0

LUCAS DEL CASTANHEL DIAS

ALGORITMOS DE APROXIMAÇÃO E PROGRAMAÇÃO DINÂMICA

Proposta de Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. André Luis Vignatti

CURITIBA

2015

LUCAS DEL CASTANHEL DIAS

ALGORITMOS DE APROXIMAÇÃO E PROGRAMAÇÃO DINÂMICA

Proposta de Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. André Luis Vignatti

CURITIBA

2015

SUMÁRIO

RESUMO

Para muitos problemas de otimização atualmente abordados encontrar a solução ótima é intratável , isto porque não conhecemos algoritmos que os solucionem em tempo polinomial. Sabendo desta limitação, podemos sacrificar a otimalidade (Por um valor conhecido) [31] e aplicar uma solução tratável, em algo que chamamos de esquemas de aproximação polinomial. Veremos neste trabalho definições do que são estes esquemas de aproximação e alguns algoritmos criados a partir deste conceito para resolver quatro problemas: Problema da Mochila, Escalonamento de tarefas em máquinas paralelas, Empacotamento e Caixeiro viajante eucludiano. Apresentamos também a proposta de continuidade do trabalho, a qual definimos como sendo estudar algoritmos em grafos de disco unitários, em especial [26], onde recentemente descobriu–se um esquema de aproximação para o problema da cobertura de vértices em grafos de disco unitário com pesos.

ABSTRACT

For many optimization problems actually seen, find the optimatial solution remains intractable, this because we do not know polynomial time algorithms for such problems. Aware of this limitation, we can sacrifice optimality (for a known value) [31] and apply a tractable solution, something we call Polynomial Time Approximation Schemes (PTAS). In this work we will see definitions on what are Poliynomial Time Aproximation Schemes and some algorithms created from this idea to solve four problems: The knapsack Problem, Scheduling jobs on identical parallel machines, Bin–packing and the Euclidean traveling salesman problem. We also present our future work, which we choose to study algorithms for unit disk graphs, in special [26], where a PTAS was recently discovered for the Weighted Unit Disk Cover Problem.

CAPÍTULO 1

INTRODUÇÃO

Esta proposta trata de dois conceitos: Algoritmos de aproximação e programação dinâmica. Veremos como ambas as técnicas podem ser combinadas para chegarmos próximo da solução ótima em problemas NP-Completos ou NP-difíceis, como em [2], em tempo polinomial em relação a um parâmetro de ajuste.

Ao contrário da maioria dos trabalhos que citamos, os quais trabalham com problemas específicos, focamos até o momento em estudar algoritmos de aproximação que usam programação dinâmica como técnica de projeto.

Destinaremos o restante desta introdução para apresentar estes dois conceitos e falar da organização desta proposta de qualificação.

1.1 Algoritmos de aproximação

Até o presente momento, alguns problemas abordados em ciência da computação não possuem algoritmos polinomiais capazes de calcular sua solução ótima. Estes problemas estão comumente nas classes NP-Completo e NP-difícil [13], cuja principal característica é a de que os algoritmos mais rápidos conhecidos executam em tempo exponencial em relação ao tamanho da entrada.

Sabendo que não podemos chegar à otimalidade sem gastar tempo exponencial, podemos sacrificá-la para encontrar uma solução próxima do ótimo resolvendo-a em tempo polinomial. Os algoritmos de aproximação provém soluções próximas as quais podemos quantificar a distância entre a solução e o valor ótimo para determinada instância e em tempo polinomial.

1.2 Programação dinâmica

Programação dinâmica é uma técnica, proposta inicialmente por Richard Bellmann [4]. Ela consiste em dividir um problema em instâncias menores, de solução mais simples. A solução global é formada a partir da combinação destas pequenas soluções.

Veremos algumas facetas da programação dinâmica em problemas mais simples e posteriormente como técnica de projeto de algoritmos para obter aproximações de problemas difíceis em tempo polinomial.

1.3 A proposta

Escolhemos estudar grafos de disco unitários como proposta. A justificativa é que recentemente descobriu-se um esquema de aproximação para o Problema da cobertura de vértices em discos unitários com peso [26]. Definiremos este e apresentaremos uma breve visão de sua definição quando falarmos da proposta, no capítulo 4.

Considerarmos estudar o esquema de aproximação para o problema em questão devido a este utilizar programação dinâmica e tratar de problemas bastante abordados atualmente.

Nosso objetivo é implementar o PTAS descrito em [26] e compará-lo com soluções heurísticas adaptadas de [27], pois o problema é um pouco diferente das soluções abordadas na literatura.

1.4 Organização do documento

O restante do documento está organizado da seguinte maneira:

No capítulo 2 apresentamos as definições de algoritmos de aproximação, programação dinâmica (E exemplos de uso da técnica) e de problemas com soluções obtidas através da combinação entre as duas técnicas (Problema da mochila, empacotamento, caixeiro viajante euclidiano, etc).

No capítulo 3 apresentamos esquemas de aproximação que usam programação dinâmica como técnica. A base deste capítulo é fornecida com base no livro de Williamsom e

Shimoys [32].

No capítulo 4 apresentamos a proposta de continuidade do trabalho. O cronograma de atividades é apresentado na seção 4.2.

CAPÍTULO 2

DEFINIÇÕES

Neste capítulo apresentaremos a definição de algoritmos de aproximação (Definição 2.1), Esquemas de aproximação polinomial (Seção 2.2) e programação dinâmica (Seção 2.3).

Veremos também ao longo deste capítulo definições de problemas com soluções obtidas através de técnicas de PTAS e programação dinâmica.

Estas definições são baseadas em [32].

2.1 Algoritmos de aproximação

Definição 2.1.1. *Um algoritmo α-aproximação para um problema de otimização é um algoritmo de tempo polinomial que, para todas as instâncias do problema, produz uma solução α vezes a solução ótima.*

Para um algoritmo α-aproximação, chamamos de α a garantia de performance do algoritmo. Em [32] convenciona-se que em problemas de maximização o valor de α é inferior a 1 e em problemas de minimização, superior a 1.

O estudo dos algoritmos de aproximação é vinculado a diversos objetivos: Estudo de problema de otimização, estudo matemático rigoroso das heurísticas associadas às soluções de problemas, estudo das métricas de dificuldade de solução dos algoritmos, dentre ouras.

Muitos problemas de otimização abordados atualmente são problemas NP-difíceis. Veremos que a partir de algoritmos de aproximação, é possível encontrar soluções próximas do valor ótimo com custos de tempo parametrizados em relação à α-aproximação.

Associada à definição de algoritmos de aproximação, veremos uma classe interessante destes. Os esquemas de aproximação polinomial.

2.2 Esquemas de aproximação polinomial

Definição 2.2.1. *Um esquema de aproximação polinomial é uma família de algoritmos A_ϵ, onde, para cada A_ϵ é uma $(1+\epsilon)$ aproximação para problemas de minimização e $(1-\epsilon)$ para problemas de maximização.*

A definição de esquema de aproximação polinomial deriva do inglês *Polynomial-time Approximation Scheme* (PTAS). Por simplicidade, adotaremos esta última sigla no restante deste documento.

Muitos problemas possuem PTAS, como o problema da mochila e do Caixeiro Viajante Euclidiano. Entretanto, alguns problemas são tão difícieis que não há PTAS salvo se $P = NP$ [1].

Veremos PTAS referentes a alguns problemas, os quais usam a técnica de programação dinâmica.

2.3 Programação dinâmica

Programação dinâmica é uma técnica de otimização que transforma um problema complexo em uma sequência de sub-problemas [7]. Sua principal característica é a natureza multi-estágio de um procedimento de otimização.

Com programação dinâmica, o que em geral queremos fazer é dividir um problema em um número razoável de subproblemas, onde razoável pode ser, por exemplo, $O(n^2)$ o tamanho da instância. A diferença entre esta abordagem e outra por divisão e conquista é a possibilidade de sobreposição entre estes subproblemas [6].

Apresentaremos programação dinâmica usando-a para resolver o problema da subsequência crescente máxima, extraído de [12].

2.3.1 O problema da subsequência crescente máxima

Ilustraremos a técnica de programação dinâmica apresentando um algoritmo que resolve o problema da subsequência crescente máxima.

Suponha que o vetor $A[1 \ldots n]$ é uma sequência de números naturais. Uma subsequência de $B[1 \ldots k]$ é o que sobra depois que um conjunto arbitrário de termos é apagado, ou seja, os elementos deste conjunto podem encontrar-se em qualquer posição dentro de A.

Se o vetor $B[1 \ldots k]$ é uma subsequência crescente de $A[1 \ldots n]$ então teremos índices $i_1, i_2, \ldots, i_k$ tais que $1 \leq i_1 < i2 < \ldots < i_k \leq n$ de forma que $B[1] = A[i_1], B[2] = A[i_2], \ldots, B[k] = A[i_k]$, além disso temos que:

$$A[i_1] \leq A[i_2] \leq \ldots \leq A[i_k] \tag{2.1}$$

Uma subsequência crescente de $A[1 \ldots n]$ é máxima se não existe nenhuma outra mais longa.

Definimos então o problema da subsequência crescente máxima.

Definição 2.3.2 (Problema da subsequência crescente máxima)**.** *Dado um vetor* $A[1 \ldots n]$*, determinar qual o tamanho de sua subsequência crescente máxima.*

O algoritmo $MaxSubsequenciaCrescente(A, n)$ resolve o problema da subsequência crescente máxima usando programação dinâmica.

Algoritmo 2.1: $MaxSubsequenciaCrescente(A, n)$

```
1  Entrada: A[1...n], vetor de números naturais e n, índice de um item arbitrário
                de A
2  para m ← 1 até n execute
3  |  c[m] ← 1
4  |  para i ← m − 1 decrescendo até 1 execute
5  |  |  se A[i] ≤ A[m] e c[i] + 1 > c[m] então
6  |  |  |  c[m] ← c[i] + 1
7  |  |  fim
8  |  fim
9  fim
10 retorna max(c[1...n])
```

Provaremos que $MaxSubsequenciaCrescente(A, n)$ resolve o subsequência crescente

máxima. Provaremos em seguida que este algoritmo executa em tempo $O(n^2)$.

Teorema 2.1. *Para um dado vetor $A[1 \dots n]$, $MaxSubsequenciaCrescente$ encontra o tamanho subsequência crescente máxima.*

Demonstração. O seguinte invariante de laço é válido:

- I_8 : $c[m]$ retorna o tamanho da subsequência crescente máxima para um dado valor de m.

$c[m]$ inicia em um, o valor mínimo para qualquer subsequência crescente. Para $m \leftarrow k$, sendo k um valor arbitrário $1 < k \leq n$, a linha 5 verifica se $A[i] \leq A[k]$, indicando que é possível formar uma sequência crescente contendo $A[i]$ e $A[k]$, e o valor desta sequência é igual ao valor da sequência até i mais um. Caso o valor de $c[i] + 1$, ou seja, da nova sequência a partir de i que inclui k seja superior à sequência de $c[k]$ escolhemos esta. Estamos acumulando assim em $A[k]$ o valor da maior subsequência crescente máxima de 1 até k. Consequentemente, conforme aumentamos o valor de k, vamos acumulando em estados, o valor da subsequência crescente máxima para o k-ésimo elemento em um estado $c[k]$. Uma vez armazenado o estado, basta procurar ao longo de $c[1 \dots n]$ aquele com maior valor e então encontramos o valor da subsequência crescente máxima.

□

Teorema 2.2. *$MaxSubsequenciaCrescente$ executa em tempo $O(n^2)$, onde n é o tamanho o vetor A.*

Demonstração. A tabela 2.1 apresenta as ordens de execução de cada linha (Seguindo a notação apresentada em [11]). Observamos que as linhas de maior ordem estão na ordem de n^2. Somando os graus de cada linha (Como apresentado em [21]) obtemos um algoritmo $O(n^2)$.

□

O algoritmo 2.1 é um exemplo de uso da técnica de programação dinâmica. Para a instância do problema instancia-se uma matriz de estados, neste caso $c[1 \dots m]$. Cada

Tabela 2.1: Número de execuções por linha do algoritmo 2.1

Linha	Número de execuções
2	$O(n)$
3	$O(n)$
4	$O(n^2)$
5	$O(n^2)$
6	$O(n^2)$
7	$O(n^2)$
8	$O(n^2)$
9	$O(n)$
10	$O(n)$

estado $c[k]$ representa o maior caminho traçado entre $A[1] \rightarrow A[k]$. A solução para $A[1 \ldots n]$ é gerada sobrepondo as soluções de todos os subvetores $A[1 \ldots k]$, para $1 \leq k < n$.

2.3.3 Formalização de programação dinâmica

Após analisar o exemplo de um algoritmo que usa a técnica, apresentaremos uma definição formal da técnica, baseada em [7].

Observamos as seguintes características no algoritmo apresentado (Assim como nos demais que utilizam programação dinâmica):

- Estágios : Os problemas são divididos em múltiplos estágios, resolvidos sequencialmente, um por vez.

- Estados : Associado com cada estágio, há os *estados* do processo, os quais fornecem informação necessária para saber quais as consequências de uma determinada decisão em ações futuras. No problema da subsequência crescente máxima os estados eram representados pelo vetor $c[m]$, que continha a maior subsequência crescente máxima para o vetor formado entre 1 e m.

- Otimização recursiva: Uso de otimização recusiva, chegando na solução para o problema de n estágios resolvendo primeiro um problema de um estágio e sequencialmente incluindo um estágio por vez e resolvendo-os até que o valor ótimo seja encontrado.

A partir das ideias acima, apresentamos a equação de *Bellmann*, falando apenas dela para problemas de maximização [23]:

$$v(x_t) = \sup_{a_t \in \Gamma(x_t)} F(x_t, a_t) + \delta v(x_{t+1}), t \in [0, n] \tag{2.2}$$

Onde:

- x_t é o vetor de estados em um estágio (ou período de tempo) t.
- $F(x_t, a_t)$ é o ganho para ir do estágio t até o estágio $t+1$ a partir de x_t tomando a ação a_t.
- $\Gamma(x_t)$ é o conjunto de todos as ações possíveis de escolher a partir do estado x_t.
- δ é o fator de desconto, tipicamente usado na análise de problemas de finanças, onde há depreciação ($\delta \leq 1$).
- sup é a função supremo.

A função $v(x_t)$ retorna o valor ótimo do problema até o estágio t.

A equação de *Bellmann*, é também chamada de *Equação da programação dinâmica*. Ela é uma equação funcional [23], pois resolvê-la envolve descobrir a função $v(x_t)$.

Estas equações funcionais podem ser resolvidas de diversas maneiras, como métodos analíticos usando o teorema do envelope (Como o exemplo dado em [23]), método dos coeficientes determinados ou analisando a solução numericamente em um computador.

No caso do problema da subsequência máxima, resolvemos o problema da equação de *Bellmann* pelo método numérico, avaliando cada estágio de forma recursiva.

Neste trabalho trataremos apenas de métodos numéricos e excluiremos casos onde a dimensionalidade dos estados pode inviabilizar uma solução. Para estes últimos existe um outro ramo de estudo, chamado de programação dinâmica aproximada, onde as equações de *Bellmann* são aproximadas com o auxílio de algumas técnicas [5].

2.4 Definições de problemas com esquemas de aproximação usando programação dinâmica

Os problemas a seguir possuem PTAS conhecido e que usam programação dinâmica. Fornecemos nesta seção suas definições. No próximo capítulo iremos descrever estes algoritmos.

O leitor mais atento pode-se perguntar como encontraremos soluções aproximadas para problemas usando programação dinâmica, se esta é aplicada em problemas para encontrar a solução ótima.

Veremos no capítulo 3 que todos estes algoritmos possuem uma característica em comum: A partir de uma instância original do problema I eles definem uma instância modificada I' com custo polinomial e em I' aplicam o algoritmo de programação dinâmica.

Ademais, sendo A um algoritmo de programação dinâmica para a instância modificada I', para todos estes problemas é válido que a solução obtida aplicando-se A em I', ou seja $A(I')$, encontraremos uma solução para a instância original I a um custo não superior a $(1+\epsilon)OPT(I)$ para problemas de minimização e $(1-\epsilon)OPT(I)$ para problemas de maximização, sendo $OPT(I)$ a solução ótima para a instância I.

Apresentamos no restante desta seção a definição de quatro problemas que estudaremos no próximo capítulo detalhadamente.

2.4.1 O problema da mochila

Um viajante depara-se com um um tesouro. Contudo, sua mochila não é grande o suficiente para levar todos os seus itens. Seu objetivo é escolher dentre todos os itens qual o subconjunto capaz de maximizar seus ganhos.

Definição 2.4.2 (Problema da mochila)**.** *Dado um conjunto de itens $I = 1 \dots n$, onde cada item i tem valor v_i e tamanho s_i. Todos os valores são inteiros positivos. A mochila possui capacidade B e o objetivo é encontrar $S \subseteq I$ tal que $\sum_{i \in S} s_i \leq B$ maximizando $\sum_{i \in S} v_i$.*

Veremos como resolver o problema da mochila com auxílio de um PTAS na seção 3.1.

2.4.3 Escalonamento de tarefas em máquinas paralelas idênticas

Precisamos executar n tarefas em m máquinas idênticas. Cada máquina executa paralelamente a outra e só pode executar uma tarefa por vez. Qual o menor tempo de execução?

Definição 2.4.4 (Problema do escalonamento de tarefas em máquinas paralelas)**.** *Dado um conjunto N tarefas de tamanho n e M máquinas idênticas de tamanho m, onde para cada tarefa $t \in N$ necessita de p_t unidades de tempo e termina de executar em c_t unidades do início da execução, determinar qual ordem de atribuição de tarefas por máquina de forma a minimizar $c_{max} = \max_{t=1...n} c_t$.*

Veremos um esquema de aproximação para o problema do escalonamento na seção 3.2.

2.4.5 Problema do empacotamento

São dadas n peças (ou itens) de tamanhos $1 > a_1 \geq a_2 \geq \ldots \geq a_n > 0$ ordenados de forma decrescente. Desejamos colocá-las no menor número possível de caixas de tamanho um.

Definição 2.4.6 (Problema do empacotamento)**.** *Seja um conjunto de itens $1 > a_1 \geq a_2 \geq \ldots \geq a_n > 0$. Desejamos empacotá-los em um conjunto caixas C, composto por c caixas de tamanho 1 de forma a minimizar C.*

Veremos um PTAS para o problema do empacotamento na seção 3.3

2.4.7 O caixeiro viajante euclidiano

O caixeiro viajante euclidiano é uma subclasse do problema do caixeiro viajante, onde as distâncias entre os nós respeitam as distâncias do plano euclidiano. Definimos abaixo o problema.

Definição 2.4.8 (Problema do Caixeiro viajante euclidiano)**.** *Dado um conjunto N composto de n pontos no espaço $\mathbb{R}^2$, onde $\forall x, y \in N, x \neq y$ com $x = (x_1, x_2), y = (y_1, y_2)$, e $d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$, encontrar o menor caminho que passe por todos os pontos.*

2.5 Considerações sobre o capítulo

Vimos neste capítulo as principais definições necessárias para o desenvolvimento deste trabalho.

Baseamos nossas definições de algoritmos de aproximação no trabalho de Williamson [32]. Vimos que os esquemas de aproximação polinomial são um caso particular das α-aproximações.

Nossa definição de programação dinâmica é adaptada de notas de aula de David Laibson [23] e do trabalho de Dmitri Bertsekas [5]. O primeiro é resultado de pesquisas em teoria econômica e o segundo em controle ótimo e programação dinâmica aproximada.

O problema da subsequência crescente máxima foi extraído das notas de aula de Paulo Feofiloff [12]. A análise por invariantes de laço foi adaptada de André Vignatti [30].

As definições dos problemas que estudaremos são baseados no livro de Williamson [32] e do trabalho de Sanjeev Arora [3].

CAPÍTULO 3

ESQUEMAS DE APROXIMAÇÃO E PROGRAMAÇÃO DINÂMICA

Veremos neste capítulo quatro algoritmos de aproximação que utilizam a técnica de programação dinâmica. Estes algoritmos resolvem os problemas da seção 2.4.

Algo em comum em todos estes algoritmos é o uso da técnica de programação dinâmica. Em todas as soluções que aqui falaremos, a partir da instância original I, criaremos uma instância modificada I' aplicando nesta um algoritmo de programação dinâmica para calcular o ótimo para I'.

A figura 3.1 esquematiza uma representação do que estudaremos nestes quatro problemas levando em conta problemas de minimização. Deixamos a seta que liga a solução para a instância do problema tracejada para demonstrar que o caminho para obtê-la é mediante ao esquema de aproximação.

Todos os esquemas de aproximação que veremos possuem três fases: Transformação da instância original para a instância modificada I' por um algoritmo que chamaramos de Arr (Arredondamento), algoritmo de programação dinâmica para encontrar a solução ótima para $A(I') = OPT(I')$ e transformação da solução $OPT(I')$ para $(1 + \epsilon)$. Cada uma das demonstrações que veremos abaixo seguirá estes três passos.

Em todos estes algoritmos, a transformação da solução encontrada para a instância modificada e a instância original é direta. Discutiremos este processo detalhadamente conforme analisarmos cada problema.

Extraímos as demonstrações do livro de Williamson e Shmoys [32].

3.1 O problema da mochila

Vimos na seção 2.4.1 a definição para o problema da mochila. Exploraremos uma solução peculiar para o problema, dada pelo algoritmo 3.1.

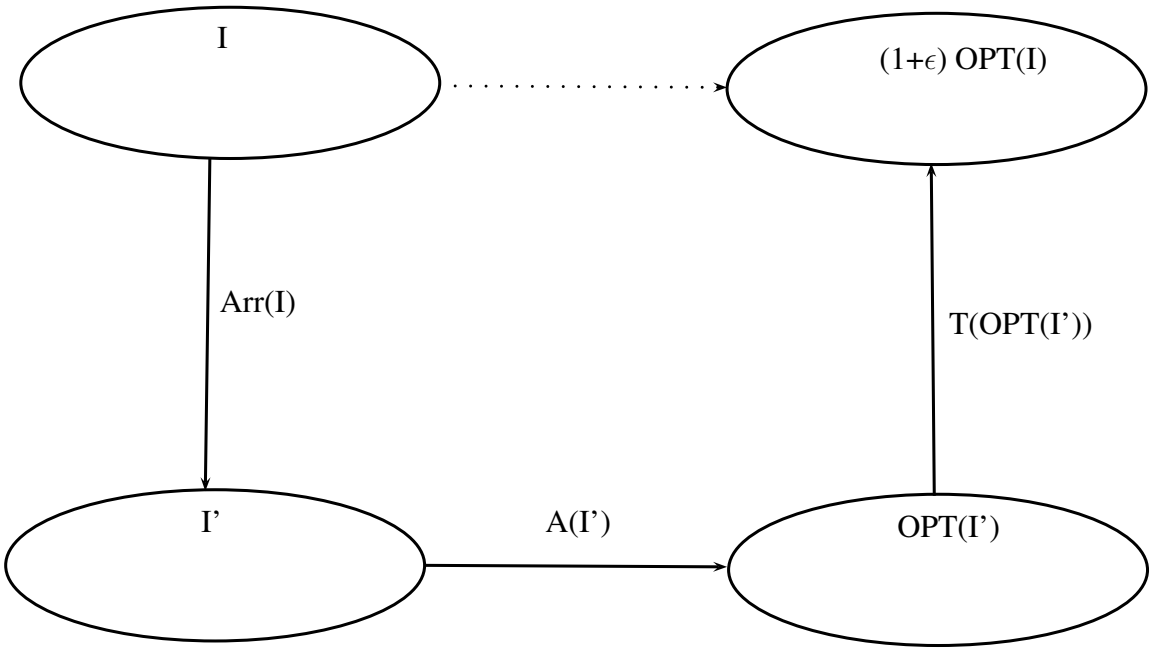

Figura 3.1: Todos os problemas que estudaremos possuem uma característica em comum: As instância original I é alterada para uma instância em I' por um algoritmo que chamamos de Arr (Arredondamento). Em I' aplicamos um algoritmo usando programação dinâmica para achar o caso ótimo. Finalmente, transformamos a solução encontrada em I' para uma solução para I a qual provaremos ser $(1+\epsilon)$ para problemas de minimização e $(1-\epsilon)$ para problemas de maximização.

Algoritmo 3.1: *Mochila*

Entrada: $I = \{1 \dots n\}$, vetor de itens. $V = \{v_1, \dots, v_n\}$, vetor de valores em I
$S = \{s_1, \dots, s_n\}$, vetor de tamanhos em I; B, tamanho da mochila
$A[1] \leftarrow \{(0,0), (s_1, v_1)\}$
para $j = 2$ *até* n **execute**
  $A[j] \leftarrow A[j-1]$
  para cada $(t, w) \in A[j-1]$ **execute**
    se $t + s_j \leq B$ **então**
      Adicione $(t + s_j, w + v_j)$ em $A[j]$
    fim
    Remova os pares dominantes de $A[j]$
  fim
fim
retorna $max_{(t,w)\in A[n]} w$

O algoritmo 3.1 usa programação dinâmica para resolver o problema da mochila. Ele é baseado nos seguintes passos: Instanciamos um vetor $A[1 \dots n]$ contendo uma lista de pares (t, w), a qual representa espaço $t \leq B$ e possui valor w. Ou seja, existe um conjunto $S \subseteq \{1, \dots, \}$ tal que $\sum_{i \in S} s_i = t \leq B$ e $\sum_{i \in S} v_i = w$. Cada elemento do vetor A é outro vetor $A[j]$, o qual não contém todos os pares possíveis até o elemento j, apenas os mais eficientes. Para fazer isto, introduzimos o conceito de pares dominantes.

Para um par (t, w) pertencente a $A[j]$, com $1 \leq j \leq n$, dizemos que este domina outro par (t', w') pertencente à $A[j]$ se $t \leq t'$ e $w \geq w'$. Ou seja, (t, w) ocupa igual ou menos espaço e valor no mínimo igual. Para $A[j]$ removeremos os pares dominados, ou seja, manteremos aqueles os quais esta propriedade não se aplique para todos os elementos de $A[j]$.

Dizemos que o vetor representado por $A[j]$ contém apenas pares dominantes, e assume a forma $(t_1, w_1), \dots, (t_k, w_k)$, com $t_1 < t_2 < \dots < t_k$ e $w_1 < w_2 < \dots < w_k$. Ou seja, o vetor $A[j]$ é ordenado em ordem crescente de peso.

Definimos V como a soma de todos os valores dos itens de uma instância do problema $V = \sum_{i=1}^{n} v_i$. Então V é o maior valor possível para o problema, e $A[j]$ pode conter no máximo $V + 1$ pares na lista (Incluímos também o par $(0, 0)$, pois assumimos na definição que os pesos e tamanhos são inteiros (A ideia de pares dominantes assegura um único par contendo o mesmo peso na lista). Pelo mesmo raciocínio, não podem existir mais de $B + 1$ pares em $A[j]$.

O algoritmo 3.1 executa os seguintes passos: Instancia o vetor $A[j]$, sendo seu primeiro elemento $A[1] \leftarrow \{(0, 0), (s_1, v_1)\}$. A partir do segundo elemento até n, o algoritmo atribui $A[j] \leftarrow A[j - 1]$ e para cada par (t, w) em $A[j]$ inclui um novo par $(t + s_j, w + v_j)$ se $t + s_j \leq B$. Finalmente o algoritmo remove os pares dominantes da lista.

A remoção dos pares dominantes pode ser feita em tempo na ordem do tamanho da lista. Mostraremos a partir do lema abaixo.

Lema 3.1 (Remoção dos pares dominantes)**.** *A remoção dos pares dominantes pode ser executada em tempo* $O(\min\{V, B\})$.

Demonstração. Suponha que estamos no início da linha 9 do algoritmo, e que $A[j]$ esteja

ordenada pelo índice t e contenha ainda os pares dominantes e dominados. Mostraremos que basta percorrer um número $O(\big|A[j]\big|) = O(\min\{V, B\})$ para remover todos os pares dominados. Suponha dois pares (t, w) e (t', w') de forma que (t, w) domina (t', w'). Precisamos avaliar dois casos, onde $t = t'$ e $t \neq t'$. No primeiro caso, sabemos que $w \leq w'$ e além disso t e t' são adjacentes em $A[j]$. Logo percorrendo a lista uma vez e comparando os pares adjacentes é suficiente para remover (t', w'). Caso exista (t'', w'') dominado por (t, w) de forma que $t = t' = t''$ basta "travar"a execução em t após t' ser removido e repetir a remoção, sem aumentar a ordem de execução do algoritmo. Resta-nos mostrar casos onde $t' > t$. Se t' for adjacente a t na ordem de execução aplica-se o mesmo procedimento descrito para $t = t'$. Caso contrário, se t' não é adajcente a t, então existe um par $(\hat{t}, \hat{w})$, tal que $t < \hat{t} \leq t'$ tal que não há relação de dominância entre $(\hat{t}, \hat{w})$ e (t, w). Neste caso sabemos que $\hat{w} > w$, $w \leq w'$ e $\hat{t} \leq t'$. Logo $(\hat{t}, \hat{w})$ domina (t', w') e ambos são adjacentes.

Finalmente, para manter o vetor contido em $A[j]$ ordenado, basta formar duas listas, uma contendo $A[j-1]$ e outra contendo os elementos formados da adição de (s_j, v_j) em $A[j-1]$ e intercalá-las, o que pode-se ser feito na ordem de $\big|A[j]\big|$ operações.

Logo o custo para ordenar e eliminar os pares dominados é de $O(\big|A[j]\big|) = O(\min\{V, B\})$ de instruções, esta última equação sendo válida porque é impossível termos um número de itens maior a $V + 1$ ou $B + 1$ pares. □

Afirmamos que o algoritmo 3.1 encontra a solução ótima para o problema da mochila, através do teorema abaixo.

Teorema 3.1. *O algoritmo 3.1 calcula a solução ótima para o problema da mochila.*

Demonstração. Provamos por indução que em j, $A[j]$ contém somente (e todos) os pares dominantes para todos os estados até j. O caso base $A[1] \leftarrow \{(0, 0), (s_1, v_1)\}$ é válido. Agora suponha que seja válido para $A[j-1]$. Sendo $S \subseteq \{1, \ldots, j\}$, e $t = \sum_{i \in S} s_i \leq B$ e $w = \sum_{i \in S} v_i$. Afirmamos que existe um par $(t', w') \in A[j]$ que domina o par (t, w).

Avaliaremos em dois casos, no primeiro $j \notin S$. Neste caso, $A[j]$ conterá apenas (t', w'), pois (t, j) é obtido de $A[j-1]$ mas removido na linha 9 do algoritmo 3.1.

No segundo caso $j \in S$. Definimos $S' = S - j$, o qual por hipótese de indução, há algum

$(\hat{t}, \hat{w}) \in A[j-1]$ que domina $\left(\sum_{i \in S'} s_i, \sum_{i \in S'} v_i\right)$, então $\hat{t} \leq \sum_{i \in S'} s_i$ e $\hat{w} \geq \sum_{i \in S'} v_i$. Então o algoritmo adicionará o par $\left(\hat{t} + s_j, \hat{w} + v_j\right)$ para $A[j]$, pois $\hat{t} + s_j \leq t \leq B$ e $\hat{w} + v_j \geq w$. Assim, se um par (t', w') domina um par (t, w) em $j-1$, este continuará dominando em j e a remoção de pares dominados em $j-1$ não afeta a solução ótima.

□

A partir do 3.1 vemos que o algoritmo consome tempo $O(n \min\{V, B\})$, pois gasta n passos para formar o vetor A e mais $O(n \min\{V, B\})$ para eliminar os pares dominados.

O algoritmo 3.1 não é um algoritmo polinomial. Rotineiramente codificamos o tamanho da mochila em base binária. Dobrar o tamanho da mochila neste caso aumenta em $\log_2 2 = 1$ o tamanho da instância, enquanto o número de passos na remoção de pares dominantes aumenta $O(B) = 2$. Aumentando em 2^{10} o valor da entrada aumentamos em 10 seu tamanho e em 1024 o número de passos executados para eliminar os pares dominantes.

Entretanto, se a entrada do algoritmo 3.1 for codificada em unário, ele executa em tempo polinomial em relação ao seu tamanho. Dizemos que este é então um algoritmo *pseudo-polinomial* [32].

Apresentamos uma maneira de obter uma $(1 - \epsilon)$-aproximação para o problema da mochila a partir de arredondamento utilizando o algoritmo 3.1.

Para fazê-lo, iremos arredondar a instância em inteiros múltiplos de uma grandeza que chamaremos de μ e converter cada valor v_i para o múltiplo mais próximo de μ. Mais precisamente, definimos uma instância modificada I', onde os valores dos itens não são mais $v_i, \ldots, v_n$, mas $v'_i, \ldots, v'_n$ e $v'_i = \lfloor v_i/\mu \rfloor$, para cada item i. Executamos então o algoritmo 3.1 usando I'. Mostraremos que se fizermos esta alteração poderemos calcular a aproximação em tempo polinomial.

Esta alteração para μ não provoca alterações muito grandes no problema original e que além disso, conseguiremos executar o algoritmo em tempo polinomial. Faremos primeiro uma estimativa: Se usarmos valores $\tilde{v_i} = v'_i\mu$ ao invés de v_i, a imprecisão em cada valor v_i seria de no máximo μ, logo cada solução factível sofreria uma mudança de no máximo μ unidades.

Desejamos que o erro seja de no máximo ϵ vezes o limitante inferior da solução do problema, o qual é o valor do maior item da mochila (A solução mais simples consiste em escolher o item de maior valor). Sendo assim, definimos M como sendo o tamanho do item de maior valor:

$$M = \max_{i \in I} v_i \tag{3.1}$$

Assim, se fizermos o erro $n\mu$ da solução igual ao erro do limitante inferior, teremos:

$$n\mu = \epsilon M \tag{3.2}$$

E para gerar I' dividiremos cada valor v_i por μ, com μ sendo:

$$\mu = \epsilon M/n \tag{3.3}$$

Com os valores divididos por μ, o vetor de valores $V' = \sum_{i=1}^{n} v'_i$ fica sendo:

$$V' = \sum_{i=1}^{n} v'_i = \sum_{i=1}^{n} \left\lfloor \frac{v_i}{\epsilon M/n} \right\rfloor \tag{3.4}$$

O último termo desta soma é na verdade, uma soma de progressão aritmética dividida por um fator ϵ. Esta soma tem uma característica de seu resultado ser $O(n^2)$. Sabemos então que $V' = O(n^2/\epsilon)$. Observando a ordem do algoritmo 3.1, se executarmos este algoritmo com esta aproximação, o tempo de execução será de:

$$O(n \min\{V, B\}) = O(n^3/\epsilon) \tag{3.5}$$

Logo o arredondamento proporcionou executar o algoritmo em tempo polinomial.

O algoritmo 3.2 converte a instância I em I' e calcula a $(1 - \epsilon)$ aproximação.

Algoritmo 3.2: AproximaçãoMochila

Entrada: $I = \{1 \dots n\}$, vetor de itens. $V = \{v_1, \dots, v_n\}$, vetor de valores em I
$S = \{s_1, \dots, s_n\}$, vetor de tamanhos em I

$M \leftarrow \max_{i \in I} v_i$

$\mu \leftarrow \epsilon M / n$

$v'_i \leftarrow \lfloor v_i / \mu \rfloor$ para todo $i \in I$

Execute o algoritmo 3.1 para a instância do problema da mochila com valores v'_i.

Finalmente nos resta mostrar que o algoritmo 3.2 calcula uma $(1 - \epsilon)$-aproximação para o problema da mochila.

Teorema 3.2. *O algoritmo 3.2 é um esquema de aproximação polinomial para o problema da mochila*

Demonstração. Precisamos mostrar que o algoritmo retorna uma solução cujo valor é pelo menos $(1 - \epsilon)$ a solução ótima. Seja S o conjunto de itens retornados pelo algoritmo e O o conjunto de itens da solução ótima. Sabemos que $M \leq OPT$, ou seja, pois qualquer solução envolve escolher um ou mais itens (Neste caso estamos escolhendo o mais valioso). Além disso, pela definição de v'_i, $\mu v'_i \leq v_i \leq \mu(v'_i + 1)$, além de $\mu v'_i \geq v_i - \mu$. Aplicando as definições de arredondamento, além de que S é a solução ótima para a instância modificada I', chegamos nas equações abaixo:

$$\begin{aligned}
\sum_{i \in S} v_i &\geq \mu \sum_{i \in S} v'_i \\
&\geq \mu \sum_{i \in O} v'_i \\
&\geq \sum_{i \in O} v_i - |O| \, \mu \\
&\geq \sum_{i \in O} v_i - n\mu \\
&= \sum_{i \in O} v_i - \epsilon M \\
&\geq OPT - \epsilon OPT = (1 - \epsilon) OPT
\end{aligned}$$

□

3.2 Escalonamento de tarefas em máquinas paralelas idênticas

Achamos conveniente falar do problema em duas partes. Na primeira, apresentaremos duas α-aproximações (Seção 3.2.1). Em seguida, falamos de um esquema de aproximação combinado a programação dinâmica para resolvê-lo (Seção 3.2.2).

Falamos brevemente das definições que usaremos no restante desta seção. Temos como entrada para o problema n tarefas e m máquinas idênticas. Cada tarefa $j = 1, \ldots, n$ executa em tempo p_j em apenas uma máquina sem interrupção. Cada máquina executa apenas uma tarefa por vez.

O objetivo é completar todas as tarefas o mais rápido possível. Definiremos como c_j o tempo no qual a tarefa termina (Presumindo que o escalonamento inicia-se no tempo 0). Queremos então encontrar $c_{\max} = \max_{j=1,\ldots,n} c_j$, o qual chamaremos de *largura* do escalonamento de tarefas.

Para uma dada instância I do problema, a solução ótima, a qual rotineiramente chamamos de $OPT(I)$ será neste problema conhecida como $c^*_{\max}$.

3.2.1 Duas α-aproximações para o problema

Veremos dois algoritmos de aproximação apresentados no livro de Williamson [32].

O primeiro é conhecido como algoritmo de busca local.

Algoritmo 3.3: Busca Local

1 **Entrada**: $J = \{1, \ldots, n\}$, vetor de tarefas. $M = \{1, \ldots, n\}$, vetor de máquinas e P $= \{p_1, \ldots, p_n\}$, vetor de tempos de processamento de cada tarefa

2 Defina $ME \leftarrow Matriz_{M \times J}$

3 Inicie ME com qualquer sequência aleatória de tarefas por máquina

4 **enquanto** $mv \leftarrow MaquinaVazia(ME, c_l - p_l)$ **faça**

5 | TrocaTarefa(ME, l, mv)

6 **fim**

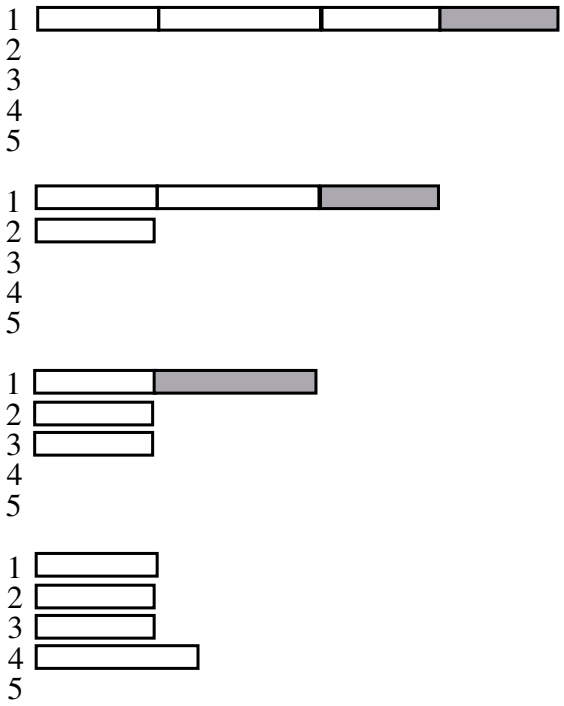

Figura 3.2: Exemplo de execução do algoritmo 3.3 supondo que o algoritmo alocou inicialmente todas as tarefas na primeira máquina.

O algoritmo 3.3 apresenta o pseudocódigo para este algoritmo de busca local. O princípio do mesmo é instanciar uma matriz de máquinas por tarefas, um escalonamento qualquer e depois, verificar se para a última tarefa há alguma máquina a qual pode completá-la antes (Basta verificar se a máquina está disponível para processar tarefas em tempo $c_l - p_l$) e então transferir a última tarefa a completar para esta máquina.

A figura 3.2 exemplifica uma execução do algoritmo.

O algoritmo executa $O(n)$ passos (Assumimos que neste caso, m é constante). Vamos agora mostrar que ele é uma 2-aproximação para o problema do escalonamento.

Vamos fornecer alguns limites para o problema. Sabemos que o tempo c^*_{max} é no mínimo o tempo da tarefa mais longa, já que todas as tarefas devem ser processadas:

$$c^*_{max} \geq \max_{j=1,\ldots,n} p_j \tag{3.6}$$

Além disso, precisamos executar $P = \sum_{j=1}^{n} p_j$ unidades de processamento para completar a execução de tarefas. Em média, uma máquina deve processar então aproximadamente P/m unidades de processamento. Sabemos que se P não for múltiplo do número

de máquinas alguma máquina executará um pouco acima da média, de forma que o tempo de execução para o caso ótimo é no mínimo:

$$c^*_{max} \geq \sum_{j=1}^{n} p_j/m \tag{3.7}$$

Para analisar o tempo do algoritmo de busca local, seja l a última tarefa a executar. O horário em que esta tarefa termina é denotado por c_l, que é o mesmo que c_max, ou tempo objetivo. Sabemos que se o algoritmo terminou com esta tarefa em c_l é porque nenhuma outra máquina estava disponível para completá-la antes. Denotamos por S_l o tempo de início da tarefa l, $S_l = C_l - p_l$. Podemos particionar esta execução em dois períodos, de 0 até S_l e p_l. Sabemos pela equação 3.6 que $p_l \leq c^*_{\max}$. Quanto ao período antes da última tarefa executar, as máquinas realizaram no máximo mS_l unidades de trabalho, o que não pode ser superior ao total de trabalho a ser feito, caso contrário a tarefa l já teria acabado. Sendo assim:

$$S_l \leq \sum_{j=1}^{n} p_j/m \tag{3.8}$$

Combinando 3.7 com 3.8 observamos que $S_l \leq c^*_{max}$. Sendo assim, a largura, ou tempo máximo de execução para este algoritmo é de $2c^*_{max}$.

Encontraremos um resultado ligeiramente melhor no teorema abaixo.

Teorema 3.3. *O algoritmo 3.3 é uma 2-aproximação*

Demonstração. Na verdade, com o mesmo raciocínio da equação 3.7 chegamos na verdade a um limite um pouco menor para S_l:

$$S_l \leq \sum_{j \neq l} p_j/m \tag{3.9}$$

Já que alguma máquina pode executar além da média e o tempo de S_l ser maior do que a média. E assim o tempo total de execução passa a ser:

$$p_l + \sum_{j \neq l} p_j/m = \left(1 - \frac{1}{m}\right) p_l + \sum_{j=1}^{n} p_j/m \tag{3.10}$$

Aplicando os limites das equações anteriores, chegamos a:

$$p_l + \sum_{j \neq l} p_j/m \leq \left(2 - \frac{1}{m}\right) c^*_{max} \tag{3.11}$$

E então melhoramos um pouco nossa 2-aproximação, cuja melhoria é mais significativa com um número menor de máquinas.

□

Apresentaremos agora um algoritmo guloso para resolver o problema. Uma parte deste algoritmo será aproveitada na próxima seção quando falarmos da aproximação que buscamos.

Chamaremos este algoritmo de *escalonamento por listas*. Dada uma lista de tarefas, arbitrariamente ordenada, este algoritmo seleciona a máquina com menor carga e atribui à ela a tarefa no topo da lista e assim sucessivamente. Não apresentaremos o pseudocódigo deste algoritmo.

O teorema abaixo é válido:

Teorema 3.4. *O algoritmo de escalonamento por listas é uma 2-aproximação para o problema*

Demonstração. Dada a saída do algoritmo de escalonamento por listas, fornecemos esta como entrada para o algoritmo de busca local. Este último, ao executar, não alterará a ordem de nenhuma tarefa, pois a última tarefa escalonada pelo algoritmo de listas foi já alocada pela máquina que poderia terminar no tempo mais recente. Sendo assim, a análise que fizemos para a última tarefa S_l também vale e assim chegamos à uma 2-aproximação. □

Na verdade, há um teorema que demonstra que este algoritmo de escalonamento é na verdade uma 4/3-aproximação para o problema do escalonamento. Não o demonstraremos aqui.

3.2.2 Esquema de aproximação polinomial para o problema do escalonamento

Veremos nesta seção dois esquemas de aproximação polinomial para o problema do escalonamento. O primeiro deles trata do número de máquinas como constante e o segundo, como entrada do problema.

Começamos com uma variação do problema onde o número m de máquinas é constante. Dado um número de entrada i, vamos definir uma família de algoritmos A_i e focar em especial em um algoritmo A_k.

Vamos dividir as tarefas em dois grupos: Longas e curtas. Uma tarefa p_l é curta se:

$$p_l \leq \frac{1}{km} \sum_{j=1}^{n} p_j \tag{3.12}$$

Caso contrário p_l é uma tarefa longa. Vale lembrar que o número de tarefas longas é menor que km, pois:

$$kmp_l > \sum_{j=1}^{n} p_j \tag{3.13}$$

Ou seja, com mais de km tarefas longas teríamos mais trabalho do que fornecido pela instância do problema.

O algoritmo A_k executará da seguinte forma: Dados k e m como constantes, encontraremos, enumerando todos os casos, o escalonamento ótimo para todas as tarefas longas. Depois, aplicaremos o algoritmo de escalonamento por listas nas tarefas curtas.

Algoritmo 3.4: A_k

1 **Entrada**: $J = \{1, \ldots, n\}$, vetor de tarefas. $M = \{1, \ldots, n\}$, vetor de máquinas e P $= \{p_1, \ldots, p_n\}$, vetor de tempos de processamento de cada tarefa

2 Defina $ME \leftarrow Matriz_{M \times J}$

3 Defina P_{long} com todas as tarefas onde $p_l > \sum_{j=1}^{n} p_j/km$

4 Defina P_{short} com todas as tarefas onde $p_l \leq \sum_{j=1}^{n} p_j/km$

5 Encontre o escalonamento ótimo para P_{long}

6 Execute o algoritmo 3.3 com os dados de entrada, matriz ME para P_short

O algoritmo 3.4 apresenta o pseudocódigo para o algoritmo A_k.

No algoritmo de aproximação constante, deduzimos a seguinte inequação para o tempo máximo de execução:

$$c_{max} \leq pl + \sum_{j \neq 1} p_j/m \tag{3.14}$$

Analisamos este algoritmo da mesma maneira que analisamos a última tarefa naquela ocasião. Nossa primeira hipótese é que esta seja uma tarefa curta. Neste caso, a equação 3.14, pois o algoritmo A_k usa o algoritmo de escalonamento por listas. Além disso, a condição $p_l \leq \frac{1}{km} \sum_{j=1}^{n} p_j$ é válida e podemos calcular:

$$c_{max} \leq \sum_{j=1}^{n} p_j/(mk) + \sum_{j \neq 1} p_j/m \leq \left(1 + \frac{1}{k}\right) \sum_{j=1}^{n} p_j/m \leq \left(1 + \frac{1}{k}\right) c^*_{max} \tag{3.15}$$

Se a última tarefa p_l for longa, então o algoritmo A_k calcula o escalonamento ótimo. Mostraremos que isto pode ser executado em tempo polinomial, desde que o número de máquinas seja constante.

Teorema 3.5. *A família de algoritmos A_i é um esquema de aproximação polinomial para o problema de minimização da largura de execução para um número constante de máquinas*

Demonstração. Como temos no máximo km tarefas longas (Vide equação 3.13) e para cada tarefa pode ser alocada ao longo de m máquinas, teremos m^{km} atribuições possíveis.

Se k e m são constantes, então o passo 5 do algoritmo 3.4 é executado em tempo $O(c)$, onde c é uma constante.

Se o escalonamento terminar com uma tarefa longa, a resposta é ótima. Caso contrário, ele será $\left(1+\frac{1}{k}\right) c^*_{max}$ e obtemos uma $(1+\epsilon)$-aproximação.

O sexto passo do algoritmo 3.4 foi analisado na seção anterior como sendo polinomial, logo A_i é um esquema de aproximação polinomial. □

Uma limitação da família de algoritmos A_i é o número de máquinas ser constante. Apresentaremos uma variação do problema para o caso onde o número de máquinas não é constante, e sim uma entrada para o problema.

Apresentamos uma família de algoritmos B_i. Analisaremos um algoritmo B_k pertencente à esta família. Vamos fixar uma largura T para a largura de execução de um determinado escalonamento, e B_k irá determinar se é impossível executar todas as tarefas em tempo T ou encontrará uma resposta com largura $\left(1+\frac{1}{k}\right) T$.

O limite inferior para T é:

$$T \geq \frac{1}{m} \sum_{j=1}^{n} p_j \tag{3.16}$$

Caso contrário nenhum escalonamento existiria.

Novamente dividiremos as tarefas em longas e curtas, sendo que o requisito fundamental é de que uma tarefa j é longa se $p_j > T/k$.

Arredondaremos os tempos de execução de todas as tarefas por seu múltiplo mais próximo de T/k^2.

O princípio de B_k é similar a A_k. Para um determinado valor de T, procuraremos pelo escalonamento das tarefas longas com número variável de máquinas em tempo inferior a T e se existir, executaremos o algoritmo de escalonamento por listas para incluir as tarefas curtas.

Novamente analisaremos a última tarefa. Primeiro consideraremos o caso onde ela é longa. Se não há escalonamento mais curto do que no tempo T, então o algoritmo decide que não há. Caso contrário, suponha S o conjunto de tarefas atribuídos para uma máquina

específica. Como cada tarefa na máquina é longa, $p_j \leq T/k$ então se há escalonamento em tempo inferior a $T, |S| \leq k$, caso contrário o tempo de escalonamento seria superior a T.

Além disso, arredondamos os tempos de execução em T/k^2 unidades. Graças a este arredondamento, o tempo real de uma tarefa longa na instância original I difere de no máximo, T/k^2 unidades. Sabendo que em S temos no máximo k tarefas, segue que:

$$\sum_{j \in S} p_j \leq T + k(T/k^2) = \left(1 + \frac{1}{k}\right) T. \tag{3.17}$$

E assim, se encontrarmos um escalonamento inferior a T na instância modificada I' (Originada do arredondamento), teremos este limite na instância original.

Agora consideraremos o caso onde uma tarefa pequena l é a última a ser executada. Sabemos que o algoritmo encontrou um escalonamento para as tarefas grandes inferiores a T (Caso contrário não chegaria ao passo das tarefas pequenas). Assim sabemos que:

$$\sum_{j=1}^{n} p_j/m \leq T \tag{3.18}$$

E $p_l < T/k$. Então a equação abaixo é válida:

$$p_l + \sum_{j \neq l} p_j < T/k + T = \left(1 + \frac{1}{k}\right) T. \tag{3.19}$$

Falta agora explicar como faremos para decidir se há escalonamento para as tarefas longas em tempo inferior a T. Usaremos programação dinâmica para este propósito.

Se há uma tarefa longa com tempo superior a T obviamente não há resposta. Caso contrário, podemos criar um vetor com k^2 dimensões onde a i-ésima dimensão representa o número de tarefas arredondadas de tamanho iT/k^2. Logo haverão no máximo $O(n^{k^2})$ entradas. O que é polinomial em relação ao tamanho da entrada.

Cada tarefa longa possui um tempo de processamento de no mínimo T/k. Então há no máximo k tarefas atribuídas para uma máquina. Estamos interessados entretanto, no conjunto de todas as combinações de tarefas que cabem em uma máquina em tempo

inferior a T.

Definimos uma sequência $(s_1, s_2, \ldots, s_{k^2})$ como sendo uma configuração de máquina se:

$$\sum_{i=1}^{k^2} s_i iT/k^2 \leq T \tag{3.20}$$

Ou seja, uma configuração de máquina representa uma combinação de tarefas longas arredondadas que conseguimos escalonar em uma máquina sem ultrapassar o tempo T.

Vamos chamar de $\mathcal{C}$ o conjunto de todas as configurações de máquina. Há no máximo $(k+1)^{k^2}$ configurações, pois precisamos considerar o caso de existirem 0 tarefas em uma máquina.

Calcularemos por programação dinâmica o número mímimo de máquinas para executar este número de tarefas. Denotamos por $OPT(n_1, n_2, \ldots, n_{k^2})$ o número mínimo de máquinas suficiente para escalonar esta entrada. Este valor segue esta relação de recorrência:

$$OPT(n_1, \ldots, n_{k^2}) = 1 + \min_{(s1,\ldots,s_{k^2})\in\mathcal{C}} OPT(n_1 - s_1, \ldots, n_{k^2} - s_{k^2}) \tag{3.21}$$

Vemos que esta última equação é uma equação de *Bellmann* tal como a equação 2.2. Seus termos são:

- $x_t = (n_1, \ldots, n_{k^2})$ é o vetor de estados em um estágio (ou período de tempo) t, neste caso é o número de tarefas ainda não atribuídas.
- $F(x_t, a_t) = 1$ é o ganho para ir do estágio t até o estágio $t+1$ a partir de x_t tomando a ação a_t.
- $\Gamma(x_t) = \mathcal{C}$ é o conjunto de todos as ações que podemos tomar no estado x_t, neste caso as configurações de máquina.
- $\delta = 1$ é o fator de desconto. Neste problema não há desconto.

A equação de *Bellmann* é calculada passo a passo, iterativamente. A solução é calculada da seguinte maneira: Cada estado ($OPT(n_1, \ldots, n_{k^2})$) representa um determinado

de tarefas ainda não escalonadas. Elas são calculadas partindo da entrada do problema, contendo todas as tarefas, subtraindo-se as configurações de máquina sucessivamente.

$(OPT(n_1, \ldots, n_{k^2}))$ retorna o menor número de máquinas necessário para escalonar todas as tarefas. Para implementar esta função basta subtrair sucessivamente de cada estado de entrada todas as configurações de máquina e o mínimo número de estágios até $(OPT(0, \ldots, 0))$, quando todas as tarefas forem escalonadas.

Este algoritmo é polinomial, pois teremos $O(n^{k^2})$ estados e $O(n(k+1)^{k^2})$ operações para resolver cada estado, pois executaremos no máximo n escalonamentos até atribuirmos todas as tarefas.

Vamos mostrar o procedimento para determinar se há um escalonamento T para as tarefas longas a partir de uma bisseção. Sabemos que o que o escalonamento ótimo encontra-se limitado no intervalo $[L_0, U_0]$, onde L_0 e U_0 são dados abaixo:

$$L_0 = \max \left\{ \left\lceil \sum_{j=1}^{n} p_j/m \right\rceil , \max_{j=1,\ldots,n} p_j \right\} \tag{3.22}$$

$$U_0 = \left\lceil \sum_{j=1}^{n} p_j/m \right\rceil + \max_{j=1,\ldots,n} p_j \tag{3.23}$$

Procuramos por T nos intervalos acima, com $T = \lfloor \frac{L+U}{2} \rfloor$. E executamos o algoritmo B_k, com $k = \lceil 1/\epsilon \rceil$. Caso o algoritmo encontre um escalonamento para o valor de T fazemos $U = T$. Caso contrário, fazemos $L = T + 1$ e a bisseção termina quando $U = L$ e o algoritmo retorna um escalonamento de tempo no máximo $(1 + \epsilon)U$.

Não demonstraremos que a bisseção é polinomial, embora seja simples de verificá-lo, pois a distância entre os seus limites superior e inferior é de no máximo a largura do processo mais lento. Como dividimos o intervalo por 2 a cada busca, chegamos à resposta na ordem logarítmica deste valor. Como tomamos o valor de T como sendo o piso, sabemos que L sempre será inferior a $c_{\max}*$ e então $(1 + \epsilon)L \leq (1 + \epsilon c^*_{max}$. Observamos que o algoritmo através da bisseção obtém uma $(1 + \epsilon)$-aproximação.

3.3 Problema do empacotamento

Vamos revisar as notações do problema do empacotamento. Como entrada do problema fornecemos um conjunto de n itens (ou peças) de tamanhos $a_1, a_2, \ldots, a_n$ tal que:

$$1 > a_1 \geq a_2 \geq \ldots \geq a_n > 0 \tag{3.24}$$

Queremos minimizar o número de *caixas* (Ou pacotes) necessários para guardar todos estes itens. Tendo cada caixa tamanho igual a um.

Vamos apresentar um teorema que fornece um limite à aproximação do problema do empacotamento:

Teorema 3.6. *A menos que $P = NP$ não pode existir uma α-aproximação para o problema do empacotamento para $\alpha < 3/2$.*

Demonstração. Há um problema de decisão relativo ao problema do empacotamento chamado de *problema da partição*. São dados n inteiros positivos $b_1, b_2, \ldots, b_n$ cuja soma é igual a $B = \sum_{i=1}^{n} b_i$, sendo que B é par. Queremos saber se é possível dividir os itens em dois conjuntos S e T tal que:

$$\sum_{i \in S} b_i = \sum_{i \in T} b_i \tag{3.25}$$

Sabemos que este problema é NP-completo. Afirmamos que é possível reduzir o problema da partição no problema do empacotamento.

Para isto, sendo B a soma de todos os itens no problema da partição, fazemos as entradas $a_1, \ldots, a_n$ do problema do empacotamento serem: $a_i = 2b_i/B$ e verificar se podemos empacotar todos os itens em 2 caixas.

Esta premissa é válida por que se há uma resposta para o problema da partição, então a soma de cada conjunto é igual a $B/2$. Ao multiplicarmos os itens b_i por $2/B$ na verdade convertemos uma partição exatamente em uma caixa de tamanho 1. Se há uma solução para o problema do empacotamento, então há uma solução para o problema da partição.

Logo um fator de aproximação abaixo de 3/2 nos permitiria chegar a solução ótima

do problema da partição, o que nos faria deduzir que $P = NP$.

□

Vamos mostrar entretanto, que mesmo que a aproximação por um fator menor que 3/2 não exista, é possível obter uma $\alpha + c$-aproxmação. Onde c é uma constante.

Vamos considerar um algoritmo guloso para preencher as caixas. Começamos com um conjunto de itens ordenados arbitrariamente. Para cada item deste conjunto, o algoritmo irá tentar inserí-lo no conjunto de caixas existentes a partir da primeira criada até a última. Caso nenhuma das caixas comporte-o ele criará uma nova caixa e irá acrescentar o item nesta.

O algoritmo 3.5 apresenta o pseudocódigo para este algoritmo guloso. A linha 4 do mesmo é percorrida em sequência, ou seja, as caixas são verificadas da primeira caixa criada até a última sucessivamente para verificar se alguma delas comporta o item a.

Algoritmo 3.5: Algoritmo de encaixe para o problema do empacotamento

```
1  Entrada: A = 1 > a_1 ≥ a_2 ≥ ... ≥ a_n > 0, conjunto de itens entre 0 e 1
2  Defina C como lista de caixas, C ← {}
3  para cada a ∈ A execute
4      se ∃c ∈ C que comporta A então
5          adicione a a c
6      senão
7          Crie novo c
8          Adicione a a c
9          Adicione c a C
10     fim
11 fim
```

Vamos analisar a performance do algoritmo 3.5. Se pegarmos a solução e parearmos a primeira caixa com a segunda, a terceira com a quarta, e assim por diante, a soma de cada par é maior ou igual a um. Isto acontece porque o algoritmo que apresentamos empacotou um item a_i arbitrário na caixa $n + 1$ porque ele não coube na caixa n.

Então se usamos l caixas, a soma dos tamanhos de todos os itens a_1, a_2, a_n, que deno-

taremos por TAM, de Tamanho, $TAM(I) = \sum_{i=1}^{n} a_i$ deve ser pelo menos $\left\lfloor \frac{l}{2} \right\rfloor$.

Sabemos também que a seguinte condição é válida:

$$OPT(I) \geq TAM(I) \tag{3.26}$$

Pois mesmo a solução ótima deve ter número de caixas maior ou igual ao tamanho da instância.

A resposta l do algoritmo de encaixe é então:

$$l \leq 2TAM(I) + 1 \leq 2OPT(I) + 1 \tag{3.27}$$

E então o algoritmo é uma α-aproximação acrescida de um fator constante.

Vamos apresentar um algoritmo de aproximação para o problema do empacotamento. Esta definição não se encaixa no que definimos de PTAS. Williamson em seu livro [32] chama-a de esquema de aproximação assintótico. Definiremos aqui da mesma maneira:

Definição 3.3.1 (Esquema de aproximação assintótico)**.** *Um esquema de aproximação assintótico é uma família de algoritmos* $\{ A_\epsilon \}$ *com uma constante* c *tal que para* $\epsilon > 0$*,* A_ϵ *retorna uma solução no máximo* $(1 + \epsilon)OPT + c$ *para problemas de minimização.*

Vamos apresentar um esquema de aproximação assintótico para o problema do empacotamento. Em especial vamos usar o mesmo conceito de programação dinâmica que usamos para resolver o problema do escalonamento de máquinas.

O princípio deste algoritmo é similar ao do problema do escalonamento. Vamos separar os itens em tamanhos grande e pequeno. Aplicaremos o algoritmo de programação dinâmica para os itens grandes e demonstraremos que a adição dos itens pequenos não aumenta muito o tamanho da solução.

Consideraremos uma peça como pequena se seu tamanho superar uma quantidade que chamaremos de γ. Também consideramos que o algoritmo de empacotamento das peças grandes resulta em l caixas.

O seguinte lema é válido:

Lema 3.2. *Um empacotamento de todas as peças de tamanho maior a um valor γ em l caixas pode ser estendido para um empacotamento de toda a entrada por* $\max\left\{l, \frac{1}{1-\gamma}TAM(I)+1\right\}$ *caixas.*

Demonstração. Suponha que um algoritmo tenha previamente empacotado itens de tamanho superior a γ em l caixas. Neste caso, iremos acrescentar agora um conjunto de itens de tamanho menor ou igual a γ. Suponha primeiro que este empacotamento de itens que chamaremos de pequenos caixa em todas as caixas já utilizadas Neste caso continuamos com l caixas.

Agora suponhamos que novas caixas foram utilizadas para empacotar os itens pequenos, resultando em $k+1$ caixas. Neste caso, cada caixa deve estar cheia em pelo menos $1-\gamma$, caso contrário caberiam mais itens nestas caixas. Sendo assim, a equação vale:

$$k(1-\gamma) \leq TAM(I) \tag{3.28}$$

Pois o tamanho deve ser pelo menos o quanto temos em cada uma das k primeiras caixas, pois não sabemos quanto pode ter sobrado para a última caixa. Resolvendo para k:

$$k \leq \frac{TAM(I)}{(1-\gamma)} \tag{3.29}$$

□

Vamos fazer $\gamma = \epsilon/2$, neste caso, segundo o lema 3.2, poderemos chegar a uma aproximação assintótica $(1+\epsilon)OPT(I)+1$):

$$\frac{TAM(I)}{(1-\gamma)} + 1 = \frac{TAM(I)}{(1-\epsilon/2)} + 1 \leq (1+\epsilon)TAM(I) + 1 \tag{3.30}$$

Finalmente se uma peça pequena tem tamanho $\epsilon/2$, então haverá no máximo $2/\epsilon$ peças pequenas por caixa. A partir da demonstração do lema 3.2 e da equação 3.30 assumimos que não é mais necessário discutir o problema para as peças pequenas.

Falaremos apenas das peças grandes agora. Consideraremos como I a uma instância

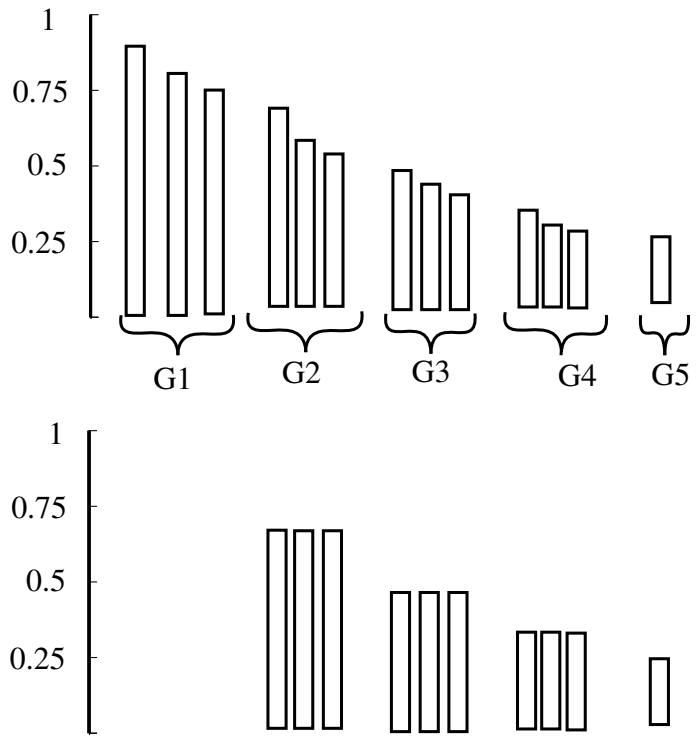

Figura 3.3: Agrupamento linear em instância do problema do empacotamento. Agrupamos os itens em grupos de k itens (Neste caso $k = 3$), eliminamos os itens do primeiro grupo e para cada grupo, alteramos os tamanhos dos itens menores para o tamanho do maior item. Esta imagem é baseada em uma obtida em [32]

do problema do empacotamento contendo apenas peças grandes. Vamos mostrar como reduzir o número de peças grandes a partir de uma instância I' e usar um algoritmo em I' para encontrar um empacotamento.

A instância I' é criada pelo que chamaremos de *agrupamento linear*. Na instância I ordenamos os itens em ordem decrescente e agrupamos os itens em grupos contendo k deles. O último grupo pode conter um número $h \leq k$ peças.

Para construir a instância I' eliminamos os itens do primeiro grupo e para os demais alteramos os tamanhos dos itens menores em cada grupo para o mesmo tamanho do maior item. Este processo é melhor visualizado na figura 3.3.

Provaremos um lema referente à instância I'.

Lema 3.3. *Seja I' a entrada obtida a partir da entrada I aplicando um agrupamento linear de tamanho k. Então:*

$$OPT(I') \leq OPT(I) \leq OPT(I') + k$$

E assim um empacotamento em I' resulta em um empacotamento em I com no máximo k caixas adicionais.

Demonstração. Vamos separar a prova por inequações. Sendo primeiro esta:

$$OPT(I') \leq OPT(I) \tag{3.31}$$

Suponha que tenhamos o empacotamento ótimo para I em mãos. Então, vamos formar um empacotamento $A(I')$ em I' a partir dele. Onde houverem itens pertencentes ao primeiro grupo de I trocaremos por itens do primeiro grupo de I'. Onde houverem itens do segundo grupo de I trocar pelo do segundo grupo de I e assim sucessivamente. Como há uma correspondência entre estes grupos e além disso, os grupo correspondente em I' tem peças menores ou iguais ao de I então o tamanho do empacotamento em I' é menor que $OPT(I)$. Não sabemos se este empacotamento é ótimo, mas ele é certamente maior ou igual ao ótimo, logo $OPT(I') \leq A(I') \leq OPT(I)$ e a inequação é válida.

A segunda inequação:

$$OPT(I) \leq OPT(I') + k \tag{3.32}$$

Vamos mostrá-la partindo da hipótese que temos a solução ótima em I', ou seja $OPT(I')$. Então, vamos construir uma solução $A(I)$ em I a partir dos seguintes passos. Para os itens do primeiro grupo de I', substitua-os pelos itens do segundo grupo em I, para os itens do segundo grupo em I' substitua-os pelos do terceiro grupo em I e assim por diante. Como forçamos o tamanho dos itens do grupo j em I' a serem do tamanho do maior item do grupo $j+1$ na instância I então obviamente a solução em I (Até o momento) é menor que em I', faltando apenas acrescentar os itens do primeiro grupo, que foram descartados de I'. Para estes, basta criar k caixas adicionais e adicionar cada um deles. Criamos então, a partir de $OPT(I')$, uma solução $A(I) \geq OPT(I)$. Esta inequação então vale:

$$A(I) \leq OPT(I') + k \tag{3.33}$$

Pois ao trocarmos o grupo j em I' pelo grupo $j+1$ em I na verdade reduzimos um pouco o tamanho de I. Os k itens adicionais são as caixas que criamos para os k itens do primeiro grupo. $A(I)$ é maior ou igual ao ótimo para I, sendo assim:

$$OPT(I) \leq A(I) \leq OPT(I') + k \tag{3.34}$$

E então a segunda inequação vale.

□

A transformação para a instância I' reduz o número de pedaços longos para n/k. Como I não tem pedaços pequenos, então:

$$TAM(I) \geq \epsilon n/2 \tag{3.35}$$

Fazemos então $k = \lfloor \epsilon TAM(I) \rfloor$. Além disso $\epsilon TAM(I) \geq 1$ caso contrário teríamos $(TAM(I))/(\epsilon/2) = (1/\epsilon)/(2/\epsilon) = 2/\epsilon^2$ peças, logo poderíamos resolver I sem a necessidade de uma instância modificada I'.

O número de peças na instância modificada I' é de n/k peças. Substituindo para os valores de k e n temos:

$$n/k \leq 2n/(\epsilon TAM(I)) \leq 4/\epsilon^2 \tag{3.36}$$

Onde assumimos n/k maior ou igual a um, então $n/k \leq 2n/k$, já que caso contrário, teremos menos de uma peça e a análise não fará sentido.

Para resolver a instância I' usamos o mesmo algoritmo de programação dinâmica que usamos para resolver o problema do escalonamento de máquinas.

Ou seja, vamos definir uma sequência $(s_1, s_2, \ldots, s_{n/k})$, onde $s_i \leq k, 1 \leq i \leq n/k$, pois podem haver k ou menos itens para cada tipo de configuração e um vetor auxiliar de tamanhos dos itens $(t_1, t_2, \ldots, t_{n/k})$, ou seja $0 < t_i < 1, 1 \leq i \leq n/k$. Logo ambas as

sequências tem tamanho polinomial, pois o tamanho da primeira lista é de no máximo $k^{n/k}$ e a segunda contém apenas n/k números reais entre 0 e 1.

Vamos definir a configuração de caixa $\mathcal{C}$ como o conjunto de todas as sequências $(s_1, s_2, \ldots, s_{n/k})$ tais que:

$$\sum_{i=1}^{n/k} s_i \cdot t_i \leq 1 \tag{3.37}$$

Então $\mathcal{C}$ contém todas as configurações de empacotamento possíveis em uma caixa. Agora é possível aplicar um algoritmo de programação dinâmica idêntico ao que usamos para o do escalonamento de máquinas, representado pela equação de *Bellmann* abaixo:

$$OPT(n_1, \ldots, n_i, \ldots, n_{n/k}) = 1 + \min_{(s1,\ldots,s_{n/k})\in\mathcal{C}} OPT(n_1 - s_1, \ldots, n_{n/k} - s_{n/k}) \tag{3.38}$$

Onde $(n_1, \ldots, n_i, \ldots, n_{n/k})$ é nosso vetor de estados contendo o número de peças $n_i \leq k, 1 \leq i \leq n/k$ para a instância modificada i.

Este algoritmo é idêntico ao que usamos para resolver o problema do escalonamento de tarefas em máquinas paralelas.

Finalmente vamos demonstrar que esta aproximação é uma $(1+\epsilon)+c$-aproximação.

Teorema 3.7. *Para algum $\epsilon > 0$, há um algoritmo que calcula a solução em tempo polinomial com no máximo $(1+\epsilon)OPT(I)+1$ caixas*

Demonstração. Vimos no lema 3.2 que o algoritmo usará $\max\left\{l, \frac{1}{1-\gamma}TAM(I)+1\right\}$ caixas, onde l é o número de caixas empacotadas pelo algoritmo designado para fazê-lo para as peças grandes.

Fazendo $\gamma = \epsilon/2$ o número de passos para as peças pequenas incluídas é de $(1+\epsilon)OPT(I)+1$.

Para as peças grandes o número de caixas é l. Vimos pelo lema 3.3 que o limite superior do algoritmo é $OPT(I') + k \leq OPT(I) + k$. Como fizemos $k = \lfloor \epsilon TAM(I) \rfloor$, então $\epsilon OPT(I) \leq k$ e então:

$l \leq OPT(I) + k \leq OPT(I) + \epsilon OPT(I) \leq (1+\epsilon)OPT(I)$

□

3.4 O caixeiro viajante euclidiano

Nesta seção veremos o problema do caixeiro viajante euclidiano. Não demonstraremos todo o problema nesta proposta de qualificação. A previsão de conclusão é para o final de julho de 2015.

A estratégia principal da prova é esta: A partir de uma instância do problema é possível criar uma instância reduzida do problema, e partir desta instância, dividí-la em quadrados com certa aleatoriedade nesta divisão Finalmente é possível demonstrar [32] que é possível encontrar com probabilidade $\frac{1}{2}$ uma rota que encontre $(1+\epsilon)OPT(I)$.

Mostraremos aqui a partir de uma instância I do problema como criar a instância I' (No lema 3.4) e enunciaremos os teoremas fundamentais para a prova do problema, os quais estamos atualmente estudando.

Lema 3.4. *Dado um esquema de aproximação polinomial para uma Instância modificada I' do Caixeiro Viajante Euclidiano, podemos obter um esquema de aproximação polinomial para qualquer instância do problema I*

Demonstração. Esta instância I' é uma instância que chamaremos de "boa". Veremos como criá-la a partir de uma instância qualquer I.

Seja L o tamanho do menor quadrado alinhado nos eixos x e y que contenha todos os pontos da instância. Assim:

$$L = \max(\max_i x_i - \min_i x_i, \max_i y_i - \min_i y_i) \tag{3.39}$$

Como a instância deve possuir no mínimo dois pontos, sabemos também que a seguinte condição é válida:

$$L \leq OPT(I) \tag{3.40}$$

Para fazer a instância "boa", criamos uma grade de linhas horizontais e verticais onde

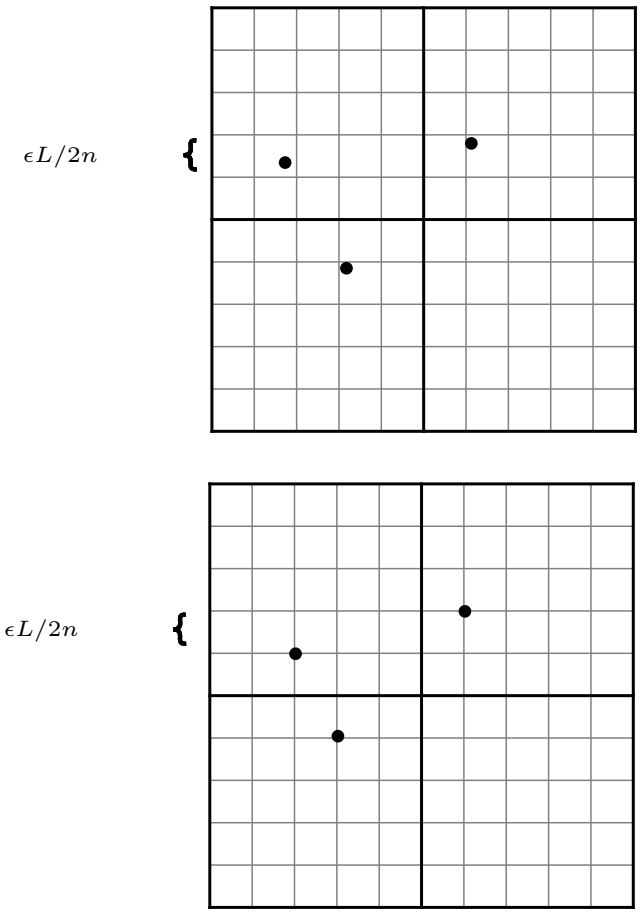

Figura 3.4: Construção de uma instância "boa" a partir de uma instância de entrada I. Os pontos são movidos para as linhas de grade, na intersecção entre as linhas verticais e horizontais.

o espaço entre elas é igual a $\epsilon L/2n$. Movemos então cada ponto para o ponto de grade (formado pela intersecção entre as linhas horizontais e verticais, como na figura 3.4) mais próximo. Como movemos estes pontos, por uma distância não maior à $\epsilon L/2n$ então a distância entre dois pontos quaisquer muda por no máximo $\pm 2\epsilon L/2n$. Assim, para uma determinada rota, seu custo aumenta em no máximo $\pm \epsilon L$.

Agora aumentamos o espaçamento entre as linhas de grade em $8n/\epsilon L$, então o espaço entre as linha de grade é de no máximo $\frac{\epsilon L}{2n} \cdot \frac{8n}{\epsilon L} = 4$. Movemos a origem dos eixos para o ponto inferior esquerdo, assim todos os pontos possuem coordenadas (x, y) não negativas. A distância máxima entre dois pontos era antes de $2L$ passa a ser agora $2L \cdot \frac{8n}{\epsilon L} = O(n)$. As coordenadas (x, y) passam para a faixa $[0, O(n)]$.

O custo de um caminho qualquer na instância original I que custava um valor C passa a custar entre $\frac{8n}{\epsilon L}(C - \epsilon L)$ e $\frac{8n}{\epsilon L}(C + \epsilon L)$.

Então seja $OPT(I)$ o custo da rota ótima para a instância I, $OPT(I')$ para o ótimo na instância I', C' o custo retornado pelo PTAS para I' e C o custo da rota correspondente na instância original, obtida através de uma transformação em C'.

Sabemos que por hipótese há um PTAS para I', então $C' \leq (1+\epsilon)OPT(I')$. Sabemos também que $OPT(I') \leq \frac{8n}{\epsilon L}(OPT(I) + \epsilon L)$. Unindo todas as equações, temos:

$$\frac{8n}{\epsilon L}(C - \epsilon L) \leq C' \leq (1+\epsilon)OPT(I') \leq (1+\epsilon)\frac{8n}{\epsilon L}(OPT(I) + \epsilon L) \tag{3.41}$$

Se olharmos os lados extremos da inequação temos:

$$C - \epsilon L \leq (1+\epsilon)(OPT(I) + \epsilon L) \tag{3.42}$$

Sabemos que $L \leq OPT(I)$, então:

$$C \leq (1 + 3\epsilon + \epsilon^2)OPT(I) \tag{3.43}$$

Um valor pequeno de ϵ nos permite escolher $\epsilon' = (1 + 3\epsilon + \epsilon^2)$ e obtemos uma $(1+\epsilon')$-aproximação.

Logo, se houver um esquema de aproximação na instância modificada I', então há um esquema de aproximação em I.

□

O próximo passo da prova é demonstrar como obter um PTAS para a instância modificada I'.

Embora ainda não tenhamos terminado o estudo da demonstração, vamos apresentar a notação necessária para falar de dois teoremas, os quais após provados, asseguram um esquema de aproximação polinomial.

Tendo a instância I' em mãos, redefinimos L como sendo o quadrado que contém todos os itens desta instância modificada. Definimos então L' como sendo a menor potência de 2 que é pelo menos $2L$.

Vamos desenhar um quadrado de lado L' que contenha todos os pontos e então dividir

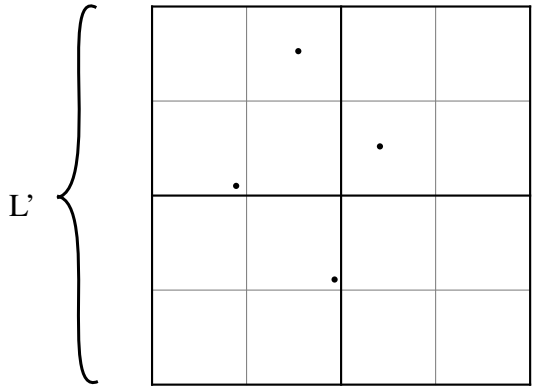

Figura 3.5: Exemplo de uma dissecção. O primeiro nível, para o quadrado de lado L' é o nível 0 da disseção. Há quatro quadrados no nível 1 e 16 no nível 2.

cada lado por 2 (Figura 3.5), originando quatro quadrados internos, repetiremos este processo até que o tamanho de cada quadrado seja igual a um. Como as coordenadas de cada ponto estão na faixa $[0, O(n)]$, L' é também $O(n)$ e esse processo de divisão sucessiva gasta $O(\log n)$ instruções.

O quadrado L' pode ser posicionado em várias coordenadas e ainda assim cobrir todos os pontos, mais precisamente nas coordenadas (a, b), para $a, b \in (-L'/2, 0]$ com o ponto de origem no canto inferior esquerdo. Esta propriedade será utilizada nos teoremas abaixo para obter um PTAS a partir do posicionamento probabilístico de L'. Chamaremos o processo de escolha de L' em coordenadas (a, b) e sua divisão em quadrados menores de (a, b)-dissecção.

O próximo passo é, ao longo de cada subquadrado gerado a partir de L' em uma (a, b)-dissecção vamos inserir pontos ao longo de suas linhas de grade chamadas de portais. Para cada portal de nível i, colocamos portais ao longo dos quatro cantos do quadrado e mais $m - 1$ ao longo das linhas de grade.

Consideraremos rotas que entram e saem de cada quadrado somente por estes portais e chamaremos estas de *p-rotas*. Dizemos que uma $p-$rota é $r-$*light* se ela cruza, para cada subquadrado da dissecção, apenas r vezes cada um de seus lados.

A partir do lema e das notações que definimos, podemos apresentar dois teoremas que estamos estudando suas demonstrações e ainda não os provamos.

Teorema 3.8. *Se escolhermos inteiros* (a, b) *no intervalo* $(-L'/2, 0]$ *uniformemente e*

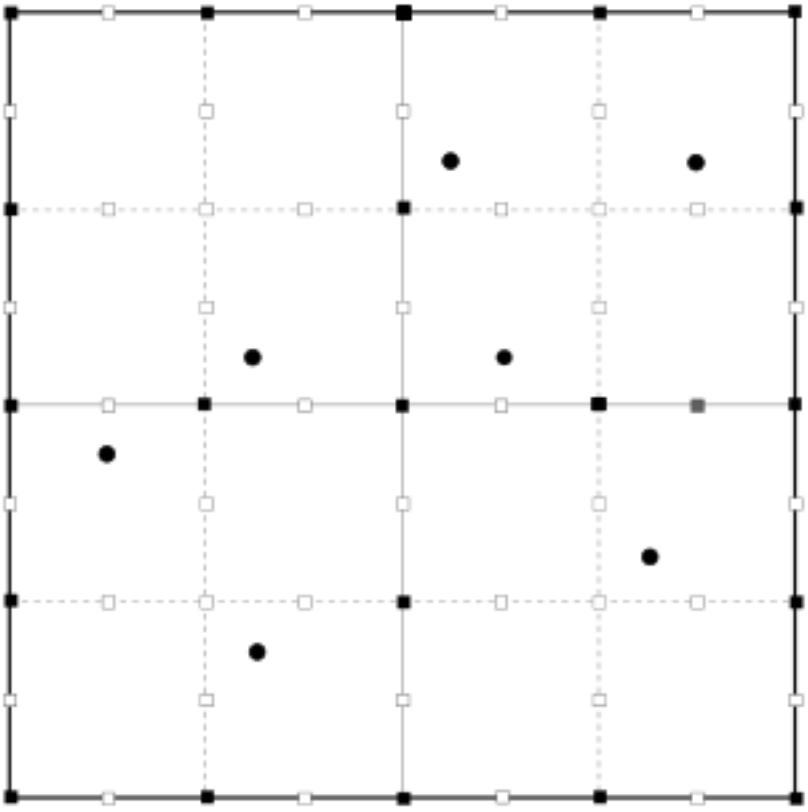

Figura 3.6: Visualização dos portais em uma dissecção. Os quadrados de fundo preto representam portais para os níveis 1 e 2 da dissecção. Os quadrados de fundo branco representam portais para o nível 2 apenas. Fonte: [32]

aleatoriamente, então com probabilidade de $1/2$ *a dissecção* (a, b) *tem uma p-rota* $r-light$ *de custo no máximo* $(1+\epsilon)OPT$ *para um parâmetro de portais* $m = O\left(\frac{1}{\epsilon}\log L'\right)$ *e* $r = 2m + 4$

e

Teorema 3.9. *Se escolhermos inteiros* (a, b) *no intervalo* $(-L'/2, 0]$ *uniformemente e aleatoriamente, então com probabilidade de* $1/2$ *a dissecção* (a, b) *tem uma p-rota* $r-light$ *de custo no máximo* $(1+\epsilon)OPT$ *para um parâmetro de portais* $m = O\left(\frac{1}{\epsilon}\log L'\right)$ *e* $r = O\left(\frac{1}{\epsilon}\right)$

Estes teoremas diferem quanto ao parâmetro r. Os passos necessários para concluí-los são: Demonstrar que é possível encontrar a menor p-rota por programação dinâmica em tempo polinomial e que é possível obter um esquema de aproximação polinomial se arbitrarmos a origem do quadrado L' com a (a, b)-dissecção.

Deixaremos estes como tarefa pendente até o final de julho no cronograma desta proposta de qualificação.

3.5 Considerações sobre o capítulo

Vimos neste capítulo quatro problemas que possuem PTAS conhecido e usam programação dinâmica: Problema da mochila, escalonamento de tarefas em máquinas paralelas idênticas, empacotamento e caixeiro viajante euclidiano. As demonstrações foram baseadas no livro de Williamsom [32].

Citamos aqui os autores de cada um dos algoritmos que estudamos. Para o problema da mochila, o algoritmo 3.1 foi apresentado por Lawler [24]. O esquema de aproximação do algoritmo 3.2 foi demonstrado por Ibarra e Kim [20].

O esquema de aproximação para o escalonamento de tarefas com número constante de máquinas foi demonstrado por Graham [15]. Para o caso onde o número de máquinas faz parte da entrada do problema a demonstração foi realizada por Hochbaum e Shmoys [18].

O esquema de aproximação para o problema do empacotamento foi demonstrado por Fernandez de la Vega e Lueker [9].

O PTAS para o problema do caixeiro viajante foi proposto por Arora [2]. Não terminamos a demonstração deste problema, deixando-o como tarefa futura.

CAPÍTULO 4

PROPOSTA E OBJETIVOS

Além do término da demonstração do esquema de aproximação polinomial para o problema do caixeiro viajante euclidiano, apresentamos como proposta estudar um outro PTAS.

Este algoritmo apresentado em [26] resolve um problema que faz parte de uma classe interessante de problemas: A dos grafos de discos unitários.

Falaremos primeiro da definição de grafo de disco unitário.

Definição 4.0.1 (Grafo de disco unitário)**.** *Um grafo de disco unitário $G(V, E)$ é um grafo formado pela intersecção de círculos unitários no plano euclidiano, onde cada círculo é um vértice $v \in V$ e para cada par de círculos, há uma aresta $e \in E$ se e somente se houver uma intersecção entre círculos.*

Um grafo de disco unitário é formado por círculos no plano euclidiano, correspondendo cada círculo em uma vértice e cada intersecção de círculos uma aresta.

A figura ilustra um conjunto de círculos unitários e seu correspondente grafo.

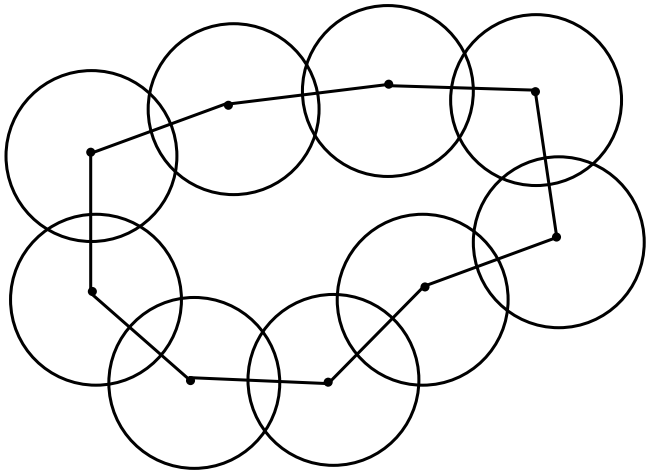

Figura 4.1: Exemplo de grafo de disco unitário

Vários trabalhos discutem grafos de discos unitários na literatura, como [8], [27]. Estes possuem aplicações diretas em redes de cobertura de sensores [19], redes *ad-hoc* [22], etc.

Estudaremos alguns trabalhos referentes a grafos de discos unitários, em especial aqueles aqui referenciados. Em seguida estudaremos o problema da cobertura de vértices para estes grafos.

Recentemente, um PTAS para o problema da cobertura de vértices em grafos de discos unitários foi descoberto [26], entretanto, carecem na literatura estudos comparativos entre este algoritmo e o uso de heurísticas para a solução deste problema, como as apresentadas em [27].

Vamos definir o problema da cobertura de vértices em discos unitários com peso (Baseada em [26]):

Definição 4.0.2 (Problema da cobertura de vértices em discos unitários com peso)**.** *Dado um conjunto* $\mathcal{D} = \{D_1, \ldots, D_n\}$ *de n discos unitários, e um conjunto* $\mathcal{P} = \{P_1, \ldots, P_m\}$ *de m pontos no plano euclidiano* $\mathbb{R}^2$*. Cada disco* D_i *tem peso* $w(D_i)$*. Escolher um conjunto de discos para cobrir todos os pontos em* $\mathcal{P}$ *, minimizando o peso total dos discos escolhidos.*

Este problema introduz outro tipo de grafo de disco unitário contendo pesos nas arestas. Por simplicidade omitiremos esta definição. Há um equivalente do mesmo problema para grafos de disco unitários apresentado em [26]. No restante desta discussão trataremos da versão com peso.

Enquanto estudávamos cada um dos problemas, uma das maiores dúvidas que tivemos foi quanto à observação do viés provocado pelo ajuste do fator de aproximação do algoritmo em seu tempo de execução. Sabemos que muitos destes problemas (Como o da cobertura de vértices em grafos de discos unitários) possuem também heurísticas conhecidas.

Sendo assim, pretendemos comparar o PTAS para o problema da cobertura de vértice em discos unitários com as heurísticas apresentadas no outro trabalho, visando determinar qual o desempenho destas técnicas, sendo que definiremos abaixo o que consideramos como indicadores de desempenho.

4.1 Metodologia

Implementaremos as soluções em linguagem de programação (C/C++). A princípio o estudo seguirá o seguinte critério: Para uma dada instância I do problema, avaliaremos um conjunto de algoritmos A_i, $i \geq 2$ por valor da solução e número de instruções executadas nos trechos de maior complexidade. Definiremos estes trechos mediante análise assintótica nos algoritmos apresentados, seguindo o mesmo procedimento que usamos para avaliar o algoritmo da subsequência crescente máxima, apresentado na seção 2.3.1.

Justificamos contar o número de instruções nos trechos de maior complexidade em detrimento do tempo de execução por este último estar bastante atrelado à arquitetura e estado do dispositivo que executa o algoritmo. Esta análise é mais precisa e podemos obtê-la declarando uma variável e incrementando esta toda vez que executada uma instrução, a qual sabemos que no modelo RAM possui um valor constante $O(1)$ por instrução [2].

Buscamos na literatura alguns exemplos de comparação de algoritmos para problemas de otimização. Alguns deles comparam inclusive heurísticas, como [29]. O autor do trabalho em questão usa dois parâmetros para comparar algoritmos de rotas de veículos: Distância média da melhor solução encontrada (Chamado de *average gap*) e tempo de execução. Consideramos a primeira medida interessante e utilizaremos esta em nossos estudos.

Em [27] apresenta-se uma heurística que obtém, para um dado grafo de discos unitários, uma 10-aproximação, obtida a partir de algumas premissas. Acreditamos que esta heurística pode ser nosso ponto de partida.

Como os problemas diferem um pouco dos apresentados nesse artigo, precisaremos adaptar as heurísticas, algo que também consideremos em nosso estudo, mais precisamente em nossa terceira etapa.

Entretanto, não sabemos se as heurísticas deste trabalho serão suficientes para o problema ou precisaremos definir novas heurísticas a partir de nosso conhecimento sobre os problemas. Em [25] discute-se o processo de criação destas heurísticas, algo que precisamos também levar em consideração quando estudarmos os problemas em grafos de discos unitários.

Nosso objetivo, além de verificar a aplicabilidade de um PTAS proposto na literatura é avaliar as consequências de aproximar-se mais da otimalidade. Arora [2], por exemplo em seu artigo que trata do esquema de aproximação polinomial comenta que sua solução é mais lenta que algumas heurísticas conhecidas para o caixeiro viajante euclidiano.

Algo também curioso que possamos verificar com o número de instruções é qual a ordem de execução (grau do polinômio) o qual o algoritmo assintoticamente encontra-se. Isto é interessante para discutirmos tratabilidade, pois segundo a tese de *Cobham-Edmonds*, todo programa polinomial pode é (tratável ou computável) em um dispositivo físico [10]. Veremos se mesmo implementável, um PTAS pode ser útil nestas condições em detrimento de uma heurística.

Sendo assim estamos interessados em avaliar as seguintes características: Facilidade de codificação de cada uma das abordagens, impactos nos ajustes dos parâmetros de otimização do PTAS, distância média entre a solução encontrada pelo PTAS e as heurísticas para a melhor solução encontrada após várias execuções e número de instruções gastas na etapa de maior complexidade dos algoritmos.

Veremos na próxima seção o cronograma de atividades.

4.2 Cronograma

As etapas de conclusão do mestrado são:

1. Conclusão de créditos em disciplina
2. Estudar esquemas de aproximação polinomial e programação dinâmica, incluindo o problema do caixeiro viajante euclidiano
3. Estudar definição de grafos de discos unitários.
4. Estudar trabalhos de PTAS em grafos de discos unitários
5. Implementar os algoritmos em linguagem de programação (C/C++)
6. Comparar os algoritmos nos critérios de : Análise assintótica, contagem de instruções nos trechos de maior complexidade, facilidade de codificação e impactos nos

Tabela 4.1: Atividades para o ano de 2014

Número da atividade	Jan	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												

ajustes do PTAS

7. Compactar resultados

8. Escrever a dissertação

9. Defesa

10. Escrever o artigo científico

Fornecemos abaixo o cronograma de tarefas já concluídas e futuras, este cronograma é dividido entre as tabelas 4.1, 4.2 e 4.3.

A previsão de defesa é para Junho de 2016. Separamos uma tarefa específica para definir as métricas de comaparação apropriadas, já que podemos incluir outras ainda não planejadas.

Incluímos também uma tarefa de compactação de resultados do estudo comparativo, onde faremos a síntese das conclusões para inclusão dos mesmos na dissertação e artigo científico.

A dissertação pode ser realizada em conjunto com as tarefas do ano de 2016, já que até lá teremos concluído os estudos com grafos de disco unitário e os itens já estudados, faltando apenas documentar e apresentar o estudo comparativo.

Tabela 4.2: Atividades para o ano de 2015

Número da atividade	Jan	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez
2												
3												
4												
5												
6												
7												
8												
9												
10												

Tabela 4.3: Atividades para o ano de 2016

Número da atividade	Jan	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez
2												
3												
4												
5												
6												
7												
8												
9												
10												

CAPÍTULO 5

CONSIDERAÇÕES FINAIS

Apresentamos neste trabalho as atividades desenvolvidas e a proposta de continuidade de trabalho do mestrado. Resumimos nesta seção um pouco do que falamos em cada um dos capítulos deste trabalho e fazemos algumas considerações sobre cada um deles.

Vimos no capítulo 2 as definições de PTAS, programação dinâmica e dos problemas que tratamos. Apresentamos uma definição um pouco mais formal de programação dinâmica, algo que em geral é omitida na literatura de algoritmos de aproximação. Ademais, consideramos no início desta proposta de dissertação estudar problemas de programação dinâmica aproximada antes de escolhermos estudar esquemas de aproximação polinomial, chegando a inclusive estudar parte introdutória do livro de Dmitri Bertsekas , fornecido em [5]. Parte das definições foram aproveitadas deste trabalho.

No capítulo 3 apresentamos esquemas de aproximação polinomial para os problemas da mochila, escalonamento de tarefas em máquinas paralelas, empacotamento e caixeiro viajante euclidiano. Estes algoritmos foram fruto de diversos trabalhos, sendo que baseamos as descrições destes algoritmos no livro de Williamson e Shimoys [32].

No capítulo 4 apresentamos a proposta de continuidade do trabalho. Escolhemos estudar grafos de disco unitário devido à descoberta recente de alguns esquemas de aproximação polinomial para estes algoritmos. O assunto é referenciado também em alguns trabalhos de [19], [16] e [26] para tratar de cobertura de redes de sensores, algo que empresas como o Google [14] estão pesquisando para tentar levar *internet* à regiões remotas do planeta.

Por simplicidade e redução de tempo omitimos alguns outros itens interessantes no estudo de problemas de aproximação, como a das classes $PTAS$ e APX, que resumem a aproximalidade de problemas [28], definidas abaixo:

Definição 5.0.1 (APX)**.** *Um problema P pertence à APX se há uma α-aproximação*

para P, sendo α uma constante.

Definição 5.0.2 (PTAS)**.** *Um problema P pertence à PTAS se há um esquema de aproximação polinomial para P*

Embora não tenhamos dito anteriormente, o conceito de PTAS engloba classes de problemas que admitem aproximações polinomiais. Entretanto, algumas classes de problemas admitem apenas aproximações por um fator constante, como é o caso da classe *APX*. Nestas, infelizmente é impossível obter um PTAS para um problema, mas é possível obter aproximações constantes, como o caso do problema *MAX-3SAT* [17].

Gostaríamos de falar mais sobre este além de outros tópicos que eventualmente estudamos e aprendemos enquanto fizemos este trabalho. Deixaremos isto para uma data futura.

BIBLIOGRAFIA

[1] Paola Alimonti e Viggo Kann. Some apx-completeness results for cubic graphs. *Theoretical Computer Science*, 237(1):123–134, 2000.

[2] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM (JACM)*, 45(5):753–782, 1998.

[3] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, e Mario Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, maio de 1998.

[4] Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716, 1952.

[5] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 2nd edition, 2000.

[6] Avrim Blum. Dynamic programming, Agosto de 2009.

[7] Stephen P. Bradley, Arnoldo C. Hax, e Thomas L. Magnanti. *Applied mathematical programming*. Addison-Wesley, Reading Mass., 1977.

[8] Brent N Clark, Charles J Colbourn, e David S Johnson. Unit disk graphs. *Discrete mathematics*, 86(1):165–177, 1990.

[9] W Fernandez De La Vega e George S. Lueker. Bin packing can be solved within 1+ ε in linear time. *Combinatorica*, 1(4):349–355, 1981.

[10] Steven Homer et al. Turing and the development of computational, complexity. Relatório, 2011.

[11] Paulo Feofiloff. Análise assintótica: ordens o, omega e theta. Aula, Abril de 2015.

[12] Paulo Feofiloff. Subsequência crescente máxima. Aula, Abril de 2015.

[13] Lance Fortnow. The status of the p versus np problem. *Commun. ACM*, 52(9):78–86, setembro de 2009.

[14] Google. Project loon. Página, 2015.

[15] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[16] Himanshu Gupta, Samir R. Das, e Quinyi Gu. Connected sensor cover: Self-organization of sensor networks for efficient query execution. *Proceedings of the 4th ACM International Symposium on Mobile Ad Hoc Networking &Amp; Computing*, MobiHoc '03, p ginas 189–200, New York, NY, USA, 2003. ACM.

[17] Johan Håstad. Some optimal inapproximability results. *J. ACM*, 48(4):798–859, julho de 2001.

[18] Dorit S Hochbaum e David B Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM)*, 34(1):144–162, 1987.

[19] Lingxiao Huang, Jian Li, e Qicai Shi. Approximation algorithms for the connected sensor cover problem. *CoRR*, abs/1505.00081, 2015.

[20] Oscar H. Ibarra e Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, outubro de 1975.

[21] Tibor Janosi. Introduction to asymptotic analysis. Aula, 2004.

[22] Fabian Kuhn, Rogert Wattenhofer, e Aaron Zollinger. Ad-hoc networks beyond unit disk graphs. *Proceedings of the 2003 joint workshop on Foundations of mobile computing*, p ginas 69–78. ACM, 2003.

[23] David Laibson. Introduction to dynamic programming. Aula, 2010.

[24] E.L. Lawler. Fast approximation algorithms for knapsack problems. *Foundations of Computer Science, 1977., 18th Annual Symposium on*, p ginas 206–213, Oct de 1977.

[25] Douglas B. Lenat. The nature of heuristics. (SSL-81-01/CIS-12), 1981.

[26] Jian Li e Yifei Jin. A PTAS for the weighted unit disk cover problem. *CoRR*, abs/1502.04918, 2015.

[27] Madhav V Marathe, Heinz Breu, Harry B Hunt, Shankar S Ravi, e Daniel J Rosenkrantz. Simple heuristics for unit disk graphs. *Networks*, 25(2):59–68, 1995.

[28] University of Waterloo. Complexityzoo. Página, 2015.

[29] Stefan Ropke. Heuristic and exact algorithms for vehicle routing problems. 2005.

[30] André Vignatti. Análise de algoritmos. Aula, 2013.

[31] André Luís Vignatti. Aproximação e Compartilhamento de Custos em Projeto de Redes. 2006.

[32] David P. Williamson e David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, New York, NY, USA, 1st edition, 2011.