Problema do Caminho Mínimo

Shortest Path Problem

O problema do caminho mínimo ou caminho mais curto, consiste em encontrar o melhor caminho entre dois nós. Assim, resolver este problema pode significar determinar o caminho entre dois nós com o custo mínimo, ou com o menor tempo de viagem.

Numa rede qualquer, dependendo das suas características, podem existir vários caminhos entre um par de nós, definidos como origem e destino. Entre os vários caminhos aquele que possui o menor "peso" é chamado de caminho mínimo. Este peso representa a soma total dos valores dos arcos que compõem o caminho e estes valores podem ser: o tempo de viagem, a distância percorrida ou um custo qualquer do arco.

Applications of the shortest path problem include those in road networks, logistics, communications, electronic design, power grid contingency analysis, and community detection. (https://developer.nvidia.com/discover/shortest-path-problem)

1) Modelagem matemática

O modelo matemático para o problema de caminho mais curto do nó 1 ao nó n de um grafo G=(N,E), N = {1, 2, ..., n}.

Variáveis:

$x_{ij} \in \{0,1\} \rightarrow$ ativação, ou não do arco (i, j)

Parâmetros:

$c_{ij} \rightarrow$ custo unitário do fluxo em (i, j)

$S(j) \rightarrow$ é o conjunto dos nós sucessores de j

$P(j) \rightarrow$ é o conjunto dos nós predecessores de j

Função objetivo:

$$\min Z = \sum_{i=1}^{n} \sum_{j \in S(i)} c_{ij} x_{ij}$$

Restrições:

$$\sum_{j \in S(1)} x_{1j} = 1$$

$$\sum_{i \in P(n)} x_{in} = 1$$

$$\sum_{i \in P(j)} x_{ij} = \sum_{k \in S(j)} x_{jk}\,, j = 2, \dots, n-1$$

Para resolução do problema de caminho mínimo, existem algoritmos alternativos mais simples e mais eficientes do que o método simplex.

2) Algoritmo de Dijkstra

O algoritmo de Dijkstra é uma solução para o problema do caminho mínimo de origem única. Funciona em grafos orientados e não orientados, no entanto, **todas as arestas devem ter custos não negativos**. Se houver custos negativos, usa-se o algoritmo de Bellman-Ford.

Entrada: Grafo ponderado G=(N,E) e nó origem O ∈ N, de modo que todos os custos das arestas sejam não negativos.

Saída: Comprimentos de caminhos mais curtos (ou os caminhos mais curtos) de um nó origem O ∈ N para todos os outros nós.

Como funciona?

O algoritmo de Dijkstra identifica, a partir do nó O, qual é o custo mínimo entre esse nó e todos os outros do grafo. No início, o conjunto S contém somente esse nó, chamado *origem*. A cada passo, selecionamos no conjunto de nós sobrando, o que está mais perto da origem. Depois atualizamos, para cada nó que está sobrando, a sua distância em relação à origem. Se passando pelo novo nó acrescentado a distância ficar menor, é essa nova distância que será memorizada.

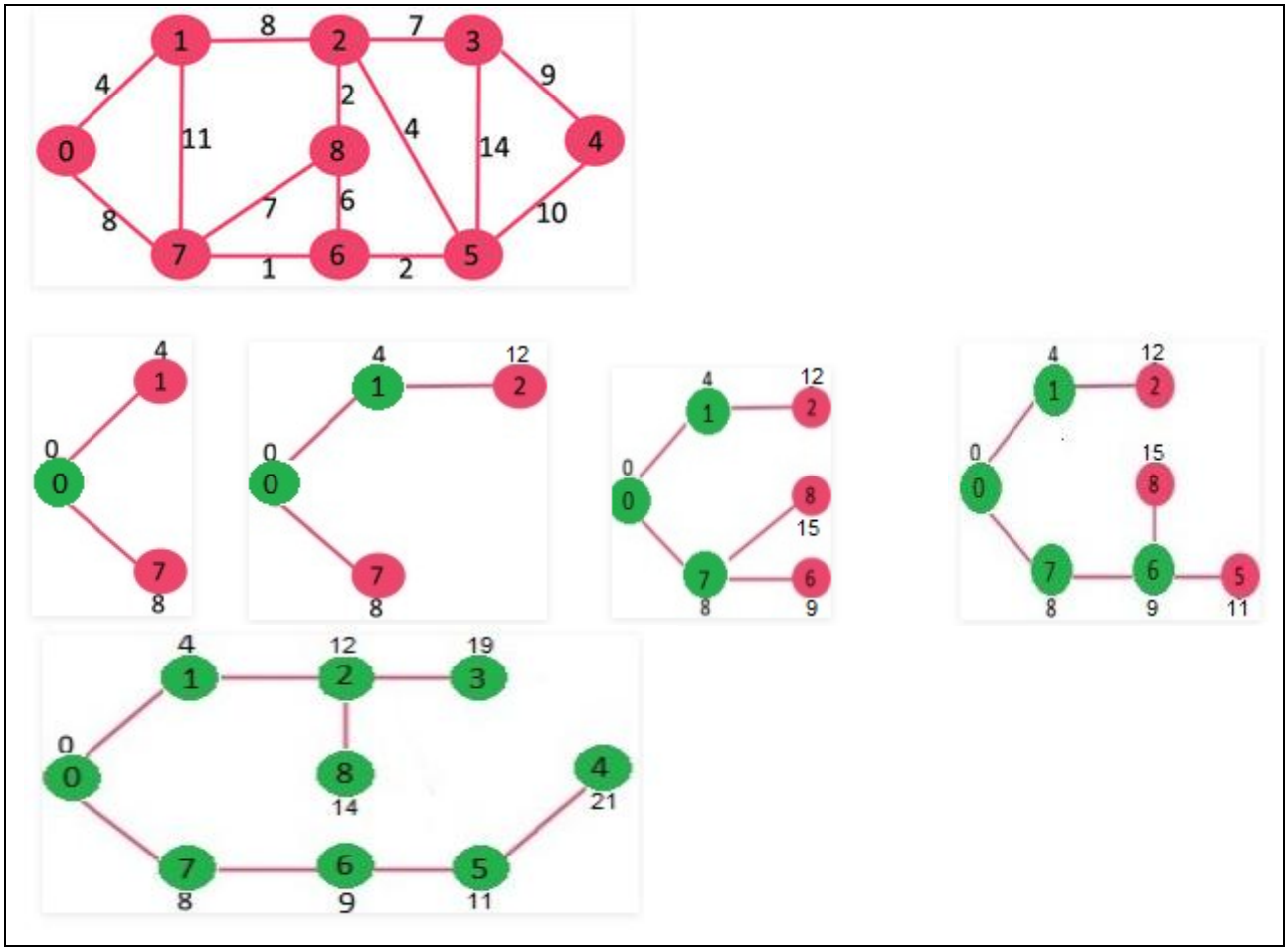
Escolhido um nó como origem da busca, este algoritmo calcula, então, o custo mínimo deste nó para todos os demais nós do grafo. O procedimento é iterativo, determinando, na iteração 1, o nó mais próximo do nó O, na segunda iteração, o segundo nó mais próximo do nó O, e assim sucessivamente, até que em alguma iteração todos os n sós sejam atingidos.

Seja G=(N,E) um grafo orientado e s um nó de G:

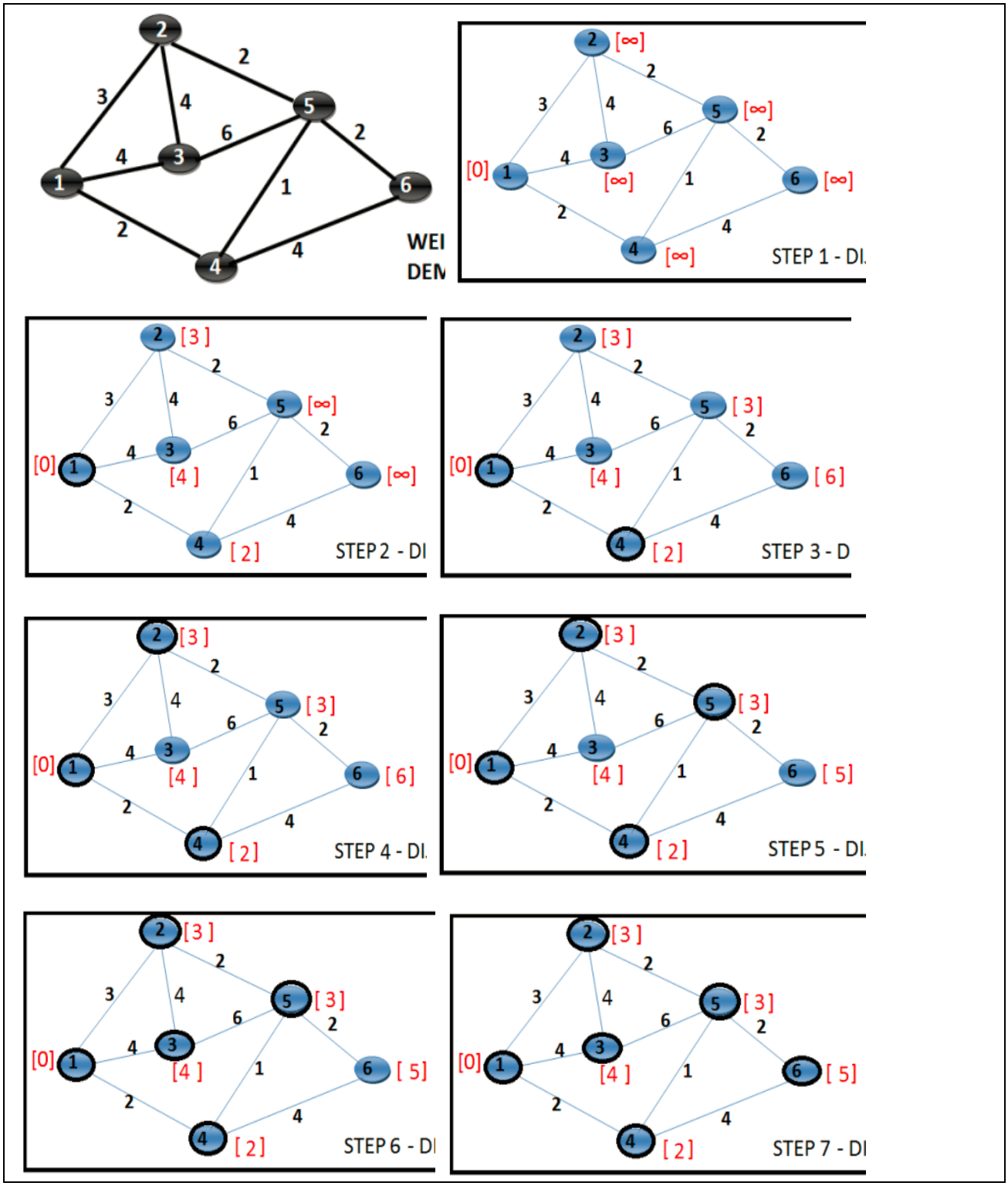
- Atribua valor zero à estimativa do custo mínimo do nó O (a origem da busca) e infinito às demais estimativas;

- Atribua um valor qualquer aos precedentes (o precedente de um nó t é o nó que precede t no caminho de custo mínimo de s para t);
- Enquanto houver nó aberto:
  - seja k um nó ainda aberto cuja estimativa seja a menor dentre todos os nós abertos;
  - feche o nó k;
  - Para todo nó j ainda aberto que seja sucessor de k faça:
    - some a estimativa do nó k com o custo do arco que une k a j;
    - caso esta soma seja melhor que a estimativa anterior para o nó j, substitua-a e anote k como precedente de j.

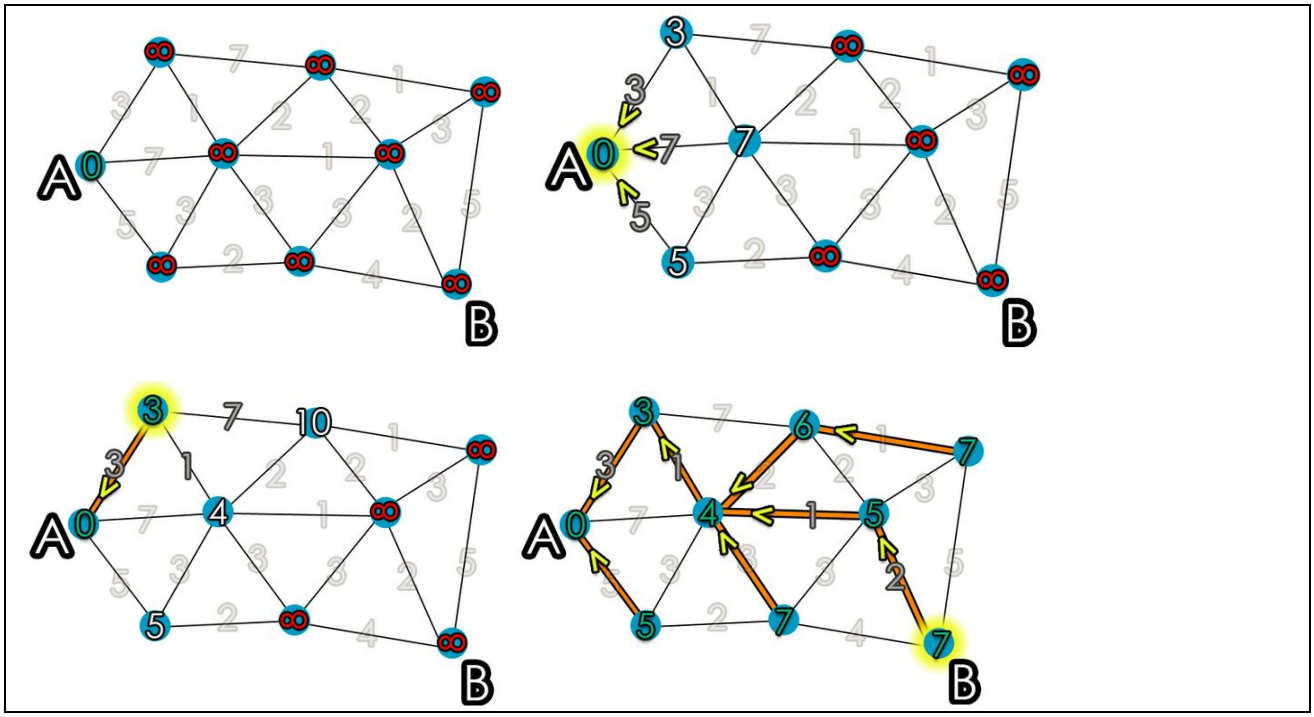
Exemplo 1



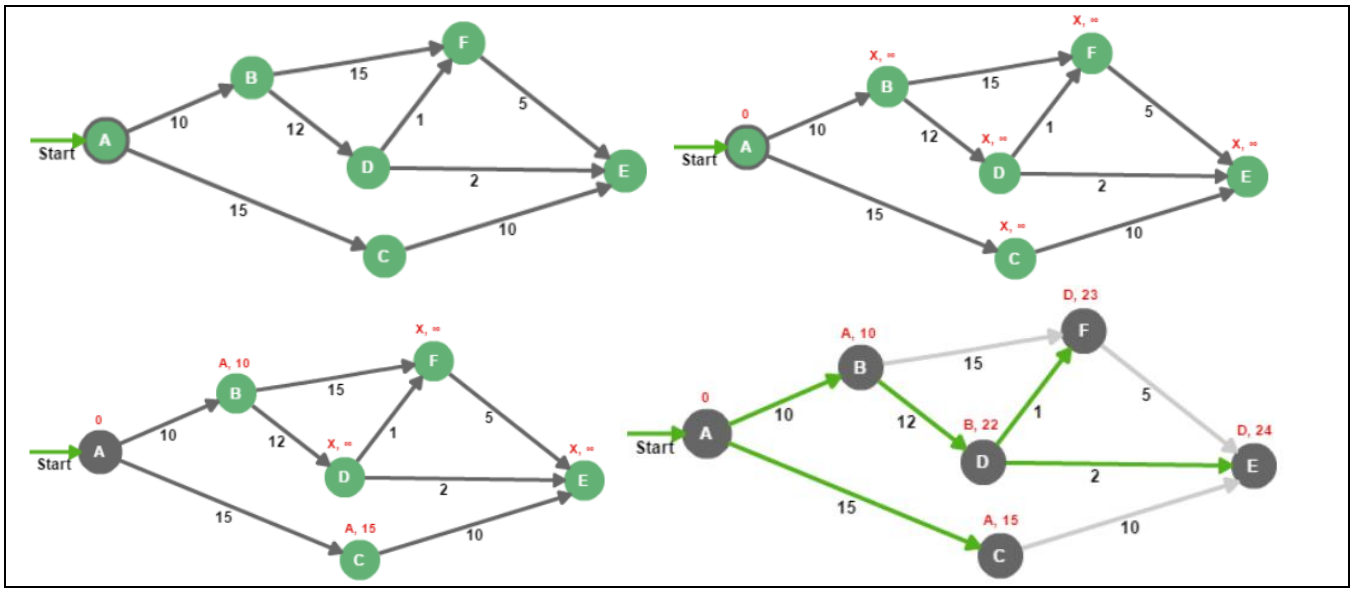
Exemplo 2


WEI
DEM
STEP 1 - DI
STEP 2 - DI
STEP 3 - D
STEP 4 - DI
STEP 5 - DI
STEP 6 - DI
STEP 7 - DI

Exemplo 3



Exemplo 4



3) Algoritmo de Floyd-Warshall (https://www.quora.com/Why-is-the-order-of-the-loops-in-Floyd-Warshall-algorithm-important-to-its-correctness)

O algoritmo de Floyd é um algoritmo que resolve o problema de determinar o caminho mais curto entre todos os pares de nós em um grafo orientado e ponderado.

Trata-se de um algoritmo que utiliza matrizes para determinar os caminhos mínimos entre todos os pares de nós da rede. No algoritmo de Floyd são feitas n iterações que

corresponde ao número de nós da rede. A cada iteração corresponde uma matriz $D_{n\times n}$ cujos valores são modificados utilizando uma fórmula de recorrência:

$$d_{ij}^{n} = \min\{d_{ik}^{n-1} + d_{kj}^{n-1}, d_{ij}^{n-1}\}.$$

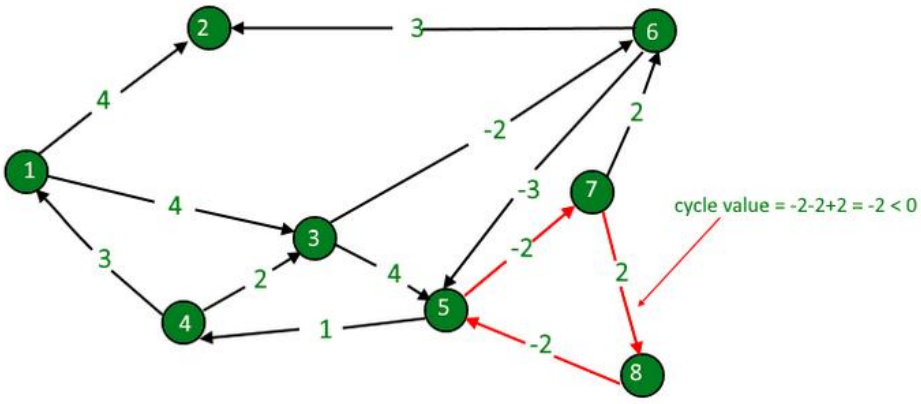
Determina-se inicialmente a matriz D^0 cujos valores correspondem aos comprimentos dos arcos se estes existem senão os valores são ∞. Depois se calcula D^1 de D^0 e D^2 de D^1 até se obter D^n de D^{n-1}. A matriz D^n é a matriz final que apresenta as distâncias mínimas entre todos os nós da rede. A ideia básica deste algoritmo é verificar a cada iteração se a inclusão de um nó k intermediário no caminho de i para j pode reduzir o tamanho de um caminho já determinado.

- Numere os nós do grafo de $1, \ldots, n$. Defina a matriz D^0 cujos valores d_{ij}^0 correspondem ao tamanho/comprimento dos arcos (i, j) se existir o arco no grafo; caso contrário considere $d_{ij}^0 = \infty$, e faça os elementos da diagonal da matriz, $d_{ii}^0 = 0$ para todo i.
- Para cada $k = 1,\ldots,n$ determine sucessivamente os elementos da matriz D^k a partir dos elementos da matriz D^{k-1} utilizando a expressão acima.

Este processo é repetido até $k = n$, e neste caso o valor do caminho mínimo de todos os pares (i, j) do grafo estão definidos na matriz D^n.

Entrada: Grafo ponderado G=(N,E). As arestas do grafo **podem ter valores negativos, mas o grafo não pode conter nenhum ciclo de valor negativo**.

Saída: Matriz n × n com os custos dos menores caminhos entre todos os nós de N.



Um ciclo negativo é aquele em que a soma total do ciclo é negativa.

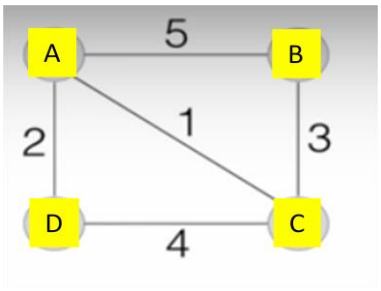
Simultaneamente à construção da matriz de distâncias mínimas D^k entre todos os nós, pode-se construir a matriz dos predecessores P^k. A matriz P^k pode ser lida da seguinte forma: se queremos reconstruir o caminho (mais curto) entre os nós i e j, então observamos o elemento nas coordenadas correspondentes. Se seu valor for 0, não haverá caminho entre esses nós; caso contrário, o valor do elemento denota o predecessor de j no caminho de i a j. Repetimos esse procedimento enquanto o nó anterior não é igual a i.

A matriz P^k é assim construída

$$P_{ij}^{(0)} = \begin{cases} 0 & i = j \text{ ou } D_{ij} = \infty \\ i & \text{caso contrário} \end{cases}$$

$$P_{ij}^{(n)} = \begin{cases} P_{ij}^{(n-1)} & D_{ij}^{(n-1)} \leq D_{ik}^{(n-1)} + D_{kj}^{(n-1)} \\ P_{kj}^{(n-1)} & D_{ij}^{(n-1)} > D_{ik}^{(n-1)} + D_{kj}^{(n-1)} \end{cases}$$

Exemplo 1

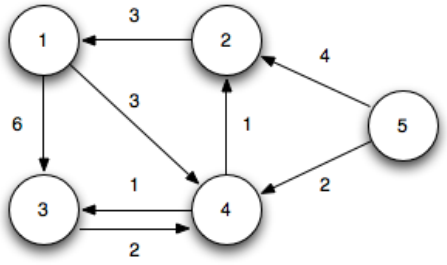


Origem	Destino	Distancia	Menor Caminho
A	B	4	A->C->B
A	C	1	A->C
A	D	2	A->D
B	C	3	B->C
B	D	6	B->C->A->D
C	D	3	C->A->D

D3	A	B	C	D
A	-	4	1	2
B	4	-	3	6
C	1	3	-	3
D	2	6	3	-

P3	A	B	C	D
A	-	C	A	A
B	C	-	B	A
C	C	C	-	A
D	D	C	A	-

Exemplo 2 http://faculty.ycp.edu/~dbabcock/PastCourses/cs360/lectures/lecture23.html



Initialization: $(k = 0)$

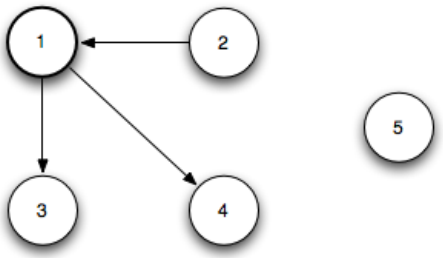
D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	∞	∞	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	/	/	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

Iteration 1: ($k = 1$) Shorter paths from $2 \rightsquigarrow 3$ and $2 \rightsquigarrow 4$ are found through vertex 1



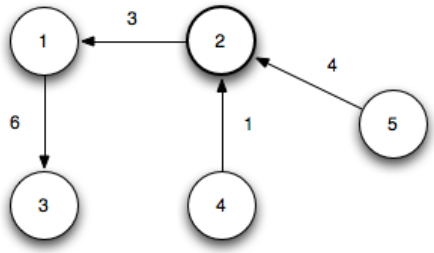
D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	**9**	**6**	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	**1**	**1**	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

Iteration 2: ($k = 2$) Shorter paths from $4 \rightsquigarrow 1$, $5 \rightsquigarrow 1$, and $5 \rightsquigarrow 3$ are found through vertex 2



D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	**4**	1	1	0	∞
5	**7**	4	**13**	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	**2**	4	4	/	/
5	**2**	5	**2**	5	/

Iteration 3: ($k = 3$) No shorter paths are found through vertex 3

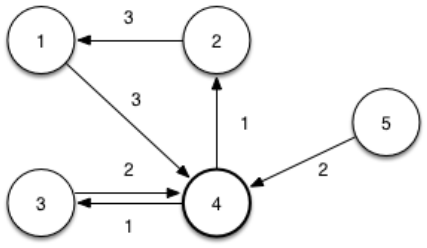
D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	4	1	1	0	∞
5	7	4	13	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	2	4	4	/	/
5	2	5	2	5	/

Iteration 4: ($k = 4$) Shorter paths from $1 \rightsquigarrow 2$, $1 \rightsquigarrow 3$, $2 \rightsquigarrow 3$, $3 \rightsquigarrow 1$, $3 \rightsquigarrow 2$, $5 \rightsquigarrow 1$, $5 \rightsquigarrow 2$, $5 \rightsquigarrow 3$, and $5 \rightsquigarrow 4$ are found through vertex 4



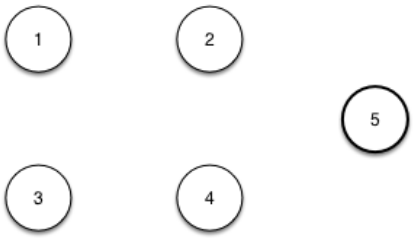
D

	1	2	3	4	5
1	0	**4**	**4**	3	∞
2	3	0	**7**	6	∞
3	**6**	**3**	0	2	∞
4	4	1	1	0	∞
5	**6**	**3**	**3**	2	0

Π

	1	2	3	4	5
1	/	**4**	**4**	1	/
2	2	/	**4**	1	/
3	**2**	**4**	/	3	/
4	2	4	4	/	/
5	2	**4**	**4**	5	/

Iteration 5: ($k = 5$) No shorter paths are found through vertex 5



D

	1	2	3	4	5
1	0	4	4	3	∞
2	3	0	7	6	∞
3	6	3	0	2	∞
4	4	1	1	0	∞
5	6	3	3	2	0

Π

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/

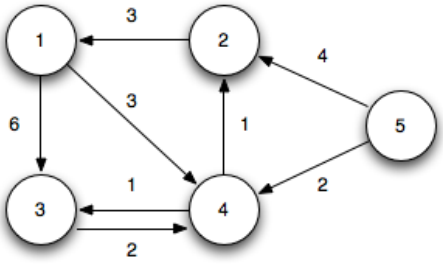
The final shortest paths for all pairs is given by

D

	1	2	3	4	5
1	0	4	4	3	∞
2	3	0	7	6	∞
3	6	3	0	2	∞
4	4	1	1	0	∞
5	6	3	3	2	0

Π

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/



Algoritmo de Bellman-Ford: O algoritmo de Bellman-Ford resolve o problema do caminho mais curto de única origem para o caso mais geral, no qual os custos das arestas podem ser negativos.

Algoritmo de Johnson: Enquanto o algoritmo de Floyd-Warshall é eficiente para grafos densos, se o grafo é esparso uma alternativa para todos os pares de estratégia de caminho mais curto conhecida como algoritmo de Johnson pode ser usada. Esse algoritmo basicamente usa o Bellman-Ford para detectar qualquer ciclo de custo negativo e, em seguida, emprega a técnica de reponderação das arestas para permitir que o algoritmo de Dijkstra encontre os caminhos mais curtos.

Algoritmo A* (http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html)

Derivado do algoritmo de Dijkstra, A* é a opção mais popular para a busca de caminhos, porque é bastante flexível e pode ser usada em uma ampla variedade de contextos.

Problema de Fluxo Máximo

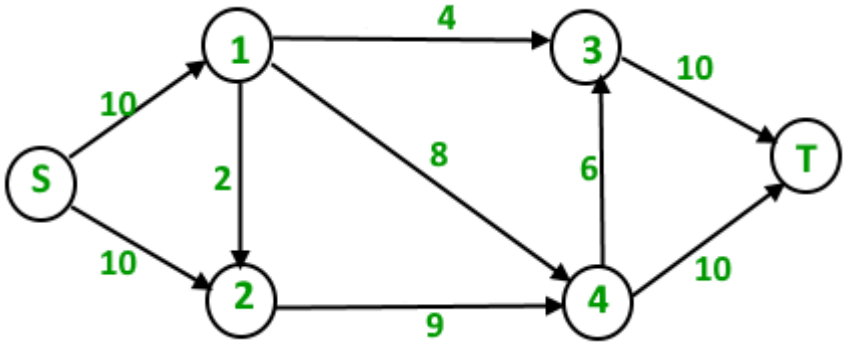
Dado um grafo que representa uma rede de fluxo onde cada aresta tem uma capacidade. Também são dados dois vértices: fonte 's' e sumidouro 't' no grafo G=(N,E). O problema consiste em encontrar o fluxo máximo possível de s para t com as seguintes restrições:

i. Cada aresta $(u, v) \in E$ tem uma capacidade não negativa $c(u, v) \geq 0$

ii. O fluxo de entrada é igual ao fluxo de saída para todos os nós, exceto s e t.

Se $(u, v) \notin E$, então $c(u, v) = 0$. A fonte produz o material a uma taxa constante e o sumidouro consome o material a uma taxa constante.

Por exemplo



https://www.geeksforgeeks.org/dinics-algorithm-maximum-flow/

Algoritmo

Os passos de cada iteração do algoritmo podem ser resumidos do seguinte modo:

Passo 1: Escolhe-se um caminho qualquer desde a origem até ao sumidouro via arestas cuja capacidade é positiva (>0).

Passo 2: Procurar nesse caminho o arco orientado com menor capacidade c.

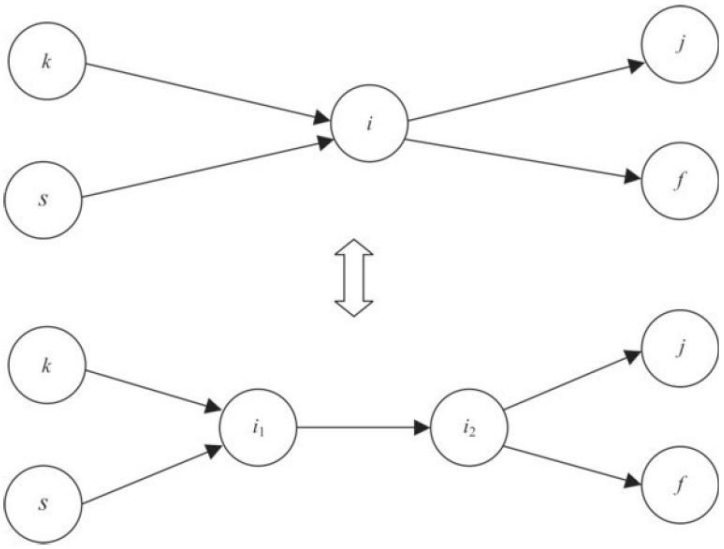
Passo 3: Subtrair de c a capacidade de fluxo em cada aresta do caminho no sentido direto e aumentar de c a capacidade das arestas no sentido inverso.

Passo 4: Regressar ao **Passo 1**. Se já não existir nenhum caminho em que todas as arestas tenham capacidade positiva, então o fluxo máximo já está determinado.

E se ocorrerem limitações nos arcos e nos nós?

Em alguns problemas de otimização em redes nos deparamos com limitações de fluxo nos nós juntamente com as limitações nos arcos. Neste caso, podemos reescrever o problema em outro equivalente, no qual as limitações de fluxo estão somente nos arcos. Para isso, pegue cada nó i com limitações de fluxo no grafo original e substitua-o por dois nós i_1 e i_2. No novo grafo, o arco (k,i) do grafo original é substituído por um arco (k,i_1), exatamente com os mesmos parâmetros de custos, limitações de capacidade etc., o arco (i,f) do grafo original é substituído por um arco (i_2,f), exatamente com os mesmos parâmetros de custo, limitações de capacidade etc., e adicionamos um arco (i_1,i_2) com custo 0 e com as limitações de capacidade do nó i

(figura abaixo). Neste novo grafo o arco (i_1,i_2) restringe o fluxo do nó i conforme se deseja. (ARENALES)



Algoritmo Ford-Fulkerson + Breadth-first search -BFS ≡ Algoritmo de Edmonds-Karp

Curiosidade: algoritmo guloso – *greedy algorithm*

São tipicamente usados para resolver problemas de otimização. Por exemplo, o algoritmo para encontrar o caminho mais curto entre duas cidades.

Um algoritmo guloso escolhe a estrada que parece mais promissora no instante atual e nunca muda essa decisão, independentemente do que possa acontecer depois.

A cada iteração:

a) seleciona um elemento conforme uma função gulosa;
b) marca-o para não considerá-lo novamente nos próximos estágios;
c) atualiza a entrada;
d) examina o elemento selecionado quanto sua viabilidade;
e) decide a sua participação ou não na solução.

Funcionamento: Constrói uma solução, elemento a elemento

a) A cada passo é adicionado um único elemento.
b) O elemento escolhido é o "melhor" segundo a função objetivo.
c) O método termina quando todos os elementos foram analisados e inseridos na solução.

Características dos algoritmos gulosos

- Para construir a solução ótima existe um conjunto ou lista de candidatos.
- São acumulados um conjunto de candidatos considerados escolhidos, e o outro de candidatos considerados rejeitados.
- Existe uma função que verifica se um conjunto particular de candidatos produz uma solução (sem considerar otimalidade no momento).
- Outra função verifica se um conjunto de candidatos é viável (também sem se preocupar com a otimalidade).
- Uma função de seleção indica a qualquer momento quais dos candidatos restantes é o mais promissor.
- Uma função objetivo fornece o valor da solução encontrada, como o comprimento do caminho construído (não aparece de forma explícita no algoritmo guloso).
- Quando funciona corretamente, a primeira solução encontrada é sempre ótima.
- A função de seleção é geralmente relacionada com a função objetivo.
- Se o objetivo é:
  - Maximizar ⇒ provavelmente escolherá o candidato restante que proporcione o maior ganho individual;
  - Minimizar ⇒ então será escolhido o candidato restante de menor custo.
- O algoritmo nunca muda de ideia:
  - Uma vez que um candidato é escolhido e adicionado à solução ele lá permanece para sempre;
  - Uma vez que um candidato é excluído do conjunto solução ele nunca mais é reconsiderado.

Em Algoritmos gulosos a construção de uma solução gulosa consiste em selecionar sequencialmente o elemento de N que minimiza o incremento no custo da solução parcial, eventualmente descartando alguns já selecionados, de tal forma que ao final se obtenha uma solução viável. O incremento no custo da solução parcial é chamado de função gulosa. Por escolher a cada passo considerando apenas a próxima decisão, chama-se também de algoritmo míope, pois enxerga somente o que está mais próximo.

Problemas que podem ser resolvidos por algoritmos gulosos têm duas propriedades principais:

- **Subestrutura ótima**: a solução ótima para um problema incorpora a solução ótima do(s) subproblema(s);
- **Propriedade de escolha gulosa**: escolhas localmente ótimas levam a uma solução globalmente ótima.