NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Graph Algorithms; Dynamic Programming

Pedro Barahona
DI/FCT/UNL
Computational Methods
1st Semester 2021/22

Graphs

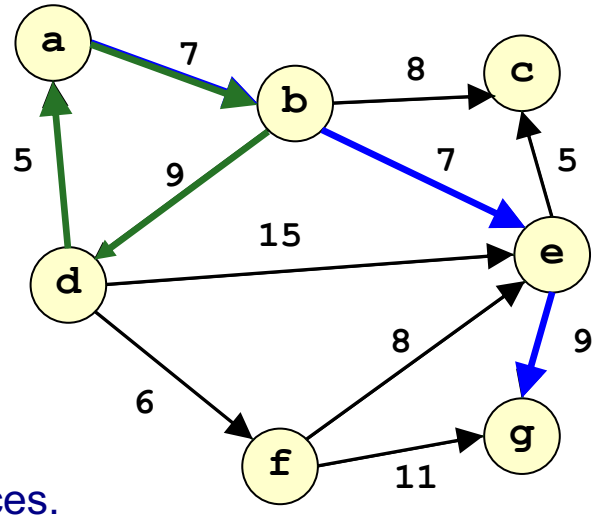
NOVA SCHOOL OF SCIENCE & TECHNOLOGY

- Graphs are a very common data structure that is useful to model a number of "network" applications, where a number of "agents" have direct connections between (some of) them.
- They range from networks of physical services (telecommunications, roads, water distribution) to more virtual services (e.g. social networks) or even to more abstract models (neighbouring countries, teams playing in several competitions, …).
- Formally, a **graph** is defined as a pair **<V,E>** where
  - **V** is a set of **vertices** (or **nodes**)
  - **E** is a set of **edges** (or **arcs**), each connecting two of the vertices
- Two characteristics of the edges, weights and direction, might be considered, leading to different types of graphs:
  - **Weighted Graphs** – Each edge has a weight, usually a positive number
  - **Directed Graphs –** Each edge has a direction, connecting one vertice to another, but not the other way round

Graphs

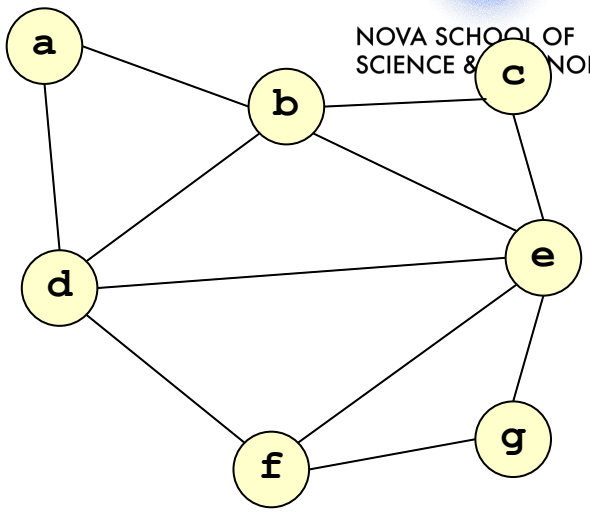
Example:

- An unweighted, undirected graph
- A weighted, undirected graph
- A weighted, directed graph



- A **path** is a sequence of connected vertices.
  - **Example**: Path: a → b → e → g
  - **Note**: A path is directional, even if the underlying graph is not.
- A **cycle** is a path starting and ending in the same vertex.
  - **Example:** Cycle: a → b → d → a

Graphs



- Two nodes are **adjacent (or neighbours)** if there is an edge between them.
  - Example: **adjacent(e,f)** but not **adjacent(a,g)**
- The **degree** of a vertex is the number of its adjacent vertices
  - Example: **degree(e) = 5**, **degree(b) = 4**
- A **graph ordering** is the assignment of a total order to the nodes of the graph, (i.e. the assignment of values **1..n** to the **n** nodes of a graph)
  - Example: **O = a < b < c < d < e < f < g**
- The **width of a node given a graph ordering**, is the number of adjacent nodes lower in the ordering.
  - Example: **width(e,O) = 3** , i.e. nodes **b,c,d** are lower in **O**
- The **width of a graph given a graph ordering**, is the maximum width of its nodes given that ordering.
  - Example: **width(G,O)** = 3 , since **e** is the node with highest width in **O**
- The **width of a** graph is the minimum width of the graph over all its orderings.

Properties of Graphs

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

- In general, given a graph, there are several problems that may be considered to compute some properties of the graphs, such as:
  - **Connectedness:** Is there a path **connecting** any two vertices of a graph?
  - What is the **shortest path** (number of edges, sum of the edges weights) between any two vertices?
  - What is the **width** of a graph?
  - Are there **cycles** in the graph, or is it a **tree** (i.e. with a unique path between two vertices, or equivalently the graph has width 1)?
  - What is the **shortest spanning tree**?
  - Are there **Hamiltonian** cycles in the graph (including all vertices only once – except the initial/final vertex). Which one(s) is the **shortest**?
  - Are there **cliques** in the graph - subset of the graph where any two nodes are adjacent). Which one(s) is **maximal** (have more nodes).
  - Is it possible to **colour** a graph with a set of colours, such that two adjacent vertices have different colours? What is the **minimum** cardinality of such set?

Properties of Graphs

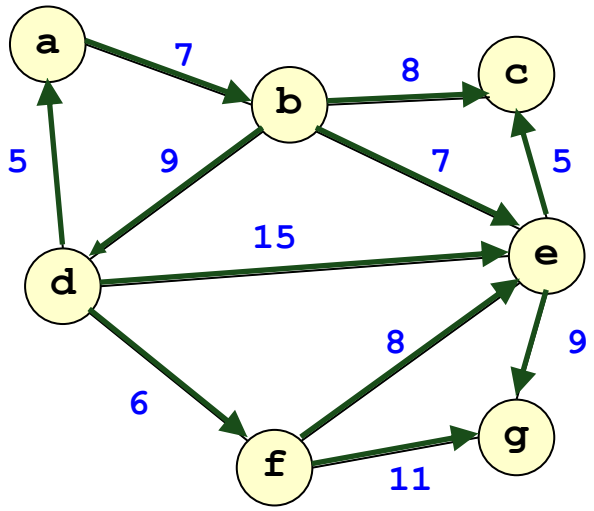
NOVA SCHOOL OF SCIENCE & TECHNOLOGY

- The problems above, and many others, are typically posed in many applications, and so a number of algorithms have been studied to solve them.
- But before studying some of these algorithms, it is important to adopt a **representation** (or **encoding**) for the implementation of a graph.
- Here we will present the two most common encodings:
  - **Adjacency matrix.**
  - **Adjacency lists.**
- The **adjacency matrix** is possibly the most intuitive way of implementing a graph. Given a **graph** with **n** vertices and some graph ordering, the adjacency matrix is a **square n × n Boolean matrix G**, whose elements $\mathbf{G_{i,j}}$ contain information about the edges between nodes **i** and **j**.
  - In an unweighted graph, the elements are Booleans
  - In a weighted graph, the elements are the weights
  - In a undirected graph the matrix is symmetric, otherwise it is usually asymmetric.

Graphs

Example:

	a	b	c	d	e	f	g
a	0	7	-1	-1	-1	-1	-1
b	-1	0	8	9	7	-1	-1
c	-1	-1	0	-1	-1	-1	-1
d	5	-1	-1	0	15	6	-1
e	0	-1	5	-1	0	-1	9
f	-1	-1	-1	-1	8	0	11
g	-1	-1	-1	-1	9	-1	0

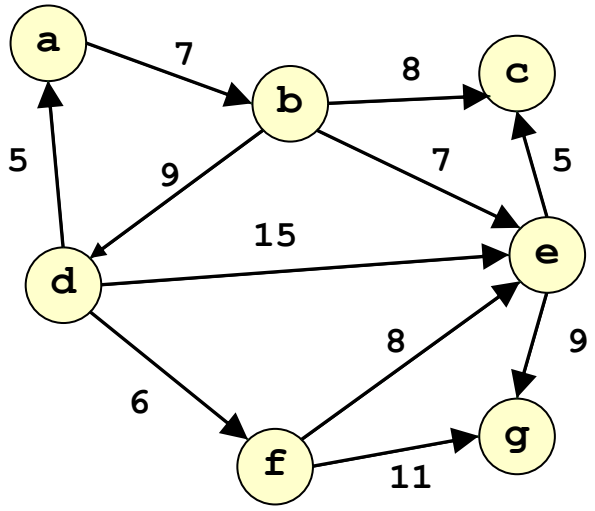


Properties of Graphs

- The adjacency matrix is a very inefficient representation of **sparse** graphs, i.e. where only a "few" of the potential arcs are presented. In this case, of the n^2 elements of the matrix only a (small) fraction of them are non-zero.
- To avoid this waste of space, one may adopt an **adjacency lists**, i.e. a set of lists each representing, for each node, the information about its neighbours (taking into account the directedness).

	a	b	c	d	e	f	g
a	0	7	-1	-1	-1	-1	-1
b	-1	0	8	9	7	-1	-1
c	-1	-1	0	-1	-1	-1	-1
d	5	-1	-1	0	15	6	-1
e	0	-1	5	-1	0	-1	9
f	-1	-1	-1	-1	8	0	11
g	-1	-1	-1	-1	9	-1	0

a	b:7		
b	c:8	d:9	e:7
c			
d	a:5	e:15	f:6
e	e:5	g:9	
f	e:8	g:11	
g			



- The space required is thus **O(|E|)** which is much less than **O(|V²|)** for sparse graphs.

Types of Algorithms

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

- As we will see, some of these problems require algorithms whose asymptotical complexity is **polynomial** on **n**, the **input size** of the problem. Assuming that reads from and writes to memory are *basic operations,* **polynomial algorithms** require **$O(n^k)$** *basic operations,* where **k** is an integer, typically small.
- Problems that can be solved by polynomial algorithms are said to be in class **P**.
- Other algorithms have exponential complexity, i.e. require **$O(k^n)$** *basic operations.* Problems that can only be solved by these are said to be in class **NP**.
- Take a computer where each elementary operation takes 1 nsec. The following table shows the "practical" consequences of the problem being in **P** or in **NP**. Here the size **n** is the size of an input vector or matrix, or the size **|V|** or **|E|** of a graph.

n^1: **Search** in a vector; **n^2**: **Sorting** (naïf) a vector; **n^3**: Matrix **multiplication**

n	10	20	30	40	50	60	70
n^1	10 nsec	20 nsec	30 nsec	40 nsec	50 nsec	60 nsec	70 nsec
n^2	100 nsec	400 nsec	900 nsec	1.6 μsec	2.5 μsec	3.6 μsec	4.9 μsec
n^3	1 μsec	8 μsec	27 μsec	64 μsec	125 μsec	216 μsec	343 μsec
2^n	1 μsec	1 msec	1 sec	18 min	13 days	37 years	37 K years

Connectedness of Graphs

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Problem (Connectedness): Check whether a graph G is connected.

- The definition of connectedness of a graph depends on its type:
  - An undirected graph is **connected** if there is a path between any two nodes of the graph.
  - A directed graph is **strongly connected** is there is a path between any two nodes of the graph, respecting the direction of the its arcs.
  - A directed graph is **weakly connected** is there is a path between any two nodes of the corresponding undirected graph.
- Here we will study the case for the undirected graphs, which is easier to decide, since paths (being reflexive, symmetric and transitive) create classes of equivalence.
- We will shortly present an algorithm that checks the connectedness of undirected graphs, as a side-effect of finding a minimal spanning tree.

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

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

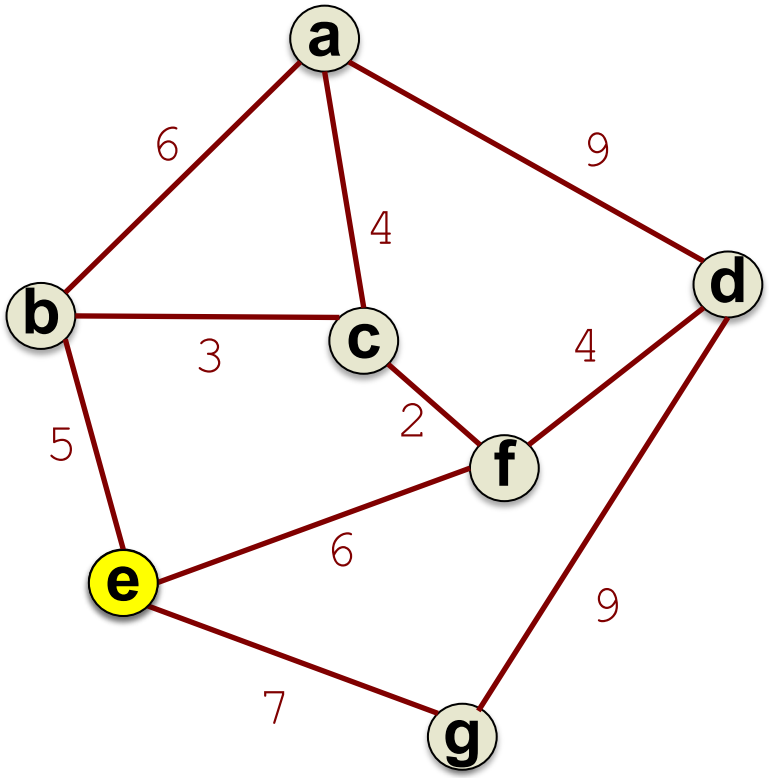
- A **spanning tree** is a subset of a connected graph that has the topology of a tree and covers all nodes of the graph.
- It has many applications, namely to provide services to a number of sites (the nodes) that can be interconnected in several ways (by a graph), but using the a minimal number of 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

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

- 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 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**.

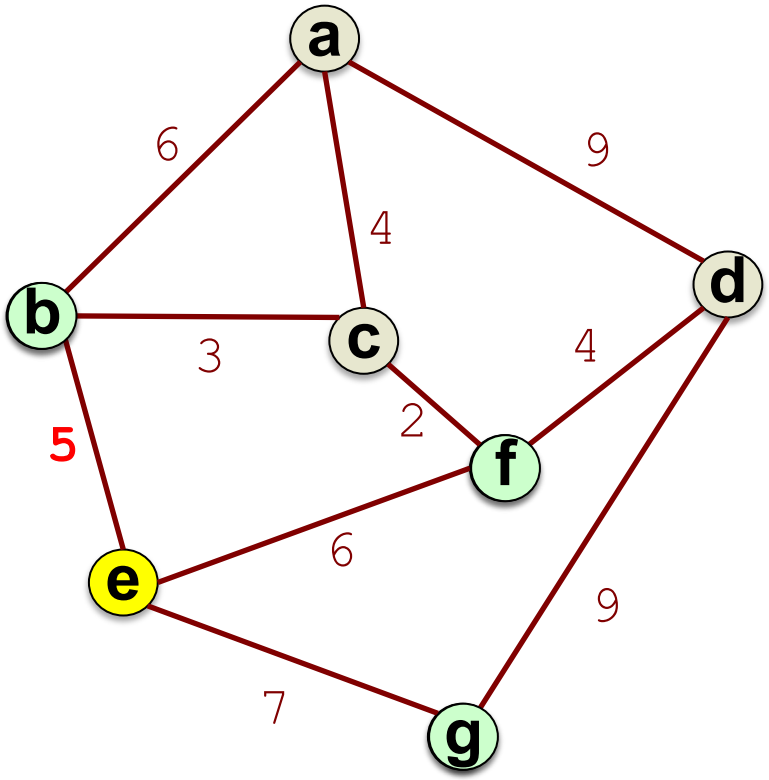
Minimum Spanning Tree: Prim's Algorithm



- 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 = {}
```

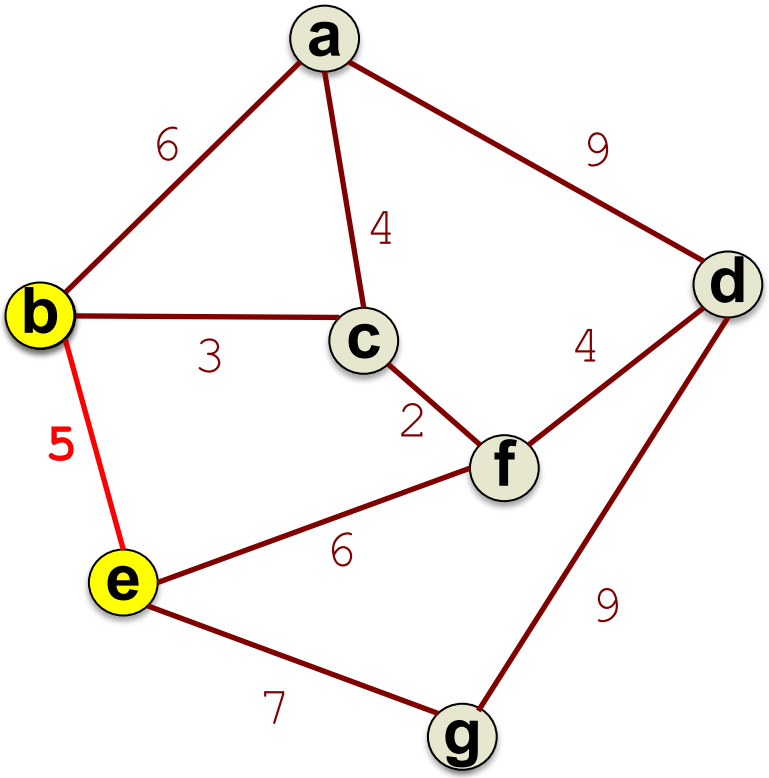
Minimum Spanning Tree: Prim's Algorithm



- 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 = {}
```

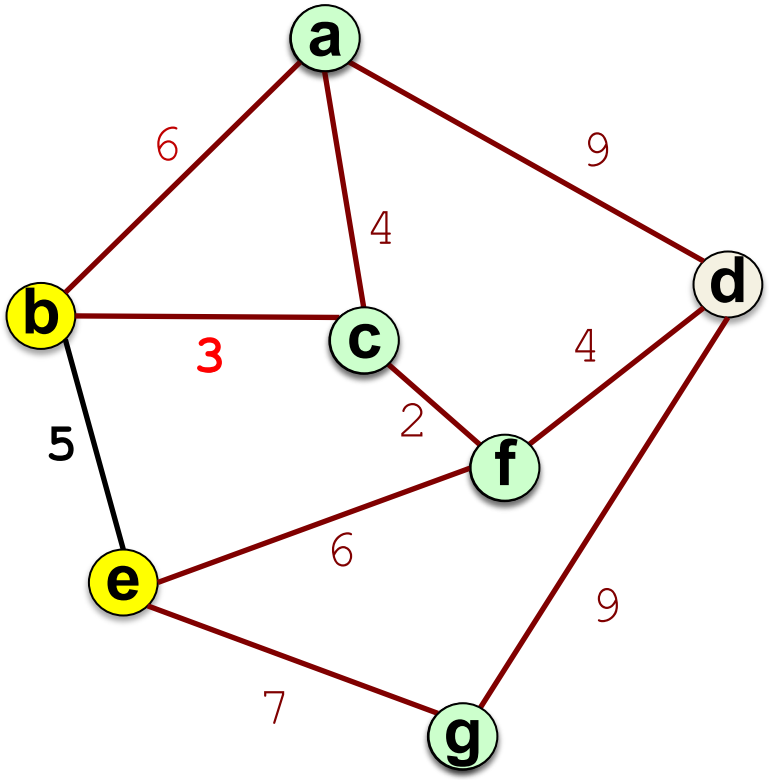
Minimum Spanning Tree: Prim's Algorithm



- 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>}
```

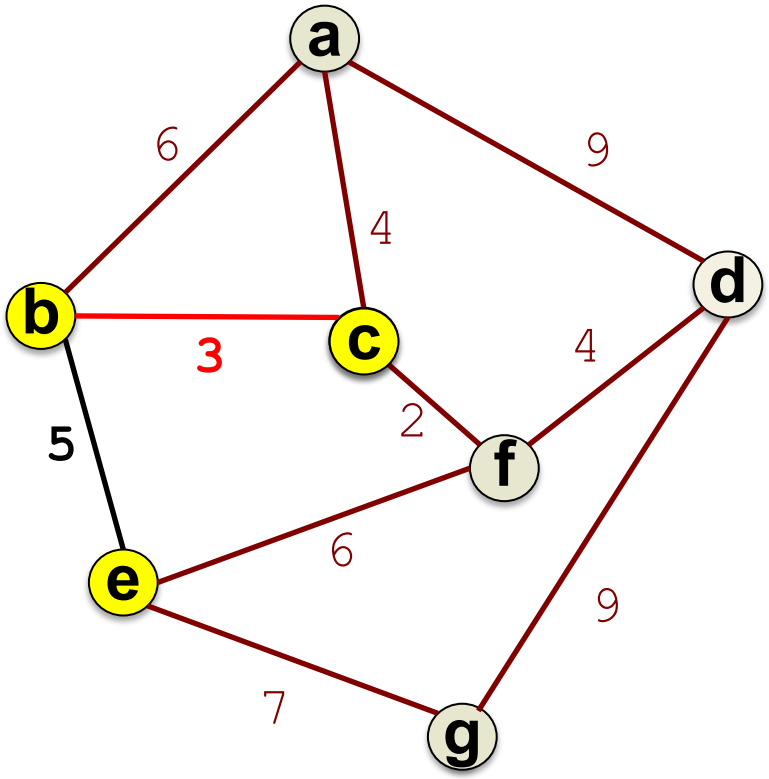
Minimum Spanning Tree: Prim's Algorithm



- 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>}
```

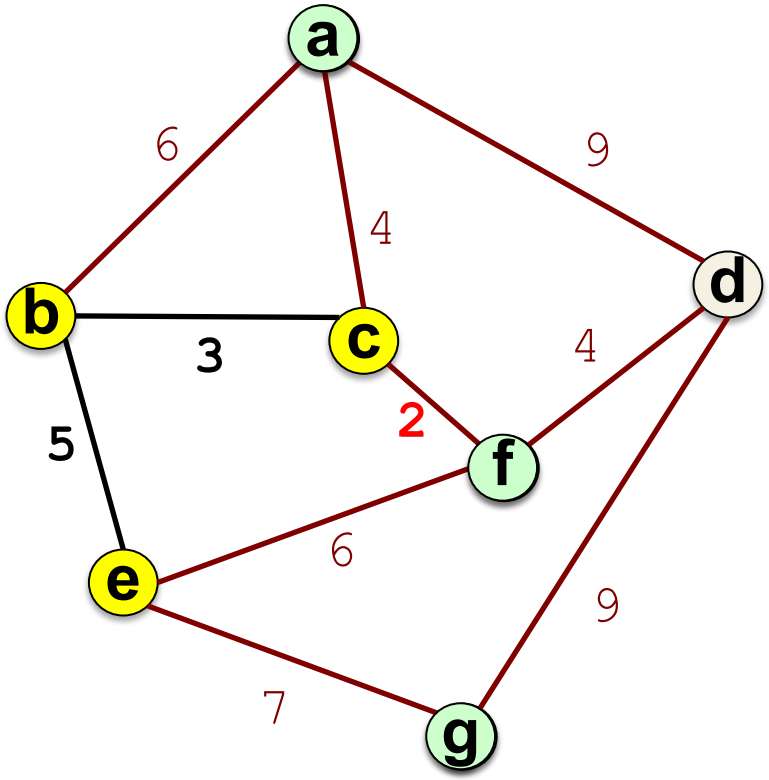
Minimum Spanning Tree: Prim's Algorithm



- 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>}
```

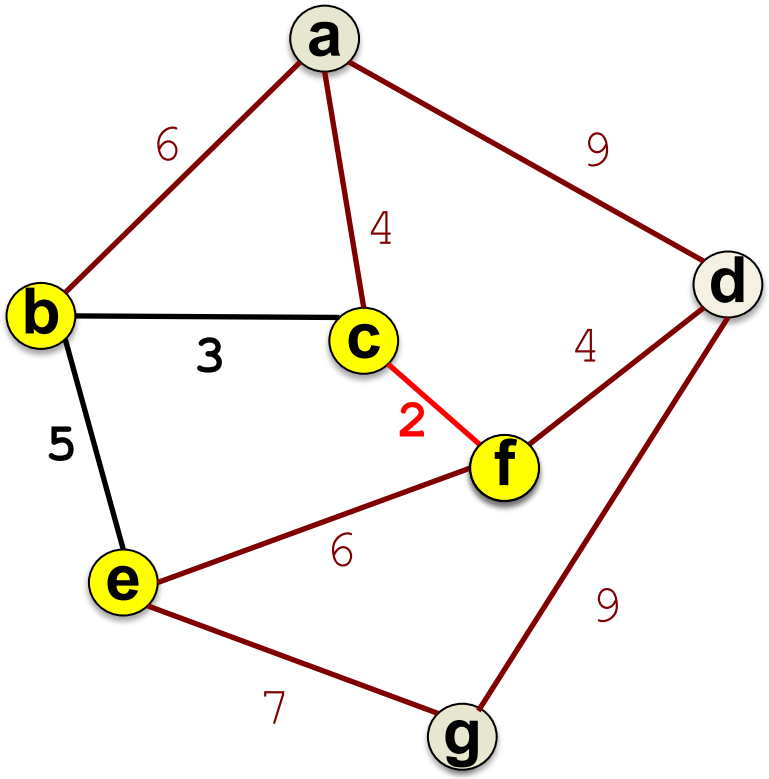
Minimum Spanning Tree: Prim's Algorithm



- 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>}
```

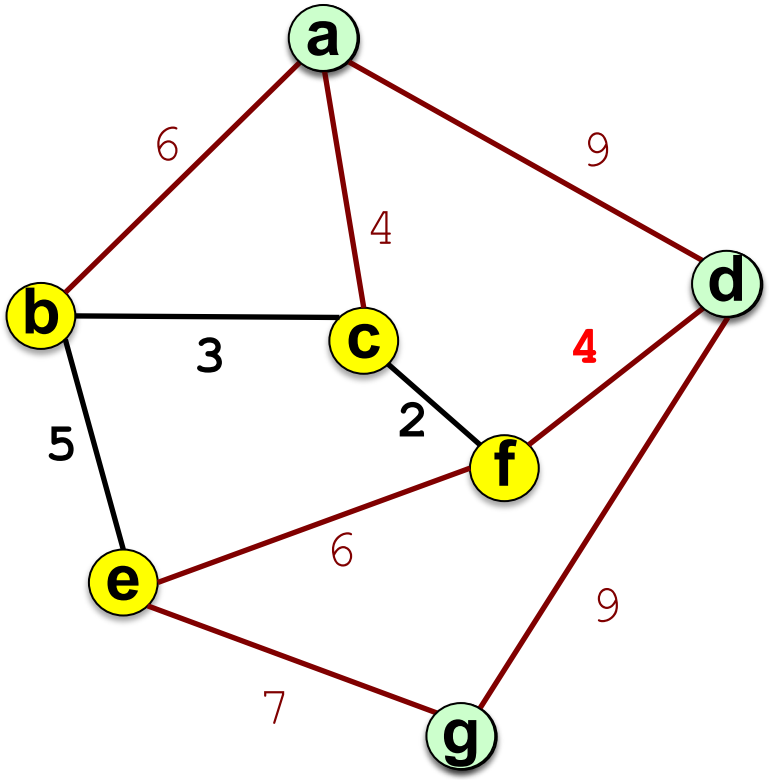
Minimum Spanning Tree: Prim's Algorithm



- 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>}
```

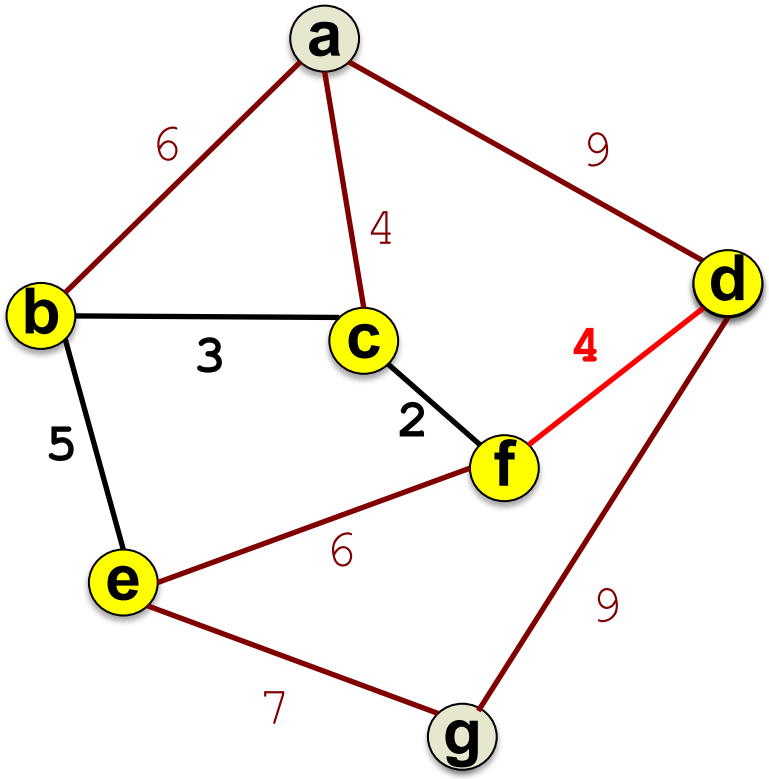
Minimum Spanning Tree: Prim's Algorithm



- 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>}
```

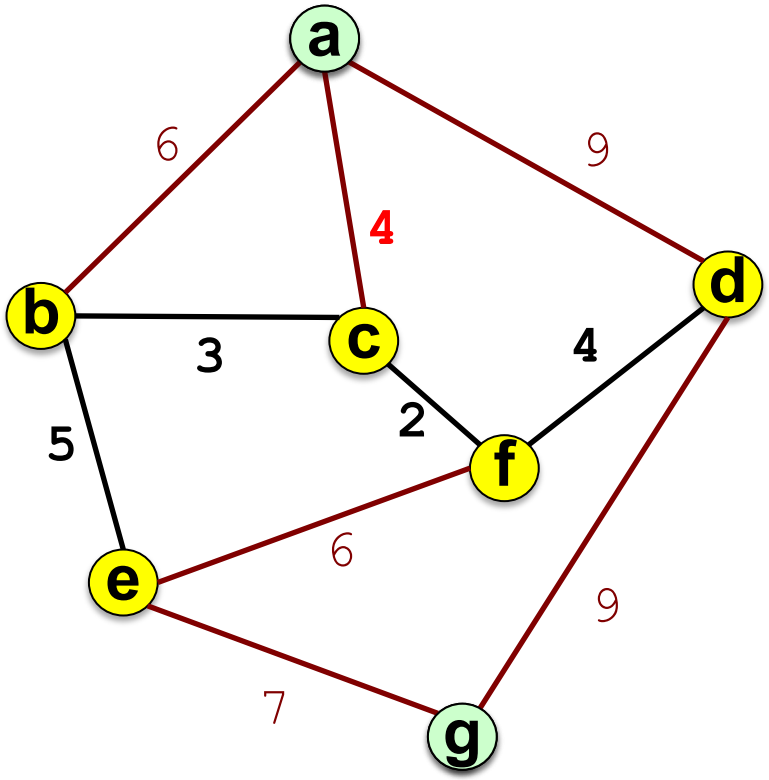
Minimum Spanning Tree: Prim's Algorithm



- 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>}
```

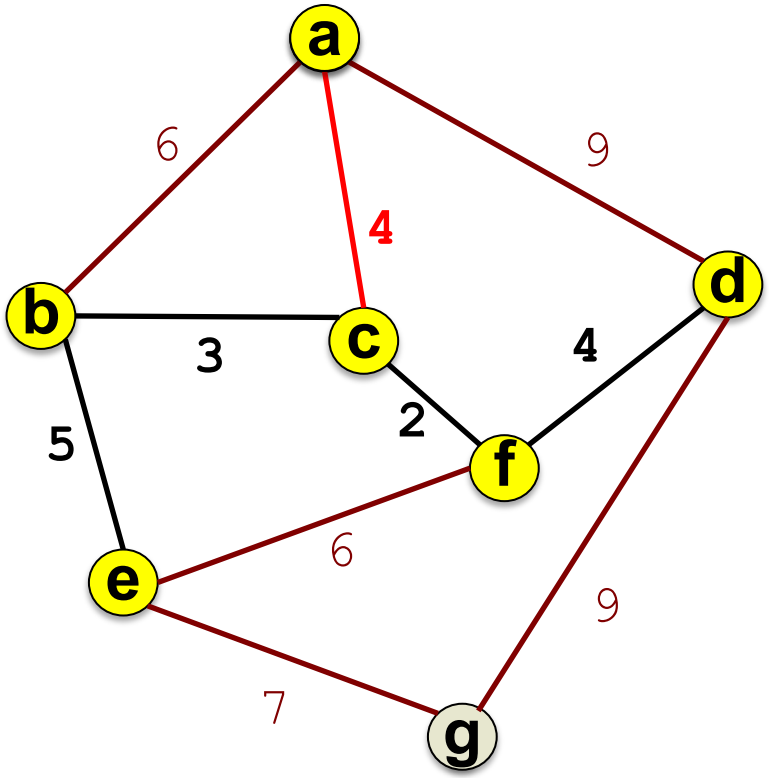
Minimum Spanning Tree: Prim's Algorithm



- 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>}
```

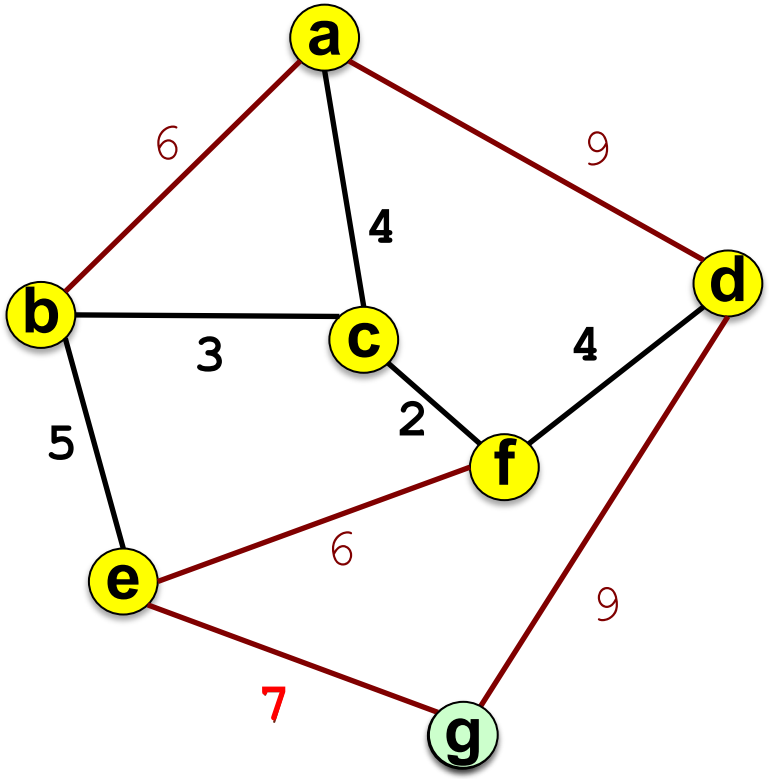
Minimum Spanning Tree: Prim's Algorithm



- 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>}
```

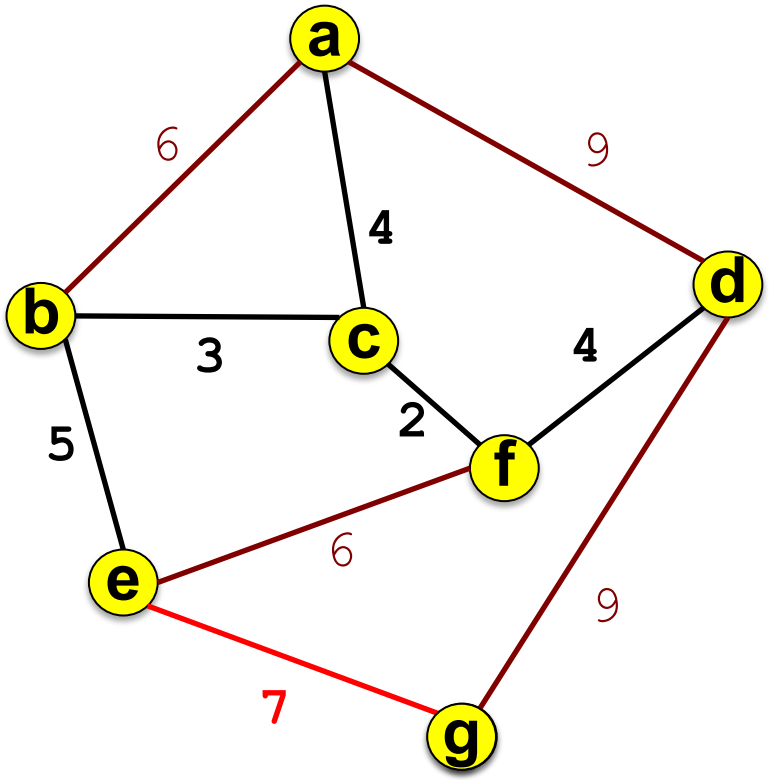
Minimum Spanning Tree: Prim's Algorithm



- 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>}
```

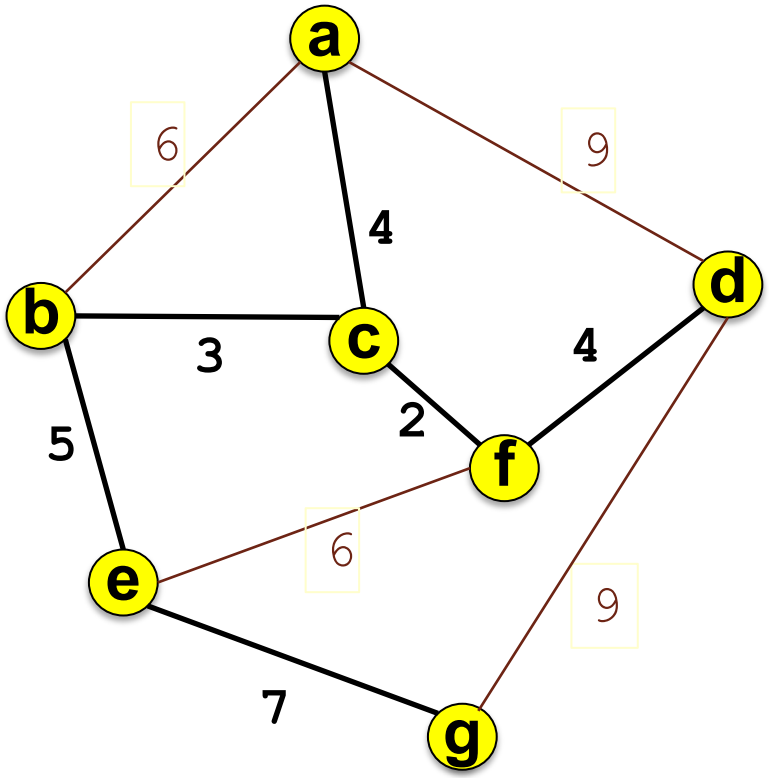
Minimum Spanning Tree: Prim's Algorithm



- 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

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

- 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 simple implementation that nonetheless is acceptable for relatively large graphs.
    1. Select arbitrarily a node from the tree to initialise the **In** set (here node 0).
    2. Put the other nodes in the **Out** set.
    3. Initialises the Minimum spanning tree **T** to an empty graph.
    4. Update and eventually returns the Minimum spanning tree **T.**

```
def prim(G):
    """ Returns the minimum spanning tree of graph G,
    using the prim algorithm"""
    n = len(G)
    In = [0]
    Out = [i for i in range(1,n)]
    row = [-1 for i in range(n)]       # an n-vector with -1s
    T = [row.copy() for i in range(n)] # an nxn-matrix with -1s
    while ...:
        ...
    return (T)
```

Minimum Spanning Tree: Prim's Algorithm

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

- The tree is then updated as follows:
  1. While there are nodes in the **Out** set,
     i. Find the arc of least cost between a node **u** in the **In** set and a node **v** from the **Out** set (if there is one!);
     ii. If no arc is selected, that means the graph is not connected and should be returned (together as the remaining Out nodes)
     iii. Include the least cost arc in the current **MST**.
     iv. Transfer the node from the **Out** set to the **In** set and

```
while len(Out) > 0:
    min_arc = math.inf
    u = v = 0
    for i in In:
        for j in Out:
            if G[i][j] > 0 and G[i][j] < min_arc:
                u = i
                v = j
                min_arc = G[i][j]
            ...
```

Minimum Spanning Tree: Prim's Algorithm

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

- The tree is then updated as follows:
  1. While there are nodes in the **Out** set,
     i. Find the arc of least cost between a node **u** in the **In** set and a node **v** from the **Out** set (if there is one!);
     ii. If no arc is selected, that means the graph is not connected the "disconnected" nodes, **Out** , are returned; otherwise
     iii. Include the least cost arc in the current **MST**.
     iv. Transfer the node from the **Out** set to the **In** set

```
while len(Out) > 0:
    ...
                u = i
                v = j
    ...
    if u == v == 0:
        return (Out)
    T[u][v] = G[u][v]
    T[v][u] = G[v][u]
    In.append(v)
    Out.remove(v)
```

Minimum Spanning Tree: Prim's Algorithm

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

- The complete algorithm is shown below:

```
def prim(G):
    """ ... """
    n = len(G)
    In = [0]
    Out = [i for i in range(1,n)]
    row = [-1 for i in range(n)]
    T = [row.copy() for i in range(n)]
    while len(Out) > 0:
        min_arc = math.inf
        u = 0
        v = 0
        for i in In:
            for j in Out:
                if G[i][j] > 0 and G[i][j] < min_arc:
                    u = i
                    v = j
                    min_arc = G[i][j]
        if u == 0 and v == 0:
            return (Out)
        T[u][v] = G[u][v]
        T[v][u] = G[v][u]
        In.append(v)
        Out.remove(v)
    return (T)
```

Minimum Spanning Tree: Prim's Algorithm

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

- 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|).

```
while len(Out) > 0:
  ...
  for i in In:
      for j in Out:
          ...
```

- Finding the minimal cost arc, when **k** nodes are already in the **In** list, requires two nested loops over ranges with **k** and **n-k** values, i.e. at most $\mathbf{n^2/4}$ (for **k** = n/2) executions of the body of the loop
- 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** algorithm, 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:
- The initial shortest path between any two nodes, is the direct distance (that can be infinite).
- A list **In** is initialised with all **n** nodes;
- The current shortest distance between two nodes is then updated by checking whether an indirect path exists passing in each of the nodes in list **In**.
- The final result is a matrix with all minimal distances between any two nodes.

Shortest Paths – Floyd-Warshall's Algorithm

- The algorithm can thus be implemented as follows:
  1. Initialise a matrix **S** of shortest paths with the adjacency matrix (that is, only direct distances between any two nodes are initially considered).
     - Of course, nodes that are not directly connected by an arc have a distance of -1 at this stage. For convenience, we will assign them to **inf**.
  1. Now, for all values **k** in the **In** list (i.e. from 0 to n-1) iterate.
     - In iteration **k**, update **S**, by considering all indirect paths between nodes **i** and **j** passing through node **k.**
  3. After the last iteration, matrix S contains all shortest paths between any two nodes of G.

- 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, coming shortly, allows the paths to be obtained.

Shortest Paths – Floyd-Warshall's Algorithm

- The first loop sets the initial paths between nodes i and j to be the direct paths, if they exist.
- The shortest paths are updated by considering the triangular inequality, with paths passing through the previous values of **k**.

```
def floyd(G):
 """ Returns the minimum distances between any two modes
    of graph G, using the Floyd-Warshall's algorithm."""
    n = len(G)
    S = [ [ math.inf for i in range(n) ] for j in range(n) ]
    for i in range(n):
        for j in range(n):
            if G[i][j] != -1:
                S[i][j]= G[i][j]
    for k in range(n):
      for i in range(n):
         for j in range(n):
            if S[i][k] + S[k][i]  < S[i][j]:
               S[i][j]  = S[i][k]  + S[k][j]
    return S
```

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 in range(n):
    for i in range(n):
        for j in range(n):
```

- The complexity of the algorithm is thus **$O(|V|^3)$.**
- Notice that algorithms to compute shortest paths between 2 nodes, like the Dijkstra algorithm, have complexity **$O(|V|^2)$**, but they do not consider the distance between all the 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 a matrix is computed during the FW algorithm, to it possible to later compute the shortest path from some node **i** to another node **j**.
- Matrix **Next** plays this role. In such matrix, **`Next[i][j] = k`** means that the shortest path from node **i** to node **j**, starts in arc `i` → `k`.
- All that is needed is to compute matrix **Next** during the FW algorithm. To do so, all values **`Next[i][j]`** should be initialised with **j** (in the beginning, i.e. before exploring the graph, the best path is the direct path).
- In fact, if there is no connection between nodes **i** and **j**, **`Next[i][j]`** should be initialised to **`inf`**, to account for that non-connection.
- Then, if a better path is found through node **k**, **`Next[i][j]`** must be updated to **`Next[i][k]`**, i.e. to go from **i** to **j**, one should start in the best arc to go from **i** to **k**.
- Better paths are found in the inner loop of the FW algorithm, so one needs simply to add some extra instructions to function floyd just developed.

Path Reconstruction – Floyd-Warshall's Algorithm

- The initialisation of matrix Next (i.e. **Next[i][j] = j**) can be implemented as:

```
Next = [ [ j for j in range(n) ] for i in range(n) ]
```

- If there is no connection, this should be accounted for in **Next:**

```
if G[i][j] != -1 :
    ...
else:
    Next[i][j] = -1
```

- The update of the elements of Next may be done in the inner loop of the floyd function, taking into account that
  - If a better path is found, through node **k**, **Next[i][j]** must be updated to **Next[i][j]**, i.e. to go from **i** to **j**, one should start in the best arc to go from **i** to **k**.

```
for k in In:
    for i in range(n):
        for j in range(n):
            if S[i][k] + S[k][i]  < S[i][j]:
                S[i][j]  = S[i][k]  + S[k][j]
                Next[i][j] = Next[i][k]
```

Path Reconstruction – Floyd-Warshall's Algorithm

- The completed Floyd function is thus shown below (changes in bold).

```
def floyd(G):
    """ ..."""
    n = len(G)
    S = [ [ math.inf for i in range(n) ] for j in range(n) ]
    Next = [ [ j for j in range(n) ] for i in range(n) ]
    for i in range(n):
        for j in range(n):
            if G[i][j] != -1:
                S[i][j]= G[i][j]
            else:
                Next[i][j] = -1
    for k in range(n):
      for i in range(n):
         for j in range(n):
            if S[i][k] + S[k][i]  < S[i][j]:
               S[i][j]  = S[i][k]  + S[k][j]
               Next[i][j] = Next[i][k]
    return (S, Next)
```

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

```
def path(u,v,Next):
    """ Returns the shortest path between nodes u and v,
    according to matrix Next computed with function floyd."""
    if Next[u][v] == -1:
        return []
    P = [u]
    while u != v:
        u = Next[u][v]
        P.append(u)
    return P
```

- The first test checks whether there is any path between nodes **u** and **v**. If not it returns an empty path.
- Otherwise the path is "reconstructed", starting in node **u** …
- … and continuing until node v is reached.
- With this reconstruction technique, the complexity of the FW algorithm is not changed, and the paths are only computed when needed.