# O Problema do Multicorte Mínimo

Pedro Luis Furio Raphael
Orientador: Professor Paulo Feofiloff

8 de fevereiro de 2010

# Sumário

# Parte I

# Objetiva

## 1 Introdução

Considere um digrafo $G = (V, E)$, com uma função peso $p$ que associa um numero não-negativo a cada arco de $E$.

Diremos que um *par fonte-sorvedouro* é um par ordenado $(s, t)$ de vértices distintos. Diremos que $s$ é a *fonte* e $t$ é o *sorvedouro*.

Um par fonte-sorvedouro é *relevante* se existe um caminho da fonte ao sorvedouro no digrafo. Caso contrário, ele será *irrelevante*.

Diremos que um conjunto de arcos $E'$ *separa* um par fonte-sorvedouro $(s, t)$ se o par é irrelevante em $G - E'$.

Considere um conjunto $S$ de pares fonte-sorvedouro. Definiremos como *multicorte* de $S$ um conjunto de arcos que separa todos os pares elementos de $S$.

O problema do multicorte de peso mínimo consiste em encontrar um multicorte $M$ que tenha peso mínimo, ou seja, tal que $\sum_{e \in M} p(e)$ seja o menor possível.

Note que, pela restrição que a fonte tem que ser distinta do sorvedouro na definição de par fonte-sorvedouro, para qualquer instância existe um multicorte.

Este problema é NP-difícil [6, 4]. Existem algoritmos de aproximação para ele com fator de aproximação $O(\sqrt{n})$ [7], sendo $n$ é o número de vértices do digrafo.

Há uma versão do problema do multicorte mínimo para grafos não-digiridos. Esta versão não é equivalente a que estamos tratando aqui. [3]. Esta versão também é NP-difícil [6, 4], mas admite algoritmo de aproximação com fator $O(\log K)$ [5], sendo $K = |S|$.

Nesta monografia, analisaremos o problema para dois tipos específicos de digrafos, árvores e anéis, e mostraremos algoritmos que resolvem este problema em tempo polinomial para estes digrafos. Este trabalho é uma versão melhorada da parte principal dos artigos Costa et al. [2] e Bentz et al. [1].

## 2 Árvores

Um digrafo $T = (V, E)$ é uma *árvore* se respeita as seguintes propriedades:

- existe um único vértice em $T$, chamado raiz, cujo grau de entrada é 0.
- todos os demais vertices de $T$ tem grau de entrada 1;
- para todo vértice $v \in V$ existe um caminho da raiz a $v$.

Segue daí que o caminho da raiz para um vértice qualquer é único. Definiremos a *distância* da raiz até um vertice $v \in V$ como sendo o número de arcos do único caminho da raiz para $v$. Denotaremos por *dist(v)* esta distância.

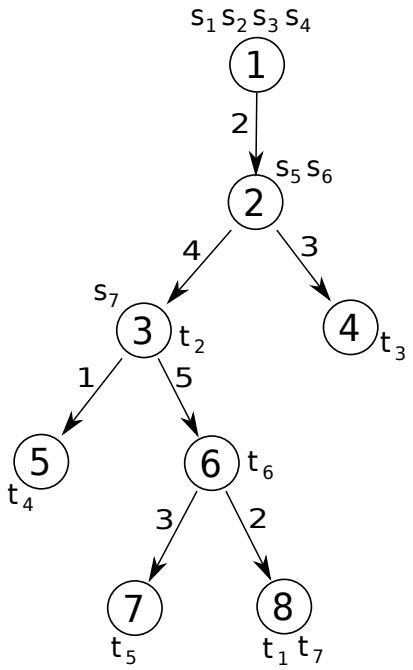


Figura 1: Exemplo de árvore, associada com uma função peso e um conjunto $S$. Nesta figura, $dist(8) = 4$. Esta figura esta no artigo Costa et al. [2]

Seja $(s, t)$ um par fonte-sorvedouro qualquer. Existe no máximo um caminho da fonte $s$ ao sorvedouro $t$. Isto pode ser deduzido intuitivamente. Comece por $t$ e ande na direção oposta ao arco que chega nele. Deste modo, uma das duas coisas será verdadeira: alcançaremos $s$ e portanto este é o único caminho de s a t, ou alcançaremos a raiz e não há caminho de $s$ a $t$. Fica assim bem definida a distância de $s$ a $t$.

## 2.1 Algoritmo

Mostraremos um algoritmo polinomial para árvores. Este recebe uma árvore $T$, uma função peso $p$ e um conjunto $S$ de pares fonte sorvedouro relevantes. O algoritmo devolve um multicorte de $S$ que tem peso mínimo.

ALGORITMO MultiCorteMínimoÁrvore$(T, p, S)$

01. sejam $(s_1, t_1), \ldots, (s_K, t_K)$ os elementos de $S$
    ordenados de modo que $dist(s_1) \leq \cdots \leq dist(s_K)$
02. $M \leftarrow \emptyset$
03. $p' \leftarrow p$
04. **para** $k \leftarrow K$ **decrescendo até** $1$ **faça**
05. $\quad c_k \leftarrow$ caminho de $s_k$ para $t_k$
06. $\quad f_k \leftarrow \min_{e \in c_k}\{p'(e)\}$
07. $\quad$ **para cada** arco $e \in c_k$ **faça**
08. $\quad\quad p'(e) \leftarrow p'(e) - f_k$
09. $\quad\quad$ **se** $p'(e) = 0$
10. $\quad\quad\quad$ **então** $M \leftarrow M \cup \{e\}$
11. **para** $k \leftarrow 1$ **até** $K$ **faça**
12. $\quad$ **se** $f_k > 0$ **então**
13. $\quad\quad c_k \leftarrow$ caminho de $s_k$ para $t_k$
14. $\quad\quad e \leftarrow$ primeiro arco de $c_k$ que está em $M$
15. $\quad\quad M \leftarrow M - c_k$
16. $\quad\quad M \leftarrow M \cup \{e\}$
17. **devolva** $M$

## 2.2 Correção

Este algoritmo tem duas fases, a primeira é o loop das linhas 4-10 e a segunda é o loop da linha 11-16. No final da primeira fase temos que

$$M \text{ é um multicorte de } S \tag{1}$$

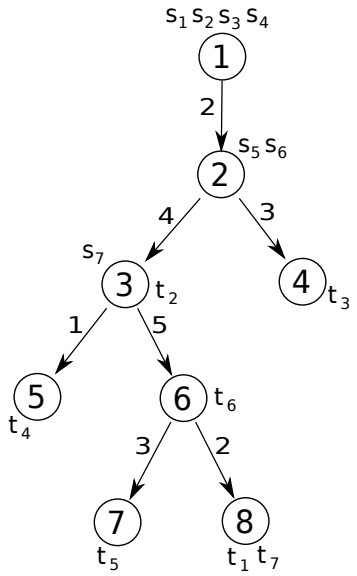


Figura 2: Exemplo de árvore, com uma função peso associada e um conjunto $S = \{(s_1, t_1), \ldots, (s_7, t_7)\}$ de pares fonte-sorvedouro. Note que $dist(s_1) \leq \cdots \leq dist(s_7)$. Um multicorte de peso mínimo para $S$ contém os arcos 1-2 e 3-6. Esta figura foi retirada do artigo Costa et al. [2]

e, para cada aresta $e$,

$$\sum_{i \in I(e)} f_i \leq p(e), \tag{2}$$

sendo que $I(e)$ é o conjunto $\{i \,|\, e \in c_i\}$, onde $c_i$ é o caminho de $s_i$ a $t_i$.

Na segunda fase, imediatamente antes de se testar a condição $k = K$ na linha 11 temos os seguintes invariantes:

*i1:* $M$ é um multicorte de $S$ e

*i2:* para todo $i = \{1, \ldots, k-1\}$, se $f_i > 0$ então $| c_i \cap M | \leq 1$.

A prova de *i2* é relativamente simples. A prova de *i1* merece mais cuidado.

*Prova de i1.* Basta mostrar que neste ponto (no início de uma iteração) o conjunto $M$ contém pelo menos um arco de cada caminho de $s_i$ a $t_i$, $i \in \{1, \ldots, K\}$. Na primeira iteração esta propriedade vale, como observamos em 1.

Suponha, agora, que no início uma iteração qualquer, $M$ é um multicorte. Provaremos que no final desta iteração $M$ continua a ser um multicorte.

Se $f_k = 0$ então nada foi feito pelo algoritmo nesta iteração e $M$ continua um multicorte. Suponha agora que $f_k > 0$. Temos que verificar, para $i \in \{1, \ldots, K\}$, se pelo menos um arco do caminho $c_i$ está em $M$.

*i)* Se $i = k$, o arco $e$ de $c_i$ continua em $M$.

*ii)* Se $c_i \cap c_k = \emptyset$, existe, no início da iteração, um arco de $c_i$ em $M$ e durante a iteração o algoritmo só suprime arcos de $c_k$.

*iii)* Se $c_i \cap c_k \neq \emptyset$ e $i < k$, então, se algum arco de $c_i$ estava em $M - c_k$ ele continuará em $M$. Senão, se todo arco de $c_i$ está em $M \cap c_k$ então o arco $e$ da linha 14 está em $c_i$ e portanto continuará em $M$.

*iv)* Se $c_i \cap c_k \neq \emptyset$ e $i > k$, então temos a seguinte situação: durante a execução do loop das linhas 4-10, o par fonte-sorvedouro $(s_i, t_i)$ foi tratado antes do par $(s_k, t_k)$. Como $f_k$ é positivo, $f_i$ também é positivo e pelo menos um arco $e_i$ de $c_i$ foi acrescentado a $M$. Se $e_i$ pertencesse a $c_k$ teriamos $p'(e_i) = 0$ antes do cálculo de $f_k$ e portanto $f_k$ seria nulo. Deste modo, este arco não será suprimido pela iteração corrente.

Concluimos que $M$ é um multicorte no fim da fase 2. □

Segue do invariante *i2* que

$$\sum_{i=1}^{K} f_i \geq p(M).$$

Por outro lado, para qualquer multicorte $M'$ e quaisquer $f'_1, \ldots, f'_k$ temos que

$$\sum_{i=1}^{K} f'_i \leq \sum_{e \in M'} \sum_{i \in I(e)} f'_i \leq \sum_{e \in M'} p(e) = p(M'),$$

por causa de (2).

Portanto o peso do multicorte $M$ é mínimo.

## 2.3 Análise de complexidade

A análise de complexidade deste algoritmo é simples. Ela depende da complexidade de se descobrir um caminho de um vértice a outro em uma árvore e do tamanho

do conjunto $S$. Para encontrar o caminho de um vértice a outro em uma árvore, digamos $v$ e $w$, partimos de $w$ e percorremos ao contrário o único arco que chega em $w$. Desta forma, supondo que o caminho de $v$ a $w$ existe, chegaremos a $v$ em no máximo $n$ passos, onde $n$ é o número de vértices da árvore. Assim, achar um caminho de um vértice a outro qualquer em uma árvore consome tempo $O(n)$.

A linha 5 consome tempo $O(n)$. Na linha 6 também consome tempo $O(n)$. O bloco de linhas 7-10 também consome tempo $O(n)$.

O **para** que começa na linha 3 e acaba na 10 é executado $K$ vezes, então, esta fase do algoritmo consome tempo $O(Kn)$.

O bloco de linhas 12-16 consome tempo $O(n)$. Este bloco é executado $K$ vezes, e portanto, a segunda fase consome tempo $O(Kn)$.

Desta forma, o algoritmo consome tempo $O(Kn)$.

Podemos fazer uma análise um pouco mais fina. O que o algoritmo faz é percorrer para cada par fonte-sorvedouro o caminho da fonte ao sorvedouro, ou seja, no final da execução o algoritmo realizou um número proporcional a soma das distâncias fonte-sorvedouro. Note que, em geral (especialmente para uma árvore aleatória), esta soma fica muito abaixo de $Kn$, pois o tamanho máximo de um caminho é a altura da árvore, que é muito menor que $n$ em média. Este fato será importante na seção 5.

### 2.3.1 Número de pares fonte-sorvedouro

O tamanho de $S$ pode ser tão grande quanto $n^2 - n$. Mesmo que todos os pares sejam relevantes, $S$ ainda pode ter $\frac{n^2-n}{2}$.

Diremos que um par $(s_j, t_j)$ está contido em um par $(s_i, t_i)$, se o caminho de $s_j$ a $t_j$ está contido no caminho de $s_i$ a $t_i$. Escrevemos $(s_j, t_j) \subseteq (s_i, t_i)$ se $(s_j, t_j)$ está contido em $(s_i, t_i)$.

Se $(s_j, t_j) \subseteq (s_i, t_i)$ então qualquer conjunto de arcos que separa $(s_j, t_j)$ também separa $(s_i, t_i)$.

Se $t_i = t_j$, então ou $(s_j, t_j) \subseteq (s_i, t_i)$ ou $(s_i, t_i) \subseteq (s_j, t_j)$, portanto, podemos supor sem perda de generalidade que $S$ não tem dois pares com o mesmo sorvedouro. Logo $K \leq n - 1$.[1] Se nos limitarmos a conjuntos $S$ deste tipo, podemos dizer que o algoritmo consome tempo $O(n^2)$.

[1]Por algum estranho motivo, o artigo [2] ignora este fato e introduz uma complexa estrutura de dados para lidar com o caso em que $K$ é da ordem de $n^2$.

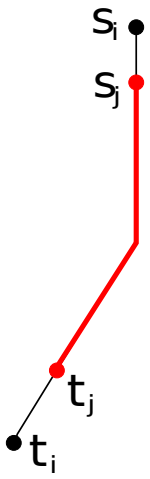


Figura 3: Exemplo de um caminho contido em outro. Claramente se separarmos $s_j$ de $t_j$ separamos $s_i$ de $t_i$.

# 3 Áneis

Um *anel* $R = (V, E)$ é um digrafo com as seguintes propriedades:

- todos os vértices tem grau de entrada 1 e grau de saida também 1;
- para todo par de vértices $v, w$ existe um caminho de $v$ para $w$.

Note que se retirarmos um arco de um anel ele passa a ser uma árvore cuja raiz é a ponta final do arco retirado.

Anéis são usados na prática na industria de Telecomunicações. Veja, por exemplo, o produto SONET ring[8].

## 3.1 Algoritmo

Segue o pseudo-código do algoritmo polinomial para anéis. O algoritmo recebe um anel $R$, uma função peso $p$ e um conjunto $S$ de pares fonte-sorvedouro relevantes e devolve um multicorte mínimo de $S$.

ALGORITMO MultiCorteMínimoAnel($R, p, S$)

01. $M \leftarrow E$
02. $(s, t) \leftarrow$ elemento qualquer de $S$

03. **para cada** arco $e$ no caminho de $s$ a $t$ **faça**
04. $T \leftarrow R - e$
05. $S' \leftarrow \{(s,t) \in S \mid (s,t)$ é relevante em $T\}$
06. $M' \leftarrow$ MultiCorteMínimoÁrvore$(R, p, S')$
07. $M' \leftarrow M' \cup \{e\}$
08. **se** $p(M') < p(M)$ **então** $\triangleright\ p(A) = \sum_{e \in A} p(e)$
09. $M \leftarrow M'$
10. **devolva** $M$

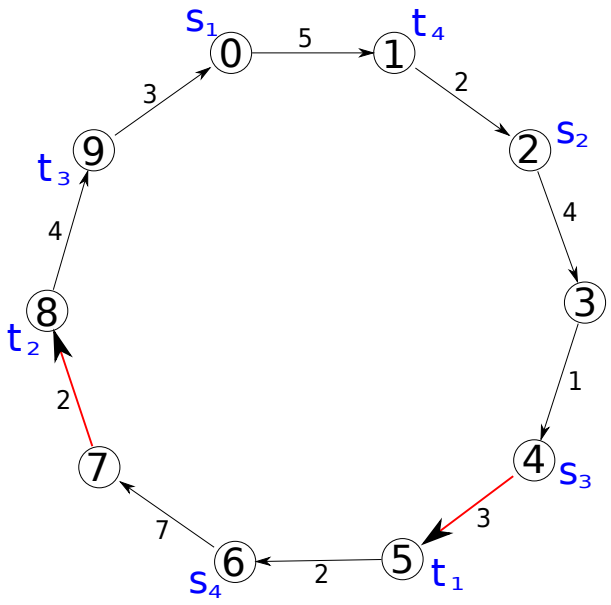


Figura 4: Exemplo de anel, associada com uma função peso e um conjunto $S$. O arco 4-5 e o arco 7-8 formam um multicorte de peso mínimo de $S$.

## 3.2 Correção

Suponha que temos um multicorte mínimo $M_*$ para $S$ no anel $R$. Na linha 2 do algoritmo escolhemos um par fonte-sorvedouro $(s,t)$ qualquer de $S$. Como $M_*$ é um multicorte, então ele contém um arco $e_*$ do caminho de $s$ a $t$. Em alguma iteração do loop das linhas 3-9 teremos $e = e_*$. Mas $M_* - \{e_*\}$ é um multicorte mínimo de $S$ em $T = R - \{e\}$. Se não fosse, então $M_*$ não seria mínimo para o anel, pois $M' \cup \{e\}$ (com $M'$ definido na linha 6) é um multicorte no anel $R$. Logo, $p(M \cup \{e_*\}) = p(M_*)$.

Concluimos que $M \cup \{e_*\}$ é um multicorte mínimo de $S$ em $R$.

### 3.3 Análise de Complexidade

A linha 6 consome tempo $O(Kn)$, sendo $n$ o número de vértices do anel $R$. Como o caminho de $s$ a $t$ em $R$ não contém mais do que $n-1$ arcos, a complexidade do algoritmo é $O(Kn^2)$.

## 4 Implementação

Para ver o que acontece na prática, implementei os dois algoritmos aqui descritos, na linguagem Java. A implementação esta disponível em www.linux.ime.usp.br/~plrapha/mac499/.

A implementação está dividida 3 pacotes de classes: main, classesUteis e digrafos.

O pacote main contém 3 classes, sendo estas Main.java, onde ocorre todo o processo de entrada e saida do programa, CorteMínimo.java, onde está implementando o algoritmo para árvores e para anéis descrito aqui e a classe Verificador.java, onde está implementado um método de verificação da resposta que consiste de duas fases, um que verifica se a soma dos fluxos é igual ao peso do multicorte encontrado pelo algoritmo e outro que verifica se este multicorte encontrado é realmente um multicorte.

O pacote digrafos contém 4 classes, sendo estas Digrafo.java, Vertice.java, Arco.java e Par.java. Estas classes representam a estrutura de um digrafo.

O pacote classesUteis é formado por 2 classes, sendo estas Gerador.java, que contém métodos que geram uma árvore com pesos aleatórios nos arcos e um conjunto $S$ de pares fonte-sorvedouro relevantes também aleatórios, e Ordena.java, onde está implementado um algoritmo de ordenação, que é usado no início do algoritmo. O algoritmo usado para gerar árvores aleatórias funciona da seguinte maneira: começa criando a raiz da árvore e após isso, para cada novo vértice, sorteia um que já está na árvore e cria um arco indo deste vértice para o novo, sendo que este arco tem peso aleatório. Já no caso de anéis, somente é necessário sortear o peso do novo arco, já que um vértice $i$ está sempre associado ao vértice $i+1$. Na geração dos pares, o algoritmo funciona da seguinte maneira: sorteia o sorvedouro, acha o caminho da raiz ao sorvedouro e sorteia um dos vértices dentre os que fazem parte do caminho

para ser a fonte. Além disso faz a verificação para garantir que não existam dois pares com o mesmo sorvedouro.

Para executar o programa primeiramente é necessário escolher qual algoritmo deve ser executado, passando como argumento na linha de comando o número 1 para árvores e 2 para anéis. Feito isso, existem duas formas de prover dados: preparar um arquivo em formato válido e passá-lo na linha de comando ou usar o gerador de digrafos e pares aleatórios implementado. Esta escolha também é feita através dos argumentos passados ao programa. No primeiro caso, o segundo argumento deve ser 1, seguido do nome do arquivo. No segundo deve ser 2, seguido pelo número de vértices que a árvore terá, seguido pelo número de pares que o conjunto $S$ terá.

No caso de árvores, um arquivo é válido se tem o seguinte formato: começa com $n$, o número de vértices da árvore, seguido por uma sequência de $n-1$ linhas com 3 números em cada, $(v, w, p)$, que denotam um arco de $v$ para $w$ com peso $p$. Estas linhas devem obedecer a seguinte regra: na linha $i$, $w$ deve ser $i+1$ e $v$ deve ser menor que $w$. A seguir, um número natural $K$, o número de pares fonte-sorvedouro de $S$ e, finalmente, uma sequência de $K$ linhas com 2 números $(s, t)$, que denotam a fonte e o sorvedouro do par. Note que os pares especificados devem ser relevantes.

No caso de anéis, um arquivo é válido se tem o seguinte formato: começa com $n$, o número de vértices do anel, seguido por uma sequência de $n$ linhas com 3 números em cada, $(v, w, p)$, que denotam um arco de $v$ para $w$ com peso $p$. Estas linhas devem obedecer a seguinte regra: na linha $i$, $w$ deve ser $i+1$ e $v$ deve ser $i$, com exceção da última linha, onde $w$ deve ser 0 e v deve ser $n-1$. A seguir, um número natural $K$, o número de pares fonte-sorvedouro de $S$ e uma sequência de $K$ linhas com 2 números $(s, t)$, que denotam a fonte e o sorvedouro do par. Note que $s$ deve ser diferente de $t$.

Após a entrada dos dados, o programa se encarregará de rodar o algoritmo e exibirá a resposta, além do tempo que demorou para encontrá-la.

# 5 Testes

O teste mais importante a se fazer é o de tempo de execução. Este teste tem o intuito de comprovar empiricamente os resultados obtidos na análise, bem como nos ajudar a prever o comportamento do algoritmo para determinada entrada. Este teste tem no eixo $y$ o tempo em milisegundos e no eixo $x$ a soma da distância dos

pares fonte-sorvedouro em $S$. O teste foi feito cosiderando a soma das distâncias no eixo $x$ ao invés do valor de $n$ pelo fato apresentado no final da seção 2.3.

Foi escrito um script para rodar os testes. Este script está disponível em www.linux.ime.usp.br/~plrapha/mac499/. Basicamente, o script apenas executa o programa para árvores com os argumentos necessários para que uma árvore e um conjunto $S$ sejam gerados de forma aleatória e coloca a resposta em um arquivo. Após ter gerado tal arquivo, o programa gnuplot foi usado para plotar o gráfico.

Nos testes feitos, o método que determina o multicorte foi chamado mais de uma vez, para a mesma entrada. Isto foi motivado pelo fato de que para instâncias pequenas, com $n$ e $K$ até 100000 o algoritmo é muito rápido, terminando o trabalho em poucos milisegundos, fazendo com que qualquer variação de tempo devido a ações do sistema operacional fiquem em evidência no gráfico. Nos gráficos apresentados abaixo, cada instância foi executada 30 vezes, e seus tempos somados.

Os gráficos das figuras 5 e 6 foram feitos usando $K = \frac{n}{2}$, com $n$ começando em 1000 e atingindo 30000, em passos de 500. Os dois gráficos correspondem a mesma entrada (geradas com a mesma semente). Isto sugere que as variações no tempo para uma mesma entrada acontecem devido a execuções de processos pelo sistema operacional, que fogem do controle do programa. Descontada as varoações o formato dos gráficos é consistente com o o que foi constatado na seção 2.3.

Os gráficos das figuras 7 e 8 foram feitos da mesma maneira que os anteriores, no entanto o valor de $K$ não aumenta, permanece em 500, enquanto o valor de $n$ começa em 1000 e cresce de 500 em 500, até atingir 100000. A variação do tempo com a soma das distancias fonte-sorvedouro é surpreendentemente errática.

A explicação para este fenomeno parece ser a seguinte: na linha 12 do algoritmo é feita a verificação $f_k > 0$, para um par fonte-sorvedouro $k \in S$. Caso essa expressão retorne "true", o algoritmo executa $O(n)$ operações. Caso essa expressão retorne "false", o algoritmo pula este passo. Desta forma, caso muitos pares fonte-sorvedouro estejam contidos uns nos outros, muitas vezes essa expressão devolverá "false" e o algoritmo economizará algum tempo.

Além disso, podemos perceber duas coisas: apenas alguns milisegundos separam o tempo máximo do mínimo, o que faz com que qualquer economia na execução faça a diferença no gráfico e, apesar de $n$ ter atingido um valor grande como 100000, a soma do comprimento dos caminhos ainda sim é muito pequena para uma boa comparação.

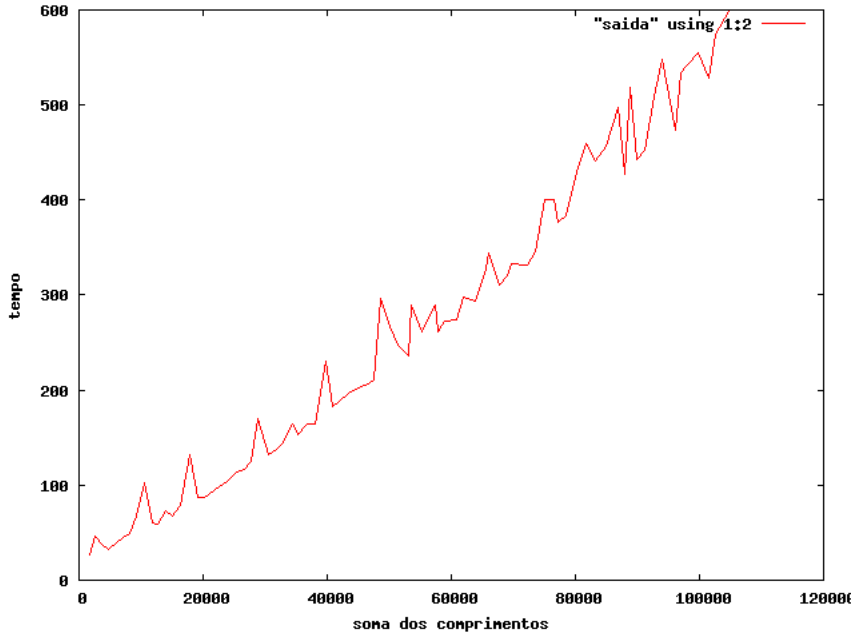


Figura 5: Gráfico de tempo em milisegundos pela soma das distancias fonte-sorvedouro. Neste teste $K = \frac{n}{2}$

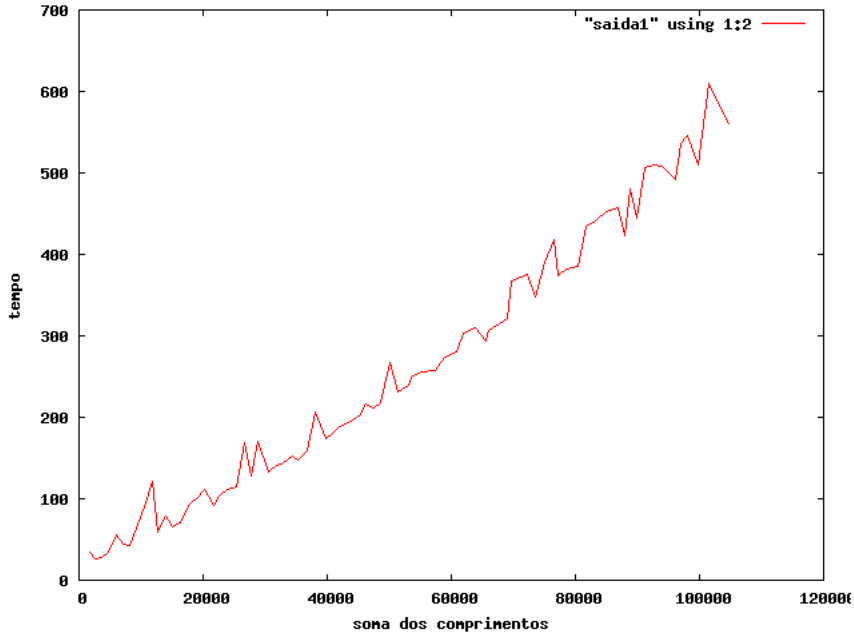


Figura 6: Gráfico de tempo em milisegundos pela soma das distancias fonte-sorvedouro. Neste teste $K = \frac{n}{2}$

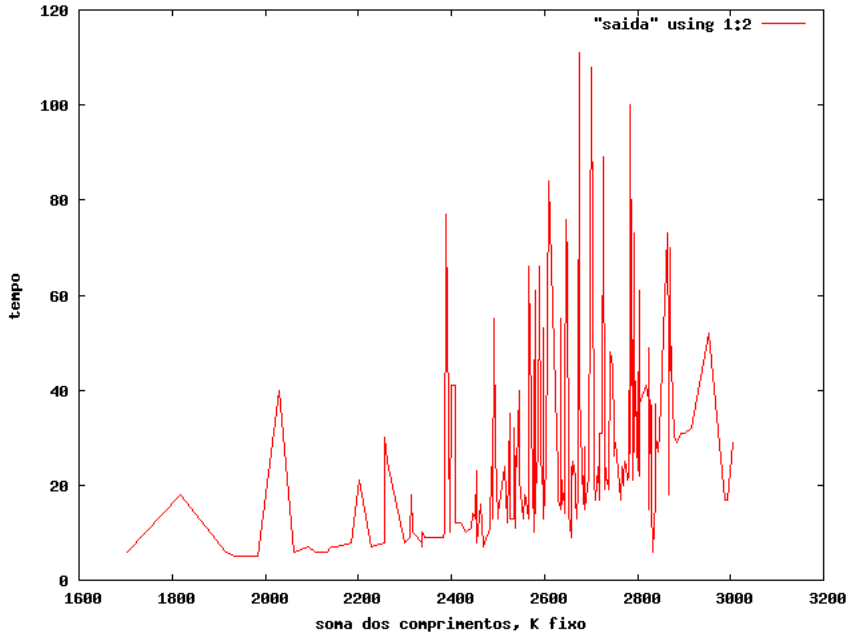


Figura 7: Gráfico de tempo em milisegundos pela soma das distancias fonte-sorvedouro. Neste teste $K = \frac{n}{2}$, com $K$ fixo valendo 500.

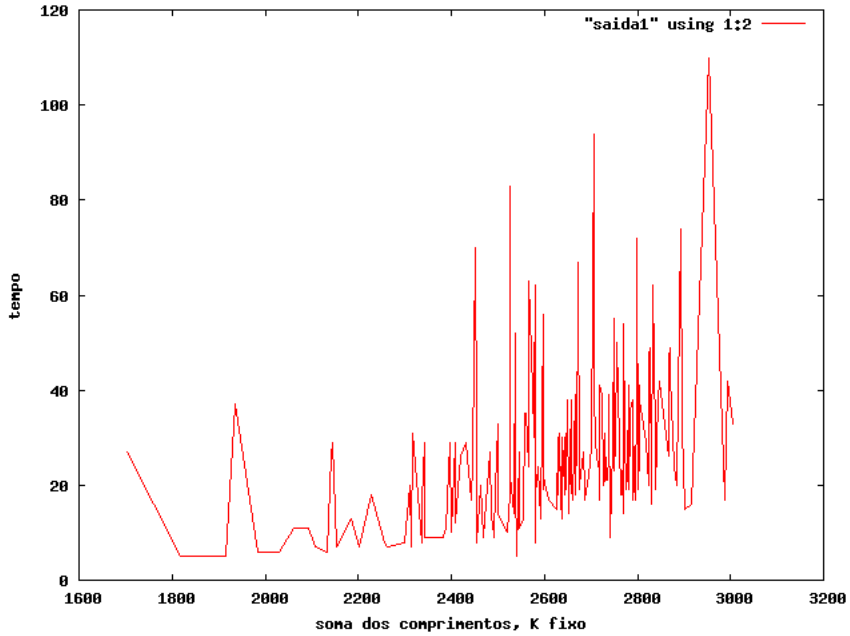


Figura 8: Gráfico de tempo em milisegundos pela soma das distancias fonte-sorvedouro. Neste teste $K = \frac{n}{2}$, com $K$ fixo valendo 500.

De uma maneira geral, levando em conta toda a discussão desta seção, concluimos que o algoritmo realmente se comporta como previsto na teoria.

# Parte II

# Subjetiva

## 1 Desafios

Os principais desafios encontrados foram: entender o problema, organizar os pensamentos, escrever o texto e o código e testar.

Meu trabalho baseou-se, principalmente, em dois artigos. Um deles se propunha a analisar o problema para árvores e a escrever um algoritmo polinomial para elas, mas apresentava uma linguagem pouco clara e usava uma estrutura de dados complicada para provar a complexidade. Já o outro se propunha apenas a analisar o problema para anéis de maneira muito superficial, citando outro artigo, cujo conteúdo não era trivial, de maneira vaga para dar uma idéia do algoritmo. Desta forma, com a ajuda do professor Paulo, tive muito trabalho separando as informações relevantes das que se desviavam do escopo deste trabalho.

Este trabalho de coleta de informações se estendeu durante os primeiros meses do ano letivo, ao mesmo tempo em que tentava organizar os pensamentos e fazer os primeiros rascunhos. Achei muito difícil, como ainda acho hoje, escrever um texto claro e preciso. Muitas revisões tiveram que ser feitas e muitas coisas simplesmente foram cortadas do trabalho, por fugir do tema principal, ou seja, desenvolver um algoritmo preciso para cada um dos problemas.

Já chegando perto do final do prazo, comecei a escrever o código e posteriormente os testes. Novamente não foi fácil, muitas revisões tiveram que ser feitas. Tomei a decisão de usar Java ao invés de C, o que me trouxe alguns problemas, principalmente na hora de testar. Como no início os resultados se mostravam longe do esperado, tanto em termos de clareza dos gráficos gerados quanto em termos de complexidade, tive que aprender como a estrutura interna do Java funciona, e acabei adquirindo um conhecimento inesperado, mas válido. Modifiquei a implementação que estava fazendo e a simplifiquei, deixando o código melhor e mais rápido.

Em todos esses desafios pude contar com a ajuda do professor Paulo, que tem uma vasta experiencia. Muitas reuniões foram feitas e pude extrair delas um conhecimento valioso que será muito útil, não só na área acadêmica, mas também no mercado de trabalho.

# 2 Disciplinas

Listarei aqui as máterias que considero as mais importantes do BCC.

- **Introdução a Computação.** Confesso que não tinha a menor idéia do que era o BCC até o início desta matéria no primeiro semestre. Na verdade, eu tinha uma idéia completamente errada e esta matéria me ajudou a corrigí-la.

- **Princípios de Desenvolvimento de Algoritmos.** A primeira matéria que entramos em contato com algoritmos de fato, e aprendemos a analisá-los. Primeiro contato com o C.

- **Estruturas de Dados.** Nesta matéria começamos a aprender as inúmeras estruturas existentes, quase todas não triviais, que nos permitem fazer coisas fantásticas gastando tempo comprovadamente pequeno. Muito importante como bagagem para as próximas matérias que listarei.

- **Algoritmos em Grafos.** Primeiro contato com grafos, seus usos e algoritmos básicos. Uma matéria com vários eps que tratam de problemas que aparecem na prática. Eu gostei muito desta matéria e foi ela que me motivou a fazer este trabalho tratando de um problema relacionado a grafos.

- **Análise de Algoritmos.** Finalmente aprendemos várias técnicas de análise e desenvolvimento de algoritmos, o que nos permitem melhorar e muito nossa capacidade de produzir bons códigos.

- **Desafios de Programação.** Esta matéria é aquela onde você realmente prova que tem os conceitos de estrutura de dados, grafos e análise consolidados na sua cabeça. Como não precisamos nos preocupar com layout do código podemos programar da maneira mais natural ao nosso próprio entendimento, o que nos ajuda a perceber falhas conceituais que passaram despercebidas nas outras matérias.

- **Otimização Combinatória.** Uma matéria muito teórica sobre um tópico muito legal. Nos ajuda a perceber se queremos seguir a carreira acadêmica ou não.

# 3 Futuro

A grande pergunta que eu e muitos da minha turma tentamos responder frequentemente: E agora? Mestrado, ou mercado de trabalho?

A resposta para esta pergunta ainda não está clara para mim, ora quero muito seguir a carreira acadêmica, ora gostaria de poder me dedicar em tempo integral a trabalhar. Se for continuar a carreira acadêmica, pretendo cursar algumas disciplinas como Introdução a Teoria do Grafos e Teoria dos Grafos, bem como Algoritmos de Aproximação. Acredito que estas matérias me permitiriam uma maior compreensão dos problemas com que me depararia no mestrado e de suas respectivas soluções.

# Referências


[1] C. Bentz, M.C. Costa, L. Létocart, and F. Roupin. Multicuts and integral multiflows in rings. *European Journal of Operational Research*, 196(3):1251–1254, 2009.

[2] M.C. Costa, L. Létocart, and F. Roupin. A greedy algorithm for multicut and integral multiflow in rooted trees. *Operations Research Letters*, 31(1):21–27, 2003.

[3] M.C. Costa, L. Létocart, and F. Roupin. Minimal multicut and maximal integer multiflow: a survey. *European Journal of Operational Research*, 162(1):55–69, 2005.

[4] E. Dahlhaus, D.S. Johnson, C.H. Papadimitriou, P.D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SIAM Journal on Computing*, 23 (4):864, 1994.

[5] N. Garg, V.V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi) cut theorems and their applications. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, page 707. ACM, 1993.

[6] N. Garg, V.V. Vazirani, and M. Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18(1):3–20, 1997.

[7] A. Gupta. Improved results for directed multicut. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 454–455. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2003.

[8] Wikipedia. Synchronous optical networking. `http://en.wikipedia.org/wiki/Synchronous_optical_networking`.