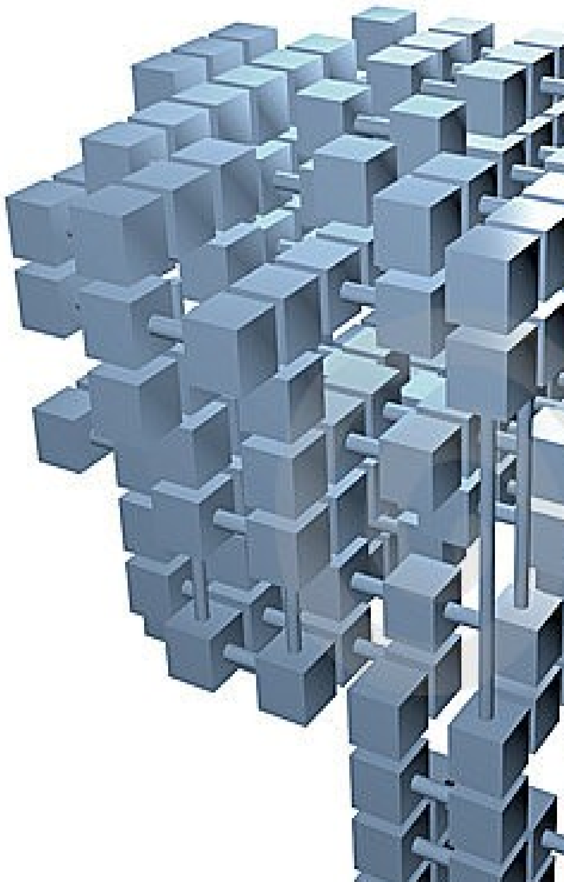
Campus de Frutal

UEMG UNIVERSIDADE DO ESTADO DE MINAS GERAIS

SISTEMAS DE INFORMAÇÃO

Estrutura de Dados 1

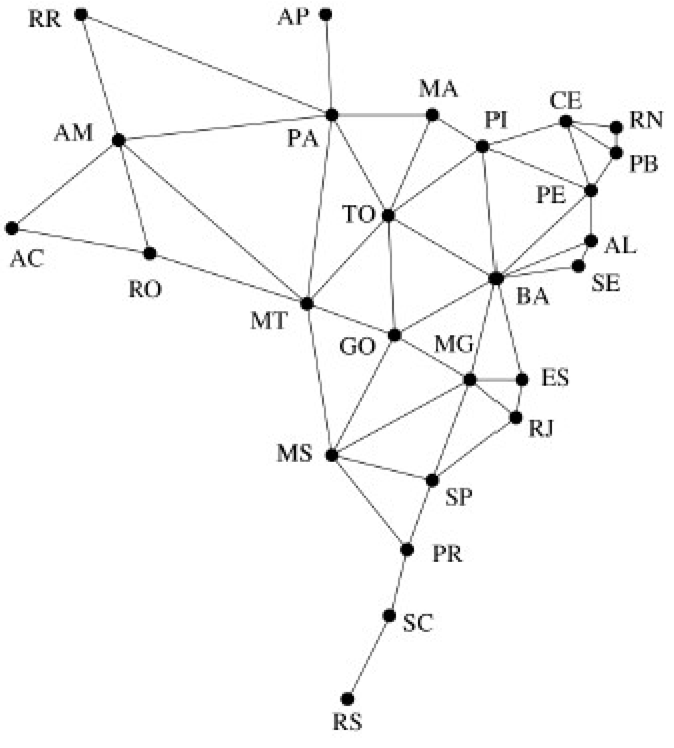
Grafos



Prof. Ivan José dos Reis Filho
ivanfilhoreis@gmail.com

Grafos

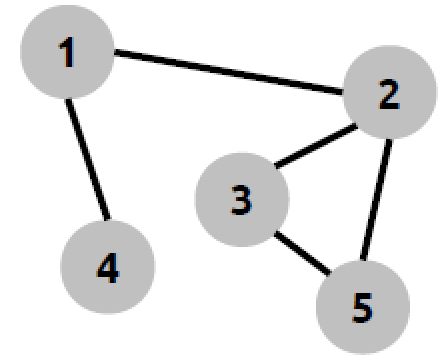
- Modelo formal para representar relacionamentos entre objetos:
  - Exemplos: geográficos, web sites, circuitos elétricos, redes de computadores, redes sociais, moléculas, etc ...
  - Utilização dos algoritmos de teoria dos grafos
  - Garantias teóricas de “melhor”, “Mínimo”, “Máximo”, etc ...



Grafos

- Aplicações abstraídas em grafos (***G***):

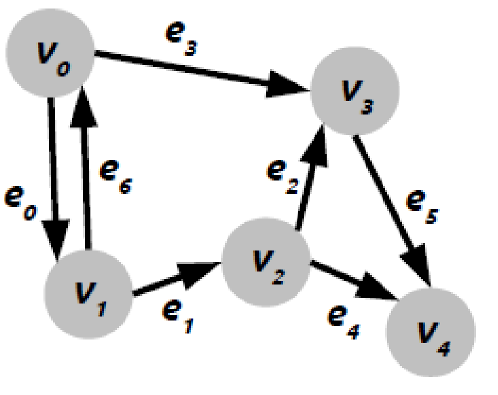
Aplicação	Vértices	Arestas
Geográfica	Cidades	Estradas
Web Sites	Páginas	Links
Circuitos	Componente	Fio
Redes	Computador	Link (cabo)
Rede Social	Pessoas	Amizades



- ***G=(V,E)***, onde:
  - ***V*** é um conjunto de vértices (ou nós, ou pontos)
  - ***E*** é um conjunto de arestas (ou linhas, ou arcos), sendo formado por dois vértices de ***V***
  - Exemplo:
    - ***V = { 1, 2, 3, 4, 5 } e E = { {1,2}, {1,4}, {2,3}, {2,5}, {3,5} }***

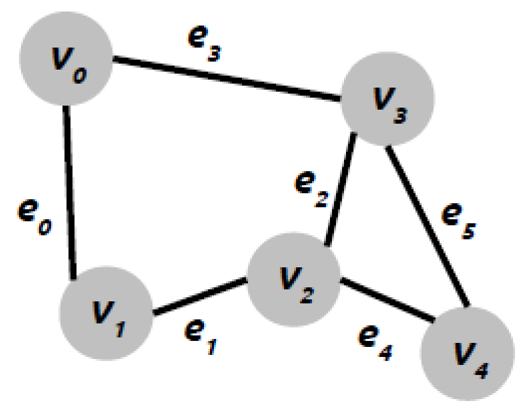
$G=(V,E)$

- ***E*** pode ser um conjunto de pares **ordenados**:
  - Aresta direcionadas (<u>dígrafo</u>)
  - Vértice <u>origem</u> e vértice <u>destino</u>
  - $V = \{ v_0, v_1, v_2, v_3, \dots \}$ e $E = \{ e_0, e_1, e_2, e_3, \dots \}$
  - $e_0 = \{v_0, v_1\}$ e $e_6 = \{v_1, v_0\}$
  - $e_0 \neq e_6$
  - $\{v_3, v_2\} \notin E$
  - $\{v_2, v_3\} \in E$



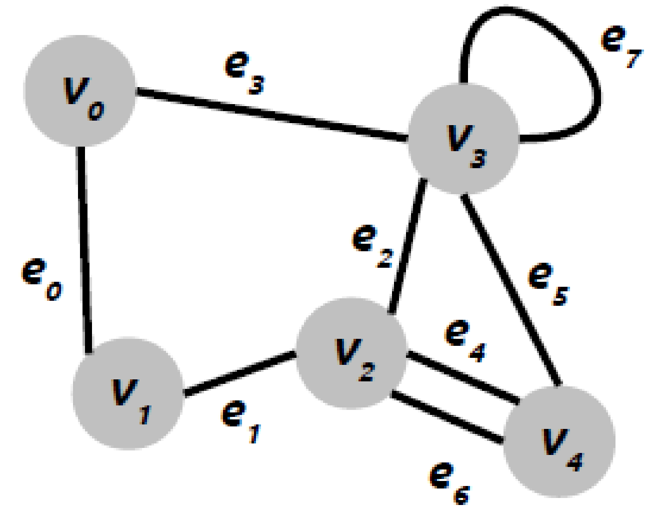
G=(V,E)

- ***E*** pode ser um conjunto de pares **não-ordenados**:
  - Aresta não direcionada
  - $V = \{ v_0, v_1, v_2, v_3, \ldots \}$ e $E = \{ e_0, e_1, e_2, e_3, \ldots \}$
  - $e_0 = \{v_0,v_1\}$ ou $e_0 = \{v_1,v_0\}$
  - $\{v_3,v_2\}$ Î E
  - $\{v_2,v_3\}$ □ E

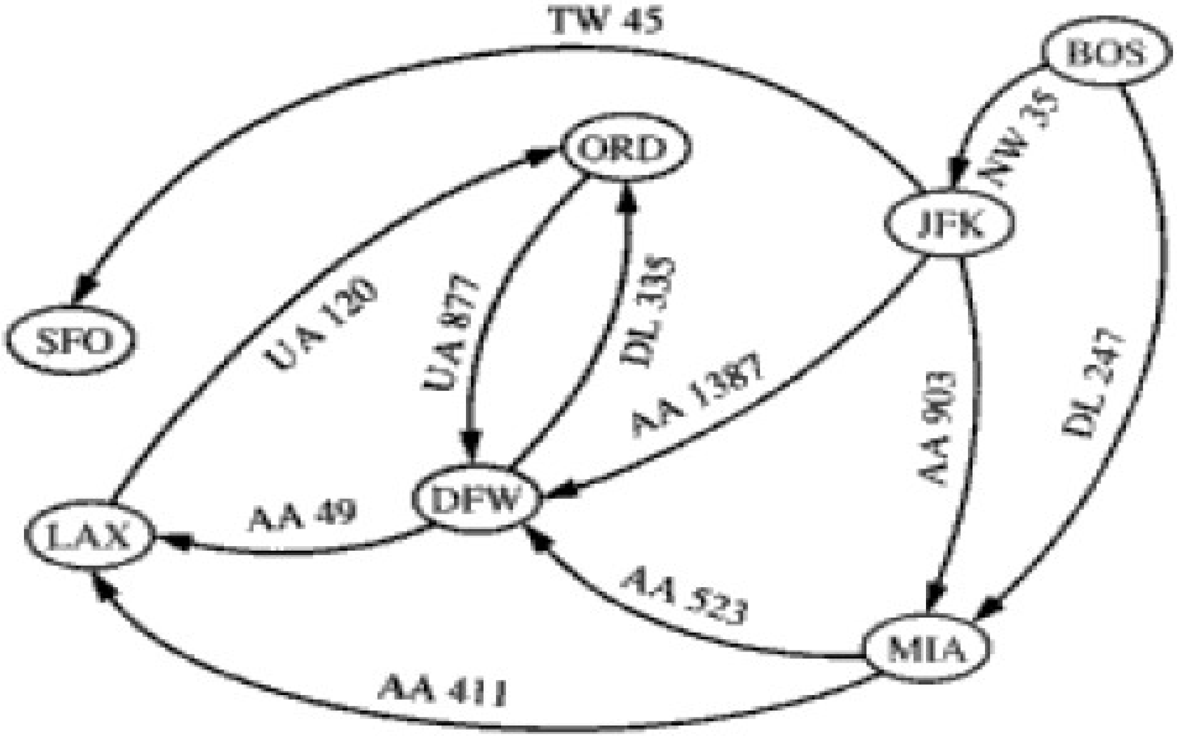


G=(V,E)

- ***v_0 e v_1 são vértices adjacentes***
- A aresta ***e_5 é <u>incidente</u> ao vértice v_3***
- Os vértices ***v_0 e v_1 são vértices finais da aresta e_0***
- O vértice ***v_1 possui grau 2***
  - Em dígrafos: grau de “chegada” e grau de “saída”
- Anomalias:
  - As arestas ***e_4 e e_6 são paralelas (ou múltiplas)***
  - A aresta ***e_7 é um laço***



Exemplo



- Grafo Ordenado representando um rede de voos. Os pontos finais da aresta UA 120 são LAX e ORD. Portanto, LAX e ORD são adjacentes. O grau de entrada DFW é 3 e o de saída é 2

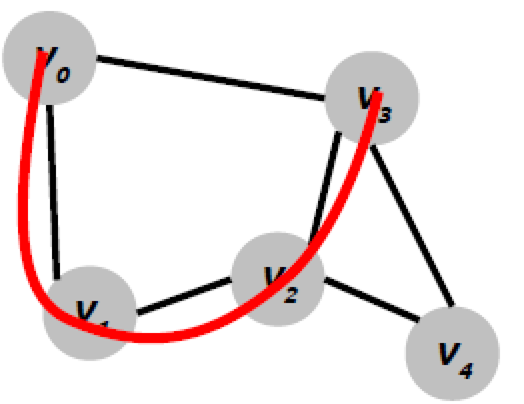
Grafos

- Um caminho (*path*) é uma sequência de vértices

$$C=\{v_0, v_1, v_2, \ldots, v_{k-1}, v_k\}$$

onde existem arestas conectando cada par adjacente desta sequência

$$E=\{v_0v_1, v_1v_2, \ldots, v_{k-1}v_k\}$$

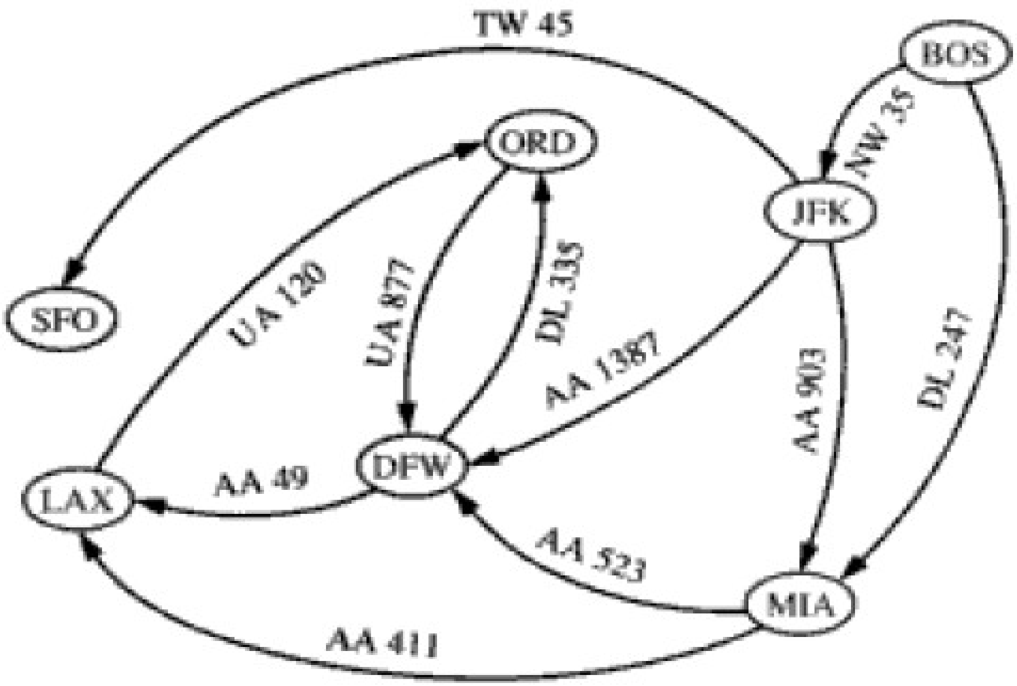


Grafos

- Um caminho é dito ser simples se todos os vértices e arestas são distintos.
- Um ciclo é um caminho onde o primeiro vértice é igual ao último.
- Um ciclo é simples quando todos os vértices e arestas são distintos (exceto o último vértice).

Grafos

- Um grafo é dito conexo, se para quaisquer 2 vértices existe um caminho entre eles;



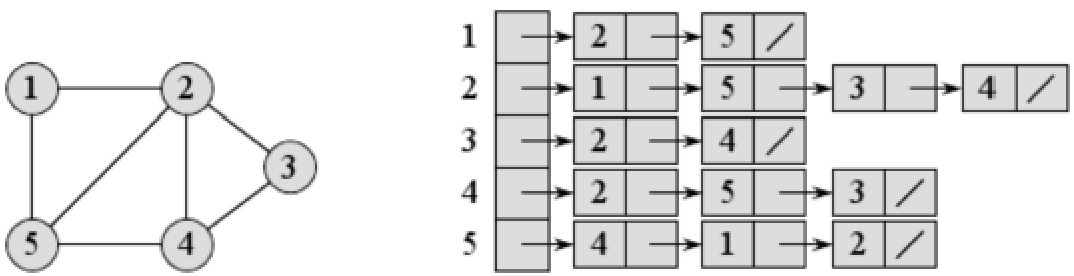
V(BOS, JFK e MIA) e E(AA 903 e DL 247): Formam um subgrafo.

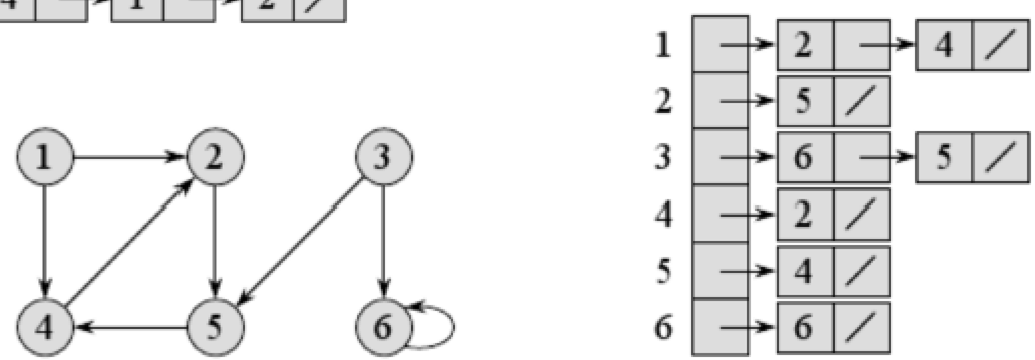
Representação Computacional

- Como armazenar um grafo?
  - Lista de arestas
  - Vértices com listas dos adjacentes
  - Matriz de adjacência
  - ...
- Memória: densidade do grafo
- Processamento: otimização para as operações mais frequentes

Representação Computacional

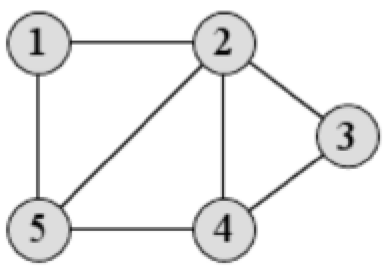
- Lista de adjacências:
  - Consiste em um vetor (ou lista) de vértices, onde para cada vértice temos uma lista de vértices adjacentes;



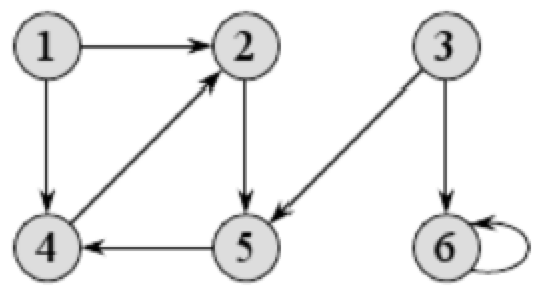


Representação Computacional

- Matriz de adjacência:
  - Consiste em uma matriz quadrada de dimensão **|V|**, onde para cada elemento **a_{ij}=1** se existir uma aresta entre **v_i** e **v_j**, ou **a_{ij}=0** caso contrário;



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



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

Representação Computacional

- Outras formas:
  - Lista de arestas:
    - Consiste em um vetor (ou lista) de arestas, onde para cada aresta temos os dois vértices adjacentes (finais);
  - Vértices com listas de arestas
    - Consiste em um vetor (ou lista) de vértices, onde para cada vértice temos uma lista de arestas incidentes;
  - ...

Representação Computacional

- Qual é o melhor? depende ...
  - Densidade dos grafos
  - Operações:
    - Quais os vértices vizinhos de **v?**
    - Quais as arestas incidentes em **v?**
    - **v_0 e v_1 são adjacentes ?**
    - **e_0 e e_1 são adjacentes ?**
    - Inserir um novo vértice
    - Inserir uma nova aresta
    - ...

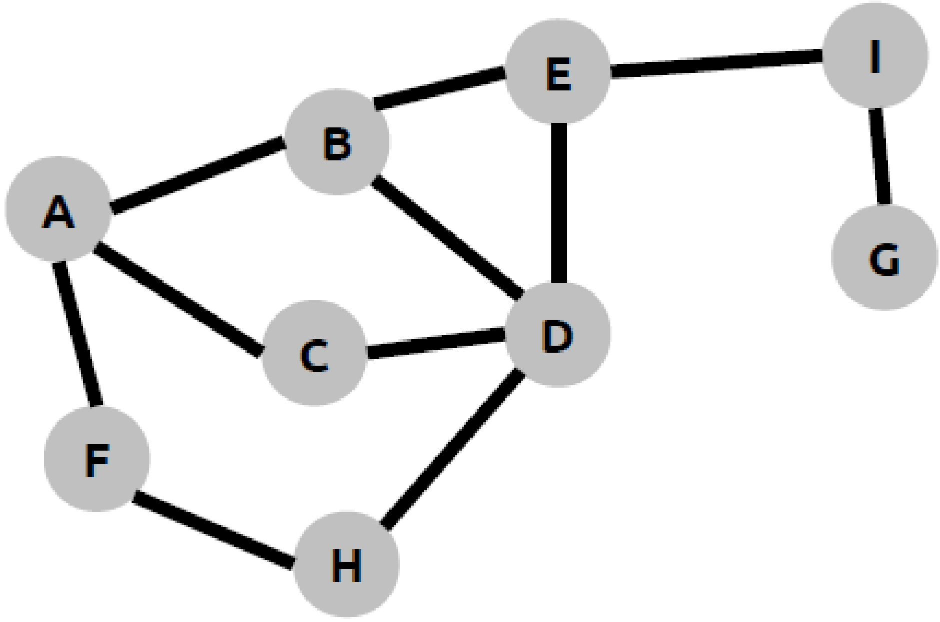
Percursos

- Em profundidade (*Depth-First)*
- Em largura (Breadth-*First)*

- Podemos utilizar em:
  - Buscas
  - Detectar componentes conexas
  - Obter uma arvore geradora
  - Achar um caminho entre 2 vértices
  - Detectar ciclos

Percurso em Profundidade

- Exemplo:



Percurso em Profundidade

- Versão recursiva:

```
void depthFirstRecursive( Graph &g, int vertex )
{
    g.setVisited( vertex );

    // processing(vertex)

    for (int k=0; k < g.numberOfNeighbours(vertex); k++) {
        int nb = g.neighbour(v,k);
        if ( !g.isVisited( nb ) ) {
            depthFirstRecursive( g, nb );
        }
    }
}
```

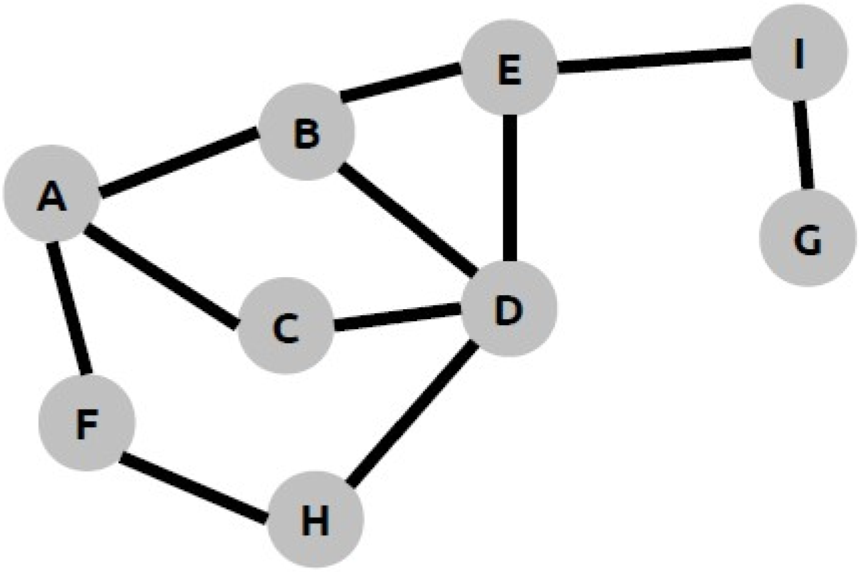
Percurso em Profundidade

- Versão iterativa:

```
void depthFirst( Graph &g, int vertex )
{
    Stack<int> s;
    s.push( vertex );
    g.setVisited( vertex );

    while ( ! s.empty() ) {
        int v = s.pop();

        // processing(v)

        for (int k=0; k < g.numberOfNeighbours(v); k++) {
            int nb = g.neighbour(v,k);
            if ( !g.isVisited( nb ) ) {
                s.push( nb );
                g.setVisited( nb );
            }
        }
    }
}
```

Percurso em Largura

- Exemplo:



Percurso em Largura

- Versão iterativa:

```
void breadthFirst(Graph &g, int vertex)
{
    Queue<int> q;
    q.put(vertex);
    g.setVisited(vertex);

    while (!q.empty()) {
        int v = q.get();

        // processing(v)

        for (int k=0; k < g.numberOfNeighbours(v); k++) {
            int nb = g.neighbour(v,k);
            if ( !g.isVisited( nb ) ) {
                q.put( nb );
                g.setVisited( nb );
            }
        }
    }
}
```

Utilização dos Percursos

- Como adaptamos para resolver cada um dos problemas abaixo?
  - Buscas
  - Detectar componentes conexas
  - Obter uma arvore geradora
  - Achar um caminho entre 2 vértices
  - Detectar ciclos

Buscas

- Começar em um vértice “qualquer” e parar quando encontrar o vértice procurado
  - Análogo a busca linear em um lista
  - Inviável para grandes grafos
- Exemplo:

```
int depthFirstSearch( Graph<T> &g, T key )
{
    Stack<int> s;
    s.push( 0 ); // some vertex
    g.setVisited( 0 );
    while ( ! s.empty() ) {
        int v = s.pop();
        if( g.getKey(v) == key )
            return v;
        for (int k=0; k < g.numberOfNeighbours(v); k++) {
            int nb = g.neighbour(v,k);
            if ( !g.isVisited( nb ) ) {
                s.push( nb );
                g.setVisited( nb );
            }
        }
    }
    return -1;
}
```

Conectividade (Grafos não direcionados)

- Começar em um vértice "qualquer" e marcar todos os vértices que alcançarmos. A próxima componente conexa será um vértice não marcado
- Exemplo:

```
int numberOfComponents( Graph &g ) {
    int count = 0;
    for( int v = 0; v < g.numberOfVertices(). v++ ) {
        if(!g.isVisited( v )) {
            count++;
            breadthFirst(g,v);
        }
    }
}
```

Árvore Geradora

- Arestas visitadas pela primeira vez em um percurso
- Exemplo:

```
void depthFirst( Graph &g, int vertex )
{
    Stack<int> s;
    s.push( vertex );
    g.setVisited( vertex );

    while ( ! s.empty() ) {
        int v = s.pop();

        for (int k=0; k < g.numberOfNeighbours(v); k++) {
            int nb = g.neighbour(v,k);
            if ( !g.isVisited( nb ) ) {
                g.setEdge( v, nb );
                s.push( nb );
                g.setVisited( nb );
            }
        }
    }
}
```

Caminho entre dois vértices

- Menor caminho (considerando número de arestas): busca em largura a partir de um dos vértice até o outro
- Exemplo:

```
void breadthFirst(Graph &g, int v1, int v2)
{
    Queue<int> q;
    q.put(v1);
    g.setVisited(v1);

    while (!q.empty()) {
        int v = q.get();
        if( v == v2 )
            return;
        for (int k=0; k < g.numberOfNeighbours(v); k++) {
            int nb = g.neighbour(v,k);
            if ( !g.isVisited( nb ) ) {
                q.put( nb );
                g.setVisited( nb );
                g.prox(nb, v); // caminho
            }
        }
    }
}
```

Detecção de ciclos

- Quando encontramos um vértice que já foi visitado em um percurso
- Exemplo:

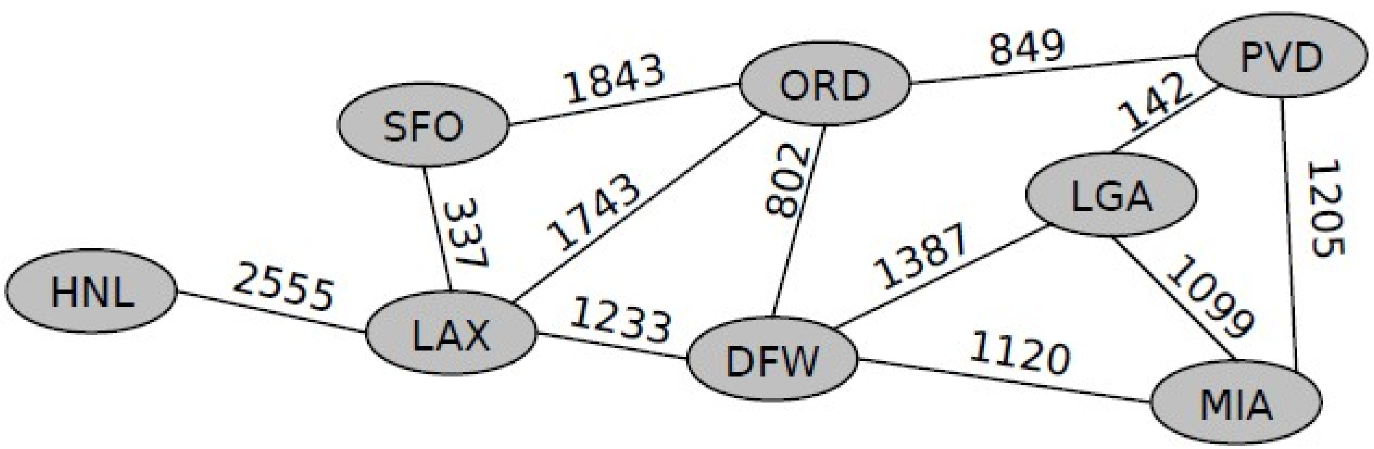
```
Stack<int> depthFirst( Graph &g, int vertex )
{
    Stack<int> s;
    s.push( vertex );
    g.setVisited( vertex );
    while ( ! s.empty() ) {
        int v = s.pop();
        for (int k=0; k < g.numberOfNeighbours(v); k++) {
            int nb = g.neighbour(v,k);
            if ( !g.isVisited( nb ) ) {
                s.push( nb );
                g.setVisited( nb );
            } else {
                Stack<int> ciclo;
                int vtemp;
                do {
                    vtemp = s.pop();
                    ciclo.push(vtemp);
                } while( vtemp == nb );
                return ciclo;
            }
        }
    }
}
```

Próximos tópicos

- Grafos ponderados:
- Caminho mínimo
- Arvore Geradora mínima

Grafos Ponderados

- Cada aresta possui um valor numérico associado a ela.
- As arestas podem representar distâncias, custos, etc ...
- ***G=(V,E, w), onde w : E → R***



Próximos tópicos

- Grafos ponderados:
- Caminho mínimo
- Arvore Geradora mínima

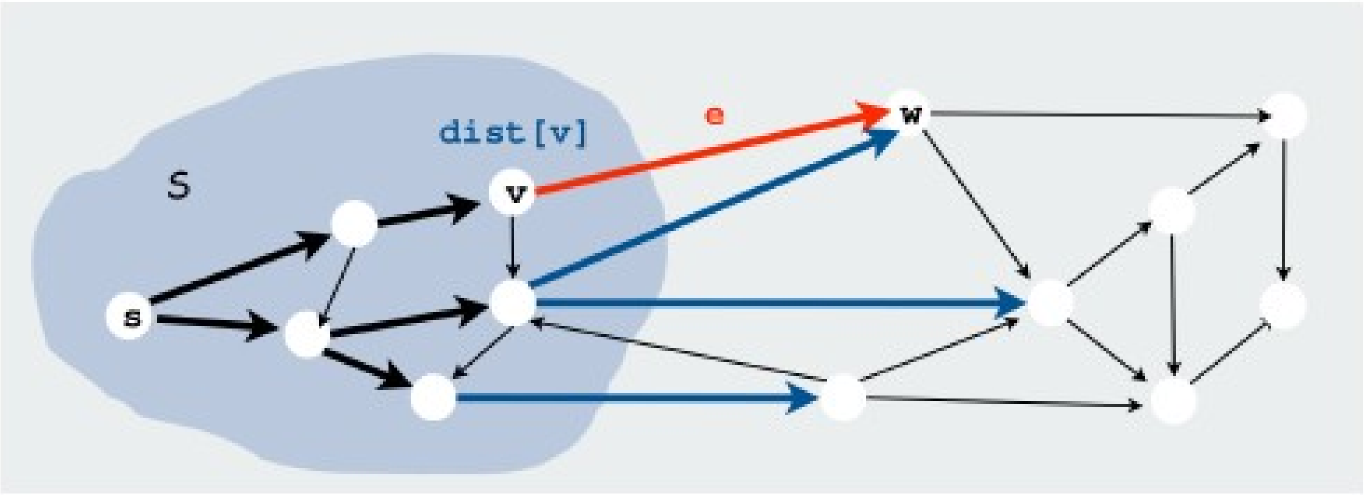
Próximos tópicos

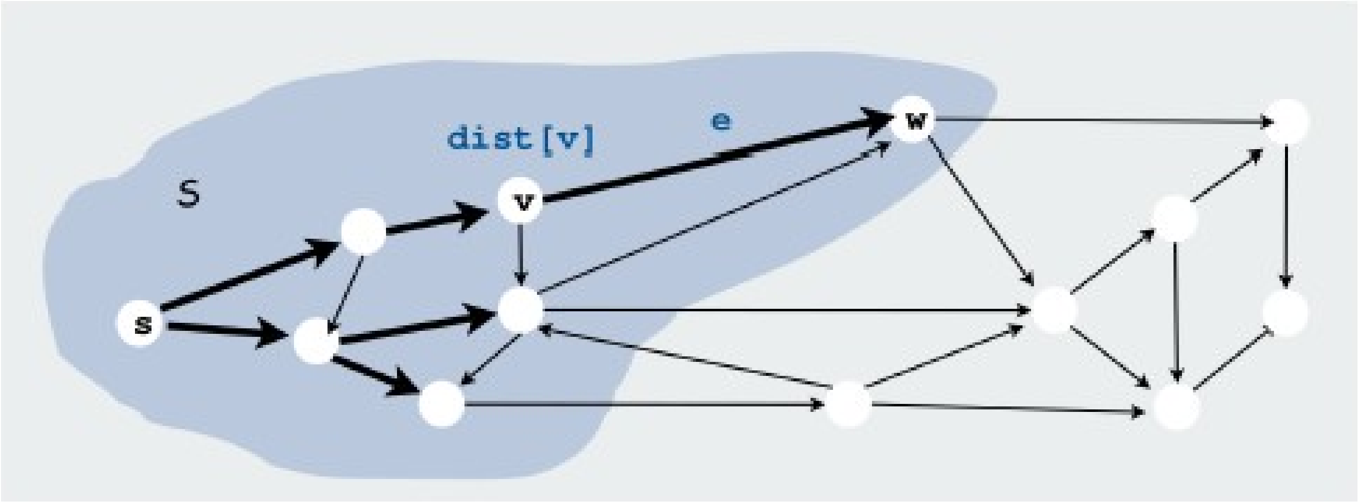
- Menor caminho
  - Dado um grafo ponderado, determinar o caminho entre dois vértices que possui a menor soma dos pesos
    - Sub-caminhos também são mínimos
    - Existe uma árvore de caminhos mínimos a partir de um vértice
- Árvore geradora mínima
  - Dado um grafo ponderado, determinar a árvore geradora que possui a menor soma dos pesos

Dijkstra

- Lembra o busca em largura, mas é orientado ao menor caminho até o momento (com a ajuda de uma *PriorityQueue)*
- Calcula a distância mínima de todos os vértice a partir de uma origem
- Assume:
  - Grafo conexo
  - Não-direcionado
  - Pesos positivos

Dijkstra

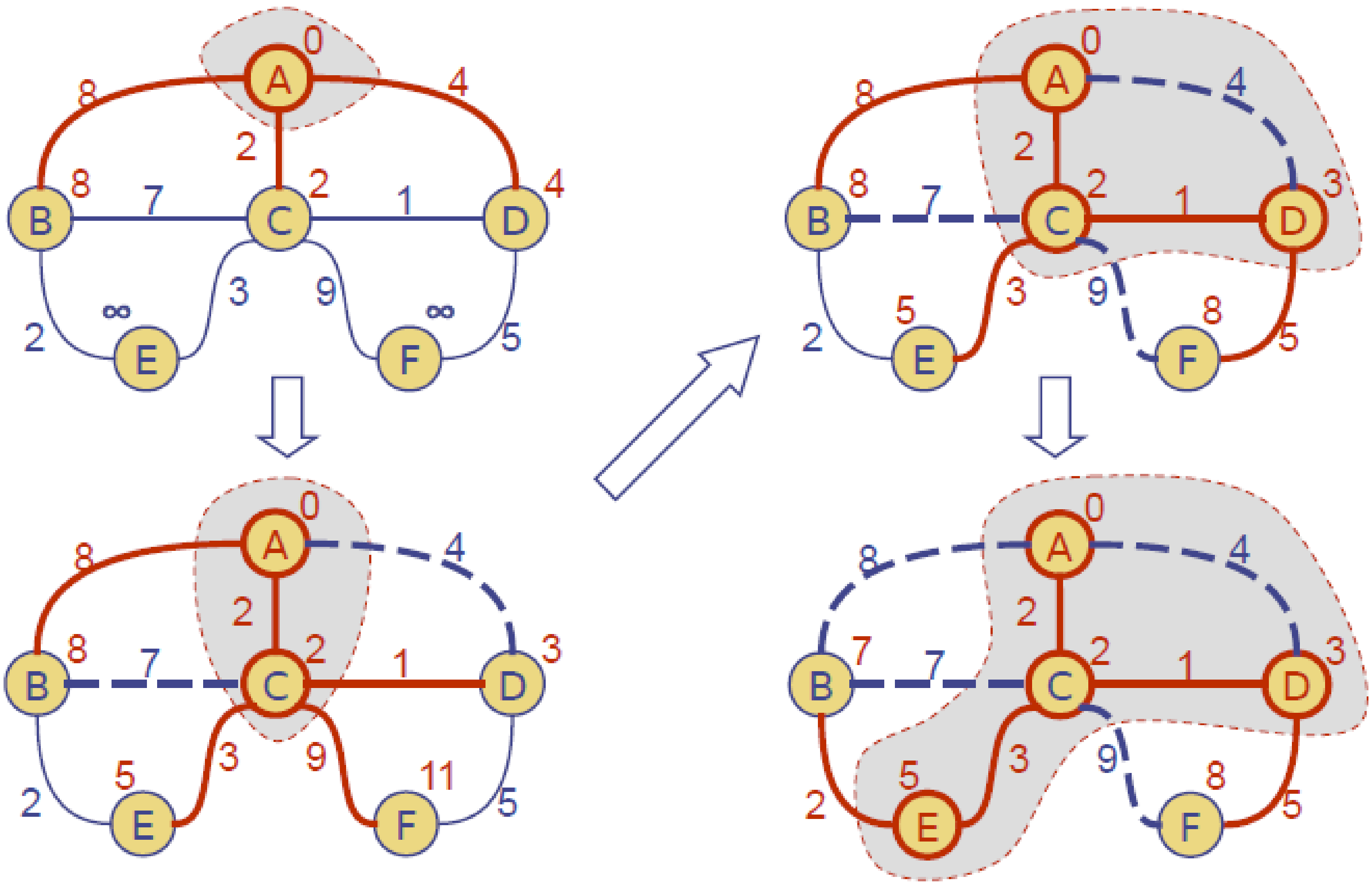




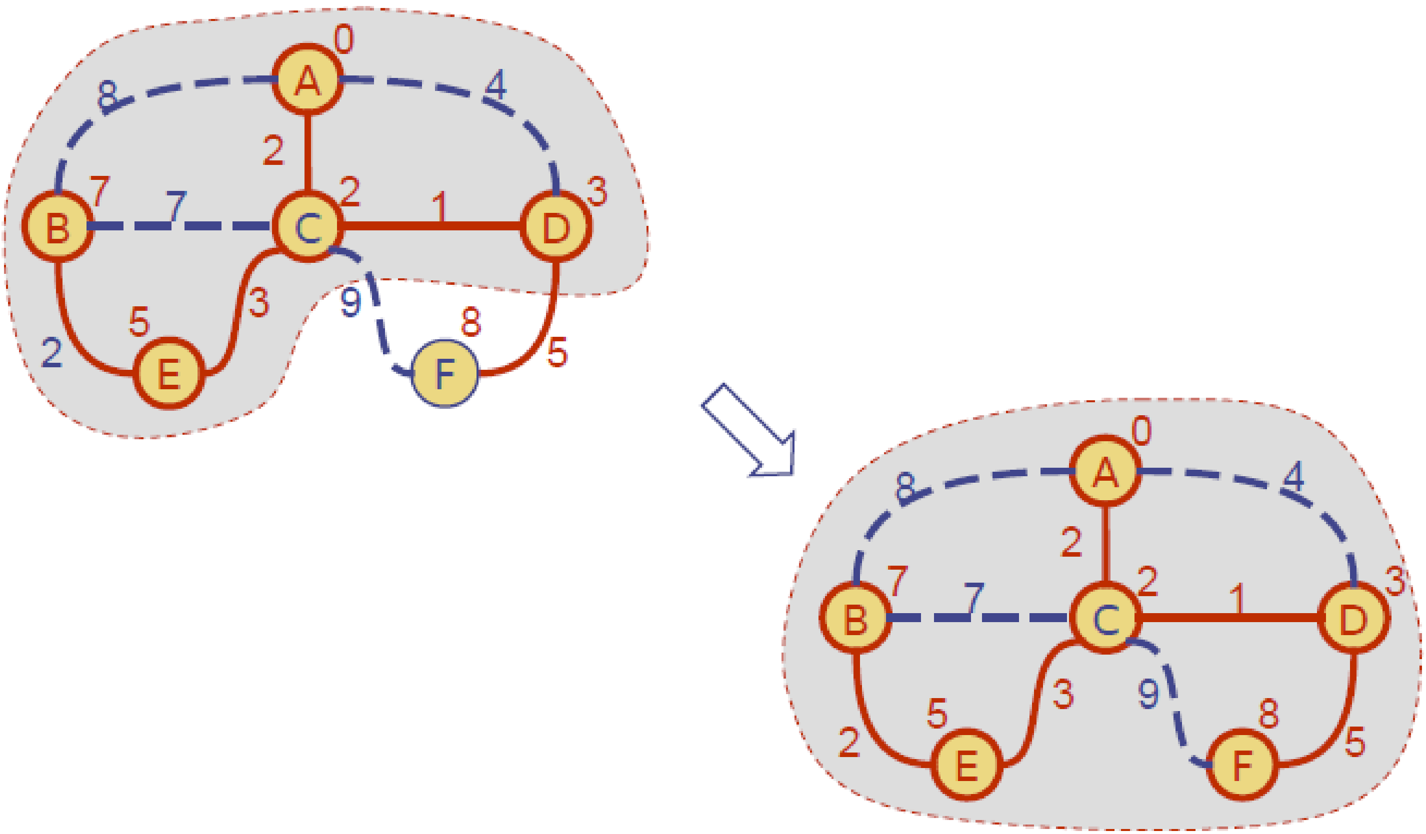
Dijkstra

```
Algorithm DijkstraDistances(G, s)
    Q ← new heap-based priority queue
    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
        l ← Q.insert(getDistance(v), v)
        setLocator(v,l)
    while ¬Q.isEmpty()
        u ← Q.removeMin()
        for all e ∈ G.incidentEdges(u)
            z ← G.opposite(u,e)
            r ← getDistance(u) + weight(e)
            if r < getDistance(z)
                setDistance(z,r)
                Q.replaceKey(getLocator(z),r)
```

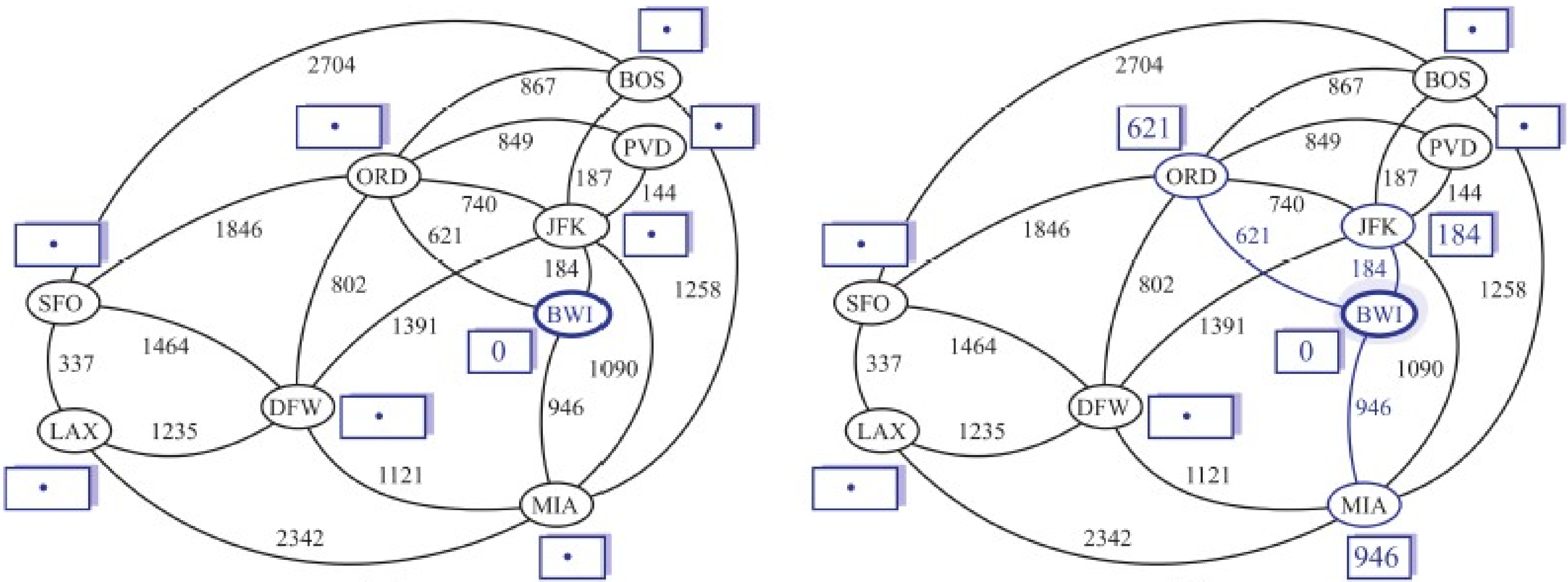
Dijkstra - Exemplo



Dijkstra – Exemplo (Cont.)

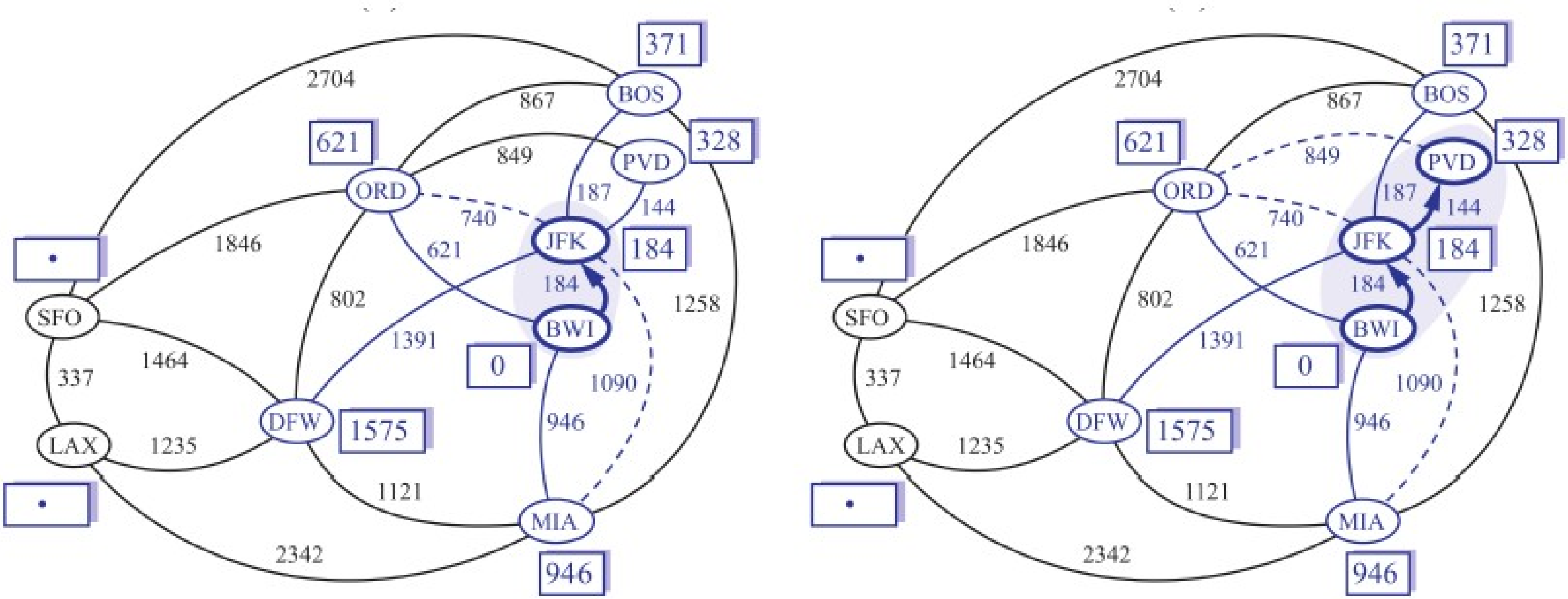


Dijkstra - Exemplo



(a) (b)

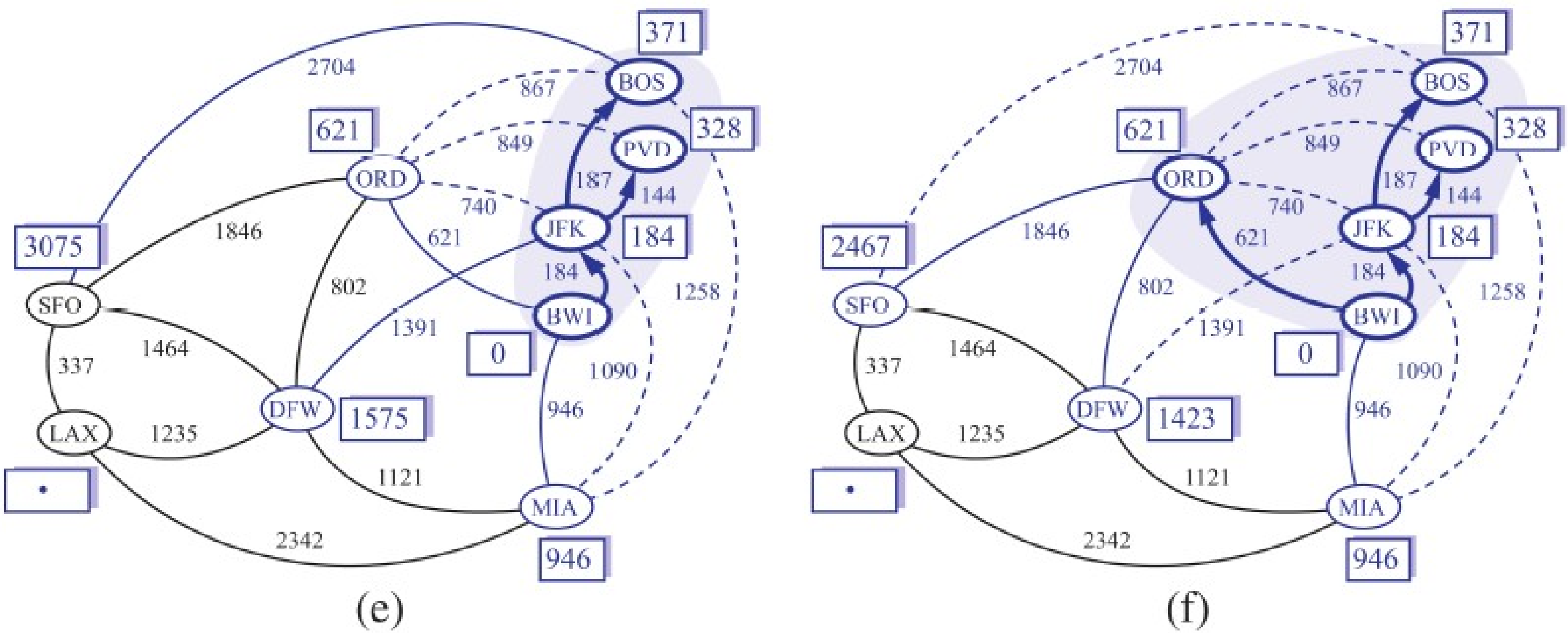
Dijkstra – Exemplo (Cont.)



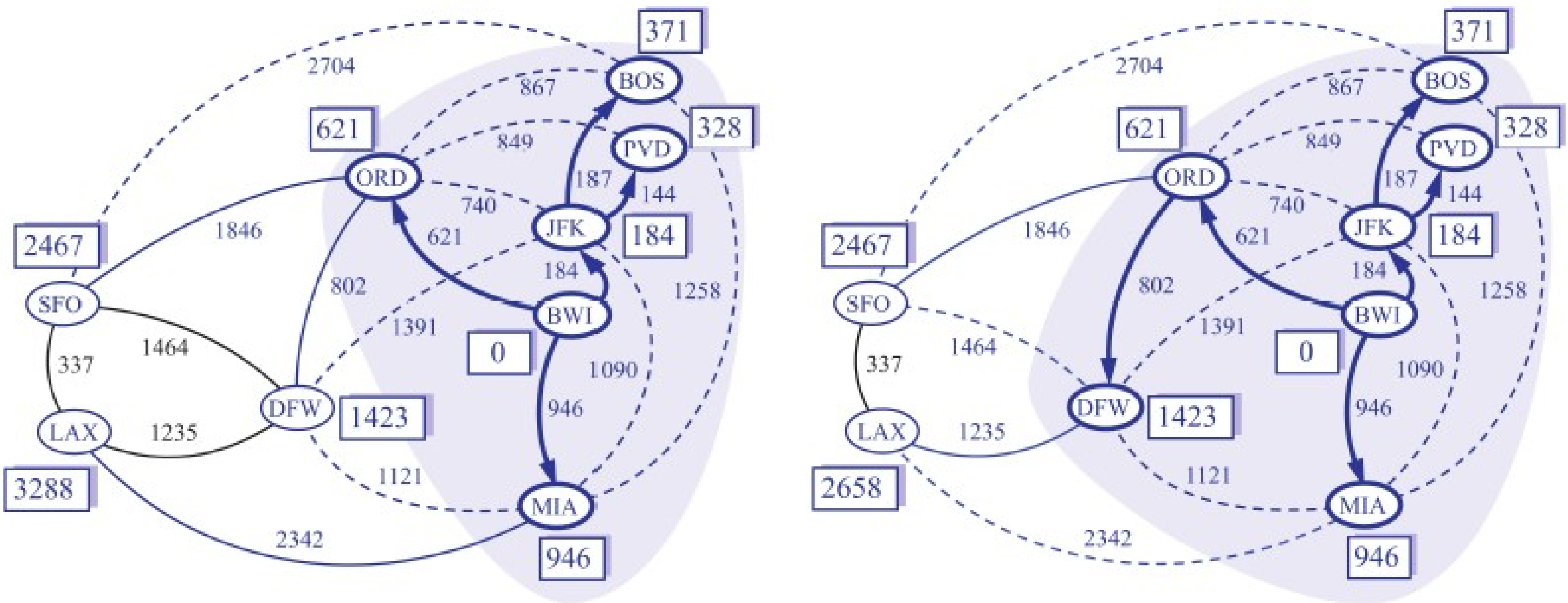
(c)

(d)

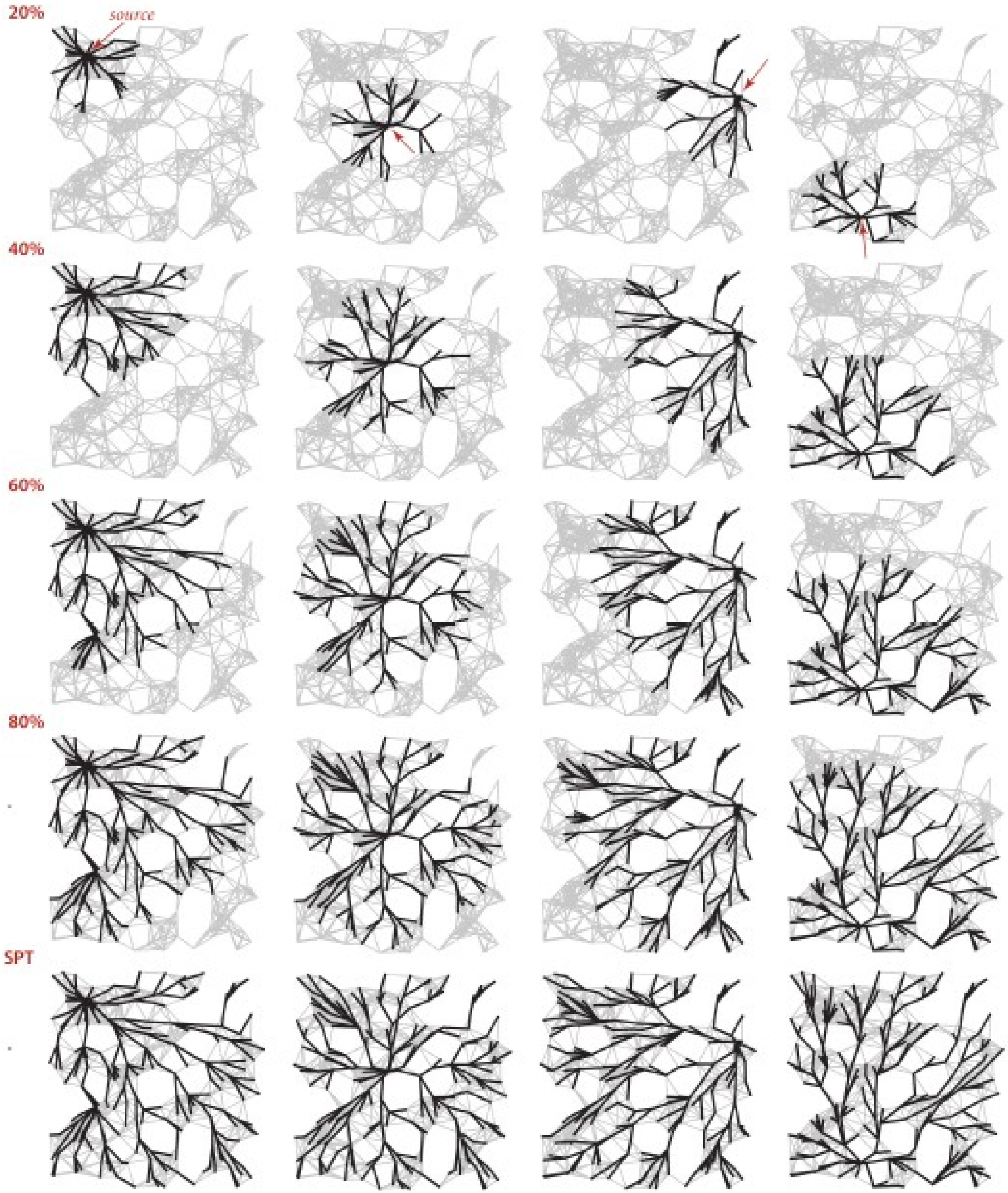
Dijkstra – Exemplo (Cont.)



Dijkstra – Exemplo (Cont.)



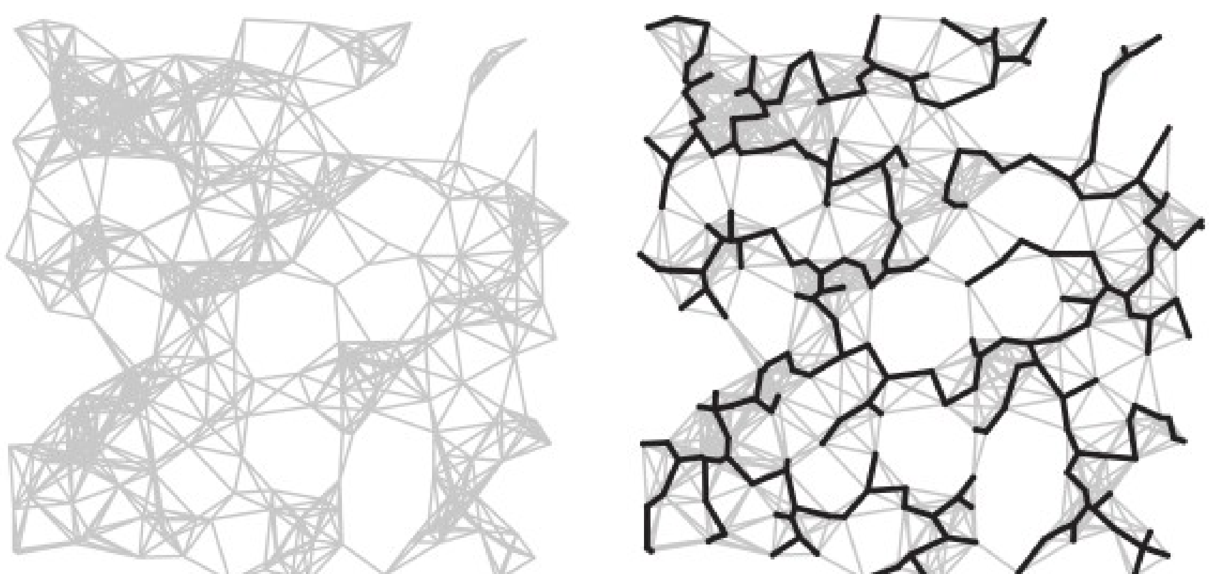
(g) (h)



Ivan José dos Reis Filho

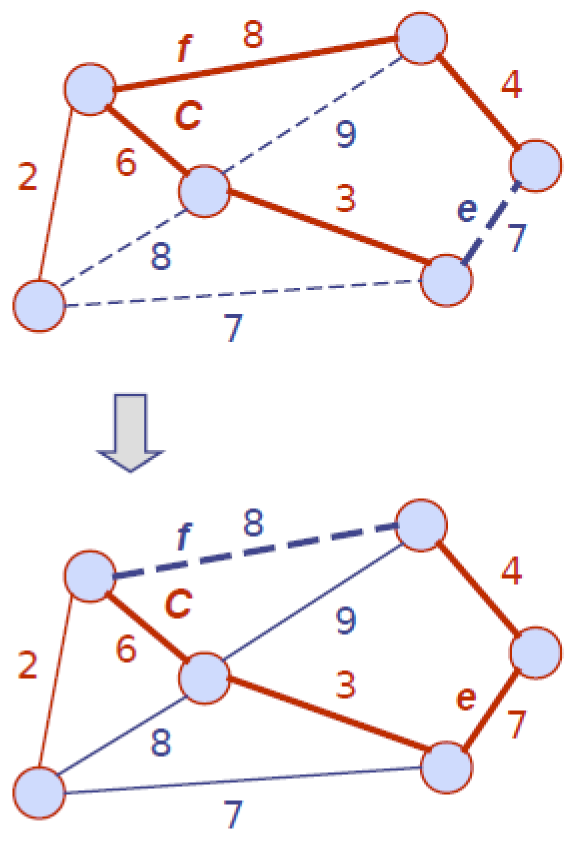
Minimum spanning tree
(Árvore Geradora Mínima)

- Subgrafo de G, contendo todos os vértices, <u>conectado</u>, <u>sem ciclos</u> e a soma de todas as arestas é o <u>menor</u> de todas as árvores geradores possíveis



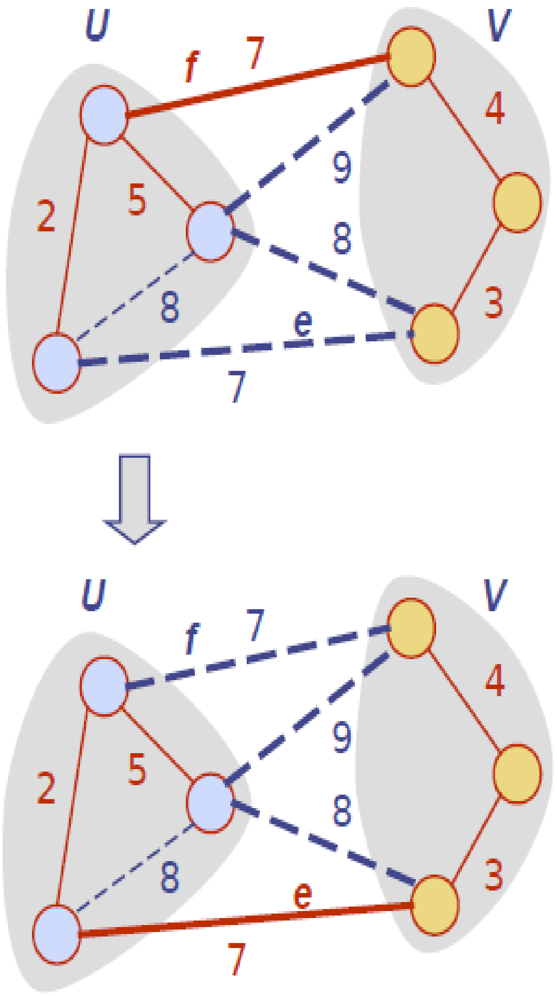
Minimum spanning tree

- Propriedades:
  - Considerando:
    - Grafo ponderado ***G=(V,E,w)***
    - ***T é uma árvore geradora mínima*** de ***G***
  - Ciclo: Seja ***e uma aresta de G*** que não está em ***T e C o ciclo*** formado por ***e em T :***
    - Todas as arestas de ***C possuem*** pesos menores (ou iguais) a ***e***

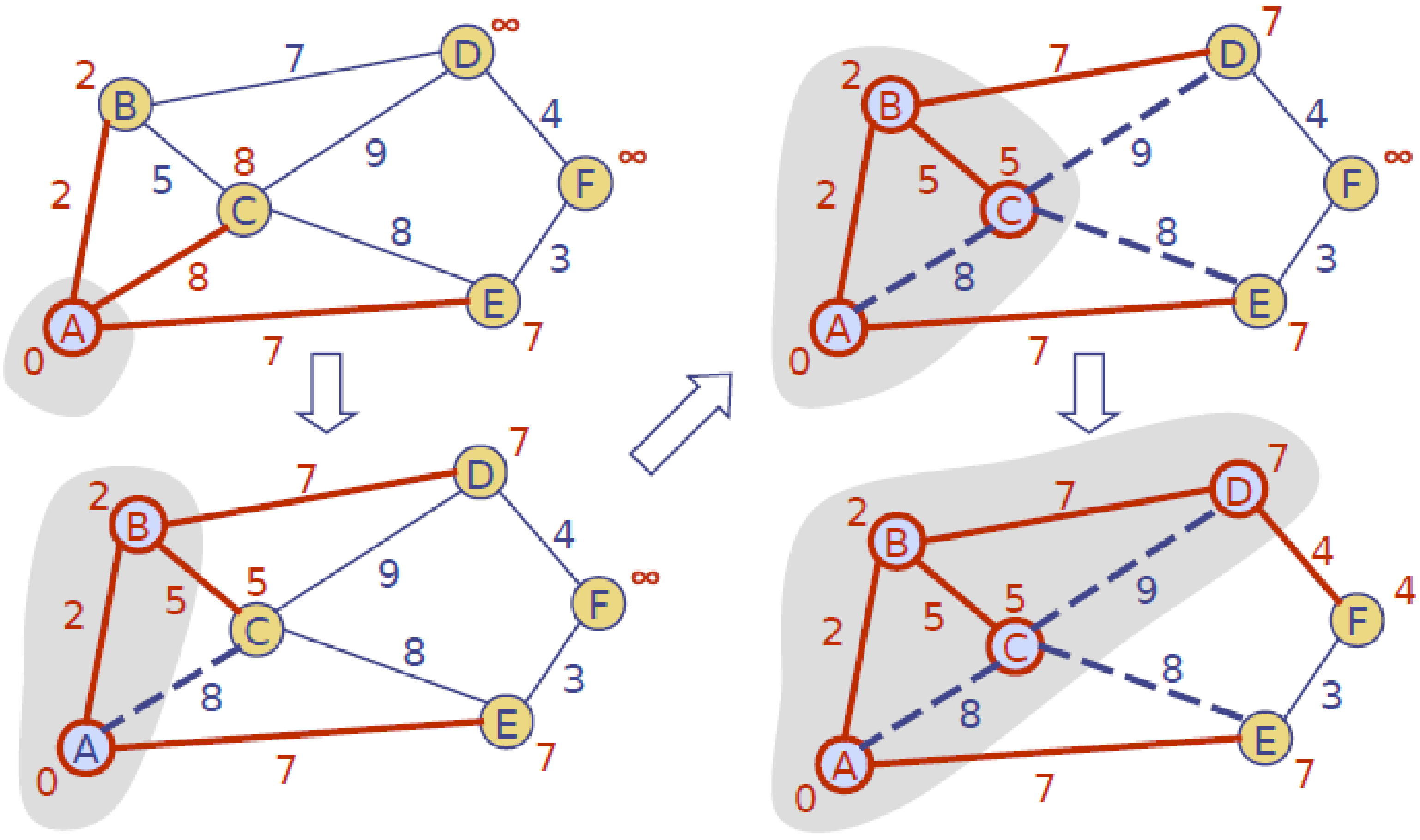


Minimum spanning tree

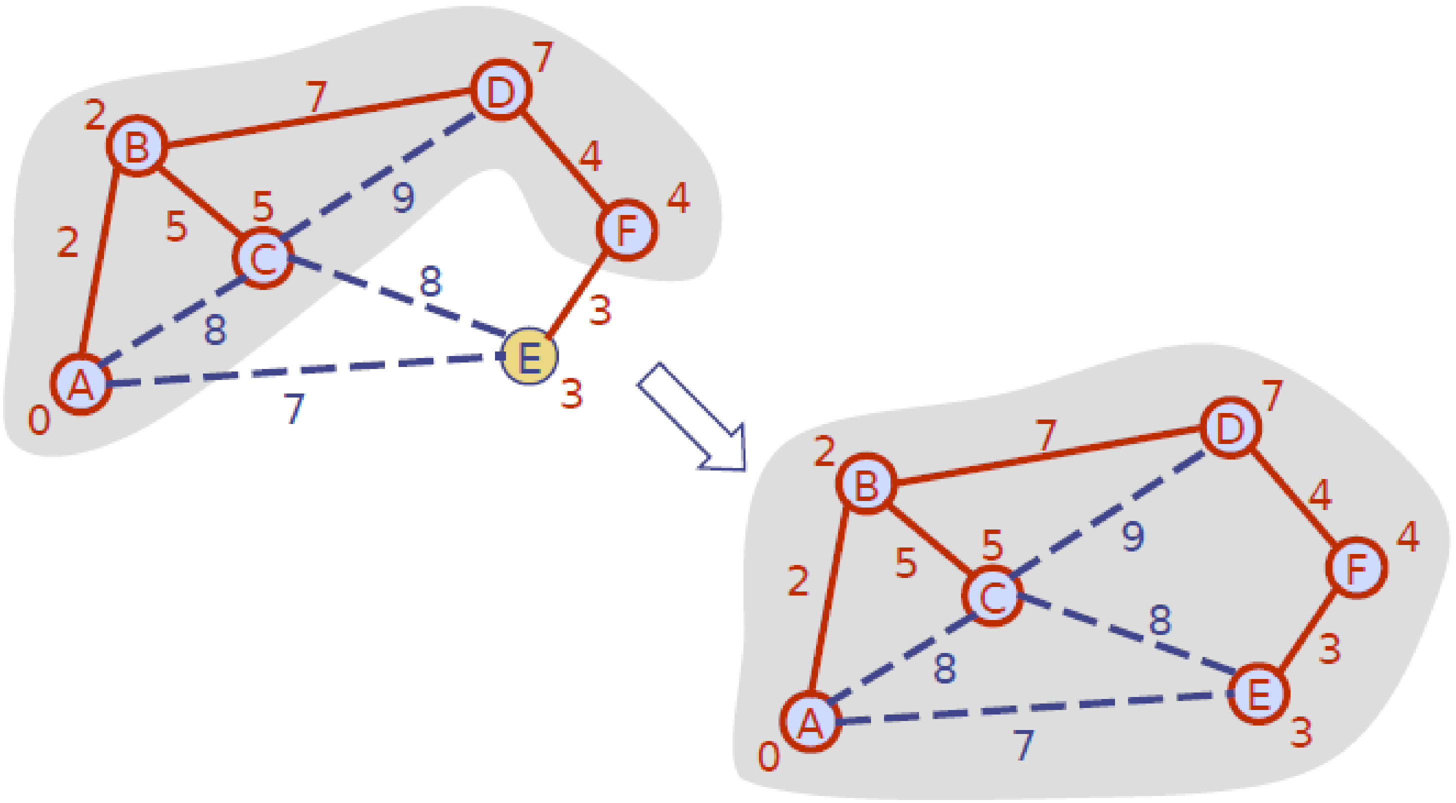
- Propriedades:
  - Considerando:
    - Grafo ponderado ***G=(V,E,w)***
    - ***T é uma árvore geradora*** mínima de ***G***
    - Uma partição dos vértices de ***G*** em dois subconjuntos ***U e V***
  - Partição: Seja ***e aresta de T*** que está entre os conjuntos ***U*** e ***V :***
    - Todas as arestas de ***G que estão*** entre os subconjuntos tem peso menores (ou iguais) a ***e***



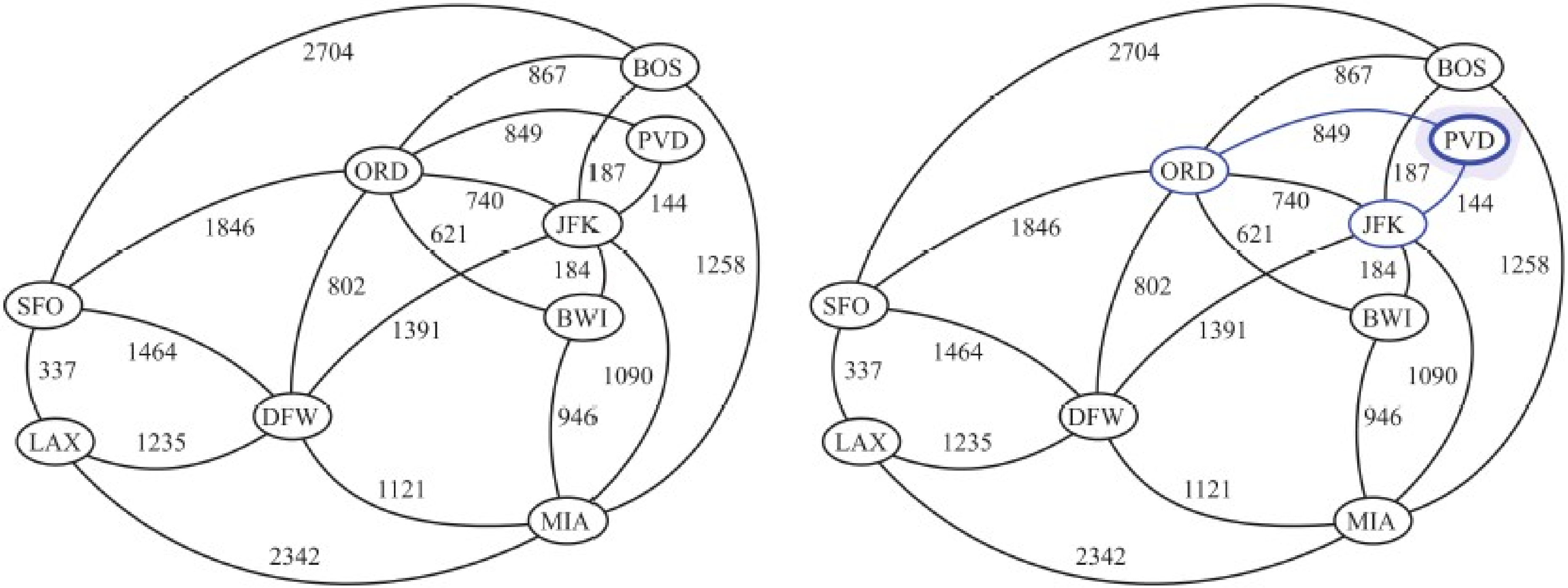
Prim-Jarnik (Exemplo)



Prim-Jarnik – Exemplo (Cont.)

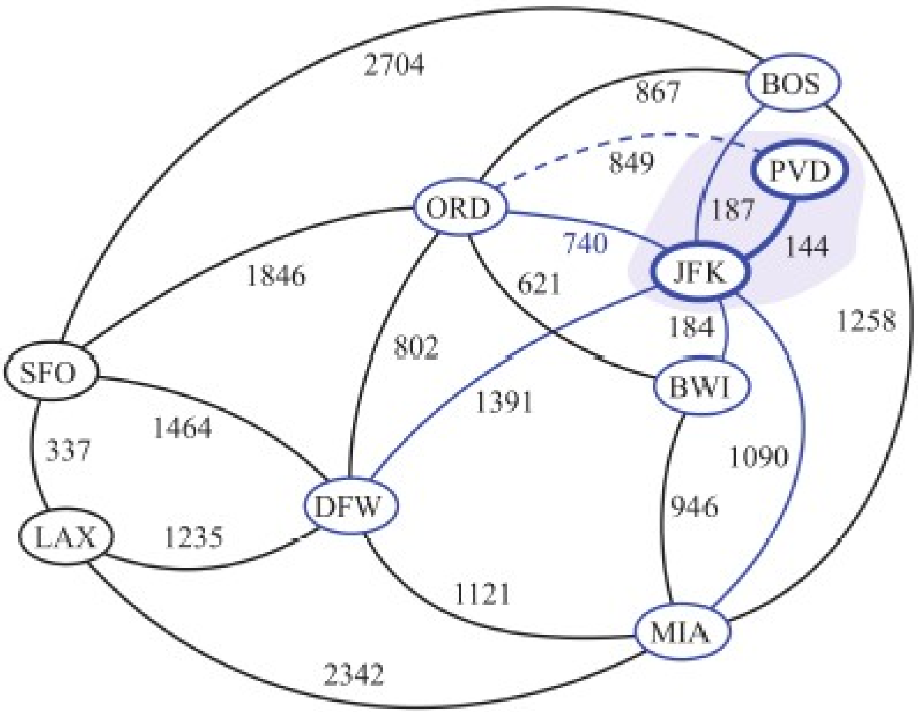


Prim-Jarnik (Exemplo)

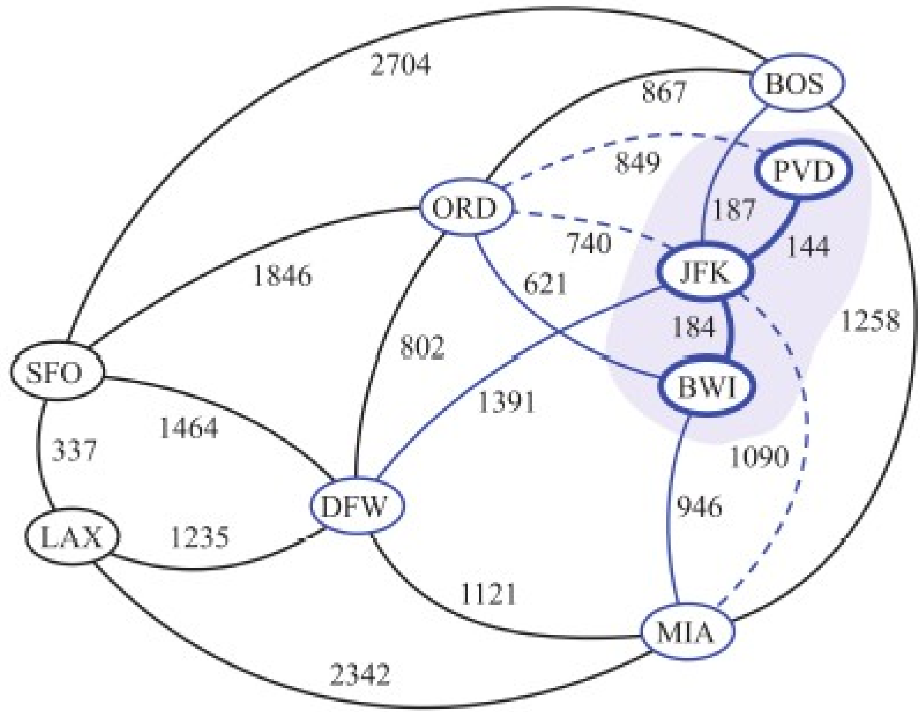


(a) (b)

Prim-Jarnik – Exemplo (Cont.)

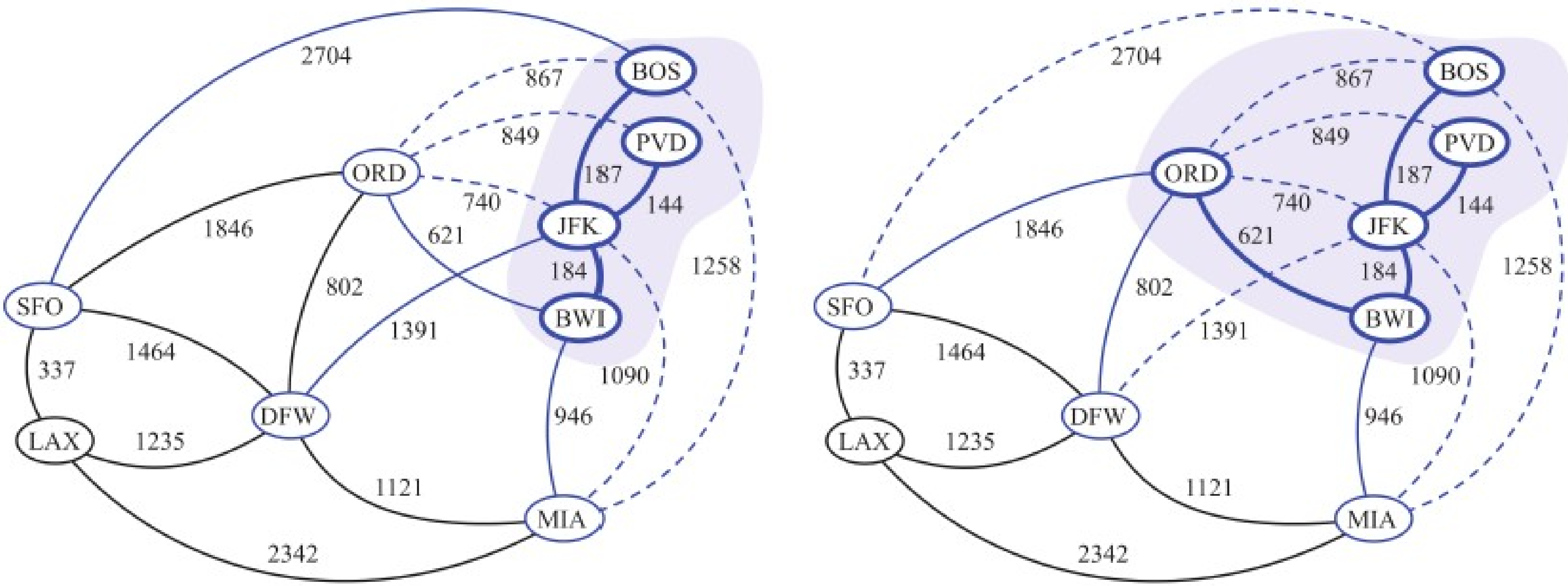


(c)



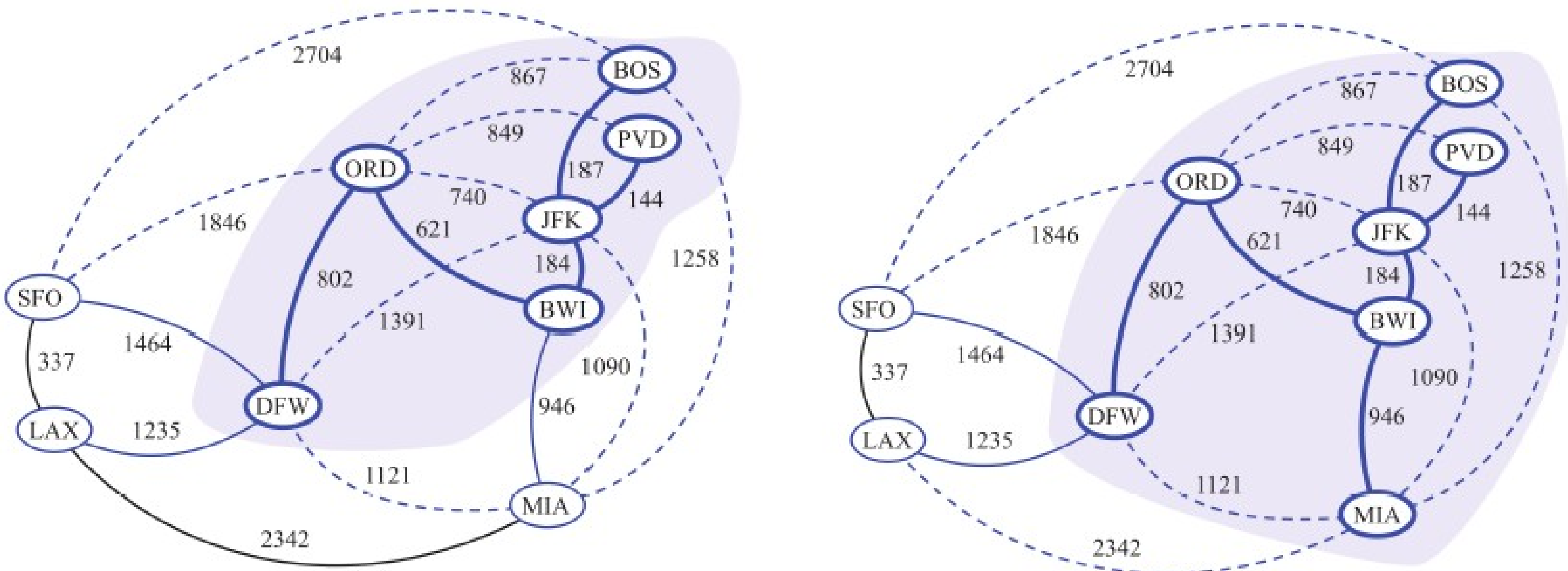
(d)

Prim-Jarnik – Exemplo (Cont.)



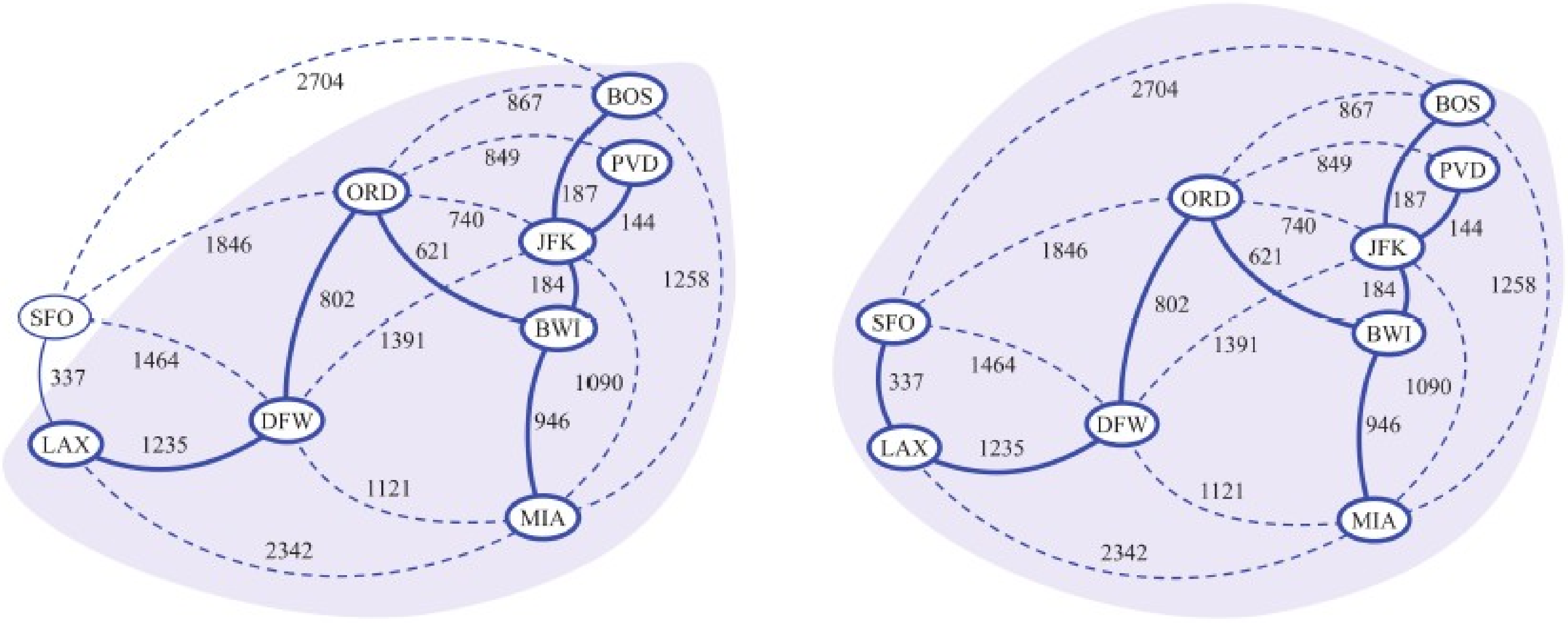
(e) (f)

Prim-Jarnik – Exemplo (Cont.)



(g)
(h)

Prim-Jarnik – Exemplo (Cont.)

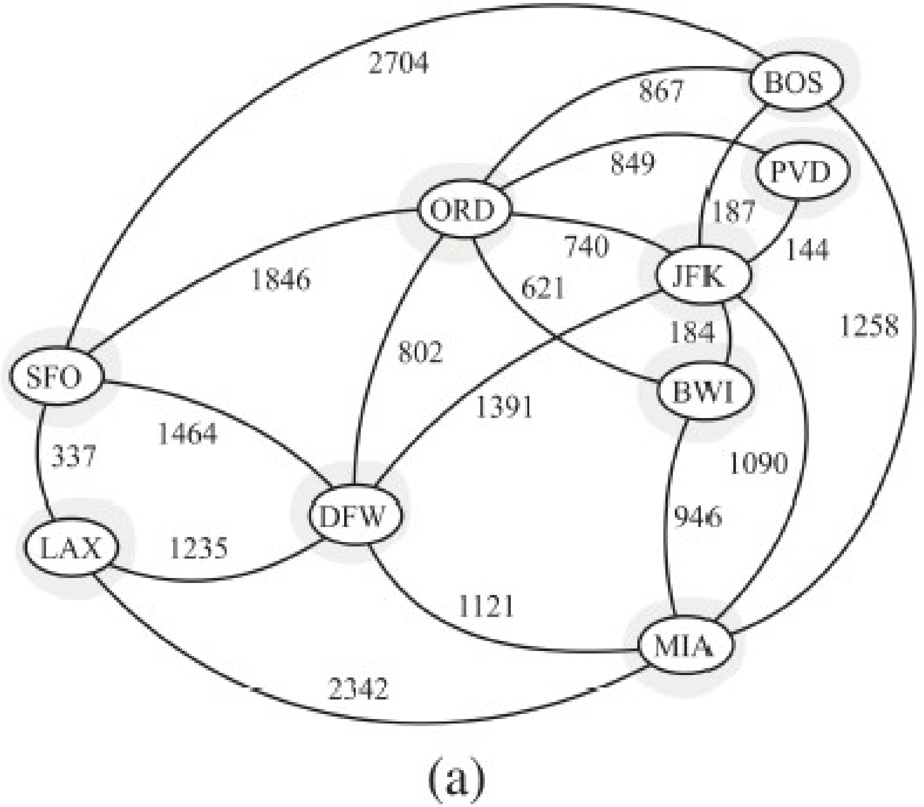


(i) (j)

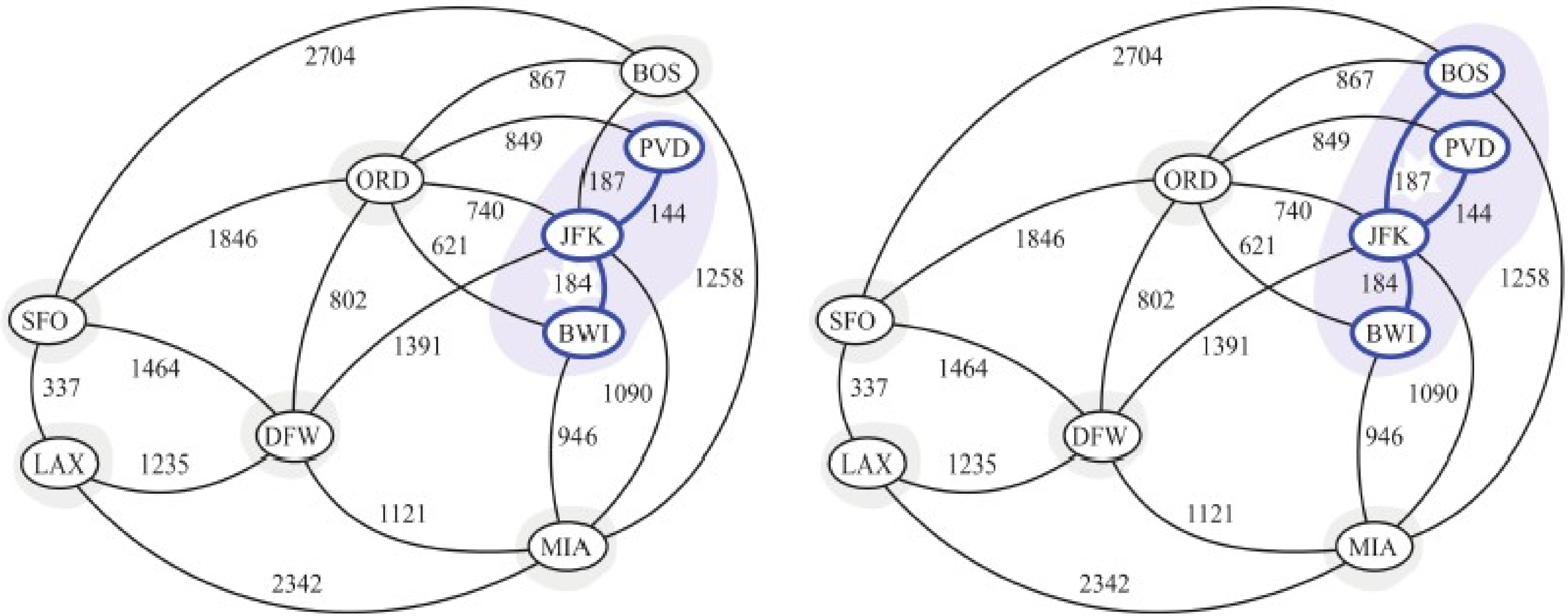
Prim-Jarnik

```
Algorithm PrimJarnikMST(G)
  Q ← new heap-based priority queue
  s ← a vertex of G
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
    setParent(v, ∅)
    l ← Q.insert(getDistance(v), v)
    setLocator(v,l)
  while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u) and e ∈ Q
      z ← G.opposite(u,e)
      r ← weight(e)
      if r < getDistance(z)
        setDistance(z,r)
        setParent(z,e)
        Q.replaceKey(getLocator(z),r)
```

Kruskal - Exemplo

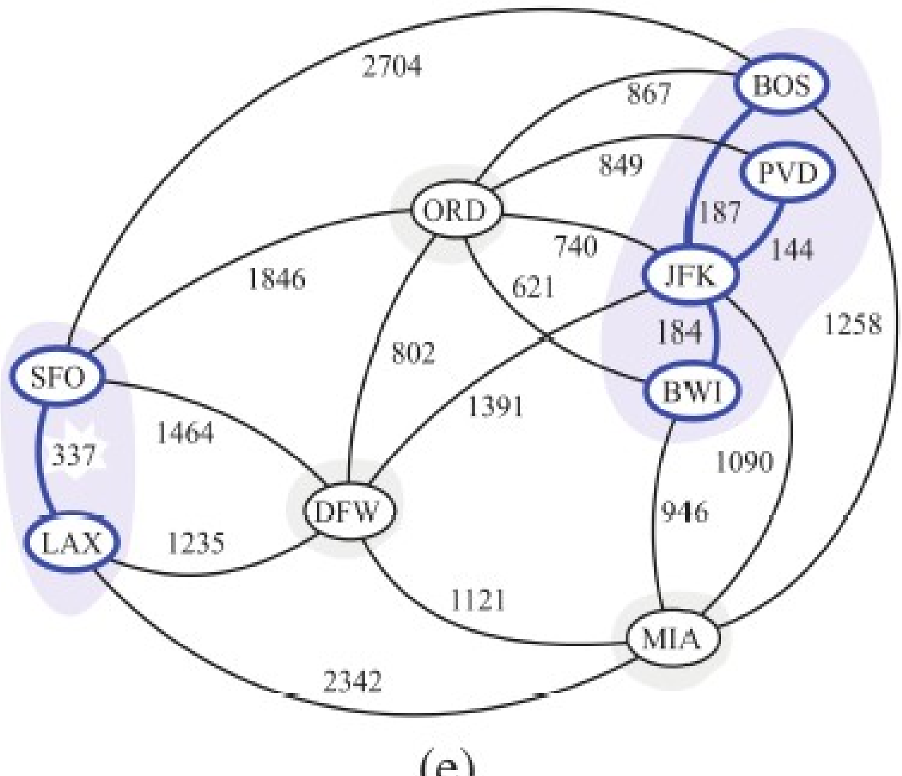


Kruskal – Exemplo (Cont.)

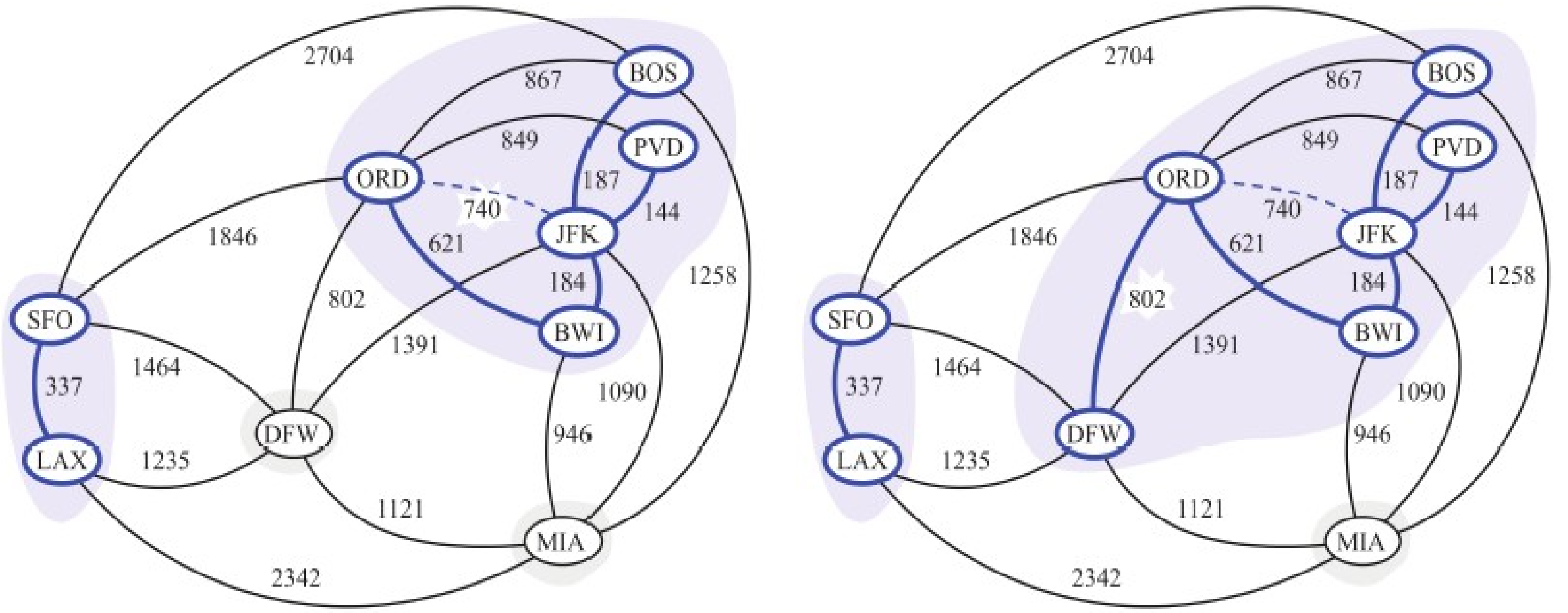


(c) (d)

Kruskal – Exemplo (Cont.)

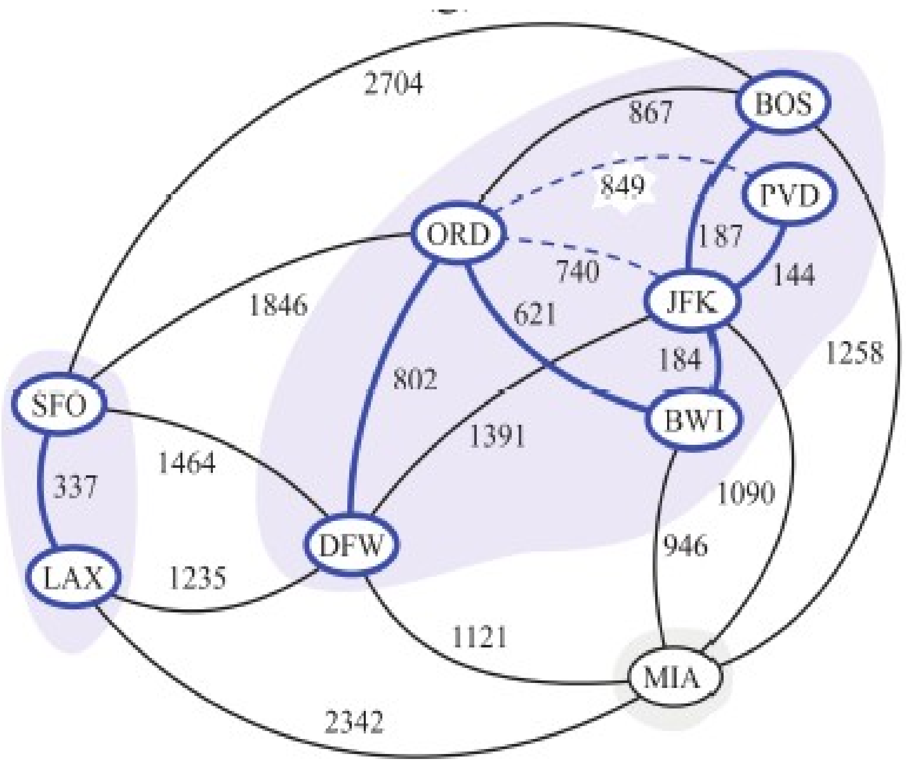


Kruskal – Exemplo (Cont.)

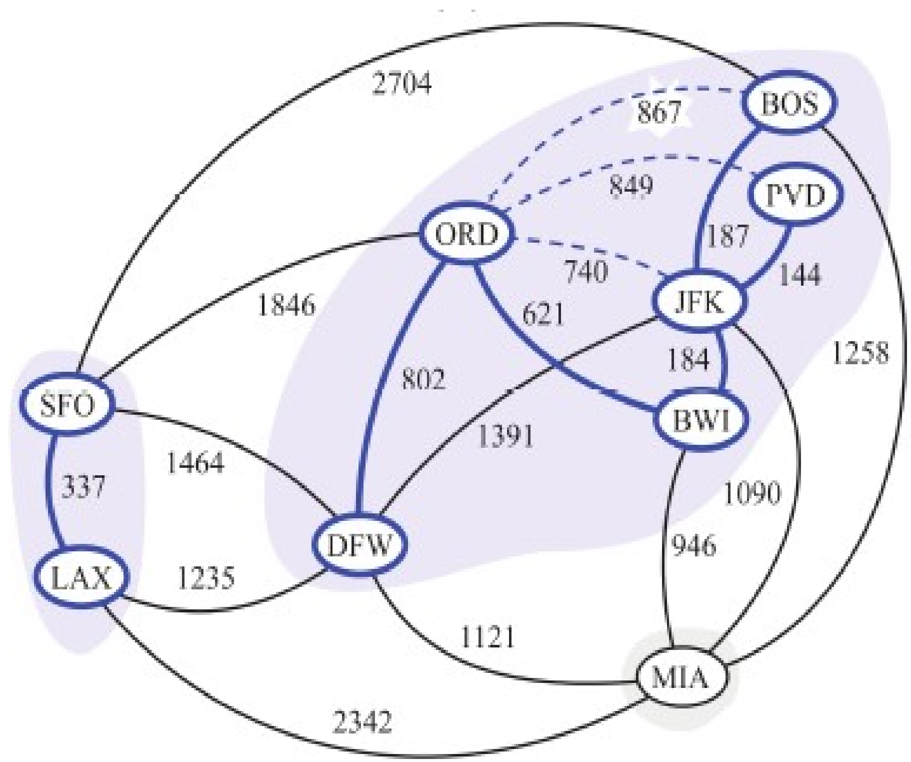


(g) (h)

Kruskal – Exemplo (Cont.)

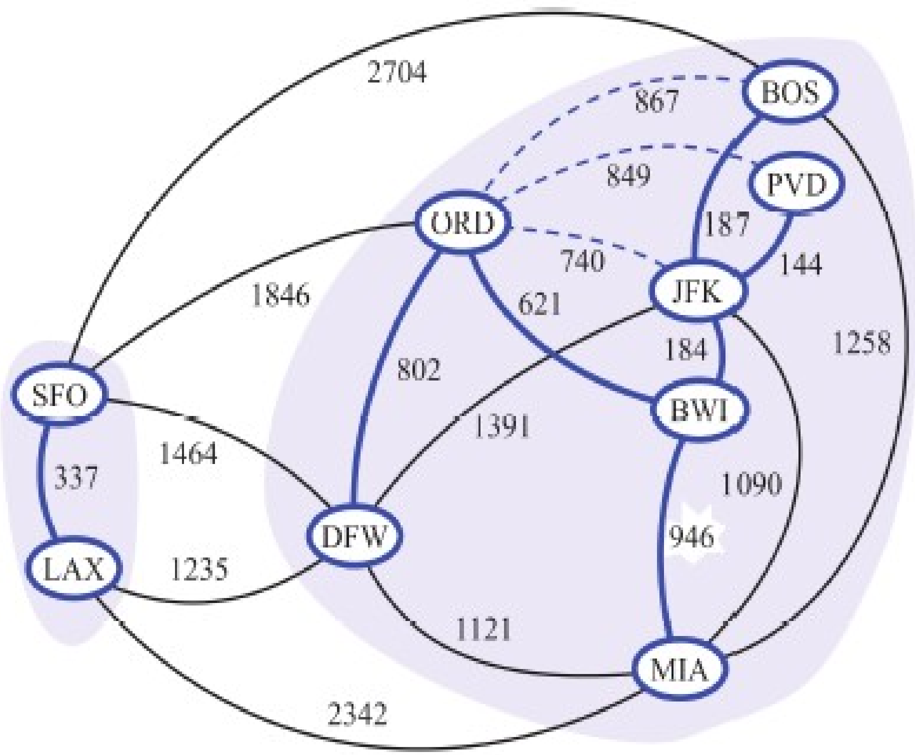


(i)

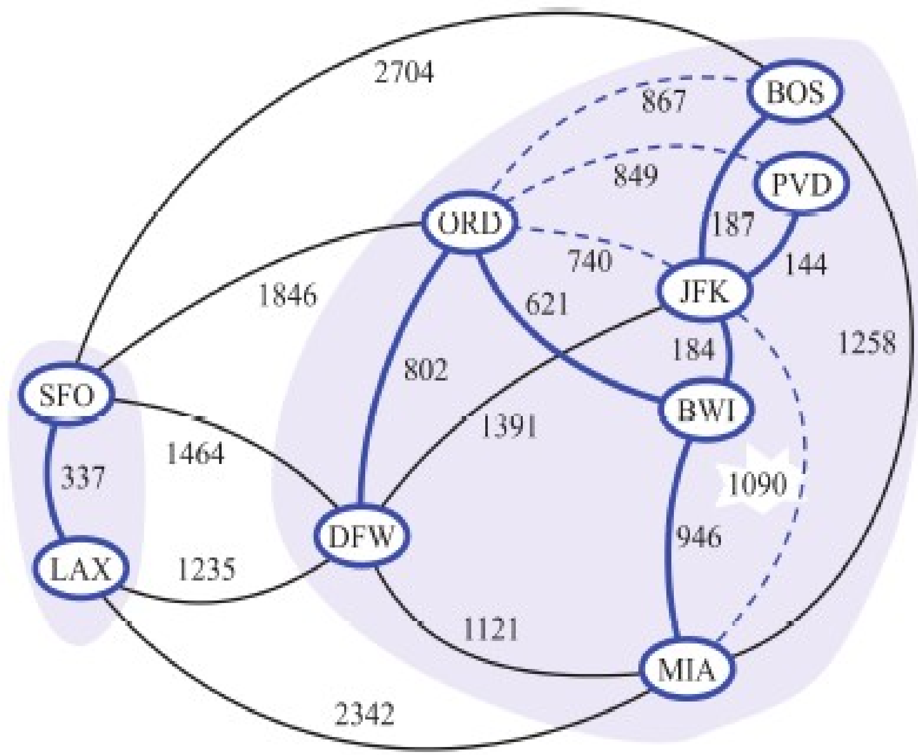


(j)

Kruskal – Exemplo (Cont.)



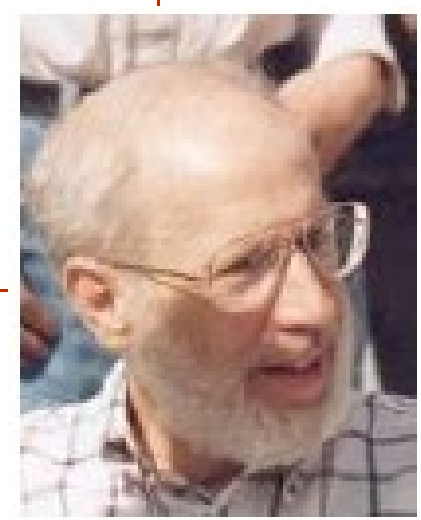
(k)

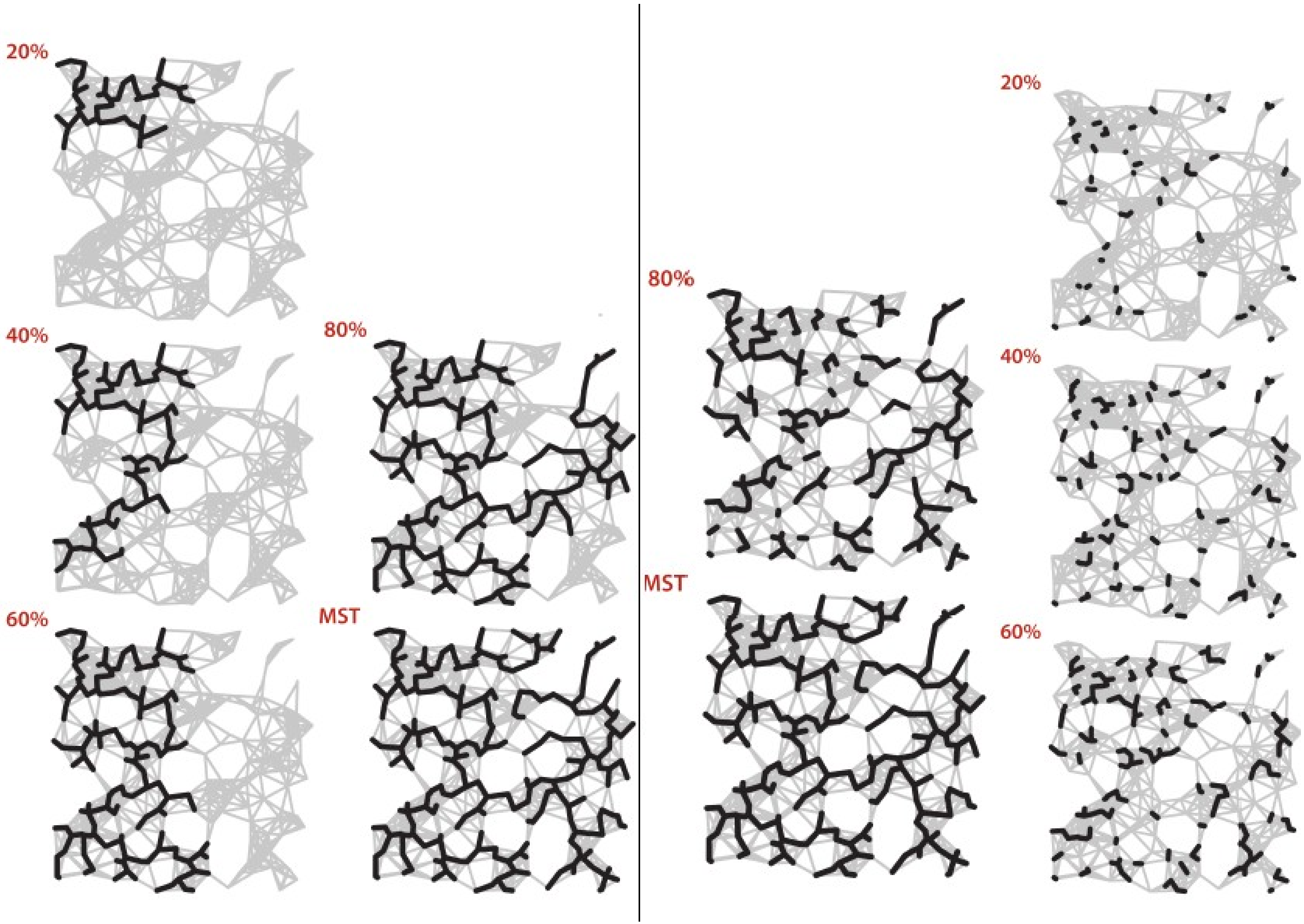


(l)

Kruskal

```
Algorithm KruskalMST(G)
   for each vertex V in G do
      define a Cloud(v) of ← {v}
   let Q be a priority queue.
   Insert all edges into Q using their  weights as the key
   T ← Ø
   while T has fewer than n-1 edges do
      edge e = T.removeMin()
      Let u, v be the endpoints of e
      if Cloud(v) ≠  Cloud(u) then
         Add edge e to T
         Merge Cloud(v) and Cloud(u)
   return T
```




20%
40%
80%
60%
MST
80%
MST
20%
40%
60%

DFS.	Take edge from vertex which was discovered most recently.
BFS.	Take from vertex which was discovered least recently.
Prim.	Take edge of minimum weight.
Dijkstra.	Take edge to vertex that is closest to s.