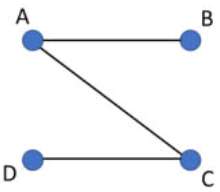
Algoritmos em Grafos

Paulo Cezar Pinto Carvalho

1. Grafos e grafos direcionados

Um grafo é uma estrutura matemática que descreve conexões entre objetos. É comum representarmos um grafo geometricamente. Os objetos (chamados de **nós** ou **vértices**) são representados por pontos e a existência de conexões entre eles (chamadas de **arestas** ou **arcos**) são representadas por linhas.

Exemplo: **A**na, **B**eatriz, **C**arlos e **D**ora estão em um grupo de WhatsApp. **A**na e **B**eatriz, **A**na e **C**arlos, e **C**arlos e **D**ora se conhecem pessoalmente. Utilizando as iniciais dos nomes para nomear as pessoas, essas conexões são representadas pelo grafo de vértices A, B, C e D e de arestas AB, BC e CD, representado geometricamente abaixo:



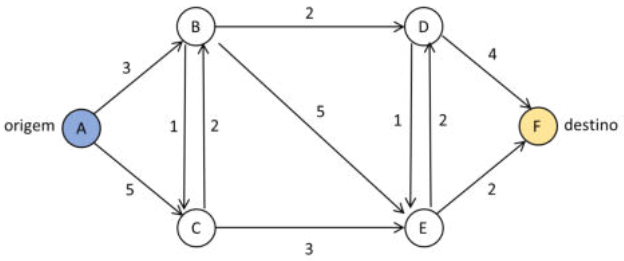
No exemplo anterior, as conexões entre os objetos são mútuas (ou seja, correspondem a um par **não ordenado** de objetos). Há situações, no entanto, em que é desejável modelar conexões **de** um objeto **para** outro (e não conexões **entre** objetos). Neste caso, essas conexões correspondem a um par **ordenado** de objetos e são representados por uma seta; grafos assim são chamados de grafos **orientados** ou **direcionados**.

Exemplo: O grafo abaixo ilustra as ruas de um bairro. Os vértices são as esquinas e as arestas correspondem a ruas e indicam o sentido permitido de tráfego. Quando a rua é de mão dupla, há setas em ambos os sentidos.

Em muitas situações envolvendo grafos, é natural associar um valor a cada aresta do grafo. Na situação anterior, podemos, por exemplo, associar a cada aresta o tempo gasto no deslocamento ao longo de cada aresta; esta é a informação essencial para o problema de encontrar a rota ótima entre dois pontos em uma cidade, que resolvemos diariamente em nossos celulares. Examinamos este problema em mais detalhe a seguir.

2. O problema do caminho mínimo

Dado um grafo direcionado, com um custo associado a cada aresta, desejamos encontrar o caminho de menor custo conectando um ponto de origem e um ponto de destino. No caso da rota ótima em uma cidade, o custo associado a uma aresta é o tempo de deslocamento ao longo dela; em outras situações, pode ser o comprimento ou o custo de uma passagem entre os extremos da aresta. A figura abaixo mostra um exemplo, em que o vértice de origem é A e o de destino é F.



Embora tenhamos dito acima que temos um grafo direcionado, o problema se aplica também a grafos não direcionados, já que essa situação é um caso particular do anterior, bastando considerar que a cada aresta não direcionada correspondem arestas direcionadas em ambas as direções.

3. Representação computacional de um grafo

Antes de descrever e analisar algoritmos que resolvam o problema do caminho mínimo, é necessário observar que, para representar um grafo em um computador, precisamos descrever os vértices e as arestas que os conectam, com seu respectivo custo. Abaixo, vemos duas possíveis formas de se fazer isto.

- Pela lista de adjacências, que, como o nome indica, lista as arestas que partem de cada vértice. Para o gráfico do exemplo, a lista de adjacências é:

De	Para	Custo
A	B	3
	C	5
B	C	1
	D	2
	E	5
C	B	2
	E	3
D	E	1
	F	4
E	D	2
	F	2

- Pela matriz de adjacências, que é uma matriz **A** = $|V| \times |V|$ (onde $|V|$ denota o número de vértices), em que o valor de cada elemento a_{ij} é o custo da aresta ligando o vértice *i* ao vértice *j*; se não estiver tal aresta, indicamos ∞ naquela posição. Para o gráfico do exemplo, a matriz de adjacências é:

	A	B	C	D	E	F
A	∞	3	5	∞	∞	∞
B	∞	∞	1	2	5	∞
C	∞	2	∞	∞	3	∞
D	∞	∞	∞	∞	1	4
E	∞	∞	∞	2	∞	2
F	∞	∞	∞	∞	∞	∞

4. Algoritmo de força bruta

O problema do caminho mínimo admite uma solução por força-bruta, que consiste em enumerar cada sequência de pontos intermediários entre A e B, verificar se cada uma delas é possível e, em caso afirmativo, calcular seu custo. No caso do nosso exemplo, em que há 6 vértices, a tabela abaixo mostra a quantidade de caminhos a serem examinados e quais deles de fato existem no grafo, com seu custo.

Número de nós intermediários	Quantidade de possíveis caminhos	Caminhos viáveis	Custo total
0	1	--	--
1	4	--	--
2	4.3 = 12	ABDF	9
		ABEF	10
		ACEF	10
3	4.3.2 = 24	ABCEF	9
		ABDEF	**8**
		ABEDF	14

		ACBDF	13
		ACBEF	14
		ACEDF	14
4	4.3.2.1 = 24	ABCEDF	13
		ACBDEF	12
		ACBEDF	18

Portanto, há 1 + 4 + 12 + 24 + 24 = 65 possibilidades a examinar, grande parte das quais corresponde a caminhos inexistentes, mas que o algoritmo teria que verificar que de fato é assim. Examinando a tabela, observamos que o caminho de custo mínimo é ABDEF, de custo igual a 8.

Mas será que este é um algoritmo prático para resolver problemas de tamanho real? Para responder a esta pergunta, vamos contar quantos caminhos o algoritmo tem que examinar em um grafo com $|V|$ vértices, em que há $n = |V| - 2$ possíveis intermediários. Vamos contar separadamente os caminhos com as diversas possibilidades para o número de vértices intermediários.

- Passando por todos os n vértices: $n.\ (n-1).\ \ldots.\ 1 = n!$
- Passando por $n-1$ vértices: $n.\ (n-1).\ \ldots.\ 2 = n!$
- Passando por $n-2$ vértices: $n.\ (n-1).\ \ldots.\ 3 = \frac{n!}{2!}$
- Passando por $n-3$ vértices: $n.\ (n-1).\ \ldots.\ 4 = \frac{n!}{3!}$
- ...
- Passando por 1 vértice: $n = \frac{n!}{(n-1)!}$
- Passando por nenhum vértice: $1 = \frac{n!}{n!}$

Somando os números de possibilidades obtemos

$$n!\left(1 + 1 + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!}\right)$$

O valor entre parênteses se aproxima do número e quando n cresce. De modo geral, para todo $n > 1$, esse valor está entre 2 e 3 e, portanto, o número de caminhos está entre $2n!$ e $3n!$.

Por exemplo, se o grafo possui 30 vértices, o número de caminhos a examinar é superior a $2.\ 28! = 6 \cdot 10^{29}$. Um computador capaz de examinar 1 bilhão (10^9) de caminhos por segundo, necessitaria mais de $6 \cdot 10^{20} = 2 \cdot 10^{13}$ anos = 20 trilhões de anos para examinar todas as possibilidades!

Isto mostra que, para que o problema possa ser resolvido de maneira prática (por exemplo, ao consultar uma rota em um celular), o problema precisa ser resolvido de modo muito mais eficiente.

5. O Algoritmo de Dijkstra

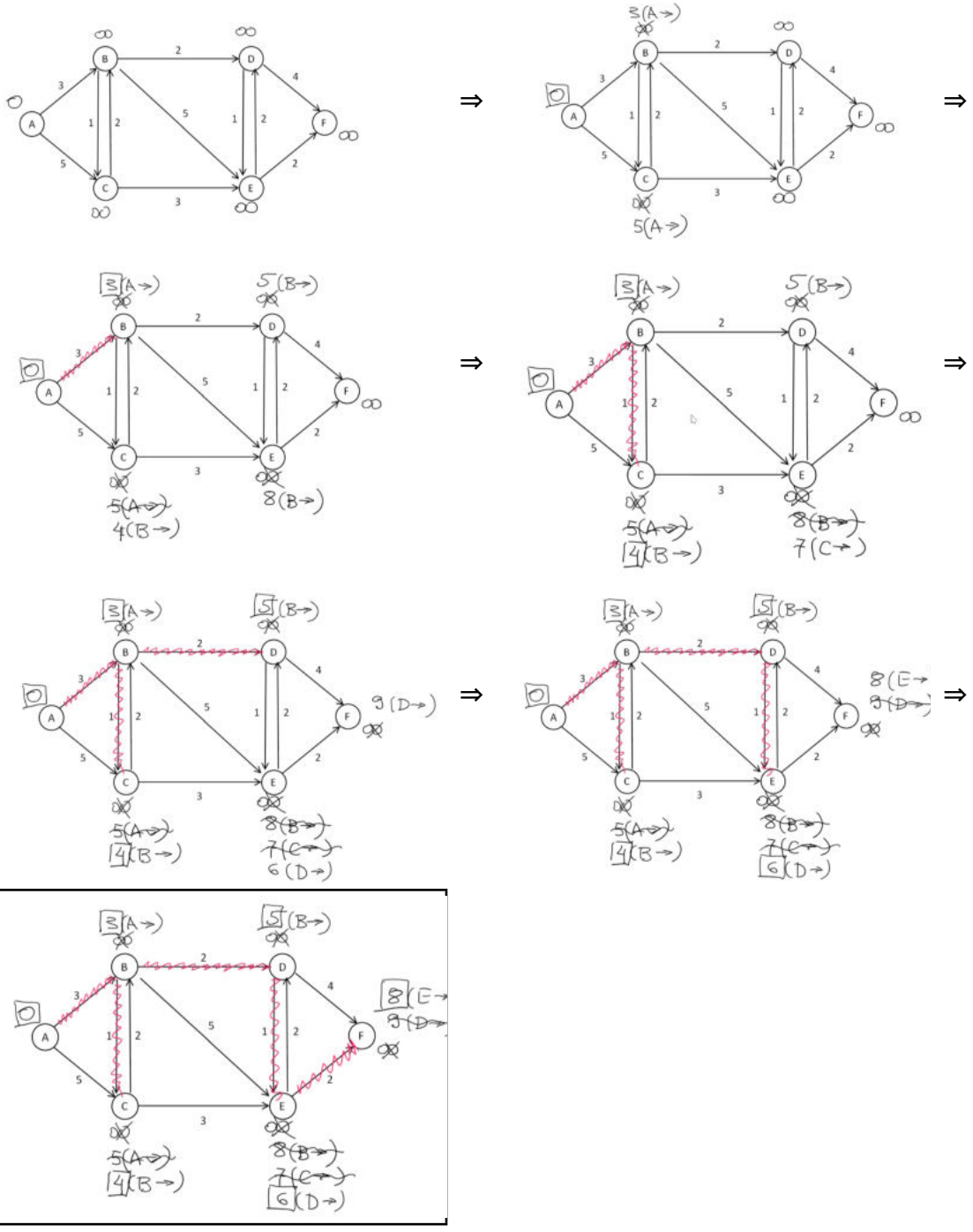
Este algoritmo, que resolve de modo eficiente o problema de encontrar o caminho mínimo entre um vértice de origem e um vértice de destino em um grafo, direcionado ou não, com um custo não negativo associado a cada aresta, foi concebido por Edsger Dijkstra em 1956. Na verdade, ele acha o melhor caminho para ir da origem a cada um dos demais vértices. A ideia fundamental é a de que é imediato encontrar o caminho mínimo para um dos vértices, aquele que está ligado, com custo mínimo, diretamente à origem. No nosso exemplo, a origem é A e o vértice diretamente ligado à origem com custo mínimo é B. Como os custos são não negativos, não é possível encontrar um caminho alternativo melhor para B, pois ele passaria necessariamente por C e teria um custo maior. Esta ideia leva ao seguinte algoritmo:

Algoritmo de Dijkstra

- Atribua valor zero à estimativa do custo mínimo para a origem e infinito às demais estimativas.
- Considere que todos os vértices estão **abertos**.
- Enquanto houver vértice aberto:
    - seja *v* um vértice ainda aberto cuja estimativa de custo seja a menor dentre todos os vértices abertos (no passo inicial, esse vértice é a origem);
    - feche o vértice *v*
    - para todo vértice *w* ainda aberto para o qual existe a aresta *vw*:
        - some a estimativa do vértice *v* com o custo da aresta *vw*;
        - caso esta soma seja melhor que a estimativa anterior para o vértice *w*, substitua-a e anote *v* como precedente de *w*.

Ao final do algoritmo, encontramos o custo do caminho mínimo da origem até cada um dos demais vértices. Além disso, os caminhos podem ser reconstruídos utilizando o precedente de cada vértice.

Na aula gravada, fazemos o passo a passo do algoritmo de Dijkstra para o nosso exemplo. As telas para as diversas etapas são dadas abaixo.



Ao final da execução do algoritmo, podemos concluir que o caminho de custo mínimo de A a F tem custo 8. Além disso, vemos que, para chegar a F, devemos vir de E; para chegar a E, devemos vir de D; para chegar a D, devemos vir de B; finalmente, para chegar a B, devemos vir diretamente de A. Logo, o caminho ótimo é ABDEF.

6. **Complexidade do algoritmo de Dijkstra**
   A complexidade do algoritmo de Dijkstra é $O(|V|^2)$, já que o processo de fechamento de vértices é executado uma vez para cada vértice e a busca do vértice a ser fechado e a atualização das distâncias dos vértices ligados a ele requer $O(|V|)$ operações.
   Na prática, é possível melhorar o desempenho do algoritmo usando estruturas de dados mais eficientes e empregando regras heurísticas que o aceleram. Mesmo sem essas melhorias, o algoritmo resolve em fração de segundos o problema do caminho mínimo em um grafo com 30 vértices, que vimos ser impossível de resolver com força bruta.

7. **Implementação em Python**
   O algoritmo pode ser facilmente implementado usando uma linguagem de programação. Abaixo, vemos como fazê-lo em Python. Se desejar, você pode executá-lo em https://www.onlinegdb.com/online_python_compiler.

```
# Criando a matriz de adjacências
INF = float('inf')
adj  =  [[INF,   3,   5, INF, INF, INF],
         [INF, INF,   1,   2,   5, INF],
         [INF,   2, INF, INF,   3, INF],
         [INF, INF, INF, INF,   1,   4],
         [INF, INF, INF,   2, INF,   2],
         [INF, INF, INF, INF, INF, INF]]
numero_de_vertices = len(adj[0])
fechado = [0]
distancia = [INF]
anterior = [-1]

for i in range(numero_de_vertices-1):
  fechado.append(0)
  distancia.append(INF)
  anterior.append(-1)

origem = ord(input("Qual é a origem?")) - ord('A')
distancia[origem] = 0

for vertice in range(numero_de_vertices):
  # Encontrando o próximo vértice a ser fechado
  a_fechar = -1
  # Determinando o vértice não fechado com distância mínima
  for i in range(numero_de_vertices):
    if fechado[i] == 0 and (a_fechar < 0 or distancia[i] <= distancia[a_fechar]):
        a_fechar = i
  fechado[a_fechar]= 1
  for vizinho in range(numero_de_vertices):
    # Calculando a nova distancia aos vizinhos do vértice que foi fechado
    # passando por esse vértice e atualizando a distancia provisoria
    if adj[a_fechar][vizinho] < INF and fechado[vizinho] == 0:
      nova_distancia = distancia[a_fechar] + adj[a_fechar][vizinho]
      if distancia[vizinho] > nova_distancia:
        distancia[vizinho] = nova_distancia
        anterior[vizinho] = a_fechar

# Imprimindo os caminhos de menor custo
print("Saindo da origem " + chr(ord('A')+ origem), ":")
for v in range(numero_de_vertices):
    if (v != origem):
        if anterior[v] == -1:
            print("Não há caminho para o vértice ", chr(ord('A') + v))
        else:
            caminho = chr(ord('A') + v)
            prev = v
            while(prev != origem):
                prev = anterior[prev]
                caminho = chr(ord('A') + prev) + caminho
            print("Caminho para o vértice " + chr(ord('A') + v)+ ": " + caminho + ", de custo " +
str(distancia[v]))
```

Veja o resultado produzido pelo programa quando a origem é A.

```
Qual é a origem?A
Saindo da origem A :
Caminho para o vértice B: AB, de custo 3
Caminho para o vértice C: ABC, de custo 4
Caminho para o vértice D: ABD, de custo 5
Caminho para o vértice E: ABDE, de custo 6
Caminho para o vértice F: ABDEF, de custo 8
```

Estrada Dona Castorina, 110 Jardim Botânico - CEP 22460-320 Rio de Janeiro, RJ - Brasil Phone: +55 21 2529-5275
www.impa.br

Finalmente, veja o que ocorre quando a origem é B.

```
Qual é a origem?B
Saindo da origem B :
Não há caminho para o vértice  A
Caminho para o vértice C: BC, de custo 1
Caminho para o vértice D: BD, de custo 2
Caminho para o vértice E: BDE, de custo 3
Caminho para o vértice F: BDEF, de custo 5
```