Alunos: Vinicius Guimarães. RA: 802431
Vitor Enzo. RA: 802123

/ /

(1) Considerando um grafo $G = (V, E, W)$, ponderado com função de custo sendo $w: E \rightarrow R^+$. O peso de um caminho dado por $P = V_0 V_1 V_2 \dots V_3$ é definido da seguinte forma:

$$w(P) = \sum_{i=1}^{n} w(V_{i-1}, V_i)$$

Assim, um caminho ótimo P^* de V_0 a V_n é aquele que minimiza $w(P)$, ou seja $P^* = \underset{P}{\text{argmin}}\, W(P)$ quando existir um caminho P de V_0 até V_n. Se não existir caminho P de V_0 até V_n, então o custo do caminho é infinito: $w(P) = \infty$

Dessa forma, considerando o caminho:

$P = V_0 \xrightarrow{P_{0i}} V_i \xrightarrow{P_{ij}} V_j \xrightarrow{P_{jn}} V_n$, onde $P = V_0 V_1 V_2 \dots V_n$ é o caminho mínimo de V_i a V_j.

O peso total desse caminho P é dado por: $w(P) = w(P_{0i}) + w(P_{ij}) + w(P_{jn})$.

Para provar que os subcaminhos também são mínimos, então:

- Supondo que exista caminho P'_{ij} de V_i a V_j tal que $w(P'_{ij}) < w(P_{ij})$
- Então o caminho $\overline{P}$ é definido como

  $\overline{P} = V_0 \xrightarrow{P_{0i}} V_i \xrightarrow{P'_{ij}} V_j \xrightarrow{P_{jn}} V_n$

  Com peso $W(\overline{P}) = W(P_{0i}) + W(P'_{ij}) + W(P_{jn})$, na qual $W(\overline{P}) < W(P)$ que é uma contradição, uma vez que P foi definido como sendo o caminho mínimo

- Portanto, não existe P'_{ij} diferente de P_{ij} com $W(P'_{ij}) < W(P_{ij})$, ou seja, os subcaminhos são mínimos (ótimos).

Digitalizado com CamScanner

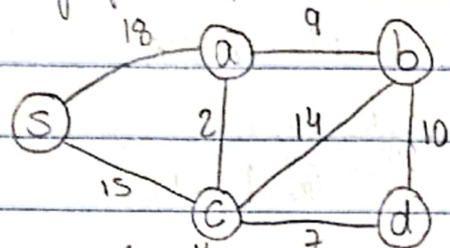
/ /

2 O algoritmo de Dijkstra utiliza a programação dinâmica, com a estratégia bottom-up para a resolução dos problemas, da seguinte forma:

1) Não recalcula os custos dos caminhos a cada interação, ou seja, armazena os valores dos caminhos mínimos em uma fila de prioridades Q.

2) Constrói a solução ótima a partir da solução dos subproblemas menores (subestrutura ótima de caminhos mínimos possui sobreposição)

Considerando o grafo abaixo

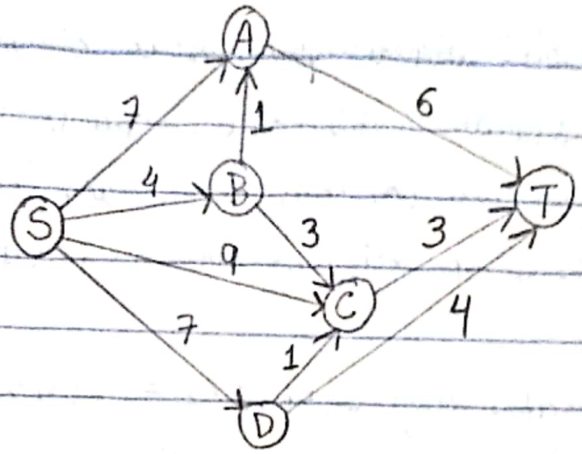


Observa-se as seguintes iterações e fila de prioridades

FILA DE PRIORIDADES						ORDEM DE ACESSO AOS VÉRTICES			
	s	a	b	c	d	u	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
$\lambda^0(v)$	$\boxed{0}$	∞	∞	∞	∞	s	$\{a, c\}$	$\lambda(a) = \min\{\lambda(a), \lambda(s) + w(s,a)\}$	$\pi(a) = s$
		18	∞	$\boxed{15}$	∞			$= \min\{\infty, 18\} = 18$	$\pi(c) = s$
								$\lambda(c) = \min\{\infty, 15\} = 15$	
						c	$\{a, b, d\}$	$\lambda(a) = \min\{\lambda(a), \lambda(c) + w(c,a)\}$	$\pi(a) = c$
								$= \min\{18, 17\} = 17$	
						⋮	⋮	⋮	⋮

Note que a cada iteração, utiliza-se os valores anteriormente calculados para que não seja preciso calcular novamente.

③ Dado o grafo:



FILA DE PRIORIDADES

	S	A	B	C	D	T
λ^0	$\boxed{0}$	∞	∞	∞	∞	∞
		7	$\boxed{4}$	9	7	∞
		$\boxed{5}$		7	7	∞
				$\boxed{7}$	7	11
					$\boxed{7}$	10
						$\boxed{10}$

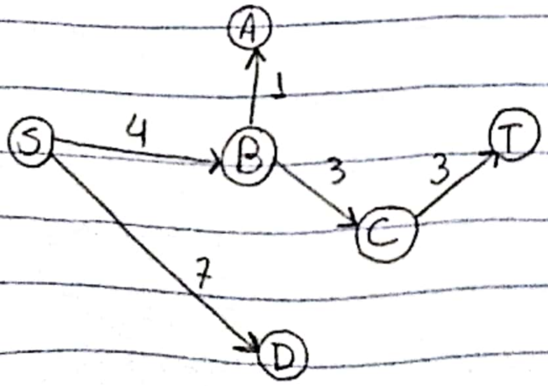
ORDEM DE ACESSO AOS VÉRTICES

u	$V'=\{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
S	$\{A,B,C,D\}$	$\lambda(A)=\min\{\infty,7\}=7$	$\pi(A)=S$
		$\lambda(B)=\min\{\infty,4\}=4$	$\pi(B)=S$
		$\lambda(C)=\min\{\infty,9\}=9$	$\pi(C)=S$
		$\lambda(D)=\min\{\infty,7\}=7$	$\pi(D)=S$
B	$\{A,C\}$	$\lambda(A)=\min\{7,5\}=5$	$\pi(A)=B$
		$\lambda(C)=\min\{9,7\}=7$	$\pi(C)=B$
A	$\{T\}$	$\lambda(T)=\min\{\infty,11\}=11$	$\pi(T)=A$
C	$\{T\}$	$\lambda(T)=\min\{11,10\}=10$	$\pi(T)=C$
D	$\{T\}$	$\lambda(T)=\min\{10,11\}=11$	
T	$\emptyset$	—	—

ÁRVORE DE CAMINHOS MÍNIMOS

Antecessor:

S	A	B	C	D	T
—	B	S	B	S	C



OBS: NO PDF não tem questão 4.

(5) Para analisar a complexidade do algoritmo de Dijkstra usando estruturas de dados estáticas, pode-se considerar:

- $G = (V, E)$ representado por uma matriz de adjacência
- Fila de prioridades Q representada por um array estático de n elementos

Com isso, obtém-se:

- Inicialização dos $\lambda(v)$ sendo $O(n)$
- Inserção dos vértices na fila Q é $O(n)$
- Loop while é executado n vezes (uma vez para cada vértice $v \in Q$)
- $u = ExtractMin(Q)$ sendo $O(n)$ → Serve para encontrar o menor elemento do vetor
- Atualização do valor de $\lambda(v)$ sendo $O(1)$, mas executa $K = d(u)$ vezes, onde $d(u)$ é o grau de u
- Decrease_Key é $O(1)$, pois é acesso direto

Assim, a função $T(n)$, que mede a complexidade do algoritmo fica sendo:

$$T(n) = O(n) + O(n) + O(n) \cdot O(n) + (O(1) + O(1)) \cdot (d(v_1) + d(v_2) + \dots + d(v_n))$$

Considerando que a soma dos graus de um grafo é igual a 2 vezes o número de arestas, então:

$$T(n) = O(n) + O(n^2) + O(1) \cdot O(m)$$

Considerando que, em grafos simples $m < n^2$, temos finalmente que a eficiência do algoritmo é $O(n^2)$

/ /

(6) Para analisar a complexidade do algoritmo de Dijkstra usando estruturas de dados dinâmicas, pode-se considerar:

- $G = (V, E)$ representado por lista de adjacência
- Fila de prioridades Q representada por um heap binário

Com isso, obtém-se:

- Inicialização dos $\lambda(v)$ sendo $O(n)$
- Inserção do vértice na fila Q é $O(\log n)$ por conta de ser árvore binária
- Loop while é executado n vezes (uma para cada vértice $v \in Q$)
- $u = ExtractMin(Q)$ é $O(\log n)$ → busca em árvore binária
- Atualização do valor de $\lambda(v)$ é $O(1)$, porém executa $K = d(u)$ vezes, onde $d(u)$ é o grau de u
- Decrease-Key é $O(\log n)$ → árvore binária

Assim, a função $T(n)$, que mede a complexidade do algoritmo fica sendo:

$$T(n) = O(n) + O(\log n) + O(n) * O(\log n) + (O(1) + O(\log n)) \cdot (d(v_1) + d(v_2) + \dots + d(v_n))$$

$$T(n) = O(n) + O(\log n) + O(n \log n) + O(m \log n) = O(n \log n) + O(m \log n)$$

$$T(n) = O((n+m) \log n)$$

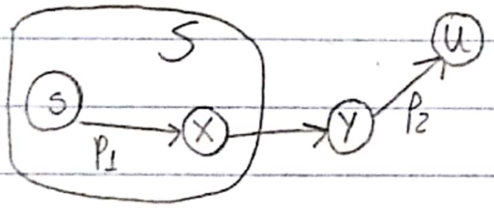
Como um grafo conexo $m > n-1$, então

$$T(n) = O(m \log n)$$

Portanto, se o grafo no qual Dijkstra vai ser aplicado for mais denso (m grande), compensa utilizar estruturas de dados estáticas. Já se o grafo for menos denso, então compensa utilizar estruturas de dados dinâmicas.

7 Sabendo que o algoritmo de dijkstra termina com $\lambda(v) = d(s,v)$, onde $d(s,v)$ é a menor distância possível de s até v, então faremos uma prova por contradição

- Considerando que u seja o primeiro vértice no qual $\lambda(u) \neq d(s,u)$, quando u entra em S
- Assim, $u \neq s$ pois se não teríamos $\lambda(s) = d(s,s) = 0$
- Por isso, existe um caminho P_{su} pois se não $\lambda(u) = d(s,u) = \infty$. Portanto existe um caminho mínimo P^*_{su}
- Antes de adicionarmos u em S, P^*_{su} possui $s \in S$ e $u \in V-S$
- Sendo y o 1º vértice em P^*_{su} tal que $y \in V-S$ e sendo x o seu predecessor, com $x \in S$, então há:



- Como $x \in S$, $\lambda(x) = d(s,x)$ e no momento em que x foi inserido em S, a aresta (x,y) foi relaxada, ou seja

$$\lambda(y) = \lambda(x) + w(x,y) = d(s,x) + w(x,y) = d(s,y)$$

- Porém, y antecede u no caminho e como os pesos das arestas são positivos, então $d(s,y) \leq d(s,u)$, ou seja $\lambda(y) = d(s,y) \leq d(s,u) \leq \lambda(u)$
- Entretanto, como u e y pertencem a V-S, quando u é escolhido para entrar em S temos que $\lambda(u) \leq \lambda(y)$
- Como $\lambda(y) \leq \lambda(u)$ e $\lambda(u) \leq \lambda(y)$ então temos que $\lambda(u) = \lambda(y)$ e assim:

$$\lambda(y) = d(s,y) = d(s,u) = \lambda(u)$$, que gera uma contradição

Por isso, $\nexists\, u \in V$ tal que $\lambda(u) \neq d(s,u)$ quando u entra em S

* Portanto, o algoritmo de dijkstra sempre termina com $\lambda(v) = d(s,v)$, onde $d(s,v)$ é a distância mínima de s até v.

7) Sabendo que um caminho mínimo é composto pela origem, destino e vértices intermediários da seguinte forma:

$$P = v_1 \; \underbrace{v_2 \;\; v_3 \;\; v_4 \ldots v_{n-1}}_{\text{Vértices Intermediários}} \; v_n$$

Considerando $i \xrightarrow{P_1} k \xrightarrow{P_2} j$, sabe-se que de acordo com a estrutura ótima dos caminhos mínimos, P_1 é caminho mínimo de i até K e P_2 é caminho mínimo de K até j. Ambos caminhos possuem vértices intermediários em V_{K-1}.

Sendo $d_{ij}^{(K)}$ o custo do caminho mínimo de i até j com todos os vértices intermediários em V_K. Quando $K=0$, não há vértices intermediários $d_{ij}^{(0)} = w_{ij}$, onde a matriz de adjacência W é dada por

$$w_{ij} \begin{cases} 0 & , \text{se } i=j \\ w(v_i,v_j) & , \text{se } i\neq j \text{ e } (v_i,v_j)\in E \\ \infty & , \text{se } i \neq j \text{ e } (v_i,v_j)\notin \end{cases}$$

Assim, com base na análise anterior, pode-se utilizar programação dinâmica com estratégia bottom-up, da seguinte forma:

$$d_{ij}^{(K)} = \begin{cases} w_{ij} & , \text{se } K=0 \\ \min\{d_{ij}^{(K-1)}, d_{iK}^{(K-1)} + d_{Kj}^{(K-1)}\} \end{cases}$$

Dessa forma, como para qualquer caminho P, todos os vértices intermediários pertencem ao conjunto $\{1,2,\ldots,n\}$, então a matriz $D^{(n)} = d_{ij}^{(n)}$ vai conter a resposta desejada.

Portanto, ao não repetir cálculos dos subproblemas com armazenamento dos valores em uma matriz, está utilizando programação dinâmica, de forma que, ao executar o cálculo $\min\{d_{ij}^{(K-1)}, d_{iK}^{(K-1)} + d_{Kj}^{(K-1)}\}$, obtém-se os caminhos mínimos no final da execução.

QUESTÃO 8

```
import java.util.Arrays;

public class Floyd {
    public static int INFINITO = Integer.MAX_VALUE;

    public static void floydWarshall(int [][] grafo) {
        int n = grafo.length;

        // Fazendo uma cópia do grafo para não alterar o grafo original
        int [][] distancias = new int[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                distancias[i][j] = grafo[i][j];
            }
        }

        // Executando o algoritmo de Floyd Warshall
        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (distancias[i][k] != INFINITO
                            && distancias[k][j] != INFINITO &&
distancias[i][k] + distancias[k][j] < distancias[i][j]) {
                        distancias[i][j] = distancias[i][k] +
distancias[k][j];
                    }
                }
            }
        }

        // Imprimindo a matriz de distâncias mínimas
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (distancias[i][j] != INFINITO) {
                    System.out.print(distancias[i][j] + " ");
                } else {
                    System.out.print("INF ");
                }
            }
            System.out.println("");
        }
```

```java
    }

    public static void main(String[] args) {
        int[][] grafoInicial = {
                {0, 2, INFINITO, 1, 8},
                {6, 0, 3, 2, INFINITO},
                {INFINITO, INFINITO, 0, 4, INFINITO},
                {INFINITO, INFINITO, 2, 0, 3},
                {3, INFINITO, INFINITO, INFINITO, 0}
        };

        // Chamando função para executar o algoritmo
        floydWarshall(grafoInicial);
    }
}
```

```
[0, 2, 3, 1, 4]
[6, 0, 3, 2, 5]
[10, 12, 0, 4, 7]
[6, 8, 2, 0, 3]
[3, 5, 6, 4, 0]
```