Algoritmos de Aproximação para Problemas de Escalonamento em Máquinas

Eduardo Candido Xavier

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Algoritmos de Aproximação para Problemas de Escalonamento em Máquinas

Eduardo Candido Xavier[1]

Janeiro de 2003

Banca Examinadora:

- Prof. Dr. Flavio Keidi Miyazawa
  Instituto de Computação, Unicamp (Orientador)
- Prof. Dr. Arnaldo Vieira Moura
  Instituto de Computação, Unicamp
- Prof. Dr. Carlos Eduardo Ferreira
  Instituto de Matemática e Estatística, USP
- Prof. Dra. Yoshiko Wakabayashi
  Instituto de Matemática e Estatística, USP (Suplente)

[1] **Auxílio financeiro da FAPESP processo 01/04412-4 e do CNPq**

Substitua pela ficha catalográfica

Algoritmos de Aproximação para Problemas de Escalonamento em Máquinas

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Eduardo Candido Xavier e aprovada pela Banca Examinadora.

Campinas, 15 de fevereiro de 2003.

Prof. Dr. Flavio Keidi Miyazawa
Instituto de Computação, Unicamp (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Substitua pela folha com a assinatura da banca

Resumo

Neste trabalho estudamos diversos problemas de escalonamento considerados NP-difíceis. Assumindo a hipótese de que $P \neq \mathrm{NP}$, sabemos que não existem algoritmos eficientes para resolver tais problemas. Por isso houve um grande avanço no desenvolvimento de algoritmos de aproximação, que são algoritmos eficientes (complexidade polinomial) e que geram soluções com garantia de qualidade. Nos últimos anos, diversas técnicas surgiram para o desenvolvimento de algoritmos de aproximação como o método Primal-Dual e Programação Semidefinida. Neste trabalho, apresentamos um estudo de algumas das técnicas envolvidas no desenvolvimento de algoritmos de aproximação. Tais técnicas são exemplificadas com algoritmos de aproximação para problemas de escalonamento em máquinas. Também implementamos alguns dos algoritmos estudados e fazemos uma comparação prática entre eles. Além disso, propomos uma mudança em um dos algoritmos e mostramos que este obtém melhores resultados na prática. Apresentamos também algoritmos de aproximação para uma variação do problema da mochila. Tal problema tem aplicações práticas na indústria metalúrgica e ainda em problemas de escalonamento.

Abstract

In this work we study several scheduling problems that are NP-hard. If we consider that $P \neq \mathrm{NP}$, we know that there are no efficient algorithms to solve these problems. Because this, there were a lot of improvement in the field of approximation algorithms, that are efficient algorithms (polynomial complexity time) that produces solutions with quality guarantee. In the last years, several new approaches have been used in the development of approximation algorithms like the Primal-Dual method and Semidefinite Program. In this work, we study several techniques used in the development of approximation algorithms using scheduling problems. We implemented several studied scheduling algorithms and compare them in practice. We propose a modification in one of the algorithms and show that it produces solutions with better quality. We also present approximation algorithms to a generalized version of the knapsack problem. This problem appears in the metal industry and has applications in scheduling problems.

À minha mãe Dita e meu pai João.

Agradecimentos

Baseado na minha experiência pessoal, o que não representa estatisticamente um conjunto de amostras razoável, os agradecimentos são a vitrine das dissertações e teses. Quando passo pela sala do café, sempre dou uma olhada nas dissertações e teses do instituto e a primeira coisa que olho são os agradecimentos. Depois é que dou uma olhada no resumo para saber o que realmente foi feito no trabalho. Estou considerando a seguinte hipótese nestes agradecimentos: eu sou normal o suficiente para constituir um conjunto de amostras dos homens, o que pode ser bastante irreal mas me considero assim. Desta forma, tentarei fazer os agradecimentos parecerem bastante agradáveis, para que o leitor olhe pelo menos o resumo do que fiz. Para que a partir do resumo, o leitor leia o restante desta dissertação, é preciso um trabalho mais árduo, começando pela boa escrita da dissertação. Então é aqui que começa os agradecimentos. Agradeço ao meu orientador, o prof. Flavio Miyazawa, por ler e reler este trabalho, mostrando erros e dando idéias para melhora-lo. Agradeço à ele também pela ótima orientação que me deu nos últimos dois anos e principalmente à sua paciência.

Existem muitas pessoas que gostaria de agradecer, mas não me lembrarei de todas neste breve momento que tenho para escrever os agradecimentos. Se alguém ficar magoado por não ter seu nome aqui (o que não é la grande coisa), antecipadamente peço desculpas. Para diminuir o número de pessoas infelizes e para facilitar o meu trabalho, considero nestes agradecimentos a duplicação de nomes. Quando agradecer ao Fernando por exemplo, fique claro que estou agradecendo a todos os Fernandos que conheço, inclusive aos que venha conhecer.

Agradeço aos amigos de república, Lásaro, Flavinho, Flavão, Borin, e Lucien, por me aturarem e aguentarem minha chatice por tanto tempo.

Agradeço ao pessoal da banda, menos ao Keops (brincadeira), Gleison, Leandro Peludo e Chico.

Agradeço aos vários amigos e colegas do IC, Amanda, Alexandre, Baiano, Bartho, Bazinho, Chenca, Daniele, Eduardo, Evandro, Fernando, Fabio, Fileto, Guilherme, Gregorio, Gustavo, Guido, Henrique, Jamanta, Luiz, Luis, Leeiza, Magrão, Marcio, Marilia (Luke também), Michele, Nahri, Nielsen, Rodrigo Buzatinho, Ricardo, Silvania, Schubert, Thaisa, Wesley, Wanderley, Zeh e muitos outros que fizeram o meu mestrado muito mais divertido.

Agradeço aos amigos de Curitiba, Márcio, Davi, Celso, Angelo, Angela, Paulo Iguaçu e

outros, por fazerem minhas visitas a cidade mais saudosistas.

Agradeço ao pessoal do meu laboratótio, Glauber (pelas batucadas e músicas), Juliana Fofinha, Chenca (pelos agradecimentos) e Silvana. Principalmente a Silvana pelos maravilhosos momentos que me proporcionou e por ler algumas frases aleatórias do texto. Então há marcas dela neste trabalho também.

Agradeço aos meus professores da graduação na UFPR, em especial ao meu orientador de iniciação científica, Alexandre Direne (vulgo Alexd), por me ensinar como um sorriso é valioso e pelos dois anos e meio de orientação e também ao professor Renato Carmo, por ensinar tão bem teoria de computação, o que me ajudou a fazer um mestrado na área.

Agradeço também ao Hermes e Renato por me darem alguns momentos de descontração em casa e para provar a JuJu (agradecimentos à você também) que eu faria isto.

Agradeço aos meus irmãos Zinho, Sandre e Cesar por sempre terem me aturado.

Agradeço à minha querida mãe Dita e meu querido pai João, pela educação, preocupação, amor e paciência, que me ajudaram a chegar até aqui. Também agradeço ao meu pai pela revisão do texto e por finalmente me fazer aprender um pouco sobre o uso de vírgulas.

Agradeço finalmente à toda população brasileira, principalmente a mais carente, que contribuiu de forma involuntária com recursos financeiros para o desenvolvimento deste trabalho e agradeço por consequência a Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) e ao CNPq.

Sabiá que anta atrás de João de Barro vira servente de pedreiro.

(Autor deconhecido)

Encare o sol de frente e as sombras desabarão atrás de ti.

(Autor deconhecido)

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Neste trabalho apresentamos diversos algoritmos de aproximação voltados para problemas de escalonamento de tarefas. Muitas das variações de problemas de escalonamento são problemas de otimização que pertencem a classe NP-difícil. Problemas de otimização, na sua forma geral, têm como objetivo maximizar ou minimizar uma função definida sobre um certo domínio. A teoria clássica de otimização trata do caso em que o domínio é infinito. Já no caso dos chamados problemas de otimização combinatória, o domínio é tipicamente finito; além disso, em geral é fácil listar os seus elementos e também testar se um dado elemento pertence a esse domínio. Ainda assim, a idéia ingênua de testar todos os elementos deste domínio na busca pelo melhor mostra-se inviável na prática, mesmo para instâncias de tamanho moderado.

Neste trabalho, assumimos a hipótese de que $P \neq NP$. Desta forma, tais problemas de escalonamento e muitos outros problemas de otimização que são NP-difíceis, não possuem algoritmos eficientes para resolve-los de forma exata. Muitos destes problemas aparecem em aplicações práticas e há um forte apelo econômico para resolve-los. Problemas de escalonamento aparecem na alocação de tarefas em máquinas, alocação de registradores em códigos gerados por compiladores, na alocação de recursos em linhas de produção de indústrias, dentre outros. Neste trabalho, consideramos problemas de alocação de tarefas, que podem estar sujeitas a várias restrições, em máquinas de tal forma a minimizar alguma função de custo associada ao problema. Exemplo de casos envolvidos neste tipo de problema é a obtenção de escalonamentos de tarefas em computadores onde a média de atendimento de uma tarefa seja minimizada, ou que tarefas importantes tenham maior prioridade para serem finalizadas, ou mesmo a obtenção de um escalonamento que gaste tempo total mínimo.

Como não conseguimos resolver tais problemas de forma exata e eficiente, buscamos alternativas que possam ser úteis. Existem vários métodos que são muito utilizados na prática como o uso de heurísticas, programação inteira, métodos híbridos, redes neurais, algoritmos genéticos, dentre outros. Outra forma de resolução, é a utilização de algoritmos de aproximação. Neste caso, o algoritmo sacrifica a otimalidade em troca da garantia de uma solução

aproximada computável eficientemente. Certamente, o interesse é, apesar de sacrificar a otimalidade, faze-lo de forma que ainda possamos dar boas garantias sobre o valor da solução obtida, procurando ganhar o máximo em termos de eficiência computacional.

Em linhas gerais, algoritmos de aproximação são aqueles que não necessariamente produzem uma solução ótima, mas soluções que estão dentro de um certo fator da solução ótima. Esta garantia deve ser satisfeita para todas as instâncias do problema. Desta forma devemos dar uma demonstração formal deste fato.

1.1 Objetivos do Trabalho

O objetivo principal deste trabalho é estudar técnicas usadas no desenvolvimento de algoritmos de aproximação exemplificando-as com o uso de problemas de escalonamento. Além disso, estudamos o comportamento de alguns algoritmos aproximados na prática. Para tanto, implementamos alguns algoritmos e comparamos suas performances, tanto de tempo quanto de qualidade de soluções geradas. Também buscamos o desenvolvimento de novos algoritmos. Neste caso, apresentamos algoritmos para uma versão do problema da mochila que tem aplicações em problemas de escalonamento.

1.2 Organização do Texto

A organização do documento está baseada principalmente nos tipos de técnicas empregadas no desenvolvimento de algoritmos de aproximação. Antes de tudo, damos uma introdução a Algoritmos de Aproximação e falamos sobre o problema de escalonamento em máquinas (cap. 2).

Nos demais capítulos, apresentamos vários algoritmos de aproximação para problemas de escalonamento, exemplificando algumas das técnicas usadas na construção de algoritmos de aproximação, como métodos combinatórios (cap. 3), programação linear (cap. 4), métodos probabilísticos (cap. 5) e programação semidefinida (cap. 6).

É importante lembrar que existem inúmeros algoritmos de aproximação para problemas de escalonamento e não é nossa intenção mostrar todos os algoritmos estudados. Algoritmos de aproximação para problemas de escalonamento foram estudados desde a década de 60 [18]. Assim, buscamos colocar no texto algoritmos que achamos exemplificar bem uma dada técnica ou que mostram algum fato que julgamos importante.

O capítulo 7 é um resumo contendo os melhores resultados para problemas de escalonamento em máquinas. Este resumo contém os melhores e mais importantes resultados que pudemos encontrar durante o mestrado. Não esperamos que ele esteja completo e muito menos que contenha os resultados mais recentes para cada tipo de problema. A área de algoritmos de

aproximação é bastante ativa e novos resultados aparecem regularmente.

Os capítulos 8 e 9 são artigos que escrevemos. O primeiro artigo mostra uma comparação prática entre alguns algoritmos de escalonamento e o segundo são algoritmos aproximados propostos para uma versão generalizada do problema da mochila que tem aplicação em escalonamento de máquinas.

Finalmente, no capítulo 10 apresentamos as conclusões deste trabalho.

Capítulo 2

Preliminares

Este capítulo contém, de forma resumida, definições e noções básicas que serão necessárias no decorrer da leitura do trabalho. Apresentamos definições básicos sobre algoritmos de aproximação, dando definições relacionadas sobre o tema. Discutimos também brevemente técnicas usadas no desenvolvimento de algoritmos aproximados. Também falamos sobre problemas de escalonamento, mostrando termos usados neste trabalho bem como definições que utilizamos.

2.1 Algoritmos de Aproximação

Nesta seção damos uma breve visão sobre algoritmos de aproximação mostrando uma notação utilizada e também conceitos básicos sobre o tema.

Dado um algoritmo $\mathcal{A}$, para um problema de minimização, se I for uma instância para este problema, vamos denotar por $\mathcal{A}(I)$ o valor da solução devolvida pelo algoritmo $\mathcal{A}$ aplicado à instância I e vamos denotar por $\mathrm{OPT}(I)$ o correspondente valor para uma solução ótima de I. Diremos que um algoritmo tem um fator de aproximação α, ou é α-aproximado, se $\mathcal{A}(I)/\mathrm{OPT}(I) \leq \alpha$, para toda instância I. No caso dos algoritmos probabilísticos, consideramos a desigualdade $E[\mathcal{A}(I)]/\mathrm{OPT}(I) \leq \alpha$, onde a esperança $E[\mathcal{A}(I)]$ é tomada sobre todas as escolhas aleatórias feitas pelo algoritmo. É importante ressaltar que algoritmos de aproximação considerados neste trabalho têm complexidade de tempo polinomial.

Ao elaborar um algoritmo aproximado, o primeiro passo é buscar uma prova de seu fator de aproximação. Outro aspecto interessante é verificar se o fator de aproximação α demonstrado é o melhor possível. Para isto, devemos encontrar uma instância cuja razão entre a solução obtida pelo algoritmo e sua solução ótima é igual, ou tão próximo quanto se queira, de α. Neste caso, dizemos que o fator de aproximação do algoritmo é justo, ou seja, seu fator de aproximação não pode ser melhorado.

Nos últimos anos surgiram várias técnicas de caráter geral para o desenvolvimento de algoritmos de aproximação. Algumas destas são: *arredondamento de soluções via programação linear, dualidade em programação linear e método primal-dual, algoritmos probabilísticos e sua desaleatorização, programação semidefinida, provas verificáveis probabilisticamente e a impossibilidade de aproximações*, dentre outras (veja [13, 22, 37, 38]).

Uma estratégia comum para se tratar problemas de otimização combinatória é formular o problema através de um sistema de programação linear inteira e resolver a relaxação linear deste, uma vez que isto pode ser feito em tempo polinomial. O uso de programação linear tem sido usado para a obtenção de algoritmos aproximados através de diversas maneiras. Uma muito comum é o uso de arredondamentos das soluções fracionárias do programa linear. Outra técnica, é resolver o sistema dual do programa linear, em vez do primal, e em seguida obtemos uma solução com base nas variáveis duais. Outra técnica mais recente, é o uso do método de aproximação primal-dual, que tem sido usado para obter diversos algoritmos combinatórios usando a teoria de dualidade em programação linear. Neste caso, o método é em geral combinatório, não requerendo a resolução de programas lineares e consiste de uma generalização do método primal-dual tradicional.

Já no caso de algoritmos probabilísticos, o algoritmo contém passos que dependem de uma seqüência de bits aleatórios. Neste caso, a análise da solução gerada pelo algoritmo é calculada com base no valor esperado da solução. É interessante observar que apesar do modelo parecer restrito, a maioria dos algoritmos probabilísticos pode ser desaleatorizada, através do

método das esperanças condicionais, tornando-se algoritmos determinísticos (veja [13]). A versão probabilística é, em geral, mais simples de se implementar e mais fácil de se analisar que a correspondente versão determinística. Além disso, muitos dos algoritmos de aproximação combinam o uso de técnicas de programação linear com técnicas usadas em algoritmos probabilísticos, considerando o valor das variáveis obtidas pela relaxação linear como probabilidades. O uso destas técnicas tanto isoladamente como em conjunto tem sido usado nos últimos anos com sucesso em diversos problemas.

No caso da técnica de programação semidefinida temos um sistema quadrático, que se escrito em certas condições pode ser resolvido em tempo polinomial. A vantagem deste método é que muitos problemas podem ser representados através de modelos de programação semidefinida que muitas vezes nos leva a melhores limitantes. Goemans e Williansom [17] apresentaram uma forma bastante inovadora de se arredondar as soluções do sistema quadrático, através do arredondamento probabilístico, considerando cada uma das variáveis do sistema como um vetor na esfera unitária. No capítulo 7 apresentamos algoritmos para problemas de escalonamento de tarefas desenvolvido por Skutella que usam formulações baseadas em matrizes semidefinidas.

Do ponto de vista teórico, os algoritmos de aproximação mais desejados são aqueles que obtêm valores mais próximos do ótimo. Os algoritmos que encontram soluções com valor tão próximo quanto se queira de uma solução ótima, possuem uma denominação própia. Tais algoritmos têm fatores de aproximação $(1+\epsilon)$, no caso de problemas de minimização, e $(1-\epsilon)$, no caso de problemas de maximização, onde ϵ é uma constante positiva que pode ser tomada tão pequena quanto se queira. Chamamos de PTAS (Polynomial Time Approximation Scheme) os algoritmos que têm tais fatores de aproximação e têm tempo de execução polinomial na entrada. Chamamos de FPTAS (Fully Polynomial Time Approximation Scheme) os algoritmos que têm tais fatores de aproximação e têm tempo de execução polinomial na entrada e em $\frac{1}{\epsilon}$. Logo, dentre os dois tipos, os algoritmos mais desejados são os FPTAS.

Outro tópico importante em algoritmos de aproximação é inaproximalidade de problemas. Dado um certo problema Q, dizemos que este problema possui fator de inaproximalidade α, se não puder existir um algoritmo α-aproximado para Q. Uma das maneiras para se demonstrar tais resultados, é mostrar que se existir um algoritmo α-aproximado para um problema Q, então podemos resolver de maneira ótima em tempo polinomial um problema Q' que seja NP-difícil. Resultados importantes nesta área foram feitos com a utilização de provas verificáveis probabilisticamente, devido a Arora *et al.* [5, 6]. Para mais detalhes sobre resultados de inaproximalidade veja [4, 37].

2.2 Problemas de Escalonamento

Nesta seção definimos alguns conceitos relacionados a problemas de escalonamento e apresentamos algumas definições e notações que serão utilizadas no restante deste trabalho.

Problemas de escalonamento têm sido uma das principais áreas de pesquisa no desenvolvimento de algoritmos de aproximação. Vale dizer que é atribuído a um problema de escalonamento de tarefas em computadores paralelos o primeiro algoritmo de aproximação (veja Graham [18]).

Neste trabalho, usamos a seguinte notação. Temos um conjunto de tarefas $J = \{1, \ldots, n\}$ e um conjunto de máquinas $M = \{1, \ldots, m\}$. As tarefas a serem escalonadas são denotadas por j, e n denota o número de tarefas. As máquinas (processadores) são denotadas por i, $1 \leq i \leq m$ onde m é o número de máquinas.

Os atributos que uma tarefa $j \in J$ pode ter em um escalonamento são:

- *tempo de processamento*, denotado por p_{ij}, que depende da máquina i onde a tarefa j será processada. No caso especial onde temos apenas uma máquina, ou todas as máquinas são idênticas, denotamos a requisição de processamento apenas por p_j. A tarefa j deve ser processada por um respectivo período de tempo em uma das m máquinas. Pode ocorrer que $p_{ij} = \infty$, indicando que a tarefa j não poderá ser executada na máquina i;

- *tempo de término*, denotado por C_j, que indica o tempo em que a tarefa j é completada;

- *peso da tarefa*, denotado por w_j, que indica a prioridade que a tarefa tem para ser terminada;

- *tempo de liberação* , denotado por r_j, antes do qual a tarefa não pode ser processada.

Nesta dissertação assumimos, a não ser que seja dito o contrário, que todos os valores descritos são inteiros não negativos.

Outra condição comum em escalonamentos é a precedência entre tarefas. Denotamos por $j \prec k$ se a tarefa j deve ser completada antes da tarefa k começar.

Dado um conjunto de tarefas $J = \{1, \ldots, n\}$ e um conjunto de máquinas $M = \{1, \ldots, m\}$, definimos um *escalonamento não-preemptivo* como uma atribuição de cada tarefa $j \in J$ para um par máquina-tempo (i, t), indicando que a tarefa j é executada na máquina i e inicia sua execução no tempo t. Dado uma tarefa j e sua atribuição (i, t), um escalonamento não-preemptivo deve satisfazer a restrição de que nenhuma outra tarefa k pode ser atribuída para um par (i, t') com $t' \in [t, t + p_{ij})$. Definimos um *escalonamento preemptivo* como uma atribuição de cada tarefa j a um conjunto de triplas $Y = \{(i_1, t_1, f_1), \ldots, (i_x, t_x, f_x)\}$. Uma tripla (i, t, f) representa a máquina que j executa, o seu tempo de início e fim respectivamente. Dado uma tarefa j e seu conjunto de atribuições Y, o escalonamento preemptivo deve satisfazer a restrição de que

nenhuma tarefa k pode ser atribuída para uma tripla (i', t', f') tal que exista par $(i', t, f) \in Y$ com $[t', f') \cap [t, f) \neq \emptyset$ e ainda

$$\sum_{l=1}^{x} \frac{f_l - t_l}{p_{i_l j}} = 1.$$

Desta forma, um escalonamento pode ser preemptivo ou não. De maneira simplificada, escalonamentos preemptivos são aqueles que uma tarefa pode ser repetidamente interrompida e continuada depois, na mesma ou em outra máquina. Escalonamentos não-preemptivos são aqueles que uma tarefa deve ser processada de maneira ininterrupta.

Definimos o tempo de término de um escalonamento, o tempo de término da última tarefa a ser completada. Denotamos o tempo de término do escalonamento por C_{max}. Desta forma temos que $C_{max} = \max\{C_j, \quad j \in J\}$.

Como nem todas as condições acima precisam estar presentes em um problema de escalonamento, Graham, Lawler, Lenstra e Rinnooy Kan [19] (veja também [27]) apresentaram um esquema de classificação para estes problemas denotado pela tripla $\alpha|\beta|\gamma$. A seguir apresentamos os valores de α, β e γ para os problemas de nosso interesse:

- O termo α é a característica da máquina que pode ser 1, P ou R. Quando temos apenas uma máquina usamos $\alpha = 1$. Quando temos várias máquinas paralelas idênticas usamos $\alpha = P$. O caso geral onde as máquinas não têm nenhuma relação entre si, é denotado com $\alpha = R$. Junto com as letras P e R pode-se colocar o número de máquinas no ambiente de escalonamento. Assim, $P2$ indica que temos duas máquinas idênticas em paralelo.

- O termo β pode ser vazio ou conter as características das tarefas: r_j (ou r_{ij}), *prec* e *pmtn*. A inclusão do tipo r_j indica que as tarefas têm tempo de liberação, *prec* indica que as tarefas podem ter precedência entre elas e *pmtn* indica que o escalonamento pode ser preemptivo.

- O termo γ refere-se à função objetivo. Caso tenhamos $\gamma = \sum w_j C_j$ estamos minimizando o tempo de finalização ponderado das tarefas. Quando temos $\gamma = C_{\max}$ o objetivo é minimizar o tempo máximo para completar todas as tarefas (*makespan*). Note que este último pode ser reduzido para o caso onde temos precedência e pesos. Para isso, basta colocar todos os pesos iguais a 0 e inserir uma nova tarefa com peso unitário e que sucede todas as demais tarefas.

Uma distinção feita em relação a um algoritmo para escalonamento é se ele é *on-line* ou *off-line*. Um algoritmo *off-line* tem como entrada todos os dados (p_j, r_j, etc...) relativos ao conjunto de tarefas. Neste caso o algoritmo tem previamente estes dados e constrói o escalonamento a partir deles. Já um algoritmo *on-line* constrói um escalonamento a medida que o tempo passa e que novas tarefas são liberadas. Se uma tarefa j é liberada no tempo t, então o algoritmo só

toma conhecimento dos dados relativos a j no tempo t. No tempo t o algoritmo só possui dados das tarefas k para as quais $r_k \leq t$.

Um tipo de escalonamento que vem sendo estudado recentemente é o caso em que as tarefas podem executar em vários processadores ao mesmo tempo. Este tipo de escalonamento é chamado *escalonamento em multiprocessadores*. Cada tarefa j está associada a uma função $p_j(S)$ onde $S \subseteq M$, de tal forma que o tempo de processamento de j está em função do subconjunto de processadores onde ela será executada. Este problema é chamado de *escalonamento maleável* e o termo β no esquema de classificação conterá o indicador set. Um escalonamento é *não maleável* se o conjunto de máquinas nas quais a tarefa irá executar está fixo. Desta forma, a tarefa tem tempo de processamento p_j e subconjunto de processadores τ_j fixo onde será executada. Para este problema o termo β conterá o indicador fix_j.

Nas próximas seções, quando estivermos descrevendo os algoritmos, consideramos que estes recebem como parâmetros o conjunto J de tarefas e o conjunto M de máquinas. Dado uma tarefa j, sempre consideramos que os atributos relacionados a esta tarefa, como r_j, p_j, C_j, são inerentes a tarefa j. Se um algoritmo recebe como parâmetro uma tarefa j, estamos consequentemente recebendo os dados relativos a esta tarefa.

Dado um conjunto de máquinas $M = \{1, \ldots, m\}$, o escalonamento não preemptivo retornado por um algoritmo será uma sequência de listas $M_1, \ldots, M_m$. Cada lista M_i contém uma sequência de tarefas $M_i = (j_{i_1}, \ldots, j_{i_l})$ indicando que o escalonamento deve executar as tarefas $j_{i_1}, \ldots, j_{i_l}$ nesta ordem na máquina i, sempre respeitando os tempos de liberação das tarefas. Além disso, consideramos que as listas estão vazias no início do algoritmo. Desta forma não colocamos os passos para iniciar valores destas listas.

Dado um escalonamento gerado por algum algoritmo, assumimos nas próximas seções, que o valor deste escalonamento representa o valor da função objetivo que estamos interessados em minimizar. Desta forma, nos referimos sempre ao valor do escalonamento sem explicitar qual é o objetivo de minimização do escalonamento. Além disso, denotamos o valor de um escalonamento ótimo por OPT.

Capítulo 3

Algoritmos Combinatórios

A utilização de algoritmos puramente combinatórios é a primeira que nos vem em mente quando tentamos resolver um problema. Não existem regras básicas para o desenvolvimento de algoritmos de aproximação através de algoritmos combinatórios, a não ser a utilização das técnicas básicas já conhecidas como divisão e conquista, algoritmos gulosos, programação dinâmica, etc. É claro que não estamos restritos a estas técnicas quando tentamos resolver qualquer problema e algoritmos de aproximação também não devem ficar restritos. O importante em algoritmos de aproximação é a busca de limitantes para a solução ótima. Sempre devemos ter em mente que é necessário uma maneira de se comparar o valor das soluções geradas pelo nosso algoritmo com o valor de uma solução ótima, por isso a importância de limitantes bons. Limitantes bons são aqueles que têm valores próximos do ótimo. Se estamos querendo minimizar o tempo de término de um escalonamento por exemplo, um limitante para o ótimo é o maior tempo de processamento de uma tarefa. Note que fatores de aproximação baseados unicamente neste limitante podem não ser bons. Considere por exemplo, o escalonamento de n tarefas com mesmo tempo de processamento em uma única máquina. Claramente qualquer escalonamento gerado deve gastar pelo menos n vezes o tempo da tarefa mais longa. Os algoritmos propostos devem, de alguma forma, gerar soluções que fazem uso de estruturas do problema que possam ser comparadas com o ótimo. Por estruturas do problema queremos dizer dados das instâncias do problema que possam ser obtidas facilmente e que são comparáveis ao ótimo. Um exemplo é o maior tempo de processamento de uma tarefa como vimos a pouco. Nas seções seguintes daremos exemplos de algoritmos combinatórios e como o fator de aproximação é calculado através de limitantes.

3.1 Algoritmo de Graham para $P||C_{max}$

O algoritmo dado nesta seção foi proposto por Graham [18] em 1966 e possui caráter histórico, pois foi o primeiro algoritmo conhecido a ter uma prova de fator de aproximação. O algoritmo trata do caso $P||C_{max}$, onde temos um conjunto de máquinas idênticas e nosso objetivo é minimizar o tempo de término do escalonamento.

O algoritmo de Graham tem como entrada o conjunto de tarefas J e o conjunto de máquinas M. Cada tarefa j possui tempo de processamento $p_j > 0$ que pertence aos racionais. Temos que encontrar uma partição $\{M_1, \ldots, M_m\}$ das tarefas $\{1, \ldots, n\}$ que minimize $\max_i(t(M_i))$ onde $t(M_i) = \sum_{j \in M_i} p_j$, ou seja, minimize o tempo de término do escalonamento.

3.1.1 O Algoritmo

O algoritmo proposto por Graham baseia-se em uma idéia muito simples: aloque as tarefas uma a uma, destinando cada tarefa à máquina menos ocupada. Caso tenhamos mais de uma máquina para alocação de uma tarefa, o algoritmo atribui a tarefa de forma arbitrária à uma das máquinas. Por esse critério, a escolha da máquina que irá receber determinada tarefa não depende dos tempos das tarefas que ainda não foram atribuídas a nenhuma máquina. Na figura 3.1 apresentamos o algoritmo que denotamos por *GH*.

ALGORITMO GH(J, M)

1. para i de 1 a m faça
2. $\quad M_i \leftarrow \emptyset$
3. para j de 1 a n faça
4. $\quad$ seja k uma máquina tal que $t(M_k)$ é mínimo
5. $\quad M_k \leftarrow M_k||j$
6. devolva $M_1, \ldots, M_m$

Figura 3.1: Algoritmo de Graham.

Como exemplo, considere a entrada ($\{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 10\}$,2). Na figura 3.2 apresentamos o escalonamento que o algoritmo gera.

M1	1	1	1	1	1	10
M2	1	1	1	1	1	

Figura 3.2: Escalonamento gerado.

O escalonamento gerado pelo algoritmo *GH* tem tempo de término igual a 15 enquanto o escalonamento ótimo tem tempo de término igual a 10 e pode ser visto na figura 3.3.

M1	1	1	1	1	1	1	1	1	1	1
M2	10									

Figura 3.3: Escalonamento ótimo.

Para esta instância, o algoritmo *GH* tem uma aproximação de $\frac{15}{10} = 1{,}5$. Graham foi o precursor da área de algoritmos de aproximação, pois propôs este algoritmo juntamente com uma prova de seu fator de aproximação.

3.1.2 Razão de Aproximação

Nesta seção mostramos o fator de aproximação do algoritmo *GH*. Vamos mostrar como obter o limite de aproximação através da analise de limitantes inferiores. O próximo teorema mostra o resultado.

Teorema 3.1.1 *O algoritmo GH é uma* $(2 - \frac{1}{m})$*-aproximação para o problema* $P||C_{max}$.

Prova. Podemos perceber que ao final da execução do algoritmo, ele produzirá uma partição $\{M_1, \ldots, M_m\}$ de $\{1, \ldots, n\}$, ou seja, um escalonamento viável. Isto ocorre já que todas tarefas serão escalonados.

Vamos achar agora limitantes inferiores para o ótimo. Sabemos que

- $OPT(m, n, t) \geq p_{max} = \max_j p_j$ e que
- $OPT(m, n, t) \geq \frac{1}{m} \sum_{j=1}^{n} p_j$.

A primeira desigualdade vem do fato de que temos que executar a maior tarefa. A segunda desigualdade vem do fato de que se pudéssemos dividir e atribuir o processamento total em partes iguais para cada uma das máquinas, este nos daria o menor C_{max} possível.

Seja τ o valor do escalonamento gerado pelo algoritmo *GH* ao final de uma iteração qualquer. Seja w a tarefa que foi atribuída à máquina k nesta iteração. É claro que $t(M_k)$ é mínimo antes da atribuição da tarefa w.

Vamos mostrar que $\tau - p_{max} \leq t(M_k)$ antes da atribuição de w. Seja f a máquina tal que $t(M_f) = \tau$. Se $f = k$ então vale que $\tau - p_{max} \leq t(M_k)$ antes da atribuição de w. Caso contrário, seja y a última tarefa processada na máquina f. Também é válido que $\tau - p_{max} \leq t(M_f) - p_y \leq t(M_k)$ antes da atribuição de w. Portanto, nenhuma máquina pode estar desocupada no tempo $\tau - p_{max}$. Logo vale a desigualdade

$$\sum_{j=1}^{n} p_j \geq m(\tau - p_{max}) + p_{max}.$$

Consequentemente temos

$$OPT \geq \frac{1}{m}\sum_{j=1}^{n} p_j \geq \tau - (\frac{m-1}{m})p_{max}. \tag{3.1}$$

Trocando τ na desigualdade (3.1) obtemos

$$\tau \leq OPT + (\frac{m-1}{m})p_{max}.$$

Como $p_{max} \leq OPT$ a seguinte desigualdade é valida

$$\tau \leq (1 + \frac{m-1}{m})OPT = (2 - \frac{1}{m})OPT.$$

A desigualdade acima é válida para o escalonamento gerado no fim de cada iteração, portanto também é válida para o último escalonamento gerado. $\square$

3.2 Um PTAS para o problema $P||C_{max}$

O algoritmo desta seção, proposto por Hochbaum e Shmoys [23], mostra o forte relacionamento entre problemas de empacotamento e escalonamento. O problema $P||C_{max}$ é fortemente relacionado com o problema de empacotamento unidimensional que pode ser definido como:

- Dados recipientes de tamanho t e n itens $\{a_1, \ldots, a_n\}$, cada item a_i com tamanho p_{a_i}, devemos empacotar estes n itens no menor número de recipientes possível. O empacotamento deve satisfazer a restrição:

  1. Dado um recipiente R e um conjunto $\{x_1, \ldots, x_k\}$ de itens empacotados em R, deve-se respeitar $\sum_{i=1}^{k} p_{x_i} \leq |R|$, onde $|R|$ é o tamanho do recipiente R.

Considere uma instância do problema $P||C_{max}$ onde temos n tarefas com requerimento de processamento $\{p_1, \ldots, p_n\}$ que devem ser executadas em m máquinas. Note que existe um escalonamento com tempo de término $C_{max} = t$ se, e somente se, podemos empacotar n itens de tamanho $\{p_1, \ldots, p_n\}$ em m recipientes de capacidade t. Dado uma instância $I = \{p_1, \ldots, p_n\}$, seja $bins(I, t)$ o menor número de recipientes de capacidade t necessários para empacotar I. O menor tempo de término C_{max} para o problema $P||C_{max}$ será dado por: $\min\{t : bins(I, t) \leq m\}$.

Seja $LB = \max\{\frac{1}{m}\sum_{i=1}^{n} p_i, p_{max}\}$. Como vimos na seção anterior, LB é um limitante inferior e $2LB$ é um limitante superior para o ótimo do problema $P||C_{max}$. Podemos determinar o menor C_{max} com uma busca binária neste intervalo com a restrição de que $bins(I, C_{max}) \leq m$. Na seção seguinte apresentamos um algoritmo polinomial para um caso particular do problema de empacotamento unidimensional.

3.2.1 Algoritmo para Empacotamento em Recipientes

Nesta seção apresentamos um algoritmo polinomial ótimo para resolver instâncias do problema de empacotamento em recipientes de tamanho t, quando o número de tamanhos diferentes dos itens é limitado por uma constante k. Uma instância deste problema, pode ser definida por uma k-tupla $(i_1, \ldots, i_k)$ especificando quantos objetos de cada tamanho existem. Assim, vamos supor que o algoritmo recebe um conjunto de itens I e gera a partir deste conjunto, a tupla representante. Seja $BINS(t_1, \ldots, t_k)$ o menor número de recipientes necessários para empacotar os itens representados pela tupla $(t_1, \ldots, t_k)$. O algoritmo, denotado por *PackHS*, recebe como parâmetro os itens e um valor especificando o tamanho dos recipientes. O algoritmo pode ser visto na figura 3.4.

Dado a tupla $(n_1, \ldots, n_k)$, o algoritmo primeiramente calcula o conjunto Q dado por todas k-tuplas $(q_1, \ldots, q_k)$ tal que $BINS(q_1, \ldots, q_k) = 1$ e $0 \leq q_i \leq n_i, 1 \leq i \leq k$. Claramente

ALGORITMO PackHS(I,t)

1. particione $I = I_1 \cup \ldots \cup I_k$ tal que itens de I_i são todos do mesmo tamanho, $1 \leq i \leq k$
2. gere tupla $(n_1, \ldots, n_k)$ tal que $n_i = |I_i|$, $1 \leq i \leq k$
3. gere tupla $(s_1, \ldots, s_k)$ tal que $s_i = p_x$, $x \in I_i$, $1 \leq i \leq k$
4. $Q \leftarrow \emptyset$
5. para cada tupla $(q_1, \ldots, q_k)$ tal que $0 \leq q_i \leq n_i, 1 \leq i \leq k$, faça
6. se $\sum_{i=1}^{k} s_i \cdot q_i \leq 1$ então
7. $Q \leftarrow Q \cup \{(q_1, \ldots, q_k)\}$
8. $BINS((q_1, \ldots, q_k)) \leftarrow 1$
9. para cada tupla $(i_1, \ldots, i_k)$; $0 \leq i_i \leq n_i, 1 \leq i \leq k$ faça
10. $BINS(i_1, \ldots, i_k) \leftarrow 1 + \min\{BINS(i_1 - q_1, \ldots, i_k - q_k) : \quad \forall \quad (q_1, \ldots, q_k) \in Q\}$
11. retorne $BINS$

Figura 3.4: Algoritmo para empacotar itens no menor número de recipientes.

o conjunto Q possui menos que n^k elementos. Em seguida o algoritmo calcula entradas de uma tabela k-dimensional. Cada entrada da tabela é definida por $BINS(i_1, \ldots, i_k)$ para todo $(i_1, \ldots, i_k) \in \{0, \ldots, n_1\} \times \{0, \ldots, n_2\} \times \ldots \times \{0, \ldots, n_k\}$. Cada entrada $(i_1, \ldots, i_k)$ da tabela representa o menor número de recipientes necessários para empacotar estes itens. A tabela é iniciada com $BINS(q) = 1$ para cada $q \in Q$. As demais entradas da tabela são calculadas usando-se a seguinte recorrência:

$$BINS(i_1, \ldots, i_k) = 1 + \min\{BINS(i_1 - q_1, \ldots, i_k - q_k) : \quad \forall \quad (q_1, \ldots, q_k) \in Q\}$$

O algoritmo pode ser implementado de forma a calcular cada entrada da tabela com complexidade de tempo $O(n^k)$ e a tabela inteira com complexidade de tempo $O(n^{2k})$. Vamos supor nas seções seguintes, que junto com a função $BINS$, o algoritmo devolve um empacotamento dos itens.

3.2.2 O Algoritmo para o Problema $P||C_{max}$

Nesta seção apresentamos o algoritmo que gera o escalonamento utilizando o algoritmo *PackHS* da seção anterior. Para tanto, temos que limitar o número de diferentes tempos de processamentos arredondando-os.

Seja ϵ um número positivo e $t \in [LB, 2LB]$. Na figura 3.5 apresentamos um algoritmo, denotado por *Round*, que arredonta os tempos de processamento das tarefas a serem escalonadas, de tal forma a ter um número constante de tempos de processamento diferentes.

O algoritmo devolve dois conjuntos, $R1$ e $R2$. O conjunto $R2$ tem as tarefas consideradas pequenas. Dizemos que uma tarefa é pequena se seu tamanho é menor do que ϵt. O conjunto

ALGORITMO Round(J,ϵ,t)

1. para cada $j \in J$ faça
2. $R1 \leftarrow \emptyset$
3. $R2 \leftarrow \emptyset$
4. se $p_j \geq \epsilon t$ então
5. seja i o inteiro tal que $t\epsilon(1+\epsilon)^i \leq p_j < t\epsilon(1+\epsilon)^{i+1}$
6. $j' = j$
7. $p_{j'} \leftarrow t\epsilon(1+\epsilon)^i$
8. $R1 \leftarrow R1 \cup \{j'\}$
9. senão
10. $R2 \leftarrow R2 \cup \{j\}$
11. retorne $R1$ e $R2$

Figura 3.5: Algoritmo para arredondar tamanho dos itens.

$R1$ contém as tarefas consideradas grandes. Estas tarefas têm seu tempo de processamento arredondado da seguinte forma: cada p_j no intervalo $[t\epsilon(1+\epsilon)^i, t\epsilon(1+\epsilon)^{i+1})$ é trocado por $p'_j = t\epsilon(1+\epsilon)^i$ para $i \geq 0$. Os p'_j criados podem assumir no máximo $k = \lceil \log_{1+\epsilon} \frac{1}{\epsilon} \rceil$ valores distintos. Com isso, podemos determinar um empacotamento ótimo para estes itens usando o algoritmo *PackHS*. Como o arredondamento reduz o tamanho de cada item por um fator de no máximo $1+\epsilon$, se considerarmos o seus tamanhos originais, o empacotamento será válido para recipientes de tamanho $t(1+\epsilon)$. Consideramos as tarefas como objetos a serem empacotados.

Na figura 3.6 apresentamos um algoritmo, denotado por *PackHS2*, que empacota as tarefas do conjunto J em recipientes de tamanho $t(1+\epsilon)$.

Todos os itens são empacotados em recipientes de tamanho $t(1+\epsilon)$. Os itens grandes do conjunto $R1$ são empacotados de forma ótima pelo algoritmo *PackHS*. O algoritmo então empacota os objetos pequenos de maneira gulosa nos espaços que possivelmente sobraram. Só é utilizado um novo recipiente se todos os demais estiverem cheios por pelo menos t.

Vamos denotar por $\alpha(J,t,\epsilon)$ o número de recipientes usados pelo algoritmo *PackHS2* para empacotar todos os itens. O lema abaixo mostra que o número de recipientes usados pelo algoritmo *PackHS2*, para empacotar todas as tarefas, não é maior do que o número de recipientes usados por uma solução ótima. Vale lembrar que no algoritmo é utilizado recipientes de tamanho $t(1+\epsilon)$ enquanto a solução ótima que empacota estes itens usa recipientes de tamanho t.

Lema 3.2.1 *O número de recipientes usados pelo algoritmo PackHS2 para empacotar uma instância (J,t,ϵ) é no máximo o número de recipientes usados em uma solução ótima, ou seja, $\alpha(J,t,\epsilon) \leq bins(J,t)$.*

Prova.

```
ALGORITMO PackHS2(J,ϵ,t)
Round(J,ϵ,t)
PackHS(R1,t)
seja M = {1, ..., l}, o conjunto de recipientes com empacotamento de R1
para cada j ∈ R2 faça
    para i ← 1 até l faça
        se j pode ser empacotado em i então
            M_i ← M_i||j
            break
    se j não foi empacotado então
        l ← l + 1
        crie novo recipiente l
        M_l ← M_l||j
retorne M_1, ..., M_l
```

Figura 3.6: Algoritmo para empacotar todos os itens (tarefas).

Os itens grandes são empacotados de maneira ótima pelo algoritmo *PackHS*. Se todos os itens pequenos puderem ser empacotados sem utilizar nenhum novo recipiente, então o lema vale já que os itens grandes foram empacotados de maneira ótima. Se o algoritmo *PackHS2* precisar criar novos recipientes, é porque todos os demais estão ocupados com pelo menos t de tamanho, já que os recipientes tem tamanho $t(1+\epsilon)$ e todos os itens pequenos tem tamanho menor que $t\epsilon$. Logo, todos os recipientes estão ocupados com pelo menos t com exceção do último recipiente criado. Como $bins(J,t)$ considera recipientes de tamanho t, este terá que usar pelo menos a mesma quantidade de recipientes.

□

Como o valor de um escalonamento ótimo para o problema $P||C_{max}$ é dado por $OPT = \min\{t : bins(J,t) \leq m\}$, temos, aplicando o lema 3.2.1, que $\min\{t : \alpha(J,t,\epsilon) \leq m\} \leq OPT$. Temos que achar então o menor valor de t tal que $\alpha(J,t,\epsilon) \leq m$. O algoritmo da figura 3.7, denotado por *HS*, gera o escalonamento para o problema $P||C_{max}$.

O algoritmo faz uma busca binária com t no intervalo $[LB, 2LB]$ até que a busca diminua para um intervalo de tamanho ϵLB. Para cada valor t, é gerado um empacotamento com o algoritmo *PackHS2*. A resposta do algoritmo *HS* é o empacotamento com o menor t tal que $\alpha(J,t,\epsilon) \leq m$. O tamanho deste empacotamento é dado por t_{min}.

Na busca binária, o algoritmo considera o intervalo $[LB, 2LB]$ dividido em intervalos discretos de tamanho ϵLB. Existem $\frac{1}{\epsilon}$ intervalos de tamanho ϵLB entre LB e $2LB$. Logo, o algoritmo faz uma busca binária em $\frac{1}{\epsilon}$ intervalos e isto é feito com $\lceil \log_2 \frac{1}{\epsilon} \rceil$ iterações. Seja T o ponto mais a direita do intervalo em que o algoritmo termina a busca, ou seja, $T = MAX$

ALGORITMO HS(J,M,ϵ)

1. $MIN \leftarrow LB$
2. $MAX \leftarrow 2LB$
3. enquanto $MAX - MIN > \epsilon LB$ faça
4. $\quad t \leftarrow \frac{MAX-MIN}{2} + MIN$
5. $\quad$ PackHS2(J, ϵ, t)
6. $\quad$ se $\alpha(J, t, \epsilon) \leq m$ então
7. $\quad\quad MAX \leftarrow t$
8. $\quad\quad t_{min} \leftarrow t$
9. $\quad$ senão
10. $\quad\quad MIN \leftarrow t$
11. retorne t_{min} e empacotamento $M_1, \ldots, M_m$ gerado por PackHS2

Figura 3.7: Algoritmo que gera o escalonamento das tarefas.

quando o algoritmo termina. O lema abaixo mostra um limite para o valor de t_{min} obtido na busca binária.

Lema 3.2.2 *Seja T o ponto mais a direita no intervalo em que o algoritmo encerra a busca binária. Vale que $T \leq (1+\epsilon)OPT$.*

Prova.

O algoritmo faz uma busca binária até que o intervalo de busca diminua para um tamanho de ϵLB, ou seja até termos $MAX - MIN < \epsilon LB$. Sabemos que $\min\{t : \alpha(J, t, \epsilon) \leq m\}$ está no intervalo $[MIN, MAX]$ e consequentemente no intervalo $[T - \epsilon LB, T]$. Logo temos

$T \leq \min\{t : \alpha(J, t, \epsilon) \leq m\} + \epsilon LB$.

Como LB é um limitante inferior do ótimo, temos que $T \leq (1+\epsilon)OPT$.

□

Com o limitante obtido no lema 3.2.2, podemos mostrar que o algoritmo *HS* é um PTAS para o problema $P||C_{max}$. Note que o algoritmo não é um FPTAS porque o algoritmo *PackHS* tem complexidade de tempo $O(n^{2\lceil \log_{1+\epsilon} \frac{1}{\epsilon} \rceil})$

Teorema 3.2.3 *Dado uma instância (J, M) para o problema $P||C_{max}$ e um ϵ tal que $1 > \epsilon > 0$, o algoritmo HS produz um escalonamento tendo tempo de término no máximo $(1+3\epsilon)OPT$.*

Prova.

O algoritmo acha um escalonamento de tamanho no máximo $(1+\epsilon)T$ já que o algoritmo *PackHS2* empacota as tarefas em recipientes de tamanho no máximo $(1+\epsilon)T$. Como $T \leq$

$(1+\epsilon)OPT$ pelo lema 3.2.2, temos um limite para o tamanho do escalonamento de $(1+\epsilon)^2 OPT$. Mas $(1+\epsilon)^2 = 1 + 2\epsilon + \epsilon^2 \leq (1 + 3\epsilon)$, já que $1 > \epsilon$.

$\square$

3.3 Uma 2-aproximação para $1|r_j|\sum C_j$

O algoritmo desta seção, proposto por Phillips, Stein e Wein [30], é genérico podendo ser aplicado para transformações gerais de escalonamentos preemptivos para não preemptivos. A forma apresentada aqui, será especifica para o problema $1|r_j|\sum C_j$, onde deve-se montar um escalonamento em uma única máquina minimizando a soma dos tempos de término das tarefas. Cada tarefa j possui um tempo de liberação r_j, antes do qual não pode ser processada. No final desta seção, mostramos que o fator de aproximação deste algoritmo é justo. O algoritmo, que denotamos por *PSW*, é apresentado na figura 3.8. O algoritmo usa uma função *first*, que retorna o primeiro elemento de uma lista.

ALGORITMO PSW(J)

1. seja S um escalonamento ótimo para o problema $1|r_j, pmtn|C_j$ com as tarefas J
2. seja C_j^p o tempo de término de cada tarefa j no escalonamento S
3. seja L uma lista com tarefas de J ordenadas pelos respectivos valores C_j^p
4. enquanto $L \neq \emptyset$ faça
5. $\quad j \leftarrow$ *first*(L)
6. $\quad L \leftarrow L \setminus \{j\}$
7. $\quad$ escalone j de forma não preemptiva a partir de C_j^p
8. $\quad$ atualize os valores de C_k^p para todos $k \in L$

Figura 3.8: Algoritmo para transformar escalonamento preemptivo em não preemptivo.

Após gerar um escalonamento preemptivo, o algoritmo ordena as tarefas por ordem de término do escalonamento preemptivo. O algoritmo gera um outro escalonamento não preemptivo da seguinte forma. Para cada tarefa j, é alocado, a partir do tempo C_j^p, espaço suficiente para executa-la de forma não preemptiva. Neste processo, ao alocar espaço para uma tarefa, as demais tarefas que terminam depois de C_j^p são deslocadas para frente, por isso é atualizado seus valores de tempo de término na linha 8 do algoritmo. Como exemplo considere a figura 3.9.

Baker [8] apresenta um algoritmo polinomial para o problema $1|r_j, pmtn|\sum C_j$ que denotamos por *Bk*. O algoritmo *Bk* é baseado na seguinte idéia: a qualquer instante, execute a tarefa com menor tempo para ser finalizada. Apresentamos o algoritmo *Bk* na figura 3.10. No algoritmo temos uma lista L que está sempre ordenada em ordem crescente por tempo de processamento das tarefas. A função *first* retorna o primeiro elemento da lista e a função *last* retorna o tempo restante que falta para a última tarefa em execução terminar. Denotamos por $t(m)$ o tempo de término da última tarefa em execução na máquina m.

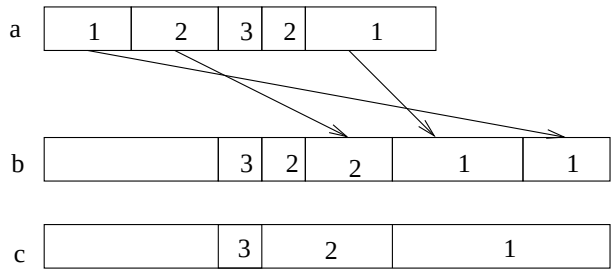

Figura 3.9: Temos três tarefas para escalonar $\{1, 2, 3\}$. O escalonamento ótimo preemptivo é apresentado em (a). Em (b) aloca-se espaço para que o escalonamento fique não preemptivo. Em (c) temos o escalonamento não preemptivo.

3.3.1 Resultados

Usando o algoritmo *Bk* como o algoritmo preemptivo no passo 1 do algoritmo *PSW*, obtemos uma 2-aproximação para o caso não preemptivo. Dado uma tarefa j, denotamos seu tempo de término no escalonamento preemptivo por C_j^p e seu tempo de término no escalonamento não preemptivo por C_j^n. O próximo teorema mostra o fator de aproximação do algoritmo *PSW* para o problema $1|r_j|C_j$.

Teorema 3.3.1 *O algoritmo PSW é uma 2-aproximação para o problema* $1|r_j|\sum C_j$ *quando utilizamos o algoritmo Bk no passo 1.*

Prova. Seja j uma tarefa qualquer. Seja C_j^p o tempo de término da tarefa j no escalonamento preemptivo e C_j^n o tempo de término no escalonamento não preemptivo. Considere o último pedaço de j que foi escalonado no preemptivo, cujo tamanho é q_j. Este pedaço foi escalonado do tempo $C_j^p - q_j$ até C_j^p. O algoritmo *PSW* insere $p_j - q_j$ unidades de tempo a partir de C_j^p e escalona j de maneira não preemptiva no bloco resultante de tamanho p_j. Desta maneira todas as tarefas que executam após C_j^p são empurradas sem alterar a ordem do escalonamento e sem violar os r_j. O algoritmo *PSW* remove todos os pedaços de j que ocorriam antes de $C_j^p - k_j$. Neste processo temos que:

$$C_j^n \leq (C_j^p + \sum_{k:C_k^P < C_j^P} p_k) + p_j \tag{3.2}$$

O algoritmo executa j a partir de C_j^p somado com o deslocamento que ocorreu devido ao processamento não preemptivo das tarefas k que terminaram antes de j. A desigualdade (3.2) pode ser reescrita da seguinte forma:

$$C_j^n \leq C_j^p + \sum_{k:C_k^P \leq C_j^P} p_k \tag{3.3}$$

ALGORITMO Bk(J)

```
L ← ∅
seja l o menor tempo de liberação das tarefas
para cada j ∈ J com r_j = l faça
    insira j em L
    J ← J − j
r_L ← l
enquanto (L ≠ ∅) faça
    se t(M_1) ≤ r_L então
        M_1 ← M_1||first(L)
        se L = ∅ então
            seja l o menor tempo de liberação das tarefas em J
            para cada j ∈ J com r_j = l faça
                insira j em L
                J ← J − j
            r_L ← l
    senão
        se p_first(L) < last(M_1) então
            troque a primeira tarefa de L com o último processo da máquina m_1
        senão
            se J = ∅ então
                r_L ← t(M_1)
            senão
                seja l o menor tempo de liberação das tarefas em J
                se l > t(M_1) então
                    r_L ← t(M_1)
                senão
                    para cada j ∈ J com r_j = l faça
                        insira j em L
                        J ← J − j
                    r_L ← l
retorne M_1
```

Figura 3.10: Algoritmo de Baker para o caso preemptivo.

O somatório é menor ou igual que C_j^p já que estamos somando os tempos de processamento das tarefas que terminam antes de j mais seu próprio requisito de processamento p_j. Logo vale

que $C_j^n \leq 2C_j^p$ e portanto

$$\sum_{j\in J} C_j^n \leq 2\sum_{j\in J} C_j^p \leq 2OPT,$$

já que o escalonamento preemptivo ótimo é um limitante para o escalonamento não preemptivo ótimo. □

Note que poderíamos obter um outro escalonamento não preemptivo da seguinte forma: Seja j a última tarefa a terminar, ou seja, cujo C_j^p é máximo. Podemos escalonar todas tarefas a partir de C_j^p até $2C_j^p$ de maneira não preemptiva. Este escalonamento é viável já que podemos escalonar de forma não preemptiva as tarefas no mesmo espaço de tempo gasto na forma preemptiva. Também respeitamos os r_j pois é claro que em C_{max}^p todas as tarefas já foram liberadas. Mas ao escalonarmos desta forma, não temos nenhuma garantia das relações entre C_i^p e C_i^n das tarefas i, com exceção da última tarefa j. Podemos notar então, que ao trabalharmos com algoritmos de aproximação, é importante que tenhamos sempre alguma forma de relacionar nossos resultados com o valor de uma solução ótima.

3.3.2 Aproximação Justa

Quando desenvolvemos algoritmos de aproximação é interessante mostrar que o algoritmo tem fator de aproximação justo, ou seja, mostrar que existem instâncias para as quais o algoritmo retorna o valor limitante. Isto nos permite mostrar que o algoritmo proposto não pode ter seu fator de aproximação melhorado. Como exemplo mostramos que o algoritmo *PWS* da seção anterior tem fator de aproximação justo.

Teorema 3.3.2 *O fator de aproximação do algoritmo PSW é justo.*

Prova. Considere a seguinte instância para o algoritmo: No tempo 0 uma tarefa j_1 com requisito de processamento B é liberada; no tempo $B-2$ uma tarefa j_2 com requisito de processamento 1 é liberada e no tempo $B+1$, x tarefas com requisito de processamento 1 são liberadas. O escalonamento ótimo preemptivo para esta instância é obtido da seguinte forma. Escalone a tarefa j_1 até o tempo $B-2$, pare-a e execute a tarefa j_2 até o tempo $B-1$. Termine de executar j_1 até o tempo $B+1$. Depois execute as x tarefas restantes. Neste caso temos

$\sum C_j^p = B-1+B+1+\sum_{i=B+2}^{B+x+1} i = 2B+Bx+\frac{(2+x+1)x}{2} = 2B+Bx-1+\frac{(x+1)(x+2)}{2}$.

O escalonamento ótimo não preemptivo é obtido executando a tarefa j_1 até o tempo B e executando as demais tarefas em seguida. Este escalonamento tem valor maior em uma unidade do que o escalonamento preemptivo.

O algoritmo por sua vez, transforma o escalonamento preemptivo em um escalonamento não preemptivo onde j_2 termina no tempo $B-1$, j_1 termina no tempo $2B-1$ e as x tarefas são executadas em seguida. Temos então

$\sum C_j^n = B-1+2B-1+\sum_{i=2B}^{2B-1+x} i = 3B+2Bx-2+\frac{x(x-1)}{2}$.

Fazendo $x = \sqrt{B}$ temos que $\lim_{B\to\infty} \frac{2Bx+3B-2+\frac{x(x-1)}{2}}{Bx+2B-1+\frac{(x+1)(x+2)}{2}} = 2$.

Logo, a transformação do escalonamento preemptivo para o não preemptivo usando este algoritmo, tem fator de aproximação justo. $\square$

Capítulo 4

Algoritmos Baseados em Programação Linear

Neste capítulo apresentamos algoritmos que utilizam programação linear para obtenção de aproximações para problemas de escalonamento. Um programa linear, é um problema de otimização onde o objetivo é otimizar uma função linear com variáveis reais, cujos valores devem satisfazer um conjunto de restrições lineares. A função a ser otimizada é chamada função objetivo. Se restringirmos as variáveis do programa linear para valores inteiros, temos então um programa linear inteiro. Existem métodos eficientes para a resolução de programas lineares. Muitos problemas de otimização podem ser facilmente escritos como programas lineares inteiros. Mas, de modo geral, a resolução de um programa linear inteiro é NP-difícil. Uma estratégia é relaxar o programa linear inteiro e resolvê-lo de forma fracionária. Os algoritmos que apresentamos tentam de alguma forma obter soluções a partir do resultado fracionário de um programa linear. Este resultado é um limitante para o ótimo, já que o espaço de soluções do programa relaxado contém o espaço de soluções inteiras. Muitos dos algoritmos arredondam a solução fracionária tentando limitar as perdas. Nas seções seguintes apresentamos alguns algoritmos que utilizam programação linear. Não é nossa intenção mostrar resultados relacionados a teoria de programação linear. Apenas usamos resultados que sabemos serem válidos e os aplicamos a algoritmos de aproximação. Para mais detalhes sobre teoria de programação linear veja [10, 14].

4.1 Uma 2-Aproximação para $R||C_{max}$

O algoritmo desta seção foi proposto por Lenstra, Shmoys e Tardos [28]. Este algoritmo faz um arredondamento da solução de um programa linear obtendo uma 2-aproximação para o problema $R||C_{max}$. Neste problema temos um conjunto de máquinas não relacionadas e devemos minimizar o tempo de término do escalonamento gerado. Apresentamos a seguir um programa linear para este problema.

4.1.1 Formulação Linear do Problema

Lenstra, Shmoys e Tardos propuseram uma formulação linear para este problema usando variáveis binárias x_{ij} para cada par $j \in J$, $i \in M$. A variável x_{ij} tem valor 0 se a tarefa j é alocada ao processador i e tem valor 1 caso contrário. Além disso, há na formulação uma variável t, que representa a função objetivo, ou seja, $t = C_{max}$. A programa linear é apresentado a seguir:

$$\text{Min} \quad t$$

$$\begin{array}{rcll} \sum_{i \in M} x_{ij} & = & 1 \quad \forall j \in J & (4.1) \\ \sum_{j \in J} x_{ij} p_{ij} & \leq & t \quad \forall i \in M & (4.2) \\ x_{ij} & \geq & 0 \quad \forall i \in M, \quad \forall j \in J. & (4.3) \end{array}$$

A restrição (4.1) garante que toda tarefa será atribuída a exatamente uma máquina. A restrição (4.2) garante que nenhuma máquina executará mais do que o tempo t.

O problema com esta formulação é que a razão entre a solução ótima inteira deste sistema e a solução ótima fracionária pode ser muito grande como mostra o seguinte lema.

Lema 4.1.1 *Existe uma instância para o problema $R||C_{max}$ para o qual a razão entre a solução ótima inteira e a solução ótima fracionária do programa linear acima é m.*

Prova. Considere uma instância onde temos apenas uma tarefa com requisito de processamento m em cada uma de m máquinas idênticas. A solução ótima inteira é m. A solução fracionária do programa linear alocaria $\frac{1}{m}$ de processamento para cada máquina resultando em uma solução fracionária de valor 1. □

Logo, não conseguimos nenhuma aproximação melhor do que m se usarmos como limitante para o ótimo, apenas o valor da solução ótima fracionária do programa linear dado. Isto mostra a importância de se obter bons limitantes. Portanto, ao invés de usar a formulação dada, o algoritmo resolve vários programas lineares com a restrição de que o valor do escalonamento

achado deve ser maior do que o tempo de processamento de cada uma das tarefas nas máquinas em que executam. Seja $T \in Z^+$ um valor que achamos ser o valor ótimo. Definimos também o conjunto $S_T = \{(i,j)|p_{ij} \leq T\}$. Com isso, temos uma família de programas lineares *LP(T)*, um para cada valor possível de $T \in Z^+$. Só usamos valores de T para o qual todas tarefas podem ser processadas em pelo menos uma máquina. O programa linear *LP(T)* usa variáveis x_{ij} somente para os pares $(i,j) \in S_T$. Desta forma, é obtido um valor de escalonamento mínimo onde desconsidera-se a construção de escalonamentos como o do lema 4.1.1. Para cada valor T, o algoritmo resolve o programa linear *LP(T)* abaixo e apenas considera se ele admite solução ou não.

$$\text{LP(T)} \qquad \begin{array}{rcll} \sum_{i:(i,j)\in S_T} x_{ij} & = & 1 & \forall j \in J \\ \sum_{j:(i,j)\in S_T} x_{ij}p_{ij} & \leq & T & \forall i \in M \\ x_{ij} & \geq & 0 & \forall (i,j) \in S_T \end{array}$$

4.1.2 Propriedades dos Pontos Extremais

Descrevemos aqui algumas propriedades dos pontos extremais do programa linear *LP(T)*. Pontos extremais são soluções ótimas de um programa linear.

Lema 4.1.2 *Qualquer ponto extremal para LP(T) tem no máximo $n+m$ variáveis diferentes de zero.*

Prova. Seja $r = |S_T|$ o número de variáveis que *LP(T)* possui. Da teoria de poliedros, sabemos que uma solução viável para *LP(T)* é um ponto extremal se, e somente se, ela satisfaz r restrições linearmente independentes com igualdade. Destas r restrições, pelo menos $r-(n+m)$ devem ser escolhidas das últimas desigualdades ($x_{ij} \geq 0$). As variáveis destas últimas restrições receberão 0, logo, teremos no máximo $n+m$ variáveis diferentes de zero.

□

Dada uma solução x do programa linear *LP(T)*, dizemos que uma tarefa j é integralmente atribuída em x se ela é inteiramente alocada a uma única máquina. Caso contrário, dizemos que a tarefa j é alocada de forma fracionária.

Lema 4.1.3 *Qualquer ponto extremal que é solução de LP(T) deve ter pelo menos $n-m$ tarefas integralmente atribuídas.*

Prova. Considere uma solução que é ponto extremal para *LP(T)* e sejam α e β o número de tarefas alocadas integralmente e de forma fracionária, respectivamente. Cada tarefa alocada de forma fracionária deve ser alocada a pelo menos 2 máquinas. Desta forma temos:

$$\alpha + \beta = n \quad e \quad \alpha + 2\beta \leq n + m$$

A última desigualdade segue do lema anterior. Temos que $\alpha + 2\beta = n + \beta \leq n + m$. Isto implica que $\beta \leq m$. Sabemos que a seguinte igualdade é valida,

$$\alpha + \beta - m = n - m.$$

Como $\beta \leq m$, temos que $(\beta - m) \leq 0$ e portanto vale que $\alpha \geq n - m$.

□

Dada uma solução x de *LP(T)* que é ponto extremal, definimos um grafo $G_x = (J, M, E)$ bipartido onde J e M são vértices e cada aresta de E será definida da seguinte forma: $(j, i) \in E$ se, e somente se, $x_{ij} \neq 0$. Definimos também o grafo H_x que é o grafo induzido de G_x possuindo apenas arestas (j, i) para x_{ij} tal que $0 < x_{ij} < 1$. Dizemos que um grafo conexo com K vértices é uma pseudo árvore se ele contém no máximo K arestas. Dizemos que um grafo é uma pseudo floresta se cada um de seus componentes for uma pseudo árvore. Os próximos dois lemas nos mostram algumas propriedades dos grafos definidos.

Lema 4.1.4 *O grafo G_x é uma pseudo floresta.*

Prova. Seja G_c um componente conexo de G_x. Vamos mostrar que G_c é uma pseudo árvore. Restringimos o programa linear *LP(T)* e x apenas para este componente G_c, obtendo $LP_c(T)$ e x_c. O programa linear $LP_c(T)$ tem apenas as variáveis relativas a G_c assim como x_c é o vetor restrito a estas variáveis. Seja $x_{\bar{c}}$ o vetor solução das demais variáveis de x. Vamos mostrar que x_c é ponto extremal de $LP_c(T)$. Suponha que x_c não seja ponto extremal. Neste caso x_c é uma combinação convexa de outras duas soluções x_a e x_b, de $LP_c(T)$. Temos que $x_c = \alpha x_a + \beta x_b$, onde $\beta = 1 - \alpha$. Vamos montar outros dois vetores x_1 e x_2 da seguinte forma: x_1 receberá os valores de $x_{\bar{c}}$ multiplicados por $\frac{1}{2\alpha}$ e os valores de x_a nas respectivas posições assim como em x; x_2 receberá os valores de $x_{\bar{c}}$ multiplicados por $\frac{1}{2\beta}$ e os valores de x_b. Assim temos que $x = \alpha x_1 + \beta x_2$, o que implica que x não é um ponto extremal. Logo x_c é ponto extremal de $LP_c(T)$. Aplicando o lema 4.1.2 mostramos que G_c é uma pseudo árvore.

□

Lema 4.1.5 *O grafo G_x tem um emparelhamento perfeito.*

Prova. Cada tarefa alocada integralmente tem exatamente uma aresta incidente em G_x. Retirando estas tarefas de G_x juntamente com suas arestas, obtemos o grafo H_x. Como a quantidade de vértices e arestas retiradas são iguais, temos que H_x é uma pseudo floresta. Em H_x cada tarefa tem grau pelo menos 2, portanto todas as folhas são máquinas. Em seguida retiramos de maneira sucessiva, uma máquina e a tarefa incidente na respectiva aresta do grafo H_x, obtendo

no final, ciclos pares já que H_x é bipartido. Casando alternadamente as arestas desses ciclos finalizamos o nosso emparelhamento. Como exemplo, na figura 4.1 apresentamos um possível grafo H_x. As arestas mais escuras correspondem ao emparelhamento.

□

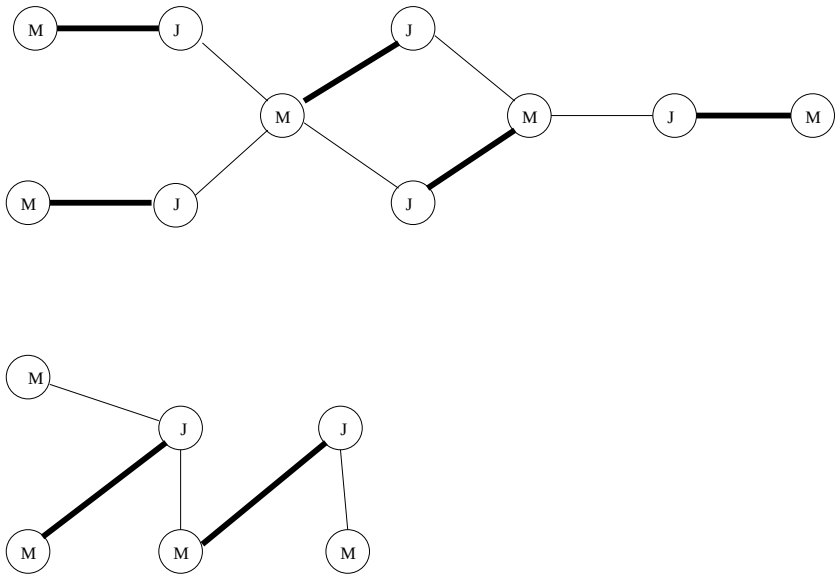

Figura 4.1: Atribuição a partir do matching.

4.1.3 O Algoritmo

Descrevemos agora o algoritmo e provamos que este é uma 2-aproximação para $R||C_{max}$. Primeiramente vamos calcular um limitante para o espaço de busca. Aloque cada tarefa em uma máquina onde sua execução é realizada no menor tempo possível. Seja α o tempo de término do escalonamento obtido. O algoritmo, denotado por *LST*, segue na figura 4.2.

Com uma busca binária no intervalo $[\frac{\alpha}{m}, \alpha]$ o algoritmo obtém o menor valor $T \in Z^+$ para o qual *LP(T)* possui solução viável. Seja T^* este valor e x a correspondente solução que é ponto extremal. As tarefas com valores integrais em x são alocadas em suas máquinas especificadas. Com a solução fracionária de x é construído o grafo H_x e um emparelhamento perfeito sobre este grafo. As demais tarefas são alocadas de acordo com o emparelhamento.

ALGORITMO LST(J, M)

1. $T \leftarrow \alpha$
2. $MIN \leftarrow \frac{\alpha}{m}$
3. $MAX \leftarrow \alpha$
4. enquanto $MAX - MIN > 1$ faça
5. $\quad T' \leftarrow \lceil \frac{MAX-MIN}{2} \rceil + MIN$
6. $\quad$ se $LP(T2)$ possui solução então
7. $\quad\quad T \leftarrow T'$
8. $\quad\quad MAX \leftarrow T'$
9. $\quad$ senão
10. $\quad\quad MIN \leftarrow T'$
11. seja $x = (x_{11}, \ldots, x_{nm})$ a solução de $LP(T)$
12. para $i \leftarrow 1$ até m faça
13. $\quad$ para $j \leftarrow 1$ até n faça
14. $\quad\quad$ se $x_{ij} = 1$ então
15. $\quad\quad\quad M_i \leftarrow M_i||j$
16. construa o grafo H_x para tarefas não alocadas
17. seja $P = \{(j_w, m_w), \ldots, (j_r, m_r)\}$ o emparelhamento perfeito de H_x
18. para cada aresta $(j, i) \in P$ faça
19. $\quad M_i \leftarrow M_i||j$
20. Devolva $M_1, ..., M_m$

Figura 4.2: Algoritmo para escalonamento de máquinas não relacionadas.

O teorema a seguir conclui esta seção mostrando o fator de aproximação do algoritmo.

Teorema 4.1.6 *O algoritmo LST é uma 2-aproximação para $R||C_{max}$.*

Prova. Seja T^* o valor ótimo obtido pelo algoritmo após a busca binária e x a correspondente solução que é ponto extremal. Como $T^* \leq OPT$, as tarefas que foram atribuídas de forma integral na solução x geram um escalonamento com tempo de término menor ou igual a T^*. As tarefas que têm solução fracionária são atribuídas segundo o emparelhamento perfeito. Pelo lema 4.1.3, sabemos que existem no máximo m tarefas atribuídas desta forma. Logo, cada máquina recebe no máximo uma tarefa a mais. Como para toda tarefa restringimos o problema a instâncias onde $p_{ij} \leq T^*$, temos que o escalonamento gerado é no máximo $2OPT$. $\square$

4.2 Um PTAS para $P3|fix_j|C_{max}$

Este problema é chamado de escalonamento de tarefas multiprocessadas em máquinas dedicadas. O algoritmo que apresentamos foi desenvolvido por Amoura *et al.* [2]. Vamos primeiramente dar algumas definições sobre o problema. Cada tarefa j tem tempo de processamento p_j e conjunto τ_j de máquinas que irá processar. Dizemos que τ_j é o *tipo* da tarefa j. Durante a execução da tarefa j, os processadores de τ_j estão dedicados exclusivamente a sua execução. Dadas duas tarefas j e k, dizemos que elas são *compatíveis* se $\tau_j \cap \tau_k = \emptyset$. Informalmente, duas tarefas são compatíveis se podem ser processadas simultaneamente. Chamamos de uma *configuração* uma coleção C de tipos tal que:

- $\forall \tau_k, \tau_j \in C$, $\tau_k \cap \tau_j = \emptyset$.
- $\cup_{\tau_k \in C} \tau_k = \{1, \ldots, m\}$.

Na próxima seção mostramos que para o caso $Pm|fix_j, pmtn|C_{max}$ existe um algoritmo ótimo polinomial. Na seção seguinte usamos o algoritmo preemptivo ótimo para obter um PTAS para o caso $P3|fix_j|C_{max}$.

4.2.1 Um algoritmo linear para o caso $Pm|fix_j, pmtn|C_{max}$

No caso $Pm|fix_j, pmtn|C_{max}$ temos um número fixo m de processadores. Vamos denotar por $\#C$ o número total de configurações em função de m. Vamos montar uma formulação linear para este problema em função das configurações. Temos variáveis x_i para cada configuração C_i. Particionamos as tarefas em função do seu tipo τ. Temos os subconjuntos:

$$J^\tau = \{j \in J | \text{ tarefa } j \text{ tem tipo } \tau\}$$

Para cada tipo τ também calculamos

$$D^\tau = \sum_{k \in J^\tau} p_k.$$

Com isso podemos montar o seguinte programa linear que denotamos por *LPP*.

$$\text{(LPP)} \quad \begin{array}{rcll} \text{Min} & \sum_{i=1}^{\#C} x_i & & \\ \sum_{i|\tau \in C_i} x_i & \geq & D^\tau & \forall \tau \subseteq \{1, \ldots, m\} \\ x_i & \geq & 0 & i = 1, \ldots, \#C \end{array}$$

As variáveis x_i recebem valores que representam o tempo que as máquinas processam a configuração C_i. Desta forma, o programa linear minimiza o tempo total C_{max} (que é o tempo correspondente à execução das configurações que recebem valores diferentes de 0), com a restrição de que deve ser alocado espaço suficiente para executar todas as tarefas. Na figura 4.3 apresentamos o algoritmo que denotamos por *ABKM*$_{pmtn}$.

O algoritmo primeiramente monta e resolve o programa linear *LPP*. Para cada variável x_i^*, obtida na resolução do programa linear *LPP*, diferente de zero, o algoritmo atribui processos a configuração correspondente ocupando todo o tempo relativo a esta variável. Processos que não podem ser executados totalmente nesta configuração são interrompidos e recomeçados posteriormente em outra configuração.

ALGORITMO ABKM$_{pmtn}(J, M)$

1. resolva programa linear LPP obtendo uma solução ótima $(x_1^*, \ldots, x_{\#C}^*)$
2. para $i \leftarrow 1$ até $\#C$ faça
3. para cada tipo τ faça
4. $t_i^\tau \leftarrow x_i^*$
5. para cada tarefa j faça
6. para $i \leftarrow 1$ até $\#C$ faça
7. se $\tau_j \in C_i$ então
8. seja t_i^τ o tempo disponível para o tipo τ_j em C_i
9. se $t_i^\tau \geq p_j$ então
10. execute j integralmente em C_i
11. $t_i^\tau \leftarrow t_i^\tau - p_j$
12. $p_j \leftarrow 0$
13. *break*
14. senão
15. execute fração t_i^τ de p_j em C_i
16. $p_j \leftarrow p_j - t_i^\tau$
17. $t_i^\tau \leftarrow 0$

Figura 4.3: Algoritmo para escalonamento preemptivo de tarefas multiprocessadas.

O seguinte teorema mostra que o algoritmo *ABKM*$_{pmtn}$ é ótimo e pode ser executado em tempo linear.

Teorema 4.2.1 *O escalonamento gerado pelo algoritmo ABKM*$_{pmtn}$ *é ótimo e é gerado com complexidade de tempo* $O(n)$.

Prova. Primeiramente vamos analisar a complexidade do algoritmo. Dado que o número m de processadores é constante, temos um número constante de configurações e portanto um número

constante de variáveis. Da mesma forma, o número de tipos diferentes é em função de m, o que implica que temos um número constante de restrições. Logo a resolução do programa linear *LPP* tem complexidade $O(1)$, assim como a execução dos laços sobre tipos e configurações. Para escalonar as tarefas, as atribuímos aos espaços representados pelas configurações (passo 5-17). Isto pode ser feito com complexidade $O(n)$.

Como o escalonamento é preemptivo, o programa linear *LPP* resolve de forma ótima o problema. O algoritmo gera um escalonamento exatamente de tamanho $\sum_{i=1}^{\#C} x_i$ que é a solução ótima de *LPP*.

□

Usando idéias deste algoritmo, é possível desenvolver um PTAS para o caso não preemptivo. Veremos isto na próxima seção.

4.2.2 O PTAS para $P3|fix_j|C_{max}$

A idéia do algoritmo é separar o conjunto de tarefas em grandes L, e pequenas T, de forma a ter um número constante de processos grandes. O algoritmo gera todas as possibilidades de escalonar os processos grandes. Já os processos pequenos são escalonados de forma ótima dentro dos espaços que sobram no escalonamento dos processos grandes, usando um algoritmo parecido com o algoritmo *ABKM*$_{pmtn}$ da seção anterior. O escalonamento dos processos pequenos é preemptivo e posteriormente transformado em não preemptivo. A transformação para um escalonamento não preemptivo é feita de forma a obter um escalonamento cujo valor não é muito distante do ótimo.

Dado um conjunto de tarefas grandes L, definimos um *quadro* de L como um subconjunto de tarefas compatíveis de L. Um *escalonamento relativo* E é uma sequência de quadros Q_i de L, tal que cada tarefa ocorre em uma subsequência de quadros e quadros consecutivos são diferentes. O último quadro do escalonamento relativo deve ser o conjunto vazio. Um escalonamento relativo representa uma ordem de escalonamento das tarefas grandes. Note que dado qualquer escalonamento das tarefas J, podemos associar o escalonamento das tarefas grandes L com uma sequência de quadros. Toda vez que uma tarefa grande termina ou começa, temos um novo quadro. Desta forma, podemos montar um escalonamento relativo dado qualquer escalonamento de J.

Dado um tipo τ definimos uma *τ-configuração* C_τ^u como uma coleção de tipos $\tau_u \subseteq \tau$ tal que:

1. $\forall \tau_u, \tau_v \in C_\tau^u, \quad \tau_u \cap \tau_v = \emptyset$
2. $\cup_{\tau_u \in C_\tau^u} \tau_u = \tau$.

Como exemplo considere que tenhamos um conjunto de tarefas grandes $\{1, 2, 3\}$ e um conjunto de tarefas pequenas $\{4, 5, 6, 7, 8, 9\}$. Na figura 4.4 temos um escalonamento relativo e

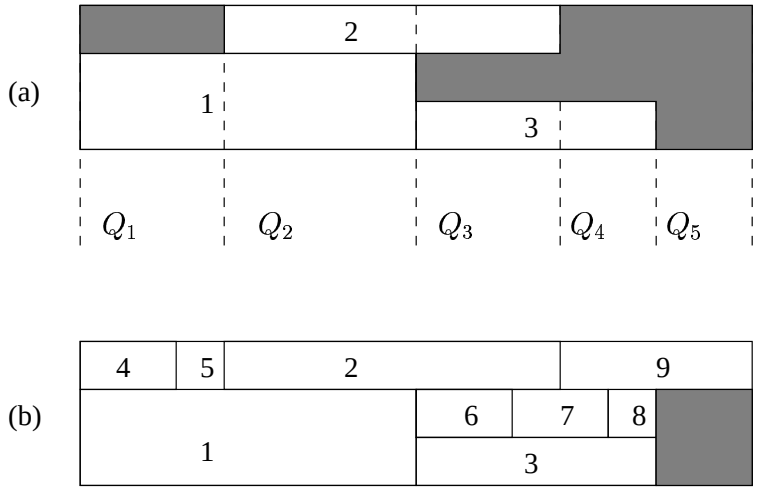

Figura 4.4: Em (a) temos um escalonamento relativo e em (b) um escalonamento das tarefas pequenas respeitando o escalonamento relativo.

um escalonamento das tarefas pequenas respeitando o escalonamento relativo. Na figura temos cinco quadros $Q_1, \ldots, Q_5$.

Seja $\#C_\tau$ o número de τ-configurações e ainda $D = \sum_{j \in J} p_j$. Antes de mostrarmos o algoritmo vamos definir um programa linear, que assim como o da seção anterior (de fato é baseado nele), é resolvido em tempo linear e gera um escalonamento ótimo para o caso onde apenas o escalonamento das tarefas pequenas é preemptivo. Dado um escalonamento relativo E das tarefas grandes L, temos as seguintes variáveis:

1. Variável t_i para cada quadro de E que representa o instante de término do quadro Q_i. Desta forma t_{f-1} é o instante de término do último processo grande. O tempo de término do escalonamento é representado pela variável t_f.

2. Variável e_τ para cada tipo τ, que representa o tempo total durante o qual processadores $\{1, 2, 3\} \setminus \tau$ estão processando tarefas de L e que processadores de τ não estão processando tarefas de L.

3. Para cada tipo τ e cada τ-configuração C_τ^u, definimos uma variável x_τ^u que desempenha o mesmo papel que a variável x_i da formulação da seção anterior. Estas variáveis representam o tempo de processamento das tarefas pequenas sobre a configuração definida por C_τ^u.

Definimos o conjunto Y como o conjunto de todos os tipos τ, tal que existe algum quadro durante o qual processadores $\{1, 2, 3\} \setminus \tau$ estão processando tarefas de L e que processadores de τ não estão processando tarefas de L. Dado um tipo τ, definimos o conjunto X^τ como o

conjunto de quadros Q_i tal que o conjunto de processadores usados no quadro Q_i é igual a $\{1,2,3\} \setminus \tau$. Com isso temos o seguinte programa linear que denotamos por *LPN*:

$$\text{Min} \quad t_f$$

$$\text{(LPN)} \qquad \begin{array}{rcll} t_i & > & t_{i-1} & \forall i \in \{1,\ldots,f\} \quad (4.1) \\ t_{k_j} - t_{i_j} & = & p_j & \forall j \in L \quad (4.2) \\ e_\tau & = & \sum_{i \in X^\tau} t_i - t_{i-1} & \forall \tau \subseteq Y \quad (4.3) \\ \sum_{C_\tau^u} x_\tau^u & \leq & e_\tau & \forall \tau \subseteq Y \quad (4.4) \\ \sum_{\tau \supseteq \tau' | \tau \in Y} \quad \sum_{u | \tau' \in C_\tau^u} x_\tau^u & \geq & D^{\tau'} & \forall \tau' \subseteq \{1,2,3\} \quad (4.5) \\ t_i & \geq & 0 & \forall i \in \{1,\ldots,f\} \quad (4.6) \\ x_\tau^u & \geq & 0 & \forall \tau\text{-configuração}; \tau \subseteq Y \quad (4.7) \end{array}$$

A restrição (4.1) é apenas para assegurar um escalonamento viável. Para cada tarefa grande j, temos o quadro i_j de início de j e o quadro k_j, que é o quadro de término de j. A restrição (4.2) força que o tempo entre estes dois quadros seja exatamente o tempo de processamento de j. Na restrição (4.3) temos o cálculo do tempo das variáveis e_τ. A restrição (4.4) garante que o tempo livre e_τ é suficiente para execução das respectivas τ-configurações. Finalmente na restrição (4.5) é garantido que haverá espaço suficiente para executar todas tarefas pequenas. A variável $D^{\tau'}$ representa o tempo total de todas as tarefas pequenas do tipo τ'. Nesta restrição é somado os tempos de todas τ-configurações que tem τ' como um de seus tipos.

Antes de mostrarmos o algoritmo para o problema $P3|fix_j|C_{max}$, damos um exemplo para melhor compreensão do programa linear *LPN*. Suponha que tenhamos 6 tarefas com requisitos de processamento e processadores dados pela tabela abaixo.

Tarefa	j_1	j_2	j_3	j_4	j_5	j_6
p_j	30	29	28	3	2	1
Processadores necessários	1	2	3	3	3,2	1,2,3

Tabela 4.1: Tabela com dados das tarefas

Suponha que $L = \{j_1, j_2, j_3\}$ e que $T = \{j_4, j_5, j_6\}$. Neste caso temos que $D = 93$. Suponha que tenhamos um escalonamento relativo E dado pela figura 4.5, ou seja, $E = (\{1,2,3\}, \{1,2\}, \{1\}, \emptyset)$.

Temos variáveis de tempo dos quadros $t_0, \ldots, t_4$. Consideramos variáveis para os tipos $\tau = \{3\}$, $\tau' = \{2,3\}$, $\tau'' = \{1,2,3\}$. Para cada um dos tipos τ,τ' e τ'' temos variáveis e_τ, $e_{\tau'}$ e $e_{\tau''}$ que indicam o tempo livre para cada um dos respectivos tipos. Temos ainda as variáveis referentes as τ-configurações. Para o tipo τ temos apenas a variável x_τ^3 indicando o tempo que

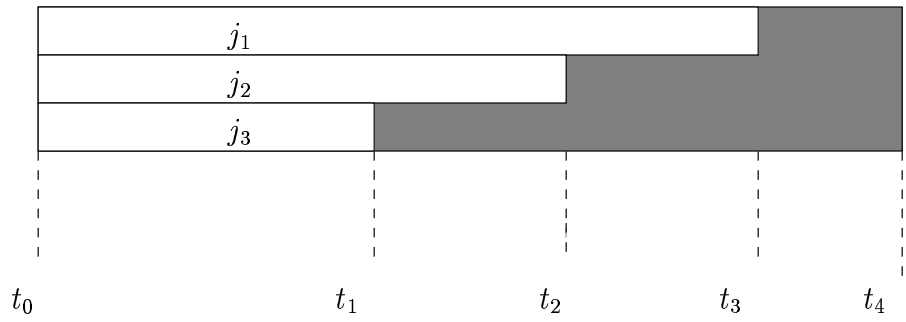

Figura 4.5: Um dos possíveis escalonamentos relativos.

alguma tarefa executa usando o processador número 3. Para o tipo τ' temos variáveis $x_{\tau'}^{23}$, que representa o tempo para uma tarefa que usa os processadores 2 e 3 simultaneamente, e $x_{\tau'}^{2,3}$ que representa o tempo para tarefas que usam o processador 2 ou o processador 3. Finalmente para o tipo τ'' temos variáveis $x_{\tau''}^{123}$, $x_{\tau''}^{12,3}$, $x_{\tau''}^{1,23}$, $x_{\tau''}^{2,13}$ e $x_{\tau''}^{1,2,3}$. Com estas variáveis podemos montar o programa linear abaixo.

$$\text{Min } t_4$$

$$t_0 < t_1, \quad t_1 < t_2, \quad t_2 < t_3, \quad t_3 < t_4$$

$$t_0 = 0, \quad t_1 - t_0 = 28, \quad t_2 - t_0 = 29, \quad t_3 - t_0 = 30$$

$$e_\tau = t_2 - t_1$$

$$e_{\tau'} = t_3 - t_2$$

$$e_{\tau''} = t_4 - t_3$$

$$x_\tau^3 \leq e_\tau$$

$$x_{\tau'}^{23} + x_{\tau'}^{2,3} \leq e_{\tau'}$$

$$x_{\tau''}^{123} + x_{\tau''}^{12,3} + x_{\tau''}^{1,23} + x_{\tau''}^{2,13} + x_{\tau''}^{1,2,3} \leq e_{\tau''}$$

$$x_\tau^3 + x_{\tau'}^{2,3} + x_{\tau''}^{12,3} + x_{\tau''}^{1,2,3} \geq D^3$$

$$x_{\tau'}^{23} + x_{\tau''}^{1,23} \geq D^{23}$$

$$x_{\tau''}^{123} \geq D^{123}$$

Temos ainda as restrições de que todas as variáveis devem ser positivas. Não inserimos estas restrições para facilitar a leitura do programa linear. Na próxima seção apresentamos o algoritmo.

4.3 O algoritmo

Note que dado um escalonamento relativo das tarefas grandes, o programa linear *LPN* resolve de forma ótima o problema onde tarefas pequenas podem ser interrompidas e continuadas mais tarde. Na figura 4.6 apresentamos o algoritmo para o problema $P3|fix_j|C_{max}$. O algoritmo usa a rotina *ABKM'*$_{pmtn}$, que é uma modificação do algoritmo *ABKM*$_{pmtn}$ da seção anterior. Esta rotina escalona as tarefas pequenas nos intervalos e_τ da mesma forma que *ABKM*$_{pmtn}$ escalona as tarefas de forma preemptiva.

Primeiramente, o algoritmo ordena as tarefas em ordem decrescente de valores de processamento. Depois disso ele acha um valor inteiro $i \leq \frac{1}{\epsilon}$ tal que $p_{7^i} + \ldots + p_{7^{i+1}-1} < \epsilon$. Seja $k = 7^i$. O algoritmo particiona as tarefas $J = T \cup L$ onde L são as k maiores tarefas. Com isso, o algoritmo constrói todos os escalonamentos relativos E possíveis das tarefas grandes. Para cada um destes escalonamentos relativos é montado e resolvido o programa linear *LPN*. Para cada escalonamento relativo E temos um escalonamento *SP(E)* com tarefas pequenas em modo preemptivo. Este escalonamento é gerado da mesma forma que o escalonamento preemptivo da seção anterior. Para cada escalonamento *SP(E)*, o algoritmo o transforma em um não preemptivo *SNP(E)*, executando ao fim do escalonamento as tarefas pequenas que foram interrompidas em algum momento de sua execução. Ao final é retornado o melhor escalonamento *SNP(E)* gerado.

Vamos a uma análise do algoritmo. Vamos supor que $0 < \epsilon < 1$ e que as tarefas de uma dada instância são tais que $D = 1$. Caso isto não ocorra podemos fazer com que as tarefas tenham esta característica. Para isso consideramos tarefas com novo processamento $p'_j = \frac{p_j}{D}$. Vamos mostrar que um escalonamento ótimo das tarefas com tempos de processamento modificados nos leva a um escalonamento ótimo das tarefas com tempos originais. Desta forma, podemos considerar instâncias com os tempos de tarefas modificados e conseguir aproximações para estes que também são aproximações para o problema original.

Teorema 4.3.1 *Seja (J, M, ϵ) uma instância para o problema $P3|fix_j|C_{max}$. Um escalonamento ótimo das tarefas j com processamentos $p'_j = \frac{p_j}{D}$ é um escalonamento ótimo das tarefas com processamentos originais. O inverso também é válido.*

ALGORITMO ABKM(J, M, ϵ)

1. seja $j_0, \ldots, j_{n-1}$ as tarefas em ordem decrescente de valores p_j
2. seja $i \leq \frac{1}{\epsilon}$ o menor valor inteiro não negativo tal que $p_{7^i} + \ldots + p_{7^{i+1}} \leq \epsilon$
3. $k \leftarrow 7^i$
4. $L \leftarrow \{j_0, \ldots, j_k\}$
5. $T \leftarrow J - L$
6. seja E^* o conjunto de todos escalonamentos relativos das tarefas de L
7. para cada $E \in E^*$ faça
8. resolva programa linear LPN obtendo solução ótima $(x_{\tau_1}(1), \ldots, x_{\tau_q}(w))$
9. seja $SP(E)$ o escalonamento gerado por ABKM$'_{pmtn}((x_{\tau_1}(1), \ldots, x_{\tau_q}(w)), E, T)$
10. $I \leftarrow \emptyset$
11. para cada $j \in T$ faça
12. se j foi interrompida em algum momento em $SP(E)$ então
13. $I \leftarrow I \cup j$
14. $SNP(E) \leftarrow SP(E)$
15. para cada $j \in I$ faça
16. retire j de $SNP(E)$
17. escalone j de forma ininterrupta ao final de $SNP(E)$
18. $MIN \leftarrow$ algum $E \in E^*$
19. para cada $E \in E^*$ faça
20. se $t(SNP(E)) < t(SNP(MIN))$ então
21. $MIN \leftarrow E$
22. retorne $SNP(MIN)$

Figura 4.6: Algoritmo para escalonamento não preemptivo de tarefas multiprocessadas.

Prova. Seja OPT' um escalonamento ótimo das tarefas modificadas e $C_{OPT'}$ o seu tempo de término. Seja X um escalonamento com as tarefas na mesma ordem em que aparecem neste escalonamento OPT', mas com as tarefas com seus tempos originais. Seja OPT um escalonamento ótimo das tarefas com tempos originais. Suponha por absurdo que $C_X > C_{OPT}$. C_{OPT} é dado por $\sum_{r \in R} p_r$ onde R é uma sequência de tarefas no escalonamento que leva até a última tarefa a ser escalonada. Seja S a sequência no escalonamento X. É claro que $\sum_{s \in S} \frac{p_s}{D} = C_{OPT'}$. Como $\sum_{r \in R} p_r < \sum_{s \in S} p_s$ temos que $\sum_{r \in R} \frac{p_r}{D} < \sum_{s \in S} \frac{p_s}{D}$. Logo, OPT' não era escalonamento ótimo das tarefas modificadas. Também é fácil ver que o inverso também é válido.

□

Para ver que o passo 2 do algoritmo faz sentido devemos observar o seguinte lema:

Lema 4.3.2 *Seja* $1 \geq p_0 \geq p_1 \geq \ldots \geq p_{n-1} \geq 0$ *uma sequência de valores tal que* $\sum_i p_i = 1$,

e seja $\epsilon > 0$. *Existe* $i \leq \frac{1}{\epsilon}$ *tal que* $p_{7^i} + \ldots + p_{7^{i+1}-1} \leq \epsilon$.

Prova. Vamos decompor a sequência $p_0 + \ldots + p_{n-1}$ em blocos $B_0 = p_0 + \ldots + p_6$, $B_2 = p_7 + \ldots + p_{7^2-1}, \ldots, B_i = p_{7^i} + \ldots + p_{7^{i+1}-1}$. Como a soma de todos os blocos é 1, no máximo $\frac{1}{\epsilon}$ blocos têm valor maior do que ϵ. Pegando o primeiro bloco B_i com valor menor que ϵ temos que $i \leq \frac{1}{\epsilon}$. □

Caso tenhamos $n < 7^{\frac{1}{\epsilon}}$, então temos um número constante de tarefas e podemos tentar todas as possibilidades de escalonamento destas tarefas como se todas fossem grandes. Precisamos agora, mostrar que o número de escalonamentos relativos é polinomial. O próximo lema mostra que na verdade, este número é limitado por uma constante.

Lema 4.3.3 *O número total de escalonamentos relativos é no máximo* $(k^3)^{2k}$

Prova. O número de processos grandes é $|L| = k$. É criado um novo quadro sempre que um processo grande começa ou termina. Existem $2k$ eventos deste tipo. Cada quadro consiste da escolha de no máximo três processos dentre k. Temos um limite superior de k^3. Logo existem no máximo $(k^3)^{2k}$ diferentes escalonamentos relativos. □

O algoritmo resolve o programa linear *LPN* que é parecido com o da seção anterior. É fácil perceber que dentre todos os escalonamentos gerados, um corresponderá ao ótimo no que diz respeito as tarefas grandes. Como as tarefas pequenas são escalonadas de forma ótima nos espaços vazios entre o escalonamento das tarefas grandes, então este escalonamento tem valor que é um limitante inferior para o ótimo. As tarefas pequenas que foram interrompidas são escalonadas de forma não preemptiva ao final do escalonamento. O próximo teorema mostra o limite para o tamanho do escalonamento gerado.

Teorema 4.3.4 *O algoritmo ABKM é um PTAS para* $P3|fix_j|C_{max}$.

Prova. Seja OPT' o valor de um escalonamento ótimo considerando as tarefas com tempos de processamentos modificados. Sabemos que o tamanho de um escalonamento relativo é no máximo $2k$. Com isso temos no máximo $m(2k) = 6k$ tarefas pequenas que são interrompidas. Desta forma, sabemos que o atraso total gerado no final do escalonamento é no máximo $p_k + \ldots + p_{7k-1}$ que foi escolhido no passo 2 do algoritmo de forma a ser menor que ϵ. Logo, o tamanho total do escalonamento é de $OPT' + \epsilon$. Se consideramos os tempos de processamento originais das tarefas, temos que o tamanho do escalonamento é limitado por $OPT + \epsilon D$, mas $OPT \geq \frac{D}{3}$. Logo, temos um limite de $OPT + 3\epsilon OPT$ para o escalonamento gerado.

□

Vale ressaltar que o algoritmo desta seção pode ser estendido para o caso $Pm|fix_j|C_{max}$ onde m é uma constante. A versão estendida foi mostrada de forma independente por Amoura *et al.* [3] e por Jansen e Porkolab [26], que também apresentaram um PTAS para o problema maleável $Pm|set|C_{max}$.

Capítulo 5

Algoritmos baseados em métodos probabilísticos

Algoritmos probabilísticos são aqueles que durante algum passo de sua execução fazem escolhas aleatórias[7]. Desta forma, quando executamos o algoritmo, podemos obter soluções diferentes para uma mesma instância de entrada. Muitos dos algoritmos que utilizam métodos probabilísticos também usam programação linear. A maneira como a solução fracionária é interpretada é que muda. Em algoritmos probabilísticos, a solução fracionária é vista como uma probabilidade e de alguma forma geramos uma solução baseada nestas probabilidades. É importante ressaltar que apesar dos resultados de aproximação serem probabilísticos, muitos destes algoritmos podem ser desaleatorizados através do método das esperanças condicionais. Desta forma obtemos um algoritmo com aproximação determinística. Nas seções seguintes exemplificaremos este método.

5.1 Uma $(2+\epsilon)$-aproximação para $R|r_{ij}|\sum w_jC_j$

O algoritmo desta seção foi proposto por Schulz e Skutella [32]. Trata-se de uma $(2+\epsilon)$-aproximação para o problema $R|r_{ij}|\sum w_jC_j$, onde temos máquinas não relacionadas, um conjunto de tarefas com diferentes tempos de liberação r_{ij} para cada máquina e devemos minimizar a soma ponderada dos tempos de término das tarefas. O algoritmo apresenta de forma bastante clara como podemos obter algoritmos de aproximação através de análises probabilísticas. Além disso, o algoritmo é um ótimo exemplo de como transformar programas lineares para que passem a ter tamanho polinomial com uma perda de ϵ na aproximação.

Primeiramente definimos um programa linear que modela o problema $R|r_{ij}|\sum w_jC_j$. Temos as seguintes variáveis:

1. Variável $C_j^{LP_R}$, que representa o tempo de término da tarefa j.

2. Variável y_{ijt}, que indica se a tarefa j está sendo processada na máquina i durante o intervalo de tempo $(t, t+1]$.

Seja $T = \max_{(i,j)} r_{ij} + \sum_{j\in J} \max_i p_{ij}$. A constante T é um limite superior para o tempo de término do escalonamento ótimo. Temos então a seguinte formulação relaxada, que denotamos por LP_R:

$$\text{Min} \sum_{j\in J} w_jC_j^{LP_R}$$

(LP^R)

$$\sum_{i=1}^{m}\sum_{t=r_{ij}}^{T}\frac{y_{ijt}}{p_{ij}} = 1 \quad \forall j \in J \tag{5.1}$$

$$\sum_{j\in J} y_{ijt} \leq 1 \quad \forall i \in M, \quad \text{e} \quad t = 0, ...T \tag{5.2}$$

$$C_j^{LP_R} = \sum_{i=1}^{m}\sum_{t=0}^{T}(\frac{y_{ijt}}{p_{ij}}(t+\frac{1}{2}) + \frac{1}{2}y_{ijt}) \quad \forall j \in J \tag{5.3}$$

$$C_j^{LP_R} \geq \sum_{i=1}^{m}\sum_{t=0}^{T} y_{ijt} \quad \forall j \in J \tag{5.4}$$

$$y_{ijt} = 0 \quad \forall i \in M, \quad \forall j \in J, \quad t = 0, ..., r_{ij}-1 \tag{5.5}$$

$$y_{ijt} \geq 0 \quad \forall i \in M, \quad \forall j \in J, \quad t = r_{ij}, ..., T \tag{5.6}$$

Vamos a uma descrição deste programa linear. Temos que minimizar a soma ponderada dos tempos de término das tarefas. A equação (5.1) nos assegura que uma tarefa j deve ser totalmente executada. A equação (5.2) nos garante que num dado intervalo de tempo em uma certa máquina, esta executa no máximo uma tarefa. Na equação (5.4) temos que o tempo de término de uma tarefa será sempre maior do que o seu tempo de processamento. A equação (5.5) nos garante que nenhuma tarefa é executada antes de ser liberada. Para analisar a equação (5.3) temos que observar o seguinte resultado:

Lema 5.1.1 *Dado um escalonamento viável não preemptivo para o problema $R|r_{ij}|\sum w_jC_j$ e uma tarefa j, o tempo de término desta tarefa é dado por:*

$$\sum_{t=t_i}^{t_f-1}(\frac{1}{t_f-t_i}(t+\frac{1}{2})+\frac{1}{2})$$

onde t_i é o tempo de início de j e t_f é o seu tempo de término. Note que dado um escalonamento não preemptivo viável, esta equação corresponde exatamente a equação (5.3).

Prova.

Resolvendo o somatório temos,

$$\begin{aligned}\textstyle\sum_{t=t_i}^{t_f-1}(\frac{1}{t_f-t_i}(t+\frac{1}{2})+\frac{1}{2}) &= \textstyle\sum_{t=t_i}^{t_f-1}\frac{1}{t_f-t_i}t+\sum_{t=t_i}^{t_f-1}\frac{1}{t_f-t_i}\frac{1}{2}+\sum_{t=t_i}^{t_f-1}\frac{1}{2}\\ &= \frac{t_i+t_f-1}{2}+\frac{1}{2}+\frac{t_f-t_i}{2}\\ &= t_f\end{aligned}$$

Logo, dado um escalonamento não preemptivo viável, a equação (5.3) corresponde exatamente ao tempo de término de uma tarefa.

□

5.1.1 O Algoritmo probabilístico

Nesta seção apresentamos um algoritmo que arredonda de forma probabilística uma solução do programa linear LP_R. O algoritmo é apresentado na figura 5.1 e o denotamos por *LPRound*.

O algoritmo primeiramente calcula uma solução ótima y do programa linear LP_R. Depois disso, para cada tarefa j, é calculado um valor α_j aleatório uniformemente distribuído no intervalo $[0,1]$. Durante a execução do laço dos passos 5-13, o algoritmo atribui cada tarefa para um par máquina-tempo (i,t) de forma aleatória e uniformemente distribuída com probabilidade $\frac{y_{ijt}}{p_{ij}}$. Além desta atribuição, é atribuído um valor t_j para cada tarefa j, que representa o tempo

para qual a tarefa j foi atribuída na máquina i. Ao final, o algoritmo escalona as tarefas nas máquinas que foram atribuídas por ordem de valores t_j sempre respeitando os tempos de liberação r_{ij} das tarefas.

ALGORITMO LPRound(J, M)

1. monte o programa linear LP_R com dados de entrada
2. seja y um vetor de soluções ótimo para LP_R
3. para cada $j \in J$ faça
4. $\quad\alpha_j \leftarrow random(0,1)$
5. para cada $j \in J$ faça
6. $\quad S \leftarrow 0$
7. $\quad$para $i \leftarrow 1$ até m faça
8. $\quad\quad$para $t \leftarrow 1$ até T faça
9. $\quad\quad\quad S \leftarrow S + \frac{y_{ijt}}{p_{ij}}$
10. $\quad\quad\quad$se $\alpha_j \leq S$ então
11. $\quad\quad\quad\quad M_i \leftarrow M_i||j$
12. $\quad\quad\quad\quad t_j \leftarrow t + random(0,1)$
13. $\quad\quad\quad\quad$break
14. ordene a lista M_i, $i = 1, \ldots, m$ por ordem de valores t_j das tarefas
15. retorne $(M_1, \ldots, M_m)$

Figura 5.1: Algoritmo probabilístico.

Através dos cálculos das esperanças dos tempos de términos das tarefas podemos obter limites para o escalonamento gerado pelo algoritmo.

Lema 5.1.2 *Seja y uma solução ótima de LP_R e considere o escalonamento construído pelo algoritmo LPRound. Temos que para cada tarefa j no escalonamento vale:*

a) O valor esperado de t_j é $\sum_{i=1}^{m}\sum_{t=0}^{T}\frac{y_{ijt}}{p_{ij}}(t+\frac{1}{2})$

b)O tempo de processamento esperado de cada tarefa j no escalonamento construído pelo algoritmo é dado por $\sum_{i=1}^{m}\sum_{t=0}^{T} y_{ijt}$

Prova. Como o algoritmo atribui t_j uniformemente no intervalo $(t, t+1]$, o valor esperado de t_j é $(t+\frac{1}{2})$. Somando o valor sobre todas as probabilidades de atribuição temos que (a) é valido. Dado um par (i,j) o tempo de processamento de j é p_{ij}. Somando todas as probabilidades de atribuição sobre os valores de processamento relativos à cada atribuição temos que a esperança de processamento de uma tarefa j é

$$\sum_{i=1}^{m}\sum_{t=0}^{T}\frac{y_{ijt}}{p_{ij}}p_{ij}.$$

Portanto (b) também é valido.

□

Com o resultado deste lema, podemos limitar os tempos de término das tarefas. Desta forma, podemos obter limitantes probabilísticos para o valor do escalonamento gerado. O próximo lema mostra o limite probabilístico para o tempo de término de uma tarefa.

Lema 5.1.3 *O valor esperado do tempo de término de uma tarefa j em um escalonamento construído pelo algoritmo LPRound é dado por:*

$$E[C_j] \leq 2\sum_{i=1}^{m}\sum_{t=0}^{T}\frac{y_{ijt}}{p_{ij}}(t+\frac{1}{2}) + \sum_{i=1}^{m}\sum_{t=0}^{T} y_{ijt}.$$

Prova.

A esperança do tempo de término da tarefa j pode ser calculada através da esperança de início da tarefa j mais a esperança de seu tempo de processamento,

$$E[C_j] = E[S_j] + E[p_j].$$

Note que $E[p_j]$ foi calculada no lema 5.1.2. Assim, vamos limitar a esperança do tempo de início da tarefa j. Suponha que j tenha sido atribuída a um certo par máquina-tempo (i,t). A esperança de tempo de início nesta atribuição, pode ser dada pelo tempo de processamento esperado que ocorre antes de j mais o tempo que a máquina i fica inativa. O tempo inativo da máquina i, pode ser limitado por t. Como a tarefa j foi atribuída ao par (i,t), então pelo menos no tempo t a tarefa pode iniciar.

O tempo de processamento esperado que ocorre antes de j, é dado pela probabilidade de uma tarefa k ser atribuída a máquina i e a um tempo anterior a t_j. Vamos supor que todas as tarefas que são atribuídas ao par máquina-tempo (i,t) são executadas antes de j, ou seja, recebem valores menores que t_j. O tempo esperado de processamento que ocorre antes de j é dado por:

$$\begin{aligned}\sum_{k\neq j} Pr[k \prec_i j]p_{ik} &\leq \sum_{k\neq j}\sum_{l=0}^{t-1} p_{ik}\frac{y_{ikl}}{p_{ik}} + \sum_{k\neq j} p_{ik}\frac{y_{ikt}}{p_{ik}} \\ &\leq t+1.\end{aligned}$$

Portanto, fixado uma tarefa $j \in J$ e uma atribuição (i,t), temos o limite para $E[S_j]$ de $2(t+\frac{1}{2})$. Somando este valor sobre todas as probabilidades de atribuição máquina-tempo e usando o lema 5.1.2 para limitar o tempo de processamento esperado temos:

$$E[C_j] \leq 2\sum_{i=1}^{m}\sum_{t=0}^{T}\frac{y_{ijt}}{p_{ij}}(t+\frac{1}{2}) + \sum_{i=1}^{m}\sum_{t=0}^{T} y_{ijt}.$$

□

Com o resultado deste lema, podemos limitar o valor esperado do escalonamento gerado pelo algoritmo *LPRound*. O próximo lema formaliza o resultado.

Teorema 5.1.4 *O valor esperado do escalonamento construído pelo algoritmo LPRound é limitado por 2OPT.*

Prova. Basta perceber, usando o lema 5.1.3, que o valor do tempo de término esperado de cada tarefa é menor ou igual a duas vezes o limite da restrição (5.3) do programa linear LP_R. Dada uma tarefa j, seja $C_j^{LP^R}$ o valor do tempo de término da tarefa em uma solução ótima para o programa linear LP^R. Desta forma temos

$$\begin{aligned} E[\textstyle\sum_j w_jC_j] = & \quad E[w_1C_1] + \ldots + E[w_nC_n] \\ \leq & \quad 2w_1C_1^{LP^R} + \ldots + 2w_nC_n^{LP^R} \\ \leq & \quad 2\textstyle\sum_j w_jC_j^{LP^R} \\ \leq & \quad 2OPT. \end{aligned}$$

□

5.1.2 Desaleatorizando o Algoritmo

O algoritmo dado na seção anterior, constrói um escalonamento com valor esperado de duas vezes o ótimo. Isto significa que na média ele retorna um escalonamento com esta propriedade, mas pode ser que existam casos em que o escalonamento produzido seja muito ruim. Apresentamos nesta seção, uma desaleatorização do algoritmo, para que desta forma tenhamos uma 2-aproximação determinística. Vale lembrar que o método apresentado aqui vale para muitos outros casos onde se obtém fatores de aproximação probabilístico. Para mais detalhes sobre o método veja [13].

O método baseia-se na idéia de considerar as alternativas de decisões probabilísticas uma a uma e sempre escolher a mais promissora. Uma alternativa é dita mais promissora se seu valor esperado é o menor possível, já que estamos tratando de um problema de minimização. Para mostrar o funcionamento da técnica no algoritmo anterior, considere uma pequena modificação que não altera o resultado do valor esperado do escalonamento. Considere que cada tarefa tem seu índice j. No passo 13 do algoritmo *LPRound*, vamos supor que os valores t_j recebem o valor t do respectivo intervalo em que a tarefa foi atribuída. Desta forma, tarefas atribuídas para um mesmo par (i,t) terão o mesmo valor para t_j e a ordem de escalonamento será determinada pelo índice j das tarefas. Tarefas com menor índice devem ser executadas antes.

Usando as mesmas idéias do lema 5.1.3 podemos provar o próximo lema, que também mostra um limite para tempo de término das tarefas.

Lema 5.1.5 *O valor esperado de tempo de término de uma tarefa j, com o algoritmo modificado, pode ser limitado por:*

$$E[C_j] \leq \sum_{i=1}^{m}\sum_{t=0}^{T}\frac{y_{ijt}}{p_{ij}}(p_{ij}+t+\sum_{k\neq j}\sum_{l=0}^{t-1}y_{ikl}+\sum_{k<j}y_{ikt})$$

Prova. Assim como no lema 5.1.3 podemos calcular a esperança do tempo de término como

$$E[C_j] = E[S_j] + E[p_j].$$

Considere que a tarefa j foi atribuída a um certo par (i,t), máquina-tempo.

Como a tarefa j foi atribuída ao par (i,t), então seu processamento é dado por p_{ij}. Para calcular $E[S_j]$, usamos as idéias do lema 5.1.3. O tempo inativo de i pode ser limitado por t. Como j é atribuído para o par (i,t) então pelo menos no tempo t a tarefa pode iniciar.

O tempo esperado de processamento que ocorre antes de j é dado por:

$$\begin{aligned}\sum_{k\neq j}Pr[k\prec_i j]p_{ik} &\leq \textstyle\sum_{k\neq j}\sum_{l=0}^{t-1}p_{ik}\frac{y_{ikl}}{p_{ik}}+\sum_{k<j}p_{ik}\frac{y_{ikt}}{p_{ik}}\\ &\leq \textstyle\sum_{k\neq j}\sum_{l=0}^{t-1}y_{ikl}+\sum_{k<j}y_{ikt}.\end{aligned}$$

Somando estes valores sobre todas probabilidades de atribuição máquina-tempo temos:

$$E[C_j] \leq \sum_{i=1}^{m}\sum_{t=0}^{T}\frac{y_{ijt}}{p_{ij}}(p_{ij}+t+\sum_{k\neq j}\sum_{l=0}^{t-1}y_{ikl}+\sum_{k<j}y_{ikt})$$

□

No processo de desaleatorização, fazemos escolhas determinísticas baseadas nos valores probabilísticos. Estamos interessados em calcular esperanças dado que certas escolhas foram feitas. Seja $K \subseteq J$ um conjunto de tarefas que tenham atribuição a pares máquina-tempo definidas. Para cada $k \in K$ usamos variáveis $x_{ikt} \in \{0,1\}$ que indicam se a tarefa k foi atribuída ao par (i,t) ou não. Com isto, podemos calcular esperanças dado que fizemos decisões sobre o conjunto K. Suponha que modificamos o algoritmo para fazer atribuição probabilística apenas para as tarefas não pertencentes a K. Dado uma tarefa j, podemos calcular a esperança de tempo de término, denotada por $E_K[j]$, desta tarefa no escalonamento gerado pelo algoritmo. Os próximos dois lemas nos mostram como calcular esta esperança.

Lema 5.1.6 *Seja $j \notin K$. O limite para o tempo de término esperado de j no escalonamento gerado pelo algoritmo, dado que as tarefas do conjunto K tenham atribuições definidas, é:*

$$E_K[C_j] \leq \sum_{i=1}^{m}\sum_{t=0}^{T}\frac{y_{ijt}}{p_{ij}}(p_{ij}+t+\sum_{k\in K}\sum_{l=0}^{t-1}x_{ikl}p_{ik}+\sum_{k\in K,k<j}x_{ikt}p_{ik}+\sum_{k\in J\setminus(K\cup j)}\sum_{l=0}^{t-1}y_{ikl}+\sum_{k\in J\setminus K,k<j}y_{ikt}).$$

Prova. Suponha que j seja atribuída ao par máquina-tempo (i,t). Usando as mesmas idéias do lema 5.1.3, temos um limite de tempo para inatividade da máquina i de t. O processamento de j é p_{ij}. Os somatórios

$$\sum_{k\in K}\sum_{l=0}^{t-1}x_{ikl}p_{ik}+\sum_{k\in K,k<j}x_{ikt}p_{ik}$$

correspondem a esperança de processamento que ocorre antes de j e os somatórios

$$\sum_{k\in J\setminus(K\cup j)}\sum_{l=0}^{t-1}y_{ikl}+\sum_{k\in J\setminus K,k<j}y_{ikt}$$

correspondem ao processamento que necessariamente ocorre antes de j. Somando estes valores sobre as probabilidades de atribuição da tarefa j a pares máquina-tempo, temos a validade do lema.

□

Lema 5.1.7 *Seja $j\in K$, tal que a tarefa j é atribuída ao par (i,t) $(x_{ijt}=1)$. O limite para o tempo de término esperado de j no escalonamento gerado pelo algoritmo, dado que as tarefas do conjunto K tenham atribuições definidas, é:*

$$E_K[C_j] \leq p_{ij}+t+\sum_{k\in K}\sum_{l=0}^{t-1}x_{ikl}p_{ik}+\sum_{k\in K,k<j}x_{ikt}p_{ik}+\sum_{k\in J\setminus(K\cup j)}\sum_{l=0}^{t-1}y_{ikl}+\sum_{k\in J\setminus K,k<j}y_{ikt}$$

Prova. Neste caso, sabemos que o tempo de inatividade da máquina i é limitado por t e que o processamento de j é p_{ij}. O valor esperado de processamento que ocorre antes de j é dado por

$$\sum_{k\in K}\sum_{l=0}^{t-1}x_{ikl}p_{ik}+\sum_{k\in K,k<j}x_{ikt}p_{ik}+\sum_{k\in J\setminus(K\cup j)}\sum_{l=0}^{t-1}y_{ikl}+\sum_{k\in J\setminus K,k<j}y_{ikt}.$$

Com isso finalizamos o lema.

□

O lema abaixo nos mostra o resultado necessário para a obtenção de uma aproximação determinística. Sabemos, pelo lema 5.1.3, que a esperança do tempo de término de uma tarefa j no escalonamento gerado pelo algoritmo é duas vezes o tempo de término da tarefa em um

escalonamento ótimo. O próximo lema mostra que podemos tomar decisões, baseado em esperanças, e gerar um escalonamento com uma tarefa atribuída determinísticamente de tal forma que a esperança do valor do escalonamento gerado não é maior do que a esperança do valor do escalonamento com esta tarefa atribuída de forma probabilística.

Lema 5.1.8 *Seja y uma solução ótima para o programa linear LP_R. Seja $K \subseteq J$, onde as tarefas pertencentes a K têm atribuição fixa (i,t). Seja $j \in J \setminus K$. Existe uma atribuição de j para um par (i_{min}, t_{min}) tal que:*

$$E_{K\cup\{j\}}[\sum_l w_lC_l] \leq E_K[\sum_l w_lC_l]$$

Prova.

Vamos denotar por $E^{(i,t)}_{K\cup\{j\}}$ a esperança $E_{K\cup\{j\}}$ com a tarefa j atribuída ao par (i,t). O lado direito da desigualdade pode ser escrito como uma combinação convexa de esperanças

$$E^{(i,t)}_{K\cup\{j\}}[\sum_l w_lC_l]$$

para todas as atribuições possíveis de j para um par (i,t). Observando os lemas 5.1.6 e 5.1.7, basta multiplicarmos cada uma das probabilidades $\frac{y_{ijt}}{p_{ij}}$ por cada uma das esperanças $E^{(i,t)}_{K\cup\{j\}}[\sum_l w_lC_l]$. Temos:

$$\begin{aligned}
E_K[\textstyle\sum_l w_lC_l] &= E_K[w_1C_1] + \ldots + E_K[w_nC_n] \\
&= (\tfrac{y_{1j1}}{p_{1j}}E^{(1,1)}_{K\cup\{j\}}[w_1C_1] + \ldots + \tfrac{y_{mjT}}{p_{mj}}E^{(m,T)}_{K\cup\{j\}}[w_1C_1]) + \ldots \\
&\quad +(\tfrac{y_{1j1}}{p_{1j}}E^{(1,1)}_{K\cup\{j\}}[w_nC_n]) + \ldots + \tfrac{y_{mjT}}{p_{mj}}E^{(m,T)}_{K\cup\{j\}}[w_nC_n]) \\
&= \tfrac{y_{1j1}}{p_{1j}}(E^{(1,1)}_{K\cup\{j\}}[w_1C_1] + \ldots + E^{(1,1)}_{K\cup\{j\}}[w_nC_n]) + \ldots \\
&\quad +\tfrac{y_{mjT}}{p_{mj}}(E^{(m,T)}_{K\cup\{j\}}[w_1C_1] + \ldots + E^{(m,T)}_{K\cup\{j\}}[w_nC_n])
\end{aligned}$$

Seja (i_{min}, t_{min}) o par para o qual obtemos o menor valor esperado. Vale a desigualdade

$$\frac{y_{1j1}}{p_{1j}}(E^{(1,1)}_{K\cup\{j\}}[w_1C_1]+\ldots+E^{(1,1)}_{K\cup\{j\}}[w_nC_n])+\ldots+\frac{y_{mjT}}{p_{mj}}(E^{(m,T)}_{K\cup\{j\}}[w_1C_1]+\ldots+E^{(m,T)}_{K\cup\{j\}}[w_nC_n]) \geq$$

$$\frac{y_{1j1}}{p_{1j}}(E^{(i_{min},t_{min})}_{K\cup\{j\}}[w_1C_1] + \ldots + E^{(i_{min},t_{min})}_{K\cup\{j\}}[w_nC_n]) + \ldots$$

$$+\frac{y_{mjT}}{p_{mj}}(E^{(i_{min},t_{min})}_{K\cup\{j\}}[w_1C_1] + \ldots + E^{(i_{min},t_{min})}_{K\cup\{j\}}[w_nC_n]) =$$

$$E^{(i_{min},t_{min})}_{K\cup\{j\}}[\sum_l w_lC_l].$$

□

Podemos alterar o algoritmo *LPRound* para que este passe a ser determinístico. Na figura 5.2 é apresentado uma versão determinística do algoritmo, que chamamos de *SSk*. No algoritmo *SSk* calculamos as esperanças $E^{(i,t)}_{K\cup\{j\}}[\sum_l w_lC_l]$ para todos os pares (i,t) e atribuímos a tarefa j para o par que tenha o menor valor esperado. Desta forma, estamos tomando decisões determinísticas e o lema 5.1.8 juntamente com o teorema 5.1.4, nos garantem que o valor do escalonamento gerado é limitado por $2OPT$.

ALGORITMO SSk(J, M)

1. monte programa linear LP_R com dados de entrada
2. seja y um vetor de soluções ótimo para LP_R
3. $K \leftarrow \emptyset$
4. $(i_j, t_j) \leftarrow (0,0)$, $j = 1, \ldots, n$
5. para cada $j \in J$ faça
6. $\quad MIN \leftarrow \infty$
7. $\quad$ para $i \leftarrow 1$ até m faça
8. $\quad\quad$ para $t \leftarrow 1$ até T faça
9. $\quad\quad\quad$ calcule $E^{(i,t)}_{K\cup j}[\sum_l w_lC_l]$
10. $\quad\quad\quad$ se $MIN > E^{(i,t)}_{K\cup j}[\sum_l w_lC_l]$ então
11. $\quad\quad\quad\quad (i_j, t_j) \leftarrow (i,t)$
12. $\quad\quad\quad\quad MIN \leftarrow E^{(i,t)}_{K\cup j}[\sum_l w_lC_l]$
13. $\quad K \leftarrow K \cup \{j\}$
14. para cada $j \in J$ e seu respectivo par (i_j, t_j) faça
15. $\quad M_{i_j} \leftarrow M_{i_j}||j$
16. ordene a lista M_i, $i = 1, \ldots, m$ por ordem de valores t_j das tarefas
17. retorne $(M_1, \ldots, M_m)$

Figura 5.2: Algoritmo determinístico.

5.1.3 Polinomialidade do Algoritmo

O número de restrições do programa linear LP_R não é polinomial já que T não é limitado polinomialmente. Há uma maneira bastante conhecida, introduzida por Hall, Shmoys e Wein [21], que transforma programas lineares como LP_R em programas lineares de tamanho polinomial com uma perda de ϵ na aproximação. Nesta seção apresentamos esta técnica.

Dado $\beta > 0$, seja L o menor inteiro tal que $(1+\beta)^L \geq T+1$. O novo problema se restringirá a L intervalos onde :

$$L = \lceil \log_{1+\beta}(T+1) \rceil$$

Com isso temos um número polinomial de intervalos em relação ao tamanho da entrada.

Sejam os intervalos $I_l = ((1+\beta)^{l-1}, (1+\beta)^l]$ para $1 \leq l \leq L$. Denotamos por $|I_l|$ o tamanho do intervalo, ou seja $|I_l| = \beta(1+\beta)^{l-1}$. Introduzimos variáveis y_{ijl} onde $y_{ijl}|I_l|$ é o tempo que a tarefa j é processada na máquina i no intervalo I_l. Abaixo apresentamos um novo programa linear de tamanho polinomial, que denotamos por LP':

$$Min \quad \sum_{j=1}^{n} w_jC_j$$

(LP')

$$\sum_{i=1}^{m}\sum_{l=0}^{L}\frac{y_{ijl}|I_l|}{p_{ij}} = 1 \quad \forall j \in J$$

$$\sum_{j\in J} y_{ijl} \leq 1 \quad \forall i \in M, \quad \text{e} \quad l = 0, ..., L$$

$$C_j^{LP'} = \sum_{i=1}^{m}\sum_{l=0}^{L}(\frac{y_{ijl}|I_l|}{p_{ij}}(1+\beta)^{l-1} + \frac{1}{2}y_{ijl}|I_l|) \quad \forall j \in J \quad (5.11)$$

$$y_{ijl} = 0 \quad \forall i \in M, \quad \forall j \in J, \quad (1+\beta)^l \leq r_{ij} - 1$$

$$y_{ijl} \geq 0 \quad \forall i \in M, \quad \forall j \in J, \quad \forall l = 0, ..., L$$

Na figura 5.3 apresentamos o algoritmo polinomial, denotado por SSk_{pol}, que é uma modificação do algoritmo *LPRound*. A única diferença está na resolução do programa linear, onde aqui é utilizado o programa linear LP', e na forma da atribuição probabilística das tarefas a pares de intervalo-máquina.

ALGORITHM SSk_{pol} (J, M, ϵ)

1. monte programa linear LP' com dados de entrada
2. seja y uma solução ótima do programa linear LP'
3. para cada $j \in J$ faça
4. $\quad \alpha_j \leftarrow random(0,1)$
5. para cada $j \in J$ do
6. $\quad S \leftarrow 0$
7. $\quad$ para $i \leftarrow 1$ até m faça
8. $\quad\quad$ para $l \leftarrow 1$ até L faça
9. $\quad\quad\quad S \leftarrow S + \frac{y_{ijl} \cdot |I_l|}{p_{ij}}$
10. $\quad\quad\quad$ se $\alpha_j \leq S$ então
11. $\quad\quad\quad\quad M_i \leftarrow M_i || j$
12. $\quad\quad\quad\quad t_j \leftarrow l$
13. $\quad\quad\quad\quad$ break
14. ordene lista M_i, $i = 1, \ldots, m$ por ordem de valores t_j das tarefas
15. retorne $(M_1, \ldots, M_m)$

Figura 5.3: Algoritmo probabilístico polinomial.

Temos que calcular as esperanças sobre este novo algoritmo para mostrarmos que as perdas geradas pelo programa linear LP' não são grandes. Os próximos lemas têm suas demonstrações muito parecidas com as dos lemas da seção anterior.

Lema 5.1.9 *A esperança de tempo de término $E[C_j]$ de uma tarefa $j \in J$ em um escalonamento gerado pelo algoritmo SSK_{pol} é:*

$$E[C_j] = \sum_{i=1}^{m} \sum_{l=0}^{L} y_{ijl}|I_l|.$$

Prova. Basta multiplicarmos cada uma das probabilidades de atribuição da tarefa j pelos respectivos valores de processamento nestas atribuições e temos que:

$$E[C_j] = \sum_{i=1}^{m} \sum_{l=0}^{L} p_{ij} \frac{y_{ijl}|I_l|}{p_{ij}}.$$

□

Lema 5.1.10 *O valor esperado do tempo de início de uma tarefa j no escalonamento construído pelo algoritmo SSk_{pol}, dado que ela foi atribuída a um par (i, l), é no máximo $2(1+\beta)(1+\beta)^{l-1}$.*

Prova. Seja j uma tarefa atribuída ao par máquina-intervalo (i,l). O tempo de inatividade da máquina i é limitado por $(1+\beta)^{l-1}$, pois neste intervalo de tempo a tarefa j já pode ser executada. O tempo de processamento esperado que ocorre antes de j é dado por:

$$\sum_{k\neq j} p_{ik}\sum_{q=0}^{l-1}\frac{y_{ikq}|I_q|}{p_{ik}}+\sum_{k\neq j} p_{ik}\frac{y_{ikl}|I_l|}{p_{ik}}=$$
$$(1+\beta)^{l-1}+\beta(1+\beta)^{l-1}=$$
$$(1+\beta)^{l-1}(1+\beta)$$

Logo o tempo de início esperado da tarefa j é:

$$(1+\beta)^{l-1}+(1+\beta)^{l-1}(1+\beta)\leq 2(1+\beta)(1+\beta)^{l-1}$$

□

Para finalizar, temos que calcular o valor esperado do tempo de término de uma tarefa. Isto é feito no próximo lema.

Lema 5.1.11 *O valor do tempo de término esperado de uma tarefa j no escalonamento gerado pelo algoritmo SSk_{pol} é limitado por $2(1+\beta)C_j^{LP'}$.*

Prova.

O tempo de término esperado da tarefa j é dado pelo tempo de início esperado mais o tempo de processamento esperado desta. Nos dois lemas anteriores temos os resultados para estes valores. Usando a restrição (5.11) de LP' podemos limitar o tempo de término da tarefa j. A restrição é dada abaixo:

$$C_j^{LP'}=\sum_{i=1}^{m}\sum_{l=0}^{L}\frac{y_{ijl}|I_l|}{p_{ij}}(1+\beta)^{l-1}+\sum_{i=1}^{m}\sum_{l=0}^{L}\frac{1}{2}y_{ijl}|I_l| \quad (5.11).$$

Usando os dois lemas anteriores obtemos a esperança do tempo de término de j:

$$E[C_j]\leq 2(1+\beta)\sum_{i=1}^{m}\sum_{l=0}^{L}\frac{y_{ijl}|I_l|}{p_{ij}}(1+\beta)^{l-1}+\sum_{i=1}^{m}\sum_{l=0}^{L}y_{ijl}|I_l|.$$

Logo, a esperança do valor de tempo de término da tarefa j é limitada por:

$$E[C_j]\leq 2(1+\beta)C_j^{LP'}.$$

□

Para qualquer $\epsilon>0$ podemos escolher $\beta=\frac{\epsilon}{2}$. O algoritmo SSk_{pol} pode ser desaleatorizado da mesma forma que o algoritmo *LPRound*. O teorema abaixo finaliza esta seção.

Teorema 5.1.12 *O algoritmo SSk_{pol} desta seção é uma $(2+\epsilon)$ aproximação para o problema $R|r_{ij}|w_jC_j$*

Prova. Seja $C_j^{LP'}$ o valor de tempo de término de uma tarefa j em uma solução ótima para o programa linear LP'. Basta aplicarmos o lema 5.1.11 com $\beta = \frac{\epsilon}{2}$. Como o valor do programa linear LP' é um limitante para o ótimo temos

$$\begin{aligned} E[\textstyle\sum_j w_jC_j] = & \quad E[w_1C_1] + \ldots + E[w_nC_n] \\ \leq & \quad (2+\epsilon)w_1C_1^{LP'} + \ldots + (2+\epsilon)w_nC_n^{LP'} \\ \leq & \quad (2+\epsilon)\textstyle\sum_j w_jC_j^{LP'} \\ \leq & \quad 2OPT. \end{aligned}$$

$\square$

Capítulo 6

Algoritmos baseados em formulações com Matrizes Semidefinidas

Muitos problemas são melhores escritos usando-se restrições não lineares. Desta forma, temos formulações que não podem ser resolvidas por métodos para resolução de programas lineares. Se a formulação for escrita de forma estritamente quadrática, ou seja, todas variáveis da formulação tem grau zero ou dois, podemos relaxar esta formulação para uma outra formulação vetorial. A formulação vetorial por sua vez, é equivalente a resolução de um programa semidefinido que pode ser resolvido em tempo polinomial. Muitos problemas têm sido satisfatoriamente resolvidos de maneira aproximada usando-se programação semidefinida. A escrita de uma formulação estritamente quadrática tem sido bastante utilizada para resolver problemas de forma aproximada (veja por exemplo [17]), mas existem formulações que não são estritamente quadráticas e fazem uso de matrizes semidefinidas para serem resolvidas em tempo polinomial. Veremos como matrizes semidefinidas aparecem na resolução de problemas de escalonamento. Não é nossa intenção apresentar detalhes de programação semidefinida. Apenas mostramos que é possível formular problemas quadráticos que possam ser resolvidos através de programação semidefinida. Para mais detalhes sobre o tema existem várias referências como [38, 37, 16].

6.1 Uma 2-aproximação para $R||\sum w_jC_j$

O algoritmo desta seção foi proposto por Skutella [33] e utiliza uma formulação quadrática para modelar o problema $R||\sum w_jC_j$, onde devemos minimizar a soma ponderada dos tempos de términos das tarefas em máquinas não relacionadas. Apesar da formulação não ser estritamente quadrática, Skutella mostrou como relaxar a formulação para obtenção de um sistema convexo com matrizes semidefinidas. Desta forma, podemos resolver o programa quadrático em tempo polinomial. De uma forma geral, quando temos um sistema convexo, podemos resolvê-lo em tempo polinomial. Na seção seguinte apresentamos a formulação do problema.

6.1.1 Formulação do problema

Nesta seção apresentamos formulações quadráticas para o problema. Vamos ver alguns detalhes que precisamos para modelar o problema $R||\sum w_jC_j$. Smith [36] mostrou que o problema $1||\sum w_jC_j$ é resolvido de forma polinomial usando-se a seguinte regra:

- Escalone as tarefas em ordem decrescente de valores $\frac{w_j}{p_j}$.

Desta forma, o nosso problema consiste em achar uma partição das tarefas nas máquinas. Para cada máquina i, definimos uma ordem das tarefas, executando j antes de k na máquina i ($j \prec_i k$) se $\frac{w_j}{p_{ij}} > \frac{w_k}{p_{ik}}$. Caso $\frac{w_j}{p_{ij}} = \frac{w_k}{p_{ik}}$ a ordem é definida pelo índice, deste modo $j \prec_i k$ se $j < k$.

Para cada máquina $i = 1, ..., m$ e para cada tarefa $j = 1, ..., n$, temos uma variável binária a_{ij} que recebe 1 caso a tarefa j seja atribuída a máquina i e 0 caso contrário. Temos também uma variável C_j para cada tarefa que representa o tempo de término desta. Temos a seguinte formulação quadrática, denotada por *IQP*:

$$\text{Min} \quad \sum_{j\in J} w_jC_j$$

$$\text{(IQP)} \qquad \begin{array}{rcll} \sum_{i=1}^{m} a_{ij} & = & 1 & \forall j \in J \quad (6.1) \\ C_j & = & \sum_{i=1}^{m} a_{ij}(p_{ij} + \sum_{k\prec_i j} a_{ik}p_{ik}) & \forall j \in J \quad (6.2) \\ a_{ij} & \in & \{0,1\} & \forall i \in M, \quad \forall j \in J \quad (6.3) \end{array}$$

Segue uma descrição da formulação. A restrição (6.1) nos garante que toda tarefa será escalonada em alguma máquina. Dada uma tarefa j, a restrição (6.2) calcula o tempo de término desta tarefa, que é dado pelo tempo de processamento que ocorre antes desta mais seu próprio processamento.

Esta formulação modela de forma correta o nosso problema. Podemos relaxar a formulação fazendo com que $a_{ij} \geq 0$ ao invés de termos a formulação inteira. Chamamos a formulação relaxada de *QP*. Note que não temos uma formulação estritamente quadrática, assim vamos reescrever esta formulação em uma nova formulação que seja convexa para que possamos resolvê-la em tempo polinomial.

A formulação *QP* pode ser reescrita como a formulação QP' descrita a seguir:

$$(QP') \quad \begin{array}{rcll} \text{Min} \quad c^T a + & \frac{1}{2}a^T Da & & (6.4) \\ \sum_{i=1}^{m} a_{ij} & = & 1 & \forall j \in J \\ a & \geq & 0 & \end{array}$$

Nesta formulação, o vetor $a \in \mathbb{R}^{mn}$ corresponde a todas as variáveis a_{ij} de *QP*. As variáveis a_{ij} em a estão ordenadas primeiramente pelas máquinas $i = 1, ..., m$ e para cada máquina i, temos uma ordenação das tarefas de acordo com a regra ($\prec_i$) definida anteriormente. O vetor $c \in \mathbb{R}^{mn}$ têm os valores $c_{ij} = w_j p_{ij}$ e está ordenado da mesma forma que o vetor a. A matriz $D = (d_{(ij)(hk)})$ é simétrica e tem dimensões $mn \times mn$ cujos valores são preenchidos segundo a regra:

$$d_{(ij)(hk)} = \begin{cases} 0 & \text{se} \quad i \neq h \quad \text{ou} \quad j = k \\ w_j p_{ik} & \text{se} \quad i = h \quad \text{e} \quad k \prec_i j \\ w_k p_{ij} & \text{se} \quad i = h \quad \text{e} \quad j \prec_i k \end{cases}$$

Como as entradas de D são iguais a zero quando $i \neq h$, temos uma matriz decomposta em m blocos $D_i, i = 1, ..., m$ na diagonal de D. Todas as outras posições tem valor 0. Como as tarefas estão ordenadas segundo a regra $\prec_i$, cada um dos blocos D_i é da seguinte forma:

$$D_i = \begin{pmatrix} 0 & w_2 p_{i1} & w_3 p_{i1} & ... & w_n p_{i1} \\ w_2 p_{i1} & 0 & w_3 p_{i2} & ... & w_n p_{i2} \\ w_3 p_{i1} & w_3 p_{i2} & 0 & ... & w_n p_{i3} \\ ... & ... & ... & & ... \\ w_n p_{i1} & w_n p_{i2} & w_n p_{i3} & ... & 0 \end{pmatrix} \tag{6.5}$$

Problemas da forma Min $c^T x + \dfrac{1}{2} x^T Dx$ sujeitos a restrições da forma $Ax = b$, podem ser resolvidos em tempo polinomial se D for uma matriz semidefinida [12]. Desta forma, temos que transformar a matriz D em uma matriz semidefinida obtendo assim um programa semidefinido que pode ser resolvido em tempo polinomial.

Skutella mostrou como transformar a matriz D, somando a ela uma fração positiva, para que esta fique semidefinida. Dados vetores binários $a \in \{0, 1\}^{mn}$, pode-se reescrever o termo $c^T a$

da função objetivo (6.4) como $a^T \text{diag}(c) a$, onde $\text{diag}(c)$ corresponde a uma matriz diagonal cujas entradas coincidem com as entradas do vetor c. Seja $\beta > 0$. Temos a seguinte função objetivo modificada:

$$\text{Min} \quad (1-\beta) \cdot c^T a + \frac{1}{2} a^T (D + 2\beta \cdot \text{diag}(c)) a \tag{6.6}$$

Como o vetor c é não negativo, note que

$$-\beta \cdot c^T a + \beta \cdot a^T \text{diag}(c) a \leq 0$$

para vetores arbitrários $a \in [0,1]^{mn}$. A igualdade é alcançada quando $a \in \{0,1\}^{mn}$. Desta forma, a nova função objetivo (6.6) pode obter valores menores do que a função objetivo (6.4). Além disso, a função objetivo (6.6) é não crescente em relação ao valor β, por isso temos que achar o menor valor possível para β tal que $D + 2\beta \cdot \text{diag}(c)$ passe a ser semidefinida. O próximo lema mostra qual é este valor.

Lema 6.1.1 *Se $\beta \geq \frac{1}{2}$, então a matriz $(D + 2\beta \cdot diag(c))$ é positiva semidefinida. Além disso, o menor valor possível de β para que matrizes da forma $(D + 2\beta \cdot diag(c))$ sejam semidefinidas é $\beta = \frac{1}{2}$.*

Prova.

Primeiramente consideramos $\beta = \frac{1}{2}$ e mostramos que $D + \text{diag}(c)$ é semidefinida. Um bloco D_i diagonal da matriz $D + \text{diag}(c)$ pode ser visto como:

$$D_i = \begin{pmatrix} w_1 p_{i1} & w_2 p_{i1} & w_3 p_{i1} & \dots & w_n p_{i1} \\ w_2 p_{i1} & w_2 p_{i2} & w_3 p_{i2} & \dots & w_n p_{i2} \\ w_3 p_{i1} & w_3 p_{i2} & w_3 p_{i3} & \dots & w_n p_{i3} \\ \dots & \dots & \dots & & \dots \\ w_n p_{i1} & w_n p_{i2} & w_n p_{i3} & \dots & w_n p_{in} \end{pmatrix} \tag{6.7}$$

Skutella mostrou que esta matriz é semidefinida dado que $\frac{w_1}{p_1} \geq, ..., \geq \frac{w_n}{p_n}$.

Considere agora que $\beta > \frac{1}{2}$. A matriz $(D + 2\beta \cdot \text{diag}(c))$ é a soma de duas matrizes semidefinidas, $D + \text{diag}(c)$ e $(2\beta - 1) \cdot \text{diag}(c)$, portanto o lema vale.

Para mostrar que o menor valor possível de β para que a matriz $(D + 2\beta \cdot \text{diag}(c))$ seja semidefinida é $\beta \geq \frac{1}{2}$ considere o seguinte exemplo de matriz D:

$$D = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{6.8}$$

Esta matriz corresponde ao caso onde temos apenas uma máquina e duas tarefas, cada uma com processamento 1 e peso 1. As entradas da diagonal de $(D + 2\beta \cdot \text{diag}(c))$ é igual a 2β de tal forma que ela só é semidefinida se $\beta \geq \frac{1}{2}$.

$\square$

Este lema nos permite trabalhar com uma nova formulação, denotado por *CQP*, que apresentamos a seguir.

$$\text{Min} \quad \tfrac{1}{2}c^Ta + \tfrac{1}{2}a^T(D + \text{diag}(c))a$$

$$(CQP) \qquad \begin{array}{rcll} \sum_{i=1}^m a_{ij} & = & 1 & \forall j \in J \\ C_j & = & \sum_{i=1}^m a_{ij}(p_{ij} + \sum_{k \prec_{ij}} a_{ik}p_{ik}) & \forall j \in J \\ a & \geq & 0 & \end{array}$$

6.1.2 O algoritmo

Nesta seção apresentamos o algoritmo que faz arredondamento probabilístico de uma solução do programa semidefinido CQP. Na figura 6.1 apresentamos o algoritmo, que denotamos por SkQ.

Primeiro o algoritmo calcula o vetor ótimo a do programa semidefinido CQP e depois atribui cada tarefa de forma probabilística a uma das máquinas. Cada tarefa é atribuída independentemente de forma aleatória a cada uma das máquinas com probabilidade a_{ij}. Dada uma atribuição de tarefas as máquinas, o algoritmo gera um escalonamento baseado no algoritmo de Smith.

ALGORITHM SkQ (J, M)

1. monte programa semidefinido CQP com dados de entrada
1. seja a um vetor que é solução ótima de CQP
2. para cada $j \in J$ faça
3. $\quad \alpha_j \leftarrow random(0, 1)$
4. para cada $j \in J$ faça
5. $\quad S \leftarrow 0$
6. $\quad$ para $i \leftarrow 1$ até m faça
7. $\quad\quad S \leftarrow S + a_{ij}$
8. $\quad\quad$ se $\alpha_j \leq S$ então
9. $\quad\quad\quad M_i \leftarrow M_i||j$
10. $\quad\quad\quad$ break
11. ordene as tarefas em M_i, $i = 1, \ldots, m$ pela função $\prec_i$
12. retorne $(M_1, \ldots, M_m)$

Figura 6.1: Algoritmo de Skutella baseado em uma formulação semidefinida.

Para calcular a razão de aproximação do algoritmo vamos obter antes alguns resultados

relativos ao arredondamento probabilístico.

Lema 6.1.2 *Considere que o algoritmo SkQ é executado com um vetor ótimo $\bar{a}$ obtido por uma solução ótima do programa quadrático QP ao invés do programa semidefinido CQP. A esperança do tempo de término de uma tarefa j no escalonamento montado pelo algoritmo é igual ao tempo de término no escalonamento ótimo.*

Prova.

Se a tarefa j é atribuída à máquina i, a esperança do valor de tempo de término é dado pela esperança de processamento que ocorre antes de j mais seu próprio processamento. Somando sobre todas as possibilidades de atribuição de j temos:

$$E(C_j) = \sum_{i=1}^{m}(\bar{a}_{ij}p_{ij} + \sum_{k\prec_i j}\bar{a}_{ij}\bar{a}_{ki}p_{ik})$$

que pela restrição (6.2) de QP temos

$$E(C_j) = C_j^{QP} = \sum_{i=1}^{m}\bar{a}_{ij}(p_{ij} + \sum_{k\prec_i j}\bar{a}_{ik}p_{ik}).$$

□

Vamos calcular agora a esperança do valor obtido pelo algoritmo *SkQ*. Dado um vetor viável a para o programa semidefinido CQP denotamos por $CQP(a)$ o valor obtido pelo programa semidefinido com este vetor.

Lema 6.1.3 *O valor esperado do escalonamento gerado pelo algoritmo SkQ é menor ou igual a 2 vezes o valor de um escalonamento ótimo.*

$$E(\sum_j w_jC_j) \leq 2OPT$$

Prova.

Seja $\bar{a}$ um vetor ótimo para QP'. Vimos que QP' é equivalente a QP. Usando este fato e o lema anterior podemos limitar o valor esperado do escalonamento gerado pelo algoritmo *SkQ* da seguinte forma:

$$E(\sum_j w_jC_j) = c^T\bar{a} + \frac{1}{2}\bar{a}D\bar{a}$$

Note que

$$c^T\bar{a} + \frac{1}{2}\bar{a}D\bar{a} = \frac{1}{2}c^T\bar{a} + \frac{1}{2}\bar{a}(D + \text{diag}(c))\bar{a} + \frac{1}{2}(c^T\bar{a} - \bar{a}\cdot\text{diag}(c)\bar{a})$$

ou seja,

$$\begin{aligned} E(\textstyle\sum_j w_jC_j) = & \quad CQP(\bar{a}) + \frac{1}{2}(c^T\bar{a} - \bar{a}\cdot \text{diag}(c)\bar{a}) \\ \leq & \quad 2CQP(\bar{a}) \\ \leq & \quad 2OPT. \end{aligned}$$

A primeira desigualdade vem do fato que $CQP(\bar{a}) \geq \frac{1}{2}c^T\bar{a}$. A segunda vem do fato de que o programa CPQ subestima o valor do programa quadrático QP' que é um limitante para o ótimo.

□

Podemos também desaleatorizar este algoritmo através do método das esperanças condicionais de forma a obter uma aproximação determinística. O próximo corolário finaliza esta seção.

Corolário 6.1.4 *O algoritmo SkQ pode ser desaleatorizado obtendo uma 2-aproximação para o problema $R||\sum w_jC_j$.*

Capítulo 7

Resumo de Resultados

Nesta seção apresentamos um resumo de resultados de aproximação para problemas de escalonamento. O objetivo deste resumo é identificar os melhores resultados de artigos estudados durante o mestrado. Certamente o resumo não está completo, mas serve para dar uma visão mais ampla da área. Apresentamos duas tabelas contendo resultados de aproximação para vários problemas de escalonamento bem como a referência de onde encontrar de forma mais detalhada estes resultados. Note que não temos resultados para algumas variações do problema na tabela abaixo. Pode realmente não existir resultados para estes problemas ou simplesmente não achamos resultados para eles. Vale ressaltar que muitos resultados na tabela foram obtidos de outro problema mais genérico. Se tivermos um resultado para $P|r_j|\sum w_jC_j$ por exemplo, podemos estender este resultado para o problema $P|r_j|\sum C_j$ já que este é um caso particular do primeiro.

Problema	*Off-line*	*On-line*
$1\|prec\| \sum w_j C_j$	(2) [20]	
$1\|prec\| \sum C_j$	(2) [20]	
$1\|prec, pmtn\| \sum w_j C_j$	(2) [20]	
$1\|prec, pmtn\| \sum C_j$	(2) [20]	
$1\|r_j\| \sum w_j C_j$	$(1+\epsilon)$ [1]	$(3+\epsilon)$ [20]
$1\|r_j\| \sum C_j$	$(1+\epsilon)$ [1]	$(3+\epsilon)$ [20]
$1\|r_j, pmtn\| \sum w_j C_j$	$(1+\epsilon)$ [1]	$(4/3)$ [31]
$1\|r_j, pmtn\| \sum C_j$	(1) [8]	(1) [8]
$1\|r_j, prec\| \sum w_j C_j$	(3) [29]	
$1\|r_j, prec\| \sum C_j$	(3) [29]	
$1\|r_j, prec, pmtn\| \sum w_j C_j$	(2) [20]	
$1\|r_j, prec, pmtn\| \sum C_j$	(2) [20]	

Tabela 7.1: Resultados com os melhores fatores de aproximação para as variações do problema de escalonamento.

Problema	*Off-line*	*On-line*
$P\|\|\sum w_jC_j$	$(1+\epsilon)$ [35]	$(4+\epsilon)$ [20]
$P\|\|\sum C_j$	(1) [24]	$(4+\epsilon)$ [20]
$P\|r_j\|\sum w_jC_j$	$(1+\epsilon)$ [1]	$(4+\epsilon)$ [20]
$P\|r_j\|\sum C_j$	$(1+\epsilon)$ [1]	$(4+\epsilon)$ [20]
$P\|r_j,pmtn\|\sum w_jC_j$	$(1+\epsilon)$ [1]	
$P\|r_j,pmtn\|\sum C_j$	$(1+\epsilon)$ [1]	
$P\|prec\|\sum w_jC_j$	(7) [29]	
$P\|prec\|\sum C_j$	(7) [29]	
$P\|prec,pmtn\|\sum w_jC_j$	$(3-\frac{1}{m})$ [20]	
$P\|prec,pmtn\|\sum C_j$	$(3-\frac{1}{m})$ [20]	
$P\|r_j,prec\|\sum w_jC_j$	(7) [29]	
$P\|r_j,prec\|\sum C_j$	(7) [29]	
$P\|r_j,prec,pmtn\|\sum w_jC_j$	(3) [20]	
$P\|r_j,prec,pmtn\|\sum C_j$	(3) [20]	
$Rm\|r_j\|\sum w_jC_j$	$(1+\epsilon)$ [1]	
$Rm\|r_j,pmtn\|\sum w_jC_j$	$(1+\epsilon)$ [1]	
$R\|r_j,prec\|C_{max}$	$\theta(\log m)$ [11]	
$R\|r_j,prec\|\sum w_jC_j$	$\theta(\log m)$ [11]	
$R\|prec\|\sum w_jC_j$	$\theta(\log m)$ [11]	
$R\|prec\|\sum C_j$	$\theta(\log m)$ [11]	
$R\|prec,pmtn\|\sum w_jC_j$		
$R\|prec,pmtn\|\sum C_j$		
$R\|\|\sum C_j$	1 [9]	
$R2\|\|\sum w_jC_j$	1.276 [33]	
$R\|\|\sum w_jC_j$	$(3/2)$ [34]	(8) [20]
$R\|r_{ij}\|\sum w_jC_j$	(2) [34]	(8) [20]
$R\|pmtn\|\sum w_jC_j$	(2) [34]	
$R\|r_{ij}\|\sum C_j$	(2) [34]	
$R\|r_{ij},pmtn\|\sum w_jC_j$	(3) [34]	
$R\|r_{ij},pmtn\|\sum C_j$	(3) [34]	
$R\|r_{ij},prec\|\sum w_jC_j$		
$R\|r_{ij},prec\|\sum C_j$		
$R\|r_{ij},prec,pmtn\|\sum w_jC_j$		
$R\|r_{ij},prec,pmtn\|\sum C_j$		
$Pm\|fix_j,pmtn\|C_{max}$	(1) [2]	
$Pm\|fix_j\|C_{max}$	$(1+\epsilon)$ [2]	
$Pm\|set\|C_{max}$	$(1+\epsilon)$ [26]	
$Pm\|fix_j\|\sum C_j$	$(1+\epsilon)$ [25]	
$Pm\|fix_j,pmtn,r_j\|\sum w_jC_j$	$(1+\epsilon)$ [25]	

Tabela 7.2: Resultados com os melhores fatores de aproximação para as variações do problema de escalonamento.

Capítulo 8

Implementação de Algoritmos de Aproximação para Problemas de Escalonamento

8.1 Prólogo

O artigo a seguir resume um conjunto de testes que realizamos com alguns algoritmos de aproximação para escalonamento. Até onde sabemos, muito pouco se sabe sobre o comportamento prático de algoritmos de aproximação para problemas de escalonamento. Os resultados mostram que apesar de alguns algoritmos terem fatores de aproximação altos, os valores obtidos estão muito próximos do valor ótimo. Também implementamos uma heurística para o problema mais genérico estudado. O algoritmo é baseado em um dos algoritmos apresentados. Os testes realizados mostram que esta heurística é melhor do que vários dos algoritmos apresentados.

8.2 Artigo

Practical Comparison of Approximation Algorithms for Scheduling Problems [1]

E.C. Xavier[2] F.K. Miyazawa[2]

Abstract

In this paper we consider practical aspects of approximation algorithms for scheduling problems. We implement some approximation algorithms and study their practical performance. The scheduling problems considered are all *NP*-hard. For the more general problem considered, we modify one of the algorithms and get better results.

Key Words: Approximation algorithms, Scheduling, Asymptotic performance.

8.2.1 Introduction

Several job scheduling problems in parallel machines that appears in practice are *NP*-hard. In these scheduling problems, we have a set of jobs with some attributes, that have to be processed in a set of machines minimizing the average completion time. If we consider that $P \neq NP$, we can not solve these problems to optimality efficiently. This motivates the development of approximation algorithms, that are efficient and produces results with quality guarantee. We implemented some approximation algorithms for scheduling jobs in parallel machines and studied their performance in practice.

Given an algorithm $\mathcal{A}$ for a minimization problem and an instance I of this problem, we denote by $\mathcal{A}(I)$ the value of the solution returned by $\mathcal{A}$ when applied to the instance I and we denote by $OPT(I)$ the value of an optimal solution to I. We say that an algorithm $\mathcal{A}$ has an approximation factor α, or is α-approximated, if $\mathcal{A}(I)/\mathrm{OPT}(I) \leq \alpha$, for all instances I. When the algorithm $\mathcal{A}$ is probabilistic and the inequality $E[\mathcal{A}(I)]/\mathrm{OPT}(I) \leq \alpha$ is valid for all instances I, where $E[\mathcal{A}(I)]$ is the expected value of the solution returned by algorithm $\mathcal{A}$, we say that $\mathcal{A}$ is a probabilistic α-approximated algorithm.

For all problems considered in this paper, we denote by $J = \{1, \ldots, n\}$ the set of jobs and $M = \{1, \ldots, m\}$ the set of machines. When machines are unrelated (*e.g.* have different

[1]This research was partially supported by FAPESP project 01/04412-4, MCT/CNPq under PRONEX program (Proc. 664107/97-4) and CNPq (Proc. 470608/01–3, 464114/00–4, 300301/98–7).

[2]Instituto de Computação — Universidade Estadual de Campinas, Caixa Postal 6176 — 13084–971 — Campinas–SP — Brazil, {eduardo.xavier, fkm}@ic.unicamp.br.

processing speed) we denote by p_{ij} the processing time of job j when executed on machine i, and by p_j when all machines are identical. We denote by r_j the release date of job j, which represents the moment that a job can start and denote by w_j the importance weight of finishing the job earlier. The completion time of the job is denoted by C_j.

We use the notation $\alpha|\beta|\gamma$ introduced by Graham, Lawler, Lenstra and Rinnooy Kan [3]. In the following we detail the terms used in this paper under this notation. The term α corresponds to the machine environment, P for identical machines or R for unrelated machines. The term β tell us some restrictions about jobs, if they have release dates, r_j, if the jobs can be interrupted and continued later, $pmtn$, etc... Finally the term γ indicates the objective function we want to minimize.

All problems we consider are non-preemptive. We implemented algorithms for the following problems: $P|r_j|\sum C_j$, $P||\sum w_jC_j$, $P|r_j|\sum w_jC_j$, $R||\sum w_jC_j$ and $R|r_j|\sum w_jC_j$. For the problem $P|r_j|\sum C_j$ we implemented the algorithm developed by Phillips *et al.* [7]. It is a combinatorial algorithm based in a heuristic for the preemptive case. For the problem $P||\sum w_jC_j$ we implemented the algorithm of Kawaguchi and Kyan [6], that is based in a list scheduling heuristic. For the problems $P|r_j|\sum w_jC_j$ and $R|r_j|\sum w_jC_j$ we implemented algorithms of Schulz and Skutella [8]. The algorithm for the first problem is combinatorial and the second is based in a solution of a linear program. Both algorithms are probabilistic. For the problem $R||\sum w_jC_j$ we implemented the algorithm developed by Skutella [9] that is based on a semidefinite program.

There are a lot of work in the development of approximation algorithms, but very few consider practical performance analysis. In [5], Hepner and Stein present an implementation of a PTAS for a single machine scheduling with release dates. There are PTAS algorithms for parallel machines as presented by Afrati et al. [1], but their implementation requires extra efforts and generally results in algorithms with time complexity represented by high degree polynomials.

All algorithms are implemented in C. For the algorithms that require solutions of linear or quadratic programs we use the Xpress-MP library, of Dash Optimization [2]. Based in the practical results, we propose a simple modification on the algorithm presented by Schulz and Skutella [8] for $R|r_j|\sum w_jC_j$. The algorithm obtain solutions with better quality.

The paper is organized as follows. In the second section we present the algorithms and give some insight of how they works. In the third section we present the results obtained in practice.

8.2.2 Algorithms

In this section we describe the algorithms and how they were implemented. We do not show how their approximate factors are obtained. The interested reader can find more details about the approximation results of these algorithms in the references.

Algorithm PSW **for** $P|r_j|\sum C_j$

The algorithm of this section was developed by Phillips *et al.* [7] which we denote by PSW. It finds a solution in two phases. In the first phase, the algorithm solves the preemptive version of this problem and in the second phase it uses an algorithm that converts the preemptive schedule to a non-preemptive schedule. The preemptive version of this problem is already *NP*-hard, and is solved by a 2-approximation algorithm. The algorithm that converts to a non-preemptive schedule produces a new schedule that is at most three times worse than the preemptive schedule. This leads to a 6-approximation algorithm for the problem $P|r_j|\sum C_j$.

The algorithm for the preemptive schedule is based in the following idea: at any time, execute m jobs with the shortest remaining amount of work. We present the algorithm in figure 8.1 and we denote it by *Preemptive*. We assume in the algorithm, that the machines are always ordered in non-decreasing order of processing time. The processing time of a machine i, is the finishing time of the last job that executes in it, which is denoted by $p(M_i)$. We assume that we have a minimum heap ordered by the processing time of jobs, wich we denote by *HEAP*. The heap contains only jobs that were released, *i.e.*, all jobs in the heap are ready to be processed. We change the release date of jobs in the heap to the release date of the heap which is denoted by r_{HEAP}. The function *first*(*HEAP*) returns a job with smallest processing time in the *HEAP* and the function *last*(M_i) return the remaining processing time of the last job executing in machine i.

In steps 1-6 the algorithm initializes, putting the jobs with the smallest release date in the heap. The loop in step 7 is done until all jobs becomes scheduled. In step 8 the algorithm tests if the machine less occupied has processing time less than or equal to the release date of the jobs in the heap. In this case, the first job of the heap is executed in this machine, otherwise, we have three other possibilities. If there is any job been executed having processing time bigger than the processing time of the job in the heap, the algorithm change these jobs (steps 17-19). In other case we put more jobs in the heap or set its release date (steps 20-31).

Notice that preemptions of jobs occurs only when we put new jobs in the heap, *i.e.*, jobs that are already in the heap does not fall in step 19, so the number of preemptions in each machine can be overestimated by n. The time complexity of the implemented algorithm is $O(n(\log n + m))$.

The algorithm PSW, which is presented in figure 8.2, uses the preemptive scheduled generated by the algorithm *Preemptive*. It schedules each job j in the machine which completed j in the preemptive schedule.

The algorithm generates a List M_i, for each machine i, of jobs ordered by their preemptive completion times C_j in the preemptive schedule. For each machine i, the algorithm PSW generates a non-preemptive schedule with jobs in the order specified by M_i, with the constraint that no job j starts before its release date.

The complexity of this algorithm is $O(n \log n + m)$ plus the complexity to generate the

```
ALGORITHM Preemptive(J, M)
 1.  HEAP ← ∅
 2.  let l be the minimum release date of jobs
 3.  for all jobs j ∈ J with r_j = l do
 4.      insert j in HEAP
 5.      J ← J − j
 6.  r_HEAP ← l
 7.  while(HEAP ≠ ∅)
 8.      if p(M_1) ≤ r_HEAP then
 9.          M_1 ← M_1||first(HEAP)
10.          if HEAP = ∅ then
11.              let l be the next minimum release date of jobs in J
12.              for all jobs j ∈ J with r_j = l do
13.                  insert j in HEAP
14.                  J ← J − j
15.              r_HEAP ← l
16.      else
17.          for i ← 1 to m do
18.              if p_first(HEAP) < last(M_i) then
19.                  change the first job in HEAP with last job in machine M_i and break
20.          if the algorithm had not changed jobs in step 19 then
21.              if J = ∅ then
22.                  r_HEAP ← p(M_1)
23.              else
24.                  let l be the next minimum release date of jobs in J
25.                  if l > p(M_1) then
26.                      r_HEAP ← p(M_1)
27.                  else
28.                      for all jobs j ∈ J with r_j = l do
29.                          insert j in HEAP
30.                          J ← J − j
31.                      r_HEAP ← l
32.  return (M_1, ..., M_m)
```

Figura 8.1: Algorithm that generate the preemptive schedule.

preemptive schedule. It was shown that the produced schedule is at most six times bigger than an optimal schedule.

ALGORITHM PSW(J,M)

1. generate a preemptive schedule $(L_1, \ldots, L_m)$ of (J, M) with algorithm Preemptive
2. order each list M_i by the preemptive completion time C_j of jobs
3. return $(M_1, \ldots, M_m)$

Figura 8.2: Algorithm that generate the non-preemptive schedule.

Algorithm KK **for the problem** $P||\sum w_j C_j$

The algorithm of this section is an extension of the problem $1||\sum w_j C_j$ for the problem $P||\sum w_j C_j$. The problem $1||\sum w_j C_j$ is solved optimally with the following algorithm developed by Smith [10]: order jobs in non-decreasing order of $\frac{p_j}{w_j}$ and schedule the jobs in this order. The heuristic for the parallel machine case is an extension: order jobs in non-decreasing order of $\frac{p_j}{w_j}$ and schedule jobs in this order every time a machine becomes free. Kawaguchi and Kyan [6] have shown that this algorithm produces schedules with a factor of $(\frac{\sqrt{2}+1}{2})$ of the optimal. The algorithm, which we denote by KK, is presented in figure 8.3 and have complexity time $O(n \log n + nm)$.

ALGORITHM KK(J, M)

1. order jobs in J in non-decreasing order of $\frac{p_j}{w_j}$
2. $M_i \leftarrow \emptyset$ for $i = 1, \ldots, m$
3. for each $j \in J$ in order do
4. let i be the less ocuppied machine
5. $M_i \leftarrow M_i || j$
6. return $(M_1, \ldots, M_m)$

Figura 8.3: Algorithm of Kawaguchi and Kyan.

Algorithm SZSK **for the problem** $P|r_j|\sum w_j C_j$

The algorithm of this section was developed by Schulz and Skutella [8]. We denote this algorithm by SZSK. The algorithm is a 2-probabilistic approximation algorithm and can also be derandomized. We implemented the probabilistic algorithm and run it 100 times and get the best generated schedule. With experimental results, we saw that running the algorithm more than 100 times does not lead to much better results. The algorithm is related to the linear formulation for a single machine problem presented below. We have variables y_{jt}, for each job j and for each time interval t that a job can run. We also have variables C_j, that represents the finishing time of job j. The constant T is an upper bound for the completion time of a job. The relaxed linear program, denoted by *LPS*, is the following:

$$\text{Min} \quad \sum_{j \in J} w_j C_j$$

$$\text{(LPS)} \quad \begin{array}{rcll} \sum_{t=r_j}^{T} y_{jt} & = & p_j & \forall j \in J \\ \sum_{j \in J} y_{jt} & \leq & 1 & t = 0, ..., T \\ C_j & = & \frac{p_j}{2} + \frac{1}{p_j} \sum_{t=r_j}^{T} y_{jt}(t + \frac{1}{2}) & \forall j \in J \\ y_{jt} & = & 0 & \forall j \in J \text{ and } t = 0, ..., r_j - 1 \\ y_{jt} & \geq & 0 & \forall j \in J \text{ and } t = r_j, ..., T \end{array}$$

It was shown that we can solve this linear program using a combinatorial algorithm [8]. Suppose we have just one machine m times faster than the machines considered. Consider the processing times of the jobs to be m times smaller. Construct a preemptive schedule for this single machine with the new processing times using the following rule: at any time, construct a preemptive schedule S on the new single machine by scheduling, among the available jobs, the one with the smallest $\frac{p_j}{w_j}$ ratio. The resulting schedule corresponds to an optimum solution to the formulation. Each variable y_{jt} recieve value 1 if job j was processed during time $[t-1, t)$ in the generated schedule.

Notice that the algorithm *Preemptive* is easily modified to solve the formulation and can be implemented to run in $O(n \log n)$. After this, we construct a schedule based in probabilistic assignments. We choose for each job j, a variable α_j uniformly distributed from the interval $[0, 1]$. Then we consider the probabilistic finishing time, *i.e.*, the first time in the schedule where the total amount of work done is $p_j \alpha_j$. We denote this value by $C_j(\alpha_j)$. The algorithm is presented in figure 8.4. The attribute j_{mach} of each job j, stores the machine where it must be executed.

ALGORITHM SZSK(J, M)

1. $M_i \leftarrow \emptyset$ for $i = 1, \ldots, m$
2. solve the linear program LPS with the combinatoric algorithm
3. for each $j \in J$ do
4. $\quad \alpha_j \leftarrow rand(0, 1)$
5. $\quad j_{mach} \leftarrow rand(1, m)$
6. let L be a list of the jobs ordered by their $C_j(\alpha_j)$
7. for each $j \in L$ in order do
8. $\quad M_{j_{mach}} \leftarrow M_{j_{mach}} || j$
9. return $(M_1, \ldots, M_m)$

Figura 8.4: Combinatoric algorithm of Schulz and Skutella.

The complexity time of the algorithm SZSK is $O(n \log n)$.

Algorithm SK **for** $R||\sum w_j C_j$

This algorithm is based in a semidefinite formulation and is presented by Skutella [9]. We denote the algorithm by SK. The quadratic program for this problem has binary variables a_{ij}, that says that a job j is to be processed in machine i, if and only if $a_{ij} = 1$ and variables C_j that represents the finish time of job j. We also have a function $\prec_i$ that specify the execution order of a job pair j, k in some machine i. Job j must be processed before k in machine i if $\frac{w_j}{p_{ij}} \geq \frac{w_k}{p_{ik}}$. The quadratic program is shown bellow:

$$\text{Min} \quad \sum_{j \in J} w_j C_j$$

$$\begin{array}{rcll} C_j & = & \sum_{i=1}^{m} a_{ij} \cdot (p_{ij} + \sum_{k \prec_i j} a_{ik} p_{ik}) & \forall j \in J \\ a_{ij} & \in & \{0,1\} & \forall i \in M, \quad \forall j \in J \end{array}$$

Skutella have shown that this formulation is equivalent to the following quadratic formulation

$$\text{Min} \quad c^T a + \frac{1}{2} a^T D a$$

$$\begin{array}{rcll} \sum_{i=1}^{m} a_{ij} & = & 1 & \forall j \in J \\ a & \geq & 0 & \end{array}$$

where $a \in \mathbb{R}^{mn}$ is a vector of all variables a_{ij} lexicographically ordered with respect to the natural order $1, 2, \ldots, m$ of the machines and then, for each machine i, the jobs ordered according to $\prec_i$. The vector $c \in \mathbb{R}^{mn}$ is given by $c_{ij} = w_j p_{ij}$ and $D = (d_{(ij)(hk)})$ is a symmetric $mn \times mn$-matrix given by

$$d_{(ij)(hk)} = \begin{cases} 0 \text{ if } i \neq h \text{ or } j = k \\ w_j p_{ik} \text{ if } i = h \text{ and } k \prec_i j \\ w_k p_{ij} \text{ if } i = h \text{ and } j \prec_i k. \end{cases}$$

It is shown that this problem can be solved in polynomial time if, and only if, matrix D is positive semidefinite. This motivates the construction of a new formulation, which we call *QPS*. The formulation is given bellow where $(D + \text{diag}(c))$ is positive semidefinite and $\text{diag}(c)$ is a diagonal matrix with the vector c.

$$
\text{(QPS)} \quad \begin{array}{lcll} \text{Min} & \frac{1}{2}c^T a + & \frac{1}{2}a^T(D + \text{diag}(c))a & \\ \sum_{i=1}^{m} a_{ij} & = & 1 & \forall j \in J \\ a & \geq & 0 & \end{array}
$$

We solve this formulation using the Xpress quadratic solver. Given a solution for *QSP*, we assign jobs to machine i with probability a_{ij}. It is shown that this is a probabilistic $2-$approximation algorithm. For the special case of identical parallel machines, the optimal solution to the above formulation is given by $a_{ij} = \frac{1}{m}$ for each one of the variables. Thus we implemented a combinatorial algorithm for this especial case attributing each job to a machine with probability $\frac{1}{m}$ and then executing the jobs in each machine by the specified order $\prec_i$. We denote this combinatorial algorithm by SK-C.

The algorithm for the general case is given in figure 8.5. We execute the algorithm 100 times, except the resolution of the *QSP* which is executed only once, returning the best generated schedule. Its complexity time is $O(n \log n + nm)$ plus the complexity time to solve *QSP*.

ALGORITHM SK(J, M)

1. $M_i \leftarrow \emptyset$ for $i = 1, \ldots, m$
2. let a be the vector solution of the quadratic program QPS
3. for each $j \in J$ do
4. $\quad \alpha_j \leftarrow rand(0, 1)$
5. for each $j \in J$ do
6. $\quad S \leftarrow 0$
7. $\quad$ for $i \leftarrow 1$ to m do
8. $\quad\quad S \leftarrow S + a_{ij}$
9. $\quad\quad$ if $\alpha_j \leq S$ then
10. $\quad\quad\quad M_i \leftarrow M_i || j$
11. $\quad\quad\quad$ break
12. order jobs in each M_i by $\prec_i$, $i = 1, \ldots, m$
13. return $(M_1, \ldots, M_m)$

Figura 8.5: Algorithm of Skutella based in a quadratic formulation.

Algorithm SZSK2 **for** $R|r_j|\sum w_j C_j$

This is also a probabilistic algorithm and is, presented by Schulz and Skutella [8]. The algorithm, denoted by SZSK2, is based in the solution of a linear formulation and is a generalization

of algorithm SZSK. As in the linear program of algorithm SZSK, the basic formulation have variables C_j that represents the finish time of job j and variables y_{ijt} that says that job j is processed in machine i at time t, for all possible unit time intervals that a machine can execute. The maximum time that a machine can execute is denoted by T. The formulation is exponential, but it can be made polynomial using interval times that increase exponentially in their size. We have intervals $I_l = ((1+\beta)^{l-1}, (1+\beta)^l]$ and binary variables y_{ijl} that indicates the interval that a job run despite the time that the job run. We represent the size of an interval I_l by $|I_l|$. The relaxed formulation is given bellow

$$\text{Min} \quad \sum_{j=1}^{n} w_j C_j$$

$$\text{(LPSS)} \qquad \begin{array}{rcl} \sum_{i=1}^{m} \sum_{l=0}^{L} \frac{y_{ijl}|I_l|}{p_{ij}} & = & 1 \quad \forall j \in J \\ \sum_{j \in J} y_{ijl} & \leq & 1 \quad \forall i \in M \text{ and } l = 0, ...L \\ C_j & = & \sum_{i=1}^{m} \sum_{l=0}^{L} (\frac{y_{ijl}|I_l|}{p_{ij}}(1+\beta)^{l-1} + \frac{1}{2} y_{ijl}|I_l|) \quad \forall j \in J \\ y_{ijl} & = & 0 \quad \forall i \in M, \forall j \in J, (1+\beta)^l \leq r_{ij} - 1 \\ y_{ijl} & \geq & 0 \quad \forall i \in M, \forall j \in J, l = 0, ..., L \end{array}$$

The algorithm solves the linear program *LPSS* and assign each job j to a machine-interval pair (i, I_l) at random with probability $\frac{y_{ijl} \cdot |I_l|}{p_{ij}}$. The jobs assigned to a machine i are scheduled in non-decreasing order of intervals assignment. If there are more than one job assigned to the same pair (i, I_l), the algorithm schedule them in order of their value j. The linear program is also solved with the Xpress solver. For a given $\epsilon > 0$ we set $\beta = \frac{\epsilon}{2}$. This algorithm is a probabilistic $(2+\epsilon)$-approximation algorithm. As in the algorithm SK, the probabilistic assignment step is executed 100 times and is returned the best generated schedule. The algorithm is presented in figure 8.6. Its time complexity is $O(nm \log T + n \log n)$ plus the time complexity to solve *LPSS*.

Heuristic Algorithm for $R|r_j| \sum w_j C_j$

In this section we present a simple modification in the algorithm SZSK2. The modified algorithm is denoted by SSHP. In [4], Hariri and Potts present a simple heuristic algorithm for $1|r_j| \sum w_j C_j$ used to find an upper bound for a branch and bound algorithm. The algorithm is as follows:

1. Let S be the set of all (unsequenced) jobs, let $H = 0$ and $k = 0$ and find $T = min_{j \in S}\{r_j\}$.
2. Find the set $S' = \{j | j \in S, r_j \leq T\}$ and find a job i with $i \in S'$ and $\frac{w_i}{p_i} = max_{j \in S'}\{\frac{w_j}{p_j}\}$.

ALGORITHM SZSK2(J, M, ϵ)

1. $M_i \leftarrow \emptyset$ for $i = 1, \ldots, m$
2. let y be the vector solution of the linear program $LPSS$
3. $t_j \leftarrow \infty$ for each $j \in J$
4. for each $j \in J$ do
5. $\quad \alpha_j \leftarrow rand(0, 1)$
6. for each $j \in J$ do
7. $\quad S \leftarrow 0$
8. $\quad$ for $i \leftarrow 1$ to m do
9. $\quad\quad$ for $l \leftarrow 1$ to L do
10. $\quad\quad\quad S \leftarrow S + \frac{y_{ijl} \cdot |I_l|}{p_{ij}}$
11. $\quad\quad\quad$ if $\alpha_j \leq S$ then
12. $\quad\quad\quad\quad M_i \leftarrow M_i || j$
13. $\quad\quad\quad\quad t_j \leftarrow l$
14. $\quad\quad\quad\quad$ break
15. order list M_i in non-decreasing order of values t_j, $i = 1, \ldots, m$
16. return $(M_1, \ldots, M_m)$

Figura 8.6: Probabilistic algorithm of Schulz and Skutella.

3. Set $k = k + 1$, sequence job i in position k, set $T = T + p_i$, set $H = H + w_i T$ and set $S = S - \{i\}$.

4. If $S = \emptyset$, then stop with the sequence generated having H as its cost. Otherwise set $T = max\{T, \min_{j \in S}\{r_j\}\}$ and go to step 2.

In algorithm SZSK2, the jobs are assigned to pairs machine-interval and them executed in each machine by the order of intervals assignments. In algorithm SSHP, the assignment step is done as in algorithm SZSK2, but the jobs assigned to a machine i are scheduled using the algorithm of Hariri and Potts.

8.2.3 Practical Analysis of the Implemented Algorithms

In this section, we present the results of our tests. Since some problems are particular cases of others, we made several different tests. Each subsection is reserved for one case. Before present each test we show how the data was generated. In each test we generate 100 jobs with processing requirement uniformly chosen from the interval $[1, 100]$, w_j chosen from the interval $[1, 10]$. When the problem require release dates, the data is generated using the same approach used by Hariri and Potts [4]. The release dates are uniformly chosen from the interval

$[0, E[p]n\gamma]$. This simulates the arrival of the n jobs from a stable queue according to a Poisson process with parameter γ [5]. The time in all tables is given in seconds. The ratio in the table corresponds to $\frac{V}{LB}$, where V is the value found by the algorithm and LB is a lower bound for the optimal. We made tests with 2, 5, 7 and 10 machines. As was done in [5], we use five different instances in each test problem, so the results in the tables corresponds to the mean of five tests. The algorithms were tested on a AMD Athlon 1.2Ghz with 800 MB of RAM.

Test for the problem $P||\sum w_jC_j$

In this problem we used the algorithms KK, SzSk and Sk-C. Table 8.1 show the results of this tests. The LB corresponds to the fractional optimal solution of the quadratic formulation QPS of algorithm Sk. It generates a better lower bound than the linear formulation $LPSS$.

Problem	LB	Algorithm	Value	Time	Ratio
$P2\|\|\sum w_jC_j$	382923.95	KK	383017.6	0.01	1.0002
		SzSk	383258.8	0.11	1.0008
		Sk-C	383319.6	0.05	1.0010
$P5\|\|\sum w_jC_j$	145821.3	KK	146088.2	0.01	1.0018
		SzSk	148134.2	0.17	1.0158
		Sk-C	147933.6	0.07	1.0144
$P7\|\|\sum w_jC_j$	115054.88	KK	115431.0	0.01	1.0032
		SzSk	117479.4	0.11	1.0210
		Sk-C	117882.6	0.07	1.0245
$P10\|\|\sum w_jC_j$	81997.11	KK	82516.2	0.01	1.0063
		SzSk	85775.2	0.11	1.0460
		Sk-C	85646.6	0.06	1.0445

Tabela 8.1:

The algorithms got very good results in practice. As we can see, the ratio grows when we use more machines. For algorithm KK the increase is very small. For the other ones the growth is more representative. We believe that this is because the number of possible ways of assign a job to a machine increases and the probabilistic algorithm would have to test much more possibilities to take better results.

Test for the problem $P|r_j|\sum C_j$

To solve the problem $P|r_j|\sum C_j$ we used the algorithms PSW, SzSk, SzSk2 and SSHP. Despite algorithm SzSk is the combinatorial version of SzSk2 for identical machines, we

also run algorithm SzSk2. The algorithms SzSk2 and SSHP were executed with parameter $\epsilon = 0.3$. We made different tests using different parameters γ to generate the release dates. We used parameter $\gamma = 0.2$, $\gamma = 0.4$ and $\gamma = 0.6$. The LB is the fractional optimal solution of the linear program of algorithm SzSk2 with $\epsilon = 0.3$. It is interesting to note that this lower bound may be far away from the integer optimal, because an optimal integer solution to the formulation $LPSS$ is already a relaxation to the original problem $P|r_j|\sum C_j$. Tables 8.2, 8.3 and 8.4 show the results obtained for these tests.

Problem with $\gamma = 0.2$	LB	Algorithm	Value	Time	Ratio
$P2\|r_j\|\sum C_j$	94590.59	PSW	109136.8	0.01	1.1537
		SzSk	124153.6	0.01	1.3125
		SzSk2	113298.6	5.08	1.1977
		SSHP	105280.4	5.79	1.1130
$P5\|r_j\|\sum C_j$	52569.48	PSW	60768.4	0.01	1.1559
		SzSk	75553.8	0.25	1.4372
		SzSk2	63130.6	58.19	1.2008
		SSHP	60418.6	52.95	1.1493
$P7\|r_j\|\sum C_j$	45222.02	PSW	57953.6	0.01	1.2815
		SzSk	68479.4	0.01	1.5142
		SzSk2	59530.8	90.13	1.3164
		SSHP	58486.2	90.17	1.2933
$P10\|r_j\|\sum C_j$	42052.44	PSW	53526.4	0.01	1.2728
		SzSk	62120.2	0.24	1.4772
		SzSk2	55645.2	171.29	1.3232
		SSHP	54845.2	183.71	1.3042

Tabela 8.2:

Notice that the algorithm SSHP get better results when we have few machines and small values of γ. This algorithm obtained the best results when we have just two machines. The algorithm PSW and SSHP are the best in all cases despite of the algorithm PSW has the biggest approximation factor. Other interesting point is that the ratio for the algorithm SzSk get better results when we have more machines with bigger values of γ. The algorithm SzSk2 obtained better results than the algorithm SzSk for all cases, except when we have big values of γ and more machines as we can see in table 8.4. Analyzing the fractional solution of the linear program used by algorithm the SzSk2, we can see that the solver generates an optimal fractional solution using less machines in such a way that variables of some machines are little used. Consequently, the generated schedule have some machines that are almost unused. The algorithm SzSk is the combinatorial version of SzSk2 but the jobs are attributed to all machi-

Problem with $\gamma = 0.4$	LB	Algorithm	Value	Time	Ratio
$P2\|r_j\|\sum C_j$	103833.92	PSW	126569.0	0.01	1.2189
		SzSk	162040.8	1.73	1.5605
		SzSk2	133894.2	6.52	1.2895
		SSHP	121682.4	6.11	1.1718
$P5\|r_j\|\sum C_j$	82872.40	PSW	104561.0	0.01	1.2617
		SzSk	124759.4	0.26	1.5054
		SzSk2	107531.0	62.90	1.2975
		SSHP	105297.2	59.61	1.2705
$P7\|r_j\|\sum C_j$	86762.90	PSW	108252.4	0.01	1.2476
		SzSk	119628.8	2.15	1.3788
		SzSk2	111726.6	112.95	1.2877
		SSHP	109449.6	107.21	1.2614
$P10\|r_j\|\sum C_j$	82787.75	PSW	103936.8	0.01	1.2554
		SzSk	107757.0	0.17	1.3016
		SzSk2	107522.0	218.68	1.2987
		SSHP	105247.4	221.41	1.2712

Tabela 8.3:

nes uniformly. This also explains why the algorithm SSHP when compared to the algorithm PSW, get better results using two machines than 7 and 10 machines.

Test for the problem $R||\sum w_j C_j$

In this problem we use the algorithms Sk, SzSk2 and SSHP. For the tests in table 8.5 we choose p_{ij} uniformly from the interval $[1, \ldots, 100]$. In the tests done in table 8.6 we choose processing times to give the idea that we have machines faster than others. Using two machines we choose processing times from the interval $[1, \ldots, 50]$ for the first machine and from $[50, \ldots, 100]$ for the second machine. Using five machines we take processing times from intervals, $[1, \ldots, 20], [20, \ldots, 40], \ldots, [80, \ldots, 100]$. Using seven machines we take processing times from intervals, $[1, \ldots, 15], [15, \ldots, 30], \ldots, [90, \ldots, 100]$. In the tests with ten machines, we choose processing times from intervals $[1, \ldots, 10][10, \ldots, 20][20, \ldots, 30], \ldots, [90, \ldots, 100]$. We use $\epsilon = 0.1$ in SzSk2 and SSHP. The LB corresponds to the fractional solution found by the quadratic formulation QSP of the algorithm Sk.

As we can see all algorithms produces schedules very close to the optimal. In general the algorithm SSHP produces better schedules.

Problem with $\gamma = 0.6$	LB	Algorithm	Value	Time	Ratio
$P2\|r_j\|\sum C_j$	134019.98	PSW	168188.8	0.01	1.2549
		SzSk	216423.0	1.55	1.6148
		SzSk2	175249.6	7.24	1.3076
		SSHP	165981.0	6.60	1.2384
$P5\|r_j\|\sum C_j$	124523.46	PSW	155055.8	0.01	1.2451
		SzSk	176075.4	0.10	1.4139
		SzSk2	162046.8	62.90	1.3013
		SSHP	156834.4	67.48	1.2594
$P7\|r_j\|\sum C_j$	126095.88	PSW	157384.2	0.01	1.2481
		SzSk	165377.2	2.21	1.3115
		SzSk2	162347.6	120.04	1.2874
		SSHP	158676.2	116.25	1.2583
$P10\|r_j\|\sum C_j$	121859.84	PSW	152096.0	0.01	1.2481
		SzSk	153544.8	0.07	1.2600
		SzSk2	158748.8	246.22	1.3027
		SSHP	153621.8	243.68	1.2606

Tabela 8.4:

Problem	LB	Algorithm	Value	Time	Ratio
$R2\|\|\sum w_jC_j$	194112.01	Sk	194216.4	1.13	1.0005
		SzSk2	194149.8	6.40	1.0001
		SSHP	194143.8	6.21	1.0001
$R5\|\|\sum w_jC_j$	37616.6	Sk	37763.0	38.31	1.0038
		SzSk2	37644.0	19.86	1.0007
		SSHP	37627.4	22.24	1.0002
$R7\|\|\sum w_jC_j$	26049.94	Sk	26305.0	89.36	1.0097
		SzSk2	26154.2	26.15	1.0040
		SSHP	26140.2	26.06	1.0034
$R10\|\|\sum w_jC_j$	11337.05	Sk	11666.2	200.79	1.0290
		SzSk2	11474.6	40.15	1.0121
		SSHP	11450.2	40.79	1.0099

Tabela 8.5:

Comparison for problem $R|r_j|\sum w_jC_j$

In this case we use the algorithms SzSk2 and SSHP to solve problem $R|r_j|\sum w_jC_j$. Table 8.7 show the results of the tests. We use $\gamma = 0.2$ to generate release dates. The LB

Problem *	LB	Algorithm	Value	Time	Ratio
$R2\|\|\sum w_jC_j$	246745.49	SK	246873.6	0.97	1.0005
		SzSK2	246811.2	6.12	1.0002
		SSHP	246783.8	6.13	1.0001
$R5\|\|\sum w_jC_j$	73513.03	SK	73934.6	37.41	1.0057
		SzSK2	73670.0	25.80	1.0021
		SSHP	73659.6	27.71	1.0019
$R7\|\|\sum w_jC_j$	51544.92	SK	52211.6	98.06	1.0129
		SzSK2	51828.4	25.90	1.0054
		SSHP	51849.2	26.14	1.0059
$R10\|\|\sum w_jC_j$	28453.73	SK	30516.2	207.62	1.0724
		SzSK2	29472.2	53.07	1.0357
		SSHP	29415.8	52.16	1.0338

Tabela 8.6:

is the optimal fractional solution of the linear program $LPSS$ with $\epsilon = 0.3$ and we also use this ϵ value in both algorithms. We emphasize that this lower bound may be far away from the integer optimal solution, since an integer optimal solution to $LPSS$ is already a relaxation of the problem $R|r_j|\sum w_jC_j$. The algorithm SSHP get better results in all tests.

Problem *	LB	Algorithm	Value	Time	Ratio
$R2\|r_j\|\sum w_jC_j$	283584.32	SzSK2	352346.2	5.958	1.2424
		SSHP	332137.8	5.686	1.1712
$R5\|r_j\|\sum w_jC_j$	200944.77	SzSK2	257145.8	51.83	1.2796
		SSHP	251919.2	53.73	1.2536
$R7\|r_j\|\sum w_jC_j$	201034.5	SzSK2	254033.2	109.32	1.2636
		SSHP	250165.8	94.71	1.2436
$R10\|r_j\|\sum w_jC_j$	203814.6	SzSK2	256892.4	186.38	1.2604
		SSHP	253180.0	185.56	1.2422

Tabela 8.7:

8.2.4 Conclusion

We present computational results for some approximation algorithms for scheduling on parallel machines. As expected, the practical solutions yields ratios better than the approximation

factor of the presented algorithms. We also present a heuristic algorithm that get better results in almost all cases studied.

8.2.5 Bibliography

1. F. Afrati, E. Bampis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Skutella, M. Sviridenko and C. Stein. Approximation schemes for minimizing average weighted completion time with release dates. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS'99)* pp. 32-44, 1999

2. Dash Optimization. Xpress-MP Release 13. *Xpress-MP Manual*, 2002.

3. E. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287-326, 1979.

4. A. M. A. Hariri and C. N. Potts. An algorithm for single machine sequencing with release dates to minimize total weighted completion time. *Discrete Applied Mathmatics* 5, 99-109, 1983.

5. C. Hepner and C. Stein. Implementation of a PTAS for Scheduling with Release Dates. In *3rd Workshop on Algorithm Engineering and Experiments (ALENEX 2001). Lecture Notes in Computer Sciense* 2513 pp. 202-215, 2001.

6. T. Kawaguchi and S. Kyan. Worst Case Bound of an LRF Schedule for the Mean Weighted Flow-Time Problem. *SIAM J. Computing* 4, 1986.

7. C. Phillips and C. Stein and J. Wein. Minimizing Average Completion time in the Presence of Release Dates. *Mathematical Programming B* 82, 1998.

8. A. S. Schulz and M. Skutella. Scheduling Unrelated Machines by Randomized Rounding. *SIAM Journal on Discrete Mathematics* Volume 15, Number 4, 2002.

9. M. Skutella. Semidefinite Relaxations for Parallel Machine Scheduling. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS'98)* pp. 472-481, 1998.

10. W. E. Smith. Various optimizers for single-stage production. *Naval Res. Logist. Quart.*, 1956, 3 pp. 58-66.

Capítulo 9

Algoritmos de Aproximação para uma Versão Generalizada do Problema da Mochila

9.1 Prólogo

O artigo a seguir trata de uma generalização do problema da mochila. No problema padrão da mochila temos como dados de entrada, o tamanho K da mochila, um conjunto S de items e funções s_e e v_e que mapeam o tamanho e o valor de cada item $e \in S$ respectivamente. O objetivo do problema é achar um subconjunto $U \subseteq S$ tal que $\sum_{u \in U} s_u \leq K$ e o valor $\sum_{u \in U} v_u$ seja maximizado. No problema que consideramos, além dos dados de entrada apresentados, temos também uma outra função c_e que mapeia uma classe para o item e, temos tamanho de divisórias de compartimentos d e limites d_{min} e d_{max} para o tamanho de compartimentos. No problema generalizado temos que achar um subconjunto $U \subseteq S$ cujo valor $\sum_{u \in U} v_u$ seja maximizado e uma partição $U_1, \ldots, U_k$ deste conjunto satisfazendo as seguintes restrições: *(i)* Para cada $U_i, 1 \leq i \leq k$ temos que $d_{min} \leq \sum_{u \in U_i} s_u \leq d_{max}$; *(ii)* Para cada $U_i, 1 \leq i \leq k$ temos que todos os items de U_i pertencem a uma mesma classe; *(iii)* $\sum_{u \in U} s_u + d \cdot k \leq K$.

Este problema tem aplicações práticas em problemas de corte e processamento de bobinas de metais na indústria metalúrgica e é exemplificada no artigo. Apesar do problema ser de empacotamento, não é difícil achar aplicações de tal problema em escalonamento. Como já foi visto no capítulo 3, seção 3.2, problemas de escalonamento e empacotamento são fortemente relacionados. Para exemplificar o uso do problema apresentado como um de escalonamento, suponha que tenhamos uma máquina m, que pode executar vários tipos de serviços mas depende da acoplagem a ela de alguns dispositivos. Desta forma se a máquina for executar uma determinada tarefa que necessite dispositivo c_1 e em seguida outra tarefa com dispositivo c_2, é necessário pará-la para trocar os dispositivos. Suponha que este tempo seja d. Cada tarefa j

possui tempo de processamento p_j, peso w_j e dispositivo necessário c_j. Além disso, suponha que nosso objetivo é maximizar o peso total das tarefas executadas até um total de tempo K. Este problema é facilmente mapeado ao problema da mochila generalizada.

Um resumo do artigo foi publicado no *IV* ALIO/EURO Workshop on Applied Combinatorial Optimization, que ocorreu no Chile em novembro de 2002. A versão completa, apresentada aqui, foi submetida para a revista *Discrete Applied Mathmatics* e ainda não recebemos resposta.

9.2 Artigo

Approximation Schemes for a Class-Constrained Knapsack Problem[1]

E. C. Xavier[2] F. K. Miyazawa[2]

Abstract

We consider approximation algorithms for a class-constrained version of the knapsack problem: Given an integer K, a set of items S, each item with value, size and a class, find a subset of S of maximum total value such that items are grouped in compartments. Each compartment must have only items of the same class and must be separated by the subsequent compartment by a wall division of size d. Moreover, two subsequent wall divisions must stay a distance of at least $d_{\min}$ and at most $d_{\max}$. The total size used by compartments and by wall divisions must be at most K. This problem have practical applications on cutting-stock problems.

Key Words: Approximation algorithms, knapsack problem.

9.2.1 Introduction

We consider approximation algorithms for a class-constrained version of the knapsack problem which we call *Generalized-Knapsack* (G-KNAPSACK). Given an integer K, a set of items $S = \{1, \ldots, n\}$, each item i with value v_i, size s_i and a class c_i, wall divisions of size d, find a set $M \subseteq S$ of maximum total value and a partition of M into compartments $C_1, \ldots, C_k$, each compartment with size $\sum_{e \in C_i} s_e$ where the following conditions are valid: *(i)* items in the same compartment have the same class, *(ii)* two subsequent compartments are separated by a wall division, *(iii)* the total size used by compartments and by wall divisions must be at most K, *(iv)* each compartment size must be at least $d_{\min}$ and at most $d_{\max}$.

This problem has a practical motivation in the roll cutting in the iron and steel industry. Ferreira *et al.* [3] present a cutting problem where a raw material roll must be cut into final rolls grouped by certain properties after two cutting phases. The rolls obtained after the first phase, called primary rolls, are submitted to different processing operations (tensioning, tempering, laminating, hardening etc.) before the second phase cut. Due to technological limitations,

[1]This research was partially supported by FAPESP project 01/04412-4, MCT/CNPq under PRONEX program (Proc. 664107/97-4) and CNPq (Proc. 470608/01–3, 464114/00–4, 300301/98–7).

[2]Instituto de Computação — Universidade Estadual de Campinas, Caixa Postal 6176 — 13084–971 — Campinas–SP — Brazil, {eduardo.xavier, fkm}@ic.unicamp.br.

primary rolls have a maximum and minimum allowable width and each cut generate a loss in the roll width.

In table 9.1, we present some common characteristics for final rolls. We consider three classes in this problem, one for each different thickness. The hardness interval of items with the same thickness are overlapped in a common interval to satisfy all hardness requirements. If there are items for which hardness cannot be assigned to the same thickness class, a new class must be defined for these ones.

Width (mm)	Hardness ($kg \cdot mm^{-2}$)	Thickness (mm)
50	50 to 70	4.50
74	60 to 75	4.50
93	32 to 39	3.50
35	32 to 41	3.50
20	20 to 30	2.50
100	24 to 35	2.50

Tabela 9.1: Characteristics of final rolls

In the example of table 9.1, we have a raw material roll of size K (1040 mm) which is first cut in primary rolls according to the different classes. In the example, the roll is first cut in three primary rolls. Each primary roll is processed by different operations to acquire the required thickness and hardness before obtaining the final rolls. Each cutting in the roll material generates a loss due to the width of the cutter knife. The cuts done at the first phase corresponds to the size of the wall division of our problem. The cuts of the second phase can be associated to the size of the items. This way, we only worry about the loss generated by the first phase cut. The figure 9.1 show the process.

Each processing operation has a high cost which implies items to be grouped before each processing operations, where each group corresponds to one compartment. In the above example we can consider three different classes and six different items. The size of the raw roll material corresponds to the size of the knapsack and the size of the wall division corresponds to the loss generated by the first cutting phase. The distances $d_{\min}$ and $d_{\max}$ are the minimum and maximum allowable width of the primary rolls.

Given a family of algorithms A_ϵ, for any $\epsilon > 0$, and an instance I for some problem P we denote by $A_\epsilon(I)$ the value of the solution returned by algorithm A_ϵ when executed on instance I and by $\mathrm{OPT}(I)$ the value of an optimal solution for this instance. We say that A_ϵ is a polynomial time approximation scheme (PTAS) for a maximization problem if for any $\epsilon > 0$ and any instance I we have $A_\epsilon(I) \geq (1-\epsilon)\mathrm{OPT}(I)$. If the algorithm is also polynomial in $1/\epsilon$ we say that A_ϵ is a fully polynomial time approximation scheme (FPTAS).

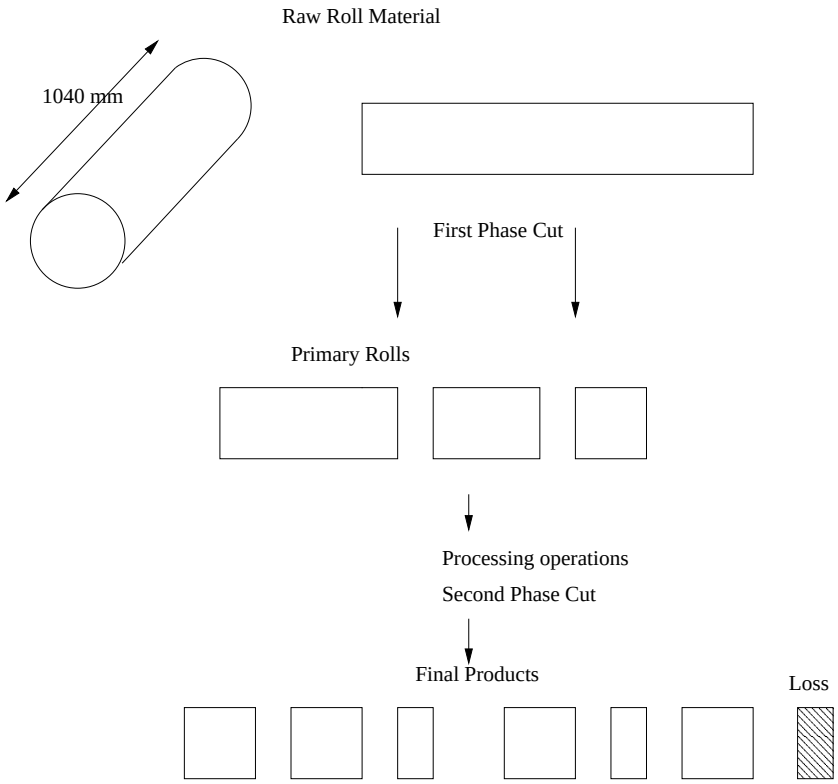

Figura 9.1: The two-phase cutting stock problem.

Results : In this paper we present a FPTAS and a PTAS for two restricted versions of the G-KNAPSACK. We also present a proof that a less restricted version of the G-KNAPSACK problem cannot be approximated unless $P = NP$. The algorithms we present are developed in two phases. In the first phase we have to solve a small problem for each class. In the second phase we use the results of each class to solve the fully problem using a dynamic programming approach.

For the FPTAS we consider at most a constant k of different item sizes for each class. The algorithm is simple and uses an algorithm to find an optimal bin packing of items. The bins have size $d_{\max}$ and must be filled by at least $d_{\min}$.

The PTAS is for the case when $d_{\min} = 0$. The algorithm is more complex and uses some nice ideas proposed by Chekuri and Khanna [2]. In this case, the algorithm find a set of items that can be packed into bins of size $d_{\max}$ and have a corresponding value very close to the optimal. The items are packed using known algorithms for bin packing.

Related Work : The knapsack problem is a well studied problem in the literature and a FPTAS was shown by Ibarra and Kim [5]. In [1], Caprara *et al.* present approximation schemes for two restricted versions of the knapsack problem, KKP and E-KKP. The KKP is the knapsack problem where the number of items chosen in the solution is at most k and the E-KKP has

the number of items chosen in the solution exactly k. Chekuri and Khanna [2] present a PTAS for the Multiple Knapsack Problem (MKP). In this problem we have a set B of bins, each bin $j \in B$ with capacity b_j, and a set S of items, each item i with size s_i and profit p_i. The objective is to find a subset $U \subseteq S$ of maximum profit such that U has a feasible packing in B. In [6], Schachnai and Tamir present a PTAS to a class-constrained multiple knapsack problem (CCMK) and class-constrained bin-packing problem (CCBP). In both problems we have N bins, each bin j with C_j compartments and size V_j. We also have a set I of items, each item $u \in I$ have class c_u, size s_u and profit p_u. In problem CCMK, we have to find a packing of items into the bins B, such that the value of the items packed is maximized and each bin j has at most C_j items of different classes. In problem CCBP the bins have size 1 and C compartments. The goal is to find a legal placement of all items in a minimal number of bins. If we consider that we do not have the restrictions *(ii)* and *(iv)* of G-KNAPSACK and the size of the wall is 0 then we can solve G-KNAPSACK using the algorithm to CCMK developed by Schachnai and Tamir.

Organization : In section 9.2.2, we present a generic algorithm to solve the G-KNAPSACK problem. This generic algorithm depends on the solution of another problem that we call SMALL. In sections 9.2.3 and 9.2.4 we present two algorithms to special cases of the problem SMALL which yields to a FPTAS and to a PTAS for the problem G-KNAPSACK. Finally, in section 9.2.5 we present an inaproximality result to G-KNAPSACK problem.

9.2.2 Generic Algorithm

In this section we present some notation and formally define the problem G-KNAPSACK. Furthermore, we present a general approximation scheme considering the existence of an algorithm to a restricted problem.

For most approximation schemes, it is sufficient to prove that the solution generated is $O(\epsilon)$ from the optimum solution, since we can obtain a solution that is ϵ from the optimum solution simple rescaling the parameter ϵ. Therefore, the following claim is valid.

Claim 9.2.1 *Given a constant $\epsilon > 0$, if A_ϵ is a polynomial time algorithm for a maximization problem P, such that for any instance I of P, we have $A_\epsilon(I) \geq (1 - O(\epsilon))\mathrm{OPT}(I)$, then there is a polynomial time algorithm B_ϵ such that $B_\epsilon(I) \geq (1 - \epsilon)\mathrm{OPT}(I)$.*

An *instance* I for the G-KNAPSACK is a tuple $(S, c, s, v, K, d, d_{\max}, d_{\min})$ where S is a set of items, c, s and v are class, size and value functions over S, respectively; d is the size of wall divisions, and $d_{\max}$ and $d_{\min}$ are bounds for the compartments size. All values are non-negatives. We denote by n and m, the number of items in the set S and the number of classes, respectively.

Given a set S and a numeric function f over S, we denote by $f(S)$ the sum $\sum_{e \in S} f(e)$.

A *solution* Q of an instance I for problem G-KNAPSACK is a pair (S', P') where $S' \subseteq S$ and P' is a partition of S', $P'_1, \ldots, P'_k$, each set P'_i has only items of the same class and are such that $d_{min} \leq s(P'_i) \leq d_{max}$ and $s(Q) \leq K$ where $s(Q) = \sum_{i=1}^{k}(s(P'_i) + d)$.

The *size of a solution* $Q = (S', P')$, where $P' = (P'_1, \ldots, P'_k)$, is equal to $s(Q)$ and the *value* of the solution Q, denoted by $v(Q)$, is equal to $v(S')$.

The problem G-KNAPSACK can be defined as follows: Given an instance I, find a solution Q of maximum value.

The crucial point for the algorithm of this section is a subroutine for a problem we call SMALL. The algorithm is the same for the two problems we consider in the next two sections, differing only on the way problem SMALL is solved.

PROBLEM SMALL: Given an instance I for G-KNAPSACK, where all items are of the same class and given a value w, find a solution Q of I with value w and smallest size.

We say that an algorithm SS_ϵ is ϵ-*relaxed* for problem SMALL if given an instance I and value w the algorithm generates a solution Q with $(1-\epsilon)w \leq v(Q) \leq w$ and $s(Q) \leq s(O)$, where O is a solution with value w and smallest size. Such solution Q is called an ϵ-*relaxed* solution.

In figure 9.2 we present an approximation scheme for G-KNAPSACK using a subroutine to solve problem SMALL. The algorithm uses a rounding idea presented by Ibarra and Kim [5] for the knapsack problem.

In steps 1–3 the original instance is reparameterized in such a way that each new item value v'_e, is a non-negative integer value bounded by $\lceil n/\epsilon \rceil$. Therefore, the value of any solution is bounded by $V = O(n^2/\epsilon)$. This leads to a polynomial time algorithm using a dynamic programming approach with only $O(\epsilon)$ loss on the total value of the solution found by the algorithm.

In steps 4–8, the algorithm generates ϵ-relaxed solutions for each problem SMALL obtained from the reparameterized instances of each class and each possible value w. The solutions are stored in variables $A_{j,k}$, for each class j and each possible value k.

In steps 9–14 problem G-KNAPSACK is solved using dynamic programming. We have table $T_{j,w}$ indexed by classes j and all possible values w. It stores the smaller solution using items of classes $\{1, \ldots, j\}$ that have value w. The basic idea is to solve the following recurrence:

$$T_{j,w} := \min\{T_{j-1,w}, A_{j,w}, \min_{1 \leq k < w}\{T_{j-1,k} + A_{j,w-k}\}\}.$$

Finally, given that there are m classes, in steps 16–17 a solution generated with maximum value w is returned.

To prove that G_ϵ is an approximation scheme we consider that algorithm SS_ϵ, used as subroutine, is an ϵ-relaxed algorithm for problem SMALL.

Lemma 9.2.2 *If the algorithm G_ϵ uses an ϵ-relaxed algorithm as subroutine and if $O_{j,w}$ is a*

ALGORITHM $\mathrm{G}_\epsilon(I)$ where $I = (S, c, s, v, K, d, d_{\max}, d_{\min})$
Subroutine: SS_ϵ (ϵ-relaxed algorithm for problem SMALL).

1. *% reparameterize the instance by value*
2. let $P \leftarrow \max\{v_e : e \in S\}$, $L \leftarrow \epsilon P/n$ and $V \leftarrow \lceil n^2/\epsilon \rceil$, where $n = |S|$
3. for each item $e \in S$ do $v'_e \leftarrow \lfloor \frac{v_e}{L} \rfloor$
4. *% generate an ϵ-relaxed solution for each class*
5. for class $j \leftarrow 1$ to m do
6. for value $w \leftarrow 1$ to V do
7. let S_j be the set of items in S with class j
8. $A_{j,w} \leftarrow SS_\epsilon(S_j, s, v', K, d, d_{\max}, d_{\min}, w)$
9. *% for each possible value w, find solution for* G-KNAPSACK *with classes* $\{1, \ldots, j\}$
10. for value $w \leftarrow 1$ to V do
11. $T_{1,w} \leftarrow A_{1,w}$
12. for class $j \leftarrow 2$ to m do
13. for value $w \leftarrow 1$ to V do
14. $T_{j,w} \leftarrow$ (solution in $\{T_{j-1,w}, A_{j,w}, \min_{1 \le k < w}\{T_{j-1,k} + A_{j,w-k}\}\}$
of value in $[(1-\epsilon)w, w]$ and minimum size)
15. let Q be the solution $T_{m,w}$, $1 \le w \le V$ with maximum value w and size $s(Q) \le K$
16. return Q.

Figura 9.2: Generic algorithm for G-KNAPSACK using subroutine for problem SMALL

solution using classes $\{1, \ldots, j\}$, with $w := v'(O_{j,w})$ and minimum size, then $T_{j,w}$ exists and $v'(T_{j,w}) \ge (1-\epsilon)w$ and $s(T_{j,w}) \le s(O_{j,w})$.

Proof. First, note that if a solution $O_{j,w}$ with value $w := v'(O_{j,w})$ using items $\{1, \ldots, j\}$ exists, then a solution for $T_{j,w}$ also exists. We can prove this fact by induction on the number of classes. The base case consider only items with class 1 and can be proved from the fact that subroutine SS_ϵ is an ϵ-relaxed algorithm, (that is, $T_{1,w} = A_{1,w}$).

Consider a solution $O_{j,w}$ with value $w := v'(O_{j,w})$ using items $\{1, \ldots, j\}$.

If $O_{j,w}$ uses only items of class j, then we have a solution $A_{j,w}$ which is obtained from subroutine SS_ϵ, which by assumption is an ϵ-relaxed algorithm. Therefore, $v'(A_{j,w}) \ge (1-\epsilon)v'(O_{j,w})$ and $s(A_{j,w}) \le s(O_{j,w})$.

If $O_{j,w}$ uses only items of classes $1, \ldots, j-1$, by induction, $T_{j-1,w}$ exists and $v'(T_{j-1,w}) \ge (1-\epsilon)v'(O_{j,w})$ and $s(T_{j-1,w}) \le s(O_{j,w})$.

If $O_{j,w} = (S, P)$ uses items of class j and items of other classes, denote by $O_1 = (S_1, P_1)$ and $O_2 = (S_2, P_2)$ two solutions obtained partitioning $O_{j,w}$ such that P_1 contains all parts of P with items of class different than j and P_2 contains all parts with items of class j. Let

$k := v'(O_1)$. By induction, there are solutions $T_{j-1,k}$ and $A_{j,w-k}$ such that

$$v'(T_{j-1,k}) + v'(A_{j,w-k}) \geq (1-\epsilon)k + (1-\epsilon)(w-k) = (1-\epsilon)v'(O_{j,w}) \quad \text{and}$$

$$s(T_{j-1,k}) + s(A_{j,w-k}) \leq s(O_1) + s(O_2) = s(O_{j,w}).$$

□

Theorem 9.2.3 *If I is an instance for* G-KNAPSACK *and SS_ϵ is an ϵ-relaxed polynomial time algorithm for* SMALL *then the algorithm G_ϵ with subroutine SS_ϵ is a polynomial time approximation scheme for* G-KNAPSACK. *Moreover, if SS_ϵ is also polynomial time in $1/\epsilon$, the algorithm G_ϵ is a fully polynomial time approximation scheme.*

Proof. Let O be an optimum solution for instance I. Let w be the value of an optimal solution considering the rounded items.

$$\begin{aligned}
v(T_{m,w}) &= \sum_{e \in T_{m,w}} v_e \geq \sum_{e \in T_{m,w}} v'_e L = L v'(T_{m,w}) \\
&\geq L(1-\epsilon)w \geq L(1-\epsilon)\sum_{e \in O} v'_e \\
&\geq L(1-\epsilon)\sum_{e \in O}(\frac{v_e}{L} - 1) \geq L(1-\epsilon)(\sum_{e \in O}\frac{v_e}{L} - n) \\
&= (1-\epsilon)(\sum_{e \in O} v_e - nL) = (1-\epsilon)(\mathrm{OPT} - \epsilon P) \\
&\geq (1-\epsilon)(\mathrm{OPT} - \epsilon \mathrm{OPT}) \\
&\geq (1-2\epsilon)\mathrm{OPT}
\end{aligned}$$

Since the solution returned by the algorithm G_ϵ has value at least $v(T_{m,w})$, we have that $G_\epsilon(I) \geq (1-2\epsilon)\mathrm{OPT}$.

If m and n are the number of classes and the number of items, respectively, the time complexity of the algorithm G_ϵ is $O(mn^4/\epsilon^2 + mn^2/\epsilon \cdot T_{SS})$, where T_{SS} is the time complexity of subroutine SS_ϵ. Therefore, if SS_ϵ is polynomial time in n (and in $1/\epsilon$) then algorithm G_ϵ is a (fully) polynomial time approximation scheme. □

In the next two sections, we present two approximations schemes for restricted versions of problem G-KNAPSACK. The approximations schemes are based on algorithm G_ϵ and differ only in the way problem SMALL is solved. From now on we consider the items after the rounding process.

9.2.3 The FPTAS

In this section, we consider the G-KNAPSACK problem restricted to k different item sizes in each class. We present a fully polynomial time approximation scheme. The algorithm is

the same presented in the previous section. In this case, we only need to present an ϵ-relaxed algorithm, used as subroutine by the algorithm G_ϵ, that is polynomial time both in the input size and in $1/\epsilon$. In fact, we show that an algorithm for problem SMALL does not need to compute solutions for every value w to obtain a fully polynomial time approximation scheme for problem G-KNAPSACK. Given such a subroutine, the approximation result follows from theorem 9.2.3.

The next result state the difficulty of the case we are considering.

Theorem 9.2.4 *The problem with the restriction that we have at most a constant k of different sizes in each class is still NP-hard.*

Proof. The theorem is valid since the knapsack problem is a particular case when each item is of a different class and $d_{\max} = \infty$, $d_{\min} = 0$ and $d = 0$. □

The k-PACK problem

Before presenting the algorithm to solve problem SMALL, consider the problem, denoted by k-PACK, which consists in packing n one-dimensional items with at most k different item sizes into the minimum number of bins of size $d_{\max}$, each bin filled by at least $d_{\min}$.

The algorithm to solve problem k-PACK uses a dynamic programming strategy combined with the generation of all configurations of packing items into one bin. In the figure 9.3 we present the algorithm that generates a function B that returns the minimum number of bins to pack an input list, under the restrictions of the problem k-PACK. For our purposes, we also need that the function B returns the partition of the input list into bins. For simplicity, we let to the interested reader its conversion to an algorithm that also returns the partition of the input list into bins.

In steps 1–4, the algorithm generates all possible subset of items that can be packed in one bin. In each iteration of the while command, steps 5–11, the algorithm uses the knowledge of instances that uses i bins to compute instances that uses $i + 1$ bins.

The following theorem is straightforward.

Theorem 9.2.5 *The algorithm P_k generates a function that returns the minimum number of bins to pack any sublist of the input list L of problem k-PACK. Moreover, the algorithm P_k has a polynomial time complexity.*

Solving problem SMALL

The following lemma states the relationship of a solution for problem SMALL and the problem k-PACK.

Lemma 9.2.6 *If $O = (L, P)$ is an optimum solution of an instance $I = (S, s, v', K, d, d_{\max}, d_{\min}, w)$ for the problem SMALL, $L \subseteq S$ and $P = (P_1, \ldots, P_k)$ then $k \geq B(L)$, where $B(L)$ is the minimum number of bins to pack L into bins of size $d_{\max}$, filled by at least $d_{\min}$.*

ALGORITHM $\mathrm{P}_k(L, d_{\min}, d_{\max})$

1. let $s_1, \ldots, s_k$ the k different sizes occurring in list L,
2. let d_i be the number of items in L of size s_i, $i = 1, \ldots, k$,
3. let Q_1 be the set of all tuples $(q_1, \ldots, q_k)$ such that $0 \leq q_i \leq d_i$, $i = 1, \ldots, k$
4. and $d_{\min} \leq \sum_{i=1}^{k} q_i s_i \leq d_{\max}$.
5. let $i \leftarrow 1$
6. while $(d_1, \ldots, d_k) \notin Q_i$ do
7. $Q_{i+1} \leftarrow \emptyset$
8. for each $q' \in Q_1$ and $q'' \in Q_i$ do
9. $q \leftarrow q' + q''$
10. if $q \notin Q_i$ and $(q_1, \ldots, q_k) \leq (d_1, \ldots, d_k)$ then $Q_{i+1} \leftarrow Q_{i+1} + q$
11. $i \leftarrow i + 1$
12. let $B(q) \leftarrow j$ for all $q \in Q_j$, $1 \leq j \leq i$
13. return B

Figura 9.3: Algorithm to find the minimum number of bins to pack any subset of L

Proof. Note that items packed in the optimum solution O are separated by wall divisions of size d and two subsequent wall divisions defining a compartment must be a distance between $d_{\min}$ and $d_{\max}$. Items in compartments can be considered as a packing into bins of size $d_{\max}$ occupied by at least $d_{\min}$. Since $B(L)$ is the minimum number of such bins to pack L, the number of compartments of L is at least $B(L)$. □

Corollary 9.2.7 *If $O = (L, P)$ is an optimum solution of an instance $I = (S, s, v', K, d, d_{\max}, d_{\min}, w)$ for problem* SMALL*, then $s(O) \geq s(L) + B(L)d$.*

In figure 9.4, we present an algorithm for solving a relaxed version of problem SMALL, which is sufficient to our purposes. The algorithm first consider all possible configurations of solutions for problem SMALL, without considering the value of each item. This step is performed by a subroutine to solve problem k-PACK. Instead of finding each possible attribution of values for each configuration, the algorithm only generates valid configurations with maximum value. For a given value w, the algorithm only returns a solution if the value is a maximum value for some configuration. Notice that we return the smallest packing that have the given value.

Theorem 9.2.8 *If I is an instance for* G-KNAPSACK *with at most k different item sizes and algorithm G_ϵ is executed with subroutine k-SS then $G_\epsilon(I) \geq (1 - \epsilon)\mathrm{OPT}(I)$.*

Proof. Consider an optimum solution O for the instance I with the rounded items . Let Q_c be the set of items of class c used in this optimal solution. This set of items corresponds to a

---

ALGORITHM k-$SS(S, s, v', K, d, d_{\max}, d_{\min}, w)$
Subroutine: P_k (Subroutine to solve problem k-PACK).

1. let $B \leftarrow P_k(S, d_{\min}, d_{\max})$
2. let $s_1, \ldots, s_k$ the k different sizes occurring in list L,
3. let d_i be the number of items in L of size s_i, $i = 1, \ldots, k$,
4. let Q be the set of all tuples $(q_1, \ldots, q_k)$ such that $0 \leq q_i \leq d_i$, $i = 1, \ldots, k$
5. for each $q = (q_1, \ldots, q_k) \in Q$ do
6. let $P(q)$ the packing obtained using function $B(q)$ placing for each size s_j,
7. i_j items of S with size s_j and greatest values
8. let $Q_w \leftarrow \{q \in Q : w = v(P(q))\}$
9. if $Q_w \neq \emptyset$ then
10. return $q \in Q_w$ such that $s(q)$ is minimum.
11. else
12. return $\emptyset$

Figura 9.4: Algorithm to find the minimum number of bins to pack any subset of L

configuration q_c that is packed optimally by corollary 9.2.7. In algorithm k-SS we return items of maximum value corresponding to this configuration. If k-SS does not return this optimal solution to G_ϵ, it is because it finds another configuration with the same value but with smaller size in line 10 of the algorithm. It follows from theorem 9.2.3 that the optimal solution found by algorithm G_ϵ with k-SS is a FPTAS to G-KNAPSACK. □

9.2.4 The PTAS

In this section, we present an algorithm to solve the problem SMALL with the restriction that $d_{\min} = 0$. This restriction is based in our weakness to find packings with a minimum filling. In section 9.2.5 we present an inaproximality result that show that the full version of G-KNAPSACK problem can not be approximated unless $P = NP$.

The algorithm of this section uses some nice ideas proposed by Chekuri and Khanna [2]. Given an instance $I = (S, s, v', K, d, d_{\max}, d_{\min}, w)$ for the problem SMALL, the algorithm performs two basic steps. First, the algorithm finds a set $U \subseteq S$ with $v'(U) \geq (1 - O(\epsilon))w$ such that its packing has size no bigger than an optimal packing of value w. This is shown in the next subsection. In the second step, the algorithm packs a set $U' \subseteq U$ such that $v(U') \geq (1 - O(\epsilon))v'(U)$.

Finding the Items

Given an instance I we first have to find a subset of the items with total value very close to w. The algorithm, denoted by $Find$, is presented in figure 9.5. It finds a polynomial number of sets, such that at least one has value close to w and its packing size is at most the size of the packing of the optimal solution.

ALGORITHM $Find(S, \epsilon, w)$
Subroutine: $Round$ (Subroutine to round the values of the items).

1. for each $e \in S$ do
2. $\quad v''_e \leftarrow (1+\epsilon)^k$ where $(1+\epsilon)^k \leq v'_e < (1+\epsilon)^{k+1}$
3. let $h \leftarrow \lfloor \log \frac{n}{\epsilon} \rfloor$
4. for each $i \in \{0, \ldots, h\}$ do
5. $\quad$ let S_i be the set of items with value $(1+\epsilon)^i$ in S
6. $\quad$ let $(e^i_1, \ldots, e^i_{n_i})$ the items in S_i sorted in non-decreasing order of size
7. let T be the set of all possible tuples $(k_1, \ldots, k_h)$
such that $k_i \in \{0, \ldots, \frac{h}{\epsilon}\}, 0 \leq i \leq h$ and $\sum_{i=0}^{h} k_i = \frac{h}{\epsilon}$.
8. for each value w' in the interval $[(1-\epsilon)w, w]$ do
9. $\quad$ for each tuple $(k_1, \ldots, k_h)$ in T do
10. $\quad\quad$ for each $i \in \{0, \ldots, h\}$ do
11. $\quad\quad\quad$ let $U_i = \{e^i_1, \ldots, e^i_j\}$ such that $v(\{e^i_1, \ldots, e^i_{j-1}\}) < k_i(\frac{\epsilon w'}{h}) \leq v(\{e^i_1, \ldots, e^i_j\})$
12. $\quad\quad$ $U \leftarrow (U_1 \cup \ldots \cup U_h)$
13. $\quad\quad$ $Q \leftarrow Q + U$.
14. return Q.

Figura 9.5: Algorithm to find sets with value very close to a given value w

In the first step, the algorithm $Find$ round down each item value to the nearest power of $(1+\epsilon)$. From now on, consider the items with the new values. Given a set O, with $v'(O) = w$ and smallest size, we have with the rounding that $\frac{w}{(1+\epsilon)} \leq v''(O) \leq w$. With the rounding specified we have $h = \lfloor \log \frac{n}{\epsilon} \rfloor$ different values of items. The items are grouped by their values in sets $S_1, \ldots, S_h$, such that items in the same set have the same value.

Instead of looking for the sets $O_i = O \cap S_i, i = 1, \ldots, h$, the algorithm $Find$, finds subsets U_i with values very close to $v'(O_i)$ whose packing is not larger than an optimal packing of O.

For each index i, the algorithm finds an integer value $k_i \in \{0, \ldots, \frac{h}{\epsilon}\}$ such that $k_i(\frac{\epsilon w}{h}) \leq v''(O_i) < (k_i+1)(\frac{\epsilon w}{h})$. The algorithm generate all possible tuples $(k_1, \ldots, k_h)$ and one will satisfy this condition. First we show that such values k_i exist in such a way that they are very close to the optimal.

Lemma 9.2.9 *If $O \subseteq S$ is a set with $v'(O) = w$ and smallest size, then there exists a valid*

tuple $(k_1, \ldots, k_h)$ *such that for each* i *we have* $k_i \in \{0, \ldots, \frac{h}{\epsilon}\}$ *and* $\sum_{i=1}^{k} k_i(\frac{\epsilon w}{h}) \geq (1-\epsilon)w$.

Proof. Let $O = O_1 \cup \ldots \cup O_h$ where $O_i = S_i \cap O, 1 \leq i \leq h$. For each i, let $k_i = \lfloor \frac{v''(O_i)h}{\epsilon w} \rfloor$ and we have:

$$k_i(\frac{\epsilon w}{h}) = \lfloor \frac{v''(O_i)h}{\epsilon w} \rfloor (\frac{\epsilon w}{h}) \geq (\frac{v''(O_i)h}{\epsilon w} - 1)\frac{\epsilon w}{h} = v''(O_i) - \frac{\epsilon w}{h}.$$

Thus, for each i we loose at most $\frac{\epsilon w}{h}$, so the total loss is ϵw.

□

The first idea is to test all possible values k_i, but this does not leads to a polynomial time algorithm. In this case we have $O((\log n)^{\log n})$ possibilities to test. To limit the number of tests polynomially, we use the fact that $\sum_i k_i \leq \frac{h}{\epsilon}$. Note that if this condition is not satisfied we would obtain sets with values bigger than w. The next two lemmas, presented by Chekuri and Khanna [2], limit the number of different tuples to be considered.

Lemma 9.2.10 *Let* f *be the number of* g*-tuples of non negative integer values such that the sum of the values of the tuple is* d*. Then*

$$f = \binom{d+g-1}{g-1}$$

Lemma 9.2.11 *The number of different tuples of index* k_i *that we have to test is* $O(n^{\frac{1}{\epsilon}})$.

Proof. Let $d = \frac{h}{\epsilon}$ and $g = h$. From lemma 4.2 we know that the number of possibilities to test is

$$f = \binom{d+g-1}{g-1}.$$

That is,

$$f = \frac{(d+g-1)!}{(g-1)!d!} = \frac{(d+g-1)\ldots(d+1)}{(g-1)!} \leq \frac{(d+g-1)^{g-1}}{(g-1)!}.$$

Using the Stirling approximation for the factorial we have $(g-1)! \geq (\frac{(g-1)^{g-1}}{e})$. Therefore,

$$f \leq (\frac{e(d+g-1)}{g-1})^{g-1}.$$

Since $h + \frac{h}{\epsilon} \leq \alpha(\frac{h}{\epsilon} - 1)$ for some constant α, we have the new limit:

$$f \leq (\frac{e\alpha(g-1)}{g-1})^{g-1} = (e\alpha)^{g-1} = O(n^{\frac{1}{\epsilon}}).$$

□

The enumeration of the tuples is done in step 7 of the algorithm and is used in the loop of step 9. Notice that we have a polynomial number of tuples. Now we show how the algorithm get the items. Given a set S_i and a value k_i the algorithm take items as follows:

1. In step 6 the algorithm sort the set S_i in non-decreasing order of items size.

2. Items are taken in non-decreasing order of size until the total value of the items becomes bigger or equal than $k_i(\frac{\epsilon w}{h})$ in step 11.

The next lemma state that at least one of the sets obtained this way have value very close to the optimal and that its packing size is not bigger than the packing size of the optimal set O.

Lemma 9.2.12 *Let O be a solution with value w and smallest size and $O_1, \ldots, O_h$ the partition of O such that $O_i = O \cap S_i, i = 1, \ldots, h$. There are values $k_1, \ldots, k_h$ and sets $U_1, \ldots, U_h$ obtained by the algorithm $Find$, such that $v(U_1 \cup \ldots \cup U_h) \geq v(O)(1 - O(\epsilon))$ and that the packing size of the set U is not larger than the packing size of O.*

Proof. Consider an integer $i, 0 \leq i \leq h$. From lemma 4.1 there exists an integer k_i such that $k_i(\frac{\epsilon w}{h}) \leq v''(O_i) < (k_i+1)(\frac{\epsilon w}{h})$. Since the algorithm $Find$ tests all possibilities of k_i, one tuple satisfy the above condition. In step 11 the algorithm take items $e_i \in S_i$ that can fall in one of this two cases:

1) Suppose that $v''_{e_i} \leq \frac{\epsilon w}{h}$. The algorithm take items until their total value become greater or equal than $k_i(\frac{\epsilon w}{h})$. Taking items this way the loose in value is not more than $\frac{\epsilon w}{h}$ because $v''(O_i) < (k_i + 1)(\frac{\epsilon w}{h})$. If all the sets O_i fall in this case the total loss is at most ϵw.

2) Suppose that $v''_{e_i} > \frac{\epsilon w}{h}$. The algorithm take items until their total value become greater or equal than $k_i(\frac{\epsilon w}{h})$. We know that $v''_{e_i} > \frac{\epsilon w}{h}$, and in this case there is no loss because the algorithm take exactly the value of $v''(O_i)$.

Note that all items in the set O_i have the same value and so $v''(O_i)$ is multiple of the value of the items. The algorithm never take more items than the set O_i, *i.e.*, $|U_i| \leq |O_i|$. The algorithm takes the items in non-decreasing order of size obtaining a packing with size not larger than the packing of the items in O_i. ∎

Remind that O corresponds to a set such that $v'(O) = w$ before the rounding in step 1 of the algorithm $Find$ and that its packing size is minimum. After the rounding we have $\frac{w}{(1+\epsilon)} \leq v''(O) \leq w$ wich implies that $v''(O) \in [(1-\epsilon)w, w]$ since

$$w - \frac{w}{(1+\epsilon)} = \frac{\epsilon w}{1+\epsilon} \leq \epsilon w.$$

If we test all integer possible values w' in the interval $[(1-\epsilon)w, w]$ we have at most ϵw possibilities. Since the maximum value of w is $O(\frac{n^2}{\epsilon})$ we have at most $O(n^2)$ possible tests. If we take $w' = \lceil v''(O) \rceil$ this value satisfy:

$$k_i \frac{\epsilon w'}{h} \leq v''(O_i) < (k_i + 1)\frac{\epsilon w'}{h}, \quad 0 \leq i \leq h.$$

According to lemma 9.2.12 we have a loss of at most $\epsilon w'$ which corresponds to a small fraction of the value of O.

$$\epsilon w' \leq \epsilon(v''(O)+1) \leq 2\epsilon v''(O)$$

The search for the value of w' corresponds to the loop in step 8 of the algorithm.

At last, the algorithm generate a polynomial number of sets, at least one with value $v = (1 - O(\epsilon))w$ and that can be packed optimally. In the next section we see how these sets can be packed.

Packing the Items

In the previous section we have obtained at least one set U of items such that its total value is very close to the optimal O and that its packing size is not larger than the packing size of O. The packing is obtained using known algorithms for bin packing. Notice that an optimal bin packing using bins of size $d_{\max}$ is a limit for the optimal packing in compartments. This is because we choose items of smaller size and possibly less items than the optimal set. Also notice that packing the items with this algorithm we can respect only the maximum limit of a compartment, $d_{\max}$.

Fernadez de la Vega and Lueker [7] have shown how to build an algorithm that pack items using at most $(1+\epsilon)OPT + 1$ bins. Frieze and Clarke [4] found a PTAS for the problem of packing a set of items in m bins maximizing the total value of the packed items, where m is a constant.

We call by A_{FVL} the algorithm of Fernadez de la Vega and Lueker, and by A_{FC} the algorithm of Frieze and Clarke. The algorithm $Pack$, to pack a set U, is presented in figure 9.6. It first uses the algorithm A_{FVL} to pack the items in U. If the number of bins used is smaller than $\frac{1}{\epsilon} + 2$ it just forget the packing and pack the set using the algorithm A_{FC}. In the other case we just take the OPT' bins with biggest value. We denote by $size(N)$ the number of bins used in packing N.

Lemma 9.2.13 *If P is the solution corresponding to the packing generated by the algorithm $Pack$ over the set U then $v(P) \geq (1 - O(\epsilon))v(U)$ and its size is smaller than the optimal packing.*

Proof. If $OPT' \geq \frac{1}{\epsilon}$ then we loose the value of $\epsilon OPT' + 1$ bins. Of course $OPT' \leq OPT$, where OPT is the minimum number of bins to pack the items in U. Each one of the bins have value at most $\frac{v(U)}{size(N)}$. Then we loose at most

$$\frac{v(U)}{size(N)}(\epsilon OPT' + 1) \leq \frac{v(U)}{OPT}(\epsilon OPT + 1) = \epsilon v(U) + \frac{v(U)}{OPT}.$$

Since $OPT \geq OPT' > \frac{1}{\epsilon}$ an upper bound for the loose is $2\epsilon v(U)$.

Therefore, $v(P) \geq (1 - 2\epsilon)v(U)$.

ALGORITHM $Pack(U)$
Subroutine: A_{FVL} and A_{FC} (Subroutines to pack the items).

1. let $N \leftarrow A_{FVL}(U)$
2. let $OPT' \leftarrow \frac{size(N)-1}{1+\epsilon}$
3. if $OPT' \geq \frac{1}{\epsilon}$ then
4. let P be the OPT' bins with biggest values in N
5. else
6. $P_{FC} \leftarrow \emptyset$
7. for $j \leftarrow 1$ to $\frac{1}{\epsilon} + 2$ do
8. $P_{FC} \leftarrow P_{FC} + A_{FC}(U, j)$
9. Let P be the smaller packing in P_{FC} such that $v(P) \geq (1-\epsilon)v(U)$
10. return P

Figura 9.6: Algorithm to pack the items

If $OPT' < \frac{1}{\epsilon}$ then we have that $size(N) \leq \frac{1}{\epsilon} + 2$. In this case P is obtained running the algorithm A_{FC} with j bins $1 \leq j \leq \frac{1}{\epsilon} + 2$. We take the smaller packing of the items such that the total loose is at most $\epsilon v(U)$. □

The ϵ-relaxed algorithm

In this section we present the ϵ-relaxed algorithm for the problem SMALL with the restriction that $d_{\min} = 0$. The algorithm is presented in figure 9.7. It is very simple and just use the algorithms presented in sections 4.1 and 4.2.

ALGORITHM $Small(S, s, v, K, d, d_{\max}, 0, w)$
Subroutine: $Find$ and $Pack$ (Subroutines of the previous sections).

1. let $Q \leftarrow Find(S, \epsilon, w)$
2. $MIN \leftarrow \infty$
3. for each $U \in Q$ do
4. $M \leftarrow Pack(U)$
5. if $size(M) < MIN$ and $v(M) > (1-(5+2)\epsilon)w$ then
6. $M2 \leftarrow M$
7. $MIN \leftarrow size(M)$
8. return $M2$

Figura 9.7: Algorithm to solve Small Problem

Given a value w, the algorithm $Find$ generates a polynomial number of sets U as shown in section 4.1. At least one of the sets have value very close to the value of the optimal set O

and have a packing of size at most the size of the optimal. For all possibilities of U we pack it with algorithm $Pack$. The algorithm $Find$ generate a loss of at most 5ϵ. The algorithm $Pack$ generate a loss of at most 2ϵ. The solution returned by the algorithm is the packing of smaller size such that the loose is at most $(2+5)\epsilon w$. We can conclude with the theorem:

Theorem 9.2.14 *Algorithm G_ϵ is a PTAS to G-Knapsack if $d_{\min} = 0$, when executed with the ϵ-relaxed algorithm $Small$ of this section.*

9.2.5 Inaproximality of the G-KNAPSACK problem

In this section we present an inaproximality result for the G-KNAPSACK problem. We prove that the full version of the problem where $0 < d_{\min} \leq d_{\max}$ can not be approximated to any factor unless $P = NP$. The proof is made reducing the partition problem, with instance I_1 and items with total size K, to an instance of the G-KNAPSACK problem with the same set of items and $d_{\min} = d_{\max} = \frac{K}{2}$. It is not hard to see that with this instance, G-KNAPSACK problem have only two possible solutions, one with size 0 and other with size $\frac{K}{2}$. The last solution is obtained if and only if instance I_1 has a solution.

We can also prove that the G-KNAPSACK can not be approximated when $0 < d_{\min} < d_{\max}$. The next theorem states this result.

Theorem 9.2.15 *There is a gap-introducing reduction transforming instance I_1 of the Partition Problem (PP) to an instance I_2 of the* G-KNAPSACK *problem such that:*

- *If instance I_1 is satisfiable then $OPT(I_2) = d_{\min}$, and*
- *if I_1 is not satisfiable, $OPT(I_2) = 0$.*

Proof. Let $I_1 = (S, s)$ be the instance of Partition Problem where each item $e \in S$ has size s_e. We construct an instance I_2 such that $OPT(I_2) \in \{0, d_{\min}\}$. Let $I_2 = (S, c, s', v, K, 0, d_{\max}, d_{\min})$ be the instance of G-KNAPSACK problem. For each item $e \in S$ we have that $v_e = s_e \cdot \alpha$ and $s'_e = s_e \cdot \alpha$, where α is an integer constant. Of course the partition problem is satisfiable with instance (S, s) if and only if it is satisfiable with instance (S, s'). All items in S belongs to the same class. Let $d_{\min} = \frac{\sum_{e \in S} s'_e}{2}$ and $d_{\max} = d_{\min} + (\alpha - 1)$. Let $K = d_{\max}$. Notice that there is no wall divisions.

First note that any size s'_e is multiple of α and therefore, any solution of I_2 is also multiple of α. Since $d_{\min}$ is multiple of α, we have

$$d_{\min} < d_{\max} < d_{\min} + \alpha$$

and we conclude that there is no solution to instance I_2 with size greater than $d_{\min}$.

If instance I_1 is satisfiable then the optimal solution of instance I_2 has value $d_{\min}$ and the knapsack is filled until $d_{\min}$. If instance I_2 is not satisfiable then the only solution with size multiple of α that respects the limits $d_{\min}$ and $d_{\max}$ has value 0 and it packs no items.

□

Corollary 9.2.16 *There is no r-approximation algorithm for* G-KNAPSACK *problem when* $0 < d_{\min} \leq d_{\max}$ *, $r > 0$, unless $P = NP$.*

9.2.6 Conclusion

We present approximation algorithms to some restricted versions of G-knapsack. We also prove that the full version of G-KNAPSACK problem cannot be approximated. The problem has a practical motivation in the iron and steel industry where a problem of cutting rolls appears.

9.2.7 Bibliography

1. A. Caprara, H. Kellerer, U. Pferschy and D. Pisinger. Approximation Algorithms for Knapsack Problems with Cardinality Constraints. *European Journal of Operational Research*, 123:333–345, 2000.

2. C. Chekuri and S. Khanna. A ptas for the multiple knapsack problem. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 213–222, 2000.

3. J.S. Ferreira, M.A. Neves, P. Fonseca, and P.F. Castro. A two-phase roll cutting problem. *European Journal of Operational Research*, 44:185–196, 1990.

4. A. M. Frieze and M. R. B. Clarke. Approximation algorithms for the m-dimensional 0-1 knapsack problem: worst-case and probabilistic analyses. *European Journal of Operational Research*, 15:100–9, 1984.

5. O.H. Ibarra and C.E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the Association for Computing Machinery*, 22:463–468, 1975.

6. H. Shachnai and T. Tamir. Polynomial time approximation schemes for class-constrained packing problems. *Proc. of the 3rd International Workshop on Approximation Algorithms for Combinatorial Optimization*, (APPROX) 2000.

7. W. Fernandez de la Vega and G.S. Lueker. Bin packing can be solved within $1 + \epsilon$ in linear time. *Combinatorica*, 1(4):349–355, 1981.

Capítulo 10

Conclusão

Neste trabalho apresentamos algumas técnicas envolvidas no desenvolvimento de algoritmos de aproximação. As técnicas apresentadas foram exemplificadas através de problemas de escalonamento com algoritmos vistos em alguns dos artigos estudados durante o mestrado. Apresentamos algoritmos que achamos destacar bem a técnica empregada ou possuem alguma característica que achamos ser relevante. Esperamos desta forma, dar uma boa visão da área de algoritmos de aproximação apesar de usarmos apenas algoritmos para problemas de escalonamento. Apresentamos também, um resumo com os melhores resultados para cada problema de escalonamento. Os resultados são baseados nos artigos estudados, por isso não esperamos que o resumo esteja completo e contenha os resultados mais recentes. Além disso, apresentamos dois artigos. O primeiro é uma comparação prática entre alguns algoritmos de aproximação. O segundo são algoritmos de aproximação para casos particulares de uma variação do problema da mochila.

Analisando os resultados práticos dos algoritmos implementados, percebemos que estes atingem valores bem mais próximo do ótimo do que o limite de seus fatores de aproximação. Isto mostra que apesar da área ser alavancada por resultados teóricos, muitos dos algoritmos podem ser utilizados na prática, até mesmo aqueles com fatores de aproximação elevados. O algoritmo PSWapresentado no capítulo 8 por exemplo, possui fator de aproximação igual a 6, mas analisando os resultados práticos para o problema $P|r_j|\sum C_j$, podemos ver que as soluções geradas estão com um fator de aproximação para um limitante do ótimo, variando de 1.15 até 1.28. Vale ressaltar que o limitante para o ótimo é o resultado fracionário de um programa linear, que mesmo sendo resolvido de forma inteira, ainda assim seria um limitante para o escalonamento ótimo. Logo, nossa intuição é que os resultados obtidos por estes algoritmos sejam muito próximos do ótimo. Os algoritmos executados para o problema $R||\sum w_jC_j$ por exemplo, atingem na prática fatores de aproximação que variam entre 1.0001 a 1.07, quando comparados com um limitante inferior para o ótimo. Isto mostra que algortimos de aproximação podem ser utilizados na prática como heurísticas ou mesmo como geradores de limitantes superiores para

algoritmos exatos baseados em métodos como *branch-and-cut*, *branch-and-bound* etc.

Finalmente, apresentamos um PTAS um FPTAS para casos particulares de um problema que é uma generalização do problema da mochila. Tal problema, definido no capítulo 9, tem aplicações em escalonamento. O problema foi primeiramente estudado de forma prática por Ferreira, Neves, Fonseca e Castro [15] através de heurísticas. O problema tem aplicações práticas na indústria metalúrgica na área de processamento de rolos de metais. Até onde sabemos, tal problema não tinha sido estudado com o foco de algoritmos de aproximação. Mostramos que o problema não pode ser aproximado a menos que $P = NP$, e apresentamos um PTAS e um FPTAS para dois casos particulares do problema. Os algoritmos que apresentamos têm complexidade de tempo relativamente alta. Portanto, seria interessante apresentar novos algoritmos aproximados, que mesmo com fatores de aproximação maiores, tenham complexidade de tempo aceitável na prática.

Referências Bibliográficas

[1] F. Afrati, E. Bambis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Skutella, C. Stein, and M. Srividenko. Approximation schemes for minimizing average weighted completion time with release dates. In *Proc. of the 40th IEEE Symposium on Foundations of Computer Science*, pages 32–44, 1999.

[2] C. Kenyon A.K. Amoura, E. Bampis and Y. Manoussakis. Scheduling independent multiprocessor tasks. In *Proc. 5th European Symposium on Algorithms*, pages 1–12, 1997.

[3] C. Kenyon A.K. Amoura, E. Bampis and Y. Manoussakis. Scheduling independent multiprocessor tasks. *Algorithmica*, 32:247–261, 2002.

[4] S. Arora and C. Lund. *Hardness of approximations*. PSW Publishing, 1997.

[5] S. Arora, C. Lund, R. Motwani, and M. Szegedy. Proof verification and intractability of approximation problems. In *Proc. 33th IEEE Annual Symposium on Foundations of Computer Sciense*, pages 106–113, 1998.

[6] S. Arora and S. Safra. Probabilistic checking of proofs: a new characterization of NP. In *Proc. 33th IEEE Annual Symposium on Foundations of Computer Sciense*, pages 2–13, 1992.

[7] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and approximation: combinatorial optimization problems and their approximality properties*. Springer-Verlag, 1999.

[8] K. R. Baker. Introduction to sequencing and scheduling. Wiley, 1974.

[9] J.L. Bruno, E.G. Coffman Jr, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17:382–387, 1974.

[10] V. Chavatal. *Linear Programming*. W. M. Freeman and Company, 1983.

[11] F. A. Chudak and D. B. Smhyos. Approximation algorithms for precedence constrained scheduling problems on parallel machines that run at diferent speeds. *Journal of Algorithms*, 30:323–343, 1999.

[12] S. J. Chung and K. G. Murty. Polynomially bounded ellipsoid algorithms for convex quadratic programming. *Nonlinear Programming*, 4:434–485, 1981.

[13] C. E. Ferreira, C. G. Fernandes, F. K. Miyazawa, J. A. R. Soares, J. C. Pina Jr., K. S. Guimarães, M. H. Carvalho, M. R. Cerioli, P. Feofiloff, R. Dahab, and Y. Wakabayashi. *Uma introdução sucinta a algoritmos de aproximação*. Colóquio Brasileiro de Matemática – IMPA, Rio de Janeiro–RJ, 2001.

[14] C. E. Ferreira and Y. Wakabayashi. *Combinatória Poliédrica e Planos-de-Corte Faciais*. 10° Escola de Computação, 1996.

[15] J. S. Ferreira, M. A. Neves, P. Fonseca, and P. F. Castro. A two-phase roll cutting problem. *European Journal of Operational Research*, 44:185–196, 1990.

[16] M. X. Goemans. Semidefinite programming and combinatorial optimization. *Mathematical Programming*, 79:143–161, 1997.

[17] M.X. Goemans and D.P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the Association for Computing Machinery*, 42:1115–1145, 1995.

[18] R.L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[19] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[20] Leslie A. Hall, Andreas S. Schulz, David B. Shmoys, and Joel Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of Operations Research*, 22:513–544, 1997.

[21] Leslie A. Hall, David B. Shmoys, and Joel Wein. Scheduling to minimize average completion time: off-line and on-line algorithms. In *Proc. of the Sixth ACM-SIAM Symposium on Discrete Algorithms*, pages 142–151, January 1996.

[22] D.S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997.

[23] D.S. Hochbaum and D.B. Shmoys. A polynomial approximation scheme for machine scheduling on uniform processors: using the dual approach. *SIAM J. Comp.*, 17:539–551, 1988.

[24] W. Horn. Minimizing average flow time with parallel machines. *Operations Research*, 21:846–847, 1973.

[25] F. Afrati E. Bampis A. V. Fishkin K. Jansen and C. Kenyon. Scheduling to minimize the average completion time of dedicated tasks. *Lecture Notes on Computer Sciense*, pages 454–??, 2000.

[26] K. Jansen and L. Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. In *Proc. of 10th annual ACM-SIAM symposium on Discrete algorithms*, pages 490–498, 1999.

[27] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *Handbooks in Operations Research and Management Science*, volume 4, chapter Sequencing and scheduling: algorithms and complexity, pages 445–522. North Holland, 1993.

[28] J. K. Lenstra, D. B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[29] C. Phillips, C. Stein, and J. Wein. Scheduling jobs that arrive over time. In *Proc. of the 4th Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes on Computer Science*, pages 86–97, 1995.

[30] C. Phillips, C. Stein, and J. Wein. Minimizing average completion time in the presence of release dates. *Mathematical Programming*, 82:199–223, 1998.

[31] A. Schulz and M. Skutella. The power of alpha-points in preemptive single machine scheduling. Technical report, Techische Universitat Berlin, 1999.

[32] A. Schulz and M. Skutella. Scheduling unrelated machines by randomized rounding. *SIAM Journal on Discrete Mathematics*, 15(4):450–469, 2002.

[33] M. Skutella. Semidefinite relaxations for parallel machine scheduling. In *Proc. of the 39th IEEE Symposium on Foundations of Computer Science*, pages 472–481, 1998.

[34] Martin Skutella. Convex quadratic and semidefinite programming relaxations in scheduling. Technical report, Techische Universitat Berlin, 1999.

[35] Martin Skutella and G. Woeginger. A ptas for minimizing total weigthed completion time on identical parallel machines. In *Proc. of the 31th ACM Symposium on the Theory of Computing*, 1999.

[36] W.E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[37] V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2000.

[38] D. Williamson. "Lecture notes on approximation algorithms". Technical Report RC 21409, T.J. Watson Research Center (IBM Research Division), Yorktown Heights, New York, 1998.