Universidade Nova de Lisboa
OMNIS CIVITAS CONTRA SE DIVISA NON STABIT
Faculdade de Ciências e Tecnologia

Graph Algorithms: Dynamic Programming

Pedro Barahona
DI/FCT/UNL

Métodos Computacionais
1st Semestre 2017/2018

Dynamic Programming: Algorithms for Graphs

- Most graph properties address optimisation goals, namely
  a. Shortest paths
  b. Minimum Spanning Trees
  c. Minimum Hamiltonian tours (Traveling Salesman)
  d. Minimum number of colours
- Some of these properties (e.g. **a** and **b**, but not **c** nor **d**), can be computed by polynomial algorithms.
- In most cases, algorithms to compute the optima may follow a methodology, **dynamic programming**, based on Mathematical Induction on the Integers:
  - Once an optimal solution is obtained with **n** nodes, extend it to **n+1** nodes.
- We will see two examples of this, in the following algorithms
  - Minimum Spanning Tree – **Prim's Algorithm**
  - Shortest Paths – **Floyd-Warshall's Algorithm**

Minimum Spanning Tree: Prim's Algorithm

- A **spanning tree** is a subset of a connected graph that has the topology of a tree and covers all nodes of the graph.
- It has many applications, namely to provide services to a number of sites (the nodes) that can be interconnected in several ways (by a graph), but using the a minimal number of connections that allow all sites to be reached, i.e. a single path connecting any two nodes.
- Among these spanning trees one is usually interested in **minimum spanning trees** (MST) that minimise the sum of the costs of the arcs selected for the tree.
- There are many polynomial algorithms that may be used to compute these MSTs, the most common ones are the Kruskal's and the Prim's algorithms.
- Given the similarities between the latter and the algorithm to check connectedness of a graph, we will address now the **Prim's Algorithm**.

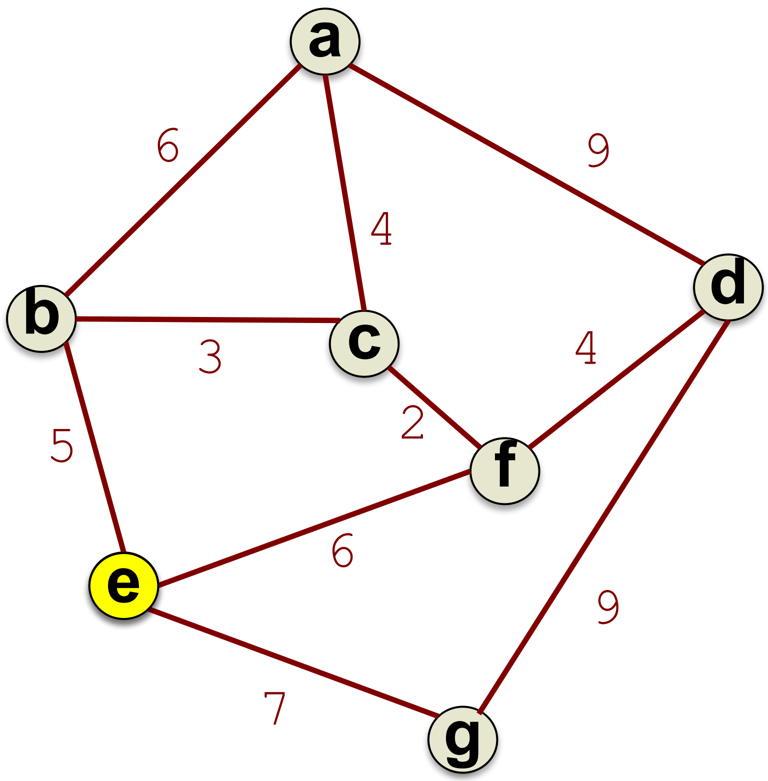
Minimum Spanning Tree: Prim's Algorithm

- The Prim´s algorithm is an example of **Dynamic Programming** that extends a MST with **n** nodes to **n+1** nodes, with an eager selection of the new node (i.e. once the node is selected, the selection **is not backtracked** for alternatives).

- The algorithm can be understood as a process of increasing the size of a current MST, starting with 1 node and ending with all the nodes, and specified as follows:
    - Maintain two sets of nodes: **In** and **Out**, where **In** is the set of nodes already included in a current **MST** and **Out** are those not yet included.

    1. Select arbitrarily a node from the tree to initialise the **In** set, and put the others in the **Out** set;
    2. While there are nodes in the **Out** set,
        i. Find which node from the **Out** set has an arc of least cost to one connecting it to one of the nodes of the **In** set;
        ii. Transfer the node from the **Out** set to the **In** set and include the least cost arc in the current **MST**.

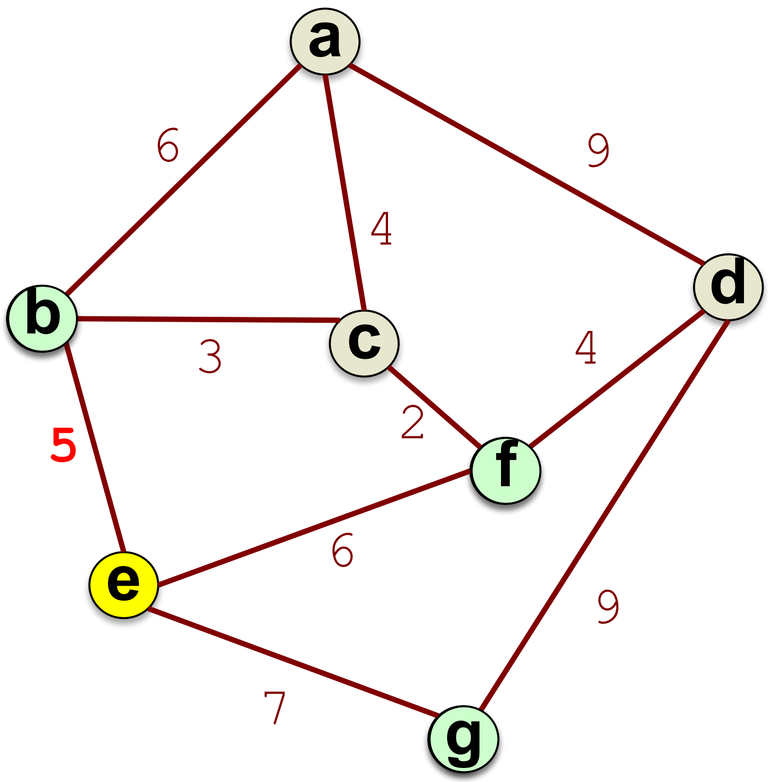
Minimum Spanning Tree: Prim's Algorithm



- Start with an arbitrary node in the **`In`** set
- Start with the **`Out`** set with all the other nodes
- Initialise the **`MST`** to empty

```
In = [e]
Out = [a,b,c,d,f,g]
MST = {}
```

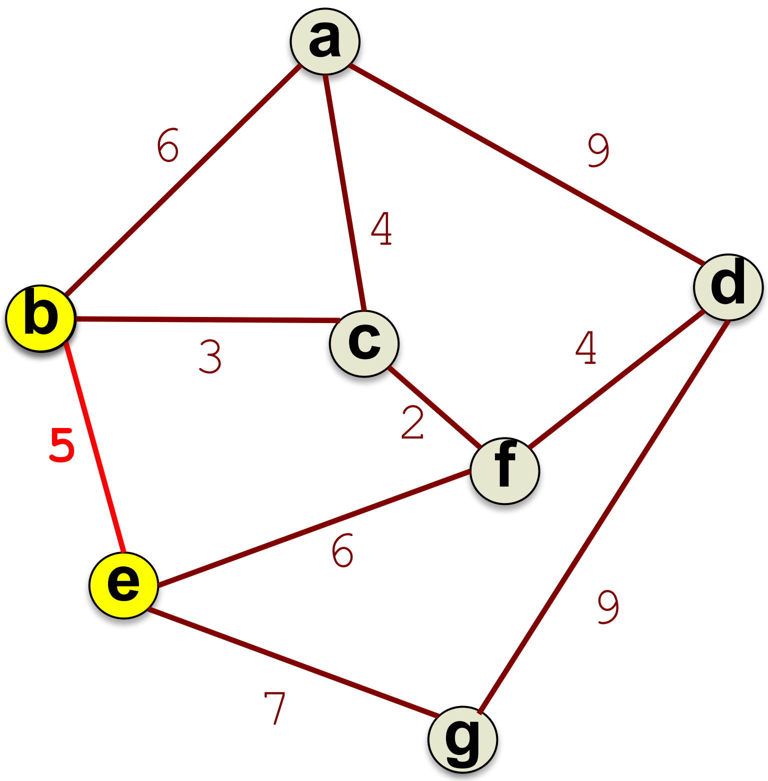
Minimum Spanning Tree: Prim's Algorithm



- Check all arcs between nodes in the **`In`** and **`Out`** sets
- Chose that with minimum cost

```
In = [e]
Out = [a,b,c,d,f,g]
MST = {}
```

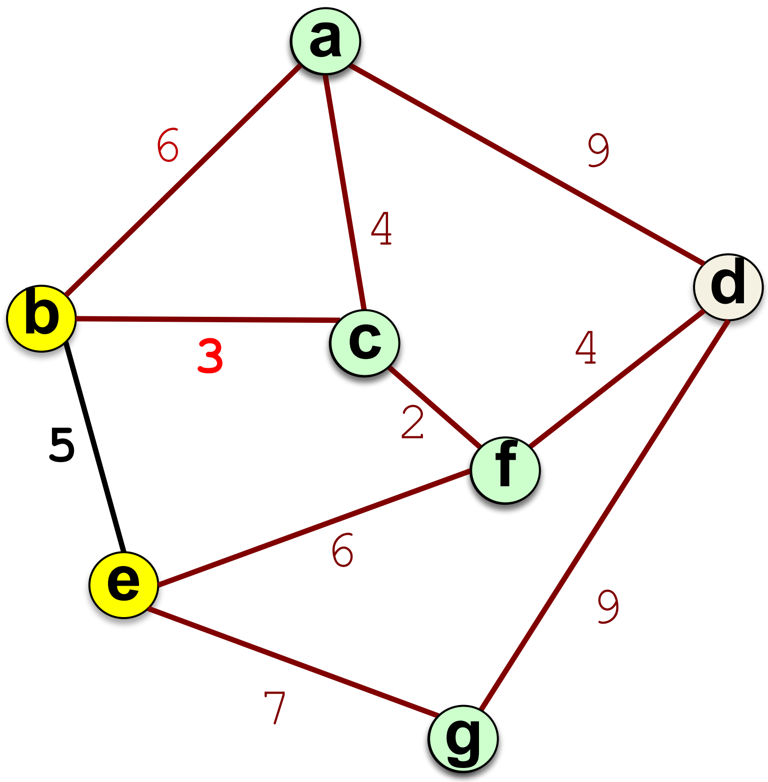
Minimum Spanning Tree: Prim's Algorithm



- Move the out node of the arc from the **Out** to the **In** set.
- Include the arc in the **MST**

```
In = [e,b]
Out = [a,c,d,e,f,g]
MST = {<b,e>}
```

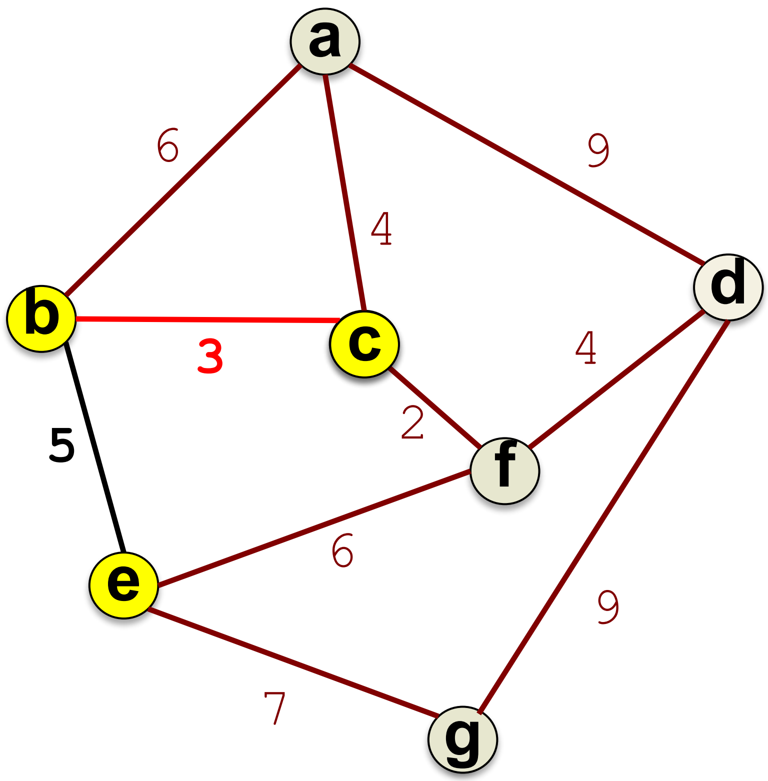
Minimum Spanning Tree: Prim's Algorithm



- Check all arcs between nodes in the **`In`** and **`Out`** sets
- Chose that with minimum cost

```
In = [b,e]
Out = [a,c,d,f,g]
MST = {<b,e>}
```

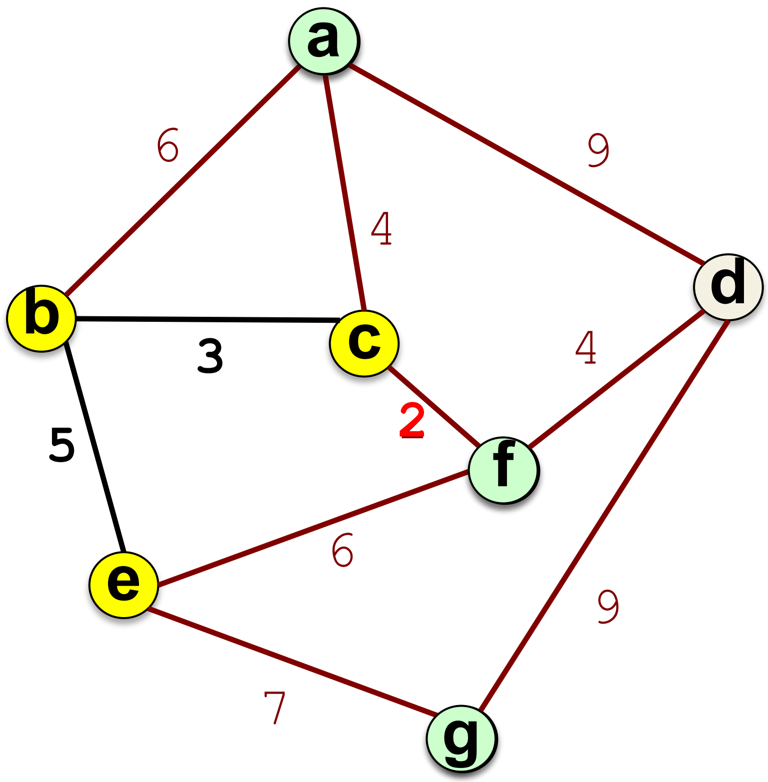
Minimum Spanning Tree: Prim’s Algorithm



- Move the out node of the arc from the **Out** to the **In** set.
- Include the arc in the **MST**

```
In = [b,c,e]
Out = [a,d,f,g]
MST = {<b,e>, <b,c>}
```

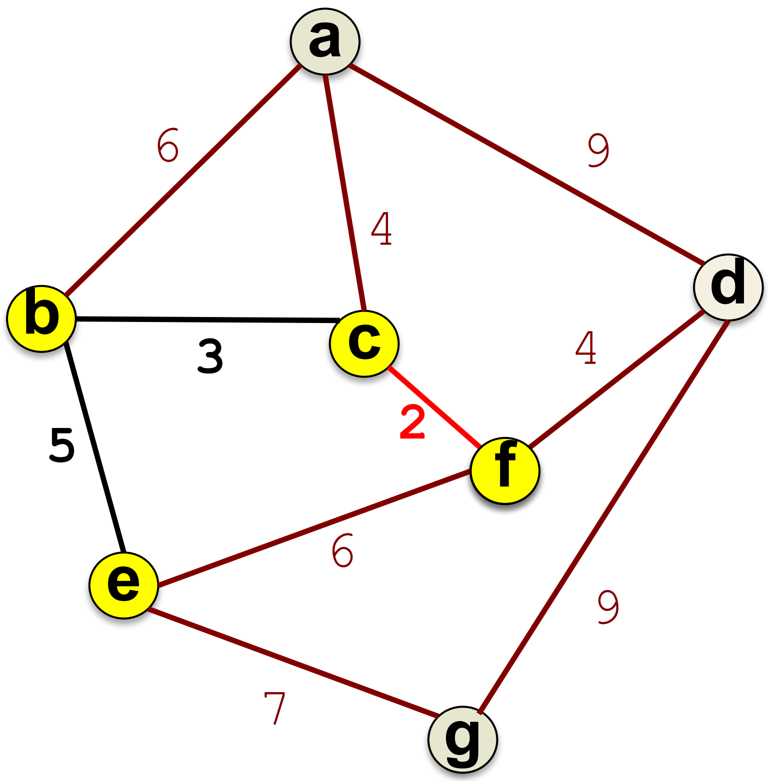
Minimum Spanning Tree: Prim's Algorithm



- Check all arcs between nodes in the **`In`** and **`Out`** sets
- Chose that with minimum cost

```
In = [b,c,e]
Out = [a,d,f,g]
MST = {<b,e>, <b,c>}
```

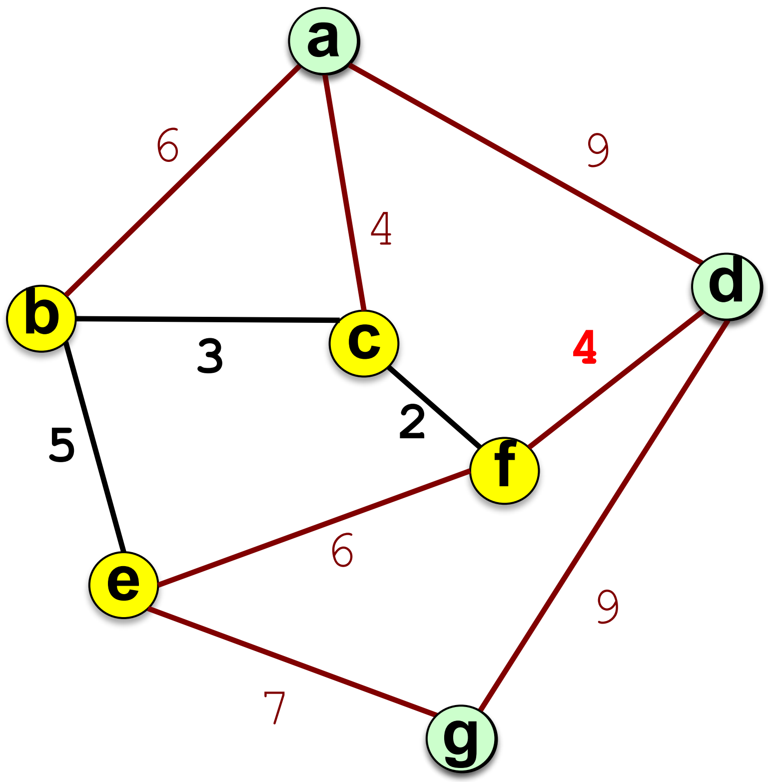
Minimum Spanning Tree: Prim's Algorithm



- Move the out node of the arc from the **Out** to the **In** set.
- Include the arc in the **MST**

```
In = [b,c,e,f,]
Out = [a,d,g]
MST = {<b,e>, <b,c>,
       <c,f>}
```

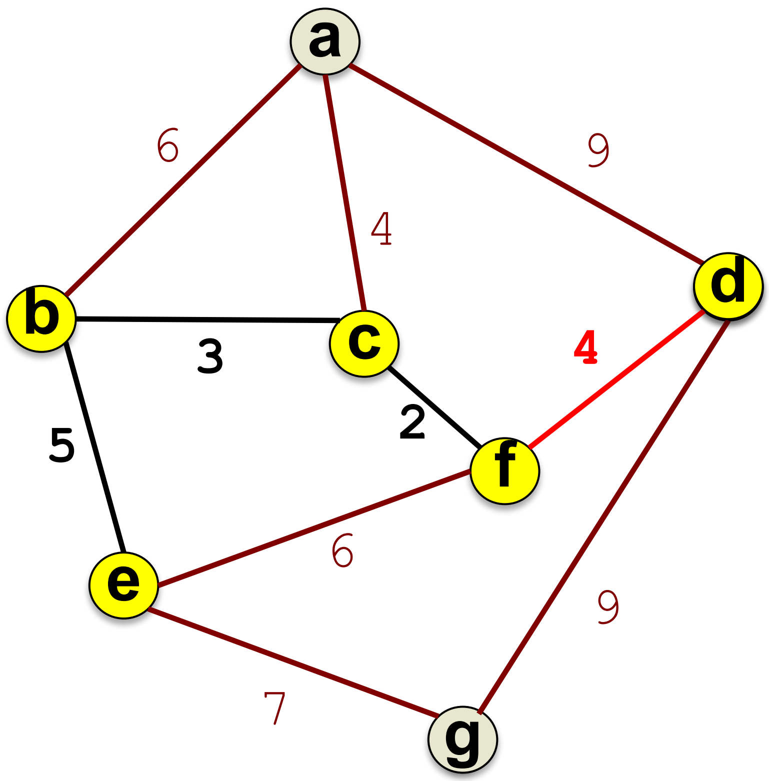
Minimum Spanning Tree: Prim's Algorithm



- Check all arcs between nodes in the **`In`** and **`Out`** sets
- Chose that with minimum cost

```
In = [b,c,e,f]
Out = [a,d,g]
MST = {<b,e>, <b,c>,
       <c,f>}
```

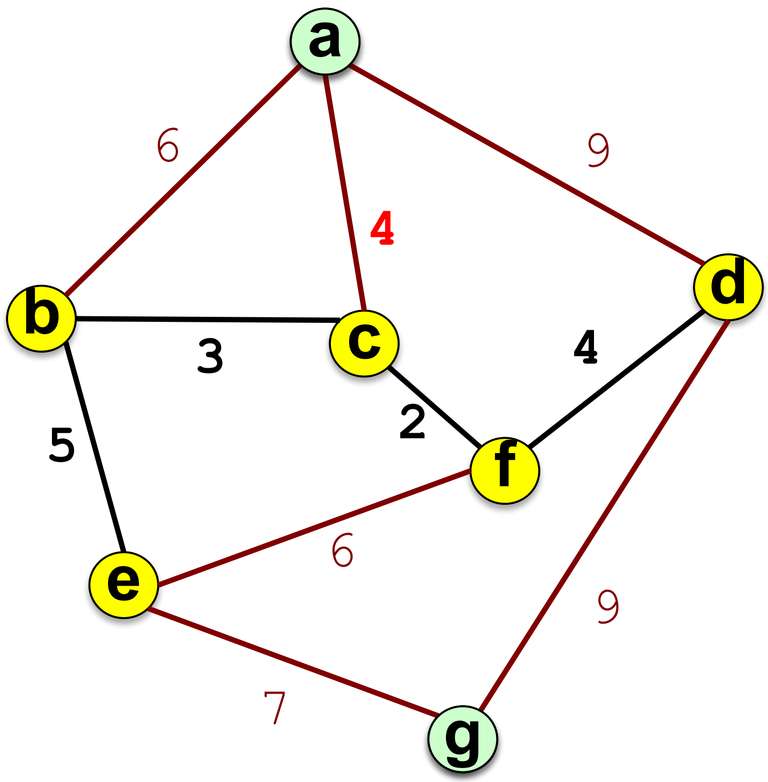
Minimum Spanning Tree: Prim's Algorithm



- Move the out node of the arc from the **Out** to the **In** set.
- Include the arc in the **MST**

```
In = [b,c,d,e,f]
Out = [a,g]
MST = {<b,e>, <b,c>,
       <c,f>, <d,f>}
```

Minimum Spanning Tree: Prim's Algorithm

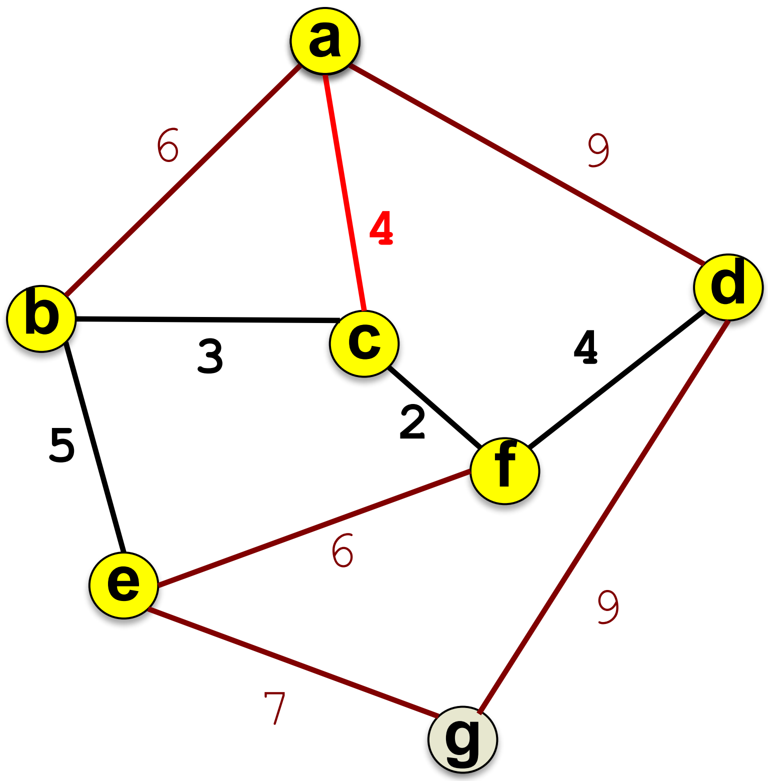


- Check all arcs between nodes in the **In** and **Out** sets
- Chose that with minimum cost

```
In = [b,c,d,e,f]
Out = [a,g]
MST = {<b,e>, <b,c>,
       <c,f>, <d,f>}
```

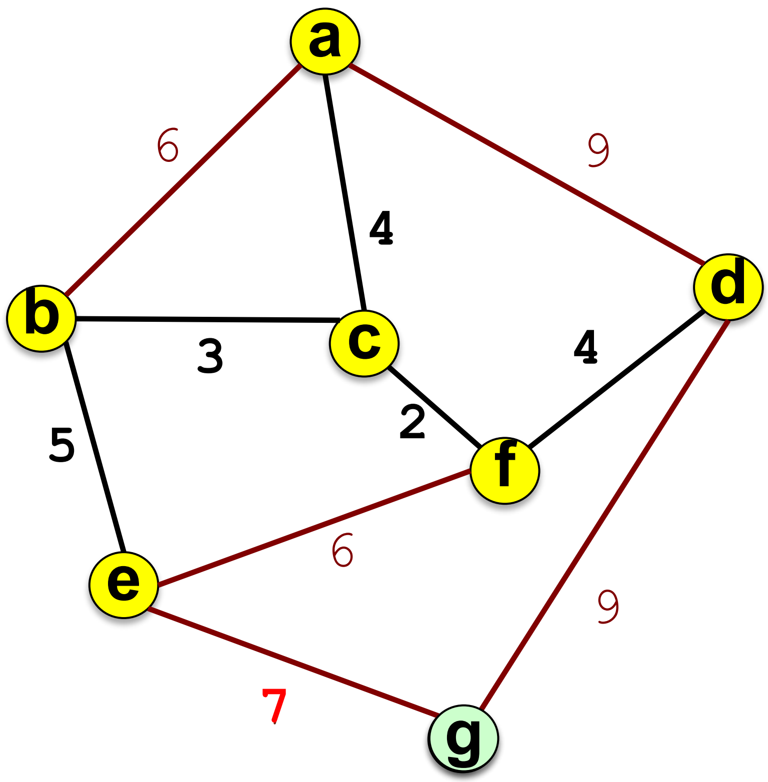
Minimum Spanning Tree: Prim's Algorithm

- Move the out node of the arc from the **`Out`** to the **`In`** set.
- Include the arc in the **`MST`**




```
In = [a,b,c,d,e,f]
Out = [g]
MST = {<b,e>, <b,c>,
       <c,f>, <d,f>
       <a,c>}
```

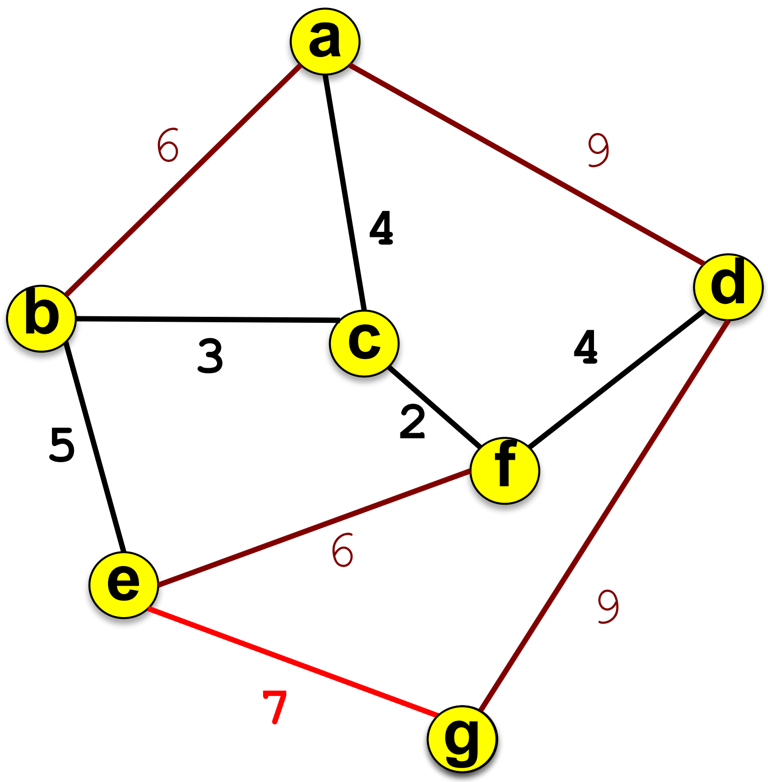
Minimum Spanning Tree: Prim's Algorithm



- Check all arcs between nodes in the **`In`** and **`Out`** sets
- Chose that with minimum cost

```
In = [a,b,c,d,e,f]
Out = [g]
MST = {<b,e>, <b,c>,
       <c,f>, <d,f>
       <a,c>}
```

Minimum Spanning Tree: Prim's Algorithm

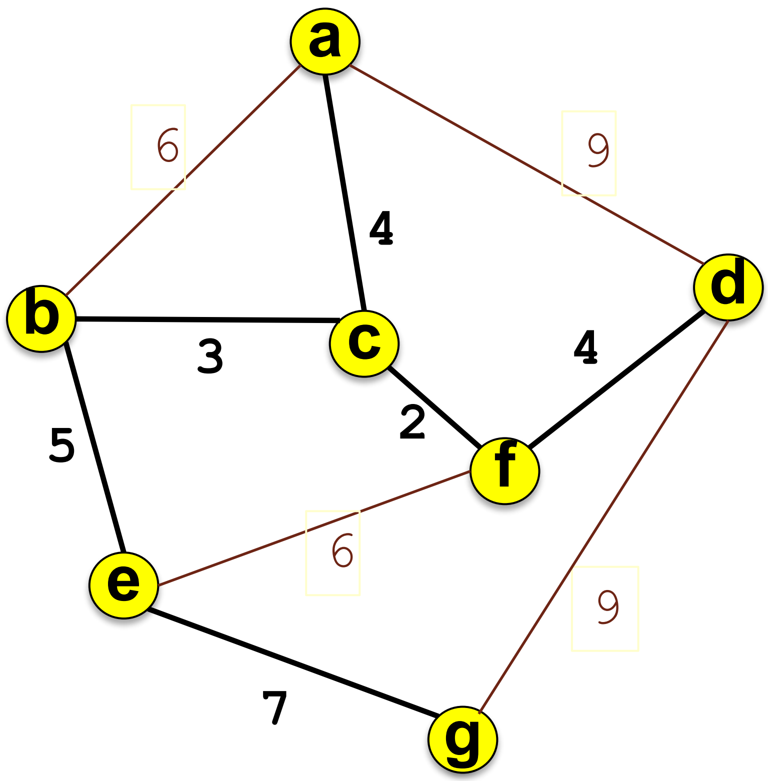


- Move the out node of the arc from the **Out** to the **In** set.
- Include the arc in the **MST**

```
In = [a,b,c,d,e,f,g]
Out = []
MST = {<b,e>, <b,c>,
        <c,f>, <d,f>
        <a,c>, <e,g>}
```

Minimum Spanning Tree: Prim's Algorithm

- The **`Out`** set is now empty
- Return the **`MST`**.




```
In = []
Out = [a,b,c,d,e,f,g]
MST = {<b,e>, <b,c>,
       <a,c>, <c,f>,
       <e,g>, <d,f>}
```

Minimum Spanning Tree: Prim's Algorithm

- Several variants can be used in the implementation of the Prim's algorithm, using appropriate data structures that make it more efficient. Here we present a naïf implementation that nonetheless is sufficient for relatively large graphs.
  1. Select arbitrarily a node from the tree to initialise the **In** set, and put the others in the **Out** set (we select node 1);
  2. While there are nodes in the **Out** set,
     - i. Find which node from the **Out** set has an arc of least cost to one connecting it to one of the nodes of the **In** set;
     - ii. Transfer the node from the **Out** set to the **In** set and include the least cost arc in the current **MST**.

```
function T = prim(G);
   n = size(G,1);
   T = ones(n)*Inf;
   In = [1]; Out = 2:n;
   while length(Out) > 0
      ...
      T = ...; In = ...; Out = ...
   end
end
```

Minimum Spanning Tree: Prim's Algorithm

- The core of the algorithm is to find the arc with least cost connecting an arc between node of the **In** and **Out** sets (implemented as vectors).
- This can be performed with a standard search for a minimum value in a matrix, but in this case, restricted to indices of nodes in the **In** and **Out** sets.
- Additionally, the position **p** of the node in the **Out** vector is stored, to make it easy to remove it from the **Out** set.
- Finally, the arc is added to **T**, the current **MST**, and the **In** and **Out** sets updated.

```
minArc = Inf;
for i = 1: length(In)
   for j = 1:length(Out)
      if G(In(i),Out(j)) < minArc
         minArc = G(In(i),Out(j));
         u = In(i); v = Out(j);
         p = j;
      end;
   end;
end;
T(u,v) = G(u,v); T(v,u) = G(v,u)
In = [v,In]; Out = [Out(1:p-1),Out(p+1:end)]
```

Minimum Spanning Tree: Prim's Algorithm

- The complete algorithm is shown below:

```
function T = prim(G);
   n = size(G,1);
   T = ones(n)*Inf;
   In = [1]; Out = 2:n;
   while length(Out) > 0
      minArc = Inf;
      for i = 1: length(In)
         for j = 1:length(Out)
            if G(In(i),Out(j)) < minArc
               minArc = G(In(i),Out(j));
               u = In(i); v = Out(j);
               p = j;
            end;
         end;
      end;
      T(u,v) = G(u,v); T(v,u) = G(v,u)
      In = [v,In]; Out = [Out(1:p-1),Out(p+1:end)]
   end
end
```

Minimum Spanning Tree: Prim's Algorithm

- It is easy to prove, by induction, that the algorithm is correct. If T is an MST with least cost with **n** nodes, adding to it the least cost arc will make it an MST with least cost with **n+1** nodes (adding any other arc would lead to a higher cost spanning tree).
- As to the worst cost complexity of the algorithm, with this implementation, we notice that the while loop is executed **n-1** times (n is the number of nodes of the graph, |V|).
- Finding the minimal cost arc requires two nested loops over ranges with **k** and **n-k** values, that is at most $\mathbf{n^2/4}$ executions (for k = n/2) of the body of the loop

```
for i = 1: length(In)
   for j = 1:length(Out)
      …
   endfor;
endfor
```

- All operations in the loop are "basic", and so the complexity of this implementation of the Prim's algorithm is $O(n*n^2/4)$ i.e. $\mathbf{O(|V|^3)}$ (where |V| = n).
- **Note:** Implementations with priority queues and other advanced data structures have better complexity, namely **O(|E|+Vlog|V|).**

Shortest Paths – Floyd-Warshall's Algorithm

- There are many algorithms for finding shortest paths between nodes of weighted graphs. They include algorithms to find one shortest path between two nodes , like the Dijskstra algorith, or to find all shortest paths between any two nodes of the graph, namely the **Floyd-Warshall's** (**FW**) algorithm.
- As the previous one, the **FW** algorithm explores dynamic programming in the following way:
- If a shortest path is considered between any two nodes, considering all paths through a **List** of **In** nodes with **n** nodes, these shortest paths can be updated by extending the list of **In** nodes with an extra node.
- Starting with an empty **List**, and including one node at a time, the final results is the set of shortest paths between any two nodes.

Shortest Paths – Floyd-Warshall's Algorithm

- The algorithm can thus be specified as follows:
  1. Initialise a matrix **S** of shortest paths with the adjacency matrix, that is only considering the direct distances between any two nodes.
     - Of course, nodes that are not connected by an arc have infinite distance between them at this stage
  1. Now, for all values k from 1 to n iterate
     - On iteration k, update S, by considering all indirect paths passing through node k.
  3. After the last iteration the set of all shortest paths between all nodes is stored in matrix S.

- Notice that this algorithm only computes the paths with shortest distance between any two nodes but does not return what these paths are.
  - In fact, a small addition to the algorithm allows the paths to be reconstructed.

Shortest Paths – Floyd-Warshall's Algorithm

- The implementation of this algorithm is shown below:

```
function S = floyd(M)
    S = M;
    n = size(S,1);
    for i = 1:n  S(i,i) = 0; end
    for k = 1:n
        for i = 1:n
            for j = 1:n
                if S(i,k) + S(k,j) < S(i,j)
                    S(i,j) = S(i,k) + S(k,j);
                end
            end
        end
    end
end
```

- The external for loop guarantees that all paths, between nodes **i** and **j**, consider, all the paths through nodes **k** (**1, 2, 3, …, n),** previously computed.
- The shortest paths are updated by considering the triangular inequality, with paths passing through the previous values of **k**.

Shortest Paths – Floyd-Warshall's Algorithm

- The correction of the algorithm can be proved by induction on the number of nodes considered in indirect paths (left as exercise).
- As to the complexity, it is easy to see that the algorithm requires 3 nested loops of size **n**, with a basic operation in the body,

```
 for k = 1:n
      for i = 1:n
         for j = 1:n
            if S(i,k) + S(k,j) < S(i,j)
               S(i,j) = S(i,k) + S(k,j);
            end
         end
      end
   end
end
```

- The complexity of the algorithm is thus **O(|V|³).**
- Notice that algorithms to compute shortest paths between 2 nodes, like the Dijskstra algorithm have complexity **O(|V|²)**, but only consider a pair (not all) of nodes.

Path Reconstruction – Floyd-Warshall's Algorithm

- The previous algorithm does not provide the shortest paths between any two nodes, but rather the shortest distances of any path between the nodes.
- Nevertheless, these paths may be easily reconstructed if the initial arc of any shortest path between two nodes is recorded in a matrix **Next** (for next node).
- For every pair **<i,j>** the matrix is initialised with **j**, i.e. it assumes that a direct path from **i** to **j** with no intermediate nodes is the best (so far).

```
for i = 1:n
   for j = 1:n
      Next(i,j) = j;
   end
end
```

- In the inner loop of the FW algorithm, if a new shortest path is found, the new leading arc is updated accordingly

```
if S(i,k) + S(k,j) < S(i,j)
   S(i,j) = S(i,k) + S(k,j);
   Next(i,j) = Next(i,k);
end
```

Path Reconstruction – Floyd-Warshall's Algorithm

- The extended FW algorithm is shown below, returning the Next matrix.

```
function [S,Next] = floyd(M)
   S = M;
   n = size(S,1);
   for i = 1:n
      for j = 1:n
         Next(i,j) = j;
      end
   end
   for k = 1:n
      for i = 1:n
         for j = 1:n
            if S(i,k) + S(k,j) < S(i,j)
               S(i,j) = S(i,k) + S(k,j);
               Next(i,j) = Next(i,k);
            end
         end
      end
   end
end
```

Path Reconstruction – Floyd-Warshall's Algorithm

- Once the matrix **Next** is returned the path between any two nodes, **u** and **v**, can be obtained by following the trail indicated by this matrix, as shown below

```
function P = path(u,v,Next)
   if N(u,v) == inf
      P = [];
      return;
   end
   P = [u];
   while u != v
      u = Next(u,v)
      P = [P,u];
   end
end
```

- The first test checks whether there is a "real" path between nodes **u** and **v**.
- Otherwise the path is "reconstructed", starting iin node **u**.
- With this reconstruction technique, the complexity of the FW algorithm is not changed, and the paths are only computed when needed.