7 Caminho mínimo

Um tópico bastante importante da Teoria dos Grafos trata de problemas que envolvem algum tipo de caminhamento na estrutura de um grafo. Nós já abordamos alguns conceitos e aplicações de caminhamento no Capítulo 3. Agora vamos explorar em mais detalhes problemas envolvendo caminhamento em grafos, tais como caminhos mínimos em grafos valorados, bem como os famosos problemas do carteiro chinês e do caixeiro viajante.

7.1 Problema do caminho mínimo

O problema do caminho mínimo em grafos valorados é um dos problemas mais conhecidos e com enorme gama de aplicações. O problema consiste em encontrar um caminho com custo total mínimo entre um vértice inicial e um vértice final. Este tipo de resultado pode ser obtido tanto em grafos dirigidos quanto em grafos não dirigidos, e os algoritmos para isso podem ser aplicados em ambos os tipos de grafos. Antes de ver os algoritmos, precisamos de algumas definições:

Definição 7.1 (Custo de um caminho)**.** *O custo de um caminho $p = \langle v_0, v_1, \ldots, v_k \rangle$ entre os vértices v_0 e v_k, denotado por $w(p)$, é igual ao somatório dos custos de todas as arestas valoradas do caminho, ou seja: $w(p) = \sum_{i=1}^{k} w_{i-1,i}$.*

Definição 7.2 (Custo do caminho mínimo)**.** *O custo do um caminho mínimo do vértice v_i para o vértice v_j é definido por:*

$$\delta_{i,j} = \begin{cases} \min\{w(p) : v_i \rightsquigarrow v_j\} & \text{se } \exists \text{ caminho de } v_i \text{ para } v_j, \\ \infty & \text{caso contrário.} \end{cases}$$

Definição 7.3 (Caminho mínimo)**.** *O caminho mínimo entre dois vértices v_i e v_j é definido como qualquer caminho p com custo igual a $\delta_{i,j}$.*

7.1.1 Algoritmo de Dijkstra

O problema do caminho mínimo é um dos mais conhecidos da Teoria dos Grafos, com inúmeras aplicações práticas, como, por exemplo, encontrar a menor rota entre duas localizações em um mapa viário. A seguir, temos uma definição formal do problema do caminho mínimo.

Definição 7.4 (Problema do caminho mínimo)**.** *Sejam v_i e v_j dois vértices de um grafo valorado conexo. Encontre $\delta_{i,j}$.*

O algoritmo Dijkstra (1959), apresenta uma solução para o problema do caminho mínimo, encontrando todos os caminhos mínimos de um determinado vértice inicial arbitrário para todos os demais vértices do grafo. Com isso, o algoritmo produz uma árvore de caminhos mínimos, tendo o vértice inicial como sua raiz. Assumimos que todas as arestas possuem custos não negativos, pois não há garantia de que o algoritmo encontre uma solução correta no caso de arestas negativas. Ou seja, $w_{i,j} \geq 0$ para cada aresta $(v_i, v_j) \in E$. Note também que, se todas as arestas do grafo tiverem o mesmo custo, o problema se reduz à uma busca em largura (veja a seção 3.2). Basicamente, o algoritmo funciona da seguinte maneira:

- inicia-se a partir de um vértice de sua escolha. A partir disso, o algoritmo analisa o grafo para encontrar os caminhos mínimos deste vértice para os demais;
- o algoritmo mantém os custos atualmente conhecidos do vértice inicial até todos os demais vértices. À medida que o algoritmo vai executando, estes custos são atualizados;
- em cada iteração, o algoritmo conclui o cálculo final do custo do vértice inicial (origem) para um determinado vértice de destino. Neste momento, o vértice destino precisa de alguma forma ser sinalizado como "visitado" ou "encerrado" e deve estar devidamente posicionado na árvore de caminhos mínimos;
- o processo continua até que todos os vértices do grafo possíveis de serem visitados[1] sejam incluídos na árvore de caminhos mínimos.

Para implementação do algoritmo de caminho mínimo a partir de um vértice origem, um dos pseudocódigos mais populares é apresentado por Cormen et al. (2001). A árvore de caminhos mínimos é construída passo a passo e é estruturada por meio de referências ao vértice predecessor (nó pai) de cada vértice da árvore. O algoritmo 7.1 tem os seguintes elementos:

- s: representa o vértice inicial, origem dos caminhos mínimos;
- d_v: custo do caminho mínimo do vértice s (vértice inicial) até o vértice v. Este custo é inicializado com valor infinito e vai sendo atualizado à medida que o algoritmo analisa o grafo e calcula caminhos mínimos;
- ρ_v: referência ao vértice predecessor do vértice v na árvore de caminhos mínimos;
- S: é o conjunto de todos os vértices cujo caminho mínimo já foi calculado pelo algoritmo;
- Q: é uma fila de prioridades (mínima) de vértices, tendo como chave o valor de d_v. A operação REMOVE-MINIMO(Q) retira da fila e retorna o vértice com menor valor atual de d_v;

[1] Se o grafo conexo for dirigido, é possível que haja vértices que não possam ser alcançados a partir do vértice inicial escolhido.

- $adj(u)$: é uma iteração com todos os vértices adjacentes ao vértice u (veja seção 2.2);

Algoritmo: Dijkstra(G, w, s)

para cada $v \in V$ **faça**
    $d_v \leftarrow \infty$;
    $\rho_v \leftarrow nil$;

$d_s \leftarrow 0$;
$S \leftarrow \emptyset$;
$Q \leftarrow V$;

enquanto $Q \neq \emptyset$ **faça**
    $u \leftarrow REMOVE - MINIMO(Q)$;
    $S \leftarrow S \cup \{u\}$;
    para cada *vértice* $v \in adj(u)$ **faça**
        se $v \in Q$ e $[d_v > (d_u + w_{u,v})]$ **então**
            $d_v \leftarrow (d_u + w_{u,v})$;
            $\rho_v \leftarrow u$;

Algoritmo 7.1: Algoritmo de Dijkstra

Uma operação importante neste pseudocódigo é o relaxamento de aresta. Quando um determinado vértice u é considerado "encerrado", todos os seus vértices adjacentes v são verificados para fazer o relaxamento de aresta. Isto significa que, ao se calcular o caminho mínimo até u, verifica-se a possibilidade de haver para o vértice v um caminho passando por u que seja mais barato do que o caminho atual para v. Neste caso o custo do caminho para v é atualizado, bem como seu predecessor na árvore de caminhos mínimos, que passa a ser o vértice u.

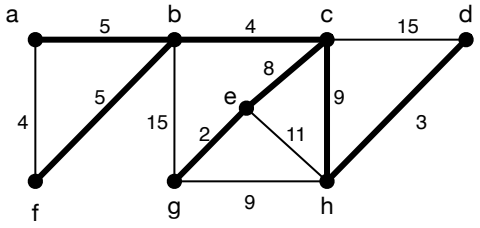
Esta operação ocorre no comando condicional dentro do laço principal do algoritmo: a atualização de custo é realizada pela expressão $d_v \leftarrow (d_u + w_{u,v})$ e a atualização de caminho é feita pela expressão $\rho_v \leftarrow u$.

Exemplo 7.1. *A Figura 7.1 mostra um exemplo de árvore de custo mínio gerada com o algoritmo de Dijkstra, tenho do vértice b como vértice inicial. As arestas que compõem a árvore são aquelas mostradas com linhas mais grossas. A tabela apresenta os valores de* d_v *e* ρ_v *calculados pelo algoritmo.*

7.1.2 Algoritmo de Floyd

O algoritmo de Floyd utiliza um procedimento incremental sobre a matriz de custos do grafo para encontrar os caminhos mínimos entre todos os pares de vértices. O grafo pode ser dirigido ou não dirigido e pode, inclusive ter arestas com custos negativos, porém não pode ter ciclos com custo total negativo. Este algoritmo foi publicado na sua forma atual

Figura 7.1 – Árvore de caminhos mínimos produzida pelo algoritmo de Dijkstra



vértice	a	b	c	d	e	f	g	h
d_v	5	0	4	16	12	5	14	13
ρ_v	b	nil	b	h	c	b	e	c

Fonte: o autor.

por Robert Floyd (FLOYD, 1962), mas essencialmente é o mesmo algoritmo publicado por Bernard Roy (ROY, 1959) e por Stephen Warshall (WARSHALL, 1962). Por este motivo, o algoritmo também é conhecido como Floyd-Warshall, Roy-Warshall ou Roy-Floyd.

Seja $G = (V, E, W)$ um grafo valorado com n vértices, inicia-se o cálculo dos caminhos mínimos a partir da matriz de custos do grafo. Em seguida, são calculadas n matrizes de distância D de dimensões $n \times n$. Um elemento $d_{i,j}$ em uma matriz de distâncias contém o custo (atual) do caminho mínimo com origem em v_i e destino em v_j. O valor final dos caminhos mínimos vai estar na última das n matrizes geradas.

Para descrever o algoritmo, os índices i e j, representam respectivamente linha e coluna das matrizes, correspondendo aos vértices de origem e destino, com $i = 1, 2, \ldots, n$ e $j = 1, 2, \ldots, n$. O índice k representando a iteração do algoritmo, com $k = 0, 1, 2, \ldots, n$.

As matrizes de distância são definidas da seguinte forma, na qual os elementos da *k-ésima* matriz são calculados a partir dos elementos da matriz anterior:

$$d_{i,j}^{k} = \begin{cases} w_{i,j} & \text{se } k = 0, \\ \min(d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1}) & \text{se } k \geq 1. \end{cases}$$

Note que a expressão $\min(d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1})$ promove o mesmo efeito de cálculo de relaxamento de aresta que é feito no algoritmo de Dijkstra.

Sempre que o custo do caminho mínimo entre um determinado par de vértices é atualizado, também é necessário atualizar os dados que mantém os caminhos (ou rotas). Tanto nos algoritmos de busca quanto no algoritmo de Dijkstra, os caminhos formam árvores tendo o vértice inicial na raiz. Foi visto que tais árvores podem ser facilmente armazenadas mantendo-se referências a vértices predecessores em vetores de roteamento.

Como o algoritmo de Floyd gera caminhos tomando cada um dos vértices do grafo como origem, são necessários n vetores de roteamento. Uma forma conveniente de manter estes dados é por meio de Matrizes de Roteamento, denotadas por R. Um elemento $\rho_{i,j}$ de uma matriz de roteamento armazena uma referência para o predecessor do vértice v_j num caminho iniciado no vértice v_i. Em outras palavras, cada linha i da matriz de roteamento

R é um vetor de roteamento armazenando os caminhos com origem no vértice v_i.

O algoritmo de Floyd constrói k matrizes de roteamento. A matriz inicial (para $k = 0$) é dada da seguinte forma:

$$\rho^0_{i,j} = \begin{cases} \text{nil} & \text{se } i = j \text{ ou } w_{i,j} = \infty, \\ v_i & \text{se } i \neq j \text{ e } w_{i,j} < \infty. \end{cases}$$

Para $k \geq 1$, havendo uma atualização de caminho mínimo passando pelo vértice v_k, ou seja, $v_i \rightsquigarrow v_k \rightsquigarrow v_j$, deve haver a atualização de vértice predecessor:

$$\rho^k_{i,j} = \begin{cases} \rho^{k-1}_{i,j} & \text{se } d^{k-1}_{i,j} \leq d^{k-1}_{i,k} + d^{k-1}_{k,j}, \\ \rho^{k-1}_{k,j} & \text{se } d^{k-1}_{i,j} > d^{k-1}_{i,k} + d^{k-1}_{k,j}. \end{cases}$$

O pseudocódigo do algoritmo 7.2 incorpora as inicializações e atualizações das matrizes D e R para cálculo dos caminhos mínimos entre todos os pares de vértices.

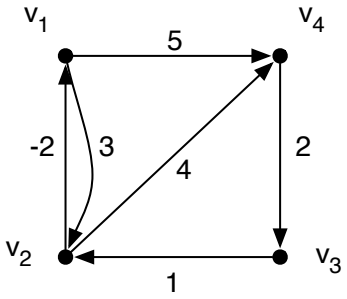
Algoritmo: Foyd(G, W)

para cada $i \leftarrow 1$ **até** n **faça**
  para cada $j \leftarrow 1$ **até** n **faça**
    $d^0_{i,j} \leftarrow w_{i,j}$;
    $\rho^0_{i,j} \leftarrow v_i$;
para cada $k \leftarrow 1$ **até** n **faça**
  para cada $i \leftarrow 1$ **até** n **faça**
    para cada $j \leftarrow 1$ **até** n **faça**
      se $(d^{k-1}_{i,k} + d^{k-1}_{k,j}) < d^{k-1}_{i,j}$ **então**
        $d^k_{i,j} \leftarrow d^{k-1}_{i,k} + d^{k-1}_{k,j}$;
        $\rho^k_{i,j} \leftarrow \rho^{k-1}_{k,j}$;
      senão
        $d^k_{i,j} \leftarrow d^{k-1}_{i,j}$;
        $\rho^k_{i,j} \leftarrow \rho^{k-1}_{i,j}$;

Algoritmo 7.2: Algoritmo de Floyd

Exemplo 7.2. *A Figura 7.2 mostra um exemplo de digrafo valorado com aresta negativa, e as respectivas matrizes de custo e de roteamento geradas pelo algoritmo de Floyd para cada uma das k iterações.*

Figura 7.2 – Digrafo valorado com as matrizes D e R



$k = 0$

$$D^0 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ -2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array} \qquad R^0 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} nil & v_1 & nil & v_1 \\ v_2 & nil & nil & v_2 \\ nil & v_3 & nil & nil \\ nil & nil & v_4 & nil \end{bmatrix} \end{array}$$

$k = 1$

$$D^1 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ -2 & 0 & \infty & 3 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array} \qquad R^1 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} nil & v_1 & nil & v_1 \\ v_2 & nil & nil & v_1 \\ nil & v_3 & nil & nil \\ nil & nil & v_4 & nil \end{bmatrix} \end{array}$$

$k = 2$

$$D^2 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ -2 & 0 & \infty & 3 \\ -1 & 1 & 0 & 4 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array} \qquad R^2 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} nil & v_1 & nil & v_1 \\ v_2 & nil & nil & v_1 \\ v_2 & v_3 & nil & v_1 \\ nil & nil & v_4 & nil \end{bmatrix} \end{array}$$

$k = 3$

$$D^3 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ -2 & 0 & \infty & 3 \\ -1 & 1 & 0 & 4 \\ 1 & 3 & 2 & 0 \end{bmatrix} \end{array} \qquad R^3 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} nil & v_1 & nil & v_1 \\ v_2 & nil & nil & v_1 \\ v_2 & v_3 & nil & v_1 \\ v_2 & v_3 & v_4 & nil \end{bmatrix} \end{array}$$

$k = 4$

$$D^4 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 3 & 7 & 5 \\ -2 & 0 & 5 & 3 \\ -1 & 1 & 0 & 4 \\ 1 & 3 & 2 & 0 \end{bmatrix} \end{array} \qquad R^4 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} nil & v_1 & v_4 & v_1 \\ v_2 & nil & v_4 & v_1 \\ v_2 & v_3 & nil & v_1 \\ v_2 & v_3 & v_4 & nil \end{bmatrix} \end{array}$$

Fonte: o autor.

Para interpretar os resultados do algoritmo de Floyd, devemos observar as últimas matrizes de distância e roteamento geradas (neste caso, D^4 e R^4). Por exemplo, na Figura 7.2, o custo do caminho mínimo de v_2 para v_3 está localizado no elemento $d^4_{2,3}$ e seu valor é igual a 5. Já o caminho mínimo entre v_2 e v_3 deve ser obtido na segunda linha da matriz R^4, o qual é $p : \langle v_2, v_1, v_4, v_3 \rangle$. Devemos começar pelo último vértice do caminho (v_3 neste caso) e a partir dele, encontrar seu predecessor em um caminho que começou em v_2, ou seja, o elemento $\rho^4_{2,3} = v_4$. Repete-se o processo sucessivamente até chegar ao vértice inicial do caminho (v_2). A seguir veremos o pseudocódigo para realizar este procedimento.

7.1.3 Imprimindo caminhos na matriz de roteamento

Os caminhos gerados pelo algoritmo de Floyd, armazenados em uma matriz de roteamento R, podem ser reconstituídos por meio do algoritmo 7.3. Note que o algoritmo é recursivo, e pode ser considerado uma generalização do algoritmo 3.5.

Algoritmo: ImprimeCaminhoMatriz(R, v_i, v_j)

se $v_i = v_j$ **então**
  imprime(v_i);
senão
  se $\rho_{i,j} = nil$ **então**
    imprime(“*não existe caminho de* v_i *para* v_j ”);
  senão
    ImprimeCaminhoMatriz($R, v_i, \rho_{i,j}$);
    imprime(v_j);

Algoritmo 7.3: Procedimento para imprimir caminhos

7.2 Problema do Carteiro Chinês

O problema do carteiro foi estudado pelo matemático chinês Kwan Mei-Ko em 1962 (KWAN, 1962) e consiste em encontrar um caminho fechado com custo mínimo em um grafo conexo valorado. O termo "carteiro"é uma analogia ao serviço de entrega de correspondência, imaginando que um carteiro precisaria entregar correspondência em todas as ruas de um bairro, voltando ao seu ponto de partida e além disso caminhando a menor distância total possível.

Se o grafo valorado for euleriano, a solução para o problema é trivial. Como um ciclo euleriano percorre todas as arestas somente uma vez, este já seria a rota com custo mínimo escolhida pelo carteiro e o seu custo total é igual à soma dos custos de todas as arestas.

Por outro lado, se o grafo não for euleriano, o requisito de passarmos por todas as ruas (isto é, todas as arestas) faz com que seja necessário definir um trajeto repetindo

algumas ruas. A questão fundamental é decidir quais ruas repetir de forma que se acrescente o menor custo adicional possível.

7.2.1 Eulerização de grafos

Para resolver a questão de repetição das "ruas" na rota do carteiro, vejamos o processo de "eulerização" de grafos não-eulerianos, descrito por Saoub (2017). Existem duas condições para um grafo não ser euleriano: não ser conexo ou não ter todos os seus vértices de grau par. O processo de "eulerização" de grafos conexos, aborda a questão dos graus dos vértices para obter um grafo euleriano a partir de um grafo não-euleriano conexo.

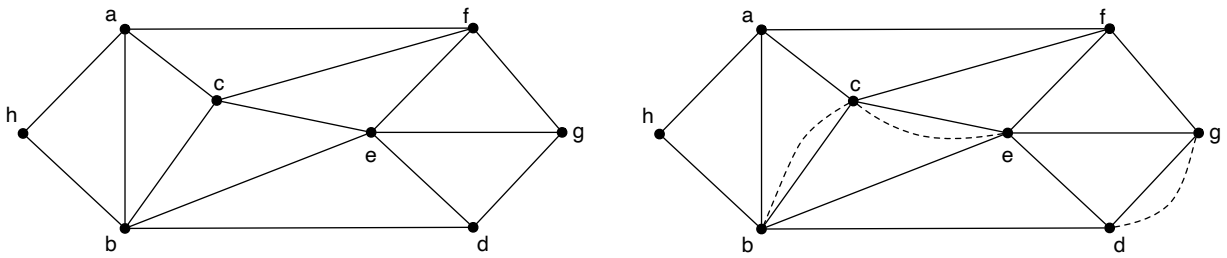
Definição 7.5 (Eulerização)**.** *Dado um grafo conexo $G = (V, E)$, uma Eulerização de G é o grafo $G^e = (V, E^e)$ tal que:*

1. *G^e é obtido pela duplicação de arestas de G; e*
2. *todos os vértices de G^e possuem grau par.*

Uma maneira de se obter a eulerização de um grafo é duplicando as arestas que ligam pares de vértices de grau ímpar. Caso reste algum par de vértices de grau ímpar não adjacentes, deve-se duplicar arestas ao longo de um caminho entre estes dois vértices. Tal caminho é chamado **caminho artificial** e as arestas duplicadas são **arestas artificiais**.

A Figura 7.3 mostra um exemplo deste processo. O grafo G da Figura 7.3a tem 4 vértices de grau ímpar. Obtemos a eulerização de G mostrada na Figura 7.3b criando uma aresta artificial entre os vértices g e d, e um caminho artificial entre os vértices e e b, representados pelas arestas tracejadas. Note que os vértices de grau par no meio de um caminho artificial (como o vértice c, por exemplo) continuam tendo grau par após a criação do caminho.

Figura 7.3 – Exemplo de eulerização



(a) Grafo G (b) Grafo G^e

Fonte: o autor.

7.2.2 Algoritmo para o problema do carteiro chinês

O pseudocódigo do algoritmo 7.4 encontra a solução do problema do carteiro chinês em grafos valorados determinando um caminho fechado com custo mínimo que passe por todas as arestas. A ideia básica é promover a eulerização do grafo construindo caminhos artificiais com custo mínimo entre pares de vértices de grau ímpar.

Algoritmo: CarteiroChines(G, W)

`// Inicialização`
$V^{par} \leftarrow$ {conjunto dos vértices de grau par};
$G^e \leftarrow G$;

`// Calcula os caminhos mínimos`
Execute $FLOYD(G, W)$ para construir a matriz de distâncias D^n;

`// Remove da matriz as linhas e colunas dos vértices de grau par`
$D^{impar} \leftarrow D^n -$ (linhas e colunas de V^{par});

`// Laço principal`
enquanto $D^{impar} \neq \emptyset$ **faça**
    Determine em D^{impar} o par de vértices v_i e v_j com menor custo $d_{i,j}^{impar}$;
    Construa um caminho artificial de v_i para v_j com custo $d_{i,j}^{impar}$ no grafo G^e;
    `// Remove da matriz as linhas e colunas de` v_i `e` v_j
    $D^{impar} \leftarrow D^{impar} -$ (linhas e colunas de v_i e v_j);

`// Encontrando o resultado final`
Encontre um ciclo euleriano do grafo G^e;

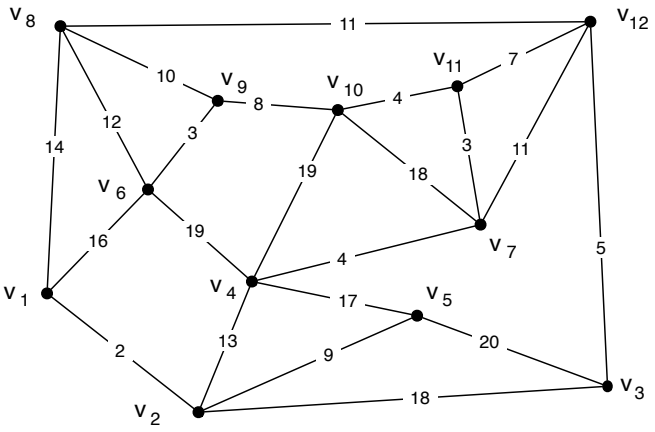
Algoritmo 7.4: Problema do carteiro chinês

Inicialmente, determinamos quais são os vértices de grau par e criamos um grafo G^e que, ao final do algoritmo, será uma eulerização de G com custo mínimo. Em seguida chamamos o algoritmo de Floyd calculamos os caminhos mínimos entre todos os pares de vértices e obtemos a matriz de distâncias (ou custos) mínimas D^n. Como só precisamos encontrar caminhos artificiais entre pares de vértices de grau ímpar, removemos as linhas e colunas de vértices de grau par da matriz D^n para construir uma nova matriz chamada D^{impar} que contém somente as linhas e colunas, e portanto, os caminhos mínimos, dos vértices de grau ímpar.

O laço principal do algoritmo encontra na matriz D^{impar} o par de vértices v_i e v_j com menor custo mínimo atual e constrói no grafo G^e o respectivo caminho artificial. As linhas e colunas correspondentes a v_i e v_j são removidas da matriz D^{impar} e processo se repete enquanto a matriz não estiver vazia. Ao final, o grafo G^e corresponde a uma eulerização do grafo G com custo mínimo e basta encontrar um ciclo euleriano em G^e para obter a solução do problema.

Exemplo 7.3. *Dado o grafo $G = (V, E)$ da Figura 7.4, observamos que os vértices v_1, v_3, v_4, v_5, v_9 e v_{11} possuem grau ímpar, enquanto que os vértices v_2, v_6, v_7, v_8, v_{10} e v_{12} possuem grau par. Após calcular os caminhos mínimos entre todos os pares de vértices e eliminar da matriz D^n as linhas e colunas referentes aos vértices de grau par, obtemos a matriz D^{impar} mostrada ao lado do grafo.*

Figura 7.4 – Primeira iteração do carteiro chinês

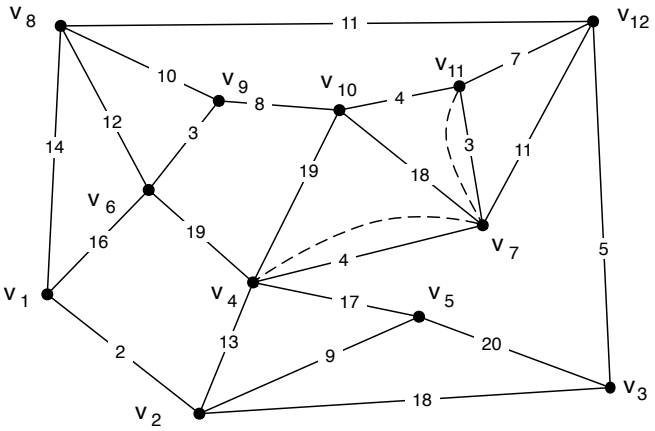


$$D^{impar} = \begin{array}{c} \\ v_1 \\ v_3 \\ v_4 \\ v_5 \\ v_9 \\ v_{11} \end{array} \begin{array}{c} \begin{array}{cccccc} v_1 & v_3 & v_4 & v_5 & v_9 & v_{11} \end{array} \\ \begin{bmatrix} 0 & 20 & 15 & 11 & 19 & 22 \\ 20 & 0 & 19 & 20 & 24 & 12 \\ 15 & 19 & 0 & 17 & 19 & \mathbf{7} \\ 11 & 20 & 17 & 0 & 30 & 24 \\ 19 & 24 & 19 & 30 & 0 & 12 \\ 22 & 12 & \mathbf{7} & 24 & 12 & 0 \end{bmatrix} \end{array}$$

Fonte: adaptado de Rabuske (1992).

O caminho mínimo entre os vértices v_4 e v_{11} é o menor dentre todos os caminhos mínimos da matriz D^{impar}. Portanto, é construído o caminho artificial entre estes dois vértices, mostrado em linhas tracejadas na Figura 7.5. As linhas e colunas referentes aos vértices v_4 e v_{11} são removidas da matriz D^{impar}.

Figura 7.5 – Segunda iteração do carteiro chinês

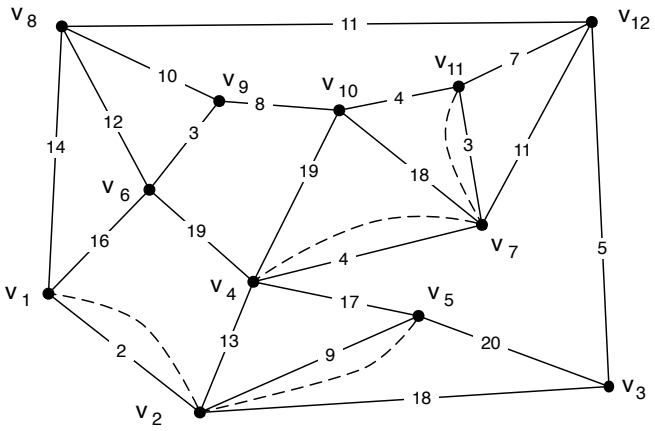


$$D^{impar} = \begin{array}{c} \\ v_1 \\ v_3 \\ v_5 \\ v_9 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_3 & v_5 & v_9 \end{array} \\ \begin{bmatrix} 0 & 20 & 11 & 19 \\ 20 & 0 & 19 & 24 \\ 11 & 20 & 0 & 30 \\ 19 & 24 & 30 & 0 \end{bmatrix} \end{array}$$

Fonte: adaptado de Rabuske (1992).

Repetindo o mesmo procedimento, cria-se um caminho artificial com custo 11 entre os vértices v_1 e v_5. A matriz D^{impar} resultante tem um único caminho mínimo (Figura 7.6).

Figura 7.6 – Terceira iteração do carteiro chinês

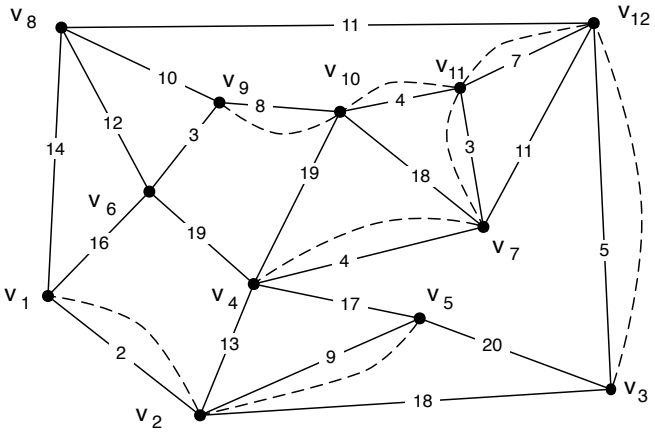


$$D^{impar} = \begin{array}{c} \\ v_3 \\ v_9 \end{array} \begin{array}{c} \begin{array}{cc} v_3 & v_9 \end{array} \\ \begin{bmatrix} 0 & 24 \\ 24 & 0 \end{bmatrix} \end{array}$$

Fonte: adaptado de Rabuske (1992).

Na figura Figura 7.7, podemos ver o resultado final do algoritmo, cujo laço principal se encerra quando a matriz D^{impar} fica vazia. Note que, considerando a existência das arestas artificiais, o grafo resultante é uma eulerização de G. Por fim, basta encontrar um ciclo euleriano incluindo as arestas artificiais.

Figura 7.7 – Resultado final do carteiro chinês



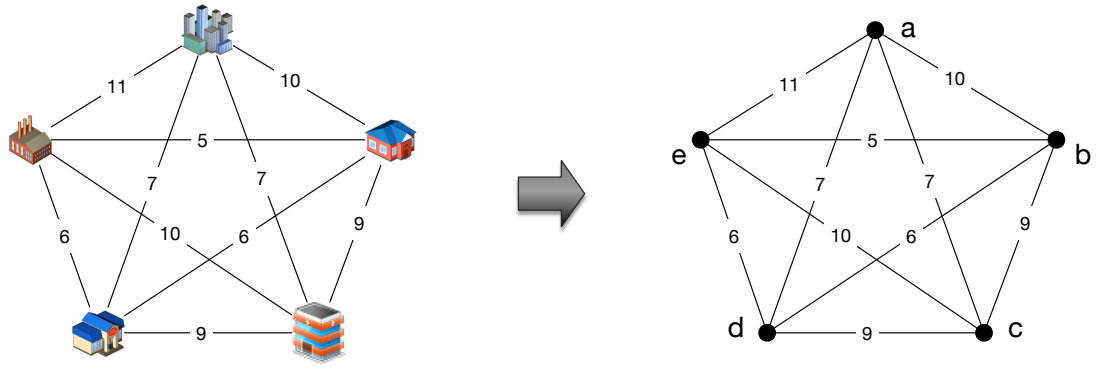
$$D^{impar} = \emptyset$$

Fonte: adaptado de Rabuske (1992).

7.3 Problema do Caixeiro Viajante

O problema do caixeiro viajante talvez seja um dos problemas mais famosos da computação e da pesquisa operacional. Seu nome se deve ao exemplo básico no qual um "caixeiro viajante"[2] precisa visitar um determinado número de cidades retornando à sua cidade de origem e gastando o mínimo possível em passagens aéreas. Em sua formulação clássica, existem voos entre todos os pares de cidades e o problema é modelado por um grafo completo valorado, com os vértices representando as cidades e os custos das arestas representando o preço das passagens aéreas entre cada par de cidades. A Figura 7.8 exemplifica a modelagem de uma instância do problema com 5 cidades.

Figura 7.8 – Modelagem do problema do caixeiro viajante



Fonte: o autor.

7.3.1 Algoritmo força bruta

Em resumo, o problema consiste em encontrar um ciclo hamiltoniano de custo mínimo em um grafo completo valorado (SAOUB, 2021). Sabemos que todos os grafos completos com 3 ou mais vértices são hamiltonianos, então poderíamos imaginar um algoritmo do tipo força bruta relativamente simples para resolver o problema: bastaria enumerar todos os ciclos hamiltonianos e escolher aquele com menor custo. Esta solução torna-se inviável porque a quantidade de ciclos hamiltonianos em um grafo completo K_n é igual a $(n-1)!/2$, levando o algoritmo a ter ordem de complexidade $O(n!)$.

O algoritmo força bruta pode ser realizado pelos seguintes passos:

Entrada: um grafo completo valorado K_n.

Passos:

1. Escolha um vértice inicial arbitrário v;

[2] "Caixeiro viajante" é um termo antigo (e já um tanto fora de moda!) para denominar um representante comercial ou vendedor ambulante.

2. Encontre todos os ciclos hamiltonianos iniciando pelo vértice v. Calcule o custo total de cada ciclo;
3. Compare todos os $(n-1)!/2$ ciclos. Escolha um com o menor custo total.

Saída: um ciclo hamiltoniano com custo mínimo.

O Quadro 6 traz a quantidade de ciclos hamiltonianos em grafos completos em função da sua quantidade de vértices , bem como os tempos aproximados de processamento para encontrar todos os ciclos considerando que cada ciclo levaria 0,2 nanossegundo para ser calculado (0,0000000002 segundo, ou 0,2 ηs). Note que para encontrar um ciclo hamiltoniano em um grafo completo, basta determinar uma permutação de seus vértices.

Quadro 6 – Quantidade de ciclos hamiltonianos em grafos completos

n	ciclos	tempo
1	0	0
5	12	$2,4\ \eta s$
10	181.440	$36,29\ \mu s$
15	43.589.145.600	$8,71 s$
20	$6,08 \times 10^{16}$	140 dias
30	$8,84 \times 10^{20}$	28 trilhões de anos

Fonte: o autor.

7.3.2 Heurística do vizinho mais próximo

Apesar de obter uma solução ótimo, algoritmo de força bruta sofre uma explosão combinatória devido à quantidade $O(n!)$ de permutações geradas, tornando-se inviável já com uma quantidade relativamente pequena de vértices. Dada esta dificuldade , podemos recorrer a heurísticas para nos ajudar na escolha entre as várias alternativas de ciclos hamiltonianos buscando soluções não exatas, porém suficientemente boas (isto é, relativamente próximas da solução exata). De fato, Sipser (1996) prova que o problema do caixeiro viajante é $\mathcal{NP}$-difícil e por isso os algoritmos que rodam rápido não garantem solução ótima e vice-versa (LAWLER et al., 1985).

Novamente, uma solução relativamente simples é a heurística do vizinho mais próximo (GROSS; YELLEN; ZHANG, 2013). A ideia básica é iniciar o ciclo hamiltoniano em um vértice qualquer e, a cada passo, percorrer uma aresta escolhendo um novo vértice adjacente com custo mínimo.

O algoritmo com heurística do vizinho mais próximo é descrito pelos seguintes passos:

Entrada: um grafo completo valorado K_n.

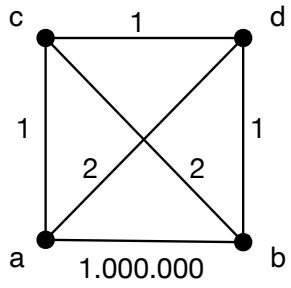
Passos:

1. Escolha um vértice inicial arbitrário v_i. Marque este vértice como visitado;
2. Faça $u \leftarrow v_i$;
3. Dentre todas as arestas incidentes ao vértice u, escolha a aresta (u, v) com menor custo. Se duas opções diferentes tiverem o mesmo custo escolha uma delas aleatoriamente;
4. Marque a aresta (u, v) como percorrida e vá para o vértice v. Marque o vértice v como visitado;
5. Faça $u \leftarrow v$;
6. Repita os passos (3), (4) e (5), considerando somente arestas para vértices não visitados;
7. Feche o ciclo adicionando a aresta do último vértice visitado v até o vértice inicial v_i;

Saída: um ciclo hamiltoniano.

A heurística de vizinho mais próximo encontra rapidamente um resultado e é fácil de implementar. Em geral, o algoritmo pode funcionar muito bem, porém não há garantia de que encontre uma solução ótima. Em alguns casos, como explicam Gross, Yellen e Anderson (2018), o algoritmo pode resultar em ciclos hamiltonianos particularmente ruins (isto é, com custo elevado) como pode ser visto no grafo da Figura 7.9. Se iniciarmos pelo vértice a, esta heurística leva ao ciclo hamiltoniano $\langle a, c, d, b, a\rangle$, com custo 1.000.003, enquanto que há um ciclo hamiltoniano neste grafo com custo 6.

Figura 7.9 – Exemplo de resultado ruim com heurística do vizinho mais próximo



Fonte: o autor.

7.3.3 Heurística da desigualdade do triângulo

Em alguns casos particulares é possível utilizar métodos alternativos que eventualmente garantem a qualidade da solução encontrada. O teorema 7.1 enunciado por Sahni e Gonzalez (1976) trata da garantia de desempenho de algoritmos para o problema do caixeiro viajante.

Teorema 7.1. *Se existe um algoritmo aproximado em tempo polinomial cuja solução para cada instância do problema geral do caixeiro viajante nunca é pior do que uma constante rr vezes a solução ótima, então $P = NP$.*

Para grafos valorados que satisfazem a **desigualdade do triângulo**, temos o chamado "Problema Métrico do Caixeiro Viajante", que pode ser resolvido em tempo polinomial com garantia de qualidade da solução obtida. O termo desigualdade do triângulo refere-se ao fato conhecido na geometria que atesta que nenhum lado de um triangulo é maior que a soma dos outros dois lados e é definida da seguinte forma:

Definição 7.6 (Desigualdade do triângulo)**.** *Seja $G = (V, E)$ um grafo simples valorado com vértices $v_1, v_2, \ldots, v_n$, de tal forma que a aresta (v_i, v_j) tem custo c_{ij}. Então G satisfaz a desigualdade do triângulo se $c_{ij} \leq c_{ik} + c_{kj}$ para todos i, j e k.*

O algoritmo para o problema métrico do caixeiro viajante utiliza diversos conceitos vistos nos capítulos anteriores tais como ciclos eulerianos e árvores geradoras de custo mínimo. Para instâncias de grafos que satisfaçam a desigualdade do triângulo, é garantido que o algoritmo a seguir encontra um ciclo hamiltoniano com custo, no pior caso, nunca superior ao dobro do custo ótimo.

Entrada: um grafo completo valorado K_n que satisfaça a desigualdade do triângulo.

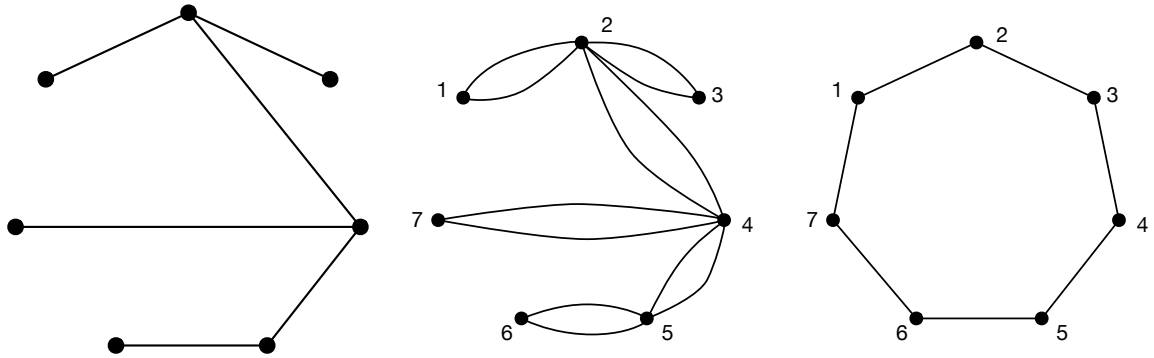
Passos:

1. Encontre uma árvore geradora de custo mínimo T^* do grafo K_n;
2. Crie um grafo euleriano G^e duplicando todas as arestas de T^*;
3. Encontre um ciclo euleriano W em G^e;
4. Construa um ciclo hamiltoniano H em G a partir do ciclo W da seguinte maneira: siga a sequência de arestas e vértices de W até que a próxima aresta da sequência esteja conectada a um vértice previamente visitado. Neste ponto, pule para o próximo vértice não visitado utilizando um atalho (ou seja, uma aresta que não faz parte de W). Retome o percorrimento de W, pegando atalhos sempre que necessário, até que todos os vértices tenham sido visitados. Complete o ciclo hamiltoniano retornando ao vértice inicial pela aresta que o conecta até o último vértice visitado.

Saída: um ciclo hamiltoniano.

Exemplo 7.4. *Suponha que a árvore da Figura 7.10a é uma árvore geradora de custo mínimo T^* de um grafo K_7 valorado (na figura, o grafo original e os custos das arestas foram omitidos). A Figura 7.10b mostra o ciclo euleriano W formado pela duplicação das arestas do grafo, com os números representando a ordem de visitação dos vértices no ciclo euleriano. A Figura 7.10c mostra o ciclo hamiltoniano H resultante quando o ciclo euleriano é modificado com os "atalhos" formados por arestas não pertencentes à árvore geradora de custo mínimo. A desigualdade do triângulo garante que esses atalhos são realmente atalhos.*

Figura 7.10 – Exemplo do algoritmo métrico do caixeiro viajante



(a) T^* (b) W (c) H

Fonte: o autor.

7.3.4 Aplicações do problema do caixeiro viajante

O problema do caixeiro viajante naturalmente surge em muitas aplicações em transportes e logística. Contudo, devido à simplicidade do modelo, ele pode ser aplicado em inúmeras situações práticas em outras áreas. A seguinte lista mostra alguns exemplos de aplicação:

- **Programação de tarefas em série:** suponha que existem n tarefas a serem executadas em série (por exemplo, por uma única máquina). O tempo necessário para preparar a máquina para executar a tarefa j logo após a execução da tarefa i é representado por c_{ij}. Para encontrar uma sequência de todas as tarefas minimizando o tempo total, modelamos um grafo dirigido com n vértices correspondendo às tarefas e as arestas valoradas correspondendo ao tempo de preparação entre as tarefas. A sequência ótima de tarefas corresponde a um ciclo hamiltoniano de custo mínimo.

- **Perfuração de placas:** uma aplicação clássica é a programação de máquinas para perfuração de uma série de furos em placas de circuito impresso ou de qualquer outro objeto. O grafo é modelado com os furos sendo os vértices e as arestas valoradas

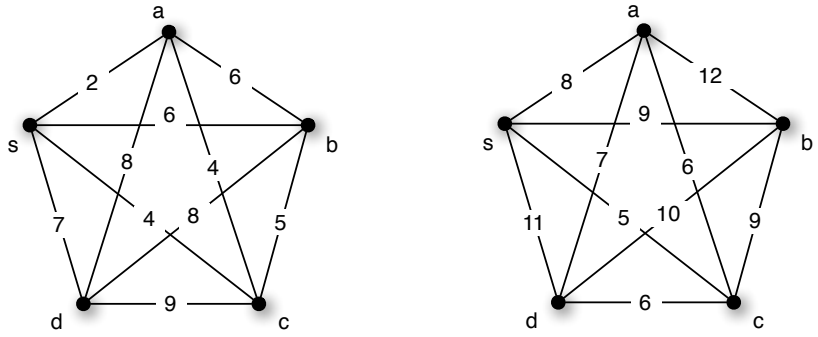
representando o tempo de deslocamento da broca entre dois furos. A tecnologia de furação pode variar de uma indústria para outra, mas sempre que o tempo de deslocamento da broca for significativo em relação ao tempo total de furação da placa, o problema do caixeiro viajante pode ter um papel importante nos ganhos de produtividade e redução de custos.

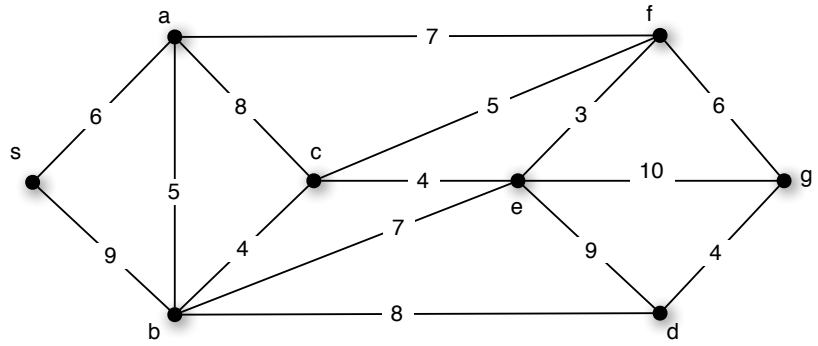
- **Programação de satélites:** o programa da Agência Aeroespacial Norte Americana (NASA) chamado *Starlight Interferometer Program* teve como objetivo prover uma tecnologia para detecção de bioassinaturas no espaço (como água em estado líquido, por exemplo) utilizando imagens geradas por dois satélites operando em conjunto. Bailey, McLain e Beard (2001) publicaram um estudo no qual aplicam o problema do caixeiro viajante para minimizar o uso de combustível para manobrar os satélites nas operações de mira. Os objetos celestes a serem observados foram modelados como vértices e os custos das arestas valoradas representam a quantidade de combustível necessária para reposicionar a mira dos dois satélites de um objeto celeste para outro.

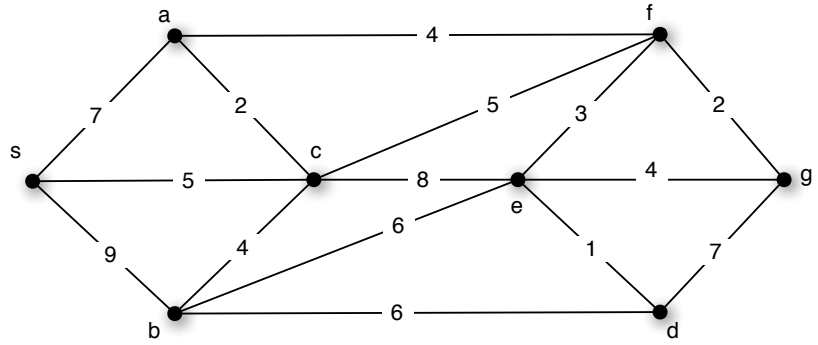
- **Programação de rota de entrega:** esta é uma das aplicações mais conhecidas, na qual os vértices são os pontos de entrega e as arestas representam o tempo (ou distância) entre dois pontos. Um exemplo com importância histórica é a programação de rota de ônibus escolares, que motivou Flood e Savage (1948), pioneiros da pesquisa operacional, a cunharem o termo “Problema do Caixeiro Viajante”.

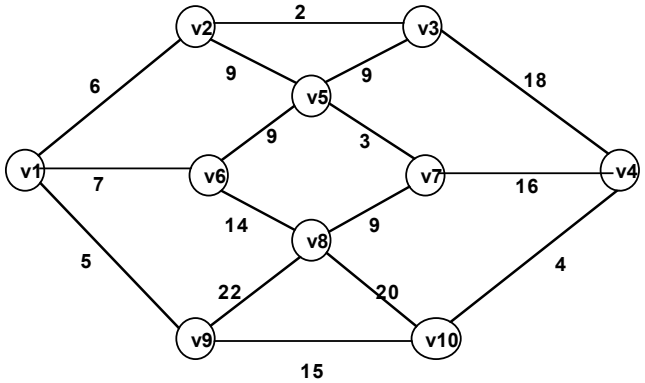
7.4 Exercícios

1. Dados os grafos abaixo, utilize o algoritmo de Dijkstra para calcular os caminhos mínimos entre todos os pares de vértices. O algoritmo deverá ser usado n vezes, uma vez para cada vértice como origem.







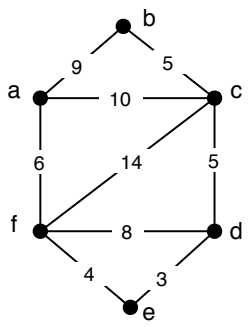


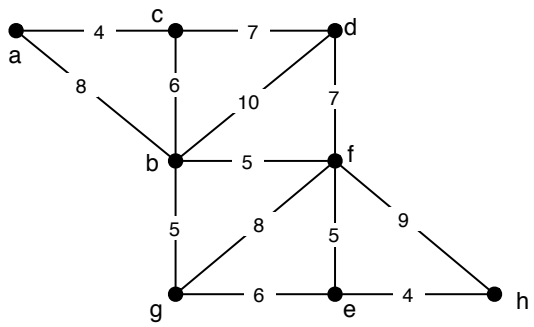
2. Implemente o algoritmo de Floyd em uma linguagem de programação de sua preferência.

3. Nos grafos do exercício 1, encontre os caminhos mínimos entre todos os pares de vértices utilizando o algoritmo de Floyd. Construa também as matrizes de roteamento.

4. Para os cinco grafos do exercício 1, resolva o Problema do Carteiro Chinês, mostrando as principais etapas para chegar à solução, conforme mostrado nos slides de aula.

5. Para cada um dos grafos abaixo, resolva o Problema do Carteiro Chinês, mostrando as principais etapas para chegar à solução, conforme mostrado nos slides de aula.





6. (PosComp – 2007) Considere o problema do caixeiro viajante, definido como se segue. Seja S um conjunto de n cidades (com $n \geq 0$) e $d_{ij} > 0$ a distância entre as cidades i e j, $i, j \in S$, $i \neq j$. Define-se um percurso fechado como sendo um percurso que parte de uma cidade $i \in S$, passa exatamente uma vez por cada cidade de $S - i$, e retorna à cidade de origem. A distância de um percurso fechado é definida como sendo a soma das distâncias entre cidades consecutivas no percurso. Deseja-se encontrar um percurso fechado de distância mínima. Suponha um algoritmo guloso que, partindo da cidade 1, move-se para a cidade mais próxima ainda não visitada e que repita esse processo até passar por todas as cidades, retornando à cidade 1. Considere as seguintes afirmativas.

   I. Todo percurso fechado obtido com esse algoritmo tem distância mínima.
   II. O problema do caixeiro viajante pode ser resolvido com um algoritmo de complexidade linear no número de cidades.
   III. Dado que todo percurso fechado corresponde a uma permutação das cidades, existe um algoritmo de complexidade exponencial no número de cidades para o problema do caixeiro viajante.

   Em relação a essas afirmativas, pode-se afirmar que:

   a) I é falsa e III é correta.
   b) I, II e III são corretas.
   c) apenas I e II são corretas.
   d) apenas I e III são falsas.
   e) I, II e III são falsas.

7. (PosComp – 2014) Assinale a alternativa que apresenta, corretamente, o algoritmo utilizado para determinar o caminho mínimo entre todos os pares de vértices de um grafo.

   a) Bellman-Ford.
   b) Floyd-Warshall.
   c) Dijkstra.
   d) Kruskal.
   e) Prim.