Aula 21 – Complexidade Computacional

Notas de Aula de Teoria dos Grafos

Prof[a.]: Patrícia D. L. Machado

UFCG – Unidade Acadêmica de Sistemas e Computação

Sumário

Nesta aula, apresentamos uma breve introdução a algoritmos e conceitos de complexidade computacional. Adicionalmente, apresentamos noções básicas sobre aproximações e heurísticas, tendo como exemplo uma heurística para determinar uma árvore geradora de peso mínimo.

Introdução

Complexidade computacional é um ramo da teoria da computação que tem como objetivo classificar problemas computacionais de acordo com sua dificuldade e estudar o relacionamento entre diferentes classes de problemas.

Um problema computacional é definido como uma tarefa que, em princípio, é passível de ser resolvida por um computador, ou seja, pode ser resolvida através da aplicação mecânica de passos, usualmente definidos como um algoritmo.

Um problema é considerado como inerentemente difícil se a sua solução requer recursos significativos, seja qual for o algoritmo usado.

A complexidade de um problema é investigada através de modelos matemáticos ou funções de cálculo que permitem quantificar recursos necessários para resolvê-los, tais como tempo e armazenamento. Também são utilizadas outras medidas de complexidade, tais como a quantidade de comunicação, o número de portas de um circuito e o número de processadores em uma computação paralela.

Um dos papéis da teoria da complexidade computacional é determinar os limites práticos sobre o que os computadores podem e não podem fazer.

Problemas e Algoritmos

A **instância de um problema** é o problema definido em termos de um exemplo específico. Como exemplo, uma instância do problema do menor caminho é o problema de encontrar o menor caminho entre 2 vértices de um grafo específico.

Um **algoritmo** para solucionar um problema é um procedimento computacional bem definido que aceita *qualquer instância de um problema* como entrada e retorna uma solução para o problema como saída.

Como já discutimos antes, muitos problemas práticos podem ser formulados em termos de grafos. Desta forma, o desenvolvimento de algoritmos eficientes para resolver estes problemas é um dos grandes desafios para a Teoria dos Grafos.

Para tal, dois aspectos precisam ser considerados:

- O algoritmo proposto é **correto** com relação às propriedades definidas para a solução (saída)
- O algoritmo é **eficiente**, ou seja, possui um tempo de resposta compatível com as expectativas para seu uso.

A correção é tratada por técnicas de verificação como a prova de teoremas enquanto que a segunda é tratada pela complexidade computacional. Em linhas gerais, a **complexidade computacional** de um algoritmo corresponde ao número de passos computacionais básicos necessários a sua execução.

No geral, a complexidade depende do tamanho e da natureza da entrada do algoritmo. No caso de grafos, a complexidade pode ser representada através de uma função que considere os valores n e $m,$ ou seja, *numero de vértices e arestas do grafo de entrada*. Neste caso, o algoritmo pode levar até $n * m$ passos para visitar o grafo inteiro.

O limite superior de passos necessários a execução de um algoritmo considera a execução do algoritmo **no pior caso**. Se a complexidade de um algoritmo tem seu limite superior representado por um polinômio, o algoritmo é chamado de **tempo polinomial**. Por exemplo, todo algoritmo que consome no máximo $8n^3 + 2n^2 + 5$ unidades de tempo, sendo n o tamanho da entrada é tempo polinomial.

Problemas podem ser agrupados em classes de acordo com sua ordem de complexidade. Uma classe de complexidade é um conjunto de problemas de complexidade relacionada. Usualmente, são definidas com base no tipo de problema computational, por exemplo, problemas de decisão; no modelo computacional, por exemplo máquinas de Turing determinísticas e não-determinísticas; e nas classes de funções que limitam sua complexidade com base em um recurso como, por exemplo, tempo polinomial, tempo exponencial, dentre outros. A Figura 1 ilustra as classes de complexidade mais conhecidas.

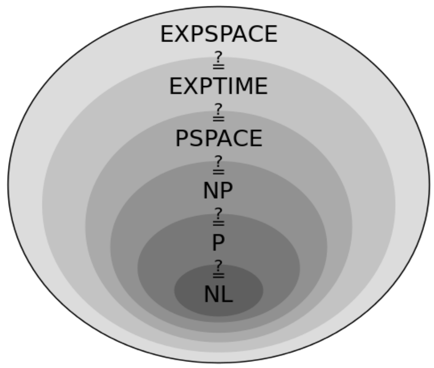


Figura 1[1]

O estudo de classes de complexidade computacional usualmente tem como enfoque problemas de decisão. Um problema de decisão é uma questão do tipo "sim" ou "não". Por exemplo, o grafo *G* apresenta um clique de tamanho *n*? O grafo *G* é bipartido?

Assume-se que tais problemas representam os problemas relacionados. Por exemplo, determinar se um grafo *G* possui um clique de tamanho *n* inclui o problema de encontrar no grafo um clique de tamanho *n*. O problema de determinar se um grafo G é bipartido, envolve encontrar uma bipartição (sim) ou um ciclo ímpar (não). Assim, apenas os problemas mais gerais de decisão costumam ser investigados por serem mais abrangentes.

Classes P e NP

A classe de problemas que podem ser resolvidos por algoritmos tempo polinomial é chamada de **P**. Este algoritmos são usualmente considerados computacionalmente viáveis, mesmo para grafos grandes. Em contraste, algoritmos com complexidade exponencial são usualmente não-viáveis até mesmo para entradas pequenas.

[1] http://en.wikipedia.org/wiki/Computational_complexity_theory

Algoritmos de busca em largura e altura são exemplos de algoritmos tempo polinomial. Outros exemplos de problemas na classe **P**: equação do segundo grau, máximo divisor comum, caminho mínimo.

Algoritmos para determinar cliques máximo não são tempo polinomial. Portanto, o problema de encontrar um clique máximo não faz parte da classe **P**.

Se, para uma determinada instância de um problema, cuja resposta é "sim" ou "não", existe um certificado que valida este fato e pode ser checado em tempo polinomial então o chamamos de **certificado sucinto**. Problemas na classe **P** usualmente possuem um certificado sucinto tanto para as respostas "sim" quanto as respostas "não". Exemplo: Determinar se um grafo é bipartido:

- Certificado sucinto que valida o "sim": a bipartição
  - Dada uma bipartição (X, Y), basta checar se todas as arestas tem um terminal em X e um terminal em Y;
- Certificado sucinto que valida o "não": um ciclo ímpar
  - Todo grafo que não é bipartido possui um ciclo ímpar.

As classes **NP** e **coNP** representam problemas para os quais algoritmos tempo polinomial podem ainda não serem conhecidos. Os representantes destas classes que não pertencem a P são problemas computacionalmente difíceis.

Com enfoque em problemas de decisão, podemos dizer que um problema pertence a classe **NP** se, dada uma instância do problema cuja resposta é "sim", existe um certificado que sucinto que valida este fato, mas não necessariamente existe um certificado sucinto que valide uma instância do problema cuja resposta é "não". Exemplo: Determinar se um grafo possui um Ciclo Hamiltoniano:

- Certificado que valida o "sim": o ciclo que pode ser checado em tempo polinomial.
- Certificado que valida o "não": não é conhecido

De forma análoga, um problema de decisão pertence a classe **co-NP** se, dada uma instância qualquer do problema cuja resposta é "não", existe um certificado sucinto que valida este fato, mas não necessariamente existe um certificado sucinto que valide uma instância do problema cuja resposta é "sim". Exemplo: Determina se uma expressão lógica é uma tautologia:

- Certificado que valida o "sim": não é conhecido

- Certificado que valida o "não": uma atribuição de valores-verdade para o qual a expressão avalia para falso pode ser checada em tempo polinomial.

Estas classes podem ser definidas através de uma hierarquia, onde a classe **P** esta contida em **NP** já que os problemas de **P** possuem um certificado sucinto para o "sim". A classe **P** esta contida em **coNP** já que os problemas de **P** possuem um certificado sucinto para o "não". E, consequentemente, **P** está contida na intersecao entre **NP** e **coNP**.

$$P \subseteq NP$$

$$P \subseteq coNP$$

$$P \subseteq NP \cap co - NP$$

A seguir, estudaremos algumas conjecturas que embasam esta hierarquia de classes.

P versus NP

Classificado como um dos problemas do milênio, pelo ***Clay Mathematics Institute***, a relação entre as classes P e NP é até o momento estabelecida pelas seguintes conjecturas.

Conjectura 1. $P \neq NP$

Acredita-se que a classe **P** seja diferentes da classe **NP**, apesar de não existir uma prova para tal. Portanto, este fato é estabelecido através da conjectura de *Cook-Edmonds-Levin*.

Para provar que **P** é diferente de **NP**, é necessário necessário mostrar que existem problemas em **NP** para os quais não existe algoritmo tempo polinomial. A conjectura se baseia no fato de que algoritmos para problemas **NP**-completos, tal como o do ciclo hamiltoniano e o do clique máximo, não são conhecidos, mas não conseguimos provar que eles não existem. Um problema é dito ser **NP**-completo quando todos os problemas da classe NP são tão difíceis quanto este.

Por outro lado, demonstrar que **P** = **NP** implica em encontrar um algoritmo tempo polinomial para um problema **NP**-completo qualquer. Qualquer problema **NP**-completo pode ser reduzido a outro **NP**-completo em tempo polinomial. Assim através de um algoritmo para um problema, podemos encontrar algoritmos tempo-polinomial para todos os outros, fazendo com que todos os problemas **NP** passem a ser também da classe **P**.

Quais as consequências práticas de **P = NP?**

A criptografia, tão importante em transações comerciais, baseia-se no uso de problemas **NP** para criptografar informações sigilosas. Quebrar a criptografia envolve resolver em tempo polimomial um problema da classe **NP**. Como isto não é possível, senhas não podem ser trivialmente quebradas. Desta forma, as consequências de **P = NP** poderiam ser potencialmente catastróficas para nossa sociedade.

Conjectura 2. $P = NP \cap co - NP$

Muitos problemas de decisão que pertencem a **NP ∩ co-NP** também pertencem a P. Caso especial: o problema de decidir se um número é primo. Apesar de ser conhecido o fato de que este pertence a **NP ∩ co-NP,** um algoritmo tempo polinomial foi descoberto apenas em 2004 quando então este problema passou a ser classificado como sendo da classe **P**. A partir de então a Conjectura 2 foi estabelecida.

Algoritmos de Aproximação

Para problemas cujos algoritmos conhecidos possuem complexidade além da polinomial, tal como o problema do caxeiro viajante (exponencial), pode-se optar por soluções não otimizadas. Dado um número real $t \geq 1$, um ***t-algoritmo de aproximação*** para um problema de otimização é um algoritmo que aceita qualquer instância de um problema e retorna uma solução viável cujo valor não é maior que t vezes o valor ótimo. Quanto menor o valor de t, melhor a aproximação.

Como exemplo, considere uma aproximação para o problema do caixeiro viajante, conhecida como *TSP Metric* (onde TSP vem de *Travel Salesman Problem*). Dado: um grafo ponderado completo *G* cujos pesos satisfazem a inequação: $w(xy) + w(yz) \geq w(xz), para\ x, y, z\ \in V(G)$, encontrar um ciclo hamiltoniano *C* de *G* com peso mínimo. *TSP Metric* é um 2-algoritmo de aproximação, tempo polinomial, para resolver este problema. No pior caso, ele retorna, em tempo polinomial, um ciclo hamiltoniano com 2 vezes o peso do ciclo de valor mínimo. A Figura 2 apresenta um exemplo de um grafo que satisfaz a inequação acima. A classe **TwoApproxMetricTSP** da **JGraphT** implementa este algoritmo.

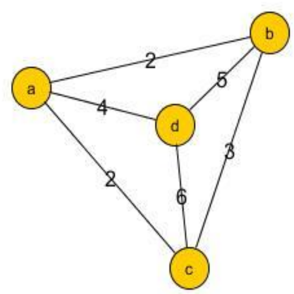


Figura 2

Heurísticas

Da mesma forma que aproximações, heurísticas também podem ser aplicadas para tratar, de forma não-otimizada, problemas para os quais não existem algoritmos tempo polinomial.

Uma **heurística** é um procedimento computacional, geralmente baseado em regras simples, a partir do qual soluções aproximadas podem ser geradas, usualmente com base em intuição. Uma **heurística gulosa** é um procedimento que seleciona a melhor opção disponível em cada estágio sem considerar consequências futuras. Como exemplo, considere uma heurística que comumente aplicamos para dar o troco de um pagamento usando a menor quantidade de moedas possível. Neste caso, o troco é construído iniciando pelas moedas de maior valor e usando tantas quanto possível.

Quando a heurística gera resultados ótimos, o procedimento é chamado de **algoritmo guloso**.

Algoritmo de Boruvka-Kruskal

Vejamos agora o exemplo de uma heurística gulosa que, por sempre retornar um resultado ótimo, é um algoritmo guloso, proposto por *Boruvka-Kruskal.*

A entrada é um grafo ponderado *G* e a saída é uma árvore geradora *T* de *G* com peso mínimo e o valor deste peso. Note que esta árvore não possui raiz.

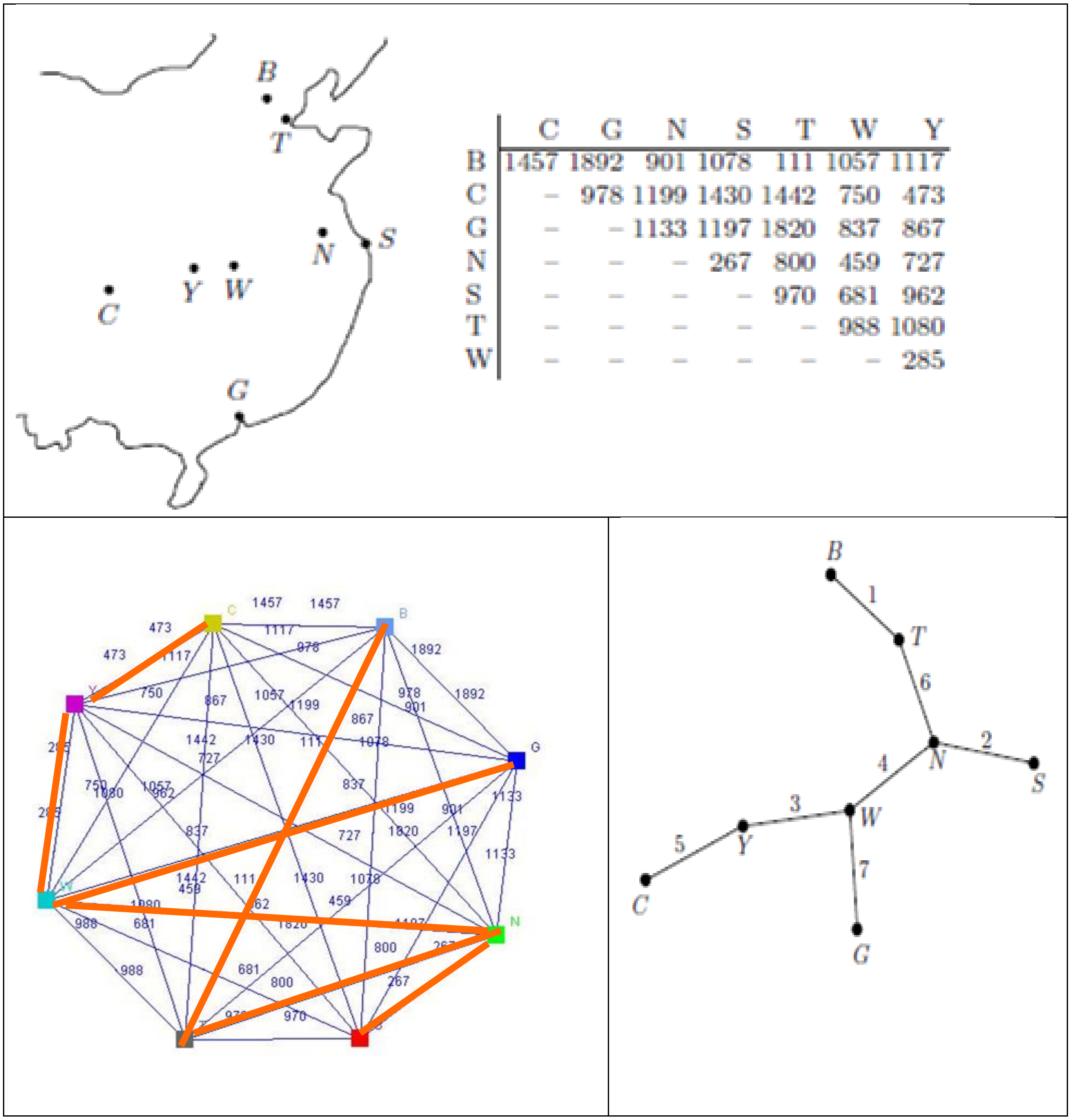
O algoritmo, listado abaixo, começa com um subgrafo vazio F e encontra uma sequencia de florestas aninhadas, terminando com uma árvore ótima. Esta sequência é construída através da adição de arestas, uma por vez, de tal forma que a aresta escolhida em cada estágio é uma de peso mínimo, desde que o subgrafo restante seja também uma floresta.

1	$F := \emptyset$, $w(F) := 0$ *(F é o conjunto de arestas da floresta atual)*
2	**Enquanto** $\exists\, e \in E \setminus F$ ***tal que*** $F \cup \{e\}$ seja o conjunto de arestas de uma floresta ***faça***
3	Escolha uma aresta e de peso mínimo
4	Substitua F por $F \cup \{e\}$ e $w(F)$ por $w(F) + w(e)$
5	**Fim Enquanto**
6	Retorne $(T = (V, F), w(F))$

No passo 1, o conjunto de arestas da floresta é inicializado como vazio. Nos passos sucessivos de 2 a 5, o algoritmo segue um ciclo repetitivo onde uma aresta e do grafo de peso mínimo é escolhida para ser adicionada a floresta. Será escolhida a aresta de menor peso tal que $F \cup \{e\}$ continue sendo um conjunto de arestas de uma floresta, ou seja, adição da aresta e a F não forme um ciclo na floresta que está sendo construída. O ciclo repetitivo pára quando a floresta for formada por uma única árvore que neste caso será a árvore geradora de peso mínimo.

Esta heurística foi inspirada por problemas como o do *grid elétrico chinês*. Considere as localidades representadas por letras no mapa da Figura 3. O grafo que modelo a conectividade entre as localidades é um grafo completo ponderado como podemos observar. Para melhor visualização, considere que as distâncias entre as localidades, ou pesos das arestas do grafo, estão representadas por uma matriz.

Como o grid deve ser construído de tal forma que a distância total de conexão seja a menor possível? A solução é encontrar a árvore geradora de peso mínimo. Aplicando a heurística de Boruvka-Kruskal encontramos a árvore representada a direita e cujas arestas estão hachuriadas no grafo em laranja. A ordem de inclusão das arestas na árvore está indicada pelo valor numérico expresso na aresta da árvore. Por exemplo, a primeira aresta a ser incluída foi BT, a segunda foi NS e assim sucessivamente.



	C	G	N	S	T	W	Y
B	1457	1892	901	1078	111	1057	1117
C	–	978	1199	1430	1442	750	473
G	–	–	1133	1197	1820	837	867
N	–	–	–	267	800	459	727
S	–	–	–	–	970	681	962
T	–	–	–	–	–	988	1080
W	–	–	–	–	–	–	285

Figura 3

Teorema 1. Toda árvore gerada pela heurística de Boruvka-Kruskal é uma árvore ótima.

De acordo com este teorema, Boruvka-kruskal é um algoritmo.

A **JGraphT** apresenta diferentes algoritmos para o problema de encontrar uma árvore geradora de peso mínimo disponíveis no pacote org.jgrapht.alg.spanning:

- **BoruvkaMinimumSpanningTree** (Complexidade da ordem: $O((E+V)logV)$) – Implementação da Heurística de Boruvka-Kruskal, na qual, em grafos onde arestas possuem pesos idênticos, arestas com pesos iguais são ordenadas lexicograficamente;
- **KruskalMinimumSpanningTree** (Complexidade da ordem: $O(ElogE)$) – Implementação da Heurística de Boruvka-Kruskal. Se o grafo for conectado, ele calcula a árvore geradora mínima, caso contrário, ele calcula a floresta geradora mínima.
- **PrimMinimumSpanningTree** (Complexidade da ordem: $O\big(|E|+|V|log(|V|)\big)$ – implementação do algoritmo de Prim que encontra uma árvore ou uma floresta geradora (caso grafo desconectado) de peso mínimo em um grafo não-direcionado.

onde |E| e |V| representam a quantidade de arestas e vértices do grafo, respectivamente.

Considerando o grafo apresentado na Figura 4, os algoritmos retornam as seguintes árvores:

- **BoruvkaMinimumSpanningTree**:
  Spanning-Tree [weight=16.0, edges=[(a : f), (b : e), (c : d), (a : b), (b : c)]]
- **KruskalMinimumSpanningTree**:
  Spanning-Tree [weight=16.0, edges=[(b : e), (a : b), (b : c), (c : d), (a : f)]]
- **PrimMinimumSpanningTree**:
  Spanning-Tree [weight=16.0, edges=[(f : a), (d : c), (c : b), (e : b), (b : a)]]

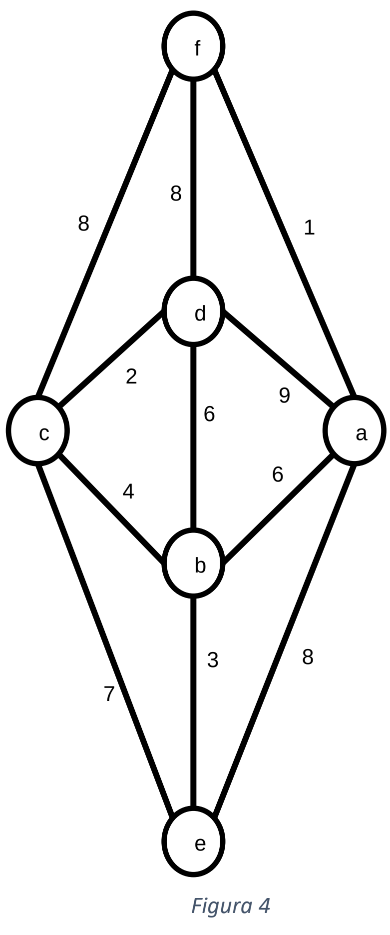


Figura 4

A fim de ilustrar as diferenças entre as ordens de complexidade dos diferentes algoritmos, considere a aproximação do fator tempo abaixo ilustrado para um grafo com 11 arestas e 6 vértices e para um outro grafo com 300 arestas e 100 vértices. Note que, a medida que aumentamos o tamanho do grafo, o algoritmo de PRIM passa a ter uma melhor performance, tal como previsto na função que determina as ordens de complexidade dos algoritmos considerados.

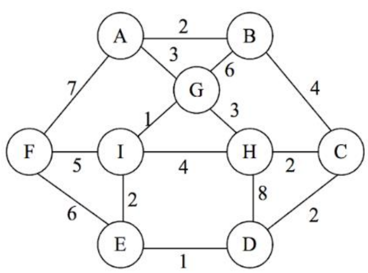
	Grafo 1		**Grafo 2**	
	|E| = 8	|V| = 6	|E| = 300	|V| = 100
Boruvka	25.0		1842.0	
Kruskal	16.6		1711.13	
Prim	18.7		760.5	

Exercícios Propostos

1. Modifique o algoritmo de Kruskal para determinar a árvore geradora de peso máximo.

2. Explique porque a conjecturas de Cook-Edmonds-Levin e Edmond podem ser verdadeiras? O que as tornariam falsas?

3. Aplique o algoritmo de BORUVKA-KRUSKAL no grafo abaixo.



4. (Exercício Adicional) Explique o algoritmo de Jarník-Prim (https://pt.wikipedia.org/wiki/Algoritmo_de_Prim) e compare com o algoritmo de BORUVKA-KRUSKAL.

Referências

J. A. Bondy and U. S. R. Murty.Graph Theory. Springer, 2008,2010.

- 8.1 e 8.5 (só o algoritmo de Kruskal)

Computational Complexity Theory in Wikipedia (http://en.wikipedia.org/wiki/Computational_complexity_theory)