Grafos III

Joaquim Madeira

15/06/2021

Sumário

- Recap
- Determinação de Caminhos Mais Curtos (“Shortest-Paths”)
- O Algoritmo de Bellman-Ford
- Utilização de uma STACK/QUEUE como estrutura de dados auxiliar
- O Algoritmo de Dijkstra
- Sugestões de leitura

Recapitulação

Estrutura de dados

```c
struct _GraphHeader {
  unsigned short isDigraph;
  unsigned short isComplete;
  unsigned short isWeighted;
  unsigned int numVertices;
  unsigned int numEdges;
  List* verticesList;
};
```

```c
struct _Vertex {
  unsigned int id;
  unsigned int inDegree;
  unsigned int outDegree;
  List* edgesList;
};
```

```c
struct _Edge {
  unsigned int adjVertex;
  int weight;
};
```

Graph.h

```
typedef struct _GraphHeader Graph;

Graph* GraphCreate(unsigned short numVertices, unsigned short isDigraph,
                   unsigned short isWeighted);

Graph* GraphCreateComplete(unsigned short numVertices,
                           unsigned short isDigraph);

void GraphDestroy(Graph** p);

Graph* GraphCopy(const Graph* g);

Graph* GraphFromFile(FILE* f);
```

Graph.h

```
// Vertices

unsigned int* GraphGetAdjacentsTo(const Graph* g, unsigned int v);

// *** NEW ***
int* GraphGetDistancesToAdjacents(const Graph* g, unsigned int v);

//
// For a graph
//
unsigned int GraphGetVertexDegree(Graph* g, unsigned int v);

//
// For a digraph
//
unsigned int GraphGetVertexOutDegree(Graph* g, unsigned int v);
```

Graph.h

```c
unsigned short GraphAddEdge(Graph* g, unsigned int v, unsigned int w);

unsigned short GraphAddWeightedEdge(Graph* g, unsigned int v, unsigned int w,
                                    int weight);

// CHECKING

unsigned short GraphCheckInvariants(const Graph* g);

// DISPLAYING on the console

void GraphDisplay(const Graph* g);

void GraphListAdjacents(const Graph* g, unsigned int v);
```

Algoritmo recursivo

Travessia em Profundidade (vértice v)

Marcar v como visitado

Para cada vértice w adjacente a v

Se w não está marcado como visitado

Então efetuar a Travessia em Profundidade (w)

- Resultado ?
- Ficam marcados todos os vértices alcançados

Travessia em Profundidade – Depth-First

Travessia em Profundidade (vértice v)

Marcar v como visitado

Para cada vértice w adjacente a v

Se w não está marcado como visitado

Então efetuar a Travessia em Profundidade (w)

- Resultado ?
- Ficam marcados todos os vértices alcançáveis

Travessia em Profundidade – Depth-First

```
Travessia em Profundidade (vértice v)
    stack = Criar um STACK vazio
    Push(stack, v)
    Marcar v como visitado
    Enquanto NãoVazio(stack) fazer
        v = Pop(stack)
        Para cada vértice w adjacente a v
            Se w não está marcado como visitado
            Então Push(stack, w)
                Marcar w como visitado
```

Vértices alcançáveis

- Determinar o conjunto dos vértices alcançáveis significa encontrar um caminho entre o vértice inicial e cada um dos vértices alcançados
  - Pode não ser o caminho mais curto !!
  - Porquê ?

- Árvore de caminhos com raiz no vértice inicial
- Como registar a árvore ?
- Fácil : registar o predecessor de cada vértice no caminho a partir do vértice inicial
- Fazer o "traceback" para obter a sequência de vértices definindo o caminho

GraphDFSRec.c

```c
struct _GraphDFSRec {
  unsigned int* marked;
  int* predecessor;
  Graph* graph;
  unsigned int startVertex;
};
```

```c
static void _dfs(GraphDFSRec* traversal, unsigned int vertex) {
  traversal->marked[vertex] = 1;

  unsigned int* neighbors = GraphGetAdjacentsTo(traversal->graph, vertex);

  for (int i = 1; i <= neighbors[0]; i++) {
    unsigned int w = neighbors[i];
    if (traversal->marked[w] == 0) {
      traversal->predecessor[w] = vertex;
      _dfs(traversal, w);
    }
  }

  free(neighbors);
}
```

GraphDFSRec.h

```c
typedef struct _GraphDFSRec GraphDFSRec;

GraphDFSRec* GraphDFSRecExecute(Graph* g, unsigned int startVertex);

void GraphDFSRecDestroy(GraphDFSRec** p);

// Getting the result

unsigned int GraphDFSRecHasPathTo(const GraphDFSRec* p, unsigned int v);

Stack* GraphDFSRecPathTo(const GraphDFSRec* p, unsigned int v);

// DISPLAYING on the console

void GraphDFSRecShowPath(const GraphDFSRec* p, unsigned int v);
```

Travessia por Níveis – Breadth-First

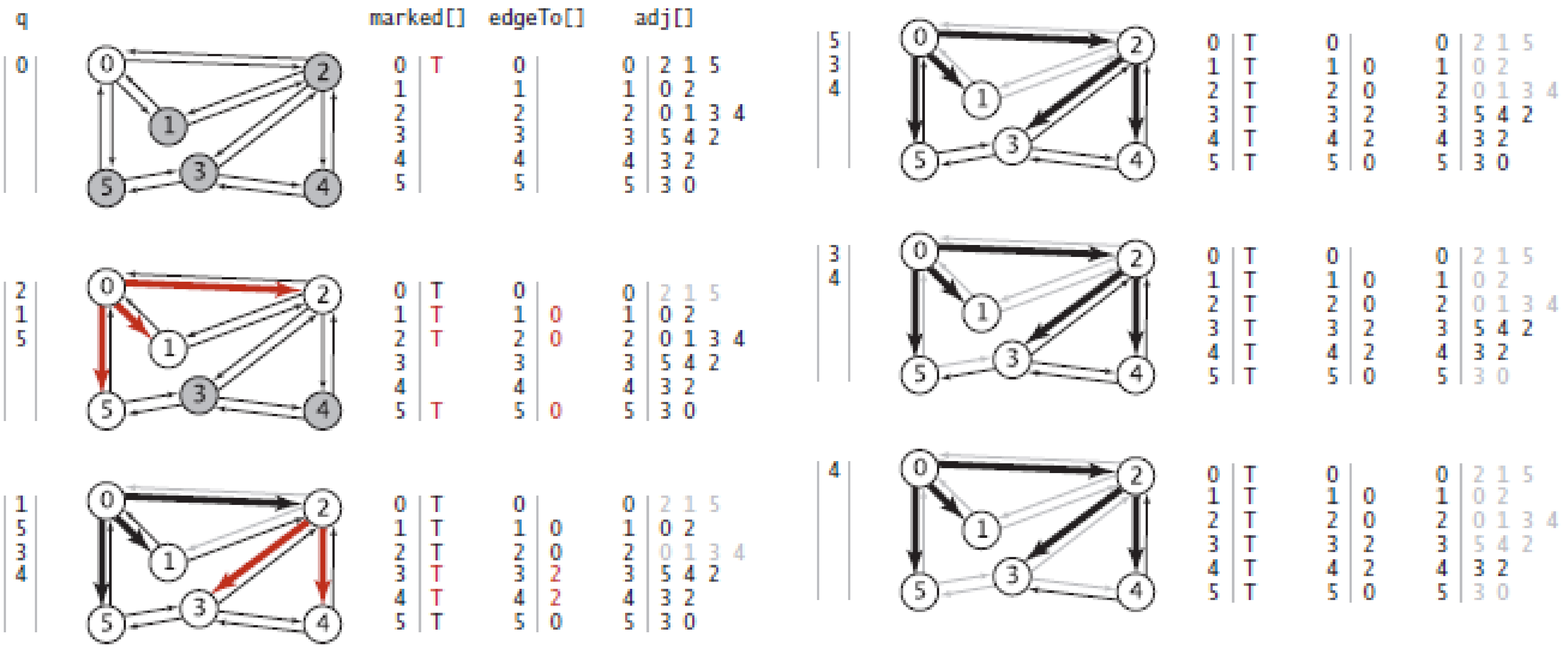
```
Travessia por Níveis(vértice v)
    queue = Criar FILA vazia
    Enqueue(queue, v)
    Marcar v como visitado
    Enquanto NãoVazia(queue) fazer
        v = Dequeue(queue)
        Para cada vértice w adjacente a v
            Se w não está marcado como visitado
            Então Enqueue(queue, w)
                Marcar w como visitado
```

Breadth-First – Caminhos mais curtos

- É encontrado o caminho mais curto entre o vértice inicial e cada um dos vértices alcançados
  - Porquê ?

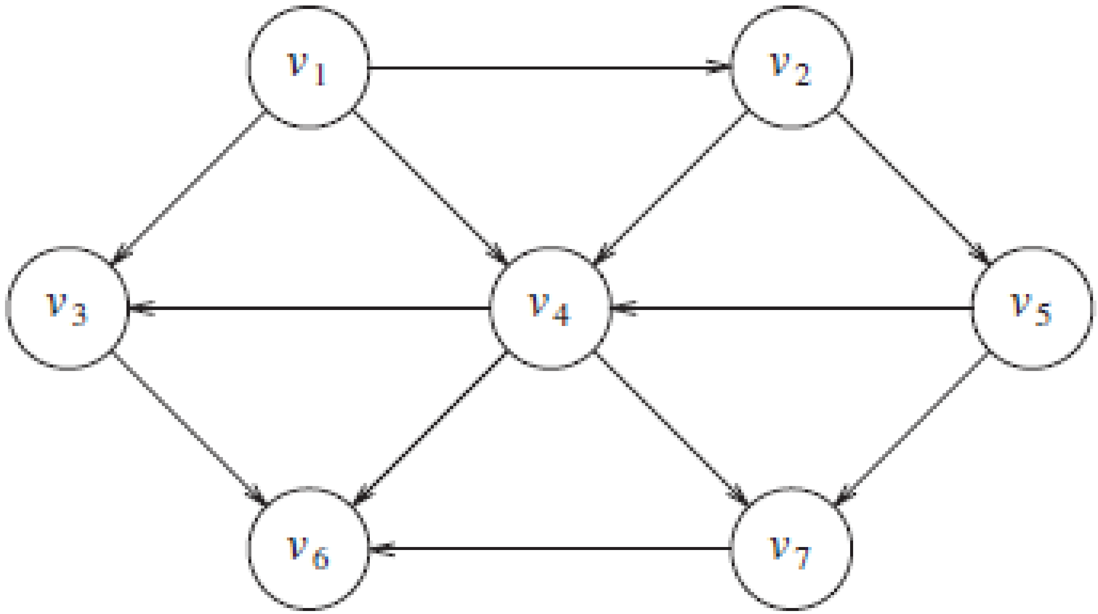
- Árvore de caminhos mais curtos com raiz no vértice inicial
- Registar o predecessor de cada vértice no caminho a partir do vértice inicial
- E a distância (i.e., nº de arestas) para o vértice inicial
- Fazer o "traceback" para obter a sequência de vértices definindo o caminho

Árvore dos caminhos mais curtos



[Sedgewick/Wayne]

Exemplo



- Possíveis sequências de vértices ?
- v1, v2, v5, v4, v3, v7, v6 OU v1, v2, v5, v4, v7, v3, v6
- Como determinar ?

Algoritmo – Manter o conjunto de candidatos

Registar num array auxiliar numEdges o InDegree de cada vértice

Criar uma FILA vazia e inserir na FILA todos os vértices v com numEdges[v] == 0

Enquanto a FILA não for vazia

    v = retirar próximo vértice da FILA

    Imprimir o seu ID

    Para cada vértice w adjacente a v

        numEdges[w] --

        Se numEdges[w] == 0 Então Inserir w na FILA

- PROBLEMA : o que acontece se existir um ciclo ??

Caminhos Mais Curtos

Caminho mais curto ?

Caminhos Mais Curtos

- Problema de otimização combinatória
- De todas as soluções possíveis, determinar a de menor custo/distância
- Podem existir soluções ótimas alternativas
    - Caminhos distintos com o mesmo custo/distância total

- Grafo / Grafo Orientado : contar o nº de arestas do caminho
- Rede : somar o valor de distância associado a cada aresta do caminho

Ciclos ?

- Um caminho mais curto não contém um ciclo !! Porquê ?

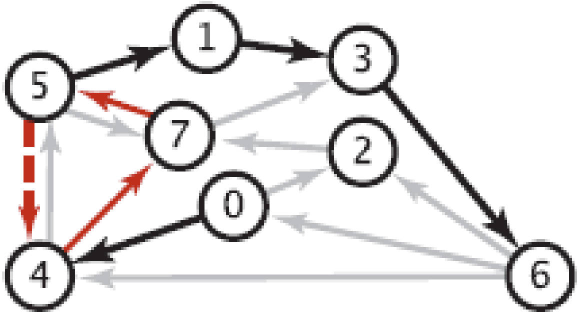
- Ciclo de custo positivo vs Ciclo de custo negativo

- Num grafo conexo e com um ciclo negativo não é possível definir um caminho mais curto !!
- Num grafo orientado fortemente conexo e com um ciclo negativo não é possível definir um caminho mais curto !!
- Porquê ?

Exemplo

digraph

4->5	0.35
5->4	-0.66
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



negative cycle (-0.66 + 0.37 + 0.28)

5->4->7->5

shortest path from 0 to 6

0->4->7->5->4->7->5...->1->3->6

[Sedgewick & Wayne]

Vários problemas – Determinar :

- O caminho mais curto entre um vértice s e um vértice t
- O caminho mais curto entre um vértice s e cada um dos outros vértices alcançáveis a partir de s
  - Árvore dos caminhos mais curtos com raiz no vértice s
- O caminho mais curto entre qualquer par de vértices
- Os K caminhos mais curtos entre um vértice s e um vértice t
- …

Árvore dos caminhos mais curtos de s para t

- Associar um rótulo (“label”) a cada vértice : (dist[v], pred[v])

- Como inicializar ?

- No final do algoritmo o que representa ?
- pred[v] : o predecessor no caminho mais curto a partir de s
- dist[v] : o custo/distância associado ao caminho mais curto a partir de s

- Fazer o “traceback” do caminho mais curto !!

Inicialização dos rótulos

- Para cada vértice $v \neq s$

$$dist[v] = +\infty \qquad\qquad pred[v] = -1$$

- Para o vértice s

$$dist[s] = 0 \qquad\qquad pred[s] = -1$$

Em que caso já sabemos resolver ?

- Grafo / Grafo orientado
- SEM custos / distâncias associados às arestas

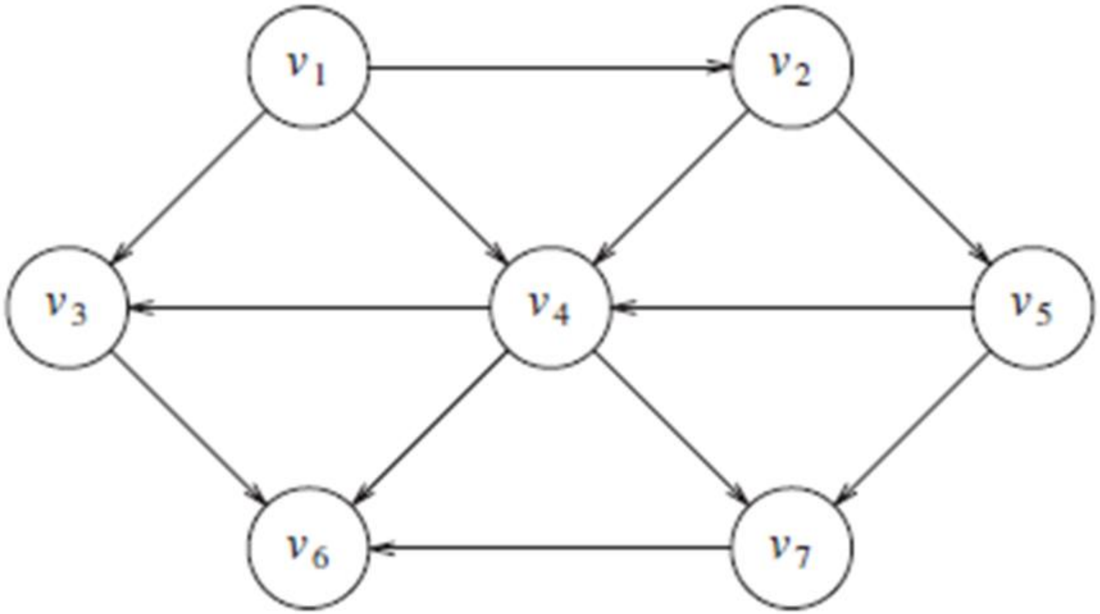
- Usar a travessia por níveis !!

Travessia por Níveis – Breadth-First

```
queue = Criar FILA vazia
Enqueue(queue, s)
Enquanto NãoVazia(queue) fazer
        v = Dequeue(queue)
        Para cada vértice w adjacente a v
                Se dist[w] == +∞
                Então Enqueue(queue, w)
                        dist[w] = dist[v] + 1
                        pred[w] = v
```

Tarefa – Executar o algoritmo

- Qual é o caminho mais curto entre v1 e v5 ?
- E o caminho mais curto entre v1 e v6 ?
- Há caminhos ótimos alternativos ?
- De que depende a sua escolha ?



[Weiss]

O Algoritmo de Bellman-Ford

Algoritmo de Bellman-Ford

- Versátil : permite pesos negativos nas arestas
- MAS não um ciclo negativo
  - Já sabemos…

- Mais lento do que algoritmos alternativos !! Porquê ?
- Como melhorar ?

- Ordem de complexidade ?
- Melhor caso vs Pior caso

Algoritmo

```
Inicializar os rótulos dos vértices
Para i = 1 até (numVértices – 1) fazer          // (V–1) vezes
        Para cada aresta (u, v) fazer
                Se dist[u] + peso(u,v) < dist[v]   // Alternativa
                Então                              // Atualizar
                        dist[v] = dist[u] + peso(u,v)
                        pred[v] = u
```

Como melhorar ?

- Em que condição pode o ciclo externo ser executado menos vezes ?

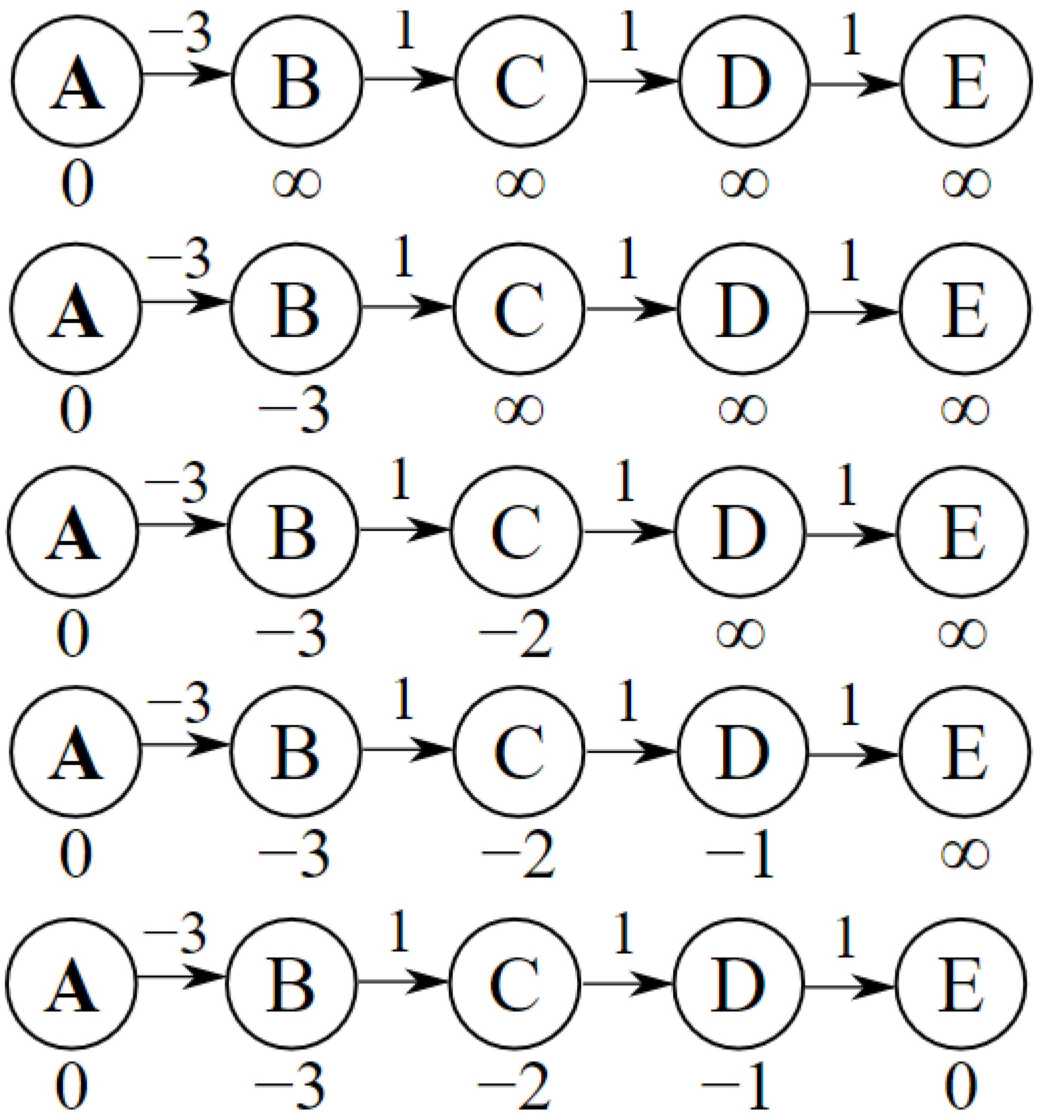
- A partir do instante em que há a certeza de que nenhum rótulo poderá vir a ser melhorado !!

- Como verificar ? -> Usar uma flag !!

Exemplo simples

- Arestas processadas da esquerda para a direita
  - São suficientes 2 iterações do ciclo externo

- Arestas processadas da direita para a esquerda
  - São necessárias as 4 iterações do ciclo externo

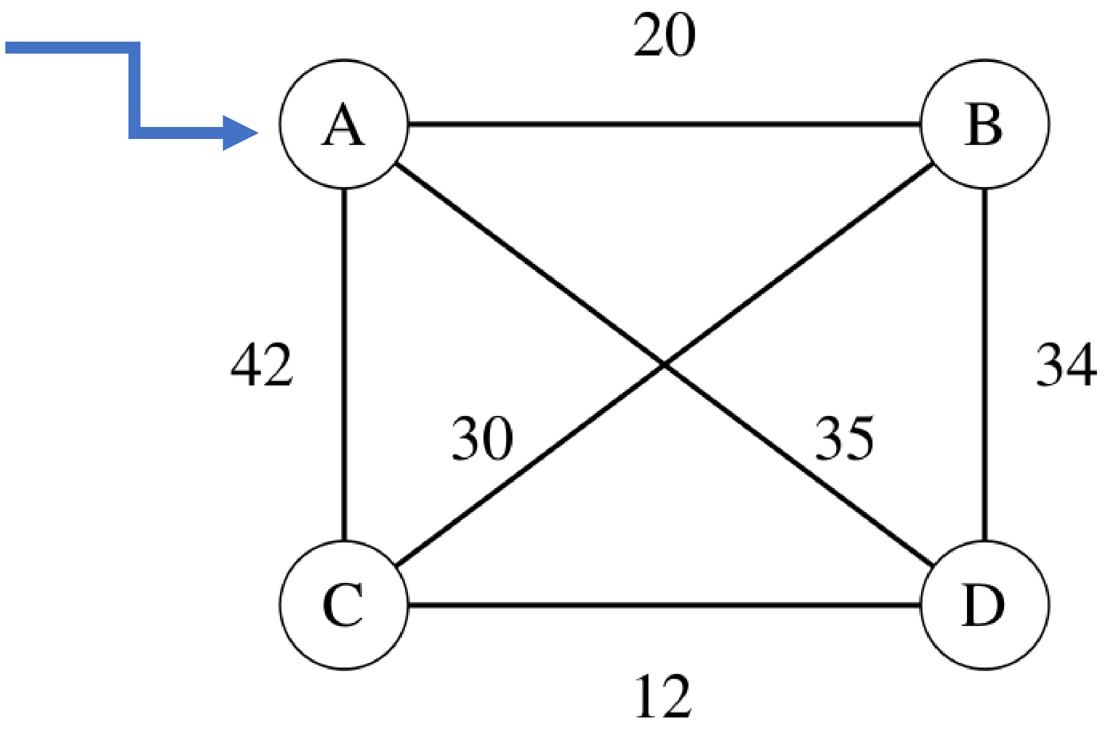


[Wikipedia]

Ordem de complexidade

- Nº de comparações
- Pior Caso: **O(V x E)**
- Melhor Caso: **O(V)**
- Como melhorar ?

Tarefa – Executar o algoritmo



[Wikipedia]

Usar uma QUEUE/STACK para referenciar os candidatos

IDEIA

- Não é necessário percorrer sempre todo o conjunto de arestas
- É suficiente considerar as arestas que poderão levar à correção dos rótulos de alguns vértices
- Quais são ?
- As arestas (u, v) para as quais o rótulo do vértice u foi alterado !!
- Como fazer ?
- Manter um conjunto dos vértices cujos rótulos foram alterados
  - Os vértices candidatos

Conjunto de vértices candidatos

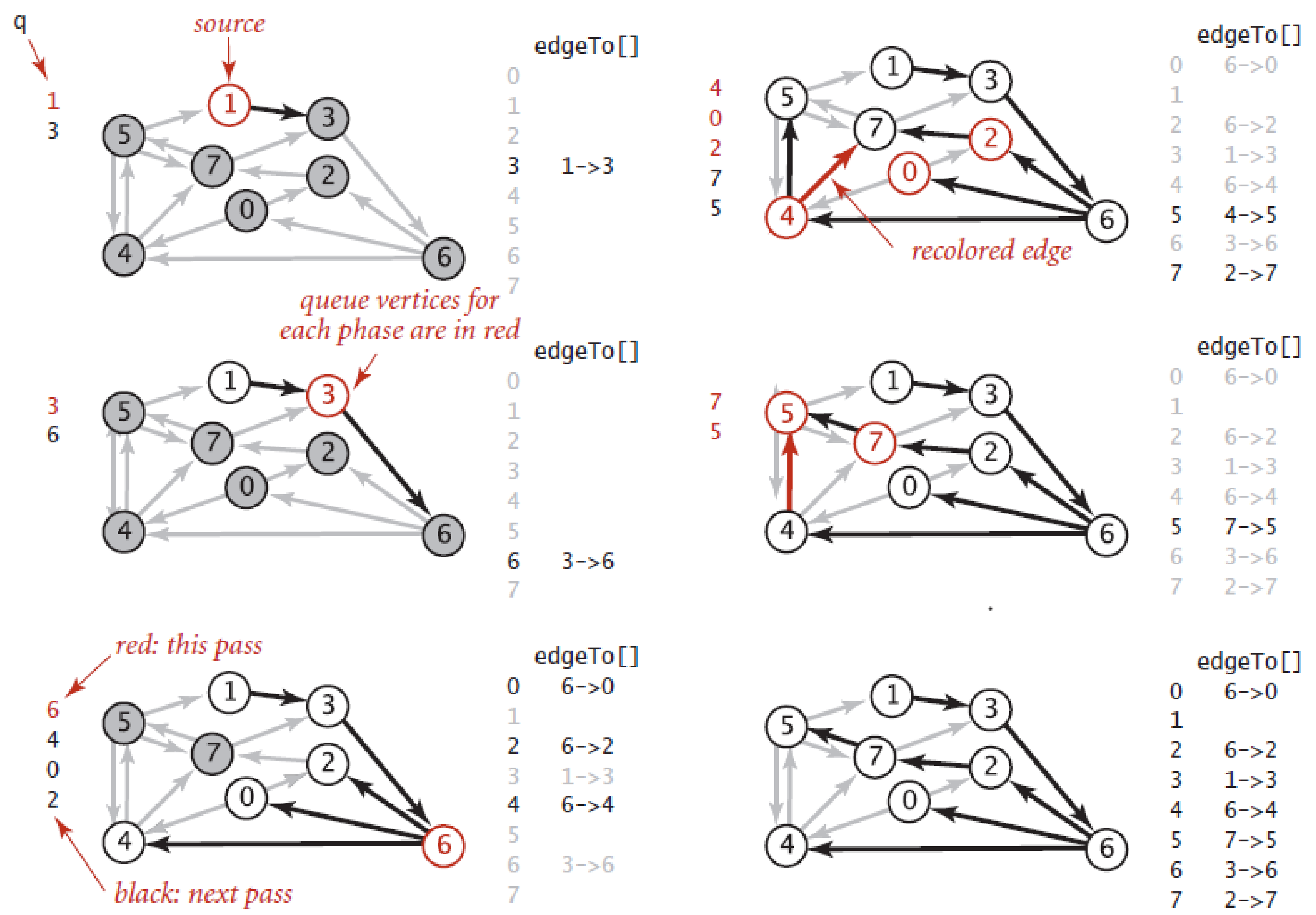
- Usar STACK ou QUEUE, como nas travessias
- STACK : grafo esparso – porquê ?
- QUEUE : grafo denso – porquê ?

- Há outras estruturas de dados ou regras para escolher o próximo vértice candidato a ser explorado
- DEQUE / …

Algoritmo

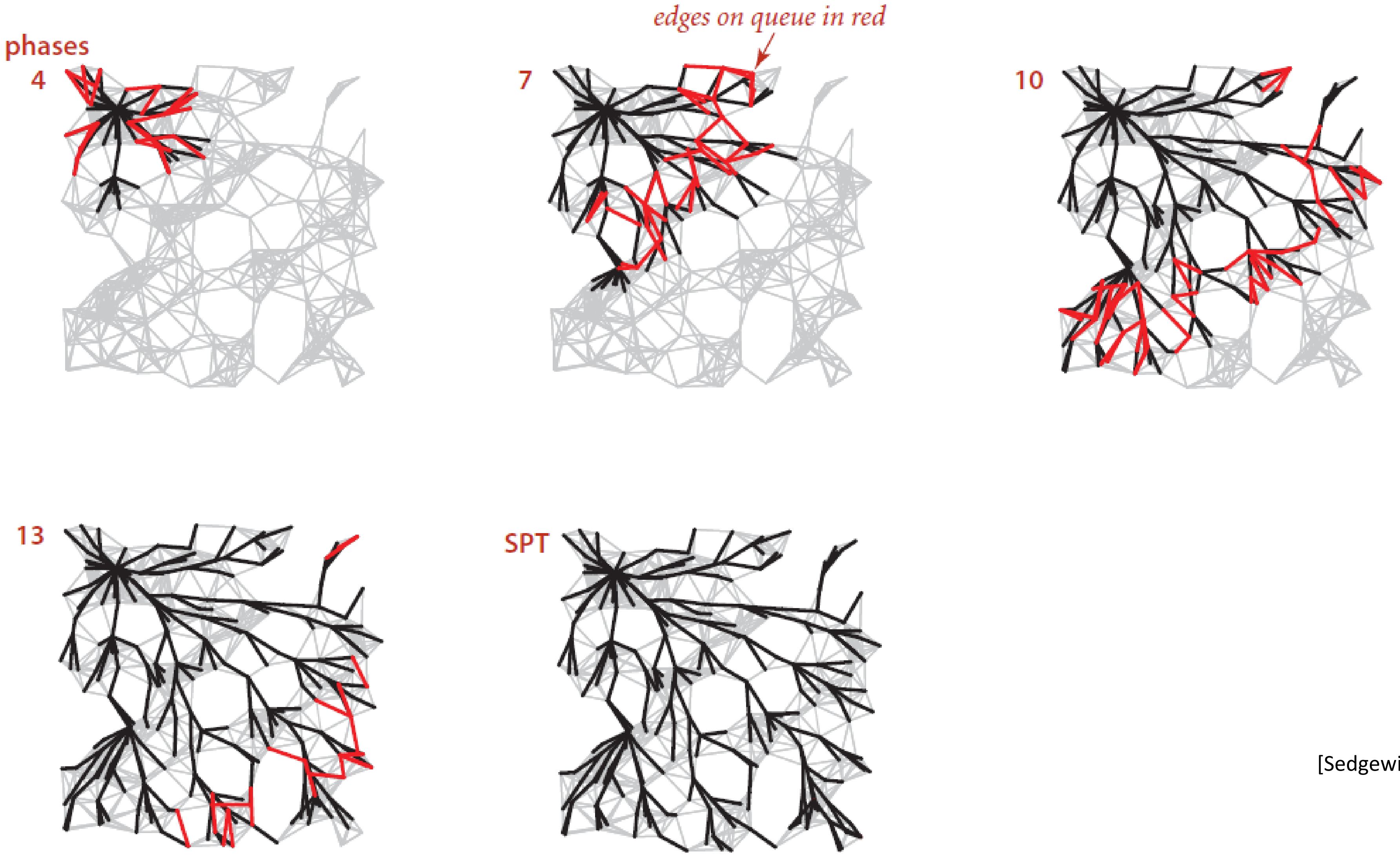
```
Inicializar os rótulos dos vértices
conjCandidatos = { s };
Enquanto conjCandidatos ≠ { } fazer
    u = próximoElemento(conjCandidatos);            // Depende da EDados
    conjCandidatos = conjCandidatos – {v};
    Para cada vértice v adjacente a u
        Se dist[u] + peso(u,v) < dist[v]             // Alternativa
        Então  dist[v] = dist[u] + peso(u,v);        // Atualizar
               pred[v] = u;
               Se v não pertence conjCandidatos
               Então conjCandidatos = conjCandidatos U { v };
```

Exemplo usando uma FILA / QUEUE



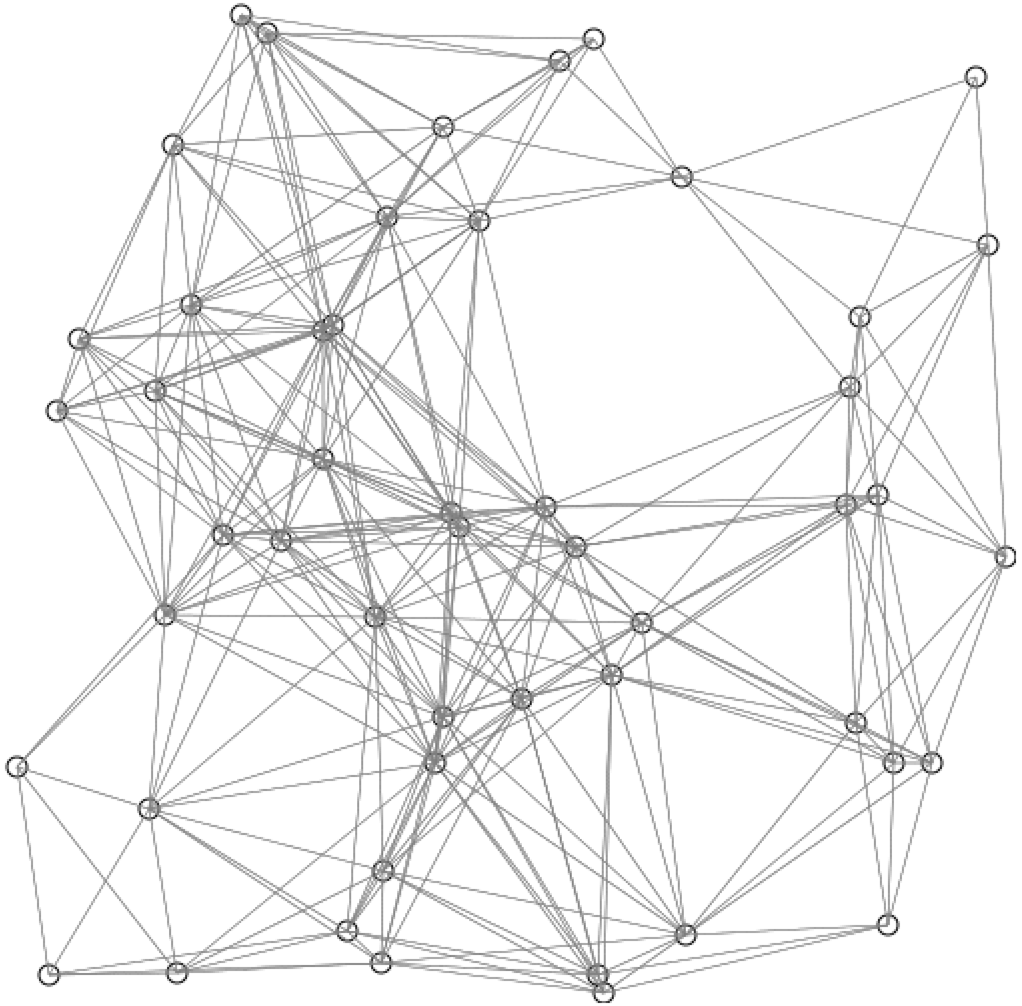
[Sedgewick & Wayne]

Exemplo usando uma FILA / QUEUE



[Sedgewick & Wayne]

Exemplo usando uma FILA / QUEUE

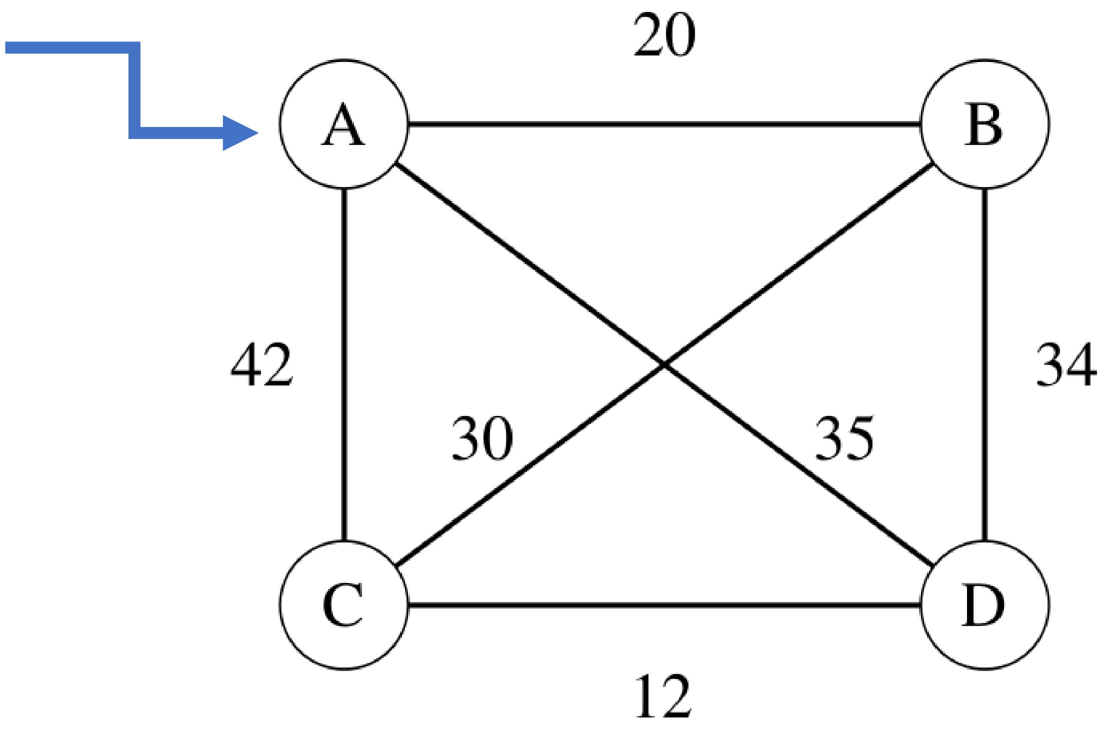


[Wikipedia]

Ordem de complexidade

- Nº de comparações
- Pior Caso : **O(V x E)**
- Na prática é mais eficiente : **O(V + E)**
- Como melhorar ?

Tarefa – Executar o algoritmo



[Wikipedia]

O Algoritmo de Dijkstra

Estratégia voraz / “greedy”

- Construir a solução passo-a-passo
- Efetuar a escolha ótima em cada instante
- Essa escolha é irreversível !!

- No caso do Algoritmo de Dijkstra, a estratégia voraz conduz sempre à solução ótima

- MAS, para outros problemas/algoritmos isso pode não acontecer e obtemos apenas uma aproximação da solução ótima
  - Heurística vs Algoritmo

IDEIA

- Determinar os sucessivos caminhos mais curtos de acordo com a ordem das suas distâncias para o vértice inicial s
- Qual é o próximo vértice a ser adicionado à árvore dos caminhos mais curtos ?
  - O que tiver, nesse instante, a menor distância para o vértice s
  - Estratégia voraz / "greedy"
- Manter ordenado o conjunto de vertices candidatos

ATENÇÃO

- Não são permitidas arestas com pesos negativos !!

- Porquê ?

Estrutura de dados

- Como manter ordenado o conjunto dos vértices candidatos ?
- Ordem parcial vs Ordem total

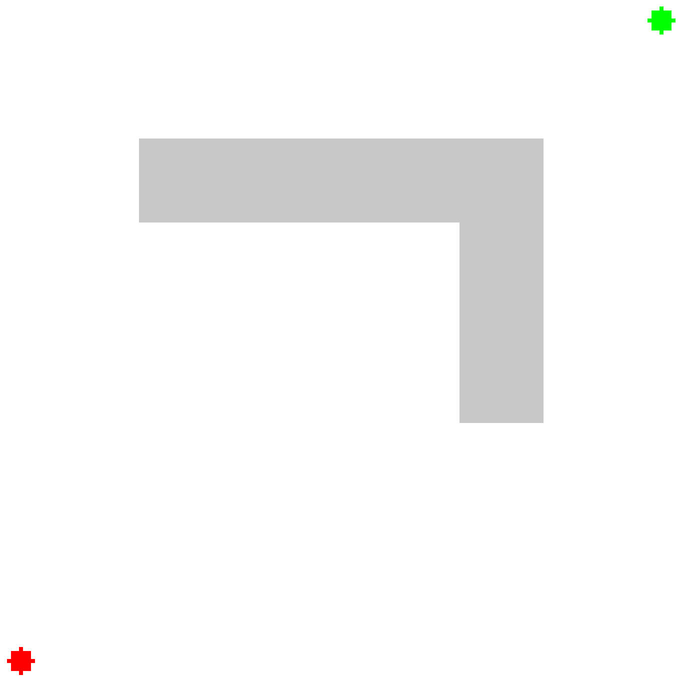
- Usar um MIN-HEAP / PRIORITY QUEUE
- Obter o próximo vértice candidato sem grande esforço computacional

- Há outras estruturas de dados que se podem usar
- A ordem de complexidade do algoritmo depende da estrutura de dados escolhida

Algoritmo

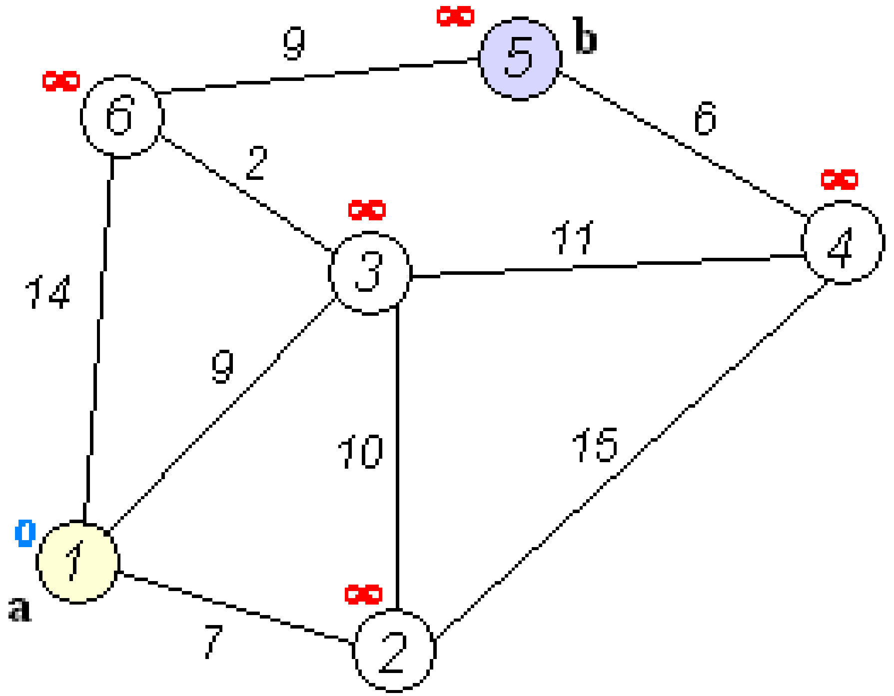
```
Inicializar os rótulos dos vértices
conjCandidatos = { s };
Enquanto conjCandidatos ≠ { } fazer
    u = removerMenor(conjCandidatos);        // Reordenação implícita
    Para cada vértice v adjacente a u que ainda não pertence à solução
        Se dist[u] + peso(u,v) < dist[v]
        Então  dist[v] = dist[u] + peso(u,v);
               pred[v] = u;
               Se v não pertence conjCandidatos
               Então conjCandidatos = conjCandidatos U { v };
               Senão reposicionar v no conjunto ordenado de candidatos
```

Exemplo – Robot Motion Planning



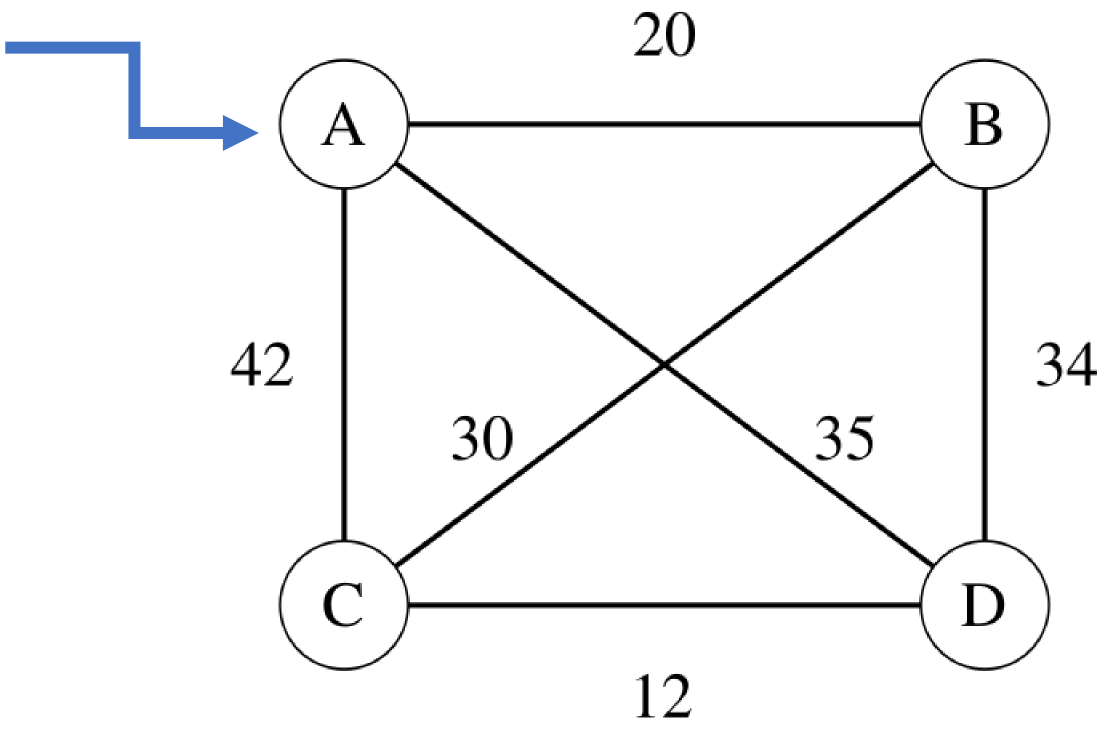
[Wikipedia]

Exemplo



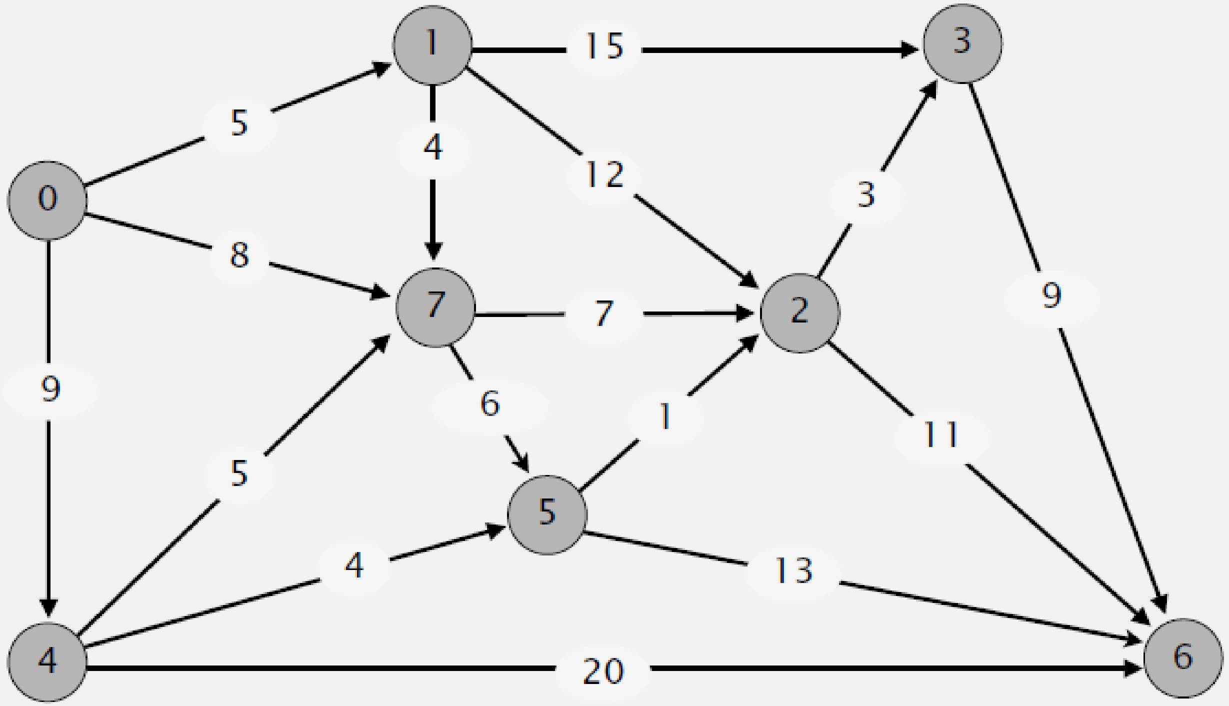
[Wikipedia]

Tarefa – Executar o algoritmo



[Wikipedia]

Tarefa – Executar o algoritmo

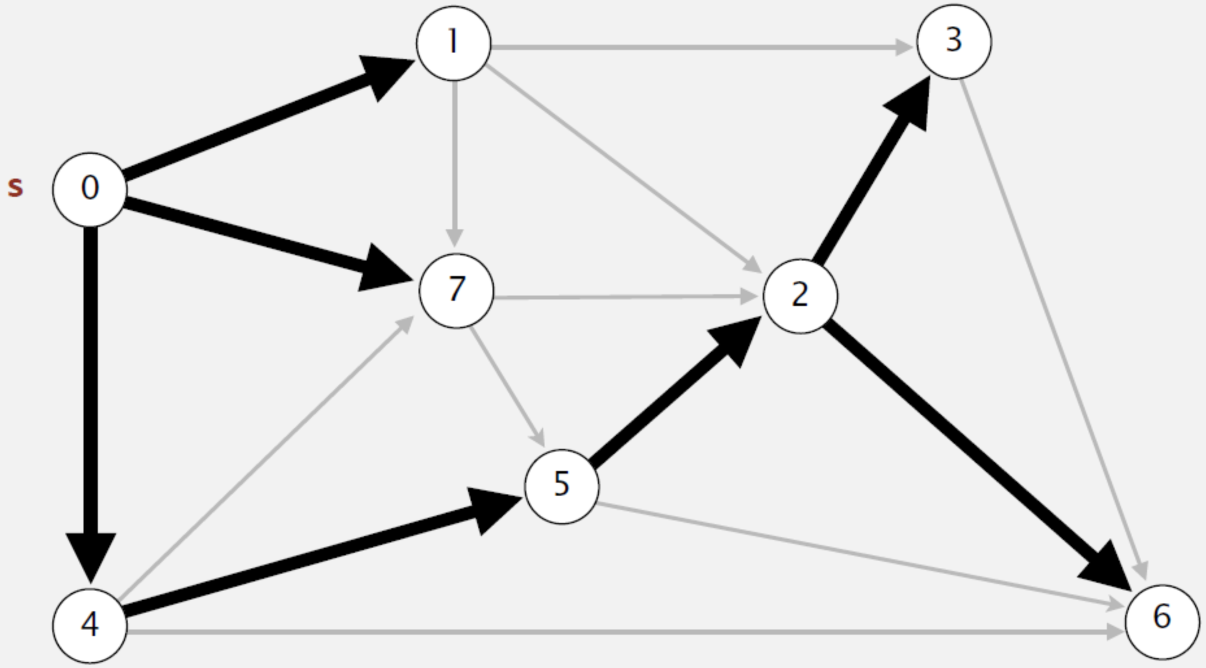


an edge-weighted digraph

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

[Sedgewick & Wayne]

Árvore dos caminhos mais curtos

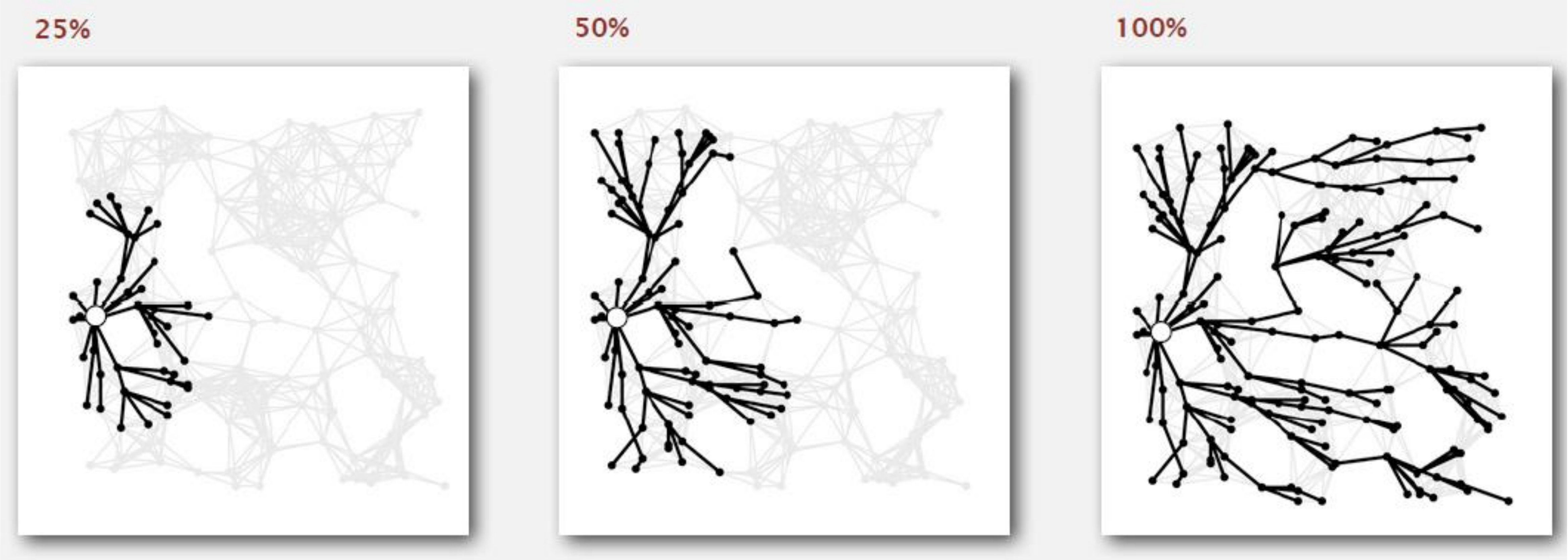


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

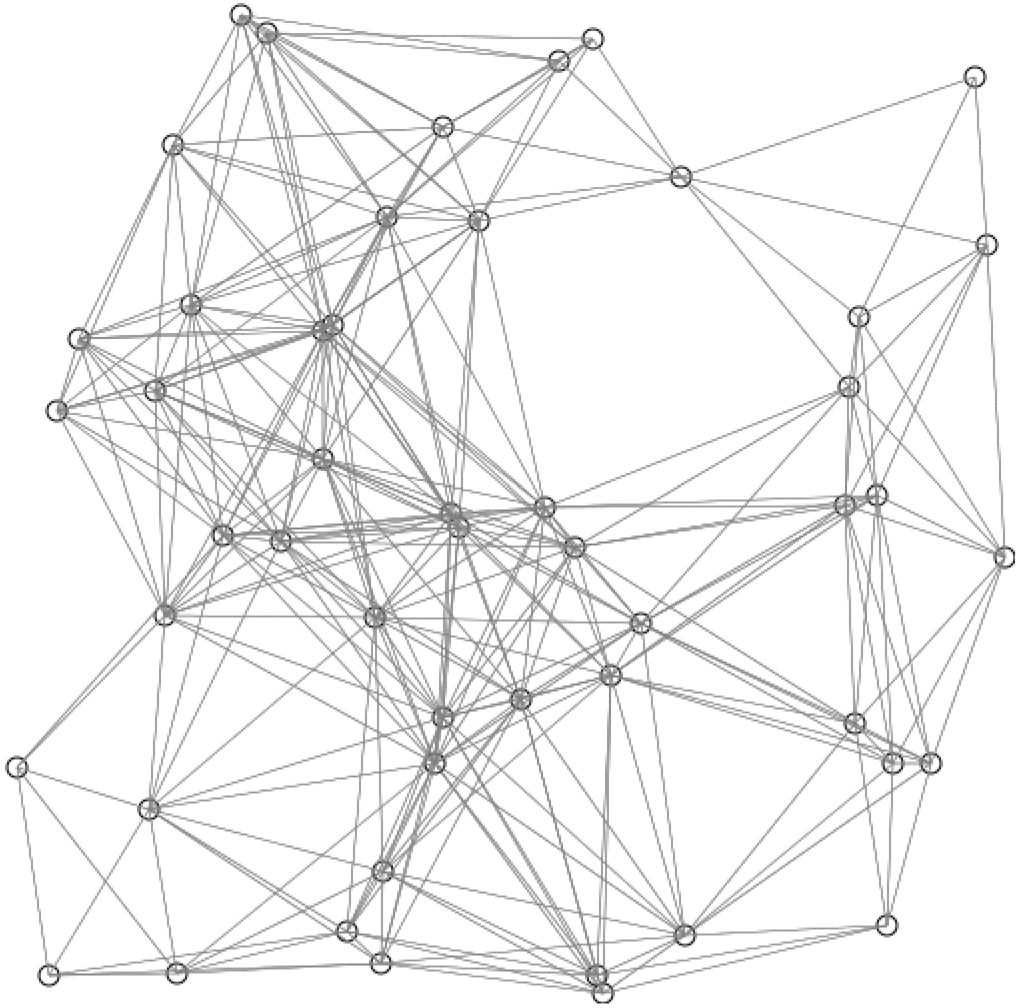
[Sedgewick & Wayne]

Exemplo



[Sedgewick & Wayne]

Exemplo – Distância Euclideana



[Wikipedia]

Fila com Prioridade – Qual escolher ?

PQ implementation	insert	delete-min	decrease-key	total
array	1	V	1	V^2
binary heap	log V	log V	log V	E log V
d-way heap (Johnson 1975)	$d \log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap (Fredman-Tarjan 1984)	1 †	log V †	1 †	E + V log V

† amortized

[Sedgewick & Wayne]

Ordem de complexidade – MIN-HEAP binária

- Nº de comparações
- Pior Caso : **O(E log V)**
- Casos típicos : **O(E log V)**

Tarefa

- Analisar os exemplos nos ficheiros adicionais

Sugestões de Leitura

Sugestões de leitura

- M. A. Weiss, *“Data Structures and Algorithm Analysis in C++”*, 4th. Ed., Pearson, 2014
  - Chapter 9

- R. Sedgewick and K. Wayne, “*Algorithms*”, 4th. Ed., Addison-Wesley, 2011
  - Chapter 4