Uma proposta de eliminação paralela para o problema da mochila ilimitada

Flavio Regis de Arruda

DISSERTAÇÃO APRESENTADA AO
INSTITUTO DE MATEMÁTICA E
ESTATÍSTICA DA
UNIVERSIDADE DE SÃO PAULO
COMO PARTE DOS REQUISITOS
PARA OBTENÇÃO DO GRAU DE
MESTRE EM CIÊNCIA DA COMPUTAÇÃO

Orientador: Prof Alfredo Goldman vel Lejbman

— São Paulo, 27 de julho de 2004—

Resumo

Um dos problemas mais conhecidos em otimização combinatória é o problema da mochila. Devido a sua importância, diversos autores buscaram formas eficientes de resolvê-lo. Por um lado, diversos estudos da paralelização deste problema foram feitos. Por outro lado, uma das alternativas é a redução do espaço de busca através de técnicas de eliminação de objetos. Neste trabalho nós tentamos unir as duas possibilidades, apresentando algoritmos paralelos para a eliminação de objetos. Para validação de alguns dos nossos algoritmos, eles foram implementados em Java e executados em um aglomerado de computadores.

Abstract

One of the most well known problems in the field of combinatorial optimization is the knapsack problem. Due to its importance, many authors studied efficient ways of solving it. On one hand, many studies presenting parallel solutions have been proposed. On the other hand, one of the alternatives is the reduction of the search space through the objects elimination's technique. In this work we tried to join these two possibilities, presenting parallel algorithms to eliminate redundant terms in knapsack problems. To validate some of our algorithms, they were implemented in Java and executed in a cluster of computers.

Aos meus pais: Maria Aparecida e Joaquim Carlos.

Agradecimentos

Agradeço ao meu orientador, **Alfredo Goldman vel Lejbman**, pela imensa paciência que teve comigo durante estes quase 3 anos de trabalho. Se por inúmeras vezes tive que interromper minha pesquisa em decorrência de problemas pessoais e profissionais, tendo um impacto negativo no planejamento do trabalho, o professor Alfredo sempre demonstrou compreensão, otimismo e lutou por manter-me motivado. Seus comentários também foram decisivos por uma grande melhora na qualidade deste trabalho.

Agradeço a cada um dos meus amigos da **Ala Oeste**, este inseparável grupo de amigos formado durante a graduação no IME, que resiste ao tempo e às mudanças, e me deu muito apoio durante todo o Mestrado. Gostaria de explicitar um agradecimento especial ao **Rodrigo Nonamor Pereira Mariano de Souza** (ou simplesmente: **Souzá**) que esteve ao meu lado nos melhores (e piores) momentos destes quase 3 anos, além de me ajudar na difícil tarefa de conciliar ser um Arquiteto de Sistemas, Professor Universitário e Mestrando. Não bastasse tudo isso, o Souzá foi fundamental na sempre irracionalmente complexa tarefa de cumprir à risca toda burocracia imeana. Rodrigo, este trabalho seria impossível se eu não tivesse podido contar com sua ajuda incondicional.

Duas mulheres contribuíram, e muito, para a conclusão deste trabalho: minha namorada **Luanna** e minha mãe **Maria**. Estas mulheres fazem valer o ditado de que por trás de todo feito masculino, existe sempre grandes mulheres que lhe dão suporte.

*Last, but not least, I would like to thank **Vincent Poirrez**, from IRISA - France, for his help with EDUK algorithm.*

Toda teoria deve ser feita pra ser posta em prática, e toda prática deve obedecer uma teoria. Só os espíritos superficiais desligam a teoria da prática, não olhando que a teoria não é senão uma teoria de uma prática, e a prática não é senão a prática de uma teoria. Quem não sabe nada de um assunto, e consegue alguma coisa nele por sorte ou acaso, chama 'teórico' a quem sabe mais, e, por igual acaso, consegue menos. Quem sabe, mas não sabe aplicar - isto é, quem afinal não sabe, porque não saber aplicar é uma maneira de não saber - tem rancor a quem aplica por instinto, isto é, sem saber que realmente sabe. Mas em ambos os casos, para o homem são de espírito e equilibrado de inteligência, há uma separação abusiva. Na vida superior a teoria e a prática completam-se. Foram feitas uma para a outra.

– Fernando Pessoa, Sociologia do Comércio, Lisboa 1926.

Toda a nossa ciência, comparada com a realidade, é primitiva e infantil - e, no entanto, é a coisa mais preciosa que temos.

– Albert Einstein.

ÍNDICE

1	Introdução	19
1.1	Objetivo do trabalho	19
1.2	Organização do texto	19
2	O problema da mochila e suas variantes	21
2.1	O problema	21
2.2	Variações do problema da mochila	23
2.3	Algoritmos exatos para resolver KP	25
2.3.1	Algoritmos de <i>branch and bound</i>	25
2.3.2	Algoritmos de programação dinâmica	27
2.4	Algoritmos de aproximação e heurísticas	30
2.5	Instâncias do problema da mochila	30
2.5.1	Instâncias não-correlatas	31
2.5.2	Instâncias fracamente correlatas	31
2.5.3	Instâncias fortemente correlatas	32
2.5.4	Instâncias <i>subset-sum</i>	32
3	Relações de dominância e periodicidade	33
3.1	Periodicidade	33
3.2	Relações de dominância	35
3.2.1	Dominância simples	35
3.2.2	Dominância múltipla	36
3.2.3	Dominância coletiva	41
3.2.4	Dominância por <i>threshold</i>	42
3.3	Síntese do capítulo	44
4	Alguns algoritmos sequenciais para resolução do UKP.	47
4.1	Algoritmo básico	47
4.2	Algoritmo MTU2	49
4.2.1	Limitantes superiores para UKP_c^N	50
4.2.2	O algoritmo	52
4.3	Algoritmo de Garfinkel e Nemhauser	57

4.4	Algoritmo EDUK	59
5	Problemas da mochila no contexto do paralelismo	65
5.1	Um algoritmo eficiente para resolução do problema da mochila binária em hipercubos	66
5.2	Solução paralela de UKP usando programação dinâmica em arquitetura sistólica	67
5.2.1	Algoritmo	70
5.3	Um algoritmo eficiente para resolução do problema da mochila ilimitada em hipercubos . . .	71
5.3.1	Escalonamento	72
6	Eliminação paralela de termos redundantes	75
6.1	Pesquisa	75
6.1.1	Algoritmo utilizado	75
6.2	Resultados obtidos e análise	76
7	Conclusão	79
7.1	Oportunidades para pesquisas futuras	79
	Referências Bibliográficas	81
	Índice Remissivo	84

LISTA DE FIGURAS

2.1	Representação gráfica das soluções gulosa e ótima.	23
2.2	Exemplo de execução de algoritmo <i>branch and bound</i>	27
2.3	Instâncias não-correlatas.	31
2.4	Instâncias fracamente correlatas.	31
2.5	Instâncias fortemente correlatas.	32
2.6	Instâncias subset-sum.	32
3.1	Relação de dominância simples.	36
3.2	Relação de dominância múltipla.	37
3.3	Relação de dominância coletiva.	42
3.4	Relação de dominância por <i>threshold</i>	43
4.1	Árvore de decisão do algoritmo MTU2.	56
5.1	Cálculo de $F(k, j)$ no algoritmo de Lin e Storer usando c processadores.	68
5.2	Exemplo de grafo de dependência para uma instância de UKP.	69
5.3	Representação de um Elemento de Processamento (EP).	70
5.4	Um anel de tamanho p com fila.	71
5.5	Malha irregular $NM(4, 7)$ com saltos $(w_i)_{i=0}^3 = (2, 3, 4, 5)$	72
6.1	Gráfico da eliminação usando dominância simples para 5.000 e 10.000 objetos.	77
6.2	Gráfico da eliminação usando dominância simples para 50.000 e 100.000 objetos.	77
6.3	Gráfico da eliminação usando dominância múltipla para 5.000 e 10.000 objetos.	78
6.4	Gráfico da eliminação usando dominância múltipla para 50.000 e 100.000 objetos.	78

LISTA DE TABELAS

2.1	Tabela com valores da função objetivo de um algoritmo de programação dinâmica.	29
2.2	Tabela com valores do objeto de maior índice em um algoritmo de programação dinâmica. .	29
3.1	Exemplo de uma iteração do algoritmo de dominância múltipla de Pisinger.	40
3.2	Relações de dominância.	45
4.1	Simulação do algoritmo básico para resolução de $\mathbf{UKP}_c^N$	50
4.2	Simulação do algoritmo GN para resolução de $\mathbf{UKP}_c^N$	60
4.3	Simulação do algoritmo EDUK para resolução de $\mathbf{UKP}_c^N$	64

INTRODUÇÃO

O problema da mochila é um dos problemas fundamentais do campo da Programação Inteira. É conhecido que muitos problemas de Programação Inteira podem ser transformados em um Problema da Mochila [Pis95, Sal75].

Além de sua importância teórica intrínseca, os problemas da mochila possuem aplicação na resolução de diversos problemas que aparecem na indústria, no gerenciamento de finanças e algoritmos de criptografia, dentre outras áreas. Como o problema é $\mathcal{NP}$ -difícil [GJ79] diversos pesquisadores [AF94a, AF94b, AG91, AQ92, AR97, GH90, CJ92, GT02, LSS88a, LS91, MAR⁺00, Pis95] buscaram formas de resolvê-lo em paralelo. Por outro lado, pesquisadores trabalhando na resolução seqüencial do problema [Dud91, MT90b, Pis95, RAR00, ZB97] identificaram e aplicaram com êxito a eliminação de termos que não fazem parte de uma solução ótima, usando a propriedade denominada como relação de dominância. Graças às relações de dominância, é freqüentemente possível reduzir o tamanho de instâncias de problemas contendo centenas ou milhares de termos para instâncias com apenas alguns poucos termos [Dud91].

1.1 Objetivo do trabalho

Nosso objetivo com este trabalho é mostrar que na grande parte dos casos é possível obter-se grandes ganhos na resolução de problemas da mochila efetuando-se a eliminação de termos dominantes através de algoritmos paralelos. Apesar de nossos esforços, não encontramos na literatura nenhum trabalho que relacionasse paralelismo e relações de dominância. Assim, este trabalho tem o objetivo de explorar a idéia de trazer para o campo de resolução paralela do problema da mochila as relações de dominância. Para tal, estudamos alguns algoritmos seqüenciais que se valem de relações de dominância, tanto em uma fase de pré-processamento, quanto embutidos dentro do próprio algoritmo. Em seguida, estudamos alguns algoritmos paralelos para a resolução do problema da mochila. Por fim, tentamos combinar as idéias de forma a explorar em quais condições é vantajoso paralelizar a eliminação de termos dominados em instâncias do problema da mochila, baseando-se para tal tanto em um estudo teórico quanto através de experimentos práticos.

1.2 Organização do texto

No Capítulo 2 é apresentado o problema da mochila e suas mais importantes variantes, seguido de um apanhado geral das principais estratégias de resolução, além de apresentarmos algumas das instâncias clássicas do problema da mochila.

No Capítulo 3 são apresentadas duas propriedades matemáticas muito importantes na redução da quantidade de computação necessária para a resolução de problemas da mochila: periodicidade e relações de dominância. Na seção sobre relações de dominância são descritas várias relações conhecidas, e, ilustradas com exemplos.

No Capítulo 4 são apresentando alguns algoritmos sequenciais exatos para a resolução do problema da mochila que fazem uso de alguma das propriedades vistas no capítulo 3. Cobrimos desde um algoritmo básico para resolução do problema da mochila, até alguns algoritmos mais modernos e sofisticados que fazem uso das propriedades descritas no capítulo 3.

No Capítulo 5 são vistos alguns algoritmos paralelos para a resolução do problema da mochila.

No Capítulo 6 apresentamos nosso trabalho na tentativa de aplicar algoritmos paralelos na eliminação de objetos dominados na resolução de problemas da mochila. São apresentados alguns experimentos utilizando instâncias de diversos tamanhos e porcentagens (pré-conhecidas) de elementos dominados. Neste capítulo apresentamos os resultados e nossas conclusões sobre quando e quanto é vantajoso paralelizar a eliminação de objetos.

Por fim, no Capítulo 7 sintetizamos o trabalho e colocamos uma lista de oportunidades para pesquisas futuras.

O PROBLEMA DA MOCHILA E SUAS VARIANTES

2.1 O problema

O problema da mochila (*Knapsack Problem* ou somente *KP*) pode ser enunciado em linguagem natural da seguinte forma:

Um viajante levará consigo apenas uma mochila para sua viagem. Sua mochila possui uma dada capacidade e deve ser preenchida com diferentes tipos de objetos que lhe serão úteis durante a viagem. Cada objeto possui um peso e um dado valor para o viajante e este possui apenas uma unidade de cada objeto a ser escolhido. Quais objetos devem ser levados pelo viajante em sua mochila de forma a maximizar o valor do seu conteúdo?

Sejam w_j o peso do j -ésimo objeto e p_j seu valor. Se um objeto j aparece na mochila, definimos $x_j = 1$, caso contrário, $x_j = 0$. Se denotarmos por c a capacidade da mochila, por n a quantidade de objetos disponíveis para escolha e por $F(c, n)$ o maior valor possível para a mochila usando os n objetos, então o problema pode ser formulado algebricamente como segue:

$$F(c, n) = \max \sum_{j=1}^n p_j x_j \quad (p_j \in \mathbb{N}^*) \quad (2.1)$$

$$\text{sujeito a } \sum_{j=1}^n w_j x_j \leq c \quad (w_j, c \in \mathbb{N}^*) \quad (2.2)$$

Onde: $x_j = 0$ ou $x_j = 1$, $j \in N = \{1, 2, \dots, n\}$.

Esta versão do problema é conhecida como *problema da mochila binária* (*Binary KP* ou *0-1KP*).

Como um outro exemplo de aplicação do problema da mochila, suponha que um gerente de uma empresa possua no seu orçamento c unidades monetárias (u.m) para investir em projetos dentro do seu departamento. Após uma pesquisa realizada por sua equipe, o gerente recebe um relatório com n diferentes projetos que trariam reduções de custo ou aumento de produtividade ao departamento como um todo. Associado a cada projeto j existe um retorno de p_j u.m e um custo para sua realização de c_j u.m. O gerente pode encontrar uma distribuição ótima de seu orçamento resolvendo um problema da mochila binária.

Existem também aplicações na área da criptografia, engenharia naval, gerenciamento de projetos e finanças entre outras [MT90b, Pis95].

A formulação simples do problema da mochila pode levar a imaginar que sua solução seja igualmente simples. Porém isso não se comprova e a razão está em sua natureza essencialmente combinatória.

Uma tentativa ingênua de se obter todas as possíveis combinações de objetos para escolher a que maximiza o conteúdo da mochila se demonstra proibitiva na prática. Como descrito em [MT90b], para $n = 60$ a análise de todas as 2^{60} possíveis soluções levaria mais de 30 anos para ser calculada em um computador capaz de examinar 1 bilhão de possíveis soluções por segundo.

Após um pouco mais de reflexão, pode-se tentar resolver o problema através do seguinte algoritmo guloso:

ALGORITMO GULOSO

```

1  Ordene os objetos tal que  $p_j > p_{j+1}$ 
2  Para  $j \leftarrow 1$  até  $n$ 
3      Se  $w_j > c$  então  $x_j \leftarrow 0$ 
4      Senão
5           $x_j \leftarrow 1$ 
6           $c \leftarrow c - w_j$ 
```

Esse algoritmo é muito mais rápido que o da primeira tentativa pois enquanto aquele tinha uma complexidade de tempo de $O(2^n)$ este tem complexidade de tempo $O(n \log n)$ (ordenação na linha 1) para encontrar uma solução. Porém esse algoritmo não encontra a melhor solução sempre. Para mostrar isso, basta aplicar o algoritmo guloso para o seguinte contra-exemplo:

$$\begin{aligned}
 p_1 &= 60, p_2 = 100, p_3 = 120, \\
 w_1 &= 10, w_2 = 20, w_3 = 45, \\
 c &= 50.
 \end{aligned}$$

A aplicação do algoritmo guloso apresentado acima daria a resposta: $x_1 = 0, x_2 = 0, x_3 = 1$ porém a solução ótima é $x_1 = 1, x_2 = 1, x_3 = 0$. A Figura 2.1 ilustra o exemplo.

O problema da mochila binária vem sendo extensivamente estudado nas últimas três décadas e existem três grandes motivos para tal:

- Muitos problema de Programação Inteira podem ser transformado em um problema da mochila [Sal75, Pis95];
- Aparece com grande frequência como um subproblema de muitos problemas mais complexos [Pis95];
- Surge em diversas situações práticas como na escolha de portfólios de investimento, manufatura de placas de sistemas integrados e até mesmo em criptografia [Lag, MT90b, Pis95].

Veremos no Capítulo 4 algoritmos exatos para resolver problemas da mochila.

Figura 2.1: Representação gráfica das soluções gulosa e ótima.

2.2 Variações do problema da mochila

Generalizando a Equação 2.1 obtêm-se algumas variantes do problema da mochila. Em [MT90b], são abordadas as seguintes variantes:

Problema da Mochila Limitada:

Dados um conjunto N de n objetos com valores positivos p_j , pesos w_j , cada um tendo b_j cópias, e uma mochila de capacidade inteira e positiva c , determine um vetor $(x_1, x_2, \dots, x_n)$, $0 \leq x_j \leq b_j$ e $x_j \in \mathbb{N}$, que satisfaça as condições:

$$\begin{aligned} \max \quad & \sum_{j=1}^n p_j x_j, \\ \text{sujeito a} \quad & \sum_{j=1}^n w_j x_j \leq c. \end{aligned}$$

Problema da Mochila Ilimitada (UKP):

Dados um conjunto N com n objetos possuindo valores positivos p_j e pesos w_j , e uma mochila de capacidade inteira e positiva c , determine um vetor $(x_1, x_2, \dots, x_n)$, $x_j \in \mathbb{N}$ que satisfaça as condições:

$$\begin{aligned} \max \quad & \sum_{j=1}^n p_j x_j, \\ \text{sujeito a} \quad & \sum_{j=1}^n w_j x_j \leq c. \end{aligned}$$

Este problema é uma generalização do Problema da Mochila Limitada no qual $b_j = \infty, \forall j \in N$.

Problema da Soma de Subconjuntos:

Dados um conjunto N de n objetos com pesos w_j , e uma mochila de capacidade inteira e positiva c , determine um vetor $(x_1, x_2, \dots, x_n)$, $x_j \in \{0, 1\}$, que satisfaça as condições:

$$\begin{aligned} & \max \sum_{j=1}^n w_j x_j, \\ & \text{sujeito a } \sum_{j=1}^n w_j x_j \leq c. \end{aligned}$$

Este problema é um caso particular do 0-1KP em que $w_j = p_j, \forall j \in N$.

Change Making Knapsack Problem:

Dados um conjunto N de n objetos com valores p_j , pesos w_j , cada um tendo b_j cópias, e uma mochila de capacidade inteira e positiva c , determine um vetor $(x_1, x_2, \dots, x_n)$, $0 \leq x_j \leq b_j, x_j \in \mathbb{N}$, que satisfaça as condições:

$$\begin{aligned} & \max \sum_{j=1}^n x_j, \\ & \text{sujeito a } \sum_{j=1}^n w_j x_j = c. \end{aligned}$$

Podemos descrever este problema como o de trocar uma determinada quantia de cédulas por moedas, de forma a manter o valor monetário, porém maximizando o número de moedas.

Problema da Mochila Binária Múltipla:

Dados um conjunto de n objetos e m mochilas de capacidade inteira e positiva c_i , dizemos que $x_{ij} = 1$ se o j -ésimo objeto for colocado na mochila i e $x_{ij} = 0$ caso contrário. Determine a matriz x_{ij} que satisfaça as condições:

$$\begin{aligned} & \max \sum_{i=1}^m \sum_{j=1}^n p_j x_{ij}, \\ & \text{sujeito a } \sum_{j=1}^n w_j x_{ij} \leq c_i, \quad i = 1, \dots, m, \\ & \sum_{i=1}^m x_{ij} \leq 1, \quad j = 1, \dots, n. \end{aligned}$$

Em [MT90b] é demonstrado que todos os problemas apresentados acima fazem parte da classe dos problema $\mathcal{NP}$ -difíceis [GJ79].

As pesquisas em torno de soluções de KP podem ser subdivididas em dois grupos:

- Algoritmos de solução exata
 - Algoritmos de *branch and bound*;
 - Algoritmos de programação dinâmica.
- Algoritmos de aproximação.
- Heurísticas.

2.3 Algoritmos exatos para resolver KP

2.3.1 Algoritmos de *branch and bound*

Branch and bound é uma técnica amplamente utilizada em problemas de otimização. O conjunto das possíveis soluções do problema formam uma árvore de busca repletas de nós. No problema da mochila, cada **nó** representa o estado da mochila (valor e peso) se nela for incluída uma determinada quantidade de objetos. Definindo-se o nó raiz da árvore como estando no nível zero, no k -ésimo nível ficam os nós representando a inclusão ou não do k -ésimo objeto. A aresta ligando um nó do k -ésimo nível ao seu pai no nível $k-1$ armazena o número de vezes que o k -ésimo objeto faz parte dessa solução. A árvore de busca possui altura $n + 1$.

Ao se percorrer cada nó da árvore calcula-se um valor **limite** (*bound* em inglês) do *melhor*¹ valor que pode ser atingido a partir desse nó na árvore. É calculado também um peso para esse nós, consistindo na soma do peso do seu nó pai com o peso do objeto ao qual representa vezes o número armazenado na aresta que o liga ao seu pai.

Um nó é chamado de **promissor** se possuir um limitante melhor que a melhor solução encontrada até o momento. Se um nó não for promissor, as soluções que envolvem aquele nó são descartadas e ele é dito **podado** (*pruned* em inglês). A árvore é percorrida através de uma busca em profundidade e a solução ótima é formada pelo conjunto dos valores das arestas do caminho que vai da raiz até o nó com o melhor valor.

Como citado anteriormente, uma busca por todo o espaço de soluções possui complexidade exponencial, porém a utilização da técnica de *branch and bound* permite obter um resultado muito melhor no caso médio. Esse ganho se dá através de duas maneiras:

- Podam-se os nós que conduzem a soluções impossíveis, ou seja, se a soma dos pesos dos itens que já foram selecionados, somado ao peso do item associado ao nó sendo analisando for maior que a capacidade da mochila, então esse nó (e toda sua subárvore) são podados da árvore de busca.

¹O sentido da palavra *melhor* depende do problema estudado e do tipo de otimização que se busca (maxização ou minimização).

- Podam-se os nós que não têm chance de conduzir a uma solução ótima. Isso é feito calculando-se um limitante superior para a solução em cada nó (através de uma função de limitação ou *bounding function*) e comparando-a com o melhor valor obtido até o momento. Para o problema da mochila ilimitada, vamos adotar a seguinte função para o cálculo do valor de um nó que se refere ao k -ésimo ($k < n$) objeto (supondo-se que os objetos estejam ordenados tal que $p_i/w_i > p_{i+1}/w_{i+1}$):

$$\sum_{i=1}^k p_i x_i + (c - \sum_{i=1}^k w_i x_i) p_{k+1}/w_{k+1},$$

se $c - \sum_{i=1}^k w_i x_i \geq w_j$ para algum $j > k$; (2.3)

ou

$$\sum_{i=1}^k p_i x_i,$$

se $c - \sum_{i=1}^k w_i x_i < w_j, \forall j > k$. (2.4)

O valor de x_i é dado pelo valor na aresta que liga o nó no nível x_i a seu pai no caminho avaliado.

A Fórmula (2.3) possui duas partes. A primeira parte é o valor de todas as variáveis que já foram atribuídas anteriormente, essas variáveis possuem $\sum w_i x_i$ em peso e a capacidade residual é $c - \sum w_i x_i$. Devido a ordenação dos objetos, o valor máximo que poderia ser atingido aproveitando-se essa capacidade residual é $c - (\sum_{i=1}^k w_i x_i) p_{k+1}/w_{k+1}$ contanto que x_{k+1} não tenha que ser um inteiro (a ausência da restrição de x ser um inteiro é conhecida como relaxação contínua).

Dessa forma, se o valor de todo nó é dado por (2.3), então seu valor é um limitante superior do valor de seus filhos. Dessa forma, atingiu-se o objetivo de poder descartar uma subárvore pois se o valor de um nó for menor ou igual ao da melhor solução conhecida até o momento, então esse nó e seus descendentes podem ser podados da árvore de busca.

Porém se (2.4) for válida, então nenhum item a mais pode ser posto na mochila e $\sum_{i=1}^k p_i x_i$ é o valor desse nó, que também é uma folha.

A Figura 2.2 ilustra a execução do algoritmo para a seguinte instância de problema:

$$\begin{aligned} p_1 &= 9, p_2 = 5, p_3 = 3, p_4 = 1, \\ w_1 &= 7, w_2 = 4, w_3 = 3, w_4 = 2, \\ c &= 10. \end{aligned}$$

Na Figura 2.2, quando $x_i = x$ significa que o primeiro objeto entra x ($x \in \mathbb{N}_+$) vezes como parte do conjunto solução. A solução ótima é dada pelo nó representado como um quadrado. O algoritmo cria os nós A, B, C, D, E, F, G, H, I, J, K, L, M na respectiva ordem. Note que os nós C, D, F e G são podados pois possuem peso superior à capacidade da mochila. Os nós K, L e M são podados pois

não são promissores, ou seja, possuem valor menor ou igual ao da melhor solução conhecida até o momento².

Figura 2.2: Exemplo de execução de algoritmo *branch and bound*.

Análise: No pior caso nenhuma subárvore é podada e o algoritmo requer $O(2^n)$ computações. No entanto, para um grande número de instâncias, menos computações são necessárias pois um grande número de subárvores é podada da árvore de decisão.

Uma solução mais sofisticada e eficiente de implementação da técnica de *branch and bound* para resolução do problema da mochila ilimitada é vista na seção 4.2.

2.3.2 Algoritmos de programação dinâmica

Programação dinâmica é outra técnica algorítmica comum na solução de problemas combinatórios. Essa técnica mostra-se sempre viável em problemas de otimização que possuem subestrutura ótima, ou seja, uma solução ótima para o problema contém soluções ótimas de subproblemas e onde os subproblemas são interdependentes, isto é, a solução de um subproblema altera a solução de outros subproblemas. Em

²Coincidentemente neste exemplo, esta é também a solução ótima.

uma solução do problema usando recursão, essa característica de interdependência e subestrutura ótima é ignorada, conseqüentemente o algoritmo faz diversas vezes computações que já foram feitas anteriormente.

Nos algoritmos que utilizam a técnica de programação dinâmica os resultados dos subproblemas são armazenados em uma tabela, de modo que possam ser consultados posteriormente, quando da resolução de subproblemas maiores.

Para entender como isso se aplica na resolução de KP, vamos primeiramente definir a função $F_k(\hat{c})$:

$$F_k(\hat{c}) = \max \sum_{j=1}^k p_j x_j \quad (0 \leq k \leq n),$$

com

$$\sum_{j=1}^k w_j x_j \leq \hat{c} \quad (0 \leq \hat{c} \leq c).$$

Ou seja, a função $F_k(\hat{c})$ é o maior valor possível usando somente os k primeiros itens, quando o limite máximo de peso é $\hat{c}$. Portanto:

$$F_0(\hat{c}) = 0, 0 \leq \hat{c} \leq c,$$

e

$$F_k(0) = 0, 0 \leq k \leq n.$$

Para deduzir a fórmula de $F_k(\hat{c})$ para k e $\hat{c}$ genéricos, é necessário estar atento às seguintes características do problema:

Quando calculando $F_k(\hat{c})$ ou o k -ésimo elemento está sendo usado (ao menos uma vez se estivermos resolvendo UKP), ou o k -ésimo elemento nunca é usado. Se o k -ésimo elemento não é usado, não se altera o valor de $F_k(\hat{c})$, de onde se conclui que nesse caso $F_k(\hat{c}) = F_{k-1}(\hat{c})$. Se o k -ésimo item é utilizado, então o valor da capacidade da mochila (sua capacidade residual) é reduzido para $c - w_k$. Para que o problema total seja ótimo é necessário que esta capacidade residual seja utilizada de forma ótima, ou seja, seu valor deve ser $F_k(\hat{c} - w_k)$. No entanto o k -ésimo item utilizado possui um valor e nesse caso $F_k(\hat{c}) = F_k(\hat{c} - w_k) + p_k$. Portanto, a fórmula geral para $F_k(\hat{c})$ é definida recursivamente da seguinte forma:

$$F_k(\hat{c}) = \max\{F_{k-1}(\hat{c}), F_k(\hat{c} - w_k) + p_k\}. \quad (2.5)$$

Um algoritmo de programação dinâmica para resolver instâncias de KP deve armazenar em uma tabela de dimensões $n \times c$ todos os valores de $F_k(\hat{c})$. Ela é calculada na ordem:

$$F_1(1), F_1(2), \dots, F_1(c); F_2(1), F_2(2), \dots, F_2(c); F_n(1), F_n(2), \dots, F_n(c).$$

Ou seja, a tabela é preenchida da esquerda para a direita e de cima para baixo. Suponha que tenhamos o seguinte problema da mochila ilimitada a ser resolvido:

$$\begin{aligned} p_1 &= 1, p_2 = 3, p_3 = 5, p_4 = 9, \\ w_1 &= 2, w_2 = 3, w_3 = 4, w_4 = 7, \\ c &= 10. \end{aligned}$$

Então um algoritmo de programação dinâmica produziria a Tabela 2.1.

$k \setminus \hat{c}$	1	2	3	4	5	6	7	8	9	10
1	0	1	1	2	2	3	3	4	4	5
2	0	1	3	3	4	6	6	7	9	9
3	0	1	3	5	5	6	8	10	10	11
4	0	1	3	5	5	6	9	10	10	12

Tabela 2.1: Tabela com valores da função objetivo de um algoritmo de programação dinâmica.

Observe que usando apenas a recursão 2.5 e a tabela 2.1 é possível se determinar o valor da mochila de solução ótima mas não se possui informação alguma sobre quais objetos fazem parte desta solução. Para obter esta informação é necessário fazer uso de uma nova tabela, onde a célula na linha k e coluna $\hat{c}$ representa qual o maior índice dos objetos utilizados.

Para tal, definimos a função $I_k(\hat{c})$ de forma recursiva, como a seguir:

$$I_k(\hat{c}) = \begin{cases} I_{k-1}(\hat{c}), & \text{se } F_{k-1}(\hat{c}) > F_k(\hat{c} - w_k) + p_k, \\ k, & \text{se } F_{k-1}(\hat{c}) \leq F_k(\hat{c} - w_k) + p_k. \end{cases} \quad (2.6)$$

Onde $I_1(k) = 1$ se $F_1(\hat{c}) \neq 0$ e $I_1(k) = 0$ se $F_1(\hat{c}) = 0$.

Note que a equação 2.6 detém uma grande relação com a equação 2.5. Isso porque se $F_{k-1}(\hat{c}) > F_k(\hat{c} - w_k) + p_k$ então o k -ésimo objeto não faz parte da solução ótima ($x_k = 0$) em $F_k(\hat{c})$, e assim, o maior índice para uma solução ótima de $F_k(\hat{c})$ é o maior índice do subproblema $F_{k-1}(\hat{c})$ que é $I_{k-1}(\hat{c})$. Como no algoritmo de programação dinâmica em cada iteração o índice k do objeto é maior ou igual que o anterior, se o k -ésimo objeto faz parte da solução ótima de $F_k(\hat{c})$ então este é o maior índice que faz parte da solução ótima do subproblema $F_k(\hat{c})$. Os valores de $I_k(\hat{c})$ para o exemplo dado estão descritos na tabela 2.2.

$k \setminus \hat{c}$	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	1	1	1	1	1	1
2	0	1	2	2	2	2	2	2	2	2
3	0	1	2	3	3	3	3	3	3	3
4	0	1	2	3	3	3	4	3	4	4

Tabela 2.2: Tabela com valores do objeto de maior índice em um algoritmo de programação dinâmica.

Assim como a tabela 2.1, a tabela 2.2 é contruida pelo algoritmo de programação dinâmica da esquerda para a direita e de cima para baixo.

Mas como utilizar a tabela 2.2 para determinar os valores de $\vec{x}$ para uma solução ótima do problema? De forma geral, se $I_k(\hat{c}) = q$, então $x_q \geq 1$ em $F_k(\hat{c})$. Como o q -ésimo elemento é usado ao menos uma vez, então a capacidade da mochila é reduzida para $c - w_q$, e deve-se então observar o valor de $I_k(\hat{c} - w_q)$.

Portanto, partindo-se da posição $I_n(c)$ (última linha e coluna da tabela com os valores dos índices máximos), incrementa-se o valor do q -ésimo objeto em $\vec{x}$ toda vez que $I_n(\hat{c}) = q$, repetindo-se o procedimento para $I_n(\hat{c} - w_q)$ até que $I_n(\hat{c}) = 0$.

No nosso exemplo, para determinarmos $\vec{x}$ ótimo para o problema olharíamos $I_4(10)$ que é igual a 4, significando que o quarto objeto aparece ao menos uma vez em uma solução ótima, passa-se então a $I_4(10 - w_4) = I_4(10 - 7) = I_4(3) = 2$, significando que o segundo objeto aparece ao menos uma vez em uma solução ótima. Por fim $I_4(3 - w_2) = I_4(0) = 0$, indicando que o algoritmo chegou ao fim e que uma solução ótima para nosso problema de exemplo é $\vec{x} = (0, 1, 0, 1)$.

Análise: Os algoritmos baseados em técnicas de programação dinâmica, como acima, requerem $O(nc)$ computações já que para cada $\hat{c}$, $0 < \hat{c} \leq c$ calcula-se o valor de $F_k(\hat{c})$ e $I_k(\hat{c})$ para cada um dos n elementos em $|N|$. Vale notar que apesar de o número de computações ser $O(nc)$ ele não é polinomial para todas as instâncias do problema, já que a entrada c pode ser fixada como sendo exponencialmente maior que n e portanto teríamos uma instância consumindo $O(n2^n)$ computações. Algoritmos que apresentam esta característica são chamados de pseudo-polinomiais.

2.4 Algoritmos de aproximação e heurísticas

Como já foi dito anteriormente, o problema da mochila faz parte da classe de problemas $\mathcal{NP}$ -difícil [GJ79] e portanto existem algumas instâncias do problema que não podem ser resolvidas em uma quantidade de tempo aceitável com os algoritmos conhecidos atualmente. Em tais situações, faz-se necessário utilizar algoritmos que apesar de não garantirem uma solução ótima, produzem uma solução cuja distância da solução ótima é limitado por uma constante ϵ tal que:

$$\epsilon \geq \frac{z - z^*}{z^*}, \quad (2.7)$$

onde z é o valor da solução aproximada e z^* é o valor da solução ótima.

Os algoritmos mais sofisticados de aproximação são os chamados *algoritmos de aproximação totalmente polinomial*. Tais algoritmos são tais que para qualquer $\epsilon > 0$ o algoritmo executa em tempo (e espaço) polinomial ao tamanho da entrada com razão $1/\epsilon$. Deve-se observar que não são conhecidos algoritmos de aproximação totalmente polinomiais para problemas $\mathcal{NP}$ -difíceis no sentido forte, pois se assim fosse o caso, seria possível provar que $\mathcal{P} = \mathcal{NP}$ [Iba87].

Existem também outras técnicas baseadas em heurísticas para a resolução aproximada de problemas da mochila. A maior parte delas, aplicada dentro de técnicas de programação dinâmica ou *branch and bound*, onde a enumeração é terminada (segundo algum critério) antes de se ter analisado todos os elementos.

Como esta dissertação está focada em algoritmos exatos para a resolução do problema da mochila, não entraremos em maiores detalhes sobre algoritmos de aproximação e outras heurísticas.

2.5 Instâncias do problema da mochila

Na literatura, as diversas possíveis instâncias do problema da mochila são divididas em classes ou categorias que apresentam uma determinada propriedade, que podem afetar em maior ou menor grau os algoritmos de resolução do problema.

2.5.1 Instâncias não-correlatas

Nessas instâncias não existe correlação entre o peso e o valor dos objetos. Tais instâncias aparecem em algumas situações práticas onde o valor do objeto não depende do peso, como no caso do viajante que prepara sua mochila para uma viagem e muitas vezes objetos pequenos e leves acabam tendo maior valor que objetos grandes.

Figura 2.3: Instâncias não-correlatas.

Instâncias não-correlatas do problema da mochila são em geral fáceis de serem resolvidas pois apresentam uma grande variedade de pesos e também facilitam a eliminação de objetos tanto por técnicas de dominância³ como por testes de limites.

2.5.2 Instâncias fracamente correlatas

Nessas instâncias existe uma correlação entre o peso e o valor dos objetos. Tais instâncias aparecem em situações práticas onde o valor do objeto é função do peso, podendo variar dentro de uma margem. Tal situação ocorre na prática em investimentos onde a rentabilidade é maior para investimentos que exigem um maior aporte de capital, ou em investimentos financeiros em que o risco é maior para investimentos com maior possibilidade de ganho.

Figura 2.4: Instâncias fracamente correlatas.

Instâncias fracamente correlatas do problema da mochila são um pouco mais difíceis de serem resolvidas pois a correlação torna mais difícil a eliminação de objetos por testes de limites.

³No capítulo 3 será definido formalmente o conceito de dominância.

2.5.3 Instâncias fortemente correlatas

Nessas instâncias existe uma forte correlação entre o peso e o valor dos objetos. Aparecem em situações onde o valor dos objetos é uma função linear do peso acrescida (ou decrescida) de uma constante.

Figura 2.5: Instâncias fortemente correlatas.

Instâncias fortemente correlatas são muito difíceis de serem resolvidas e por isso são utilizadas como recurso para se avaliar o quão bom um algoritmo é para a resolução de problemas da mochila difíceis.

2.5.4 Instâncias *subset-sum*

Nessas instâncias o valor de um objeto é exatamente igual ao seu peso.

Figura 2.6: Instâncias subset-sum.

Em todas as classes de instâncias expostas acima a dificuldade de um problema é dada também por outros parâmetros como a capacidade c da mochila, a faixa de valores que o peso e o valor podem assumir, assim como o tamanho n do problema.

RELAÇÕES DE DOMINÂNCIA E PERIODICIDADE

Como dito anteriormente, o problema da mochila ilimitada (UKP) é $\mathcal{NP}$ -difícil, mas pode ser resolvido em complexidade de tempo pseudo-polinomial $O(nc)$ através de técnicas de programação dinâmica. Por essa razão surgiram diversos trabalhos com o objetivo de diminuir o espaço de busca do problema, seja descartando objetos que não são relevantes para o problema (através das chamadas relações de dominância), seja pela constatação de que a partir de um dado valor de capacidade, apenas o objeto com a maior razão p_i/w_i contribui para a solução do problema.

Neste capítulo estudamos estas duas técnicas de redução do espaço de busca em UKP. Como será visto mais a frente, ambas são empregadas de forma bastante engenhosa em um algoritmo descrito na seção 4.4.

3.1 Periodicidade

A propriedade de periodicidade em UKP é conhecida desde 1966 [GG66] e se baseia na constatação de que, a partir de uma dada capacidade y^* , somente um tipo de objeto contribui na solução. Este tipo de objeto é o que possui maior razão $\rho_i = p_i/w_i$ e será chamado de b , vamos denotar seu peso e valor por w_b e p_b , respectivamente.

Graças à periodicidade, o conhecimento da solução do problema para cada capacidade abaixo de y^* é suficiente para resolver o problema para qualquer capacidade. Matematicamente esta propriedade se formula como a seguir:

$$F(c, N) = \lceil (c - y^*)/w_b \rceil p_b + F(c - \lceil (c - y^*)/w_b \rceil w_b, N), \quad \forall c \geq y^*. \quad (3.1)$$

O primeiro termo de (3.1) denota o valor obtido da mochila preenchida apenas com o objeto b da capacidade y^* até c , enquanto a outra parcela é o valor ótimo do subproblema da mochila com capacidade $c - \lceil (c - y^*)/w_b \rceil w_b$.

A propriedade de periodicidade pode ser deduzida, e calcular-se um y^* como acima, da seguinte maneira: Imagine, sem perda de generalidade, que temos os objetos dispostos em ordem estritamente decrescente de ρ , portanto $b = 1$. Queremos mostrar que existe um valor de y^* tal que $\forall y : y \geq y^*$ há uma solução ótima tal que $x_1 \geq 1$.

Para uma mochila de capacidade y , uma solução possível (porém não necessariamente ótima) é: $x_1 = \lfloor y/w_1 \rfloor$ e $x_j = 0$, $j = 2, \dots, n$. Denotando o respectivo valor desta possível solução como θ_1 , temos:

$$\theta_1 = p_1 \left\lfloor \frac{y}{w_1} \right\rfloor \geq p_1 \left(\frac{y}{w_1} - 1 \right) = \frac{p_1 y}{w_1} - p_1 = \rho_1(y - w_1). \quad (3.2)$$

Por sua vez, um limitante superior para o valor de uma mochila com capacidade y tal que $x_1 = 0$ é

$$\theta_2 = \frac{p_2 y}{w_2} = \rho_2 y. \quad (3.3)$$

Usando as equações (3.2) e (3.3), temos que a condição necessária para que $x_1 \geq 1$ em alguma solução ótima é $\rho_1(y - w_1) \geq \rho_2 y$ ou equivalentemente:

$$y^* \leq \frac{p_1}{\rho_1 - \rho_2}. \quad (3.4)$$

Em outras palavras, para y maior do que y^* , certamente $x_1 \geq 1$; uma solução ótima para este valor de y é

$$F(y, N) = p_1 + F(y - w_1, N). \quad (3.5)$$

Cabe enfatizar que x pode ser maior ou igual a 1 em uma solução ótima mesmo para $y < y^*$, porém o que a inequação 3.4 nos garante é que certamente $x \geq 1$ para $y \geq y^*$.

Vamos definir agora $\pi(y) = \rho_1 y - F(y, N)$, ou seja, $\pi(y)$ é o valor ótimo para a problema removendo-se a restrição de que o vetor solução $\vec{x}$ seja inteiro (primeira parcela), subtraído do valor ótimo, para o mesmo problema, porém mantendo-se a integralidade para os valores de $\vec{x}$ (segunda parcela). Demonstraremos que $\pi(y)$ é periódica:

Para um valor de y suficientemente grande ($y \geq y^*$):

$$\pi(y - w_1) = \rho_1(y - w_1) - F(y - w_1, N) = \rho_1 y - \rho_1 w_1 - (F(y, N) - p_1) = \rho_1 y - F(y, N) = \pi(y).$$

Isso mostra que a função $\pi(y)$ é periódica, com período igual a w_1 para $y > y^*$, como queríamos demonstrar.

Exemplo:

Suponha a seguinte instância do problema da mochila:

$$\begin{aligned}
p_1 &= 18, p_2 = 14, p_3 = 8, p_4 = 4, p_5 = 0 \\
w_1 &= 15, w_2 = 12, w_3 = 7, w_4 = 4, w_5 = 1 \\
\rho_1 &= 1.2, \rho_2 = 1.167, \rho_3 = 1.143, \rho_4 = 1, \rho_5 = 0 \\
c &= 1000.
\end{aligned}$$

Então:

$$\frac{p_1}{\rho_1 - \rho_2} = \frac{18}{18/15 - 14/12} = 540.$$

Assim, para capacidades maiores do que 540 a solução ótima certamente terá $x_1 \neq 1$. Portanto, para capacidades acima deste valor é possível subtrair w_1 de c até que a capacidade fique inferior a 540. No exemplo, teremos $x_1 \geq 30$ já que $1000 - 30w_1 = 550 > 540$.

3.2 Relações de dominância

Durante os anos de estudo sobre a solução de problemas da mochila, logo se observou que mesmo nas instâncias com milhares de objetos, somente alguns poucos objetos faziam parte da solução ótima. Como o problema é $\mathcal{NP}$ -difícil, é extremamente importante identificar quais objetos podem ser descartados, antes mesmo que se inicie o cálculo da solução ótima ou o mais cedo possível. Desta constatação nasceu a teoria das relações de dominância em problemas da mochila.

Definição 3.1 Para um dado objeto k pertencente a uma instância do problema da mochila, denomina-se o termo k de dominado (ou redundante) se o valor da solução ótima não se altera quando k for removido do problema. Ou seja, para um termo redundante k pode-se fixar $x_k = 0$ no problema original.

3.2.1 Dominância simples

Teorema 3.1 Sejam j e k dois objetos pertencentes a uma instância do problema da mochila ilimitada. Se

$$w_j \leq w_k \text{ e } p_j \geq p_k \quad (3.6)$$

então o objeto k é dominado pelo objeto j .

Prova: Seja $x_k = \alpha > 0$ e $x_j = \beta$ uma possível solução. Uma solução melhor pode ser construída fazendo $x_k = 0$ e $x_j = \beta + \alpha$. De fato, a nova solução é possível já que através de (3.6) temos que $\alpha w_j \leq \alpha w_k$ e $\alpha p_j \geq \alpha p_k$. Portanto $F(c, N \setminus \{k\}) \geq F(c, N)$.

Definição 3.2 Sejam j e k dois objetos pertencentes a uma instância de um problema da mochila. Se j e k satisfazem (3.6) então diz-se que o objeto k é simplesmente dominado pelo objeto j .

Esta relação de dominância foi proposta pela primeira vez por Gilmore and Gomory [GG63].

Exemplo: Suponha a seguinte instância do problema da mochila:

$$\begin{aligned} p_1 &= 302, p_2 = 580, p_3 = 301, p_4 = 601, p_5 = 605, p_6 = 322, p_7 = 330 \\ w_1 &= 120, w_2 = 245, w_3 = 130, w_4 = 260, w_5 = 310, w_6 = 194, w_7 = 190 \\ c &= 10. \end{aligned}$$

Se representarmos por $i \prec j$ se o objeto i for dominado pelo objeto j , então no exemplo $3 \prec 1, 6 \prec 7$.

Uma representação gráfica desta relação de dominância pode ser vista na Figura 3.1.

Algoritmo 3.1 Seja A o conjunto formado por objetos de um problema da mochila.

O algoritmo abaixo remove todos os elementos simplesmente dominados com $O(|A|^2)$ operações.

REMOVE DOMINADOS SIMPLES (A)

- 1 Ordene os elementos tal que $w_j < w_{j+1}$
e se $w_j = w_{j+1}$ então $p_j \geq p_{j+1}$, $j = 1, \dots, |A| - 1$
- 2 Para $j \leftarrow 1$ até $|A| - 1$ faça
- 3 Para $k \leftarrow j + 1$ até $|A|$ faça
- 4 Se a condição (3.6) for válida faça
- 5 $A \leftarrow A \setminus \{k\}$
- 6 Retorne A

Figura 3.1: Relação de dominância simples.

3.2.2 Dominância múltipla

Teorema 3.2 Sejam j e k dois objetos pertencentes a uma instância do problema da mochila ilimitada. Se:

$$\lfloor w_k/w_j \rfloor p_j \geq p_k \quad (3.7)$$

então o objeto k é dominado pelo objeto j .

Prova: Seja $x_k = \alpha > 0$ e $x_j = \beta$ uma possível solução. Uma solução melhor pode ser construída fazendo $x_k = 0$ e $x_j = \beta + \lfloor w_k/w_j \rfloor \alpha$. De fato, a nova solução é possível já que $\lfloor w_k/w_j \rfloor \alpha w_j \leq \alpha w_k$ e apresenta um valor igual ou melhor que a solução anterior, pois através de (3.7) temos que: $\lfloor w_k/w_j \rfloor \alpha p_j \geq \alpha p_k$, e portanto $F(c, N \setminus \{k\}) \geq F(c, N)$.

Definição 3.3 Sejam j e k dois objetos pertencentes a uma instância de um problema da mochila. Se j e k satisfazem (3.7) então diz-se que o objeto k é multiplamente dominado pelo objeto j .

Exemplo: Suponha a seguinte instância do problema da mochila:

$$\begin{aligned} p_1 &= 300, p_2 = 580, p_3 = 301, p_4 = 601, p_5 = 605, p_6 = 322, p_7 = 310 \\ w_1 &= 120, w_2 = 245, w_3 = 130, w_4 = 260, w_5 = 310, w_6 = 194, w_7 = 190 \\ c &= 10. \end{aligned}$$

Se representarmos por $i \prec j$ se o objeto i for dominado pelo objeto j , então no exemplo apenas o objeto 2 seria dominado, já que $2 \prec 1$.

Uma representação gráfica desta relação de dominância pode ser vista na Figura 3.2.

Algoritmo 3.2 Seja A o conjunto formado por objetos de um problema da mochila. O algoritmo abaixo remove todos os elementos multiplamente dominados em $O(|A|^2)$ operações.

```

REMOVE MT ( $A$ )
1  Ordene os elementos tal que  $p_j/w_j < p_{j+1}/w_{j+1}$ 
   e se  $p_j/w_j = p_{j+1}/w_{j+1}$  então  $w_j \leq w_{j+1}$ 
2  Para  $j \leftarrow 1$  até  $|A| - 1$  faça
3    Para  $k \leftarrow j + 1$  até  $|A|$  faça
4      Se a condição (3.7) for válida faça
5         $A \leftarrow A \setminus \{k\}$ 
6  Retorne  $A$ 

```

Figura 3.2: Relação de dominância múltipla.

A relação de dominância múltipla e o algoritmo acima foram introduzidos por Martello e Toth [MT90a]. Além de ser uma generalização do conceito de dominância simples é parte fundamental do algoritmo de *branch and bound* que apresentaram para a resolução exata de grandes instâncias de UKP (centenas ou milhares de objetos). Esta relação de dominância permite uma grande redução no número de objetos do problema, principalmente para instâncias não-correlatas ou fracamente correlatas do problema da mochila. Dudzinski [Dud91] mostrou que diversas instâncias de UKP com milhares de objetos podem ser reduzidas para instâncias com menos de uma dúzia de elementos apenas aplicando-se a eliminação de termos multiplamente dominados antes de se iniciar a resolução do problema. Nessas instâncias, a redução do número de elementos é tão grande que o tempo gasto para ordenar os elementos de forma que $p_1/w_1 > p_2/w_2 > \dots > p_n/w_n$ é maior do que o tempo gasto para resolver o problema. Baseado nisso, ele propôs um algoritmo para instâncias com poucos elementos não dominados, onde a eliminação de termos se dá em complexidade de tempo $O(mn)$ (onde m é o número de objetos não dominados na instância do problema). O algoritmo de eliminação de Dudzinski não necessita pré-ordenar os objetos.

Algoritmo 3.3 Seja A o conjunto formado por objetos de um problema da mochila.

O algoritmo abaixo detecta elementos não dominados i e para cada um deles remove em tempo linear todos os elementos multiplamente dominados por i .

ALGORITMO DUDZINSKI (A)

```

1  Para  $i \leftarrow 1$  até  $|A|$  faça
2    Para  $j \leftarrow i + 1$  até  $|A|$  faça
3      Se  $\lfloor w_i/w_j \rfloor p_j \geq p_i$  faça
4        troca( $i, j$ ) {Objeto na posição  $i$  passa para posição  $j$  e vice-versa.}
5    Para  $j \leftarrow i + 1$  até  $|A|$  faça
6      Se  $\lfloor w_j/w_i \rfloor p_i \geq p_j$  faça
7         $A \leftarrow A \setminus \{j\}$ 
```

Apesar de no pior caso o algoritmo acima ter complexidade $O(n^2)$ ele é mais vantajoso que o algoritmo MT quando $m < \log n$.

Algoritmo de Pisinger

Um algoritmo seqüencial mais rápido que o algoritmo MT para eliminação de termos redundantes baseados em redundância múltipla foi apresentado por Pisinger [Pis95] em 1995. O algoritmo proposto se baseia no fato de que não é necessário ordenar os objetos pela relação p_j/w_j mas apenas dispô-los em uma seqüência não decrescente de pesos w .

Proposição 3.1 O algoritmo MT funciona igualmente bem, se os objetos estão ordenados em ordem não-decrescente de pesos (desempatando de forma que $p_j \geq p_{j+1}$).

Prova: A condição (3.7) não ocorre se $w_i > w_j$, ou se $p_i < p_j$ no caso de $w_i = w_j$. Sendo assim é necessário ordenar apenas em ordem não-decrescente de peso, e em tempo linear resolver os casos em que os pesos sejam idênticos. O algoritmo DESEMPATA, descrito a seguir, recebe como entrada uma instância de UKP (conjunto A) após ter sido realizada uma ordenação não-decrescente de pesos e realiza os desempates em $O(n)$.

ALGORITMO DESEMPATA (A)

```

1   $m \leftarrow 1$ 
2  Para  $j \leftarrow 2$  até  $|A|$  faça
3      Se  $p_j > p_m$  faça
4      Se  $w_j \neq w_m$  faça
5           $m \leftarrow m + 1$ 
6      troca( $j, m$ )

```

Definição 3.4 Definimos $d_{ij} = \lfloor w_j/w_i \rfloor$ para $i < j$.

Devido a ordenação dos objetos pelo peso, temos como consequência da definição:

Proposição 3.2 O valor d_{ij} aumenta quando j aumenta e decresce quando i aumenta.

Proposição 3.3 Se diversos objetos $i_1, \dots, i_h$ têm o mesmo valor para o quociente d_{ij} para um dado j então somente o último objeto deve ser considerado para testar dominância. Além disso, se um conjunto de objetos i possui $d_{ij} = 1$ para um objeto j , então nenhum dos objeto i jamais dominará o objeto j e portanto não precisam ser testados quanto a dominância.

Prova: Como tanto o valor quanto o peso estão aumentando para $i_1, \dots, i_h$ então o maior valor para $d_{ij}p_i$ ocorre para o índice i_h . Assim, se a condição (3.7) não ocorre para i_h , então a condição também não se verificará para $i_1, \dots, i_{h-1}$. A segunda parte da proposição é uma consequência do aumento do valor dos objetos, pois se $d_{ij} = 1$ então $d_{ij}p_i < p_j$ já que $i < j$.

Definição 3.5 Para um dado objeto j , definimos l_d^j como o índice do objeto *representante* dos objetos correspondentes ao quociente d .

$$l_d^j = \left(\max_{i < j} i : \left\lfloor \frac{w_j}{w_i} \right\rfloor \geq d \right) \quad (3.8)$$

O objeto *representante* possui este nome pelo fato de representar todos os objetos que possuem o mesmo valor de d_{ij} no que se refere à dominância. Portanto, para saber se um objeto é dominado ou não, basta compará-lo com os representantes dos diversos d_{ij} para este objeto.

Proposição 3.4 O valor de l_d^j é inversamente proporcional a d e diretamente proporcional a j , ou seja: $l_d^j \geq l_{d+1}^j$ e $l_d^j \leq l_d^{j+1}$

Como os valores para os pesos estão distribuídos dentro de uma faixa de valores de um intervalo limitado $[R1, R2]$, e portanto não podem existir mais do que $D = \lfloor R2/R1 \rfloor$ diferentes valores para $d_{i,j}$. Assim, armazenando os valores de l_d^j calculados na iteração anterior, é possível obter um algoritmo de eliminação de objetos dominados em tempo $O(Dn)$ como a seguir:

ALGORITMO MIN (A)

```

1   $m \leftarrow 1$ 
2  Para  $i \leftarrow 1$  até  $D$  faça  $l_i \leftarrow 0$ 
3  Para  $j \leftarrow 2$  até  $n$  faça
4       $troca(j, m + 1)$ 
5      Para  $i \leftarrow 1$  até  $m$  faça
6           $d \leftarrow \lfloor w_j / w_i \rfloor$ 
7          Se  $d = 1$  então  $m \leftarrow m + 1$  e break
8          Se  $l_d > i$  então  $i \leftarrow l_d$  senão  $l_d \leftarrow i$ 
9          Se  $d * p_i \geq p_j$  então break

```

O algoritmo acima, recebe como entrada A a saída do algoritmo DESEMPATA e então testa para cada $j = 1, \dots, n$ se um dos objetos não dominados precedentes $i = 1 \dots, m$ domina j . Assim, para cada par i, j de objetos é encontrada a razão d (linha 6) e este valor é usado para achar o valor do objeto *representante* l_d da iteração anterior. Se $l_d > i$ então como $d_{i,j} = d$ para todo o intervalo $[i, l_d[$ pode-se pular para o índice l_d fazendo $i \leftarrow l_d$ (linha 8). Agora ou i é o índice de um objeto *representante* ou então move-se l_d para a frente até que ele se torne um *representante*.

Exemplo: No algoritmo, se o objeto $(p_j, w_j) = (32, 30)$ estiver sendo avaliado, então teríamos:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
p_i	4	5	7	8	9	10	12	16	17	19	22	25	28	30	31
w_i	5	7	8	9	10	12	13	15	18	19	21	24	26	27	28
d_{ij}	6	4	3	3	3	2	2	2	1	1	1	1	1	1	1

Tabela 3.1: Exemplo de uma iteração do algoritmo de dominância múltipla de Pisinger.

Note que somente os objetos 1, 2, 5 e 8 são *representantes*, e assim, somente eles devem ser comparados com o objeto j para detecção de dominância. Note que $j \prec 8$ e portanto o algoritmo remove objeto j .

Análise: Como o algoritmo considera cada objeto j uma vez e como i percorre todo o conjunto de objetos não dominados que pode ser no máximo m , temos que o algoritmo possui complexidade $O(mn)$. No entanto a linha 8 do algoritmo conduz a uma complexidade $O(Dn)$, já que cada uma das iterações do laço mais interno cai em um dos seguintes casos:

- l_d aponta para um objeto *representante* para o valor de d corrente. Há no máximo D testes de dominância para cada objeto j , e a complexidade nesse caso é $O(Dn)$.
- l_d não aponta para um objeto *representante*, então ele deve ser deslocado para a frente. Nesse caso existem no máximo D ponteiros l_d e cada um dele pode se mover no máximo m vezes para a frente e a complexidade nesse caso é $O(Dn)$.

Pisinger, em sua dissertação de doutorado [Pis95] apresenta diversos resultados comparando o algoritmo de eliminação acima com os algoritmos Dudzinski e MT e demonstra que seu algoritmo é melhor que os

outros. Somente quando o crescimento dos pesos é exponencial que todos os algoritmos degeneram para a complexidade de tempo $O(n^2)$, que é a melhor possível.

A superioridade de seu algoritmo se dá pelo fato de que os dois limites superiores para ele $O(mn)$ e $O(Dn)$ complementam um ao outro muito bem.

3.2.3 Dominância coletiva

Teorema 3.3 *Dado um problema da mochila ilimitada, sejam k e um conjunto de objetos $J \subseteq N \setminus \{k\}$ pertencentes a este problema. O objeto k é dito dominado se*

$$w_k \geq \sum_{j \in J} w_j \quad e \quad p_k \leq \sum_{j \in J} p_j \quad (3.9)$$

Prova: Seja $\vec{x}$ uma solução ótima do problema da mochila, e $\vec{y}$ a solução obtida através da substituição de cada ocorrência do k -ésimo objeto em $\vec{x}$ pelos objetos em J . Então, graças a equação (3.9), temos que

$$\begin{aligned} \sum_{i \in \vec{x}} w_i &\geq \sum_{j \in \vec{y}} w_j, \\ e \\ \sum_{i \in \vec{x}} p_i &\leq \sum_{j \in \vec{y}} p_j. \end{aligned}$$

Portanto $\vec{y}$ é uma solução ótima, o que nos mostra que o k -ésimo objeto pode ser removido do problema, sem alterar a solução ótima do mesmo, ou seja, $F(c, N) = F(c, N \setminus \{k\})$.

Definição 3.6 *Seja J um subconjunto dos objetos pertencentes a uma instância de um problema da mochila e k um outro objeto pertencente ao problema. Se eles satisfazem a condição (3.9) então diz-se que o objeto k é coletivamente dominado pelos objetos do subconjunto J .*

Exemplo: Suponha a seguinte instância do problema da mochila:

$$\begin{aligned} p_1 &= 300, p_2 = 580, p_3 = 301, p_4 = 601, p_5 = 605, p_6 = 322, p_7 = 310 \\ w_1 &= 120, w_2 = 245, w_3 = 130, w_4 = 260, w_5 = 310, w_6 = 194, w_7 = 190 \\ c &= 10. \end{aligned}$$

Se representarmos por $i \prec j$ se o objeto i for dominado pelo objeto j , então no exemplo seriam dominados o objeto 2, 4, 5, já que $2 \prec 1$, $4 \prec \{1, 3\}$ e $5 \prec \{1, 7\}$.

Uma representação gráfica desta relação de dominância pode ser vista na Figura 3.3.

Como se pode perceber, dominância coletiva é uma generalização do conceito de dominância múltipla. Um teste para encontrar todos os elementos coletivamente dominados em uma fase de pré-processamento é computacionalmente proibitivo já que sua resolução é equivalente a resolver um problema da mochila.

Figura 3.3: Relação de dominância coletiva.

3.2.4 Dominância por *threshold*

Teorema 3.4 *Dado um problema da mochila ilimitada, sejam k e um conjunto de objetos $J \subseteq N \setminus \{k\}$ pertencentes a este problema. O objeto k é dito dominado se existir um inteiro positivo α tal que:*

$$\alpha w_k \geq \sum_{j \in J} w_j \quad e \quad \alpha p_k \leq \sum_{j \in J} p_j. \quad (3.10)$$

Prova: Seja $\vec{x}$ uma solução ótima do problema da mochila, e $\vec{y}$ a solução obtida através da substituição de cada uma das α ocorrências do k -ésimo objeto em $\vec{x}$ pelos objetos em J . Então, graças a equação (3.10), temos que

$$\sum_{i \in \vec{x}} w_i \geq \sum_{j \in \vec{y}} w_j,$$

e

$$\sum_{i \in \vec{x}} p_i \leq \sum_{j \in \vec{y}} p_j.$$

Portanto $\vec{y}$ é uma solução ótima, o que nos mostra que as α cópias do k -ésimo objeto podem ser removidas do problema, sem alterar a solução ótima do mesmo. Assim, em uma solução ótima, o objeto k aparece no máximo α vezes.

Definição 3.7 *Seja J um subconjunto dos objetos pertencentes a uma instância de um problema da mochila e k um outro objeto pertencente ao problema. Se eles satisfazem a condição (3.10) então diz-se que o objeto k é dominado por *threshold* pelos objetos do subconjunto J .*

Definição 3.8 O *threshold* do termo k , denotado como $t(k)$ é o maior valor de αw_k tal que a condição (3.10) não seja válida.

Desta forma $x_k = \alpha$, $x_i = 0$, $i \neq k$ é a única solução ótima para UKP com capacidade αw_k e conjunto de objetos N , em outras palavras, quando a capacidade da mochila é igual a αw_k somente o k -ésimo objeto faz parte da solução ótima, aparecendo α vezes.

Exemplo: Suponha a seguinte instância do problema da mochila:

$$\begin{aligned} p_1 &= 200, p_2 = 150, p_3 = 600 \\ w_1 &= 100, w_2 = 150, w_3 = 350 \\ c &= 1000. \end{aligned}$$

Se representarmos por $i \prec j$ se o objeto i for dominado pelo objeto j , então $2 \prec 1$ quando $c \geq 300$, $2 \prec 3$ quando $c \geq 350$ e $3 \prec 1$ quando $c \geq 700$.

Uma representação gráfica desta relação de dominância pode ser vista na Figura 3.4.

Figura 3.4: Relação de dominância por *threshold*.

A eliminação dos objetos dominados por *threshold* constitui uma generalização da dominância coletiva (nesta última $\alpha = 1$), foi apresentada por Andonov [RAR00] e utilizada como meio de se obter uma drástica redução no espaço de busca de UKP em um algoritmo de programação dinâmica batizado de EDUK [RAR00, PYA02].

Em [RAR00] é provado que existe uma relação entre a dominância por *threshold* e a propriedade de periodicidade (apresentada na seção 3.1) que permite testar facilmente se a periodicidade já foi atingida.

Definição 3.9 Para qualquer capacidade y , definimos o conjunto $J(y)$ como

$$J(y) = \{j \in N : t(j) > y\}.$$

Ou seja, $J(y)$ denota o conjunto dos objetos que não são dominados por *threshold* para a capacidade y . O teorema a seguir liga a periodicidade com a dominância por *threshold*.

Teorema 3.5 O conjunto $J(y)$ é unitário, se e somente se a periodicidade já foi atingida e y satisfaz a equação (3.1).

Prova:

$\Rightarrow$ Se a capacidade y^* é a menor capacidade para a qual o conjunto $J(y^*)$ é um unitário, então ele só possui o objeto b . Assim, somente este tipo de objeto pode continuar preenchendo a mochila de forma ótima, ou seja, para todo $y > y^*$ temos que $F(y, N) = F(y - w_b, N) + p_b$, caracterizando a periodicidade.

$\Leftarrow$ Seja k um tipo de objeto diferente de b , mostraremos que o *threshold* de k está abaixo de y . Seja t' o menor múltiplo de w_k tal que $t' \geq y^*$. Então, pela equação (3.1) existe um vetor solução $\vec{x} \in \text{Opt}(\text{UKP}_t^N)$ tal que $x_b \geq 1$ o que implica que $x_k \leq y^*/w_k$. Portanto o objeto k aparece sempre no máximo x_k vezes, e assim seu *threshold* $t(k) \leq x_k w_k \leq y^*$ e $k \notin J(y)$.

Corolário 3.1 Seja $y^* = \min\{y : |J(y)| = 1\}$, ou seja, y^* é igual ao menor valor de y tal que o conjunto $J(y)$ seja unitário.

Este corolário fornece uma condição necessária para se identificar se foi atingida a periodicidade. Veremos na Seção 4.4 como esta condição é utilizada em um algoritmo pra resolução do UKP.

3.3 Síntese do capítulo

Como o problema da mochila é $\mathcal{NP}$ -difícil, não é conhecida nenhuma solução polinomial para o problema. Por isso, é interessante utilizar algoritmos que forneçam recursos para diminuir o tamanho do espaço de busca quando se quer a solução ótima. As duas principais técnicas são relações de dominância e periodicidade.

A propriedade de periodicidade no problema da mochila ilimitada (UKP) estabelece que a partir de uma dada capacidade, apenas o elemento com melhor razão entre valor e peso é que continua a ser acrescentado na solução ótima do problema.

As relações de dominância nos permitem identificar quais objetos não aparecem em uma solução ótima do problema da mochila. Existem relações de dominância tanto para o problema da mochila binária quanto para o problema da mochila ilimitada.

No geral as relações de dominância são exploradas em uma fase de pré-processamento, mas existem também algoritmos que eliminam objetos de forma embutida na própria resolução do problema da mochila.

Existe um compromisso entre a quantidade de objetos dominados que uma relação de dominância pode eliminar e o esforço computacional gasto para tal. As relações de dominância coletiva e dominância por *threshold*, por exemplo, necessitam que se compute um problema semelhante ao problema da mochila para que possam ser postas em prática.

A Tabela 3.2 descreve as relações de dominância em ordem crescente de eliminação de objetos (e consequentemente de complexidade) e respectivo autor da relação de dominância.

No capítulo seguinte, serão apresentados alguns algoritmos sequenciais para o problema da mochila, alguns dos quais utilizando relações de dominância.

Dominância simples	Gilmore e Gomory (1966)
Dominância múltipla	Marthello e Toth (1990)
Dominância coletiva	Pisinger (1995)
Dominância por <i>threshold</i>	Andonov et Al (2000)

Tabela 3.2: Relações de dominância.

ALGUNS ALGORITMOS SEQUENCIAIS PARA RESOLUÇÃO DO UKP.

Neste Capítulo veremos quatro algoritmos sequenciais para a resolução do problema da mochila ilimitada. Primeiro, veremos um algoritmo de programação dinâmica bastante simples, apenas para exemplificar melhor a técnica e mostrar como ela é implementada em um algoritmo. Na Seção 4.2 apresentamos o algoritmo de *branch and bound* descrito em [MT90a], que além de resolver eficientemente mesmo grandes instâncias do problema da mochila, faz uso da propriedade de dominância múltipla. Na Seção 4.3 apresentamos o algoritmo de programação dinâmica, de Garfinkel e Nemhauser [GN72] e que faz uso de recorrências que tornam a resolução de UKP mais rápida que a de algoritmos básicos de programação dinâmica. Finalmente, na seção 4.4, descrevemos o algoritmo EDUK, que usa de forma bastante engenhosa as propriedades de periodicidade e dominância por *threshold* para uma resolução eficiente do problema da mochila.

Notação:

Denotaremos por $\mathbf{UKP}_\gamma^S$ um sub-problema da mochila com capacidade γ e cujos objetos estão em $S \subseteq N$. Portanto, o problema original se descreve como $\mathbf{UKP}_c^N$, onde $|N| = n$. $F(\gamma, S)$ denota o valor ótimo que pode ser obtido para a mochila $\mathbf{UKP}_\gamma^S$. Para todo vetor $\vec{x}$ de dimensão n em $\mathbb{Z}_+^n$, vamos denotar por $W(\vec{x}) = \sum x_i w_i$ como o peso de $\vec{x}$ e $P(\vec{x}) = \sum x_i p_i$ como o seu valor. O par $(W(\vec{x}), P(\vec{x}))$ representa o ponto de solução de $\vec{x}$. $\text{Opt}(\mathbf{UKP}_c^N)$ denota o conjunto dos vetores $\vec{x}$ que são soluções de $\mathbf{UKP}_c^N$ tais que $P(\vec{x}) = F(c, N)$. Também denotaremos por ρ_i a razão entre p_i/w_i e por b o objeto com maior ρ .

4.1 Algoritmo básico

Nesta seção, descreveremos um algoritmo para resolução do problema da mochila ilimitada que faz uso da recursão clássica de Bellman [Bel57] apresentada na Seção 2.3.2. O algoritmo aqui apresentado é bastante simples, porém muito bom para se observar como um algoritmo para resolução de $\mathbf{UKP}_c^N$ faz uso de programação dinâmica.

Recordando, a recursão de Bellman é

$$\begin{cases} F(y, k) = \max\{F(y, k-1), F(c - w_k, k) + p_k\}, \forall k, y > 0, \\ F(0, k) = F(y, 0) = 0, \forall k, y > 0, \\ F(y, k) = \infty, \forall y < 0. \end{cases}$$

A idéia por trás desta recursão é que se $x_k = 0$ em uma solução ótima, então $F(y, k) = F(y, k - 1)$. Se $x_k > 0$ em alguma solução ótima, então este problema é equivalente ao problema de encontrar a solução ótima usando os mesmos k objetos, porém para a mochila de capacidade $c - w_k$, com função objetivo adicionada de p_k ; assim $F(y, k) = F(c - w_k, k) + p_k$. Isto é feito para $k = 1, \dots, n$ e $y = 0, \dots, n$.

Após o algoritmo computar o valor ótimo da função objetivo $F(c, N)$ precisa calcular qual o valor de $\vec{x}$ que propicia esta solução. Isto é feito em uma etapa do algoritmo de programação dinâmica que ficou conhecida como *backtracking phase*, onde se usa uma tabela auxiliar, que permite fazer uma busca do fim para o início na tabela F .

Chamaremos de $p(y, k)$ uma posição na tabela auxiliar. Esta tabela é preenchida em conjunto com a tabela F , e é definida recursivamente como segue:

$$p(y, k) = \begin{cases} 0, & \text{se } F(y, k) = F(y, k - 1); \\ 1, & \text{se } F(y, k) > F(y, k - 1). \end{cases} \quad (4.1)$$

O algoritmo básico para a resolução de UKP_c^N é descrito a seguir:

ALGORITMO UKP BÁSICO (c, N)

```

1    $k \leftarrow 0$ 
2   Para  $y \leftarrow 0$  até  $c$  faça  $F(y, 0) \leftarrow 0$ 
3    $y \leftarrow 1$ 
4   Enquanto  $k < n$  faça
5        $y \leftarrow 1$ 
6        $k \leftarrow k + 1$ 
7       Enquanto  $y < w_k$  faça
8            $F(y, k) \leftarrow F(y, k - 1)$ 
9            $p(y, k) \leftarrow 0$ 
10           $y \leftarrow y + 1$ 
11          Enquanto  $y \leq c$  faça
12               $V \leftarrow p_k + F(y - w_k, k)$ 
13              Se  $V > F(y, k - 1)$  então
14                   $F(y, k) \leftarrow V$ 
15                   $p(y, k) = 1$ 
16              senão
17                   $F(y, k) \leftarrow F(y, k - 1)$ 
18                   $p(y, k) = 0$ 
19               $y \leftarrow y + 1$ 
20  {Terminou de computar  $F(c, N)$ . Começa a fase de backtracking}
21   $z \leftarrow 0$ 
22   $y \leftarrow c$ 
23  Enquanto  $k > 1$  faça
24      Se  $p(y, k) = 1$  então

```

```

25       $z \leftarrow z + 1$ 
26       $y \leftarrow y - w_k$ 
27      senão
28           $x_k \leftarrow z$ 
29           $z \leftarrow 0$ 
30           $k \leftarrow k - 1$ 
31  devolva  $\vec{x}$ 

```

A função da variável z no algoritmo é contar o número de vezes que x_k aparece na solução ótima.

Exemplo:

Considere a seguinte instância do problema da mochila ilimitada:

$$\begin{aligned}
 p_1 &= 11, p_2 = 7, p_3 = 5, p_4 = 1 \\
 w_1 &= 6, w_2 = 4, w_3 = 3, w_4 = 1 \\
 c &= 25.
 \end{aligned}$$

A Tabela 4.1 simula a execução do algoritmo para as funções $F(y, k)$ e $p(y, k)$.

Para saber o valor ótimo, basta observar o valor de $F(25, 4)$ que é 45. Para encontrar o vetor $\vec{x}$ com a solução ótima, a fase de *backtracking* executa da seguinte maneira:

Inicia com $k = 4$ e verifica que $p(25, 4) = 0$, portanto $x_4 = 0$. Decrementa k em uma unidade e verifica que $p(25, 3) = 1$, portanto $x_3 \geq 1$ e faz $y = 25 - w_3 = 25 - 3 = 22$. O algoritmo então examina $p(22, 3) = 0$ e portanto o terceiro objeto só aparece uma vez na solução ótima ($x_3 = 1$). Continuando desta maneira, chega-se que $\vec{x} = (3, 1, 1, 0)$.

Em [GN72] é sugerido que ao invés de se utilizar a tabela $p(y, k)$, utilize-se apenas 1 bit em $F(y, k)$, com isso, toda vez que $p(y, k) = 1$ então $F(y, K) = -F(y, k)$ e quando $p(y, k) = 0$ então $F(y, K) = F(y, k)$.

Análise: Este algoritmo possui complexidade de tempo e memória pseudo-polinomial $O(nc)$. Para maiores detalhes, ver a Seção 2.3.2.

4.2 Algoritmo MTU2

Em um artigo publicado em 1990 [MT90a], os pesquisadores Martello e Toth descrevem um algoritmo de *branch and bound* eficiente mesmo para instâncias muito grandes de UKP_c^N . Os resultados das experiências feitas por esses pesquisadores mostrou que o algoritmo não só consegue resolver, dentro de tempos aceitáveis, instâncias de tamanho muito superior às conseguidas até então por algoritmos de programação dinâmica, como também é melhor que os demais algoritmos de *branch and bound* conhecidos na época.

Um dos pontos chave para o bom desempenho do algoritmo foi a identificação pelos autores da relação de dominância múltipla (ver Seção 3.2.2) no problema da mochila ilimitada. Outros dois pontos importantes do algoritmo é a utilização de limitantes superiores (*upper bounds*) melhores para podar nós na árvore de busca de uma solução ótima (Seção 4.2.1), e a constatação que, no geral, somente os objetos com maior valor de $\rho_j = p_j/w_j$ é que fazem parte de uma solução ótima. Estes objetos fazem parte do que os autores denominaram de problema *core*, conceito que é apresentado na subseção 4.2.2.

y	$F(y, 1)$	$p(y, 1)$	$F(y, 2)$	$p(y, 2)$	$F(y, 3)$	$p(y, 3)$	$F(y, 4)$	$p(y, 4)$
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	2	1
3	0	0	0	0	5	1	5	0
4	0	0	7	1	7	0	7	0
5	0	0	7	1	7	0	8	1
6	11	1	11	0	11	0	11	0
7	11	1	11	0	12	1	12	0
8	11	1	14	1	14	0	14	0
9	11	1	14	1	16	1	16	0
10	11	1	18	1	18	0	18	0
11	11	1	18	1	19	1	19	0
12	22	1	22	0	22	0	22	0
13	22	1	22	0	23	1	23	0
14	22	1	25	1	25	0	25	0
15	22	1	25	1	27	1	27	0
16	22	1	29	1	29	0	29	0
17	22	1	29	1	30	1	30	0
18	33	1	33	0	33	0	33	0
19	33	1	33	0	34	1	34	0
20	33	1	36	1	36	0	36	0
21	33	1	36	1	38	1	38	0
22	33	1	40	1	40	0	40	0
23	33	1	40	1	41	1	41	0
24	44	1	44	0	44	0	44	0
25	44	1	44	0	45	1	45	0

Tabela 4.1: Simulação do algoritmo básico para resolução de $\mathbf{UKP}_c^N$.

4.2.1 Limitantes superiores para $\mathbf{UKP}_c^N$

Como explicado na Seção 2.3, os algoritmos de *branch and bound* para resolução do problema da mochila fazem uso de uma função de limite superior (*upper bound*) do valor da função objetivo para podar nós que não tem condição de superar a melhor solução conhecida até então. Supondo-se, sem perda de generalidade, que os objetos estão ordenados em ordem não crescentes de ρ , então no algoritmo MTU2 são utilizadas as seguintes funções de limitação:

$$U_0 = \left\lfloor p_1 \frac{c}{w_1} \right\rfloor. \quad (4.2)$$

A estratégia por trás desta solução é que se relaxarmos a condição de que x_1 precisa ser inteiro, então a solução ótima é dada por: $x_1 = \frac{c}{w_1}$ e $x_j = 0 (1 < j \leq n)$ e o valor ótimo desta mochila é dada por $\frac{cp_1}{w_1}$.

No entanto, como $\lfloor \frac{cp_1}{w_1} \rfloor \leq \frac{cp_1}{w_1}$, então temos que U_0 é um limitante superior mais próximo do valor ótimo de UKP_c^N .

Se impusermos a condição de x_1 ser inteiro tal que $x_1 \leq \lfloor c/w_1 \rfloor$ e relaxarmos a integralidade para x_2 , então:

$$U_1 = \left\lfloor \frac{c}{w_1} \right\rfloor p_1 + \left\lfloor p_2 \frac{\bar{c}}{w_2} \right\rfloor, \quad (4.3)$$

onde:

$$\bar{c} = c \bmod w_1. \quad (4.4)$$

Note que $U_1 \leq U_0$ pois enquanto U_0 é baseado no valor obtido ao se preencher integralmente com o primeiro objeto, U_1 preenche a mochila com o que for possível (respeitando a integralidade) do primeiro objeto e o restante da mochila com o segundo objeto.

Martello e Toth definem então um limitante superior ainda menor que U_0 e U_1 da seguinte maneira:

$$U_2 = \max(U^0, U^1), \quad (4.5)$$

onde:

$$z' = \left\lfloor \frac{c}{w_1} \right\rfloor p_1 + \left\lfloor \frac{\bar{c}}{w_2} \right\rfloor p_2, \quad (4.6)$$

$$c' = \bar{c}(\bmod w_2), \quad (4.7)$$

$$U^0 = z' + \left\lfloor c' \frac{p_3}{w_3} \right\rfloor, \quad (4.8)$$

$$U^1 = z' + \left\lfloor p_2 - (w_2 - c') \frac{p_1}{w_1} \right\rfloor. \quad (4.9)$$

Ou seja, U^0 é análogo a U_1 porém a variável com integralidade relaxada é x_3 ; U^1 é definido como sendo o valor obtido ao se acrescentar mais uma unidade do objeto 2 a z' em detrimento da redução proporcional da contribuição do objeto 1 (essa diminuição é necessária, já que a capacidade da mochila deve ser respeitada).

Vamos agora provar que $U_2 \leq U_1$:

Partindo de (4.3) e (4.6) podemos reescrever U_1 tal que $U_1 = z' + \lfloor c' p_2 / w_2 \rfloor$; e portanto $U^0 \leq U_1$. Usando o fato de que $\rho_j \geq \rho_{j+1}$ e a equação 4.7, então temos que $c' < w_2$ e portanto $U^1 \leq U_1$. Assim $U_2 \leq U_1$.

O limitante U^1 é obtido forçando que ao menos $\lfloor \bar{c}/w_2 \rfloor + 1$ cópias do segundo objeto sejam utilizadas, a custa de ao menos $\lfloor (w_2 - c')/w_1 \rfloor$ cópias do primeiro objeto da solução obtida em z' e liberando

$c' + \lceil (w_2 - c')/w_1 \rceil w_1$ unidades de capacidade na mochila. Partindo-se das idéias acima, pode-se substituir U^1 pelo limitante superior abaixo:

$$\bar{U}^1 = z' + \left\lfloor \left(c' + \left\lceil \frac{w_2 - c'}{w_1} \right\rceil w_1 \right) \frac{p_2}{w_2} - \left\lceil \frac{w_2 - c'}{w_1} \right\rceil p_1 \right\rfloor. \quad (4.10)$$

A idéia por trás deste limitante é que $c' + \left\lceil \frac{w_2 - c'}{w_1} \right\rceil w_1 \geq w_2$, então $\bar{U}^1 \leq U^1$ e consequentemente o limitante U_3 (definido em 4.11) é um limitante superior para UKP_c^N , e para qualquer instância, $U_3 \leq U_2$.

$$U_3 = \max(U^0, \bar{U}^1). \quad (4.11)$$

A idéia por trás de U_3 é que $\bar{U}^1$ é obtido pela troca de capacidades preenchidas com objetos do primeiro objeto, por capacidades preenchidas pelo segundo objeto.

Portanto os limitantes superiores estudados nesta seção possuem a seguinte relação: $U_3 \leq U_2 \leq U_1 \leq U_0$.

Exemplo:

Suponha a seguinte instância do problema da mochila ilimitada:

$$\begin{aligned} p_1 &= 20, p_2 = 5, p_3 = 1 \\ w_1 &= 10, w_2 = 5, w_3 = 3 \\ c &= 39. \end{aligned}$$

Então teremos:

$$U_0 = 78.$$

$$U_1 = 60 + \lfloor 9 \frac{5}{5} \rfloor = 69.$$

$$U^0 = 65 + \lfloor 4 \frac{1}{3} \rfloor = 66.$$

$$U^1 = 65 + \lfloor 5 - 1 \frac{20}{10} \rfloor = 68.$$

$$U_2 = 68.$$

$$\bar{U}^1 = 65 + \lfloor (4 + \lceil \frac{1}{10} \rceil 10) \frac{5}{5} - \lceil \frac{1}{10} \rceil 20 \rfloor = 59.$$

$$U_3 = 66.$$

4.2.2 O algoritmo

O algoritmo MTU2 nasceu como uma evolução do algoritmo MTU1 [MT77a] (também para resolução de UKP_c^N) desenvolvido pelos mesmos pesquisadores no final da década de 70. Nos experimentos com o MTU1 foi notado que [MT77b]:

- Apesar da complexidade de tempo no pior caso ser exponencial, a maior parte das instâncias de UKP_c^N podem ser resolvidas dentro de um tempo razoável, mesmo para instâncias muito grandes;
- quando a situação acima ocorre, o tempo gasto para ordenar os objetos em ordem não crescente de ρ é responsável por grande parte do tempo computacional da solução;
- somente os objetos com maior valor de ρ acabam fazendo parte da solução ótima apresentada pelo algoritmo.

Os pesquisadores Balas e Zemel notaram propriedades semelhantes no problema da mochila binária e apresentaram um algoritmo [BZ80] que resolve o problema da mochila binária ordenando, na maior parte das instâncias, apenas um pequeno subconjunto dos objetos (denominado de *core*). Formalmente, o *core* é definido como:

$$C = \{1, 2, \dots, \bar{n} \equiv \max\{j : x_j > 0\}\}.$$

O problema *core* é definido como:

$$F(c, C) = \max \sum_{j \in C} p_j x_j \quad (p_j > 0) \quad (4.12)$$

$$\text{sujeito a } \sum_{j \in C} w_j x_j \leq c \quad (w_j, c > 0) \quad (4.13)$$

Onde: $x_j \geq 0$ e inteiro.

Se fosse possível conhecer de antemão o valor de $\bar{n}$, então para resolver UKP_c^N bastaria fazer com que $x_j = 0$ para todo j tal que $\rho_j < \rho_{\bar{n}}$, e resolver o problema *core* ordenando os objetos por ordem não crescente de ρ . Como é impossível obter de antemão o valor de $\bar{n}$, o algoritmo tenta resolver o problema usando um cálculo heurístico $\vartheta = \max(100, \lfloor n/100 \rfloor)$ determinado através de experimentos.

Algoritmo 4.4 O algoritmo MTU2 é descrito abaixo:

ALGORITMO MTU2 (c, N, ϑ)

- 1 $k \leftarrow 0$
- 2 Faça cópias dos objetos em N para dentro de $\bar{N}$
- 3 Repita
- 4 $k \leftarrow \min(k + \vartheta, |N|)$
- 5 Pegue o k -ésimo maior valor de ρ
- 6 $G = \{j \in \bar{N} : \rho_j > \rho_k\}$
- 7 $E = \{j \in \bar{N} : \rho_j = \rho_k\}$
- 8 $\bar{E} = \text{subconjunto de } E \text{ tal que } |\bar{E}| = k - |G|$
- 9 $C = G \cup \bar{E}$

```

10   Ordene os objetos em  $C$  em ordem não crescente de  $\rho$ .
12   Resolva  $\mathbf{UKP}_c^C$  usando MTU1, tal que  $z$  e  $\vec{x}$  sejam a saída deste algoritmo.
13   Se  $k = \varnothing$  {primeira iteração} então
14     calcule  $U_3$ 
15   Se  $z < U_3$  então {redução}
16     Para cada  $j \in \overline{N}/C$  faça
17        $u \leftarrow U_1(j)$ 
18       Se  $u > z$  então  $u \leftarrow U_3j$ 
19       Se  $u \leq z$  então  $\overline{N} \leftarrow \overline{N}/\{j\}$ 
20   Até que  $z = U_3$  ou  $\overline{N} = C$ 
21   Para cada  $j \in N/C$  faça  $x_j \leftarrow 0$ 

```

Algoritmo 4.5 Segue abaixo o algoritmo MTU1 usado pelo MTU2. Este algoritmo pressupõe que os objetos estejam ordenados em ordem não crescente de ρ .

ALGORITMO MTU1 (c, N)

```

1   1) {Inicialização}
2    $z \leftarrow 0$ 
3    $\hat{z} \leftarrow 0$ 
4    $\hat{c} \leftarrow c$ 
5    $p_{n+1} \leftarrow 0$ 
6    $w_{n+1} \leftarrow \infty$ 
7   Para  $k \leftarrow 1$  até  $n$  faça  $\hat{x}_k \leftarrow 0$ 
8    $U \leftarrow U_3$ 
9   Para  $k \leftarrow n$  até 1 faça  $m_k \leftarrow \min w_i : i > k$ 
10   $j \leftarrow 1$ 
11  2) {Constrói uma nova solução corrente}
12  Enquanto  $w_k > \hat{c}$  faça
13    Se  $z \geq \hat{z} + \lfloor \hat{c}p_{j+1}/w_{j+1} \rfloor$  então vá para 5
14     $y \leftarrow \lfloor \hat{c}/w_j \rfloor$ 
15     $u \leftarrow \lfloor (\hat{c} - yw_j)p_{j+1}/w_{j+1} \rfloor$ 
16    Se  $z \geq \hat{z} + yp_j + u$  então vá para 5
17    Se  $u = 0$  então vá para 4
18  3) {Salve a solução corrente}
19     $\hat{c} \leftarrow \hat{c} - yw_j$ 
20     $\hat{z} \leftarrow \hat{z} + yp_j$ 
21     $\hat{x}_j \leftarrow y$ 
22     $j \leftarrow j + 1$ 
23    Se  $\hat{c} \geq m_{j-1}$  então vá para 2
24    Se  $z \geq \hat{z}$  então vá para 5
25     $y \leftarrow 0$ 
26  4) {Atualiza a melhor solução}
27     $z \leftarrow \hat{z} + yp_j$ 
28    Para  $k \leftarrow 1$  até  $j - 1$  faça  $x_k \leftarrow \hat{x}_k$ 

```

```

29   $x_j \leftarrow y$ 
30  Para  $k \leftarrow j + 1$  até  $n$  faça  $x_k \leftarrow 0$ 
31  Se  $z = U$  então FIM
32  5) {Backtrack}
33   $i \leftarrow 0$ 
34   $i \leftarrow \max(k < j : \hat{x}_k > 0)$ 
35  Se  $i = 0$  então FIM {Não existe solução}
36   $\hat{c} \leftarrow \hat{c} + w_i$ 
37   $\hat{z} \leftarrow \hat{z} - p_i$ 
38   $\hat{x}_i \leftarrow \hat{x}_i - 1$ 
39  Se  $z \geq \hat{z} + \lfloor \hat{c}p_{i+1}/w_{i+1} \rfloor$  então
40    {Remove todos os objetos  $i$ }
41     $\hat{c} \leftarrow \hat{c} + w_i \hat{x}_i$ 
42     $\hat{z} \leftarrow \hat{z} - p_i \hat{x}_i$ 
43     $\hat{x}_i \leftarrow 0$ 
44     $j \leftarrow i$ 
45    Vá para 5).
46   $j \leftarrow i + 1$ 
47  Se  $\hat{c} - w_i \geq m_i$  então vá para 2)
48   $h \leftarrow i$ 
49  6) {Tenta substituir um objeto do tipo  $i$  por objetos do tipo  $h$ }
50   $h \leftarrow h + 1$ 
51  Se  $z \geq \hat{z} + \lfloor \hat{c}p_h/w_h \rfloor$  então vá para 5)
52  Se  $w_h = w_i$  então vá para 6).
53  Se  $w_h > w_i$  então
54    Se  $w_h > \hat{c}$  ou  $z \geq \hat{z} + p_h$  então vá para 6)
55     $z \leftarrow \hat{z} + p_h$ 
56    Para  $k \leftarrow 1$  até  $n$  faça  $x_k \leftarrow \hat{x}_k$ 
57     $x_h \leftarrow 1$ 
58    Se  $z \leftarrow U$  então FIM
59     $i \leftarrow h$ 
60    Vá para 6)
61  Senão
62    Se  $\hat{c} - w_h < m_{h-1}$  então vá para 6)
63     $j \leftarrow h$ 
64    Vá para 2)
65  FIM

```

Exemplo:

Suponha a seguinte instância do problema da mochila ilimitada

$$\begin{aligned}
 p_1 &= 20, p_2 = 39, p_3 = 52, p_4 = 58, p_5 = 31, p_6 = 4, p_7 = 5 \\
 w_1 &= 15, w_2 = 30, w_3 = 41, w_4 = 46, w_5 = 25, w_6 = 4, w_7 = 5 \\
 c &= 101.
 \end{aligned}$$

Se $\vartheta = 4$, então o problema *core* é definido por:

$$\begin{aligned} p_1 &= 20, p_2 = 39, p_3 = 52, p_4 = 58 \\ w_1 &= 15, w_2 = 30, w_3 = 41, w_4 = 46 \\ c &= 101. \end{aligned}$$

Aplicando o algoritmo de eliminação de objetos multiplamente dominados da seção 3.2.2 (algoritmo MT), são eliminados os objetos 2 e 4 e portanto o problema resultante é:

$$\begin{aligned} p_1 &= 20, p_3 = 52 \\ w_1 &= 15, w_3 = 41 \\ c &= 101. \end{aligned}$$

Este problema é então resolvido por *branch and bound*, gerando a árvore da Figura 4.1.

Figura 4.1: Árvore de decisão do algoritmo MTU2.

Note que o valor devolvido pelo algoritmo MTU1 ao problema *core* ($z = 132$) não é igual ao limitante $U_3 = 133$. Portanto o algoritmo entra na etapa de redução, calculando $U_1(j)$ para $j = 5, 6$ e 7 .

$$U_1(5) = 31 + \left(100 + \left\lfloor 1 \frac{52}{41} \right\rfloor \right) = 132 \leq z;$$

$$U_1(6) = 4 + \left(120 + \left\lfloor 7 \frac{52}{41} \right\rfloor \right) = 132 \leq z;$$

$$U_1(7) = 31 + \left(120 + \left\lfloor 6 \frac{52}{41} \right\rfloor \right) = 132 \leq z;$$

Como todos os objetos fora do *core* foram reduzidos, então a solução obtida como saída do algoritmo MTU1 usando apenas os objetos do *core* já era a solução ótima, e assim $\vec{x} = (4, 0, 1, 0, 0, 0, 0)$.

Análise: Como explicado na seção 2.3 os algoritmos de *branch and bound* possuem, no pior caso, complexidades de tempo e espaço exponenciais. No entanto, os autores do algoritmo relatam experimentos usando diversos tipos de instâncias do problema da mochila (fortemente correlatas, fracamente correlatas e não-correlatas), e os resultados demonstram que o algoritmo MTU2 consegue resolver instâncias do problema com até 250.000 objetos em menos de 20 segundos, para todas as instâncias, em uma HP 9000/840 [MT90a].

4.3 Algoritmo de Garfinkel e Nemhauser

Em 1972, Garfinkel e Nemhauser descreveram em [GN72] um algoritmo baseado em uma recorrência que acabou se tornando clássica para a resolução do UKP usando técnicas de programação dinâmica, pois é mais eficiente para muitas instâncias do UKP que a recorrência (2.5), necessitando realizar menos cálculos.

Nesta seção descreveremos este algoritmo, mas primeiro vamos apresentar os argumentos matemáticos que lhe dão sustentação.

Definição 4.10 Seja $D(y)$ o conjunto dos objetos que aparecem uma ou mais vezes na solução ótima de UKP_y^N , ou seja $D(y) = \{j | \vec{x} \in \text{UKP}_y^N \text{ e } x_j > 0\}$.

Observe que se $F(y, N) > 0$, então

$$F(y, N) = p_j + F(y - w_j, N), \quad j \in D(y), \quad (4.14)$$

e

$$F(y, N) > p_k + F(y - w_k, N), \quad k \notin D(y). \quad (4.15)$$

Definição 4.11 Vamos denotar por $S(y)$ o conjunto $\{j | w_j \leq y\}$.

Sem perda de generalidade, podemos assumir que $j \neq k \implies w_j \neq w_k$, então $|S(y)| \leq \min(y, n)$. Segue de $D(y) \subseteq S(y)$, (4.14) e (4.15) que

$$F(y, N) = \begin{cases} \max_{j \in S(y)} p_j + F(y - w_j, N) & \text{se } S(y) \neq \emptyset; \\ 0 & \text{se } S(y) = \emptyset. \end{cases} \quad (4.16)$$

Uma forma de calcular $F(c, N)$ é calcular $F(y, N)$ através de um algoritmo de programação dinâmica para $y = 1, \dots, c$. Note porém que para cada valor de $y \geq \max_{j \in N}(w_j)$ a recorrência (4.16) precisa realizar $|S(y)|$ adições e comparações para calcular (4.16). O algoritmo proposto pelos autores propõe uma melhora na recorrência (4.16) que permite uma grande redução no número de cálculos realizados.

Seja

$$Q(y) = \{j | j \in S(y) \text{ e } j \leq d(y - w_j)\}. \quad (4.17)$$

Onde $d(y)$ é definido como

$$d(y) = \begin{cases} \min\{j | j \in D(y)\} & \text{se } D(y) \neq \emptyset; \\ n & \text{se } D(y) = \emptyset. \end{cases} \quad (4.18)$$

Como consequência de (4.18) e (4.14), e que, se $j \in D(y)$ então $D(y - w_j) \subseteq D(y)$, e temos que

$$d(y) \leq d(y - w_j), \quad \forall j \in D(y). \quad (4.19)$$

A nova recorrência fica:

$$F(y, N) = \begin{cases} \max_{j \in Q(y)} p_j + F(y - w_j, N) & Q(y) \neq \emptyset \\ 0 & Q(y) = \emptyset. \end{cases} \quad (4.20)$$

Para aplicar a recorrência acima no cálculo de UKP_c^N suponha que $F(0, N), \dots, F(y - 1, N)$ e $d(0), \dots, d(y - 1)$ são conhecidos, então a seguinte proposição é verdadeira:

Proposição 4.5 $\forall j \in S(y)$ a adição $p_j + F(y - w_j, N)$ só é realizada se $j \leq d(y - w_j)$.

Prova:

Suponha que $j \notin D(y)$, então não é necessário comparar j para a obtenção de $g(y)$.

Suponha que $j \in D(y)$, então $g(y) = p_j + g(y - w_j)$. Da definição de $d(y)$ e como $d(y - w_j) = k$, temos que $k \in D(y - w_j)$.

Pelo princípio de otimalidade [Bel57] temos que $g(y) = p_k + g(y - w_k)$

Portanto é possível obter $g(y)$ usando o k -ésimo objeto e conseqüentemente não é necessário comparar j para a obtenção de $g(y)$.

Assim, em cada iteração quanto menor for o valor de $d(y)$, menos computações precisam ser realizadas. Assim é desejável que os objetos sejam indexados de forma que aqueles que tem maior chance de aparecerem em uma solução ótima tenham índices menores que os demais. Uma indexação com tal propriedade é uma tal que ordena o índice dos objetos por ordem não-crescente de ρ (lembrando que $\rho_j = p_j/w_j$), $\rho_1 \geq \rho_2 \geq \dots \geq \rho_n$.

O algoritmo de programação dinâmica que utiliza a recorrência (4.20) é descrito a seguir. Ele recebe como entradas a capacidade c da mochila e o conjunto dos tipos de objetos N e devolve como resposta um

vetor $\vec{x}$ com a quantidade de vezes que cada objeto deve aparecer em uma solução ótima. Deve-se atentar que a indexação da resposta em $\vec{x}$ é a indexação obtida após os objetos em N terem sido ordenados em ordem não-crescente de ρ .

ALGORITMO GN (c, N)

```

1  Ordene os objetos de forma que  $\rho_1 \geq \rho_2 \geq \dots \geq \rho_n$ 
2  Faça  $F(0, N) \leftarrow 0$ ,  $d(0) \leftarrow n$  e  $y \leftarrow 0$ 
3  Faça
4       $y \leftarrow y + 1$ 
5       $j \leftarrow 1$ 
6      Enquanto  $j \leq n$  faça
7          Se  $y - w_j \geq 0$  e  $j \leq d(y - w_j)$  então
8               $V \leftarrow F(y - w_j, N) + p_j$ 
9              Se  $V \geq F(y, N)$  então  $F(y, N) = V$  e  $d(y) = j$ 
10              $j \leftarrow j + 1$ 
11  Enquanto  $y \leq c$ 
12  {Backtracking}
13  Enquanto  $F(y, N) \geq 0$  faça
14       $x_{d(y)} \leftarrow x_{d(y)} + 1$ 
15       $y \leftarrow y - w_{d(y)}$ 
16  Retorne  $\vec{x}$ 

```

Exemplo:

Considere a seguinte instância do problema da mochila ilimitada:

$$\begin{aligned}
 p_1 &= 11, p_2 = 7, p_3 = 5, p_4 = 1 \\
 w_1 &= 6, w_2 = 4, w_3 = 3, w_4 = 1 \\
 c &= 25.
 \end{aligned}$$

A Tabela 4.2 simula a execução do algoritmo para as funções $F(y, k)$ e $d(y)$

Quando o algoritmo termina de computar $F(25, 4) = 45$ e $d(25) = 1$, então ele entra na etapa de *backtracking*, onde $d(25) = 1$, $d(25 - w_1) = d(19) = 1$, $d(19 - w_1) = d(13) = 1$, $d(7 - w_1) = d(1) = 4$ e portanto o $\vec{x} = (4, 0, 0, 1)$.

Análise:

O algoritmo é $O(c)$ em consumo de memória e realiza $O(nc)$ computações. No entanto, como para muitas instâncias o valor de $d(y)$ acaba sendo pequeno durante boa parte do algoritmo, na prática são realizados muito menos cálculos.

4.4 Algoritmo EDUK

Em 2000 Andonov, Poirriez e Rajopadhye [RAR00] apresentaram um algoritmo de programação dinâmica que se vale de todas as propriedades de dominância e periodicidade para UKP conhecidas. Até então, diver-

y	$F(y, 1)$	$d(y)$
0	0	0
1	1	4
2	2	4
3	5	3
4	7	2
5	8	2
6	11	1
7	12	1
8	14	2
9	16	1
10	18	1
11	19	1
12	22	1
13	23	1
14	25	1
15	27	1
16	29	1
17	30	1
18	33	1
19	34	1
20	36	1
21	38	1
22	40	1
23	41	1
24	44	1
25	45	1

Tabela 4.2: Simulação do algoritmo GN para resolução de UKP_c^N .

Os algoritmos se valiam independentemente de algumas dessas propriedades mas somente neste trabalho é que foi apresentado um algoritmo que se beneficiava de todas as propriedades conhecidas para UKP.

Os autores começam definindo uma nova relação de dominância, chamada de dominância por *threshold*, que estabelece que a partir de uma certa capacidade t_j o objeto j não contribui mais para uma solução ótima. Mostra-se então que a dominância por *threshold* é uma generalização do conceito de dominância coletiva e estabelece uma relação entre esta relação de dominância e a bem conhecida propriedade de periodicidade em UKP (todos estes resultados são apresentados em maiores detalhes na seção 3.2.4).

A partir daí é apresentado um algoritmo batizado com o nome de EDUK (*Efficient Dynamic Programming Algorithm for Unbounded Knapsack Problem*), que procura se valer da relação de dominância por *threshold* (e seu relacionamento com as propriedades de dominância coletiva e periodicidade) para resolver de forma eficiente o problema da mochila ilimitada. A utilização da dominância por *threshold* numa etapa de pré-processamento do UKP não é vantajosa, já que para cada termo que se deseje verificar se é dominado, é necessário resolver um problema equivalente ao problema da mochila. A idéia do EDUK é usar o fato

de que durante a resolução do problema por programação dinâmica são calculados diversos subproblemas da mochila e embutir a eliminação dos objetos dominados dentro do próprio algoritmo de resolução do problema.

Para tal, o algoritmo faz uso da função $l(y) \equiv d(y)$ definida em (4.18) além de uma outra função $ll(j, y)$ definida abaixo:

$$ll(j, y) = \begin{cases} \max\{y' | y' < y\}, & \text{se } l(y') = j; \\ 0, & \text{se } \forall y' < y \Rightarrow l(y') \neq j. \end{cases} \quad (4.21)$$

O *threshold* de um objeto j (denotado por $t(j)$) é o maior valor de capacidade para o qual este objeto seja o único necessário para se obter uma solução ótima. Portanto, para todos os objetos $j \neq b$, se $y > y^*$ (y^* foi definido na Seção 3.1) então $t(j) \leq ll(j, y) < ll(b, y)$.

Assim, dado o conjunto

$$J(y) = \{j \in N | w_j \leq y \text{ e } (ll(j, y) + w_j) > y\}, \quad (4.22)$$

o algoritmo calcula UKP_c^N através da seguinte recorrência

$$F(y, N) = \max_{j \in J(y)} p_j + F(y - w_j, N), \quad y = 1, \dots, c. \quad (4.23)$$

Onde: $F(0, 0) = 0$ e $F(y, N) = -\infty$ para $y < 0$.

Como se pode perceber, a recorrência (4.23) é semelhante a recorrência usada no algoritmo de Garfinkel e Nemhauser, excetuando-se que os objetos calculados estão no conjunto $J(y)$ dos objetos que não são dominados por *threshold* até a capacidade y .

Da mesma forma que no algoritmo de Garfinkel e Nemhauser, pequenos valores de $l(y)$ proporcionam uma grande redução no número de computações realizadas, daí também ser conveniente no EDUK ordenar os objetos em $J(y)$ em ordem não crescente de p . Porém, note que enquanto no algoritmo de Garfinkel e Nemhauser esta ordenação é realizada sobre o conjunto N , no EDUK esta ordenação é realizada sobre o conjunto $|J|$, que possui cardinalidade muito menor que N .

Infelizmente em nenhum dos trabalhos os autores apresentam uma descrição do algoritmo de forma estruturada. Todas as descrições do EDUK publicadas são feitas em linguagem natural. O autor desta dissertação entrou em contato com os autores do EDUK e perguntou sobre a possibilidade de acesso ao código-fonte da implementação do algoritmo, o que foi respondido com uma negativa.

Assim, a descrição do algoritmo que segue abaixo é baseada no entendimento das descrições publicadas em pelos autores do algoritmo [RAR00, PYA02] e testada com uma implementação em Java para diversas instâncias de UKP.

```

ALGORITMO EDUK ( $c, N$ )
1  {Inicialização}
2  Retire do conjunto  $N$  o objeto  $j$  com menor peso e adicione-o ao conjunto  $J$ 
3  Para  $y \leftarrow 0$  até  $w_j$  faça
4       $F(y, N) \leftarrow 0$ 
5       $l(y) = n$ 
6       $c_0 \leftarrow w_j$ 
7       $F(w_j, N) \leftarrow p_j$ 
8       $l(y) \leftarrow j$ 
9       $ll(j) \leftarrow y$ 
10      $y \leftarrow w_j$ 
11     {Fase de redução}
12     Enquanto  $|N| > 0$  faça
13         Retire do conjunto  $N$  o objeto  $j$  com menor peso
14         Para  $y \leftarrow c_0 + 1$  até  $w_j$ 
15             Calcule  $F(y, N)$  usando a recursão (4.23)
16             Atualize  $l$  e  $ll$ 
17             Se  $w_j > F(y, N)$  então {Objeto  $j$  não é coletivamente dominado}
18                  $F(y, N) \leftarrow p_j$ 
19                  $l(y) \leftarrow j$ 
20                  $ll(j) \leftarrow y$ 
21                  $U \leftarrow U \cup \{j\}$ 
22             Para todo  $j \in J$ 
23                 Se  $ll(j) + p_j < y$  {Objeto  $j$  é dominado por threshold}
24                 Faça  $J \leftarrow J \setminus \{j\}$ 
25              $c_0 \leftarrow w_j$ 
26         {Fase padrão}
27         Enquanto  $y \leq c$  e  $|J| > 1$  faça
28              $y \leftarrow y + 1$ 
29             Calcule  $F(y, N)$  usando a recursão (4.23)
30             Atualize  $l$  e  $ll$ 
31             Para todo  $j \in J$ 
32                 Se  $ll(j) + w_j \leq y$  {Objeto  $j$  é dominado por threshold}
33                 Faça  $J \leftarrow J \setminus \{j\}$ 
34         Se  $y < c$  então  $\{|J| = 1$ , pode-se usar propriedade de periodicidade de UKP}
35         Seja  $b \in J$ 
36          $x_b \leftarrow (c - y)/w_b + 1$ 
37          $y \leftarrow c - x_b w_b$ 
38         {Backtracking}
39         Enquanto  $F(y, N) \geq 0$  faça
40              $x_{l(y)} \leftarrow x_{l(y)} + 1$ 
41              $y \leftarrow y - w_{l(y)}$ 
42     Retorne  $\vec{x}$ 

```

Seja c_0 o valor da maior capacidade tal que, para todo $y \leq c_0$, o valor ótimo já foi calculado pelo

EDUK. Sejam também J e N os estados destes conjuntos para a capacidade c_0 , identificamos os seguintes invariantes no Algoritmo EDUK:

- Todos os objetos em N possuem pesos estritamente maiores que c_0 ;
- Todos os objetos em J possuem pesos menores que c_0 e não são coletivamente dominados;
- Na fase padrão, ao final da iteração, J não possui nenhum objeto que é *threshold* dominado.

As linhas 15, 16 e 29, 30 do Algoritmo 4.4 indicam que se deve calcular $F(y, N)$ usando a recursão (4.23) e também atualizar l e ll . Isto foi feito de forma a privilegiar a facilidade de entendimento do EDUK, no entanto, indicamos a seguir a maneira de se realizar estas operações.

PROCEDIMENTO RECORRÊNCIA ()

```

1  max ← 0
2  Para todo  $j \in J$  faça
3    Se  $j \leq l(y - w_j)$  então
4       $V \leftarrow F(y - w_j, N) + p_j$ 
5      Se  $V > \text{max}$  então
6        max ←  $V$ 
7       $l(y) \leftarrow j$ 
8       $ll(j) \leftarrow y$ 
9   $F(y, N) \leftarrow \text{max}$ 

```

Exemplo:

Suponha a seguinte instância do problema da mochila ilimitada

$$\begin{aligned}
 p_1 &= 300, p_2 = 580, p_3 = 301, p_4 = 601, p_5 = 605, p_6 = 322, p_7 = 310 \\
 w_1 &= 120, w_2 = 245, w_3 = 130, w_4 = 260, w_5 = 310, w_6 = 194, w_7 = 190 \\
 c &= 2900.
 \end{aligned}$$

A Tabela 4.3 simula a execução do algoritmo para as funções $F(y, k)$ e $l(y)$ e J .

Na coluna $l(y)$ os valores subscritos indicam que $t(l(y)) = y$ e portanto esta é a última vez que este objeto será acrescentado numa solução ótima para qualquer capacidade daqui por diante. Note que a periodicidade é detectada pelo algoritmo quando $y = 914$ (o conjunto J possui apenas um elemento), a partir de então o algoritmo faz $x_1 = (2900 - 914)/120 = 16$ e entra na fase de *backtracking* que é executada exatamente da mesma forma que no algoritmo GN, descrito na seção anterior. Com isso o vetor solução é $\vec{x} = (22, 0, 2, 0, 0, 0, 0)$.

Análise: O algoritmo EDUK possui complexidade de tempo $O(nc)$. No entanto, resultados apresentados em [RAR00] demonstram que esta análise é muito conservadora. Na grande maioria das instâncias do problema da mochila, o EDUK realiza uma quantidade muito menor de cálculos (em virtude das relações de dominância e periodicidade), apresentando um desempenho melhor do que o algoritmo MTU2[MT90a], aceito até então como o melhor algoritmo para resolver UKP para instâncias de tamanho grande (da ordem de milhares de objetos).

y	$F(y, N)$	$l(y)$	J	y	$F(y, N)$	$l(y)$	J	y	$F(y, N)$	$l(y)$	J
0	0	7	$\emptyset$	500	1202	1	$\{1,3\}$	730	1801	1	$\{1,3\}$
120	300	1	$\{1\}$	510	1203	1	$\{1,3\}$	740	1802	1	$\{1,3\}$
130	301	3	$\{1,3\}$	520	1204	3	$\{1,3\}$	750	1803	1	$\{1,3\}$
190	310	<u>7</u>	$\{1,3,7\}$	550	1210	1	$\{1,3\}$	760	1804	1	$\{1,3\}$
194	322	<u>6</u>	$\{1,3,6,7\}$	554	1222	1	$\{1,3\}$	770	1805	1	$\{1,3\}$
240	600	1	$\{1,3,6,7\}$	564	1223	1	$\{1,3\}$	780	1806	<u>3</u>	$\{1,3\}$
250	601	1	$\{1,3,6,7\}$	574	1224	1	$\{1,3\}$	790	1810	1	$\{1,3\}$
260	602	3	$\{1,3,6,7\}$	584	1225	3	$\{1,3\}$	794	1822	1	$\{1,3\}$
310	610	1	$\{1,3,6,7\}$	600	1500	1	$\{1,3\}$	804	1823	1	$\{1,3\}$
314	622	1	$\{1,3,6,7\}$	610	1501	1	$\{1,3\}$	814	1824	1	$\{1,3\}$
324	623	3	$\{1,3,6,7\}$	620	1502	1	$\{1,3\}$	824	1825	1	$\{1,3\}$
360	900	1	$\{1,3,6,7\}$	630	1503	1	$\{1,3\}$	834	1826	1	$\{1,3\}$
370	901	1	$\{1,3,6,7\}$	640	1504	1	$\{1,3\}$	840	2100	1	$\{1,3\}$
380	902	1	$\{1,3,6\}$	650	1505	3	$\{1,3\}$	850	2101	1	$\{1,3\}$
390	903	3	$\{1,3\}$	670	1510	1	$\{1,3\}$	860	2102	1	$\{1,3\}$
430	910	1	$\{1,3\}$	674	1522	1	$\{1,3\}$	870	2103	1	$\{1,3\}$
434	922	1	$\{1,3\}$	684	1523	1	$\{1,3\}$	880	2104	1	$\{1,3\}$
444	923	1	$\{1,3\}$	694	1524	1	$\{1,3\}$	890	2105	1	$\{1,3\}$
454	924	3	$\{1,3\}$	704	1525	1	$\{1,3\}$	900	2106	1	$\{1,3\}$
480	1200	1	$\{1,3\}$	714	1526	3	$\{1,3\}$	910	2110	1	$\{1,3\}$
490	1201	1	$\{1,3\}$	720	1800	1	$\{1,3\}$	914	2122	1	$\{1\}$

Tabela 4.3: Simulação do algoritmo EDUK para resolução de $\mathbf{UKP}_c^N$.

PROBLEMAS DA MOCHILA NO CONTEXTO DO PARALELISMO

Existem diversos estudos e algoritmos para a resolução de problemas da mochila em arquiteturas paralelas. Esses algoritmos são projetados para arquiteturas que vão desde o modelo teórico PRAM e cobrem praticamente todas as arquiteturas paralelas existentes. As técnicas mais utilizadas são:

- variações do algoritmo de duas-listas [HS74, CFH89, Kar84], onde os objetos são divididos em duas ou mais listas e então, em paralelo, é resolvido em cada lista seu subproblema e a solução de cada lista é combinada em uma solução final.
- adaptação para resolução em paralelo da técnica de programação dinâmica [LSS88b, LS91, GT02];
- adaptação para resolução em paralelo da técnica de *branch and bound* [KT88];
- transformar o problema da mochila em um outro problema de solução eficiente conhecida [Ten90];
- resolução de forma aproximada.

Em [GW93] é realizado um *survey* das soluções existentes para diversas arquiteturas paralelas.

Neste capítulo são apresentados 3 algoritmos para a resolução paralela do problema da mochila. Todos fazem uso da técnica de programação dinâmica, sendo que dois são projetados para hipercubos (Seção 5.1 e 5.3) e um é projetado para um vetor sistólico (Seção 5.2).

Apresentamos primeiro o algoritmo descrito em [LS91] para a resolução do problema da mochila binária em hipercubos. Boa parte das soluções paralelas para a resolução de problemas da mochila até então (final da década de 80) usava quantidade exponencial de processadores (em função do tamanho da entrada) para encontrar a solução ótima. Em 1988, Lee *et al.* [LSS88b] propuseram um algoritmo, baseado na técnica de divisão e conquista, para arquitetura de hipercubos, que podia executar em máquinas com qualquer número de processadores. Entretanto, o algoritmo era pouco eficiente e não escalava bem para valores grandes de capacidade ($c \gg O(n)$) e muitos processadores. O algoritmo de Lin e Storer, descrito na seção 5.1, é um aprimoramento da estratégia empregada em [LSS88b], conseguindo um algoritmo eficiente e escalável para arquitetura de hipercubos.

Na seção 5.2 é apresentado um algoritmo paralelo para a resolução do problema da mochila em arquitetura sistólica linear de Andonov e Rajopadhye [AR97]. A estratégia empregada pelos autores serve tanto para resolver o problema da mochila binária quanto para resolver o problema da mochila ilimitada, mas os autores deram ênfase maior no algoritmo e nos experimentos realizados para este último caso. Este

algoritmo usa uma representação em forma de grafo dirigido das dependências entre os valores da função objetivo dentro da técnica de programação dinâmica, transformando a paralelização do problema da mochila num problema de escalonamento de tarefas interdependentes a processadores alinhados sequencialmente em VLSI.

Na Seção 5.3 é apresentado um algoritmo de Goldman e Trystram para a resolução do problema da mochila ilimitado em hipercubos, descrito em [GT02]. Como o algoritmo da seção 5.2, este algoritmo também transforma a paralelização da resolução via programação dinâmica do problema da mochila num problema de escalonamento. Porém, nesse caso, o algoritmo é desenhado para arquitetura de hipercubos.

A escolha por estudar estes algoritmos se deveu ao fato de serem os melhores conhecidos para suas respectivas arquiteturas.

5.1 Um algoritmo eficiente para resolução do problema da mochila binária em hipercubos

Nesta seção, apresentamos o algoritmo de Lin e Storer [LS91] para a resolução do problema da mochila binária em hipercubos. Este algoritmo não faz suposições sobre o número de processadores da máquina, e é capaz de resolver o problema da mochila binária em $O(\frac{nc}{p})$ em uma arquitetura EREW PRAM de p processadores. Como visto na Seção 2.3.2 o melhor algoritmo seqüencial conhecido para resolução do problema da mochila possui complexidade de tempo $O(nc)$, portanto o algoritmo de Lin e Storer tem *speedup* ótimo ($O(p)$).

Como é habitual em algoritmos de programação dinâmica, o algoritmo é dividido em duas fases, uma que computa o valor ótimo da função objetivo e uma outra, de *backtracking*, que encontra o $\vec{x}$ com a solução ótima. Na primeira etapa o algoritmo se vale de uma formulação multi-estágio do problema da mochila.

Seja $\mathbf{BKP}_c^N$ o problema da mochila binária de capacidade c e para n objetos, contidos no conjunto N . Seja N_j definido como sendo o subconjunto de N contendo os primeiros j objetos, tal que $N_n \equiv N$ então $\mathbf{BKP}_k^{N_j}$ é o subproblema com capacidade k ($0 \leq k \leq c$) contendo os primeiros j ($1 \leq j \leq n$) objetos. Seja $F(k, j)$ definido como sendo o valor ótimo de $\mathbf{BKP}_c^{N_j}$, então pelo princípio de otimalidade temos que

$$F(k, j) = \max\{F(k, j-1), F(k - w_j, j-1) + p_j\}, \quad (5.1)$$

onde $F(k, 0) = 0$ para $k \geq 0$.

O algoritmo atribui para cada um dos p processadores disponíveis $\lceil (c+1)/p \rceil$ elementos necessários para o cálculo de $F(c, j)$. Assim, o primeiro processador p_1 calcula $F(1, j), F(2, j), \dots, F(\lceil (c+1)/p \rceil, j)$; o segundo processador, p_2 , calcula $F(\lceil (c+1)/p \rceil + 1, j), F(\lceil (c+1)/p \rceil + 2, j), \dots, F(2\lceil (c+1)/p \rceil, j)$; e assim por diante.

O algoritmo é descrito a seguir. Para facilitar a descrição definimos $d = \lceil (c+1)/p \rceil$. Cada processador (p) possui um número que vai de 0 até o número total de processadores ($p_{\max}$) menos um, e sabe de antemão qual o seu próprio número. A notação $F_j(p, k)$ denota o valor de $\mathbf{BKP}_c^{N_j}$ calculado pelo processador de índice p .

ALGORITMO LINSTORER(c, N)

```

1  {Inicialização}
1  Para  $p \leftarrow 1$  até  $p_{\max}$  faça em paralelo
2    Para  $i \leftarrow 0$  até  $d - 1$  faça
3       $F_0(y, i) \leftarrow 0$ 
5  {Cálculo da função objetivo}
4  Para  $j \leftarrow 1$  até  $n$  faça em paralelo
5    Para  $i \leftarrow 0$  até  $d - 1$  faça em paralelo
6      Se  $(0 \leq p < (w_j - i)/d)$  então  $F_j(p, i) \leftarrow F_{j-1}(p, i)$ 
7      Se  $((w_j - k)/d \leq p < p_{\max})$  então
8         $F_j(p, i) \leftarrow \max\{F_{j-1}(p, i), F_{j-1}(\lfloor (pd + i - w_j)/d \rfloor, (pd + i - w_j) \bmod d) + p_j\}$ 
9  {Backtracking}
10 Para  $p \leftarrow 1$  até  $p_{\max}$  faça em paralelo
11    $W \leftarrow c$ 
12    $P \leftarrow f_n(p_{\max} - 1, c \bmod d)$ 
13 Para  $j \leftarrow n$  até 1 faça em paralelo
14   Se  $F_{j-1}(\lfloor W/d \rfloor, W \bmod d) = P$  então  $x_j \leftarrow 0$ 
15   Senão
16      $x_j = 1$ 
17      $P \leftarrow P - p_j$ 
18      $W \leftarrow W - w_j$ 

```

Note que o cálculo de $F(k, j)$ em uma iteração depende apenas dos valores já calculados em uma iteração anterior. A Figura 5.1 ilustra o procedimento para o cálculo de $F(k, j)$ em uma máquina com c processadores.

Análise: Em uma arquitetura EREW PRAM, a fase de cálculo da função objetivo tem complexidade de tempo $O(nd) = O(nc/p)$, enquanto que na fase de *backtracking* o algoritmo possui complexidade de tempo $O(n)$. Portanto, a complexidade de tempo do algoritmo é $O(nc/p)$. Como dito anteriormente, seu *speedup* é $O(p)$ e sua eficiência por processador é $O(1)$, portanto ótima.

Para a arquitetura de hipercubo, como o tempo de comunicação de dados entre dois processadores pode levar até $\log p$ unidades de tempo, temos que a complexidade de tempo do algoritmo de Lin e Storer é $O(\frac{nc \log p}{p})$ em um hipercubo com p processadores.

5.2 Solução paralela de UKP usando programação dinâmica em arquitetura sistólica

Em 1997 Andonov e Rajopadhye publicaram um artigo [AR97] em que apresentavam uma solução paralela para o problema da mochila ilimitada para arquiteturas sistólicas, bem como detalhes de implementação da solução em VLSI. A solução é baseada na técnica de programação dinâmica e é independente do número de EPs (Elementos de Processamento) que compõem o vetor sistólico. Assumindo-se que hajam q EPs e que cada um possua memória α de memória, então a complexidade de tempo média da solução é de $O(\frac{nc}{q} \lceil \frac{w_{\max} + w_{\min}}{\alpha} \rceil)$, onde $w_{\max}$ e $w_{\min}$ representam o máximo e o mínimo dentre os pesos dos objetos. Assim, sempre que $\alpha \geq (w_{\max} + w_{\min})$, então o algoritmo possuirá aceleração ótima. Nesta seção faremos a

Figura 5.1: Cálculo de $F(k, j)$ no algoritmo de Lin e Storer usando c processadores.

suposição que cada EP possui mais do que $w_{\max} + w_{\min}$ de memória, de forma que possamos capturar a idéia principal do algoritmo sem entrar na complexidade técnica para lidar com problemas de conflito por falta de memória. Em [AR97] é feita uma análise minuciosa das alterações necessárias quando $\alpha < (w_{\max} + w_{\min})$ e seu impacto na ordem do tempo de execução do algoritmo.

O algoritmo calcula $F(c, N)$, porém para tal usa a seguinte recorrência para o cálculo do valor ótimo:

$$f(y, j) = \begin{cases} 0 & \text{se } y = 0 \text{ ou } j = 0, \\ f(y, j-1) & \text{se } y < w_j \text{ e } j > 0, \\ \max(f(y, j-1), f(y-w_j, j) + p_j) & \text{se } y \geq w_j \text{ e } j > 0. \end{cases} \quad (5.2)$$

Onde $F(c, N) = f(c, n)$, $0 \leq y \leq c$ e $0 \leq j \leq n = |N|$.

A comunicação necessária para o cálculo de $F(c, N)$ pode ser representada através de um grafo dirigido, chamado grafo de dependência (GD) $\mathcal{G} = (\mathcal{D}, \mathcal{A})$ onde $\mathcal{D} = \{(y, j) \in \mathbb{N}^2 : 0 \leq y \leq c, 1 \leq j \leq n\}$ é o conjunto de vértices enquanto o conjunto $\mathcal{A}$ é o conjunto de arestas. Cada vértice $(y, j) \in \mathcal{D}$ representa uma operação realizada pelo algoritmo, enquanto as arestas representam as dependência dos dados.

A Figura 5.2 ilustra um exemplo de grafo de dependência para uma instância de UKP com $n = 3$, $c = 6$, $w_1 = 4$, $w_2 = 3$ e $w_3 = 2$.

Note que em todas as colunas do grafo, a definição da aresta de dependência é dada pelo valor do peso dos objeto (que é um dos dados de entrada para o problema) o que transforma a recorrência (5.2) difícil de ser mapeada para um vetor sistólico através de métodos tradicionais, como o de síntese [RF90] por exemplo. No entanto, como em cada coluna são realizadas as mesmas operações e a dependência se dá ou

Figura 5.2: Exemplo de grafo de dependência para uma instância de UKP.

entre colunas adjacentes ou dentro da mesma coluna é possível mapear o GD em um vetor sistólico de n EPs idênticas chamadas de C_k , $k = 1, 2, \dots, n$, responsáveis pelo processamento na coluna k do GD.

Essa maneira de proceder no mapeamento de um GD é também chamada de método de projeção de dependência e usa as seguintes funções:

- **Função de ordem temporal:** é um mapeamento $t : \mathcal{D} \rightarrow \mathbb{N}$, tal que se a computação no vértice $v \in \mathcal{D}$ depende de $w \in \mathcal{D}$ então $t(v) > t(w)$.
- **Função de alocação:** é um mapeamento $a : \mathcal{D} \rightarrow [1, q]$, tal que $a(v)$ é o número do EP que executa os cálculos das operações pela qual o vértice $v \in \mathcal{D}$ é responsável. A função de alocação a deve ser escolhida de forma que nenhuma EP tenha mais que uma operação a realizar num dado instante.
- **Função de endereçamento:** é um mapeamento $addr : \mathcal{D} \rightarrow [1, \alpha]$, tal que $addr(v)$ é o endereço da localização na memória do EP $a(v)$, onde os dados de $v \in \mathcal{D}$ são armazenados.

Para este algoritmo os autores definiram que para qualquer $(y, j) \in \mathcal{D}$:

- $a(y, j) = (j \bmod p) + 1$: Dessa forma para qualquer y , o valor de $f(y, j)$, $j = 1, 2, \dots, n$ é computado sequencialmente pelo mesmo EP (de índice $(j \bmod p) + 1$);

- $t(y, j) = (\lfloor j/p \rfloor)c + (j \bmod p) + y$: Então $t(c, n) = (\lfloor n/p \rfloor)c + (n \bmod p) + c$ no caso geral, porém quando $p = n$, então $t(c, n) = c + n$.

5.2.1 Algoritmo

Agora que o escalonamento do problema e dos processadores já foi definido, apresentamos o algoritmo que é executado por cada EP. Cada EP C_k armazena o valor p_k , w_k e k nos registradores p , w e k respectivamente, enquanto que os dados de entrada em C_0 durante os primeiros $c + 1$ instantes t , $t = 0, 1, \dots, c$ são $f_{in} = 0$, $u_{in} = 0$ e $y_{in} = t$. A Figura 5.3 ilustra a EP, enquanto o algoritmo executado por cada EP C_k , $k = 1, 2, \dots, n$, é dado logo a seguir.

Figura 5.3: Representação de um Elemento de Processamento (EP).

ALGORITMO EP

```

1  repita
2     $addr \leftarrow y_{in} \bmod w$ 
3    Se  $y_{in} < w$  então
4       $f_{out} \leftarrow f_{in}$ 
5       $u_{out} \leftarrow u_{in}$ 
6    Senão  $f_{out} \leftarrow \max(f_{in}, F(addr) + p)$ 
7      Se  $f_{in} \leq F(addr) + p$  então
8         $u_{out} \leftarrow k$ 
9      Senão  $u_{out} \leftarrow u_{in}$ 
10    $y_{out} \leftarrow y_{in}$ 
11    $F(addr) \leftarrow f_{out}$ 

```

Análise: Se existirem n EP's, então pela função de ordem temporal o algoritmo tem complexidade de tempo $O(n + c)$ para resolver o problema. No caso geral se houver menos do que n EP's ($p < n$), então a arquitetura sistólica deve possuir o formato de um anel e uma fila Q que conecta o primeiro ao último EP conforme descrito na Figura 5.4.

A fila recebe os dados do EP _{p} e envia para o EP₁ quando necessário. Assim, o vetor sistólico tem de ser executado $\lceil n/p \rceil$ vezes para calcular $F(c, n)$, e a eficiência do algoritmo é $O(nc/p + n)$.

Figura 5.4: Um anel de tamanho p com fila.

5.3 Um algoritmo eficiente para resolução do problema da mochila ilimitada em hipercubos

Descrevemos aqui um algoritmo eficiente para a resolução de UKP em hipercubos de autoria de Goldman e Trystram [GT02]. Neste algoritmo a etapa de programação dinâmica para resolução do problema da mochila é representada como um grafo de dependência (assim como no algoritmo da seção 5.2) que possui a estrutura de uma malha (*mesh*) irregular. O algoritmo escalona de forma eficiente a computação nesta malha, de forma que é possível obter uma solução eficiente para UKP_c^N , em um hipercubo.

O algoritmo representa a comunicação paralela entre os processadores por um grafo acíclico onde os vértices representam as tarefas (ou instruções) e os arcos denotam as relações de precedência entre estas tarefas. O processo de paralelização consiste em atribuir a cada tarefa um tempo de execução e um dado processador. Este processo é também chamado de escalonamento do problema.

Note que esta abordagem é semelhante a utilizada na seção 5.2, porém, enquanto lá a solução foi desenhada para uma arquitetura sistólica e implementada em VLSI, o algoritmo aqui descrito é desenhado para a arquitetura de hipercubos em um modelo muito semelhante ao do comportamento de máquinas SIMD. Cada processador do hipercubo pode se comunicar através de pequenas mensagens com todos os processadores vizinhos em um único passo (ou tempo de execução unitário).

Definição 5.12 Uma malha $NM(n, c)$ com saltos $(w_i)_{i=0}^{n-1}$ é o grafo dirigido cujos vértices são o conjunto de pares (i, j) ($0 \leq i \leq n-1$ e $0 \leq j \leq c$) e dado um vértice (i, j) há uma arco de (i, j) para $(i+1, j)$ e outro arco de (i, j) para $(i, j+w_i)$ se $0 \leq i < n-1$ e $0 \leq j \leq c-w_i$.

Denotaremos $w_{\min} = \min_i w_i$ e $w_{\max} = \max_i w_i$.

A Figura 5.5 ilustra uma malha irregular $NM(4, 7)$ para objetos com peso 2, 3, 4 e 5.

Assim, o cálculo de $F(c, n)$ através de um algoritmo de programação dinâmica é representado pelo grafo de precedência $NM(n, c)$ com $(w_i)_{i=0}^{n-1}$ saltos e cada tarefa (i, j) deve executar o seguinte:

Figura 5.5: Malha irregular $NM(4, 7)$ com saltos $(w_i)_{i=0}^3 = (2, 3, 4, 5)$.

$$\text{Tarefa}(i, j) \left\{ \begin{array}{ll} \text{Se } i = 0 \text{ e } j < w_0 & \begin{array}{l} \text{Se } (n \geq 2) \text{ envie } 0 \text{ para } (i + 1, j), \\ \text{Se } (j \leq c - w_0) \text{ envie } p_0 \text{ para } (i, j + w_0). \end{array} \\ \text{Se } i = 0 \text{ e } j \geq w_0 & \begin{array}{l} \text{Receba } d \text{ de } (i, j - w_0), \\ \text{Se } (n \geq 2) \text{ envie } d \text{ para } (i + 1, j), \\ \text{Se } (j \leq c - w_0) \text{ envie } d + p_0 \text{ para } (i, j + w_0). \end{array} \\ \text{Se } i > 0 \text{ e } j < w_i & \begin{array}{l} \text{Receba } d \text{ de } (i - 1, j), \\ \text{Se } (i < n - 1) \text{ envie } d \text{ para } (i + 1, j), \\ \text{Se } (j \leq c - w_i) \text{ envie } d + p_i \text{ para } (i, j + w_i). \end{array} \\ \text{Se } i > 0 \text{ e } j \geq w_i & \begin{array}{l} \text{Receba } d_1 \text{ de } (i - 1, j) \text{ e } d_2 \text{ de } (i, j - w_i), \\ \text{Calcule } d = \max\{d_1, d_2\} \\ \text{Se } (i < n - 1) \text{ envie } d \text{ para } (i + 1, j), \\ \text{Se } (j \leq c - w_i) \text{ envie } d + p_i \text{ para } (i, j + w_i). \end{array} \end{array} \right.$$

5.3.1 Escalonamento

Para determinar um escalonamento, é necessário definir uma função de alocação e uma função de ordem temporal. A estratégia de alocação considera hipercubos como o produto cartesiano

$$H(z) = H(\lceil \lg w_{\min} \rceil) \otimes H(z').$$

Na descrição do algoritmo os $2^{z'}$ hipercubos $H(\lceil \lg w_{\min} \rceil)$ são identificados como $H_0, \dots, H_{2^{z'}-1}$ de tal forma que dois índices consecutivos denotam hipercubos adjacentes.

Seja P um dos $2^{\lceil \lg w_{\min} \rceil}$ caminhos Hamiltonianos de $H(\lceil \lg w_{\min} \rceil)$, então P_k denota o caminho associado a H_k ($0 \leq k < 2^{\lceil \lg w_{\min} \rceil}$) e cada um de seus vértices é identificado com $0, \dots, w_{\min} - 1$. Cada linha da malha irregular, denotada por L_i ($0 \leq i < n$) é executada no caminho $P_{i \bmod 2^{z'}}$. Por não alterar a complexidade

da análise e simplificar a apresentação, supõem-se que $2^{z'} = p/w_{\min}$, onde p é o número de processadores e $w_{\min}$ é objeto de menor peso.

Os autores definem

$$M = \max_i \left\lfloor \frac{c}{w_i} \right\rfloor \left\lceil \frac{w_i}{w_{\min}} \right\rceil, \quad (5.3)$$

ou seja, M é o maior tempo de execução dentre todas as linhas da malha, e

$$N = 2^{z'} = \frac{p}{w_{\min}}, \quad (5.4)$$

representa o número de caminhos hamiltonianos.

- **Função de alocação:** Tarefa(i, j) é executada pelo processador $P_{i \bmod N}^{(j \bmod w_i) \bmod w_{\min}}$.
- **Função de ordem temporal:** Tarefa(i, j) é executada no instante:

$$i(\lg w_{\min} + 2) + \left\lfloor \frac{j}{w_i} \right\rfloor \left\lceil \frac{w_i}{w_{\min}} \right\rceil + \left\lfloor \frac{j \bmod w_i}{w_{\min}} \right\rfloor + \left\lfloor \frac{i}{N} \right\rfloor t_w,$$

onde o valor de t_w é o tempo de espera, cujo valor depende do número de processadores, e é dado por:

$$t_w = \begin{cases} 0 & \text{se } p \geq M \left\lceil \frac{w_{\min}}{\lg w_{\min} + 2} \right\rceil, \\ M - (\lg w_{\min} + 2)N & \text{caso contrário.} \end{cases}$$

Análise: Vamos primeiro supor que exista um número ilimitado de processadores. Então, o tempo mínimo de execução do grafo de dependência do problema da mochila é maior ou igual ao tamanho do maior caminho no grafo. Na i -ésima linha de $NM(n, c)$ existem $\left\lfloor \frac{c}{w_i} \right\rfloor$ ou $\left\lfloor \frac{c}{w_i} \right\rfloor - 1$ saltos. Como a linha de saltos de menor comprimento (e consequentemente de maior número de saltos) é a linha do objeto $w_{\min}$ e existem ao todo n linhas, então o maior caminho no gráfico é no máximo igual a

$$n + \left\lfloor \frac{c}{w_{\min}} \right\rfloor - 1. \quad (5.5)$$

Supondo-se que o número de processadores p é maior ou igual a $M \left\lceil \frac{w_{\min}}{\lg w_{\min} + 2} \right\rceil$, então a última linha da malha irregular será executada (graças a função de ordem temporal) no instante $(n - 1)(\lg w_{\min} + 2)$ e sua execução levará $\left\lfloor \frac{c}{w_{n-1}} \right\rfloor \left\lceil \frac{w_{n-1}}{w_{\min}} \right\rceil$ unidades de tempo. Assim, neste caso, o tempo de execução máximo para o algoritmo é:

$$(n - 1)(\lg w_{\min} + 2) + M. \quad (5.6)$$

Supondo-se que o número de processadores p é menor que $M \left\lceil \frac{w_{\min}}{\lg w_{\min} + 2} \right\rceil$, então existe um tempo de espera entre a execução de cada linha da malha irregular, ou seja, $t_w \neq 0$. Neste caso, o tempo de execução é no máximo igual a

$$(n - 1)(\lg w_{\min} + 2) + M + \left\lceil \frac{n}{N} \right\rceil t_w. \quad (5.7)$$

Ou seja, o tempo é igual ao de (5.6) mais $\lfloor \frac{n}{N} \rfloor$ vezes o tempo de espera. Assim, possui máximo igual a

$$M \left(\frac{nw_{\min}}{p} + 1 \right). \quad (5.8)$$

Podemos passar para a análise da complexidade de tempo do algoritmo com e sem tempo de espera.

Se $t_w = 0$, como o valor de M é $\Theta(\frac{c}{w_{\min}})$, então a complexidade de tempo do algoritmo descrito, num hipercubo, é $\Theta(n \lg w_{\min} + \frac{c}{w_{\min}})$, se houverem ao menos $\frac{c}{\lg w_{\min}}$ processadores.

Se $t_w \neq 0$, ou seja, existe um tempo de espera entre a execução das linhas na malha irregular, então a complexidade de tempo do algoritmo descrito, num hipercubo, é $\Theta(\frac{nc}{p} + \frac{c}{w_{\min}})$, se houver um número de processadores entre $2^{\lceil \lg w_{\min} \rceil}$ e $\frac{c}{\lg w_{\min}}$.

Assim, para um número suficiente de processadores ($p > nw_{\min}$), a complexidade é dominada por $O(\frac{c}{w_{\min}})$. Senão, a complexidade de tempo é $O(\frac{nc}{p})$ que é a complexidade do algoritmo seqüencial (usando programação dinâmica) dividida pelo número de processadores.

ELIMINAÇÃO PARALELA DE TERMOS REDUNDANTES

Conforme visto no Capítulo 5, diversos estudos de algoritmos para a resolução do problema da mochila em paralelo são encontrados na literatura. Também encontram-se publicações sobre relações de dominância em algoritmos seqüenciais [Dud91, MT90b, Pis95]. No entanto, apesar de uma extensa busca bibliográfica, não tomamos ciência da existência de nenhum trabalho publicado sobre a implementação de algoritmos paralelos de eliminação de termos dominados.

Isso despertou nosso interesse de pesquisar maneiras de se paralelizar alguns algoritmos de eliminação de termos dominantes e descobrir se estes algoritmos poderiam ser tão úteis para algoritmos paralelos de resolução de problemas da mochila como o são para algoritmos seqüenciais.

Neste capítulo descrevemos nosso estudo do comportamento de versões paralelas dos algoritmos de dominância simples e múltipla de forma independente do algoritmo de resolução do problema da mochila. Somente estes algoritmos (versões seqüenciais e paralela) foram implementados. Para os testes utilizamos um aglomerado de PCs (Pentium III 733MHZ-256Mb) com sistema operacional Linux em uma rede Ethernet 100. Os experimentos e resultados descritos neste capítulo foram apresentados no WSCAD'02 [AG02].

6.1 Pesquisa

Pela originalidade de nossa abordagem, decidiu-se testar a viabilidade da paralelização da eliminação dos termos dominados para diversos tamanhos de instâncias do problema e para diversas porcentagens diferentes de termos dominados. Foram gerados dois conjuntos de arquivos de testes, um para dominância simples e outro para dominância múltipla. Em cada um dos conjuntos haviam 20 arquivos, onde variamos o número de objetos do problema (5.000, 10.000, 50.000 e 100.000) e a porcentagem de elementos dominados (10%, 30%, 50%, 70% e 90%). Para cada uma dessas 40 instâncias do problema da mochila foram medidos os tempos de execução seqüencial e paralelo, este último utilizando $p = 2, 4$ e 8 processadores, onde p é o número de processadores utilizados. Na eliminação paralela, em cada uma das máquinas foram instalados serviços implementados em Java com métodos remotos (RMI) de eliminação de termos dominados, utilizando os algoritmos seqüenciais de eliminação simples e múltipla.

6.1.1 Algoritmo utilizado

Seja $A = \{a_0, \dots, a_{n-1}\}$ o conjunto formado pelos objetos de uma instância do problema da mochila ilimitada (UKP). A eliminação é feita da seguinte maneira:

ELIMINAÇÃO PARALELA(A)

- 1 Para $i = 0$ até $p - 1$ faça
- 2 envie ao processador p_i os objetos com índices no intervalo $i \left\lfloor \frac{n}{p} \right\rfloor \dots (i + 1) \left\lfloor \frac{n}{p} \right\rfloor$
- 3 Para $i = 1$ até p faça em paralelo
- 4 execute método remoto de eliminação no processador p_i
- 5 devolva objetos não eliminados
- 6 execute o método local de eliminação para os termos recebidos.

Do ponto de vista do desenvolvimento da aplicação, a escolha por desenvolver utilizando-se Java RMI mostrou-se vantajosa no momento de transformar uma implementação de objetos locais em objetos remotos. Outro ponto de destaque é o fato de permanecer transparente para as demais classes do sistema a localização (remota ou local) dos objetos contendo os métodos de eliminação utilizados.

6.2 Resultados obtidos e análise

Os resultados obtidos com os testes realizados utilizando a estratégia descrita na seção anterior estão representados nos gráficos das Figuras 6.1, 6.2, 6.3 e 6.4. Em cada figura temos no eixo x o número de processadores utilizados, e no eixo y o tempo total do experimento. Como foi verificado que os algoritmos paralelos utilizados são bem estáveis, isto é não houve variações de tempo entre diversas execuções com o mesmo conjunto de dados, cada ponto nos gráficos corresponde ao tempo de uma execução.

Na Figura 6.1 podemos ver os resultados dos algoritmos sequencial e paralelo para eliminação usando dominância simples. Como nessa figura as instâncias de teste têm poucos objetos (até 10.000), mesmo quando existe a eliminação de 90% deles, não é possível obter ganho de tempo em relação ao algoritmo sequencial. Isto se deve ao fato de que o custo das comunicações iniciais (dividir os objetos entre as máquinas) e finais (concentrar os objetos após a primeira etapa de eliminação) é considerável. Também é interessante notar que para dois processadores este custo chega a ser maior do que o ganho proporcionado pela eliminação simultânea.

Na Figura 6.2 podemos ver que quando as instâncias são maiores e o número de objetos eliminados é maior do que 10% é possível obter um pequeno ganho de desempenho, mesmo para dois processadores.

Nas Figuras 6.3 e 6.4 temos os algoritmos com eliminação por dominância múltipla. Nestes casos o uso de até quatro processadores apresenta melhoras evidentes no tempo de processamento, mesmo para instâncias pequenas e um número reduzido de elementos dominados. Pode-se notar que para alguns exemplos, vide Figura 6.3, foi possível obter uma aceleração superlinear (ex: 10.000 objetos com 90% de dominados). Isso se deve ao fato de que em alguns testes o número de elementos eliminados na última fase é bem reduzido, isto é, quase todos os elementos são eliminados em paralelo em tempo proporcional a $(n/2)^2$.

Para instâncias maiores o algoritmo de eliminação por dominância múltipla apresenta resultados promissores sempre (Figura 6.4). Neste caso é possível constatar melhoras no tempo de execução mesmo quando se passa de 4 para 8 máquinas.

De uma maneira geral, percebe-se que em ambos os algoritmos, quanto maior a porcentagem de elementos dominados e o número de objetos, maior a aceleração obtida, principalmente no algoritmo que usa dominância múltipla. Entretanto a melhora no consumo de tempo, quando se passa de 4 para 8 máquinas, não chega a ser significativa.

Figura 6.1: Gráfico da eliminação usando dominância simples para 5.000 e 10.000 objetos.

Figura 6.2: Gráfico da eliminação usando dominância simples para 50.000 e 100.000 objetos.

Figura 6.3: Gráfico da eliminação usando dominância múltipla para 5.000 e 10.000 objetos.

Figura 6.4: Gráfico da eliminação usando dominância múltipla para 50.000 e 100.000 objetos.

CONCLUSÃO

Nos Capítulos 2 ao 5 descrevemos nosso estudo sobre o problema da mochila durante este mestrado. Procuramos descrever os principais algoritmos de resolução sequencial e paralela, além das técnicas de redução do espaço de busca.

O trabalho realizado no Capítulo 6 mostrou que é interessante utilizar algoritmos mais sofisticados para a eliminação de objetos no caso de grandes instâncias do problema da mochila.

Os experimentos também mostram que algoritmos de eliminação baseados na relação de dominância simples, além de reduzirem menos o espaço de busca em um UKP do que algoritmos que usam dominância múltipla, também apresentam um baixo ganho quando paralelizados, sendo que chegam até mesmo a ter um desempenho pior em relação a implementação sequencial quando, por exemplo, se utilizam 2 processadores.

Observamos que o ganho na paralelização de algoritmos de dominância (especialmente dominância múltipla) além de ser diretamente proporcional ao tamanho da instância cresce com o aumento da porcentagem de objetos dominados dentro dessa instância. Esse é um resultado muito interessante, já que, quanto maiores os benefícios do uso de eliminação de termos dominantes para algoritmos que resolvem UKP sequencialmente, mais vantajoso se torna empregar a paralelização desta etapa do problema da forma que fizemos neste trabalho. Assim, esperamos ter contribuído para demonstrar que o estudo de algoritmos paralelo de técnicas de eliminação de termos dominados é vantajosa para grandes instâncias do problema da mochila ilimitada, já que esta etapa de pré-processamento é responsável pela maior parte do tempo gasto na resolução do algoritmo [Dud91].

Também é interessante notar que a linguagem Java apresenta recursos suficientes para implementações paralelas que podem apresentar ganhos de desempenho interessantes [PL01].

7.1 Oportunidades para pesquisas futuras

Durante nosso trabalho de pesquisa surgiram diversos questionamentos e idéias que consideramos interessantes como tópicos de pesquisas futuras. Infelizmente problemas para utilizar aglomerados de máquinas no IME e falta de tempo hábil nos impossibilitaram de pesquisar mais profundamente estes assuntos:

- **Heurísticas para distribuição dos objetos:** Como o objeto com maior relação p_i/w_i (vamos chamar este objeto de b) é o que possui maior probabilidade de eliminar outros objetos, uma heurística interessante para melhorar a performance dos algoritmos propostos no Capítulo 6 é distribuir uma cópia de b em cada um dos subconjuntos para eliminação.

- **Paralelização da relação de dominância coletiva:** A relação de dominância coletiva pode ser entendida como a realização de diversos problemas da mochila. A paralelização da implementação desta relação de dominância pode ser beneficiar das técnicas apresentadas no Capítulo 5?
- **Paralelização do algoritmo da Seção 4.4:** O algoritmo EDUK para resolução seqüencial do problema da mochila, faz um uso bastante interessante de eliminação de objetos dominados durante a execução da fase que calcula a recorrência da função objetivo. Paralelizar o algoritmo EDUK seria extremamente interessante.

REFERÊNCIAS BIBLIOGRÁFICAS

- [AF94a] V. Aleksandrov e S. Fidanova, *Non-uniform recurrence equations on 2d regular arrays*, Advance in Numerical Methods and Applications, in Proc. of NMA94, August 1994, pp. 217–225. Citado na(s) página(s) 19
- [AF94b] V. Aleksandrov e S. Fidanova, *On the expected execution time for a class of non uniform recurrence equations mapped onto 1d array*, Parallel Algorithms and Applications 1 (1994), 303 – 314. Citado na(s) página(s) 19
- [AG91] R. Andonov e F. Gruau, *A 2D modular toroidal systolic array for the Knapsack problem*, ASAP’91, 1991, pp. 458 – 472. Citado na(s) página(s) 19
- [AG02] F. Arruda e A. Goldman, *Eliminação paralela de termos dominantes no problema da mochila*, WS-CAD’02, 2002, pp. 89 – 94. Citado na(s) página(s) 75
- [AQ92] R. Andonov e P. Quinton, *Efficient linear systolic array for Knapsack problem*, CONPAR’92, LNCS, no. 634, 1992, pp. 247 – 258. Citado na(s) página(s) 19
- [AR97] R. Andonov e S. Rajopadhye, *Knapsack on VLSI: from algorithm to optimal circuits*, IEEE Transactions on Parallel and Distributed Systems 8 (1997), no. 6, 545 – 562. Citado na(s) página(s) 19, 65, 67, 68
- [Bel57] R. Bellman, *Dynamic programming*, Princeton University Press, 1957. Citado na(s) página(s) 47, 58
- [BZ80] E. Balas e E. Zemel, *An algorithm for large zero-one knapsack problems*, Operational Research (1980), no. 28, 1130–1154. Citado na(s) página(s) 53
- [CFH89] M. Cosnard, A. Ferreira, e H. Herbelin, *The two list algorithm for the Knapsack problem on a FPS T20*, Parallel Computing 3 (1988/89), no. 9, 385 – 388. Citado na(s) página(s) 65
- [CJ92] G-H. Chen e J-H. Jang, *An improved parallel algorithm for 0/1 knapsack problem*, Parallel Computing 18 (1992), no. 7, 811–821. Citado na(s) página(s) 19
- [Dud91] K. Dudzinski, *A note on dominance relation in unbounded knapsack problem*, Operations Research Letters 10 (1991), 417–419. Citado na(s) página(s) 19, 38, 75, 79
- [GG63] P.C. Gilmore e R.E Gomory, *A linear programming approach to the cutting stock problem*, Operations Research 11 (1963), 863–888. Citado na(s) página(s) 36
- [GG66] P.C. Gilmore e R.E Gomory, *The theory and computation of knapsack fuctions*, Operations Research 14 (1966), 1045–1074. Citado na(s) página(s) 33
- [GHCJ90] M-S. Chern G-H. Chen e J-H. Jang, *Pipeline architectures for dynamic programming algorithms*, Parallel Computing 13 (1990), no. 1, 111 – 117. Citado na(s) página(s) 19

- [GJ79] M.R. Garey e D.S. Johnson, *Computers and intractability: A guide to the theory of np-completeness*, W.H. Freeman and Company, San Francisco, 1979. Citado na(s) página(s) 19, 25, 30
- [GN72] R. Garfinkel e G. Nemhauser, *Integer programming*, John Wiley and Sons, 1972. Citado na(s) página(s) 47, 49, 57
- [GT02] A. Goldman e D. Trystram, *An efficient parallel algorithm for solving the Knapsack problem on hypercube*, Journal of Parallel and Distributed Computing (2002), To appear. Citado na(s) página(s) 19, 65, 66, 71
- [GW93] T. Gerasch e P. Wang, *A survey of parallel algorithm for one-dimensional integer Knapsack problems*, Infor 32 (1993), no. 3, 163–186. Citado na(s) página(s) 65
- [HS74] E. Horowitz e S. Sahni, *Computing partitions with applications to the Knapsack problem*, Journal of ACM 21 (1974), 277 – 292. Citado na(s) página(s) 65
- [Iba87] T. Ibaraki, *Enumerative approaches to combinatorial optimization*, Annals of Operations Research 10-11 (1987), 376–388. Citado na(s) página(s) 30
- [Kar84] E. Karnin, *A parallel algorithm for the Knapsack problem*, IEEE Transactions on Computers 33 (1984), no. 5, 404 – 408. Citado na(s) página(s) 65
- [KT88] G.A.P Kindervater e H.W.J.M Trienekens, *Experiments with parallel algorithms for combinatorial problems*, European Journal of Operational Research (1988), no. 33, 65–81. Citado na(s) página(s) 65
- [Lag] Michail G. Lagoudakis, *The 0-1 knapsack problem: An introductory survey*. Citado na(s) página(s) 22
- [LS91] J. Lin e J. Storer, *Processor efficient hypercube algorithm for the Knapsack problem*, Journal of Parallel and Distributed Computing 13 (1991), no. 3, 332 – 337. Citado na(s) página(s) 19, 65, 66
- [LSS88a] J. Lee, E. Shragowitz, e S. Sahni, *A hypercube algorithm for the 0/1 Knapsack problem*, Journal of Parallel and Distributed Computing 5 (1988), no. 4, 438–456. Citado na(s) página(s) 19
- [LSS88b] J. Lee, E. Shragowitz, e S. Sahni, *A hypercube algorithm for the 0/1 Knapsack problem*, Journal of Parallel and Distributed Computing 5 (1988), no. 4, 438–456. Citado na(s) página(s) 65
- [MAR⁺00] D.G. Morales, F. Almeida, C. Rodríguez, J. L. Roda, I. Coloma, e A. Delgado, *Parallel dynamic programming and automata theory*, Parallel Computing 26 (2000), 113 – 134. Citado na(s) página(s) 19
- [MT77a] S. Martello e P. Toth, *Branch and bound algorithms for the solution of general unidimensional knapsack problems*, Advances in Operations Research, North-Holland (1977). Citado na(s) página(s) 52
- [MT77b] S. Martello e P. Toth, *Computational experiences with large-size unidimensional knapsack problems*, Presented at the TIMS/ORSA Joint National Meeting, San Francisco (1977). Citado na(s) página(s) 52
- [MT90a] S. Martello e P. Toth, *An exact algorithm for large unbounded knapsack problems*, Operations Research Letters (1990), no. 9, 15–20. Citado na(s) página(s) 38, 47, 49, 57, 63
- [MT90b] S. Martello e P. Toth, *Knapsack problems: Algorithms and computer implementations*, John Wiley and Sons, New York, London, Sydney, 1990. Citado na(s) página(s) 19, 21, 22, 23, 25, 75
- [Pis95] D. Pisinger, *Algorithms for knapsack problems*, PhD thesis, University of Copenhagen, Dept. of Computer Science (1995). Citado na(s) página(s) 19, 21, 22, 38, 40, 75
- [PL01] C.M. Pancake e C. Lengauer, *Special issue on high performance java*, Communications of ACM 44 (2001), no. 10, 99–101. Citado na(s) página(s) 79
- [PYA02] V. Poirriez, N. Yanev, e R. Andonov, *Towards reduction of the class of intractable unbounded knapsack problem*, Rapport de Recherche LAMIH ROI (2002). Citado na(s) página(s) 43, 61

- [RAR00] V. Poirriez R. Andonov e S. Rajopadhye, *Unbounded Knapsack problem: Dynamic programming revisited*, European Journal of Operational Research **123** (2000), 394 – 407. Citado na(s) página(s) 19, 43, 59, 61, 63
- [RF90] S. V. Rajopadhye e R. M. Fujimoto, *Synthesizing systolic arrays from recurrence equations.*, Parallel Computing **14** (1990), 163–189. Citado na(s) página(s) 68
- [Sal75] H.M. Salkin, *Integer programming*, Addison-Wesley, Reading, Mass, 1975. Citado na(s) página(s) 19, 22
- [Ten90] S. Teng, *Adaptive parallel algorithms for integral Knapsack problems*, Journal of Parallel and Distributed Computing **8** (1990), 400–406. Citado na(s) página(s) 65
- [ZB97] N. Zhu e K. Broughan, *On dominated terms in the general Knapsack problem*, Operations Research Letters **21** (1997), 31–37. Citado na(s) página(s) 19

ÍNDICE REMISSIVO

A

Algoritmo

- Algoritmo básico para UKP, 47
- Algoritmo de Pisinger, 38
- EDUK, 59
- Garfinkel e Nemhauser, 57
- MTU2, 49

B

- Branch and Bound, 25

C

- Conclusão, 79

I

- Instâncias do problema da mochila, 30
- Instâncias do problema da mochila
 - fortemente correlatas, 31
 - fracamente correlatas, 31
 - não-correlatas, 30
 - subset-sum, 32

P

- Periodicidade, 33
- Periodicidade
 - Equação, 33
- Problema da mochila
 - 0-1KP, 21
 - Algoritmos de aproximação, 30
 - binária, 21
 - BKP
 - Hipercubos, 66
 - Change Making Knapsack Problem, 24
 - enunciado, 21
 - no contexto do paralelismo, 65
 - no contexto do paralelismo
 - 0-1KP, 66
 - mochila binária, 66
 - mochila ilimitada, 67, 71
 - UKP, 67, 71
 - Problema da Mochila Binária Múltipla, 24
 - Problema da Mochila Ilimitada, 23
 - Problema da Mochila Limitada, 23
 - Soma de Subconjunto, 24
 - UKP, 23
 - UKP
 - Hipercubos, 71
 - solução para arquitetura sistólica, 67
 - Programação Dinâmica, 27

R

- Redução do espaço de busca em KP, 33
- Redução do espaço de busca em KP
 - Periodicidade, 33
 - Relações de dominância, 35
 - Relações de dominância
 - Dominância múltipla, 36
 - Dominância simples, 35
- Relações de dominância, 35
- Relações de dominância
 - Dominância coletiva, 41
 - Dominância múltipla, 36
 - Dominância múltipla
 - Algoritmo de Pisinger, 38
 - Dominância por threshold, 41
 - Dominância simples, 35
 - Redução paralela, 75

Redução paralela

Algoritmo, 75

Análise dos resultados obtidos, 76

Pequisa, 75

S

Subset Sum, 24

U

UKP

Periodicidade, 33

Soluções seqüenciais, 47

Soluções seqüenciais

Algoritmo básico, 47

EDUK, 59

Garfinkel e Nemhauser, 57

MTU2, 49