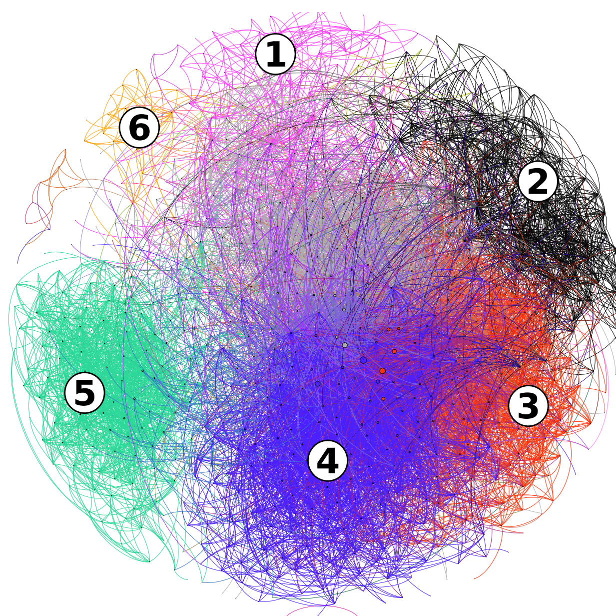
Rafael de Santiago

Anotações para a Disciplina de Grafos

Versão de 27 de março de 2025



Universidade Federal de Santa Catarina

Sumário

List of Algorithms . . . 5
1 Introdução . . . 7
1.1 Definições Iniciais . . . 7
1.1.1 Grafos Valorados ou Ponderados . . . 8
1.1.2 Grafos Orientados . . . 10
1.1.3 Hipergrafo . . . 11
1.1.4 Multigrafo . . . 11
1.1.5 Grau de um Vértice . . . 12
1.1.6 Igualdade e Isomorfismo . . . 12
1.1.7 Partição de Grafos . . . 12
1.1.8 Matriz de Incidência . . . 12
1.1.9 Operações com Grafos . . . 13
1.1.10 Vizinhança . . . 14
1.1.11 Grafo Regular . . . 15
1.1.12 Grafo Simétrico . . . 15
1.1.13 Grafo Anti-simétrico . . . 15
1.1.14 Grafo Completo . . . 15
1.1.15 Grafo Complementar . . . 15
1.1.16 Percursos em Grafos . . . 16
1.1.17 Cintura e Circunferência . . . 16
1.1.18 Planaridade . . . 16
1.1.19 Vértice de Corte . . . 17
1.1.20 Ponte . . . 17
1.1.21 Base . . . 17
1.1.22 Anti-Base . . . 17
1.1.23 SCIE (Subconjunto Internamente Estável ou conjunto independente) . . . 18
1.1.24 SCEE (Subconjunto Externamente Estável ou conjunto dominante) 18
2 Representações Computacionais . . . 19
2.1 Lista de Adjacências . . . 19
2.2 Matriz de Adjacências . . . 20
2.3 Exercícios . . . 21
3 Buscas em Grafos . . . 23
3.1 Busca em Largura . . . 23
3.1.1 Complexidade da Busca em Largura . . . 24
3.1.2 Propriedades e Provas . . . 25
3.1.2.1 Caminhos Mínimos . . . 25
3.1.2.2 Árvores em Largura . . . 28
3.2 Busca em Profundidade . . . 28
3.2.1 Complexidade da Busca em Profundidade . . . 30
4 Caminhos e Ciclos . . . 31
4.1 Caminhos e Ciclos Eulerianos . . . 31
4.1.1 Algoritmo de Hierholzer . . . 33
4.2 Caminhos e Ciclos Hamiltonianos . . . 38
4.2.1 Caixeiro Viajante . . . 38
5 Caminhos de Custo Mínimo . . . 41
5.1 Propriedades de Caminhos Mínimos . . . 43

5.1.1 Propriedade de subcaminhos de caminhos mínimos o são (esses subcaminhos também são caminhos mínimos) 43
5.1.2 Propriedade de desigualdade triangular 43
5.1.3 Propriedade de limite superior 44
5.1.4 Propriedade de inexistência de caminho 45
5.1.5 Propriedade de convergência 45
5.1.6 Propriedade de relaxamento de caminho 46
5.1.7 Propriedade de relaxamento e árvores de caminho mínimo . . . 47
5.1.8 Propriedade de subgrafo dos predecessores 48
5.2 Bellman-Ford 50
5.2.1 Complexidade de Bellman-Ford 51
5.2.2 Corretude de Bellman-Ford 51
5.3 Dijkstra 53
5.3.1 Complexidade de Dijkstra 54
5.3.2 Corretude do Algoritmo de Dijkstra 55
5.4 Floyd-Warshall 56
5.4.1 Complexidade de Floyd-Warshall 57
5.4.2 Corretude de Floyd-Warshall 58
5.4.3 Construção de Caminhos Mínimos para Floyd-Warshall 58
5.5 Dúvidas Frequentes ou Importantes 60

6 Problemas de Travessia 63
6.1 Problema dos Canibais e Missionários 64
6.1.1 Construção e Busca 66
6.2 Problemas dos Potes de Vinho 66
6.3 Problema da Fuga dos Ladrões 67
6.4 Problema dos três maridos ciumentos 67
6.5 Problema de Agrupamento de Indivíduos 67

7 Conectividade 69
7.1 Componentes Fortemente Conexas 69
7.1.1 Complexidade do Algoritmo de Componentes Fortemente Conexas 71
7.1.2 Corretude do Algoritmo de Componentes Fortemente Conexas 72
7.1.2.1 Propriedades de Buscas em Profundidade 72
7.1.2.2 Propriedades de Componentes Fortemente Conexas . 74
7.2 Ordenação Topológica 76
7.2.1 Complexidade da Ordenação Topológica 78
7.2.2 Corretude da Ordenação Topológica 78

8 Árvores Geradoras Mínimas 79
8.1 Propriedades do Método Genérico 80
8.2 Algoritmo de Kruskal 81
8.2.1 Complexidade do Algoritmo de Kruskal 82
8.3 Algoritmo de Prim 83
8.3.1 Complexidade do Algoritmo de Prim 84

9 Fluxo Máximo 85
9.1 Redes de Fluxo 85
9.2 Rede Residual 87
9.3 Caminhos Aumentantes 87
9.4 Cortes de Redes de Fluxo 87
9.5 Ford-Fulkerson 88
9.5.1 Complexidade 89
9.6 Edmonds-Karp 90
9.6.1 Complexidade 91

10 Emparelhamento **93**
10.1 Emparelhamento Máximo em Grafos Bipartidos 93
10.1.1 Resolução por Fluxo Máximo 93
10.1.1.1 Complexidade 95
10.1.2 Algoritmo de Hopcroft-Karp 95
10.1.2.1 Algoritmo detalhado 96
10.1.2.2 Complexidade 98
10.2 Emparelhamento Máximo em Grafos 98
11 Coloração de Grafos **103**
11.1 Aplicações 103
11.2 Heurística DSATUR 104
11.3 Algoritmo de Lawler 105
11.3.1 Complexidade 106
11.4 Listar Conjuntos Independentes Maximais 107
12 Caminho Crítico **109**
12.1 Listar Conjuntos Independentes Maximais 111
12.2 Cálculo do caminho crítico 111
Referências **115**
A Revisão de Matemática Discreta **117**
B Estruturas de Dados Auxiliares **119**
B.1 Conjuntos Disjuntos 119

Índice de algoritmos

1 Criação de uma lista de adjacências para um grafo dirigido e ponderado. 20
2 Criação de uma matriz de adjacências para um grafo dirigido e ponderado. 21

3 Busca em largura. 24
4 Busca em profundidade. 29
5 Algoritmo principal da chamada da busca em profundidade. 30
6 VisitaBuscaProfundidade. 30

7 Algoritmo de Hierholzer. 34
8 Algoritmo de Auxiliar "$buscarSubciclo$". 35
9 Algoritmo de Bellman-Held-Karp. 39

10 Inicialização de G. 42
11 Relaxamento de v. 43
12 Algoritmo de Bellman-Ford. 50
13 Algoritmo de Dijkstra. 54
14 Algoritmo de Floyd-Warshall. 57
15 Algoritmo de Floyd-Warshall com Matriz dos Predecessores. 59
16 Print-Shortest-Path. 60

17 Algoritmo de Componentes-Fortemente-Conexas 70
18 DFS de Cormen et al. (2012). 71
19 DFS-Visit de Cormen et al. (2012). 71
20 DFS para Ordenação Topológica 77
21 DFS-Visit-OT. 77

22 Método Genérico de Cormen et al. (2012). 80
23 Algoritmo de Kruskal. 82
24 Algoritmo de Prim. 84

25 Algoritmo de Ford-Fulkerson. 89
26 Busca em Largura para Edmonds-Karp. 90

27 Adaptação de entrada de Emparelhamento Máximo para um algoritmo de Fluxo Máximo. 94
28 Resolução de Emparelhamento Máximo através um algoritmo de Fluxo Máximo. 95
29 Algoritmo de Hopcroft-Karp. 96
30 Algoritmo de Hopcroft-Karp detalhado. 96
31 BFS 97
32 DFS 97
33 Emparelhamento máximo para grafos não-dirigidos e não-ponderados. 99
34 AumentanteAlternante 100

35 Blossom 100
36 Algoritmo de Edmonds (Blossom). 101

37 Método DSATUR 105
38 Algoritmo de Lawler 106
39 Listar Conjuntos Independentes Maximais 107

40 Listar Conjuntos Independentes Maximais 111

CAPÍTULO 1

Introdução

1.1 Definições Iniciais

Antes de visitar a representação de grafos, é importante que saibamos o que são vértices e arestas. Vértices geralmente são representados como unidades, elementos ou entidades, enquanto as arestas representam as ligações/conexões entre pares de vértices. Geralmente, chamaremos o conjunto de vértices de V e o conjunto de arestas de E. Define-se que $E \subseteq V \times V$. Também usaremos n e m para denotarem o número de vértices e arestas respectivamente, então $n = |V|$ e $m = |E|$. O número de arestas possível em um grafo é $\frac{n^2-n}{2}$.

Um grafo pode ser representado de duas formas (CORMEN et al., 2012). A primeira forma é chamada de lista de adjacências e tem mais popularidade em artigos científicos. Nela, o grafo é representado como uma dupla para especificar vértices e arestas. Por exemplo, para um grafo G, pode-se dizer que o mesmo é uma dupla $G = (V, E)$, especificando assim que o grafo G possui um conjunto V de vértices e E de arestas. A segunda forma seria uma através de uma matriz binária, chamada de matriz de adjacência. Normalmente representada pela letra $A(G)$, a matriz é definida por $A(G) \in \{0,1\}^{|V|\times|V|}$, a qual seus elementos $a_{u,v} = 1$ se existir uma aresta entre os vértices u e v. Um exemplo das das formas para um mesmo grafo pode ser visualizado no Exemplo 1.1.1.

Example 1.1.1. A Figura 1 exibe um grafo de 4 vértices e 4 arestas. Na representação por listas de adjacências, o grafo pode ser representado da seguinte forma

$$G = (\{1,2,3,4\},\{\{1,2\},\{1,4\},\{2,4\},\{3,4\}\}). \tag{1.1}$$

A representação por uma matriz de adjacência ficaria assim

$A(G) =$

	1	2	3	4
1	0	1	0	1
2	1	0	0	1
3	0	0	0	1
4	1	1	1	0

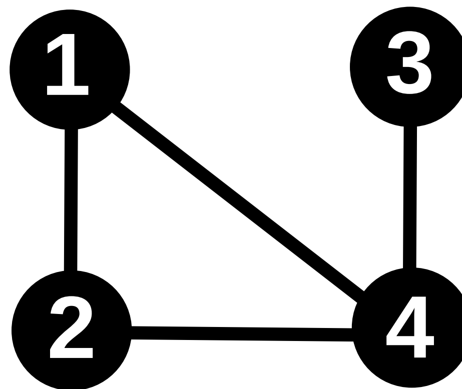


Figura 1 – Exemplo de grafo com 4 vértices e 4 arestas.

1.1.1 Grafos Valorados ou Ponderados

Um grafo é valorado quando um peso ou valor é associado a suas arestas. Na literatura, a definição do grafo passa a ser uma tripla $G = (V, E, w)$, na qual V é o conjunto de vértices, E é o conjunto de arestas e $w : e \in E \to \mathbb{R}$ é a função que especifica o valor.

Quando não se possui valornas arestas, parte-se de uma relação binária entre existir ou não uma aresta entre dois vértices. Neste caso, se u e v possui uma aresta, geralmente se simboliza essa ligação com o valor 1, e se não existir 0.

Em uma matriz de adjacências para grafos valorados, o valor das arestas aparecem nas células da matriz. Em um par de vértices que não possui valor estabelecido (não há aresta), representa-se com uma lacuna ou com um valor simbólico para o problema que o grafo representa. Por exemplo, se os valores representam as distâncias, geralmente se associa o valor infinito aos pares de vértices que não possuem arestas.

Um exemplo de grafo valorado e suas representações pode ser visualizado no Exemplo 1.1.2.

Example 1.1.2. A Figura 2 exibe um grafo valorado de 4 vértices e 4 arestas. Na representação por listas de adjacências, o grafo pode ser representado da seguinte forma

$$G = (\{1,2,3,4\}, \{\{1,2\}, \{1,4\}, \{2,4\}, \{3,4\}\}, w). \tag{1.2}$$

A função w teria os seguintes valores: $w(\{1,2\}) = 8$, $w(\{1,4\}) = 9$, $w(\{2,4\}) = 5$ e $w(\{3,4\}) = 7$.

A representação por uma matriz de adjacência ficaria assim

$A(G) =$

	1	2	3	4
1	0	8	0	9
2	8	0	0	5
3	0	0	0	7
4	9	5	7	0

ou desta outra forma para o caso de uma aplicação a problemas que envolvam distâncias

$A(G) =$

	1	2	3	4
1	∞	8	∞	9
2	8	∞	∞	5
3	∞	∞	∞	7
4	9	5	7	∞

.

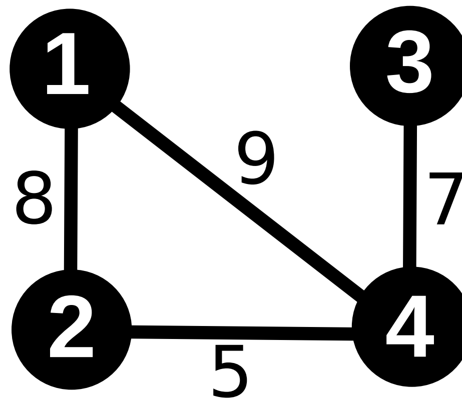


Figura 2 – Exemplo de grafo valorado com 4 vértices e 4 arestas.

Grafos com Sinais

Para representar alguns problemas, utiliza-se valores negativos associados às arestas. Um exemplo disso, seriam grafos que representem relações de amizade e de inimizade. Para amizade, utiliza-se o valor 1 e para inimizade o valor -1. Nesse caso, não dizemos que o grafo é valorado ou ponderado, mas sim um grafo com sinais. Quando os valores negativos e positivos podem ser diferentes de 1 e -1, diz-se que os grafos são valorados e com sinais.

1.1.2 Grafos Orientados

Um grafo orientado é aquele no qual suas arestas possuem direção. Nesse caso, não chamamos mais de arestas e sim de arcos. Um grafo orientado é definido como uma dupla $G = (V, A)$, a qual V é o conjunto de vértices e A é o conjunto de arcos. O conjunto de arcos é composto por pares ordenados (u, v), os quais $u, v \in V$ e representam um arco saindo de u e incidindo em v. Duas funções importantes devem ser consideradas nesse contexto: a função de arcos saintes $\delta^+(v) = \{(v, u) : (v, u) \in A\}$ e arcos entrantes $\delta^-(v) = \{(u, v) : (u, v) \in A\}$.

O Exemplo 1.1.3 exibe a representação de um grafo orientado.

Example 1.1.3. A Figura 3 exibe um grafo orientado de 4 vértices e 4 arestas. Na representação por listas de adjacências, o grafo pode ser representado da seguinte forma

$$G = (\{1,2,3,4\}, \{(1,4),(2,1),(4,2),(4,3)\}). \tag{1.3}$$

A representação por uma matriz de adjacência ficaria assim

$$A(G) = \begin{array}{c|c|c|c|c|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 0 & 0 & 1 \\ \hline 2 & 1 & 0 & 0 & 0 \\ \hline 3 & 0 & 0 & 0 & 0 \\ \hline 4 & 0 & 1 & 1 & 0 \\ \hline \end{array}.$$

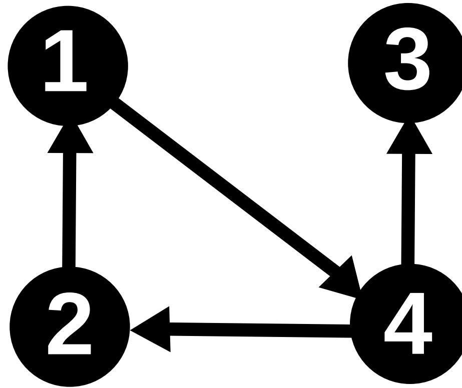


Figura 3 – Exemplo de grafo orientado com 4 vértices e 4 arcos.

1.1.3 Hipergrafo

Um hipergrafo $H = (V,E)$ é um grafo no qual as arestas podem conectar qualquer número de vértices. Cada aresta é chamada de hiperaresta $E \subseteq 2^V \setminus \{\{\}\}$.

1.1.4 Multigrafo

Um multigrafo $G = (V,E)$ é um grafo que permite múltiplas arestas para o mesmo par de vértices. Logo, não se tem mais um conjunto de arestas, mas sim uma tupla de arestas. Para o exemplo da Figura 4, têm-se $E = (\{1,2\},\{1,2\},\{1,4\},\{2,4\},\{3,4\},\{3,4\})$.

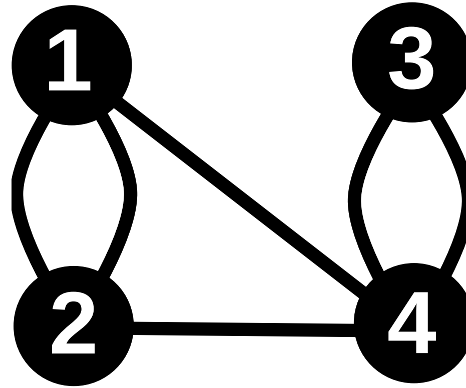


Figura 4 – Exemplo de um multigrafo com 4 vértices e 6 arestas.

1.1.5 Grau de um Vértice

O grau de um vértice é a quantidade de arestas que se conectam a determinado vértice. É denotada por uma função d_v, onde $v \in V$. Em um grafo orientado, o número de arcos saintes para um vértice v é denotado por d_v^+, e o número de arcos entrantes é denotado por d_v^-.

1.1.6 Igualdade e Isomorfismo

Diz-se que dois grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$ são iguais se $V_1 = V_2$ e $E_1 = E_2$. Os dois grafos são considerados isomorfos se existir uma função bijetora (uma-por-uma) para todo $v \in V_1$ e para todo $u \in V_2$ preserve as relações de adjacência (NETTO, 2006).

1.1.7 Partição de Grafos

Uma partição de um grafo é uma divisão disjunta de seu conjunto de vértices. Um grafo $G = (V, E)$ é dito k-partido se existir uma partição $P = \{p_i | i = 1, \ldots, k \wedge \forall j \in \{1, \ldots, k\}, j \neq i(p_i \cap p_j \neq \{\})\}$. Quando $k = 2$, dize que o grafo é bipartido (NETTO, 2006).

1.1.8 Matriz de Incidência

Sobre um grafo orientado $G = (V, E)$, uma matriz de incidência $B(G) = \{+1, -1, \}^{|V| \times |A|}$ mapeia a origem e o destino de cada arco no grafo G. Dado um arco (u, v), $b_{u,(u,v)} = +1$

e $b_{v,(u,v)} = -1$ (NETTO, 2006).

1.1.9 Operações com Grafos

As seguintes operações binárias são descritas em Netto (2006):

- União: Dados os grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$;
- Soma (ou *join*): Dados os grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, $G_1 + G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{\{u, v\} : u \in V_1 \wedge v \in V_2\})$;
- Produto cartesiano: Dados os grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, $G_1 \times G_2 = (V_1 \times V_2, E)$, onde $E = \{\{(v, w), (x, y)\} : (v = x \wedge \{w, y\} \in E_2) \vee (w = y \wedge \{v, x\} \in E_1)\}$. $G_1 \times G_2$ e $G_2 \times G_1$ são isomorfos;
- Composição ou produto lexicográfico: Dados os grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, $G_1 \circ G_2 = (V_1 \times V_2, E)$, onde $E = \{\{(v, w), (x, y)\} : (\{v, x\} \in E_1 \vee v = x) \wedge \{w, y\} \in E_2\}$;
- Soma de arestas: Dados os grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, os quais $V_1 = V_2$, $G_1 \oplus G_2 = (V_1, E_1 \cup E_2)$.

A seguinte operação unária é descrita em Netto (2006):

- Contração de dois vértices: Dado um grafo $G = (V, E)$ e dois vértices $u, v \in V$, a operação de contração desses dois vértices em G, gera um grafo $G' = (V', E')$ o qual $V' = V \setminus \{u, v\} \cup \{uv\}$ e $E' = \{\{x, y\} \in E : x \neq u \wedge x \neq v\} \cup \{\{x, uv\} : \{x, u\}, \{x, v\} \in E\}$.

Outras operações sobre grafos são descritas na literatura. Esse texto irá omití-las por enquanto para que sejam utilizados no momento mais oportuno. Sõ elas: inserção e remoção de vértices e arestas, desdobramento de um vértice. Essa última depende do contexto de aplicação.

1.1.10 Vizinhança

A vizinhança de vértices é diferente para grafos não-orientados e orientados. Para um grafo grafo não-orientado $G = (V, E)$, uma função de vizinhança é definida por $N : v \in V \rightarrow \{u \in V : \{v, u\} \in E\}$ e indica o conjunto de todos os vizinhos de um vértice específico. Para o grafo do Exemplo 1.1.1, $N(1) = \{2, 4\}$.

Para um grafo orientado $G = (V, A)$, diz-se que um vértice $u \in V$ é sucessor de $v \in V$ quando $(v, u) \in A$; e $u \in V$ é antecessor de $v \in V$ quando $(u, v) \in A$. As funções de vizinhança para um grafo orientado G são $N^+ : v \in V \rightarrow \{u \in V : (v, u) \in A\}$, $N^- : v \in V \rightarrow \{u \in V : (u, v) \in A\}$, e $N(v) = N^+(v) \cup N^-(v)$.

Diz-se que a vizinhança de v é fechada quando esse mesmo vértice se inclui no conjunto de vizinhos. A função que representa vizinhança fechada v é simbolizada neste texto como $N_*(v) = N(v) \cup \{v\}$.

As funções de vizinhança também podem ser utilizadas para identificar um conjunto de vértices vizinhos de um grupo de vértices em um grafo $G = (V, E)$ (orientado ou não). Nesse contexto, $N(S) = \bigcup_{v \in S} N(v)$, $N^+(S) = \bigcup_{v \in S} N^+(v)$, e $N^-(S) = \bigcup_{v \in S} N^-(v)$.

As noções de sucessor e antecessor podem ser aplicadas iterativamente. As Equações (1.4), (1.5), (1.6) e (1.7) exibem exemplos de fechos transitivos diretos.

$$N^0(v) = \{v\} \tag{1.4}$$

$$N^{+1}(v) = N^+(v) \tag{1.5}$$

$$N^{+2}(v) = N^+\left(N^{+1}(v)\right) \tag{1.6}$$

$$N^{+n}(v) = N^+\left(N^{+(n-1)}(v)\right) \tag{1.7}$$

Chama-se de fecho transitivo direto aqueles que correspondem aos vizinhos sucessivos e os inversos os que correspondem aos vizinhos antecessores. Um fecho transitivo

direto de um vértice v de um grafo $G = (V, E)$ são todos os vértices atingíveis a partir v no grafo G; ele é representado pela função $R^+(v) = \bigcup_{k=0}^{|V|} N^{+k}(v)$. Um fecho transitivo inverso de v é o conjunto de vértices que atingem v; ele é representado pela função $R^-(v) = \bigcup_{k=0}^{|V|} N^{-k}(v)$. Diz-se que w é descendente de v se $w \in R^+(v)$. Diz-se que w é ascendente de v se $w \in R^-(v)$.

1.1.11 Grafo Regular

Um grafo não-orientado $G = (V, E)$ que tenha $d(v) = k \forall v \in V$ é chamado de grafo k-regular ou de grau k. Um grafo orientado $G_o = (V, A)$ que possui a propriedade $d^+(v) = k \forall v \in V$ é chamado de grafo exteriormente regular de semigrau k. Se G_o tiver $d^-(v) = k \forall v \in V$ é chamado de grafo interiormente regular de semigrau k.

1.1.12 Grafo Simétrico

Um grafo orientado $G = (V, A)$ é simétrico se $(u, v) \in A \iff (v, u) \in A \forall u, v \in V$.

1.1.13 Grafo Anti-simétrico

Um grafo orientado $G = (V, A)$ é anti-simétrico se $(u, v) \in A \iff (v, u) \notin A \forall u, v \in V$.

1.1.14 Grafo Completo

Um grafo completo $G = (V, E)$ é completo se $E = V \times V$.

Grafos bipartidos completos $G_B = ((X, Y), E)$ possuem $E = X \times Y$.

1.1.15 Grafo Complementar

Para um grafo $G = (V, E)$, um grafo complementar é definido por $G^c = \overline{G} = (V, (V \times V) \setminus E)$.

1.1.16 Percursos em Grafos

"Um percurso, itinerário ou cadeia é uma família de ligações sucessivamente adjacentes, cada uma tendo uma extremidade adjacente a anterior e a outra à subsequente (à execeção da primeira e da última)" (NETTO, 2006). Diz-se que um percurso é aberto quando a última ligação é adjacente a primeira. Têm-se desse modo um ciclo.

Um percurso é considerado simples se não repetir ligações (NETTO, 2006).

Caminhos são cadeias em grafos orientados.

Circuitos são ciclos em grafos orientados.

1.1.17 Cintura e Circunferência

Cintura de um grafo G é comprimendo do menor ciclo existente no grafo. É representada pela função $g(G)$. A circunferência é comprimento do maior ciclo. A circunferência do grafo G é representada pela função $c(G)$.

1.1.18 Planaridade

Netto (2006) comenta que a noção de planaridade se baseia na ideia da representação de um conjunto de elementos em um plano, como em um mapa. Nesse contexto, um grafo planar pode ser representado em um plano sem que suas arestas se cruzem. Um grafo plano é um grafo planar que foi desenhado numa superfície plana.

Netto (2006) cita três resultados mais conhecidos para considerar que um grafo é planar:

- (**Harary**) Um grafo é planar sse não tiver, como subgrafo, um grafo homeomorfo[1] a K_5[2] ou a $K_{3,3}$[3].

[1] Um grafo H é homeomorfo a um grafo G, se H pode ser obtido de G pela inserção de vértices de grau 2 em pontos intermediários de suas arestas (substitui uma aresta $\{u, v\}$, adicionando um vértice w e duas arestas $\{u, w\}$ e $\{v, w\}$) (NETTO, 2006).

[2] Grafo K_5: grafo completo com cinco vértices.

[3] Grafo $K_{3,3}$: grafo bipartido com 3 vértices para cada parte e cada parte se conecta a outra completamente (9 arestas).

- (**Wagner**) Um grafo G é planar sse ele não contiver um subgrafo contratível a K_5 ou a $K_{3,3}$.
- (**Whitney**) Um grafo G 2-conexo[4] é planar sse possui um dual[5].

1.1.19 Vértice de Corte

Um vértice é dito ser um vértice de corte se sua remoção (juntamente com as arestas a ele conectadas) provoca um redução na conexidade do grafo (passa a ter mais componentes desconexas).

1.1.20 Ponte

Uma aresta é dita ser um a ponte se sua remoção provoca um redução na conexidade do grafo (passa a ter mais componentes desconexas).

1.1.21 Base

Uma base de um grafo $G(V,E)$ é um subconjunto $B \subseteq V$, tal que:

- dois vértices quaisquer de B não são ligados por nenhum caminho;
- todo vértice não pertencente a B pode ser atingido por um caminho partindo de B.

1.1.22 Anti-Base

Uma anti-base de um grafo $G(V,E)$ é um subconjunto $A \subseteq V$, tal que:

- dois vértices quaisquer de A não são ligados por nenhum caminho;
- de todo vértice não pertencente a A pode se atingir A por um caminho.

[4] Grafo k-conexo: são necessários desconectar no mínimo k vértices para desconectar o grafo conexo.

[5] Um grafo dual D é obtido a partir de um grafo G tal que cada vértice de D corresponde a uma face em G e cada aresta em D conecta duas faces adjacentes em G.

1.1.23 SCIE (Subconjunto Internamente Estável ou conjunto independente)

Seja $G(V,A)$ um grafo não orientado. Diz-se que $S \subset V$ é um subconjunto internamente estável se dois vértices quaisquer de S nunca são adjacentes entre si.

Agora, se dado um SCIE S não existe um outro SCIE S' tal que $S \subset S'$, então S é dito ser um SCIE maximal.

1.1.24 SCEE (Subconjunto Externamente Estável ou conjunto dominante)

Seja $G(V,A)$ um grafo orientado. Diz-se que $T \subset V$ é um subconjunto externamente estável se todo vértice não pertencente a T tiver pelo menos um vértice de T como sucessor. A Figura 5 traz um grafo dirigido com o conjunto dominante $\{A,B,C\}$.

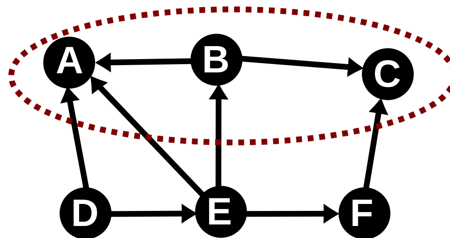


Figura 5 – Exemplo de conjunto SCEE.

Se dado um SCEE T não existe um outro SCEE T' tal que $T' \subset T$, então T é dito ser um SCEE minimal. Este conceito também pode ser aplicado a grafos não-dirigidos, bastando que considere-se que todo vértice exterior a T deva ter como adjacente pelo menos um vértice de T.

CAPÍTULO 2

Representações Computacionais

Duas formas de representação computacional de grafos são amplamente utilizadas. São elas "listas de adjacências" e "por matriz de adjacências" (CORMEN et al., 2012). Elas possuem vantagens e desvantagens principamente relacionadas à complexidade computacional (consumo de recursos em tempo e espaço). Detalhes sobre vantagens e desvantagens não aparecerão nesse documento. Um de nossos objetivos do momento será implementar e avaliar as duas formas de representação.

2.1 Lista de Adjacências

A representação de um grafo $G = (V, E)$ por listas de adjacências consiste em um arranjo, chamado aqui de *Adj*. Esse arranjo é composto por $|V|$ listas, e cada posição do arranjo representa as adjacências de um vértice específico (CORMEN et al., 2012). Para cada $\{u, v\} \in E$, têm-se $Adj[u] = (\ldots, v, \ldots)$ e $Adj[v] = (\ldots, u, \ldots)$ quando G for não-dirigido. Quando G for dirigido, para cada $(u, v) \in E$, têm-se $Adj[u] = (\ldots, v, \ldots)$.

Para grafos ponderados, Cormen et al. (2012) sugere o uso da própria estrutura de adjacências para armazenar o peso. Dado um grafo ponderado não-dirigido $G = (V, E, w)$, para cada $\{u, v\} \in E$, têm-se $Adj[u] = \big(\ldots, \big(v, w(\{u, v\})\big), \ldots\big)$ e $Adj[v] = \big(\ldots, \big(u, w(\{u, v\})\big), \ldots\big)$. Quando o grafo for dirigido, para cada $(u, v) \in E$, têm-se $Adj[u] = \big(\ldots, \big(v, w((u, v))\big), \ldots\big)$.

O Algoritmo 1 representa a carga de um grafo dirigido e ponderado $G = (V, A, w)$ em uma lista de adjacências *Adj*.

Algoritmo 1: Criação de uma lista de adjacências para um grafo dirigido e ponderado.

```
Input :um grafo dirigido e ponderado G = (V, A, w)
1 criar arranjo Adj[|V|]
2 foreach v ∈ V do
3     Adj[v] ← listaVazia()
4 foreach (u, v) ∈ A do
5     Adj[u] ← Adj[u] ∪ (v, w((u, v)))
6 return Adj
```

2.2 Matriz de Adjacências

Uma matriz de adjacência é uma representação de um grafo através de uma matriz A. Para um grafo não-dirigido $G = (V, E)$, $A = \mathbb{B}^{|V| \times |V|}$, na qual cada elemento $a_{u,v} = 1$ e $a_{v,u} = 1$ se $\{u, v\} \in E$; $a_{u,v} = 0$ e $a_{v,u} = 0$ caso $\{u, v\} \notin E$. Para todo grafo não-dirigido G, $a_{u,v} = a_{v,u}$.

Para um grafo dirigido $G = (V, X)$, $A = \mathbb{B}^{|V| \times |V|}$, na qual cada elemento $a_{u,v} = 1$ se $(u, v) \in A$; $a_{u,v} = 0$ e $a_{u,v} = 0$ caso $(u, v) \notin X$.

Para um grafo não-dirigido e ponderado $G = (V, E, w)$, a matriz será formada por células que comportem o tipo de dado representado pelos pesos. Assumindo que os pesos serão números reais, então a matriz de adjacências será $A = \mathbb{R}^{|V| \times |V|}$. Cada elemento $a_{u,v} = w(\{u, v\})$ e $a_{v,u} = w(\{u, v\})$ se $\{u, v\} \in E$; $a_{u,v} = \epsilon$ e $a_{v,u} = \epsilon$ caso $\{u, v\} \notin E$. ϵ é um valor que representa a não conexão, geralmente 0, $+\infty$ ou $-\infty$ dependendo do contexto de aplicação.

O Algoritmo 2 representa a carga de um grafo dirigido e ponderado $G = (V, A, w)$ em uma matriz de adjacências *Adj*.

Algoritmo 2: Criação de uma matriz de adjacências para um grafo dirigido e ponderado.

Input : um grafo dirigido e ponderado $G = (V, A, w : A \rightarrow \mathbb{R})$, um símbolo ϵ que representa a não adjacência

```
1 Adj ← ℝ^{|V|×|V|}
2 foreach v ∈ V do
3     foreach u ∈ V do
4         Adj_{u,v} ← ε
5 foreach (u, v) ∈ A do
6     Adj_{u,v} ← w((u, v))
7 return Adj
```

2.3 Exercícios

Implemente as duas bibliotecas para grafos. Preencha a seguinte tabela a partir da análise computacional, de acordo com as operações abaixo determinadas.

	Lista de Adjacências	**Matriz de Adjacências**
Inserção de vértice		
inserção de arestas		
Remoção de vértice		
Remoção de arestas		
Teste se $\{u, v\} \in E$		
Percorrer vizinhos		
Grau de um vértice		

CAPÍTULO 3

Buscas em Grafos

3.1 Busca em Largura

Dado um grafo $G = (V, E)$ e uma origem s, a **busca em largura** (Breadth-First Search - BFS) explora as arestas/arcos de G a partir de s para cada vértice que pode ser atingido a partir de s. É uma exploração por nível. O procedimento descobre as distâncias (número de arestas/arcos) entre s e os demais vértices atingíveis de G. Pode ser aplicado para grafos orientados e não-orientados (CORMEN et al., 2012).

O algoritmo pode produzir uma árvore de busca em largura com raiz s. Nessa árvore, o caminho de s até qualquer outro vértice é um caminho mínimo em número de arestas/arcos (CORMEN et al., 2012).

O Algoritmo 3 descreve as operações realizadas em uma busca em largura. Nele, criam-se três estruturas de dados que serão utilizadas para armazenar os resultados da busca. O arranjo C_v é utilizado para determinar se um vértice $v \in V$ foi conhecido ou não; D_v determina a distância percorrida até encontrar o vértice $v \in V$; e A_v determina o vértice antecessor ao $v \in V$ em uma busca em largura a partir de s (CORMEN et al., 2012).

Algoritmo 3: Busca em largura.

```
Input   :um grafo G = (V, E), vértice de origem s ∈ V
   // configurando todos os vértices
1  C_v ← false ∀ v ∈ V
2  D_v ← ∞ ∀ v ∈ V
3  A_v ← null ∀ v ∈ V
   // configurando o vértice de origem
4  C_s ← true
5  D_s ← 0
   // preparando fila de visitas
6  Q ← Fila()
7  Q.enqueue(s)
   // propagação das visitas
8  while Q.empty() = false do
       // visitando o vértice u
9      u ← Q.dequeue()
10     foreach v ∈ N(u) do
11         if C_v = false then
               // conhecendo o vértice v
12             C_v ← true
13             D_v ← D_u + 1
14             A_v ← u
15             Q.enqueue(v)
16 return (D, A)
```

3.1.1 Complexidade da Busca em Largura

O número de operações de enfileiramento e desenfileiramento é limitado a $|V|$ vezes, pois visita-se no máximo $|V|$ vértices. Como as operações de enfileirar e desenfileirar podem ser realizadas em tempo $\Theta(1)$, então para realizar estas operações demanda-se tempo de $O(|V|)$. Deve-se considerar ainda, que muitas arestas/arcos incidem em vértices já visitados, então inclui-se na complexidade de uma BFS a varredura de todas as adjacências, que demandaria $\Theta(|E|)$. Diz-se então, que a complexidade computacional da BFS é $O(|V| + |E|)$.

3.1.2 Propriedades e Provas

3.1.2.1 Caminhos Mínimos

A busca em largura garante a descoberta dos caminhos mínimos em um grafo não-ponderados $G = (V, E)$ de um vértice de origem $s \in V$ para todos os demais atingíveis. Para demonstrar isso, Cormen et al. (2012) examina algumas propriedades importantes a seguir. Considere a distância de um caminho mínimo $\delta(s, v)$ de s a v como o número mínimo de arestas/arcos necessários para percorrer esse caminho.

Lema 3.1.1. *Seja $G = (V, E)$ um grafo orientado ou não-orientado e seja $s \in V$ um vértice arbitrário, então $\delta(s, v) \leq \delta(s, u) + 1$ para qualquer aresta/arco $(u, v) \in E$.*

Prova: Se u pode ser atingido a partir de s, então o mesmo ocorre com v. Desse modo, o caminho mínimo de s para v não pode ser mais longo do que o caminho de s para u seguido pela aresta/arco (u, v) e a desigualdade vale. Se u não pode ser alcançado por s, então $\delta(s, u) = \infty$, e a desigualdade é válida. ■

Lema 3.1.2. *Seja $G = (V, E)$ um grafo orientado ou não-orientado e suponha que G tenha sido submetido ao algoritmo BFS (Algoritmo 3) partindo de um dado vértice de origem $s \in V$. Ao parar, o algoritmo BFS satisfará $D_v \geq \delta(s, v) \forall v \in V$.*

Prova: Utiliza-se a indução em relação ao número de operações de enfileiramento (*enqueue*). A hipótese indutiva é $D_v \geq \delta(s, v) \forall v \in V$.

A base da indução é a situação imediatamente após s ser enfileirado na linha 7 do Algoritmo 3. A hipótese indutiva se mantém válida nesse momento porque $D_s = 0 = \delta(s, s)$ e $D_v = \infty \geq \delta(s, v) \forall v \in V \setminus \{s\}$.

Para o passo da indução, considere um vértice v não-conhecido ($C_v =$**false**) que é descoberto depois do último desenfileirar. Consideramos que o vértice desenfileirado é $u \in V$. A hipótese da indução implica que $D_u \geq \delta(s, u)$. Pela atribuição da linha 13 e pelo Lema 3.1.1, obtem-se

$$D_v = D_u + 1 \geq \delta(s,u) + 1 \geq \delta(s,v). \tag{3.1}$$

Então, o vértice v é enfileirado e nunca será enfileirado novamente porque ele também é marcado como conhecido e as operações entre as linhas 12 e 15 são apenas executadas para vértices não-conhecidos. Desse modo, o valor de D_v nunca muda novamente e a hipótese de indução é mantida. ■

Lema 3.1.3. *Suponha que durante a execução do algoritmo de busca em largura (Algoritmo 3) em um grafo $G = (V,E)$, a fila Q contenha os vértices $(v_1, v_2, \ldots, v_r)$, onde v_1 é o início da fila e v_r é o final da fila. Então, $D_{v_r} \leq D_{v_1} + 1$ e $D_{v_i} \leq D_{v_{i+1}}$ para todo $i \in \{1,2,\ldots,r-1\}$.*

Prova: A prova é realizada por indução relacionada ao número de operações de fila.

Para a base da indução, imediatamente antes do laço de repetição (antes da linha 8), têm-se apenas o vértice s na fila. O lema se mantém nessa condição.

Para o passo da indução, deve-se provar que o lema se mantém para depois do desenfileiramento quanto do enfileiramento de um vértice. Se o início v_1 é desenfileirado, v_2 torna-se o início. Pela hipótese de indução, $D_{v_1} \leq D_{v_2}$. Então $D_{v_r} \leq D_{v_1} + 1 \leq D_{v_2} + 1$. Assim, o lema prossegue com v_2 no início.

Quando enfileira-se um vértice v (linha 15), ele se torna v_{r+1}. Nesse momento, já se removeu da fila o vértice u cujo as adjacências estão sendo analisadas, e pela hipótese de indução, o novo início v_1 deve ter $D_{v_1} \geq D_u$. Assim, $D_{v_{r+1}} = D_v = D_u + 1 \leq D_{v_1} + 1$. Pela hipótese indutiva, têm-se $D_{v_r} \leq D_u + 1$, portanto $D_{v_r} \leq D_u + 1 = D_v = D_{v_{r+1}}$ e o lema se mantém quando um vértice é enfileirado. ■

Corolário 3.1.4. *Suponha que os vértices v_i e v_j sejam enfileirados durante a execução do algoritmo de busca em largura (Algoritmo 3) e que v_i seja enfileirado antes de v_j. Então, $D_{v_i} \leq D_{v_j}$ no momento que v_j é enfileirado.*

Prova: Imediata pelo Lema 3.1.3 e pela propriedade de que cada vértice recebe um valor D finito no máximo uma vez durante a execução do algoritmo. ■

Teorema 3.1.5. *Seja $G = (V, E)$ um grafo orientado ou não-orientado, e suponha que o algoritmo de busca em largura (Algoritmo 3) seja executado em G partindo de um dado vértice $s \in V$. Então, durante sua execução, o algoritmo descobre todo o vértice $v \in V$ atingível por s. Ao findar sua execução, o algoritmo retornará a distância mínima entre s e $v \in V$, então $D_v = \delta(s, v) \forall v \in V$.*

Prova: Por contradição, suponha que algum vértice receba um valor de distância não igual à distância de seu caminho mínimo. Seja v um vértice (com $\delta(s, v)$) que recebe tal valor de distância D_v incorreto. O vértice v não poderia ser s, pois o algoritmo define $D_s = 0$, o que estaria correto. Então deve-se encontrar um outro $v \neq s$. Pelo Lema 3.1.2, $D_v \geq \delta(s, v)$ e portanto, temos $D_v > \delta(s, v)$. O vértice v deve poder ser visitado a partir de s, se não puder, $\delta(s, v) = \infty \geq D_v$. Seja u o vértice imediatamente anterior a v em um caminho mínimo de s a v, de modo que $\delta(s, v) = \delta(s, u) + 1$. Como $\delta(s, u) < \delta(s, v)$, e em razão de selecionar-se v, têm-se $D_u = \delta(s, u)$. Reunindo essas propriedades, obtém-se a seguinte hipótese de contradição

$$D_v > \delta(s, v) = \delta(s, u) + 1 = D_u + 1. \tag{3.2}$$

Considere o momento que o algoritmo opta por desenfileirar o vértice u de Q. Nesse momento, o vértice v pode ter sido não-conhecido, conhecido e já foi removido da fila, ou conhecido e está na fila. O restante da prova analisa cada um desses casos:

- Se v é não-conhecido ($C_v =$**false** e lembrando que v é vizinho de u), então a operação na linha 13 define $D_v = D_u + 1$, contradizendo o que é dito na Equação 3.2.

- Se v já foi conhecido ($C_v =$**true**) e foi removido da fila, pelo Corolário 3.1.4, têm-se $D_v \leq D_u$ que também contradiz o que é dito na Equação 3.2.

- Se v já foi conhecido e permanece na fila, quando v fora enfileirado w era o vértice antecessor imediato no caminho até v, logo $D_v = D_w + 1$. Considere também

que w já foi desenfileirado. Porém, pelo Corolário 3.1.4, $D_w \leq D_u$, então, temos $D_v = D_w + 1 \leq D_u + 1$, contradizendo a Equação 3.2.

■

3.1.2.2 Árvores em Largura

O algoritmo de busca em largura (Algoritmo 3) criar uma árvore de busca em largura à medida que efetua busca no grafo $G = (V, E)$. Também chamada de "subgrafo dos predecessores", uma árvore de busca em lagura pode ser definida como $G_\pi = (V_\pi, E_\pi)$, na qual $V_\pi = \{v \in V : A_v \neq \textbf{null}\} \cup \{s\}$ e $E_\pi = \{(A_v, \pi, v) : v \in V_\pi \backslash \{s\}\}$.

3.2 Busca em Profundidade

A busca em profundidade (Depth-First Search - DFS) realiza a visita a vértices cada vez mais profundos/distantes de um vértice de origem s até que todos os vértices sejam visitados. Parte-se a busca do vértice mais recentemente descoberto do qual ainda saem arestas inexploradas. Depois que todas as arestas foram visitadas no mesmo caminho, a busca retorna pelo mesmo caminho para passar por arestas inexploradas. Quando não houver mais arestas inexploradas a busca em profundidade pára (CORMEN et al., 2012).

O Algoritmo 4 apresenta um pseudo-código para a busca em profundidade. Note que no lugar de usar uma fila, como na busca em largura (vide Algoritmo 3, utiliza-se uma pilha. Os arranjos C_v, T_v, e A_v $\forall v \in V$ são respectivamente o arranjo de marcação de visitados, do tempo de visita e do vértice antecessor à visita.

Cormen et al. (2012) afirma que é mais comum realizar a busca em profundidade de várias fontes. Desse modo, seu livro reporta um algoritmo que sempre que um subgrafo conexo é completamente buscado, parte-se de um outro vértice de origem não-visitado ainda (um vértice não atingível por s).

Algoritmo 4: Busca em profundidade.

```
    Input  : um grafo G = (V, E), vértice de origem s ∈ V
    // configurando todos os vértices
 1  C_v ← false ∀ v ∈ V
 2  T_v ← ∞ ∀ v ∈ V
 3  A_v ← null ∀ v ∈ V
    // configurando o vértice de origem
 4  C_s ← true
 5  tempo ← 0
    // preparando fila de visitas
 6  S ← Pilha()
 7  S.push(s)
    // propagação das visitas
 8  while S.empty() = false do
 9      tempo ← tempo +1
10      u ← S.pop()
11      T_u ← tempo
12      foreach v ∈ N(u) do
13          if C_v = false then
14              C_v ← true
15              A_v ← u
16              S.push(v)
17  return (C, T, A)
```

É mais comum encontrar a Busca em Profundidade através de um algoritmo recursivo na literatura. Nos Algoritmos 5 e 6, é possível visualizar esta versão da busca. O primeiro algoritmo corresponde a preparação das estruturas de dados e chamada inicial ao método de visita. O segundo corresponde ao método de visita, que efetiva a busca em profundidade. Note que no lugar da pilha, a recursividade controla a sequência de visitas aos vértices.

Algoritmo 5: Algoritmo principal da chamada da busca em profundidade.

Input : um grafo dirigido não ponderado $G = (V, E)$, um vértice de origem $s \in V$

```
// Configurando todos os vértices
1 C_v ← false ∀ v ∈ V
2 T_v ← ∞ ∀ v ∈ V
3 F_v ← ∞ ∀ v ∈ V
4 A_v ← null ∀ v ∈ V
// configurando o tempo de início
5 tempo ← 0
6 VisitaBuscaProfundidade(G, s, C, T, A, F, tempo)// chamar Algoritmo 6
7 return (C, T, A, F)
```

Algoritmo 6: VisitaBuscaProfundidade.

Input : um grafo $G = (V, E)$, vértice de origem $v \in V$, e os vetores C, T, A e F, e uma variável *tempo* $\in \mathbb{Z}^+_*$

```
1 C_v ← true
2 tempo ← tempo +1
3 T_v ← tempo
4 foreach u ∈ N⁺(v) do
5     if C_u = false then
6         A_u ← v
7         DFS-Visit(G, u, C, T, A, F, tempo)
8 tempo ← tempo +1
9 F_v ← tempo
```

3.2.1 Complexidade da Busca em Profundidade

Da mesma maneira que a complexidade da busca em largura, a busca em profundidade possui complexidade $O(|V| + |E|)$. As operações da pilha resultariam tempo $O(|V|)$. Muitos arestas/arcos incidem em vértices já visitados, então inclui-se na complexidade de uma DFS a varredura de todas as adjacências, que demandaria $\Theta(|E|)$.

CAPÍTULO 4

Caminhos e Ciclos

Este capítulo tem o objetivo de introduzir problemas importantes envolvendo caminhos e ciclos. Dois problemas clássicos são definidos e algoritmos são apresentados.

Antes de iniciar a abordagem de problemas e algoritmos, é importante entender o que é um caminho e um ciclo, para estabelecer suas diferenças no contexto de grafos. Um caminho[1] é uma sequência de vértices $\langle v_1, v_2, \ldots v_n \rangle$ conectados por uma aresta ou arco. Gross e Yellen (2006) definem um caminho como um grafo com dois vértices com grau 1 e os demais vértices com grau 2, formando uma estrutura linear. Um ciclo (ou circuito)[2] é uma cadeia fechada de vértices $\langle v_1, v_2, \ldots, v_n, v_1 \rangle$ onde cada par consecutivo é conectado por uma aresta ou arco. É como um caminho com o fim e o início conectados.

4.1 Caminhos e Ciclos Eulerianos

Dois problemas são apresentados a seguir. Eles estão relacionados a encontrar percursos que passem por todas as arestas em grafos não-dirigidos. Há versões de problemas envolvendo percursos Eulerianos para grafos dirigidos, mas não será o foco deste curso.

---

[1] Em inglês, chamado de *path*.
[2] Em inglês, chamado de *cycle* ou *circuit*.

O problema do **Caminho (ou Trilha) Euleriano** pode ser definido como: encontrar um caminho $p = \langle v_1, v_2, \ldots, v_k \rangle$ em um grafo não-orientado $G = (V, E)$, que passe por todas as arestas de G, no qual nenhuma aresta é repetida e $v_1 \neq v_k$ ou $v_1 = v_k$.

O problema do **Ciclo Euleriano** pode ser definido como: encontrar um ciclo $p = \langle v_1, v_2, \ldots, v_k, v_1 \rangle$ em um grafo não-orientado $G = (V, E)$, que passe por todas as arestas de G, no qual nenhuma aresta é repetida.

Em um problema de decisão relacionado a caminhos ou ciclos eulerianos, deseja-se saber se há um caminho ou um ciclo euleriano no grafo de entrada. Um grafo é dito Euleriano se possui um ciclo Euleriano. Se houver um caminho Euleriano, mas não um ciclo Euleriano, diz-se que o grafo é semi-Euleriano.

Curiosidade

Os problemas de caminho e ciclo Euleriano surgiram com o conhecido problema das Sete Pontes de Königsberg por Euler em 1736. O problema consistia em atravessar as todas as sete pontes da cidade de Königsberg da Prussia (hoje Kaliningrado na Rússia) sem repetí-las.

Observando um mapa antigo das sete pontes, você consegue determinar o caminho Euleriano?

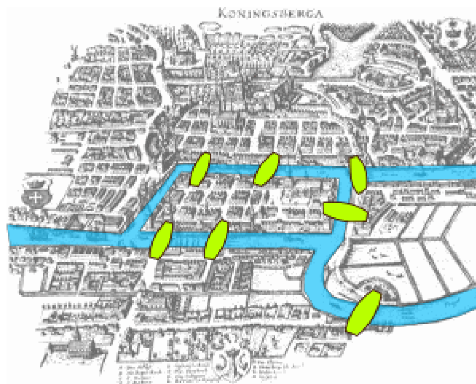


Figura 6 – Mapa das sete pontes de Königsberg na época de in Euler (GIUSCA, 2005).

4.1.1 Algoritmo de Hierholzer

O algoritmo de Hierholzer (Algoritmo 7) foi publicado em artigo datado de 1873, como publicação póstuma a seu criador, Carl Hierholzer (falecido em 1871). Através desse trabalho, Carl Hierholzer provou que o que Leonhard Euler tinha razão ao conjecturar que

"Todo grafo conectado possui um Ciclo Euleriano se e somente se todos os seus vértices possuem grau par."[3]

. A partir dessa definição, uma dúvida pode permanecer: e sobre um grafo com vértice(s) de grau zero, não teriam Ciclo Euleriano? A resposta mais adequada é teriam. Aceitamos que essa definição de conectado, no contexto de Ciclo Euleriano, desconsidera os vértices de grau zero fora da única componente conectada do grafo. O algoritmo proposto identifica o ciclo Euleriano em tempo $O(|E|)$.

[3] Será que podemos estender uma propriedade semelhante para Caminhos Eulerianos?

Algoritmo 7: Algoritmo de Hierholzer.

```
Input   :um grafo não-orientado G = (V, E)
   // Sem arestas, retornar o ciclo euleriano vazio.
1  if |E| = {} then
2      return (true, ⟨⟩ )
   // Caso existam arestas, seguimos a busca pelo ciclo.
3  C_{u,v} ← 0   ∀{u, v} ∈ V
4  foreach {u, v} ∈ E do
5      C_{u,v} ← C_{u,v} + 1
6  v ← selecionar um v ∈ V arbitrariamente, que esteja conectado a uma aresta.
   // "buscarSubciclo" invoca o Algoritmo 8
7  (r, Ciclo) ← buscarSubciclo(G, v, C)
8  if r = false then
9      return (false,null)
10 else
11     if ∃{u, v} ∈ E : C_{u,v} > 0 then
12         return (false,null)
13     else
14         return (true, Ciclo)
```

Algoritmo 8: Algoritmo de Auxiliar “$buscarSubciclo$”.

```
Input  : um grafo não-orientado G = (V, E), um vértice v ∈ V, o vetor de arestas visitadas C
1  Ciclo ← (v)
2  t ← v
3  repeat
       // Só prossegue se existir uma aresta não-visitada conectada a Ciclo.
4      if ∄u ∈ N(v) : C_{u,v} > 0 then
5          return (false,null)
6      else
7          {v, u} ← selecionar uma aresta {v, u} ∈ E, para qualquer u, tal que C_{v,u} > 0
8          C_{v,u} ← C_{v,u} − 1
9          v ← u
           // Adiciona o vértice v ao final do ciclo.
10         Ciclo ← Ciclo · (v)
11 until v = t
   /* Para todo vértice x no Ciclo que tenha uma aresta adjacente não
      visitada.                                                        */
12 foreach x ∈ {u ∈ Ciclo : ∃{u, w} ∈ {e ∈ E : C_e > 0}} do
13     (r, Ciclo′) ← buscarSubciclo(G, x, C)
14     if r = false then
15         return (false,null)
16     Assumindo que Ciclo = ⟨v1, v2, ..., x, ..., v1⟩ e Ciclo′ = ⟨x, u1, u2, ..., uk, x⟩, alterar
        Ciclo para Ciclo = ⟨v1, v2, ..., x, u1, u2, ..., uk, x, ..., v1⟩, ou seja, inserir o Ciclo′ no
        lugar da posição de x em Ciclo.
17 return (true, Ciclo)
```

Explique porque a complexidade de Algoritmo de Hierholzer é de $O(|E|)$.

Lema 4.1.1. *Seja $G = (V, E)$ um grafo não-dirigido no qual todas as arestas pertençam à única componente conectada e todos os vértices possuem grau par, é sempre possível*

encontrar obter um ciclo em G que não repita arestas.

Prova: A prova é realizada por contradição. Imagine que comecemos a criar um ciclo p por um vértice arbitrário $v_1 \in V$, conectado a uma aresta em G. Chamaremos de aresta marcada a que já foi colocada no ciclo e d'_v a quantidade de arestas conectadas a v que ainda não estão marcadas.

Por contradição, tentamos demonstrar que não é possível obter um ciclo em G, iremos tentar criar um caminho que nunca encontrará novamente v_1. Ao iniciar por v_1, selecionamos arbitrariamente um vértice adjacente, chamado aqui de u, e o adicionamos a p. Desse modo, a aresta $\{v_1, u\}$ é considerada como marcada. Nesse momento, d'_{v_1} e d'_u se tornam números ímpares. Como o grau de u é par e maior que 0, ele terá alguma aresta não marcada que permitirá que o caminho continue a ser construído. Repete-se o processo até que não seja mais possível encontrar arestas não marcadas, adicionando um novo vértice ao caminho que esteja conectado ao último vértice adicionado em p utilizando uma aresta não marcadas. Note que é impossível não retornar a v_1 e estabelecer um ciclo, pois todo vértice recém adicionado terá uma aresta não marcada para adicionar alguém ao caminho, a não se que se tenha retornado a v_1, portanto formando um ciclo, o que é uma contradição. Dessa forma, a prova por contradição está completa. ■

Lema 4.1.2. *Seja $G = (V, E)$ um grafo não-dirigido com dois ciclos p_1 e p_2 que não compartilham arestas, eles podem ser unidos para formar um único ciclo, caso compartilhem um vértice.*

Prova: A prova é dada por construção. Considere $p_1 = \langle u_1, u_2, \ldots, u_1 \rangle$ e $p_2 = \langle v_1, v_2, \ldots, v_1 \rangle$. Para construir um único ciclo:

1. considere que o vértice x que seja comum aos dois ciclos;

2. conside $p_1 = \langle u_1, u_2, \ldots, x, \ldots, u_1 \rangle$ e $p_2 = \langle v_1, v_2, \ldots, x, \ldots, v_1 \rangle$;

3. formatar o ciclo p_2 para iniciar e começar em x, tornando-se desse modo $p_2 = \langle x, \ldots, v_1, v_2, \ldots, x\rangle$;

4. copiar p_1 para r, o ciclo resultante, desse modo $r = \langle u_1, u_2, \ldots, x, \ldots, u_1\rangle$;

5. substituir o vértice x em r pelo ciclo na ordem em p_2, obtendo $p_1 = \langle u_1, u_2, \ldots, x, \ldots, v_1, v_2, \ldots, x, \ldots, u_1\rangle$.

■

Lema 4.1.3. *Seja $G = (V, E)$ um grafo não-dirigido no qual todos os vértices possuem grau par, remover as arestas que compõem qualquer ciclo em G, todos os vértices continuarão com grau par.*

Prova: A prova é dada por contradição. Considere o ciclo $p = \langle v_1, v_2, \ldots, v_k, v_1\rangle$. Os vértices que sofrerão redução de grau serão aqueles que pertencem ao ciclo, portanto, por contradição, devemos supor que ao menos um vértice do ciclo terá grau ímpar. Isso é impossível, pois ao remover as arestas que pertença ao ciclo, os vértices pertencentes ao ciclo sofrerão sempre o decréscimo de dois em seus graus (um para aresta que entra e outro para a que sai). ■

Teorema 4.1.4. *Um grafo não-orientado $G = (V, E)$ é (ou possui um ciclo) Euleriano se e somente se em G cada vértice tem um grau par e todas as arestas pertençam a uma única componente conectada no grafo.*

Prova: A prova será realizada por contradição. Considerando o Lema 4.1.1 há ao menos um ciclo. Se houver mais de um ciclo, e se eles se conectam em vértices, é possível converter esses ciclos em um apenas pelo Lema 4.1.2.

Então devemos imaginar arestas que não pertençam a um ciclo presente no grafo. Ao remover os ciclos formados, ainda assim, teremos vértices com grau par pelo Lema 4.1.3. Essas arestas podem estar em uma única ou em várias componentes conectadas. Sabe-se que essa ou essas componentes terão vértices de grau par e cada componente poderá ter um ciclo (Lema 4.1.1). No caso de mais de uma componente conectada ter

sido gerada, significa que o ciclo removido conectava as componentes. Se cada uma delas tiver um ciclo, então é possível obter um ciclo de cada uma (pelo Lema 4.1.2), logo não sobrando arestas sem ciclo.

Analisando cada componente (ou a única que tenha sobra pela remoção), há de se encontrar a aresta que não está no ciclo. No entanto, cada componente conectada possui um ciclo (Lema 4.1.1) e a remoção desse ciclo gerará componente ou componentes com vértices de grau par (Lema 4.1.3) até que o grau seja igual a zero ou não haja mais componentes. Desse modo, é impossível encontrar tal aresta e completa-se a prova. ■

4.2 Caminhos e Ciclos Hamiltonianos

Ciclos ou caminhos Hamiltonianos são aqueles que percorrem todos os vértices de um grafo apenas uma vez. Mais especificamente para um ciclo Hamiltoniano, o início e o fim terminam no mesmo vértice. O nome Hamiltoniano vem de William Rowan Hamilton, o inventor de um jogo que desafia a buscar um ciclo pelas arestas de dodecaedro (figura tridimensional de 12 faces).

Um grafo é dito Hamiltoniano se possui um ciclo Hamiltoniano.

Há $|V|!$ diferentes sequências de vértices que podem ser caminhos Hamiltonianos, então, um algoritmo de força-bruta demanda muito tempo computacional. O problema de decisão para encontrar um caminho ou ciclo Hamiltoniano é considerado NP-Completo.

4.2.1 Caixeiro Viajante

Dado um grafo completo[4] $G = (V, E, w)$ no qual V é o conjunto de vértices, E é o conjunto de arestas e $w : E \rightarrow \mathbb{R}^+$ é a função dos pesos (ou custo ou distâncias), busca-se pelo ciclo Hamiltoniano de menor soma total de peso (menor custo ou distância).

[4] Em um grafo completo, o conjunto de arestas é definido por $E = V \times V$.

Um dos algoritmos mais eficientes para resolvê-lo é o de programação dinâmica Held-Karp (ou Bellman–Held–Karp, no Algoritmo 9). No entanto, o mesmo demanda tempo computacional de $O(2^{|V|}|V|^2)$.

O algoritmo de Bellman-Held-Karp é de programação dinâmica e considera que “cada subcaminho de um caminho de distância mínima é mínimo”. Segue abaixo a recorrência utilizada para a programação dinâmica do algoritmo. Ela tem o objetivo de descobrir o valor do menor caminho que começa em 1 e temina em v passando por todos os vértices que estão em S. Para isso, assumimos que $1, v \notin S$

$$OPT(S,v) = \begin{cases} w(\{1,v\}) & \text{para } |S| = 0, \\ \mathbf{min}_{u \in S, u \neq 1}\{OPT(S - \{u\}, u) + w(\{u,v\})\} & \text{para } |S| > 0. \end{cases}$$

Algoritmo 9: Algoritmo de Bellman-Held-Karp.

Input : um grafo $G = (V, E = V \times V, w)$

1. **for** $k \leftarrow 2$ ***to*** $|V|$ **do**
2. $\quad C(\{k\}, k) \leftarrow w(\{1,k\})$
3. **for** $s \leftarrow 2$ ***to*** $|V| - 1$ **do**
4. $\quad$ **foreach** $S \in \{x \subseteq \{2,3,\ldots,|V|\} : |x| = s\}$ **do**
5. $\quad\quad$ **foreach** $v \in S$ **do**
6. $\quad\quad\quad C(S,v) \leftarrow \mathbf{min}_{u \neq v, u \in S}\{C(S \backslash \{v\}, u) + w(\{u,v\})\}$
7. **return** $\mathbf{min}_{v \in V \backslash \{1\}}\{C(\{2,3,\ldots,|V|\}, v) + w(\{v,1\})\}$

Execute o Algoritmo 9 sobre o grafo $G = (V = \{1,2,3,4\}, E = V \times V, w)$, no qual $w(\{1,2\}) \rightarrow 10$, $w(\{1,3\}) \rightarrow 15$, $w(\{1,4\}) \rightarrow 20$, $w(\{2,3\}) \rightarrow 35$, $w(\{2,4\}) \rightarrow 25$ e $w(\{3,4\}) \rightarrow 30$.

A resposta deve ser 80.

CAPÍTULO 5

Caminhos de Custo Mínimo

Em um problema de Caminho Mínimo, há um grafo ponderado orientado ou não $G = (V, E, w)$, onde V é o conjunto de vértices, E é o conjunto de arcos ou arestas, e $w : E \to \mathbb{R}$ é a função que representa o peso entre dois vértices (distância ou custo das arestas). Para um caminho $p = \langle v_1, v_2, \ldots, v_k \rangle$ seu peso é dado por $w(p) = \sum_{i=2}^{k} w\big((v_{i-1}, v_i)\big)$ (CORMEN et al., 2012).

O peso de um caminho mínimo de u a v é dado por

$$\delta(u, v) = \begin{cases} \mathbf{min}\{w(p) : u \overset{p}{\leadsto} v\}, & \text{se há um caminho de } u \text{ para } v \text{ ,} \\ \infty, & \text{caso contrário.} \end{cases} \tag{5.1}$$

Há algumas variantes para os problemas de caminho mínimo (CORMEN et al., 2012):

- Problema de caminhos mínimos de fonte única: dado um grafo ponderado $G = (V, E, w)$ e um vértice de origem $s \in V$, encontrar o caminho de custo $\delta(s, v)$ para todo o $v \in V$;

- Problema de caminhos mínimos para um destino:dado um grafo ponderado $G = (V, E, w)$, um vértice de destino $t \in V$, determinar o caminho de custo $\delta(v, t)$ para todo o $v \in V$;

- Problema de caminhos mínimos para um par: dado um grafo ponderado $G = (V, E, w)$, um vértice de origem $s \in V$ e um vértice de destino t, determinar o caminho de custo $\delta(s, t)$;

- Problema de caminhos mínimos para todos os pares: dado um grafo ponderado $G = (V, E, w)$, encontrar o caminho de curso $\delta(u, v)$ para todo o par $u, v \in V$.

Pesos Negativos

Os problemas de caminho mínimo geralmente operam sem erros em grafos com pesos negativos. Uma exceção a isso é quando há um ciclo com peso negativo. Nesse caso, nunca haverá um peso definido, pois é sempre possível diminuir o peso total do caminho percorrendo o ciclo mais uma vez.

Inicialização e Relaxamento

Os Algoritmos 10 e 11 são utilizados em diversos algoritmos de resolução de caminhos mínimos. A estrutura de dados D é referente a estimativa de caminho que será obtida ao longo da execução de um caminho mínimo para cada vértice $v \in V$. A estrutura de dados A é utilizada para identificar o vértice anterior em cada caminho mínimo para um vértice $v \in V$.

Algoritmo 10: Inicialização de G.

Input :um grafo $G = (V, E, w)$, um vértice de origem $s \in V$

```
// inicialização
```

1 $D_v \leftarrow \infty \; \forall v \in V$

2 $A_v \leftarrow$ **null** $\forall v \in V$

3 $D_s \leftarrow 0$

4 **return** (D, A)

Algoritmo 11: Relaxamento de v.

Input :um grafo $G = (V, E, w)$[a], $(u, v) \in E$, A, D

1 **if** $D_v > D_u + w\big((u, v)\big)$ **then**
2 $\quad D_v \leftarrow D_u + w\big((u, v)\big)$
3 $\quad A_v \leftarrow u$

[a] Para o caso de G ser não-dirigido, deve-se realizar o relaxamento em (u, v) e (v, u) para uma aresta $\{u, v\} \in E$, ou seja, deve-se realizar o relaxamento nos dois sentidos.

5.1 Propriedades de Caminhos Mínimos

5.1.1 Propriedade de subcaminhos de caminhos mínimos o são (esses subcaminhos também são caminhos mínimos)

Lema 5.1.1. *Dado um grafo ponderado $G = (V, E, w)$, um caminho mínimo entre v_1 e v_k $p = \langle v_1, v_2, \ldots, v_k \rangle$, suponha que, para quaisquer i e j, $1 \le i \le j \le k$, todo o subcaminho de p chamado de $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ é um caminho mínimo de v_i a v_j.*

Prova: Se o caminho p for decomposto em $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, têm-se $w(p) = w(p_{0j}) + w(p_{ij}) + w(p_{jk})$. Suponha que exista um caminho p'_{ij} de v_i a v_j com peso $w(p'_{ij}) < w(p_{ij})$. Então, $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ é um caminho de v_1 a v_k cujo o peso $w(p) = w(p_{0j}) + w(p'_{ij}) + w(p_{jk})$ é menor do que $w(p)$, o que contradiz a hipótese de que p seja um caminho mínimo de v_1 a v_k. ■

5.1.2 Propriedade de desigualdade triangular

Lema 5.1.2. *Seja $G = (V, E, w)$ um grafo ponderado e $s \in V$ um vértice de origem, então para todas as arcos/arestas $(u, v) \in E$ têm-se*

$$\delta(s, v) \le \delta(s, u) + w\big((u, v)\big). \tag{5.2}$$

Prova: Suponha que p seja um caminho entre s e v. Então, p não tem peso maior do que qualquer outro caminho de s a v. Especificamente, p não possui peso maior que o caminho de s até o vértice u que utiliza a aresta/arco (u, v) para atingir o destino v. ■

5.1.3 Propriedade de limite superior

Lema 5.1.3. *Seja $G = (V, E, w)$ um grafo ponderado dirigido ou não com a função de peso $w : E \to \mathbb{R}$. Seja $s \in V$ o vértice de origem, considera-se também o grafo G inicializado (Algoritmo 10). Então, $D_v \geq \delta(s, v)$ para todo $v \in V$, e esse invariante é mantido para qualquer sequência de etapas de relaxamentos em G (Algoritmo 11). Além disso, tão logo D_v alcance seu limite inferior $\delta(s, v)$, nunca mais se altera.*

Prova: Prova-se que o invariante $D_v \geq \delta(s, v)$ para todo o vértice $v \in V$ por indução em relação ao número de etapas de relaxamento.

Para a base da indução, $D_v \geq \delta(s, v)$ é verdadeiro após a inicialização (Algoritmo 10), pois esse procedimento define que $D_v = \infty$ para todo $v \in V \setminus \{s\}$, ou seja, $D_v \geq \delta(s, v)$ mesmo que v seja inatingível em um caminho mínimo a partir de s. Nesse momento $D_s = 0 \geq \delta(s, s)$.

Para o passo da indução, considere o relaxamento de uma aresta/arco (u, v). Pela hipótese de indução, $D_x \geq \delta(s, x)$ para todo o $x \in V$ antes do relaxamento. O único valor de D que pode mudar é D_v. Se ele mudar, têm-se

$$
\begin{aligned}
D_v &= D_u + w\big((u, v)\big) \\
&\geq \delta(s, u) + w\big((u, v)\big) \text{ (pela hipótese da indução)} \\
&\geq \delta(s, v) \text{ (pela desigualdade triangular, Lema 5.1.2)}
\end{aligned}
\tag{5.3}
$$

e, portanto o invariante é mantido.

Para demonstrar que D_v não se altera depois que $D_v = \delta(s, v)$, por ter alcaçado seu limite inferior, D_v não pode diminuir porque $D_v \geq \delta(s, v)$ e não pode aumentar porque o relaxamento não aumenta valores de D. ■

5.1.4 Propriedade de inexistência de caminho

Corolário 5.1.4. *Supõe-se que, em um grafo $G = (V, E, w)$ ponderado dirigido ou não, nenhum caminho conecte o vértice de origem $s \in V$ a um vértice $v \in V$. Então, depois que o grafo G é inicializado (Algoritmo 10), temos $D_v = \delta(s, v) = \infty$ e essa desigualdade é mantida como um invariante para qualquer sequência de etapas de relaxamento (Algoritmo 11) nas arestas de G;*

Prova: Pela propriedade de limite superior (Lema 5.1.3), têm-se sempre $\infty = \delta(s, v) \leq D_v$, portanto $D_v = \infty = \delta(s, v)$. ■

Propriedade de limite superior de relaxamento de aresta

Lema 5.1.5. *Seja $G = (V, E, w)$ um grafo ponderado dirigido ou não com a função de peso $w : E \to \mathbb{R}$. Seja $(u, v) \in E$ uma aresta/arco. Depois de relaxamento de caminho (Algoritmo 11) sobre (u, v), pode-se afirmar que $D_v \leq D_u + w\big((u, v)\big)$.*

Prova: Há duas possibilidades. Se antes de relaxar a aresta (u, v):

- $D_v > D_u + w\big((u, v)\big)$: então D_v passa a ser $D_v = D_u + w\big((u, v)\big)$;
- $D_v \leq D_u + w\big((u, v)\big)$: então não há alteração em D_v e D_u.

■

5.1.5 Propriedade de convergência

Lema 5.1.6. *Seja $G = (V, E, w)$ um grafo ponderado dirigido ou não, $s \in V$ um vértice de origem e $s \rightsquigarrow u \to v$ um caminho mínimo de s a v em G. Suponha que G seja inicializado (Algoritmo 10) e depois uma sequência de etapas de relaxamento (Algoritmo 11) é executado para todas as arestas/arcos de G. Se $D_u = \delta(s, u)$ em qualquer tempo anterior da chamada, então $D_u = \delta(s, u)$ igual em toda a chamada.*

Prova: Pela propriedade do limite superior (Lema 5.1.3), se $D_u = \delta(s, u)$ em algum momento antes do relaxamento da aresta/arco (u, v), então essa igualdade se mantém válida a partir de sua definição. Em particular, após o relaxamento (Algoritmo 11) da aresta/arco (u, v), têm-se:

$$\begin{aligned} D_v &\leq D_u + w\big((u,v)\big) \text{ pelo Lema 5.1.5)} \\ &= \delta(s,u) + w\big((u,v)\big) \\ &= \delta(s,v) \text{ pelo Lema 5.1.1.)} \end{aligned} \tag{5.4}$$

Pela propriedade do limite superior (Lema 5.1.3) $D_v \geq \delta(s, v)$, da qual concluímos que $D_v = \delta(s, v)$, e essa igualdade é mantida daí em diante. ■

5.1.6 Propriedade de relaxamento de caminho

Lema 5.1.7. *Seja $G = (V, E, w)$ um grafo ponderado dirigido ou não e $s \in V$ um vértice de origem. Considere qualquer caminho mínimo $p = \langle v_1, v_2, \ldots v_k \rangle$ de $s = v_1$ a v_k. Se G é inicializado (Algoritmo 10) e depois ocorre uma sequência de etapas de relaxamento (Algoritmo 11) que inclui, pela ordem, relaxar as arestas/arcos $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$, então $D_k = \delta(s, v_k)$ depois desses relaxamentos e todas as vezes daí em diante. Essa propriedade se mantém válida, não importa quais outros relaxamentos forem realizados.*

Prova: Esta prova é realizada por indução, na qual têm-se $D_{v_i} = \delta(s, v_i)$ depois que o i-ésimo aresta/arco do caminho p é relaxado.

Para a base, $i = 1$ antes que quaisquer arestas/arcos em p sejam relaxados. Têm-se $D_{v_1} = D_s = 0 = \delta(s, s)$ pela inicialização. Pela propriedade do limite superior (Lema 5.1.3) o valor D_s nunca se altera depois da inicialização.

Pelo passo da indução, supõe-se que $v_{i-1} = \delta(s, v_{i-1})$ e examina-se o que acontece quando se relaxa a aresta (v_{i-1}, v_i). Pela propriedade de convergência (Lema 5.1.6), após o relaxamento dessa aresta, têm-se $D_v = \delta(s, v_i)$ e essa igualdade é mantida todas as vezes depois disso. ■

5.1.7 Propriedade de relaxamento e árvores de caminho mínimo

Lema 5.1.8. *Seja* $G = (V, E, w)$ *um grafo ponderado dirigido ou não e* $s in V$ *um vértice de origem, suponha que G não possua um ciclo de peso negativo que possa ser atingido por* s*. Então, depois que o grafo G é inicializado (Algoritmo 10), o subgrafo dos predecessores* $G_\pi = (V_\pi, E_\pi)$ *forma uma árvore enraizada em* s*, e qualquer sequência de etapas de relaxamento em arestas em G (Algoritmo 11) mantém essa propriedade invariante.*

Prova: Inicialmentem o único vértice em G_π é o vértice s, e o lema é trivialmente verdade. Considere um subgrafo dos predecessores G_π que surja depois de uma sequência de etapas de relavamento.

Primeiro, prova-se que o subgrafo é acíclico. Suponha por contradição que alguma etapa de relaxamento cria um ciclo no grafo G_π. Seja $c = \langle v_1, v_2, \ldots, v_k \rangle$ o ciclo onde $v_1 = v_k$. Então, $A_{v_i} = v_{i-1}$ para $i = 1, 2, \ldots, k$ e, sem prejuízo de generalidade, pode-se supor que o relaxamento de arestas (v_{k-1}, v_k) criou o ciclo em G_π. Afirma-se que todos os vértices do ciclo c podem ser atingidos por s, pois cada um tem um predecessor não nulo (**null**). Portanto, uma estimativa de caminho mínimo fora atribuída a cada vértice em c quando um valor atribuídoo à A_v não foi igual a **null**. Pela propriedade do limite superior (Lema 5.1.3), cada vértice no ciclo c tem um peso de caminho mínimo infinito, o que implica que ele pode ser atingido por s.

Examina-se as estimativas de caminhos mínimos em c imediatamente antes de chamar o procedimento de relaxamento (Algoritmo 11) passando os parâmetros $G, (v_{k-1}, v_k), A, D$ e mostra-se que c é um ciclo de peso negativo, contradizendo a hipótese que G não possui um ciclo negativo que possa ser atingido por s. Imediatamente antes da chamada, têm0se $A_{v_i} = v_{i-1}$ para $i = 2, 3, \ldots, k-1$. Assim, para $i = 2, 3, \ldots, k-1$, a última atualização para D_{v_i} foi realizada pela atribuição $D_{v_i} \leftarrow D_{v_{i-1}} + w\big((v_{i-1}, v_i)\big)$. Se $D_{v_{i-1}}$ mudou desde então, ela diminuiu. Por essa razão, imediatamente antes da chamada de relaxamento, têm-se

$$D_{v_i} \geq D_{v_{i-1}} + w\big((v_{i-1}, v_i)\big) \forall i \in \{2, 3, \ldots, k-1\} \tag{5.5}$$

Como A_k é alterado pela chamada, imediatamente antes têm-se também a desigualdade estrita

$$D_{v_k} > D_{v_{k-1}} + w\big((v_{k-1}, v_k)\big). \tag{5.6}$$

Somando essa desigualdade estrita com as $k-1$ desigualdades (Equação (5.5)), obtêm-se a soma das estimativas dos caminhos mínimos em torno do ciclo c:

$$\sum_{i=2}^{k} D_{v_i} > \sum_{i=2}^{k} \Big(D_{v_{i-1}} + w\big((v_{i-1}, v_i)\big)\Big) = \sum_{i=2}^{k} D_{v_{i-1}} + \sum_{i=2}^{k} w\big((v_{i-1}, v_i)\big). \tag{5.7}$$

Mas,

$$\sum_{i=2}^{k} D_{v_i} = \sum_{i=2}^{k} D_{v_{i-1}}, \tag{5.8}$$

já que cada vértice no ciclo c aparece exatamente uma vez em cada somatório. Esa desigualdade implica

$$0 > \sum_{i=2}^{k} w\big((v_{i-1}, v_i)\big). \tag{5.9}$$

Assim a soma dos pesos no ciclo c é negativa, o que dá acontradição desejada.

Agora, provamos que G_π é acíclico. Para mostrar que ele forma uma árvore enraizada em s, basta provar que há um único caminho simples de s a v em G_π para cada $v \in V_\pi$.

Primeiro, deve-se mostrar que existe um caminho de s a cada vértice em $v \in V_\pi$. Os vértices em V_π são os que têm os valores A não **null**, e o vértice s. Aqui a ideia é provar a indução que existe em um caminho de s para todos os vértices em V_π.

Para concluir a prova do lema, deve-se mostrar agora que, para qualquer vértice $v \in V_\pi$, o grafo G_π contém no máximo um caminho simples de s a v. Suponha o contrário: que existam dois caminhos simples de s a algum outro vértice $p_1\langle s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$, e $p_2\langle s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$ onde $x \neq y$. Mas então $A_z = x$ e $A_z = y$ o que implica uma contradição, pois $x = y$. Conclui-se que G_π contém um caminho simples único de s a v e G_π forma uma árvore enraizada em s. ■

5.1.8 Propriedade de subgrafo dos predecessores

Lema 5.1.9. *Seja $G = (V, E, w)$ um grafo ponderado orientado ou não e um vértice de origem $s \in V$. Suponha que G não possua um ciclo de peso negativo que possa ser atingido*

por s. Chama-se de inicialização o procedimento do inicializado Algoritmo 10 e depois executar qualquer sequência de etapas de relaxamento de arestas de G (Algoritmo 11) que produza $D_v = \delta(s,v)$ para todo $v \in V$. Então, o subgrafo predecessor $G_\pi(V_\pi, E_\pi)$ é uma árvore de caminhos mínimos com uma raiz em s.

Prova: Para ilustrar a primeira propriedade, deve-se mostrar V_π é o conjunto de vértices atingidos por s. Por definição, um peso de caminho mínimo $\delta(s,v)$ é finito sse v pode ser alcançado por s. Isso implica que os vértices atingidos por s possuem peso de caminho finito. Porém, um vértice $v \in V \backslash \{s\}$ recebeu um valor finito para D_v sse $A_v \neq$**null**. Assim, os vértices em V_π são exatamente aqueles que podem ser alcançados por s.

O Lema 5.1.8 define que após a inicialização, G_π possui raiz em s e assim permanece mesmo depois de sucessivas etapas de relaxamento.

Agora, prova-se que para todo vértice em $v \in V_\pi$, o único caminho simples em G_π de s a v é o caminho mínimo de s a v em G. Seja $p = \langle v_1, v_2, \ldots v_k \rangle$, onde $v_1 = s$ e $v_k = v$. Para $i = 2, 3, \ldots, k$, temos $D_v = \delta(s, v_i)$ e também $D_v \geq D_{v_{i-1}} + w\big((v_{i-1}, v_i)\big)$, do que concluímos $w\big((v_{i-1}, v_i)\big) \leq \delta(s, v_i) - \delta(s, v_{i-1})$. A soma dos pesos ao longo de p produz

$$
\begin{aligned}
w(p) &= \sum_{i=2}^{k} w\big((v_{i-1}, v_i)\big) \\
&\leq \sum_{i=2}^{k} \delta(s, v_i) - \delta(s, v_{i-1}) \\
&= \delta(s, v_k) - \delta(s, v_0) \\
&= \delta(s, v_k)
\end{aligned}
\tag{5.10}
$$

Assim $w(p) \leq \delta(s, v_k)$. Visto que $\delta(s, v_k)$ é um limite inferior para o peso de qualquer caminho de s a v_k, conclui-se que $w(p) = \delta(s, v_k)$. Deste modo, p é um caminho mínimo de s a $v = v_k$.

■

5.2 Bellman-Ford

O algoritmo de Bellman-Ford resolve o problema de caminhos minimos de uma única fonte. Um pseudo-código está representado no Algoritmo 12. Como entrada para o algoritmo deve-se determinar um grafo ponderado orientado ou não $G = (V, E, w)$, onde V é o conjunto de vértices, E o conjunto de arestas/arcos e $w : E \to \mathbb{R}$, e um vértice de origem $s \in V$. O algoritmo devolve um valor booleano **false** quando for encontrado um ciclo de peso negativo em G. Caso contrário, retorna **true**, o antecessor de cada vértice v no caminho mínimo em A_v e a peso $\delta(s, v)$ em D_v.

O algoritmo vai progressivamente diminuindo a estimativa de peso do caminho de s a $v \in V$ até que se obtenha o caminho mínimo e $D_v = \delta(s, v)$ para todo $v \in V$.

Algoritmo 12: Algoritmo de Bellman-Ford.

```
Input :um grafo G = (V, E, w), um vértice de origem s ∈ V
   // inicialização
1  D_v ← ∞ ∀ v ∈ V
2  A_v ← null ∀ v ∈ V
3  D_s ← 0
4  for i ← 1 to |V| − 1 do
5      foreach (u, v) ∈ E do
           // relaxamento
6          if D_v > D_u + w((u, v)) then
7              D_v ← D_u + w((u, v))
8              A_v ← u
9  foreach (u, v) ∈ E do
10     if D_v > D_u + w((u, v)) then
11         return (false,null,null)
12 return (true, D, A)
```

5.2.1 Complexidade de Bellman-Ford

Quanto a complexidade computacional em tempo computacional de Bellman-Ford, observando as primeiras instruções, têm-se a inicialização que demanda $\Theta(|V|)$ pois as estruturas são inicializadas para cada vértice. A partir do primeiro conjunto de laços de repetição, há o laço mais externo que repete $|V|-1$ vezes. Para cada repetição desse laço, passa-se por cada aresta em E, logo esse primeiro conjunto de laços dita $(|V|-1)|E|$ execuções da comparação na linha 6. O último laço de repetição, faz ao máximo $|E|$ verificações da comparação na linha 10. Então, o algoritmo de Bellman-Ford é executado no tempo computacional de $O(|V||E|)$.

5.2.2 Corretude de Bellman-Ford

Lema 5.2.1. *Seja $G=(V,E,w)$ um grafo ponderado e um vértice de origem $s \in V$, suponha que G não possua nenhum ciclo de peso negativo que possa ser alcançado por s. Então, depois de executar as $|V|-1$ iterações nas linhas 4 a 8 com o algoritmo de Bellman-Ford (Algoritmo 12), têm-se $D_v = \delta(s,v)$ para todo $v \in V$.*

Prova: Prova-se o esse lema através da propriedade de relaxamento de caminho (Lema 5.1.7). Considera-se que qualquer vértice v possa ser atingido por s e seja $p = \langle v_1, v_2, \ldots, v_k \rangle$ um caminho mínimo de s a v, no qual $v_1 = s$ e $v_k = v$. Como caminhos mínimos são simples, p tem no máximo $|V|-1$ arestas/arcos, sendo $k \leq |V|-1$. Cada uma das $|V|-1$ iterações do laço da linha 4 relaxa todas as $|E|$ arestas/arcos. Entre as arestas relaxadas na i-ésima iteração, para $i = 1,2,\ldots k$, está (v_{i-1}, v_i). Então, pela propriedade de relaxamento de caminho $D_v = D_{v_k} = \delta(s, v_k) = \delta(s,v)$. ■

Corolário 5.2.2. *Seja $G=(V,E,w)$ um grafo ponderado dirigido ou não e $s \in V$ o vértice de origem, supõe-se que G não tenha nenhum ciclo negativo que possa ser atingido por s. Então, para cada vértice $v \in V$, existe um caminho de s a v sse o algoritmo de Bellman-Ford termina com $D_v < \infty$ quando é executado para G e s.*

Prova: Se $v \in V$ pode ser atingido por s, então existe uma aresta/arco (u, v). Então, $\delta(s, v) < \infty$ através da propriedade de convergência (Lema 5.1.6). ■

Teorema 5.2.3. *Considera-se o algoritmo de Bellman-Ford (Algoritmo 12) executado para um grafo $G = (V, E, w)$ e o vértice de origem $s \in V$. Se G não contém nenhum ciclo de custo negativo, que pode ser alcançado de s, então o algoritmo retorna **true**, $D_v = \delta(s, v)$ para todo $v \in V$ e o subgrafo predecessor G_π é uma árvore de caminhos mínimos com raiz em s. Se G contém um ciclo de peso negativo que possa ser atingido por s, então o algoritmo retorna **false**.*

Prova: Suponha que o grafo G não tenha um ciclo de peso negativo atingível por s. Primeiro, prova-se que $D_v = \delta(s, v)$ para todo $v \in V$. Se o vértice v pode ser atingido por s, então o Lema 5.2.1 prova essa afirmação. Se v não pode ser atingido por s, a prova decorre da propriedade da inexistência de caminho (Corolário 5.1.4). Portanto, a afirmação está provada. A propriedade de subgrafo dos predecessores (Lema 5.1.9) juntamente com essa última afirmação implica que G_π é uma árvore de caminhos mínimos. Agora, usa-se a afirmação para mostrar que Bellman-Ford retorna **true**. No término, têm-se para todas as arestas/arcos $(u, v) \in E$,

$$\begin{aligned} D_v &= \delta(s, v) \\ &\leq \delta(s, u) + w\big((u, v)\big) \text{ (pela desigualdade triangular – Lema 5.1.2)} \\ &= D_u + w\big((u, v)\big), \end{aligned} \tag{5.11}$$

e, assim, nenhum dos testes na linha 10 serão verdadeiros e Bellamn-Ford retorna **false**. Então, ele retorna **true**.

Agora, suponha que o grafo G contenha o ciclo de peso negativo que possa ser atingido por s. Seja esse ciclo $c = \langle v_1, v_2, \ldots, v_k \rangle$, onde $v_1 = v_k$. Então,

$$\sum_{i=2}^{k} w\big((v_{i-1}, v_i)\big) < 0 \tag{5.12}$$

Considere, por contradição, que o algoritmo de Bellman-Ford retorna **true**. Assim, $D_{v_i} \leq D_{v_{i-1}} + w\big((v_{i-1}, v_i)\big)$ para $i = 2, 3, \ldots, k$. Somando as desigualdades em torno do ciclo c têm-se

$$\sum_{i=2}^{k} D_{v_i} \leq \sum_{i=2}^{k} D_{v_{i-1}} + w\big((v_{i-1}, v_i)\big) = \sum_{i=2}^{k} D_{v_{i-1}} + \sum_{i=2}^{k} w\big((v_{i-1}, v_i)\big). \tag{5.13}$$

Como $v_0 = v_k$, cada vértice em c aparece exatamente apenas uma vez em cada um dos somatórios, portanto

$$\sum_{i=2}^{k} D_{v_i} = \sum_{i=2}^{k} D_{v_{i-1}} \tag{5.14}$$

Além disso, pelo Corolário 5.2.2, D_{v_i} é finito para $i = 2, 3, \ldots, k$. Assim,

$$0 \leq \sum_{i=2}^{k} w\big((v_{i-1}, v_i)\big), \tag{5.15}$$

o que contradiz a desigualdade da Equação (5.12). Conclui-se que o algoritmo de Bellman-Ford retorna **true** se o grafo G não contém nenhum ciclo negativo que possa ser alcançado a partir da fonte e **false** caso contrário.

■

5.3 Dijkstra

O algoritmo de Dijkstra resolve o problema de encontrar um problema de caminho mínimos de fonte única em um grafo $G = (V, E, w)$ ponderados dirigidos ou não. Para esse algoritmo, as arestas/arcos não devem ter pesos negativos. Então, a função de pesos é redefinida como $w : E \to \mathbb{R}^+_*$. A vantagem está em o algoritmo de Dijkstra ser mais eficiente que o do Bellman-Ford se for utilizada uma estrutura de dados adequada.

O algoritmo repetidamente seleciona o vértice de menor custo estimado até então. Quando esse vértice é selecionado, ele não é mais atualizado e sua distância é propagada para suas adjacências. A estrutura de dados C é utilizada no pseudo-código abaixo para definir se um vértice foi visitado (contém **true**) ou não (contém **false**).

Algoritmo 13: Algoritmo de Dijkstra.

Input : um grafo $G = (V, E, w : E \rightarrow \mathbb{R}^+_*)$, um vértice de origem $s \in V$

```
1  D_v ← ∞ ∀ v ∈ V
2  A_v ← null ∀ v ∈ V
3  C_v ← false ∀ v ∈ V
4  D_s ← 0
5  while ∃v ∈ V (C_v = false) do
6      u ← arg min_{v∈V} {D_v | C_v = false}
7      C_u ← true
8      foreach v ∈ N(u) : C_v = false do
9          if D_v > D_u + w((u, v)) then
10             D_v ← D_u + w((u, v))
11             A_v ← u
12 return (D, A)
```

5.3.1 Complexidade de Dijkstra

Se o algoritmo de Dijkstra manter uma fila de prioridades mínimas para mapear a distância estimada no lugar de D, o algoritmo torna-se mais eficiente que o Bellman-Ford. Seria utilizada uma operação do tipo "EXTRACT-MIN" no lugar da que está na linha 6, para encontrar o vértice com a menor distância. Ao extraí-lo da estrutura de prioridade, não mais seria necessário. Poderia-se gravar sua distância mínima em uma estrutura auxiliar e não mais utilizar a estrutura de visitas C. Ao atualizar as distâncias, poderia-se utilizar a operação de "DECREASE-KEY" da fila de prioridades no lugar da operação da linha 10.

Para essa fila de prioridades, poderia se utilizar um Heap, como o Heap Binário, no qual a implementação das operações supracitadas tem complexidade de tempo computacional $O(\log_2 n)$ para o "DECREASE-KEY" e $O(\log_2 n)$ para o "EXTRACT-MIN". Para essa aplicação, $n = |V|$. Utilizando essa estrutura de dados, sabe-se que no máximo executa-se $O(|E|)$ operações de "DECREASE-KEY" e $O(|V|)$ operações de "EXTRACT-

MIN". Então a complexidade computacional seria $O\big((|V|+|E|)\log_2|V|\big)$. Com Heap Fibonacci, consegue-se um tempo ainda melhor para a operação de "DECREASE-KEY" (em tempo amortizado).

5.3.2 Corretude do Algoritmo de Dijkstra

Teorema 5.3.1. *Dado um grafo $G = (V, E, w : E \to \mathbb{R}^+_*)$ e um vértice de origem $s \in V$, o algoritmo de Dijkstra termina com $D_v = \delta(s, v)$ para todo $v \in V$.*

Prova: Usa-se a seguinte invariante de laço: no início de cada iteração do laço das linhas 5 – 11, $D_v = \delta(s, v)$ para todo v o qual C_v =**true**. Para demonstrar isso, diz-se que $D_v = \delta(s, v)$ no momento em que v é marcado como visitado, ou seja, na linha 7. Uma vez demonstrado isso, recorre-se à propriedade do limite superior (Lema 5.1.3) para demonstrar que a igualdade é válida em todos os momentos a partir desse.

Inicialização: Inicialmente, C_u =**false** para todos $u \in V$. Então, a invariante é trivialmente verdadeiro.

Manutenção:

Por contradição, seja u o primeiro vértice para o qual $D_u \neq \delta(s, u)$ quando C_u torna-se **true**. O vértice u não pode ser s, pois $D_s = \delta(s, v) = 0$ nesse momento. Como $u \neq s$, deve existir algum caminho de s a u, senão $D_u = \delta(s, u) = \infty$ pela propriedade de inexistência de caminho (Lema 5.1.4), o que contradiz $D_u \neq \delta(s, u)$.

Assume-se então que haja um mínimo caminho p de s a u. Antes de C_u se tornar **true**, o caminho p conecta um vértice v o qual C_v =**true** a u. Decompõe-se o caminho p em $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$, no qual C_x =**true** e C_y =**false**. Afirma-se que $D_y = \delta(s, y)$ no momento que C_u se torna **true**. Para provar essa afirmação, observa-se que C_x =**true**. Então, u foi escolhido como primeiro vértice para o qual $D_u \neq \delta(s, u)$ quando C_u se torna **true**, tínha-se $D_x = \delta(s, x)$ quando C_x se tornou **true**. A aresta/arco (x, y) foi relaxada naquele momento, e a afirmação decorre da propriedade de convergência (Lema 5.1.6). Agora, pode-se obter uma contradição para provar que $D_u = \delta(s, u)$. Como

y aparece antes de u em um caminho mínimo de s a u e todos os pesos das arestas/arcos são não-negativos, temos $\delta(s,y) \leq \delta(s,u)$ e assim,

$$\begin{aligned} D_y &= \delta(s,y) \\ &\leq \delta(s,u) \\ &\leq D_u (\text{ pela propriedade do limite superior – Lema 5.1.3}). \end{aligned} \tag{5.16}$$

Porém, como C_u =**false** e C_y =**false** quando u foi escolhido na linha 6, tem-se $D_u \leq D_y$. Assim, as duas desigualdades da Equação (5.16) são de fato igualdades, o que dá

$$D_y = \delta(s,y) = \delta(s,u) = D_u. \tag{5.17}$$

Consequentemente, $D_u = \delta(s,u)$, o que contradiz a escolha de u. Conclui-se que $D_u = \delta(s,u)$ quando C_u se torna **true** e que essa igualdade é mantida até o término do algoritmo.

Término:

No término, quanto para C_v =**true** para todo $v \in V$, $D_v = \delta(s,v)$ para todo $v \in V$. ■

Corolário 5.3.2. *Ao executar algoritmo de Dijkstra (Algoritmo 13) sobre o grafo $G = (V,E,w)$ ponderado orientado ou não, sem ciclos de peso negativo, para o vértice de origem $s \in V$, o subgrafo dos predecessores G_π será uma árvore de caminhos mínimos em s.*

Prova: Imediata pelo Teorema 5.3.1 e a propriedade do subgrafo dos predecessores (Lema 5.1.9). ■

5.4 Floyd-Warshall

O algoritmo de Floyd-Warshall (Algoritmo 14) encontra o caminho mínimo para um grafo $G(V,E,w)$ ponderado dirigido ou não para todos os pares de vértices. O algoritmo

suporta arestas/arcos de pesos negativos, mas não opera em grafos com ciclos de peso negativo.

A função W (Equation (5.18)) define uma matriz de adjacências[1] para o algoritmo de Floyd-Warshall.

$$W\Big(G(V,E,w)\Big)_{uv} = \begin{cases} 0, & \text{se } u = v, \\ w\big((u,v)\big), & \text{se } u \neq v \wedge (u,v) \in E, \\ \infty, & \text{se } u \neq v \wedge (u,v) \notin E. \end{cases} \quad (5.18)$$

O algoritmo define um número de matrizes igual a $|V|$. Inicialmente, a matriz $D^{(0)}$ é definida como a matriz de adjacência de G (linha 1). Depois repete-se o procedimento de criar uma nova matriz e atualizar as distâncias de cada celula da nova matriz por $|V|$ vezes (linhas 2 a 6).

Algoritmo 14: Algoritmo de Floyd-Warshall.

Input :um grafo $G = (V,E,w)$

1 $D^{(0)} \leftarrow W\big(G\big)$

`// Assumindo que os vértices estão rotulados de` $1,2,\ldots,|V|$

2 **foreach** $k \in \{1,2,\ldots,|V|\}$ **do**

3 seja $D^{(k)} = \big(d^{(k)}_{uv}\big)$ uma nova matriz $|V| \times |V|$

4 **foreach** $u \in V$ **do**

5 **foreach** $v \in V$ **do**

6 $d^{(k)}_{uv} \leftarrow \mathbf{min}\Big\{d^{(k-1)}_{uv}, d^{(k-1)}_{uk} + d^{(k-1)}_{kv}\Big\}$

7 **return** $D^{(|V|)}$

5.4.1 Complexidade de Floyd-Warshall

O algoritmo Floyd-Warshall (Algoritmo 14) demanda $|V|^2$ operações para executar a linha 1. Como há o aninhamento de três laços limitados a $|V|$ iterações na sequên-

[1] Para grafos não-dirigidos, deve-se preencher a matriz resultante de $W(G)$ nas coordenadas (u,v) e (v,u)

cia, a operação na linha 6 será executada $|V|^3$ vezes. Logo, a complexidade de tempo computacional de Floyd-Warshall é de $\Theta(|V|^3)$.

Para este algoritmo, é interessante notar que foram utilizadas $|V|$ matrizes de $|V|$ linhas e $|V|$ colunas. Logo a complexidade de espaço para uma implementação que utilize o pseudo-código do Algoritmo 14 utilizaria espaço computacional de $\Theta(|V|^3)$. No entanto, é possível reduzir essa complexidade de espaço computacional em $\Theta(|V|^2)$.

Crie um pseudo-código para o Algoritmo 14 que demande complexidade de espaço computacional $\Theta(|V|^2)$.

5.4.2 Corretude de Floyd-Warshall

Prove que quando o Floyd-Warshall pára sobre uma entrada $G = (V, E, w)$ ele retornará as distâncias de caminhos mínimos para todo o par de vértice em V.

5.4.3 Construção de Caminhos Mínimos para Floyd-Warshall

O Algoritmo 15 além do peso dos caminhos mínimos em D, retorna a matriz dos predecessores Π, ou seja, uma matriz que indica o vértice anterior no caminho de cada coordenada u, v.

Algoritmo 15: Algoritmo de Floyd-Warshall com Matriz dos Predecessores.

```
Input   :um grafo G = (V, E, w)
1  D^(0) ← W(G)
   // Matriz dos predecessores
2  Π^(0)_uv ← (π^(0)_uv) uma nova matriz |V| × |V|
3  foreach u ∈ V do
4      foreach v ∈ V do
5          if (u, v) ∈ E then
6              π^(0)_uv ← u
7          else
8              π^(0)_uv ← null
9  foreach k ∈ V do
10     seja D^(k) = (d^(k)_uv) uma nova matriz |V| × |V|
11     seja Π^(k) = (π^(k)_uv) uma nova matriz |V| × |V|
12     foreach u ∈ V do
13         foreach v ∈ V do
               // atualizando matriz dos predecessores
14             if d^(k−1)_uv > d^(k−1)_uk + d^(k−1)_kv then
15                 π^(k)_uv ← π^(k−1)_kv
16             else
17                 π^(k)_uv ← π^(k−1)_uv
18             d^(k)_uv ← min{d^(k−1)_uv, d^(k−1)_uk + d^(k−1)_kv}
19 return (D^(|V|), Π^(|V|))
```

A impressão de cada caminho pode ser realizada através do Algoritmo 16, que recebe como entrada a matriz de predecessores gerada pelo Algoritmo 15, um vértice de origem u e um de destino v. Ao terminar, o algoritmo terá impresso na tela o caminho mínimo de u a v.

Algoritmo 16: Print-Shortest-Path.

Input : a matriz dos predecessores Π, um vértice de origem u, um vértice de destino v

```
1 if u = v then
2     print(u)
3 else
4     if π_uv = null then
5         print("Caminho inexistente de u para v")
6     else
7         Print-Shortest-Path(Π, u, π_uv)
8         print(v)
```

5.5 Dúvidas Frequentes ou Importantes

1) Qual a dificuldade de se obter algoritmos de tempo polinomial para resolver problemas caminhos de custo máximo?

Resposta: Considere um problema de encontrar caminhos simples do vértice s (origem) à t (destino) cujo a soma dos pesos em suas arestas ou arcos seja o máximo possível. O problema em questão aparece na literatura geralmente com o nome *Longest Path Problem* ou *Longest Weighted Path Problem*. Para ajudar a entender que há uma dificuldade em se encontrar algoritmos polinomiais para tal problema, utilizaremos de uma estratégia muito comum na literatura: vamos relacioná-lo com uma redução polinomial a outro problema, considerado NP-Difícil na literatura, o Problema do Caixeiro Viajante (PCV). Desse modo, a ideia é demonstrar uma maneira de resolver o PCV através de um algoritmo para o problema de caminho de custo máximo, utilizando adaptação de tempo polinomial na entrada e saída do algoritmo. Desse modo, se existir um algoritmo de tempo polinomial para o problema de caminho de custo máximo, ele resolveria o PCV em tempo polinomial, então $P = NP$.

Considere o grafo não-dirigido e completo $G = (V, E, w)$ dado como entrada para o PCV. Na adaptação polinomial da entrada, iremos construir o grafo completo $G' = (V', E', w')$.

Para isso, imagine um número K suficientemente grande. Considerando $V = \{1, 2, \ldots n\}$, $V' = (V - \{n\}) \cup \{s, t\}$. Considere também que $E' = \big\{\{u, v\} \in E : n \notin \{u, v\}\big\} \cup \big\{\{s, x\} : \{n, x\} \in E\big\} \cup \big\{\{x, t\} : \{x, n\} \in E\big\}$, $w'(\{u, v\}) = K - w(\{u, v\})$ para todo $u, v \in V - \{n\}$, $w'(\{s, x\}) = K - w(\{n, x\})$ para todo $x \in V - \{n\}$ e $w'(\{x, t\}) = K - w(\{x, n\})$ para todo $x \in V - \{n\}$. Ao submeter a entrada $\langle G', s, t\rangle$ ao algoritmo de caminho de custo máximo, ele encontra o caminho $p = \langle s, v_1, v_2, \ldots, t\rangle$. O caminho p poderia ser convertido, em tempo polinomial, no ciclo $c = \langle n, v_1, v_2, \ldots, n\rangle$, que corresponde a resposta esperada para o problema PCV, considerando grafo de entrada G. A resposta foi obtida depois de consulta ao material em Lawler (1976a).

CAPÍTULO 6

Problemas de Travessia

Em problemas de travessia, os estados possíveis podem ser representados como vértices de um grafo e as transições de um estado para outro podem ser representados como arestas ou arcos. Cada estado pode ser considerado como uma solução candidata a solução esperada. Nesses problemas, espera-se que haja um estado inicial (situação inicial) que possa atingir um estado final através de um caminho em sucessivos estados adjacentes. Depois do grafo representar o problema, seus estados e transições, uma das buscas estudadas aqui podem ser utilizadas para localizar um estado final.

Além da busca em largura e profundidade, há outros algoritmos que podem ser utilizados para realizar uma "travessia". Se há uma função que avalia quais vértices adjacentes aos já visitados são melhores, há a *Best First Search* ou Busca do Melhor Primeiro. Uma outra busca muito conhecida em problemas de localização de caminhos (*pathfinding*) é o A*, que utiliza duas funções (objetivo e heurística) para determinar de qual vértice a busca deve continuar. Há várias buscas utilizadas para travessia que foram reportadas na literatura.

É importante ressaltar que muitos dos problemas de travessia podem ter uma quantidade exponencial de estados em relação ao número de entidades que representem o tamanho. Nesse caso, se não houver uma estratégia mais eficiente de busca, as buscas em largura e profundidade podem demandar tempo exponencial determinístico, ou

seja, a execução pode demandar tempo mais elevado do que se pode aguardar para a resposta, mesmo em entradas de tamanho pequeno.

Em muitos casos, não é necessário produzir um grafo. Nesse contexto, a travessia ocorre em um grafo conceitual, onde os vértices visitados são construídos durante a busca.

Alguns dos problemas aqui citados foram obtidos a partir de Mariani (2019).

Problemas de menor caminho são um tipo bem específico de problemas de travessia.

6.1 Problema dos Canibais e Missionários

O problema dos canibais e missionários assume que três canibais e três missionários devem cruzar um rio, mas para isso possuem apenas um barco. Cada canibal e cada missionário sabe conduzir o barco. O barco suporta apenas dois indivíduos. Ao lado de cada margem, não poderá permanecer um número maior de canibais do que de missionários, pois há uma alta probabilidade de que os canibais devorem o(s) missionário(s) (MARIANI, 2019).

Existem várias formas de representar este problema. No contexto de problemas de travessia, deve-se pensar sobre a representação de cada estado, de modo que parte-se de um estado com todos os indivíduos de um lado do rio e deseja-se passá-los para outro lado. A representação de um estado que é utilizada nessa explicação é de uma dupla (α,β), na qual $\alpha,\beta \in \{1,2,3\}$ e representam o número de canibais e missionários respectivamente que permanecem no lado indesejado do rio. Enumerando todos os estados teria-se

$$\begin{aligned}\{&(0,0),(0,1),(0,2),(0,3),\\&(1,0),(1,1),(1,2),(1,3),\\&(2,0),(2,1),(2,2),(2,3),\\&(3,0),(3,1),(3,2),(3,3)\}.\end{aligned} \tag{6.1}$$

São ao todo 16 estados. O estado $(3,3)$ representa que três canibais e três missionários estão do lado indesejado do rio, ou seja, representa o estado inicial. O estado $(0,0)$ representa o estado final (MARIANI, 2019).

Em nenhum momento, pode haver mais canibais que missionários em qualquer uma das margens do rio. Então, o subconjunto de estados $\{(2,1),(3,2),(3,1)\}$ não podem ser admitidos, pois haveria um número maior de canibais que o número de missionários na margem indesejada. Há outros estados inadmissíveis que devem considerar o número maior de canibais na outra margem. Desse modo, considerando os demais estados inadmissíveis $\{(2,1),(3,2),(3,1),(1,2),(0,2),(0,1)\}$. Todos os demais estados seriam os vértices do grafo que representará o problema. Então, o conjunto de vértices $V = \{(0,0),(0,3),(1,0),(1,1),(1,3),(2,0),(2,2),(2,3),(3,0),(3,3)\}$.

O conjunto de arcos do grafo representaria então a transição possível entre os estados. Considera-se então um arco (v_1, v_2), no qual v_1 seria o estado de origem e v_2 seria o estado de destino. Considere também que $v_1 = (\alpha_1, \beta_1)$ e $v_2 = (\alpha_2, \beta_2)$. Para isso, deve-se considerar três regras básicas do problema:

- $\alpha_2 \leq \alpha_1$: o número de canibais que voltam é menor ou igual aos que foram enviados da margem indesejada;
- $\beta_2 \leq \beta_1$: o número de missionários que voltam é menor ou igual aos que foram enviados da margem indesejada;
- $1 \leq (\alpha_1 + \beta_1) - (\alpha_2 + \beta_2) \leq 2$: pode-se enviar no máximo dois indivíduos de um barco para outro.

Com essas regras, o percurso a ser descoberto deve considerar. Seja $i = 1,2,3\ldots$ a ordem do passo para encontrar um caminho a partir do estado inicial, nos passos ímpares deve-se seguir o arco em sua orientação original. Em passos pares, deve-se seguir um arco em sua direção inversa (caminho de volta à margem indesejada).

6.1.1 Construção e Busca

Pode-se realizar a construção do grafo enquanto a busca ocorre. Para isso, considere a formação de um grafo $G = (V, A)$. $(\alpha, \beta, o) \in V$, no qual $o \in \{d, e\}$ é a margem do rio (e para esquerda e d para direita). Considera-se as seguintes possibilidades de transição:

- 1 canibal;
- 1 missionário;
- 2 canibais;
- 2 missionários;
- 1 canibal e 1 missionário.

O conjunto de arcos $A = \{(v_1, v_2) : v1 = (\alpha_1, \beta_1, o_1), v_2 = (\alpha_2, \beta_2, o_2), o_1 \neq o_2\}$. O algoritmo de busca deve respeitar as restrições do problema original.

Defina um algoritmo de busca para encontrar um caminho entre os vértices $(3,3,e)$ e $(3,3,d)$.

6.2 Problemas dos Potes de Vinho

Considere que temos três potes com capacidades de 8, 5 e 3 litros, respectivamente, os quais não possuem qualquer marcação. O maior deles esta completamente cheio enquanto que os outros dois estão vazios. Estamos interessados em dividir o vinho em duas porções iguais de 4 litros, tarefa esta que pode ser realizada por transvasos sucessivos de um vaso no outro. Qual o menor número de transvasos necessários para completar a divisão (NETTO, 2006; MARIANI, 2019)?

6.3 Problema da Fuga dos Ladrões

Uma quadrilha de 3 ladrões assalta um banco e foge com uma mala de dinheiro para um aeroporto onde um avião pronto para decolar está à espera. O esconderijo é seguro, mas a fuga é difícil porque o avião só comporta 170 kg. Só um dos ladrões sabe pilotar e ele pesa 60 kg. O segundo, que é o guarda-costas do chefe, pesa 100 kg e o chefe pesa 70 kg. O chefe teme que o piloto fuja com o dinheiro (que pesa 40 kg) se tiver uma oportunidade. O piloto tem a mesma preocupação em relação ao chefe. Apenas o guarda-costas merece a confiança de ambos. A quadrilha, no entanto, já elaborou um plano de fuga capaz de satisfazer a todos. Qual é esse plano (MARIANI, 2019)?

6.4 Problema dos três maridos ciumentos

Três esposas e seus respectivos maridos desejam ir ao centro da cidade em um Corvette, o qual comporta apenas duas pessoas. Como eles poderiam deslocar-se até o centro considerando que nenhuma esposa deveria estar com um ou ambos os outros maridos a menos que seu marido também esteja presente (MARIANI, 2019)?

6.5 Problema de Agrupamento de Indivíduos

Da Silva Mendes, De Santiago e Lamb (2018) definem um problema de agrupamento de indivíduos através de uma representação em grafo realizando uma travessia através de um algoritmo semelhante ao A*. No problema, os vértices representam particionamento disjuntos de um conjunto de indivíduos. Seja $I = \{i_1, i_2, \ldots, i_n\}$ o conjunto de indivíduos, e $R \subseteq I \times I$ o conjunto de relacionamentos entre os indivíduos, precisa-se saber qual o vértice que maximiza os relacionamentos entre indivíduos do mesmo grupo [1].

O grafo considerado é $G = (V, E)$ no qual:

[1] Para aqueles que quiserem mais detalhes sobre esse projeto, basta ler o artigo Da Silva Mendes, De Santiago e Lamb (2018) ou conversar com o professor, coautor do trabalho

- V é o conjunto de soluções, na qual cada solução $v \in V$ é uma partição disjunta composta por todos os indivíduos de I;

- E é o conjunto de arestas, na qual cada aresta conecta dois vértices (duas soluções) ditos vizinhos. Para o trabalho, foi considerado que dois vértices são vizinhos se apenas há um indivíduo em uma posição distinta.

CAPÍTULO 7

Conectividade

7.1 Componentes Fortemente Conexas

Um grafo conexo é aquele no qual há um caminho entre todos os pares de vértices. É dita uma componente fortemente conexa de um grafo dirigido não-ponderado $G = (V, A)$ é um conjunto máximo de vértices $C \subseteq V$, tal que para todo o par de vértices u, v em C têm-se $u \rightsquigarrow v$ e $v \rightsquigarrow u$.

Um algoritmo para identificar as Componentes Fortemente Conexas é relatado por Cormen et al. (2012) e apresentado aqui no Algoritmo 17. Nele, utiliza-se duas vezes a busca em profundidade sugerida também por Cormen et al. (2012) (Algoritmos 18 (DFS) e 19 (DFS-Visit)). Primeiramente, faz-se a busca em profundidade para descobrir os caminhos de todos os vértices para todos os outros. Depois, percorre-se os mesmos caminhos em um grafo transposto (G^T). As árvores representadas como os antecessores em A^T trarão a resposta: cada árvore é uma componente fortemente conexa.

Algoritmo 17: Algoritmo de Componentes-Fortemente-Conexas

```
Input :um grafo dirigido não ponderado G = (V, A)
  /* Chamar a DFS (do Algoritmo 18) para computar os tempos de término para
     cada vértice                                                          */
1 (C, T, A', F) ← DFS(G)
  /* Criar grafo transposto de G, chamado de G^T.                          */
2 A^T ← {}
3 foreach (u, v) ∈ A do
4   A^T ← A^T ∪ {(v, u)} /* Inverte-se todos os arcos para G^T.            */
5 G^T ← (V, A^T)
  /* Chamar a DFS (do Algoritmo 18) alterado para que ele execute o laço da
     linha 6, selecionando vértices em ordem decrescente de F              */
6 (C^T, T^T, A'^T, F^T) ← DFS-adaptado(G^T)
  /* dar saída de cada árvore na floresta em profundidade em A^T como uma
     componente fortemente conexa.                                         */
7 return A'^T
```

Algoritmo 18: DFS de Cormen et al. (2012).

```
Input :um grafo dirigido não ponderado G = (V, E)
  // Configurando todos os vértices
1 C_v ← false ∀v ∈ V
2 T_v ← ∞ ∀v ∈ V
3 F_v ← ∞ ∀v ∈ V
4 A_v ← null ∀v ∈ V
  // configurando o tempo de início
5 tempo ← 0
6 foreach u ∈ V do
7     if C_u = false then
          // DFS-Visit é especificado no Algoritmo 19
8         DFS-Visit(G, u, C, T, A, F, tempo)
9 return (C, T, A, F)
```

Algoritmo 19: DFS-Visit de Cormen et al. (2012).

```
Input :um grafo G = (V, E), vértice de origem v ∈ V, e os vetores C, T, A e F, e uma
       variável tempo ∈ Z*+
1 C_v ← true
2 tempo ← tempo +1
3 T_v ← tempo
4 foreach u ∈ N+(v) do
5     if C_u = false then
6         A_u ← v
7         DFS-Visit(G, u, C, T, A, F, tempo)
8 tempo ← tempo +1
9 F_v ← tempo
```

7.1.1 Complexidade do Algoritmo de Componentes Fortemente Conexas

A complexidade de tempo computacional do Algoritmo 17 é dependente da complexidade de tempo do algoritmo DFS de Cormen et al. (2012) (Algoritmos 18 (DFS) e 19 (DFS-Visit)). Para o algoritmo DFS, as instruções das linhas 1 a 4 demandam uma

quantidade de instruções igual a $\Theta(|V|)$. O laço entre as linhas 6 a 8 é executado por um número de iterações dependente do número de vértices ($\Theta(|V|)$). O procedimento DFS-Visit é invocado apenas uma vez para cada vértice, graças ao vetor de visitados C. Como nesse procedimento as adjacências de cada vértice são visitadas, há também uma dependência do número de arcos saintes em cada vértice. Logo, a complexidade computacional do Algoritmo 18, e consequentemente, a do algoritmo de Componentes Fortemente Conexas, é $\Theta(|V|+|E|)$.

7.1.2 Corretude do Algoritmo de Componentes Fortemente Conexas

7.1.2.1 Propriedades de Buscas em Profundidade

Teorema 7.1.1. *Em qualquer busca em profundidade em um grafo dirigido ou não $G = (V, E)$, para quaisquer dois vértices u e v, exatamente uma das três condições é válida:*

- *Os intervalos $[T_u, F_u]$ e $[T_v, F_v]$ são completamente disjuntos, e nem u nem v é descendente do outro na floresta de profundidade.*
- *O intervalo $[T_u, F_u]$ está contido inteiramente dentro do intervalo $[T_v, F_v]$ e u é um descendente de v em uma árvore de profundidade.*
- *O intervalo $[T_v, F_v]$ está contido inteiramente dentro do intervalo $[T_u, F_u]$ e v é um descendente de u em uma árvore de profundidade.*

Prova: A prova começará com o caso de $T_u < T_v$. Para isso, consideramos dois subcasos: $T_v < F_u$ ou não. O primeiro subcaso ocorre quando $T_v < F_u$, portanto v foi descoberto quando $C_u =$**true**, ou seja, v foi descoberto mais recentemente que u, o que implica que v é descendente de u. Ao findar a busca de u (linha 9 do Algoritmo 19), todas as arestas de u foram exploradas, e consequentemente $[T_v, F_v]$ está completamente contido no intervalo $[T_u, F_u]$.

Ainda considerando $T_u < T_v$, mas no subcaso de $F_u < T_v$, devido a $T_w < F_w$ para todo $w \in V$, $T_u < F_u < T_v < F_v$, assim os intervalos $[T_u, F_u]$ e $[T_v, F_v]$ são disjuntos. Como os intervalos são disjuntos, nenhum vértice foi descoberto enquanto o outro ainda não havia findado (linha 9 do Algoritmo 19), então nenhum é descendente do outro.

O caso de $T_v < T_u$ é semelhante, com papéis de u e v invertidos no argumento anterior. ■

Corolário 7.1.2. *O vértice v é um descendente adequado do vértice u na floresta em profundidade para um grafo dirigido ou não $G = (V, E)$ sse $T_u < T_v < F_v < F_u$.*

Prova: Imediata pelo Teorema 7.1.1. ■

Teorema 7.1.3. *Em uma floresta em profundidade de um grafo dirigido ou não $G = (V, E)$, o vértice v é um descendente do vértice u sse no momento T_u, em que uma busca descobre u, há um caminho de u a v inteiramente de vértices w marcados com $C_w =$**false**.*

Prova: Se $v = u$, então o caminho de u a v não possui vértices além dele mesmo.

Agora, supõe-se que v seja um descendente próprio de u, que ainda tem $C_v =$**false** quando se define o valor de T_u. Pelo Corolário 7.1.2, $T_u < T_v$ e portanto $C_v =$**false** no tempo T_u. Visto que v pode ser descendente de u, todos os vértices w num caminho simples de u até v tem $C_w =$**false**.

Agora, supõe-se que haja um caminho de vértices w marcados com $C_w =$**false** no caminho de u a v no tempo T_u, mas v não se torna descendente de u na árvore de profundidade. Sem prejuízo, considera-se que todo o vértice exceto v ao longo do caminho se torne um descendente de u. Seja w o predecessor de v no camino, de modo que w seja um descendente de u. Pelo Corolário 7.1.2, $F_w < F_u$. Como v tem que ser descoberto depois de u ser descoberto, mas antes de w ser terminado, têm-se $T_u < T_v < F_w < F_u$. Então o Teorema 7.1.1 implica que o intervalo $[T_v, F_v]$ está contido no intervalo $[T_u, F_u]$. Pelo Corolário 7.1.2, v deve ser descendente de u. ■

7.1.2.2 Propriedades de Componentes Fortemente Conexas

Para as propriedades abaixo, considere que:

- $d(S) = \min_{v \in S}\{T_v\}$;
- $f(S) = \max_{v \in S}\{F_v\}$;

Lema 7.1.4. *Considerando C e C' componentes fortemente conexas distintas em um grafo dirigido $G = (V, A)$, seja $u, v \in C$ e $u', v' \in C'$ e suponha que G tenha um caminho $u \rightsquigarrow u'$. Então, G não pode conter um caminho $v' \rightsquigarrow v$.*

Prova: Se G possui um caminho $v' \rightsquigarrow v$, então contém os caminhos $u \rightsquigarrow u' \rightsquigarrow v'$ e $v' \rightsquigarrow v \rightsquigarrow u$ em G. Assim, u e v' podem ser visitados um a partir do outro, o que contradiz a hipótese de que C e C' são componentes fortemente conexas distintas. ■

Lema 7.1.5. *Sejam C e C' componentes fortemente conexas distintas no grafo dirigido $G = (V, A)$. Suponha que haja um arco $(u, v) \in A$, no qual $u \in C$ e $v \in C'$. Então, $f(C) > f(C')$.*

Prova: Considera-se dois casos dependendo qual das componentes fortemente conexas (C ou C') têm o primeiro vértice descoberto na busca em profundidade.

Se $d(C) < d(C')$, seja x o primeiro vértice descoberto em C. No tempo T_x, todos os vértices em C e C' são não visitados ($C_v =$**false** para todo $v \in C \cup C'$). Pode-se afirmar então que há um caminho em x a w para qualquer $w \in C'$ por causa do arco $(u, v) \in A$, então $x \rightsquigarrow u \rightarrow v \rightsquigarrow w$. Pelo Teorema 7.1.3, todos os vértices em C e C' se tornam descendentes de x ma árvore de profundidade. Pelo Corolário 7.1.2, x tem o tempo de término mais recente que qualquer um de seus descendentes, portanto $F_x = f(C) > f(C')$;

Se $d(C) > d(C')$, seja y o primeiro vértice descoberto em C'. No tempo T_y, todos os vértices w em C' têm $C_w =$**false** e G contém um caminho de y a cada vértice em C' formado apenas por vértices z com $C_z =$**false**. Pelo Teorema 7.1.3, todos os vértices em C' se tornam descendentes de y na árvore de profundidade. Pelo Teorema 7.1.3,

$F_y = F(C')$. No tempo T_y, todos os vértices w em C têm C_w =**false**. Como existe um arco (u, v) de C a C', o Lema 7.1.4 implica que não pode haver um caminho de C' a C. Consequentemente, nenhum vértice em C pode ser visitado por y. Portanto, no tempo F_y, todos os vértices w em C ainda tem C_w =**false**. Assim, para qualquer vértice em $w \in C$, têm-se $F_w > F_y$, o que implica que $f(C) > f(C')$. ■

Corolário 7.1.6. *Considerando C e C' componentes fortemente conexas distintas no grafo dirigido $G = (V, A)$, suponha que havia um arco $(u, v) \in A^T$, no qual $u \in C$ e $v \in C'$. Então, $f(C) < f(C')$.*

Prova: Como $(u, v) \in A^T$, têm-se $(v, u) \in A$. Visto que as componentes fortemente conexas em G^T são as mesmas, o Lema 7.1.5 implica que $f(C) < f(C')$. ■

Teorema 7.1.7. *O Algoritmo 17 (de Componentes-Fortemente-Conexas) encontra corretamente as componentes fortemente conexas de um grafo dirigido $G = (V, A)$ dado como sua entrada.*

Prova: A prova é realizada por indução em relação ao número de árvores de busca encontradas na busca em profundidade de G^T na linha 6. Cada árvore forma uma componente fortemente conexa.

A hipótese da indução é que as primeiras k árvores produzidas na linha 6 são componentes fortemente conexas.

A base da indução, quando $k = 0$, é trivial.

No passo da indução, supõe-se que cada uma das k primeiras árvores de profundidade produzidas na linha 6 é uma componente fortemente conexa, e consideramos a $(k+1)$-ésima árvore produzida. Seja u a raiz dessa árvore, e supondo que u esteja na componente fortemente conexa C. Como resultado do modo que se escolhe a raiz da árvore na linha 6, $F_u - f(C) > f(C')$ para qualquer componente fortemente conexa C' exceto C que ainda tenha de ser visitada. Pela hipótese de indução, no momento da busca de u na árvore de profundidade, todos os vértices w em C tem C_w =**false**. Então, pelo Teorema 7.1.3, todos os outros vértices de C são descendentes de u nessa árvore

de profundidade. Além disso, pela hipótese de indução e pelo Corolário 7.1.6, qualquer arco em G^T que saem de C devem ir até componentes fortemente conexas que já foram visitadas. Assim, nenhum vértice em uma componente fortemente conexa, exceto C, será um descendente de u durante a busca em profundidade de G^T. Portanto, os vértices da árvore de busca em profundidade em G^T enraizada em u formam exatamente uma componente fortemente conexa, o que conclui o passo de indução e a prova. ■

7.2 Ordenação Topológica

A ordenação topológica no contexto de grafos, recebe um grafo acíclico dirigido $G = (V, A)$ e ordena linearmente todos os vértices tal que se existe um arco $(u, v) \in A$ então u aparece antes de v na ordenação. O algoritmo de Ordenação Topológica tem como base uma busca em profundidade com a adição de uma lista O para inserir os vértices, como pode ser visto no Algoritmo 21. Os vértices são inseridos sempre no início da lista O logo que o algoritmo termina de visitá-lo (linha 10 do Algoritmo 21).

Algoritmo 20: DFS para Ordenação Topológica

Input : um grafo dirigido não ponderado $G = (V, A)$

```
// Configurando todos os vértices
1 C_v ← false ∀v ∈ V
2 T_v ← ∞ ∀v ∈ V
3 F_v ← ∞ ∀v ∈ V
// configurando o tempo de início
4 tempo ← 0
// Criando lista com os vértices ordenados topologicamente
5 O ← ()
6 foreach u ∈ V do
7     if C_u = false then
          // DFS-Visit-OT é especificado no Algoritmo 21
8         DFS-Visit-OT(G, u, C, T, F, tempo, O)
9 return O
```

Algoritmo 21: DFS-Visit-OT.

Input : um grafo $G = (V, E)$, vértice de origem $v \in V$, e os vetores C, T e F, e uma variável *tempo* $\in \mathbb{Z}^+_*$, uma lista O

```
1 C_v ← true
2 tempo ← tempo + 1
3 T_v ← tempo
4 foreach u ∈ N⁺(v) do
5     if C_u = false then
6         DFS-Visit-OT(G, u, C, T, F, tempo, O)
7 tempo ← tempo + 1
8 F_v ← tempo
// Adiciona o vértice v no início da lista O
9 O ← (v) ∪ O
```

7.2.1 Complexidade da Ordenação Topológica

Como a inserção no início de uma lista demanda tempo $O(1)$, a complexidade de tempo da Ordenação Topológica de um grafo acíclico é dependente da Busca em Profundidade. Logo, a complexidade da Ordenação Topológica é $O(|V| + |A|)$.

7.2.2 Corretude da Ordenação Topológica

Lema 7.2.1. *Um grafo dirigido $G = (V, A)$ é acíclico sse uma busca em profundidade de G não produz nenhum arco de retorno.*

Prova: Supondo que uma busca em profundidade produza um arco de retorno (u, v). Então, o vértice v é um ascendente (ancestral) do vértice u na floresta em profundidade. Assim, G contém um caminho de v a u, e o arco (u, v) completa o ciclo.

Supondo que G contenha um ciclo c, mostrou-se que uma busca em profundidade de G produz um arco de retorno. Seja v o primeiro vértice a ser descoberto em c e seja (u, v) o arco precedente em c. No tempo T_v, os vértices em c formam um caminho de vértices w com $C_w =$**false** de v a u. Pelo Teorema 7.1.3, o vértice u se torna um descendente de v na floresta em profundidade. Então (u, v) é um arco de retorno. ■

Teorema 7.2.2. *O Algoritmo 20 produz uma ordenação topológica de um grafo dirigido acíclico $G = (V, A)$ dado como entrada.*

Prova: Supondo que o algoritmo seja executado sobre um determinado grafo dirigido acíclico $G = (V, A)$ para determinar os tempos de término para seus vértices. É suficiente mostrar que, para qualquer par de vértices distintos $u, v \in V$, se G contém um arco de u a v, então $F_v < F_u$. Considere qualquer arco (u, v) explorado no algoritmo. Quando esse arco é explorado, v ainda não foi visitado (por DFS-Visit-OT), já que v é ascendente (ancestral) de u e (u, v) é um arco de retorno, o que contradiz o Lema 7.2.1. Portanto, v já deve ter $C_v =$**false** ou já foi visitado. Se v tem $C_v =$**false**, ele se torna um descendente de u e $F_v < F_u$. Se v já foi visitado F_v já foi definido, então $F_v < F_u$. Assim, para qualquer arco $(u, v) \in A$, têm-se $F_v < F_u$. ■

CAPÍTULO 8

Árvores Geradoras Mínimas

Uma árvore geradora é um subconjunto acíclico (uma árvore) de todos os vértices de um grafo. Dado um grafo ponderado $G = (V, E, w)$, o problema de encontrar uma arvore geradora mínima é aquele no qual busca-se encontrar uma árvore que conecte todos os vértices do grafo G, tal que a soma dos pesos das arestas seja o menor possível. Seja T uma árvore geradora, diz-se que o custo da árvore geradora de T é dado por $w(T) = \sum_{\{u,v\}\in T} w\big(\{u,v\}\big)$.

Este capítulo apresenta dois métodos para encontrar árvores geradoras mínimas: Kruskal e Prim. Esses dois algoritmos gulosos se baseiam em um método genérico apresentado por Cormen et al. (2012). Dado um grafo não-dirigido e ponderado $G = (V, E, w)$, esse método genérico encontra uma árvore geradora mínima.

O Algoritmo 22 apresenta o método genérico para uma árvore geradora mínima para G. O método se baseia na seguinte invariante de laço: “Antes de cada iteração, A é um subconjunto de alguma árvore geradora mínima.”. A cada iteração, determina-se uma aresta que pode ser adicionada a A sem violar a invariante de laço. A aresta segura é uma aresta que se adicionada a A mantém a invariante.

Algoritmo 22: Método Genérico de Cormen et al. (2012).

Input :um grafo $G = (V, E, w)$

1 $A \leftarrow \{\}$

2 **while** *A não formar uma árvore geradora* **do**

3 encontrar uma aresta $\{u, v\}$ que seja segura para A

4 $A \leftarrow A \cup \{\{u, v\}\}$

5 **return** A

8.1 Propriedades do Método Genérico

Teorema 8.1.1. *Considere um grafo conexo não-dirigido ponderado $G = (V, E, w)$. Seja $A \subseteq E$ uma árvore geradora mínima de G. Seja $(S, V \setminus S)$ qualquer conjunto de corte de G que respeita A e seja $\{u, v\}$ uma aresta leve*[1] *que cruza $(S, V \setminus S)$, então $\{u, v\}$ é uma aresta segura para A.*

Prova:

Seja T uma árvore geradora mínima que inclui A e suponha que T não contenha a aresta leve $\{u, v\}$, pois se tiver, o processo termina. Constrói-se então outra árvore geradora mínima T' que inclui $A \cup \{\{u, v\}\}$ usando uma técnica de recortar e colar, mostrando assim que $\{u, v\}$ é uma aresta segura para A.

Desde que u esteja em S e v em $V \setminus S$, a aresta $\{u, v\}$ forma um ciclo com as arestas de caminho simples p de u a v em T. No mínimo uma aresta em T está no caminho simples p e cruza o corte. Considere que $\{x, y\}$ seja essa aresta de corte. A aresta $\{x, y\}$ não está em A, pois o corte respeita A. Desde que $\{x, y\}$ está no único caminho de u a v em T, removê-lo divide T em duas componentes. Adicionar $\{u, v\}$ o reconecta a forma de uma nova árvore geradora $T' = T \cup \{\{u, v\}\} \setminus \{\{x, y\}\}$.

Depois, mostra-se que T' é uma árvore geradora mínima. Desde que $\{u, v\}$ é uma aresta leve atravessando $(S, S \setminus V)$ e $\{x, y\}$ também atravessa esse corte, $w(\{u, v\}) \leq w(\{x, y\})$. Por essa razão:

[1] Arestas de baixo custo/peso.

$$
\begin{aligned}
w(T') &= w(T) + w\big(\{u,v\}\big) - w\big(\{x,y\}\big) \\
&\leq w(T').
\end{aligned}
\tag{8.1}
$$

Mas, T é uma árvore geradora mínima, então $w(T) \leq w(T')$. Desse modo, T' precisa ser uma árvore geradora mínima também.

Falta mostrar que $\{u,v\}$ é uma aresta segura para A. Têm-se $A \subseteq T'$, desde que $A \subseteq T$ e $\{x,y\} \notin A$; então $A \cup \big\{\{u,v\}\big\} \subseteq T'$. Consequentemente, desde que T' é uma árvore geradora mínima, $\{u,v\}$ é uma aresta segura para A. ■

Corolário 8.1.2. *Considere um grafo conectado não-dirigido e ponderado $G = (V,E,w)$. Seja $A \subseteq E$ incluído em alguma árvore geradora mínima de G e seja $C = (V_C, E_C)$ um componente conectado (uma árvore) na floresta $G_A = (V,A)$. Se $\{u,v\}$ é uma aresta leve conectando C a algum outro componente em G_A, então $\{u,v\}$ é uma aresta segura.*

Prova: O corte $(V_C, V \setminus V_C)$ respeita A e $\{u,v\}$ é uma aresta leve para esse corte. Por essa razão, $\{u,v\}$ é uma aresta segura para A (com base no Teorema 8.1.1). ■

8.2 Algoritmo de Kruskal

O algoritmo de Kruskal inicia com $|V|$ árvores (conjuntos de um vértice cada). Ele encontra uma aresta segura e a adiciona a uma floresta, que está sendo montada, conectando duas árvores na floresta. Essa aresta $\{u,v\}$ é de peso mínimo. Suponha que C_1 e C_2 sejam duas árvores conectadas por uma aresta $\{u,v\}$. Visto que essa deve ser uma aresta leve, o Corolário 8.1.2 implica que $\{u,v\}$ é uma aresta segura para C_1. Kruskal é um algoritmo guloso, pois ele adiciona à floresta uma aresta de menor peso possível a cada iteração.

Um pseudo-código de Kruskal pode ser visualizado no Algoritmo 23. O algoritmo se inicia definindo as $|V|$ árvores desconectadas; cada uma contendo um vértice (linhas 2 a 4). Então, cria-se uma lista das arestas E' ordenadas por peso (linha 5). Depois,

iterativamente, se tenta inserir uma aresta leve em duas árvores que contém nodos não foram conectadas ainda (linhas 7 a 9). Quando a inserção ocorre, a estrutura de dados S_v que mapeia a árvore de cada vértice v é é atualizada. O procedimento se repete até que todas as arestas tenham sido avaliadas no laço.

Algoritmo 23: Algoritmo de Kruskal.

```
   Input  :um grafo G = (V, E, w)
1  A ← {}
2  S ← vetor de |V| elementos vazios
3  foreach v ∈ V do
4      S_v ← {v}
5  E' ← lista de arestas ordenadas por ordem crescente de peso
6  foreach {u, v} ∈ E' do
7      if S_u ≠ S_v then
8          A ← A ∪ {{u, v}}
9          x ← S_u ∪ S_v
10         foreach y ∈ x do
11             S_y ← x
12 return A
```

8.2.1 Complexidade do Algoritmo de Kruskal

A complexidade de tempo do algoritmo de Kruskal depende da estrutura de dados utilizada para implementar o mapeamento das árvores de cada vértice, ou seja, da estrutura S do Algoritmo 23. Para a implementação mais eficiente, verifique as operações de conjuntos disjuntos no Capítulo 21 de Cormen et al. (2012). Sabe-se que para ordenar o conjunto de arestas na linha 5, deve-se executar um algoritmo de ordenação. O mais eficiente conhecido demanda tempo $\Theta(|E| \log_2 |E|)$.

Qual a estrutura de dado implementa a versão mais eficiente do Algoritmo de Kruskal?

8.3 Algoritmo de Prim

O algoritmo de Prim é (também) um caso especial do método genérico apresentado no Algoritmo 22. O algoritmo de Prim é semelhante ao algoritmo de Dijkstra, como pode ser observado no pseudo-código do Algoritmo 24. As arestas no vetor A formam uma árvore única. Essa árvore tem raízes em um vértice arbitrário r. Essa arbitrariedade não gera problemas de corretudo no algoritmo, pois uma árvore geradora mínima deve conter todos os vértices do grafo. A cada iteração, seleciona-se o vértice que é atingido por uma aresta de custo mínimo (linha 7), sendo que o vértice selecionado adiciona suas adjacências e pesos na estrutura de prioridade Q, que, futuramente, selecionará a chave de menor custo, ou seja, o vértice conectado a árvore que possui o menor custo. Pelo Corolário 8.1.2, esse comportamento adiciona-se apenas arestas seguras em A.

Antes de cada iteração do laço da linha 6, a árvore obtida é dada por $\{\{v, A_v\} : v \in V\setminus\{r\}\setminus Q\}$. Os vértices que já participam da solução final são $V\setminus Q$. Além disso, para todo $v \in Q$, se $A_v \neq$ **null**, então $K_v < \infty$ e K_v é o peso de uma resta leve $\{v, A_v\}$ que conecta o vértice v a algum vértice que já está na árvore que está sendo construída. As linhas 7 e 8 indentificam um vértice $u \in Q$ incidente em alguma aresta leve que cruza o corte $(V\setminus Q, Q)$, exceto na primeira iteração. Ao remover u de Q, o mesmo é acrescido ao conjunto $V\setminus Q$ na árvore, adicionando a aresta (u, A_u) na solução.

Se G é conexo, para montar a árvore geradora mínima a partir do vetor resultante A, deve-se criar o conjunto $\{\{v, A_v\} : v \in V\setminus\{r\}\}$.

Algoritmo 24: Algoritmo de Prim.

```
Input :um grafo G = (V, E, w)
1  r ← selecionar um vértice arbitrário em V
   // Definindo o vetor dos antecessores A e uma chave para cada vértice K
2  A_v ← null ∀v ∈ V
3  K_v ← ∞ ∀v ∈ V
4  K_r ← 0
   // Definindo a estrutura de prioridade de chave mínima Q
5  Q ← (V, K)
6  while Q ≠ {} do
7      u ← arg min_{v∈Q}{K_v}
8      Q ← Q\{u}
9      foreach v ∈ N(u) do
10         if v ∈ Q ∧ w({u, v}) < K_v then
11             A_v ← u
12             K_v ← w({u, v})
13 return A
```

8.3.1 Complexidade do Algoritmo de Prim

Para implementar o algoritmo de Prim de maneira mais eficiente possível, deve-se encontrar uma estrutura de prioridade que realize as operações de extração do valor mínimo e da alteração das chave de maneira rápida.

Qual a complexidade em tempo computacional da versão mais eficiente do Algoritmo de Prim?

CAPÍTULO 9

Fluxo Máximo

O problema de fluxo máximo pode ser formalizado da seguinte forma: dado um grafo dirigido e ponderado (rede residual) $G = (V, A, c)$, um vértice $s \in V$ de origem de fluxo e um vértice de destino $t \in V$, deseja-se despachar um fluxo de s para t de acordo com as capacidades dos arcos dadas por c no qual a soma total de fluxo que incide em t seja máxima. Desse modo, o fluxo despachado em cada arco deve ser menor ou igual a sua capacidade. O fluxo total que chega a um vértice $v \in V - \{s, t\}$ deve ser o mesmo que sai do vértice (sob pena de acúmulo de fluxo no vértice, que não tem reservatório para fluxo).

Um exemplo de aplicação pode ser dado para um projeto de malha viária terrestre. Imagine que G seja a malha de uma cidade nas quais os veículos entram em s e saiam em t. Deseja-se despachar o máximo de veículos possíveis. Para isso, devemos respeitar as capacidades de cada via (arcos). Exemplos de aplicação podem ser obtidos para despacho de líquidos (água, esgoto) ou eletricidade.

9.1 Redes de Fluxo

Uma rede de fluxo, no contexto da Teoria dos Grafos, é um grafo dirigido ponderado $G = (V, A, c)$, na qual V representa o conjunto de vértices, A o conjunto de arcos e

$c : V \times V \rightarrow \mathbb{R}^+$ é a função de capacidade de cada arco. Impõe-se ainda que se há um arco (u, v) não há um arco (v, u). Se $(u, v) \notin A$, então $c\big((u, v)\big) = 0$. Há dois vértices distintos: o vértice de origem ou fonte, geralmente chamado de s, e o vértice de destino ou sorvedouro, chamado de t. or conveniência, assume-se que todo o vértices $v \in V \setminus \{s, t\}$, $s \rightsquigarrow v \rightsquigarrow t$ (CORMEN et al., 2012).

Um fluxo em G é uma função $f : V \times V \rightarrow \mathbb{R}$ que satisfaz as seguintes propriedades:

- Restrição de capacidade: para todo $u, v \in V$, $0 \leq f\big((u, v)\big) \leq c\big((u, v)\big)$;

- Conservação de fluxo: para todo $u \in V \setminus \{s, t\}$,

$$\sum_{v \in V} f\big((v, u)\big) = \sum_{v \in V} f\big((u, v)\big).$$

  Quando $(u, v) \notin A$, então $f\big((v, u)\big) = 0$.

  A quantidade não negativa $f\big((v, u)\big)$ é denominada o fluxo do vértice u a v. O valor de fluxo é dado por $F = \sum_{v \in V} f\big((s, v)\big) - \sum_{v \in V} f\big((v, s)\big)$, ou seja, o total de fluxo que sai de s menos o total de fluxo que entra em s.

  Quando $(u, v), (v, u) \in A$, deve-se remover (u, v) de A, adicionar um novo vértice v' a V, adicionar os arcos (u, v') e (v', v) a A definido as capacidades $c\big((u, v')\big) \rightarrow c\big((u, v)\big)$ e $c\big((v', v)\big) \rightarrow c\big((u, v)\big)$ e $c\big((u, v)\big) \rightarrow 0$.

Para múltiplas origens $s_1, s_2, \ldots, s_k$ pode-se adicionar um vértice inicial s' a G e os arcos (s', s_i) com capacidade $c\big((s', s_i)\big) \rightarrow \infty$ para todo $i \in \{1, 2, \ldots, k\}$.

Para múltiplos destinos $t_1, t_2, \ldots, t_l$ pode-se adicionar um vértice final t' a G e os arcos (t_j, t') com capacidade $c\big((t_j, t')\big) \rightarrow \infty$ para todo $j \in \{1, 2, \ldots, l\}$.

O problema de fluxo máximo busca encontrar o maior fluxo em G de s a t tal que as capacidades de cada arco sejam respeitadas.

9.2 Rede Residual

Uma rede residual consiste em um grafo $G_f = (V, A_f, c_f)$ composto por arcos com capacidades $c_f\big((u,v)\big) = c\big((u,v)\big) - f\big((u,v)\big)$ para todo arco A. Para cada arco (u,v) em A, têm-se um arco invertido em A_f com capacidade $c_f\big((v,u)\big) = f\big((u,v)\big)$. Se um arco $(u,v) \notin A$, então $c_f\big((v,u)\big) = 0$. Desse modo, $|A_f| = 2|A|$.

9.3 Caminhos Aumentantes

Em uma rede de fluxo G, um caminho aumentante p é um caminho simples de s a t na rede residal G_f. Ele aumenta o fluxo na rede sem ferir as restrições de capacidades impostas em c. A capacidade residual de p é $c_f(p) = \mathbf{min}\{c_f\big((u,v)\big) : (u,v) \in p\}$.

Lema 9.3.1. *Seja $G = (V, A, c)$ uma rede de fluxo com fonte s e sorvedouro em t, seja p um caminho aumentante em G e seja f um fluxo em G. Defina a função $f_p : V \times V \to \{R\}$ por*

$$f_p\big((u,v)\big) = \begin{cases} c_f(p) & se (u,v) \in p, \\ 0 & caso\ contrário. \end{cases}$$

Então, f_p é um fluxo em G_f com valor $c_f(p) > 0$.

Prova: Página 524 de Cormen et al. (2012).■

Corolário 9.3.2. *Seja $G = (V, A, c)$ uma rede de fluxo, seja f um fluxo em G e seja p um caminho aumentante em G_f. Considere a função f_p e suponha que aumenta-se f adicionando f_p. Então a função $f \uparrow f_p$ é um fluxo em G.*

Prova: Página 525 de Cormen et al. (2012).■

9.4 Cortes de Redes de Fluxo

Em uma das estratégias para se obter o fluxo máximo, busca-se caminhos aumentantes iterativamente. Como saber que quando o algoritmo termina, têm-se realmente o fluxo

máximo? O teorema de fluxo máximo/corte mínimo diz que se não houver caminho aumentante, atingiu-se o fluxo máximo.

Lema 9.4.1. *Seja $G = (V, A, c)$ uma rede de fluxo com fonte s e sorvedouro em t, e seja (S, T) qualquer corte de G. O fluxo que passa pelo corte é igual a F (o valor do fluxo).*

Prova: Página 526 de Cormen et al. (2012).■

Corolário 9.4.2. *O valor de qualquer fluxo em f em uma rede de fluxo G é limite superior pela capacidade de qualquer corte de G.*

Prova: Página 526 de Cormen et al. (2012).■

Teorema 9.4.3. *Se f é um fluxo em uma rede de fluxo $G = (V, A, c)$ com fonte s e sorvedouro t, então as seguintes condições são equivalentes:*

1. *f é um fluxo máximo em G.*
2. *A rede residual G_f não contém nenhum caminho aumentante.*
3. *F é a capacidade de corte para algum corte (S, T).*

Prova: Página 527 de Cormen et al. (2012).■

9.5 Ford-Fulkerson

O algoritmo de Ford-Fulkerson se baseia em três ideias principais: redes residuais, caminhos aumentantes e cortes. Começa-se com $f\big((u, v)\big) = 0$ para todo $u, v \in V$. O método de Ford-Fulkerson aumenta iterativamente o valor de fluxo, identificando os arcos que podem alterar seu fluxo, consultando suas capacidades. O fluxo aumenta até que não haja mais caminhos aumentantes.

O Algoritmo 25 é demonstrado abaixo.

Algoritmo 25: Algoritmo de Ford-Fulkerson.

Input : um grafo dirigido e ponderado $G = (V, A, c)$, um vértice fonte s, um vértice sorvedouro t, uma rede residual $G_f = (V, A_f, c_f)$

```
1 F ← 0
2 while existir um caminho aumentante p na rede residual de s a t do
      // Identificando a capacidade do caminho e adicionando-o ao fluxo atual
3     f_p ← min{c_f((u,v)) : (u,v) ∈ p}
4     F ← F + f_p
      // Atualizando a capacidade residual
5     foreach (u,v) ∈ p do
6         c_f((u,v)) ← c_f((u,v)) − f_p
7         c_f((v,u)) ← c_f((v,u)) + f_p
8 return F
```

9.5.1 Complexidade

A complexidade de tempo computacional de $Ford - Fulkerson$ depende do fluxo máximo. A Figura 9.6.1 demonstra um exemplo de um pior caso. O tempo para encontrar um caminho de s para t é de $O(|V| + |A|) = O(|A|)$. Devido a característica de depender do valor do fluxo, a quantidade de caminhos que podem ser encontrados é de f^*, ou seja, o valor de fluxo máximo. Então a complexidade de tempo do algoritmo é $O(|A| f^*)$.

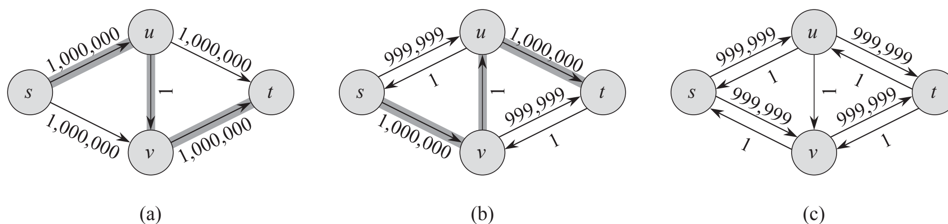


Figura 7 – Exemplo citado por Cormen et al. (2012) para demonstrar o pior caso quanto a complexidade de tempo.

9.6 Edmonds-Karp

Dado a complexidade do algoritmo de Ford-Fulkerson, é impraticável executá-lo para instâncias muito grandes que caem no caso de caminhos aumentantes pequenos. O algoritmo Edmonds-Karp propõe uma pequena adaptação para superar esse problema. No lugar de encontrar um caminho aumentante arbitrário, Edmonds-Karp usa uma busca em largura descrita no Algoritmo 26.

Algoritmo 26: Busca em Largura para Edmonds-Karp.

Input : um grafo dirigido e ponderado $G = (V, A, c)$, um vértice fonte s, um vértice sorvedouro t, uma rede residual $G_f = (V, A_f, c_f)$

```
   // configurando todos os vértices
1  C_v ← false ∀ v ∈ V
2  A_v ← null ∀ v ∈ V
   // configurando o vértice de origem
3  C_s ← true
   // preparando fila de visitas
4  Q ← Fila()
   // Iniciar busca pela fonte.
5  Q.enqueue(s)
   // propagação das visitas
6  while Q.empty() = false do
7      u ← Q.dequeue()
8      foreach v ∈ N⁺(u) do
9          if C_v = false ∧ c_f((u, v)) > 0 then
10             C_v ← true
11             A_v ← u
               // Sorvedouro encontrado. Criar caminho aumentante.
12             if v = t then
13                 p ← (t)
14                 w ← t
15                 while w ≠ s do
16                     w ← A_w
17                     p ← (w) ∪ p
18                 return p
19             Q.enqueue(v)
20 return null
```

Lema 9.6.1. *Se o algoritmo Edmonds-Karp é executar em uma rede de fluxo $G = (V, A, c)$ com a fonte s e o sorvedouro sendo t, então para todos os vértices, exceto s e t, a distância*

do caminho mínimo $\delta_f(s, v)$ na rede residual G_f aumenta monotonicamente com cada aumento de fluxo.

Prova: Página 530 de Cormen et al. (2012).■

Teorema 9.6.2. *Se o algoritmo Edmonds-Karp é executar em uma rede de fluxo $G = (V, A, c)$ com a fonte s e o sorvedouro sendo t, então o número total de aumentos de fluxo executados pelo algoritmo é de no máximo $|V| \cdot |A|$.*

Prova: Página 531 de Cormen et al. (2012).■

9.6.1 Complexidade

Como cada caminho aumentante pode ter um arco crítico[1], o número total de arcos críticos é $O(|V| \cdot |A|)$. Como cada iteração do Ford-Fulkerson pode demandar passar por no máximo $O(|A|)$ arcos, a complexidade do algoritmo Edmonds-Karp é de $O(|V||A|^2)$.

Para pensar

Será que os arcos de retorno ainda fazem sentido para o algoritmo de Edmonds-Karp? Verifique o seguinte exemplo:

[1] Considera-se arco crítico todo arco que define um fluxo para um caminho aumentante (que define o valor de $c_f(p)$ em um caminho p).

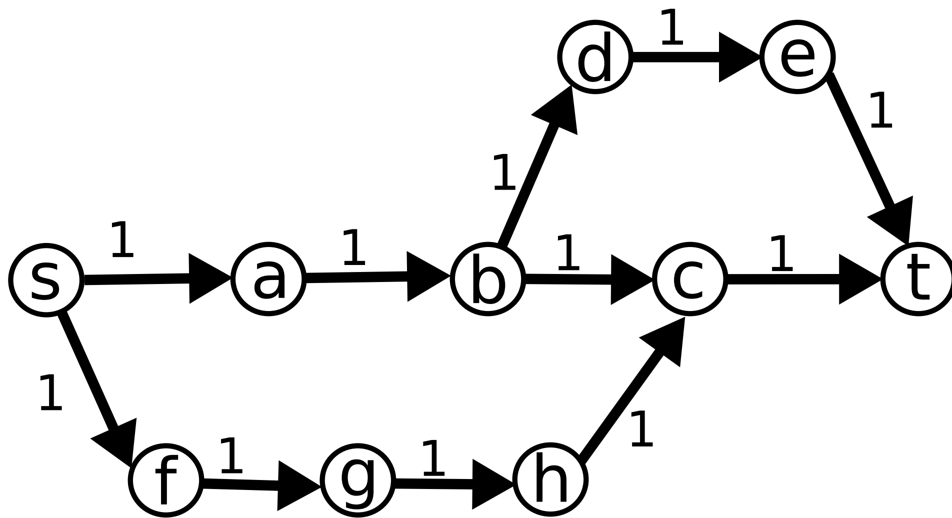


Figura 8 – Exemplo da importância dos arcos de retorno para Edmonds-Karp.

.

CAPÍTULO 10

Emparelhamento

10.1 Emparelhamento Máximo em Grafos Bipartidos

Dado um grafo bipartido não dirigido $G = (V = X \cup Y, E)$, no qual V é o conjunto de vértices bipartidos, e E é o conjunto de arestas. Considere que em cada aresta $\{x, y\}$, $x \in X$ e $y \in Y$. Considere também que X e Y são disjuntos, ou seja, $X \cap Y = \emptyset$. Um emparelhamento M em G é um subconjunto de arestas $M \subseteq E$ tal que cada vértice aparece no máximo uma vez em M. Nesse problema, deseja-se encontrar um conjunto M de maior tamanho possível (KLEINBERG; TARDOS, 2005).

10.1.1 Resolução por Fluxo Máximo

De acordo com Kleinberg e Tardos (2005), uma forma de resolver esse problema é através do uso de algoritmos de fluxo máximo. Para isso,deve-se adaptar o grafo de entrada. O Algoritmo 27 formaliza essa adaptação. A Figura 12 um exemplo de antes e depois da adaptação.

Algoritmo 27: Adaptação de entrada de Emparelhamento Máximo para um algoritmo de Fluxo Máximo.

```
   Input  :um grafo bipartido não-dirigido e não-ponderado G = (V = X ∪ Y, E)
   // Criando conjunto de vértices, considerando um novo vértice de origem s
      e um novo de destino t
1  V′ ← X ∪ Y ∪ {s, t}
   // Criando novo conjunto de arcos
2  A ← {}
   // Transformando as arestas de E em arcos
3  foreach {x, y} ∈ E do
4      considere que x ∈ X e y ∈ Y
5      A ← A ∪ {(x, y)}
   // Criando arcos entre s e x ∈ X
6  foreach x ∈ X do
7      A ← A ∪ {(s, x)}
   // Criando arcos entre s e y ∈ Y
8  foreach y ∈ Y do
9      A ← A ∪ {(y, t)}
   // Definindo os pesos de cada arco
10 criar função w : A → 1
11 G′ ← (V′, A, w)
12 return G′
```

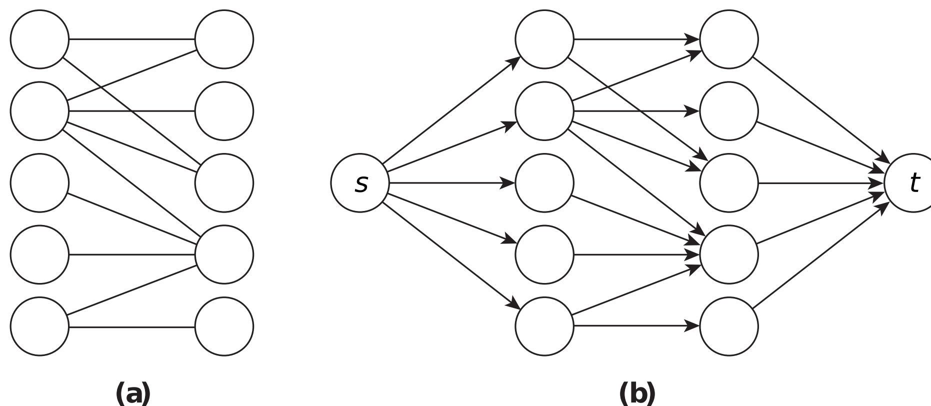


Figura 9 – Exemplo citado por Kleinberg e Tardos (2005) na tranformaçao do grafo bipartido não dirigido para um grafo dirigido e ponderado que, ao ser submetido a um algoritmo de fluxo máximo, obtém-se o emparelhamento máximo.

Depois de adaptada a entrada para o problema de Fluxo Máximo, consegue-se obter o emparelhamento submetendo-a a um algoritmo de fluxo máximo. Pode-se ainda utilizar as informações de fluxo para determinar o conjunto M.

Algoritmo 28: Resolução de Emparelhamento Máximo através um algoritmo de Fluxo Máximo.

Input : um grafo bipartido não-dirigido e não-ponderado $G = (V = X \cup Y, E)$

1 Obter G' a partir do Algoritmo 27 passando por parâmetro o grafo G.
2 Criar rede residual G'_f a partir de G'
3 Executar algoritmo de fluxo máximo passando por parâmetro: G', como rede de fluxo; G'_f, como rede residual, s, como vértice de origem; e t como vértice de destino ou sorvedouro.; `// Criando o conjunto de emparelhamento` M
4 $M \leftarrow \{\}$
5 **foreach** $\{x, y\} \in E$ **do**
6   considere que $x \in X$ e $y \in Y$
7   **if** $c_f\big((x, y)\big) = 0$ **then**
8         $M \leftarrow M \cup \big\{\{x, y\}\big\}$
9 **return** M

10.1.1.1 Complexidade

A tranformação de um grafo bipartido em uma rede de fluxo demanda tempo computacional $O(|V| + |E|)$. Supondo que utilizemos o algoritmo de Ford-Fulkerson para resolver o problema, a mesma possui a complexidade de tempo $O(|A'|f^*)$ $G' = (V', A', w)$. Sabendo que o fluxo máximo é o menor entre $|X|$ e $|Y|$, poderíamos reecrever a função $O(|A'||V|)$. Analisando o Algoritmo 27, sabe-se que $|A'| = |E| + |X| + |Y|$. Então a complexidade de tempo em usar o algoritmo de Ford-Fulkerson é $O\big(|V| + |E| + (|E| + |X| + |Y|)|V|\big) = O\big((|E| + |X| + |Y|)|V|\big) = O(|E||V| + |X||V| + |Y||V|) = O(|E||V|)$.

10.1.2 Algoritmo de Hopcroft-Karp

Um algoritmo mais eficiente do que utilizar algoritmos de Fluxo Máximo é o Hopcroft-Karp. Ele é demonstrado no Algoritmo 29. Para entender o algoritmo, devemos compreender o conjunto de caminhos aumentantes em seu contexto.

Um caminho aumentante alternante é todo o caminho possui início em vértice livre em X e o fim em um vértice livre em Y. É dito vértice livre, um vértice que não está em M. Diferente de outros caminhos que foram vistos na disciplina, um caminho aumentante aqui é uma sequência de arestas $p = < e_1, e_2, \ldots, e_m >$ no qual $e_1, e_2, \ldots, e_m \in E$.

Na linha 4 do algoritmo, utiliza-se o operador $\oplus$ (XOR). Em conjuntos $A \oplus B =$

$(A\backslash B) \cup (B\backslash A)$.

Algoritmo 29: Algoritmo de Hopcroft-Karp.

Input : um grafo bipartido não-dirigido e não-ponderado $G = (V = X \cup Y, E)$

1 $M \leftarrow \{\}$
2 **repeat**
3 $\quad P \leftarrow$ conjunto de caminhos aumentantes alternantes $p_1, p_2, \ldots, p_k$
4 $\quad M \leftarrow M \oplus \bigcup_{p \in P} p$
5 **until** $P = \{\}$
6 **return** M

10.1.2.1 Algoritmo detalhado

Algoritmo 30: Algoritmo de Hopcroft-Karp detalhado.

Input : um grafo bipartido não-dirigido e não-ponderado $G = (V = X \cup Y, E)$

1 $D_v \leftarrow \infty \; \forall v \in V$
2 $mate_v \leftarrow$ **null** $\forall v \in V$

`// tamanho do emparelhamento`

3 $m \leftarrow 0$
4 **while** $BFS(G, mate, D) =$ ***true*** **do**
5 $\quad$ **foreach** $x \in X$ **do**
6 $\quad\quad$ **if** $mate_x =$ ***null*** **then**
7 $\quad\quad\quad$ **if** $DFS(G, mate, x, D) =$ ***true*** **then**
8 $\quad\quad\quad\quad m \leftarrow m + 1$
9 **return** $(m, mate)$

Algoritmo 31: BFS

Input : um grafo bipartido não-dirigido e não-ponderado $G = (V = X \cup Y, E)$, um vetor de emparelhamento $mate$, um vetor de distâncias D

```
1  Q ← Fila()
2  foreach x ∈ X do
3      if mate_x = null then
4          D_x ← 0
5          Q.enqueue(x)
6      else
7          D_x ← ∞
8  D_null ← ∞
9  while Q.empty() = false do
10     x ← Q.dequeue()
11     if D_x < D_null then
12         foreach y ∈ N(x) do
13             if D_{mate_y} = ∞ then
14                 D_{mate_y} ← D_x + 1
15                 Q.enqueue(mate_y)
16 return D_null ≠ ∞
```

Algoritmo 32: DFS

Input : um grafo bipartido não-dirigido e não-ponderado $G = (V = X \cup Y, E)$, um vetor de emparelhamento $mate$, um vértice $x \in X$, um vetor de distâncias D

```
1  if x ≠ null then
2      foreach y ∈ N(x) do
3          if D_{mate_y} = D_x + 1 then
4              if DFS(G, mate, mate_y, D) = true then
5                  mate_y ← x
6                  mate_x ← y
7                  return true
8      D_x ← ∞
9      return false
10 return true
```

10.1.2.2 Complexidade

Cada fase do algoritmo consiste em uma busca em largura e uma em profundidade. Então, uma única fase pode utilizar tempo $O(|E|)$. Como há apenas $\sqrt{|V|}$ caminhos aumentantes alternante, a complexidade de tempo é $O(\sqrt{|V|}|E|)$.

10.2 Emparelhamento Máximo em Grafos

O emparelhamento máximo em grafos não-dirigidos e não-ponderados é o problema de encontrar o maior conjunto independente de arestas. A chave para este emparelhamento está nos caminhos aumentantes alternantes. No entanto, ao tentar encontrar esses caminhos, pode-se encontrar um ciclo que não pode ser excluído. Em 1965, Edmonds descobriu uma forma de resolver isso, mesclando os vértices desse ciclo recursivamente (*blossom*), tornando a instância de entrada menor (SCHRIJVER, 2004). Nessa seção, se conhecerá o algoritmo presente no livro Papadimitriou e Steiglitz (1982), com complexidade de tempo $O(|V|^4)$, mas há algoritmos muito mais eficientes: Algoritmo de Balinski $O(|V|^3)$, Algoritmo de Even-Kariv $O(|V|^{5/2})$, e o Algoritmo de Micali-Vazirani $O(\sqrt{|V|}|E|)$.

Algoritmo 33: Emparelhamento máximo para grafos não-dirigidos e não-ponderados.

```
Input   : um grafo não-dirigido e não-ponderado G = (V, E)
   // Vetor que identifica o par do emparelhamento
1  M_v ← null ∀ v ∈ V
   // Vetor que identifica se o vértice foi visitado ou não
2  C_v ← false ∀ v ∈ V
   // Criando o vetor de expostos
3  X_v ← null ∀ v ∈ V
   // Criando o vetor de vértices vistos
4  S_v ← false ∀ v ∈ V
   // Criando o vetor de rótulos
5  L_v ← null ∀ v ∈ V
6  while ∃u ∈ V : C_u = false ∧ M_u = null do
7      u ← selecionar um u ∈ V, o qual C_u = false ∧ M_u = null
8      C_u ← true
9      A ← {}
       // populando o vetor de expostos
10     X_v ← null ∀ v ∈ V
       // Fazer no sentido v para w e w para v
11     foreach {v, w} ∈ E do
12         if M_w = null ∧ w ≠ u then
13             X_v ← w
14         else
15             if M_w ∉ {v, null} then
16                 A ← A ∪ {(v, M_w)}
17     S_v ← false ∀ v ∈ V
18     Q ← {u}
19     L_u ← null
20     if X_u ≠ null then
21         AumentanteAlternante(G, u, M, X, L)
22     else
23         while Q ≠ {} do
24             v ← selecione um vértice em Q
25             Q ← Q \ {v}
26             foreach w ∈ V : L_w = null ∧ (v, w) ∈ A do
27                 Q ← Q ∪ {w}
28                 L_w ← v
29                 S_{M_w} ← true
30                 if X_w ≠ null then
31                     AumentanteAlternante(G, w, M, X, L)
32                     Q ← {}
33                     break
34                 else
35                     if S_w = true then
36                         Blossom(G, w)
37 return M
```

Algoritmo 34: AumentanteAlternante

```
Input  :um grafo não-dirigido e não-ponderado G = (V, E), um vértice v, os vetores M,
        X, L
1 if L_v = null then
2 |  M_v ← X_v
3 |  X_{M_v} ← v
4 else
5 |  X_{L_v} ← M_v
6 |  M_v ← X_v
7 |  M_{X_v} ← v
8 |  AumentanteAlternante(G, L_v, M, X, L)
```

Algoritmo 35: Blossom

```
Input  :um grafo não-dirigido e não-ponderado G = (V, E), um vértice b, os vetores M,
        X, L, B, Y
1 T ← buscar conjunto de vértices que estão no ciclo envolvendo b, incluindo-o
2 Y_b ← ciclo de b
3 foreach t ∈ T do
4 |  B_t ← b
  // Mesclando os vértices de B
5 Substituir qualquer instância do nodo x ∈ B no grafo auxiliar A, na fila Q e no vetor label
   por um novo vértice v_b que representa o "blossom" de b
```

Algoritmo de Edmonds:

Algoritmo 36: Algoritmo de Edmonds (Blossom).

Input : um grafo não-dirigido $G = (V, E)$, um conjunto de vértices expostos (livres) S, estrutura que mapeia quem está emparelhado para cada vértice W, vértices marcados C^V, arestas marcadas C^E, identidade de cada vértice I, dados dos blossoms B, nível $l = \langle 0 \rangle$

```
1 D_v ← ∞ ∀ v ∈ V
2 mate_v ← null ∀ v ∈ V
  // tamanho do emparelhamento
3 m ← 0
4 while BFS(G, mate, D) = true do
5     foreach x ∈ X do
6         if mate_x = null then
7             if DFS(G, mate, x, D) = true then
8                 m ← m + 1
9 return (m, mate)
```

CAPÍTULO 11

Coloração de Grafos

O problema da coloração de grafos busca encontrar um conjunto de "cores" a serem atribuídos aos vértices, sem que vértices adjacentes tenham a mesma cor. O problema tem como entrada um grafo não-dirigido e não-ponderado $G = (V, E)$. Sua versão de decisão busca recebe além de G um valor inteiro k e deseja-se saber se é possível encontrar um conjunto de "cores" (ou valores/números cromáticos) com tamanho menor ou igual a k. Na sua versão de otimização, geralmente tenta-se encontrar o conjunto mínimo de cores para G.

O problema de decisão com $k > 3$ é NP-Completo e o problema de otimização é NP-Difícil o que significa que não existe um algoritmo executado em tempo polinomial para os mesmos a não ser que $P = NP$.

11.1 Aplicações

As aplicações da coloração de grafos estão associadas a problemas de alocaçao. Seguem alguns exemplos citados por Kleinberg e Tardos (2005):

- Suponha uma coleção de n tarefas em um sistema que pode processá-los paralelamente, mas certos pares de tarefas não podem ser processados ao mesmo tempo, pois usam algum mesmo recurso. Deseja-se executar em k unidades de

tempo todos os recursos. Então, cria-se um grafo G no qual os vértices são as tarefas e as arestas ligam duas tarefas que não podem ser executadas no mesmo tempo. Ao submeter G e k a um algoritmo de decisão da coloração, pode-se obter em que unidade de tempo cada tarefa deve ser executada.

- Suponha que deseja-se construir um compilador e no processo de compilação deve-se associar cada variável a um de k resgistradores disponíveis. Aqui, para o grafo G, os vértices seriam as variáveis e as arestas conectam dois vértices correspondentes a duas variáveis que estão em uso ao mesmo tempo em algum momento do programa. Um algoritmo de coloração de decisão ajudará a escolher qual o registrador seguro para cada variável.

- Deseja-se um dos k comprimentos de onda (transmissão sem fio) para cada n dispositivos, mas se dois dispositivos estiverem muito próximos, dois comprimentos de onda devem ser atribuídos para evitar interferências. Um algoritmo de coloração indicará qual comprimento de onda deve ser atribuído a cada um dos n dispositivos.

11.2 Heurística DSATUR

Considerando a dificuldade computacional de resolver a coloração mínima, há diversos métodos heurísticos propostos na literatura para atribuir uma coloração admissível, relaxando o requisito de se obter o menor número de cores possível. Um destes métodos é o DSATUR, proposto em Brélaz (1979). Se trata de um método guloso, no qual seleciona-se iterativamente um vértice não colorido para obter uma cor. Como critério, seleciona-se o vértice não-colorido com o maior valor na característica chamada de "saturação do grau". A saturação do grau de um vértice é obtida pela quantidade de cores em vértices vizinhos. Quando há um empate neste critério, o vértice de maior grau é selecionado. O método DSATUR é apresentado no Algoritmo 37.

Algoritmo 37: Método DSATUR

```
Input   :um grafo não-dirigido e não-ponderado G = (V, E)
1  X_v ← 0 ∀ v ∈ V
2  D_v ← 0 ∀ v ∈ V
3  i ← 0
4  while i < |V| do
       // Selecionando o vértice a ser colorido
5      u ← argmax_{v∈V} {D_v : X_v = 0}
6      S ← ⋃_{v∈V: X_v=0 ∧ D_v=D_u} {v}
7      if |S| > 1 then
8          w ← argmax_{v∈S} {d(v)}
9      else
10         w ← S_0
11     u ← w
       // Colorindo o vértice u com uma cor admissível
12     cviz ← ∪_{v∈N(u)} {X_v}
13     cand ← {1, ..., maximo(cviz) + 1} − cviz
14     cor ← min(cand)
15     X_u ← cor
       // Atualizando "saturação do grau" dos vértices vizinhos de u
16     foreach v ∈ N(u) do
17         cviz' ← ∪_{w∈N(v)} {X_w}
18         D_v ← |cviz' − {0}|
19     i ← i + 1
20 return X
```

11.3 Algoritmo de Lawler

O algoritmo de Lawler resolve o problema de otimização e encontra o número correspondente a menor quantidade de cores em G (LAWLER, 1976b). O algoritmo usa os conjuntos independentes maximais[1] identificados a partir de subgrafos formados por subconjuntos de vértices.

O algoritmo de Lawler é representado no Algoritmo 38. Nesse algoritmo, $\mathbb{S}$ possui todos os subconjuntos de vértices possíveis ($2^{|V|}$ subconjuntos). Cada subconjunto é utilizado para identificar qual a coloração mínima considerando os vértices que pertence a ele. Isso se baseia na recorrência

[1] Um conjunto independente maximal é um conjunto de vértices no qual cada vértice não está conectado a cada outro, e não exista outro vértice no grafo que poderia pertencer a esse conjunto conservando essa propriedade.

$$OPT\big(G=(S,E)\big)=\begin{cases}0 & \text{se S=\{\},}\\ 1+\mathbf{min}_{I\in\mathbf{I}(G)}\Big\{OPT\Big(G'=\big(S\backslash I,\{\{u,v\}\in E: u,v\in S\backslash I\}\big)\Big)\Big\} & \text{caso contrário.}\end{cases}$$

Nessa recorrência, **I**(**G**) é um algoritmo que retorna os conjuntos independentes maximais do grafo (ou subgrafo) G.

Ainda quanto ao algoritmo, a função $f : 2^V \to \mathbb{Z}^+_*$ mapeia um subconjunto do conjunto de vértices em um número inteiro obtido a partir de uma representação binária ordenada do conjunto de vértices. O vetor X é indexado de acordo com esse número inteiro obtido por $f(\cdot)$. $X_{f(S)}$ possui o coloração mínima possível considerando o subconjunto de vértices S.

Algoritmo 38: Algoritmo de Lawler

Input :um grafo não-dirigido e não-ponderado $G=(V,E)$

1. $X \leftarrow$ vetor indexado entre 0 e $2^{|V|}-1$
2. $X_0 \leftarrow 0$
3. $\mathbb{S} \leftarrow 2^V$
   `// Suponha que` $\mathbb{S}$ `corresponde a ordem crescente dada por` $f(\cdot)$
4. **foreach** $S \in \mathbb{S}\backslash\{\{\}\}$ **do**
5. $s \leftarrow f(S)$
6. $X_s \leftarrow \infty$
7. $G' \leftarrow \big(S,\{\{u,v\}\in E: u,v\in S\}\big)$
8.   **foreach** $I \in \mathbf{I}(G')$ **do**
9. $i \leftarrow f(S\backslash I)$
10.   **if** $X_i + 1 < X_s$ **then**
11. $X_s \leftarrow X_i + 1$
12. **return** $X_{2^{|V|}-1}$

11.3.1 Complexidade

O algoritmo demanda complexidade de tempo $O(|V||E|2.4423^{|V|})$. Isso devido ao uso da função de encontrar o conjunto indenpendente. Para maiores detalhes sobre esse e outros algoritmos de coloração de grafos, veja De Lima e Carmo (2019).

11.4 Listar Conjuntos Independentes Maximais

Algoritmo 39: Listar Conjuntos Independentes Maximais

```
Input   : um grafo não-dirigido e não-ponderado G = (V, E)
   // S é uma lista de todos os subconjuntos possíveis de vértices de V,
      ordenados na ordem decrescente pela cardinalidade (tamanho) de cada
      subconjunto.
1  S ← 2^V
2  R ← {}
3  foreach X ∈ S do
       // Verifica se X é um conjunto independente
4      c ← true
5      foreach v ∈ X do
6          foreach u ∈ X do
7              if {u, v} ∈ E then
8                  c ← false
9                  break
10     if c = true then
11         remover todos os subconjuntos de X de S
12         R ← R ∪ {X}
13 return R
```

CAPÍTULO 12

Caminho Crítico

Método do Caminho Crítico (CPM - *Critical Path Method*) e a técnica de revisão e avaliação de programa (PERT - *Program Evaluation and Review Technique*) são métodos para tratar grafos de planejamento de projetos. Um projeto é definido como um conjunto de atividades, sendo que cada atividade consome tempo e recursos determinados. O objetivo dos métodos é fornecer maneiras analíticas de resolver a programação das atividades do projeto (TAHA, 2006).

Tanto para o CPM quanto para PERT, divide-se as fases de planejamento de projeto nas seguintes etapas:

1. Definição das atividades do projeto, seus tempos e recursos;
2. Construção do grafo representando as atividades seus tempos de execução as interdependências entre as tarefas;
3. Cálculo do caminho crítico;
4. Programação temporal das tarefas.

Segundo Taha (2006), CPM e PERT foram desenvolvidas independentemente uma da outra. A diferença entre elas é que CPM considera durações determinísticas para as atividades e PERT durações probabilísticas.

Cada atividade do projeto será um arco no grafo. Os vértices do grafo estabelecem as precedências entre as atividades. Há duas regras para se construir o grafo:

1. Cada atividade é representada por apenas um arco;

2. Para manter a corretude das relações de precedência, precisa-se responder às seguintes perguntas: (*i*) quais atividades devem preceder imediatamente a atividade atual; (*ii*) quais atividades devem vir depois da atividade atual; e (*iii*) quais atividades devem ocorrer concorrentemente da atividade atual.

Para responder as perguntas da segunda regra, pode-se ter que criar atividades fictícias para garantir correta precedência. Na Figura 10, há um esquema de como resolver conflito de recorrência entre duas atividades, criando uma atividade inexistente com duração que demanda 0 unidades de tempo (arcos tracejados), de acordo com Taha (2006).

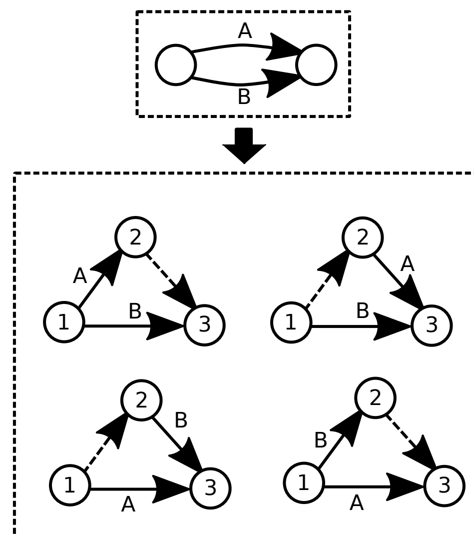


Figura 10 – Utilização de atividade fictícia para representar atividades concorrentes (TAHA, 2006).

Algoritmo 40: Listar Conjuntos Independentes Maximais

```
Input   :um grafo não-dirigido e não-ponderado G = (V, E)
// S é uma lista de todos os subconjuntos possíveis de vértices de V,
   ordenados na ordem descrescente pela cardinalidade (tamanho) da cada
   subconjunto.
1  S ← 2^V
2  R ← {}
3  foreach X ∈ S do
       // Verifica se X é um conjunto independente
4      c ← true
5      foreach x ∈ X do
6          foreach u ∈ X do
7              if {u, v} ∈ E then
8                  c ← false
9                  break
10     if c = true then
11         remover todos os subconjuntos de X de S R ← R ∪ {X}
12 return R
```

12.1 Listar Conjuntos Independentes Maximais

12.2 Cálculo do caminho crítico

De acordo com Taha (2006), executam-se cálculos com o objetivo de encontrar a duração total necessária para concluir o projeto e a classificação das atividades entre críticas[1] ou não-críticas.

Considerando que evento é um ponto no tempo em que algumas tarefas são concluídas e outras são iniciadas, define-se:

- E_j é o tempo mais cedo da ocorrência de um evento j;
- T_j é o tempo mais tarde da ocorrência de um evento j;
- D_{ij} é a duração da atividade representada pelo arco (i, j).

O cálculo do caminho crítico envolve dois passos: o *forward-pass*, que define E_j, e o *backward-pass*, que define T_j.

[1] Uma atividade é considerada crítica se há apenas como executá-la em um tempo inicial e final determinados cujo intervalo é igual a duração da referida tarefa.

Forward-pass

Inicialmente, determina-se $E_1 = 0$ para indicar que o evento inicial é o ponto de partida.

Depois, considera-se o vértice do evento j. Dados de arcos/atividades entrantes no vértice j, calcula-se

$$E_j = \mathbf{max}_{v \in N^-(j)} \{E_v + D_{vj}\}. \tag{12.1}$$

O processo é concluído quando todos os vértices/eventos j tiverem seus valores E_j calculados.

Backward-pass

Após a conclusão do *forward-pass*, inicia-se o *backward-pass* do vértice n, que representa o evento no qual todas as atividades já foram concluídas.

Inicialmente, define-se $T_n = E_n$. Depois, para cada evento j, calcula-se

$$T_j = \mathbf{min}_{v \in N^+(j)} \{T_v - D_{jv}\}. \tag{12.2}$$

O processo é concluído no evento 1, quando define-se que $E_1 = T_1 = 0$.

Folgas

De acordo com Taha (2006), folgas ou flutuações são os tempos disponíveis dentro do intervalo de tempo para uma atividade não crítica. Geralmente calcula-se a folga total e a folga livre.

A folga total é vinculada a cada atividade e é definida como $TF_{ij} = T_j - E_i - D_{ij}$. Ela representa o excesso de tempo definido entre a ocorrência mais cedo do evento i e a ocorrência mais tarde do evento j considerando a duração da atividade (i, j).

A folga livre é também vinculada a cada atividade e é definida como $FF_{ij} = E_j - E_i - D_{ij}$. Ela representa o excesso de tempo definido desde a ocorrência mais cedo do vento i até a ocorrência mais cedo do evento j considerando a duração da atividade (i, j).

Agradecimentos

Agradeço o Prof. Antônio Carlos Mariani pelo apoio e suporte quando a disciplina de Grafos fora cedida aos meus cuidados.

Referências

BRéLAZ, D. New methods to color the vertices of a graph. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 22, n. 4, p. 251–256, abr. 1979. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/359094.359101>. Citado na página 104.

CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012. Citado 18 vezes nas páginas 5, 7, 19, 23, 25, 28, 41, 69, 71, 79, 80, 82, 86, 87, 88, 89, 91 e 119.

Da Silva Mendes, R. F.; De Santiago, R.; Lamb, L. C. Novel parallel anytime a* for graph and network clustering. In: *2018 IEEE Congress on Evolutionary Computation (CEC)*. [S.l.: s.n.], 2018. p. 1–6. Citado na página 67.

De Lima, A. M.; CARMO, R. Exact Algorithms for the Graph Coloring Problem. *Revista de Informática Teórica e Aplicada*, v. 25, n. 4, p. 57, 2019. ISSN 01034308. Citado na página 106.

GIUSCA, B. *Map of Konigsberg in Euler's time showing the actual layout of the seven bridges, highlighting the river Pregel and the bridges*. 2005. Disponível em: <https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg#/media/File:Konigsberg_bridges.png>. Citado na página 32.

GROSS, J. T.; YELLEN, J. *Graph Theory and Its Applications*. FL: CRC Press, 2006. Citado na página 31.

KLEINBERG, J.; TARDOS, E. *Algorithm Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN 0321295358. Citado 3 vezes nas páginas 93, 94 e 103.

LAWLER, E. *Combinatorial optimization - networks and matroids*. New York: Holt, Rinehart and Winston, 1976. Citado na página 61.

LAWLER, E. L. A note on the complexity of the chromatic number problem. *Inf. Process. Lett.*, v. 5, p. 66–67, 1976. Citado na página 105.

MARIANI, A. C. *Problemas de Travessia*. 2019. Disponível em: <https://www.inf.ufsc.br/grafos/temas/travessia/travessia.htm>. Acesso em: 09 abr 2019. Citado 4 vezes nas páginas 64, 65, 66 e 67.

NETTO, P. O. B. *Grafos: teoria, modelos, algoritmos*. São Paulo: Edgard Blucher, 2006. Citado 4 vezes nas páginas 12, 13, 16 e 66.

PAPADIMITRIOU, C. H.; STEIGLITZ, K. *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, NJ: Prentice Hall, 1982. ISBN 0131524623 9780131524620 0486402584 9780486402581. Citado na página 98.

SCHRIJVER, A. *A Course in Combinatorial Optimization*. [S.l.: s.n.], 2004. Citado na página 98.

TAHA, H. A. *Operations Research: An Introduction (8th Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN 0131889230. Citado 4 vezes nas páginas 109, 110, 111 e 112.

APÊNDICE A

Revisão de Matemática Discreta

Conjuntos é uma coleção de elementos sem repetição em que a sequência não importa. No Brasil, utilizamos a seguinte notação para enumerar todos os elementos de um conjunto. Na Equação (A.1), é possível visualizar a representação de um conjunto denominado A, formado pelos elementos e_1, e_2, ..., e_n. Devido ao uso da vírgula como separador de decimais, usa-se formalmente o ponto-e-vírgula. Para essa disciplina, podemos utilizar a vírgula como o separador de elementos em um conjunto, desde que utilizados o ponto como separador de decimais[1]. Para dar nome a um conjunto, geralmente utiliza-se uma letra maiúscula ou uma palavra com a inicial em maiúscula.

$$A = \left\{e_1; e_2; \ldots; e_n\right\} \tag{A.1}$$

Há duas formas de definir conjuntos. A forma por enumeração por elementos, utiliza notação semelhante a da Equação (A.1). São exemplos de definição de conjuntos por enumeração:

- $N = \{\diamondsuit, \spadesuit, \heartsuit, \clubsuit\}$;
- $V = \{a, e, i, o, u\}$;
- $G = \{\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \theta, \iota, \kappa, \lambda, \mu, \nu, \xi, \pi, \rho, \sigma, \tau, \upsilon, \phi, \chi, \psi, \omega\}$;
- $R = \{-100.9, 12.432, 15.0\}$;
- $D = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

A forma por descrição de propriedades utiliza-se de uma notação que evidencia a natureza de cada elemento pela descrição de um em um formato genérico. Por exemplo o conjunto D, descrito na Equação (A.2), denota um conjunto com os mesmos elementos em $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

$$D = \{x \in \mathbb{Z} | x > 1 \land x \leq 10\} \tag{A.2}$$

. Então para que complicar utilizando uma notação não enumerativa? Por dois motivos: por questões de simplicidade, dado a quantidade de conjuntos; ou para representar conjuntos infinitos, como no exemplo dos inteiros pares $Pares = \{x \in \mathbb{Z} | x \equiv 0 \pmod 2\}$. Para o conjunto dos pares, ainda podemos utilizar uma descrição mais informal, mas

[1] Nas anotações presentes nesse documento, utiliza-se a "notação americana". Para a Equação (A.1) , teria-se $A = \left\{e_1, e_2, \ldots, e_n\right\}$.

que é dependente da conhecimento sobre a linguagem Portuguesa: $Pares = \{x \in \mathbb{Z} | \ x$ *é inteiro e par* $\}$.

Para denotar a cardinalidade (quantidade de elementos) de um conjunto, utilizamos o símbolo “|”. Para os conjuntos apresentados acima, é correto afirmar que:

- $|N| = 4$;
- $|V| = 5$;
- $|R| = 3$;
- $|D| = 10$;
- $|Pares| = \infty$.

A cardinalidade pode ser utilizada para identificar quantos símbolos são necessários para representar um elemento. Por exemplo, $|12,66| = 5$

Para denotar conjuntos vazios, adota-se duas formas de representação: $\{\}$ ou $\emptyset$. Utilizando o operador de cardinalidade, têm-se $|\{\}| = |\emptyset| = 0$.

Como principais operações entre conjuntos, pode-se destacar:

- União ($\cup$): união de dois conjuntos. Exemplo: $\{1,2,3,4,5\} \cup \{2,4,6,8\} = \{1,2,3,4,5,6,8\}$;
- Intersecção ($\cap$): intersecção de dois conjuntos. Exemplo: $\{1,2,3,4,5\} \cup \{2,4,6,8\} = \{2,4\}$;
- Diferença ($-$ ou $\backslash$): diferença de dois conjuntos. Exemplo $\{1,2,3,4,5\} \backslash \{2,4,6,8\} = \{1,3,5\}$;
- Produto cartesiano ($\times$): Exemplo $\{1,2,3\} \times \{A,B\} = \{(1,A),(2,A),(3,A),(1,B),(2,B),(3,B)\}$;
- Conjunto de partes (ou *power set*): o conjunto de todos os subconjuntos dos elementos de um conjunto. Para o conjunto $A = \{1,2,3\}$ o conjunto das partes seria $2^A = P(A) = \{\{\},\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$.

Funções são representadas de forma diferente na matemática discreta. Busca-se estabelecer a relação entre um conjunto de domínio (entrada da função) e um contradomínio (resposta da função). A Equação (A.3) exibe a forma como é utilizada para formalizar uma função. Nesse formato, passa-se a natureza da entrada e da saída de um problema. Por exemplo, a função que gera a correspondência entre o domínio dos inteiros positivos em base decimal para base binária seria $f : x \in Z^+ \rightarrow \{0,1\}^{\log_2(|x|+1)}$.

$$\text{nome da funcao} : \text{dominio} \rightarrow \text{contradominio} \tag{A.3}$$

Para representar uma coleção de itens onde a sequência importa e a repetição pode ocorrer, utiliza-se as tuplas. Uma tupla é representada da forma demonstrada na Equação (A.4).

$$A = (e_1, e_2, \ldots, e_n) \tag{A.4}$$

. Um exemplo de uma tupla, pode ser lista de chamada de uma turma ordenada lexicograficamente.

APÊNDICE B

Estruturas de Dados Auxiliares

B.1 Conjuntos Disjuntos

Para o algoritmo de Kruskal, é interessante manter um conjunto de conjuntos disjuntos, no qual cada subconjunto representa um dos elementos de uma subárvore que comporá a Árvore Geradora Mínima. Para representar esses conjunto disjuntos, utiliza-se uma estrutura de dados sugerida no livro de (CORMEN et al., 2012).

Para a estrutura de conjuntos disjuntos, imagina-se que cada conjunto disjunto será uma árvore composta por nodos que serão representados pela tripla $(p, rank, data)$, na qual p é o nodo pai, $rank$ é um limite superior da altura da árvore e $data$ é o dado armazenado naquele nodo.