e deixar "buracos" entre as disciplinas da mesma turma.

Outros critérios interessantes são: permitir aos docentes que digam qual as disciplinas que preferem ministrar e permitir também a escolha do período de aulas. Como nem sempre vai existir um escalonamento que respeite todas as condições dadas, no caso da busca de uma solução para o problema, é bem interessante a participação interativa de um programador, que colocará critérios adicionais para resolver o problema (ex: senioridade dos docentes e considerar as disciplinas para quais os docentes são capazes de ministrar, ao invés da preferência dos mesmos).

Capítulo 2

Introdução

Após termos visto alguns exemplos, assim como sua definição informal, neste capítulo apresentaremos escalonamento formalmente. Primeiramente veremos a definição e em seguida a classificação de três campos, a qual é muito utilizada na literatura.

2.1 Definição

Antes de podermos definir escalonamento, precisamos definir algumas notações relativas aos recursos e às tarefas. Na maior parte dos problemas consideramos os recursos, representados por um conjunto de m máquinas, denotamos cada máquina por m_i e o conjunto das máquinas por $\mathcal{M} = \{m_1, \ldots, m_m\}$ e n tarefas $\mathcal{T} = \{t_1, \ldots, t_n\}$.

Associadas às tarefas temos as seguintes características:

tempo de execução (p_{ij})**:** Tempo necessário para a execução completa da tarefa t_j na máquina m_i. Caso as máquinas sejam idênticas este tempo é denotado apenas por p_j. No caso de máquinas especializadas, isto é, quando a tarefa t_j só pode ser executada em um subconjunto M_j de $\mathcal{M}$, p_{ij} só é definido para $i \in M_j$.

data de disponibilidade (r_j)**:** Corresponde ao instante em que a tarefa t_j está disponível, isto é, o primeiro instante em que a tarefa t_j pode começar a ser executada.

data devida (d_j)**:** Instante de tempo no qual a tarefa t_j deve estar pronta. Conforme o problema, o término da execução da tarefa t_j após d_j é permitido, mas existe uma penalização associada no critério de otimização.

A partir destas características podemos definir um escalonamento:

Definição 2 *Um escalonamento é uma atribuição de tarefas a recursos, no tempo, que obedece às seguintes propriedades:*

- *cada tarefa é atribuída a uma máquina por vez, isto é, uma tarefa não pode ser executada por mais do que uma máquina simultaneamente;*
- *cada máquina executa no máximo uma tarefa por vez;*
- *a tarefa t_j não é executada antes da sua data de disponibilidade (r_j);*

- *todas as tarefas são executadas por completo.*

A definição anterior é bem geral e é válida para a maioria dos problemas de escalonamento clássicos. Geralmente, conforme as condições do problema, um escalonamento deverá obedecer propriedades adicionais. É interessante ressaltar que também existem problemas de escalonamento para os quais algumas das propriedades da definição anterior não são válidas; por exemplo, existem alguns modelos mais recentes nos quais uma mesma tarefa pode ser executada simultaneamente em máquinas distintas.

2.2 Notação

Com o objetivo de se criar uma classificação dos problemas de escalonamento, uma nomenclatura básica, que classifica os diversos tipos de problema de escalonamento, foi introduzida por [1]. A notação é composta por três campos $\alpha|\beta|\gamma$, onde o primeiro campo descreve os recursos, o segundo as tarefas e finalmente o último o critério de otimização.

2.2.1 Recursos - campo α

Os tipos mais comuns de recursos são às máquinas paralelas e as máquinas tipo "shop" (especializadas). No primeiro caso as tarefas podem ser completadas por uma única máquina. Para o segundo caso existem tarefas que podem precisar ser executadas em máquinas distintas.

Máquinas paralelas

Existem, com relação as máquinas paralelas, três parâmetros principais: a quantidade, o desempenho e a especifidade. O número de máquinas pode variar de uma a tantas quantas forem necessárias (geralmente denota-se este último caso através de uma quantidade infinita de máquinas). O desempenho das máquinas pode ser idêntico ou existirem máquinas com velocidades diferentes. O desempenho pode inclusive depender da tarefa a ser executada. Finalmente, podem existir tarefas que necessitem de um tipo de máquina específico para serem executadas. As principais variações estão descritas abaixo:

1: Uma máquina;

P_m **ou** P**:** m máquinas idênticas. Todas as máquinas podem executar todas as tarefas com a mesma velocidade. Caso existam restrições quanto à máquina onde a tarefa t_j pode ser executada, por exemplo em um subconjunto M_j de P_m, estas restrições farão parte do segundo campo (β).

P_∞ **ou** $\overline{P}$**:** número ilimitado de máquinas idênticas;

Q_m**:** m máquinas com velocidades diferentes, onde a velocidade de cada máquina é chamada de v_i. Logo a velocidade de execução não depende da tarefa executada. Se uma tarefa t_j é executada em uma máquina m_i o seu tempo de processamento p_{ij} é igual a p_j/v_i. Este ambiente também é conhecido por máquinas uniformes;

R_m**:** Neste caso, as velocidades de processamento das tarefas nas m máquinas não obedecem à mesma razão. A execução de uma tarefa depende ao mesmo tempo da máquina e da tarefa. A máquina m_i pode executar a tarefa t_j com velocidade v_{ij}. Logo o tempo de execução da tarefa t_j na máquina m_i é igual a p_j/v_{ij}.

É interessante observar que o ambiente P_m é um caso particular do ambiente Q_m, o qual também é um caso particular do ambiente R_m. Também podemos encontrar os ambientes do tipo Q e R com um número não limitado de máquinas.

Máquinas Shop

Em sistemas de produção as máquinas são consideradas como dedicadas, ao invés de paralelas como anteriormente. Veremos os três principais modelos de sistemas de produção.

Flow-shop - F_m: No modelo flow-shop, todas as tarefas têm o mesmo número de operações as quais são executadas na mesma ordem, por todas as m máquinas em série. Após a execução em uma máquina, a tarefa entra na fila para execução na próxima máquina. Geralmente, as filas respeitam a disciplina FIFO (*first-in, first-out*);

Open-shop - O_m: Neste modelo as tarefas também devem ser executadas por todas as m máquinas; entretanto, alguns dos tempos de execução podem ser nulos. Não existem restrições quanto à ordem das máquinas onde cada uma das tarefas deve passar;

Job-shop - J_m: Neste modelo cada uma das tarefas possui um roteiro pré-determinado. O número de máquinas, como nos modelos anteriores, também é m. Um tipo especial de job-shop onde uma tarefa pode visitar uma máquina mais do que uma vez, é denominado por recirculação (com a palavra *recrc* no segundo campo).

2.2.2 Tarefas - campo β

Nós já vimos na introdução algumas das características das tarefas, como tamanho (tempo de execução), data de disponibilidade e data de término. Veremos agora algumas das outras possibilidades mais comuns:

Tempo de preparação da máquina - s_{jk}: Dadas duas tarefas t_j e t_k, s_{jk} corresponde ao tempo de preparação de uma máquina que executou uma tarefa t_j e vai executar uma tarefa t_k. Caso este tempo também dependa da máquina m_i em questão, usa-se a notação s_{ijk}. No caso de t_j ser a primeira (ou última) tarefa executada na máquina, o tempo de preparação e denotado s_{0j} (s_{j0}).

Relações de precedência - *prec*: Quando existe, a ordem para a execução de tarefas geralmente é representada por um grafo orientado, onde os vértices correspondem às tarefas e os arcos às relações diretas de precedência. Uma arco de uma tarefa t_j a uma tarefa t_k faz com que a tarefa t_k possa iniciar sua execução apenas após o término da tarefa t_j. Caso o grafo de precedência pertença a uma família específica, ao invés de *prec* coloca-se o nome da família no segundo campo. Os mais comuns são: *intree, outree, chains, split-compute-merge*, entre outros.

Tempo de comunicação entre máquinas - c_{jk}: Esta possibilidade está sempre ligada a algum tipo de relação de precedência entre as tarefas. No caso de uma relação direta de precedência entre duas tarefas t_j e t_k, se a tarefa t_k finaliza a sua execução no instante t na máquina m_i. A tarefa t_k não pode iniciar a sua execução antes do instante t, na mesma máquina m_i, ou do instante $t + c_{jk}$ em qualquer outra máquina.

Interrupção - *prmp*: Quando é possível interromper uma tarefa, ela não precisa necessariamente terminar na mesma máquina em que começou a sua execução. O tempo de execução utilizado até a interrupção não é perdido. Quando a tarefa volta a uma máquina, ela só precisa ser executada durante o tempo restante após a última interrupção.

Restrições de escolha de máquina - M_j: Este ítem aparece no segundo campo quando a tarefa t_j só pode ser executada em um subconjunto $M_j \subset \mathcal{M}$.

Além das possibilidades anteriores, também existem várias possibilidades para algoritmos do tipo flow-shop, como: permutação (prmu) - quando a fila não é necessariamente FIFO, bloqueio (block) - quando a fila tem tamanho limitado e sem espera (nwt) - quando a fila tem tamanho zero.

No caso de novos problemas de escalonamento, que não se adaptem às variações precedentes, pode-se definir novos parâmetros para os campos, desde que os campos recursos, tarefas e critério sejam respeitados.

2.2.3 Critério de otimização - campo γ

Para podermos definir os principais critérios de otimização, precisamos antes denotar o instante em que cada tarefa t_j termina a sua execução. Este instante é normalmente chamado de C_j. Para cada tarefa é associada uma função custo $f_j(C_j)$. A função de custo total para a maioria dos critérios de otimização correspondem ao máximo,

$$\max\{f_j(C_j)|1 \leq j \leq n\},$$

ou a soma de todos os custos

$$\sum_{j=1}^{n} f_j(C_j).$$

O problema de escalonamento consiste em encontrar uma solução que minimize a função de custo total. As funções mais comuns são:

Makespan - $C_{\max}$: definido como o instante de tempo em que a última tarefa é finalizada,

$$C_{\max} = \max\{C_1, \ldots, C_n\}.$$

Neste caso o critério corresponde ao máximo e as funções $f_j()$ são funções identidade.

Tempo de total de término - $\sum C_j$: definido como a soma dos instantes de término de cada tarefa,

$$\sum C_j = \sum_{j=1}^{n} C_j.$$

Tempo de total de término ponderado - $\sum w_j C_j$: definido como a soma dos instantes de término de cada tarefa com peso w_j,

$$\sum C_j = \sum_{j=1}^{n} w_j C_j.$$

Outras funções comuns são:

$L_j = C_j - d_j$	lateness (grau de atraso, pode ser negativo)
$E_j = \max\{0, d_j - C_j\}$	earliness (grau de "adiantamento")
$T_j = \max\{0, C_j - d_j\}$	tardiness (grau de atraso efetivo)
$U_j = \begin{cases} 0 \text{ se } C_j \leq d_j \\ 1 \text{ caso contrário} \end{cases}$	penalidade unitária por atraso

A melhor forma de visualizar o comportamento das funções acima é através de uma análise gráfica. Na figura 2.1 podemos ver cada uma das curvas.

Entre os critérios mais comuns estão os seguintes critérios:

- Tardiness total ponderada - ($\sum w_j T_j$);
- Total ponderado de tarefas atrasadas - ($\sum w_j U_j$);
- Atraso máximo - ($\max L_j$).

Da mesma forma que o makespan $\max C_j$ é denotado por $C_{\max}$, os critérios de minimização também são denotados por $L_{\max}, E_{\max}, T_{\max}$ e $U_{\max}$.

Exercício 2 *Podemos ver que existem funções que penalizam o atraso (L_j) ou o término antes do tempo (E_j). Escreva uma função que penaliza ao mesmo tempo o término adiantado, ou atrasado. Modifique a função anterior, de forma que um intervalo de tolerância de ϵ da data d_j não seja penalizada.*

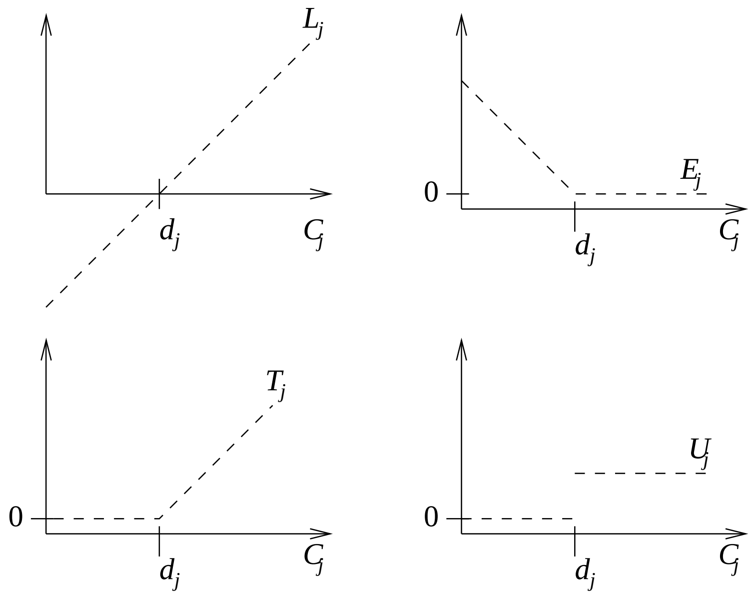


Figura 2.1: Gráficos de diversas funções comuns em escalonamentos.

2.3 Exemplos

Com o intuito de facilitar o aprendizado das diversas notações para os problemas de escalonamento, veremos alguns exemplos.

2.3.1 $P_m|prec; p_j = 1|C_{\max}$

É o problema de escalonar tarefas com tempo de execução 1 e relações de precedência arbitrárias em m máquinas idênticas, de forma que o makespan seja minimizado.

Uma instância do problema corresponde a um grafo e uma quantidade de máquinas. A figura 2.2 mostra um grafo de precedência: uma instância do problema corresponde a este grafo e um número determinado de máquinas.

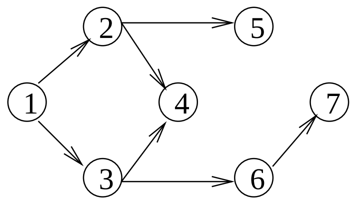


Figura 2.2: Um possível grafo de precedência para problema $P_m|prec; p_j = 1|C_{\max}$.

Na figura 2.3 podemos ver um escalonamento válido em duas máquinas com makespan 5.

Na figura 2.4 podemos ver um escalonamento, para o mesmo grafo, agora em três máquinas. O escalonamento em três máquinas tem makespan 4. É interessante notar que com o uso de uma máquina adicional foi possível encontrar (facilmente) um escalonamento melhor. Mas qual é o escalonamento com o menor makespan, para o grafo da figura 2.2? Observando que o maior caminho entre uma raiz e um vértice terminal (folha) é 4, podemos deduzir facilmente que

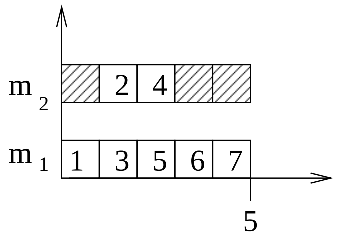


Figura 2.3: Escalonamento do grafo da figura 2.2 em duas máquinas.

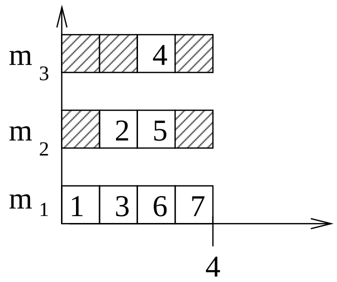


Figura 2.4: Escalonamento do grafo da figura 2.2 em três máquinas.

não é possível encontrar um escalonamento melhor (mesmo com mais máquinas). Este limite inferior, que pode ser encontrado para qualquer grafo de precedência e é denominado *caminho crítico*.

Além do caminho crítico também é interessante observar no grafo o número máximo de tarefas que podem ser executadas simultaneamente, também denominado *largura* de um grafo. No caso desta instância, a largura é 3 (os seguintes subconjuntos de tarefas podem ser executados simultaneamente: $\{4, 5, 6\}$ e $\{4, 5, 7\}$). É importante notar que dois vértices t_j e t_k podem ser executados ao mesmo tempo se e só se não existe um caminho orientado entre t_j e t_k.

Antes de passarmos para o próximo exemplo, um pouco mais de notação. Denotamos os sucessores diretos de um vértice v_1 por $succ(v_1)$, os seus predecessores são denotados por $pred(v_1)$. No grafo da figura 2.2 temos: $succ(3) = \{4, 6\}$, $pred(6) = \{3\}$ e $succ(4) = \{\emptyset\}$.

2.3.2 $P_\infty|prec; dup; p_j = 1; c_{jk} = \gamma|C_{\max}$

Escalonamento de n tarefas com relações de precedência em um número ilimitado de máquinas. Todas as tarefas têm tempo de execução 1, podem ser duplicadas e o tempo de comunicação entre quaisquer duas máquinas é sempre γ. O objetivo é de minimizar o makespan.

Veremos três escalonamentos diferentes, conforme o valor de γ para o grafo split-compute-merge da figura 2.5.

Conforme o custo da comunicação, um escalonamento com makespan ótimo duplica mais ou menos tarefas. A figura 2.6 apresenta um escalonamento para $\gamma = 0.5$. As cópias de uma tarefa são indicadas por índices *linha*. Podemos ver que a duplicação para poder executar as tarefas 9 e 10 no instante 3 sempre é interessante, independentemente do custo da comunicação. Uma eventual duplicação das tarefas 5, 6 ou 7 não permitiria que a tarefa 8 fosse executada mais cedo.

Podemos ver na figura 2.7 um escalonamento onde o custo de comunicação é maior, neste caso $\gamma = 3$. A tarefa 8, não pode começar antes pois a tarefa 7 foi executada em outra máquina, e a comunicação só ficou disponível 3

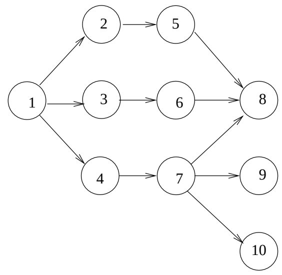


Figura 2.5: Grafo split-compute-merge.

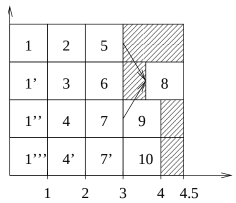


Figura 2.6: Escalonamento do grafo da figura 2.5 para $\gamma = 0.5$ em quatro máquinas.

unidades de tempo após o término da tarefa 7.

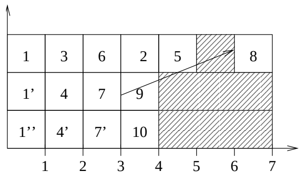


Figura 2.7: Escalonamento em três máquinas do grafo da figura 2.5 para $\gamma = 3$.

Finalmente quando $\gamma = 10$ o uso de uma comunicação entre quaisquer duas tarefas implicaria em um makespan de ao menos 12. Logo a duplicação acaba substituindo a comunicação.

2.3.3 $1|r_j; prmp|L_{\max}$

Este é o problema de encontrar um escalonamento, em uma máquina, com interrupção e datas de disponibilidade. O critério de otimização é minimizar a lateness máxima.

Na tabela 2.1 temos uma instância do problema. onde para cada tarefa t_j temos o seu tamanho, data de disponibilidade e data devida. Um escalonamento possível pode ser visto na figura 2.9. Neste caso $L_{\max} = 4$.

Para esta instância, não é difícil construir um escalonamento melhor, mas para isto vamos construir um escalonamento passo a passo. No instante zero nenhuma tarefa está disponível, logo a máquina não é utilizada. No instante 1 apenas a tarefa t_1 pode ser executada. Já no instante 2, três tarefas podem ser executadas: um pedaço de t_1, t_2 e t_3. Intuitivamente, vamos escolher a tarefa com a data devida mais próxima, isto é, t_1. Continuando com este raciocínio chegaremos ao escalonamento da figura 2.10.

Apesar do escalonamento da figura 2.10 ter o mesmo makespan que o escalonamento anterior, temos $L_1 = 4 - 2 = 2, L_2 = 0; L_3 = 6 - 4 = 2$ e $L_4 = 9 - 8 = 1$.

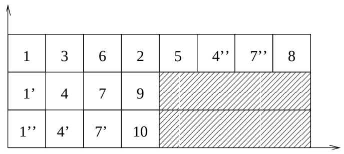


Figura 2.8: Escalonamento em três máquinas do grafo da figura 2.5 para $\gamma = 10$.

j	1	2	3	4
p_j	2	1	2	2
r_j	1	2	2	7
d_j	2	3	4	8

Tabela 2.1: Instância do problema $1|r_j; pmtn|L_{\max}$.

Exercício 3 *Escreva o algoritmo que forneceu o escalonamento anterior. No caso da instância anterior, existe um escalonamento melhor? Poderíamos escolher, para cada tarefa disponível, ao invés da tarefa t_j com data devida mais próxima o máximo da diferença entre o melhor C_j possível e r_j? Qual critério é o melhor?*

2.3.4 $J_3|p_{ij} = 1; recrc|C_{\max}$

Problema de minimizar o makespan de três máquinas job shop onde as tarefas têm tempo de execução unitário e podem passar mais do que uma vez pela mesma máquina.

Uma instância do problema é:

	etapas			
t_j	1	2	3	4
t_1	m_1	m_3	m_2	m_1
t_2	m_2	m_3		
t_3	m_3	m_1		
t_4	m_1	m_3	m_1	
t_5	m_3	m_1	m_2	m_3

Um escalonamento possível está representado na figura 2.11.

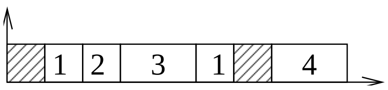


Figura 2.9: Escalonamento para uma instância do problema $1|r_j; prmp|L_{\max}$.

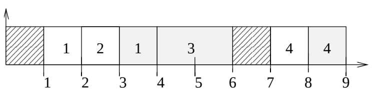


Figura 2.10: Outro escalonamento para uma instância do problema $1|r_j; prmp|L_{\max}$. As tarefas executadas após a data devida estão indicadas em cinza claro.

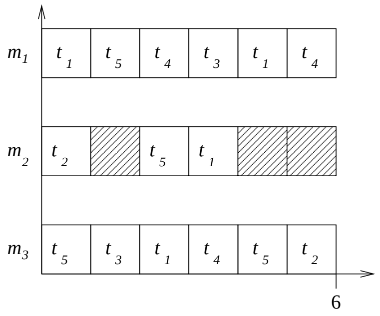


Figura 2.11: Escalonamento para uma instância do problema $J_3|p_{ij} = 1; recrc|C_{\max}$.

2.3.5 $P_m|r_j; d_j; M_j; p_j| \sum w_j T_j$

Ainda não escrevi nada a respeito deste problema. A parte interessante é que ele serve de modelo para o problema seguinte:

> Atribuição de terminais de embarque a aviões:
> "Dadas as características dos aviões e seus horários de partida, assim como a capacidade de cada terminal do aeroporto, encontrar uma atribuição válida de horários e terminais para os aviões, se possível respeitando os horários de partida".

O m corresponde ao número de terminais de embarque, o r_j ao momento em que o avião j está pronto para o embarque (*now boarding :)*), d_j corresponde ao momento em que o avião j tem que estar pronto para a decolagem, M_j o conjunto de terminais em que o avião j pode estacionar, e finalmente p_j é o tempo de embarque para o avião j.

No critério de otimização cada avião j tem um custo de penalização associado ao atraso na decolagem de w_j. A penalização só é aplicada no caso de a decolagem ser após o horário d_j.

Exemplo 10 *Para um aeroporto com 3 terminais, temos os seguintes seis aviões para escalonar:*

j	1	2	3	4	5	6
p_j	2	1	3	3	2	4
r_j	0	1	1	1	1	2
d_j	3	3	5	4	5	7
M_j	$\{1\}$	$\{1,2,3\}$	$\{1,2\}$	$\{3\}$	$\{3\}$	$\{1,2\}$
w_j	10	2	5	2	2	5

Para o qual um escalonamento possível é mostrado na figura 2.12. O valor deste escalonamento é: 2, pois o único avião atrasado é o 5, de uma unidade de tempo e seu peso e 2.

2.3.6 Um exemplo curioso

Este exemplo foi copiado de Pinedo [1]. Antes do exemplo, vejamos a seguinte definição:

Definição 3 *Um escalonamento válido é dito sem atraso se nenhuma máquina fica inativa quando existem tarefas disponíveis para serem executadas.*

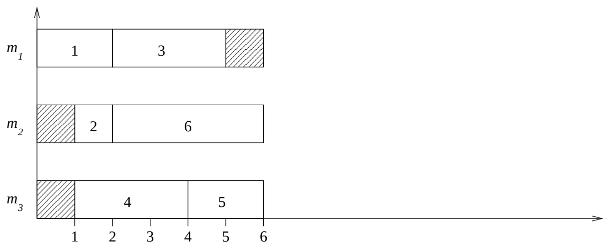


Figura 2.12: Escalonamento possível para o exemplo 10.

Exemplo 11 *Intuitivamente, os escalonamentos sem atraso parecem os melhores. Considere a seguinte instância de* $P_2|prec|C_{\max}$.

j	1	2	3	4	5	6	7	8	9	10
p_j	8	7	7	2	3	2	2	8	8	15

Com o grafo de precedência da figura 2.13.

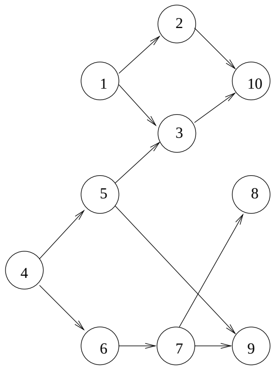


Figura 2.13: Grafo de precedência.

Um escalonamento ótimo com makespan 31 pode ser facilmente obtido. Para isto podemos usar um algoritmo guloso, que escolhe a tarefa com maior caminho crítico. Inicialmente, são alocadas as tarefas 1 e 4; em seguida, após a tarefa 4 alocamos a tarefa 5, pois o caminho até a tarefa 10 tem peso maior do que os outros. A alocação completa está na figura 2.14.

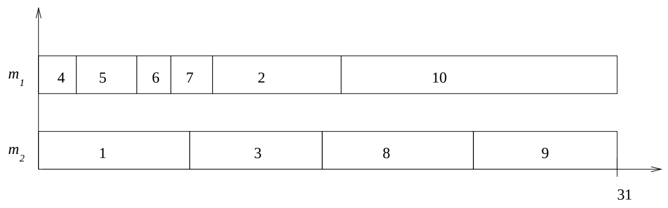


Figura 2.14: Escalonamento em dois processadores.

O que acontece se diminuirmos o tempo de processamento de cada tarefa em uma unidade? Podemos ver um escalonamento sem atraso na figura 2.15. O escalonamento apesar de tarefas de tamanhos menores, ficou com makespan maior.

Fazendo a alocação sem atraso em três processadores temos como resultado um makespan maior (no caso 36), ver figura 2.16.

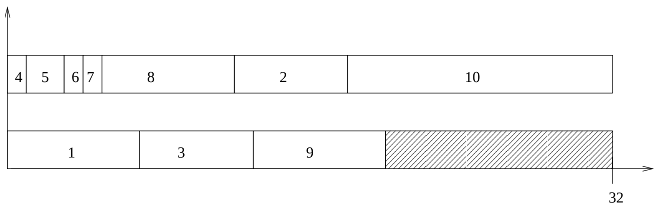


Figura 2.15: Escalonamento, com tarefas menores, em dois processadores.

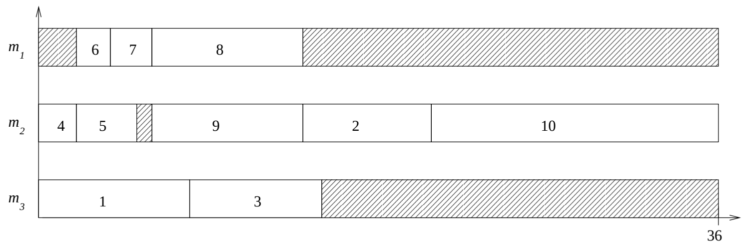


Figura 2.16: Escalonamento em três processadores.

Finalmente na próxima figura (2.17), podemos constatar que a introdução de um tempo de inatividade (quadrado cinza) propiciou um makespan menor. Este, como tem o mesmo tamanho que o caminho crítico, é ótimo.

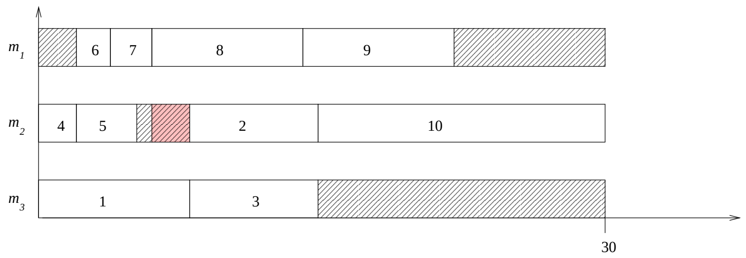


Figura 2.17: Escalonamento ótimo em três processadores.

Capítulo 3

Noções de complexidade

Veremos neste capítulo os conceitos básicos para classificar os problemas de escalonamento segundo o seu grau de dificuldade. Apesar da teoria envolvida ser complexa, vamos nos concentrar em alguns tópicos básicos que serão úteis para a resolução na prática de problemas de escalonamento.

O objetivo principal deste capítulo é de fornecer a base necessária para se analisar problemas de escalonamento, para em seguida propor soluções exatas, ou não, para os mesmos. Para uma descrição mais detalhada sugerimos o livro de Garey e Johnson [1]

3.1 Redução entre problemas de escalonamento

Antes de vermos um pouco da teoria de complexidade, vamos estudar as relações de dificuldade entre alguns problemas de escalonamento. Para isto, vamos definir uma relação de redução que, dados dois problemas, indica que um não é mais difícil do que o outro. Isto é se x se reduz a y sabemos que um algoritmo que resolve y também resolve, sem grandes modificações, x.

Vamos começar analisando os tipos de máquina. Se temos um algoritmo que resolve $R_m|\beta|\gamma$, este mesmo algoritmo também resolve $Q_m|\beta|\gamma$; para isto basta igualar v_{ij} a v_i. Da mesma forma, este algoritmo também resolve $P_m|\beta|\gamma$, igualando $v_{ij} = 1$. Finalmente este algoritmo também serve para resolver $P_2|\beta|\gamma$. Estas relações podem ser vistas na figura abaixo:

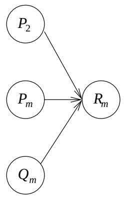


Figura 3.1: Relações de complexidade entre os tipos de máquina.

Entretanto, olhando as reduções acima, representadas por arcos vemos que, na verdade, cada um dos problemas do parágrafo anterior é um caso particular do problema seguinte. Logo, a figura que representa as relações de dificuldade é na verdade a figura 3.2. Esta, além de fornecer as informações da figura anterior pois claramente a operação de redução é transitiva, também fornece a relação entre os problemas com máquinas P_2, P_m e Q_m.

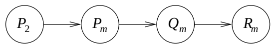


Figura 3.2: Relações de complexidade entre os tipos de máquina.

De uma forma geral temos o grafo de precedência da figura 3.3. Nesta, podemos ver que o caso com apenas uma máquina é um caso particular de todos os problemas. Vemos também que o problema do Flow-Shop é um caso particular do Job-Shop, pois basta assumir no segundo que o roteiro de todas as tarefas é o mesmo, da primeira a última máquina. Finalmente, podemos observar que existem problemas que não estão relacionado com outros.

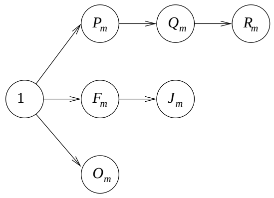


Figura 3.3: Relações de complexidade todos os vários de máquina.

Da mesma forma também existem relações de dificuldade entre os problemas de escalonamento quando variamos o campo das tarefas β. Na figura 3.4 vemos um exemplo do grau de dificuldade do problema com diversas relações de precedência entre as tarefas.

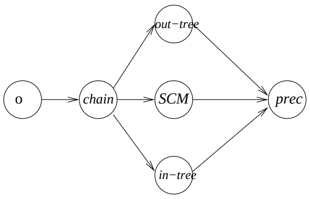


Figura 3.4: Relações de complexidade segundo as reações de precedência. O *o* indica tarefas independentes.

Quando não existem relações de precedência entre as tarefas, o problema não é mais difícil do que o problema onde as relações de precedência são cadeias. O primeiro caso, é um caso particular do segundo, no qual cada tarefa pertence a uma cadeia unitária. Da mesma forma, cadeias também são um caso particular de grafos *split-compute-merge* e de árvores. Finalmente, estes últimos são casos particulares de grafos de precedência em geral.

Existem também, como para as máquinas, parâmetros para o campo β que não estão relacionados entre si, como por exemplo r_j, s_{ij}, *prmp* e M_j. Não é possível dizer que um problema com o campo M_j não é mais difícil que um problema com o campo r_j. Por outro lado, algumas destas condições apenas complicam o problema, por exemplo o problema sem custo de comunicação entre as máquinas não é mais difícil que o problema com o parâmetro c_{ij}. Se existe um algoritmo que resolve o segundo tipo de problema, basta zerar os custos de comunicação para resolver os problemas do primeiro tipo.

Também existem relações de dificuldade entre os critérios de otimização. Como, pela primeira vez, nem todas as reduções serão simplesmente casos particulares, veremos em detalhes os arcos da figura 3.4. Na figura, os problemas de 1 a 8 correspondem aos seguintes critérios:

1	2	3	4	5	6	7	8
$C_{\max}$	$L_{\max}$	$\sum C_j$	$\sum U_j$	$\sum T_j$	$\sum w_j C_j$	$\sum w_j U_j$	$\sum w_j T_j$

- $1 \rightarrow 2$ No primeiro critério se minimiza o máximo de todos os C_j e no segundo o máximo de todos os $C_j - d_j$; logo o primeiro é um caso particular do segundo onde $d_j = 0$.

- $2 \to 4$ e $2 \to 5$ No primeiro queremos minimizar o máximo de $C_j - d_j$ e no segundo e terceiro queremos minimizar, respectivamente a soma de $\max\{0, C_j - d_j\}$ e a soma de $T_j = \{0,$ se $C_j - d_j \leq 0; 1$ caso contrário$\}$. Veremos a redução apenas para $\max\{0, C_j - d_j\}$, pois a outra é análoga. Se temos um algoritmo que resolve $\sum T_j$, este mesmo algoritmo também resolve o problema para $d_j = d_j + z$, onde z é um inteiro. Basta então aplicar o algoritmo para um valor de z grande e em seguida diminuir z de um em um até que $\sum w_j T_j$ seja diferente de zero. O z anterior fornece a resposta ao problema $C_j - d_j$.

- $3 \to 6$ Como para o caso $1 \to 2$, se um algoritmo resolve $\sum T_j$, basta fazer com que $d_j = 0$ para que o algoritmo resolva $\sum C_j$.

- $3 \to 6$, $5 \to 8$ e $4 \to 7$ Nestes casos podemos ver facilmente que os problemas são casos particulares com $w_j = 1$.

Nesta seção vimos que existe uma relação de dificuldade entre vários problemas de escalonamento. A relação que usamos, através de reduções, implica que um problema não é mais difícil do que outro. Nas próximas seções veremos a definição de redução assim como uma classificação da dificuldade de resolver problemas.

3.2 Complexidade

Muitos problemas de combinatória têm um número de casos exponencial, ou maior, de possibilidades a serem analisadas de forma a encontrar a solução desejada[1]. Às vezes é possível resolver um problema sem analisar todos os casos possíveis. Conforme a quantidade de casos que devem ser analisados, o problema pode ser classificado como fácil ou difícil.

A análise de complexidade permite determinar a classe de complexidade de um dado problema. Dizemos que um problema é fácil se existe um algoritmo que resolve o problema em um tempo limitado por um polinômio no tamanho do problema. Por outro lado o problema é difícil caso não possa ser resolvido em um tempo polinomial. Vejamos a importância desta classificação na seguinte tabela de 1979 (Garey&Johnson [1]).

complexidade	tamanho		
	10	30	60
n	10^{-5}s	3×10^{-5}s	6×10^{-5}s
n^5	.1s	24.3s	13min
2^n	10^{-3}s	17.9 min	366séc
3^n	0.059s	6.5 anos	1.3×10^{13} séc

Tabela 3.1: Tempo necessário para se resolver um problema conforme o seu tamanho em 1979.

Mesmo se em 1979 não fosse possível imaginar a enorme evolução da informática, a mesma tabela continua representativa. Exagerando um pouco, se hoje temos computadores 1 bilhão de vezes mais rápidos, isto é, o tempo de processamento de uma instrução é 1×10^{-14}, os tempos de resolução ficam:

complexidade	tamanho		
	10	30	60
n	10^{-14}s	3×10^{-14}s	6×10^{-14}s
n^5	10^{-10}s	2.4×10^{-8}s	7.8×10^{-7}s
2^n	10^{-12}s	1.8×10^{-8}s	19.23min
3^n	5.9×10^{-11}s	0.20s	1.3×10^{4} séc

Tabela 3.2: Tempo necessário para se resolver o problema em máquinas rápidas para os dias de hoje.

[1] No decorrer deste capítulo, denotaremos por solução um escalonamento válido e por solução ótima um escalonamento válido com o melhor valor possível da função objetivo.

complexidade	velocidade			
	x	$100x$	$1000x$	$1x10^9x$
n	N_1	$100N_1$	$1000N_1$	$1 \times 10^9 N_1$
n^5	N_2	$2.5N_2$	$3.98N_2$	$63N_2$
2^n	N_3	$N_3 + 6.64$	$N_3 + 9.97$	$N_3 + 29.9$
3^n	N_4	$N_4 + 4.19$	$N_4 + 6.29$	$N_4 + 18.9$

Tabela 3.3: Tamanho do problema solúvel em 1 hora.

Vemos que em 21 anos de evolução dos computadores, ainda não foi possível resolver todos os problemas exponenciais, os quais continuam impraticáveis para instâncias um pouco maiores. Também é interessante ressaltar que se o princípio de Moore continuar válido (os processadores dobram de velocidade a cada 18 meses), em 21 anos teremos computadores “apenas” dois milhões de vezes mais rápidos do que hoje.

Na tabela a seguir podemos encontrar um exemplo mais significativo. Dada a complexidade de um problema e o tamanho da instância que pode ser resolvida em uma hora em um computador com velocidade x, qual o tamanho da instância que pode ser resolvida com computadores mais rápidos, no mesmo tempo?

De uma forma geral o tamanho da instância solúvel por problemas polinomiais cresce multiplicando-se fatores ao tamanho inicial, enquanto que para problemas exponenciais o crescimento é logarítmico.

Em seguida veremos algumas notações para que as noções de redução e de complexidade de problemas possam ser formalizadas.

Denotaremos por π os problemas de escalonamento e por I suas instâncias. Por exemplo, um problema π é $P_2|p_j|C_{\max}$ e uma instância deste problema é um conjunto determinado de tarefas $\{t_1, \ldots, t_n\}$, cada tarefa t_j com o seu peso p_j. Dado um problema π, denotamos por D_π o conjunto de todas as suas instâncias. Dizemos que um algoritmo resolve um problema π se é capaz de encontrar a solução para cada instância $I \in D_\pi$.

Existem basicamente dois tipos de problema de combinatória; os de otimização e os de decisão. Em problemas de decisão a solução consiste de uma resposta: “sim” ou “não”. Nos problemas de otimização, procura-se o valor extremo de alguma função. Cada problema de otimização possui uma versão de decisão, que é responder se existe uma solução com um dado valor aceitável da função objetivo.

Exemplo 12 *Para o problema $P_2|p_j|C_{\max}$ um problema de decisão associado seria: Existe um escalonamento tal que $C_{\max} \leq \frac{\sum p_j}{2}$?*

Como um algoritmo que resolve um problema de otimização π também resolve a sua versão de decisão, este não é computacionalmente mais difícil que o problema original π. Quando um problema de decisão é difícil, o problema de otimização associado também é difícil. Logo, para o desenvolvimento da teoria de complexidade podemos utilizar apenas os problemas de decisão.

A complexidade de um problema depende da relação entre o tempo necessário para resolvê-lo e o tamanho da instância I. Como a teoria de complexidade independe do problema e do computador utilizado, usa-se em geral, modelos de computação como a máquina de Turing e a RAM (Random Access Machine). Em uma RAM, o tempo de um algoritmo pode ser medido através da soma dos custos de operações básicas necessárias à execução deste algoritmo. O custo de cada operação básica é proporcional ao espaço necessário para se armazenar os operandos. É interessante ressaltar que no caso de análise de complexidade, as máquinas de Turing e RAM são equivalentes.

O tamanho da instância I depende da codificação utilizada (por exemplo, nos computadores atuais, os números inteiros são armazenados na forma binária). Mas qualquer codificação não unária, não redundante e passível de decodificação é valida. O tamanho de uma instância I é definido como $N(I)$, o tamanho da string necessária à codificação. É interessante dizer que mesmo se o tamanho de uma instância depende, por exemplo, do número de objetos e de seus tamanhos, na maioria dos casos o número de objetos pode ser usado como tamanho da instância.

A medida do tempo de execução de um algoritmo A, para um problema de tamanho n, é o número máximo de operações necessárias em uma RAM, para qualquer instância $I \in D_\pi$, tal que $n = N(I)$. Esta medida é denominada complexidade do algoritmo A.

Um algoritmo A tem complexidade $O(f(n))$ quando a sua função de complexidade $g_A(n)$ satisfaz: $g_A(N(I)) \leq Cf(N(I))$, onde C é uma constante positiva. Quando o número de passos de $g_A(n)$ for limitado por um polinômio, dizemos que o algoritmo é polinomial. Vejamos agora a definição das duas principais classes de complexidade $\mathcal{P}$ e $\mathcal{NP}$.

Definição 4 *A classe $\mathcal{P}$ contém todos os problemas de decisão solúveis em tempo polinomial, isto é, para os quais existe um algoritmo com tempo polinomial em uma RAM.*

Existe uma classe ainda maior de problemas, para os quais uma RAM consegue verificar, em tempo polinomial, se uma solução de um problema de decisão está correta ou não. Esta solução pode ser vista como uma dica ou uma consulta a um oráculo. Por exemplo, no caso de problemas de escalonamento a dica pode ser um escalonamento.

Definição 5 *A classe $\mathcal{NP}$ contém todos os problemas de decisão, para os quais dada uma dica apropriada, podem ser verificados em uma RAM em tempo polinomial.*

Pode-se concluir facilmente que $\mathcal{P} \subset \mathcal{NP}$. Mas uma das questões mais importantes na computação hoje é saber se $\mathcal{P} = \mathcal{NP}$ ou se $\mathcal{P} \neq \mathcal{NP}$. Caso sejam iguais então existirão algoritmos polinomiais para um grande número de problemas para os quais ainda não se conhecem algoritmos polinomiais. Mas a conjectura mais provável é que $\mathcal{P} \neq \mathcal{NP}$ e neste caso as classe de complexidade podem ser representadas pela figura 3.5. Veremos em seguida as classe $\mathcal{NP}c$ e $s\mathcal{NP}c$.

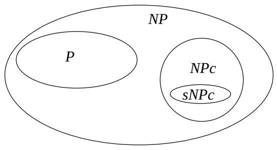


Figura 3.5: Relações entre as classes de complexidade (supondo $\mathcal{P} \neq \mathcal{NP}$).

Um outro conceito interessante em teoria de complexidade é a redução polinomial. Com esta redução podemos classificar problemas como não mais difíceis que outros.

Definição 6 *Uma redução de um problema π_2 para um problema π_1 ($\pi_2 \propto \pi_1$) é uma função $f : D_{\pi_2} \rightarrow D_{\pi_1}$ tal que:*

1. *$\forall I_2 \in D_{\pi_2}$ a resposta é sim se e só se a resposta é sim para $I_1 = f(I_2)$.*
2. *$\forall I_2 \in D_{\pi_2}$ a função f pode ser calculada em tempo polinomial em $N(I_2)$.*

Um exemplo, visto no começo do capítulo foi:

Exemplo 13 $L_{\max} \rightarrow \sum U_j$. *Os problemas de decisão associados são:*

1. *Existe um escalonamento para o problema $\alpha|\beta|L_{\max}$ tal que $L_{\max} \leq x$?*

2. Existe um escalonamento para o problema $\alpha|\beta|\sum U_j$ tal que $\sum U_j \leq y$?

Caso saibamos resolver o segundo problema, o mesmo algoritmo pode ser utilizado para o primeiro. Basta alterar d_j para $d_j + x$ e verificar se existe uma solução com $y = 0$.

Se existe uma redução de π_2 para π_1 e $\pi_1 \in \mathcal{P}$, então claramente $\pi_2 \in \mathcal{P}$. Mas a operação de redução também serve para criar uma classe de equivalência em $\mathcal{NP}$.

Definição 7 *Um problema de decisão π_1 é $\mathcal{NP}-$completo se $\pi_1 \in \mathcal{NP}$ e $\forall \pi_2 \in \mathcal{NP}, \pi_2 \propto \pi_1$.*

Logo, se existir um algoritmo polinomial para um problema $\mathcal{NP}-$completo, todos os problemas $\mathcal{NP}-$completos serão solucionáveis em tempo polinomial. O primeiro problema $\mathcal{NP}-$completo da literatura foi o problema da satisfatibilidade (dados um conjunto de variáveis e uma coleção de cláusulas, existe uma atribuição de valores às variáveis tal que todas as cláusulas sejam verdadeiras?). Hoje são conhecidos muitos problemas $\mathcal{NP}-$completos, vários deles são problemas de escalonamento. Os problemas $\mathcal{NP}-$completos mais importantes do ponto de vista do escalonamento são:

Partição Dados n inteiros positivos $s_1, \ldots, s_n$ existe um subconjunto $J \subseteq I = \{1, \ldots, n\}$ tais que

$$\sum_{i \in J} s_i = \sum_{i \in I \setminus J} s_i?$$

3-Partição Dados $3n$ inteiros positivos $a_1, \ldots, a_{3n}$ e um inteiro b tal que

$$\frac{b}{4} < a_j < \frac{b}{2}, j = 1, \ldots, 3n \text{ e } \sum_{j=1}^{3n} a_j = nb$$

existem n conjuntos disjuntos de três elementos tais que

$$\sum_{j \in S_i} a_j = b, \text{ para } i = 1, \ldots, n?$$

Circuito hamiltoniano Dado um grafo $G(V, E)$, decidir se G contém um circuito hamiltoniano (um circuito hamiltoniano é uma ordem $(v_1, \ldots, v_n)$ dos vértices de G, onde $|V| = n$, tal que $(v_i, v_{i+1}) \in E$ para $i = 1, \ldots, n-1$ e $(v_n, v_1) \in E$.

TSP Dado um grafo $G(V, E)$ onde entre os vértices v_i e v_j existe uma aresta com valor w_{ij} e um inteiro B, decidir se G contém um circuito onde cada vértice é visitado exatamente uma vez, cuja soma dos pesos das arestas seja $\leq B$.

3.2.1 Exemplos de redução

Partição $\propto P_2|p_j|C_{\max}$

O primeiro passo para mostrar que $P_2|p_j|C_{\max}$ pertence a classe $\mathcal{NP}-$*completo* é mostrar que o problema pertence a $\mathcal{NP}$. Dada um escalonamento ótimo é fácil de verificar em tempo polinomial que o makespan está abaixo, ou é igual, a um dado valor.

A redução de uma instância qualquer do problema da partição também é fácil. Dada uma instância I com n inteiros positivos $s_1, \ldots, s_n$, podemos transformar esta na seguinte instância I' de $P_2|p_j|C_{\max}$: Sejam $m = n$ tarefas com pesos $p_j = s_j$. É fácil ver que existe um escalonamento de I' com makespan igual a $\sum p_j/2$ se e só se existe uma partição do conjunto $s_1, \ldots, s_n$ de I.

3.2.2 3-Partição $\propto J_2|recrc, prmp, p_j|C_{\max}$

$J_2|recrc, prmp, p_j|C_{\max}$ pertence a $\mathcal{NP}$, pois podemos verificar em tempo polinomial se um escalonamento dado tem makespan menor ou igual a um certo valor.

Dada uma instância I do problema da 3-partição com $3n$ inteiros positivos $a_1, \ldots, a_{3n}$ e um inteiro b, vamos transformá-la na seguinte instância I' de $J_2|recrc, prmp, p_j|C_{\max}$. Seja $m = 3n + 1$ o número de tarefas, onde cada tarefa t_j é definida por:

	t_1	t_2	$\ldots$	t_j	$\ldots$	t_{m-1}
M_1	a_1	a_2	$\ldots$	a_j	$\ldots$	a_{3n}
M_2	a_1	a_2	$\ldots$	a_j	$\ldots$	a_{3n}

Isto é, cada uma das $3n$ primeiras tarefas, $t_j, 1 \leq j \leq 3n$, deve ser executada primeiro na máquina M_1 e em seguida na máquina M_2 sendo o tempo de execução em cada máquina a_j. Finalmente a tarefa t_m começa a ser processada pela máquina M_2 e tem que alternar o processamento entre as duas máquinas. O tempo total da tarefa é $2nb$ e cada um dos tempos de processamento é b.

Se existe um escalonamento com makespan $2nb$, a tarefa t_m é executada durante todo o escalonamento. Além da tarefa t_m, cada máquina deve executar cada uma das $3n$ primeiras tarefas. Como o tempo livre em cada máquina é b e a soma destas tarefas também é b, o escalonamento não terá tempo de inatividade. Como para cada tarefa o tempo de processamento p_j está no intervalo $]\frac{b}{4}, \frac{b}{2}[$ existem exatamente três tarefas alocadas em cada intervalo de tamanho b.

Logo existe um escalonamento com makespan $2nb$ se e só se existe uma 3-Partição para o conjunto I.

3.2.3 Algoritmos não tão difíceis

Mesmo se não conhecemos nenhum algoritmo polinomial para os problemas da classe $\mathcal{NP}-$completo, existem, para alguns deles, algoritmos onde a complexidade é limitada por um polinômio em $N(I)$ e $Max(I)$, o maior valor numérico que ocorre na instância I. Mesmo se estes problemas não são polinomiais, pois $Max(I)$ nem sempre é menor que um polinômio em $N(I)$ para todo $I \in D_\pi$, podem existir algoritmos pseudo-polinomiais para resolver estes problemas.

Vamos definir a seguir os problemas que são muito difíceis, isto é, a não ser que o tamanho da instância seja pequeno, o tempo necessário para que um algoritmo encontre a solução exata pode ser impraticável.

Os problemas fortemente $\mathcal{NP}-$completos são aqueles para os quais não existem algoritmos pseudo-polinomiais que resolvem tais problemas, a não ser que $\mathcal{P} = \mathcal{NP}$.

Definição 8 *Seja π_p, para o problema π e o polinômio p, o sub-problema de π obtido restringindo D_π às instâncias onde $Max(I) \leq p(N(I))$. O problema π é fortemente $\mathcal{NP}-$completo se $\pi \in \mathcal{NP}$ e $\pi_p \in \mathcal{NP}-$completo.*

As complexidades das versões de otimização dos problemas de decisão são:

$\mathcal{NP}-$completo ($\mathcal{NP}c$	$\mathcal{NP}-$difícil
fortemente $\mathcal{NP}-$completo	fortemente $\mathcal{NP}-$difícil

3.2.4 Exemplos de problemas solúveis

Quando um problema pertence a $\mathcal{P}$ (isto é a versão de decisão do problema) então é possível encontrar uma solução em tempo polinomial, o que geralmente significa em tempo aceitável. Caso o problema seja pseudo-polinomial, pode-se tentar resolver o problema por técnicas como a programação dinâmica, na qual a solução é construída a partir de soluções de sub-problemas do mesmo. Finalmente, quando o problema é fortemente $\mathcal{NP}-$difícil, ou quando não se consegue propor um algoritmo pseudo-polinomial, temos duas possibilidades: o uso de heurísticas ou o uso de técnicas "inteligentes" de enumeração.

Nesta seção veremos alguns problemas para os quais é possível encontrar a solução exata em tempo razoável.

Um algoritmo polinomial

Exemplo 14 *Resolver* $1|p_j|\sum w_j C_j$.

Para a maior parte dos problemas de escalonamento solúveis em tempo polinomial, os algoritmos utilizados são do tipo guloso. Mais especificamente, algoritmos de lista, onde as tarefas são ordenadas segundo alguma relação de ordem, para em seguida serem alocadas às máquinas.

Cada tarefa (t_j) do exemplo 14 acima possui duas características: o tempo de execução (p_j) e o peso de penalização (w_j). Observando-se que caso só existisse uma das características, seria interessante ordenar as tarefas em ordem crescente de tempo ou em ordem decrescente de pesos. Um critério que considera tanto p_j quanto w_j deve ser proporcional a w_j e inversamente proporcional a p_j. Não é difícil de deduzir que devemos ordenar as tarefas em ordem crescente de w_j/p_j.

Algoritmos pseudo-polinomiais

Para resolver os dois próximos exemplos usaremos técnicas de programação dinâmica, onde a solução é construída por etapas, a partir de soluções para sub-problemas.

Exemplo 15 *Resolver* $P_2|p_j|C_{\max}$.

Para resolver este problema, temos que encontrar um subconjunto do conjunto das tarefas tal que sua soma seja $LB = \sum p_j/2$; caso não exista este conjunto, devemos encontrar um subconjunto com soma $LB+1$ (ou simetricamente $LB-1$), e assim por diante.

Dada uma instância $t_1, \ldots, t_m$ com respectivos tempos de execução $p_1, \ldots, p_m$, uma forma de se resolver este problema é de verificar quais as somas possíveis com j tarefas, j variando de 1 a m.

É claro que com apenas uma tarefa, por exemplo t_1, as somas possíveis são: $\{0, p_1\}$, com duas tarefas, t_1, t_2 as somas possíveis são: $\{0, p_1, p_2, p_1 + p_2\}$. O número total de somas possíveis é no final 2^m (já sabíamos disto pois a versão de decisão problema pertence a classe $\mathcal{NP}-$completo).

Mas, se $f(j,c)$ denota a possibilidade de se obter com as j primeiras tarefas o tempo de processamento c, isto é, se é possível se obter o valor c com as j primeiras tarefas $f(j,c) = 1$, $f(j,c) = 0$ caso contrário. Existem duas possibilidades para que $f(j,c)$ seja igual a um, ou é possível encontrar o valor c com as $j-1$ primeiras tarefas, ou é possível encontrar o valor $c - p_j$ com as $j-1$ primeiras tarefas. Logo, podemos escrever a seguinte relação de recorrência:

$$f(j,c) = \max\{f(j-1,c); f(j-1,c-p_j)\}.$$

Com as seguintes condições de contorno:

$f(0,x) = 0; 0 \leq x \leq d;$
$f(j,0) = 0; 1 \leq j \leq m$ (Com zero objetos pode se conseguir soma zero).
$f(j,x) = 0; x < 0$ (Não dá para se conseguir somas negativas.)

Mesmo que não exista solução cuja soma seja LB, basta encontrar o maior valor de $x \leq LB$ tal que $f(m,x) \neq 0$.

Após ter encontrado o menor valor de $C_{\max}$ a partição correspondente pode ser encontrada por backtracking. As tarefas que devem ser alocadas ao outro processador correspondem ao conjunto complementar.

Exercício 4 *Resolver o problema com* $p_j = \{7, 8, 2, 4, 1\}$.

Inicialmente, somando as tarefas vemos que $LB = 11$*. Logo estamos interessados nos valores* $f(m,x)$ *para* x *variando de 0 a 11. Na tabela seguinte, as linhas correspondem ao número de tarefas e as colunas ao valor possível:*

$m \setminus x$	0	1	2	3	4	5	6	7	8	9	10	11
0	**1**	0	0	0	0	0	0	0	0	0	0	0
1 ($p_1 = 7$)	**1**	0	0	0	0	0	0	1	0	0	0	0
2 ($p_2 = 8$)	1	0	0	0	0	0	0	1	**1**	0	0	0
3 ($p_3 = 2$)	1	0	1	0	0	0	0	1	1	1	**1**	0
4 ($p_4 = 4$)	1	0	1	0	1	0	1	1	1	1	**1**	1
5 ($p_5 = 1$)	1	1	1	1	1	1	1	1	1	1	1	**1**

A primeira linha da tabela contém apenas um valor um, pois não é possível ter nenhuma soma parcial com zero tarefas, a não ser a soma zero. A segunda linha é construída a partir da primeira, usando-se a equação de recorrência, no caso: $f(1,c) = \max\{f(0,c), f(0,c-7)\}$*. Como o único valor não nulo da linha anterior é o* $f(0,0)$*, apenas os valores* $f(1,0) = f(0,0)$ *e* $f(1,7) = f(0,c-7)$ *serão iguais a um. O mesmo raciocínio é válido para as linhas seguintes. Como o valor de* $f(5,11)$ *é um, existe uma partição do conjunto de tarefas do exemplo, com a mesma soma.*

Sabendo-se que existe um partição, a mesma pode ser construída por backtracking. No caso o valor um de $f(5,11)$ *pode ter vindo de* $f(4,11)$ *e/ou de* $f(4,11-1)$*: caso seja o primeiro, a tarefa* t_j *não faz parte de um dos conjuntos; caso seja o segundo, a tarefa pertence à partição a ser construída. Escolhendo-se um dos dois (pois caso os dois sejam possíveis a escolha é indiferente), por exemplo* $f(4,10)$*, a tarefa* t_5 *faz parte da partição. A tarefa* t_4 *não faz parte deste conjunto pois* $f(3,6) = 0$*. Logo, continuamos com* $f(3,10)$*. Como* $f(2,10) = 0$*, a tarefa* t_3 *faz parte do conjunto, e continuamos com* $f(3,8)$*, onde finalmente descobrimos que a tarefa* t_2 *faz parte do conjunto. Chegando-se a* $f(2,0)$*, sabemos que um subconjunto é formado por* t_2, t_3 *e* t_5 *e o outro subconjunto é o complementar. O caminho escolhido está em negrito na tabela.*

Exemplo 16 *Resolver* $1|p_j; d_j = d| \sum w_j U_j$.

Este problema é equivalente ao problema da mochila, no qual devemos colocar em uma mochila objetos que não ultrapassem a capacidade total da mochila, maximizando o valor total. Formalmente: dada uma mochila de capacidade c e m objetos com tamanhos $p_1, \ldots, p_m$ e valores respectivos $w_1, \ldots, w_m$, encontrar um subconjunto dos objetos com tamanho total no máximo c, com o maior valor possível. Neste problema, cada objeto só pode ser colocado na mochila uma vez.

Para enxergar esta equivalência basta observar que para o problema $1|p_j; d_j = d| \sum w_j U_j$ devemos escolher quais as tarefas devem ser completadas antes do instante d, minimizando o peso das tarefas que serão executadas depois. No problema da mochila maximiza-se o tamanho, ou peso, das tarefas a serem completadas antes do instante d.

Dada uma instância qualquer do problema, com um valor para d e

t_1	t_2	$\dots$	t_j	$\dots$	t_m
p_1	p_2	$\dots$	p_j	$\dots$	p_m
w_1	w_2	$\dots$	w_j	$\dots$	w_m

Se denotarmos os valores dos sub-problemas por $\pi(j,x)$, onde j é a última tarefa e x a data devida, a resolução do problema está baseada na seguinte equação:

$$\pi(j,x) = \min\{\pi(j-1,x) + w_j, \pi(j-1,x-p_j),$$

que significa que o valor do problema com j tarefas e data limite x corresponde ao valor mínimo entre o valor do problema, sem escalonar a tarefa t_j antes da data x, e o valor do problema escalonando a tarefa t_j. Não é difícil de ver que as condições de contorno são:

$$\begin{array}{l} \pi(0,x) = 0; 0 \leq x \leq d \text{ (sem tarefas, nenhuma acaba após o prazo);} \\ \pi(j,0) = \sum_{i=1}^{j} w_j; 0 \leq j \leq m \text{ (todas as tarefas acabam após o prazo);} \\ \pi(j,y) = +\infty; y < 0 \text{ (datas negativas não existem).} \end{array}$$

Exercício 5 *Resolução do problema com 6 tarefas e $d = 14$:*

t_1	t_2	t_3	t_4	t_5	t_6
3	*3*	*3*	*4*	*9*	*5*
3	*4*	*3*	*4*	*10*	*5*

Como no exemplo anterior, vamos resolver o problema, linha a linha, a cada vez adicionando um objeto.

$j \setminus x$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	3	3	0	0	0	0	0	0	0	0	0	0	0	0
2	7	7	7	3	3	3	0	0	0	0	0	0	0	0	0
3	10	10	10	6	6	6	3	3	3	0	0	0	0	0	0
4	14	14	14	10	10	10	7	6	6	4	3	3	3	0	0
5	24	24	24	20	20	20	17	16	16	14	13	13	13	10	10
6	29	29	29	25	25	24	22	21	20	19	18	17	16	16	14

Logo, o valor mínimo de $\sum w_j U_j$ é 14. Para saber quais tarefas estão alocadas antes de d basta fazer o backtracking a partir de $\pi(6,14)$.

$$\begin{array}{l} \pi(6,14) = \pi(5,14-p_6) = \pi(5,9) \textit{ (a tarefa } t_6 \textit{ está alocada antes.)} \\ \pi(5,9) = \pi(4,9) - w_5 \textit{ (a tarefa } t_5 \textit{ é executada após d.)} \\ \pi(4,9) = \pi(3,9) - w_4 \textit{ (a tarefa } t_4 \textit{ é executada após d.)} \\ \pi(3,9) = \pi(2,9-p_3) = \pi(2,6) \textit{ (a tarefa } t_3 \textit{ está alocada antes.)} \\ \pi(2,6) = \pi(1,6-p_2) = \pi(1,3) \textit{ (a tarefa } t_2 \textit{ está alocada antes.)} \\ \pi(1,3) = \pi(0,3-p_1) = \pi(0,0) \textit{ (a tarefa } t_1 \textit{ está alocada antes.)} \end{array}$$

Logo, executando-se as tarefas t_1, t_2, t_3 e t_6 até o instante 14, a penalização será paga apenas para as tarefas t_4 e t_5. Na tabela acima, também podemos ver as soluções para outros valores de d menores que 14.

Algoritmos para problemas fortemente $\mathcal{NP}-$difíceis

Com relação aos problemas fortemente $\mathcal{NP}-$difíceis temos várias possibilidades: utilizar algoritmos simples como os de lista, usar heurísticas mais sofisticadas ou enumerar as soluções possíveis de uma forma conveniente. A vantagem das duas primeiras é de serem rápidas, apesar de nem sempre encontrarem a solução ótima. Nesta seção, por um abuso de notação, suporemos que são soluções para o problema todos os escalonamentos válidos, sendo o melhor possível denotado por solução ótima. A enumeração, mesmo que usando-se o melhor algoritmo possível, nem sempre terá tempo de execução praticável.

Outro ponto interessante é de se garantir algum critério com relação à solução encontrada. Se um algoritmo é capaz, para um problema π, de encontrar uma solução que seja no máximo k vezes pior do que a solução ótima, dizemos que esta heurística é um algoritmo de aproximação com garantia de desempenho k. É interessante ressaltar que para alguns problemas, a existência de um algoritmo de aproximação polinomial, implica que $\mathcal{P} = \mathcal{NP}$.

Além dos algoritmos de lista que podem ser usados com algoritmos de aproximação, temos também as heurísticas que são variações de algoritmos de busca local.

Considere o problema de minimizar uma função objetivo f sobre um conjunto S de soluções possíveis ou válidas. Uma forma natural de se resolver o problema é partindo de uma solução válida $x \in S$, procurar soluções melhores vizinhas a x. Denota-se a vizinhança de x por $N(x)$.

1. Escolha uma solução inicial $x \in S$;
2. Repita
3.   Procure a melhor solução $x' \in N(x)$;
4.   Se $f(x') < f(x)$ então
5. $x = x'$;
6. até que $f(x') \geq f(x)$

Antes de continuarmos, temos que conhecer o significado de vizinhança para os problemas de escalonamento. Não existe nenhuma definição, pois o conceito de vizinhança, além de ser dependente do contexto do problema, pode ser também visto de formas diferentes por diferentes pessoas. Uma definição que pode ser utilizada para a maioria do problemas leva em conta escalonamentos parecidos:

Definição 9 *Dois escalonamentos S_1 e S_2 de uma mesma instância I são vizinhos se e só se:*

- *De todas as tarefas, apenas uma é executada em máquinas diferentes nos escalonamentos S_1 e S_2, para todas as outras tarefas, exceto esta, a ordem de execução nas máquinas é a mesma; ou*
- *Todas as tarefas de S_1 são executadas nas mesmas máquinas no escalonamento S_2, mas a ordem de execução em uma das máquinas é diferente para apenas uma das tarefas.*

Mas, infelizmente, não basta a cada etapa encontrar $y \in N(x)$ que minimize a função f, pois pode-se ficar preso a mínimos locais. De forma a evitar tais situações, meta-heurísticas, as quais são capazes de escapar dos mínimos locais, são utilizadas.

Algumas das principais meta-heurísticas são:

Busca Tabu Funciona basicamente como uma busca local. A melhor solução $y \in N(x)$ é escolhida como a próxima solução. Entretanto, quando se encontra um mínimo local, isto é, uma solução x tal que $f(x) \leq f(y), y \in N(x)$, para se evitar que a heurística não procure mais soluções, uma lista Tabu é usada. Esta lista Tabu é uma fila, que não permite que a próxima solução seja uma das soluções já vistas.

1. Escolha uma solução inicial $x \in S$;

2. melhor=$f(x)$;
3. $x^* = x$;
4. lista-tabu=$\emptyset$;
5. Repita
6. Possib$(x) = \{x' \in N(x) | x'$ não é tabu OU $f(x') < f(x)\}$;
7. Escolha uma solução $\bar{x} \in$ Possib(x);
8. $x = \bar{x}$;
9. atualize a lista-tabu;
10. Se $f(x)$ <melhor então
11. $x^* = x$;
12. melhor=$f(x)$;
13. Até o critério de parada

Pode-se aprimorar este método com uma lista de boas soluções iniciais, que será usada a cada vez que o método não encontrar soluções melhores, depois de um certo período de tempo.

Simulated annealing Este método é baseado na equação que modela o resfriamento lento de um sólido (Física). Uma solução y vizinha de uma solução x, $y \in N(x)$, é aceita como nova solução se reduz o valor da função objetivo f. Por outro lado, se este mesmo vizinho aumenta o valor da função objetivo em $\Delta f > 0$, esta nova solução é aceita com probabilidade $e^{\frac{-\Delta f}{c_k}}$, onde c_k é um parâmetro de controle que corresponde à temperatura.

O método consiste em diminuir gradativamente o valor de c_k, isto é, a probabilidade de se aceitar soluções piores diminui quando o valor de c_k diminui. Logo, escapar de mínimos locais é fácil no começo do algoritmo. No algoritmo abaixo, a função rand retorna um número aleatório entre 0 e 1.

1. $k = 0$;
2. Escolha uma solução inicial $x \in S$;
3. melhor=$f(x)$;
4. $x^* = x$;
5. Repita
6. Escolha aleatoriamente uma solução $x' \in N(x)$
7. Se (rand< $e^{\frac{-\Delta f}{c_k}}$) então
8. $x = x'$
9. Se $f(x)$ <melhor então
10. $x^* = x$;
11. melhor=$f(x)$;
12. $c_{k+1} = g(c_k)$;
13. k++;
14. Até o critério de parada

Busca genética O algoritmo de busca genética tenta imitar a natureza da evolução. Cada solução possível para o problema pode ser vista como uma string, representando um gene. Boas soluções do conjunto de soluções iniciais podem ser combinadas para obter um novo conjunto de soluções. A medida de qualidade de um conjunto de soluções pode ser dada, por exemplo, pela função objetivo. Existem três formas clássicas de obter novas soluções:

- Reprodução: Copiar soluções para o conjunto das novas soluções;
- Combinação: Combinar duas soluções em uma nova solução;
- Mutação: Modificar aleatoriamente soluções dentro do conjunto de soluções.

Os novos conjuntos de soluções são obtidos iterativamente dos conjuntos anteriores, até que um conjunto, ou indivíduo, satisfaça algum critério de parada.

Branch and Bound

O principal método de enumeração é o Branch& Bound (B& B). Este método é semelhante à Divisão e Conquista, onde segundo um critério dado, algumas das conquistas não precisam ser executadas, sem prejuízo ao resultado final.

O método pode ser descrito da seguinte forma. Inicialmente o espaço de soluções S é dividido em subconjuntos disjuntos $S_1, \ldots, S_k$. Estes são subdivididos em subconjuntos até que o ponto em que os subconjuntos sejam unitários. O espaço inicial de soluções S pode ser visto como a raiz de uma árvore, onde os filhos de um nó correspondem a subdivisões do conjunto correspondente ao nó. A idéia básica do Branch&Bound é de eliminar os subconjuntos que não têm a possibilidade de conter soluções menores o quanto antes.

Um nó da árvore pode deixar de ser analisado nas seguintes condições:

1. O subconjunto S_i não contém nenhuma solução melhor do que a melhor solução conhecida até o momento.
2. O subconjunto S_i não contém soluções válidas.
3. Todas as soluções contidas em S_i não são melhores que as soluções contidas em outro subconjunto S_j. Isto é, a melhor solução em S_i é no máximo tão boa quanto a melhor solução em S_j.

O mecanismo básico utilizado em B&B consiste em conhecer alguma estimativa ou limite na função objetivo das soluções em S_i, de forma a compará-lo com estimativas de outros conjuntos ou de alguma solução. A solução inicial pode ser obtida por qualquer uma das heurísticas anteriores. Logo, um método de B&B consiste de:

Branching: Escolha do método de divisão e busca nos subconjuntos;

Bounding: Alguma forma de estimar a função objetivo das soluções de um subconjunto, de forma a eliminar os nós na árvore de busca.

O algoritmo B&B é dado abaixo:

1. Lista=$\{S\}$;
2. Best= valor de uma solução válida x;
3. enquanto (Lista$\neq$ 0)
4. Escolha e remova um nó da v Lista;
5. Gere o conjunto de filhos de v: $f_1, \ldots, f_{n_v}$;
6. Calcule o limite inferior para cada f_i, LB_i;
7. Para cada f_i
8. Se LB_i <Best então
9. se f_i contém apenas uma solução então
10. Best=LB_i;
11. $x = f_i$;
12. senão
13. adicione f_i a lista Lista

Capítulo 4

Escalonamento em uma máquina

Veremos neste capítulo vários problemas relacionados ao escalonamento em uma máquina. Estes problemas são importantes, pois além de fornecerem idéias para problemas de escalonamento em um número maior de máquinas, correspondem a casos particulares de problemas mais complicados.

Mesmo que características como a minimização do makespan ($C_{\max}$) ou a possibilidade de interrupção possam parecer a princípio irrelevantes para o escalonamento em uma máquina, os seguintes exemplos demonstram o contrário:

- $1|p_j = 1; s_{jk}|C_{\max}$[1]. Neste caso, como todas as tarefas tem a mesma duração, o makespan é dado pela soma do número de tarefas com o tempo de preparação da máquina. Logo para a minimização do makespan deve-se encontrar a ordem de tarefas cuja soma de custos de preparação seja mínima.

- $1|prmp; p_j; r_j; d_j|L_{\max}$. Este problema é bem diferente do problema $1|p_j; r_j; d_j|L_{\max}$ onde as interrupções não são permitidas. Por exemplo, para o primeiro problema caso, os escalonamentos ótimos são necessariamente sem atraso (ver definição 3). O que não é verdade para o problema onde a interrupção é permitida. Verifique isto no seguinte exemplo, com duas tarefas:

	t_1	t_2
p_j	3	2
r_j	0	1
d_j	5	3

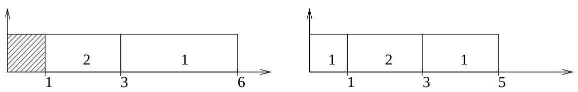


Figura 4.1: Escalonamentos ótimos, sem e com interrupção, com valores de $L_{\max}$ respectivamente 1 e 0.

4.1 Tempo total de finalização ponderado ($\sum w_j C_j$)

Nós já vimos no capítulo anterior que o problema $1|p_j|\sum w_j C_j$ é polinomial, veremos agora como provar que a regra WSPT (weighted shortest procesing time first - ordem decrescente de w_j/p_j) fornece escalonamentos ótimos. Esta regra também é conhecida como regra de Smith.

[1] Mais detalhes podem ser vistos no capítulo de exemplos

Teorema 1 *Para $1|p_j|\sum w_j C_j$ a regra de Smith é ótima.*

Prova: Vamos provar por contradição. Suponha que para uma dada instância I do problema o escalonamento WSPT não é ótimo. Logo, no escalonamento ótimo S, devem existir pelo menos duas tarefas consecutivas, t_1 e t_2 tais que

$$\frac{w_{t_1}}{p_{t_1}} < \frac{w_{t_2}}{p_{t_2}}.$$

Se chamarmos de t o tempo em que a tarefa t_1 está escalonada, o tempo deste escalonamento ótimo S é igual a:

$$T + (t + p_{t_1} w_{t_1} + (t + p_{t_1} + p_{t_2}) w_{t_2},$$

onde T é o tempo ponderado de todas as outras tarefas.

Seja S' o escalonamento obtido invertendo-se a ordem das tarefas t_1 e t_2. Como mais nenhuma tarefa tem o seu tempo de término (C_j) alterado o valor de $\sum w_j C_j$ para S' é:

$$T + (t + p_{t_2}) w_{t_2} + (t + p_{t_2} + p_{t_1}) w_{t_1}.$$

Como o escalonamento S é ótimo, temos que:

$$\begin{aligned}
&T + (t + p_{t_1} w_{t_1} + (t + p_{t_1} + p_{t_2}) w_{t_2} \leq T + (t + p_{t_2}) w_{t_2} + (t + p_{t_2} + p_{t_1}) w_{t_1},\\
&p_{t_1} w_{t_1} + p_{t_1} w_{t_2} + p_{t_2} w_{t_2} \leq p_{t_2} w_{t_2} + p_{t_2} w_{t_1} + p_{t_1} w_{t_1},\\
&p_{t_1} w_{t_2} \leq p_{t_2} w_{t_1}.
\end{aligned}$$

O leva a uma contradição.□

Vejamos agora um caso mais geral, onde existem dependências do tipo cadeia entre as tarefas. Inicialmente, vamos supor que para o problema $1|p_j; chains|\sum w_j C_j$, uma vez que uma cadeia começa a ser executada, a mesma não pode ser interrompida. Veremos agora, dadas duas cadeias de tarefas, qual das duas deve ser executada primeiro, de forma a minimizar $\sum w_j C_j$.

Lema 1 *Sejam $t_1, \ldots, t_k$ e $t_{k+1}, \ldots, t_n$, duas cadeias de tarefas, se*

$$\frac{\sum_{i=1}^{k} w_{t_i}}{\sum_{i=1}^{k} p_{t_i}} \geq \frac{\sum_{i=k+1}^{n} w_{t_i}}{\sum_{i=k+1}^{n} p_{t_i}}$$

a cadeia com as tarefas $t_1, \ldots, t_k$ deve preceder a cadeia $t_{k+1}, \ldots, t_n$.

Prova: Vamos provar por contradição. O tempo total ponderado de um escalonamento onde a cadeia $t_1, \ldots, t_k$ está escalonada antes é:

$$w_{t_1} p_{t_1} + \ldots + w_{t_k} \sum_{i=1}^{k} p_{t_i} + w_{t_k} \sum_{i=1}^{k+1} p_{t_i} + \ldots + w_{t_n} \sum_{i=1}^{n} p_{t_i}.$$

Se a cadeia $t_{k+1}, \ldots, t_n$ está escalonada antes o tempo total ponderado é:

$$w_{t_k} p_{t_k} + \ldots + w_{t_n} \sum_{i=k+1}^{n} p_{t_i} + w_{t_1} \left(p_{t_1} + \sum_{i=1}^{k+1} p_{t_i} \right) + \ldots + w_k \sum_{i=1}^{n} p_{t_i}.$$

Caso o segundo tempo seja menor que o primeiro temos:

$$\begin{aligned}
&\textstyle\sum_{i=k+1}^{n} p_{t_i} w_{t_1} + \ldots + \sum_{i=k+1}^{n} p_{t_i} w_{t_k} - \sum_{i=1}^{k} p_{t_i} w_{t_{k+1}} + \ldots + \sum_{i=1}^{k} p_{t_i} w_{t_n} > 0,\\
&\textstyle\sum_{i=k+1}^{n} p_{t_i} \sum_{i=1}^{k} w_{t_i} > \sum_{i=1}^{k} p_{t_i} \sum_{i=k+1}^{n} w_{t_i}.
\end{aligned}$$

o que contradiz a hipótese.□

No caso em que a execução de uma cadeia pode ser interrompida, devemos decidir se, e quando, fazê-lo. Para isto introduzimos um novo conceito:

Definição 10 *Dada uma cadeia de tarefas, $t_1, \ldots, t_k$, e l^* satisfazendo:*

$$\frac{\sum_{i=1}^{l^*} w_{t_i}}{\sum_{i=1}^{l^*} p_{t_i}} = \max_{1 \leq l \leq k} \left(\frac{\sum_{i=1}^{l} w_{t_i}}{\sum_{i=1}^{l} p_{t_i}} \right),$$

defini-se por fator ρ o valor da razão à esquerda.

Lema 2 *Dada uma instância do problema $1|p_j;chains|\sum w_j C_j$, para cada cadeia $t_{i_1}, \ldots, t_{i_k}$ onde o fator ρ é dado pela tarefa l^*, existe um escalonamento ótimo onde as tarefas $t_{i_1}, \ldots, t_{i_{l^*}}$ são executadas consecutivamente, e sem interrupção.*

Idéia da prova: A idéia é de mostrar por contradição que não existe um escalonamento melhor, onde esta sequência é interrompida. Supondo-se, por absurdo, que existe um escalonamento melhor com uma tarefa sendo executada entre a sequência, isto é, $t_{i_1}, \ldots, t_{i_u}, v, t_{i_{u+1}}, \ldots, t_{i_{l^*}}$. Temos as seguintes desigualdades (lema 1):

$$\frac{w_v}{p_v} < \frac{\sum_{i=1}^{u} w_{t_i}}{\sum_{i=1}^{u} p_{t_i}} \text{ e } \frac{w_v}{p_v} > \frac{\sum_{i=u+1}^{l^*} w_{t_i}}{\sum_{i=u+1}^{l^*} p_{t_i}}.$$

Como o fator ρ é determinado por l^* segue que:

$$\frac{\sum_{i=u+1}^{l^*} w_{t_i}}{\sum_{i=u+1}^{l^*} p_{t_i}} \geq \frac{\sum_{i=1}^{u} w_{t_i}}{\sum_{i=1}^{u} p_{t_i}}.$$

Como o escalonamento $t_{i_1}, \ldots, t_{i_u}, v, t_{i_{u+1}}, \ldots, t_{i_{l^*}}$ é melhor que o escalonamento $t_{i_1}, \ldots, t_{i_{l^*}}, v$ então:

$$\frac{w_v}{p_v} > \frac{\sum_{i=u+1}^{l^*} w_{t_i}}{\sum_{i=u+1}^{l^*} p_{t_i}} \geq \frac{\sum_{i=1}^{u} w_{t_i}}{\sum_{i=1}^{u} p_{t_i}}$$

o que nos leva a uma contradição. Caso a cadeia seja interrompida por uma ou mais cadeias, a prova é semelhante.

□

O lema precedente é utilizado no algoritmo seguinte:

1. Enquanto existirem tarefas a executar faça
2. encontre a cadeia ch com maior fator ρ
3. execute a cadeia ch até a tarefa l^*

Além de grafos simples como cadeias, o problema também é polinomial para árvores [1]. Podemos também estudar um problema semelhante, onde nem todas as tarefas estão disponíveis no instante 0, isto é, o problema $1|p_j;r_j|\sum w_j C_j$, o qual é $\mathcal{NP}-$difícil mesmo quando todos os pesos são iguais. Um outro problema interessante pode ser obtido permitindo a interrupção de tarefas para o problema anterior.

Dada uma instância de $1|p_j;r_j;prmp|\sum w_j C_j$ um algoritmo que parece resolver o problema é a versão com interrupção do WSPT.

1. $r(p_j)$ denota o quanto da tarefa t_j ainda deve ser executado

2. enquanto existirem tarefas a serem executadas faça
3. encontre a tarefa t_j com a maior relação $w_j/r(p_j)$
4. execute t_j durante uma unidade de tempo

Além deste algoritmo nem sempre encontrar a solução ótima, o problema $1|p_j;r_j;prmp|\sum w_jC_j$ também é fortemente $\mathcal{NP}-$difícil. Por outro lado quando todos os pesos são iguais o problema pode ser resolvido pelo algoritmo acima.

Exemplo 17 *Para a seguinte instância de $1|p_j;r_j;prmp|\sum w_jC_j$ a versão com interrupção do WSPT não encontra a solução ótima.*

	t_1	t_2	t_3
p_j	*20*	*9*	*30*
r_j	*0*	*1*	*2*
d_j	*3*	*1*	*2*

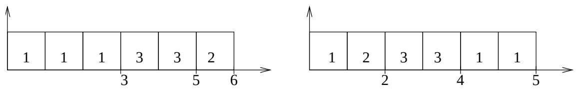


Figura 4.2: Escalonamento gerado pelo WSPT com interrupção e ótimo, com $\sum w_jC_j$ respectivamente 264 e 258.

*Na figura 4.2 temos os escalonamentos gerados pelo WSPT com interrupção e o ótimo. Os valores de $\sum w_jC_j$ são respectivamente: 3*20+5*30+6*9=264 e 2*9+4*30+6*20=258.*

4.2 $1|p_j;prec|h_{\max}$

Nesta seção $h_{\max}$ corresponde ao máximo de quaisquer funções não decrescentes $h_1,\ldots,h_n$, isto é:

$$h_{\max} = \max\{h_1(C_1),\ldots,h_n(C_n)\}.$$

Entre as formas mais comuns para $h_{\max}$, temos $L_{\max}, T_{\max}$ e $U_{\max}$.

O algoritmo que resolve este problema pode ser deduzido a partir das seguintes propriedades:

1. O escalonamento ótimo não possui tempo de inatividade, logo a última tarefa termina no instante $C_{\max} = \sum_{i=1}^{n} p_j$;
2. Começando-se, do final ao início, a partir de do instante final $C_{\max}$, a tarefa que minimiza $h_{\max}$ é t_j tal que $h_j(C_{\max}) = \min\{h_1(C_{\max}),\ldots,h_n(C_{\max})\}$;

   Para provar esta propriedade, suponha por absurdo que existe um escalonamento melhor S, onde a última tarefa não satisfaça este critério. Isto é, no escalonamento S, para última tarefa alocada j^* temos $h_{j^*}(C_{\max}) > \min\{h_1(C_{\max}),\ldots,h_n(C_{\max})\}$. Seja $t_{j'}$ a tarefa que minimiza $h_{\max}$. Construa o escalonamento S' a partir de S, de tal forma que a tarefa $t_{j'}$ seja executada por último, sem alterar a ordem das outras tarefas. Em S' todas as tarefas são executadas antes, exceto $t_{j'}$. Logo o único custo que aumentou foi o relativo a $t_{j'}$, que é por definição menor do que o custo de h_{j^*}, portanto o custo de S' é menor ou igual ao de S, contradição.
3. A partir de um escalonamento parcial onde um conjunto T de tarefas já está alocado, o próximo escalonamento parcial pode ser obtido alocando-se a tarefa t_j tal que

$$h_j(C_{\max} - \sum_{t_j \in T} p_j) = \min_{j \in \mathcal{T}\setminus T}\{h_j(C_{\max} - \sum_{t_j \in T} p_j)\}.$$

A demonstração é similar a anterior.

Logo um algoritmo de programação dinâmica resolve o problema:

1. $T = \emptyset, T^C = \{1, \ldots, n\}$;
2. T' =conjunto de tarefas sem sucessor;
3. enquanto $(T^C \neq \emptyset)$ faça
4. escolha j^* tal que $h_{j^*}\left(\sum_{t_j \in T^C} p_j\right) = \min_{j \in T'}\{h_j\left(\sum_{t_j \in T^C} p_j\right)\}$;
5. aloque j^* no instante $\left(\sum_{j \in T^C} p_j\right) - p_{j^*}$;
6. $T = T \cup \{j^*\}$;
7. $T^C = T^C - \{j^*\}$;
8. atualize T';

A cada um dos n passos do laço no máximo n tarefas devem ser analisadas, logo o algoritmo tem complexidade $O(n^2)$. Como para codificar uma instância do problema precisamos de $n \log x$ bits, onde x é o maior valor de p_j, o algoritmo é polinomial.

Um caso particular do problema é o $1|p_j|L_{\max}$, para o qual a versão com data de disponibilidade é fortemente $\mathcal{NP}-$difícil.

Teorema 2 *O problema de decisão associado a $1|p_j;r_j|L_{\max}$ é fortemente $\mathcal{NP}-$completo.*

Prova: Redução de 3-partição para $1|p_j;r_j|L_{\max}$.

1. O problema $1|p_j;r_j|L_{\max}$ pertence a $\mathcal{NP}$ pois pode se verificar em tempo polinomial se um escalonamento dado tem $L_{\max}$ menor ou igual a um valor dado B.
2. Dada uma instância $\{a_1, \ldots, a_{3t}\}$ e um valor B do problema da 3-partição, tais que $B/4 < a_j < B/2$ e $\sum_{j=1}^{3t} a_j = tB$, a seguinte instância de $1|p_j;r_j|L_{\max}$ pode ser construída. Sejam $n = 4t-1$ tarefas tais que:

t_j	$j = 1, \ldots, t-1$	t_j	$j = 1, \ldots, 4t-1$
$p_j = 1$		$p_j = a_{j-t+1}$	
$r_j = jB + (j-1)$		$r_j = 0$	
$d_j = jB + j$		$d_j = tB + (t-1)$	

Se existe um escalonamento com $L_{\max} = 0$, então cada uma das $t-1$ primeiras tarefas deve ser executada nos intervalos de tempo entre r_j e $d_j = r_j + 1$ (ver figura 4.3). As tarefas remanescentes estão alocadas em t intervalos de tamanho B. Logo, existe um escalonamento com $L_{\max} = 0$ se e só se existe uma solução para a 3-partição.$\square$

Por outro lado, a técnica de Branch& Bound se adapta bem para a resolução deste problema. Como metodologia para a divisão em sub-problemas (Branch) podemos usar a alocação de tarefas a partir do instante zero. Cada nível correspondendo a alocação de mais uma tarefa. Se a raiz corresponde a nenhum nó alocado, a mesma terá n filhos.

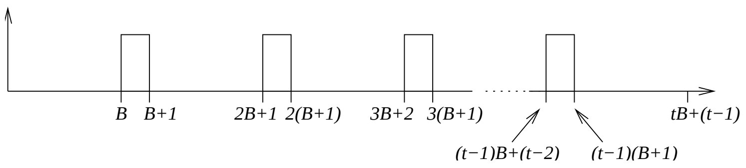


Figura 4.3: Alocação das tarefas t_j, $j = 1, \ldots, t-1$ em um escalonamento com $L_{\max} = 0$.

Em cada nível k da árvore as k primeiras tarefas estão alocadas logo estes nós terão $(n-k)$ filhos. A altura da árvore é $n+1$.

Um nó da árvore, no nível $k-1$, corresponde a $k-1$ tarefas alocadas: $t_1, \ldots, t_{k-1}$. Se t denota o makespan deste escalonamento parcial e T o conjunto de tarefas a escalonar, então uma tarefa $t_j \in T$ deve ser considerada como filho deste nó se $r_{t_j} < \min_{l \in T}\{\max\{t, r_l\} + p_l\}$. Pois caso contrário, a escolha das tarefas l que satisfazem $r_{t_k} \geq \max\{t, r_l\} + p_l$ para o escalonamento no instante t não altera o menor tempo possível para escalonar a tarefa t_k.

Limites para o Bound podem ser obtidos com heurísticas como a EDD (*Earliest Due Date*) onde as tarefas são ordenadas em ordem crescente de datas devidas. Como a heurística EDD não fornece um limite inferior usaremos o algoritmo EDD com interrupção. É interessante observar que o limite inferior assim obtido não é necessariamente atingível por um escalonamento sem interrupção. Caso o algoritmo EDD com interrupção forneça um escalonamento sem interrupção, este pode ser o novo limite superior do problema.

Exercício 6 *Resolva por B& B a seguinte instância com 4 tarefas:*

	1	2	3	4
p_j	4	2	6	5
r_j	0	1	3	5
d_j	8	12	11	10

Na figura 4.4 está esquematizado um rascunho da resolução do problema.

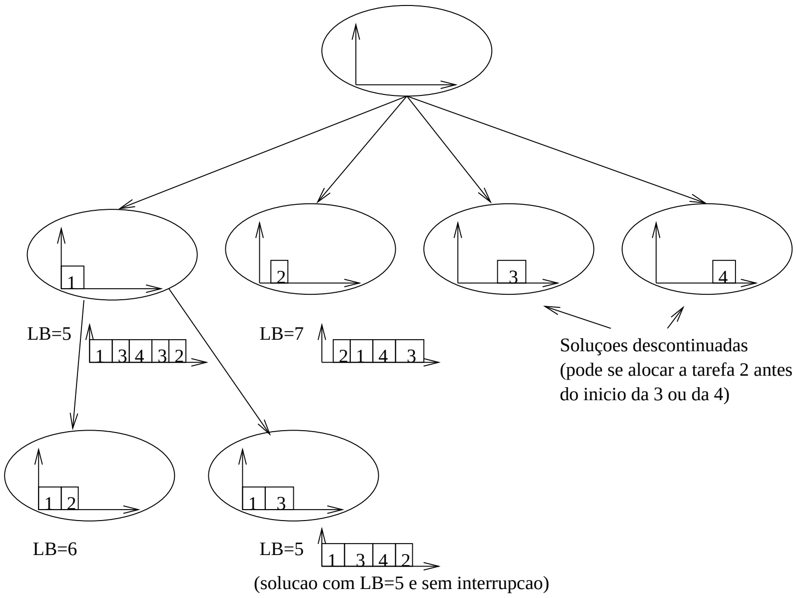


Figura 4.4: Exemplo de resolução de $1|p_j; r_j|L_{\max}$ por Branch&Bound.

4.3 $\sum U_j$ - Número de tarefas atrasadas

Neste caso como todas as tarefas tem o mesmo peso queremos maximizar o número de tarefas que é completado antes das datas devidas. Para isto o escalonamento pode ser construído tarefa a tarefa através da regra EDD. Caso a tarefa escolhida não possa ser completada a tempo, a tarefa com maior tempo de execução é retirada e a construção continua. No algoritmo abaixo T denota as tarefas escalonadas, T^D as tarefas que foram descartadas e T^C as tarefas à escalonar.

1. $T = T^D = \emptyset$, $T^C = \{1, \ldots, n\}$;
2. enquanto $(T^C \neq \emptyset)$ faça
3. escolha a tarefa j^* tal que $d_{j^*} = \min_{j \in T^C} d_j$
4. $T = T + j^*$;
5. $T^C = T^C - j^*$;
6. se $(\sum_{j \in T} > d_{j^*})$ então /* a tarefa t_j acaba após o prazo */
7. seja k^* a maior tarefa em T $(p_{k^*} = \max(j \in T)p_j$;
8. $T = T \setminus k^*$;
9. $T^D = T^D + k^*$;

Teorema 3 *O algoritmo acima contrói um escalonamento ótimo para* $1|p_j|\sum U_j$.

idéia da prova: Sem perda de generalidade supomos que as tarefas $(1, \ldots, n)$ estão ordenadas em ordem crescente de data devida $d_1 \leq d_2 \leq \ldots \leq d_n$. Seja $T_k \subset \{1, \ldots, k\}$ tal que:

1. T_k tem o maior número de tarefas, N_k, completadas antes da data devida;
2. Entre todos os subconjuntos de $\{1, \ldots, k\}$ com N_k tarefas completadas antes da data devida, as tarefas em T_k tem o menor tempo de processamento.

Observando que T_n corresponde ao maior escalonamento possível a prova é feita por indução:

1. O algoritmo constrói T_1. trivial;
2. A partir de T_k o algoritmo constrói T_{k+1}:

   caso 1: Se a tarefa $k+1$ é adicionada, $N_{k+1} = N_k + 1$ que por indução é o número máximo de tarefas. Como a tarefa $k+1 \in T_{k+1}$ o tempo é mínimo;

   caso 2: Caso contrário $N_k = N_{k+1}$, mas a tarefa com o maior tempo de execução é retirada, o que faz com que o tempo de execução de T_{k+1} seja menor ou igual ao tempo de execução de T_k.

Exemplo 18 *Dadas as 5 tarefas abaixo:*

	1	2	3	4	5
p_j	7	8	4	6	6
d_j	9	17	18	19	21

Um escalonamento ótimo tem a seguinte seqüência de tarefas $(3, 4, 5, 1, 2)$ *com* $\sum U_j = 2$.

Conforme já vimos no capítulo de complexidade, a generalização deste problema $1|p_j|\sum w_j U_j$ é equivalente ao problema da mochila.

4.4 Atraso total $\sum T_j$

Ao invés de se minimizar apenas o número de tarefas atrasadas, neste caso também queremos minimizar a soma deste atraso. O problema $1|p_j|\sum T_j$ é $\mathcal{NP}-$difícil mas existe um algoritmo com tempo pseudo polinomial que encontra a solução ótima.

Sem perda de generalidade, vamos supor que a ordem das datas devidas das tarefas é crescente, isto é $d_1 \leq d_2 \leq \ldots \leq d_n$. Seja $T_k \subset \{1, \ldots, k\}$. Se $p_k = \max\{p_1, \ldots, p_n\}$, isto é a k-ésima tarefa tem o maior tempo de processamento o seguinte lema é válido:

Lema 3 *Existe um inteiro δ entre 0 e $n-k$ tal que existe um escalonamento ótimo S onde a tarefa t_k é precedida pelas tarefas t_j com $j \leq k+\delta$ e seguida pelas tarefas t_j com $j > k+\delta$.*

Logo para construir um escalonamento ótimo de um conjunto de tarefas $\{1, \ldots, l\}$ a partir do instante t podemos usar programação dinâmica. Seja k tal que $p_k = \max\{p_1, \ldots, p_l\}$, a partir de lema anterior sabemos que para algum δ $(0 \leq \delta \leq l-k)$ existe um escalonamento ótimo que corresponde a concatenação de 3 subconjuntos:

1. das tarefas $\{1, 2, \ldots, k-1, k+1, \ldots, k+\delta\}$ em alguma ordem começando no instante t;
2. da tarefa t_k;
3. das tarefas $\{k+\delta+1, \ldots, k_l\}$ em alguma ordem.

O instante de término da tarefa k $(C_k(\delta))$ é dado por $t+\sum_{j\leq k+\delta} p_j$. Mas o processo é recursivo pois para encontrar a seqüência ótima as sub-seqüências dos ítens 1 e 3 também devem ser ótimas.

Para apresentar o algoritmo de Lawler utilizaremos as seguintes notações:

- $J(j, l, k)$ a sub-seqüência com as tarefas de j a l que tem tempo de execução menor do que p_k (seguindo o raciocínio da divisão em subconjuntos, p_k não pertence aos conjuntos dos ítens 1 e 3);
- $V(J(j, l, k), t)$ atraso total do subconjunto $J(j, l, k)$ para uma seqüência ótima que começa no instante t.

Algoritmo:

1. condições iniciais:
2. $V(\emptyset, t) = 0$, sem tarefas o atraso é nulo;
3. $V(\{j\}, t) = \max\{0, t + p_j - d_j\}$, atraso com apenas uma tarefa alocada no instante t.
4. recorrência:
5. para o cálculo de $V(J(j, l, k), t)$ seja k' a tarefa com o maior tempo de execução $(p_{k'} = \max(p_{j'}|j' \in J(j, l, k)))$;
6. $V(J(j, l, k), t) = \min_\delta(V(J(j, k'+\delta, k'), t) + \max\{0, C_{k'}(\delta) - d_{k'}\} + V(J(k'+\delta+1, l, k')), C_{k'}(\delta))$.

o valor ótimo de $\sum T_j$ é dado por $V(J(1, n, t_\infty), 0)$ onde t_∞ corresponde a uma tarefa com um tempo p_∞ maior que o maior tempo de execução entre as tarefas de 1 a n. A complexidade deste algoritmo é $O(n^4 \sum p_j)$.

Exemplo 19 *Apresentaremos a idéia de execução para o seguinte exemplo com cinco tarefas. As tarefas do exemplo já estão ordenadas em ordem crescente de data devida.*

	1	2	3	4	5
p_j	121	79	147	83	130
d_j	260	266	266	336	337

Como o maior tempo de execução é o da tarefa 3, logo:

$$V(J(1,5,t_\infty),0) = \min \begin{cases} V(J(1,3,3),0) + (347-266) + V(J(4,5,3),347); \\ V(J(1,4,3),0) + (430-266) + V(J(5,5,3),430); \\ V(J(1,5,3),0) + (560-266) + V(\emptyset,560). \end{cases}$$

Pelas condições iniciais sabemos que $V(\emptyset,560) = 0$ *e* $V(J(5,5,3),430) = 560-337$. *Em seguida os valores de* $V(J(1,3,3),0), V(J(4,5,3),347), V(J(1,4,3),0)$ *e* $V(J(1,5,3),0)$ *devem ser calculados recursivamente.*

4.5 Atraso total ponderado

Nesta seção veremos que este problema é fortemente $\mathcal{NP}-$difícil e proporemos um algoritmo de Branch and Bound.

Teorema 4 *O problema de decisão associado a* $1|p_j|\sum w_jT_j$ *é fortemente* $\mathcal{NP}-$*completo.*

Idéia da Prova:

- Dado um escalonamento é fácil verificar em tempo polinomial (no caso linear) que o valor de $\sum w_jT_j$ é menor ou igual a um valor L dado.
- Redução do problema da 3-partição:

  Dada uma instância do problema da 3-partição $a_1,\ldots,a_{3t}$, B, considere o seguinte problema com $4t-1$ tarefas:

	$t_j, j=1,\ldots,3t$	$t_j, j=3t+1,\ldots,4t-1$
d_j	0	$(j-3t)(B+1)$
p_j	a_j	1
w_j	1	2

  Seja $L=\sum_{j,k} a_ja_k + \frac{1}{2}(t-1)tB$. A prova se baseia nas seguintes constatações:

  - Caso as tarefas $t_j, j=3t+1,\ldots,4t-1$ estejam escalonadas nos intervalos $[B,B+1],[2B+1,2B+2],\ldots[(t-1)(B+1)+B,(t-1)(B+1)]$ e as tarefas $t_j, j=1,\ldots,3t$ estejam escalonadas nos intervalos $[0,B],[B+1,2B+1],\ldots[2B+2,3B+2]$ (isto só é possível se existir uma 3-partição!), o atraso total ponderado será exatamente L;
  - Caso contrário, basta verificar que alguma das partições terá tamanho maior do que B, o que faria com que o escalonamento tivesse atraso total ponderado maior do que L.

Este problema também pode ser resolvido por Branch and Bound, para eliminar parte das soluções usaremos o seguinte lema:

Lema 4 *Se existem duas tarefas* j *e* k *com* $d_j \leq d_k, p_j \leq p_k$ *e* $w_j \geq w_k$ *existe um escalonamento ótimo onde a tarefa* j *aparece antes da tarefa* k.

Para o B&B vamos alocar as tarefas do final ao início:

Branch alocação de uma nova tarefa;

Bound baseado no problema de transporte:

- cada tarefa j com tempo de execução p_j é dividida em p_j tarefas unitárias;
- a variável de decisão x_{jk} é 1 se uma unidade da tarefa j é executada no intervalo de tempo $[k-1, k]$, e 0 caso contrário;
- as variáveis de decisão devem obedecer as seguintes condições:
  $\sum_{k=1}^{C_{\max}} x_{jk} = p_j, j = 1, \ldots, n$ (todas as tarefas são executadas completamente)
  $\sum_{j=1}^{n} x_{jk} = 1, k = 1, \ldots, C_{\max}$ (a cada instante apenas uma tarefa é executada)

Uma solução para o problema de transporte, não é necessariamente um escalonamento válido para $1|p_j|\sum w_jT_j$ pois as tarefas podem ser interrompidas.

Seja c_{jk} os coeficientes tais que

$$\sum_{k=l-p_j+1}^{l} c_{jk} \leq w_j \max\{l-d_j, 0\}, j = 1, \ldots, n, l = 1, \ldots, C_{\max}. \tag{4.1}$$

Uma solução com custo mínimo é um limite inferior, pois para qualquer solução do problema de transporte com $x_{jk} = 1$ para $k = C_j - p_j + 1, \ldots, C_j$ temos

$$\sum_{k=1}^{C_{\max}} c_{jk}x_{jk} = \sum_{k=C_j-p_j+1}^{C_j} c_{jk} \leq w_j \max\{C_j - d_j, 0\}.$$

A equação 4.1 é válida para os seguintes valores de c_{jk}:

$$c_{jk} = \begin{cases} 0, \text{ para } k \leq d_j; \\ w_j, \text{ para } k > d_j. \end{cases}$$

Exemplo 20 *Estudaremos o comportamento desse algoritmo de B&B para o seguinte exemplo com 4 tarefas.*

	1	2	3	4
w_j	4	5	3	5
p_j	12	8	15	9
d_j	16	26	25	27

Somando-se os tempos de execução das tarefas sabemos que o makespan é 44. Usando-se o lema 4 devemos considerar apenas as tarefas 3 e 4 como candidatas para terminar a execução no instante 44 ($p_1 \leq p_3, d_1 \leq d_3$ e $w_1 \geq w_3$, a mesma relação é válida entre as tarefas 2 e 4). Logo, devemos calcular o limite inferior para estes dois casos. Ver figura 4.5

Quando a tarefa 4 é alocada por último obtemos um limite inferior para $|\sum w_jT_j$ de 115, por outro lado quando a tarefa 3 é a última, o limite inferior é 67. Como a resolução do problema de transporte na obtenção deste limite forneceu um escalonamento sem interrupção, esta solução é uma solução ótima.

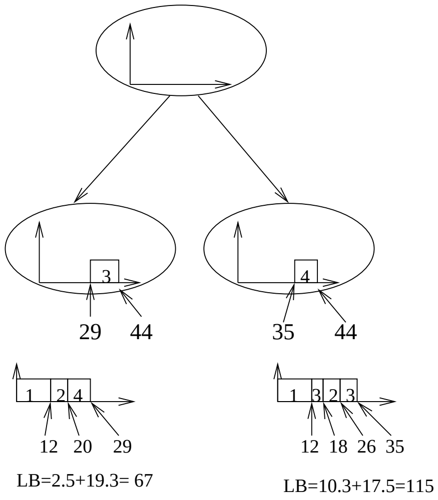


Figura 4.5: Exemplo de resolução de $1|p_j|\sum w_j T_j$ por Branch&Bound.

Capítulo 5

Máquinas paralelas P_m

As principais funções de minimização para máquinas paralelas são: makespan ($C_{\max}$), tempo total de término ($\sum C_j$) e atraso máximo $L_{\max}$.

5.1 $P_m|p_j|C_{\max}$

Conforme já vimos anteriormente o problema para m fixo igual a 2 já é $\mathcal{NP}-$difícil ($P_2|p_j|C_{\max}$) é equivalente ao problema da partição). Também já vimos a heurística LPT:

1. Ordene as tarefas em ordem decrescente em p_j;
2. $j = 1$;
3. Enquanto ($j \leq m$) faça
4. Aloque a tarefa t_j na máquina onde ela puder ser alocada antes.

Em seguida veremos como medir o quanto a solução produzida por este algoritmo está longe da solução ótima. Para isto vamos usar a razão entre uma solução LPT ($C_{\max}(LPT)$) e uma solução ótima ($C_{\max}(OPT)$). Veremos que é possível conhecer esta razão, mesmo sem o conhecimento do makespan de um escalonamento ótimo.

Teorema 5 *Para $P_m|p_j|C_{\max}$ temos que*

$$\frac{C_{\max}(LPT)}{C_{\max}(OPT)} \leq \frac{4}{3} - \frac{1}{3m}.$$

Prova por contradição: Suponha por absurdo que existe pelo menos um contra-exemplo onde

$$\frac{C_{\max}(LPT)}{C_{\max}(OPT)} > \frac{4}{3} - \frac{1}{3m}.$$

Seja $S = \{t_1, \ldots, t_n\}$ um contra-exemplo com o menor número de tarefas: n. Observando a política LPT, a última e menor tarefa, t_n é a última a começar a sua execução. Mas como S é um contra-exemplo com o menor número de tarefas, t_n também é a última a terminar, pois caso contrário $S' = \{t_1, \ldots, t_{n-1}\}$ teria o mesmo makespan que S logo:

$$\frac{4}{3} - \frac{1}{3m} < \frac{C_{\max}(LPT(S))}{C_{\max}(OPT(S))} = \frac{C_{\max}(LPT(S'))}{C_{\max}(OPT(S))} \leq \frac{C_{\max}(LPT(S'))}{C_{\max}(OPT(S'))}. \rightarrow | \leftarrow .$$

O que implica que a tarefa t_n começa no instante $C_{\max}(LPT(S)) - p_n$, como neste instante as outras máquinas ainda estão ocupadas temos

$$C_{\max}(LPT(S)) - p_n \leq \frac{\sum_{j=1}^{n-1} p_j}{m},$$

logo

$$C_{\max}(LPT(S)) \leq p_n + \frac{\sum_{j=1}^{n-1} p_j}{m} = p_n(1 - \frac{1}{m}) + \frac{\sum_{j=1}^{n} p_j}{m}$$

mas como o makespan ótimo é maior ou igual ao trabalho total dividido pelos processadores,

$$C_{\max}(OPT(S)) \geq \frac{\sum_{j=1}^{n} p_j}{m},$$

combinando as duas equações temos:

$$\frac{4}{3} - \frac{1}{3m} < \frac{C_{\max}(LPT)(S)}{C_{\max}(OPT)(S)} \leq \frac{p_n(1 - \frac{1}{m}) + \frac{\sum_{j=1}^{n} p_j}{m}}{C_{\max}(OPT)(S)},$$

logo

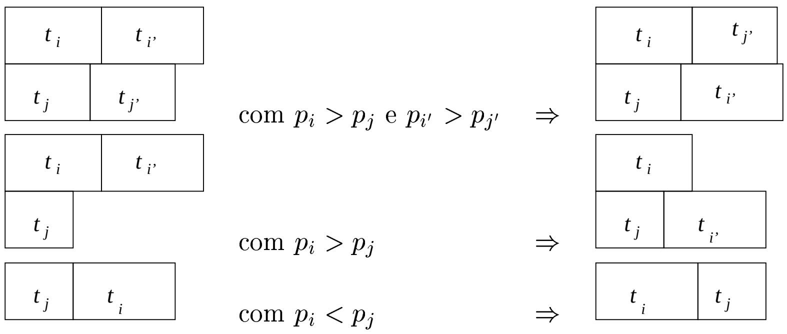
$$\frac{1}{3} - \frac{1}{3m} < \frac{p_n(1 - \frac{1}{m})}{C_{\max}(OPT(S))}$$

$$C_{\max}(OPT(S)) < \frac{p_n(1 - \frac{1}{n})}{\frac{1}{3} - \frac{1}{3m}} = 3p_n$$

que implica que no escalonamento ótimo existem no máximo duas tarefas por máquina.

Pode se mostrar que dado um escalonamento ótimo com no máximo duas tarefas por máquina este pode ser transformado em um escalonamento LPT com o mesmo makespan. Idéias:

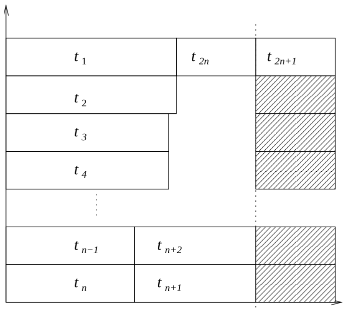
As seguintes alterações no escalonamento não aumentam o makespan:

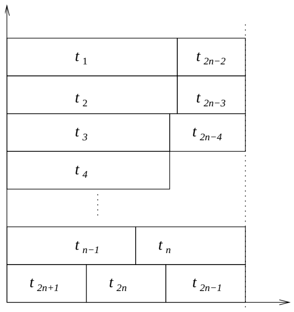


usando-se estas propriedades pode-se construir um escalonamento que seria o mesmo produzido pelo algoritmo LPT, desta forma chegamos a

$$C_{\max}(OPT(S)) = C_{\max}(LPT(S)), \rightarrow | \leftarrow .$$

Além disto, este limite é o melhor possível pois para m máquinas o seguinte conjunto de pesos de tarefas: $(p_1, \ldots, p_r) = (2m-1, 2m-1, 2m-2, 2m-2, \ldots, m+1, m+1, m, m, m)$ onde $r = 2m+1$ temos:

LPT 

ótimo 

Como o $C_{\max}$ do escalonamento LPT é $4m-1$ e o makespan do escalonamento ótimo é $3m$. A razão entre os dois é:

$$\frac{4}{3} - \frac{1}{3m}.$$

Vejamos agora o caso onde existem relações de precedência.

5.2 $P_m|prec;p_j|C_{\max}$

Primeiro supondo que o número de processadores não é limitado (por exemplo $m \geq n$) temos o seguinte algoritmo:

1. seja F o conjunto de tarefas sem predecessor;
2. enquanto ($F \neq \emptyset$)
3. escalone todas as tarefas em F;
4. atualize o conjunto F;

O makespan deste algoritmo corresponde ao caminho crítico, logo é ótimo. Para descobrir em um grafo quais tarefas fazem parte do caminho crítico podemos:

1. Descobrir o menor tempo possível de alocação para cada tarefa;
2. Executar o passo anterior no caminho inverso, e descobrir qual o maior tempo de alocação possível para cada tarefa.

As tarefas para as quais os dois tempos de alocação forem iguais fazem parte do caminho crítico.

Exercício 7 *Descubra quais tarefas fazem parte do caminho crítico no exemplo da figura 5.1. Os tempos de processamento de cada tarefa estão listados abaixo:*

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
p_j	*4*	*9*	*3*	*3*	*6*	*8*	*8*	*12*	*6*

Quando o número de máquinas é limitado os algoritmos de lista tem garantia $2 - \frac{1}{m}$.

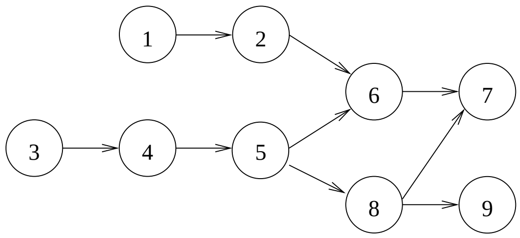


Figura 5.1: Grafo de precedência de exercício 7.

5.3 $P_m|p_j; prmp|C_{\max}$

Para a minimização do makespan com interrupções, vamos supor, sem perda de generalidade, que as tarefas estão ordenadas em ordem decrescente de p_j.

Para a resolução deste problema, poderíamos primeiramente observar um problema de programação linear equivalente:

$$
\begin{array}{l}
min C_{\max} \\
\text{sujeito a} \left\{ \begin{array}{l}
\sum_{i=1} m x_{ij} = p_j, j = 1, \ldots, n \text{ (as tarefas são totalmente executadas)} \\
C_{\max} - \sum_{i=1}^{m} x_{ij} \leq 0, j = 1, \ldots, n \text{ (o tempo de processamento de cada tarefa é } \leq C_{\max}) \\
C_{\max} - \sum_{j=1}^{n} x_{ij} \leq 0, i = 1, \ldots, m \text{ (cada máquina executa } \leq C_{\max} \text{ tarefas)}
\end{array} \right. \\
x_{ij} \geq 0, i = 1, \ldots, m, j = 1, \ldots, n.
\end{array}
$$

onde a variável x_{ij} representa o tempo gasto pela tarefa j na máquina i. Apesar deste problema ser polinomial, existe uma solução mais simples para resolver o problema de escalonamento com interrupção.

Lema 5 *O limite inferior de um escalonamento $P_m|prec; p_j|C_{\max}$ é*

$$LB = \max \left\{ p_1, \left\lceil \frac{\sum_{j=1}^{n} p_j}{m} \right\rceil \right\}.$$

Usando este lema obtemos o seguinte algoritmo:

1. Aloque as tarefas em um única máquina, uma após a outra, sem interrupção. Esta alocação terá makespan $\sum_{j=1}^{n} p_j$ que é $\leq mLB$;
2. Corte este escalonamento em m partes, a primeira $[0, LB]$, a segunda $[LB, 2LB]$ e assim por diante. Aloque cada parte em uma máquina.

Como a maior tarefa é menor ou igual a LB este escalonamento é válido. Um outro algoritmo possível é o LRPT (longest remaining processing time) onde são alocadas a cada instante uma unidade de execução de cada tarefa.

1. i=0;
2. Enquanto existirem tarefas a executar faça
3.   Aloque uma unidade de tempo das m maiores tarefas no instante i;
4. $i = i + 1$;

Apesar do algoritmo LRPT também produzir um escalonamento válido o número de interrupções pode ser bem maior do que $m-1$ (número máximo de interrupções do algoritmo anterior).

Exemplo 21 *Para a seguinte instância com 3 máquinas e 5 tarefas com pesos respectivos:* $(4,4,3,3,2,2,2)$ *o algoritmo LRPT pode produzir um escalonamento com até 18 interrupções (ver figura 5.2).*

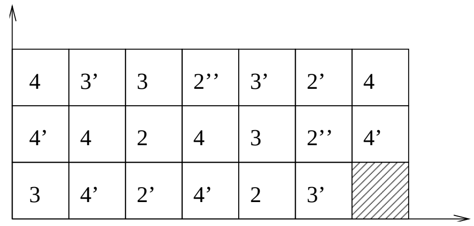


Figura 5.2: Exemplo de escalonamento gerado pelo algoritmo LRPT.

5.4 Soma do tempo de término

Nesta seção estudaremos o problema $P_m|p_j|\sum C_j$. Já sabemos que o problema $1|p_j|\sum w_jC_j$ é polinomial, logo o problema $1|p_j|\sum C_j$ também é polinomial e o escalonamento ótimo é obtido alocando-se as tarefas em ordem crescente de tempo de execução.

Este resultado também pode ser visto através das permutações $I=(i_1,\ldots,i_n)$ de $(1,\ldots,n)$. Um escalonamento pode ser representado pela ordem das tarefas, isto é, por uma permutação. Dada uma permutação I temos que:

$$\sum C_j = p_{i_1} + (p_{i_1}+p_{i_2}) + (p_{i_1}+p_{i_2}+p_{i_3}) + \ldots + \sum_{k=1}^{n} p_{i_k},$$
$$= np_{i_1} + (n-1)p_{i_2} + (n-2)p_{i_3} + \ldots + p_{i_n}.$$

De forma a minimizar esta soma, vemos que p_{i_1} deve corresponder ao menor valor de p_j, p_{i_2} ao segundo menor, e assim por diante. Logo os p_j devem estar em ordem crescente. Com este tipo de argumento provaremos o seguinte teorema:

Teorema 6 *A regra SPT (shortest processing time) é ótima para o problema* $P_m|p_j|\sum C_j$.

Idéia da Prova: Suponha inicialmente que n é múltiplo de m (caso contrário acrescente $m-n\%m$ tarefas de tamanho zero). De forma a minimizar os termos com grandes coeficientes multiplicativos devem ser alocadas n/m tarefas por máquina (caso contrário teríamos máquinas com coeficiente $\geq (n/m)+1$ e outras com coeficiente $\leq (n/m)-1$). As m maiores tarefas devem ter coeficiente 1, as segundas m maiores tarefas devem ter coeficiente multiplicativo 2, e assim por diante. Como a regra SPT contrói um escalonamento desta forma, este é ótimo.□

O problema $P_m|prec;p_j|\sum C_j$ é fortemente $\mathcal{NP}-$difícil, e o problema $P_m|p_j|\sum w_jC_j$ é $\mathcal{NP}-$difícil mas pode ser resolvido por um algoritmo pseudo-polinomial.

5.5 Atraso máximo

O problema $P_m|prmp;p_j|L_{\max}$ é um dos poucos problemas para máquinas paralelas com data devida que é polinomial.

Se existe um escalonamento com $L_{\max} = z$ temos que para cada tarefa t_j o seu tempo de término C_j é menor ou igual a $d_j + z$. Podemos encontrar, se existir, um escalonamento onde $C_j \leq d_j + z$ da seguinte forma:

Aplique a regra LRPT, começando de trás para frente, na última data devida (já levando em conta o valor de z). As datas limites do problema original serão as datas de disponibilidade deste problema. Se o escalonamento obtido for válido com todas as tarefas alocadas a partir do instante zero, então existem um escalonamento para $P_m|prmp; p_j|L_{\max}$ com $L_{\max} \leq z$.

Exemplo 22 *Verifique se existe um escalonamento com $L_{\max} = 0$ para a seguinte instância em duas máquinas:*

	1	*2*	*3*	*4*
p_j	*3*	*3*	*3*	*8*
d_j	*4*	*5*	*8*	*9*

Começando de trás para frente trocamos d_j por $r_j = 9 - d_j$.

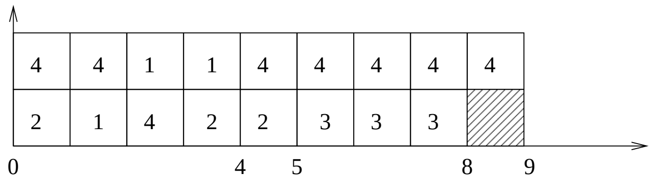


Figura 5.3: Solução com $L_{\max} = 0$.

Exercício: verificar que se a tarefa 1 tivesse tempo de execução 4, não seria possível encontrar uma solução com $L_{\max} = 0$.

Capítulo 6

Escalonamento com atraso de comunicação

Este problema está diretamente relacionado aos computadores paralelos com memória distribuída. Os processos que compõe uma aplicação paralela correspondem às tarefas, e a comunicação entre processos executados por máquinas diferentes aos atrasos de comunicação.

Três características principais serão estudadas:

- A duplicação de tarefas: possibilidade de se executar uma mesma tarefa mais do que uma vez, com o objetivo de reduzir os atrasos de comunicação;
- Quantidade de máquinas: como estes problemas são geralmente difíceis uma outra forma de reduzir o impacto dos atrasos é de se utilizar um número não limitado de máquinas;
- Atraso de comunicação: neste caso estaremos interessados na razão entre os tempos de comunicação e execução das tarefas (também conhecida por granularidade da aplicação paralela). Usa-se a seguinte razão: $\rho = \max c_{jk} / \min p_j$ onde $\rho \leq 1$ denota os problemas com pequenos tempos de comunicação (SCT) e $\rho > 1$ os problemas com tempo de comunicação grande (LCT).

Como uma das características possíveis é a duplicação, usa-se a seguinte notação:

Definição 11 *O escalonamento de uma cópia de uma tarefa t_j em uma máquina m_i no instante t é indicado pela tripla (t_j, m_i, t).*

Um arco no grafo de precedência da tarefa t_k à tarefa t_j implica que a tarefa t_j necessita dos dados produzidos pela tarefa t_k. Logo um escalonamento válido deve satisfazer:

1. Cada tarefa t_j é executada ao menos uma vez;
2. A cada instante uma máquina executa apenas uma tarefa;
3. Se existe um arco de t_k à t_j então cada cópia de t_j, (t_j, m_i, t) necessita dos dados de uma cópia de t_k, $(t_k, m_{i'}, t')$ tal que $t \geq t' + p_k$, se $i = i'$ e $t \geq t' + p_j + c_{kj}$ se $i \neq i'$.

Usaremos também as seguintes notações:

$pre(t_j)$ predecessores diretos da tarefa t_j;

$suc(t_j)$ sucessores diretos da tarefa t_j;

$p_{\min}$, $p_{\max}$ tempos de execução da menor e maior tarefa;

$c_{\min}$, $c_{\max}$ tempos da menos e da maior comunicação.

Estudaremos os problemas de minimização do makespan $P_m|prec; p_j; c_{jk}; dup|C_{\max}$. Veremos também alguns problemas onde a duplicação não é permitida. Veremos os problemas em ordem crescente de dificuldade.

6.1 $P_\infty|prec; p_j; c_{jk}; dup|C_{\max}$

6.1.1 SCT

Primeiramente estudaremos o problema assumindo que os tempos de comunicação são pequenos ($p_{\min} \geq c_{|max}$). Neste caso existe um algoritmo polinomial que escalona todas as cópias no menor instante possível (b_j). Para calcular estes tempos usamos o seguinte procedimento:

$$b_j = \begin{cases} 0, & \text{se } pre(t_j) = \emptyset \\ b_s + p_s, & \text{se } pre(t_j) = \{t_s\} \\ min_{t_s \in pre(t_j)}\{b_s + p_s, \max_{t_k \in pre(t_j)\setminus\{t_s\}}\{b_k + p_k + c_{kj}\}, & \text{caso contrário.} \end{cases}$$

Um arco (t_j, t_k) no grafo de precedência é denominado crítico se $b_j + p_j + c_{jk} > b_k$, neste caso uma cópia de t_j deve ser executada na mesma máquina onde a tarefa t_k será executada. Para obter o escalonamento removemos os arcos não críticos e alocamos os caminhos remanescentes, um por processador.

Exemplo 23 *Dado o grafo de precedência da figura 6.1, assim como os tempos de processamento e de comunicação (localizados abaixo das tarefas, ou junto dos arcos). Os instantes mínimos para alocação são:*

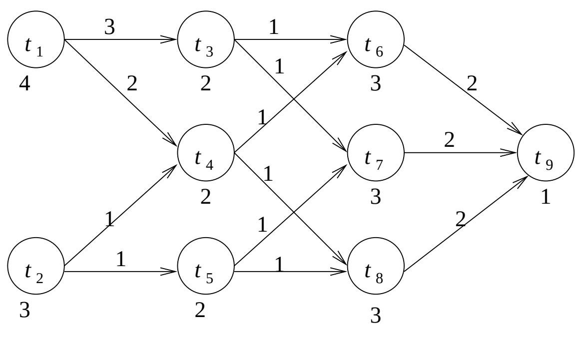


Figura 6.1: Grafo de precedência com atraso de comunicação.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
b_j	*0*	*0*	*4*	*4*	*3*	*7*	*6*	*6*	*11*

Os arcos não críticos foram removidos da figure 6.2. Basta agora alocar cada caminho em um processador (o que vai implicar na duplicação da tarefa t_1) a partir do instante b_j.

Observe que no exemplo anterior $p_{\min} < c_{\max}$, mas o algoritmo continua válido sob a seguinte hipótese: para cada tarefa t_j, $\max\{c_{kj} | t_k \in pre(t_j)\} \leq \min\{p_k | p_k \in pre(t_j)\}$. O algoritmo anterior fornece um escalonamento ótimo,

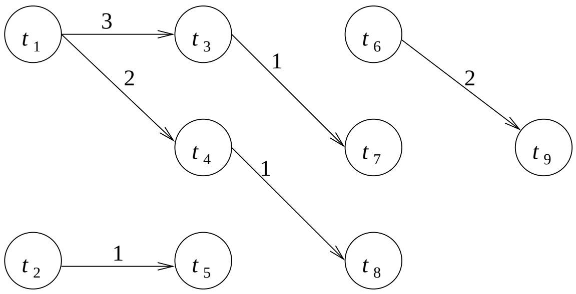


Figura 6.2: Arcos crítico do grafo de precedência da figura 6.1.

mas não minimiza o número de processadores utilizado. A minimização do número de processadores, sem alterar o makespan mínimo, é um problema $\mathcal{NP}-$difícil.

6.1.2 LCT

Antes de analisarmos o problema $P_\infty|LCM; p_j; c_{jk}|C_{\max}$ veremos um problema um pouco mais simples, mas equivalente: $P_\infty|p_j = 1; c_{jk} = \tau > 1|C_{\max}$. Esta foi a abordagem de Papadimitriou e Yannakakis em um dos trabalhos mais importantes para o escalonamento em máquinas paralelas. Neste trabalho, além de mostrar que o problema é difícil, os autores também encontraram um algoritmo de aproximação com razão 2.

Na primeira parte os autores mostrarm que decidir se existe um escalonamento de $P_\infty|p_j = 1; c_{jk} = \tau > 1|C_{\max}$ para um dado $C_{\max}$ é NP-completo.

A prova é uma redução do problema CLIQUE.

Dado um grafo $G(V, E)$ e um inteiro k, sendo que G tem ao menos $\binom{k}{2}$ arestas. Vamos construir um grafo de precedência $D(U, A)$ e dar um valor para a comunicação τ tal que D pode ser escalonado em tempo $C_{\max}$ se e só se G tem um clique de tamanho k.

D é construído da seguinte forma:

- para cada $v \in V$ existe em D um caminho $(d_{v1}, \ldots, d_{v|V|^2})$ com arcos $(d_{vi}, d_{v,i+1})$, $i = 1, \ldots, |V|^2 - 1$;
- para cada aresta $e = [u, v]$ em E existe um vértice c_{uv} em D também existe um nó t em D;
- os arcos de D são definidos da seguinte forma. Existe um arco de $(d_{v|V|^2}, c_e)$ do último vértice no caminho para os vértices correspondentes a aresta e, onde $v \in e$;
- para todo $e \in E$ existe um arco (c_e, t);
- seja $\tau = |V|^2(k-1) + \binom{k}{2}$ e $C_{\max} = |V|^2 k + |E| + 1$.

Suponha agora que exista um escalonamento onde o nó $c_{v,u}$ é executado no processador p antes do tempo $d = |V|^2 + \binom{k}{2}$. Mas, para isto é necessário que as cadeias correspondentes a u e v também sejam executadas em p, pois não há tempo para efetuar uma comunicação. Por outro lado, o vértice t também precisa ser executado no mesmo processador que todos os vértices que forem executados após, ou no instante, d. Isto implica que existem no máximo $|E| - \binom{k}{2}$ arestas executadas em tempos maiores ou iguais a d.

Cada uma das $\binom{k}{2}$ arestas restantes devem ser executadas no mesmo processador que executa os caminhos dos seus nós. Como para estas não é possível fazer comunicações, todas estas arestas, assim como t devem ser executados no mesmo processador. Além disto, este processador pode executar no máximo $C_{\max} - 1 - |E|/|V|^2 = k$ caminhos. Enretanto, isto implica que existem k vértices em V que são adjacentes a $(\frac{k}{2})$ arestas, logo um clique de tamanho k existe.

Por outro lado, se existe um clique de tamanho k, um escalonamento possível é o seguinte: execute cada caminho fora do clique em um processador, e os caminhos do clique e suas arestas em outro. Finalmente, execute os vértices remanescentes neste mesmo processador (exatamente quando chegarem as comunicações dos outros caminhos), e finalmente t.

Ver figura na página 72 do livro Scheduling Theory and its applications.

6.1.3 Um algoritmo de aproximação

Dado um grafo de precedência $G(V, A)$ e um inteiro τ seja a seguinte função:

1. se v é origem, então $e(v) = 0$;
2. caso contrário, sejam os ancestrais de v, ordenados em ordem decrescente de $e(u)$, $(e(u_1) \geq e(u_2) \geq \ldots \geq e(u_p))$. Seja $k = \min\{\tau + 1, p\}$, $e(v) = e(u_k) + k$.

Lema 6 *Não existe escalonamento onde o vértice v é escalonado antes do tempo $e(v)$.*

Prova: Por indução. Ok para vértices origem. Suponha que um vértice v pode ser escalonado em $t < e(v)$. Sejam seus k ancestrais $(u_1, \ldots, u_k)$ (conforme acima). Por indução, cada vértice u_i foi escalonado em, ou após, $e(u_i) \geq e(u_k)$. Como $t < e(v) = e(u_k) + k \leq e(u_k) + \tau + 1$, cada vértice u_i é escalonado a menos de $\tau + 1$ antes de v, logo devem estar no mesmo processador que v. Como existem k deste nós entre $e(u_k)$ e t, temos $t \geq e(u_k) + k$, ou $t \geq e(v)$, contradição.

Lema 7 *Existe um escalonamento onde cada vértice é alocado no instante $2e(v)$.*

prova: Novamente por indução, na profundidade de v. Claramente, os vértices origem podem ser escalonados no instante $t = 0$. Para o passo de indução suponha primeiro que v tem $p \leq \tau + 1$ ancestrais. Então, todos podem ser escalonados, incluindo v até o instante $e(v) + 1 \leq 2e(v)$ no mesmo processador.

Finalmente, assuma que v tem $p > \tau + 1$ ancestrais. Ordene-os em ordem decrescente de $e()$: $e(u_1) \geq e(u_2) \geq e(u_3) \geq e(u_p)$. Como $e(v) = e(u_{\tau+1}) + \tau + 1)$, temos que $e(v) - \tau - 1 \geq e(u_j)$ para $j \geq \tau + 1$. Logo, por indução, todos os ancestrais de v exceto os τ primeiros podem ser escalonados até o instante $2e(v) - 2\tau - 2$. Isto implica que a partir do instante $2e(v) - \tau - 1$ os primeiros τ vértices podem ser escalonados (em ordem reversa), e portanto v pode ser escalonado no instante $2e(v)$.

Falar da generalização.

6.2 Número infinito de máquinas, sem duplicação

Neste caso, mesmo grafos de precedência simples correspondem a problemas $\mathcal{NP}-$difíceis.

Exemplo 24 *O problema $P_\infty|SCM; p_j, c_{jk}|C_{\max}$ é $\mathcal{NP}-$difícil.*
idéia da prova: *Redução do problema da partição $\{a_1, \ldots, a_n\}$ e $\sum a_i = 2B$, com o seguinte grafo de precedência:*

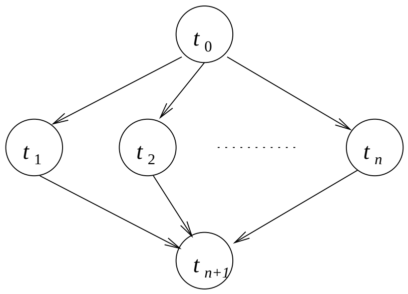


Figura 6.3: Grafo para a redução do problema da partição.

onde os tempos de execução de cada tarefa são dados abaixo e todas as comunicações valem C $(c_{jk} = C)$*:*

$p_0 = p_{n+1} = A$ *(inteiro)*
$p_j = a_j, j = 1, \ldots, n;$
$C \in \max\{B - a_{\min} + 1, \ldots, B - 1\}$, *onde* $a_{\min} = \min a_j$.

Existe uma partição do conjunto $\{a_1, \ldots, a_n\}$ *com soma* B *se e só se existe um escalonamento com makespan* $2A + C + B$. *Esta propriedade pode ser verificada observando-se a figura 6.4.*

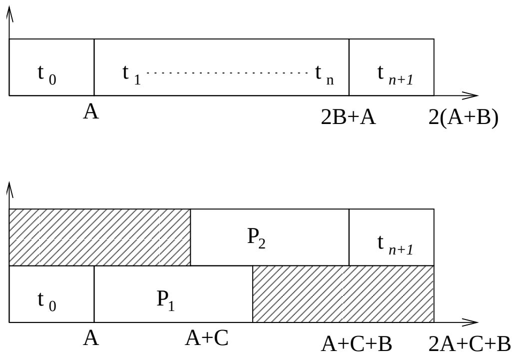


Figura 6.4: Melhor tempo de execução possível para o escalonamento em uma e duas máquinas.

Na figura 6.4, P_1, P_2 *denotam uma partição do conjunto* $\{a_1, \ldots, a_n\}$. *No caso do uso de mais máquinas 9m) o melhor caso corresponde a particionar o conjunto* $\{a_1, \ldots, a_n\}$ *em* m *conjuntos com o mesmo tamanho. Como o tempo de comunicação é grande, podemos ver que o menor makespan é obtido com duas máquinas (não é difícil de mostrar que o melhor makespan com* m *máquinas é* $2A + 2C + \max\{a_j, \frac{2B}{m}\}$*).*

Por outro lado, para grafos mais simples como o send (uma tarefa corresponde a raiz e todas as outras são nós folha) o problema é polinomial.

Vejamos a análise do seguinte exemplo: Sem perda de generalidade podemos supor que os vértices estão ordenados

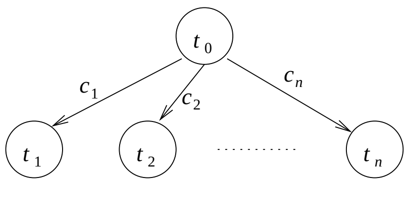


Figura 6.5: Grafo de precedência do tipo send.

de tal forma que $p_1 + c_{01} \geq p_2 + c_{02} \geq \ldots \geq p_n + c_{0n}$. Primeiramente podemos verificar que em um escalonamento ótimo somente o processador que executa a raiz tem que executar mais alguma tarefa. Em seguida podemos ver que se a tarefa t_j não é executada na mesma máquina que a tarefa t_0, as tarefas $\{t_{j+1}, \ldots, t_n\}$ também podem ser executadas cada uma em uma máquina, sem um aumento do makespan.

Observando estes dois fatos, o algoritmo tem apenas que encontrar a tarefa t_j que minimiza $\max\{\sum_{k=1}^{j} p_j, p_{j+1} + c_{0,j+1}\}$ para j entre 1 e n.

Exercício 8 *Encontre o escalonamento ótimo para o grafo send com os parâmetros abaixo:*

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
p_j	*1*	*1*	*2*	*2*	*3*	*2*	*4*	*2*	*2*
c_{0j}	*0*	*6*	*5*	*4*	*2*	*3*	*1*	*2*	*1*

Capítulo 7

Escalonamento Dinâmico

O escalonamento dinâmico ocorre quando alguma das condições do escalonamento muda com o tempo, por exemplo, o grafo de precedência pode não ser conhecido por completo, ou mesmo a alocação de uma tarefa pode sofrer alterações. O primeiro é bastante comum em computação paralela, quando tarefas sendo executadas podem criar subtarefas que até então não eram conhecidas até o momento. Esta área é bem recente, e a maioria dos resultados estão ligados ao escalonamento em computação paralela, logo vamos nos restringir a este caso.

Neste caso, existem duas possibilidades, ou todas as tarefas criadas são enviadas a um escalonador central, ou as mesmas são criadas localmente. No primeiro caso, cabe ao escalonador central a distribuição das novas tarefas. Isto geralmente é feito utilizando-se de algoritmos simples e rápidos. Os algoritmos que melhor se adaptam neste caso são os algoritmos de lista. Quando se trata de escalonamento dinâmico em máquinas paralelas não é interessante a procura pela melhor solução, pois a chegada de uma nova tarefa, que pode acontecer a qualquer instante, pode fazer com que a solução calculada já não seja a melhor possível.

Quando as tarefas são geradas localmente, pode ocorrer uma diferença de carga entre duas máquinas, por exemplo, uma muito carregada, e a outra ociosa. Nestes casos aplicam-se técnicas de balanceamento ou partilha de carga, onde tenta-se fazer uma redistribuição do trabalho durante a execução das mesmas. No caso de balanceamento procura-se fazer com que a quantidade de trabalho nas máquinas seja aproximadamente igual, ou seja que existe uma faixa de valores tal que a carga de cada uma das máquinas esteja nesta faixa. Com relação a partilha de carga o objetivo é mais simples, deseja-se garantir que nenhuma das máquinas fique ociosa enquanto existirem duas ou mais tarefas em outra máquina.

Novamente, podemos ter um controle de distribuição centralizado, que controla o nível de carga de cada máquina. Verifica os eventuais desbalanceamentos, e encontra as tarefas que se migradas vão corrigir as diferenças de carga. Ou podemos ter um controle distribuído, onde as máquinas monitoram as condições das máquinas vizinhas, e encontram eventuais diferenças de carga. Mesmo no controle distribuído temos duas possibilidades, as máquinas com mais carga podem transferir tarefas às máquinas com menos carga, ou mesmo as máquinas ociosas podem roubar tarefas das outras (*work stealing*.

Também existem duas opções com relação a migração, caso seja possível migrar uma tarefa sendo executada, interrompendo-a, temos a migração preemptiva. Caso contrário só tarefas prontas, mas ainda não alocadas podem ser migradas.

Mostrar em um exemplo a diferença das duas técnicas.

Um exemplo interessante de balanceamento dinâmico pode ser visto em Computação Móvel, onde aproveita-se de regiões de intersecção entre as ERBs para se fazer o balanceamento de carga de UMs.

7.1 Um pouco de teoria

Geralmente, os algoritmos de partilha de carga (mesmo os mais simples como os gulosos, ou de lista) permitem a obtenção de tempos de execução muito próximos ao ótimo, isto fica ainda mais evidente em programas com alto grau de paralelismo. Observe o teorema abaixo:

Teorema 7 *Seja uma máquina paralela com processadores idênticos sem custo de comunicação. Seja T_1 o tempo de execução seqüencial e T_∞ o tempo de execução em um número infinito de processadores.*

Todo escalonamento com partilha de carga em p processadores tem duração T_p limitada por:

$\max\{\frac{T_1}{p}, T_\infty\} \leq T_p \leq \frac{T_1}{p} + T_\infty$

Prova: a prova consiste em limitar o tempo inativo dos processadores pelo caminho crítico do grafo de precedência. Seja t uma tarefa que termina no makespan de T_p, e seja d a sua data de início de execução. Existem duas possibilidades, ou não existem processadores inativos antes de d (caso trivial), ou existem processadores inativos antes de d.

Neste segundo caso, seja θ a maior data com um processador inativo antes de d, como t não começou neste instante ela não estava pronta nesta data, logo um de seus predecessores estava sendo executado. Este raciocínio pode ser aplicado recursivamente até que ocorra o primeiro caso.

Seja o tempo de inatividade $I = pT_p - T_1$, temos que $I \leq (p-1)\sum_{j=1}^{n} p_j$, logo $pT_p \leq T_1 + (p-1)\sum_{j=1}^{n} p_j$.

Mas ao mesmo tempo T_∞ é maior ou igual a soma das tarefas no caminho crítico, logo:

$$T_p \leq \frac{T_1}{p} + T_\infty$$

O teorema anterior mostra que os algoritmos de partilha de carga fornecem uma execução eficaz, quando os tempos de comunicação não são considerados. Isto é, o tempo de inatividade está limitado por pT_∞, que é geralmente pequeno comparado ao tempo seqüencial. Este é um argumento muito forte em relação aos algoritmos de partilha de carga, que são mais simples e realizam menos migrações que os algoritmos de balanceamento de carga.

Em seguida, veremos mais dois resultados interessantes, o primeiro mostra que o uso de estratégias mais inteligentes de escalonamento, ou mesmo o uso de preempção não são muito úteis para a obtenção de algoritmos de escalonamento dinâmico mais eficientes. O segundo mostra que estratégias sem o conhecimento do grafo de precedência podem causar sobrecustos muito grandes no caso de máquinas com memória distribuída.

Teorema 8 *Se o tempo de comunicação não é considerado um limite inferior para o tempo de um algoritmo de escalonamento que não conhece a duração das tarefas é $2 - \frac{1}{p}$, mesmo quando a interrupção de tarefas é permitida.*

prova: Seja uma instância com uma tarefa de tamanho p e $p(p-1)$ tarefas com duração unitária. O melhor escalonamento é obtido colocando-se a tarefa grande em apenas um processador e as outras divididas entre os outros processadores. Por outro lado, sem conhecer a duração das tarefas, um escalonador pode escalonar as tarefas da pior forma possível, isto é, dividir as $p(p-1)$ tarefas entre os p processadores, e em seguida alocar a tarefa com tamanho p.

Observe o seguinte programa:

```
void f(int n) {
  if (n>1) {
    fork f(n/2);
    fork f(n/2);
  }
}
```

Este programa tem $O(n)$ operações, e gera uma árvore binária com profundidade $\log n$. Um escalonamento seqüencial vai ocupar espaço em memória proporcional a $\log n$ se a árvore é percorrida em profundidade. Já, se a árvore é percorrida por largura o gasto com memória será proporcional a n. A mesma discussão pode ser feita em máquinas paralelas.

Além disto, como pode existir dependência de dados entre programas, as vezes a migração apenas de uma tarefa pode não ser suficiente. Pode ser necessário efetuar a migração de todo um espaço de memória que corresponde aos dados (ou mesmo a pilha de execução) de uma tarefa. Este fator também deve ser considerado nos casos de migrações.

7.2 Alguns Ambientes de Escalonamento Dinâmico

Nesta seção veremos alguns dos ambientes para programação paralela que fazem alguma forma de escalonamento dinâmico. Os dois primeiros de balanceamento de carga e os seguintes de partilha de carga.

7.2.1 GTLB - Generic Threads Load Balancer

Corresponde a um núcleo balanceador de processos leves (threads) para máquinas com memória compartilhada, faz parte do ambiente PM^2 desenvolvido pela Universidade de Lille. Neste as tarefas se executam de forma assíncrona, sendo o sincronismo fornecido através de acessos a memória comum.

As tarefas criadas são imediatamente colocadas como prontas para serem executadas, e enviadas a um processador com menos tarefas. Como as tarefas podem ter durações diferentes (desconhecidas antes do término de sua execução), o ambiente também provê mecanismos de balanceamento de carga, onde um *deamon* local controla as diferenças de carga. Este *deamon* é executado concorrentemente com a aplicação. Mais especidficamente, cada máquina controla o número de tarefas em sua fila de execução, sem considerar os seus tamanhos. Quando o número de tarefas chega a um valor dado, esta informação é repassada a um vizinho (em uma topologia do tipo anel). No caso de diferenças importantes ocorre a migração, caso contrário a mensagem é repassada ao próximo vizinho.

7.2.2 DPC++ - Distributed Programming in C++

O DPC++ é um ambiente para proramação paralela em aglomerados desenvolvido na UFRGS. O ambiente permite a criação de objetos remotos em máquinas distantes. A troca de mensagens entre os objetos pode ser síncrona, quando o método retorna algo, ou assíncrona no caso de métodos `void`. Supõe-se que todos os objetos são `synchronized` para minimizar problemas de concorrência.

A distribuição de carga funciona a nível dos objetos, os quais são alocados de forma a não sobrecarregar alguma máquina da rede. Em caso de sobrecargas (dar exemplo) objetos também podem ser migrados.

O controle é feito por um nó central que armazena tabelas com os níveis de carga de cada máquina. A atualização destas tabelas é feita através do uso de objetos *espiões* que informam ao nó central eventuais mudanças de carga.

7.2.3 Cilk

Cilk é uma linguagem paralela baseada na linguagem C com suporte para primitivas do tipo *join/fork*. A criação de tarefas é explícita, e as mesmas acessam um espaço de memória comum.

Uma palavra chave *spawn* antes da chamada de uma função faz com que uma nova tarefa seja criada para executar esta função. Mas, a execução continua, sem interrupção na tarefa onde ocorreu a chamada. Para o uso de sincronizações deve ser usada a instrução *sync* que faz com que a função mãe espere o término das tarefas filhas iniciadas. Não existe nenhuma restrição com relação ao acesso concorrente à memória. Também é possível criar uma função *callback* a ser executada ao término das tarefas.

Cilk supõe o uso de memória compartilhada acessível por todos os processadores, mas versões mais novas oferecem suporte a máquinas com páginas de memória virtualmente compartilhada. Cilk é baseado em políticas de roubo de trabalho. Além disto, em cada máquina o escalonador usa uma política que prioriza a profundidade, isto é, a tarefa mãe sempre é bloqueada. Cada máquina mantém uma pilha com as tarefas prontas para serem executadas. Caso uma máquina fique inativa, a mesma "rouba" uma tarefa da base da pilha de outra máquina qualquer. Em máquinas com memória distribuída são usados mecanismos que procuram tarefas de memórias que estejam mais próximas, ou mais acessíveis.

7.2.4 Jade

Jade é uma extensão da linguagem C desenvolvida em Stanford. A linguagem C é expandida com a noção de bloco de instruções. Para cada bloco é necessário especificar os dados acessados e a forma de acesso. Para cada bloco encontrado é criada uma nova tarefa, onde devem ficar claros os dados de entrada e de saída, assim se cria um grafo de fluxo de dados. Existe uma garantia de que a execução paralela terá os mesmos resultados do que uma execução seqüêncial, para isto o ambiente gerência um grafo de precedência distribuído, logo é necessária uma forma de acesso a memória compartilhada para máquinas com memória distribuída.

As tarefas são criadas durante a execução do programa localmente, mas informações sobre as mesmas são enviadas a um escalonador central, quando as mesmas estão prontas para serem executadas. Quando um processador termina a execução de uma tarefa uma nova tarefa é escolhida, preferencialmente uma que dependa dos dados gerados pela tarefa recém terminada.

7.2.5 Análise

Todos os ambientes vistos possuem políticas de distribuição de carga extremamente simples, e rápidas. Vê se na prática o interesse por algoritmos simples, ao invés de algoritmos sofisticados, onde uma análise parcial mais cuidadosa pode não ser vantajosa.

Referências Bibliográficas

[1] nada ainda Solving sparse triangular systems on parallel computers. *International Journal of High Speed Computing*, 1:73–96, 1989.