Instituto de Matemática e Estatística
Universidade de São Paulo

Algoritmos de Aproximação e Problemas de *Clustering*

Samuel Plaça de Paula

Supervisora: Profª Cristina Gomes Fernandes

São Paulo, 2012

Sumário

I Parte Objetiva **1**

1 Introdução **2**

2 Conceitos **2**

2.1 Grafos 2

2.2 Programação linear 3

2.3 Problemas de decisão e classes de complexidade 5

2.3.1 Linguagens e representação 5

2.3.2 Problemas de decisão e as classes P e NP 6

2.3.3 NP-completude 7

2.4 Problemas de otimização 8

2.4.1 Problemas NP-difíceis 8

3 Uma Introdução a Algoritmos de Aproximação **8**

3.1 Por que algoritmos de aproximação? 8

3.2 Uma introdução às técnicas 10

3.3 Arredondamento determinístico 11

3.4 Arredondando uma solução dual 12

3.5 O método primal-dual 16

3.6 Um algoritmo guloso 17

3.7 Um algoritmo de arredondamento aleatorizado 22

4 Problemas de *Clustering* **24**

4.1 Minimização da maior distância entre elementos de um mesmo *cluster* 25

4.1.1 Um algoritmo de aproximação para k-tMM 26

4.1.2 Complexidade dos problemas de *clustering* 28

4.2 Aproximando problemas de gargalo 31

4.2.1 Aproximando o problema dos k-centros 33
4.2.2 Aproximando o *k-supplier* 34
4.3 Problemas de k-centros com tolerância a falhas 35
4.3.1 Os k-centros com α-vizinhos 35
4.3.2 Os k-centros com α-todos-vizinhos 36
4.3.3 O *k-supplier* com α-vizinhos 37

5 Conclusões **38**

II Parte Subjetiva **40**

6 O processo, desafios e frustrações **41**

7 A experiência do BCC **41**

8 Relação entre o curso e o trabalho de formatura **42**

9 Trabalhos futuros **42**

10 Agradecimentos **43**

Parte I

Parte Objetiva

1 Introdução

Encontrar um conjunto mínimo de pedaços suspeitos de código de forma que, com os pedaços tomados, se possam determinar "assinaturas" de todo um conjunto dos vírus de computador conhecidos. Determinar onde devem ser abertos centros de distribuição de um certo produto de modo que nenhuma loja que o vende esteja muito distante do centro de distribuição mais próximo. Descobrir como dividir elementos em categorias de modo que as categorias não sejam muito heterogêneas.

Há uma quantidade imensa de problemas, tais como esses, que podem ser modelados como problemas de *otimização combinatória*. Podemos resolvê-los com o auxílio de um computador. No entanto, dentre estes, muitos dos mais interessantes são computacionalmente difíceis, no sentido de que não podem ser resolvidos por algoritmos eficientes (isto é, de tempo polinomial no tamanho da descrição do problema) a menos que P = NP. Como em geral precisamos obter respostas em tempo razoável, uma maneira de enfrentar esse problema são os Algoritmos de Aproximação.

Algoritmos de aproximação são algoritmos que, para um problema de otimização, calculam eficientemente soluções aproximadamente ótimas. Ou seja, terminam em tempo polinomial e devolvem uma solução cujo valor dista do valor ótimo no máximo em um determinado fator chamado de garantia de desempenho.

Neste trabalho, descrevemos algumas técnicas de desenvolvimento de algoritmos de aproximação através de exemplos de algoritmos para variados problemas, com foco nos problemas de *clustering*, como o *k-center* e algumas de suas numerosas variantes. Estão presentes algoritmos e resultados clássicos encontrados na literatura, bem como resultados presentes em artigos científicos recentes e/ou cujos resultados são os melhores conhecidos — ou mesmo os melhores possíveis — para os problemas estudados.

2 Conceitos

Começaremos com uma breve introdução a alguns dos conceitos importantes para o trabalho. Nesta seção, trataremos de noções básicas de grafos, complexidade computacional e programação linear.

2.1 Grafos

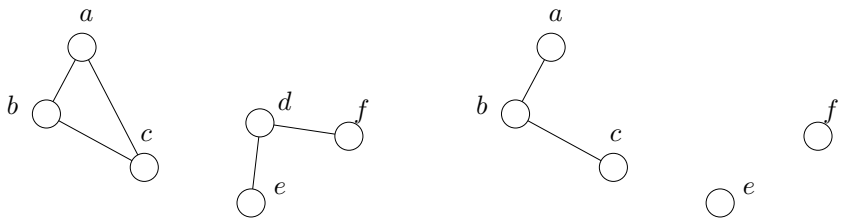
Um grafo é um conjunto de objetos, os **vértices**, acompanhado de uma lista de ligações entre pares de objetos, as **arestas**. Mais precisamente, damos a seguinte definição.

Definição 2.1. *Um* grafo *é um par* (V, E)*, onde* V *é um conjunto qualquer e* $E \subseteq \{X \subseteq V \colon |X| = 2\}$.

Isto é, E é um conjunto de pares (não ordenados) de elementos em V. Um elemento $e \in E$ tem a forma $e = \{x, y\}$, com $x \in V$, $y \in V$ e $x \neq y$. Dizemos que V é o *conjunto de vértices*, e E, o *conjunto de arestas*.

Embora em teoria dos grafos não se façam restrições sobre V, neste trabalho consideraremos sempre que o conjunto de vértices é finito.

É comum representar-se um grafo através de uma figura, onde vértices são, comumente, representados por círculos ou pontos; e as arestas, por linhas ligando vértices dois a dois. Por exemplo, considere o grafo $G = (V, E)$ com $V = \{a, b, c, d, e, f\}$, $E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{d, e\}, \{d, f\}\}$. À esquerda na figura 2.1 damos uma representação de G.



(a) Uma representação do grafo G (b) Um subgrafo H de G

Figura 2.1: Figuras representando grafos.

Além disso, um conceito comum quando se fala em grafos, e que será de fundamental importância nos problemas de *clustering*, é a existência de *custos* nas arestas. Isto é, consideramos que o grafo pode ter um custo (ou *peso*) associado a cada uma de suas arestas.

Podemos formalizar um grafo com custos nas arestas como um par (G, c), onde $G = (V, E)$ é um grafo e $c\colon E \longrightarrow X$ é uma função que associa a cada aresta um custo. Se $e \in E$ e $c(e) = x$, dizemos que x é o custo da aresta e.

Também daremos as seguintes definições sobre grafos, que são úteis para o trabalho.

Definição 2.2. *Seja $e = \{v, w\}$ uma aresta. Os vértices v e w são chamados de* pontas *da aresta e.*

Definição 2.3. *Dado um grafo $G = (V, E)$, dizemos que dois vértices $v, w \in V$ estão* conectados *se $\{v, w\} \in E$. Podemos dizer, também, que v e w são* adjacentes *ou* vizinhos*.*

Definição 2.4. *Dado um grafo $G = (V, E)$, um conjunto $S \in V$ é uma* clique *se, para quaisquer $v, w \in S$, $\{v, w\} \in E$. Isto é, se os vértices de S estão todos dois a dois conectados.*

Definição 2.5. *Dado um grafo $G = (V, E)$, um conjunto $I \in V$ é chamado* independente *se, para quaisquer $v, w \in I$, $\{v, w\} \notin E$. Isto é, se todos os vértices de I estão dois a dois* não *conectados.*

Definição 2.6. *Dado um grafo $G = (V, E)$, um grafo $H = (V', E')$ é um* subgrafo *de G se $V' \subseteq V$ e $E' \subseteq E$. Por vezes usamos a notação $H \subseteq G$ para dizer que H é subgrafo de G.*

Definição 2.7. *Um grafo $G = (V, E)$ é dito* completo *se tem todas as arestas possíveis, isto é, se $E = \{\{u, v\} \subseteq V : u \neq v\}$.*

Definição 2.8. *Dados um grafo $G = (V, E)$ e um conjunto $X \subseteq V$, o* subgrafo de G induzido por X *é o grafo $H = (X, \{e \in E\colon$ *as duas pontas de e estão em $X\}$).*

No grafo G da figura 2.1, o conjunto $\{a, b, c\}$ é uma clique, e $\{b, e, f\}$ é um conjunto independente. O grafo $H = (\{a, b, c, e, f\}, \{\{a, b\}, \{b, c\}\})$ é um exemplo de subgrafo de G, e está representado à direita, também na figura 2.1.

2.2 Programação linear

Programação linear é uma área da matemática e da computação que se dedica ao estudo e resolução dos **problemas de programação linear**. Um problema de programação linear, ou **programa linear**, é um problema de otimização que toma a seguinte forma geral: deseja-se encontrar um vetor $x \in \mathbb{R}^n$ que minimize (ou maximize) o valor de uma função linear, dado que esteja satisfeito um conjunto de restrições lineares sobre x.

O que chamamos aqui de função linear é uma função do tipo $f(x) = \langle c, x \rangle$, isto é, produto escalar de c por x, com $c \in \mathbb{R}^n$. Uma restrição linear é uma restrição em uma das seguintes formas:

- $\langle a_j, x\rangle \geq b_j$,
- $\langle a_j, x\rangle \leq b_j$, ou
- $\langle a_j, x\rangle = b_j$,

com $a_j \in \mathbb{R}^n$, $b_j \in \mathbb{R}$. Se temos um conjunto de restrições indexadas por j, com $1 \leq j \leq m$, podemos entender cada a_j como uma linha de uma matriz $m \times n$, A, e cada b_j como entrada de um vetor b.

É comum escrevermos um programa linear (PL) na seguinte disposição:

$$\begin{aligned} \min \quad & \langle c, x\rangle \\ \text{sujeito a} \quad & Ax \geq b, \\ & x \geq 0, \end{aligned} \tag{PLG}$$

com $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$ fixos para o problema.

As desigualdades, conforme dissemos anteriormente, não precisam ser todas do tipo "maior ou igual", assim como as entradas de x não precisam ser não-negativas. No entanto, essa forma é suficientemente genérica para representar todo programa linear. Isto é, qualquer programa linear admite um programa equivalente na forma geral descrita pelo PLG [1].

Um exemplo simples de problema seria o seguinte: encontrar x_1, x_2, x_3 não negativos que maximizem a função $3x_1 - x_2 + \frac{1}{2} \cdot x_3$, satisfazendo as restrições $x_1 \leq 5$, $x_2 \geq \frac{11}{4}$ e $\frac{4}{5}x_1 + x_3 = 10$. Ou, no formato usual,

$$\begin{aligned} \min \quad & x_1 - x_2 + \frac{1}{2}x_3 \\ \text{sujeito a} \quad & x_1 \leq 5, \\ & x_2 \geq \frac{11}{4}, \\ & \frac{4}{5}x_1 + x_3 = 10 \\ & x_1, x_2, x_3 \geq 0. \end{aligned}$$

O programa pode ser escrito da seguinte forma:

$$\begin{aligned} \min \quad & 1x_1 + (-1)x_2 + \frac{1}{2}x_3 \\ \text{sujeito a} \quad & (-1)x_1 + 0x_2 + 0x_3 \geq -5, \\ & 0x_1 + 1x_2 + 0x_3 \geq \frac{11}{4}, \\ & \frac{4}{5}x_1 + 0x_2 + x_3 \geq 10, \\ & \left(-\frac{4}{5}\right)x_1 + 0x_2 + (-1)x_3 \geq -10, \\ & x_1, x_2, x_3 \geq 0, \end{aligned}$$

de onde A, b e c da forma geral saem naturalmente.

Neste trabalho, entretanto, a matriz A e os vetores c e b não serão particularmente importantes para os algoritmos estudados, e por isso não os daremos explicitamente ao apresentar um programa linear.

Para nós, o fato mais importante sobre problemas de programação linear é que eles podem ser resolvidos em tempo polinomial. De fato, mais do que isso, programas lineares podem ser resolvidos muito rapidamente, por exemplo, pelo método Simplex.

2.3 Problemas de decisão e classes de complexidade

2.3.1 Linguagens e representação

Um problema de decisão, em computação, é um "conjunto de instâncias positivas". Dada uma instância I para o problema, desejamos decidir se a resposta para I é "sim" ou "não", ou seja, se I está ou não no conjunto de instâncias positivas. Daremos, mais adiante, uma definição mais formal dessa noção.

Um exemplo de problema de decisão é o problema da primalidade: dado um número natural n, desejamos saber se n é primo ou não. O conjunto de instâncias positivas, no caso, é simplesmente o conjunto $\mathbb{P}$ dos números primos. Desejamos saber se a resposta para a pergunta "$n \in \mathbb{P}$?" é "sim" ou "não".

Estamos interessados em estudar a possibilidade de resolver alguns desses problemas, e, mais que isso, desejamos saber se existem algoritmos que resolvem tais problemas de maneira eficiente. Para tanto, faz-se necessário definir o que, para nós, significa "eficiência". Pensaremos em alguma medida de eficiência em relação ao *tamanho da instância*. Começaremos, portanto, descrevendo o que chamamos de tamanho de uma instância, usando para isso o conceito de *linguagem*.

Um *alfabeto* é um conjunto Σ finito qualquer. Um elemento $\sigma \in \Sigma$ é chamado de *símbolo*. Dado um alfabeto Σ, uma *palavra* sobre Σ é qualquer sequência finita de símbolos em Σ. Denotamos por Σ^* o conjunto de todas as palavras sobre Σ.

Definição 2.9. *O* comprimento *de uma palavra $w \in \Sigma^*$, denotado por $|w|$, é o número de símbolos que compõem w.*

Por exemplo, para o alfabeto $\Sigma = \{a, b, c\}$, temos $abab$, a e $cabc$ como alguns exemplos de palavras. O conjunto de todas as palavras para esse alfabeto é $\Sigma^* = \{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, \ldots\}$. Note que incluímos uma palavra denotada por ε. Trata-se da *palavra vazia*, que tem comprimento zero e é uma palavra sobre qualquer alfabeto.

Definição 2.10. *Dado um alfabeto Σ, uma* linguagem *é um conjunto $L \subseteq \Sigma^*$.*

Ainda usando o exemplo anterior, poderíamos dar os seguintes exemplos de linguagens:

- $L = \{ab, ba, bccac\}$,
- $L = \{w \in \Sigma^* \colon |w| \text{ é par}\} = \{\varepsilon, aa, ab, ac, ba, bb, bc, \ldots, aaaa, aaab, \ldots\}$,
- $L = \{a^i \colon i \geq 2\} = \{aa, aaa, aaaa, \ldots\}$,
- $L = \emptyset$, $L = \Sigma^*$, etc.

Desejamos expressar nossos problemas de decisão como linguagens correspondentes ao conjunto de instâncias positivas. Para isso, o que chamamos de instâncias não são números, grafos, etc. diretamente, mas suas *representações* como palavras sobre algum alfabeto.

Isso faz sentido por dois principais motivos. Primeiro, como já tínhamos adiantado, para podermos falar no tamanho (ou, como definimos agora, comprimento) da instância. Segundo, porque estamos interessados em resolver os problemas usando algoritmos, e a entrada de um algoritmo é, de fato, uma representação de um objeto, e não o objeto em si.

A questão da representação é uma ideia bastante concreta: nos computadores que usamos, toda a informação é codificada, fundamentalmente, por *strings* de bits, isto é, palavras sobre o alfabeto $\{0, 1\}$.

Os computadores que conhecemos também nos sugerem que a representação via um alfabeto finito é algo bastante poderoso, uma vez que, como sabemos, objetos tão diversos como números inteiros, imagens, programas, grafos e documentos de texto podem ser todos representados por sequências de bits.

Dado um objeto $\mathcal{A}$ qualquer, denotamos por $\langle \mathcal{A} \rangle$ a *representação* de $\mathcal{A}$. Em geral não discutiremos mais o alfabeto específico sobre o qual é feita a representação. Podemos assumir que é o alfabeto binário $\Sigma = \{0, 1\}$.

2.3.2 Problemas de decisão e as classes P e NP

Usando o vocabulário que estabelecemos, podemos definir um problema de decisão simplesmente como uma *linguagem*. Por exemplo, as linguagens

- PRIMOS $= \{\langle n \rangle : n$ é um número primo$\}$,
- CLIQUE $= \{\langle G, k \rangle : G$ grafo, $k \in \mathbb{N}$ tais que G tem uma clique de tamanho $k\}$

são problemas de decisão bastante conhecidos, o primeiro dos quais já havíamos citado.

Definição 2.11. *Um algoritmo A* decide *uma linguagem L se, para qualquer entrada $w \in \Sigma^*$, A termina sua execução em tempo finito e responde corretamente se $w \in L$.*

Definição 2.12. *Um* algoritmo eficiente*, ou* de tempo polinomial*, é um algoritmo que, para qualquer entrada $w \in \Sigma^*$, tem tempo de execução limitado por um polinômio em $n = |w|$.*

Isto é, um algoritmo é eficiente se sua complexidade de tempo é $O(n^k)$ para algum k natural, onde n é o comprimento da instância.

Definição 2.13. *Um* verificador *para uma linguagem L é um algoritmo V que recebe como entrada duas palavras, w e τ, e tem a seguinte propriedade:*

$$w \in L \iff \exists \tau \text{ tal que } V \text{ responde "sim" na entrada } (w, \tau).$$

A palavra τ dessa definição pode ser entendida como um "certificado" de que w está em L. Por exemplo, para uma instância $\langle G, k \rangle \in$ CLIQUE, um certificado pode ser a palavra $\tau = \langle S \rangle$, onde S é uma clique em G com $|S| = k$. Assim, um exemplo de verificador para CLIQUE seria um algoritmo V que recebe como entrada uma instância do problema CLIQUE e (a representação de) um conjunto S, e responde "sim" se, e só se, S é uma clique de tamanho k em G. Além disso, se $\langle G, k \rangle \notin$ CLIQUE, certamente não existe nenhum τ que seja aceito por nosso verificador V como certificado válido.

Note que, nesse exemplo, podemos obter um tal verificador V que execute em tempo polinomial em $n = |\langle G, k \rangle|$. Isso porque o certificado escolhido tem tamanho polinomial na representação do grafo e pode ter sua validade verificada em tempo polinomial: trata-se do que chamamos de *certificado eficiente* (ou *certificado fácil*).

Definição 2.14. *Um* verificador polinomial *para uma linguagem L é um algoritmo V que é um verificador para L e é eficiente.*

Dizemos que L tem um certificado fácil se existe um verificador polinomial para L.

Finalmente, definimos as classes P e NP.

Definição 2.15. $P = \{L \subseteq \Sigma^* : \text{ existe algoritmo eficiente que decide } L\}$.

Definição 2.16. *$NP = \{L \subseteq \Sigma^* :$ existe verificador polinomial para $L\}$.*

Teorema 2.17. *$P \subseteq NP$.*

Demonstração. Seja $L \in$ P. Então existe um algoritmo eficiente A que decide L. Considere o seguinte algoritmo que recebe w e τ:

1. Execute A na entrada w.
2. Se A responde "sim", responda "sim"; caso contrário, responda "não".

Ou seja, o algoritmo ignora o certificado e simplesmente resolve o problema diretamente. É claro que esse algoritmo é um verificador polinomial para L, uma vez que A é eficiente. □

Não sabemos, no entanto, se P $\subsetneq$ NP [2].

2.3.3 NP-completude

Em complexidade, é comum a ideia de que um problema A se *reduz* (polinomialmente) a um problema B, isto é, de que cada instância de A é equivalente, em algum sentido, a uma instância de B (de tamanho polinomial na instância original). Damos a seguinte definição:

Definição 2.18. *Dadas duas linguagens A e B, uma* redução polinomial de A a B *é um algoritmo polinomial que, dada uma entrada $x \in \Sigma^*$, produz como saída uma palavra $y \in \Sigma^*$, com as seguintes propriedades:*

1. *$|y|$ é polinomial em $|x|$,*
2. *$x \in A \iff y \in B$.*

Se existe redução polinomial de A a B, escrevemos $A \leq_P B$.

Definição 2.19. *Um problema de decisão B é* NP-completo *se:*

1. *$B \in NP$,*
2. *$\forall A \in NP$, $A \leq_P B$.*

Clique foi um dos primeiros problemas a serem provados NP-completos [3].

Teorema 2.20. *Se $A \leq_P B$, e $B \in P$, então $A \in P$.*

Demonstração. Seja D um algoritmo polinomial que decide B. Seja R uma redução polinomial de A para B. Considere o algoritmo que, na entrada x, realiza os seguintes passos:

1. Obtenha a palavra y resultante da execução de R em x.
2. Execute D em y e dê a mesma resposta.

Tal algoritmo decide A em tempo polinomial, pois todos os seus passos são polinomiais, e, como R é uma redução, vale que $x \in A \iff y \in B$. □

Corolário 2.21. *Seja B NP-completo. Se $B \in P$, então $P = NP$.*

2.4 Problemas de otimização

Um problema de otimização é também um conjunto de instâncias. Para cada instância I, temos um conjunto de *soluções viáveis*, $\mathrm{Sol}(I)$. A cada solução viável $S \in \mathrm{Sol}(I)$, está associado um valor numérico, $\mathrm{val}(S)$.

Dada uma instância I, desejamos encontrar $S^* \in \mathrm{Sol}(I)$ tal que $\mathrm{val}(S^*)$ seja *mínimo*, para problemas de minimização, ou *máximo*, para problemas de maximização. Um tal S^* é chamado de *solução ótima*, e seu valor $\mathrm{val}(S^*)$ é o *valor ótimo*, denotado por $\mathrm{OPT}(I)$, ou simplesmente OPT, quando a instância está subentendida.

Se $\mathrm{Sol}(I) = \emptyset$, dizemos que a instância I é *inviável*.

Um exemplo de problema de otimização é o problema da clique de tamanho máximo. Trata-se de um problema de maximização. Uma instância desse problema é um grafo $G = (V, E)$. O conjunto de soluções viáveis é $\mathrm{Sol}(I) = \{S \subseteq V : S \text{ é clique em } G\}$. O valor de uma solução viável S é $\mathrm{val}(S) = |S|$.

2.4.1 Problemas NP-difíceis

Apesar de termos definido redução polinomial considerando que os dois problemas envolvidos são problemas de decisão, podemos estender esse conceito. Note que um problema de otimização, por exemplo, também pode "resolver" um problema de decisão. Tome como exemplo o problema da clique de tamanho máximo, que definimos acima. Esse problema resolve Clique da seguinte forma: dada uma instância $\langle G, k\rangle$, podemos simplesmente resolver o problema da clique máxima para G, obtendo o valor ótimo OPT. Se $\mathrm{OPT} \geq k$, então $\langle G, k\rangle \in$ Clique. Caso contrário, $\langle G, k\rangle \notin$ Clique.

Definição 2.22. *Se A e B são problemas, dizemos que A* se reduz polinomialmente *a B, denotado por $A \leq_P B$, se a existência de uma rotina polinomial que resolva B implica a existência de um algoritmo polinomial que resolve A.*

Já dissemos que Clique é NP-completo, e vimos que o problema da clique de tamanho máximo o resolve. Isso nos serve de indício de que o problema da clique de tamanho máximo não deve ser um problema fácil! De fato, trata-se de um problema *NP-difícil*.

Definição 2.23. *Seja B um problema. Dizemos que B é NP-difícil se, para qualquer $A \in NP$, vale que $A \leq_P B$.*

A seguir, apresentamos uma adaptação do primeiro capítulo do livro de Williamson e Shmoys [4] como uma introdução geral às técnicas de algoritmos de aproximação. Seguimos o conteúdo e ordem de apresentação originais, por vezes fazendo algumas modificações e complementos, em especial nas demonstrações fornecidas.

3 Uma Introdução a Algoritmos de Aproximação

3.1 Por que algoritmos de aproximação?

A otimização combinatória trata de como tomar diversas decisões do mundo real de modo a obter o melhor resultado possível dado um objetivo (ex.: maximizar lucro, minimizar gasto de material, etc.)

No entanto, os problemas de otimização combinatória mais interessantes são, em sua maioria, NP-difíceis. Isso significa que não conhecemos algoritmos eficientes que resolvam tais problemas e, de fato, a menos que P = NP, algoritmos eficientes para esses problemas *não existem*. O que, então, podemos fazer em tais casos?

Desejamos conhecer algoritmos de otimização que:

1. encontrem soluções ótimas
2. em tempo polinomial
3. para qualquer instância.

No entanto, para problemas NP-difíceis, no mínimo um desses requisitos precisa ser relaxado, a menos que P = NP.

Há abordagens que relaxam o requisito "para qualquer instância", buscando algoritmos que resolvam, em tempo polinomial, um subconjunto das instâncias de um dado problema.

Uma forma mais comum de atacar o problema é desistir da exigência de que o consumo de tempo seja polinomial. Buscam-se formas espertas de explorar todo o espaço de soluções de forma a encontrar soluções ótimas, podendo-se ter de esperar por minutos ou horas, ou mesmo correndo o risco de que a instância usada nem possa ser resolvida em tempo razoável. Isso é feito por quem resolve formulações de problemas de otimização como problemas de programação inteira, ou pesquisadores de Inteligência Artificial que usam técnicas como a busca A*, por exemplo.

No entanto, a abordagem mais comum é relaxar a restrição de otimalidade e, em vez disso, buscar uma solução "suficientemente boa", principalmente se tal solução puder ser obtida em questão de segundos — ou menos. Há uma extensa variedade de técnicas nessa categoria, e que muitas vezes resultam em soluções que são boas na prática.

Essa terceira forma de encarar os problemas NP-difíceis de otimização combinatória é a que levaremos em consideração: iremos relaxar a exigência de otimalidade, mas relaxaremos o mínimo que pudermos. Tentaremos encontrar soluções que tenham valor próximo ao da solução ótima (em termos da função objetivo para cada problema).

Definição 3.1. *Uma* α-aproximação *para um problema de otimização é um algoritmo polinomial que, para toda instância desse problema, produz uma solução cujo valor está a um fator* α *do valor de uma solução ótima.*

Nesse caso, dizemos que α é a *garantia de desempenho* do algoritmo (também chamada de *razão de aproximação* ou *fator de aproximação* do algoritmo). Adotaremos o padrão de que $\alpha > 1$ para problemas de minimização e $0 < \alpha < 1$ para problemas de maximização.

Por que, afinal, estudar algoritmos de aproximação?

- Porque precisamos solucionar problemas de otimização que aparecem com frequência no mundo real e são, muitas vezes, NP-difíceis.
- Porque bons algoritmos de aproximação para versões mais simples de um problema podem nos dar boas heurísticas para o problema original.
- Porque isso nos fornece uma base matemática rigorosa com a qual estudar heurísticas.
- Porque nos dá uma maneira de medir quão difíceis são vários problemas de otimização combinatória (em termos de quão bem eles podem ser aproximados).

- Por prazer!

Existem problemas que admitem algoritmos de aproximação realmente muito bons, a ponto de terem *esquemas de aproximação em tempo polinomial*.

Definição 3.2. *Um* esquema de aproximação em tempo polinomial *(PTAS, do inglês* polynomial-time approximation scheme*) é uma família de algoritmos $\{A_\varepsilon\}$ em que, para cada $\varepsilon > 0$, A_ε é uma $(1+\varepsilon)$-aproximação, para problemas de minimização, ou uma $(1-\varepsilon)$-aproximação, para problemas de maximização.*

3.2 Uma introdução às técnicas

Consideremos o seguinte problema: dados um conjunto $E = \{e_1, \ldots, e_n\}$, conjuntos $S_1, S_2, \ldots, S_m$ tais que cada $S_j \subseteq E$ e um peso não-negativo w_j para cada S_j, encontrar uma coleção de peso mínimo dos conjuntos que cubra E. Isto é, desejamos encontrar $I \subseteq \{1, \ldots, m\}$ que minimize $\sum_{j \in I} w_j$ sujeito a $\bigcup_{j \in I} S_j = E$. Esse problema é conhecido como *cobertura por conjuntos*. Se, para todo conjunto S_j, o peso é $w_j = 1$, o problema é chamado de cobertura por conjuntos *sem peso*.

O problema da cobertura por conjuntos é uma abstração de diversos tipos de problemas, como a *cobertura por vértices*. Em tal problema, dados um grafo (não dirigido) $G = (V, E)$ e um peso não negativo w_i para cada vértice $i \in V$, desejamos encontrar um conjunto de vértices de peso mínimo $C \subseteq V$ tal que para cada aresta $(i, j) \in E$, ou $i \in C$ ou $j \in C$. É fácil ver por que a cobertura por vértices é um caso particular da cobertura por conjuntos; o conjunto E da cobertura por conjuntos é exatamente o conjunto de arestas, e para cada $i \in V$, cria-se um conjunto $S_i = \{e \in E : e \text{ incide em } i\}$ de peso w_i (peso do vértice).

Vamos representar o problema da cobertura por conjuntos como um problema de programação linear. Teremos, para cada conjunto S_j, uma variável de decisão x_j, que indicará se S_j é incluído ou não na solução. Queremos que a variável seja booleana, e portanto formularemos o problema como um PLI (problema de programação linear inteira). Para garantir que todo elemento aparece na união dos conjuntos que compõem a solução, devemos ter a restrição

$$\sum_{j : e_i \in S_j} x_j \geq 1, \text{ para cada } e_i,\ i = 1, \ldots, n.$$

Além disso, queremos que o peso dos conjuntos que aparecem na solução seja mínimo. Ou seja, nossa função objetivo, a ser minimizada, é $\sum_{j=1}^{m} w_j x_j$. Daí, o PLI equivalente ao nosso problema é:

$$\begin{aligned} \text{minimizar } & \sum_{j=1}^{m} w_j x_j \\ \text{sujeito a } & \sum_{j : e_i \in S_j} x_j \geq 1, \quad i = 1, \ldots, n, \\ & x_j \in \{0, 1\}, \quad j = 1, \ldots, m. \end{aligned}$$

Seja Z^*_{PLI} o valor ótimo desse PLI para uma dada instância do problema. Como o PLI é um modelo exato, temos $Z^*_{PLI} = \text{OPT}$, onde OPT é o valor de uma solução ótima do problema de cobertura por conjuntos para tal instância.

No entanto, em geral, um PLI não pode ser resolvido em tempo polinomial. De fato, se pudéssemos resolver esse PLI em tempo polinomial, poderíamos resolver também a cobertura por conjuntos, que é NP-difícil, e então teríamos P = NP.

Trabalharemos, então, com um PL (problema de programação linear), que pode ser resolvido em tempo polinomial:

$$\begin{aligned} \text{minimizar } & \sum_{j=1}^{m} w_j x_j \\ \text{s.a. } & \sum_{j:e_i \in S_j} x_j \geq 1, \quad i = 1, \ldots, n, \\ & x_j \geq 0, \quad j = 1, \ldots, m. \end{aligned}$$

Note-se que é desnecessário exigir que $x_j \leq 1$ devido à nossa função objetivo (dada uma solução, um $x_j > 1$ pode ser trocado por $x_j = 1$ sem perda de viabilidade ou aumento do custo da solução).

Esse PL é uma *relaxação* do PLI, o que significa que toda solução viável do PLI é viável também no PL, e o valor (custo) de uma solução viável do PLI é o mesmo no PL. Como toda solução ótima do PLI é viável no PL e tem valor Z^*_{PLI}, uma solução ótima do PL terá valor $Z^*_{PL} \leq Z^*_{PLI} = \text{OPT}$. Isso significa que o PL é uma maneira de obter, em tempo polinomial, um limite inferior para o valor ótimo (OPT) do problema de cobertura por conjuntos.

Veremos como usar o PL para obter algoritmos de aproximação para o problema de cobertura por conjuntos.

3.3 Arredondamento determinístico

Tentemos recuperar uma solução para a cobertura de conjuntos a partir de uma solução ótima do PL, digamos x^*. Incluiremos S_j na solução se, e somente se, $x^*_j \geq 1/f$, onde f é o número máximo de conjuntos em que um elemento aparece, i.e.,

$$f = \max_{e_i \in E} |\{j : e_i \in S_j\}|. \tag{3.1}$$

Seja I o conjunto de índices j dos conjuntos incluídos nessa solução. O conjunto I indexa uma cobertura por conjuntos. Uma solução inteira para o problema $\hat{x}$ é obtida definindo-se $\hat{x}_j = 1$ se $x^*_j \geq 1/f$, e $\hat{x}_j = 0$, caso contrário. Explicitamos o algoritmo:

Algoritmo 3.1: Algoritmo de arredondamento determinístico para a cobertura por conjuntos

Entrada: Elementos e_i, conjuntos S_j e seus pesos w_j, $\quad 1 \leq i \leq n$, $1 \leq j \leq m$
Saída: Indexação I de uma cobertura por conjuntos

```
x* ← uma solução ótima do PL;
f ← max_{e_i} |{j : e_i ∈ S_j}|;
I ← ∅;
para j ← 1 até m faça
    se x*_j ≥ 1/f então
        I ← I ∪ {j};
    fim
fim
```

Lema 3.3. *A coleção de conjuntos $\{S_j : j \in I\}$, é uma cobertura por conjuntos.*

Demonstração. Diremos que um elemento e_i é *coberto* se a solução I dada pelo lema contém algum conjunto do qual e_i seja membro. Tomemos um elemento e_i qualquer e vejamos que ele está coberto. Como x^* é solução ótima, e portanto viável, do PL, sabemos que $\sum_{j:e_i \in S_j} x_j^* \geq 1$.

Pela definição de f, sabemos que e_i está em no máximo f conjuntos S_j diferentes. Isto é, há no máximo f termos nessa soma, e portanto pelo menos um desses termos tem que ser no mínimo igual a $1/f$. Então, para algum j tal que $e_i \in S_j$, vale que $x_j \geq 1/f$. Logo, $j \in I$, e o elemento e_i está coberto. □

Teorema 3.4. *O algoritmo 3.1 é uma f-aproximação para o problema de cobertura por conjuntos.*

Demonstração. Como a resolução do PL e a obtenção de uma cobertura a partir da solução do PL podem ser feitas em tempo polinomial no tamanho da instância original, segue que o algoritmo termina em tempo polinomial.

Pela construção, $j \in I$ implica que $x_j^* \geq 1/f$. Então, para cada $j \in I$, vale que $1 \leq f \cdot x_j^*$.

Além disso, como $x^* \geq 0$ (pois x^* é solução viável do PL), $f \geq 0$ e $w \geq 0$ (os pesos são não-negativos), segue que o termo fw_jx_j é não-negativo para cada $j = 1, \ldots, m$.

Assim, o valor (custo) da cobertura dada pelo algoritmo é

$$\begin{aligned}
\sum_{j \in I} w_j &\leq \sum_{j \in I} w_j \cdot (f \cdot x_j^*) \\
&\leq \sum_{j \in I} w_j \cdot (f \cdot x_j^*) + \sum_{j \notin I} w_j \cdot (f \cdot x_j^*) \\
&= \sum_{j=1}^{m} w_j \cdot (f \cdot x_j^*) \\
&= f \sum_{j=1}^{m} w_j x_j^* \\
&= f \cdot Z_{PL}^* \\
&\leq f \cdot \mathrm{OPT},
\end{aligned}$$

onde a desigualdade final vem da discussão de que $Z_{PL}^* \leq \mathrm{OPT}$. □

Para o caso especial da cobertura por vértices, $f = 2$ pois cada aresta incide em exatamente dois vértices. Portanto, para esse caso, o algoritmo acima é uma 2-aproximação.

3.4 Arredondando uma solução dual

Para começar, suponha que cada elemento e_i é taxado em um preço não negativo y_i por sua cobertura numa solução do problema de cobertura por conjuntos. É natural imaginar que alguns elementos podem ser cobertos por conjuntos de baixo peso, enquanto outros acabem exigindo a inclusão de conjuntos de grande peso na solução. Gostaríamos de capturar essa distinção cobrando pequenas taxas dos elementos no primeiro caso, enquanto elementos como no segundo caso tenham que pagar preços maiores. Para que os preços sejam razoáveis, não pode ocorrer de a soma dos preços dos elementos em um conjunto S_j seja maior que o peso do próprio conjunto, uma vez que podemos cobrir todos esses elementos pagando o peso w_j. Assim, para cada conjunto S_j, temos o seguinte limitante superior para os preços:

$$\sum_{i:e_i \in S_j} y_i \leq w_j.$$

Podemos encontrar o máximo preço total que podem ser cobrados dos elementos resolvendo o seguinte PL:

$$\begin{aligned} &\text{maximizar} \sum_{i=1}^{n} y_i \\ &\text{sujeito a} \sum_{i:e_i \in S_j} y_i \leq wj, \quad j = 1, \ldots, m, \\ &\qquad y_i \geq 0, \quad i = 1, \ldots, n. \end{aligned}$$

Esse PL é o *dual* da relaxação do problema da cobertura por conjuntos (e a este chamaremos de *primal*. Isso nos garante uma série de relações importantes, dentre as quais a propriedade da *dualidade fraca*, que diz que para quaisquer x e y viáveis no PL primal e no dual, respectivamente, vale que

$$\sum_{i=1}^{n} y_i \leq \sum_{j=1}^{m} w_j x_j,$$

isto é, o valor da solução y no dual não é maior que o valor da solução x. Em particular, nenhuma solução viável do PL dual tem valor maior que o valor de uma solução ótima do primal. Então, para qualquer y viável no dual, vale que $\sum_{i=1}^{n} y_i \leq \text{OPT}$ (pois já vimos que $Z^*_{PL} \leq \text{OPT}$). Esse fato será importante no desenvolvimento de algoritmos de aproximação.

Temos, ainda, outra propriedade importante: a chamada *dualidade forte*. É a propriedade de que uma solução ótima do PL dual e uma solução ótima do PL primal têm valores *iguais*. No nosso caso, isso significa que, se x^* é uma solução ótima do primal e y^* uma solução ótima do dual, então vale a seguinte igualdade:

$$\sum_{j=1}^{m} w_j x^*_j = \sum_{i=1}^{n} y^*_i.$$

Agora, usemos uma solução do nosso PL dual para obter uma solução da cobertura por conjuntos.

Sendo y^* uma solução ótima dual, tomemos como solução a coleção dos S_j para os quais a desigualdade dual é satisfeita *sem folga*. Isto é, tomamos os S_j para os quais $\sum_{i:e_i \in S_j} y^*_i = w_j$. Seja I' o conjunto dos índices dos conjuntos em tal coleção. O algoritmo é explicitado a seguir.

Algoritmo 3.2: Algoritmo de arredondamento dual para a cobertura por conjuntos

Entrada: Elementos e_i, conjuntos S_j e seus pesos w_j, $\quad 1 \leq i \leq n$, $1 \leq j \leq m$
Saída: Indexação I' de uma cobertura por conjuntos

```
y* ← uma solução ótima do PL dual;
I' ← ∅;
para j ← 1 até m faça
    se Σ_{i: e_i ∈ S_j} y*_i = w_j então
        I ← I' ∪ {j};
    fim
fim
```

Lema 3.5. *A coleção de conjuntos $\{S_j : j \in I'\}$ é uma cobertura por conjuntos.*

Demonstração. Suponha que exista algum elemento e_k não coberto. Então, para todo conjunto S_j contendo e_k, deve valer que

$$\sum_{i:e_i \in S_j} y^*_i < w_j. \tag{3.2}$$

Seja ε a menor diferença entre o lado direito e o esquerdo de todas as desigualdades para os conjuntos em que e_k ocorre. Isto é,

$$\varepsilon = \min_{j:e_k \in S_j} (w_j - \sum_{i:e_i \in S_j} y_i^*).$$

Pela desigualdade (3.2), sabemos que $\varepsilon > 0$.

Considere agora uma nova solução dual y' em que $y'_k = y^*_k + \varepsilon$ e todos os outros componentes de y' são iguais aos de y^*.

Para cada j tal que $e_k \in S_j$, vale que

$$\sum_{i:e_i \in S_j} y'_i = \sum_{i:e_i \in S_j} y^*_i + \varepsilon \leq w_j,$$

pela definição de ε. E, para cada j tal que $e_k \notin S_j$, temos

$$\sum_{i:e_i \in S_j} y'_i = \sum_{i:e_i \in S_j} y^*_i \leq w_j.$$

Segue, então, que y' é uma solução viável para o PL dual e, além disso,

$$\sum_{i=1}^{n} y'_i > \sum_{i=1}^{n} y^*_i,$$

pela maneira como construímos y'.

No entanto, isso contradiz a otimalidade de y^*. Logo, todo elemento necessariamente está coberto e I' é uma cobertura por conjuntos. □

Teorema 3.6. *O algoritmo 3.2 é uma f-aproximação para o problema da cobertura por conjuntos.*

Demonstração. Assim como todo PL, nosso programa dual pode ser resolvido em tempo polinomial no tamanho do PL. Como nossa construção dá um programa de tamanho polinomial no tamanho da instância original, a obtenção de y^* é um passo do algoritmo que gasta tempo polinomial.

Além disso, é evidente que nossa construção de uma cobertura por conjuntos a partir da solução do PL também pode ser feita em tempo polinomial. Assim, o algoritmo é polinomial, bastando provar sua garantia de desempenho.

A ideia central para isso é o seguinte argumento de "cobrança": quando escolhemos um conjunto S_j para estar na cobertura, nós "pagamos" por ele cobrando y^*_i de cada um de seus elementos e_i (mesmo os que já estavam cobertos).

Cada elemento e_i paga sua taxa de cobertura y^*_i no máximo uma vez para cada conjunto que o contém, e, portanto, no máximo f vezes. Então, o custo total pago é no máximo $f \cdot \sum_{i=1}^{m} y^*_i$, ou seja, f vezes o valor de uma solução ótima.

Como $j \in I'$ apenas se $w_j = \sum_{i:e_i \in S_j} y_i^*$, temos que o custo da cobertura I' é

$$\begin{aligned}
\sum_{j \in I'} w_j &= \sum_{j \in I'} \sum_{i:e_i \in S_j} y_i^* \\
&= \sum_{i=1}^{n} |\{j \in I' : e_i \in S_j\}| \cdot y_i^* \\
&\leq \sum_{i=1}^{n} f \cdot y_i^* \\
&= f \sum_{i=1}^{n} y_i^* \\
&\leq f \cdot \mathrm{OPT},
\end{aligned}$$

onde a última desigualdade vem da dualidade fraca. □

É possível mostrar que esse algoritmo não pode se sair melhor do que o da Seção 3.3. Mais precisamente, podemos mostrar que se I indexa a solução devolvida pelo algoritmo de arredondamento daquela seção, então $I \subseteq I'$. Como veremos a seguir, isso vem da propriedade de *folgas complementares* em soluções ótimas de um par primal-dual.

A dualidade fraca para o par de programas lineares que estamos considerando pode ser obtida pela seguinte cadeia de igualdades e desigualdades:

$$\begin{aligned}
\sum_{i=1}^{n} y_i &\leq \sum_{i=1}^{n} y_i \sum_{j:e_i \in S_j} x_j && (3.3) \\
&= \sum_{j=1}^{m} x_j \sum_{i:e_i \in S_j} y_i \\
&\leq \sum_{j=1}^{m} x_j w_j. && (3.4)
\end{aligned}$$

A desigualdade (3.3) se deve ao fato de que $\sum_{j:e_i \in S_j}^{m} x_j \geq 1$ para cada i, para x viável. A desigualdade (3.4) se deve ao fato de que $\sum_{i:e_i \in S_j}^{n} y_i \leq w_j$ para cada j, para y viável.

Para soluções ótimas x^* e y^*, a dualidade forte implica que seus valores são iguais, isto é, $\sum_{i=1}^{n} y_i^* = \sum_{j=1}^{m} w_j x_j^*$. Então, as desigualdades (3.3) e (3.4) devem ser obedecidas com igualdade.

A única maneira de isso ser satisfeito é que, sempre que $y_i^* > 0$, valha que $\sum_{j:e_j \in S_j} x_j^* = 1$, e sempre que $x_j^* > 0$, então $\sum_{i:e_i \in S_j} y_i^* = w_j$. Isto é, sempre que uma variável do PL (primal ou dual) é não nula, a restrição correspondente no dual ou primal deve ser satisfeita sem folgas (i.e., com igualdade).

Tal relação é conhecida como a *condição de folgas complementares*. Assim, se x^* e y^* são soluções ótimas, as folgas complementares devem ser satisfeitas. A recíproca é verdadeira: se $\hat{x}$ e $\hat{y}$ são soluções viáveis primal e dual, respectivamente, e as folgas complementares são satisfeitas, segue que $\hat{x}$ e $\hat{y}$ têm valores iguais e, portanto, são soluções ótimas.

No algoritmo de arredondamento primal, incluímos j em I se $x_j^* \geq 1/f$. Assim, $j \in I$ implica $x_j^* > 0$, que por sua vez implica $\sum_{i:e_i \in S_j} y_i^* = w_j$, satisfazendo nosso critério para incluir j em I'.

Logo, $j \in I$ implica $j \in I'$. Ou seja, $I \subseteq I'$.

3.5 O método primal-dual

Uma das desvantagens dos algoritmos apresentados nas duas seções anteriores é que ambas envolvem a resolução de um PL. Embora programas lineares possam ser resolvidos em tempo polinomial e haja algoritmos que são muito rápidos na prática (Simplex), algoritmos que resolvem um problema específico são, muitas vezes, muito mais rápidos.

A ideia básica do algoritmo dessa seção é que o algoritmo de arredondamento dual da seção anterior usa poucas propriedades da *otimalidade* da solução dual y. Em vez de realmente resolver o PL dual, podemos simplesmente construir uma solução viável com tais propriedades. Construir tal solução dual pode ser muito mais rápido do que realmente resolver o PL dual e portanto levará a um algoritmo mais rápido.

O algoritmo da seção anterior usa as seguintes propriedades:

1. $\sum_{i=1}^{n} y_i \leq \text{OPT}$, o que é verdade para qualquer y dual viável.
2. Incluímos j em I' exatamente quando $\sum_{i:e_i \in S_j} y_i = w_j$, e então I' indexa uma cobertura por conjuntos.

Esses dois fatos juntos foram usados para provar que o custo de I' não é maior do que f vezes o valor ótimo.

De fato, é a *demonstração* do Lema 3.5 (que diz que construímos uma cobertura por conjuntos válida) que mostra como obter um algoritmo que constrói uma solução dual.

Considere uma solução viável dual qualquer y, e seja T o conjunto de índices de todas as restrições do dual satisfeitas com igualdade, isto é, $T = \{j : \sum_{i:e_i \in S_j} y_i = w_j\}$.

Se T indexa uma cobertura por conjuntos, nada temos a fazer. Senão, então algum elemento e_i não está coberto, e, como mostrado na prova do Lema 3.5, é possível obter uma solução de maior valor aumentando y_i de um $\varepsilon > 0$. Mais especificamente, podemos aumentar y_i de $\min_{j:e_i \in S_j}(w_j - \sum_{k:e_k \in S_j} y_k)$, que é um valor positivo já que e_i não está coberto. Para um j que minimiza a expressão vale que a restrição referente ao S_j passa a ser satisfeita com igualdade. Além disso, a solução dual modificada continua viável. Assim, podemos adicionar um tal j a T, e assim o elemento e_i agora está coberto pelos conjuntos indexados por T. Repetimos o processo até que T seja uma cobertura por conjuntos.

Como um e_i adicional é coberto a cada passo, o processo é repetido no máximo n vezes. Para completar a descrição do algoritmo, só precisamos dar uma solução dual viável inicial. É fácil encontrar uma tal solução, visto que o vetor nulo satisfaz as restrições do PL dual.

Esse tipo de algoritmo é chamado de *primal-dual* por analogia ao método primal-dual usado em algoritmos para outros problemas de otimização, como programação linear, fluxo máximo em redes, caminhos mais curtos, etc. Um algoritmo primal-dual começa com viabilidade dual (uma solução dual viável), e usa a informação dual para inferir uma solução primal, não necessariamente viável. Se a solução primal realmente não é viável, a solução dual é modificada de modo a incrementar o seu valor, e esse processo é repetido. No nosso caso, temos:

Teorema 3.7. *O algoritmo 3.3 é uma f-aproximação para o problema da cobertura por conjuntos, onde f é o máximo número de ocorrências de um e_i nos conjuntos S_j.*

Demonstração. O algoritmo demora no máximo n passos para acabar, uma vez que a cada passo pelo menos um elemento passa a ser coberto por I. Como cada passo leva tempo polinomial, o algoritmo é polinomial.

Algoritmo 3.3: Algoritmo primal-dual para a cobertura por conjuntos.

Entrada: Elementos e_i, conjuntos S_j e seus pesos w_j
Saída: Indexação I de uma cobertura por conjuntos

$y \leftarrow 0$;
$I \leftarrow \emptyset$;
enquanto I *não é uma cobertura* **faça**
  Seja i tal que $e_i \notin \bigcup_{j \in I} S_j$;
  $\varepsilon \leftarrow \min_{j:e_i \in S_j} (w_j - \sum_{k:e_k \in S_j} y_k)$;
  $y_i \leftarrow y_i + \varepsilon$;
  Seja l com $e_i \in S_l$ tal que $\sum_{j:e_j \in S_l} y_j = w_l$;
  $I \leftarrow I \cup \{l\}$;
fim

Como sempre alteramos y de modo que não perdemos sua viabilidade dual, segue que, ao fim do algoritmo, y é viável.

Observe que, ao final do algoritmo, I é o conjunto $\{S_j : \sum_{i:e_i \in S_j} y_i = w_j\}$, que é uma cobertura por conjuntos, pela condição de parada do algoritmo.

É fácil, então, ver que a demonstração do Teorema 3.6 vale com y no lugar do y^*. Portanto, a solução produzida pelo algoritmo primal-dual tem valor no máximo $f \cdot \mathrm{OPT}$. □

3.6 Um algoritmo guloso

Até aqui, as técnicas que vimos levaram, todas, a algoritmos de aproximação com garantia de desempenho igual a f (conforme definido em 3.1). Estudaremos, agora, outro tipo de técnica. Algoritmos *gulosos* são algoritmos que, a cada passo, tomam uma decisão que seja *localmente* ótima ("gulosa"), fazendo uma escolha que otimiza essa decisão em particular, sem preocupar-se com o fato de isso levar ou não a um ótimo global. Há problemas que podem ser resolvidos por algoritmos gulosos, isto é, problemas para os quais existem algoritmos gulosos que encontram necessariamente uma solução ótima. Este não será o caso dos nossos problemas, que são difíceis. No entanto, algoritmos gulosos são, muitas vezes, heurísticas boas e bastante usadas. No caso do problema da cobertura por conjuntos, veremos que um guloso nos dará uma aproximação com garantia de desempenho que pode ser significativamente melhor que f.

Algoritmo 3.4: Um algoritmo guloso para o problema da cobertura por conjuntos.

$I \leftarrow \emptyset$;
$\hat{S}_j \leftarrow S_j, \quad \forall j$;
enquanto I *não é uma cobertura* **faça**
  $l \leftarrow \arg\min_{j:\hat{S}_j \neq \emptyset} \frac{w_j}{|\hat{S}_j|}$;
  $I \leftarrow I \cup \{l\}$;
  $\hat{S}_j \leftarrow \hat{S}_j \setminus S_l, \quad \forall j$;
fim

Este é um algoritmo guloso bastante natural para o problema da cobertura por conjuntos. A cada passo, escolhemos um conjunto que nos dê o melhor "custo-benefício", isto é, um conjunto que minimiza a razão entre seu peso e a quantidade de elementos ainda não cobertos que ele contém. No caso de empate, pegamos arbitrariamente um dos conjuntos que atinja esse mínimo. Continuamos escolhendo novos conjuntos até que todos os elementos estejam cobertos.

É claro que isso nos dá um algoritmo polinomial: a cada passo, um novo S_j é adicionado à cobertura.

Então, não podemos levar mais do que m passos. Além disso, no mínimo um elemento é coberto, então também certamente não precisamos de mais do que n passos. A cada passo, computamos $O(m)$ razões, cada uma em tempo constante. Portanto, nosso algoritmo tem complexidade $O(m \cdot \min\{m, n\})$.

Para prosseguir, precisaremos de uma nova notação e um fato.

Seja H_k o k-ésimo *número harmônico*, isto é,

$$H_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k}.$$

Note que H_k é aproximadamente $\int_1^k \frac{1}{x} = \ln k - \ln 1 = \ln k$, e portanto $H_k \approx \ln k$. Mais especificamente, $1 + \int_1^k \frac{1}{x} \leq H_k \leq \int_1^{k+1} \frac{1}{x}$, donde pode-se concluir que $1 + \ln k \leq H_n \leq \ln(k+1)$.

Fato 3.8. *Dados $a_1, \ldots, a_k$ e $b_1, \ldots, b_k$ positivos, vale que*

$$\min_{i=1,\ldots,k} \frac{a_i}{b_i} \leq \frac{\sum_{i=1}^k a_i}{\sum_{i=1}^k b_i} \leq \max_{i=1,\ldots,k} \frac{a_i}{b_i}.$$

Demonstração. Mostraremos apenas a primeira das duas desigualdades. A outra pode ser provada por uma argumentação similar.

Sejam p, q, r, s reais positivos. Se $\frac{p}{q} \leq \frac{r}{s}$, então $\frac{p}{q} \leq \frac{p+r}{q+s}$, pois

$$\begin{aligned}
\frac{p}{q} \leq \frac{r}{s} &\iff ps \leq qr (\text{pois } s, q \geq 0) \\
&\iff ps + pq \leq qr + pq \\
&\iff p(q+s) \leq q(p+r) \\
&\iff \frac{p}{q} \leq \frac{p+r}{q+s} (\text{pois } q > 0 \text{ e } q + s > 0.
\end{aligned}$$

Isso nos permite realizar uma prova por indução em k.

Para $k = 1$, temos que $\frac{a_1}{b_1} = \min_{i=1,\ldots,k} \frac{a_i}{b_i} = \frac{\sum_{i=1}^k a_i}{\sum_{i=1}^k b_i}$.

Seja, agora, $k > 1$ e suponha que a desigualdade valha para $k-1$. Seja l um índice tal que $\frac{a_l}{b_l} = \min_{i=1,\ldots,k} \frac{a_i}{b_i}$. Então, $\frac{a_l}{b_l} \leq \min_{i \neq l} \frac{a_i}{b_i}$ e, por hipótese de indução, $\min_{i \neq l} \frac{a_i}{b_i} \leq \frac{\sum_{i \neq l} a_i}{\sum_{i \neq l} b_i}$.

Então, $\frac{a_l}{b_l} \leq \frac{\sum_{i \neq l} a_i}{\sum_{i \neq l} b_i}$ e, pela propriedade que provamos no início da demonstração, segue que

$$\frac{a_l}{b_l} \leq \frac{a_l + \sum_{i \neq l} a_i}{b_l + \sum_{i \neq l} b_i} = \frac{\sum_{i=1}^k a_i}{\sum_{i=1}^k b_i}.$$

□

Teorema 3.9. *O algoritmo 3.4 é uma H_n-aproximação para o problema da cobertura por conjuntos.*

Demonstração. A intuição básica para a análise do algoritmo é a seguinte. Seja OPT o valor de uma solução ótima S^* para o problema da cobertura por conjuntos. Sabemos que uma solução ótima cobre todos os n elementos com uma solução de peso OPT. Assim, pelo menos um dos conjuntos S_j de uma solução ótima cobre seus elementos com peso médio de no máximo OPT$/n$. Isso porque se todos os

elementos tivessem peso médio superior a esse valor, a soma destes pesos seria superior a OPT, o que é um absurdo, visto que tal soma é o peso de S^*.

Da mesma forma, após k elementos serem cobertos, a solução ótima consegue cobrir os $n-k$ restantes com uma solução de peso OPT, o que implica que algum conjunto cobre seus elementos ainda não cobertos com peso médio de no máximo $\mathrm{OPT}/(n-k)$. Então, em geral, o guloso paga por volta de $\mathrm{OPT}/(n-k+1)$ para cobrir o k-ésimo elemento, dando uma garantia de desempenho igual a $\sum_{k=1}^{n} \frac{1}{n-k+1} = H_n$.

Vamos formalizar essa intuição. Seja n_k o número de elementos que ainda não foram cobertos ao começo da k-ésima iteração. Se o algoritmo acaba após l iterações, então $n_1 = n$ e $n_{l+1} = 0$. Tome uma iteração k qualquer, e seja I_k o conjunto de índices dos conjuntos escolhidos nas iterações 1 a $k-1$. Para cada $j = 1, \ldots, m$, seja $\hat{S}_j$ o conjunto de elementos não cobertos em S_j no começo dessa iteração, isto é, $\hat{S}_j = S_j - \bigcup_{p \in I_k} S_p$. Afirmamos que, para o conjunto de índice j escolhido na iteração k, vale que

$$w_j \leq \frac{n_k - n_{k+1}}{n_k} \mathrm{OPT}. \tag{3.5}$$

Dada a desigualdade 3.5, podemos provar o teorema. Seja I o conjunto de índices dos conjuntos na solução final, ou seja, $I = I_{l+1}$. Então,

$$\begin{aligned}
\sum_{j \in I} w_j &\leq \sum_{k=1}^{l} \frac{n_k - n_{k+1}}{n_k} \mathrm{OPT} \\
&\leq \mathrm{OPT} \cdot \sum_{k=1}^{l} \left(\frac{1}{n_k} + \frac{1}{n_k - 1} + \ldots + \frac{1}{n_{k+1} + 1} \right) \qquad (3.6) \\
&= \mathrm{OPT} \cdot \sum_{i=1}^{n} \frac{1}{i} \\
&= H_n \cdot \mathrm{OPT}, \qquad (3.7)
\end{aligned}$$

onde a desigualdade 3.6 vem do fato de que $\frac{1}{n_k} \leq \frac{1}{n_k - i}$ para $0 \leq i \leq n_k$.

Para provar a desigualdade 3.5, vamos primeiro argumentar que, na k-ésima iteração,

$$\min_{j : \hat{S}_j \neq \emptyset} \frac{w_j}{|\hat{S}_j|} \leq \frac{\mathrm{OPT}}{n_k}. \tag{3.8}$$

Sendo O o conjunto de índices dos conjuntos em uma solução ótima, então a desigualdade 3.8 segue do Fato 3.8, observando-se que

$$\begin{aligned}
\min_{j : \hat{S}_j \neq \emptyset} \frac{w_j}{|\hat{S}_j|} &\leq \min_{j \in O} \frac{w_j}{|\hat{S}_j|} \\
&\leq \frac{\sum_{j \in O} w_j}{\sum_{j \in O} |\hat{S}_j|} \\
&= \frac{\mathrm{OPT}}{\sum_{j \in O} |\hat{S}_j|} \leq \frac{\mathrm{OPT}}{n_k},
\end{aligned}$$

onde a última desigualdade segue do fato de que como O é uma cobertura por conjuntos, $n_k \leq n$ certamente é no máximo igual ao número de elementos cobertos por O.

Seja j o índice de um conjunto que minimiza essa razão, de modo que $\frac{w_j}{|\hat{S}_j|} \leq \frac{\mathrm{OPT}}{n_k}$. Se adicionarmos o conjunto S_j à nossa solução, então haverá $|\hat{S}_j|$ elementos não cobertos a menos, de forma que $n_{k+1} =$

$n_k - |\hat{S}_j|$. Logo,

$$w_j \leq \frac{|\hat{S}_j|\text{OPT}}{n_k} = \frac{n_k - n_{k+1}}{n_k}\text{OPT}.$$

□

Podemos melhorar um pouco a garantia de desempenho do algoritmo usando o PL dual das seções anteriores na nossa análise. Seja g o tamanho máximo de um conjunto S_j, isto é, $g = \max_j |S_j|$. Lembremos que Z^*_{PL} é o valor ótimo do nosso PL. O teorema a seguir implica diretamente que o algoritmo guloso é uma H_g-aproximação, uma vez que $Z^*_{PL} \leq \text{OPT}$.

Teorema 3.10. *O algoritmo guloso desta seção devolve uma solução indexada por I tal que $\sum_{j\in I} w_j \leq H_g \cdot Z^*_{PL}$, onde g é o tamanho máximo de um conjunto S_j.*

Demonstração. Para provar o teorema, construiremos uma solução dual y *inviável* tal que $\sum_{j\in I} w_j = \sum_{i=1}^n y_i$. Mostraremos, então, que $y' = \frac{1}{H_g} y$ é uma solução dual viável. Pelo teorema de dualidade fraca, $\sum_{i=1}^n y'_i \leq Z^*_{PL}$, de modo que $\sum_{j\in I} w_j = \sum_{i=1}^n y_i = H_g \sum_{i=1}^n y \leq H_g \cdot \text{OPT}$.

Essa técnica de construir uma solução dual inviável com o mesmo custo da solução primal viável construída e que, multiplicada por um único escalar, torna-se dual viável, chama-se *dual-fitting*.

Para construir a solução inviável y, suponha que escolhamos adicionar S_j à solução na iteração k. Então, para cada $e_i \in \hat{S}_j$, estabelecemos $y_i = \frac{w_j}{|\hat{S}_j|}$. Como cada $e_i \in \hat{S}_j$ está descoberto na iteração k e então continua coberto durante as próximas iterações do algoritmo (pois adicionamos S_j à solução), a variável y_i tem seu valor estabelecido exatamente uma vez; em particular, seu valor é definido na iteração em que o elemento e_i é coberto. Além disso, $w_j = \sum_{i:e_i\in\hat{S}_j} y_i$; isto é, o peso do conjunto S_j escolhido na k-ésima iteração é igual À soma dos y_i dos elementos de S_j que passam a ser cobertos nessa iteração. Isso implica que $\sum_{j\in I} w_j = \sum_{i=1}^n y_i$.

Falta provar que a solução dual $y' = \frac{1}{H_n} y$ é viável. Precisamos mostrar que, para cada S_j, $\sum_{i:e_i\in S_j} y'_i \leq w_j$. Tome um conjunto arbitrário S_j. Seja a_k o número de elementos nesse conjunto que estão descobertos no início da k-ésima iteração, de modo que $a_1 = |S_j|$, e $a_{l+1} = 0$. Seja A_k o conjunto dos elementos de S_j que passam a ser cobertos na k-ésima iteração, de modo que $|A_k| = a_k - a_{k+1}$. Se o conjunto S_p é escolhido na k-ésima iteração, então, para cada elemento $e_i \in A_k$, vale que

$$y'_i = \frac{w_p}{H_g|\hat{S}_p|} \leq \frac{w_j}{H_g a_k},$$

onde $\hat{S}_p$ é o conjunto dos elementos de S_p descobertos no início da k-ésima iteração. A desigualdade segue do fato de que, se S_p é escolhido na k-ésima iteração, ele deve minimizar a razão entre seu peso e

o número de elementos descobertos que ele contém. Assim,

$$\begin{aligned}
\sum_{i:e_i \in S_j} y'_i &= \sum_{k=1}^{l} \sum_{i:e_i \in A_k} y'_i \\
&\leq \sum_{k=1}^{l} (a_k - a_{k+1}) \frac{w_j}{H_g a_k} \\
&\leq \frac{w_j}{H_g} \sum_{k=1}^{l} \frac{a_k - a_{k+1}}{a_k} \\
&\leq \frac{w_j}{H_g} \sum_{k=1}^{l} \left(\frac{1}{a_k} + \frac{1}{a_k - 1} + \ldots + \frac{1}{a_{k+1} + 1} \right) \\
&\leq \frac{w_j}{H_g} \sum_{i=1}^{|S_j|} \frac{1}{i} \\
&= \frac{w_j}{H_g} H_{|S_j|} \\
&\leq w_j,
\end{aligned}$$

onde a desigualdade final segue de que $|S_j| \leq g$. Aqui vemos o motivo de dividir a solução dual por H_g: sabemos que $H_|S_j| \leq H_g$ para todo j. □

De fato, nenhuma aproximação para o problema da cobertura por conjuntos pode ter garantia de desempenho melhor do que H_n, sob uma hipótese um pouco mais forte do que P ≠ NP.

Teorema 3.11. *Se existir uma $c \ln n$-aproximação para o problema da cobertura por conjuntos sem peso para alguma constante $c < 1$, então existe um algoritmo determinístico de tempo $O(n^{O(\log \log n)})$ para todo problema NP-completo.*

Teorema 3.12. *Existe uma constante $c > 0$ tal que a existência de uma $c \ln n$-aproximação para a cobertura por conjuntos implica que $P = NP$.*

Teoremas como esses são por vezes chamados de *resultados de inaproximabilidade* ou teoremas de *dificuldade*, uma vez que mostram que é NP-difícil fornecer soluções de valor próximo ao ótimo com certas garantias de desempenho.

As f-aproximações para a cobertura por conjuntos nos dão, para o caso especial da cobertura por vértices, uma 2-aproximação. Nenhum algoritmo com garantia melhor que essa é conhecido, e dois teoremas sobre a dificuldade da cobertura por vértices foram demonstrados.

Teorema 3.13. *Se existir uma α-aproximação para o problema da cobertura por vértices com $\alpha < 10\sqrt{5} - 21 \approx 1{,}36$, então $P = NP$.*

O teorema a seguir menciona uma conjectura chamada de *conjectura dos jogos únicos*, que diz, grosso modo, que um determinado problema (chamado de problema dos jogos únicos) é NP-difícil.

Teorema 3.14. *Admitindo que a conjectura dos jogos únicos seja correta, se existir uma α-aproximação para a cobertura por vértices com constante $\alpha < 2$, então $P = NP$.*

Então, admitindo-se que P ≠ NP e o problema dos jogos únicos é NP-difícil, já encontramos essencialmente a melhor aproximação possível para o problema da cobertura por vértices.

3.7 Um algoritmo de arredondamento aleatorizado

Nesta seção trataremos de uma última técnica para a obtenção de um algoritmo de aproximação para o problema da cobertura por conjuntos. Embora o algoritmo não tenha uma garantia de desempenho melhor que a do guloso que estudamos (algoritmo 3.4), além de ser mais lento do que este, nós o estudaremos por ser um exemplo introdutório de algoritmo de aproximação aleatorizado.

Novamente, o algoritmo irá resolver um PL obtido da relaxação do PLI que modela o problema, e então arredondar a solução fracionária para uma inteira. Mas, em vez de fazê-lo de maneira determinística, o algoritmo irá fazê-lo de maneira aleatória, usando a técnica do *arredondamento aleatorizado*. Seja x^* uma solução ótima para o PL. Gostaríamos de arredondar os valores fracionários de x^* ou para 0 ou para 1 de modo a obter uma solução $\hat{x}$ para o PLI sem que o custo desta seja muito mais alto. A ideia central aqui é que iremos interpretar o valor fracionário x_j^* como a probabilidade de que $\hat{x}_j$ deveria ser 1. Assim, cada S_j é incluído na nossa solução com probabilidade x_j^*, e esses m eventos (a inclusão de S_j na solução) são eventos aleatórios independentes.

Seja X_j uma variável aleatória que vale 1, se o conjunto S_j é incluído na solução, e 0, caso contrário. Então o valor esperado da solução é

$$E\left[\sum_{j=1}^{m} w_j X_j\right] = \sum_{j=1}^{m} w_j \Pr[X_j = 1] = \sum_{j=1}^{m} w_j x_j^* = Z_{PL}^*,$$

ou seja, o valor do PL, que não é maior que OPT! No entanto, veremos que é bem provável que a solução não seja uma cobertura por conjuntos. Mesmo assim, isso ilustra bem o motivo de aproximações aleatorizadas serem capazes de fornecer aproximações bastante boas em alguns casos.

Calculemos agora a probabilidade de que um dado elemento e_i não esteja coberto por tal procedimento. É a probabilidade de que nenhum dos conjuntos contendo e_i estejam na solução, isto, é,

$$\prod_{j:e_i \in S_j} (1 - x_j^*).$$

Podemos obter um limitante superior para essa probabilidade usando o fato de que $1 - x \leq e^{-x}$ para qualquer $x \geq 0$, onde e é a base do logaritmo natural. Então,

$$\begin{aligned}
\Pr[e_i \text{ não estar coberto}] &= \prod_{j:e_i \in S_j} (1 - x_j^*) \\
&\leq \prod_{j:e_i \in S_j} e^{-x_j^*} \\
&= e^{-\sum_{j:e_i \in Sj} x_j^*} \\
&\leq e^{-1},
\end{aligned}$$

onde a última desigualdade vem da restrição do PL de que $\sum_{j:e_i \in S_j} x_j^* \geq 1$. Embora e^{-1} seja um limitante superior para a probabilidade de que um dado elemento não esteja coberto, é possível chegar arbitrariamente perto desse limitante, de modo que no pior caso é bem provável que esse procedimento de aproximação não produza uma cobertura por conjuntos.

Quão pequena essa probabilidade deve ser para que seja bastante provável que se produza uma cobertura? E mais importante, qual é a noção "correta" de "bastante provável"? Uma das possíveis maneiras de pensar nisso é impor uma garantia conforme nosso foco em algoritmos de tempo polinomial. Suponha que, para qualquer constante c, fôssemos capazes de elaborar um algoritmo polinomial cuja probabilidade de falha fosse no máximo uma polinomial inversa n^{-c}; então, dizemos ter um algoritmo que funciona *com alta probabilidade*.

Mais precisamente, teríamos uma família de algoritmos, uma vez que pode ser necessário dar algoritmos progressivamente mais lentos ou com piores garantias de desempenho para obter resultados analogamente mais seguros. Se pudéssemos elaborar um procedimento aleatorizado tal que

$$\Pr[e_i \text{ não está coberto}] \leq \frac{1}{n^c}$$

para alguma constante $c \geq 2$, então

$$\begin{aligned} \Pr[\text{existe um elemento descoberto}] &\leq \sum_{i=1}^{n} \Pr[e_i \text{ descoberto}] \\ &\leq n \cdot \frac{1}{n^c} \\ &= \frac{1}{n^{c-1}}, \end{aligned} \tag{3.9}$$

e teríamos uma cobertura com alta probabilidade. De fato, conseguimos atingir tal limitante da seguinte forma: para cada S_j, imaginamos uma moeda que dá cara com probabilidade x_j^*, e nós lançamos a moeda $c \ln n$ vezes. Incluímos S_j na solução se, e somente se, obtivermos cara em um dos lançamentos da moeda. Então, a probabilidade de S_j não ser incluído é $(1 - x_j^*)^{c \ln n}$, e

$$\begin{aligned} \Pr[e_i \text{ descoberto}] &= \prod_{j:e_i \in S_j} (1 - x_j^*)^{c \ln n} \\ &\leq \prod_{j:e_i \in S_j} e^{-x_j^*(c \ln n)} \\ &= e^{(-c \ln n) \sum_{j:e_i \in S_j} x_j^*} \\ &\leq \frac{1}{n^c}, \end{aligned} \tag{3.10}$$

como desejávamos.

Provaremos que o algoritmo tem um bom valor esperado dado que produza uma cobertura por conjuntos.

Teorema 3.15. *O algoritmo é uma $O(\ln n)$-aproximação aleatorizada que produz uma cobertura por conjuntos com alta probabilidade.*

Demonstração. Vejamos que o algoritmo é polinomial. Primeiro, resolve um PL, o que pode ser feito em tempo polinomial. Depois, para cada S_j, simula o lançamento de $c \ln n$ moedas, onde c é constante. Admitindo que cada lançamento seja feito em tempo constante, esse passo do algoritmo consome tempo $O(m \ln n)$.

Seja $p_j(x_j^*)$ a probabilidade de um dado S_j estar na solução em função de x_j^*. Pela construção do algoritmo, sabemos que

$$p_j(x_j^*) = 1 - (1 - x_j^*)^{c \ln n}.$$

Observe que, se $x_j^* \in [0, 1]$ e $c \ln n \geq 1$, então a derivada p_j' em x_j^* é limitada por

$$p_j'(x_j^*) = (c \ln n)(1 - x_j^*)^{(c \ln n)-1} \leq (c \ln n).$$

Então, como $p_j(0) = 0$ e o crescimento da função p_j é limitado superiormente por $c \ln n$ no intervalo $[0, 1]$, segue que $p_j(x_j^*) \leq (c \ln n) x_j^*$ no intervalo $[0, 1]$. Se X_j é uma variável aleatória que vale 1, se o

conjunto S_j está na solução, e 0, caso contrário, então o valor esperado do procedimento aleatório é

$$\begin{aligned} E\left[\sum_{j=1}^{m} w_j X_j\right] &= \sum_{j=1}^{m} w_j \Pr[X_j = 1] \\ &\leq \sum_{j=1}^{m} w_j (c \ln n) x_j^* \\ &= (c \ln n) \sum_{j=1}^{m} w_j x_j^* = (c \ln n) Z_{PL}^*. \end{aligned}$$

No entanto, gostaríamos de limitar o valor esperado da solução dado que uma cobertura foi produzida. Seja F o evento de a solução obtida pelo procedimento ser uma cobertura, e seja $\bar{F}$ seu complemento. Sabemos pela discussão anterior (em 3.9 e 3.10) que $\Pr[F] \geq 1 - \frac{1}{n^{c-1}}$, e também sabemos que

$$E\left[\sum_{j=1}^{m} w_j X_j\right] = E\left[\sum_{j=1}^{m} w_j X_j \middle| F\right] \Pr[F] + E\left[\sum_{j=1}^{m} w_j X_j \middle| \bar{F}\right] \Pr[\bar{F}].$$

Como $w_j \leq 0$ para todo j,

$$E\left[\sum_{j=1}^{m} w_j X_j \middle| \bar{F}\right] \geq 0.$$

Assim,

$$\begin{aligned} E\left[\sum_{j=1}^{m} w_j X_j \middle| F\right] &= \frac{1}{\Pr[F]} \left(E\left[\sum_{j=1}^{m} w_j X_j\right] - E\left[\sum_{j=1}^{m} w_j X_j \middle| \bar{F}\right] \Pr[\bar{F}] \right) \\ &\leq \frac{1}{\Pr[F]} \cdot E\left[\sum_{j=1}^{m} w_j X_j\right] \\ &\leq \frac{1}{1 - \frac{1}{n^{c-1}}} \cdot E\left[\sum_{j=1}^{m} w_j X_j\right] \\ &= \frac{(c \ln n) Z_{PL}^*}{1 - \frac{1}{n^{c-1}}} \\ &\leq 2c(\ln n) Z_{PL}^* \end{aligned}$$

para $n \geq 2$ e $c \geq 2$, que satisfazem $(1 - \frac{1}{n^{c-1}}) \geq \frac{1}{2}$. □

Embora nesse caso haja uma aproximação mais simples e rápida com uma melhor garantia de desempenho, algoritmos aleatorizados são, por vezes, mais fáceis de descrever e analisar do que algoritmos determinísticos. Muitos algoritmos aleatorizados admitem uma variante determinística que atinge a garantia de desempenho esperada do algoritmo original. No entanto, em alguns casos a versão determinística é difícil de descrever. Também pode ocorrer de a única maneira conhecida de analisar um algoritmo é analisar sua versão aleatorizada.

Com isto, encerramos nossa introdução a algoritmos de aproximação.

4 Problemas de *Clustering*

Nessa seção, trataremos do problema de particionar elementos em conjuntos (*clusters*) de acordo com suas semelhanças. Trata-se de um problema bastante genérico que aparece em diversas aplicações,

e por isso existem muitas variantes desse problema. A principal diferença entre elas é a função objetivo. Uma das formas mais simples desse problema é o chamado *k-center*, (ou problema dos *k*-centros) em que, dada uma coleção S de pontos com distâncias definidas para cada par de pontos, desejamos encontrar um conjunto $C = \{c_1, \ldots, c_k\} \subseteq S$, cujos elementos chamamos de *centros*, de forma que a maior distância de um ponto $p \in S$ ao centro mais próximo de p seja mínima. Note que nessa formulação temos implicitamente uma partição de S em k conjuntos $(B_1, \ldots, B_k)$ dada por $B_k = \{p \in S\colon$ o centro mais proximo de p é $c_k\}$. Se para um dado p houver mais de um centro que atinge a distância mínima, escolha um deles ao acaso para determinar o *cluster* de p.

Uma possível interpretação desse problema seria, por exemplo, a automatização de um processo de divisão em categorias. Podemos imaginar, como exemplo, o uso de dados de compras de consumidores. Conforme os tipos e quantidades de produtos comprados, pode-se elaborar uma maneira de medir a "distância" entre dois consumidores. Nesse contexto, o problema dos k centros pode ser visto como o problema de estabelecer k "perfis de consumidor" de maneira que os consumidores incluídos em cada perfil sejam um grupo o menos heterogêneo possível. Poderíamos usar a mesma idéia para tentar encontrar formas de categorizar alunos de um curso, deputados de uma casa legislativa, etc., conforme os critérios de comparação e representações matemáticas que pareçam convenientes ou interessantes.

Outra interpretação para esse problema é como um problema de *localização de instalações* (ou *facility location*). Os pontos representam um conjunto de locais que precisam ser supridos por certo tipo de instalação, e suas distâncias são definidas pela rota de menor custo de um ponto a outro — onde o custo pode ser a distância, o tempo, ou uma combinação desses e outros fatores. Deseja-se abrir tais instalações em k dos pontos de maneira que nenhum ponto esteja muito longe da instalação mais próxima. Por exemplo, imagine que temos um conjunto de hospitais públicos, e dispomos de recursos para abrir centros de tratamento de câncer em k deles. Desejamos saber em quais hospitais instalar tais centros de modo a minimizar o maior tempo de transporte entre um hospital comum e seu centro de tratamento mais próximo, a fim de que a transferência mais demorada de um paciente seja tão rápida quanto possível.

A interpretação do *clustering* como um problema de localização de instalações naturalmente leva a variantes do problema em que a possibilidade de falha de algumas das instalações é considerada.

A seguir, estudaremos algumas variações do problema e algoritmos de aproximação para resolvê-las.

4.1 Minimização da maior distância entre elementos de um mesmo *cluster*

Veremos aqui alguns resultados para problemas de *clustering* estudados por Gonzalez [5], adaptando e por vezes complementando seu conteúdo. Trataremos do problema de minimizar a maior distância entre qualquer par de pontos que esteja dentro de um mesmo *cluster*; veremos a complexidade de algumas de suas variações e um algoritmo de aproximação.

Seja $G = (V, E)$ um grafo não direcionado com pesos não-negativos nas arestas dados pela função $w\colon E \to \mathbb{R}_0^+$. Um ***k-split*** de G é uma partição de V em k conjuntos $(B_1, B_2, \ldots, B_k)$. Cada B_i é um **cluster**. Cada *k-split* tem um valor $\mathrm{val}\,(B_1, \ldots, B_k)$ dado pela função objetivo "MM" (max-max), definida por:

$$\mathrm{val}\,(B_1, \ldots, B_k) = \max\{M_1, \ldots, M_k\},$$

onde

$$M_i = \max\{w(e)\colon e \in E \text{ tem as pontas em } M_i\}.$$

Em geral, pensaremos no problema de otimização $\min_{(B_1,\ldots,B_k)} \mathrm{val}\,(B_1, \ldots, B_k)$.

Diremos que estamos tratando do *caso métrico* caso o grafo seja completo e suas arestas satisfaçam a desigualdade triangular. Isto é,

$$w\,(ij) \leq w\,(ik) + w\,(kj)\,, \quad \forall i \neq j \neq k \in V.$$

Note que, na verdade, não precisamos exigir que o grafo seja completo – caso não seja, para cada $ij \notin E$ podemos incluir ij em E estabelecendo $W(ij) = d(i,j)$, onde $d(i,j)$ é o custo do menor caminho de i a j no grafo. Claramente, o grafo continua respeitando a desigualdade triangular.

Adotaremos a notação de [5] para as variações desse problema. Iremos nos referir ao problema de *clustering* α-βMM, onde α é o número de *clusters* que desejamos formar e β representa uma descrição do espaço em que estão os pontos. Se o número de *clusters* for um parâmetro de entrada para o problema, então escrevemos $\alpha = k$. Se os pontos a serem divididos em *clusters* estão em um espaço euclideano de dimensão m, com a distância entre pontos definida pela distância euclideana, então $\beta = m$. Se o problema está definido no caso métrico, escrevemos $\beta = t$. Se não há nenhuma suposição sobre o grafo em questão, ou seja, no caso mais geral possível, escrevemos $\beta = g$. O "MM", finalmente, refere-se à nossa função objetivo, conforme já mencionamos.

4.1.1 Um algoritmo de aproximação para k-tMM

Seja S nosso conjunto de pontos e $d(i,j)$ a distância entre os pontos $i, j \in S$. Suponha que $|S| > k$, caso contrário o problema torna-se trivial — basta tomar $\{i\}$ como conjunto da partição para cada $i \in S$. Seja $\mathrm{OPT}(S,k)$ o valor de um *k-split* ótimo para S.

Definição 4.1. *Um conjunto $T \subseteq S$ é uma* clique *se para todo par (i,j) de pontos distintos de T, existe uma aresta com extremos i e j.*

Note que, no caso métrico, todo subconjunto de S é uma clique.

Definição 4.2. *Uma clique $T \subseteq S$ é uma* a-clique de peso h *se $|T| = a$ e $\min_{i \neq j \in T} d(i,j) = h$.*

Lema 4.3. *Se S tem uma $(k+1)$-clique de peso h, então $\mathrm{OPT}(S,k) \geq h$.*

Demonstração. Tome um *k-split* qualquer, digamos $(B_1, \ldots, B_k)$.

Seja i, $1 \leq i \leq k$, tal que $|B_i \cap T| \geq 2$. Tal i existe pois $|T| = k+1 > k$.

Sejam p, q elementos distintos em $B_i \cap T$. Segue da nossa hipótese que $d(i,j) \geq h$. Logo,

$$\mathrm{val}(B_1, \ldots, B_k) \geq h,$$

isto é, qualquer *k-split* tem valor no mínimo h. □

Considere o problema k-tMM. Isto é, o problema de particionar pontos em k conjuntos (onde k é um parâmetro do problema) de forma a minimizar o valor da partição conforme a função objetivo MM, no caso métrico. Propomos o seguinte algoritmo, bastante simples, para resolvê-lo.

Observe que o algoritmo começa colocando todos os pontos em B_1 e selecionando um elemento arbitrário como o centro c_1 desse conjunto. A cada passo ℓ, o algoritmo já tem os *clusters* $B_1, \ldots, B_\ell$, e cria um novo *cluster* com o seguinte critério: tome um ponto, digamos v, que esteja a distância máxima do centro c_j de seu conjunto B_j atual. Mova v para $B_{\ell+1}$, e tome-o como o centro $c_{\ell+1}$ desse *cluster*. Então, mova para $B_{\ell+1}$ todos os pontos que estão mais próximos de $v = c_{\ell+1}$ do que de seus respectivos centros atuais.

Teorema 4.4. *O algoritmo 4.1 é uma 2-aproximação para o k-tMM.*

Demonstração. Em primeiro lugar, o algoritmo é polinomial. A cada *cluster* criado, a lista dos pontos é percorrida para obter aquele que está a máxima distância de seu centro atual. Esse passo pode ser feito

Algoritmo 4.1: Algoritmo para o k-tMM.

Entrada: Conjunto S, distâncias d sobre S, natural k
Saída: k-*split* $(B_1, \ldots, B_k)$ de S

```
1  B_1 ← S;
2  c_1 ← elemento arbitrário em B_1;
3  para ℓ ← 1 até k − 1 faça
4  |  h ← max{d(c_j, v_i) : v_i ∈ B_j, 1 ≤ j ≤ ℓ};
5  |  Seja v_i tal que v_i ∈ B_j e d(d_i, c_j) = h;
6  |  Mova v_i para B_{ℓ+1};
7  |  c_{ℓ+1} ← v_i;
8  |  para cada v ∈ B_1 ∪ ··· ∪ B_ℓ faça
9  |  |  Seja j tal que v ∈ B_j;
10 |  |  se d(v, c_j) ≥ d(v, c_{ℓ+1}) então
11 |  |  |  Mova v de B_j para B_{ℓ+1};
12 |  |  fim
13 |  fim
14 fim
```

em tempo $O(n)$, onde $n = |S|$. Após escolhido o novo centro, é necessário novamente percorrer os pontos para verificar quais deverão ser movidos para o novo conjunto. Esse passo também é $O(n)$. Portanto, a criação de um novo *cluster* custa tempo $O(n)$. O consumo de tempo do algoritmo, portanto, é $O(kn)$, que é $O(n^2)$, visto que $k \leq n$.

Além disso, é claro que o algoritmo gera um k-*split* . Resta verificar que o valor da solução gerada não é superior a $2 \cdot \mathrm{OPT}(S)$.

Seja $(B_1, \ldots, B_k)$ a solução gerada pelo algoritmo. Seja $h = \max d(v, c_j) \colon v \in B_j$.

Se p e q são vértices no mesmo *cluster*, digamos B_j, e são tais que $\mathrm{val}\,(B_1, \ldots, B_k) = d(p, q)$, note que, como estamos no caso métrico, vale que

$$\mathrm{val}\,(B_1, \ldots, B_k) = d(p, q) \leq d(p, c_j) + d(c_j, q) \leq h + h = 2 \cdot h \tag{4.1}$$

Seja, agora v^* um vértice tal que $d(v^*, c_{j^*}) = h$, onde j^* é tal que $v^* \in B_{j^*}$, isto é, v^* está a distância máxima de seu centro c_{j^*}. Observe que, como v^* não é o centro de seu conjunto, nenhum passo do algoritmo escolheu v^* como novo centro. Ou seja, todo centro, ao ser escolhido, estava a distância *no mínimo* h de seu centro. Além disso, v^* está a distância pelo menos h de cada centro, visto que está a essa distância do centro mais próximo.

Portanto, $\{v^*, c_1, \ldots, c_k\}$ é uma $(k+1)$-clique de peso h. Segue de 4.3, portanto, que $\mathrm{OPT}(S, k) \geq h$. Disso e 4.1, temos que

$$\mathrm{val}\,(B_1, \ldots, B_k) \leq 2 \cdot h \leq 2 \cdot \mathrm{OPT}(S, k).$$

□

Antes de prosseguir, vejamos que a análise é justa. Mostraremos que é possível gerar instâncias para as quais a razão entre o valor ótimo e o valor da solução dada por 4.1 seja tão próxima de 2 quanto se queira.

Considere a seguinte classe de instâncias do k-1MM (*clustering* de pontos na reta real), que é um caso particular do k-tMM:

- $S = \{0, 1, 2 - \varepsilon, 3\}$, onde $0 < \varepsilon < 1$,
- $d(i, j)$ é a distância euclideana entre i e j (no caso 1-dimensional, $d(i, j) = |i - j|$), e
- $k = 2$

Suponha que no primeiro passo o algoritmo tome o ponto 1 como c_1. No passo seguinte, o ponto mais distante de c_1 é o ponto 3, que é então tomado como o centro c_2 do conjunto B_2. O algoritmo termina com:

- $B_1 = 0, 1, 2 - \varepsilon, B_2 = 3$,
- $\text{val}\,(B_1, B_2) = d(2 - \varepsilon, 0) = 2 - \varepsilon$.

Note que o valor ótimo para essa instância é $\text{val}\,(\{0, 1\}, \{2 - \varepsilon, 3\}) = 1 + \varepsilon$. Portanto, a razão entre o valor da solução do algoritmo e o valor ótimo é $\frac{2-\varepsilon}{1+\varepsilon}$.

4.1.2 Complexidade dos problemas de *clustering*

Demos um algoritmo de aproximação para o k-tMM, mas até agora não nos perguntamos se esse problema, ou pelo menos o k-gMM, que é mais geral, são NP-difíceis. Pelo que discutimos até agora, poderia ser que uma solução exata para o problema pudesse ser atingível em tempo polinomial mesmo que P $\neq$ NP, e nesse caso nossa aproximação não teria muito valor.

Mas, como veremos agora, é verdade que esses problemas são NP-difíceis. Daremos uma prova de que a versão de decisão do k-tMM é um problema NP-completo, conforme a que pode ser vista em [5]. Disso segue, como veremos, que a versão de otimização de k-tMM é NP-difícil. O mesmo vale para k-gMM, claro, por ser uma generalização daquele.

Mais que isso, veremos, ainda, que o problema da $(2-\varepsilon)$-aproximação para o k-tMM é NP-difícil para qualquer $\varepsilon > 0$. Ou seja, o algoritmo 4.1 é o melhor possível para esse problema, a menos que P = NP.

Considere o problema da cobertura exata por conjuntos de tamanho 3, que chamaremos de XC3:

Dados um conjunto finito de elementos $X = \{x_1, x_2, \ldots, x_{3q}\}$ e uma coleção de subconjuntos de X de tamanho 3, $C = Y \subseteq X \colon |Y| = 3$, com $|C| = m$, e nenhum elemento de X aparece em mais do que 3 elementos de C. O problema XC3 é decidir se C contém uma cobertura exata de X, isto é, um $C' \subseteq C$ tal que cada elemento de X ocorre em exatamente um elemento deC'.

Sabe-se que esse problema é NP-completo [3]. Considere agora a seguinte variação de XC3. Chamaremos esse novo problema de RXC3: a versão *restrita* de XC3.

No problema RXC3, é dada uma instância (X, C) de XC3 tal que cada elemento de X ocorre em *exatamente* 3 dos conjuntos na coleção C. O problema é, como em XC3, decidir se C contém uma cobertura exata de X.

O problema RXC3 também é NP-completo: em [5] é mostrada uma redução polinomial de XC3 a RXC3. Mostraremos que RXC3 tem uma redução polinomial à versão de decisão de k-tMM.

O problema de decisão k-tMM consiste em, dados um conjunto de pontos (vértices) S com distância $d(p, q)$ definida para cada par de vértices e obedecendo a desigualdade triangular, um inteiro positivo k e um real não negativo w, decidir se S admite um k-*split* de peso não superior a w.

Teorema 4.5. *O problema de decisão k-tMM é NP-completo.*

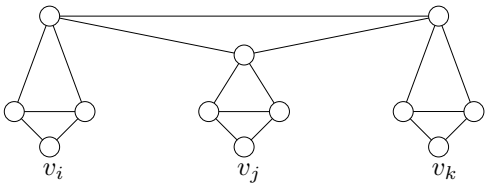
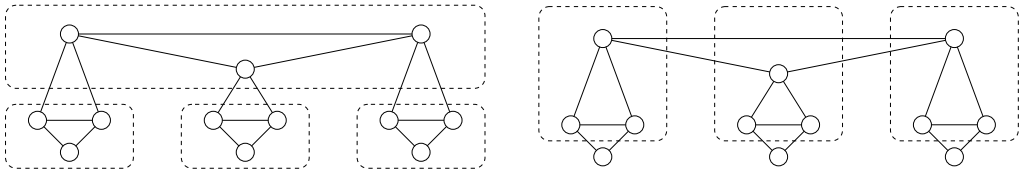


Figura 4.1: Componente representando um $Y \in C$.



(a) Agrupamento dos $Y \in C'$ (b) Agrupamento dos $Y \in C \setminus C'$

Figura 4.2: As duas possibilidades de divisão em *clusters* de um conjunto de vértices representando um $Y \in C$.

Demonstração. É fácil ver que o problema de decisão k-tMM está em NP. Dada uma instância (S, d, k, w) para o problema, um certificado para "sim" é um k-*split* $(B_1, \ldots, B_k)$ de valor $\text{val}(B_1, \ldots, B_k) \leq w$. Tal certificado tem tamanho polinomial no tamanho da entrada, e claramente pode ser verificado em tempo polinomial.

Veremos agora que RXC3 tem uma redução polinomial ao k-tMM.

Seja (X, C) uma instância do RXC3, com $|X| = 3q$, com q inteiro, e $|C| = m$. Considere a seguinte instância do k-tMM:

- O conjunto de vértices, S, contém um vértice v_i para cada $x_i \in X$, e cada $Y \in C$ introduz 9 novos vértices. Se $Y = \{v_i, v_j, v_k\}$, a estrutura mostrada na figura 4.1 representa Y.
- As distâncias d são dadas da seguinte maneira: as arestas da representação de um $Y \in C$, conforme acima, têm peso 1. As demais arestas têm peso 2;
- $k = 3m + q$; e
- $w = 1$.

Tal construção pode ser feita em tempo polinomial no tamanho da instância (X, C).

Para completar nossa prova, basta mostrar que S admite um k-*split* de valor ≤ 1 se, e somente se, C contém uma cobertura exata para X.

($\Leftarrow$) Seja $C' \subseteq C$ cobertura exata de X. Para cada $Y \in C'$, o subgrafo que representa Y é agrupado em 4 *clusters*, conforme mostrado à esquerda na figura 4.2.

Para cada $Y \in C \setminus C'$, o subgrafo correspondente é agrupado em 3 *clusters*, conforme mostrado à direita na figura 4.2.

Note que esses *clusters* dão uma partição de S, isto é, cada vértice é incluído em exatamente um *cluster* por essa regra.

Cada elemento em C contribui com 3 *clusters*, exceto aqueles em C', que adicionam mais 1 *cluster* cada.

Como C' é uma cobertura exata de X e todos os seus elementos são 3-conjuntos (conjuntos de tamanho 3), vale que $C' = |X|/3$.

Então, o número de *clusters* criados é

$$3 \cdot |C| + 1 \cdot |C'| = 3m + \frac{|X|}{3} = 3m + q = k.$$

Finalmente, em todo *cluster* vale que os vértices estão conectados apenas por arestas de peso 1. Portanto, obtivemos um *k-split* de S com peso $1 \leq w$.

($\Rightarrow$) Seja $(B_1, \ldots, B_k)$ um *k-split* de S com valor $\text{val}(B_1, \ldots, B_k) \leq w = 1$.

Note que, na nossa instância do k-tMM, não há nenhum conjunto de 4 vértices totalmente conectados por arestas de peso 1. Além disso, $|V| = 3q + 9m = 3(q + 3m) = 3k$.

Então, cada B_i tem cardinalidade $|B_i| = 3$. Pela regra de construção dessa instância, todo subgrafo que representa um $Y \in C$ está agrupado, pelo nosso *k-split* , como uma das opções da figura 4.2.

Como todo vértice está em exatamente um *cluster*, para que tenhamos k *clusters*, certamente há exatos q subgrafos representando elementos em C que estão agrupados como à esquerda na figura 4.2, e cada um deses deve incluir v_ℓ (representantes dos x_ℓ) todos diferentes, uma vez que cada vértice desses só está em *clusters* dos subgrafos agrupados da maneira (a).

Dessas observações, segue que, se tomarmos os $Y \in C$ representados por subgrafos que estão agrupados como em teremos uma cobertura exata de X. □

Esse resultado tem duas consequências imediatas.

Corolário 4.6. *O problema de otimização k-tMM é NP-difícil.*

Corolário 4.7. *O problema de otimização k-gMM é NP-difícil.*

Outro resultado segue não do Teorema 4.5, mas da redução usada em sua prova.

Teorema 4.8. *O problema da $(2-\varepsilon)$-aproximação para o k-tMM é NP-difícil para todo $\varepsilon > 0$.*

Demonstração. Seja (X, C) uma instância do RXC3. Seja (G, d, k) a instância do k-tMM obtida pela mesma redução usada na prova do Teorema 4.5.

O valor ótimo para (G, d, k) é 1, se a resposta do RXC3 para (X, C) é "sim", e 2, caso contrário.

Fixe um $\varepsilon > 0$. Uma $(2-\varepsilon)$-aproximação daria, para a instância (G, d, k), portanto, uma resposta exata: se o valor ótimo da instância é 2, então o algoritmo daria uma resposta de valor ≥ 2, que só pode ser, de fato, 2, uma vez que não há aresta com custo maior que 2; e se o valor ótimo da instância é 1, o algoritmo obrigatoriamente forneceria uma solução de valor $\leq (2-\varepsilon) \cdot 1 < 2$, que no caso só pode ser, de fato, 1 (pois toda aresta tem valor 1 ou 2).

Dessa maneira, nossa $(2-\varepsilon)$-aproximação resolveria, em tempo polinomial, o problema RXC3, que é NP-completo: dada a resposta do algoritmo para a instância (G, d, k), conseguimos determinar a resposta do problema RXC3 para a instância (X, C).

Ou seja, o problema da $(2-\varepsilon)$-aproximação para o k-tMM é NP-difícil. □

Note que os resultados sobre a complexidade e inaproximabilidade do k-tMM são especialmente importantes por tratar-se do tipo mais básico de problema de *clustering*. Muitas variações do *clustering* têm o k-tMM como um caso particular, de modo que os resultados vistos para este valem também para as variantes. Isto é, se um problema tem k-tMM como caso particular, sabemos de imediato que uma 2-aproximação para ele é o melhor que podemos esperar obter, a menos que P = NP.

4.2 Aproximando problemas de gargalo

Note que, devido à forma de sua função objetivo, o problema k-tMM tem como valores viáveis os custos das arestas do grafo. Em particular, o valor ótimo é necessariamente o custo de alguma aresta do grafo.

Hochbaum e Shmoys se referem a problemas com essa propriedade de *bottleneck problems*, ou "problemas de gargalo", e propõem um método geral para resolvê-los [6]. Um tal problema admite, para uma instância que diz respeito a um grafo G, um conjunto $\mathcal{G}$ de *subgrafos viáveis* de G, que definimos da seguinte maneira. Um subgrafo H de G está em $\mathcal{G}$ se, e somente se, existe em H uma solução viável para o problema. Portanto, basta encontrarmos $G^* \in \mathcal{G}$ cujo custo da aresta mais cara seja mínimo. Tais ideias ficarão mais claras ao estudarmos problemas concretos mais adiante.

Todos os problemas que vamos considerar daqui em diante são *métricos*, isto é, são definidos sobre grafos completos com custos respeitando a desigualdade triangular. Começaremos dando algumas definições e resultados que nos ajudarão a descrever a técnica.

Definição 4.9. *Um* algoritmo de decisão ρ-relaxado *para um problema de minimização* Π *é um algoritmo de tempo polinomial que, dada uma instância* I *de* Π *e um valor* x*, devolve* ou *uma solução viável* $S \in \mathrm{Sol}(I)$ *de valor* $\mathrm{val}(S) \leq \rho \cdot x$*,* ou *um certificado eficiente de que* $\mathrm{OPT}(I) > x$*.*

Definição 4.10. *Dado um grafo* $G = (V, E)$ *e um inteiro positivo* t*, a* t*-ésima potência de* G *é o grafo* $G^t = (V, E^t)$*, com*

$$E^t = \{\{u, v\} \colon \text{ existe em } G \text{ caminho de } u \text{ a } v \text{ de comprimento } \leq t\}.$$

Dado um grafo $G = (V, E)$ com custo c_e para cada aresta $e \in E$, defina

$$\max(G) = \max_{e \in E} c_e.$$

Lema 4.11. *Sejam* G *um grafo completo com custos* c_e *nas arestas respeitando a desigualdade triangular;* u, v *vértices distintos. Se* $P \subseteq G$ *é um caminho de* u *a* v *de comprimento* ℓ*, então*

$$c_{uv} \leq \ell \cdot \max(P).$$

Demonstração. Se $\ell = 2$, a propriedade segue diretamente da desigualdade triangular.

Seja $\ell > 2$ e suponha que a propriedade valha para caminhos de comprimento $\ell - 1$.

Seja P um caminho $i_1 i_2 \cdots i_\ell$, de comprimento ℓ, com $u = i_1$, $v = i_\ell$. Seja $P' \subseteq P$ o caminho $i_1 \cdots i_{\ell-1}$. Por hipótese de indução, vale que

$$c_{i_1 i_{\ell-1}} \leq (\ell - 1) \max(P') \leq (\ell - 1) \max(P), \tag{4.2}$$

e, pela desigualdade triangular,

$$c_{i_1 i_\ell} \leq c_{i_1 i_{\ell-1}} + c_{i_{\ell-1} i_\ell}. \tag{4.3}$$

De 4.2 e 4.3, segue que

$$c_{i_1 i_\ell} \leq (\ell - 1) \max(P) + c_{i_{\ell-1} i_\ell} \leq (\ell - 1) \max(P) + \max(P) = \ell \cdot \max(P),$$

e a prova do lema segue por indução. □

Lema 4.12. *Seja G um grafo completo com custos c_e nas arestas respeitando a desigualdade triangular. Sejam H um subgrafo de G, t um inteiro positivo. Se $\{i, j\}$ é uma aresta em H^t, então*

$$c_{ij} \leq t \cdot \max(H)$$

Demonstração. Se $\{i, j\}$ é uma aresta em H^t, então existe em H um caminho de comprimento t de i a j.

Pelo lema 4.11, segue que $c_{ij} \leq t \cdot \max(H)$. □

Lema 4.13. *Seja G um grafo completo com custos c_e nas arestas respeitando a desigualdade triangular. Sejam H um subgrafo de G, t um inteiro positivo. Se H^t contém uma solução viável S para um problema de gargalo sobre G, então S é viável em G e*

$$val\,(S) \leq t \cdot \max(H).$$

Demonstração. Note que H^t é subgrafo de G. Isso vem dos fatos de que o conjunto $V(H^t)$ de vértices de H^t é um subconjunto dos vértices de G, e de que G é grafo completo. Portanto toda solução em H^t está, de fato, em G.

Além disso, como se trata de um problema de gargalo, o valor de uma solução S em H^t é $\text{val}\,(S) = c_e$ para alguma aresta e de H^t. Assim, do lema 4.12, segue que $\text{val}\,(S) \leq t \cdot \max(H)$. □

Considere as seguintes rotinas. Dado um grafo $G = (V, E)$ com custo c_e para cada $e \in E$ e um valor x, GARGALO (G, x) devolve o grafo $G' = (V, E')$, onde $E' = \{e \in E \colon c_e \leq x\}$. Isto é, o grafo obtido removendo-se de G toda aresta de custo superior a x. A rotina TESTE recebe um subgrafo $H \subseteq G$, o conjunto $\mathcal{G}$, e um inteiro positivo t, e devolve *ou* uma solução contida em H^t, *ou* um certificado de que não existe solução em H.

Propõe-se o algoritmo 4.2 como método geral para aproximar problemas de gargalo. Note que $\mathcal{G}$ não é dado explicitamente; trata-se de uma maneira de representar genericamente as restrições do problema em questão.

Algoritmo 4.2: Método geral para aproximar problemas de gargalo.

Entrada: Grafo $G = (V, E)$, custos c nas arestas, conjunto $\mathcal{G}$
Saída: Solução S

```
1 Ordene as arestas de modo que c_1 ≤ c_2 ≤ ··· ≤ c_m;
2 i ← 1;
3 repita
4 |   G_i ← GARGALO (G, c_i) ;
5 |   S ← TESTE (G_i, G, t) ;
6 |   i ← i + 1;
7 até S ser uma solução viável ;
```

Lema 4.14. *Na iteração i do algoritmo 4.2, as linhas 4 e 5 formam um algoritmo de decisão t-relaxado para o seguinte problema: "existe em G uma solução de custo no máximo c_i?"*

Demonstração. Pela definição do grafo G_i produzido pela rotina GARGALO, vale que $\max(G_i) = c_i$.

Se o resultado S devolvido pela rotina TESTE for uma solução em G_i^t, então, pelo lema 4.13, S é uma solução viável com valor $\text{val}\,(S) \leq t \cdot \max(G_i) = t \cdot c_i$.

Se S é um certificado de que não existe solução em G_i, então não existe em G solução que use apenas arestas de custo no máximo c_i, e então S é, de fato, um certificado de que $\text{OPT} > c_i$. □

Teorema 4.15. *O algoritmo 4.2 é uma t-aproximação para o problema de gargalo descrito por $\mathcal{G}$.*

Demonstração. Seja i^* a iteração em que S recebe uma solução. Pelo lema 4.14, vale que $\text{val}(S) \leq t \cdot c_{i^*}$.

Note que $\text{OPT} \geq c_{i^*}$:

- Se $i^* = 1$, então, $c_{i^*} = c_1 \leq \text{OPT}$, uma vez que o valor ótimo é no mínimo o valor da aresta mais barata.
- Se $i^* > 1$, pelo lema 4.14, vale que $\text{OPT} > c_{i^*-1}$. Isso porque, na iteração $i^* - 1$, a rotina TESTE devolveu um certificado de falha. Como o valor ótimo é o valor de alguma aresta, $\text{OPT} > c_{i^*-1}$ implica que $\text{OPT} \geq c_{i^*}$.

Assim, $\text{val}(S) \leq t \cdot c_{i^*} \leq t \cdot OPT$. □

Assim, para usar o algoritmo 4.2 para um problema de gargalo, basta encontrar um t e uma rotina TESTE adequados. Mostraremos, a seguir dois exemplos básicos de aplicação dessa técnica.

4.2.1 Aproximando o problema dos k-centros

Já definimos, no início desta seção, o problema dos k-centros. O algoritmo 4.1 é uma 2-aproximação também para esse problema [4]. No entanto, daremos, agora, uma (outra) 2-aproximação, usando o método geral que acabamos de descrever.

Se nossa instância é um grafo completo $G = (V, E)$ com custos c_e nas arestas respeitando a desigualdade triangular e um inteiro positivo k, uma solução é um conjunto $S \subseteq V$ com $|S| = k$. Se $H \subseteq G$, dizemos que S é viável em H se todo vértice é vizinho de pelo menos um centro em H.

Como queremos encontrar uma 2-aproximação, escolhemos $t = 2$. A nossa rotina de teste é bastante simples: dado o grafo G_i, a rotina constrói o grafo G_i^2. Obtém-se, então, um conjunto independente maximal I em G_i^2. Por fim, a rotina devolve I.

Se $|I| \leq k$, então I é uma solução viável, pois em G_i^2 todo vértice é vizinho de um centro, uma vez que I é um conjunto independente maximal. Do lema 4.13, segue que

$$\text{val}(I) \leq t \cdot \max(G_i) = 2 \cdot c_i.$$

Se $|I| > k$, então I é um certificado de que $\text{OPT} > c_i$: seja $S \subseteq V$ com $|S|$, e suponha que todo vértice em G_i é vizinho de algum dos centros de S. Como $|I| > |S|$, existe um vértice $v \in S$ que é vizinho de pelo menos dois vértices, digamos u e w, de I. Mas isso é absurdo, pois se $u, w \in I$, então u e w não têm vizinhos em comum em G_i.

Assim, o algoritmo de teste tem a propriedade que desejamos: devolve ou um certificado de falha, ou uma solução de valor no máximo $t \cdot \max(G_i)$. Segue do teorema 4.15 que o algoritmo 4.2, com $t = 2$ e o algoritmo de teste que descrevemos é uma 2-aproximação para o *k-center*.

Podemos descrever o algoritmo de modo mais explícito, a fim de ilustrar melhor a própria técnica geral.

Teorema 4.16. *O algoritmo 4.3 é uma 2-aproximação para o* k-center.

Demonstração. Segue da discussão anterior. □

Algoritmo 4.3: Modificação do algoritmo 4.2 para resolver o *k-center*.

Entrada: Grafo $G = (V, E)$, custos c nas arestas, inteiro $k > 0$
Saída: Solução S

```
1 Ordene as arestas de modo que c_1 ≤ c_2 ≤ ··· ≤ c_m;
2 i ← 1;
3 repita
4 |   G_i ← GARGALO(G, c_i);
5 |   S ← conjunto maximal em G_i^2;
6 |   i ← i + 1;
7 até |S| ≤ k ;
```

4.2.2 Aproximando o *k-supplier*

O *k-supplier* é um problema bastante similar ao *k-center*. Ainda temos um grafo $G = (V, E)$ completo com custos c_e nas arestas respeitando a desigualdade triangular e um inteiro $k > 0$, e desejamos abrir k centros. A diferença é que temos também um conjunto $U \subseteq V$, e podemos escolher como centros apenas vértices em U, e, além disso, não exigimos que os vértices em U estejam próximos de algum centro.

Isto é, no *k-supplier*, queremos encontrar $S \subseteq U$ com $|S| = k$ que minimize a maior distância de um vértice $v \in W = V \setminus U$ a um centro $s \in S \subseteq U$.

Daremos uma 3-aproximação para esse problema, que é o melhor resultado a menos que P = NP [6]. Para isso, usaremos o método geral escolhendo $t = 2$ e o algoritmo de teste descrito pelo algoritmo 4.4.

Algoritmo 4.4: Rotina TESTE para o *k-supplier*

Entrada: Grafo G_i, inteiro $k > 0$, conjunto $U \subseteq V$.
Saída: Conjunto S, que é solução ou certificado de falha.

```
1  Construa o grafo G_i^2;
2  I ← conjunto independente maximal em G_i^2;
3  I' ← I ∩ W;
4  se |I'| > k então
5  |   S ← I';
6  fim
7  senão
8  |   S ← ∅;
9  |   para cada v ∈ I' faça
10 |   |   se v tem vizinhos em U no grafo G_i então
11 |   |   |   Seja w ∈ U um vizinho de v em G_i;
12 |   |   |   S ← S ∪ {w};
13 |   |   fim
14 |   |   senão
15 |   |   |   retorna I';
16 |   |   fim
17 |   fim
18 fim
```

Lema 4.17. *O algoritmo 4.4 devolve ou uma solução viável em G_i^3 ou um certificado de que* $\mathrm{OPT} > c_i$.

Demonstração. Primeiro, considere o caso em que $|I'| > k$. Então o algoritmo devolve I', que é um certificado de falha: nenhum conjunto $S \subseteq U$ com $|S| = k$ é viável em G_i. Para verificar a afirmação, basta supor que S é uma tal solução viável. Como $|I'| > |S|$, existem vértices $u, v \in I'$ que são vizinhos

do mesmo centro. Isto é uma contradição: u e v não têm vizinhos em comum em G_i, pois não são vizinhos em G_i^2. Assim, não existe solução viável em G_i, e portanto I' é um certificado de que OPT $> c_i$.

Caso $|I'| \leq k$, o algoritmo seleciona, para cada $v \in I'$, um vizinho $w \in U$ e o escolhe como centro. Se algum $v \in I'$ não é vizinho, em G_i, de nenhum vértice em U, então claro que não existe solução viável em G_i. Assim, a rotina devolve I', que também nesse caso é um certificado de que OPT $> c_i$.

Se, no entanto, o algoritmo consegue abrir um centro vizinho a cada $v \in I'$, vale que todo vértice em W é vizinho, em G_i^3, de algum centro. Se $v \in I'$, é claro que isso vale. Seja $v_1 \in U \setminus I'$. Então v tem vizinho, digamos v_2, em comum com algum vértice em I', digamos v_3 (se não, v_1 estaria em I'). Seja $s \in S$ adjacente a v_3. Então, existe um caminho de comprimento 3, em G_i, de v_1 até s: o caminho $v_1 v_2 v_3 s$. □

Desse lema e do teorema 4.15, segue que o algoritmo para problemas de gargalo, com $t = 3$ e o algoritmo de teste acima, é uma 3-aproximação.

Tal resultado é o melhor possível a menos que P = NP [6].

4.3 Problemas de k-centros com tolerância a falhas

Mostraremos, agora, duas variantes do *k-center* e uma do *k-supplier*, apresentadas como versões "tolerantes a falhas" [7]. Nos problemas de k-centros com tolerância a falhas, estamos interpretando o problema de *clustering* como um problema de localização de instalações: vemos os centros como "fornecedores" e os vértices como "clientes". Desejamos, nas variantes a seguir, garantir que, caso algum centro falhe (isto é, deixe de existir para o grafo), os vértices que dele dependem não fiquem muito distantes de seus novos centros mais próximos.

Assim, nos problemas descritos a seguir, queremos abrir k centros de modo a minimizar a maior distância de um vértice aos seus α centros mais próximos. Esse α faz parte da instância. Os problemas e algoritmos mostrados são descritos por Khuller, Pless e Sussman [7].

4.3.1 Os k-centros com α-vizinhos

O *α-neighbor k-center*, ou problema dos k-centros com α-vizinhos, define-se da seguinte maneira: dado um grafo completo G com custos c_e nas arestas respeitando a desigualdade triangular, e inteiros positivos k e α, desejamos escolher um *conjunto de centros* $S \subseteq V$ com $|S| \leq k$ que minimize a maior distância de um vértice *que não é centro* ao mais distante dentre seus α centros mais próximos.

Mais formalmente, se $S \subseteq V$ e $u \in V$, definimos $\delta^{(\alpha)}(u, S)$ como o custo da aresta que conecta u ao α-ésimo centro mais próximo de u. O problema dos k-centros com α-vizinhos é

$$\min_{S \subseteq V \ |S| \leq k} \max_{u \in V \setminus S} \delta^{(\alpha)}(u, S).$$

Note que, portanto, se um vértice é centro, não exigimos que ele esteja "perto" de $\alpha - 1$ outros centros. Essa é a diferença entre esse problema e o próximo que estudaremos.

Observe também que o *k-center* é um caso particular desse problema, para $\alpha = 1$. Portanto, a melhor razão de aproximação que podemos esperar conseguir encontrar para esse problema, a menos que P = NP, é 2. E, de fato, mostraremos um algoritmo que é uma 2-aproximação.

Esse algoritmo também usa o método geral dado pelo algoritmo 4.2. Usamos $t = 2$ e a rotina de teste descrita pelo algoritmo 4.5. A ideia do algoritmo é atribuir a cada vértice v um número de "cobertura", $C(v)$, que começa valendo zero. O número de cobertura corresponde ao número de centros já escolhidos dos quais o vértice é vizinho (se v é centro, define-se $C(v) = \alpha$ pois v já está "satisfeito"). O algoritmo executa α iterações, e, ao fim de cada iteração j, garante que cada vértice está coberto pelo menos j vezes (ou por ser um centro, ou por ser, de fato, vizinho de j dos centros já escolhidos).

Algoritmo 4.5: Rotina de teste para o α-*all-neighbor k-center*

Entrada: Grafo G_i^2, inteiros $k > 0$, $\alpha > 0$
Saída: S (solução ou certificado de falha)

```
1  S ← ∅;
2  para cada v ∈ V faça
3  |  C(v) ← 0;
4  fim
5  para j ← 1 até α faça
6  |  enquanto ∃v com C(v) < j faça
7  |  |  S ← S ∪ {v};
8  |  |  C(v) ← α;
9  |  |  para cada vizinho u de v em G_i^2 faça
10 |  |  |  C(u) ← C(u) + 1;
11 |  |  fim
12 |  fim
13 fim
```

Note que o que o algoritmo faz, implicitamente, é encontrar, a cada iteração j, um conjunto independente em G_i^2. Isso porque, se dois vértices distintos u e v são escolhidos na mesma iteração para se tornarem centros, é porque o primeiro deles a ser escolhido como centro não aumentou o número de cobertura do outro, o que só acontece se eles não são vizinhos em G_i^2.

Lema 4.18. *O algoritmo 4.5 devolve ou uma solução viável em G_i^2, ou um certificado de que* $\mathrm{OPT} > c_i$.

Teorema 4.19. *O método geral para problemas de gargalo (algoritmo 4.2), com $t = 2$ e a rotina de teste dada pelo algoritmo 4.5, é uma 2-aproximação para o problema do α-neighbor k-*center.

Demonstração. Segue do lema 4.18 e do teorema 4.15. □

4.3.2 Os k-centros com α-todos-vizinhos

Estudaremos, nesta seção, o problema chamado de α-*all-neighbor k-center*, ou problema dos k-centros com α-todos-vizinhos. Trata-se do problema sobre o qual já comentamos na seção 4.3.1; a única diferença em relação ao problema anterior é o fato de que, aqui, iremos exigir que *todos* os vértices sejam cobertos por α "próximos". Ou, mais formalmente, nosso problema é

$$\min_{S \subseteq V \ |S| \leq k} \max_{u \in V} \delta^{(\alpha)}(u, S).$$

Para esse problema, daremos uma 3-aproximação. Como esse problema é, também, uma variação do k-*center*, espera-se conseguir encontrar uma 2-aproximação. Khuller, Pless e Sussman [7] dão uma 2-aproximação para os casos especiais $\alpha = 2$ e $\alpha = 3$, mas ainda não se sabe se a a melhor razão possível (a menos que P = NP) para esse problema é menor do que 3 no caso geral [8].

O algoritmo de teste para esse problema toma a seguinte forma: dado G_i, obtém-se um conjunto independente maximal I em G_i^2. Se $\alpha \cdot |I| > k$, então não existe solução viável em G_i, pois cada elemento de I exige a criação de α centros diferentes. Nesse caso, I é certificado de que OPT $> c_i$.

Além disso, se algum vértice tem menos de $\alpha - 1$ vizinhos em G_i, certamente G_i também não é viável, pois não há como todos os vértices estarem conectados a α centros. Um tal vértice é certificado dessa afirmação.

Caso contrário, isto é, se $\alpha \cdot |I| \leq k$ e todo vértice tem pelo menos $\alpha - 1$ vizinhos em G_i, então damos uma solução viável em G_i^3 da seguinte maneira: para cada $v \in I$, escolha como centros o vértice v e $\alpha - 1$ de seus vizinhos. Como I é independente em G_i^2, todas essas vizinhanças são disjuntas, e portanto não corremos o risco de tentar repetir um vértice nesse processo.

Tome $v \in V$. Se $v \in I$, então v está a distância no máximo 1, em G_i, dos α centros mais próximos. Se $v \in V \setminus I$, então $\exists w \in I$ tal que v e w têm vizinho u em comum (pela maximalidade de I). Portanto, todos os vizinhos de w, em G_i, são vizinhos de v em G_i^3. Assim, nossa solução é viável em G_i^3, e esse algoritmo de teste nos dá uma 3-aproximação para o problema dos k-centros com α-todos-vizinhos.

4.3.3 O *k-supplier* com α-vizinhos

Esse problema é similar ao *k-supplier*, com a diferença de que todo vértice que não é candidato a centro precisa ser suprido por α centros próximos. Ou, formalmente, dado um grafo G métrico, um conjunto $U \subseteq V$ e inteiros positivos k, α, desejamos encontrar

$$\min_{S \subseteq U \ |S| \leq k} \max_{u \in V \setminus U} \delta^{(\alpha)}(u, S).$$

O *k-supplier* que já vimos é um caso particular deste, para $\alpha = 1$. Daremos, a seguir, uma 3-aproximação para o problema, que, portanto, é o melhor possível a menos que P = NP [6] [7].

A rotina de teste que daremos para esse problema é a seguinte. Dado G_i, construímos G_i^2, e H_i o subgrafo de G_i^2 induzido por $W = V \setminus U$. Obtenha um conjunto independente maximal I em H_i.

Se algum $v \in W$ não tem α vizinhos em U no grafo G_i, então OPT $> c_i$, pois em G_i não há conjunto de centros que satisfaça a todos os elementos de W.

E, se $\alpha \cdot |I| > k$, também temos um certificado de falha, uma vez que, como não têm vizinhos em comum em G_i, cada elemento de I requer a criação de α centros diferentes.

Caso contrário, podemos definir o conjunto de centros $S \subseteq U$ escolhendo α vizinhos de cada $v \in I$ (em G_i) pertencentes a U. Tal conjunto é viável em G_i^3 pois qualquer $v \in W \setminus I$ tem pelo menos um vizinho em comum com um elemento de I, de modo que está a distância no máximo 3 de toda a vizinhança (em G_i) de tal elemento em I — em particular, a distância no máximo 3 (em G_i) de α centros.

Com $t = 3$ e a rotina de teste acima, o método geral para problemas de gargalo dá uma 3-aproximação para o nosso problema.

5 Conclusões

Neste trabalho, conseguimos estudar variadas técnicas de aproximação para alguns problemas NP-difíceis. Ao estudar as técnicas de aproximação para o problema da cobertura por conjuntos, pudemos observar o quanto tais técnicas podem ser diferentes e usar ao mesmo tempo vários conceitos de computação e matemática — programação linear, grafos, probabilidade, etc.

Após nos depararmos com uma amostra da diversidade de técnicas para algoritmos de aproximação existentes, pudemos nos concentrar numa família mais específica de problemas: os problemas de *clustering*. Estes, além de terem interpretações reais interessantes, também possuem vários resultados teóricos relevantes, como resultados de inaproximabilidade e algoritmos que atingem a melhor razão possível a menos que P = NP.

O estudo de todos estes tópicos nos deu exemplos interessantes de reduções para prova de NP-completude e de inaproximabilidade; de aplicações de programação linear e teoria dos grafos; e de algoritmos em geral. Estudamos uma poderosa técnica geral para resolução de uma grande família de problemas (os problemas de gargalo).

Além disso, tivemos o desafio de tentar mostrar um conteúdo espalhado de maneira relativamente uniforme, e de tentar oferecer as demonstrações e explicações em uma linguagem mais acessível do que na maioria do material original estudado. Com isso, exercitamos a capacidade de escrita, sobretudo a de matemática e demonstrações. Algumas demonstrações dadas são bastante parecidas com as originais, mas, sempre que possível, procuramos adaptar a linguagem de modo que nos parecesse mais adequado ou compreensível.

Referências

[1] CHVÁTAL, V. *Linear programming.* Freeman, 1983.

[2] CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to algorithms.* 3. ed. The MIT Press, 2009.

[3] GAREY, M. R.; JOHNSON, D. S. *Computers and intractability: a guide to the theory of np-completeness.* Freeman, 1980.

[4] WILLIAMSON, D. P.; SHMOYS, D. B. *The design of approximation algorithms.* Cambridge University Press, 2011.

[5] GONZALEZ, T. F. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, v. 38, p. 293–306, 1985.

[6] HOCHBAUM, D. S.; SHMOYS, D. B. A unified approach to approximation algorithms for bottleneck problems. *Journal of the ACM*, v. 33, n. 3, p. 533–550, 1986.

[7] KHULLER, S.; PLESS, R.; SUSSMANN, Y. J. Fault tolerant k-center problems. *Theoretical Computer Science*, v. 242, p. 237–245, 2000.

[8] LIM, A.; RODRIGUES, B.; XANG, F.; XU, Z. k-center problems with minimum coverage. *Theoretical Computer Science*, 2004.

[9] SIPSER, M. *Introduction to the theory of computation.* 2. ed. Course Technology, Cengage Learning, 2006.

Parte II

Parte Subjetiva

6 O processo, desafios e frustrações

O tema deste trabalho de formatura foi escolhido por uma identificação que senti, ao longo do curso, com discplinas como Análise de Algoritmos, Introdução à Teoria dos Grafos e Otimização Combinatória. A professora Cristina me indicou o tema, explicando do que se tratava, e aceitei.

Foi com bastante prazer que comecei a estudar algoritmos de aproximação para a realização deste trabalho. Sempre senti que estava aproveitando muito do que havia "sofrido" ao longo da graduação, ao ter que relembrar resultados de cálculo diferencial e integral, estatística e programação linear para acompanhar os tópicos.

Sinto que o grande desafio na realização desse trabalho foi conciliar o tempo dedicado ao estudo e trabalho do TCC com as demais disciplinas. Sempre tive prazer em estudar o material relevante para meu trabalho, mas frequentemente me via impedido de fazer isso, até mesmo por períodos de semanas inteiras, por conta da demanda dos cursos.

Embora frustrantes, essas dificuldades em conciliar as diferentes tarefas sempre me permitiram perceber o quanto parecia prazeroso conseguir tempo para trabalhar no TCC!

7 A experiência do BCC

Escolhi cursar Ciência da Computação por gostar de matemática e ter interesse em alguns aspectos do que eu entendia por computação — por exemplo, a programação de computadores. Hoje diria que, portanto, não havia nenhuma razão particularmente forte para optar por esse curso. Mas diria também que foi uma escolha muito feliz.

Desde o princípio senti afinidade com o curso e, até para minha própria surpresa, nunca senti que estava no lugar errado. É claro que alguns tópicos estudados (e até uma ou outra disciplina) pareciam bem menos inspiradores que outros, e que alguns momentos foram extremamente cansativos e até frustrantes mesmo quando o motivo de tanto empenho eram estudos e trabalhos sobre coisas realmente empolgantes.

No entanto, o contato com os veteranos e professores, bem como a experiência de caminhar no curso, mostravam que (quase?) tudo que eu estava estudando era valioso: ou por ser importante para uma formação geral em computação, ou por colocar as primeiras pedras em alguns dos muitos caminhos que um computeiro pode escolher trilhar.

Para isso, ter assistido a várias apresentações de TCC, desde o primeiro ano, bem como participar de palestras e eventos, foi bastante importante. Posso ter entendido pouco de várias coisas que vi e ouvi nos dois primeiros anos, por exemplo, mas sempre era proveitoso de alguma forma. Me mostrava quantas coisas diferentes eu poderia tentar fazer, e o quanto seria bom tentar olhar com atenção e humildade para tudo com que eu me deparasse ao longo do BCC. Poder fazer um bom número de disciplinas eletivas também enriqueceu bastante essa visão das múltiplas possibilidades à frente.

Além disso, os momentos de maior esforço e cansaço se tornaram muito mais suportáveis por serem compartilhado com os vários bons amigos que encontrei entre meus colegas de BCC. E quando tais momentos passavam, ficava apenas a satisfação de sentir que mais um pequeno passo tinha sido dado (e um lembrete mental de não deixar acumular dois EPs para o fim de semana anterior a uma prova).

Sinto que cometi diversos erros ao longo da graduação, ou que, pelo menos, poderia ter feito várias coisas de modo melhor. No entanto, acho que tive a felicidade de seguir, mesmo sem perceber muito bem, vários dos passos que hoje eu consideraria os mais importantes para que o BCC seja uma boa experiência: formar uma turma (que seja mais do que um conjunto de colegas), participar de eventos,

conversar com alunos de outros anos, conversar com professores, pedir conselhos e tentar estudar todos os assuntos com boa vontade.

8 Relação entre o curso e o trabalho de formatura

Para este trabalho, como já disse, senti usar resultados de diversas e variadas disciplinas cursadas ao longo do bacharelado. Cito algumas delas:

- **MAC0110 - Introdução à Computação**
  MAC0122 - Princípios de Desenvolvimento de Algoritmos
  MAC0323 - Estruturas de Dados
  Estas disciplinas foram muito importantes para adquirir a intuição e capacidade de entendimento de algoritmos. Em especial, Estruturas de Dados foi importante para me convencer de que algumas descrições abstratas de algoritmos eram de fato implementáveis.
- **MAC0338 Análise de Algoritmos**
  MAC0430 Algoritmos e Complexidade de Computação
  Nessas disciplinas, aprendi fundamentos que são a motivação do trabalho. Em Análise tive o primeiro contato com teoria de complexidade, que foi o assunto que mais me atraiu na disciplina.
- **MAT0138 Álgebra I para Computação**
  MAT0111 - Cálculo Diferencial e Integral I
  MAT0139 - Álgebra Linear para Computação
  Considero essas disciplinas como base matemática imprescindível para a compreensão de várias demonstrações estudadas no trabalho.
- **MAC0325 - Otimização Combinatória**
  Muitos dos problemas estudados no trabalho são contextualizados nessa disciplina. Além disso, cursar MAC0325 contribuiu bastante para a minha formação geral, fornecendo um "repertório" de técnicas para modelar e resolver problemas, principalmente sobre grafos. Isso ajudou bastante na familiarização com o tema deste trabalho.
- **MAC0315 - Programação Linear**
  Nessa disciplina estudei pela primeira vez vários conceitos fundamentais para diversas técnicas de aproximação.
- **MAC0450 - Algoritmos de Aproximação**
  Disciplina que cursei já no fim da graduação, mas me deu uma visão mais ampla sobre a área e me permitiu o contato com problemas e técnicas importantes da área.
- **MAC0328 - Algoritmos em Grafos**
  MAC0320 - Introdução à Teoria dos Grafos
  Vários dos problemas e algoritmos estudados na minha iniciação científica utilizam conceitos com os quais tive contato e me familiarizei através dessas disciplinas.

9 Trabalhos futuros

Tenho intenção de continuar trabalhando nessa área, em especial com problemas de *clustering*. Estes estão aparecendo com grande destaque ultimamente, e podem modelar uma imensa variedade de problemas reais.

Meus planos para o futuro próximo são ingressar no mestrado, em que pretendo estudar problemas reais e modelá-los e resolvê-los via problemas de *clustering*, introduzindo, se necessário, novos problemas desse tipo.

Tenho interesse, também, em acompanhar a evolução de alguns resultados em aberto com os quais me deparei nesse trabalho, a exemplo da aproximação do α-*all-neighbor* k-*center*, e contribuir para a evolução da área no que possível.

10 Agradecimentos

Devo agradecer, em primeiro lugar, a meus pais e minha irmã, pelo apoio infalível que senti por parte deles durante toda minha formação pessoal e acadêmica. Sempre fui incentivado e ajudado em minhas escolhas, e isso foi da maior importância para que eu me visse motivado e capacitado a seguir crescendo. Agradeço de maneira muito especial, também, à Suzana, que conheci no início da graduação e é hoje parte fundamental da minha vida. Agradeço a ela por estar sempre a meu lado com tanta dedicação e carinho.

Gostaria, também, de agradecer a todos os meus amigos, que também foram importantes para mim na caminhada até aqui. Em especial, os amigos que fiz no IME foram de enorme importância, com sua companhia, ajuda e generosidade, para que eu conseguisse seguir em frente mesmo diante de grandes desafios ao longo da graduação. Não posso deixar de citar nomes como Felipe, Filipe, Gustavo, Heitor, Jackson, Leonardo, Nádia, Lucas, Renato, Thiago, Wallace e Wilson, mesmo temendo estar cometendo enorme injustiça para com tantos outros! Cada um deles me ajudou muito mesmo sem perceber.

Agradeço ao Giuliano, que sempre com boa vontade nos inspira e ajuda a melhorar o curso (e a vida dos alunos e professores).

Por fim, gostaria de agradecer aos professores do DCC por sempre ter a sensação de que eles se importam genuinamente com os alunos. Professores que eu gostaria de lembrar aqui de maneira especial são Carlinhos, Coelho, Gubi, Yoshiko, Paulo Feofiloff e Yoshiharu, por terem sido para mim exemplos especiais como tutores, ou mesmo por terem me inspirado com suas aulas e prazer pelo que ensinam. Preciso agradecer de maneira muito especial à Cris, a quem admiro muito como professora e sou grato por toda a paciência e dedicação ao me orientar na realização desse trabalho.