

Módulo 3

Estruturas de Dados



Lição 6

Grafos

Versão 1.0 - Mai/2007

Autor

Joyce Avestro

Equipe

Joyce Avestro
 Florence Balagtas
 Rommel Feria
 Reginald Hutcherson
 Rebecca Ong
 John Paul Petines
 Sang Shin
 Raghavan Srinivas
 Matthew Thompson

Necessidades para os Exercícios**Sistemas Operacionais Suportados****NetBeans IDE 5.5** para os seguintes sistemas operacionais:

- Microsoft Windows XP Professional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Professional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware**Nota:** IDE NetBeans com resolução de tela em 1024x768 pixel

| Sistema Operacional | Processador | Memória | HD Livre |
|---------------------------------------|--|---------|----------|
| Microsoft Windows | 500 MHz Intel Pentium III workstation ou equivalente | 512 MB | 850 MB |
| Linux | 500 MHz Intel Pentium III workstation ou equivalente | 512 MB | 450 MB |
| Solaris OS (SPARC) | UltraSPARC II 450 MHz | 512 MB | 450 MB |
| Solaris OS (x86/x64 Platform Edition) | AMD Opteron 100 Série 1.8 GHz | 512 MB | 450 MB |
| Mac OS X | PowerPC G4 | 512 MB | 450 MB |

Configuração Recomendada de Hardware

| Sistema Operacional | Processador | Memória | HD Livre |
|---------------------------------------|--|---------|----------|
| Microsoft Windows | 1.4 GHz Intel Pentium III workstation ou equivalente | 1 GB | 1 GB |
| Linux | 1.4 GHz Intel Pentium III workstation ou equivalente | 1 GB | 850 MB |
| Solaris OS (SPARC) | UltraSPARC IIIi 1 GHz | 1 GB | 850 MB |
| Solaris OS (x86/x64 Platform Edition) | AMD Opteron 100 Series 1.8 GHz | 1 GB | 850 MB |
| Mac OS X | PowerPC G5 | 1 GB | 850 MB |

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris, Windows, e Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em <http://java.sun.com/j2se/1.5.0/download.html>
- Para **Mac OS X**, Java 2 Platform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: <http://developer.apple.com/java> (é necessário registrar o download da JDK).

Para mais informações: <http://www.netbeans.org/community/releases/55/relnotes.html>

Colaboradores que auxiliaram no processo de tradução e revisão

| | | |
|-----------------------------------|------------------------------------|------------------------------------|
| Alexandre Mori | Jacqueline Susann Barbosa | Mauro Regis de Sousa Lima |
| Alexis da Rocha Silva | João Paulo Cirino Silva de Novais | Namor de Sá e Silva |
| Aline Sabbatini da Silva Alves | João Vianney Barrozo Costa | Nolyanne Peixoto Brasil Vieira |
| Allan Wojcik da Silva | José Augusto Martins Nieviadonski | Paulo Afonso Corrêa |
| André Luiz Moreira | José Ricardo Carneiro | Paulo Oliveira Sampaio Reis |
| Anna Carolina Ferreira da Rocha | Kleberth Bezerra G. dos Santos | Pedro Antonio Pereira Miranda |
| Antonio Jose R. Alves Ramos | Kefreen Ryenz Batista Lacerda | Renato Alves Félix |
| Aurélio Soares Neto | Leonardo Leopoldo do Nascimento | Renê César Pereira |
| Bárbara Angélica de Jesus Barbosa | Lucas Vinícius Bibiano Thomé | Reydersen Magela dos Reis |
| Bruno da Silva Bonfim | Luciana Rocha de Oliveira | Ricardo Ulrich Bomfim |
| Bruno dos Santos Miranda | Luís Carlos André | Robson de Oliveira Cunha |
| Bruno Ferreira Rodrigues | Luiz Fernandes de Oliveira Junior | Rodrigo Fernandes Suguiura |
| Carlos Alexandre de Sene | Luiz Victor de Andrade Lima | Rodrigo Vaez |
| Carlos Eduardo Veras Neves | Marco Aurélio Martins Bessa | Ronie Dotzlaw |
| Cleber Ferreira de Sousa | Marcos Vinicius de Toledo | Rosely Moreira de Jesus |
| Everaldo de Souza Santos | Marcus Borges de S. Ramos de Pádua | Seire Pareja |
| Fabício Ribeiro Brigagão | Maria Carolina Ferreira da Silva | Silvio Sznifer |
| Fernando Antonio Mota Trinta | Massimiliano Giroldi | Tiago Gimenez Ribeiro |
| Frederico Dubiel | Mauricio da Silva Marinho | Vanderlei Carvalho Rodrigues Pinto |
| Givailson de Souza Neves | Mauro Cardoso Mortoni | Vanessa dos Santos Almeida |

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos** - Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo** - Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes** - Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso** - Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™

Rommel Fera – Criador da Iniciativa JEDI™

1. Objetivos

Essa lição cobre a nomenclatura para o Grafo *ADT*. Discute diferentes modos para representar um grafo. Os dois algoritmos de grafos transversais também são discutidos, assim como o problema de árvores geradoras de custo mínimo e o problema do caminho mais curto.

Ao final dessa lição, o estudante deverá ser capaz de:

- Explicar conceitos básicos e definições de **grafos**
- Discutir métodos de **representação de grafos**: matriz de adjacência e lista de adjacência
- Grafos Transversais usando os algoritmos ***depth-first search*** (busca em primeira-profundidade) e ***breadth-first search*** (busca em primeira-largura)
- Entender árvores geradoras de custo mínimo para grafos não-dirigidos usando o algoritmo de ***Prim*** e de ***Kruskal***
- Resolver o problema de menor caminho com início único usando o algoritmo de ***Dijkstra***
- Resolver o problema de menor caminho para todos os pares usando o algoritmo de ***Floyd***

2. Definições e Conceitos Relacionados

Um **grafo**, $G = (V, E)$, consiste de um conjunto finito não-vazio de vértices (ou *nodes*), V , e um conjunto de arestas, E . São exemplos de grafos:

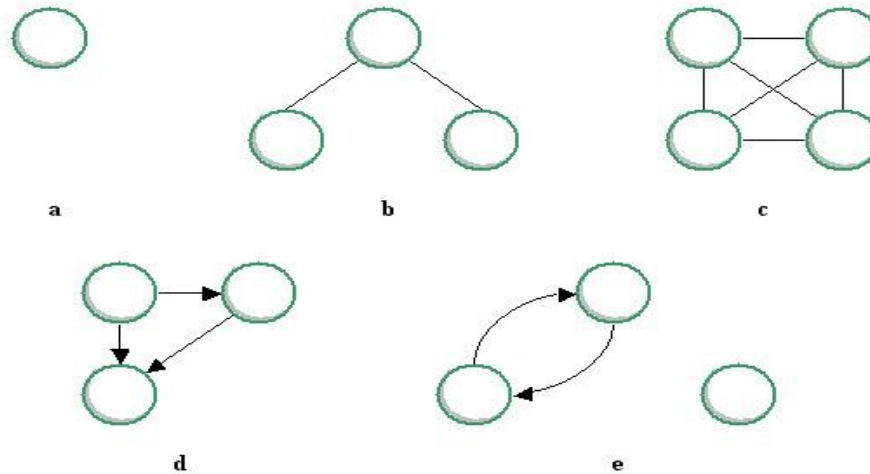
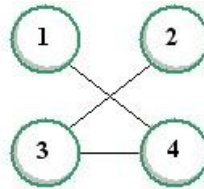


Figura 1: Exemplos de Grafos

Um grafo não-dirigido é um grafo no qual o par de vértices representando uma aresta é desordenado. Por exemplo, (i, j) e (j, i) representa o mesmo vértice.



$$V = \{1, 2, 3, 4\}$$

$$E = \{(1, 4), (2, 3), (3, 4)\}$$

Figura 2: Grafo não-dirigidos

Dois vértices i e j são **adjacentes** se $aresta(i, j)$ é uma aresta em E . A aresta (i, j) é conhecida com sendo **incidente** nos vértices i e j .

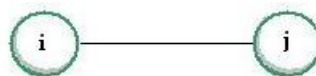


Figura 3: Aresta (i, j)

Um **grafo não-dirigido completo** é um grafo no qual uma aresta conecta todos os pares de vértices. Se um grafo não-dirigido completo tem n vértices, existirão $n(n-1)/2$ arestas nele.

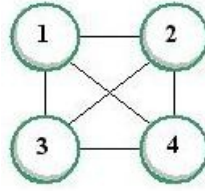
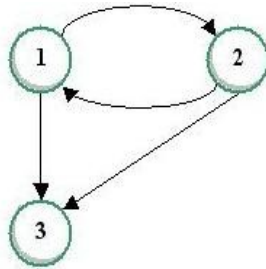


Figura 4: Grafo não-dirigido completo

Um **grafo dirigido** ou **dígrafo** é um grafo no qual cada aresta é representada por um par ordenado $\langle i, j \rangle$, onde **i** é o vértice principal e **j** é o vértice secundário da aresta. As arestas $\langle i, j \rangle$ e $\langle j, i \rangle$ são duas arestas distintas.



$$V = \{1, 2, 3\}$$

$$E = \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 1 \rangle, \langle 2, 3 \rangle\}$$

Figura 5: Grafo dirigido

Um **grafo dirigido completo** é um grafo no qual todos os pares de vértices **i** e **j** são conectados por duas arestas $\langle i, j \rangle$ e $\langle j, i \rangle$. Existem $n(n-1)$ arestas nele.

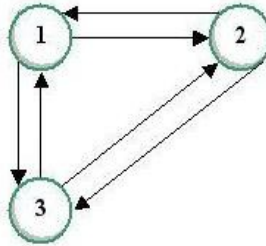
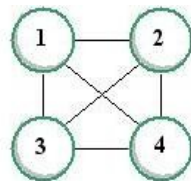
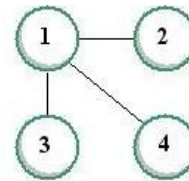


Figura 6: Grafo dirigido completo

Um subgrafo de um grafo não-dirigido (dirigido) $G = (V, E)$ é um grafo não-dirigido (dirigido) $G' = (V', E')$ no qual $V' \subseteq V$ e $E' \subseteq E$.



Graph G



Subgraph G'

Figura 7: Exemplo de subgrafo

Um **caminho** do vértice u para o vértice w em um grafo não-dirigido [dirigido] $G = (V, E)$ é uma sequência de vértices $v_0, v_1, v_2, \dots, v_{m-1}, v_m$ onde $v_0 \equiv u$ e $v_m \equiv w$, no qual $(v_0, v_1), (v_1, v_2), \dots, (v_{m-1}, v_m)$ [$\langle v_0, v_1 \rangle, \langle v_1, v_2 \rangle, \dots, \langle v_{m-1}, v_m \rangle$] são arestas em E .

O **comprimento de um caminho** refere-se ao número de arestas contidas nele.

Um **caminho simples** é um caminho no qual todos os vértices são distintos, exceto, possivelmente, o primeiro e o último.

Um caminho simples é um **ciclo simples** se ele tiver o mesmo vértice de começo e fim.

Dois vértices i e j são **conectados** se existir um caminho do vértice i para o vértice j . Se para cada par de vértices distintos i e j existir um caminho direto de i para j e para ambos os vértices, isso é conhecido como sendo **fortemente conectado**. Um subgrafo conectado máximo de um grafo não-dirigido é conhecido como **componente conectado em um grafo não-dirigido**. Em um grafo dirigido G , o **componente fortemente conectado** refere-se ao componente fortemente conectado em G .

Um **grafo ponderado** é um grafo com pesos e custos designados para suas arestas.

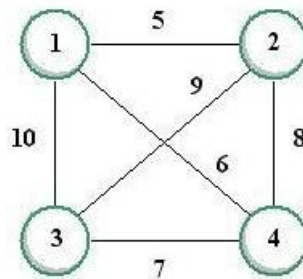
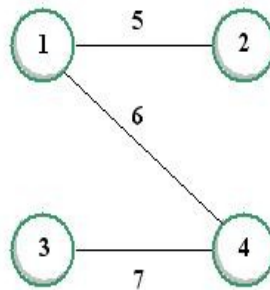


Figura 8: Grafo Ponderado

Uma **árvore geradora** é um subgrafo que conecta todos os vértices de um grafo. O **custo** de uma árvore geradora, se for ponderada, é a soma dos pesos dos galhos (arestas) da árvore geradora. Uma árvore geradora que tem o custo mínimo é conhecida como **árvore geradora de custo mínimo**. Isso não é necessariamente único para um determinado grafo.



Cost = 18

Figura 9: Árvore Geradora

3. Representação de Grafos

Existem diversas maneiras de se representar um grafo, e existem alguns fatores que devem ser considerados:

- Operações nos grafos
- Número de arestas relativas ao número de vértices no grafo

3.1. Matriz de Adjacência para grafos dirigidos

Um grafo dirigido pode ser representado usando uma matriz bidimensional, digamos A , com dimensões $n \times n$, onde n é o número de vértices. Os elementos de A são definidos como:

$$A(i,j) = 1 \text{ se a aresta } \langle i,j \rangle \text{ existir, } 1 \leq i, j \leq n \\ = 0 \text{ se a aresta } \langle i,j \rangle \text{ não existir}$$

A matriz de adjacência pode ser declarada como uma matriz de lógicos se o grafo não for ponderado. Se o grafo for ponderado, $A(i, j)$ é configurado para conter o custo da aresta $\langle i, j \rangle$, mas se não existir aresta $\langle i, j \rangle$ no grafo, $A(i, j)$ é configurado para um valor muito grande. A matriz é então chamada de **matriz custo-adjacência**.

Por exemplo, a representação de custo-adjacência do seguinte grafo é mostrada abaixo:

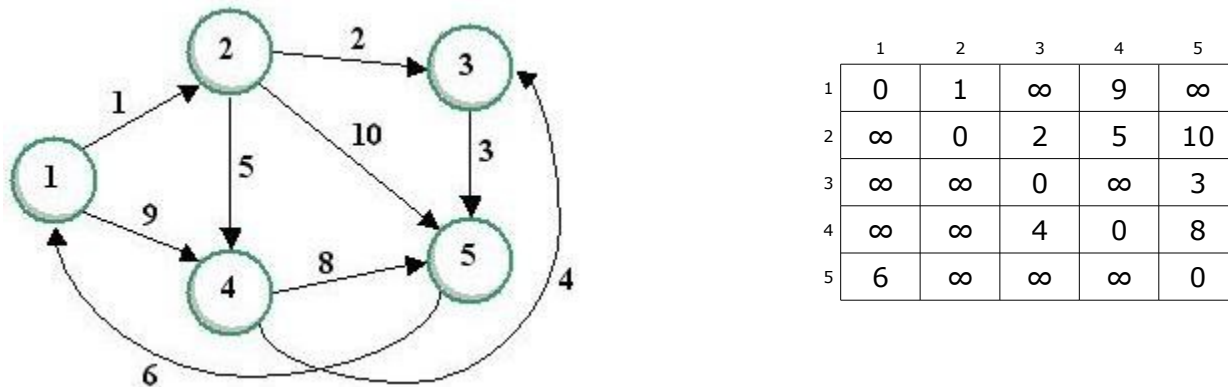


Figura 10: Representação da Matriz de Custo Adjacência para Grafos Dirigidos

Não é permitida referência a si própria, portanto, os elementos diagonais são sempre zeros. O número de elementos não-zero em A é menor ou igual a $n(n-1)$, o qual é o limite se o dígrafo é completo. O **grau de saída** do vértice i , ou seja, o número de setas derivadas dele, é o mesmo de números de elementos não-zero na linha i . O caso é parecido para o **grau de entrada** do vértice j , em que o número de setas apontando para ele é o mesmo do número de elementos não-zero na coluna j .

Com essa representação, saber se existe uma aresta $\langle i, j \rangle$ leva $O(1)$ tempo. No entanto, mesmo se o dígrafo tiver menos que n^2 arestas, a representação implica em requerimento de espaço de $O(n^2)$.

3.2. Lista de Adjacência para Grafos Dirigidos

Uma tabela seqüencial ou lista pode também ser usada para representar um dígrafo G em n vértices, digamos $LIST$. A lista é mantida como se para qualquer vértice i em G , $LIST(i)$ aponta para a lista de vértices adjacentes de i .

Por exemplo, segue a representação da lista de adjacência o grafo anterior:

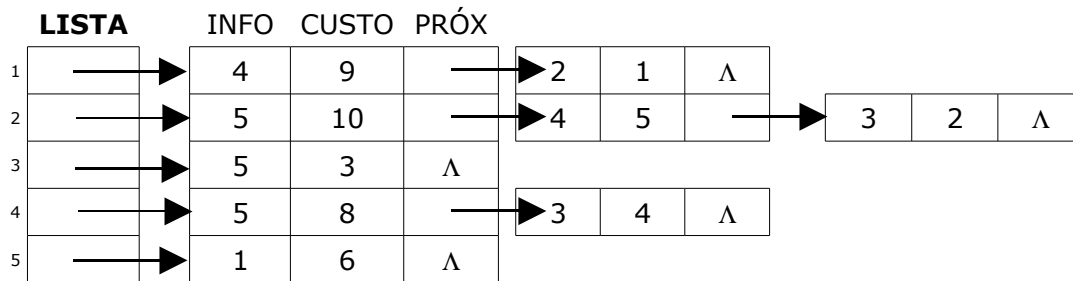


Figura 11: Representação dos custos em Lista de Adjacência para Grafos Direcionados

3.3. Matriz de adjacência para Grafos Não Direcionados

Assim como o diagrama, a matriz de adjacência pode ser usada para representar Grafos Não Direcionados. Entretanto, eles diferem no sentido que são *simétricos* para Grafos Não Direcionados, por exemplo, $A(i, j) = A(j, i)$. Para representar o Grafo são necessários elementos na menor ou maior diagonal. A outra parte pode ser considerada como **"Não se preocupe" (*)**.

Para um Grafo Não Direcionado G com n vértices, o número de elementos não-zero é $A \leq n(n-1)/2$. O limite superior é alcançado quando G é completado.

Por exemplo, o seguinte Grafo Não Direcionado não é ponderado. Por essa razão, a matriz de adjacência tem como entrada $A(i,j) = 1$ se a aresta existir, senão a entrada é 0.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | * | * | * | * |
| 2 | 0 | 0 | * | * | * |
| 3 | 0 | 1 | 0 | * | * |
| 4 | 1 | 0 | 1 | 0 | * |
| 5 | 1 | 1 | 0 | 0 | 0 |

Figura 12: Representação dos custos em uma Matriz de Adjacência para Grafos Não Direcionados

3.4. Lista de Adjacência para Grafos Não Direcionados

A representação é similar a lista de adjacência para Grafos Direcionados. Contudo, para Grafos Não Direcionados, existem duas entradas na lista para uma aresta (i, j).

Por exemplo, veja a seguir a representação da Lista de Adjacência do Grafo anterior:

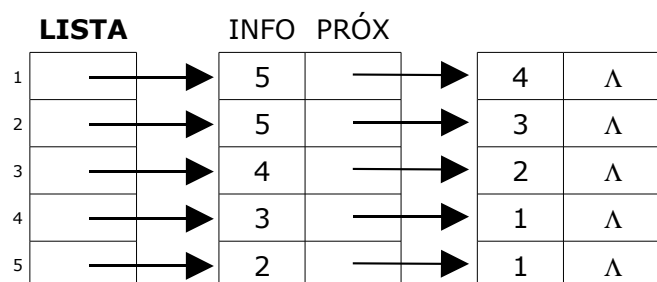
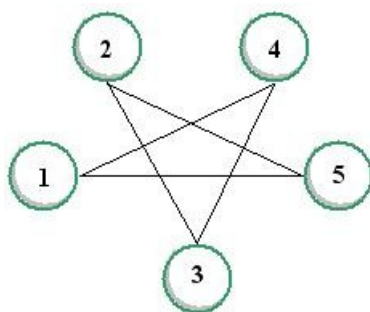


Figura 13: Representação dos custos em Lista de Adjacência para Grafos Não Direcionados

4. Percurso em Grafos

Um grafo, diferentemente de uma árvore, não tem o conceito de um *node* raiz no qual o método de percurso pode ser iniciado. Também não há uma ordem natural entre vértices de ou para o vértice mais recentemente visitado que indica o próximo a ser visitado. No percurso em grafos, também é importante perceber que desde que um vértice possa ser adjacente de ou para muitos vértices, há a possibilidade de encontrá-lo novamente. Por essa razão, existe a necessidade de indicar se um vértice já foi visitado.

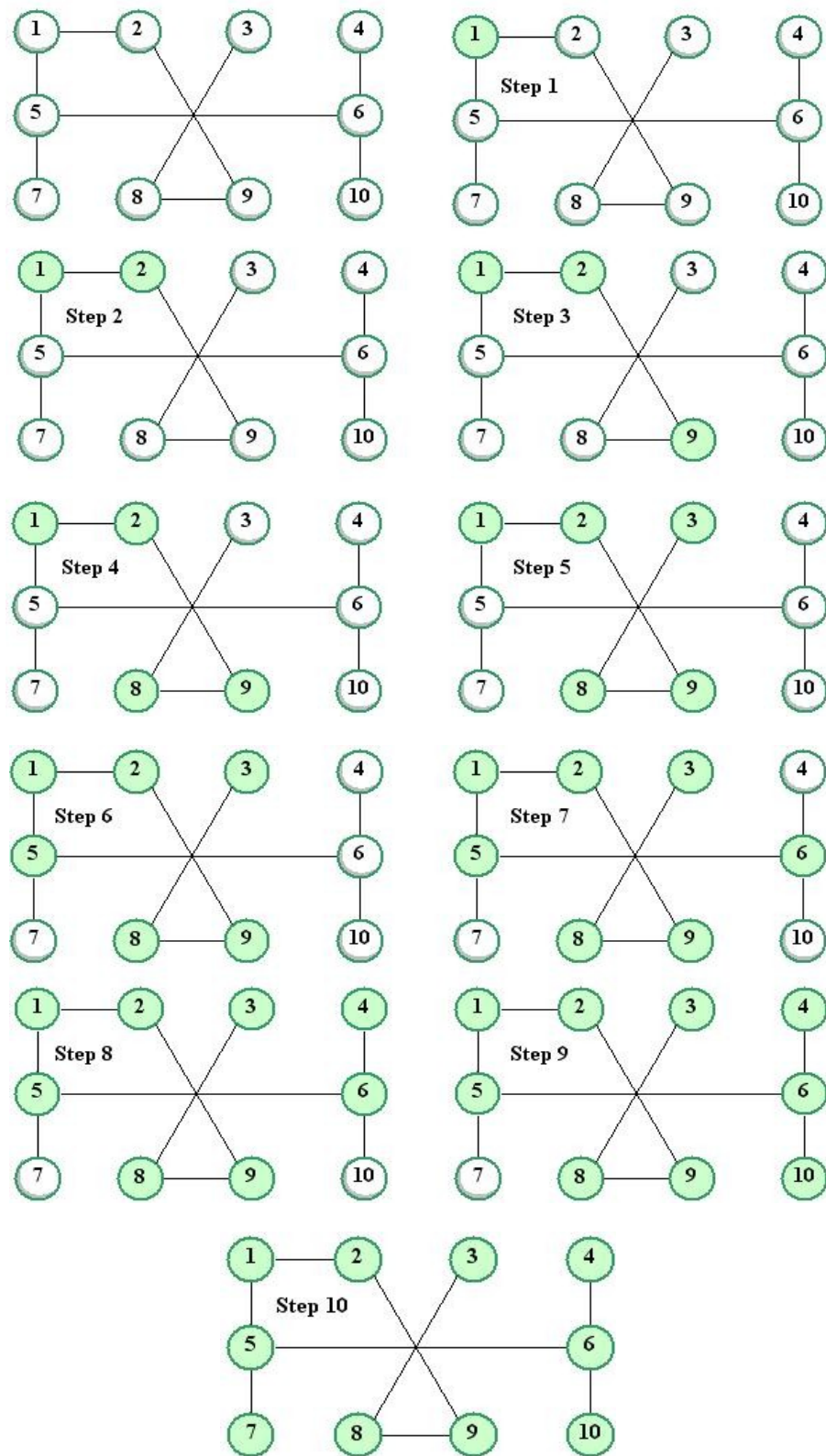
Nessa seção, cobriremos dois algoritmos de percurso em grafos: busca em profundidade e busca em largura.

4.1. Busca em Profundidade

Na busca em profundidade (*depth first search* - DFS), o grafo é percorrido da forma mais profunda possível. Dado um grafo com vértices marcados como não visitados, o percurso é executado conforme apresentado a seguir:

1. Selecione um vértice não visitado para iniciar. Se nenhum vértice for encontrado, a busca em profundidade termina
2. Marque o vértice inicial como visitado
3. Processar o vértice adjacente:
 - a) Selecione um vértice não visitado, por exemplo **u**, adjacente do vértice inicial
 - b) Marque o vértice adjacente como visitado
 - c) Inicie a busca em profundidade usando **u** como vértice inicial. Se nenhum vértice for encontrado, vá para o passo (1)
4. Se mais vértices adjacentes forem encontrados, vá para o passo (3c)

Sempre que houver ambigüidade em relação a qual deve ser o próximo vértice a ser visitado, no caso de existirem muitos vértices adjacentes, aquele com o menor número deve ser escolhido. Por exemplo, iniciando no vértice 1, a busca em profundidade percorrerá os elementos conforme o seguinte grafo:



1 2 9 8 3 5 6 4 10 7

Figura 14: Exemplo de Busca em Profundidade

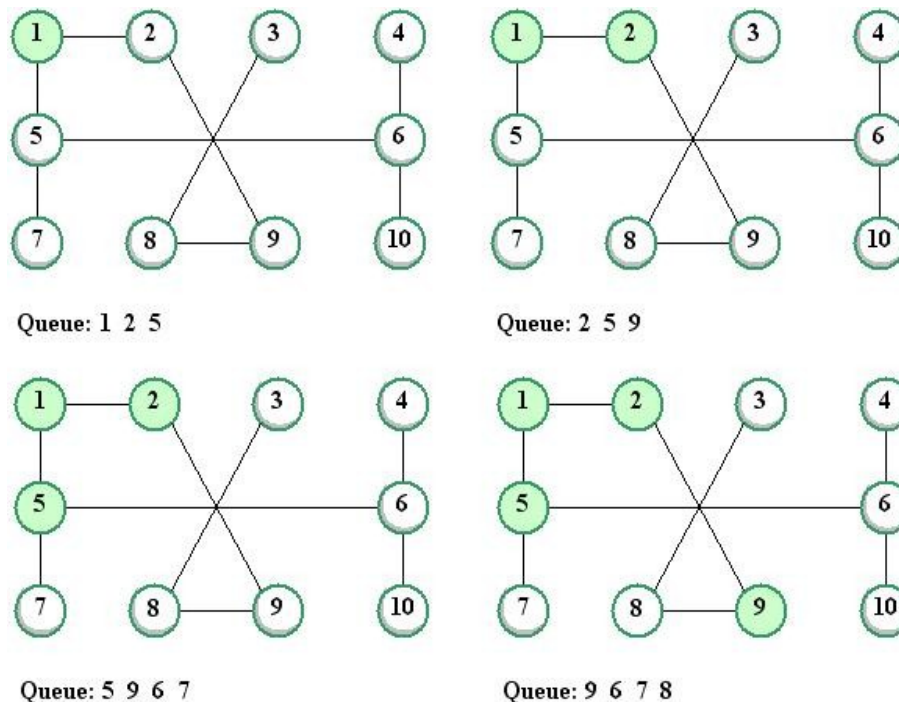
4.2. Busca em Primeira Largura

A Busca em Primeira Largura (*Breadth First Search* - BFS) percorre o grafo da forma mais ampla o possível. Dado um grafo com vértices marcados como não visitados, o percurso é executado da seguinte forma:

1. Selecione um vértice não visitado, por exemplo *i*, e marque-o como visitado. Então, busque o mais amplamente possível partindo de *i* e visitando seus vértices adjacentes.
2. Repetir (1) até que todos os vértices no grafo sejam visitados.

Assim como na busca em profundidade, no caso de dúvida sobre qual *node* será o próximo a ser visitado, deve-se considerar a ordem numérica crescente dos elementos.

Por exemplo, iniciando no vértice 1, a busca em largura percorrerá os elementos conforme o seguinte grafo:



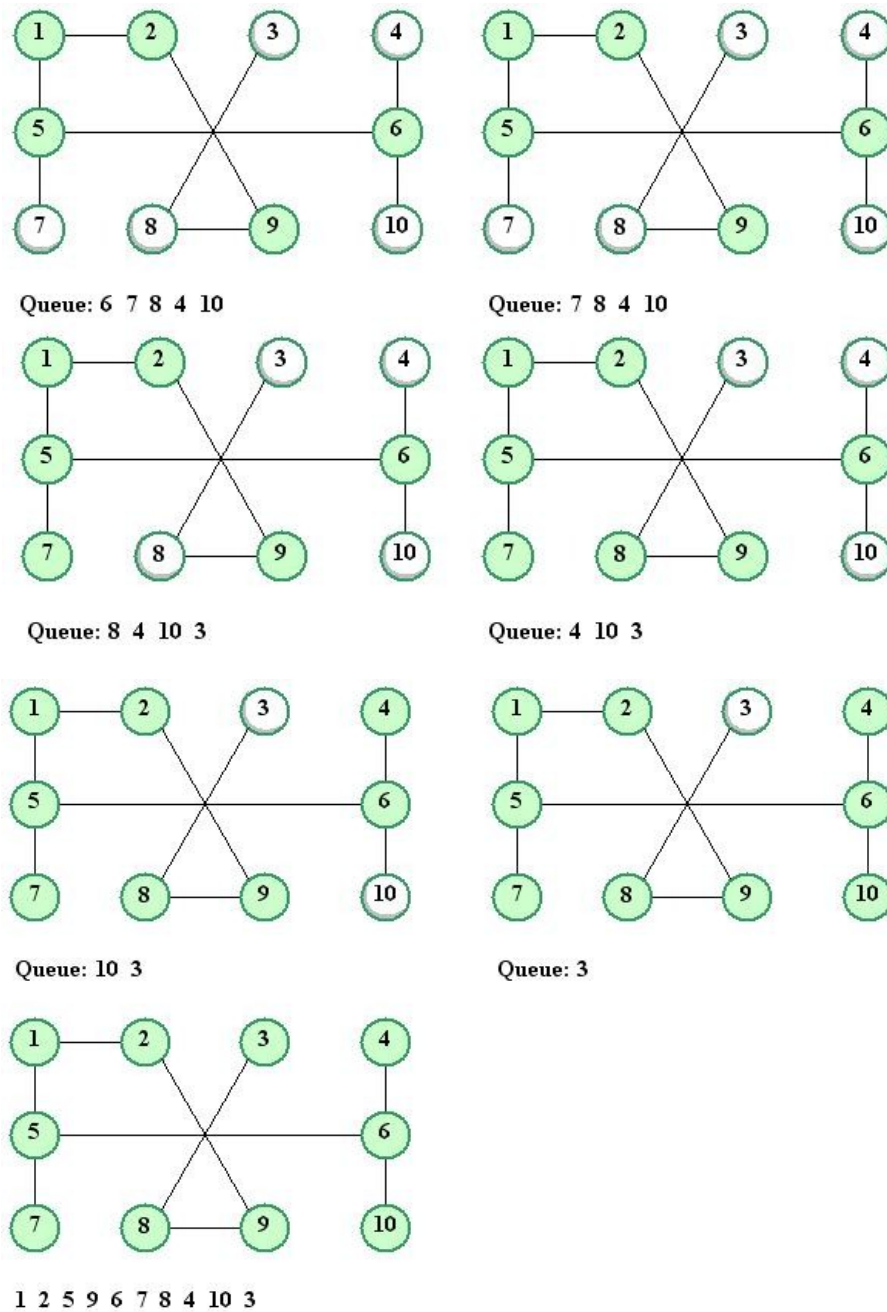


Figura 15: Exemplo de Busca em Largura

5. Árvore Geradora de Custo Mínimo para Grafos Não Direcionados

A Árvore Geradora de Custo Mínimo (*Minimum Cost Spanning Tree* - MST), como definida anteriormente, é um subgrafo de um dado grafo G , no qual todos os vértices estão conectados e tem o custo mais baixo. Isso é particularmente útil para encontrar o caminho mais barato para conectar computadores em uma rede, bem como aplicações similares.

Encontrar árvore geradora de custo mínimo para um grafo não direcionado usando abordagem de força bruta não é aconselhável se o número de árvores geradoras para n vértices distintos seja n^{n-2} . Por essa razão, é imperativo usar outra abordagem na busca da árvore geradora de custo mínimo e nessa lição, iremos cobrir algoritmos que utilizam uma **abordagem gulosa**. Nessa abordagem, uma sequência de escolhas oportunistas terão êxito na busca pelo ótimo global. Para resolver o problema da árvore geradora de custo mínimo, usaremos os algoritmos de **Prim** e **Kruskal**, os quais são, ambos, algoritmos gulosos.

5.1. Teorema MST

Considere $G = (V, E)$ um grafo não-dirigido, ponderado e conexo. Considere U seja algum conjunto apropriado de V e (u, v) seja uma aresta de menor custo tal que $u \in U$ e $v \in (V - U)$. Existe uma árvore geradora de custo mínimo T tal que (u, v) é uma aresta em T .

5.2. Algoritmo Prim

Esse algoritmo encontra a aresta de menor custo ligando algum vértice U para um vértice v em $(V - U)$ para cada passo do algoritmo:

Considere $G = (V, E)$ um grafo não-dirigido, ponderado e conexo. Considere que U indica o conjunto de vértices escolhidos e T indica o conjunto de arestas preparadas incluído em alguma instância do algoritmo.

1. Escolha um vértice inicial de V e coloque-o em U
2. Entre os vértices em $V - U$ escolha aquele vértice, v , que é conexo a algum vértice, u , em U por uma aresta de menor custo. Adicione vértice v para U e a aresta (u, v) para T
3. Repita (2) até $U = V$, em que, T é uma árvore geradora de custo mínimo para G

Por exemplo,

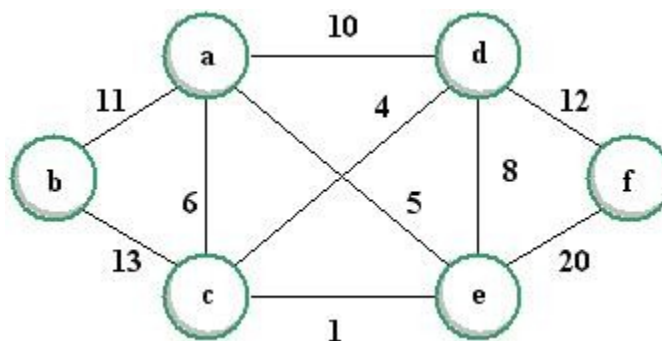


Figura 16: Um grafo não-dirigido ponderado

Tomando **a** como o vértice de partida, a figura a seguir mostra a execução do algoritmo *Prim* para resolver o problema MST:

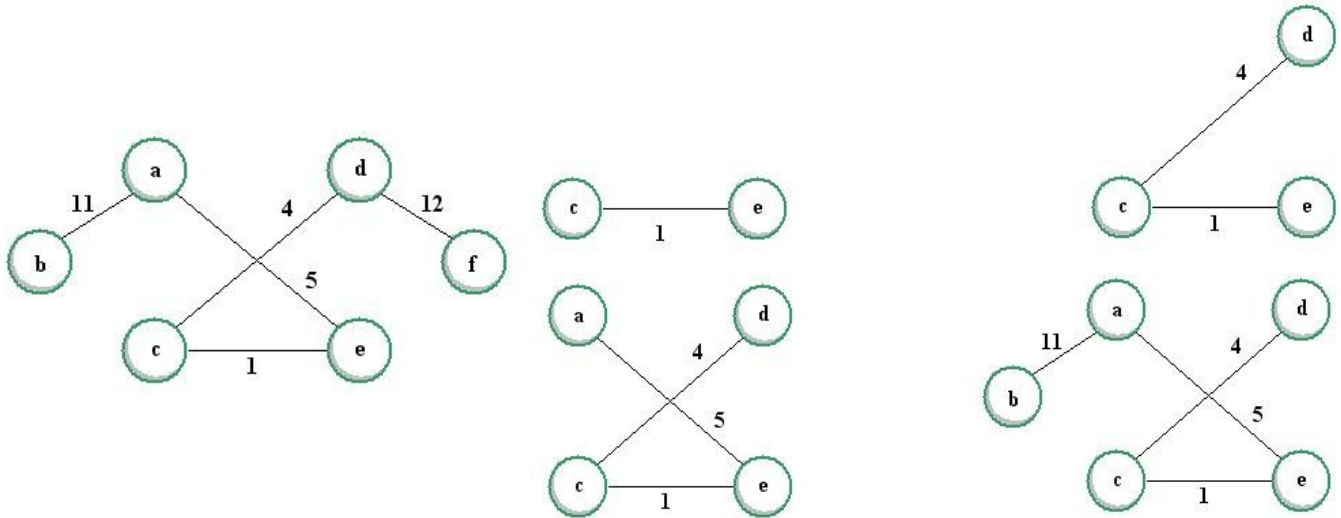


Figura 17: Resultado da aplicação do algoritmo *Prim* sobre o grafo anterior

5.3. Algoritmo *Kruskal*

Outro importante algoritmo usado para encontrar a MST foi desenvolvido por *Kruskal*. Nesse algoritmo, os vértices são listados em ordem crescente de peso. A primeira aresta a ser adicionada em **T**, que é o MST, é a de menor custo. Uma extremidade é considerada se pelo menos um dos vértices não estiver na longe da árvore encontrada.

Agora o algoritmo:

Considere $G = (V, E)$ um grafo não-dirigido, ponderado e conexo. A árvore geradora de custo mínimo, **T**, é construída aresta por aresta, com as arestas consideradas em ordem crescente de seus custos.

1. Escolha a aresta com o baixo custo como a aresta inicial.
2. A aresta de baixo custo entre as arestas restantes em **E** é considerada para inclusão em **T**. Se o ciclo for criado, a aresta em **T** é rejeitada.

Por exemplo,

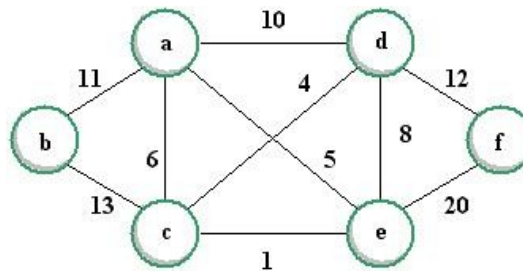
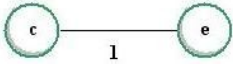
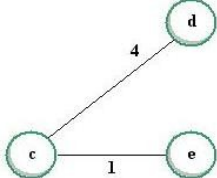
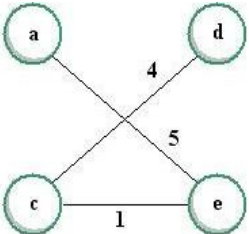
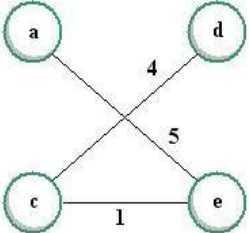


Figura 18: Um grafo não-dirigido ponderado

A tabela a seguir mostra a execução do algoritmo **Kruskal** para resolver o problema MST do grafo acima:

| Arestas | MST | U | V-U | Comentário |
|---|--|------------|---------------------|--|
| (c, e) – 1 (c, d) – 4 (a, e) – 5 (a, c) – 6 (d, e) – 8 (a, d) – 10 (a, b) – 11 (d, f) – 12 (b, c) – 13 (e, f) – 20 | | - | a, b, c, d, e, f | Lista de arestas em ordem crescente de peso |
| (c, e) – 1 aceito (c, d) – 4 (a, e) – 5 (a, c) – 6 (d, e) – 8 (a, d) – 10 (a, b) – 11 (d, f) – 12 (b, c) – 13 (e, f) – 20 | (c, e) – 1 | c, e | a, b, d, f | c e e não em U  |
| (c, d) – 4 aceito (a, e) – 5 (a, c) – 6 (d, e) – 8 (a, d) – 10 (a, b) – 11 (d, f) – 12 (b, c) – 13 (e, f) – 20 | (c, e) – 1 (c, d) – 4 | c, d, e | a, b, f | d não em U  |
| (a, e) – 5 aceito (a, c) – 6 (d, e) – 8 (a, d) – 10 (a, b) – 11 (d, f) – 12 (b, c) – 13 (e, f) – 20 | (c, e) – 1 (c, d) – 4 (a, e) – 5 | a, c, d, e | b, f | a não em U  |
| (a, c) – 6 rejeitado (d, e) – 8 (a, d) – 10 (a, b) – 11 (d, f) – 12 (b, c) – 13 (e, f) – 20 | (c, e) – 1 (c, d) – 4 (a, e) – 5 | a, c, d, e | b, f | a e c estão em U  |

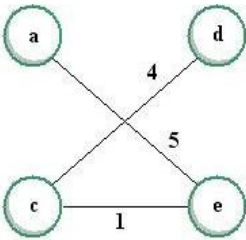
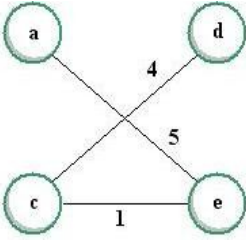
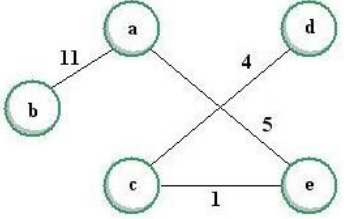
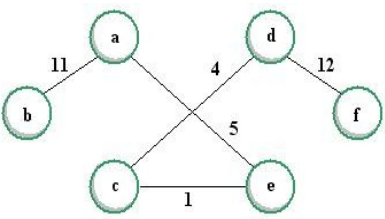
| Arestas | MST | U | V-U | Comentário |
|--|--|------------------|------|--|
| (d, e) – 8 rejeitado (a, d) – 10 (a, b) – 11 (d, f) – 12 (b, c) – 13 (e, f) – 20 | (c, e) – 1 (c, d) – 4 (a, e) – 5 | a, c, d, e | b, f | d e e estão em U  |
| (a, d) – 10 rejeitado (a, b) – 11 (d, f) – 12 (b, c) – 13 (e, f) – 20 | (c, e) – 1 (c, d) – 4 (a, e) – 5 | a, c, d, e | b, f | a e d estão em U  |
| (a, b) – 11 aceito (d, f) – 12 (b, c) – 13 (e, f) – 20 | (c, e) – 1 (c, d) – 4 (a, e) – 5 (a, b) – 11 | a, b, c, d, e | f | b não em U  |
| (d, f) – 12 aceito (b, c) – 13 (e, f) – 20 | (c, e) – 1 (c, d) – 4 (a, e) – 5 (a, b) – 11 (d, f) – 12 | a, b, c, d, e, f | | f não em U  |
| (b, c) – 13 (e, f) – 20 | | | | Todos os vértices estão agora em U COST = 33 |

Figura 19: Resultado da aplicação do algoritmo Kruskal sobre o grafo anterior

Desde que todos os vértices estejam prontos em **U**, a MST deve ser obtida. O algoritmo resultante para a MST possui o custo 33.

Nesse algoritmo, o maior fator em custo computacional é a ordenação de arestas em ordem crescente.

6. Problemas de Menor Caminho para Grafos Direcionados

Outro tipo clássico de problemas em grafos é achar o menor caminho dado um grafo ponderado. Para achar o menor caminho, é necessário obter o **comprimento** que, neste caso, é a soma dos custos não negativos de cada aresta do caminho.

Existem dois tipos de problemas com grafos ponderados:

- **Problema de Menor Caminho de Início Único (Single Source Shortest Paths (SSSP))** que determina o custo do menor caminho de um vértice inicial **u** para um vértice final **v**, onde **u** e **v** são elementos de **V**.
- **Problema de Menor Caminho para Todos os Pares (All-Pairs Shortest Paths (APSP))** que determina o custo do menor caminho de cada vértice para todos os vértices de **V**.

Vamos discutir o algoritmo criado por **Dijkstra** para resolver o Problema SSSP, e, para o Problema APSP, vamos usar o algoritmo desenvolvido por **Floyd**.

6.1. Algoritmo de Dijkstra para o Problema SSSP

Assim como os algoritmos de Prim e Kruskal, o algoritmo de Dijkstra usa o caminho "ganancioso". Neste algoritmo, para cada vértice é atribuída uma **classe** e um **valor**, onde:

- Um vértice de **Classe 1** é um vértice o qual a sua menor distância para o vértice inicial, digamos **k**, já foi encontrada; seu **valor** é igual a sua distância do vértice **k** pelo menor caminho.
- Um vértice de **Classe 2** é um vértice o qual a sua menor distância de **k** ainda precisa ser encontrada; seu **valor** é a sua distância do vértice **k** encontrada até agora.

Seja **u** o vértice inicial e **v** o vértice final. Seja **pivô** o vértice que foi mais recentemente considerado parte do caminho. Seja **caminho** de um vértice seu início direto no menor caminho. Agora o algoritmo:

1. Coloque o vértice **u** na *classe 1* e todos os outros vértices na *classe 2*
2. Defina o valor de vértice **u** para zero e o valor de todos os outros vértices para infinito
3. Faça o seguinte até o vértice **v** seja colocado na *classe 1*:
 - a. Defina o vértice **pivô** como o vértice colocado mais recentemente na *classe 1*
 - b. Ajuste todos os *nodes* da *classe 2* do seguinte modo:
 - i. Se um vértice não está conectado ao vértice *pivô*, seu valor permanece o mesmo
 - ii. Se um vértice está conectado ao vértice *pivô*, substitua seu *valor* pelo mínimo entre seu valor atual e o *valor* do vértice *pivô* mais a distância do *pivô* até o vértice na *classe 2*. Defina seu **caminho** como *pivô*
 - c. Escolha um vértice de *classe 2* com valor mínimo e coloque-o na *classe 1*

Por exemplo, dado o seguinte grafo ponderado e direcionado, encontre o menor caminho do vértice 1 ao vértice 7.

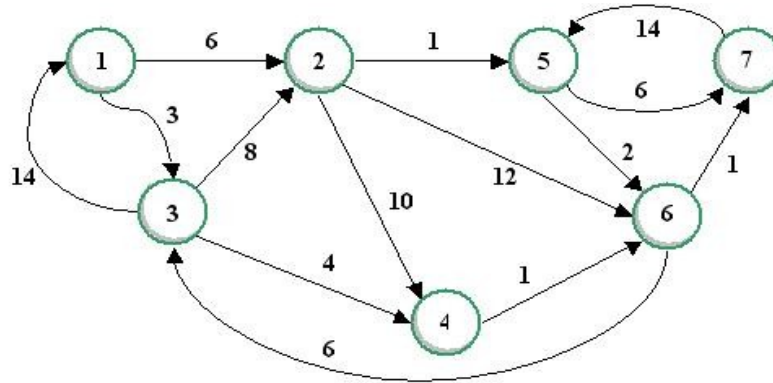


Figura 20: Um gráfico ponderado e direcionado para o exemplo SSSP

A tabela seguinte mostra a execução do algoritmo:

| | Vértice | classe | valor | caminho | Descrição |
|------|----------|--------|----------|---------|---|
| | 1 | 1 | 0 | 0 | O vértice inicial é 1. Sua classe é definida como 1. Todas as outras são definidas como 2. O valor do vértice 1 é 0 enquanto os demais forem ∞ . Os caminhos de todos os vértices são definidos como 0 uma vez que os caminhos do vértice inicial para o vértice ainda não foram encontrados. |
| | 2 | 2 | ∞ | 0 | |
| | 3 | 2 | ∞ | 0 | |
| | 4 | 2 | ∞ | 0 | |
| | 5 | 2 | ∞ | 0 | |
| | 6 | 2 | ∞ | 0 | |
| | 7 | 2 | ∞ | 0 | |
| Pivô | 1 | 1 | 0 | 0 | O vértice inicial é o primeiro pivô. Ele está conectado aos vértices 2 e 3. Portanto, os valores dos vértices estão definidos como 4 e 3 respectivamente. O vértice de classe 2 de menor valor é 3, então ele é escolhido como o próximo pivô. |
| | 2 | 2 | 4 | 0 | |
| | 3 | 2 | 3 | 0 | |
| | 4 | 2 | ∞ | 0 | |
| | 5 | 2 | ∞ | 0 | |
| | 6 | 2 | ∞ | 0 | |
| | 7 | 2 | ∞ | 0 | |
| Pivô | 1 | 1 | 0 | 0 | A classe do vértice 3 é definida como 1. Ele é adjacente aos vértices 1, 2 e 4, mas o valor de 1 e 2 é menor que o valor se o caminho que inclui o vértice 3 é considerado, então não haverá mudança em seus valores. Para o vértice 4, o valor é definido como (valor de 3) + custo(3, 4) = 3 + 4 = 7. Caminho(4) é definido como 3. |
| | 2 | 2 | 4 | 1 | |
| | 3 | 1 | 3 | 1 | |
| | 4 | 2 | 7 | 3 | |
| | 5 | 2 | ∞ | 0 | |
| | 6 | 2 | ∞ | 0 | |
| | 7 | 2 | ∞ | 0 | |
| Pivô | 1 | 1 | 0 | 0 | O próximo pivô é 2. Ele é adjacente a 4, 5 e 6. Adicionar o pivô ao menor caminho atual para 4 aumentará seu custo, mas o mesmo não ocorre com 5 e 6, onde os valores são mudados para 5 e 16 respectivamente. |
| | 2 | 1 | 4 | 1 | |
| | 3 | 1 | 3 | 1 | |
| | 4 | 2 | 7 | 3 | |
| | 5 | 2 | 5 | 2 | |
| | 6 | 2 | 16 | 2 | |
| | 7 | 2 | ∞ | 0 | |

| | Vértice | classe | valor | caminho | Descrição |
|------|----------|--------|----------|---------|--|
| Pivô | 1 | 1 | 0 | 0 | O próximo pivô é 5. Ele está conectado aos vértices 6 e 7. Adicionar o vértice 5 ao caminho mudará o valor do vértice 6 para 7 e do vértice 7 para 11. |
| | 2 | 1 | 4 | 1 | |
| | 3 | 1 | 3 | 1 | |
| | 4 | 2 | 7 | 3 | |
| | 5 | 1 | 5 | 2 | |
| | 6 | 2 | 7 | 5 | |
| | 7 | 2 | 11 | 5 | |
| Pivô | 1 | 1 | 0 | 0 | O próximo pivô é 4. Embora ele seja adjacente ao vértice 6, o valor de 6 não mudará se o pivô for adicionado ao seu caminho. |
| | 2 | 1 | 4 | 1 | |
| | 3 | 1 | 3 | 1 | |
| | 4 | 1 | 7 | 3 | |
| | 5 | 1 | 5 | 2 | |
| | 6 | 2 | 7 | 5 | |
| | 7 | 2 | 11 | 5 | |
| Pivô | 1 | 1 | 0 | 0 | Tornar 6 o pivô impõe mudar no valor do vértice 7 de 11 para 8 e também adicionar o vértice 6 ao caminho do anterior. |
| | 2 | 1 | 4 | 1 | |
| | 3 | 1 | 3 | 1 | |
| | 4 | 1 | 7 | 3 | |
| | 5 | 1 | 5 | 2 | |
| | 6 | 1 | 7 | 5 | |
| | 7 | 2 | 8 | 6 | |
| Pivô | 1 | 1 | 0 | 0 | Agora, o vértice final v é colocado na classe 1. O algoritmo termina. |
| | 2 | 1 | 4 | 1 | |
| | 3 | 1 | 3 | 1 | |
| | 4 | 1 | 7 | 3 | |
| | 5 | 1 | 5 | 2 | |
| | 6 | 1 | 7 | 5 | |
| | 7 | 1 | 8 | 6 | |

O caminho do vértice inicial 1 até o vértice final 7 pode ser obtido recuperando o valor do caminho(7) na ordem inversa, que é,

caminho(7) = 6
 caminho(6) = 5
 caminho(5) = 2
 caminho(2) = 1

Portanto, o menor caminho é **1 --> 2 --> 5 --> 6 --> 7**, e o custo é valor(7) = **8**.

6.2. Algoritmo Floyd para o problema APSP

Para encontrar o menor caminho para todos os pares, algoritmo **Dijkstra** pode ser usado com todos os pares de origens e destinos. Contudo, essa não é a melhor solução existente para o problema APSP. Uma solução mais elegante e apropriada é usar o algoritmo criado por **Floyd**.

O algoritmo faz uso da *representação de matriz de adjacência de custo* de um grafo. Ela tem uma dimensão de **n x n** para **n** vértices. Nesse algoritmo, o **COST** é dado pela matriz de adjacência. **A** é a matriz que contém o custo do caminho mais curto, inicialmente igual ao COST. Outra matriz **n x n**, **PATH**, contém os vértices eminentes ao lado do caminho mais curto:

$PATH(i,j) = 0$ inicialmente, indica que o caminho mais curto entre **i** e **j**. É a aresta (i,j) se a

mesma existir

= k se incluir **k** no caminho de **i** a **j** pela **k**-ésima interação, produz um caminho de maior custo

O algoritmo é como se segue:

1. Inicialize **A** para ser igual ao COST:

$$A(i, j) = \text{COST}(i, j), 1 \leq i, j \leq n$$

2. Se o custo de passagem atravessar o vértice intermediário **k** do vértice **i** ao vértice **j** custar menos que o acesso direto de **i** to **j**, substitua $A(i, j)$ com esse custo e atualize $\text{PATH}(i, j)$, ou seja:

Para $k = 1, 2, 3, \dots, n$

a) $A(i, j) = \text{mínimo} [A_{k-1}(i, j), A_{k-1}(i, k) + A_{k-1}(k, j)]$, $1 \leq i, j \leq n$

b) Se $(A(i, j) == A_{k-1}(i, k) + A_{k-1}(k, j))$ atribua $\text{PATH}(i, j) = k$

Por exemplo, resolva o problema APSP do grafo a seguir usando o algoritmo de **Floyd**:

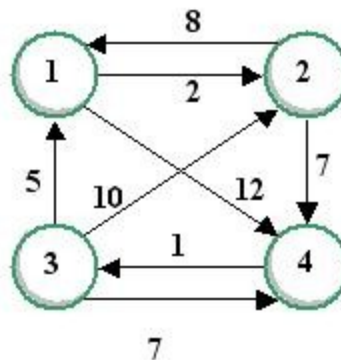


Figura 21: Um Grafo Ponderado e Direcionado para o exemplo APSP

Como auto-referência não é permitida, não é necessário computar $A(j, j)$, para $1 \leq j \leq n$. Também, para a k -ésima interação, não haverá mudanças para as k -ésimas linhas e colunas em **A** e no **PATH**, uma vez que só somará 0 ao valor atual. Por exemplo, se $k = 2$:

$$A(2, 1) = \text{mínimo}(A(2, 1), A(2, 2) + A(2, 1))$$

Como $A(2, 2) = 0$, nunca haverá mudança na k -ésima linha e coluna.

A seguir é mostrada a execução do algoritmo de **Floyd**:

| | 1 | 2 | 3 | 4 |
|---|----------|----------|----------|----|
| 1 | 0 | 2 | ∞ | 12 |
| 2 | 8 | 0 | ∞ | 7 |
| 3 | 5 | 10 | 0 | 7 |
| 4 | ∞ | ∞ | 1 | 0 |

A

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 |

PATH

Para a primeira interação $k=1$:

$$\begin{aligned}
A(2, 3) &= \text{mínimo}(A(2, 3) , A(2, 1) + A(1, 3)) = \text{mínimo}(\infty, \infty) = \infty \\
A(2, 4) &= \text{mínimo}(A(2, 4) , A(2, 1) + A(1, 4)) = \text{mínimo}(12, 20) = 12 \\
A(3, 2) &= \text{mínimo}(A(3, 2) , A(3, 1) + A(1, 2)) = \text{mínimo}(10, 7) = \mathbf{7} \\
A(3, 4) &= \text{mínimo}(A(3, 4) , A(3, 1) + A(1, 4)) = \text{mínimo}(7, 17) = 7 \\
A(4, 2) &= \text{mínimo}(A(4, 2) , A(4, 1) + A(1, 2)) = \text{mínimo}(\infty, \infty) = \infty \\
A(4, 3) &= \text{mínimo}(A(4, 3) , A(4, 1) + A(1, 3)) = \text{mínimo}(1, \infty) = 1
\end{aligned}$$

k = 1

| | 1 | 2 | 3 | 4 |
|---|----------|----------|----------|----|
| 1 | 0 | 2 | ∞ | 12 |
| 2 | 8 | 0 | ∞ | 7 |
| 3 | 5 | 7 | 0 | 7 |
| 4 | ∞ | ∞ | 1 | 0 |

A

| | 1 | 2 | 3 | 4 |
|---|---|----------|---|---|
| 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 |

PATH

Para k=2:

$$\begin{aligned}
A(1, 3) &= \text{mínimo}(A(1, 3) , A(1, 2) + A(2, 3)) = \text{mínimo}(\infty, \infty) = \infty \\
A(1, 4) &= \text{mínimo}(A(1, 4) , A(1, 2) + A(2, 4)) = \text{mínimo}(12, 9) = \mathbf{9} \\
A(3, 1) &= \text{mínimo}(A(3, 1) , A(3, 2) + A(2, 1)) = \text{mínimo}(5, 15) = 5 \\
A(3, 4) &= \text{mínimo}(A(3, 4) , A(3, 2) + A(2, 4)) = \text{mínimo}(7, 12) = 7 \\
A(4, 1) &= \text{mínimo}(A(4, 1) , A(4, 2) + A(2, 1)) = \text{mínimo}(\infty, \infty) = \infty \\
A(4, 3) &= \text{mínimo}(A(4, 3) , A(4, 2) + A(2, 3)) = \text{mínimo}(1, \infty) = 1
\end{aligned}$$

k = 2

| | 1 | 2 | 3 | 4 |
|---|----------|----------|----------|----------|
| 1 | 0 | 2 | ∞ | 9 |
| 2 | 8 | 0 | ∞ | 7 |
| 3 | 5 | 7 | 0 | 7 |
| 4 | ∞ | ∞ | 1 | 0 |

A

| | 1 | 2 | 3 | 4 |
|---|---|---|---|----------|
| 1 | 0 | 0 | 0 | 2 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 |

PATH

Para k=3:

$$\begin{aligned}
A(1, 2) &= \text{mínimo}(A(1, 2) , A(1, 3) + A(3, 2)) = \text{mínimo}(2, \infty) = 2 \\
A(1, 4) &= \text{mínimo}(A(1, 4) , A(1, 3) + A(3, 4)) = \text{mínimo}(9, \infty) = 9 \\
A(2, 1) &= \text{mínimo}(A(2, 1) , A(2, 3) + A(3, 1)) = \text{mínimo}(8, \infty) = 8 \\
A(2, 4) &= \text{mínimo}(A(2, 4) , A(2, 3) + A(3, 4)) = \text{mínimo}(7, \infty) = 7 \\
A(4, 1) &= \text{mínimo}(A(4, 1) , A(4, 3) + A(3, 1)) = \text{mínimo}(\infty, 6) = \mathbf{6} \\
A(4, 2) &= \text{mínimo}(A(4, 2) , A(4, 3) + A(3, 2)) = \text{mínimo}(\infty, 8) = \mathbf{8}
\end{aligned}$$

k = 3

| | 1 | 2 | 3 | 4 |
|---|----------|----------|----------|---|
| 1 | 0 | 2 | ∞ | 9 |
| 2 | 8 | 0 | ∞ | 7 |
| 3 | 5 | 7 | 0 | 7 |
| 4 | 6 | 8 | 1 | 0 |

A

| | 1 | 2 | 3 | 4 |
|---|----------|----------|---|---|
| 1 | 0 | 0 | 0 | 2 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 |
| 4 | 3 | 3 | 0 | 0 |

PATH

Para k=3:

$$A(1, 2) = \text{mínimo}(A(1, 2) , A(1, 4) + A(4, 2)) = \text{mínimo}(2, 17) = 2$$

$$A(1, 3) = \text{mínimo} (A(1, 3) , A(1, 4) + A(4, 3)) = \text{mínimo} (\infty, 10) = \mathbf{10}$$

$$A(2, 1) = \text{mínimo} (A(2, 1) , A(2, 4) + A(4, 1)) = \text{mínimo} (8, 13) = 8$$

$$A(2, 3) = \text{mínimo} (A(2, 3) , A(2, 4) + A(4, 3)) = \text{mínimo} (\infty, 8) = \mathbf{8}$$

$$A(3, 1) = \text{mínimo} (A(3, 1) , A(3, 4) + A(4, 1)) = \text{mínimo} (5, 13) = 5$$

$$A(3, 2) = \text{mínimo} (A(3, 2) , A(3, 4) + A(4, 2)) = \text{mínimo} (7, 15) = 7$$

$k = 4$

| | 1 | 2 | 3 | 4 |
|---|---|---|-----------|---|
| 1 | 0 | 2 | 10 | 9 |
| 2 | 8 | 0 | 8 | 7 |
| 3 | 5 | 7 | 0 | 7 |
| 4 | 6 | 8 | 1 | 0 |

A

| | 1 | 2 | 3 | 4 |
|---|---|---|----------|---|
| 1 | 0 | 0 | 4 | 2 |
| 2 | 0 | 0 | 4 | 0 |
| 3 | 0 | 1 | 0 | 0 |
| 4 | 3 | 3 | 0 | 0 |

PATH

Após a n^{th} interação, **A** contém o menor custo enquanto **PATH** contém o caminho de menor custo. Para ilustrar como usar o resultado das matrizes, vamos encontrar o caminho mais curto do vértice 1 para o vértice 4:

$$A(1, 4) = 9$$

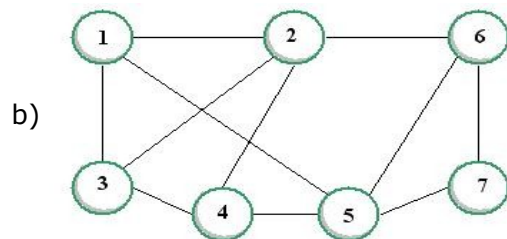
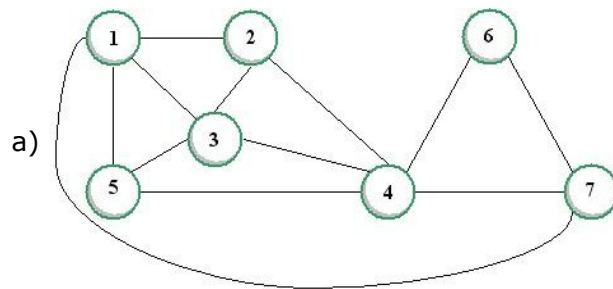
$$\text{PATH}(1, 4) = 2 \rightarrow \text{Desde que não seja 0, temos que pegar } \text{PATH}(2, 4):$$

$$\text{PATH}(2, 4) = 0$$

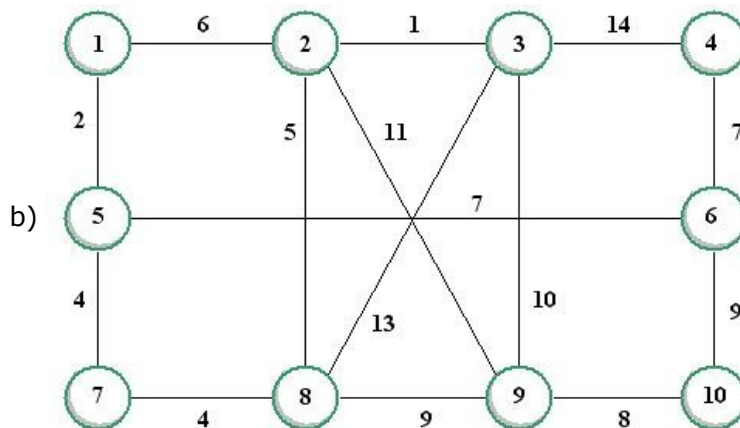
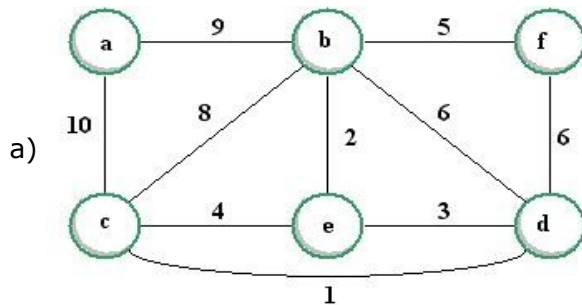
Por essa razão, o caminho mais curto do vértice 1 para o vértice 4 é **1 --> 2 --> 4** com custo 9. Até mesmo se existe uma aresta direta de 1 ao 4 (com custo 12), o algoritmo retornou outro caminho. Esse exemplo mostra que ele não é sempre a conexão direta que é retornada ao obter o caminho mais curto em um ponderado, grafo direcionado.

7. Exercícios

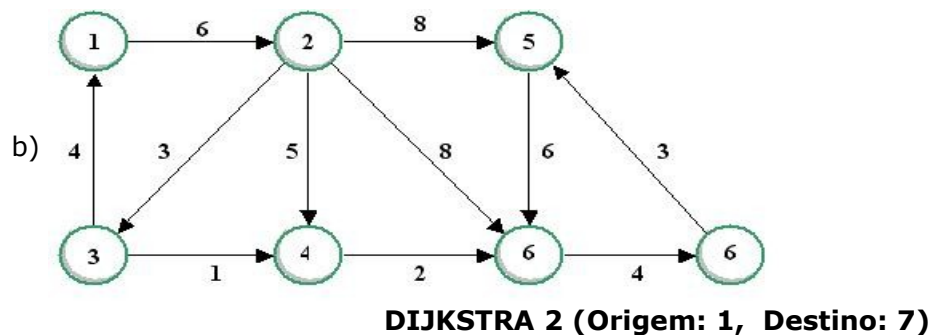
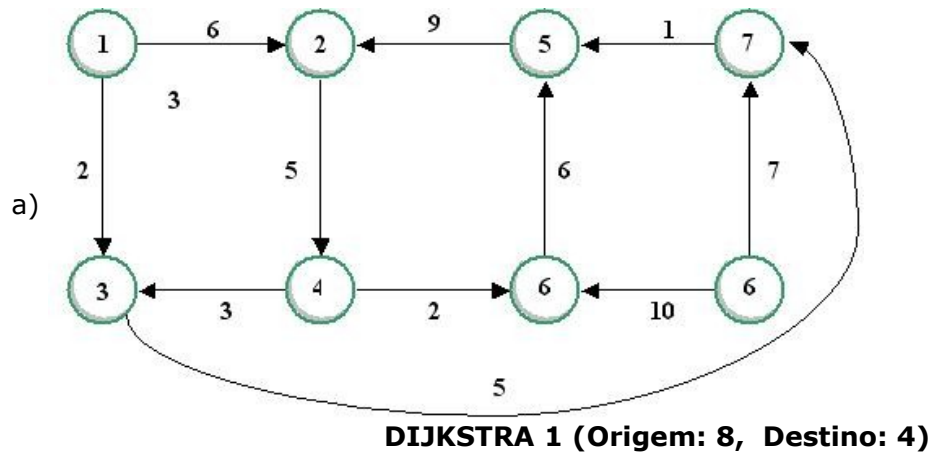
1. O que é DFS e BFS listando elementos dos seguintes grafos com 1 como vértice de partida?



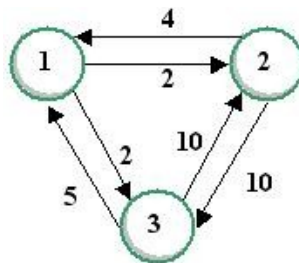
2. Encontre a árvore geradora de custo mínimo dos seguintes grafos usando os algoritmos de **Kruskal** e **Prim**. Dê o custo do MST.



3. Resolva o problema SSSP do seguinte grafo, usando o algoritmo **Dijkstra**. Mostre o valor, a classe e o caminho dos vértices para cada interação:



4. Resolva o APSP dos grafos a seguir dando as matrizes **A** e **Path** usando o algoritmo **Floyd**:



7.1. Exercícios para Programar

1. Crie uma definição de classe Java para grafos direcionados ponderados usando representação de matriz de adjacência.
2. Implemente os dois algoritmos de grafo transversal.
3. Caminho mais curto usando o algoritmo **Dijkstra**.

Implemente um caminho mais curto dado um mapa e o custo de cada aresta. O programa pedirá um arquivo de entrada contendo as origens ou destinos (vértices) e as conexões entre as localizações (aresta) com o custo. As localizações seguem a forma **(número, localização)**, onde **número** é um inteiro determinado para o vértice e **localização** é o nome do vértice. Todo par (número, localização) tem que ser localizado em uma linha separada no arquivo de entrada. Para terminar a entrada, use **(0, 0)**. Conexões serão restritas também de um mesmo arquivo. Uma definição de tem que se da forma **(i, j, k)**, uma linha por aresta, onde **i** é o número determinado para a origem, **j** o

número determinado para o destino e **k** o custo da chegada de **j** a **i** (Lembre-se, usamos grafos direcionados aqui). Conclua a entrada usando **(0, 0, 0)**.

Após a entrada, crie um mapa e mostre-o para usuário. Solicite ao usuário informar a fonte e o destino e dê o caminho mais curto usando o algoritmo **Dijkstra**.

Mostre a saída na tela. A saída consiste no caminho e seu custo. O caminho deve seguir a seguinte forma:

origem → localização 2 → ... → localização n-1 → destino

Amostragem do arquivo de entrada

```
(1, Math Building)
(2, Science Building)
(3, Engineering)
.
.
(0, 0)
(1, 2, 10)
(1, 3, 5)
(3, 2, 2)
.
.
(0, 0, 0)
```

← início da definição de aresta

Amostragem do fluxo do programa

```
[ MAP and LEGEND ]
Origem de entrada: 1
Destino de entrada: 2
Origem: Math Building
Destino: Science Building
Caminho: Math Building → Engineering → Science Building
Custo: 7
```

Parceiros que tornaram JEDI™ possível



Instituto CTS

Patrocinador do DFJUG.

Sun Microsystems

Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas

Criador da Iniciativa JEDI™.

DFJUG

Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil

Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec

Suporte e apoio financeiro e logístico a todo o processo.

Borland

Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB

Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.