UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FÉLIX CARVALHO RODRIGUES

Programação Dinâmica Eficiente com Algoritmos *Cache-Oblivious*

Trabalho de Graduação

Prof. Dr. Marcus Ritt
Orientador

Prof[a]. Dr[a]. Luciana Salete Buriol
Co-orientador

Porto Alegre, Dezembro de 2008

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice Reitor: Prof. Pedro Cezar Dutra da Fonseca
Pró-Reitor de Graduação: Prof Carlos Alexandre Netto
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador da Comissão de Graduação da CIC: Prof. Raul Fernando Weber
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“We apologise for the inconvenience”

— God’s Final Message to His Creation
(The Hitchhiker’s Guide to the Galaxy)

AGRADECIMENTOS

Agradeço principalmente ao meu pai, Osvaldir Rodrigues, por sempre acreditar em mim e me apoiar em todos os meus objetivos, não interessando o grau de dificuldade deles, assim como compreender o quanto ele significa para mim, mesmo eu não demonstrando isso.

Agradeço muito a minha mãe, Helena Carvalho, e minha irmã, Priscila Carvalho, pelo imenso carinho recebido e pela compreensão de que apesar de eu passar muito pouco tempo fisicamente com elas, eu as amo muito.

Agradeço também:

Ao meu colega e amigo Daniel Osmari, pelo modelo de determinação, inspiração e por me mostrar que eu necessito me esforçar para ser o melhor sempre.

Aos meus amigos Kao Félix, Daniel Beck, Márcio Zacarias e Letícia Nunes, pela diversão e grande apoio nas horas mais difíceis, além das constantes discussões estimulantes.

E por fim, aos meus orientadores Prof. Dr. Marcus Ritt e Profª. Drª. Luciana Salete Buriol, pela excelente orientação, pela quantidade de informação adquirida e pelo exemplo a ser seguido.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 9

LISTA DE FIGURAS 10

RESUMO 11

ABSTRACT 12

1 INTRODUÇÃO 13
1.1 Hierarquias de Memória 14
1.2 Outros Modelos 15
1.2.1 RAM 15
1.2.2 Memória Externa 16

2 ALGORITMOS *CACHE-OBLIVIOUS* 17
2.1 Modelo de cache Ideal 17
2.1.1 Política de Substituição 18
2.1.2 Níveis de Memória 18
2.1.3 Associatividade e Reposição Automática 19
2.2 Algoritmos Estudados 19
2.2.1 Maior Subseqüência Comum 20
2.2.2 Multiplicação de Matrizes 21
2.2.3 Gap Problem 22

3 *BIN PACKING* 24
3.1 Soluções para o Empacotamento Unidimensional 24
3.1.1 Heurísticas e Meta-heurísticas 24
3.1.2 Algoritmos Exatos 24

4 PROBLEMA DA MOCHILA 26
4.1 Problema 26
4.2 Variações 27
4.2.1 Problema Fracionário da Mochila 27
4.2.2 Problema Booleano da Mochila 27
4.2.3 Problema da Mochila Limitado 27
4.3 Problema da Mochila Não Limitado 28
4.3.1 Dominâncias 28

5 SOLUÇÃO *CACHE-OBLIVIOUS* PARA O PROBLEMA DA MOCHILA ILIMITADO 30
5.1 Solução Tradicional 30
5.2 Solução *Cache-Oblivious* 32
5.3 Dominâncias 33
5.4 Complexidade das Soluções 36
5.5 Resultados Computacionais 39
5.5.1 Instâncias 39
5.5.2 Comparações 40

6 CONSIDERAÇÕES FINAIS 44
6.1 Trabalhos Futuros 45

REFERÊNCIAS 46

LISTA DE ABREVIATURAS E SIGLAS

RAM	Random Access Machine
CPU	Central Processing Unit
LRU	Least Recently Used
FIFO	First In, First Out

LISTA DE FIGURAS

Figura 1.1: A memória no modelo RAM 15
Figura 1.2: O modelo de Memória Externa 16

Figura 2.1: O Modelo de cache Ideal 17

Figura 4.1: Todos os casos em que i é dominado: (a) dominância simples, (b) dominância múltipla, (c) dominância coletiva e (d) dominância limiar 29

Figura 5.1: Acesso a memória no algoritmo 5.3 37
Figura 5.2: Acesso a memória no algoritmo 5.1 38
Figura 5.3: Item i como um ponto no plano. Qualquer ponto nas áreas destacadas ou dominam o item i ou são dominados por ele. 39
Figura 5.4: Número de *cache misses* para diferentes configurações de cache com uma entrada de $W = 250.000$ e $n = 10.000$ para os algoritmos 5.1 (T) e 5.3 (CO) 41
Figura 5.5: Número de *cache misses* para as versões que utilizam Dominância para diferentes configurações de cache com uma entrada de $W = 10.000.000$ e $n = 10.000$ para os algoritmos 5.1 (T) e 5.3 (CO) 41
Figura 5.6: Tempo de execução dos algoritmos 5.1 (T) e 5.3 (C), em escala logarítmica. 42
Figura 5.7: Tempo de execução dos algoritmos 5.4 (domT) e 5.5 (domCO), em escala logarítmica. 43

RESUMO

A memória nos computadores modernos geralmente está organizada em uma hierarquia complexa. Dessa forma, torna-se importante projetar algoritmos que utilizem a cache de forma eficiente. Além disso, as configurações da memória e da cache tem grande variação de computador para computador. Assim, é necessário também que os algoritmos desenvolvidos dependam o mínimo possível de informações da máquina para usar a cache eficientemente.

No modelo de cache ideal, existem dois níveis de memória. Uma tem acesso aleatório e é infinita (memória principal), porém tem um custo associado ao seu acesso, enquanto que a outra é de acesso rápido, porém com um tamanho finito.

Um algoritmo é dito *cache-oblivious* se ele usa a cache de forma eficiente mesmo sem ter nenhuma informação sobre a cache. Para medirmos a complexidade desse tipo de algoritmo, não basta utilizarmos somente a complexidade do número de instruções executadas. Dessa maneira, utilizamos também a complexidade de *cache-misses*, que pode ser medida utilizando o modelo de cache ideal, para medir o quão eficientemente um algoritmo acessa a cache.

Existem muitos problemas ainda não analisados quanto a sua eficiência de cache. Um desses problemas é o Problema da Mochila. Nele, dado uma mochila de um certo tamanho e um conjunto de itens com um peso e um lucro associado, pede-se que se encontre a combinação de itens que caibam na mochila que resultem no maior lucro acumulado.

Esse problema é de extrema importância para várias áreas da computação, sendo subproblema de muitos problemas. Um desses problemas é o *Bin-Packing*, de inúmeras aplicações práticas.

Apresentamos, nesse trabalho, um algoritmo *cache-oblivious* para o Problema da Mochila Ilimitado. Além disso, apresentamos também uma pesquisa e análise de problemas em que já existem algoritmos *cache-oblivious* desenvolvidos.

Palavras-chave: Programação dinâmica, algoritmos *cache-oblivious*, problema da mochila ilimitado, problema bin-packing.

Efficient Cache-Oblivious Dynamic Programming Algorithms

ABSTRACT

Memory in modern computers is usually organized in a complex hierarchy. Thus, it is important to design algorithms that use the cache efficiently. Moreover, the configuration of memory and cache varies greatly from a computer to another. Therefore, it is necessary that the algorithms developed depend on the minimum information from the machine to use the cache in a efficient way.

In the ideal-cache model, there are two levels of memory. The first one has random access and is infinite (main memory), but has a cost associated with its access, while the other can be quickly accessed, but has a finite size.

An algorithm is said to be cache-oblivious if it uses the cache efficiently even without having any information about the cache. To measure the complexity of such a algorithm, it is not enough to use only the work complexity. Thus, we also use the cache-miss complexity, which can be measured using the ideal-cache model, measuring how efficiently an algorithm accesses the cache.

Many problems have not yet been analyzed for their cache efficiency. An example of such problems is the knapsack problem. Given a bag of a certain size and a number of items with a weight and profit associated to it, discover the combination of items that fit in the bag such that the profit is maximized.

The solution to this problem is of utmost importance to several areas of computer science, and subproblem of many other problems. An example of such problem is the Bin-Packing, which has many practical applications.

In this work we present a cache-oblivious algorithm to the Unlimited Knapsack Problem. Furthermore, we also present a research and analysis of problems in which a cache-oblivious algorithms has already been developed.

Keywords: dynamic programming, cache-oblivious algorithms, unbounded knapsack problem, bin-packing problem.

1 INTRODUÇÃO

Tradicionalmente o projeto de algoritmos tem como foco o desenvolvimento de algoritmos que levam em conta apenas o número de instruções executadas, sendo considerado o acesso à memória constante. Esse tipo de complexidade pode ser medida pelo modelo RAM (*Random Access Machine*), onde existe somente um processador e uma memória com acesso aleatório constante.

Com o desenvolvimento de memórias mais rápidas, porém mais caras, e a disparidade entre a velocidade do processador e das memórias mais baratas, tornou-se comum o uso de uma hierarquia de memórias nos computadores. Nessa hierarquia, somente a memória mais rápida (e também a menor) tem contato direto com o CPU. A segunda memória mais rápida tem contato com essa cache e assim por diante até a memória mais lenta e de maior capacidade.

Sendo os computadores atuais projetados com hierarquias de memória com múltiplos níveis, torna-se importante também levar em conta no projeto de algoritmos, além do número de instruções executadas, o custo para acessar os diferentes níveis de memória existentes.

O modelo de cache ideal, proposto por Prokop em (PROKOP, 1999), lida justamente com esse tipo de custo. Nele, além da complexidade do número de instruções, é possível calcular a complexidade de *cache-misses* de um algoritmo.

Há duas maneiras de se desenvolver um algoritmo que utiliza a cache eficientemente. Um algoritmo é dito *cache-aware* se ele tem algum conhecimento de como é a cache, tal como o tamanho dela ou quantas palavras existem em uma linha de cache. Pelo outro lado, um algoritmo é dito *cache-oblivious* se ele usa a cache de forma eficiente mesmo sem ter nenhuma informação sobre a cache.

Devido à grande diversidade de configurações de computadores existentes, tanto no número de níveis de memória, quanto no tamanho dessas memórias, é importante que os algoritmos desenvolvidos para usar a cache eficientemente sejam *cache-oblivious*.

Programação Dinâmica é um método para problemas que tem por característica uma subestrutura ótima e subproblemas que se sobrepõem. Um problema que tem subestrutura ótima é aquele em que as soluções ótimas para os seus subproblemas podem ser utilizadas para achar a solução ótima para o problema maior. Nesse tipo de técnica, utiliza-se a memória para guardar as soluções para os subproblemas, de forma a não ser necessário recalcular nenhuma informação.

Esse tipo de método depende pesadamente da memória, com muitos acessos sendo feitos a ela a cada passo do cálculo da solução. Dessa forma, esse tipo de algoritmo é ideal para se utilizar técnicas a fim de diminuir o número de *cache-misses*. Existem um número considerável de algoritmos *cache-oblivious* já desenvolvidos para problemas que utilizam programação dinâmica, porém a eficiência da cache na maioria dos algoritmos

conhecidos não foi estudado.

Um exemplo de problema em que ainda não se conhece um algoritmo *cache-oblivious* é o Problema da Mochila. Nele, deve-se preencher uma mochila com uma certa capacidade, tendo um *array* indicando os tipos de itens existentes, com seu peso e seu lucro, buscando obter o lucro máximo. Várias variantes são conhecidas, tais como o com itens ilimitados (Problema da Mochila Ilimitado) ou com somente um item de cada tipo (Problema da Mochila Binário), porém, não temos conhecimento de nenhum algoritmo *cache-oblivious* desenvolvido até hoje para sequer uma dessas variantes.

O Problema da Mochila Ilimitado pode ser resolvido como um subproblema em algoritmos que resolvem outros problemas mais complexos, em particular do problema *Bin-Packing* (Problema de Empacotamento Unidimensional). Para o solução do *bin-packing*, o melhor algoritmo conhecido é via o método de otimização chamado geração de colunas. Neste algoritmo, é calculado, por diversas vezes, o problema da mochila ilimitado, de forma que uma melhora no tempo de execução desse problema tem conseqüências na resolução do *bin-packing*.

Neste trabalho, procuramos desenvolver um algoritmo *cache-oblivious* para o Problema da Mochila Ilimitado, além de pesquisar e analisar problemas em que já existem algoritmos *cache-oblivious* desenvolvidos.

Esse trabalho está estruturado da seguinte maneira:

- O capítulo 1 aborda os conhecimentos mínimos necessários para o bom entendimento do trabalho. Além disso, fornece uma breve explicação de outros modelos referenciados no restante do trabalho.

- O capítulo 2 apresenta o modelo de cache ideal utilizado para o cálculo de complexidade de *cache-misses* e justifica o seu uso, assim como apresenta os algoritmos estudados e seus resultados práticos.

- No capítulo 3, o problema *Bin-Packing* é introduzido, bem como sua relação com o Problema da Mochila Ilimitado e a importância do desenvolvimento de um algoritmo *cache-oblivious* para ele.

- O capítulo 4 aborda o Problema da Mochila, explicando em detalhes as suas principais variantes e apresentando algumas características especiais da variante com quantidade de itens ilimitada.

- No capítulo 5, é apresentada a solução *cache-oblivious* para o Problema da Mochila Ilimitado e a versão tradicional, assim como a prova da complexidade de ambos. Ainda, é abordado adaptações nos algoritmos para a utilização de dominâncias para acelerar o cálculo da solução ótima.

- Por fim, no capítulo 6 são apresentadas as considerações finais e planos para trabalhos futuros.

1.1 Hierarquias de Memória

A hierarquia de memória é a forma como as diferentes memórias em um computador estão organizadas. Normalmente em uma hierarquia de memória os diferentes tipos de memórias são dispostas de forma que o processador está mais próximo da memória mais rápida, e mais longe da memória mais lenta.

A idéia de múltiplos níveis de memória está fortemente baseado na constatação de que um programa que acessou uma posição na memória provavelmente irá utilizá-lo novamente dentro de um curto espaço de tempo (localidade temporal) e que um programa que acessou uma posição de memória provavelmente utilizará as posições próximas a ela em seguida (localidade temporal).

A cache é repartida em linhas, que podem conter um número fixo de palavras. Quando um dado é acessado, se ele não está no nível mais perto da CPU, um *cache-miss* acontece e essa palavra é buscada de um nível superior. Até essa operação terminar, o programa sob execução não pode prosseguir, o que tem um impacto negativo no seu desempenho. Ao alocar esse dado, a cache trás junto as palavras próximas a esse dado, com o tamanho de uma linha dela. Assim, a localidade espacial é aproveitada. Já para se aproveitar da localidade temporal, a cache procura manter um dado o maior tempo possível em sua memória, utilizando diferentes estratégias para a substituição quando a cache está cheia.

O número de palavras total da cache geralmente é maior ou igual ao quadrado do número de palavras em uma linha da cache, ou seja, o número de linhas em uma cache é de no mínimo o número de palavras por linha da cache. Esse tipo de cache é referenciado como cache alta (*tall-cache*).

1.2 Outros Modelos

Além do modelo de cache ideal, existem outros modelos utilizados amplamente no desenvolvimento de algoritmos. A seguir apresentaremos alguns desses modelos para um melhor entendimento do trabalho.

1.2.1 RAM

Neste modelo, instruções são executadas com uma memória tão larga quanto necessária, de acesso aleatório constante. Com ele, é possível calcular tanto a complexidade de número de instruções quanto a complexidade espacial.

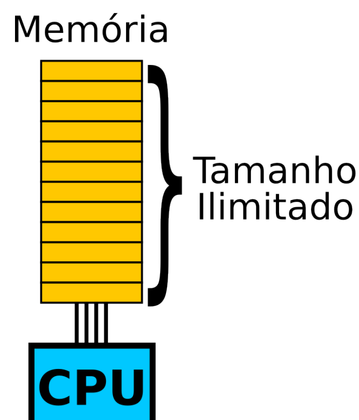

Figura 1.1: A memória no modelo RAM

Todavia, não é possível calcular a complexidade de entrada e saída da memória (ou seja, a complexidade de *cache-misses*). Logo, outros modelos devem ser utilizados para tal tarefa.

1.2.2 Memória Externa

Este modelo, também conhecido como modelo de acesso ao disco, introduz dois níveis à memória:

- Uma memória (cache) na qual o CPU acessa diretamente, que é barata de acessar, porém pequena em tamanho.
- Uma memória (disco) na qual o CPU não consegue acessar diretamente, que é custosa para acessar, mas contém uma capacidade tão grande quanto o necessário.

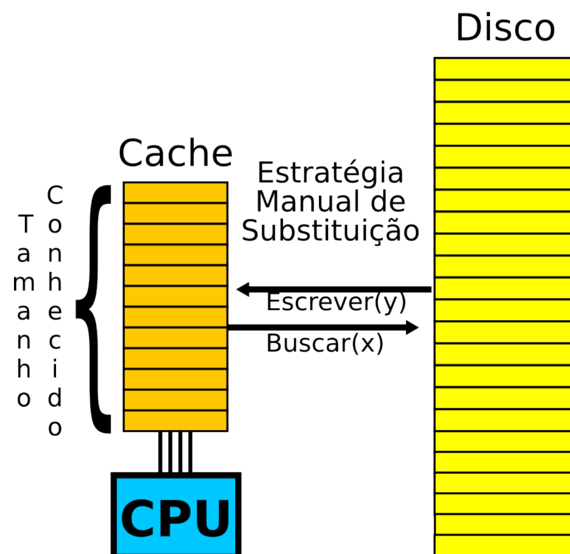

Figura 1.2: O modelo de Memória Externa

Ao contrário do modelo de cache ideal, porém, o acesso à memória é feito explicitamente, sendo necessário um conhecimento da cache, assim como definir explicitamente cada leitura ou escrita do algoritmo.

Como discos são dispositivos consideravelmente diferentes da memória, modelos de memória externa geralmente são mais complexos. Por exemplo o *parallel disk model* (PDM) de Vitter (VITTER, 2007), considera o número de discos em paralelo além do tamanho de um bloco transferido. Modelos mais sofisticados diferenciam ainda entre acesso aleatório ou seqüencial, e modelam a sobreposição (*overlap*) entre entrada/saída e computação (VITTER, 2007).

No modelo de cache ideal, como veremos a seguir, essa complexidade extra não existe, de forma que o algoritmo não necessita conhecer absolutamente nada sobre a cache, simplificando em grande parte o trabalho do programador.

2 ALGORITMOS *CACHE-OBLIVIOUS*

2.1 Modelo de cache Ideal

O modelo de cache ideal é um modelo criado para facilitar a análise de complexidade de entrada e saída. O modelo conta com dois níveis de memória, sendo o primeiro nível de memória uma memória (cache) finita com Z palavras, com um tamanho de linha de L palavras. O segundo nível de memória contém a memória principal, infinita e, assim como no modelo RAM, com acesso aleatório constante.

O processador pode acessar somente o primeiro nível de memória (cache). Se o processador necessita de algo fora dessa cache, um *cache-miss* acontece e L palavras são buscadas do segundo nível de memória para a cache. Logo, a memória principal contém uma penalidade extra para ser acessada. Essa cache é totalmente associativa, ou seja, cada linha dela pode armazenar qualquer linha da memória principal.

Utilizando esse modelo, é possível calcular a complexidade de *cache-misses* de algoritmos sem a necessidade de conhecer ou tratar com a memória, ao contrário de outros modelos conhecidos, tais como os descritos no capítulo 1.

Nesse modelo, a cache não necessariamente precisa ser alta, porém alguns algoritmos assumem uma cache alta para o cálculo de sua complexidade pessimista de *cache-miss*. Uma cache alta, ou *tall-cache* pode ser definida como uma cache em que a expressão $\frac{Z}{L} \geq L$ é verdadeira, ou seja, que $Z = \Omega(L^2)$.

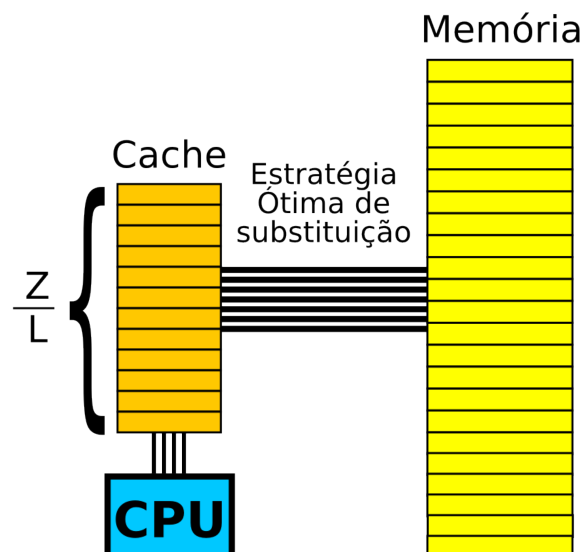

Figura 2.1: O Modelo de cache Ideal

O modelo assume algumas propriedades explicadas a seguir:

2.1.1 Política de Substituição

Quando ocorre um *cache-miss* e a cache está cheia, é necessário escolher uma linha da cache para ser substituída. No modelo de cache ideal, a linha escolhida para a substituição é aquela que mais demoraria para ser acessada no futuro. Essa política de substituição é chamada de estratégia de substituição ótima, e explora a localidade temporal dos dados o melhor possível.

Essa estratégia, obviamente, não pode ser utilizada nos computadores reais. Existem diversos algoritmos de substituição utilizados atualmente, tais como LRU, e FIFO (HENNESSY; PATTERSON, 1996). Na FIFO a linha a ser substituída é sempre o que esteve na cache por mais tempo. Já no LRU, a linha a ser substituída é a que foi menos utilizada recentemente.

Felizmente, uma cache com substituição ótima pode ser substituída por uma com substituição LRU sem perda da otimalidade assintótica, desde que seja permitida a mudança dos limites de Z da cache.

Em (FRIGO et al., 1999), foi provado que um algoritmo que causa $Q'(n; Z, L)$ *cache-misses* em um problema de tamanho n usando uma cache (Z, L) ótima causará $Q(n; Z, L) \leq 2Q'(n; \frac{Z}{2}, L)$ *cache-misses* em uma cache (Z, L) que use uma política de substituição LRU. Isso significa que, se permitirmos usar uma cache com o dobro de tamanho, a política LRU não gera mais que o dobro de *cache-misses* em relação à política de substituição ótima.

Por consequência, se um algoritmo satisfaz a condição de regularidade (FRIGO et al., 1999), o número de *cache-misses* é assintoticamente igual nas duas políticas, isto é

$$Q_{LRU}(n; Z, L) = \Theta(Q_{OPT}(n; Z, L)) \tag{2.1}$$

A condição de regularidade é satisfeita se o tamanho do crescimento assintótico da complexidade de cache não é alterado, caso o tamanho da cache seja duas vezes maior:

$$Q(n; Z, L) = O(Q(n; 2Z, L)) \tag{2.2}$$

Essa condição é razoável se o algoritmo não se aproveita mais do que linearmente do tamanho da cache e isso é satisfeito por todos algoritmos estudados nesse trabalho. Logo, como consequência da afirmação 2.1, todas as análises podem ser feitas tanto no modelo ótimo quanto no modelo LRU. É importante ressaltar, entretanto, que na prática a estratégia LRU é geralmente aproximada, o que possivelmente aumenta a diferença entre a cache ideal e as caches reais.

2.1.2 Níveis de Memória

Embora o modelo utilize apenas dois níveis de memória, ele prova resultados sobre qualquer hierarquia de níveis de memória. Para tanto, a memória deve possuir a propriedade da inclusão, que pode ser definida pelas seguintes características:

- Um dado pode estar em uma memória de nível i se e somente se ela está presente em uma memória de nível $i + 1$ (onde o nível 1 é o nível mais próximo do processador).

- Se dois elementos pertencem à mesma linha em um nível i, eles pertencem à mesma linha em um nível $i + 1$.

- O tamanho de uma memória de nível i é estritamente menor do que uma memória de nível $i + 1$.

Tendo essas propriedades garantidas, Frigo et al. provaram em (FRIGO et al., 1999) que uma cache (Z_i, L_i) em um nível i de um modelo LRU multi-nível sempre contém as mesmas linhas de cache que uma cache (Z_i, L_i) simples gerenciada por LRU desde que tenha a mesma seqüência de acessos à memória. Com isso, é possível provar que o modelo com apenas dois níveis é suficiente para provar resultados sobre modelos com hierarquia de cache mais complexas.

2.1.3 Associatividade e Reposição Automática

O modelo de cache ideal tem como característica ter a memória totalmente associativa, ou seja, qualquer bloco da memória pode ser guardado em qualquer lugar da cache.

Já a reposição automática diz que quando um bloco está para ser trazido para a cache, isso é automaticamente feito pelo *hardware*, e o algoritmo não necessita tratar essas operações de memória.

Nem sempre é assim que a cache funciona. Para a associatividade, a cache geralmente implementa uma associatividade limitada, significando que cada linha de cache pertence a um agrupamento de x blocos, sendo chamada de uma cache x-associativa. A maioria das caches variam de *1-way associativity* até *8-way associativity*. Sendo assim, faz-se necessário uma conversão para algo mais realista.

Para que o modelo de cache ideal possa ser utilizado quando o sistema não gerencia operações de memória automaticamente, é necessário provar que se pode simular uma cache com reposição automática sem um acréscimo na complexidade tanto de número de instruções quanto de *cache-misses*.

Frigo et al., em (FRIGO et al., 1999) provaram que para uma constante $\alpha > 0$, uma cache LRU $(\alpha Z, L)$ pode ser simulada em um espaço $O(Z)$ tal que cada acesso à cache tem $O(1)$ de complexidade de tempo. Com isso, o modelo de cache ideal pode ser reduzido para utilizar uma memória *1-way associativity* e um gerenciamento de memória manual.

2.2 Algoritmos Estudados

Existem muitos problemas já estudados sob o aspecto da eficiência de cache. Em (CHOWDHURY; RAMACHANDRAN, 2006), Chowdhury e Ramachandran apresentaram uma coleção de algoritmos que utilizam programação dinâmica que são *cache-oblivious* e minimizam o número de *cache-misses* transcorridos.

Além dos algoritmos apresentados, Chowdhury e Ramachandran também apresentam em (CHOWDHURY; RAMACHANDRAN, 2006) um *"framework"* para o desenvolvimento de algoritmos *cache-oblivious*. Esse *framework* pode ser utilizado para problemas que utilizam três laços aninhados, tais como o algoritmo de Floyd-Warshall, multiplicação de matrizes e o *gap problem*.

Apesar de estudarem os problemas, muito pouco é apresentado de resultados práticos. Ao se implementar na prática os algoritmos *cache-oblivious* da literatura, observa-se que para se obter uma melhora no tempo de execução em relação aos algoritmos comuns, muitos ajustes devem ser feitos, e muitas vezes quebrando o pressuposto de que o programa não tem conhecimento nenhum sobre a cache.

Uma técnica bastante utilizado nesses algoritmos é a divisão e conquista, onde o problema é recursivamente subdividido em subproblemas até que o subproblema tenha um tamanho b e então seja resolvido de maneira trivial. Um ponto de ajuste comum em quase todos os algoritmos *cache-oblivious* de programação dinâmica conhecidos é a escolha de que tamanho b deve ter. Sem o conhecimento do tamanho da cache, é necessário configurar esse tamanho ao menor possível, realizando assim várias subdivisões desnecessárias dentro da cache. Se soubermos o tamanho da cache, é possível escolher um b tal que quando o subproblema couber na cache, ele será resolvido da maneira tradicional, não fazendo assim nenhuma subdivisão desnecessária.

A seguir apresentamos alguns problemas estudados, assim como resultados práticos parciais das soluções para esses problemas. Para avaliarmos o desempenho dos algoritmos na prática, implementamos tanto a versão tradicional quanto a *cache-oblivious*. Escolhemos aplicar uma metodologia em que nenhum conhecimento prévio da cache é conhecido. Dessa forma, o b escolhido foi sempre o menor possível, para todos os problemas analisados.

2.2.1 Maior Subseqüência Comum

Dada uma seqüência $X = (x_1, ..., x_n)$ e uma seqüência $Y = (y_1, ..., y_m)$, uma seqüência $Z = (z_1, ..., z_k)$ é uma subseqüência comum de X e Y se ela é uma subseqüência de X e de Y. No problema da maior subseqüência comum, deve-se achar a subseqüência comum entre elas de maior tamanho.

Esse problema pode ser resolvido pela seguinte recorrência, onde $C(i, j)$ representa o tamanho da maior subseqüência comum de X e Y:

$$C(i,j) = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0, \\ C(i-1, j-1) + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j, \\ \max(C(i, j-1), C(i-1, j)) & \text{se } i, j > 0 \text{ e } x_i \neq y_j. \end{cases} \tag{2.3}$$

A solução por programação dinâmica segue diretamente dessa recorrência. A complexidade de tempo do algoritmo é de $\Theta(nm)$ e ele tem $O(\frac{nm}{L})$ cache misses (CHOWDHURY; RAMACHANDRAN, 2006). O algoritmo *cache-oblivious* para encontrar a maior subseqüência comum, apresentado em (CHOWDHURY; RAMACHANDRAN, 2006), tem uma complexidade de *cache-misses* de apenas $O(\frac{mn}{ZL})$, enquanto mantém a mesma complexidade de tempo $\Theta(nm)$.

A idéia por trás do algoritmo é dividir recursivamente a computação da solução do problema em quatro subproblemas de um quarto do tamanho do problema original, até que o vetor utilizado caiba inteiramente na cache. Para isso, é necessário propagar pela matriz os resultados computados pelos subproblemas para podermos recuperar a subseqüência correta.

Tabela 2.1: Tempo de execução dos algoritmos da Maior Subseqüência Comum (em segundos) para cada entrada do problema. A última linha apresenta a razão obtida (aceleração) entre o tempo dos dois algoritmos.

Algoritmo	8192	16384	32768	65536
Tradicional (LCS-T)	$0,680$	$2,770$	$12,920$	$43,639$
Cache-Oblivious (LCS-CO)	$0,650$	$2,180$	$8,580$	$39,370$
Aceleração (LCS-T/LCS-CO)	$1,046$	$1,270$	$1,506$	$1,108$

O teste foi realizado com uma instância com seqüências com 8192 a 65536 itens, escolhidos aleatoriamente entre as letras do alfabeto. Como visto na Tabela 2.1, o tempo de execução é um pouco melhor do que o algoritmo tradicional. Todavia, a melhora no tempo de execução está longe do reportado em (CHOWDHURY, 2005), que alcança uma aceleração de até $2,38$. O principal motivo disso é que a implementação do algoritmo tem o caso base definido como o menor possível, ao contrário do ocorrido em (CHOWDHURY, 2005).

2.2.2 Multiplicação de Matrizes

Dado uma matriz A de tamanho $m \times n$ e uma matriz B de tamanho $n \times p$, deve-se gerar uma matriz C de tamanho $m \times p$ contendo a multiplicação de A por B. Essa multiplicação pode ser definida como:

$$C(i,j,k) = \begin{cases} A(i,1) \times B(1,j) & \text{se } k = 1, \\ C(i,j) + A(i,k) \times B(k,j) & \text{se } i,j > 0 \text{ e } k > 1. \end{cases} \tag{2.4}$$

O algoritmo para resolução desse problema segue diretamente da definição do problema, tendo complexidade $O(mpn)$. Entretanto, existem algoritmos com menor complexidade para esse problema. Strassen, em (STRASSEN, 1969), apresenta um algoritmo com complexidade $O(n^{\log_2 7})$ e, conforme Prokop, em (PROKOP, 1999), incorre em $\Theta(\frac{n^2}{L} + \frac{n^{\log_2 7}}{L\sqrt{Z}})$ *cache misses*, que é ótimo.

O problema da multiplicação de matrizes é um problema bastante conhecido, tendo o seu algoritmo *cache-oblivious* apresentado por Prokop em (PROKOP, 1999). O algoritmo divide as matrizes A e B conforme o seu tamanho, em um dos 3 casos:

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} \tag{2.5}$$

$$\begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 \tag{2.6}$$

$$A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} \tag{2.7}$$

O primeiro caso (2.5) acontece quando $m \geq \max(n,p)$. O segundo (2.6) quando $n \geq \max(m,p)$ e o terceiro (2.7), quando $p \geq \max(n,m)$. As matrizes são divididas dessa maneira até um caso base de tamanho b, e então resolvidas de forma tradicional.

Tabela 2.2: Tempo de execução do algoritmo de Multiplicação de Matrizes (em segundos) para cada entrada do problema. A última linha apresenta a razão obtida (aceleração) entre o tempo dos dois algoritmos.

Algoritmo	256	512	1024
Tradicional (MAT-T)	$0,49$	$2,78$	$21,06$
Cache-Oblivious (MAT-CO)	$0,53$	$3,31$	$25,25$
Aceleração (MAT-T/MAT-CO)	$0,92$	$0,84$	$0,83$

Os resultados não chegam nem perto dos descritos em (PROKOP, 1999). A causa mais aparente é a mesma pela qual o algoritmo da Maior Subseqüência Comum não conseguiu repetir os resultados apresentados na literatura. Por não definirmos o caso base levando

em consideração o tamanho da cache, muitas instruções extras são realizadas para dividir a matriz, o que acaba encobrindo a vantagem do menor número de *cache misses*.

2.2.3 Gap Problem

O problema do *gap* (WATERMAN, 1995) é uma generalização do problema da distância de edição (*edit distance problem*) (LEVENSHTEIN, 1966). Seja X e Y seqüências de elementos de tamanho m e n, respectivamente, ao transformarmos X em Y, um *gap* em X corresponde a uma seqüência de exclusões consecutivas, e um *gap* em Y corresponde a uma seqüência de inserções consecutivas.

Nesse problema o custo de um *gap* não é necessariamente igual a soma dos custos de cada exclusão (ou inserção). Com isso, uma função W(i,j) é definida como o custo de excluir $x_{i+1}, ..., x_j$ de X, e uma função W'(i,j) é definida como o custo de inserir $y_{i+1}, ..., y_j$ em X. Ainda, temos o custo de substituição de um elemento de X por um de Y, definido como $S(x_i, y_j)$. Sendo D(i,j) o custo mínimo de transformar $x_1, ..., x_i$ em $y_1, ..., y_i$, a seguinte recorrência resolve o problema:

$$D(i,j) = \begin{cases} 0 & \text{se } i,j = 0, \\ W(0,j) & \text{se } i = 0, 1 \leq j \leq n. \\ W'(0,j) & \text{se } j = 0, 1 \leq i \leq m. \\ \min(D[i-1,j-1] + S(x_i,y_j), E(i,j), F(i,j)) & \text{se } i,j > 0. \end{cases} \tag{2.8}$$

onde:

$$E(i,j) = \min_{0 \leq q < j} \{D(i,q) + W(q,j)\}$$

e

$$F(i,j) = \min_{0 \leq p < i} \{D(p,j) + W'(p,i)\}$$

A solução por programação dinâmica segue pela recorrência. Para utilizarmos a idéia de dividir o problema em subproblemas menores, até que eles caibam na cache, agora é necessário propagar mais informações, uma vez que agora cada $D(i,j)$ depende, além de $D(i-1,j-1)$, de $D(0,j)...D(i-1,j)$ e $D(i,0)...D(i,j-1)$ também. Dessa forma, as funções $E(i,j)$ e $F(i,j)$ também necessitam ser subdivididas.

Para podermos utilizar os resultados calculados de E e F, definimos funções que propagam essas informações de uma subdivisão para a outra, e aplicamos essas novas funções recursivamente ao mesmo tempo em que calculamos o *Gap*.

Tabela 2.3: Tempo de execução do algoritmo do Gap (em segundos) para cada entrada do problema. A última linha apresenta a razão obtida (aceleração) entre o tempo dos dois algoritmos.

Algoritmo	512	1024	2048
Tradicional (GAP-T)	$0,439$	$4,55$	$44,30$
Cache-Oblivious (GAP-CO)	$2,36$	$17,36$	$141,89$
Aceleração (GAP-T/GAP-CO)	$0,186$	$0,26$	$0,31$

Para W e W’ foram escolhidas funções constantes, que retornavam o tamanho total da seqüência dada. Mais uma vez, o problema não foi otimizado levando em conta o tamanho da cache, e dessa forma o seu desempenho ficou longe da versão tradicional. Nesse problema em particular é especialmente importante a escolha ótima para a base, uma vez que a cada subdivisão do problema, a propagação de E e de F também é realizada de forma recursiva, gerando muitas subdivisões extras.

3 *BIN PACKING*

O problema *Bin packing* é um problema clássico da otimização combinatória. Este problema é conhecido na literatura portuguesa como o problema de empacotamento unidimensional.

O problema de empacotamento unidimensional pode ser enunciado como a seguir: Dada a capacidade W de recipientes (todos têm a mesma capacidade), um conjunto finito $I = \{1, 2, ..., n\}$ de itens, e um valor real $w_i \in [0, B]$ associado a cada item i, representando o tamanho (*size*) do item i, encontre uma partição de I em conjuntos disjuntos $I_1, I_2, ..., I_k$ tal que a soma dos tamanhos dos itens em cada I_j não seja maior que W e k seja o menor possível.

O empacotamento unidimensional é um problema classificado como NP-Difícil (GAREY; JOHNSON, 1979). Um outro problema NP-Difícil (3-Partition) é um caso especial do empacotamento unidimensional (GAREY; JOHNSON, 1979).

3.1 Soluções para o Empacotamento Unidimensional

3.1.1 Heurísticas e Meta-heurísticas

Muitas heurísticas e meta-heurísticas já foram propostas para este problema. Dentre as heurísticas, existem algumas que possuem aproximação com garantia. Dentre elas, podemos citar:

- *First Fit (FF)*: considerando os itens em qualquer ordem, insere-se os itens num recipiente, até que não se possa inserir mais nenhum item sem extrapolar a capacidade do mesmo. Quando a capacidade for alcançada, iniciar um novo recipiente. Este algoritmo é classificado como um algoritmo online, pois os elementos podem ser processados conforme são recebidos, sem necessidade de conhecer informação prévia, antes da aplicação do algoritmo. Este algoritmo possui aproximação garantida $FF(I) \geq \frac{17}{10}(OPT(I) - 1)$ (GAREY; JOHNSON, 1979).

- *First Fit Decreasing (FFD)*: para cada item, tenta inseri-lo em todos bins previamente alocados, antes de iniciar um novo bin. O FFD possui aproximação garantida $FFD(I) = \frac{11}{9}OPT(I)$ (GAREY; JOHNSON, 1979).

Outros algoritmos de aproximação para este problema, tais como o *Best Fit* e o *Best Fit Decreasing*, podem ser encontrados em (AUSIELLO et al., 1999).

3.1.2 Algoritmos Exatos

Além das heurísticas e meta-heurísticas, algoritmos exatos também foram propostos. Dentre os algoritmos exatos, destaca-se o algoritmo que usa a técnica de geração de colu-

nas. A técnica de geração de colunas é uma técnica de otimização combinatória indicada para resolver problemas que possuem um grande número de variáveis, que é o caso do problema de empacotamento unidimensional. Esta técnica pode ser vista com mais detalhes na literatura de otimização (DESAULNIERS; DESROSIERS; SOLOMON, 2005). A idéia básica do método de geração de colunas é resolver um subproblema do problema original, cuja solução ótima é a mesma que no problema original. Para determinar este subproblema, executa-se um número de iterações, que no máximo será igual ao número de variáveis do problema original, determinando-se o valor associado às variáveis a cada restrição do problema (coluna), por iteração.

Para determinar os valores destas variáveis, a cada iteração deve ser resolvido um outro problema, que no caso, é o problema da mochila. Maiores detalhes do método de geração de colunas não serão aqui expostos, visto que o mesmo requer o entendimento de procedimentos de otimização que requerem conhecimentos avançados de otimização, que não é o foco deste trabalho.

Atualmente, considerando nossa revisão bibliográfica sobre este problema, o algoritmo de geração de colunas para o empacotamento unidimensional que resolve as maiores instâncias propostas na literatura para este problema foi proposto por Applegate, Buriol, Dillard, Johnson e Shor (APPLEGATE et al., 2003). O tempo de cada iteração depende principalmente de uma resolução linear, e da resolução do problema da mochila, sendo que a otimização da resolução de ambos pode ser considerada igualmente importante para a diminuição do tempo gasto pelo método.

Visto que o problema da mochila é resolvido inúmeras vezes (a cada iteração) para resolver uma instância do problema de empacotamento unidimensional, torna-se ainda mais importante ter um algoritmo do problema da mochila que seja o mais rápido possível. Esta otimização terá um impacto direto e importante na diminuição do tempo total da resolução do problema de empacotamento unidimensional. Com esta motivação, o problema da mochila é a seguir apresentado, para depois descrevermos o método de resolução proposto.

4 PROBLEMA DA MOCHILA

O Problema da Mochila (*Knapsack Problem*) é um dos problemas mais estudados em otimização combinatória. Ele e suas variantes tem muitas aplicações práticas, e são de extrema importância para a informática teórica, uma vez que aparecem como subproblemas de vários outros problemas de otimização combinatória.

Todas as variações do Problema da Mochila (inteiro) pertencem à classe de problemas NP-Completo, ou seja, não se conhece um algoritmo que o resolva em tempo polinomial. Entretanto, utilizando programação dinâmica existem algoritmos que resolvem os diversas variações do problema da mochila em tempo pseudo-polinomial.

Para um algoritmo executar em tempo pseudo-polinomial, é necessário que o seu tempo de execução seja polinomial em relação ao valor da entrada, ao invés de ser polinomial em relação ao comprimento da entrada (o número de dígitos da entrada). Logo, um algoritmo pseudo-polinomial pode ser exponencial em relação ao comprimento da entrada.

O problema da mochila não é útil apenas em problemas de empacotamento. Existem aplicações nas mais diversas áreas como criptografia e gerenciamento de projetos. Além dessas, ele freqüentemente é um subproblema de outro problema, sendo executado diversas vezes, e seu desempenho nesses casos é crítico.

Nesse sentido, é importante encontrar maneiras de diminuir ao máximo o tempo de processamento do algoritmo, e minimizar o número de *cache-misses* é uma maneira nova de abordar esse problema.

4.1 Problema

O Problema da Mochila é definido como:

Definição 4.1.1. *Dado uma capacidade de uma mochila $W \in \mathbb{N}$, um conjunto de tipos de itens $I = \{1, ..., n\}$, tal que que cada tipo possua um peso w_i e um lucro p_i, para $1 \leq i \leq n$. Encontre o subconjunto de objetos que maximize o lucro total na mochila, respeitando a capacidade W da mochila.*

O problema da mochila tem diversas variantes, tais como só aceitar um objeto de cada tipo, limitar ou não o número de objetos por tipo, permitir que se tenha uma número de objetos fracionário, entre outros. Desses, estamos interessados no Problema da Mochila Não Limitado (*Unbounded Knapsack Problem*). A seguir, apresentamos rapidamente algumas variações importantes do problema da mochila.

4.2 Variações

4.2.1 Problema Fracionário da Mochila

Nessa variante o número de itens por tipo é fracionário, ou seja, é possível dividir um item para caber dentro da mochila.

A definição em termos de programação linear é a seguinte:

$$\text{Maximizar } \sum_{i=1}^{n} p_j x_j \tag{4.1}$$

$$\text{Sujeito a } \sum_{i=1}^{n} w_i x_i \leq W$$
$$x_i \geq 0, x_i \in \mathbb{R}^+ \mid i \in \{1, ..., n\}$$

Note que pela característica fracionária, uma solução gulosa exata é possível. Basta escolher o tipo de item i que tenha a maior razão $\frac{p_i}{w_i}$ e a resposta será $x_i = \frac{W}{w_i}$, com todos os outros x_j, $j \neq i$ iguais a 0.

4.2.2 Problema Booleano da Mochila

O Problema Booleano da Mochila (*0-1 Knapsack Problem*) é aquele em que só se pode colocar um item para cada tipo, ou seja, só pode escolher se o item faz parte da solução ótima. Definindo formalmente:

$$\text{Maximizar } \sum_{i=1}^{n} p_j x_j \tag{4.2}$$

$$\text{Sujeito a } \sum_{i=1}^{n} w_i x_i \leq W$$
$$x_i \in \{0, 1\} \mid i \in \{1, ..., n\}$$

4.2.3 Problema da Mochila Limitado

No Problema da Mochila Limitado (*Bounded Knapsack Problem*), um tipo de item i só pode aparecer na solução um numero máximo de b_i vezes. Definindo formalmente:

$$\text{Maximizar } \sum_{i=1}^{n} p_j x_j \tag{4.3}$$

$$\text{Sujeito a } \sum_{i=1}^{n} w_i x_i \leq W$$
$$x_i \in \{0, 1, ..., b_i\} \mid i \in \{1, ..., n\}$$

Esse problema já possui algumas características interessantes, porém o fato ter um número fixo de vezes que um item pode aparecer na mochila limita bastante o quanto o problema pode ser otimizado. Já o problema a seguir não possui essa limitação.

4.3 Problema da Mochila Não Limitado

O Problema da Mochila Não Limitado (Unbounded Knapsack Problem, UKP) é aquele onde o número de objetos por tipo não é limitado, ou seja, podemos utilizar quantos objetos de um tipo quanto quisermos.

Formulando como um problema de programação inteira:

$$\text{Maximizar} \quad \sum_{i=1}^{n} p_j x_j \tag{4.4}$$

$$\text{Sujeito a} \quad \sum_{i=1}^{n} w_i x_i \leq W$$
$$x_i \geq 0, x_i \in \mathbb{Z} \mid i \in \{1, ..., n\}$$

Esse problema pode ser facilmente convertido para o Problema da Mochila Limitado, apenas pondo como limite de cada tipo de objeto um valor $b_i = \lfloor \frac{W}{w_i} \rfloor$, $\forall i \in \{1, ..., n\}$. Todavia, existem algumas características desse problema que permitem otimizar a sua solução.

Para a resolução desse problema, existem duas técnicas principais utilizadas: a Programação Dinâmica e o *Branch & Bound*. Esse trabalho tem como foco exclusivamente a solução por programação dinâmica, tanto a versão trivial quanto a versão que utiliza algumas propriedades exclusivas do problema não limitado para acelerar o processamento da solução.

4.3.1 Dominâncias

Uma vez que o problema é sabidamente NP-Completo, torna-se importante a busca de maneiras de reduzir o tamanho do espaço de busca, e a eliminação de termos redundantes é uma excelente maneira de se atingir esse objetivo.

Ao compararmos tipos de itens par a par, é possível observarmos situações em que nunca utilizaríamos um tipo de item, e então podemos simplesmente não processá-lo, reduzindo o problema a ser considerado.

Dominância simples acontece quando um tipo de objeto tem um lucro associado maior e um peso associado menor do que outro. Esse conceito foi desenvolvido e provado por Gilmore e Gomory (GILMORE; GOMORY, 1961, 1963). Martello e Toth, em (MARTELLO; TOTH, 1990), estenderam o conceito de dominância para incluir dominância múltipla. Existem ainda mais tipos de dominâncias, apresentadas a seguir:

1. Um tipo de item j domina um tipo diferente de item i **se** $p_j \geq p_i$ e $w_j \leq w_i$.

2. Um tipo de item j domina multiplamente um tipo diferente de item i **se** $\lfloor \frac{w_i}{w_j} \rfloor > \frac{p_i}{p_j}$.

3. Um conjunto de tipos de itens $J \subseteq I$ domina coletivamente um tipo de item i, $i \notin J$, **se** existe um conjunto de valores $y_1, ..., y_n$ tal que

$$\sum_{j \in J} y_j w_j \leq w_i \quad e \quad \sum_{j \in J} y_j p_j \geq p_i.$$

4. Um conjunto de tipos de itens $J \subseteq I$ domina limiarmente (*threshold dominates*) um tipo de item i, $i \notin J$, **se** existe um multiplicador $\alpha \in \mathbb{N}$ e um conjunto de valores $y_1, ..., y_n$ tal que
$$\sum_{j \in J} y_j w_j \leq \alpha w_i \quad e \quad \sum_{j \in J} y_j p_j \geq \alpha p_i.$$

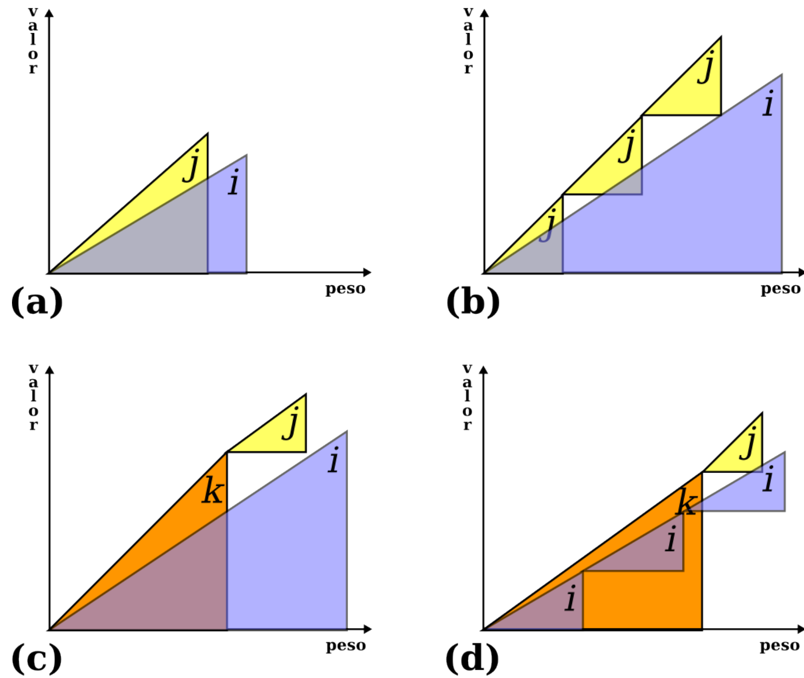

Figura 4.1: Todos os casos em que i é dominado: (a) dominância simples, (b) dominância múltipla, (c) dominância coletiva e (d) dominância limiar

Identificar esses tipos de dominâncias nem sempre é uma tarefa fácil, porém apresentamos no próximo capítulo uma versão do algoritmo utilizando programação dinâmica que calcula e se aproveita da maior parte dessas dominâncias em tempo de execução, tanto para o algoritmo que não minimiza *cache-misses* quanto para o que minimiza.

5 SOLUÇÃO *CACHE-OBLIVIOUS* PARA O PROBLEMA DA MOCHILA ILIMITADO

O Problema da Mochila Ilimitado tem melhores resultados quando resolvido por programação dinâmica ou *Branch & Bound*. Nesse trabalho nos concentramos na solução por programação dinâmica, com o intuito de minimizar o número de *cache-misses* do algoritmo, enquanto mantendo o algoritmo *cache-oblivious*.

Dado um vetor de pesos w e um vetor de lucros p, pode-se definir $Kp(i)$ como sendo o lucro máximo obtido com uma mochila de tamanho i. Logo, $Kp(i)$ pode ser computado pela seguinte relação de recorrência:

$$Kp(i) = \begin{cases} 0 & \text{se } i = 0, \\ \max_{1 \leq j \leq n}\{p_j + Kp(i - w_j) | w_j \leq i\} & \text{se } i \neq 0. \end{cases} \tag{5.1}$$

Nas próximas seções, iremos descrever os algoritmos tradicionais e a versão *cache-oblivious* que minimiza o número de *cache misses*. Após, veremos as versões que se aproveitam das dominâncias e então iremos provar a complexidade de *cache-misses* para cada algoritmo. Finalmente, iremos apresentar os resultados obtidos.

5.1 Solução Tradicional

Derivando diretamente da relação de recorrência (5.1), é possível calcular a solução ótima do problema da mochila para um tamanho W, resolvendo incrementalmente para cada mochila de tamanho $i < W$, e então usando essas subsoluções para derivar a solução para a mochila de tamanho W.

Utilizando um vetor Kp para guardar o lucro obtido para cada tamanho de mochila, é possível reutilizar as soluções já calculadas. Logo, para preencher esse vetor, deve-se para cada tamanho de mochila i, guardar o máximo lucro obtido com qualquer combinação de itens. Abaixo é descrito o algoritmo que faz justamente isso.

ALGORITMO 5.1: PROBLEMA DA MOCHILA ILIMITADO TRADICIONAL

Entrada Um vetor de lucros p e um vetor de pesos w com n elementos, e uma capacidade W.

Saída O lucro $profit$ e o peso $weight$ ótimo para a capacidade W e um vetor de itens K que torna possível a recuperação da seqüência de itens utilizados na solução ótima.

```
 1  para todo i ∈ {0, ..., W} :
 2        K[i] ← 0
 3        Kp[i] ← 0
 4
 5  profit ← 0
 6  weight ← 0
 7  para todo s ∈ {1, ..., W} :
 8        para todo i ∈ {1, ..., n} :
 9              se w_i ≤ s :
10                    se p_i + Kp[s − w_i] ≥ Kp[s] :
11                          Kp[s] ← p_i + Kp[s − w_i]
12                          K[s] ← i
13
14        se Kp[s] > profit :
15              profit ← Kp[s]
16              weight ← s
17  retorne (profit, weight, K)
```

No algoritmo, dois vetores são preenchidos simultaneamente, K e Kp. $Kp[i]$ contém o máximo lucro possível para uma mochila de tamanho i, enquanto que $K[i]$ contém o último item utilizado para se chegar a $Kp[i]$, o que torna possível recuperar os itens utilizados na construção da solução.

Em $(1 - 3)$, os vetores são inicializados. No laço $(7 - 12)$, é iterado todas as capacidades da mochila, até o tamanho W e em $(10 - 12)$ é determinado o item que fornece maior lucro para o tamanho de mochila sendo analisado (se houver um). Em $(14 - 16)$ os valores ótimos são atualizados se necessário.

Analisando o algoritmo 5.1, fica claro que a complexidade de número de instruções executadas é $O(Wn)$, uma vez que para cada tamanho da mochila, todos os itens são percorridos. Apesar disso, o algoritmo não é polinomial, e sim pseudo-polinomial em relação ao tamanho da mochila, que pode ter tamanho exponencial, e cuja entrada é $\Omega(\log W)$. Os elementos acessados a cada iteração estão ligados ao peso de cada item, inutilizando a localidade espacial no acesso à memória desse algoritmo.

Note que com esse algoritmo ainda não é possível recuperar toda a seqüência de itens utilizados. Todavia, como guardamos todos os itens utilizados no cálculo em K, é relativamente simples recuperar essa informação. O próximo algoritmo faz justamente isso.

ALGORITMO 5.2: RECUPERAÇÃO DA SEQÜÊNCIA DE ITENS DA SOLUÇÃO ÓTIMA PARA O PROBLEMA DA MOCHILA ILIMITADO

Entrada Um vetor de pesos de tipos de itens w, um vetor de itens K, o lucro $profit$ e o peso $weight$ ótimos saídos do algoritmo para o Problema da Mochila.

Saída Um vetor S com os tipos de itens utilizados na solução ótima e um vetor Sq que para cada $i \in \{0, ..., |S|\}$, $Sq[i]$ indica a quantidade do tipo de item $S[i]$ utilizado.

```
1   para todo i ∈ {0,...,W}:
2         A_i ← 0
3
4   j ← 0
5   s ← weight
6   enquanto s > 0:
7         i ← K[s]
8         se A[i] = 0:
9               j ← j + 1
10              S[j] ← i
11              Sq[j] ← 1
12              A[i] ← j
13        senão:
14              Sq[A[i]] ← Sq[A[i]] + 1
15        s ← s − w_i
16
17  retorne (S, Sq)
```

No algoritmo, o vetor A é um vetor de índices, indicando para um dado item i, em qual posição de S (e Sq) esse item se encontra. S contém os itens utilizados na solução do problema e Sq a suas quantidades. Em (8), a condição testa se o item já foi encontrado antes. Se ele foi, a quantidade dele é incrementada (14), senão, um novo lugar em S e Sq é criado para o item, e o índice dele é armazenado em A.

O algoritmo 5.2 para a recuperação da seqüência de itens utilizados claramente não interfere na complexidade de tempo dos algoritmos combinados, uma vez que é feita no máximo W operações.

5.2 Solução *Cache-Oblivious*

Uma outra opção equivalente para resolver o problema da mochila é calcular os elementos de Kp da seguinte maneira: para cada tipo de item i, $i \in \{1, ..., n\}$, calcule o máximo lucro obtido para cada tamanho de mochila. O algoritmo a seguir descreve essa modificação no algoritmo original.

ALGORITMO 5.3: PROBLEMA DA MOCHILA ILIMITADO *Cache-Oblivious*

Entrada Um vetor de lucros p e um vetor de pesos w com n elementos, e uma capacidade W.

Saída O lucro $profit$ e o peso $weight$ ótimo para a capacidade W e um vetor de itens K que torna possível a recuperação da seqüência de itens utilizados na solução ótima.

```
 1  para todo i ∈ {0, ..., W}:
 2      Kp[i] ← 0;  K[i] ← 0
 3
 4  para todo i ∈ {1, ..., n}:
 5      para todo s ∈ {w_i, ..., W}:
 6          se p_i + Kp[s − w_i] ≥ Kp[s]:
 7              Kp[s] ← p_i + Kp[s − w_i]
 8              K[s] ← i
 9
10  profit ← Kp[W];  weight ← W
11  found ← 0
12  enquanto found = 0:
13      weight ← weight − 1
14      se Kp[weight] < profit:
15          found ← 1
16          weight ← weight + 1
17
18  retorne (profit, weight, K)
```

No algoritmo, novamente dois vetores são preenchidos simultaneamente, K e Kp, com $Kp[i]$ contendo o máximo lucro possível para uma mochila de tamanho i, e $K[i]$ contendo o último item utilizado para se chegar a $Kp[i]$.

No laço $(1-2)$, os vetores são inicializados. Em $(4-8)$, para cada item i, é calculado cada $Kp[s]$, com s variando até W. Em cada iteração do laço interno $(5-8)$, é calculado se, para o $Kp[s]$ atual, é proveitoso utilizar o item i ou não. Em $(10-16)$, é calculado a capacidade da menor mochila que obtém o maior lucro, o que nem sempre é necessário.

Apesar da lógica da solução não ter se alterado no algoritmo 5.3, o padrão de acesso a memória se altera muito quando comparamos ao algoritmo 5.1. No algoritmo *cache-oblivious* o acesso à memória é seqüencial, tanto para $Kp[s-w_i]$ quanto para $Kp[s]$, já que agora é o s que varia no laço mais interno, ao invés de w_i.

5.3 Dominâncias

A fim de utilizarmos as informações sobre dominâncias explicadas na seção 4.3, é necessário fazer algumas mudanças no algoritmo. Utilizando a abordagem de programação dinâmica, três dos quatro tipos de dominâncias são testados em tempo de execução: a

dominância simples, a múltipla e a coletiva.

Para o algoritmo tradicional (5.1), é necessário, cada vez que um item dominado é encontrado, remover esse item do vetor de pesos e lucros. Isso pode ser conseguido em tempo constante através da troca dos pesos e lucros do item dominado com o começo do vetor, e alterar o começo do vetor para uma posição a mais (essa alteração seria apenas incrementar um indicador do início do vetor).

Para que essa solução funcione, entretanto, é necessário que o vetor esteja ordenado por peso, o que pode ser feito por um algoritmo como Mergesort (KNUTH, 1998, cap. 5.2.4) com complexidade de tempo $O(n \log n)$. A seguir é descrito o algoritmo que utiliza dominâncias no cálculo da solução do Problema da Mochila Ilimitado.

ALGORITMO 5.4: PROBLEMA DA MOCHILA ILIMITADO QUE UTILIZA DOMINÂNCIAS

Entrada Um vetor de lucros p e um vetor de pesos w com n elementos (ordenados pelo peso), e uma capacidade W.

Saída O lucro $profit$ e o peso $weight$ ótimo para a capacidade W e um vetor de itens K que torna possível a recuperação da seqüência de itens utilizados na solução ótima.

```
1  para todo i ∈ {0, ..., W}:
2      Kp[i] ← 0;  K[i] ← 0
3      A[i] = i
4
5  profit ← 0;  weight ← 0
6  j ← 0
7  para todo s ∈ {1, ..., W}:
8      para todo i ∈ {j, ..., n}:
9          se w_i < s:
10             se p_i + Kp[s − w_i] ≥ Kp[s]:
11                 Kp[s] ← p_i + Kp[s − w_i]
12                 K[s] ← i
13         senão:
14             se w_i = s:
15                 se Kp[s] ≥ p_i:
16                     temp ← w_i;  w_i ← w_j;  w_j ← temp
17                     temp ← p_i;  p_i ← p_j;  p_j ← temp
18                     temp ← A[i];  A[i] ← A[j];  A[j] ← temp
19                     j ← j + 1
20                 senão:
21                     Kp[s] ← p_i
22                     K[s] ← i
23     K[s] = A[K[s]]
24     se Kp[s] > profit:
25         profit ← Kp[s]
26         weight ← s
27 retorne (profit, weight, K)
```

Note que o algoritmo altera os vetores de entrada. Para evitar esse problema, é necessário fazer uma cópia do vetor dos pesos e uma do vetor dos lucros durante a inicialização do algoritmo. Além disso, também é necessário guardar os índices originais durante o algoritmo para podermos recuperar a seqüência de itens usados na solução.

Para podermos executar o algoritmo 5.2 sem modificações, é necessário fazer a ordenação dos vetores w e p novamente, para podermos recuperar os itens corretos.

No algoritmo, além de K e Kp, um vetor extra é utilizado. O vetor A contém o índice do item na configuração de itens originais, antes de qualquer troca em decorrência de descobertas de itens dominados aconteça. Em (9) é testado se o item é menor do que o tamanho da mochila sendo analisada. Se ele for menor, se analisa o item normalmente. Caso ele tenha o mesmo tamanho da mochila atual, ele pode ser um item dominado. Em (14), é testado justamente isso. Se o item for dominado, ele é posto para fora dos itens sendo considerados $(16-18)$, e o indicador do começo do vetor de itens é atualizado (19).

A adaptação no algoritmo 5.3 para utilizar as dominâncias é muito mais simples. Basta acrescentar um teste antes de executar o laço interno, verificando se o lucro do item a ser utilizado (p_i) é maior do que o lucro já calculado para a capacidade $Kp[w_i]$, se ele for menor nunca será utilizado, e se pode passar para o próximo item. A seguir é apresentado o algoritmo para essa versão.

ALGORITMO 5.5: PROBLEMA DA MOCHILA ILIMITADO *Cache-Oblivious* QUE UTILIZA DOMINÂNCIAS

Entrada Um vetor de lucros p e um vetor de pesos w com n elementos (ordenados pelo peso), e uma capacidade W.

Saída O lucro $profit$ e o peso $weight$ ótimo para a capacidade W e um vetor de itens K que torna possível a recuperação da seqüência de itens utilizados na solução ótima.

```
1  para todo i ∈ {0, ..., W} :
2        Kp[i] ← 0 ;  K[i] ← 0
3
4  para todo i ∈ {1, ..., n} :
5        se Kp[w_i] < p_i :
6              para todo s ∈ {w_i, ..., W} :
7                    se p_i + Kp[s − w_i] ≥ Kp[s] :
8                          Kp[s] ← p_i + Kp[s − w_i]
9                          K[s] ← i
10
11 profit ← Kp[W] ;  weight ← W
12 found ← 0
13 enquanto found = 0 :
14       weight ← weight − 1
15       se Kp[weight] < profit :
16             found ← 1
17             weight ← weight + 1
18
19 retorne (profit, weight, K)
```

As modificações feitas para o algoritmo 5.5 são muito menores. A única mudança é o teste em (5), que testa se o item é dominado ou não. Ainda assim, o algoritmo se aproveita dos mesmos 3 tipos de dominâncias que as do algoritmo 5.4. Além disso, não é feita nenhuma alteração nos dados de entrada, e por consequência não é necessário fazer o ordenamento após a execução do algoritmo para recuperar a seqüência de itens usados (ou guardar os vetores de w e p originais).

Vale lembrar que o fato de usar dominâncias para minimizar o número de itens analisados não altera de nenhuma maneira a complexidade dos algoritmos, uma vez que no pior caso, sempre existirá uma instância em que não ocorre nenhuma dominância. Na próxima seção veremos qual a complexidade de $cache - miss$ dos algoritmos, e depois os resultados obtidos pela nossa implementação.

5.4 Complexidade das Soluções

A seguir seguem as provas da complexidade do algoritmo 5.1 e do algoritmo 5.3. Uma vez que para cada instância (n,W) do Problema da Mochila Ilimitado existe um caso em que nenhum item é dominado, o Algoritmo 5.4 se comportará exatamente como o algoritmo tradicional (Algoritmo 5.1), e o algoritmo 5.5 se comportará exatamente como o algoritmo *cache-oblivious* normal (Algoritmo 5.3).

Sendo assim, a utilização de dominâncias para tentar diminuir o número de itens não interfere no cálculo da complexidade de *cache-miss* dos algoritmos.

Teorema 5.4.1. *Dado uma capacidade W, um conjunto de tipos de itens de tamanho n, o algoritmo para o Problema da Mochila Ilimitado Tradicional (Algoritmo 5.1) implica em $\Theta(nW)$ cache misses.*

Prova. O pior caso que pode acontecer é quando o número de itens n é maior que a cache e a distância entre qualquer item é maior que a linha da cache. Ou seja, o pior caso ocorre quando $n > \frac{Z}{L}$ e $|w_i - w_j| > L$, $i \neq j$, $\forall i, j \in \{1, .., n\}$.

Para cada iteração de i, no laço interno $(11 - 14)$, o elemento $Kp[s - w_i]$ é acessado, causando no máximo um *cache miss*. Porém, na próxima iteração não há proveito de localidade de $Kp[s - w_i]$, uma vez que $Kp[s - w_{i+1}]$ está a uma distância maior que uma linha de cache. Logo, para completar todos os itens, pelo menos n *cache misses* ocorrerão até que a cache esteja cheia.

Tendo a cache cheia, a próxima iteração de s não terá necessariamente n *cache-misses*. Pela política de substituição ótima, uma vez que a cache estiver cheia, a linha que será acessada mais longe no futuro será a escolhida para ser substituída. Como a linha que será acessada mais futuramente é sempre a que contém o último $Kp[s - w_i]$ acessado, uma linha da cache conterá ele, enquanto que as outras conterão os elementos $K[s - w_0], ..., K[s - w_{\frac{Z}{L}}]$ e seus vizinhos, ou seja, os elementos acessados no início.

Com isso, temos que para cada iteração de s em que a cache já está cheia, $n - \frac{Z}{L}$ cache misses ocorrerão. Entretanto, a cada L iterações, porém, todas as linhas da cache deverão já ter sido renovadas, com isso acrescenta-se n *cache misses* a cada L iterações. Com isso, aumentamos em $\frac{Wn}{L}$ o número total de *cache misses*.

Logo, como temos no máximo W iterações de s, teremos $W(n - \frac{Z}{L}) + \frac{Wn}{L}$ *cache-misses* ao todo. Como $n - \frac{Z}{L} \in \Theta(n)$, e $\frac{Wn}{L} \in \Theta(nW)$, teremos uma complexidade pessimista de *cache-miss* de $\Theta(nW)$.

□

Teorema 5.4.2. *Dado uma capacidade W, um conjunto de tipos de itens de tamanho n, o algoritmo para o Problema da Mochila Ilimitado Cache-Oblivious (Algoritmo 5.3) implica em $O(\frac{nW}{L})$ cache misses.*

Prova. Podemos separar no cálculo da complexidade de *cache misses* do algoritmo dois casos distintos:

1. $W \leq Z$:

   Todo o vetor K cabe na cache. Com isso, a complexidade de cache misses consiste no preenchimento da cache. Como o caso em que $W > Z$ preenche toda a cache, esse caso acaba por não influenciar na complexidade de *cache misses* do algoritmo.

2. $W > Z$:

   O vetor K não cabe na cache. Para cada iteração do laço interno $(9 - 13)$, os elementos $Kp[s]$ e $Kp[s - i_{peso}]$ são acessados, causando no máximo 2 *cache misses*. Para a próxima iteração, como o índice s só é incrementado, tanto $Kp[s+1]$ quanto $Kp[s + 1 - w_i]$ já estão na cache, e não há *cache-miss*. De fato, o próximo *cache-miss* só poderá ocorrer quando L iterações tiverem transcorrido.

   Para o próximo item i, tendo toda a cache ocupada, pela política de substituição do modelo de cache ideal, até a ultima linha da cache seria reaproveitada, diminuindo Z cache misses a cada iteração de i. Logo, como temos no máximo W iterações para cada tipo de item i, temos $\frac{W-Z}{L}$ cache misses por iteração do laço externo.

   Logo, teremos $O\left(\frac{W-Z}{L}\right)$ cache-misses por iteração de i e $O\left(\frac{n(W-Z)}{L}\right)$ *cache misses* para todo o algoritmo, o que resulta em $O\left(\frac{nW}{L}\right)$ *cache misses*.

□

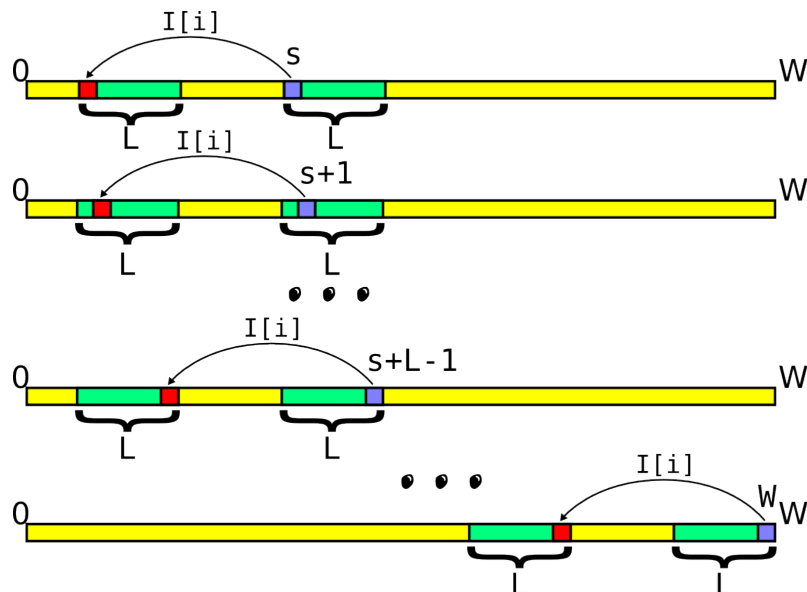

Figura 5.1: Acesso a memória no algoritmo 5.3

A principal diferença entre os dois algoritmos está em como os elementos de K são acessados. Como a versão *cache-oblivious* acessa os elementos de K em seqüência, com

um deslocamento entre os itens avaliados constante para cada iteração interna (como visto na Figura 5.1), ela se aproveita da localidade da cache, dividindo todo o trabalho do preenchimento e da atualização da cache por L.

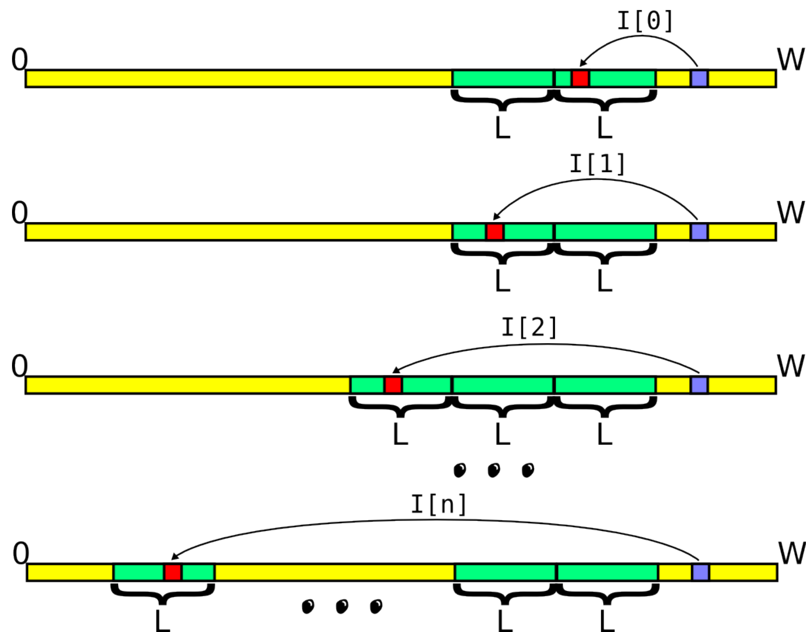

Figura 5.2: Acesso a memória no algoritmo 5.1

Já a versão tradicional tem deslocamentos que variam conforme os itens (como visto na figura 5.2), o que faz com que os acessos a K não se aproveitem da localidade, e tendo então a sua complexidade de cache-misses igual a sua complexidade de tempo.

Vale ainda destacar que o algoritmo 5.3 é ótimo, no sentido que nenhum algoritmo que resolva o problema da mochila com complexidade de tempo $\Theta(nW)$ e faça $\Theta(nW)$ acessos à memória gera menos *cache-misses* do que $O(\frac{nW}{L})$, assintoticamente. Uma vez que não é conhecido se existe uma maneira de resolver um problema NP-Completo em tempo polinomial, é possível que exista um algoritmo que tenha menos acessos a memória, e dessa maneira tenha uma complexidade de *cache-misses* inferior ao nosso algoritmo.

5.5 Resultados Computacionais

Todos os algoritmos para o problema da mochila foram implementados em C, em um Linux (kernel 2.6.27), compilados com o compilador GNU gcc 4.3.2, enquanto que os algoritmos para os outros problemas foram implementados em C++. Todos os testes foram executados em uma máquina Intel® Core™ 2 Quad (4 processadores de $2,4$ GHz cada), com 4 GB de memória principal, 4MB de cache L2 e 32KB de cache L1 (dados).

Para os testes de cache, foi utilizado o *software valgrind* (NETHERCOTE; SEWARD, 2007) com a ferramenta *cachegrind*. Essa ferramenta permite que se simule diversas configurações de cache, além de não ser necessário nenhuma modificação no código original. Uma vez que o *valgrind* faz uma simulação, o programa tem um tempo de execução multiplicado por um fator de até 30. Dessa forma, para os testes de cache, foram escolhidos problemas não muito grandes, com valores de cache menores do que as máquinas atuais normalmente possuem. Porém, para os testes de tempo de execução, os tamanhos normais de instâncias foram utilizados.

5.5.1 Instâncias

Todas as instâncias geradas são aleatórias, distribuídas uniformemente entre um intervalo de (w_{min}, w_{max}) para os pesos e de (p_{min}, p_{max}) para os lucros. Os valores w_{min}, w_{max}, p_{min} e p_{max} são informados pelo usuário para o gerador, junto com o tamanho da mochila W e o número de itens n. O gerador foi implementado em *Python*, utilizando o módulo *random* para o gerar os números aleatórios. Para esse tipo de instância, existem três casos diferentes utilizados na literatura (MARTELLO; TOTH, 1990). O primeiro é o não correlacionado, onde tanto peso quanto lucro de um item são gerados totalmente aleatórios. No segundo caso, o fracamente correlacionado, os pesos são escolhidos aleatoriamente, enquanto que os lucros são formados ao somarmos αw_i a um valor aleatório escolhido entre um intervalo $[-\beta, +\beta]$, para constantes α e β, ou seja, $p_i = \alpha w_i + \lfloor random() * 2\beta \rfloor - 2\beta$, onde $random()$ tem como saída um valor aleatório real entre $[0, 1]$. Finalmente, para o caso fortemente correlacionado, os pesos são escolhidos aleatoriamente, enquanto que os lucros derivam diretamente do peso, sendo $p_i \leftarrow \alpha w_i + \beta$, para constantes α e β.

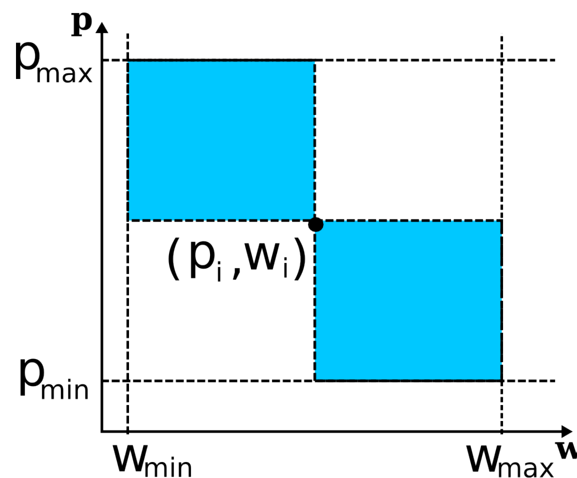

Figura 5.3: Item i como um ponto no plano. Qualquer ponto nas áreas destacadas ou dominam o item i ou são dominados por ele.

Ao escolhermos pesos e lucros não correlacionados, existe uma grande possibilidade de os itens escolhidos serem ou dominados ou dominadores. Se considerarmos um item i como um ponto (w_i, p_i) no retângulo definido pelos pontos (w_{min}, p_{min}) e (w_{max}, p_{max}), todos os pontos que estiverem na área retangular entre (w_{min}, p_i) e (w_i, p_{max}) dominarão i, e todos os pontos que estiverem na área retangular entre (w_i, p_{min}) e (w_{max}, p_i) serão dominados por i. Com isso, a cada item escolhido que não é dominado nem domina ninguém, o número de pontos disponíveis que não dominam nem são dominados diminui pela metade. A Figura 5.3 ilustra esse conceito.

Ao analisarmos o desempenho dos algoritmos sem dominâncias, essa característica do caso não correlacionado não é importante, porém para os algoritmos que utilizam dominâncias, ela é vital. Dessa forma, para todos os testes feitos, foram escolhidas instâncias geradas pelos casos fraca e fortemente correlacionados.

5.5.2 Comparações

Neste trabalho são comparados o número de *cache misses* ocorridos para os algoritmos simples e o tempo de execução de cada algoritmo apresentado. Na tabela 5.1, são apresentadas as abreviações utilizadas nas comparações.

Tabela 5.1: Abreviações dos Algoritmos

Abreviação	Descrição	Algoritmo
T	Algoritmo Tradicional (sem Dominâncias)	5.1
CO	Algoritmo *Cache-Oblivious* (sem Dominâncias)	5.3
domT	Algoritmo Tradicional (com Dominâncias)	5.4
domCO	Algoritmo *Cache-Oblivious* (com Dominâncias)	5.5

Para os testes do número de *cache-misses* ocorridos, escolheu-se tamanhos de cache entre 1KB e 8KB, em que obrigatoriamente os dados não caberiam totalmente na cache, a fim de simular uma situação real, onde o tamanho da mochila e o número de itens são muito maiores que o tamanho da cache. Optou-se por tamanho do problema, o caso em que o tamanho da mochila é 250.000 e o número de itens é de 10.000. Ainda, O tamanho da linha da cache escolhida foi o maior possível, respeitando a regra do *tall-cache* e escolhendo como tamanhos potências de 2.

Tabela 5.2: Número de *cache misses* para as versões que não utilizam Dominância, para diferentes configurações de cache com uma entrada de $W = 250.000$ e $n = 10.000$ para os algoritmos 5.1 (T) e 5.3 (CO)

Algoritmo	(Z=1024,L=32)	(Z=2048,L=32)	(Z=4096,L=64)	(Z=8192,L=64)
T (5.1)	$64,683$M	$56,049$M	$52,577$M	$38,153$M
CO (5.3)	$24,899$M	$24,889$M	$12,442$M	$12,440$M
Fator T/CO	$2,598$	$2,252$	$4,226$	$3,066$

Ao analisarmos o número de *cache-misses* ocorridos em ambos algoritmos, podemos notar que há uma redução muito significativa no número de *cache-misses* no algoritmo 5.3, como esperado. Outra observação necessária é que o total de *cache-misses* do algoritmo 5.3 (CO) diminui pela metade quando dobramos o tamanho de L. Isso indica que na prática o número de *cache-misses* é $O\left(\frac{nW}{L}\right)$ como comprovado pelo cálculo da complexidade de *cache-misses* do algoritmo.

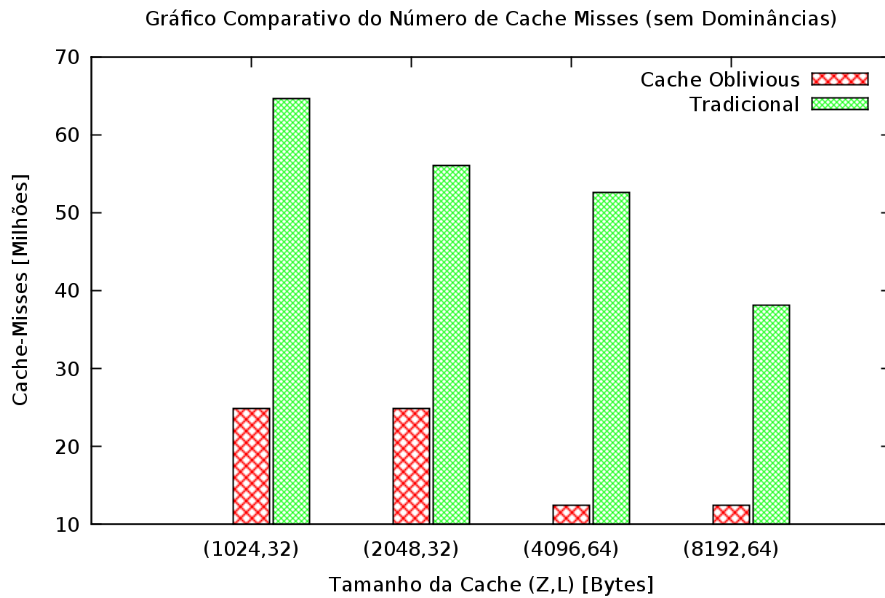

Figura 5.4: Número de *cache misses* para diferentes configurações de cache com uma entrada de $W = 250.000$ e $n = 10.000$ para os algoritmos 5.1 (T) e 5.3 (CO)

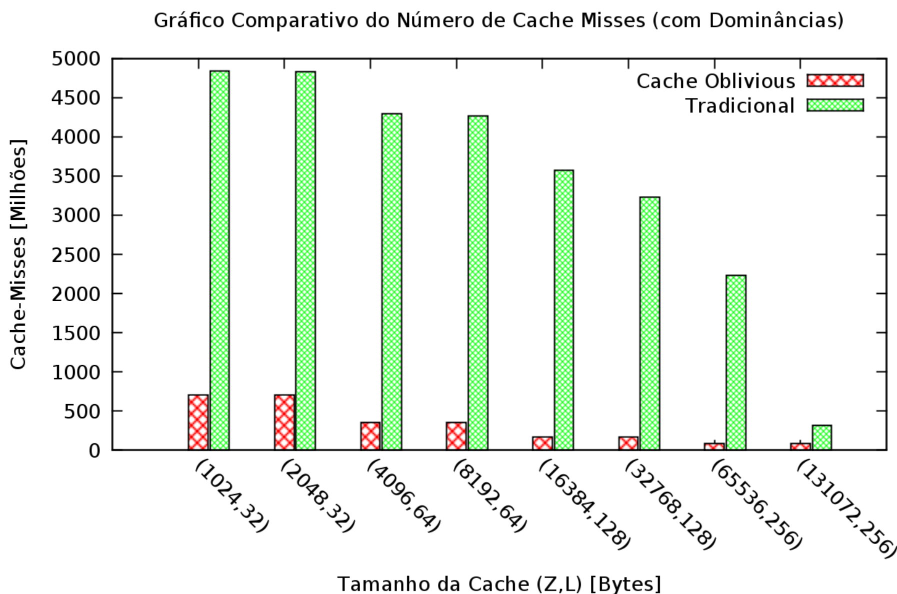

Figura 5.5: Número de *cache misses* para as versões que utilizam Dominância para diferentes configurações de cache com uma entrada de $W = 10.000.000$ e $n = 10.000$ para os algoritmos 5.1 (T) e 5.3 (CO)

Ao compararmos a Figura 5.4 com a Figura 5.5, notamos que a diferença do número de *cache misses* aumentou muito, indicando que a adaptação para utilização de dominâncias acentua as diferenças dos algoritmos. Pela dominação dos itens, eles acabam por ficarem muito mais espaçados entre si, caindo no pior caso para o algoritmo normal (a adaptação para utilizar dominâncias também tem o mesmo pior caso).

Entretanto, de nada adianta o número de *cache-misses* ser menor se o tempo de execução não obtiver nenhuma melhora. Dessa forma, foram feitos testes para medir o desempenho desses algoritmos que não utilizam informação de dominância. Devido ao alto

Tabela 5.3: Número de cache-misses de cada algoritmo com dominância (em milhões) para cada configuração de cache (Z,L)

Algoritmo	(2048,32)	(8192,64)	(32768,128)	(65536,256)
T (5.1)	4832.63M	4266.48M	3230.68M	2236.73M
CO (5.3)	703.40M	351.47M	175.14M	87.61M
Fator T/CO	$6,87$	$12,14$	$18,45$	$25,53$

tempo de execução desses algoritmos, e pelo fato deles não serem utilizados na prática, testes com um menor W e n foram realizados.

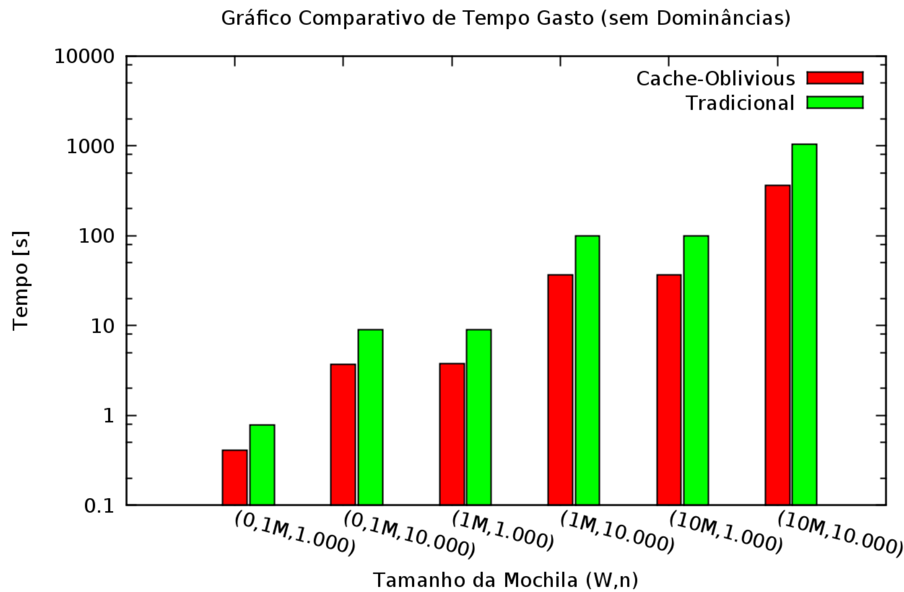

Figura 5.6: Tempo de execução dos algoritmos 5.1 (T) e 5.3 (C), em escala logarítmica.

Tabela 5.4: Tempo de execução (em segundos) e *Speed-Up* dos algoritmos sem Dominâncias.

W	n	T	CO	*Speed-Up* (T/CO)
$0,1$ M	1.000	$0,788$	$0,410$	$1,92$
$0,1$ M	10.000	$8,960$	$3,710$	$2,42$
1 M	1.000	$9,009$	$3,780$	$2,38$
1 M	10.000	$98,870$	$36,820$	$2,69$
10 M	1.000	$100,200$	$36,942$	$2,71$
10 M	10.000	$1242,341$	$362,36$	3.43

Comparando os testes na Tabela 5.4 e no gráfico da Figura 5.6, observarmos uma aceleração no tempo de execução de até $3,43$ vezes utilizando o algoritmo 5.3 (CO). Isso significa que o CO demora somente 29% do tempo que o algoritmo 5.1 (T) leva para computar a sua solução. Embora essa aceleração seja significativa, ela não terá sentido se não se verificar uma aceleração semelhante nos algoritmos que utilizam dominâncias para a diminuição da entrada.

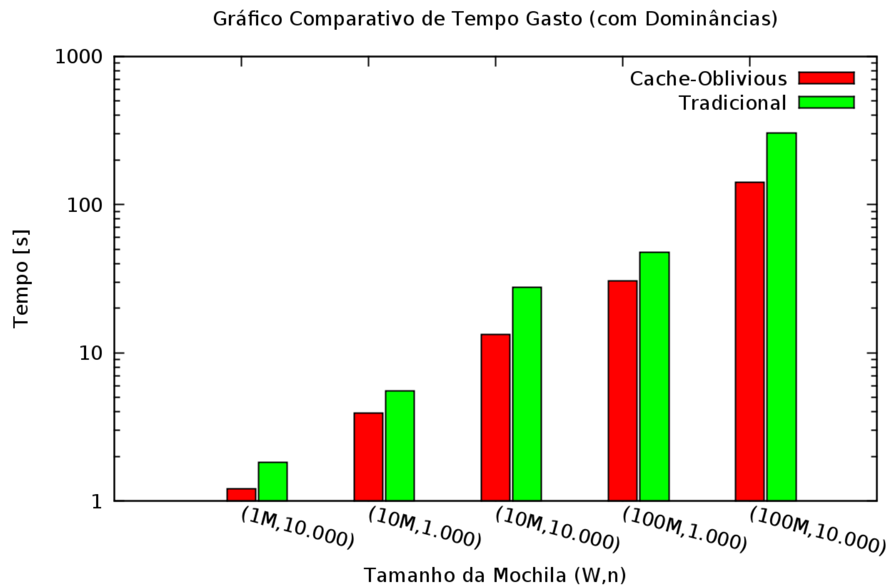

Figura 5.7: Tempo de execução dos algoritmos 5.4 (domT) e 5.5 (domCO), em escala logarítmica.

Tabela 5.5: Tempo de execução (em segundos) e *Speed-Up* dos algoritmos com Dominâncias

W	n	domT	domCO	*Speed-Up* (domT/domCO)
1 M	10.000	$1,82$	$1,21$	$1,50$
10 M	1.000	$5,55$	$3,92$	$1,42$
10 M	10.000	$27,57$	$13,33$	$2,07$
100 M	1.000	$47,53$	$30,47$	$1,56$
100 M	10.000	$304,50$	$141,36$	$2,15$

Ao verificarmos os valores na Tabela 5.5 e no gráfico da Figura 5.7, vemos que a aceleração do tempo de execução não se manteve nas versões dos algoritmos com dominâncias. Isso confirma a afirmação de que o fato do algoritmo utilizar dominâncias para diminuir o número de itens a processar não altera a complexidade de cache dele.

Alguns fatores também contribuem para a manutenção do rendimento do algoritmo *cache-oblivious*. Como afirmado anteriormente, o fato de termos menos itens causa um espaçamento maior entre eles, e por conseqüência uma pior utilização da cache pelo algoritmo tradicional. Dessa forma, a melhora na utilização da cache é justificada. Além disso, outro fator que causa um aumento na aceleração em relação ao algoritmo tradicional é o fato de que na implementação de dominância no algoritmo tradicional é necessário fazer mais operações além do teste para descobrir se um item é dominado (como fisicamente trocar elementos de lugar nos vetores de peso e lucro da entrada).

6 CONSIDERAÇÕES FINAIS

Nesse trabalho foi apresentado um estudo sobre o modelo de cache ideal e as suas propriedades e demonstrado que o modelo é factível com a maioria dos computadores atuais. Também foram estudados e implementados algoritmos *cache-oblivious* propostos na literatura, além de apresentados resultados e limitações desses algoritmos. Além disso, um novo algoritmo *cache-oblivious* foi proposto para o Problema da Mochila Ilimitado, e sua complexidade pessimista de *cache-misses* foi analisada.

Inicialmente, foi realizada uma apresentação de conceitos básicos sobre memórias cache e hierarquias de memória. Então, a título de comparação com o modelo de cache ideal, foram apresentados o modelo RAM e o modelo de memória externa. Após, foi apresentado o modelo de cache ideal. Todos os algoritmos utilizados aqui tiveram suas complexidades provadas utilizando esse modelo. O modelo apresenta algumas características que geralmente não são encontradas nas máquinas atuais, tais como a estratégia de substituição ótima e a cache totalmente associativa. Sendo assim, foi mostrado para cada caso que os resultados provados para o modelo são válidos para a grande maioria de configurações de cache encontradas na atualidade.

Foram apresentados algoritmos *cache-oblivious* propostos na literatura, expondo resultados práticos das implementações desses algoritmos, bem como as limitações deles. Os algoritmos apresentados foram o da Maior Subseqüência Comum (*Longest Commom Subsequence*), a Multiplicação de Matrizes e o Problema do *Gap* (*Gap Problem*). Em todos eles ficou claro que apesar de o algoritmo ser *cache-oblivious* para o cálculo da complexidade, é preciso quebrar o pressuposto de não ter nenhuma informação sobre a cache para se obter resultados satisfatórios na prática.

A seguir foi introduzido brevemente o problema do Empacotamento Unidimensional (*Bin-Packing Problem*), com o intuito de dar uma motivação para a escolha do Problema da Mochila Ilimitado para ser analisado em termos de eficiência de cache. Assim, foram introduzidas as principais heurísticas e metaheurísticas para resolver o problema do *bin-packing*, e mencionado como solução exata a técnica de geração de colunas e onde que o Problema da Mochila Ilimitado aparece nesse contexto.

Foi definido formalmente as variantes mais importantes do Problema da Mochila, comentando sobre a importância do problema como um todo. O problema é NP-Completo, e até hoje o melhor algoritmo conhecido tem complexidade de tempo de $O(nW)$, que é um algoritmo pseudo-polinomial no tamanho da mochila, uma vez que a entrada é $\Omega(\log W)$. Para o Problema da Mochila Ilimitado, em que a quantidade de cada tipo de item não é limitada, o conceito de dominância de itens, que permitem uma grande redução no número de itens computados de fato, foi apresentado, assim como o porquê dessas características não alterarem as complexidades do problema.

O algoritmo tradicional para a solução do Problema da Mochila Ilimitado e sua ver-

são *cache-oblivious* foi apresentada, assim como versões desses algoritmos que fazem uso de dominâncias para reduzir o número de itens computados. Então foi provado a complexidade pessimista de *cache-misses* tanto para o algoritmo tradicional quanto para o algoritmo *cache-oblivious*.

Finalmente, foram apresentados resultados práticos para as implementações dos algoritmos desenvolvidos. Nesses resultados, ficou claro que o algoritmo *cache-oblivious* apresentou um desempenho consideralvemente melhor que o algoritmo tradicional, tanto utilizando ou não dominâncias. Outro ponto importante desse algoritmo é o fato dele ser *cache-oblivious* também na prática, ao contrário da maioria dos algoritmos *cache-oblivious* propostos na literatura. Ainda, algoritmo *cache-oblivious* com dominâncias é extremamente mais simples do que o algoritmo tradicional com dominâncias, além de não alterar a entrada (o que causa a necessidade de se ordenar a entrada uma segunda vez ao término do algoritmo).

6.1 Trabalhos Futuros

Existem diversas possibilidades para trabalhos futuros, entre elas:

- Ampliar a idéia do algoritmo *cache-oblivious* para as outras variantes do problema da mochila;
- Aplicar o algoritmo implementado em algumas das diversas aplicações do Problema da Mochila Ilimitado, dentre eles, e principalmente, o Problema do Empacotamento Unidimensional (*Bin-Packing*);
- Analisar do ponto de vista de eficiência de cache as outras soluções para o Problema da Mochila Ilimitado, tais como a solução por *Branch & Bound*.

REFERÊNCIAS

APPLEGATE, D. L.; BURIOL, L. S.; DILLARD, B. L.; JOHNSON, D. S.; SHOR, P. W. The Cutting-Stock Approach to Bin Packing: theory and experiments. In: ALENEX 2003, 2003. **Anais. . .** [S.l.: s.n.], 2003.

AUSIELLO, G.; CRESCENZI, P.; GAMBOSI, G.; KANN, V.; MARCHETTI-SPACCAMELA, A.; PROTASI, M. **Complexity and approximation – Combinatorial Optimization Problems and their Approximability Properties**. [S.l.]: Springer-Verlag, 1999.

CHOWDHURY, R. A. **Experimental Evaluation of an Efficient Cache-Oblivious LCS Algorithm**. [S.l.]: University of Texas, Austin, 2005. (TR-05-43).

CHOWDHURY, R. A.; RAMACHANDRAN, V. Cache-oblivious dynamic programming. In: SODA '06: PROCEEDINGS OF THE SEVENTEENTH ANNUAL ACM-SIAM SYMPOSIUM ON DISCRETE ALGORITHM, 2006, New York, NY, USA. **Anais. . .** ACM, 2006. p.591–600.

DESAULNIERS, G.; DESROSIERS, J.; SOLOMON, M. M. **Column Generation**. [S.l.]: Springer Verlag, 2005.

FRIGO, M.; LEISERSON, C. E.; PROKOP, H.; RAMACHANDRAN, S. Cache-oblivious algorithms. In: IEE SYMPOS. FOUND. COMP. SCI., 40., 1999. **Proceedings. . .** [S.l.: s.n.], 1999. p.285–297.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractibility, A Guide to the Theory of NP-Completeness**. New York: W. H. Freeman and Company, 1979.

GILMORE, P. C.; GOMORY, R. E. A linear programming approach to the cutting stock problem. **Operations research**, [S.l.], v.9, p.848–859, 1961.

GILMORE, P. C.; GOMORY, R. E. A linear programming approach to the cutting stock problem – Part II. **Operations research**, [S.l.], v.11, p.863–888, 1963.

HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture**: a quantitative approach. Zweite.ed. [S.l.]: Morgan Kauffmann Publishers Inc., 1996.

KNUTH, D. E. **The art of computer programming**. 2nd.ed. [S.l.]: Addison-Wesley, 1998. v.III, Sorting and searching.

LEVENSHTEIN, V. Binary codes capable of correcting deletions, insertions and reversals. **Soviet Physics Doklady**, [S.l.], v.10, n.8, p.707–710, 1966.

MARTELLO, S.; TOTH, P. **Knapsack Problems. Algorithms and Computer Implementations**. Chichester: John Wiley and sons, 1990. 296p.

NETHERCOTE, N.; SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM SIGPLAN 2007 CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI 2007), 2007. **Proceedings...** [S.l.: s.n.], 2007.

PROKOP, H. **Cache-oblivious algorithms**. 1999. Tese (Doutorado em Ciência da Computação) — MIT.

STRASSEN, V. Gaussian elimination is not optimal. **Numerische Mathematik**, [S.l.], v.14, n.3, p.354–356, 1969.

VITTER, J. S. External Memory Algorithms and Data Structures: dealing with massive data. **ACM Computing Surveys**, [S.l.], v.33, n.2, Feb. 2007.

WATERMAN, M. S. **Introduction to Computational Biology**. [S.l.]: Chapman and Hall, London, UK, 1995.