MONASH College

MCD4710

Introduction to algorithms and programming

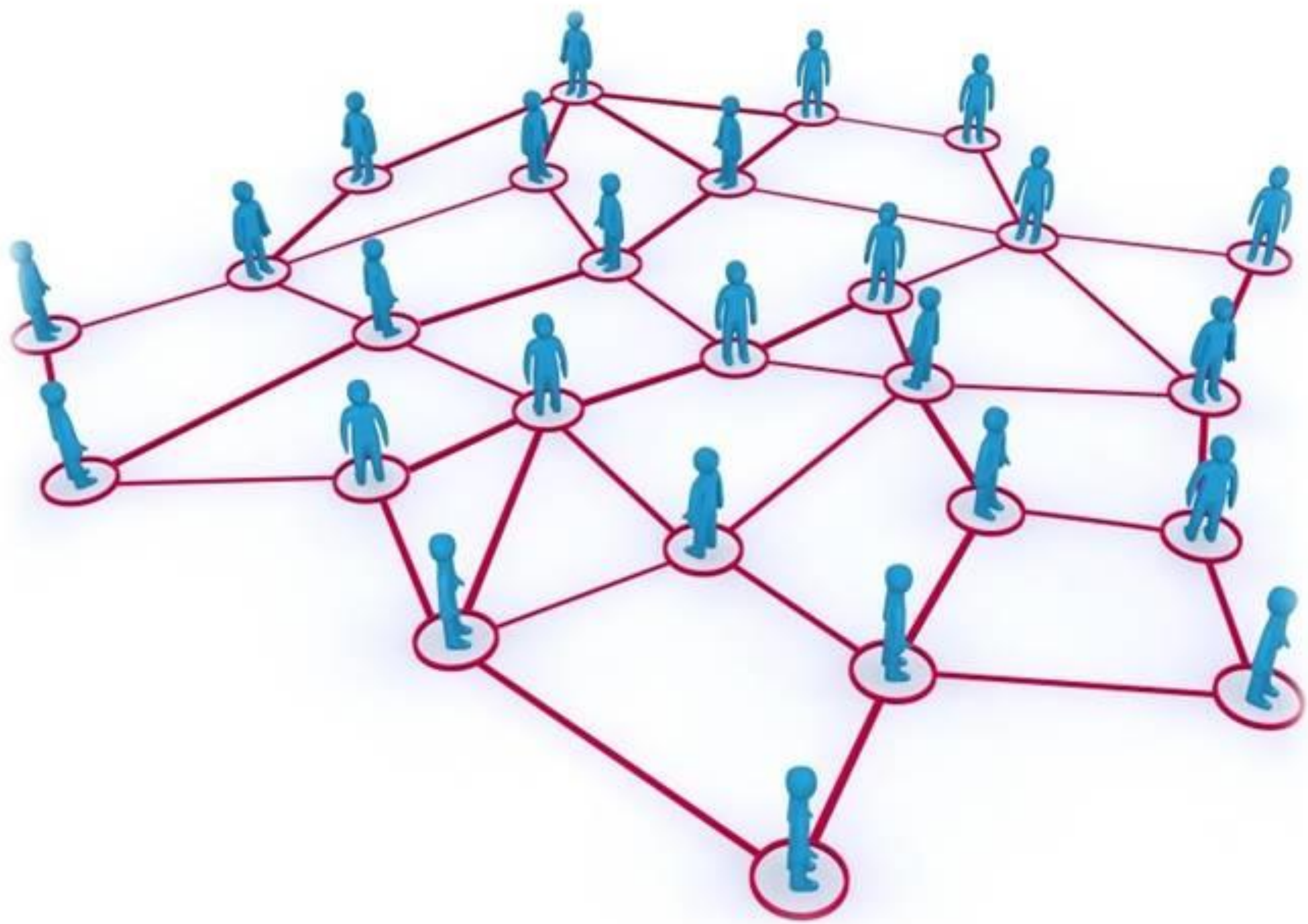
Lecture 9

Graphs and Spanning Trees

Overview

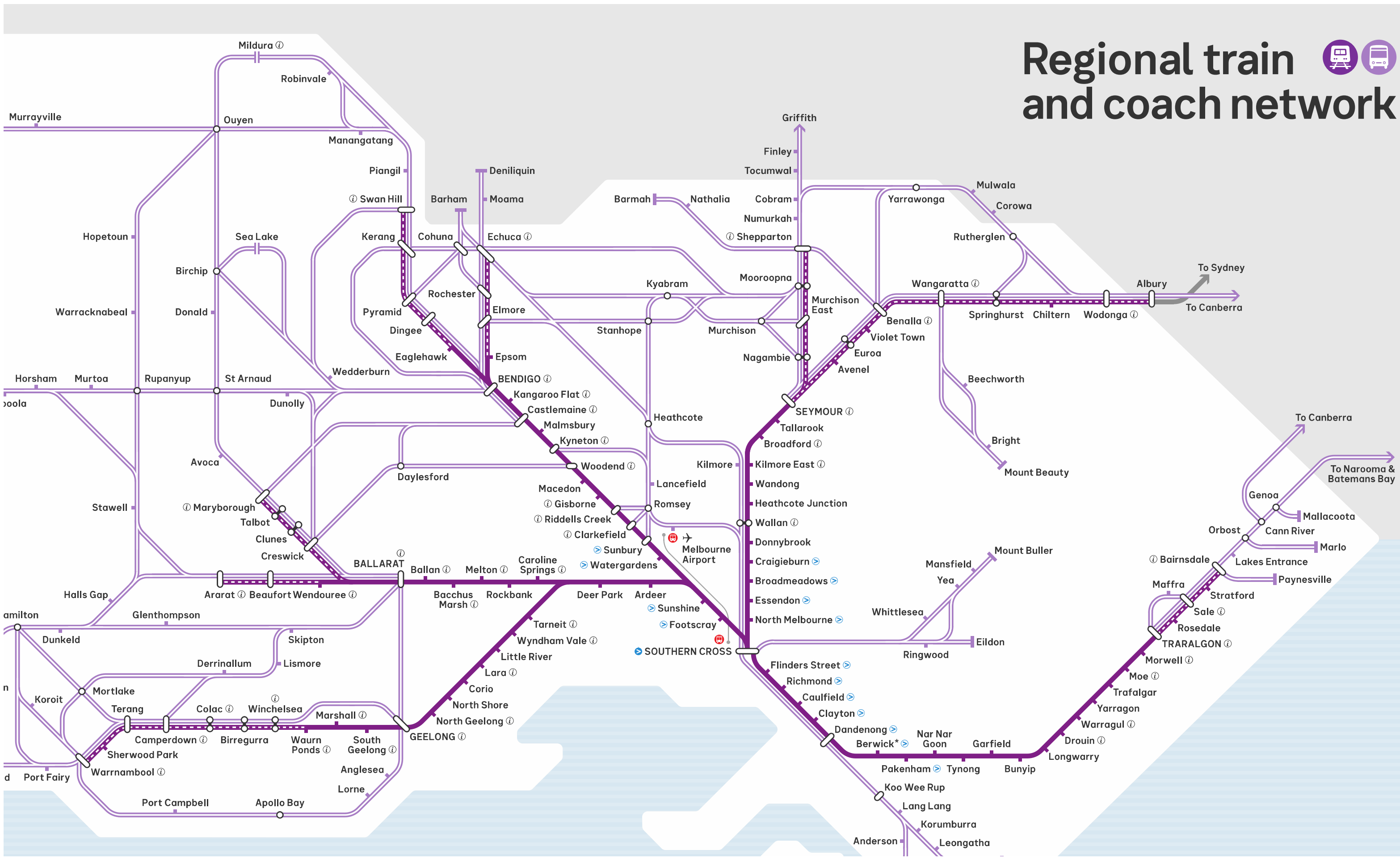
- Graphs
- Trees and Spanning Trees
- Prims algorithm (simplified)
- Problem decomposition

How to represent relational data?

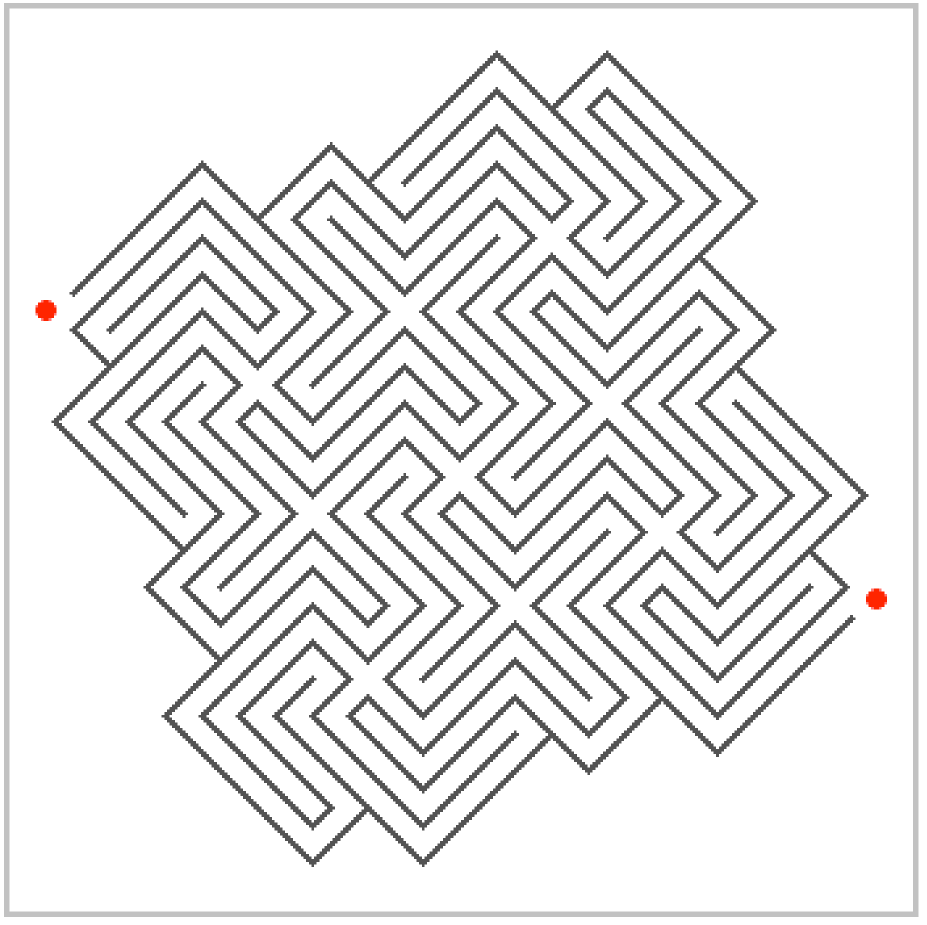


MONASH College

How to represent relational data?



How to represent relational data?

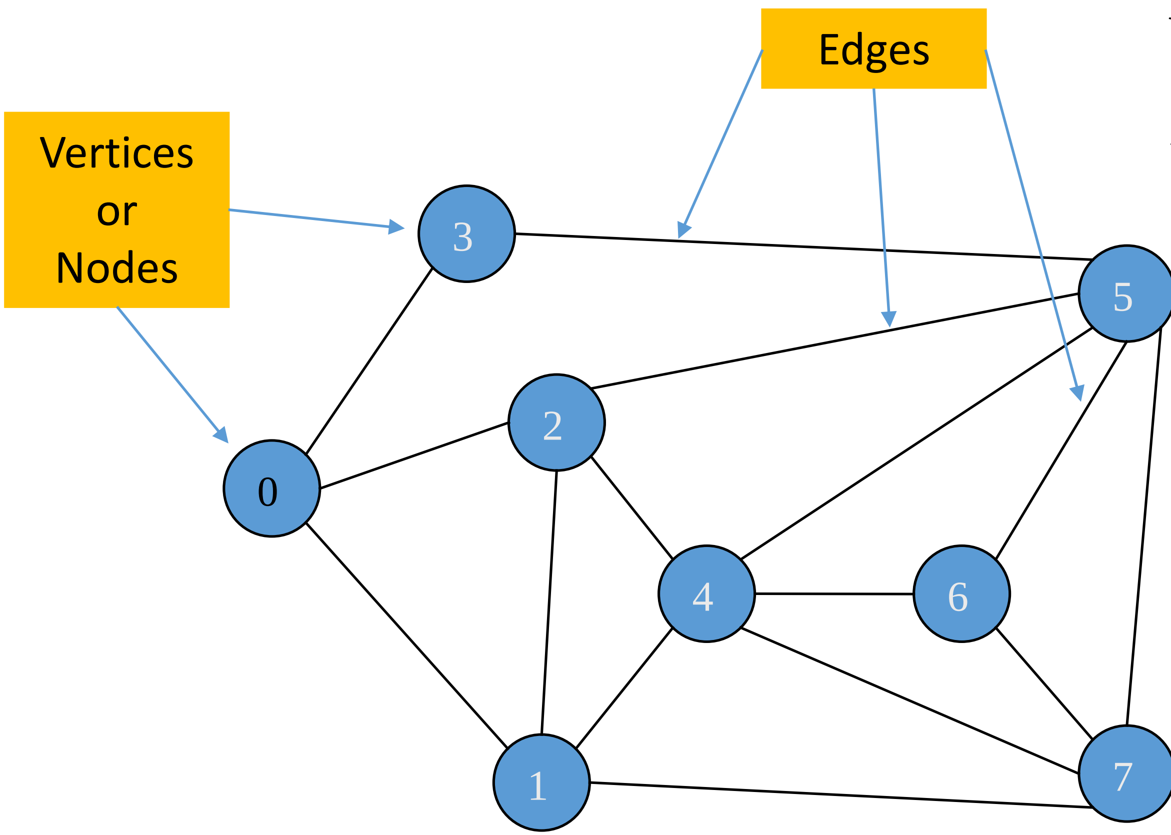


Graphs

A graph *G*=(*V*,*E*)is an ordered pair consisting of a set of vertices and a set of edges.

Definition:

Graph $G = (V, E)$ is a set of *vertices* $V = \{v_1, \ldots, v_n\}$ *and* collection of *edges* $E = \{e_1, \ldots, e_m\}$ where $e_i = (v, w)$, $v, w \in V$

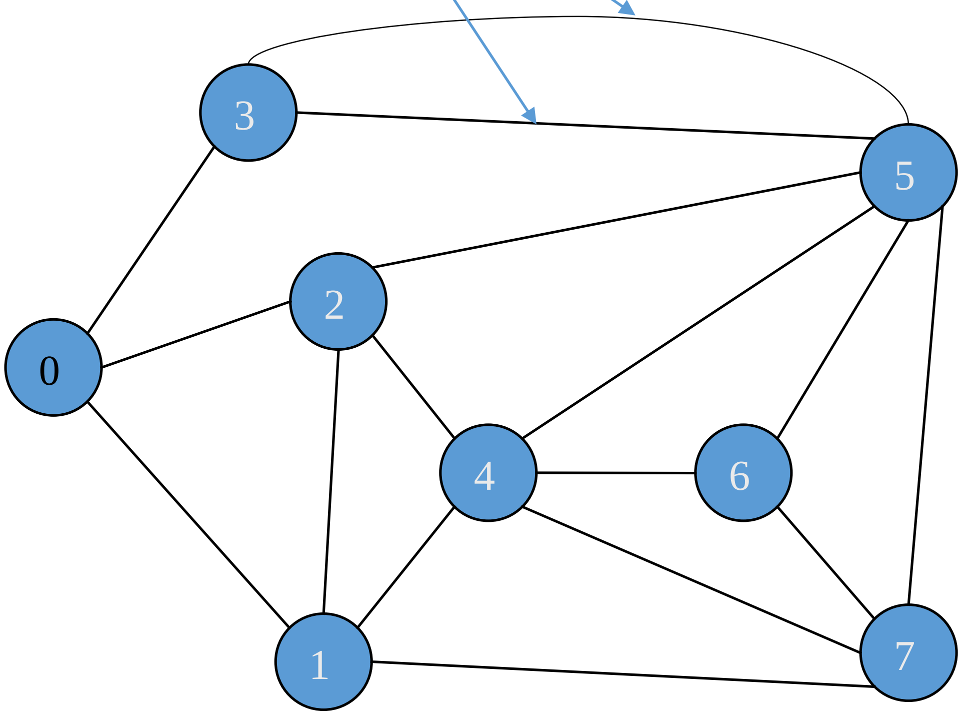


$$V = \{0, 1, \ldots, 7\}$$

$$E = \{(0,1), (0,2), (0,3), (1,2), (1,4), (1,7), (2,4), (2,5), (3,5), (4,5), (4,6), (4,7), (5,6), (5,7), (6,7)\}$$

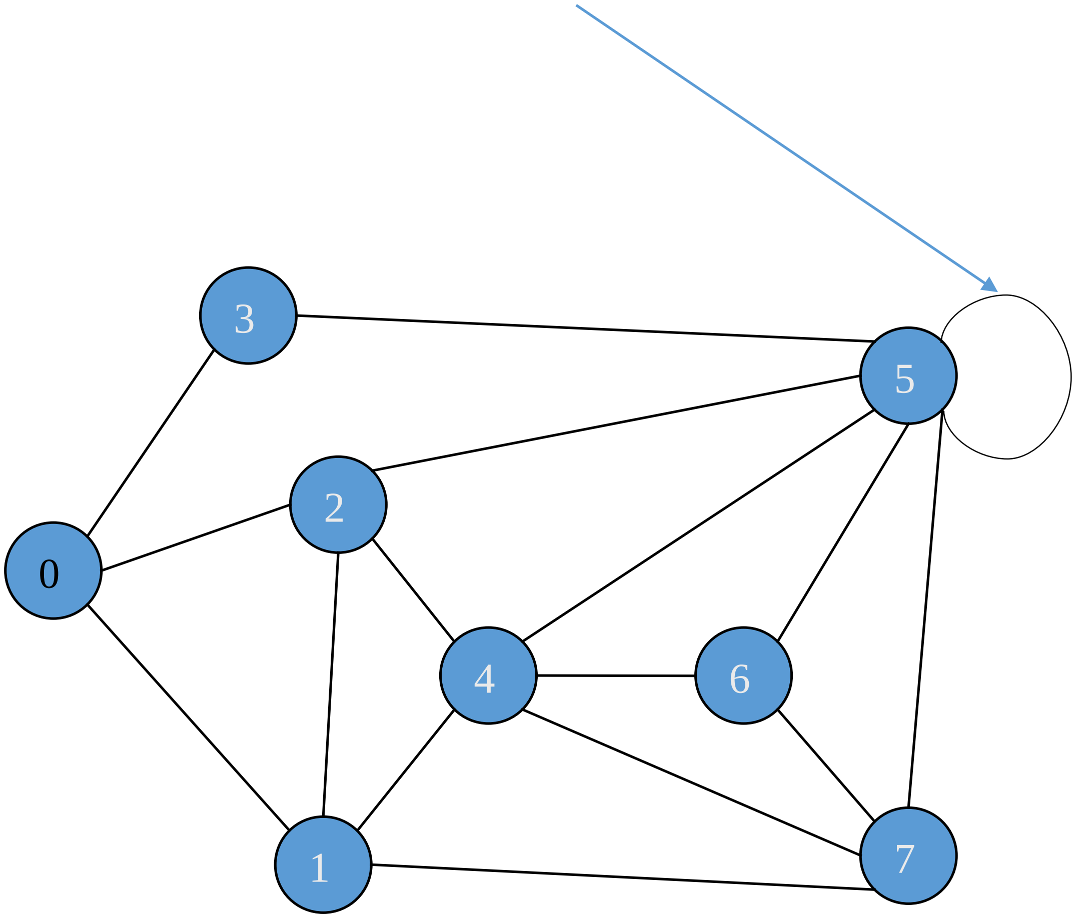
Graphs

- A graph is a multigraph if it has multiple edges joining the same pair of vertices



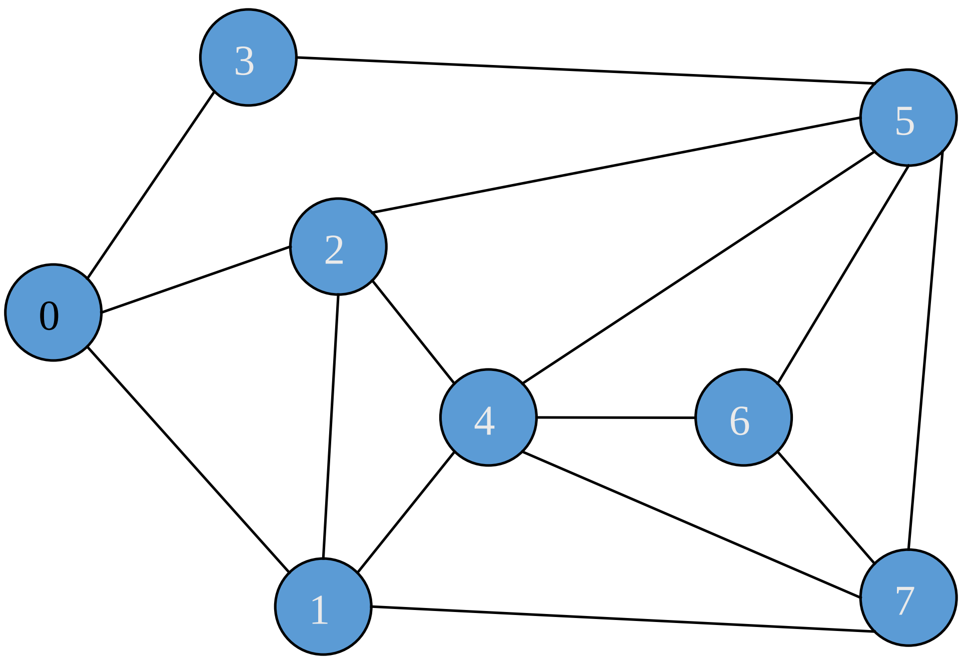
Graphs

- A loop is an edge where both end vertices are the same



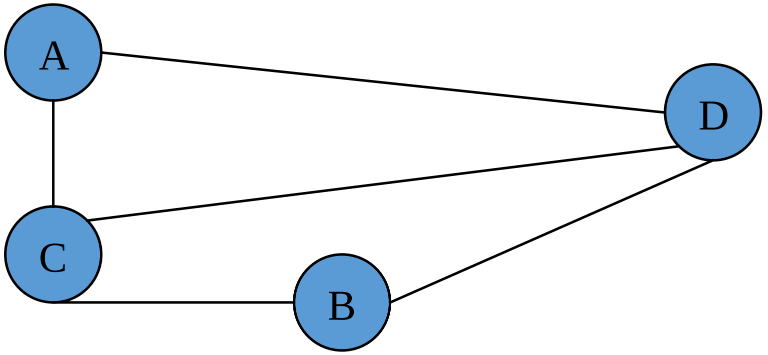
Graphs

- A graph is simple if it has no loops and no multiple edges



Graphs

- Two vertices are adjacent if they are connected by an edge
- The degree of a vertex in a simple graph is the number of edges incident to the vertex
- Vertex D has degree 3 and Vertex B has degree 2

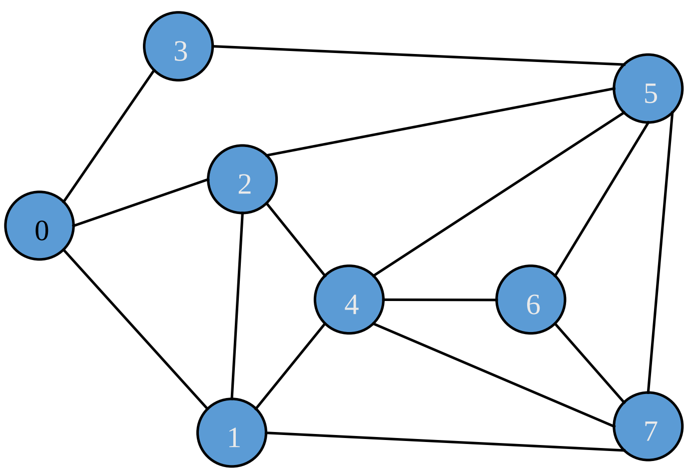


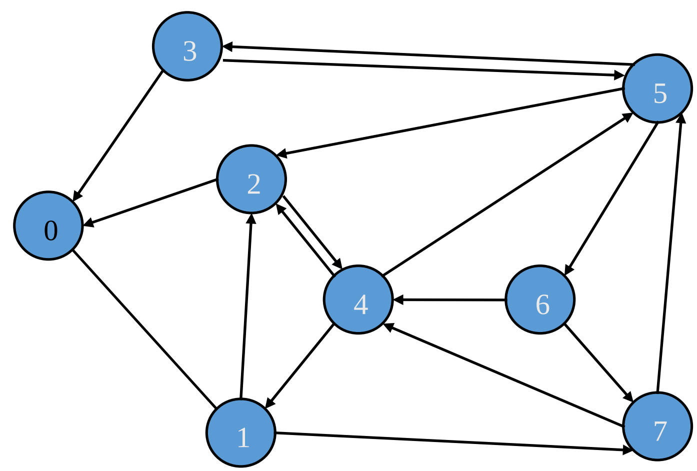
Graphs

Undirected

VS

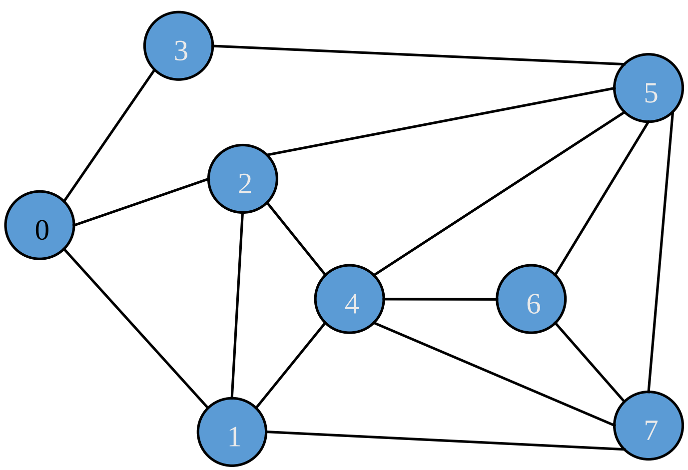
Directed

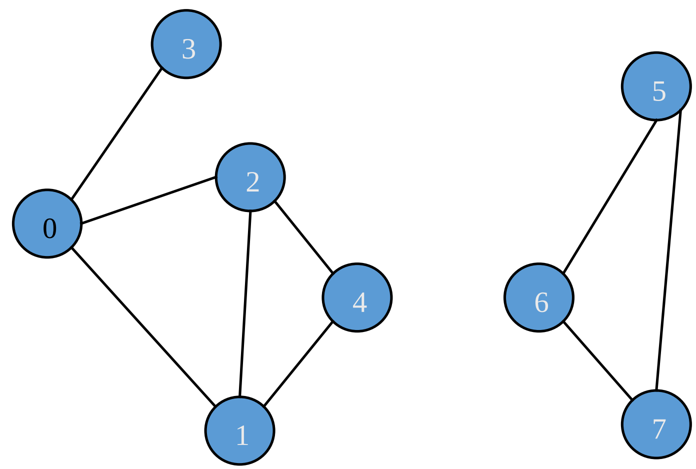




Graphs

Connected VS Unconnected



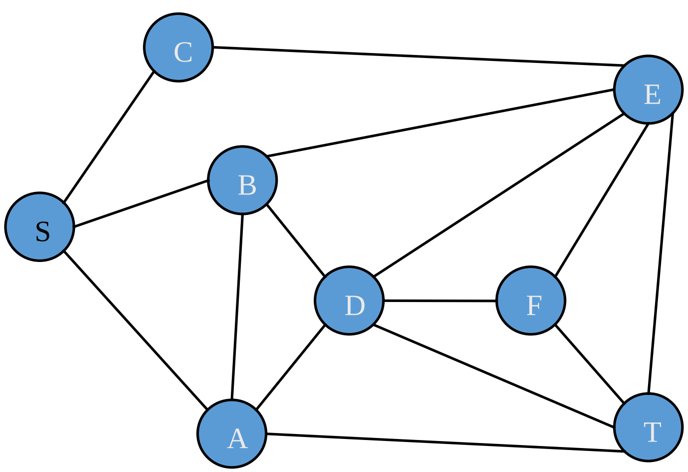


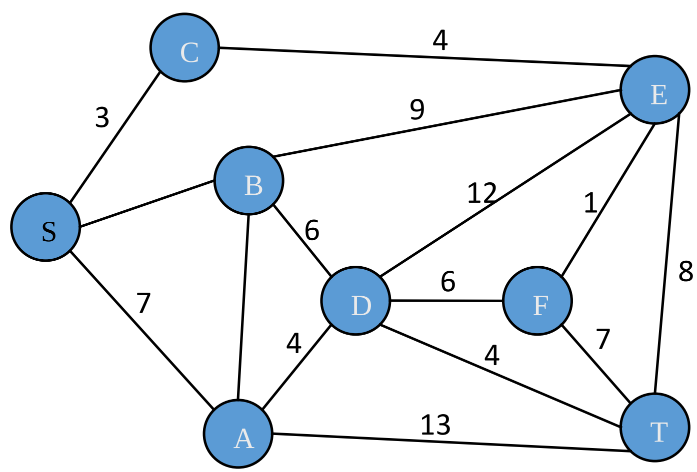
Graphs

Unweighted

VS

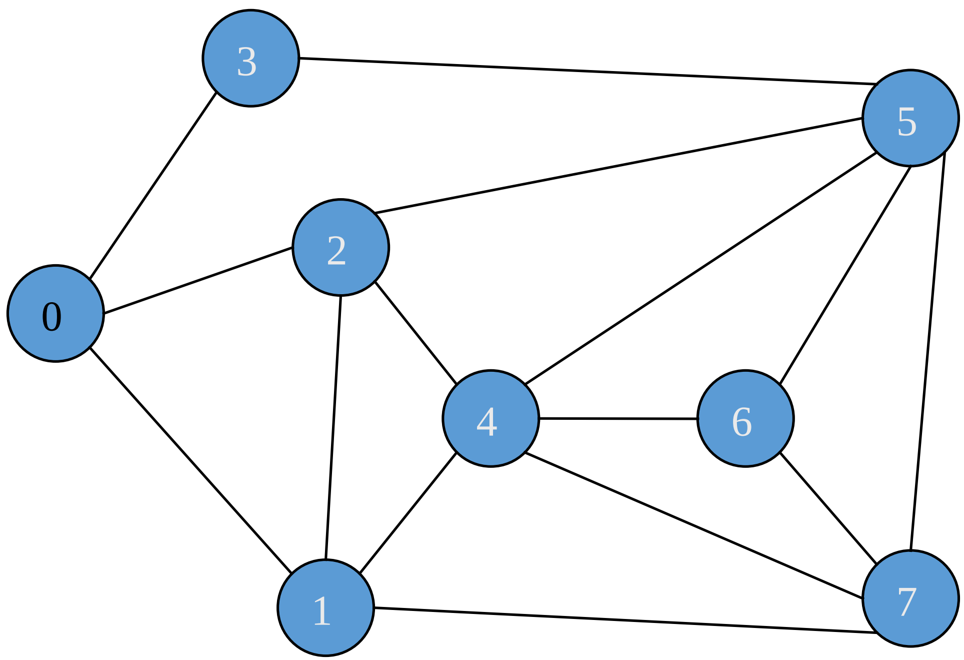
Weighted





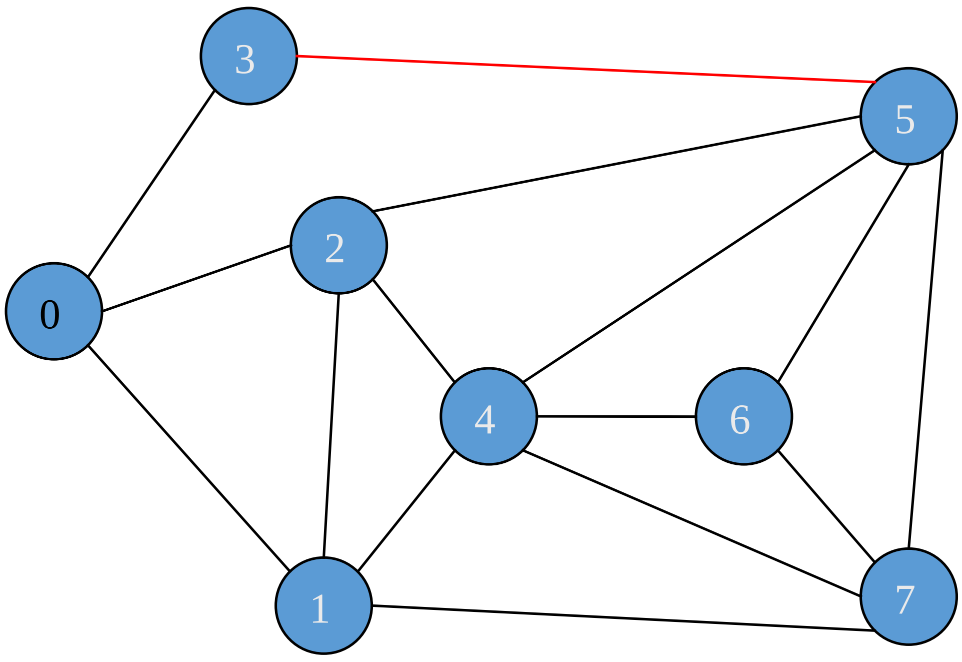
Trail

- A walk with distinct (no repeated) edges.



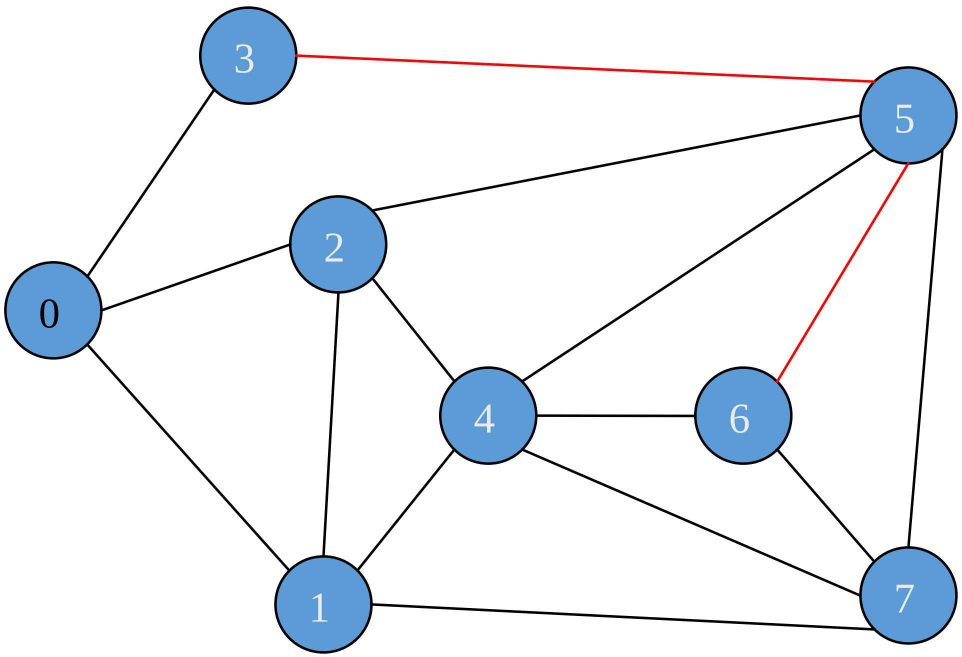
Trail

- A walk with distinct (no repeated) edges.



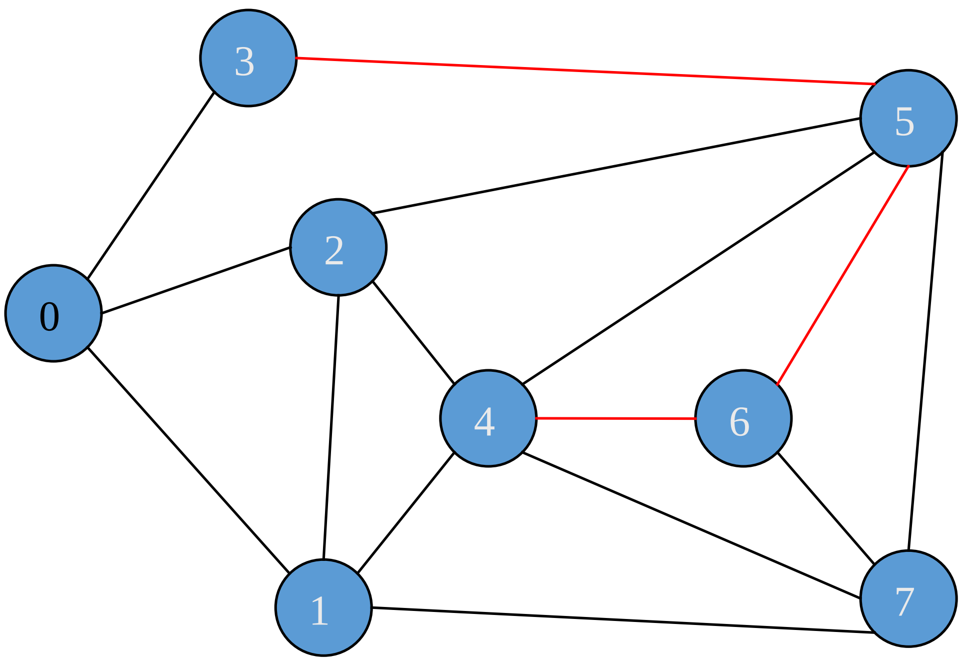
Trail

- A walk with distinct (no repeated) edges.



Trail

- A walk with distinct (no repeated) edges.

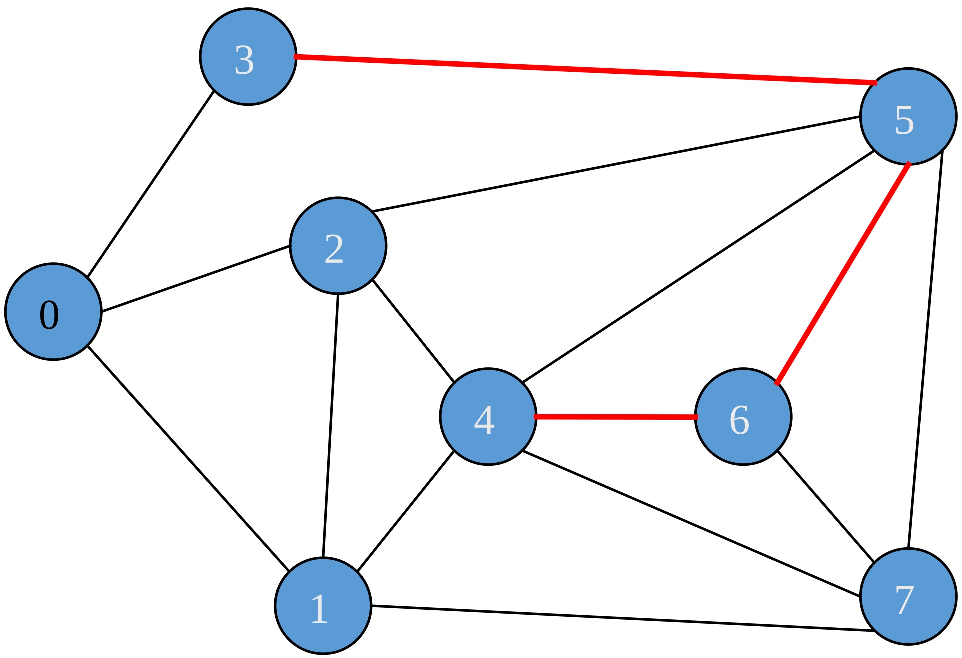


MONASH College

Trail

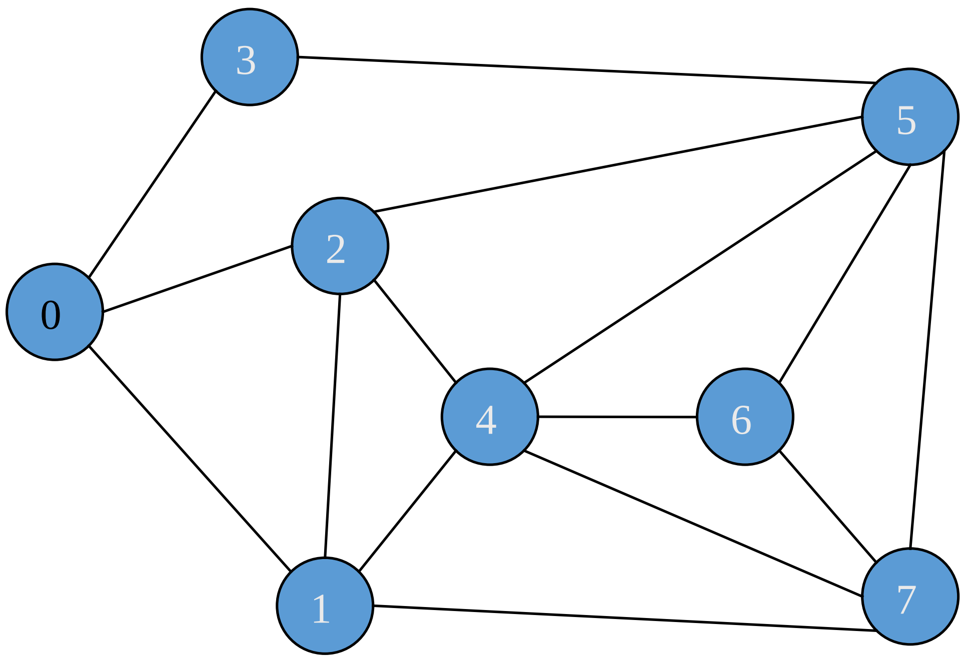
- A walk with distinct (no repeated) edges.

  Eg. (3,5),(5,6),(6,4) or (3,5,6,4)



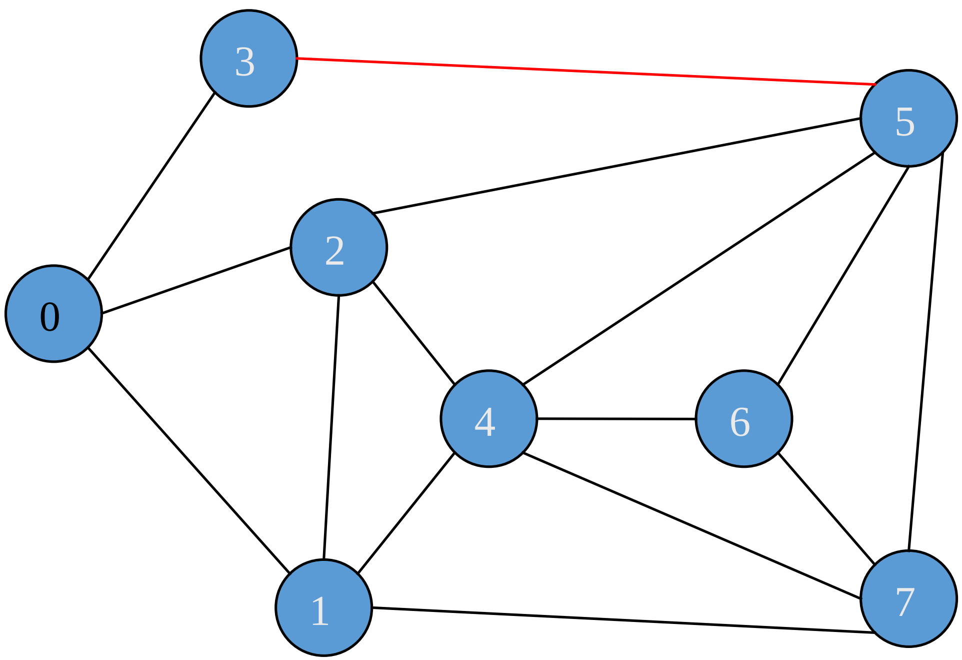
Circuit

- A circuit is a trail that begins and ends on the same vertex



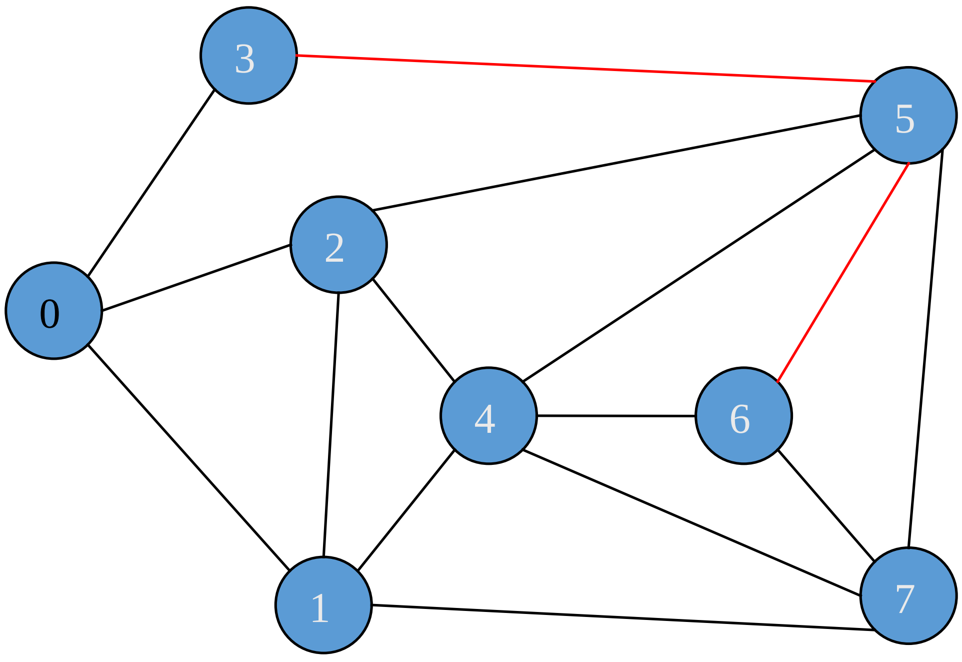
Circuit

- A circuit is a trail that begins and ends on the same vertex



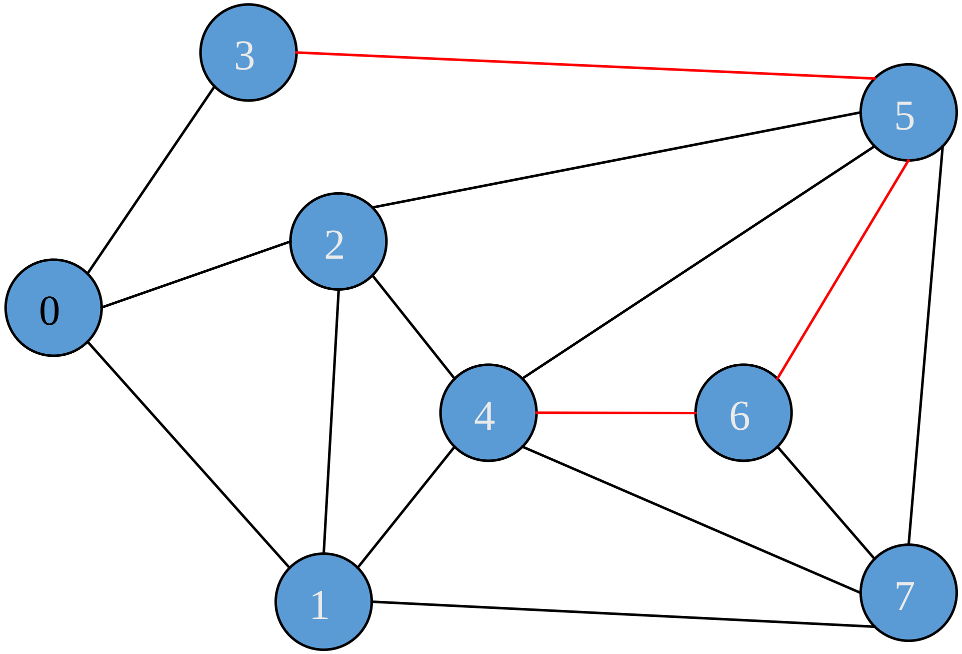
Circuit

- A circuit is a trail that begins and ends on the same vertex



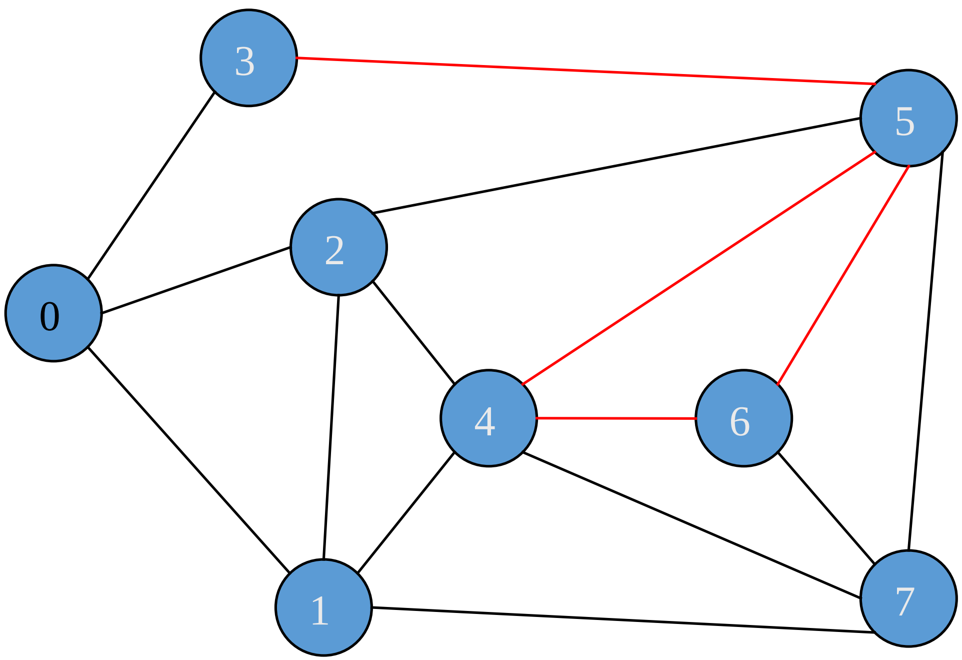
Circuit

- A circuit is a trail that begins and ends on the same vertex



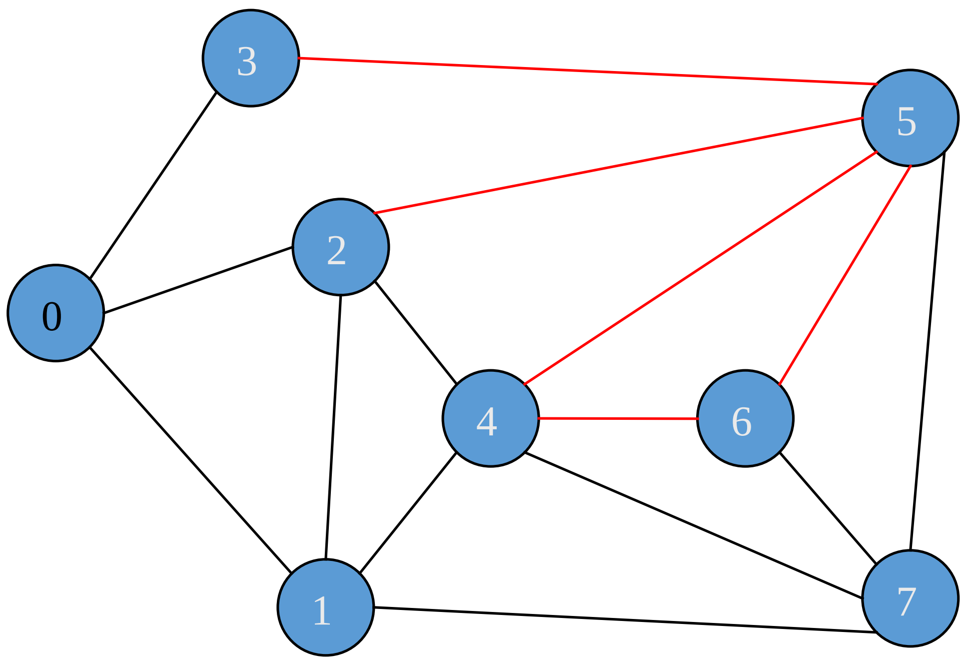
Circuit

- A circuit is a trail that begins and ends on the same vertex



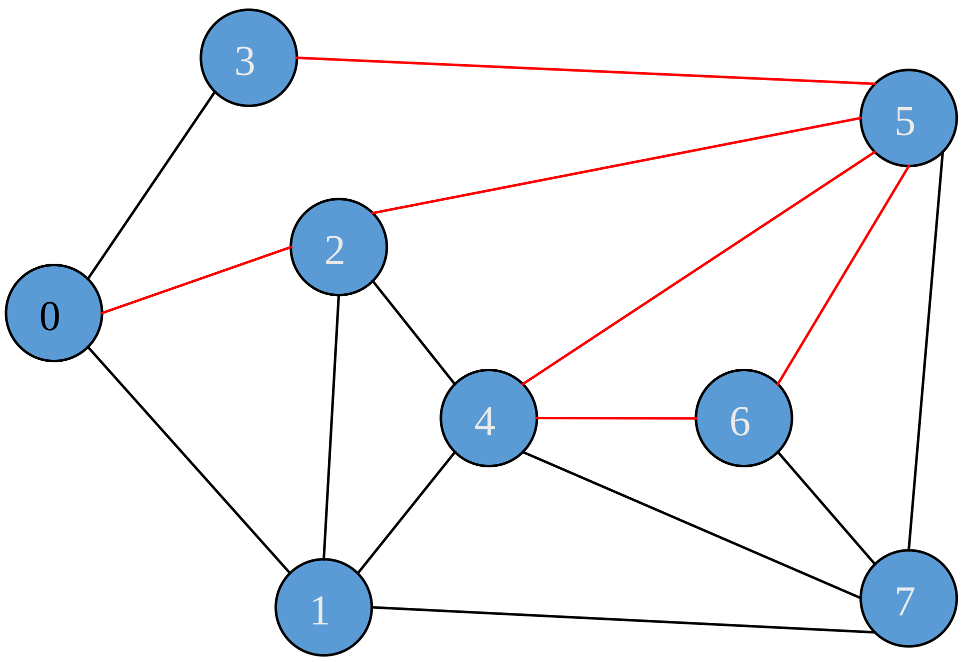
Circuit

- A circuit is a trail that begins and ends on the same vertex



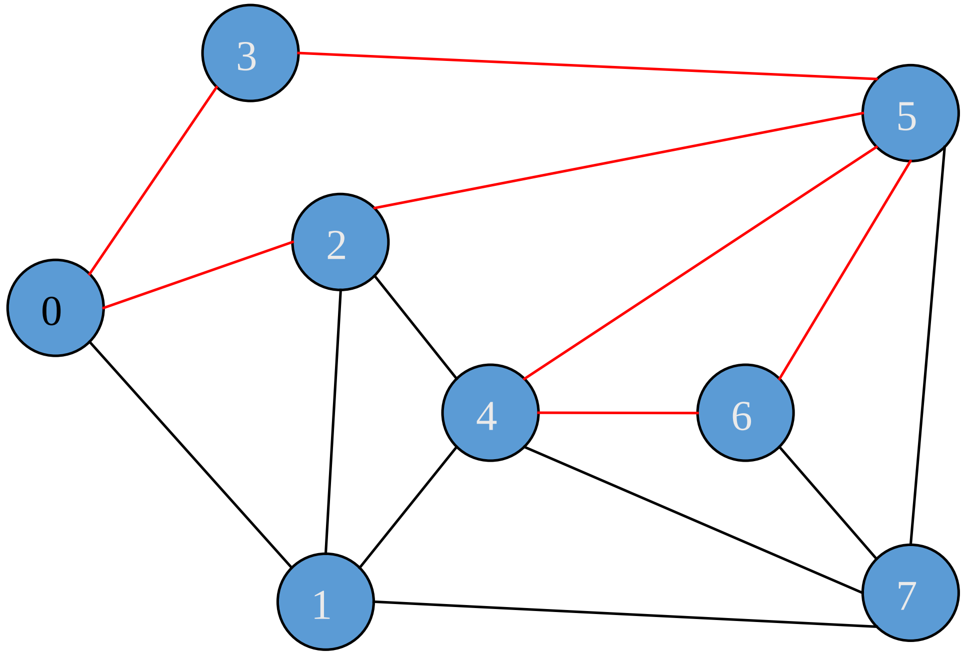
Circuit

- A circuit is a trail that begins and ends on the same vertex



Circuit

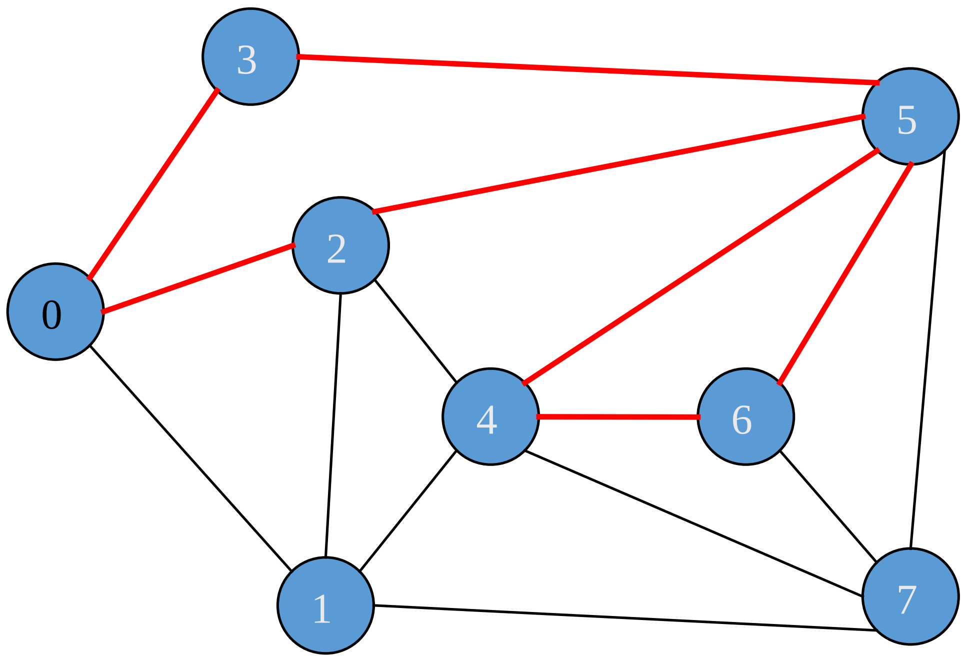
- A circuit is a trail that begins and ends on the same vertex



Circuit

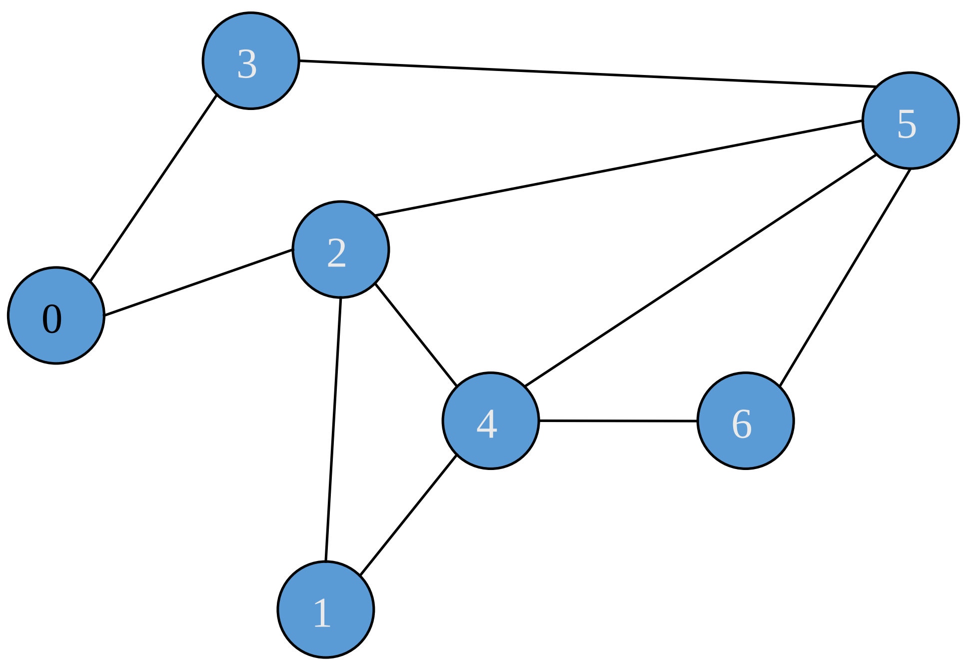
- A circuit is a trail that begins and ends on the same vertex

  Eg. (3,5),(5,6),(6,4),(4,5),(5,2),(2,0),(0,3)
  or (3,5,6,4,5,2,0,3)



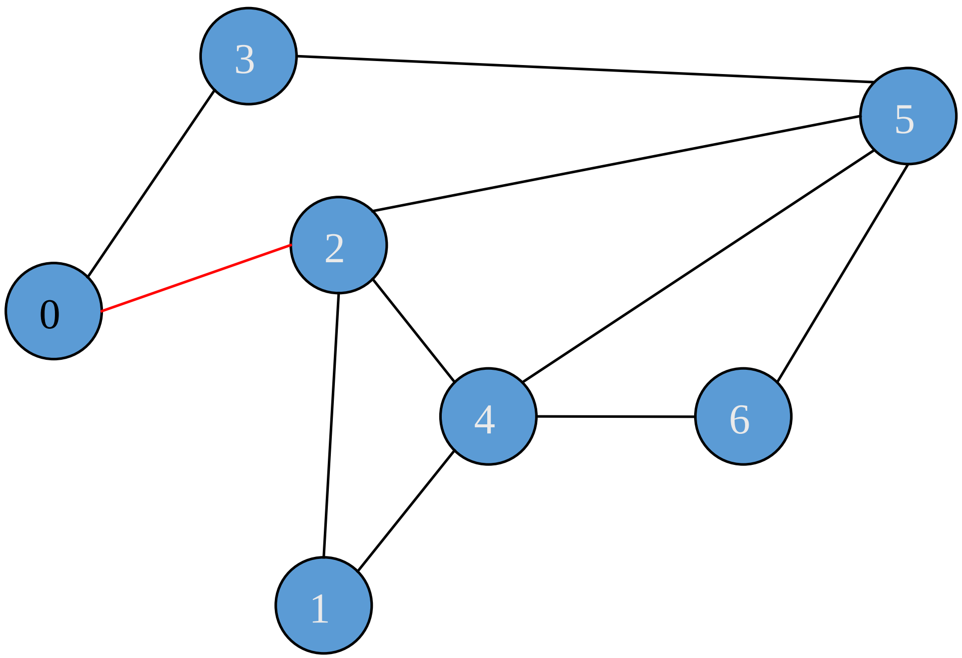
Eularian Circuit

- A connected graph is called Eulerian if you can find a trail which starts and ends at the same vertex and contains each edge exactly once. Such a trail is called an Eulerian circuit.



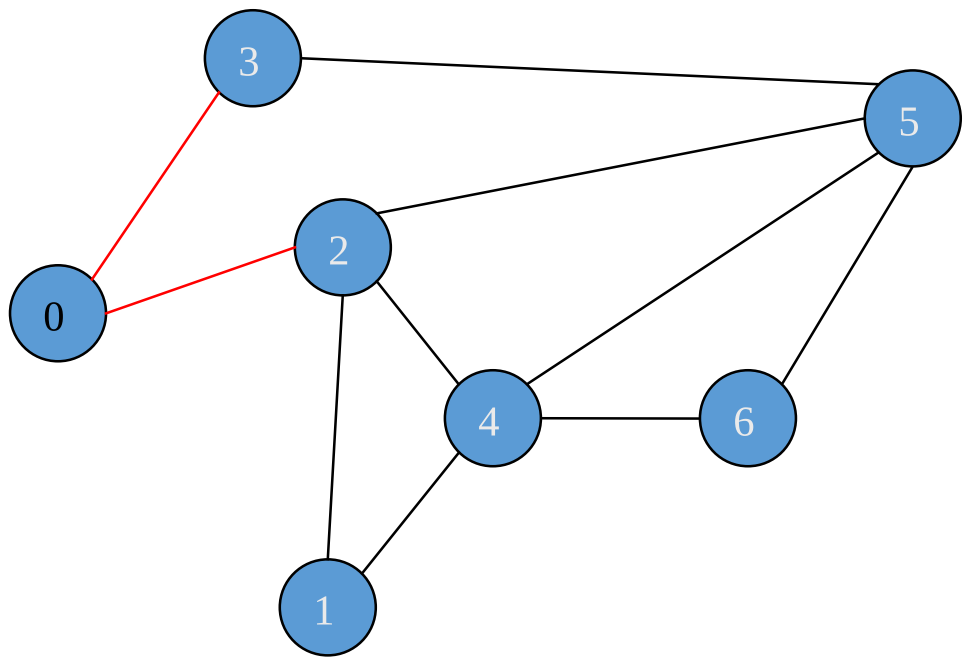
Eulerian Circuit

- A connected graph is called Eulerian if you can find a trail which starts and ends at the same vertex and contains each edge exactly once. Such a trail is called an Eulerian circuit.



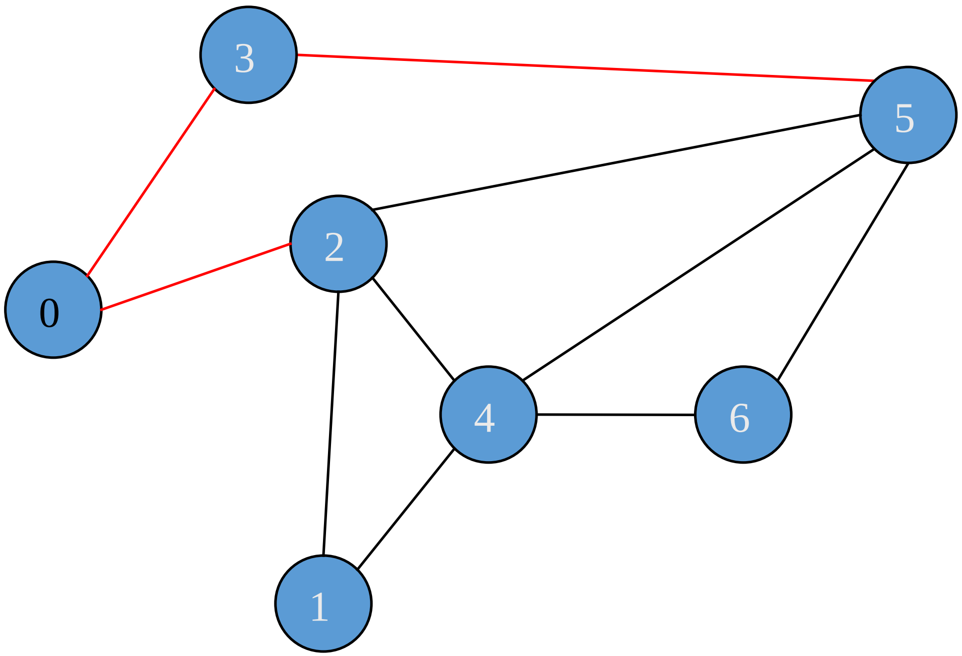
Eulerian Circuit

- A connected graph is called Eulerian if you can find a trail which starts and ends at the same vertex and contains each edge exactly once. Such a trail is called an Eulerian circuit.



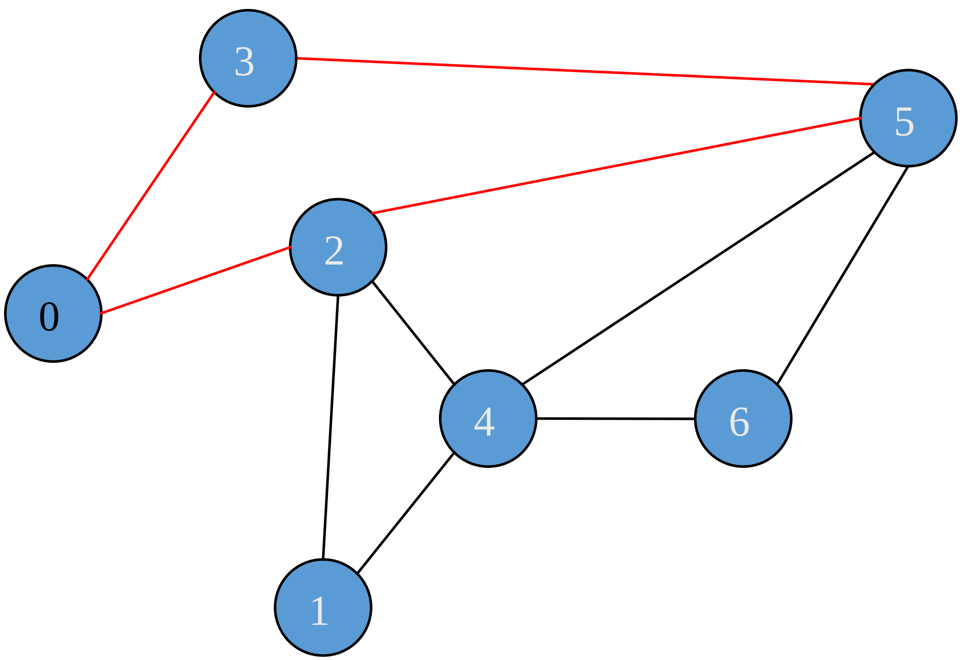
Eulerian Circuit

- A connected graph is called Eulerian if you can find a trail which starts and ends at the same vertex and contains each edge exactly once. Such a trail is called an Eulerian circuit.



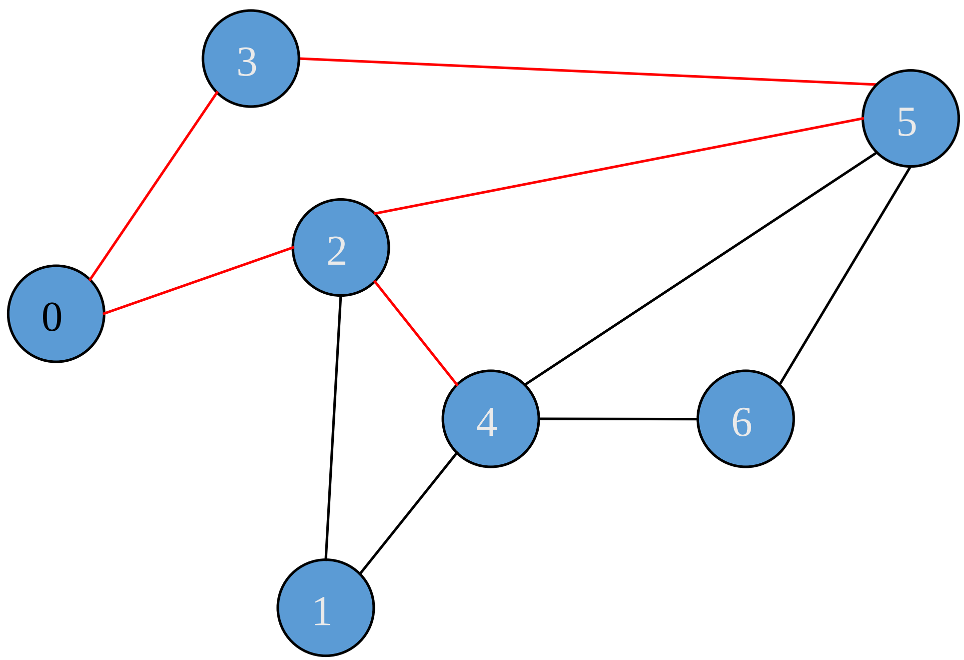
Eulerian Circuit

- A connected graph is called Eulerian if you can find a trail which starts and ends at the same vertex and contains each edge exactly once. Such a trail is called an Eulerian circuit.



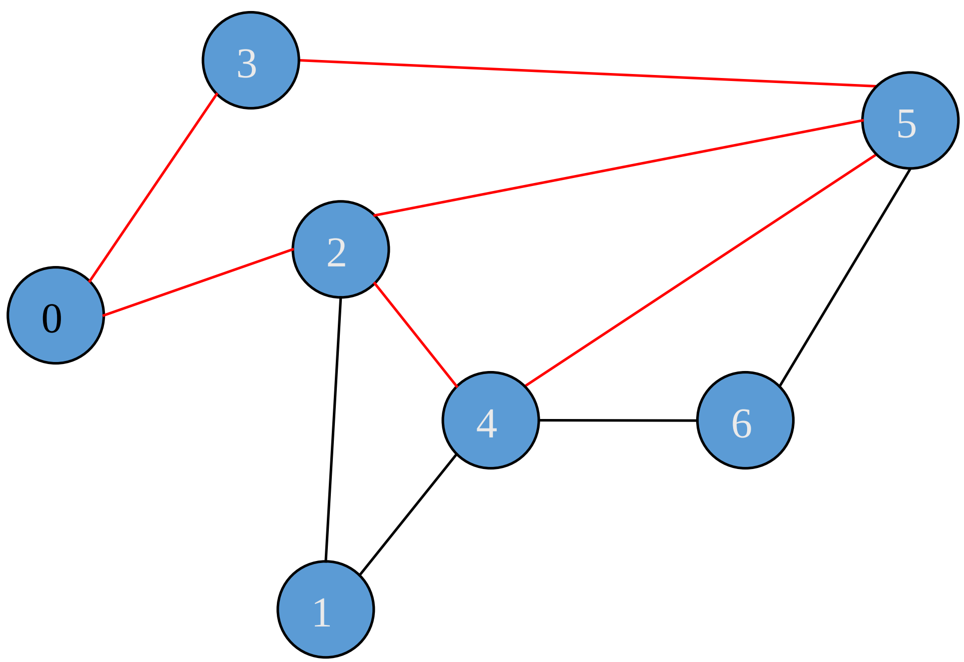
Eulerian Circuit

- A connected graph is called Eulerian if you can find a trail which starts and ends at the same vertex and contains each edge exactly once. Such a trail is called an Eulerian circuit.



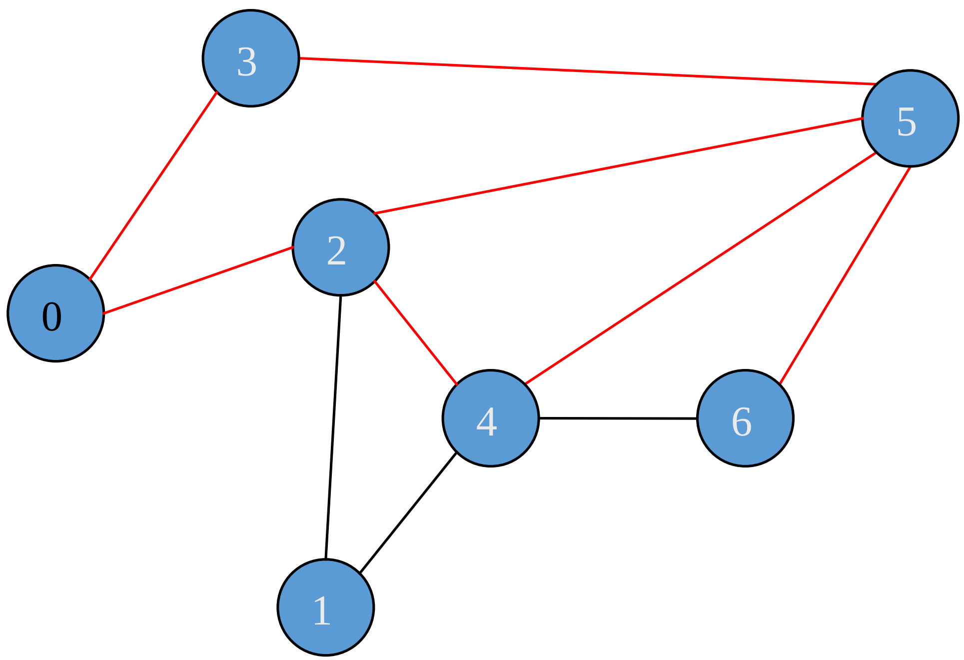
Eulerian Circuit

- A connected graph is called Eulerian if you can find a trail which starts and ends at the same vertex and contains each edge exactly once. Such a trail is called an Eulerian circuit.



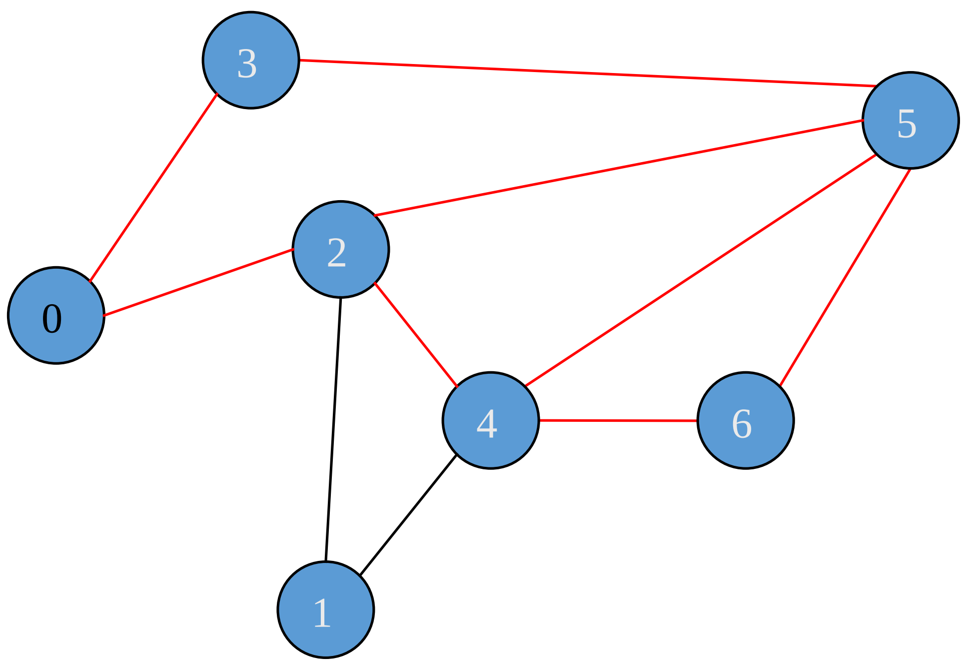
Eulerian Circuit

- A connected graph is called Eulerian if you can find a trail which starts and ends at the same vertex and contains each edge exactly once. Such a trail is called an Eulerian circuit.



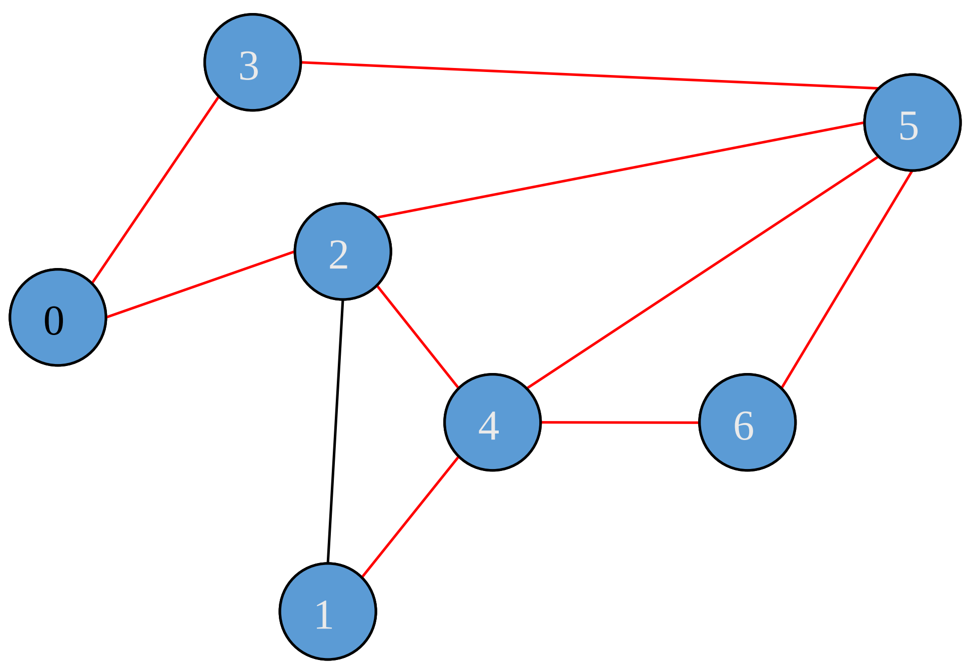
Eulerian Circuit

- A connected graph is called Eulerian if you can find a trail which starts and ends at the same vertex and contains each edge exactly once. Such a trail is called an Eulerian circuit.



Eulerian Circuit

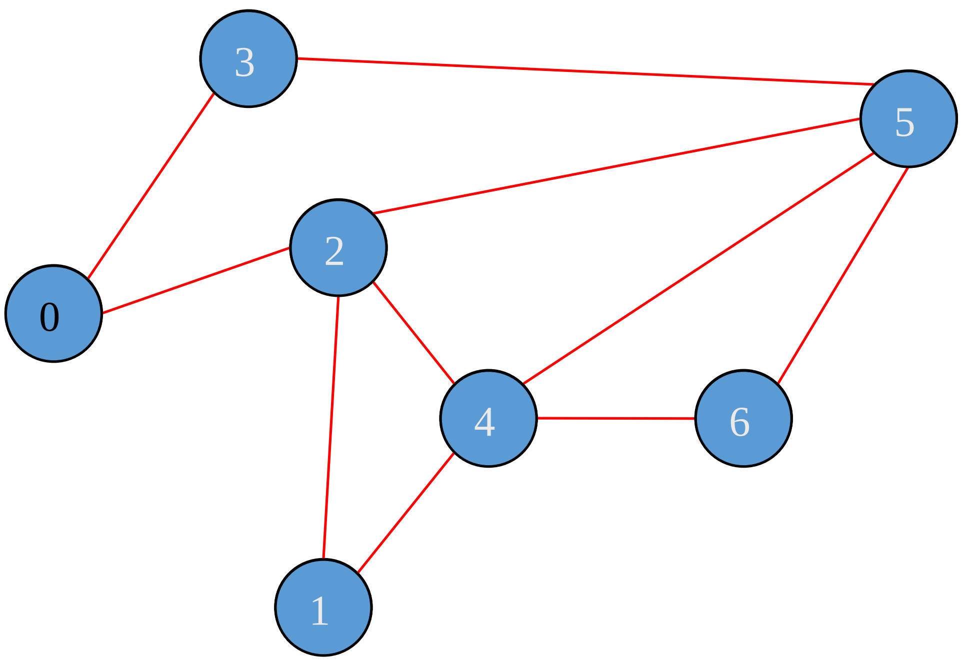
- A connected graph is called Eulerian if you can find a trail which starts and ends at the same vertex and contains each edge exactly once. Such a trail is called an Eulerian circuit.



Eulerian Circuit

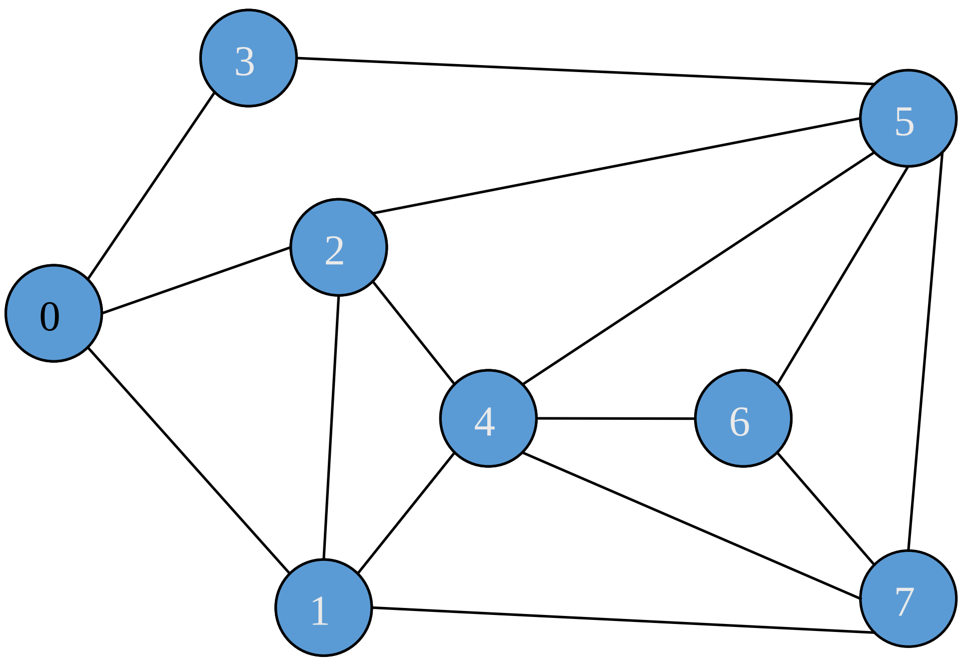
- A connected graph is called Eulerian if you can find a trail which starts and ends at the same vertex and contains each edge exactly once. Such a trail is called an Eulerian circuit.

Eg. (2,0,3,5,2,4,5,6,4,1,2)



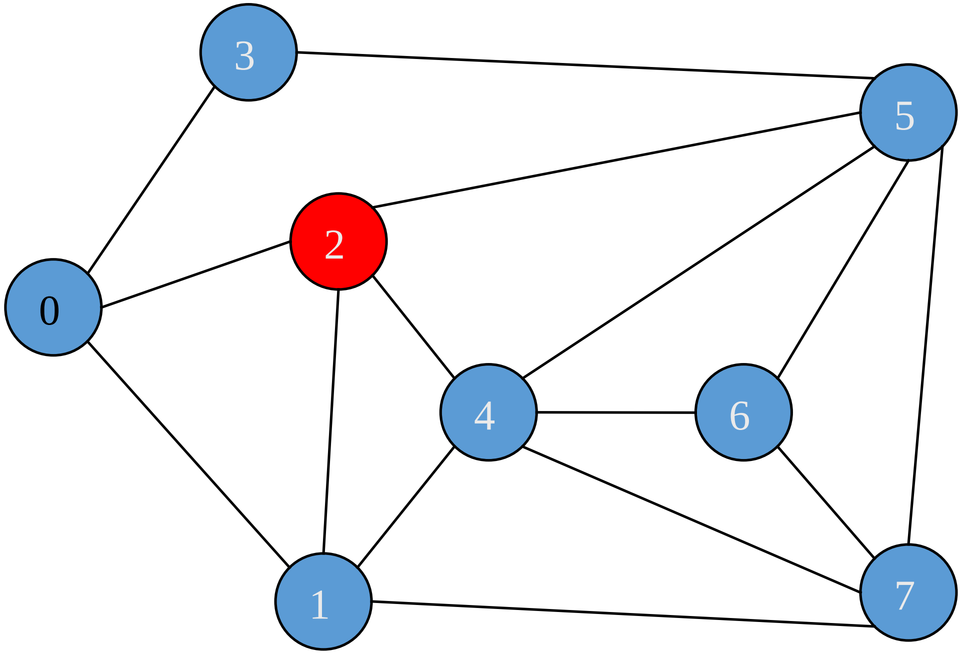
Path

- Open walk
- A walk with no repeated vertices (and therefore no repeated edges).



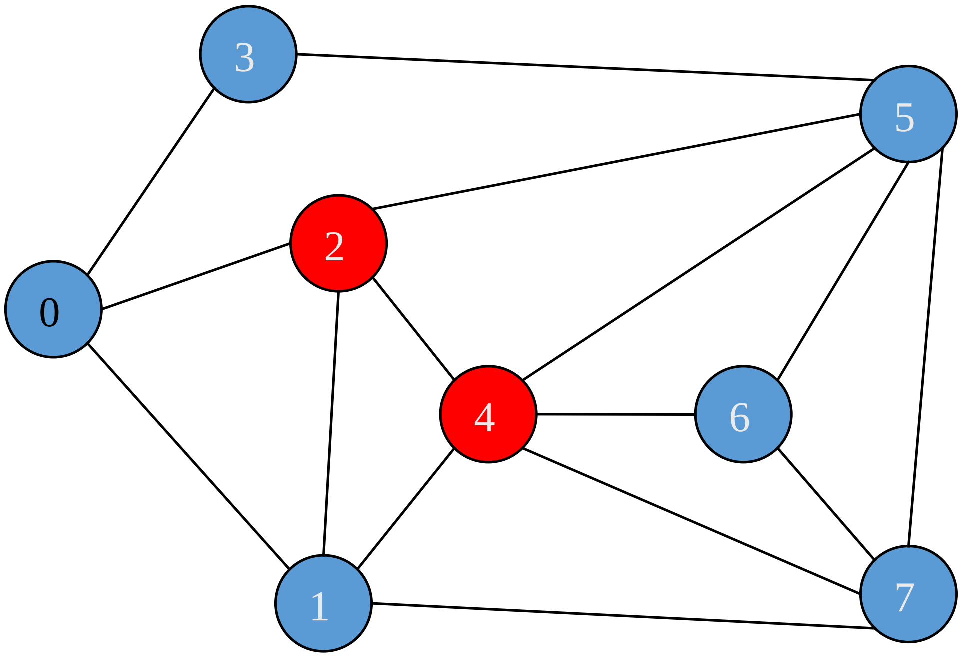
Path

- Open walk
- A walk with no repeated vertices (and therefore no repeated edges).



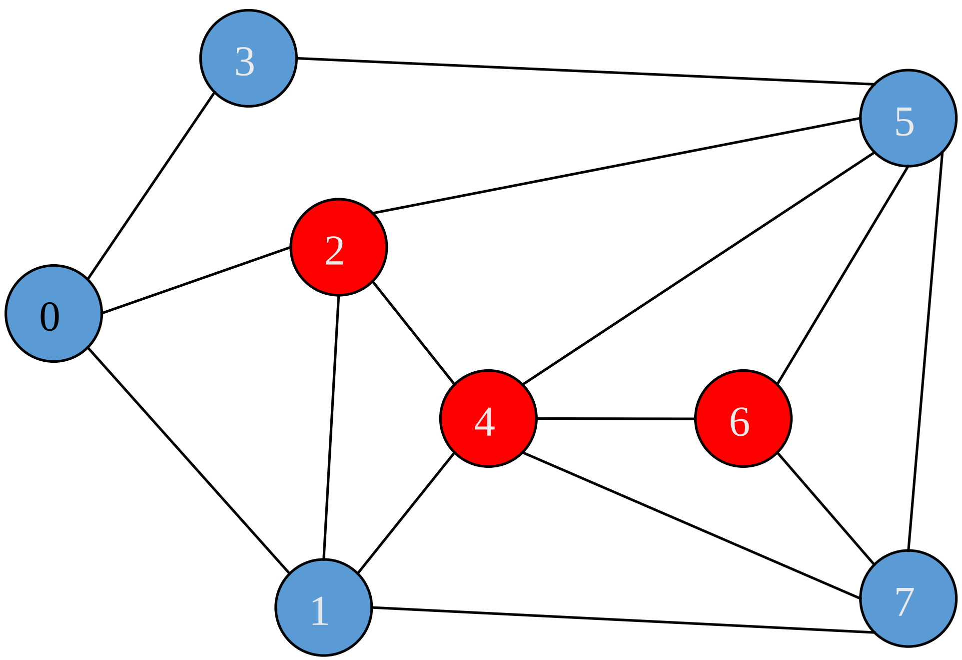
Path

- Open walk
- A walk with no repeated vertices (and therefore no repeated edges).



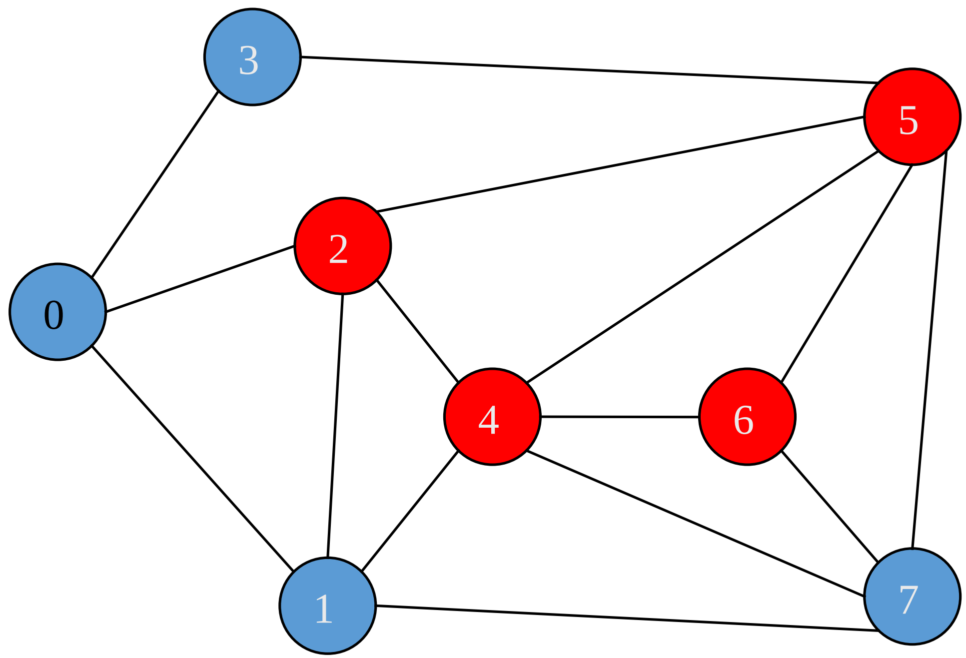
Path

- Open walk
- A walk with no repeated vertices (and therefore no repeated edges).



Path

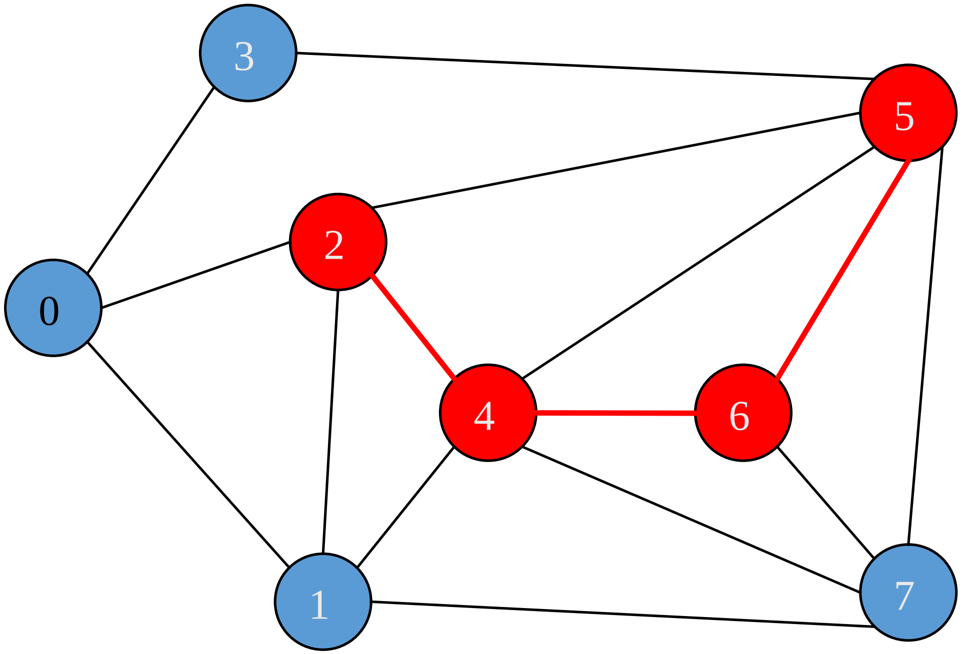
- Open walk
- A walk with no repeated vertices (and therefore no repeated edges).



Path

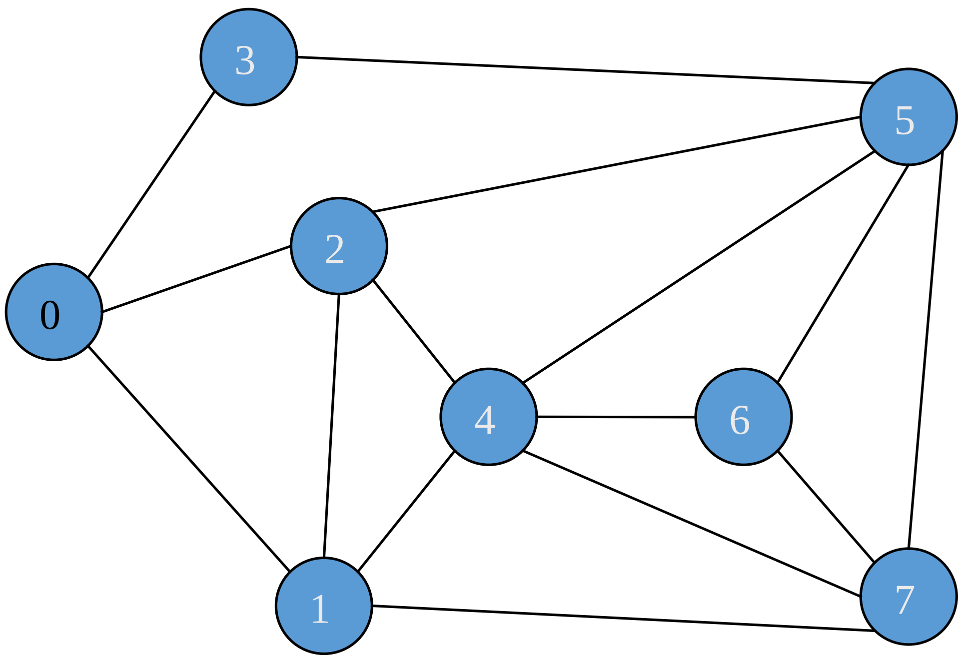
- Open walk
- A walk with no repeated vertices (and therefore no repeated edges).

Eg. (2,4,6,5)



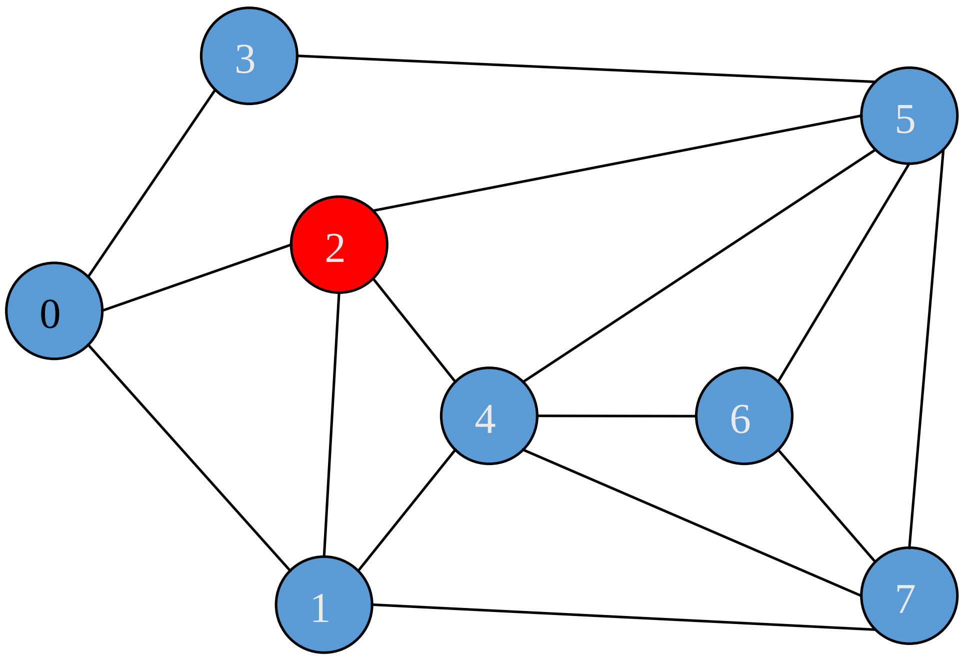
Hamiltonian Path

- A Hamiltonian path visits every vertex in the graph exactly once but it does not need to start and end at the same vertex.



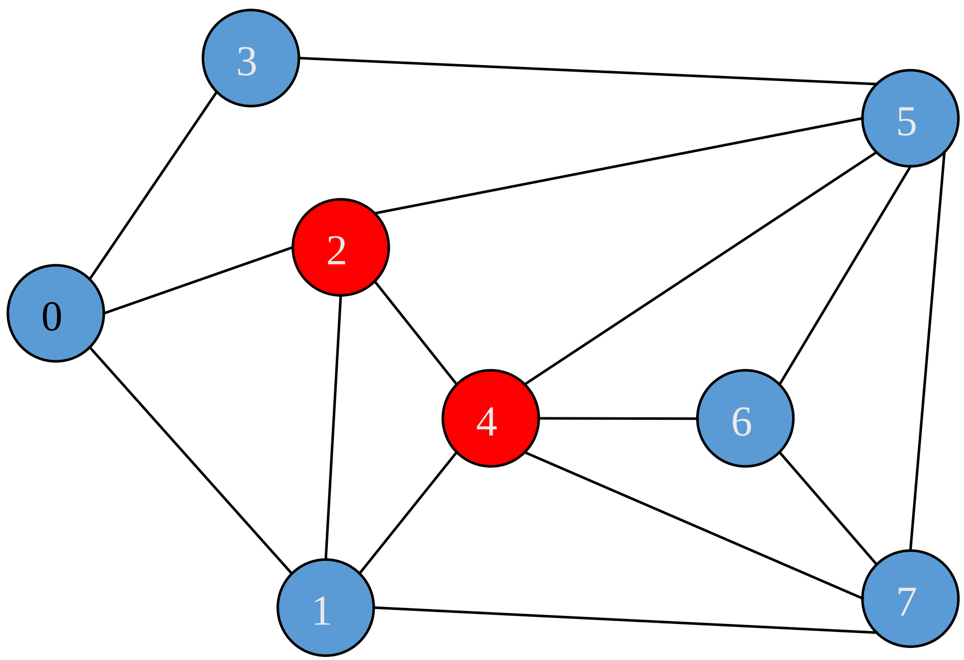
Hamiltonian Path

- A Hamiltonian path visits every vertex in the graph exactly once but it does not need to start and end at the same vertex.



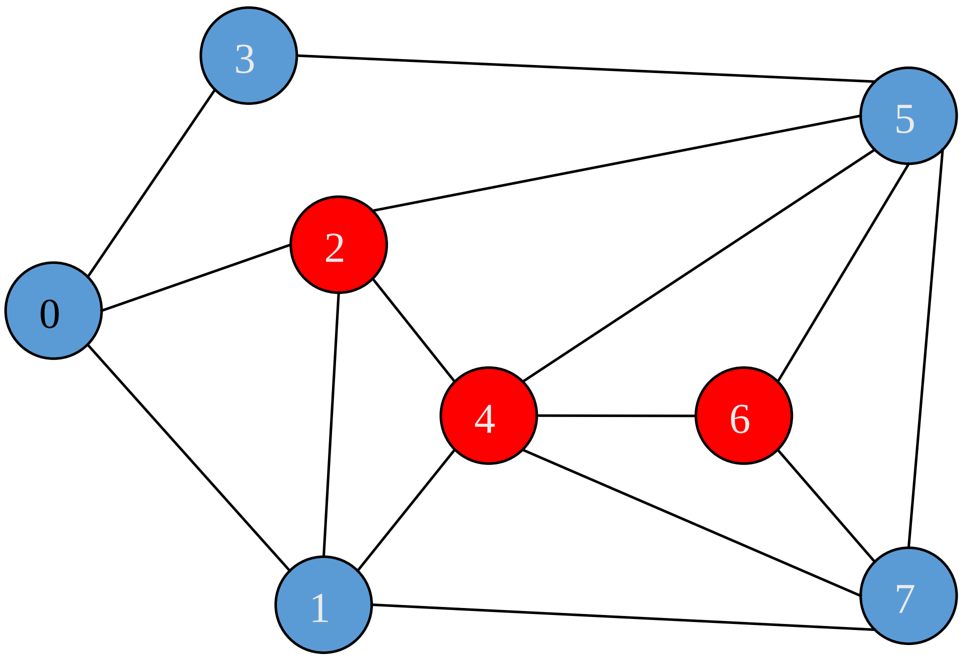
Hamiltonian Path

- A Hamiltonian path visits every vertex in the graph exactly once but it does not need to start and end at the same vertex.



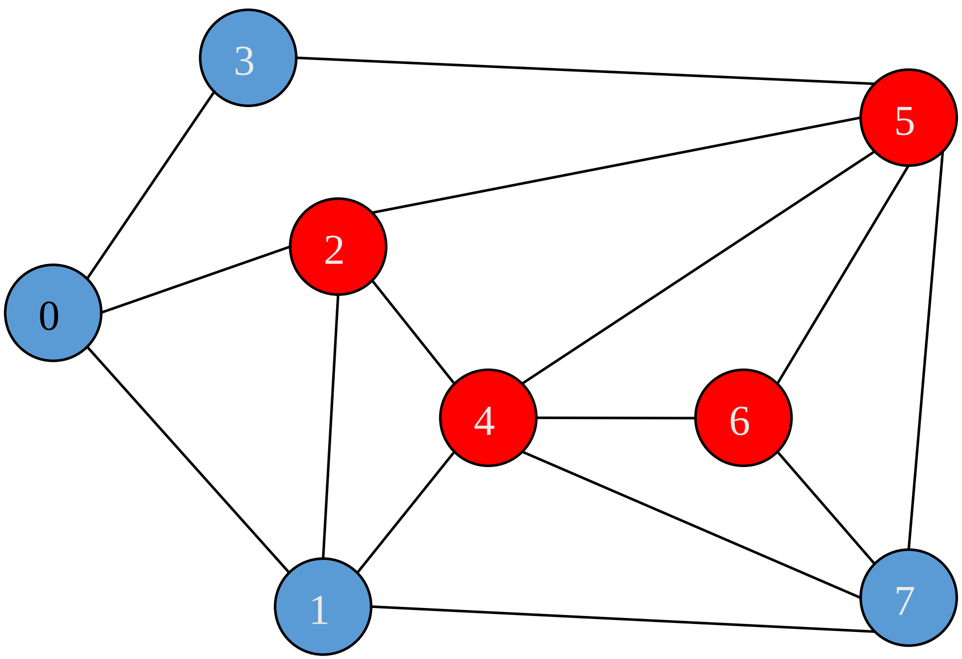
Hamiltonian Path

- A Hamiltonian path visits every vertex in the graph exactly once but it does not need to start and end at the same vertex.



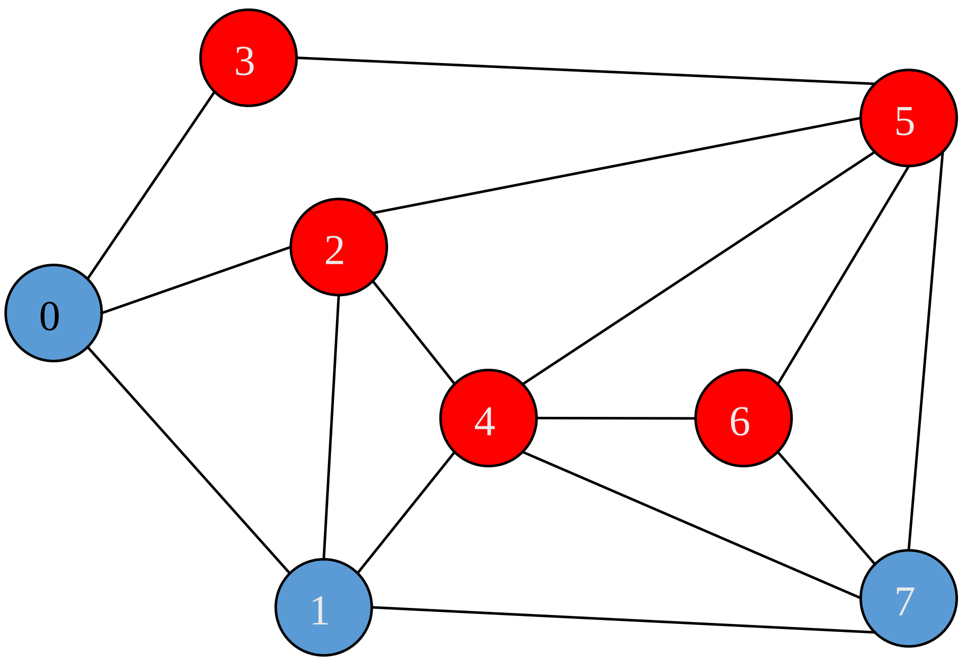
Hamiltonian Path

- A Hamiltonian path visits every vertex in the graph exactly once but it does not need to start and end at the same vertex.



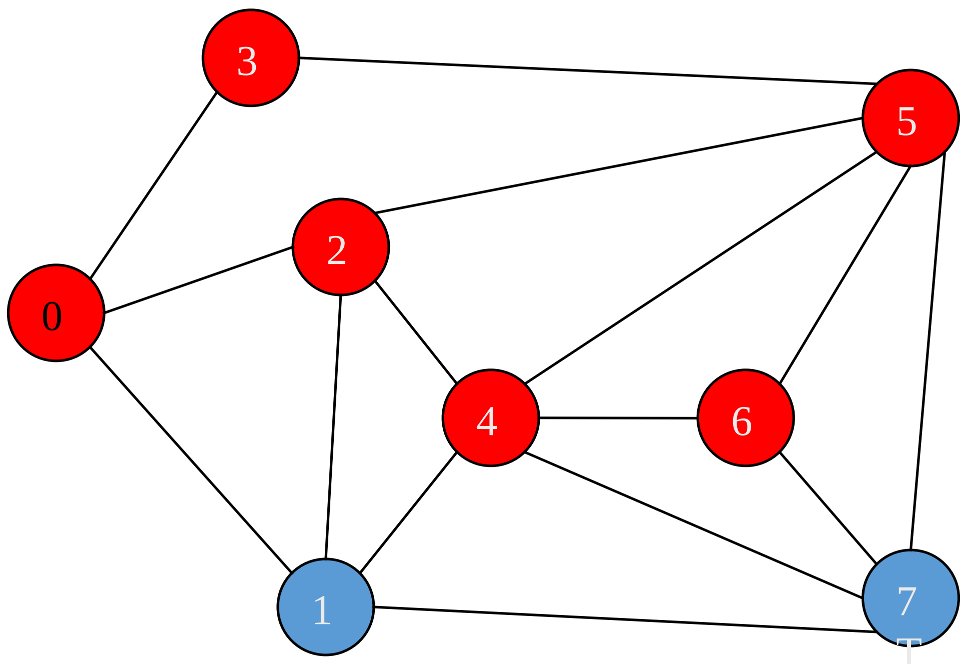
Hamiltonian Path

- A Hamiltonian path visits every vertex in the graph exactly once but it does not need to start and end at the same vertex.



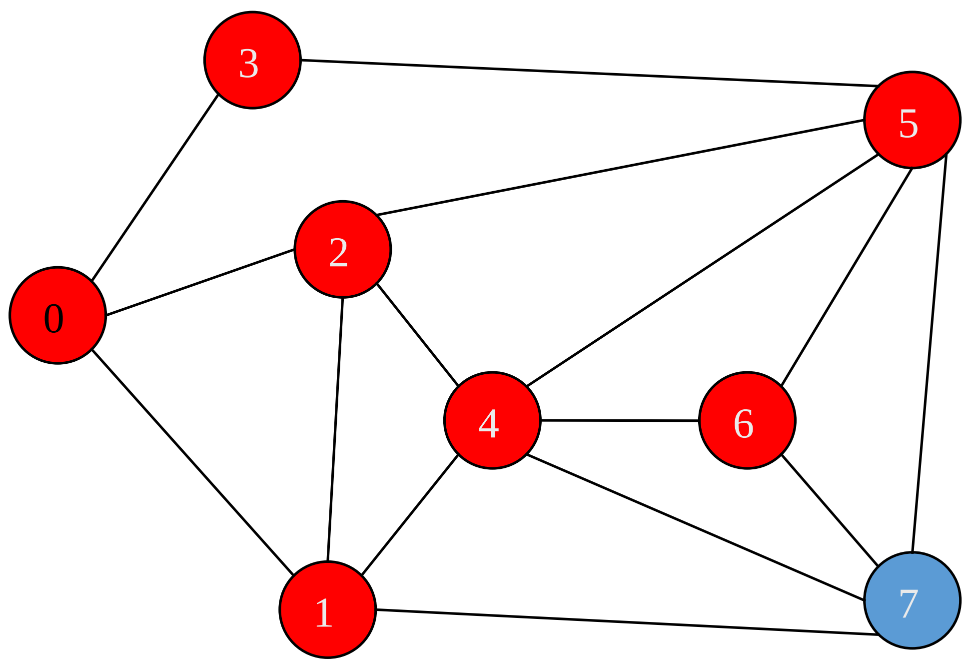
Hamiltonian Path

- A Hamiltonian path visits every vertex in the graph exactly once but it does not need to start and end at the same vertex.



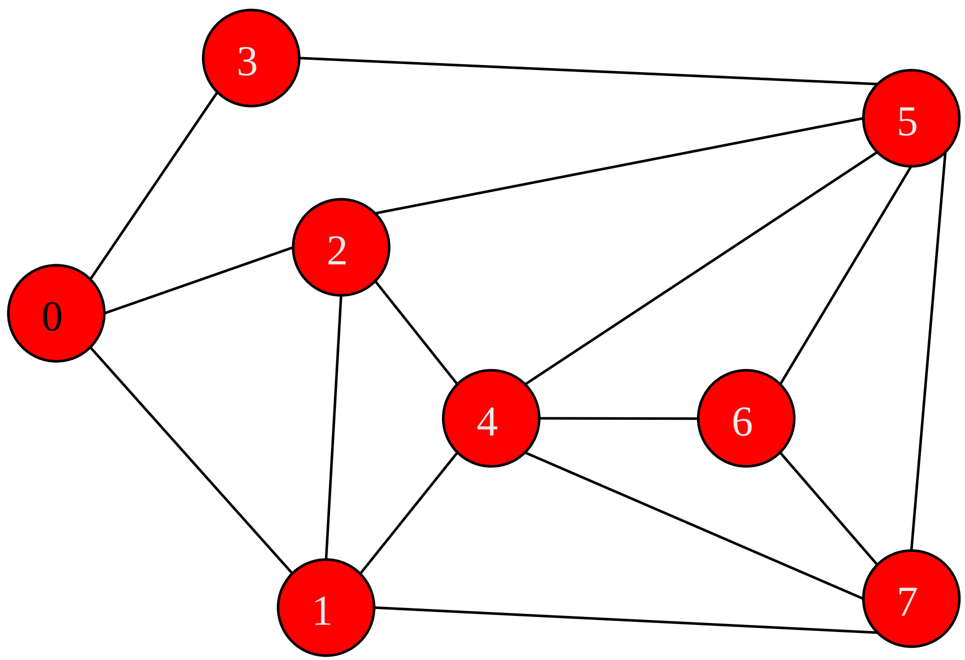
Hamiltonian Path

- A Hamiltonian path visits every vertex in the graph exactly once but it does not need to start and end at the same vertex.



Hamiltonian Path

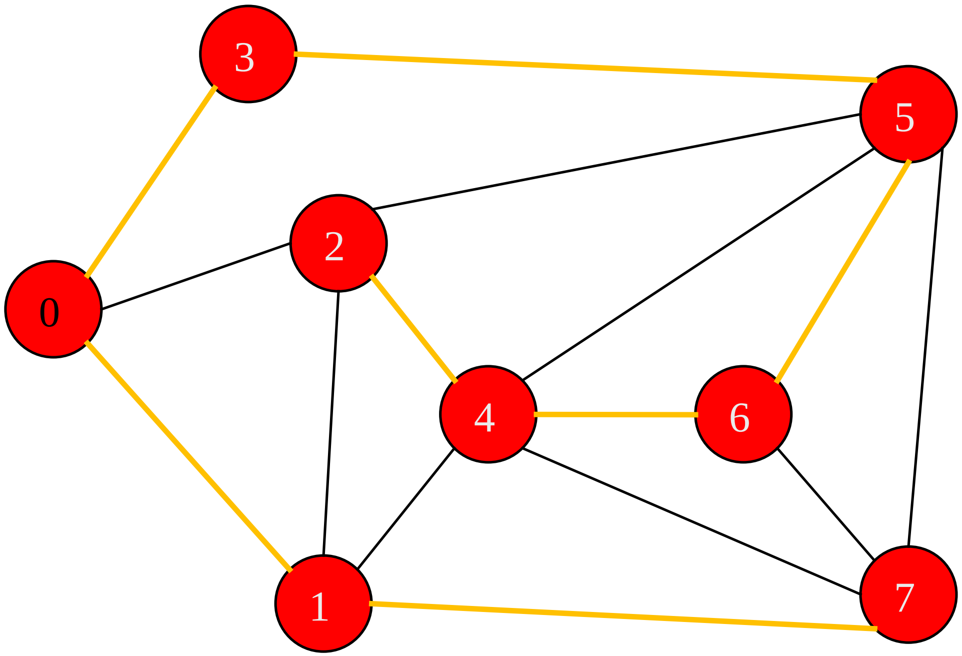
- A Hamiltonian path visits every vertex in the graph exactly once but it does not need to start and end at the same vertex.



Hamiltonian Path

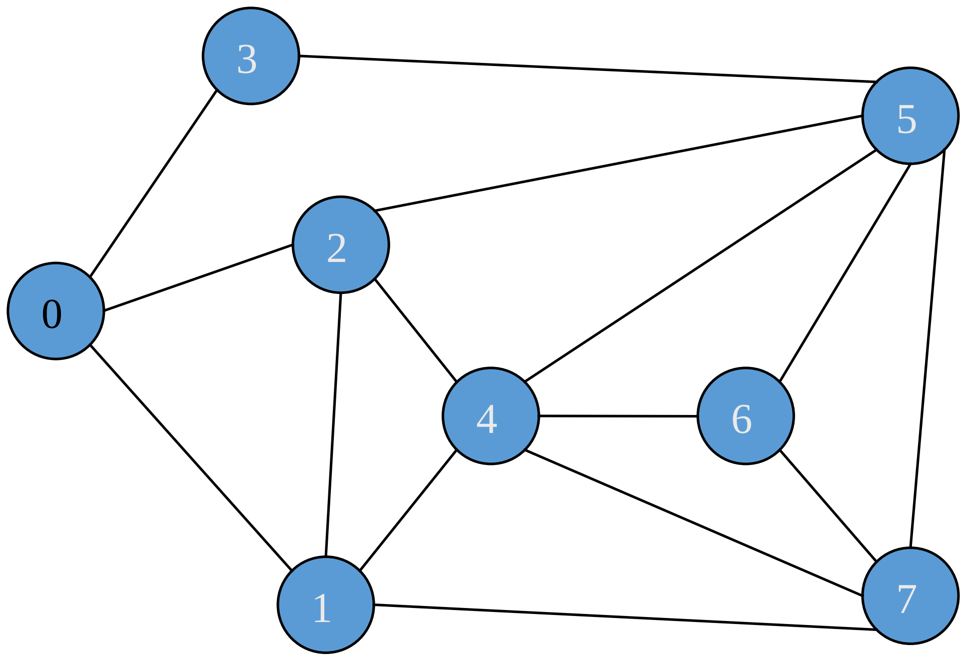
- A Hamiltonian path visits every vertex in the graph exactly once but it does not need to start and end at the same vertex.

Eg. (2,4,6,5,3,0,1,7)



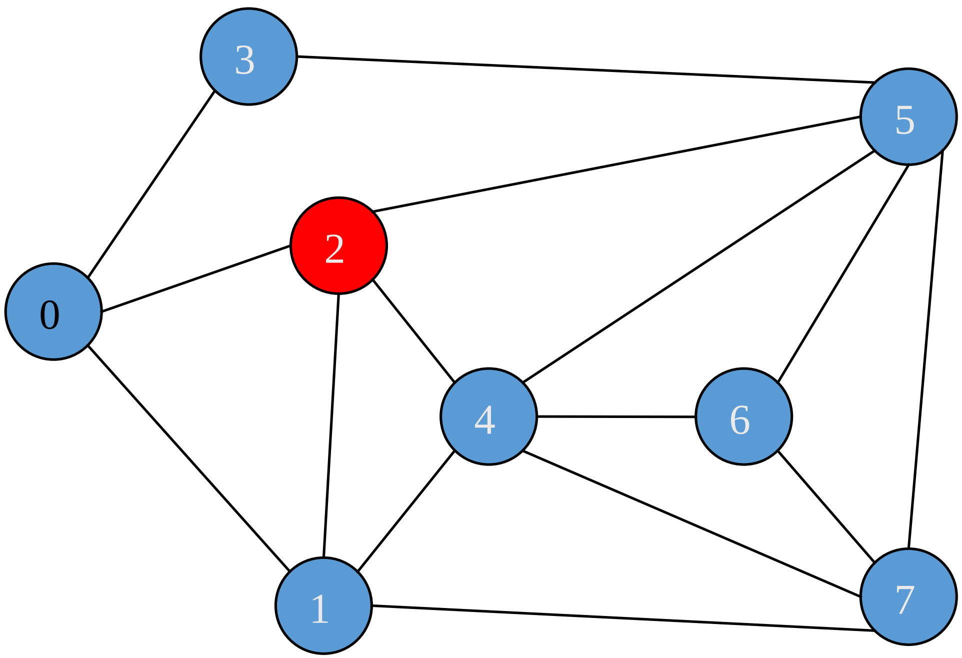
Cycle

- Closed path i.e. a path that begins and ends on the same vertex.



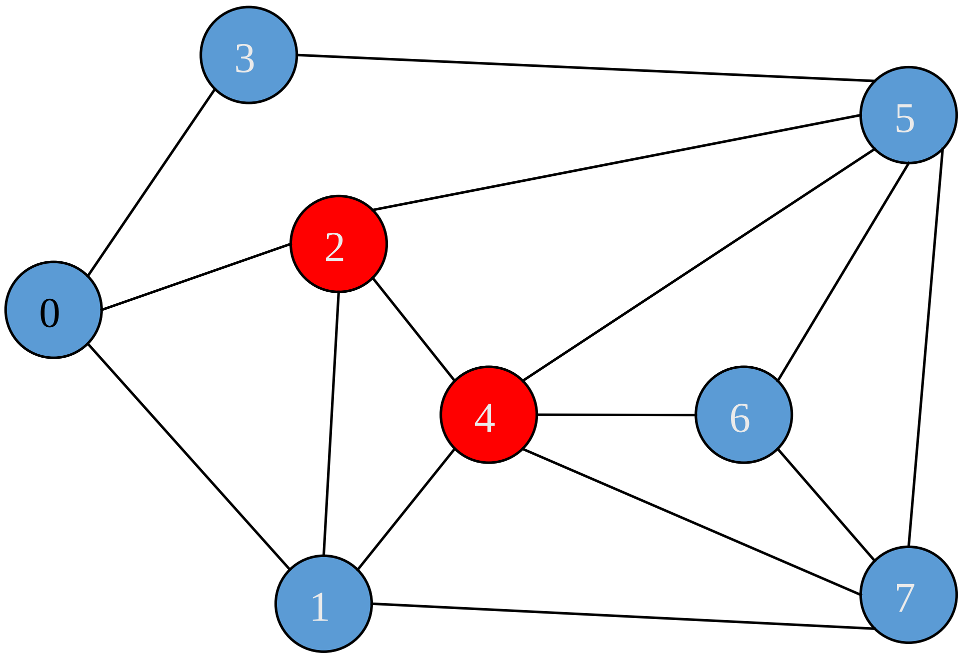
Cycle

- Closed path i.e. a path that begins and ends on the same vertex.



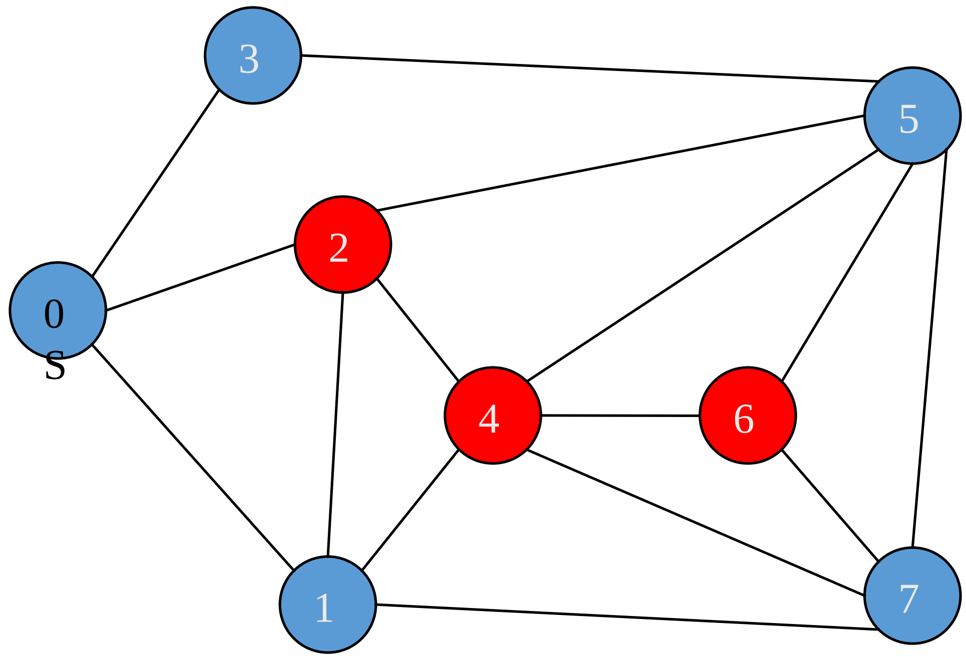
Cycle

- Closed path i.e. a path that begins and ends on the same vertex.



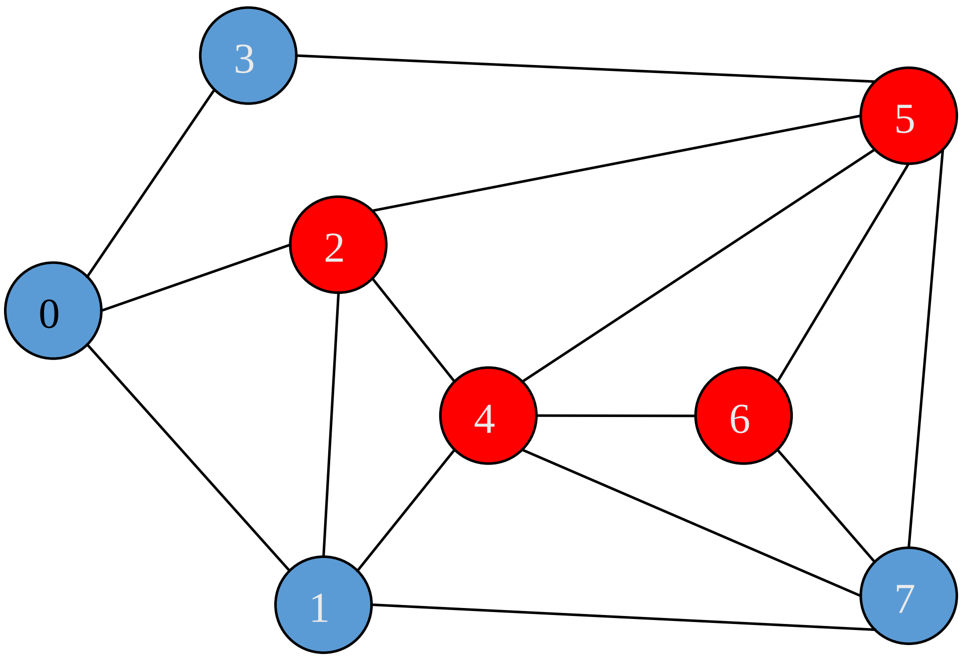
Cycle

- Closed path i.e. a path that begins and ends on the same vertex.



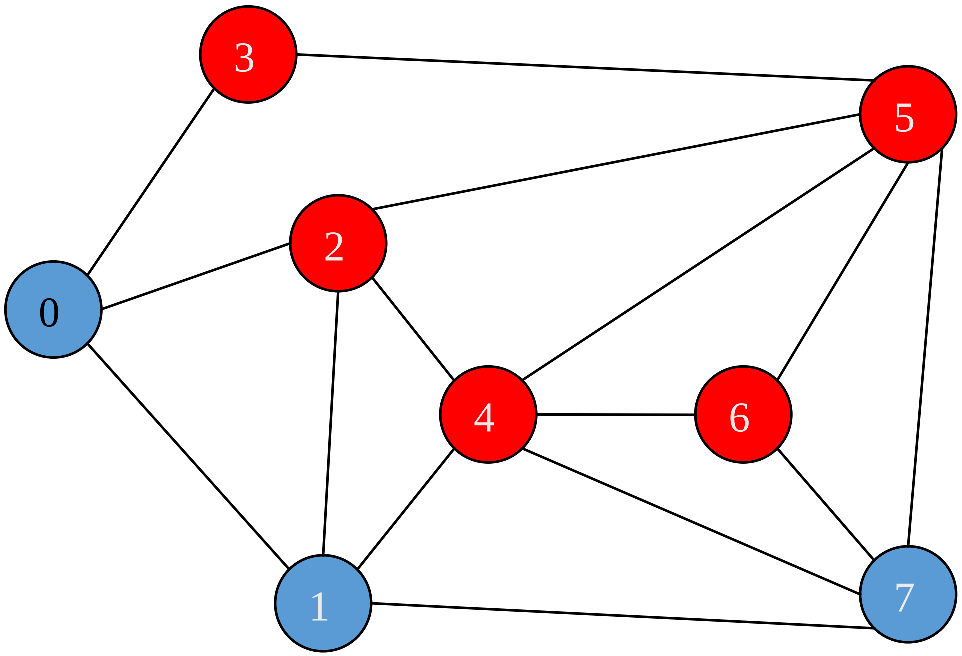
Cycle

- Closed path i.e. a path that begins and ends on the same vertex.



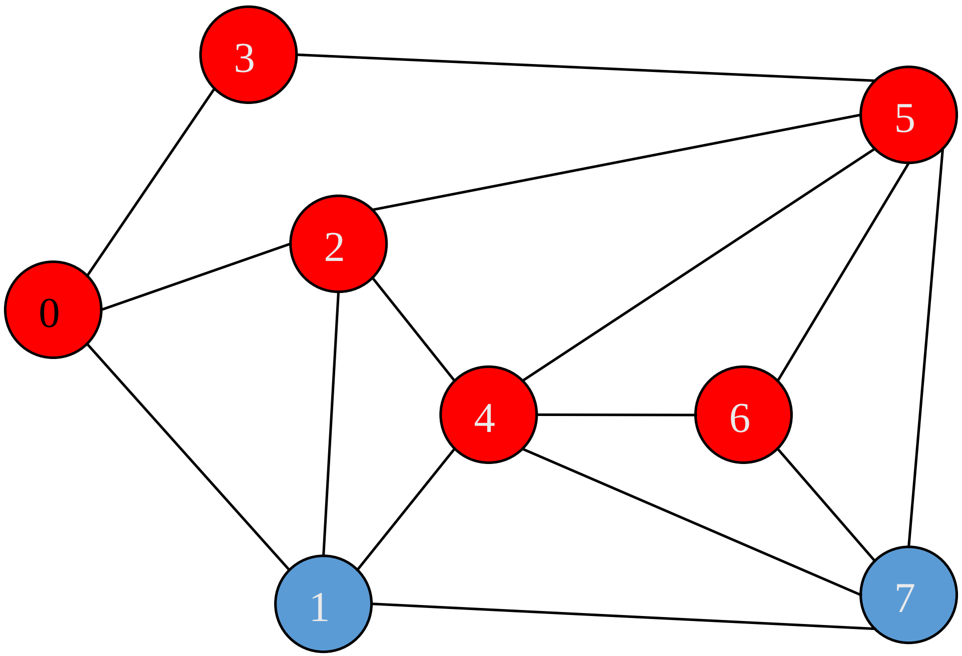
Cycle

- Closed path i.e. a path that begins and ends on the same vertex.



Cycle

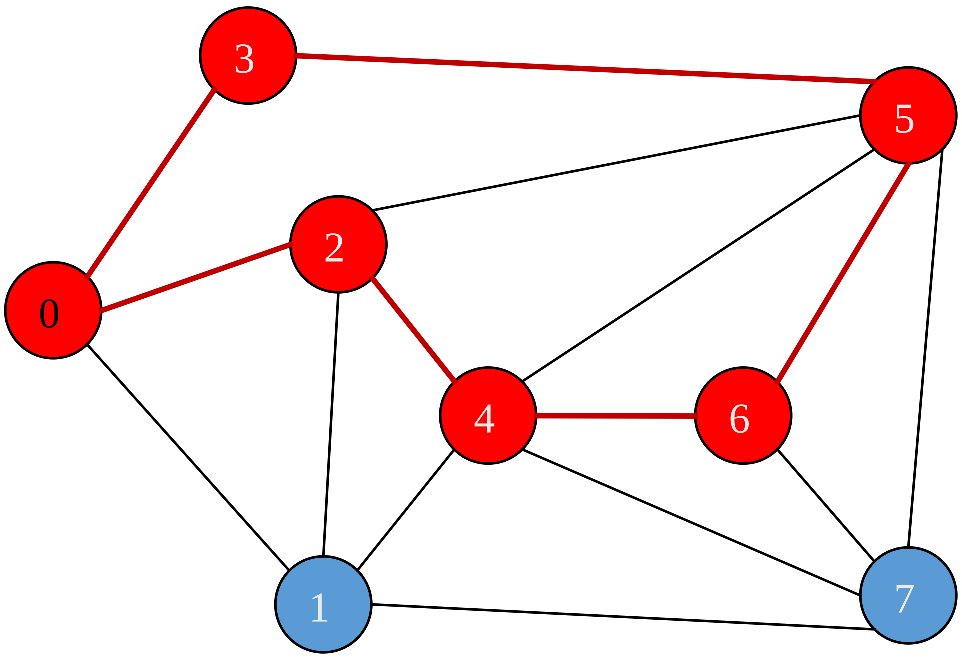
- Closed path i.e. a path that begins and ends on the same vertex.



Cycle

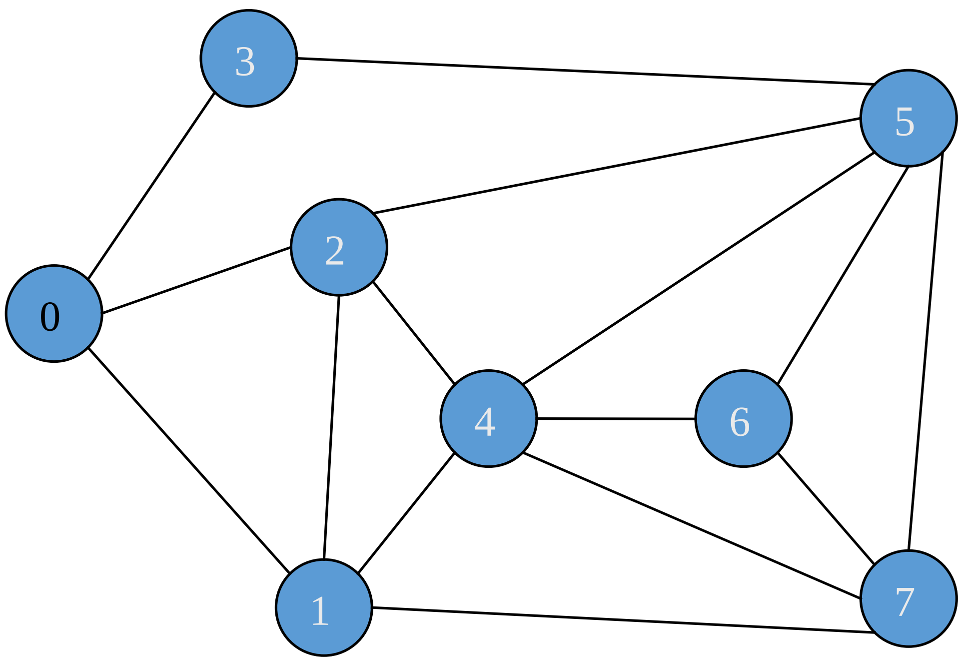
- Closed path i.e. a path that begins and ends on the same vertex.

  Eg. (2,4,6,5,3,0,2)



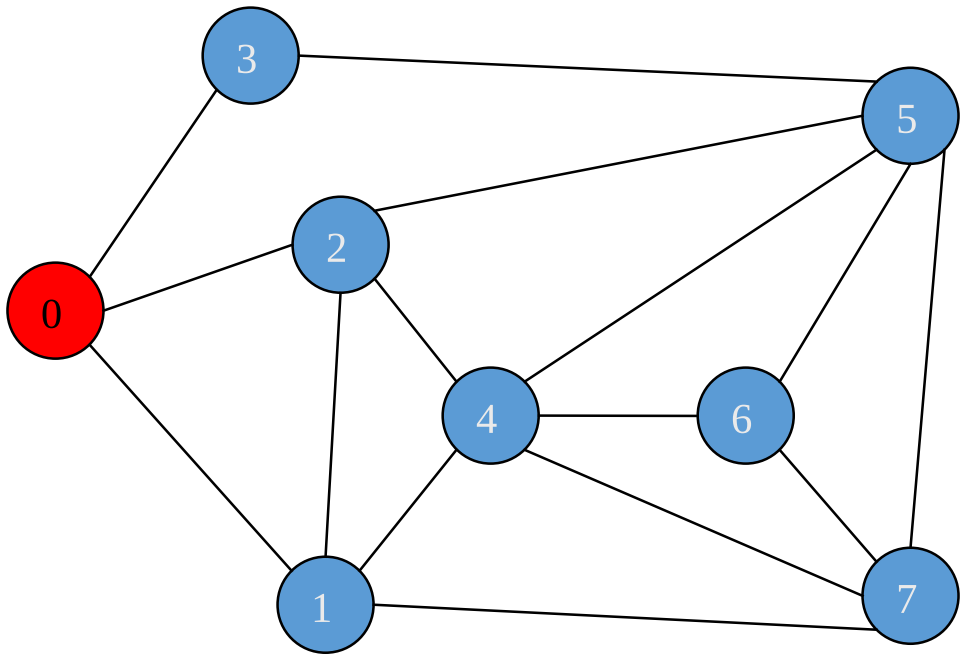
Hamiltonian Cycle

- A Hamiltonian cycle is a cycle that visits every vertex in the graph exactly once.
- A graph is called Hamiltonian if it has a Hamiltonian cycle.



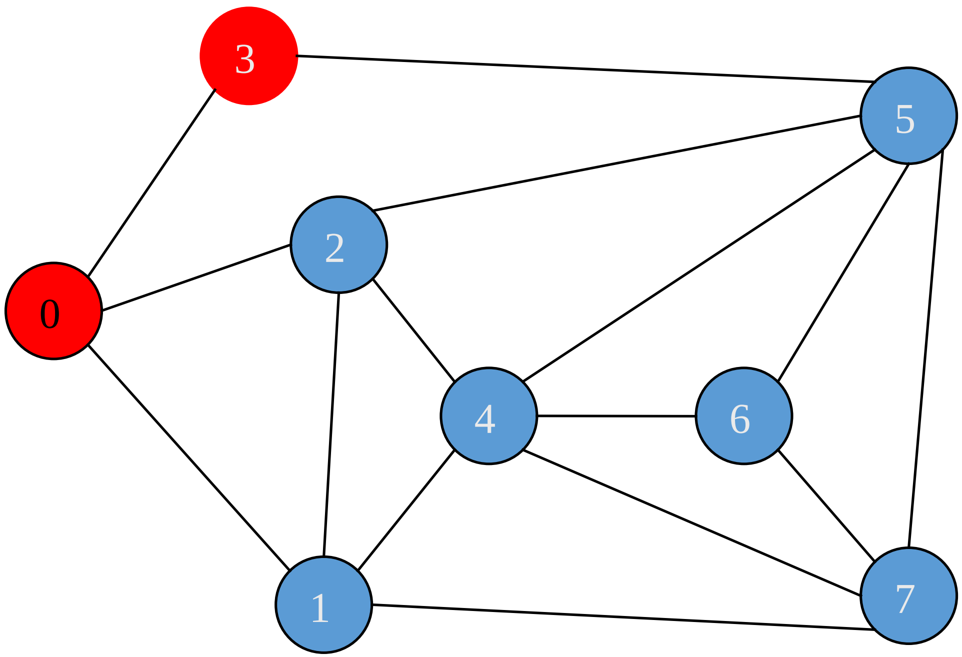
Hamiltonian Cycle

- A Hamiltonian cycle is a cycle that visits every vertex in the graph exactly once.
- A graph is called Hamiltonian if it has a Hamiltonian cycle.



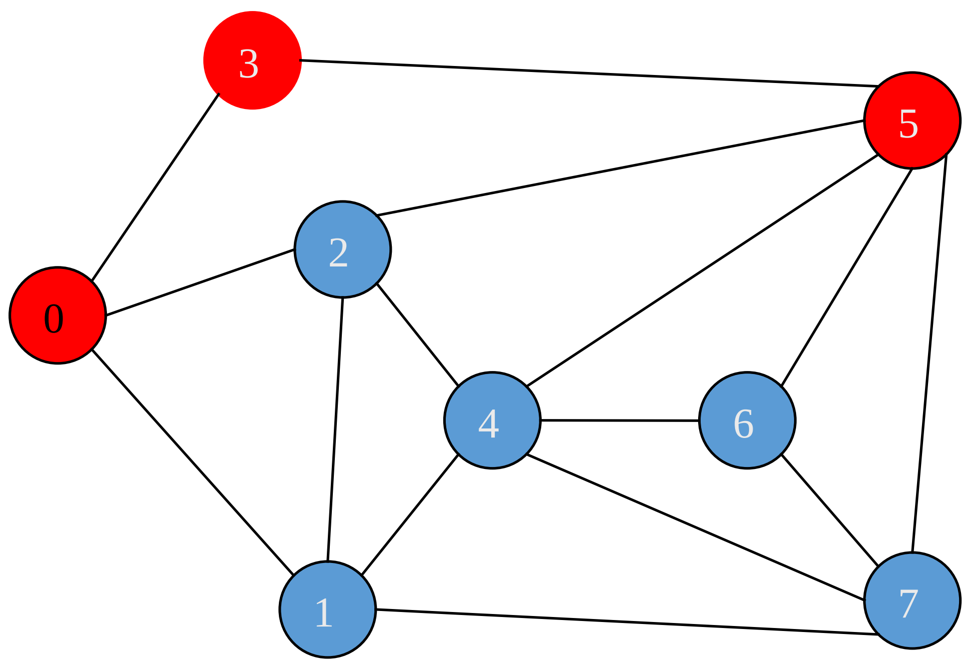
Hamiltonian Cycle

- A Hamiltonian cycle is a cycle that visits every vertex in the graph exactly once.
- A graph is called Hamiltonian if it has a Hamiltonian cycle.



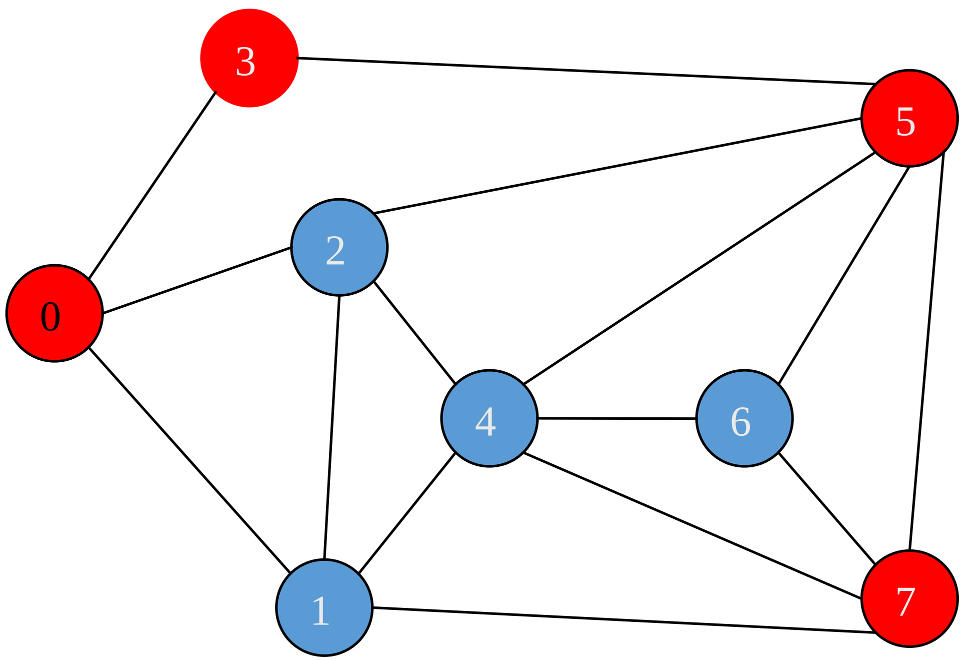
Hamiltonian Cycle

- A Hamiltonian cycle is a cycle that visits every vertex in the graph exactly once.
- A graph is called Hamiltonian if it has a Hamiltonian cycle.



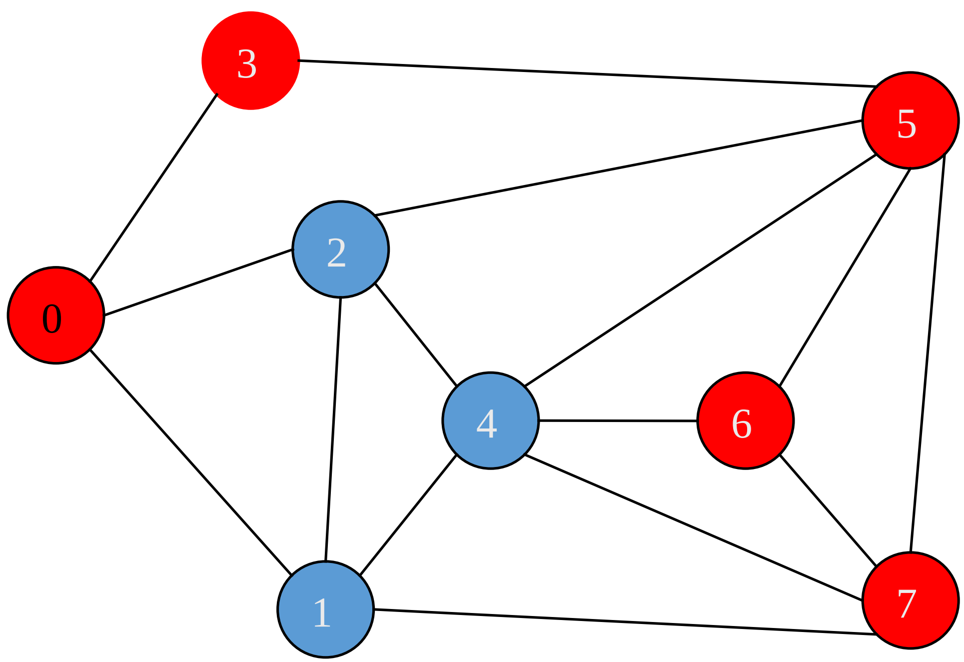
Hamiltonian Cycle

- A Hamiltonian cycle is a cycle that visits every vertex in the graph exactly once.
- A graph is called Hamiltonian if it has a Hamiltonian cycle.



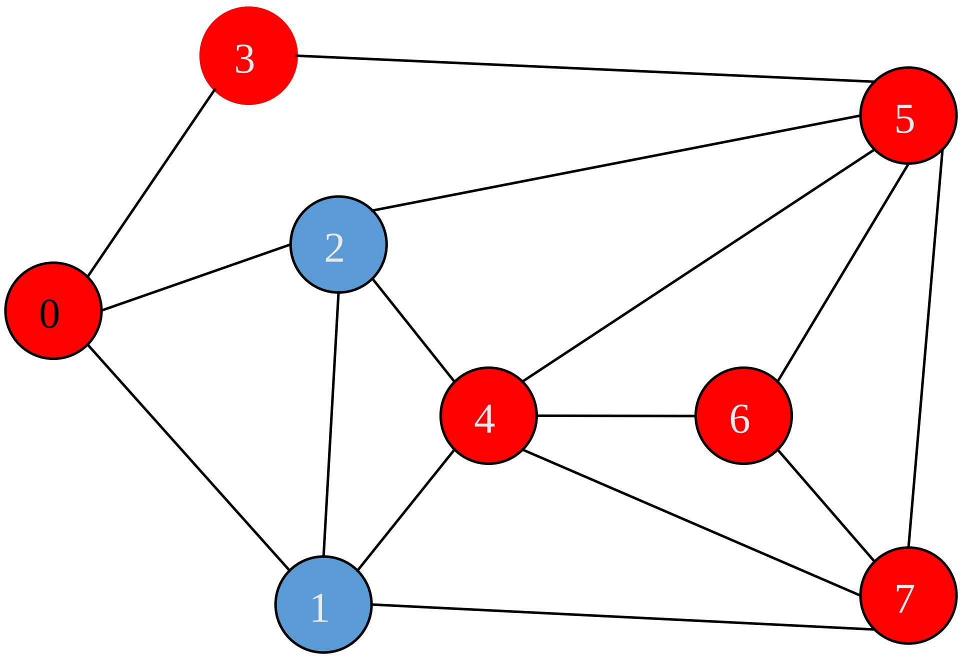
Hamiltonian Cycle

- A Hamiltonian cycle is a cycle that visits every vertex in the graph exactly once.
- A graph is called Hamiltonian if it has a Hamiltonian cycle.



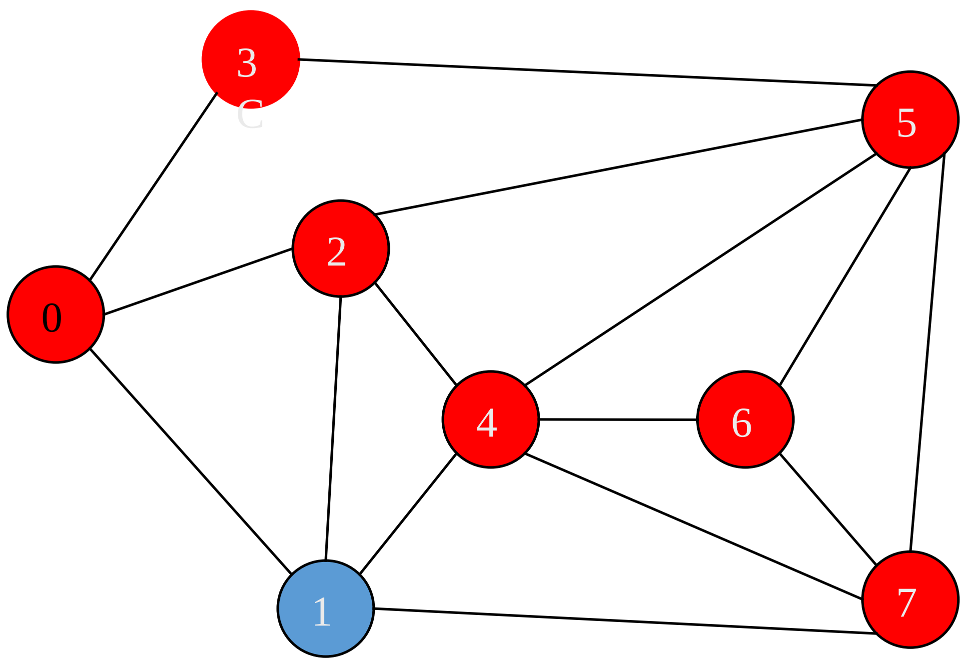
Hamiltonian Cycle

- A Hamiltonian cycle is a cycle that visits every vertex in the graph exactly once.
- A graph is called Hamiltonian if it has a Hamiltonian cycle.



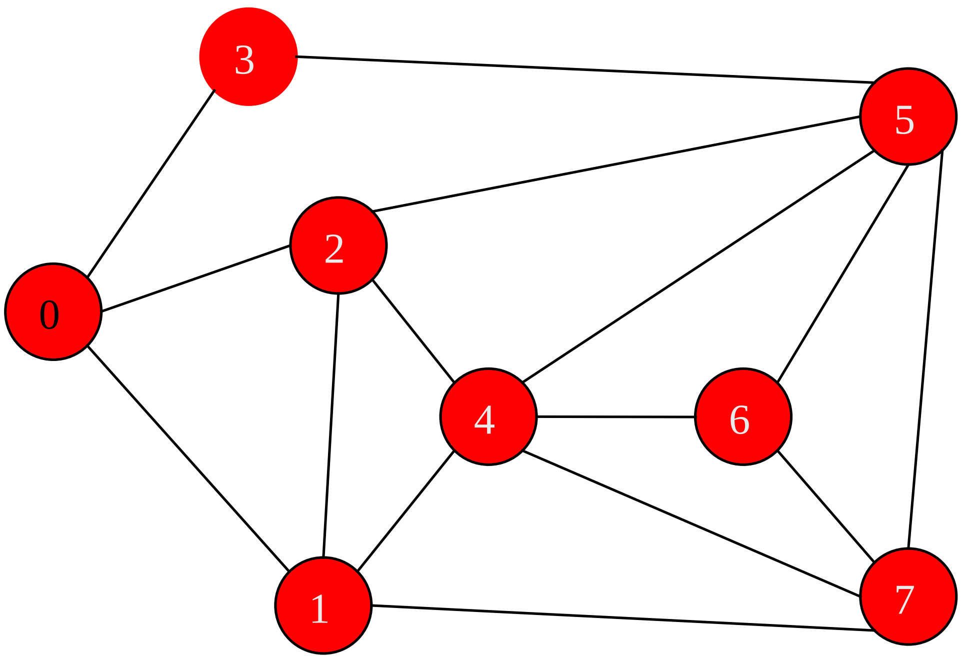
Hamiltonian Cycle

- A Hamiltonian cycle is a cycle that visits every vertex in the graph exactly once.
- A graph is called Hamiltonian if it has a Hamiltonian cycle.



Hamiltonian Cycle

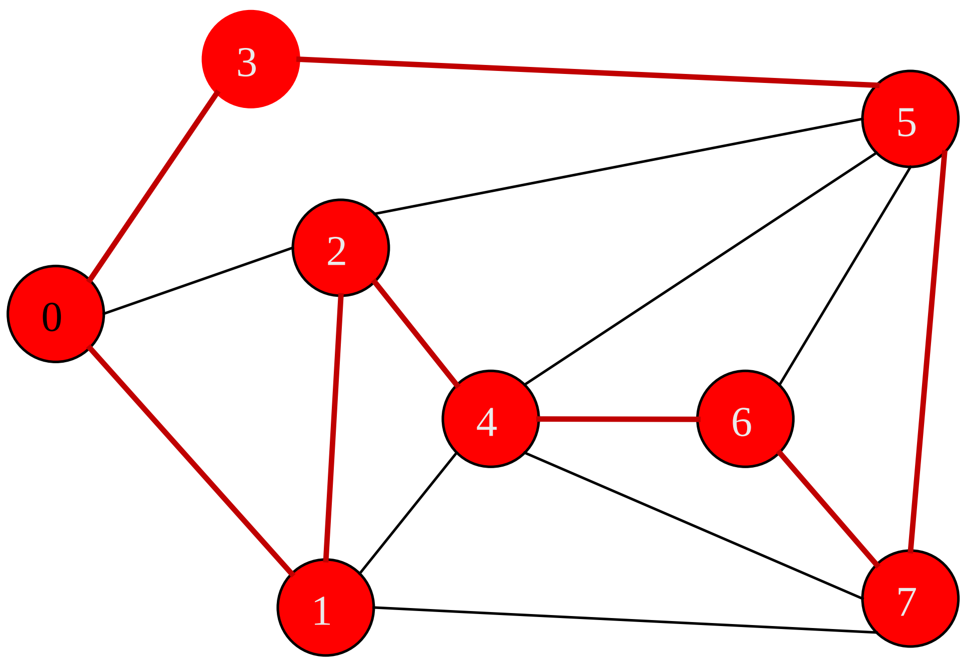
- A Hamiltonian cycle is a cycle that visits every vertex in the graph exactly once.
- A graph is called Hamiltonian if it has a Hamiltonian cycle.



Hamiltonian Cycle

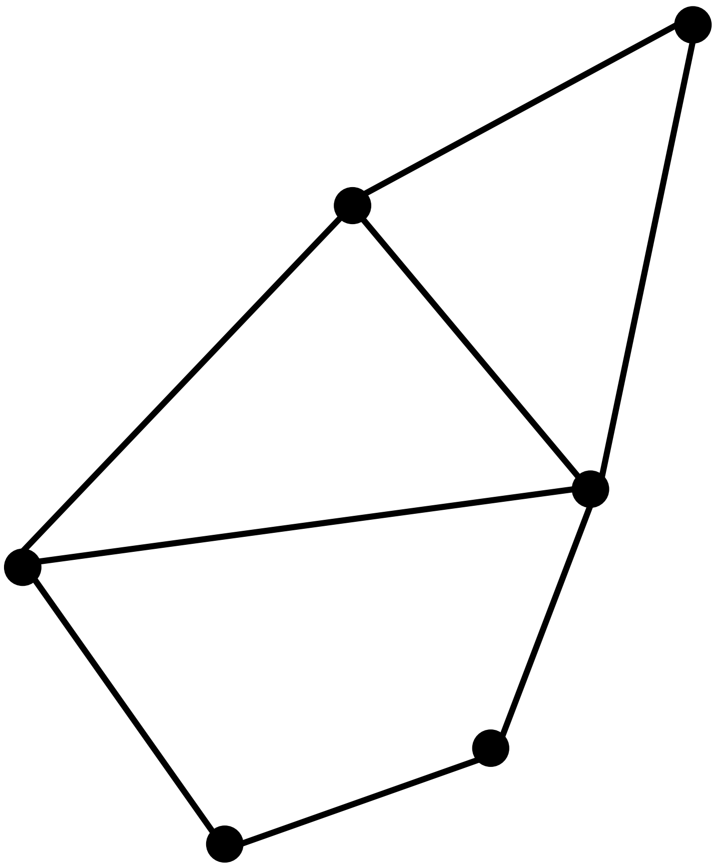
- A Hamiltonian cycle is a cycle that visits every vertex in the graph exactly once.
- A graph is called Hamiltonian if it has a Hamiltonian cycle.

Eg. (0,3,5,7,6,4,2,1,0)



Let's practice some graph notions

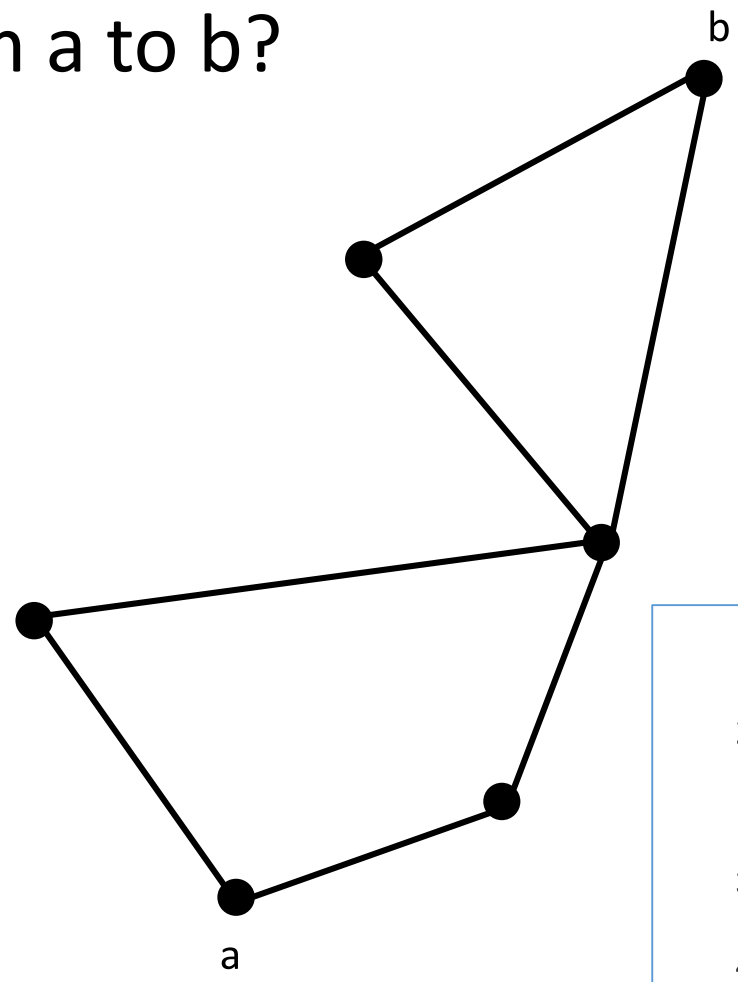
How many cycles does this graph have?



1. Visit https://flux.qa
2. Log in using your Authcate details (not required if you're already logged in to Monash)
3. Touch the + symbol and enter the code: EQMSF9
4. Answer questions when they pop up.

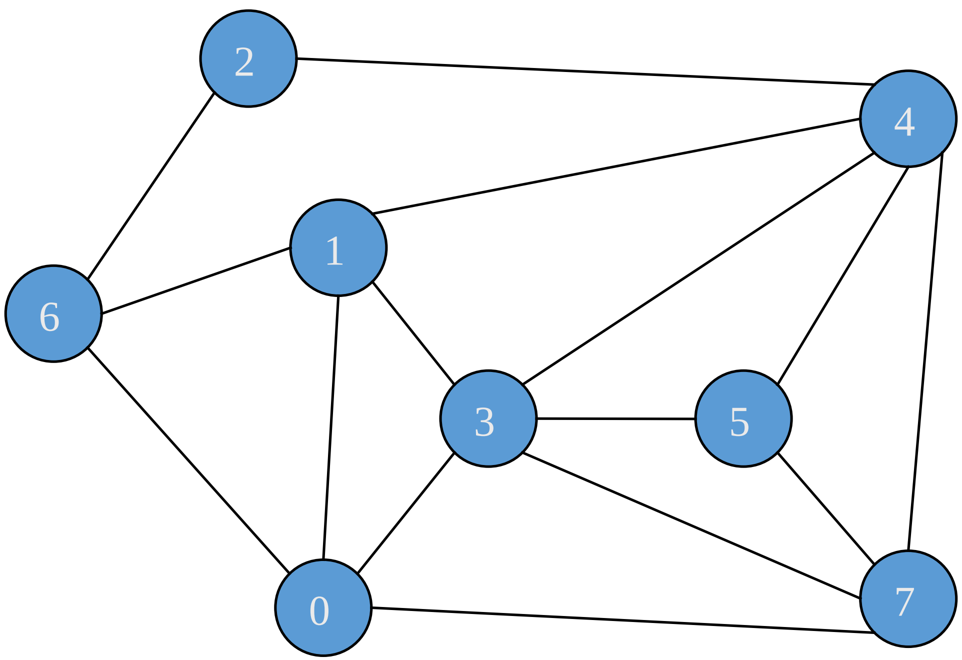
Let's practice some graph notions

How many paths are there from a to b?



1. Visit https://flux.qa
2. Log in using your Authcate details (not required if you're already logged in to Monash)
3. Touch the + symbol and enter the code: EQMSF9
4. Answer questions when they pop up.

Adjacency Matrix and List



Adjacency List

$0 \to \;\; 1, \;\; 3, \;\; 6, \;\; 7$

$1 \to \;\; 0, \;\; 3, \;\; 4, \;\; 6$

$2 \to \;\; 4, \;\; 6$

$3 \to \;\; 0, \;\; 1, \;\; 4, \;\; 5, \;\; 7$

$4 \to \;\; 1, \;\; 2, \;\; 3, \;\; 5, \;\; 7$

$5 \to \;\; 3, \;\; 4, \;\; 7$

$6 \to \;\; 0, \;\; 1, \;\; 2$

$7 \to \;\; 0, \;\; 3, \;\; 4, \;\; 5$

Adjacency Matrix

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

What if the graph is directed?

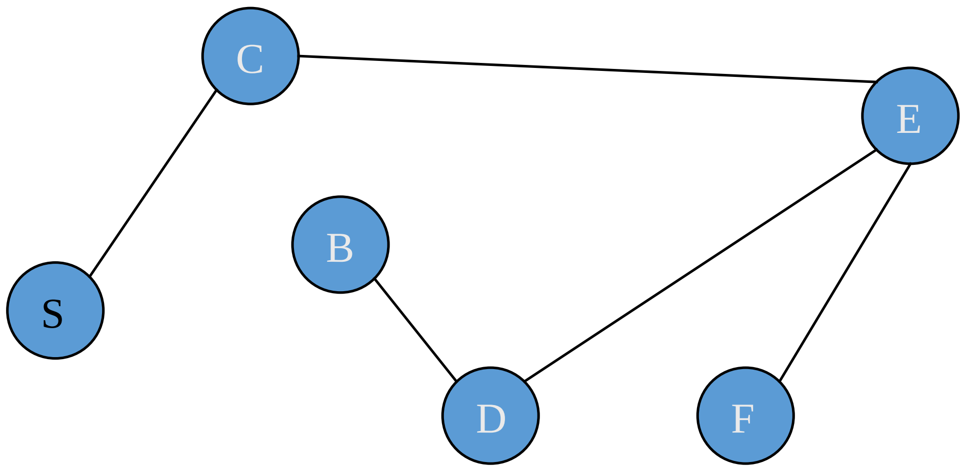
Finding the neighbours of a given vertex (graph represented as an adjacency matrix)

$$g = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

```python
def neighbours(i, g):
    """Input: vertex i, graph g
       Output: neighbours of i
       For example:
       >>> neighbours(5, graph)
       [3,4,7]
    """
    n = len(g)
    res = []
    for j in range(n):
        if g[i][j]==1:
            res.append(j)
    return res
```

Trees

- A Graph that is simple, connected and has no cycles

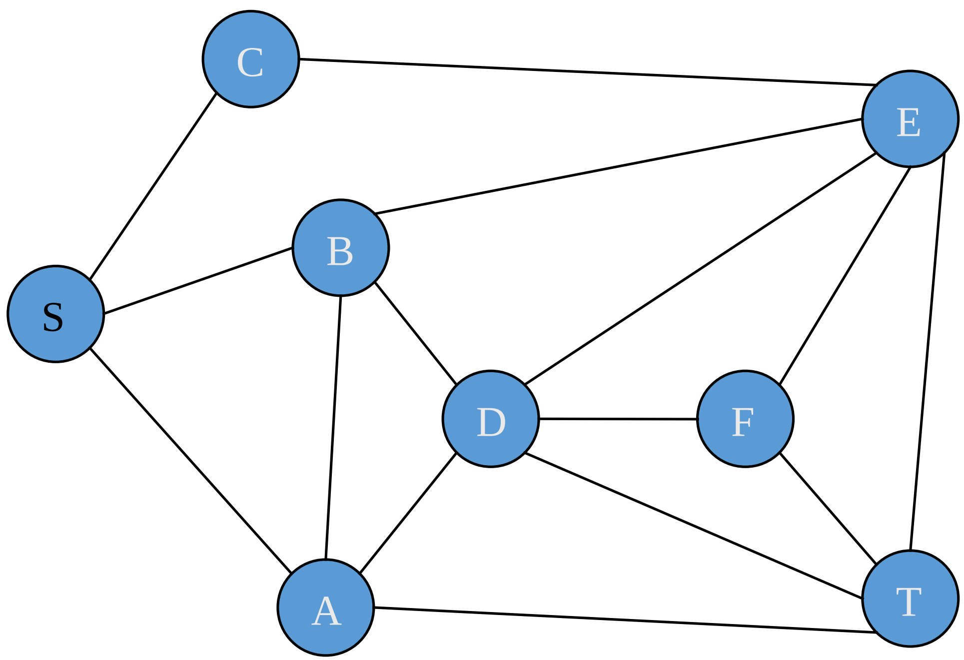


A tree

- is ***minimally connected***, i.e. removing any edge makes graph disconnected
- Contains a ***unique path*** between any two vertices

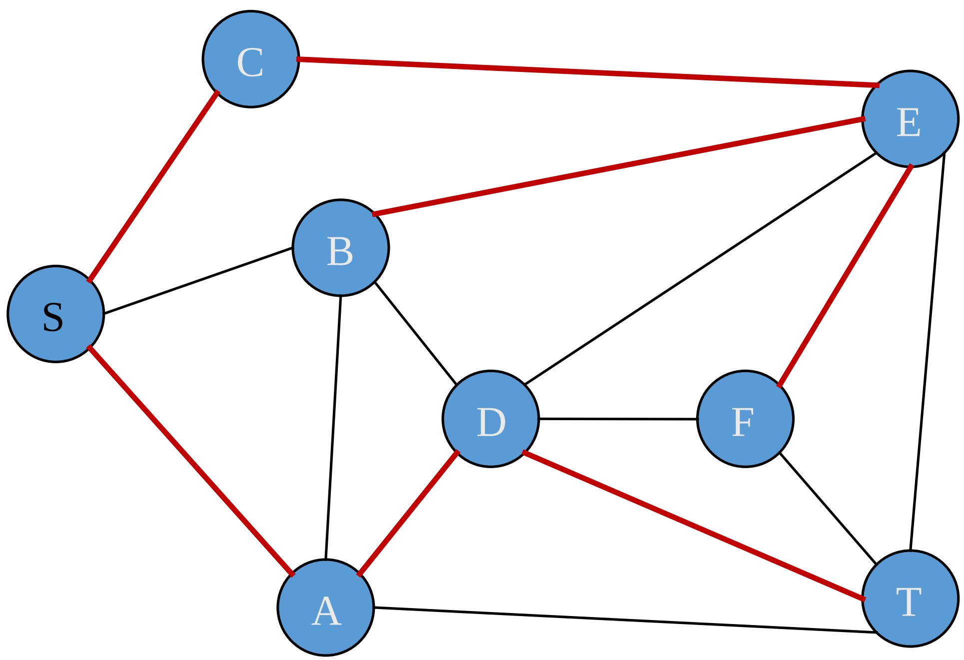
Spanning Tree of a Graph, G

- If we assume that G is simple and connected then a spanning tree of G is a tree which:
  - Contains all the vertices of G (Spans G), and
  - The edges of the tree are edges of G (Subgraph of G).



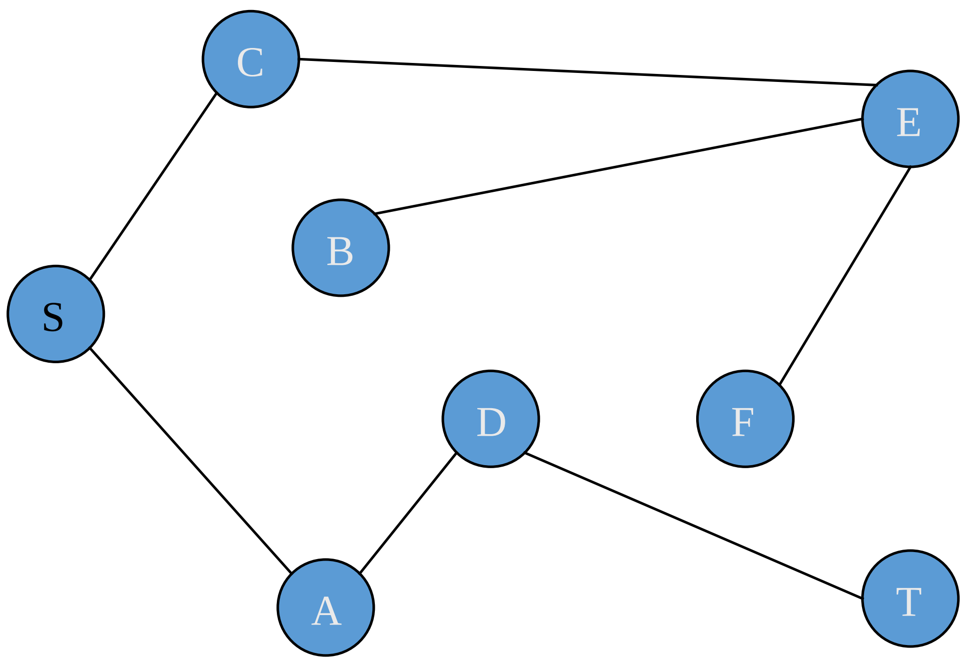
Spanning Tree of a Graph, G

- If we assume that G is simple and connected then a spanning tree of G is a tree which:
  - Contains all the vertices of G (Spans G), and
  - The edges of the tree are edges of G (Subgraph of G).

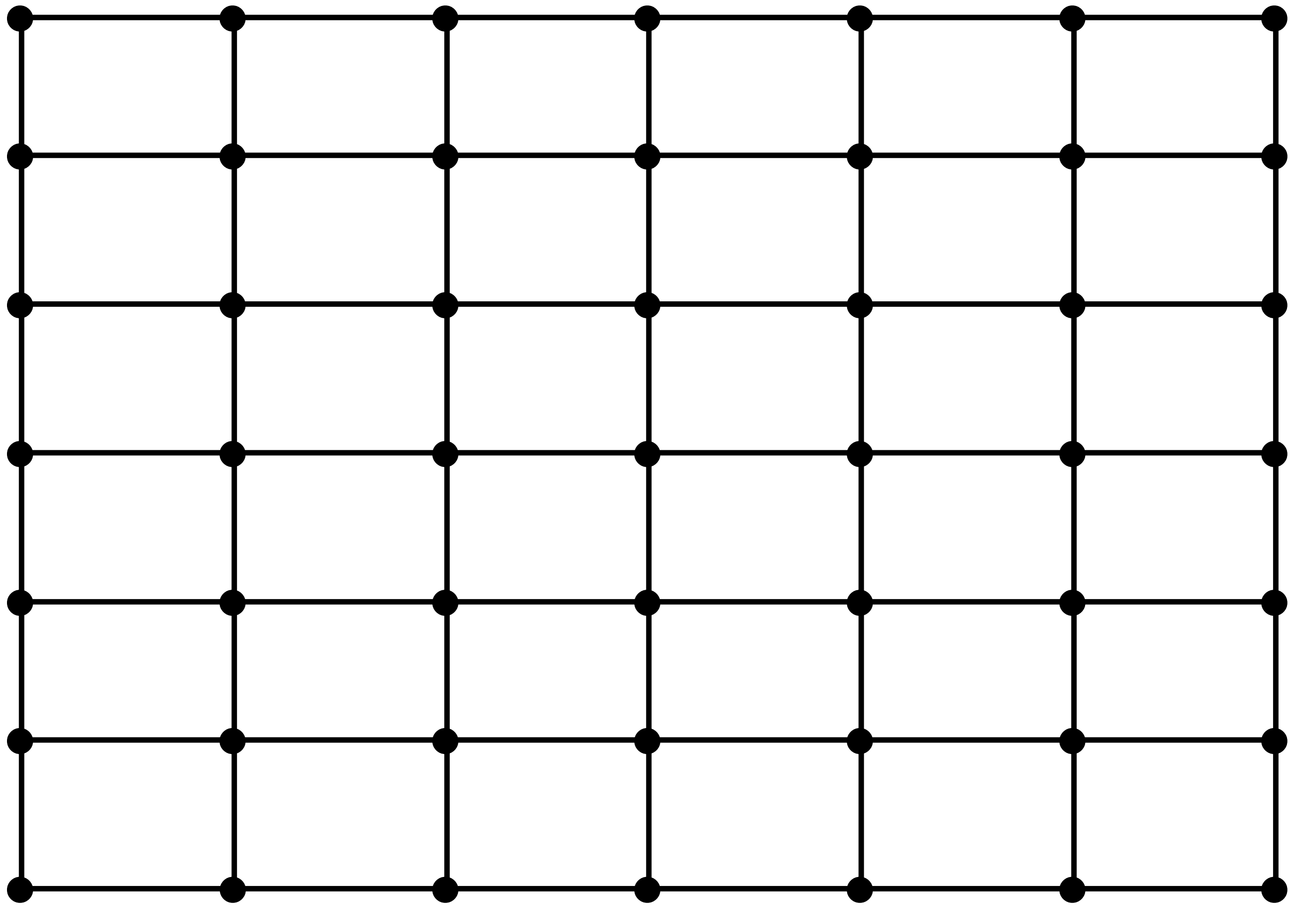


Spanning Tree of a Graph, G

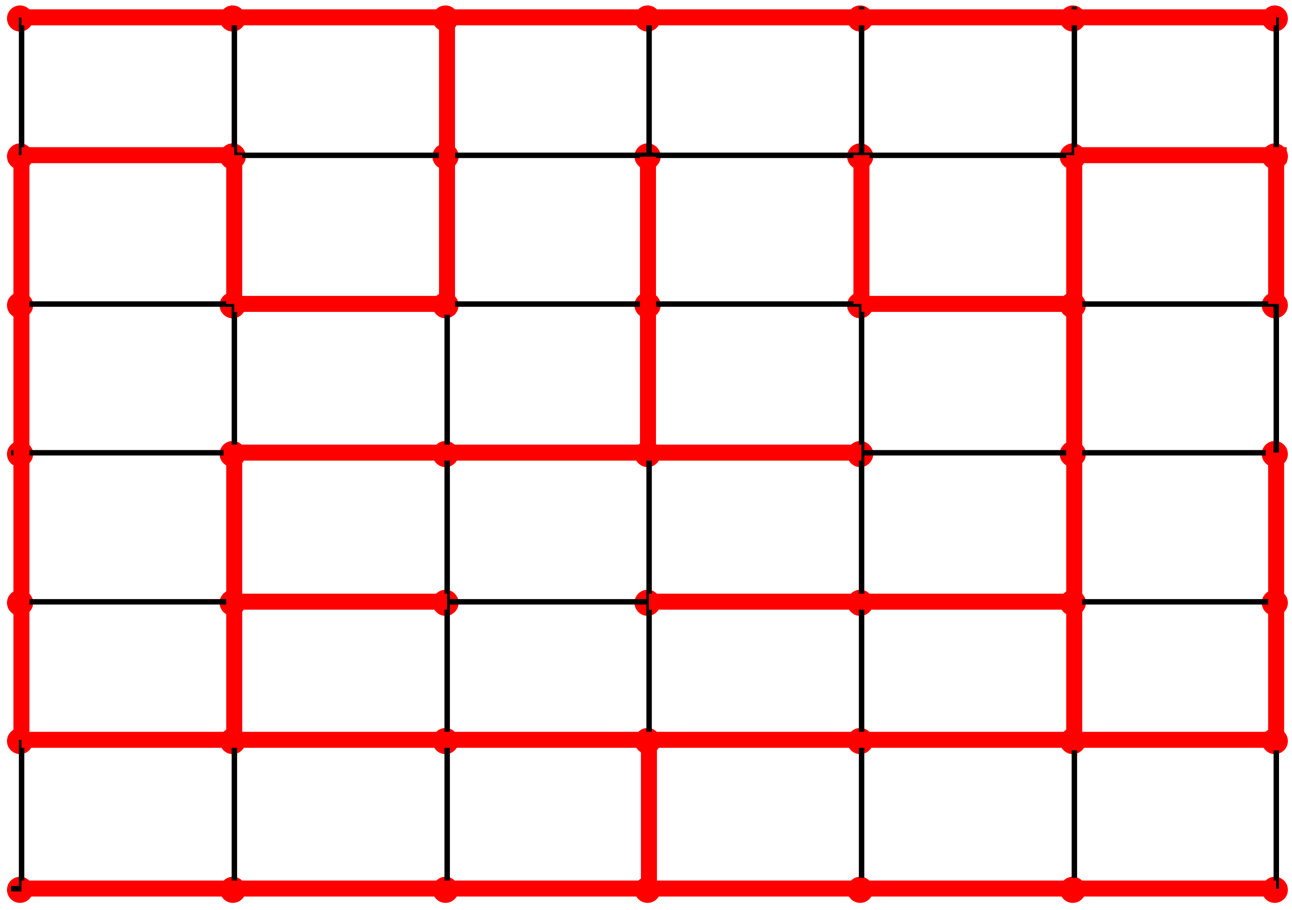
- If we assume that G is simple and connected then a spanning tree of G is a tree which:
  - Contains all the vertices of G (Spans G), and
  - The edges of the tree are edges of G (Subgraph of G).



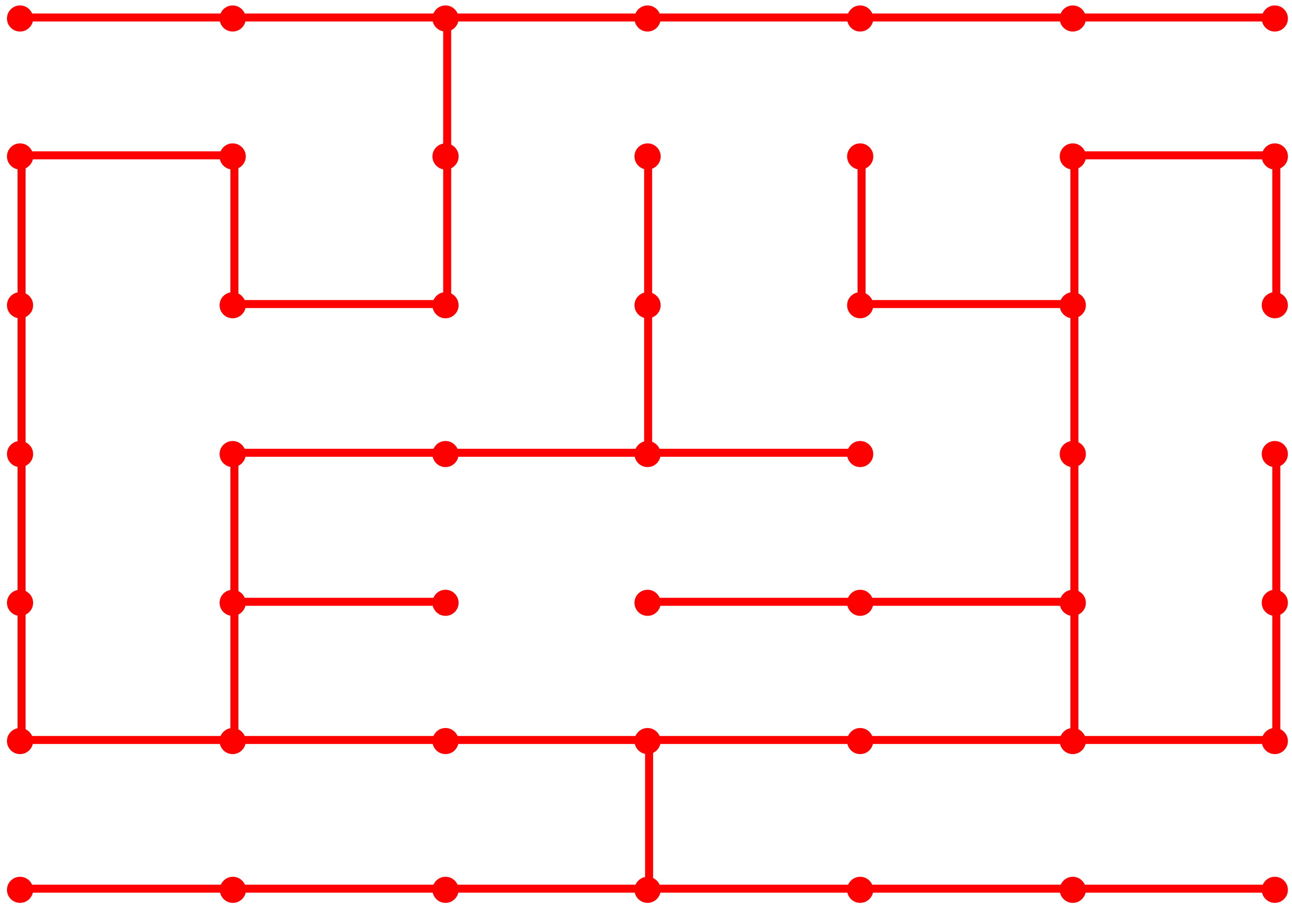
Start with a Grid



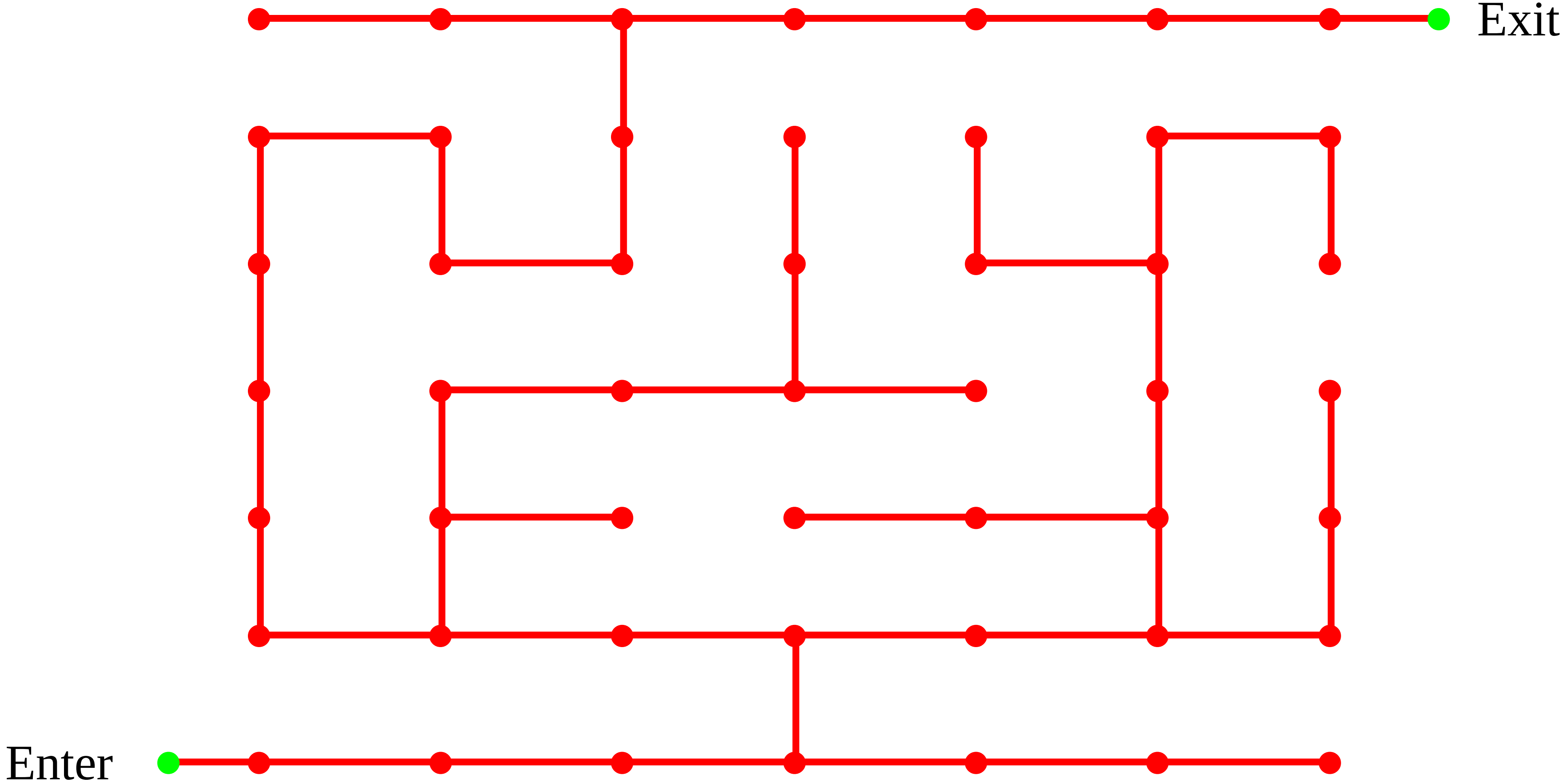
Find the Spanning Tree



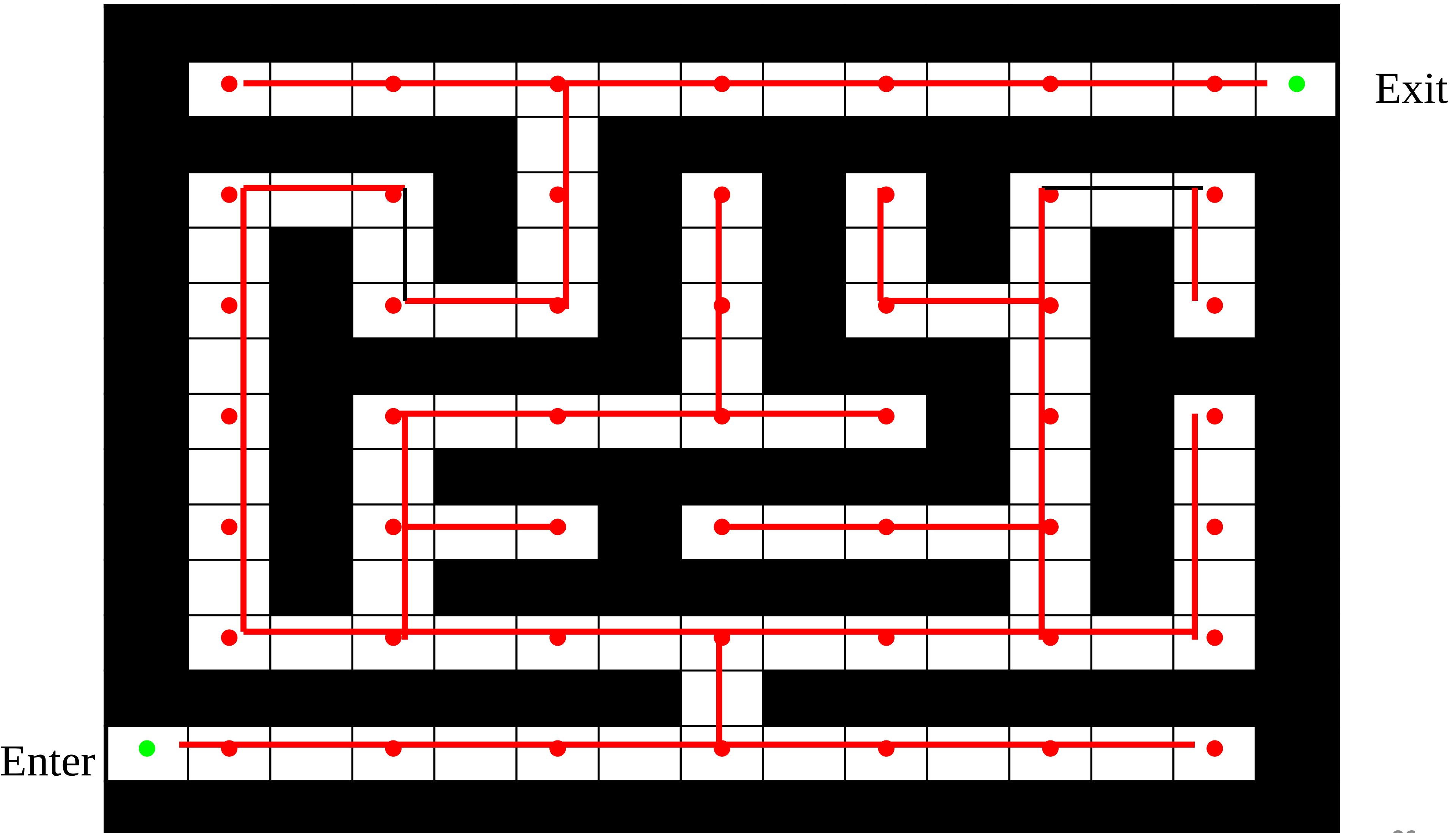
Find the Spanning Tree



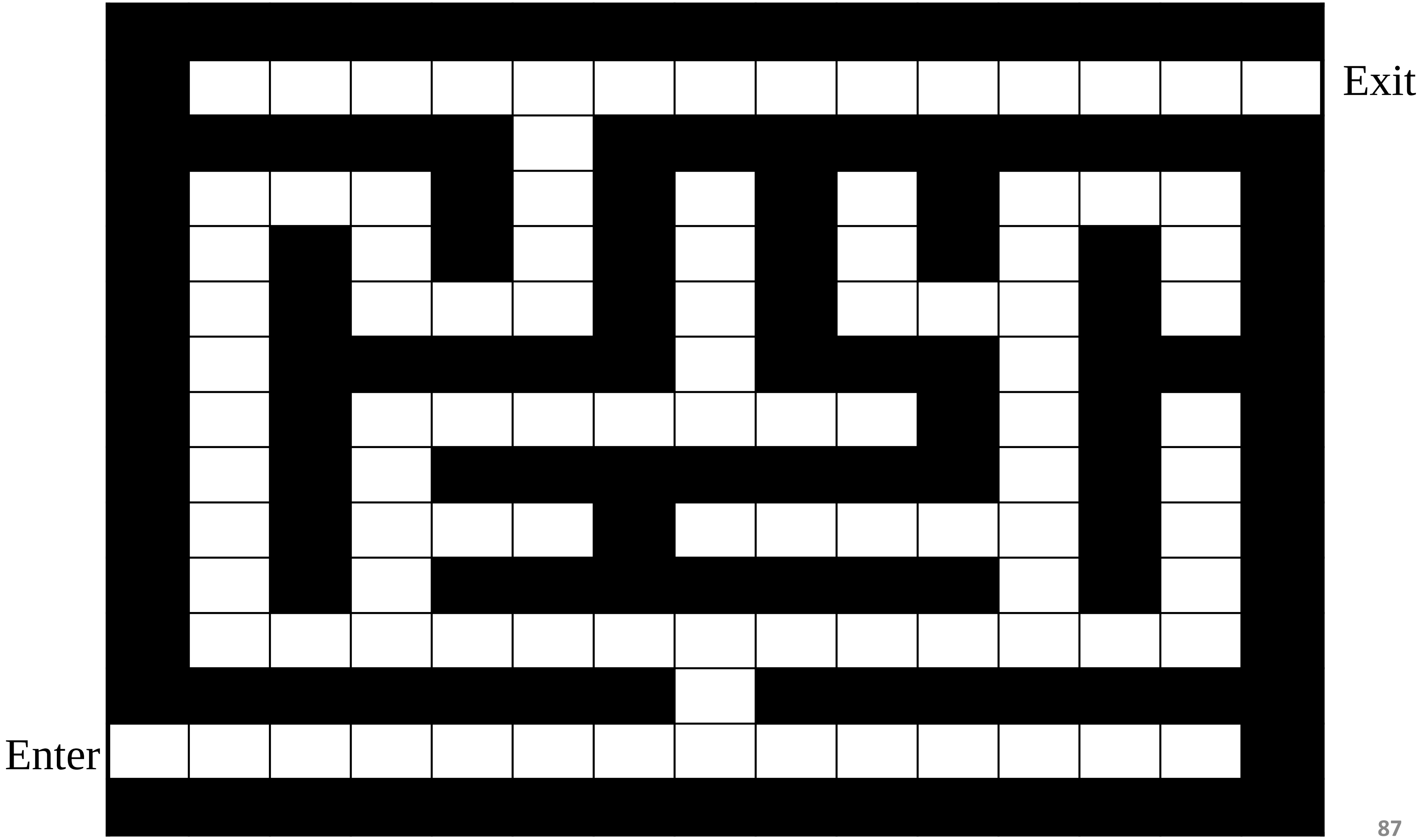
Add entrance and exit



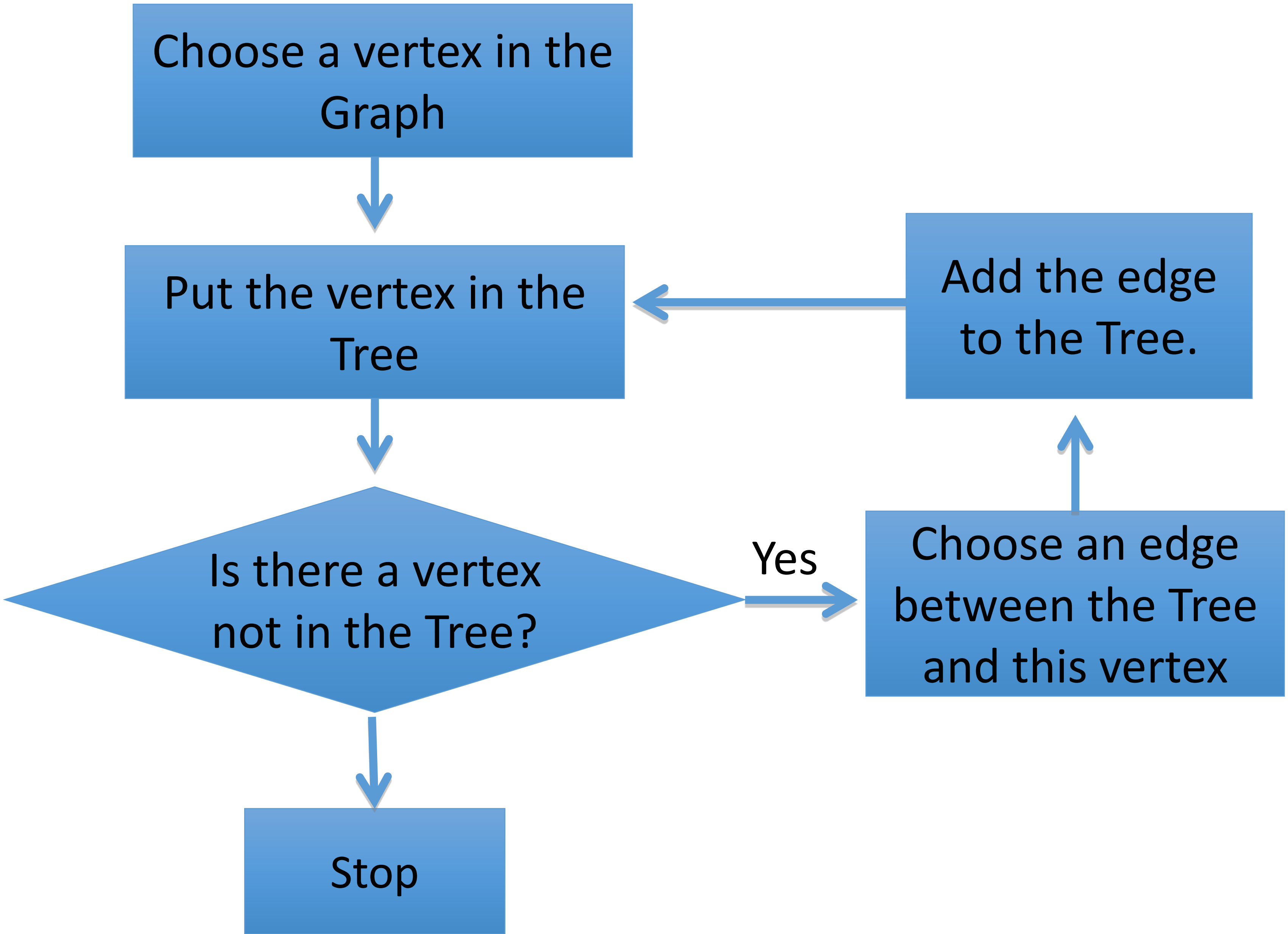
Put in walls



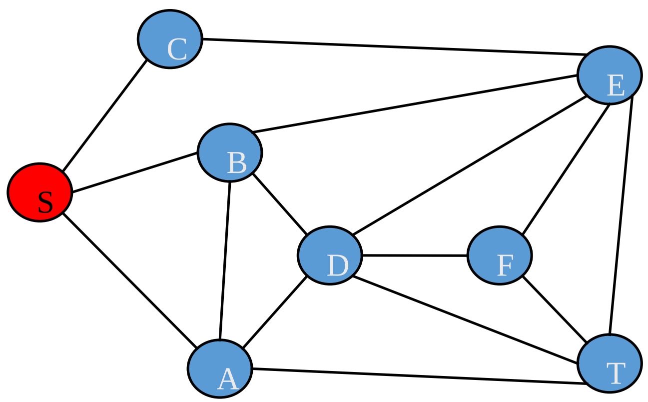
Remove Tree



Prim’s algorithm

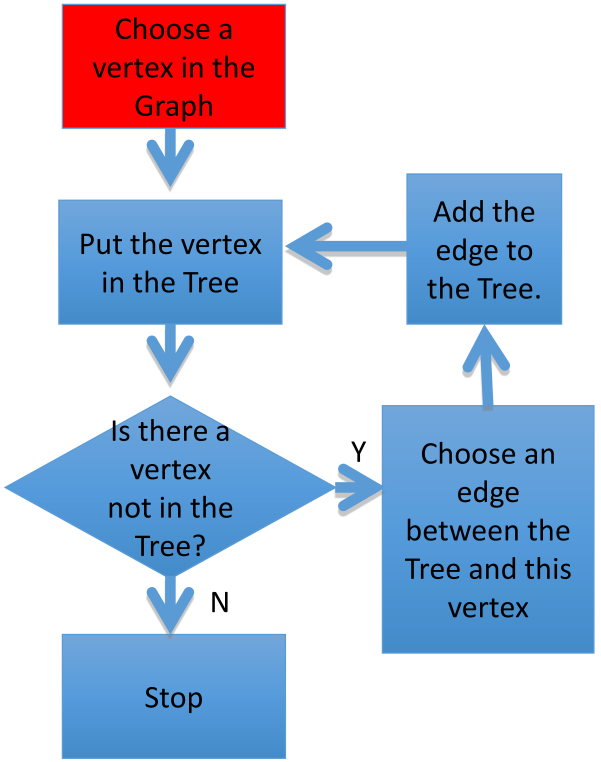


Original Graph – input to Prim's algorithm

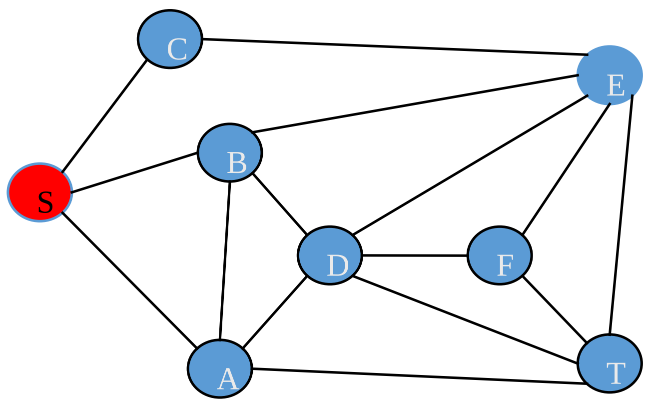


Tree created – output of Prim's algorithm

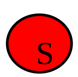
Prim's algorithm



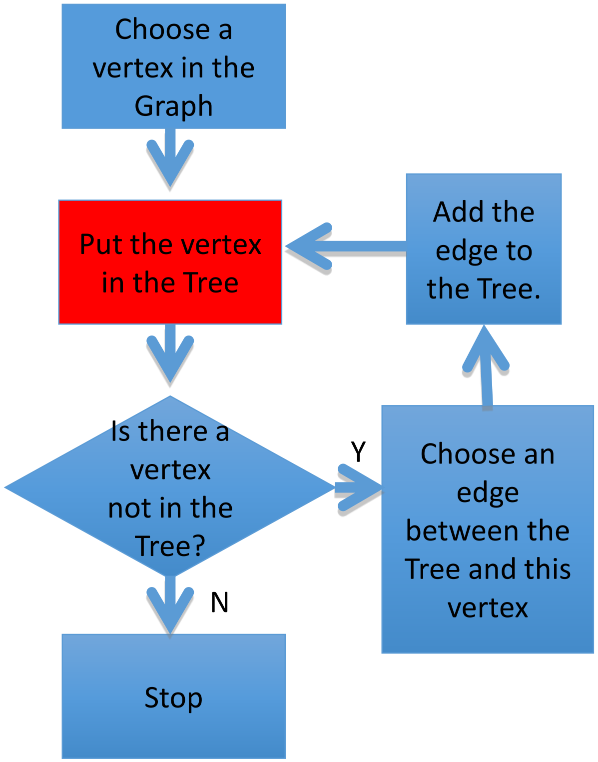
Original Graph – input to Prim's algorithm



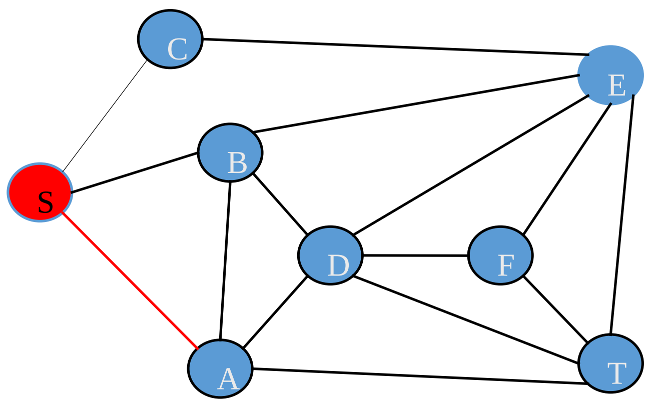
Tree created – output of Prim's algorithm



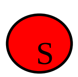
Prim's algorithm



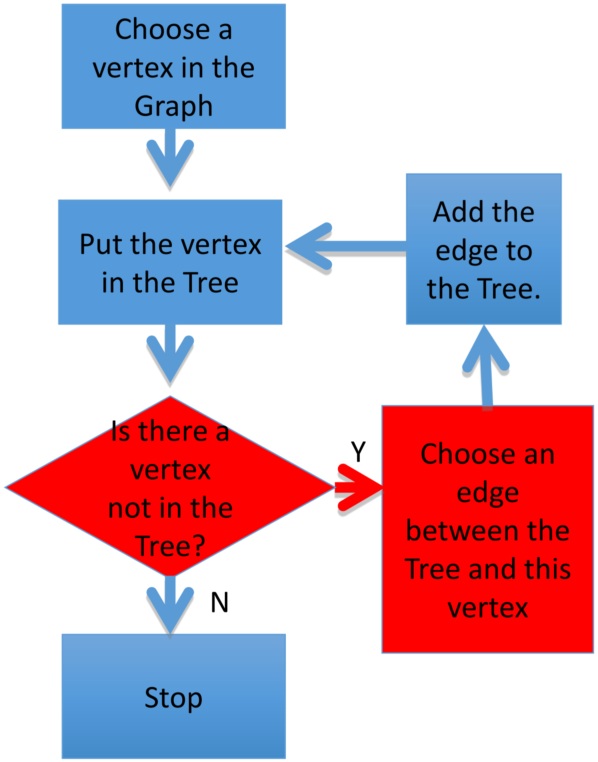
Original Graph – input to Prim's algorithm



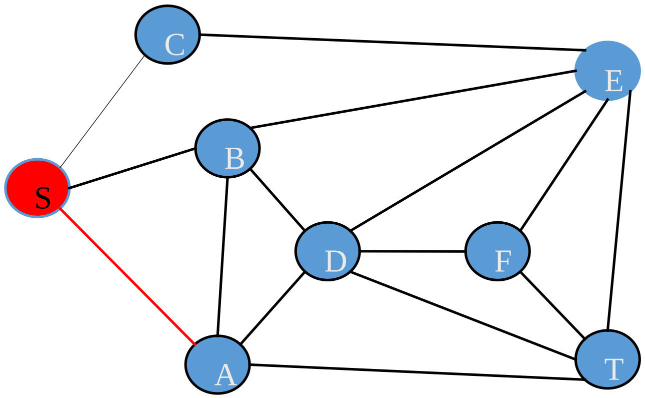
Tree created – output of Prim's algorithm



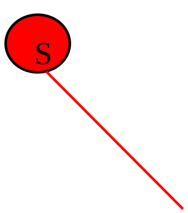
Prim's algorithm



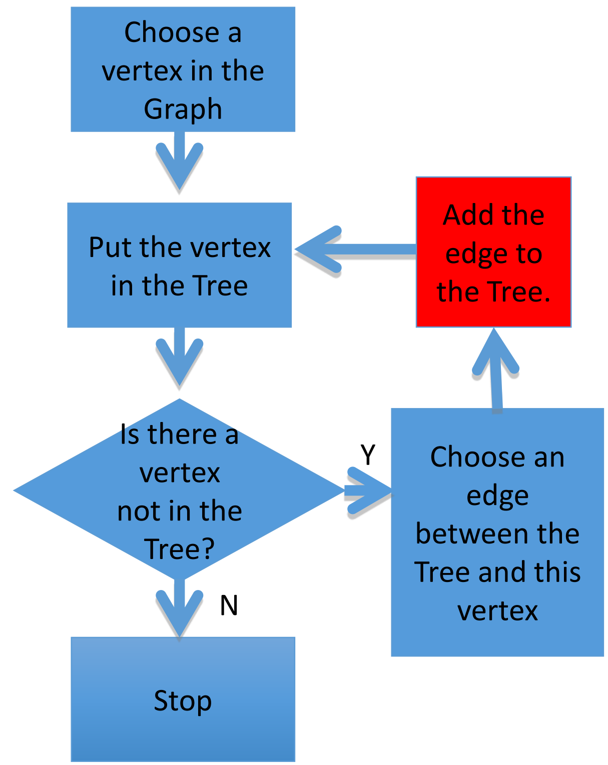
Original Graph – input to Prim's algorithm



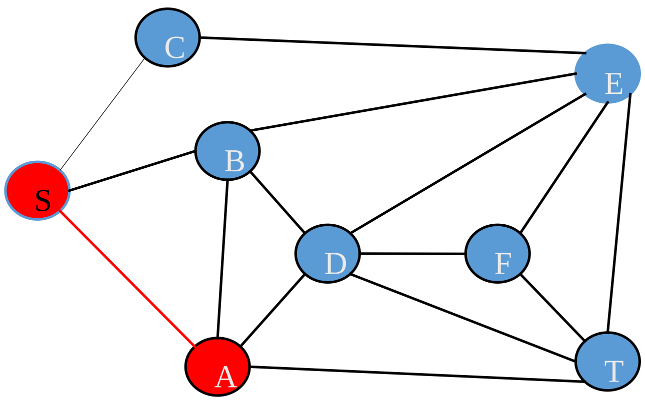
Tree created – output of Prim's algorithm



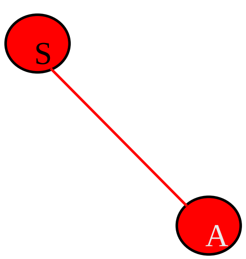
Prim's algorithm



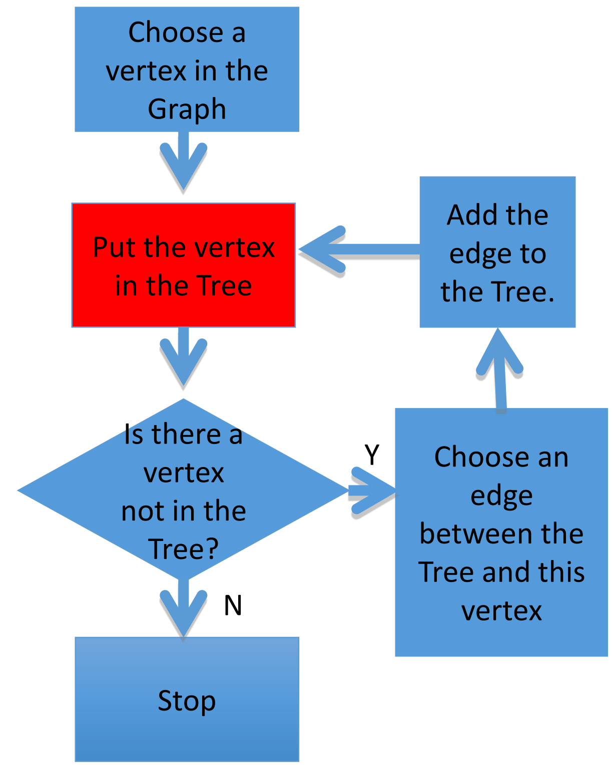
Original Graph – input to Prim's algorithm



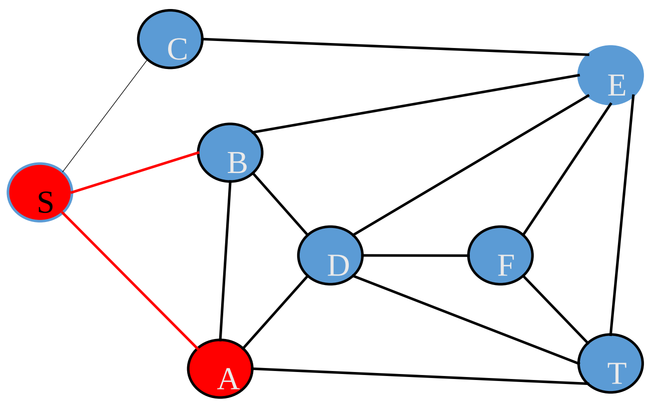
Tree created – output of Prim's algorithm



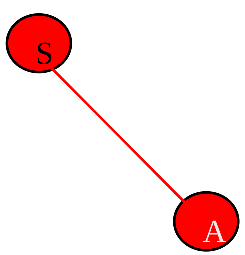
Prim's algorithm



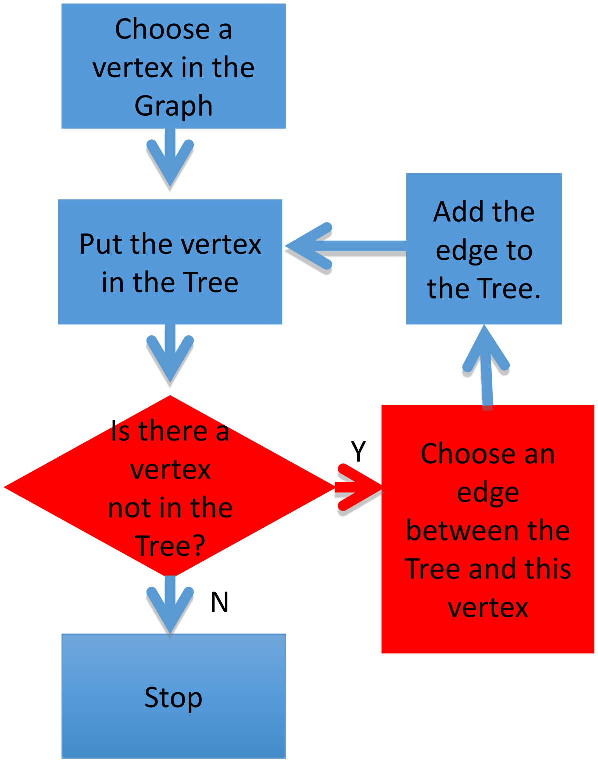
Original Graph – input to Prim's algorithm



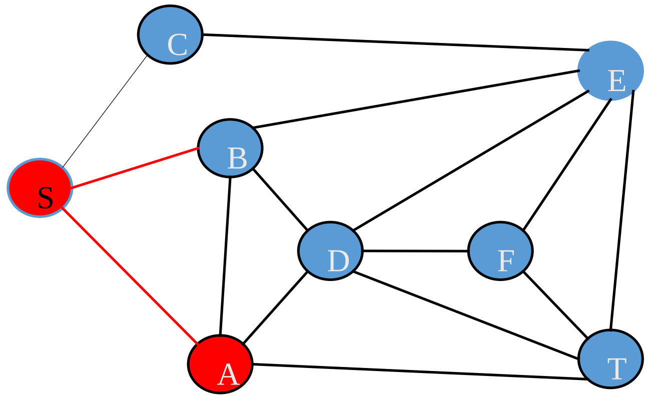
Tree created – output of Prim's algorithm



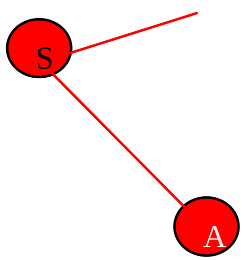
Prim's algorithm



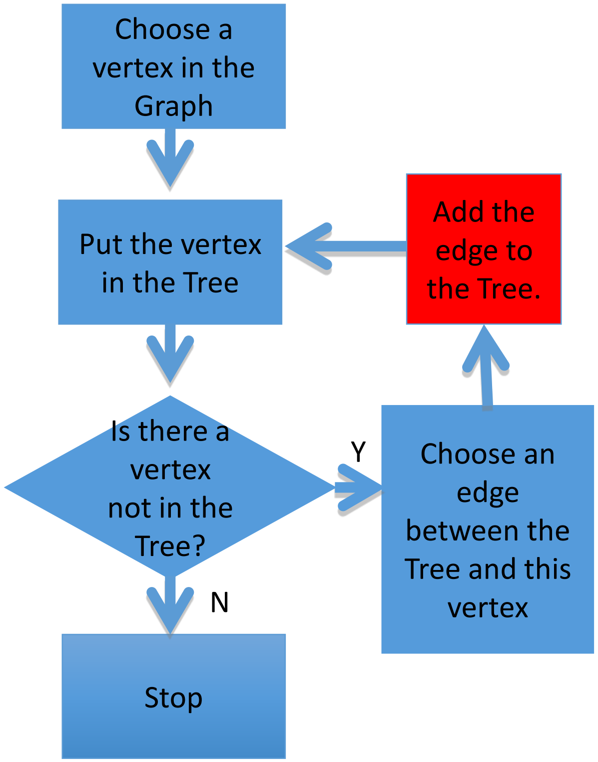
Original Graph – input to Prim's algorithm



Tree created – output of Prim's algorithm

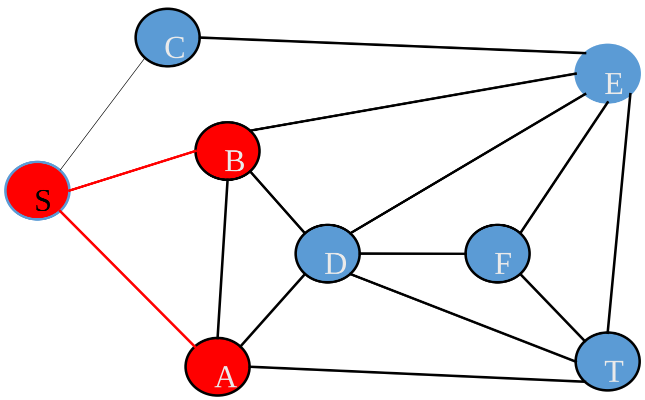


Prim's algorithm

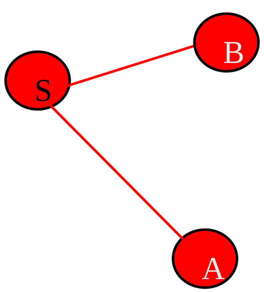


MONASH College

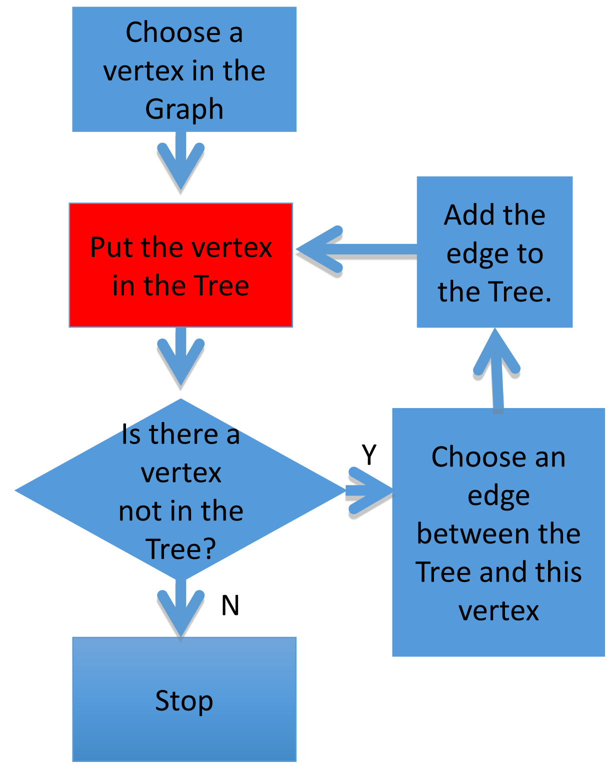
Original Graph – input to Prim's algorithm



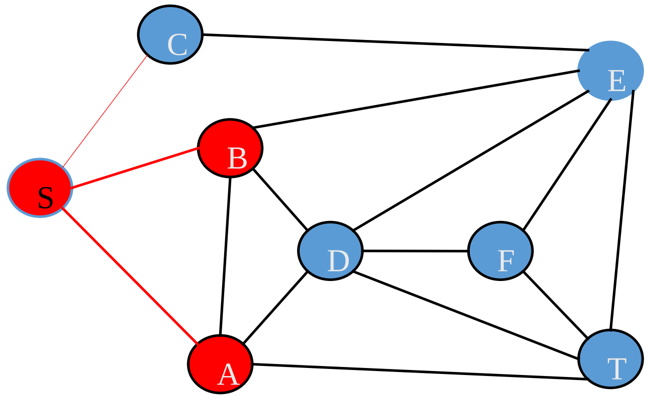
Tree created – output of Prim's algorithm



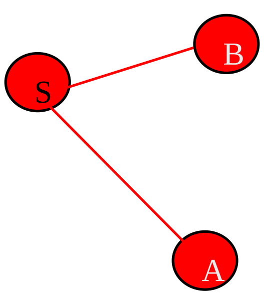
Prim's algorithm



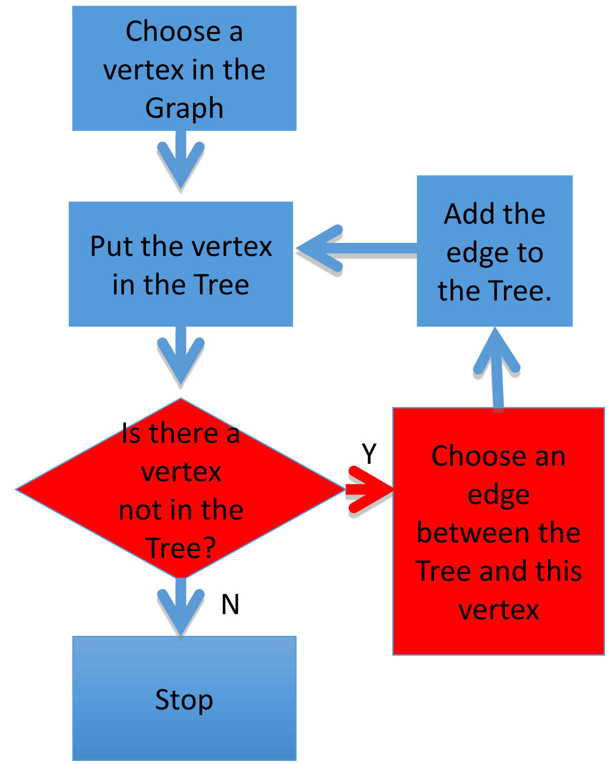
Original Graph – input to Prim’s algorithm



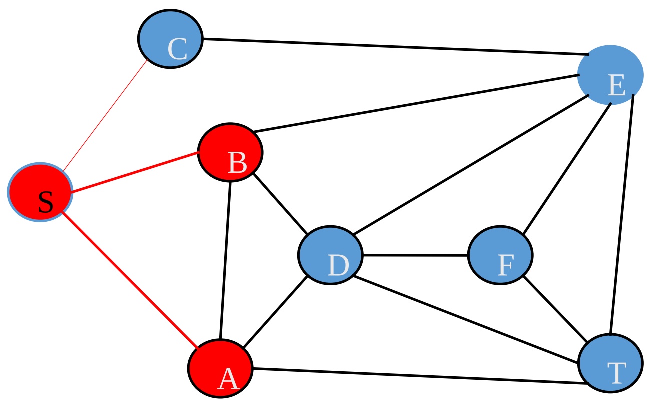
Tree created – output of Prim’s algorithm



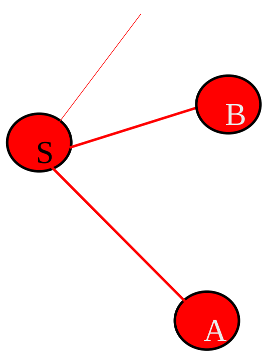
Prim’s algorithm



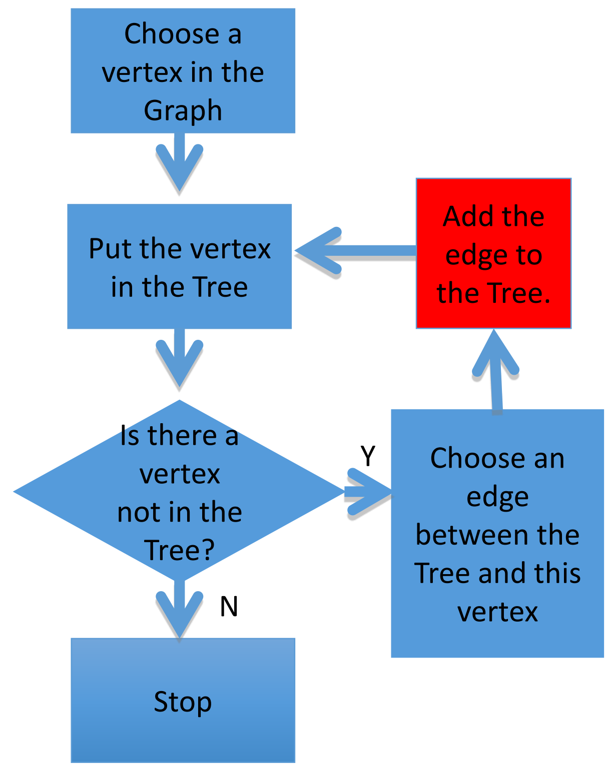
Original Graph – input to Prim's algorithm



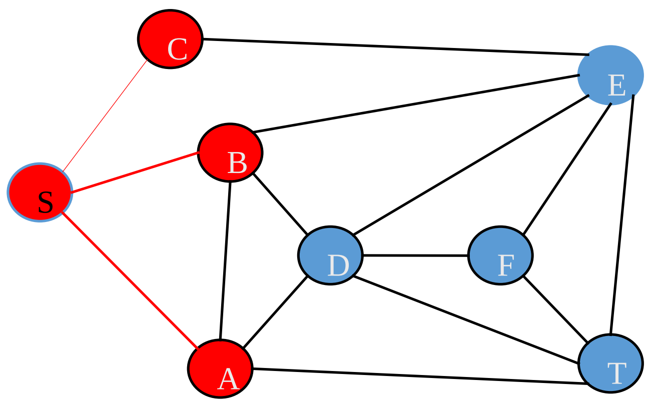
Tree created – output of Prim's algorithm



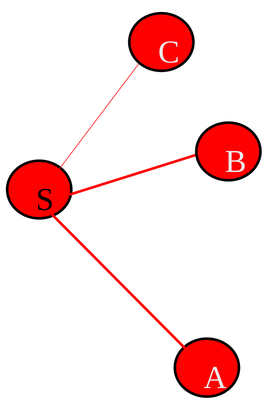
Prim's algorithm



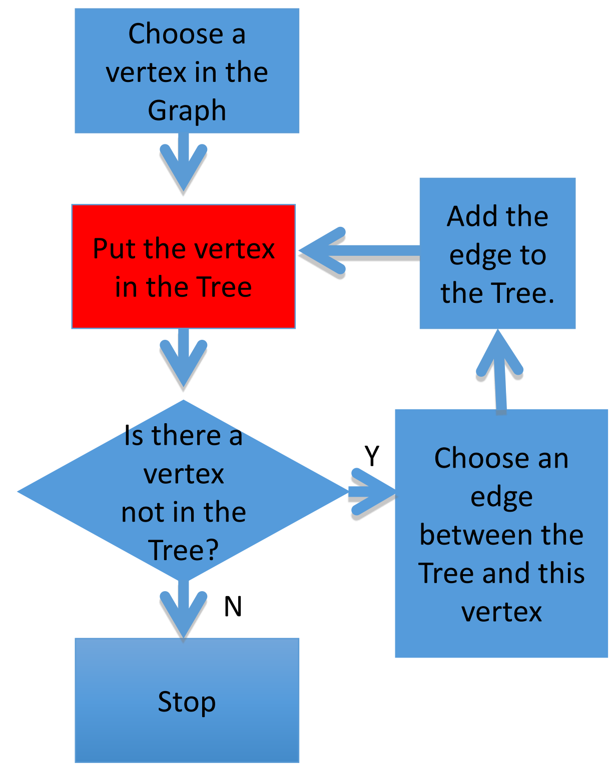
Original Graph – input to Prim's algorithm



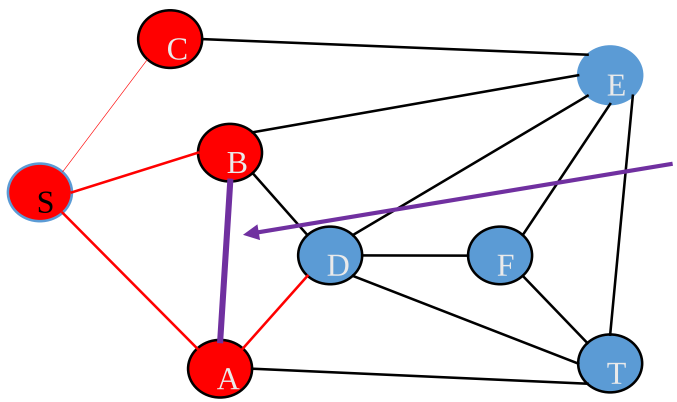
Tree created – output of Prim's algorithm



Prim's algorithm

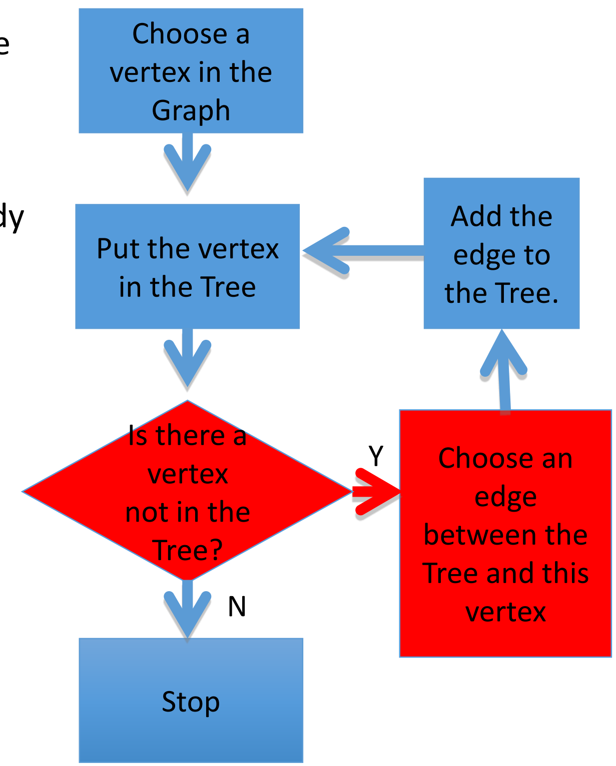


Original Graph – input to Prim's algorithm

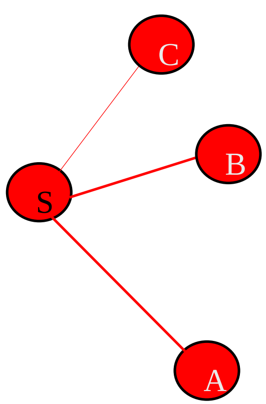


This edge creates a cycle because vertex B is already added to the tree.
So we should pick an edge that connects to a vertex that's already not in the tree

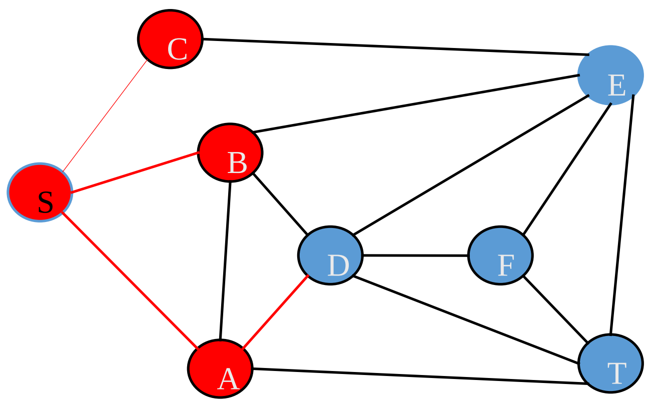
Prim's algorithm



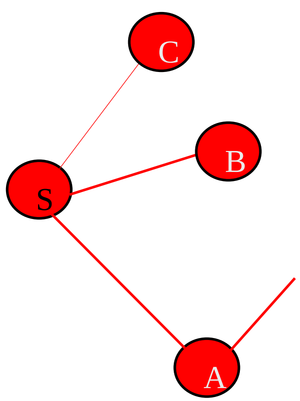
Tree created – output of Prim's algorithm



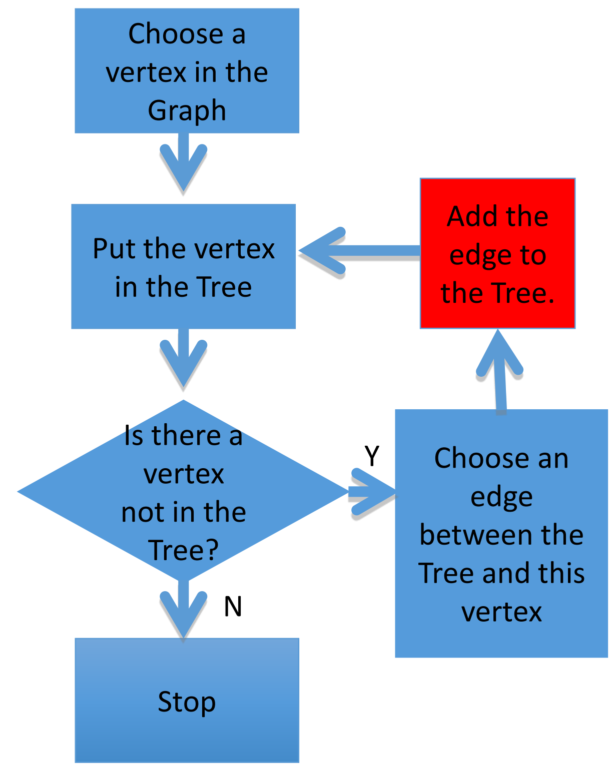
Original Graph – input to Prim's algorithm



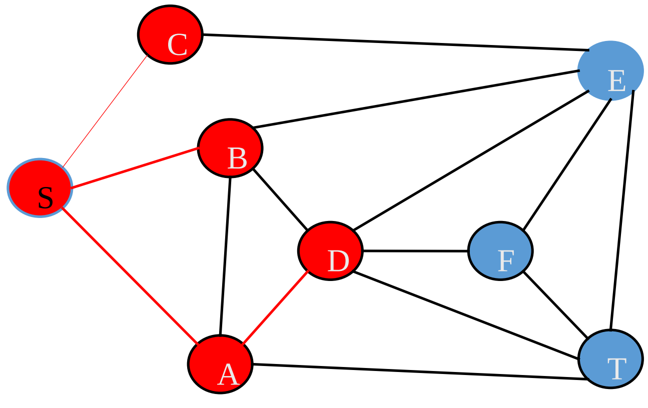
Tree created – output of Prim's algorithm



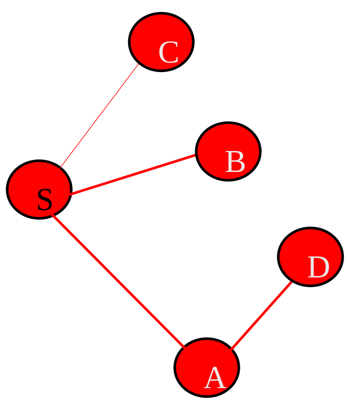
Prim's algorithm



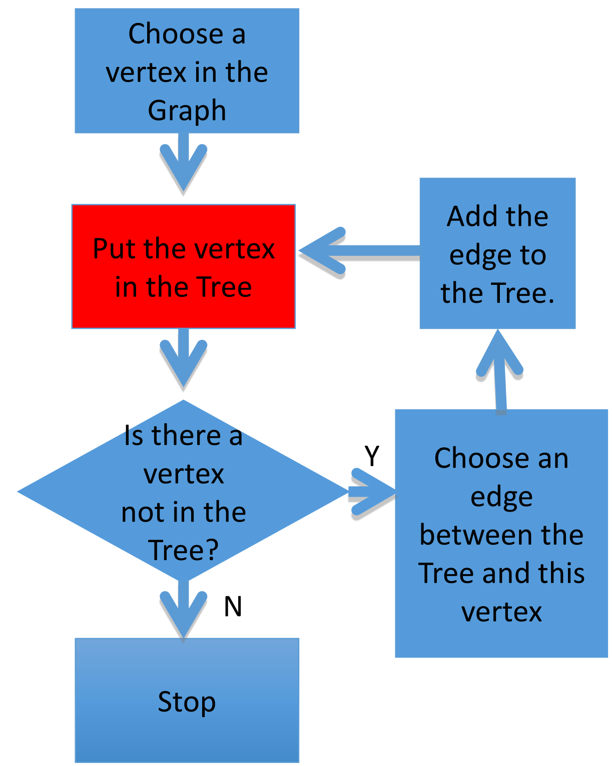
Original Graph – input to Prim's algorithm



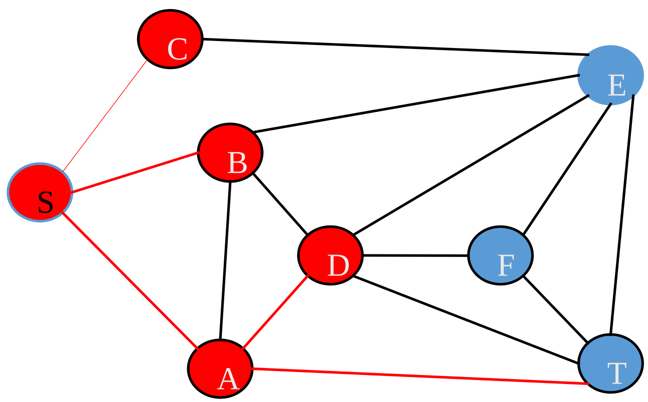
Tree created – output of Prim's algorithm



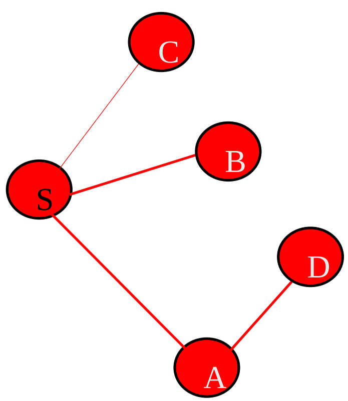
Prim's algorithm



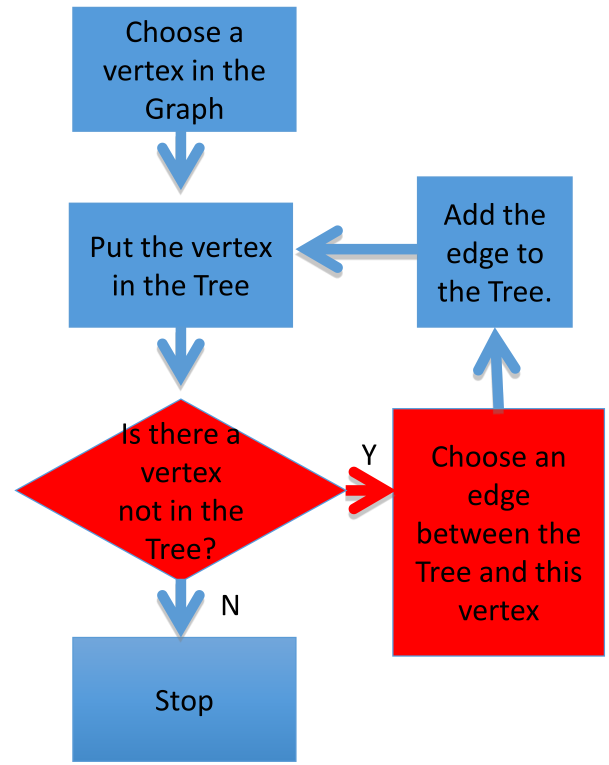
Original Graph – input to Prim's algorithm



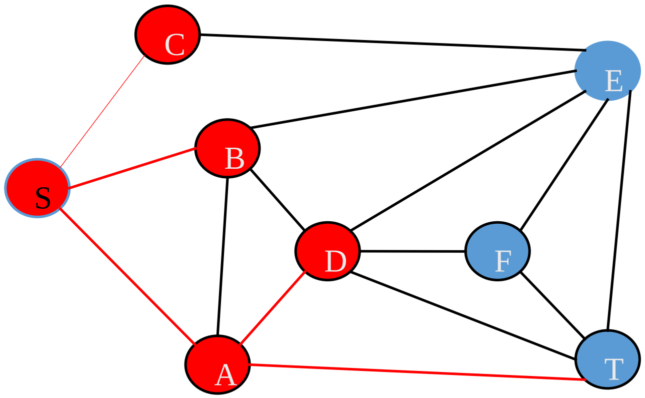
Tree created – output of Prim's algorithm



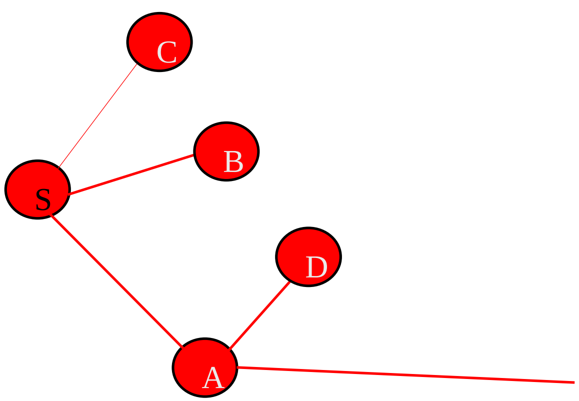
Prim's algorithm



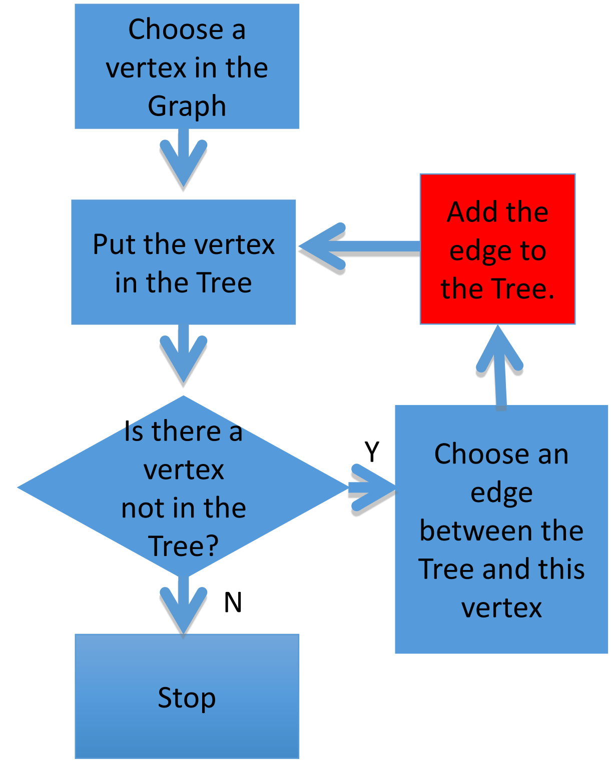
Original Graph – input to Prim's algorithm



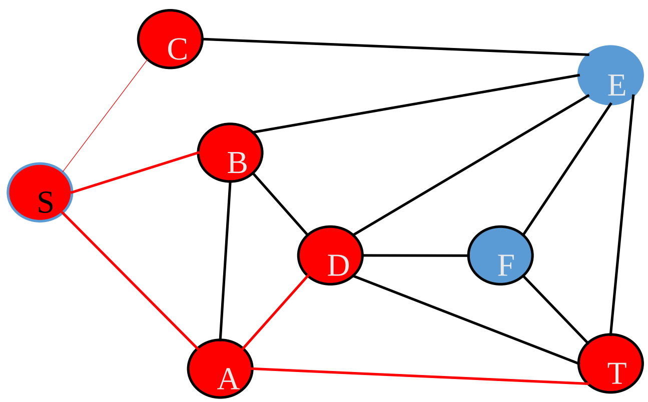
Tree created – output of Prim's algorithm



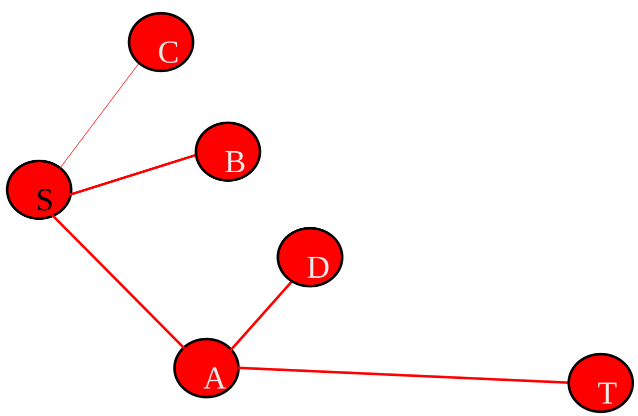
Prim's algorithm



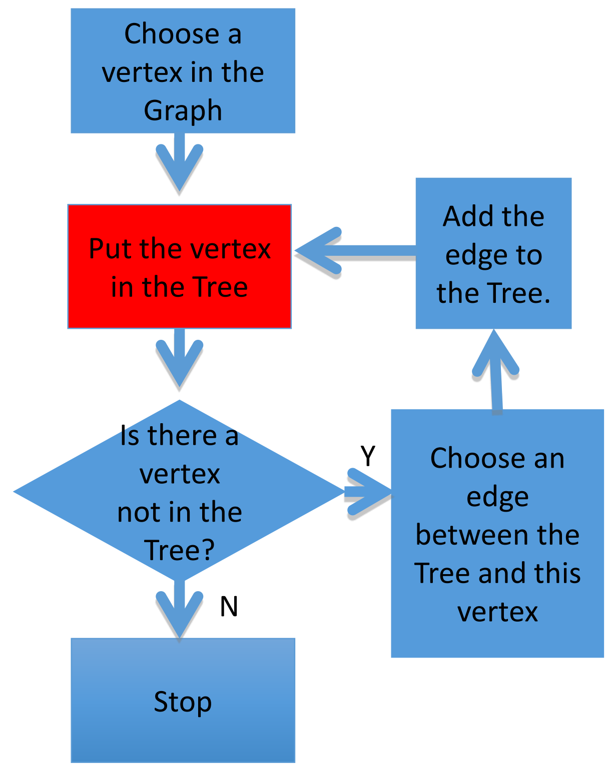
Original Graph – input to Prim's algorithm



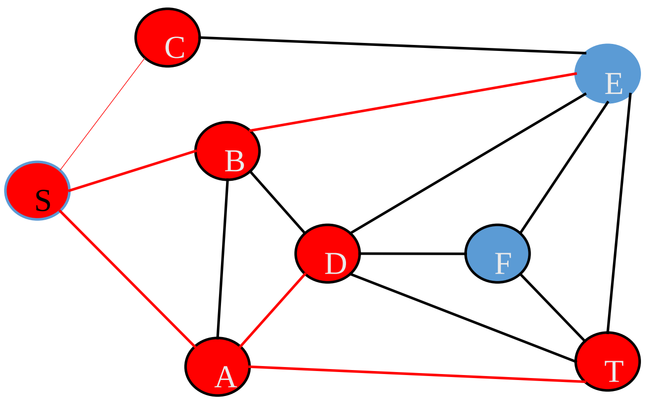
Tree created – output of Prim's algorithm



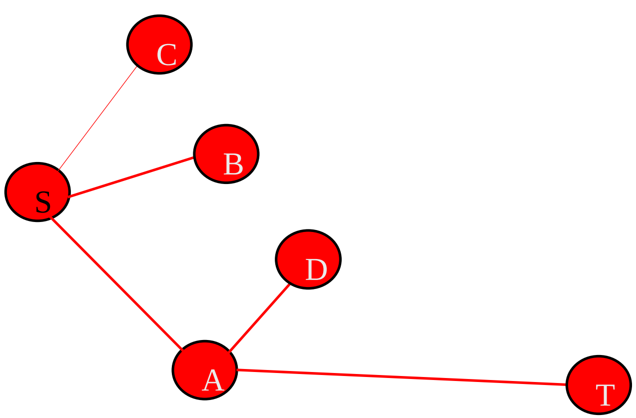
Prim's algorithm



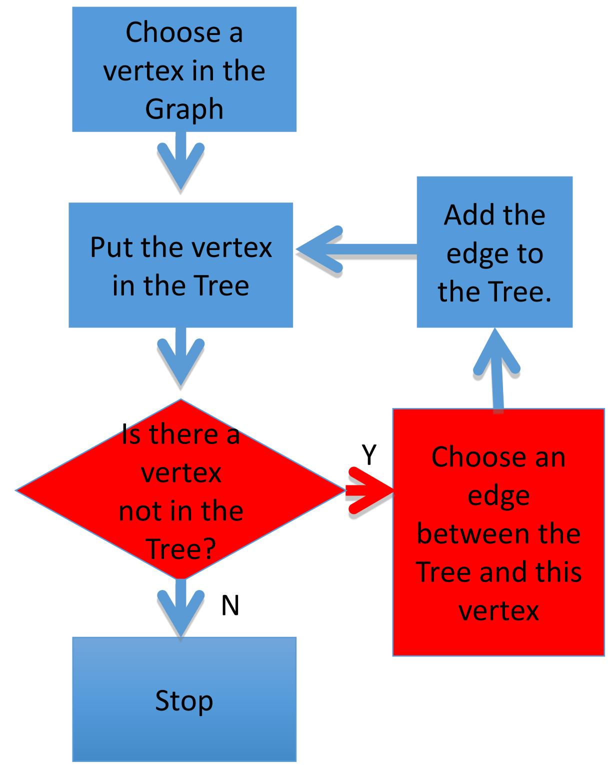
Original Graph – input to Prim’s algorithm



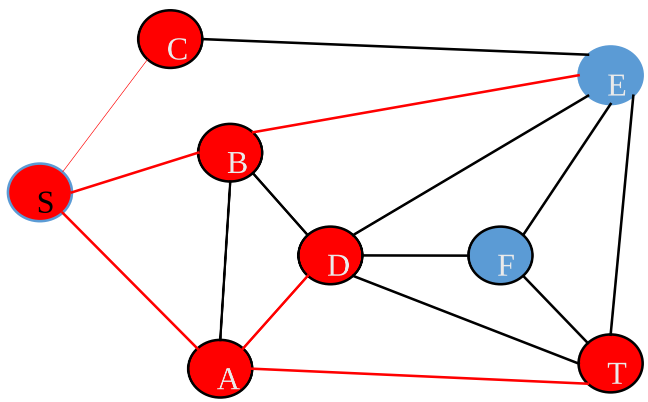
Tree created – output of Prim’s algorithm



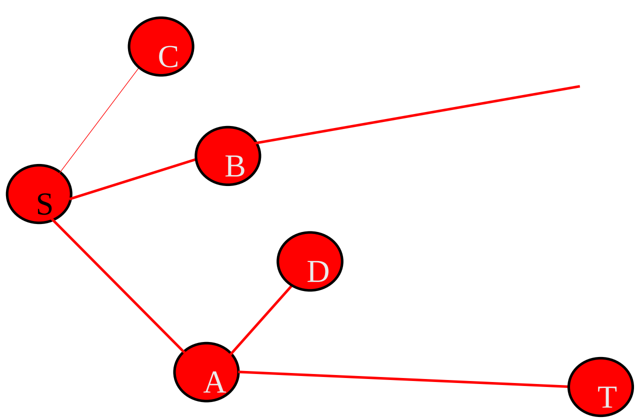
Prim’s algorithm



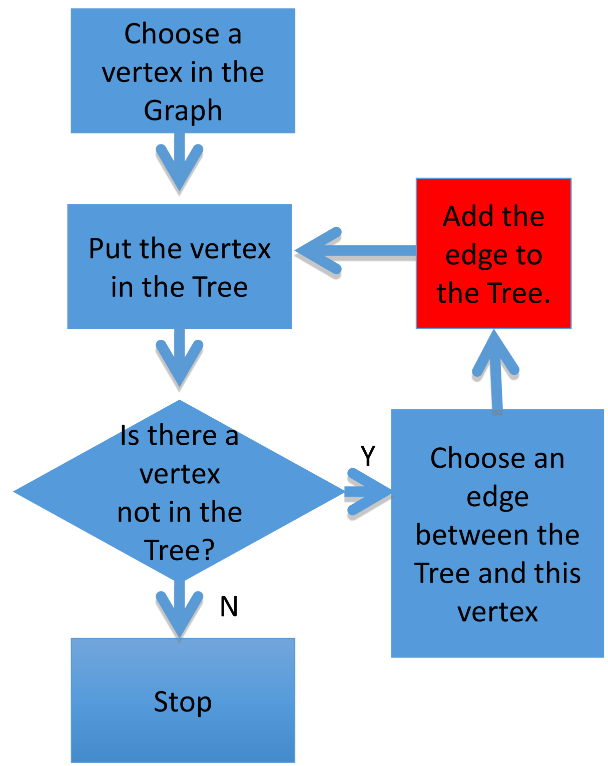
Original Graph – input to Prim's algorithm



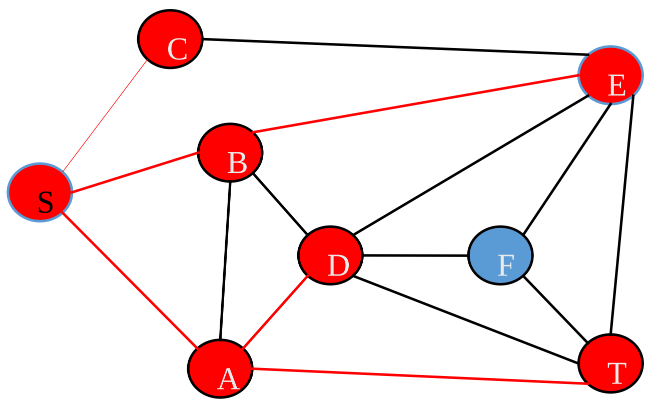
Tree created – output of Prim's algorithm



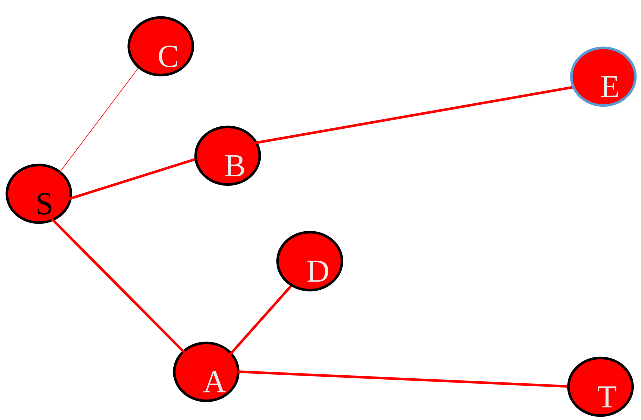
Prim's algorithm



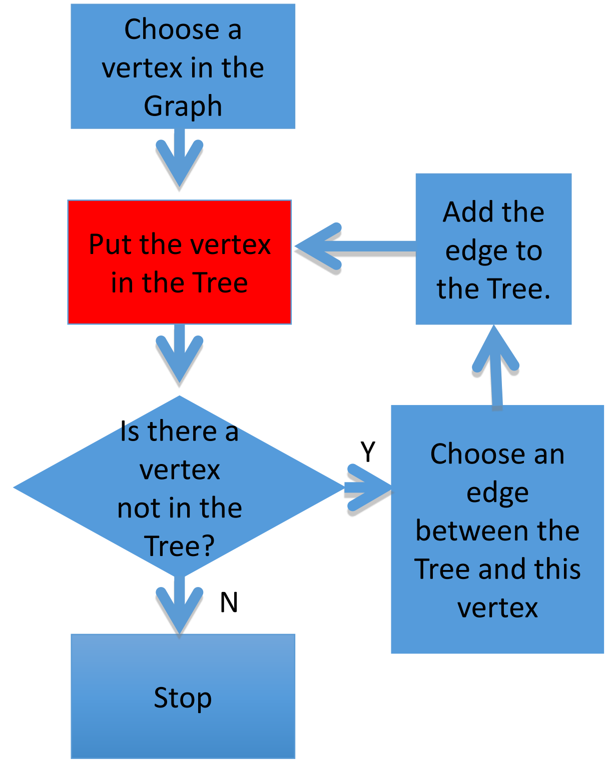
Original Graph – input to Prim's algorithm



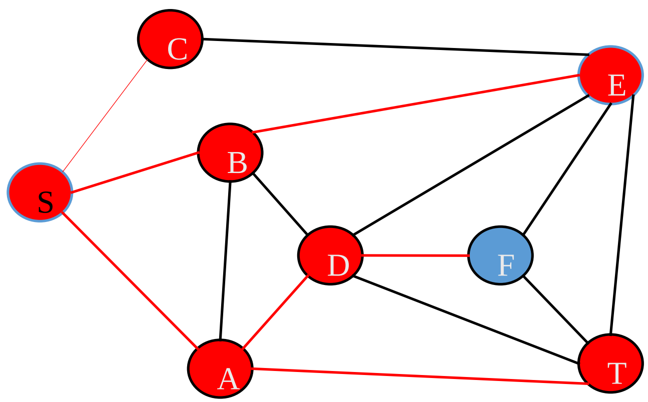
Tree created – output of Prim's algorithm



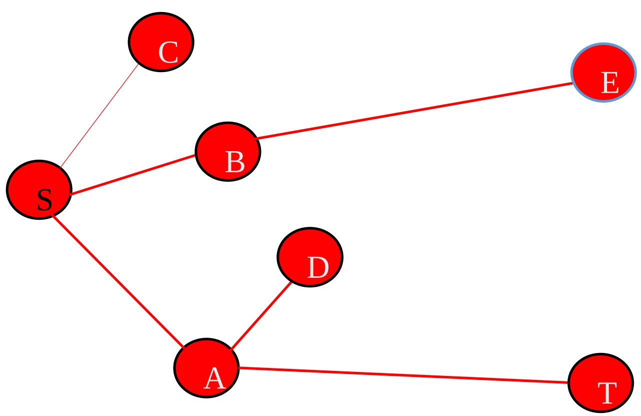
Prim's algorithm



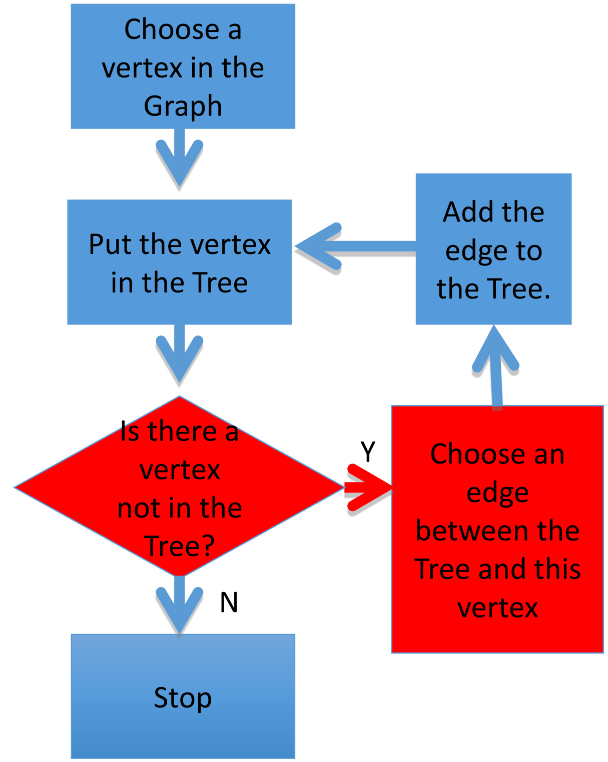
Original Graph – input to Prim's algorithm



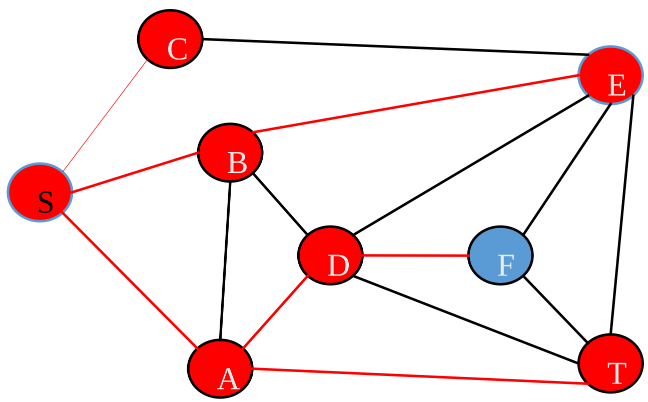
Tree created – output of Prim's algorithm



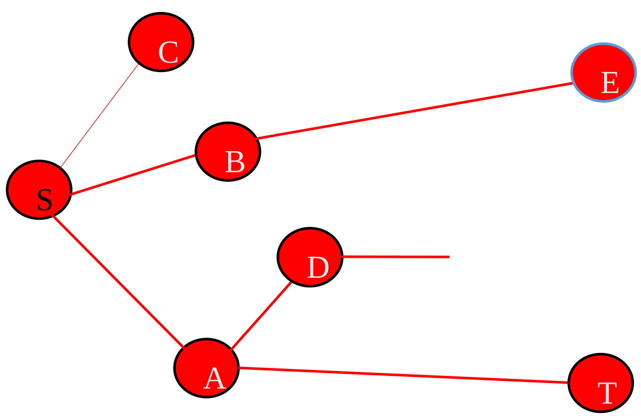
Prim's algorithm



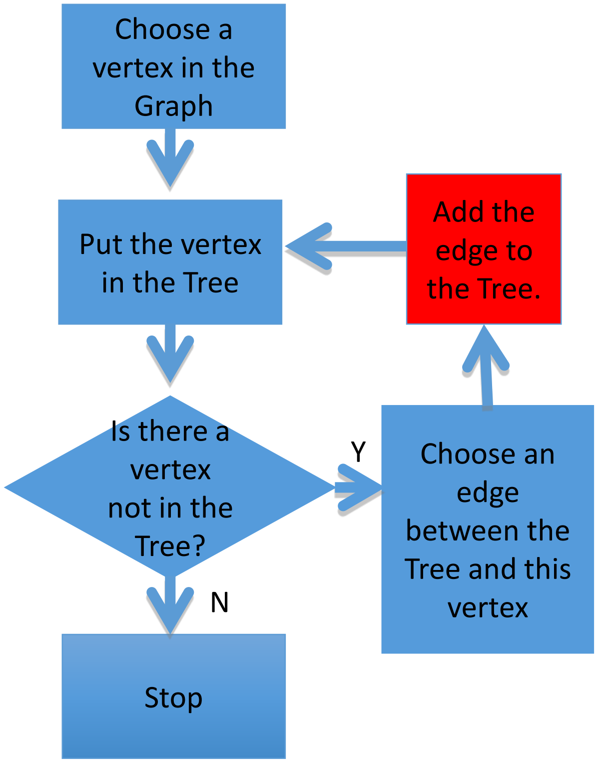
Original Graph – input to Prim's algorithm



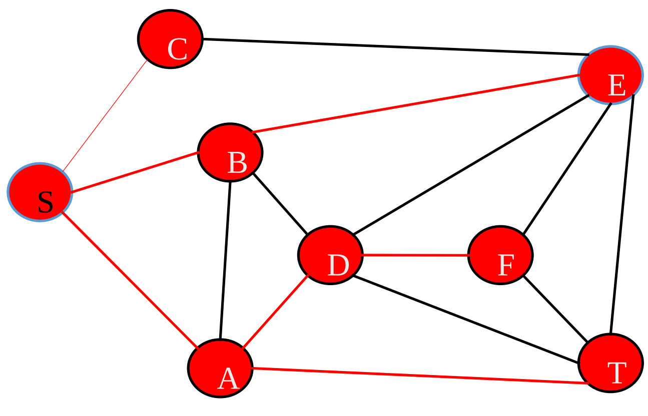
Tree created – output of Prim's algorithm



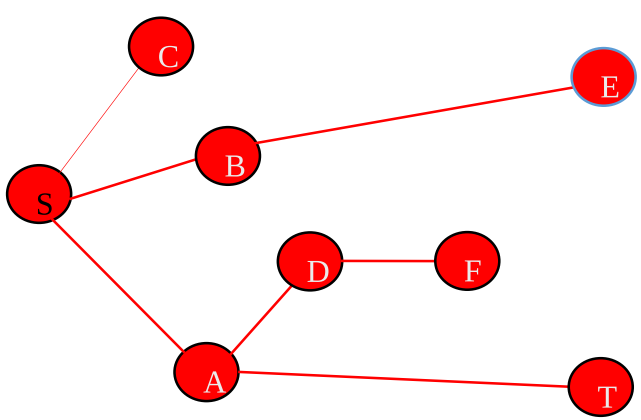
Prim's algorithm



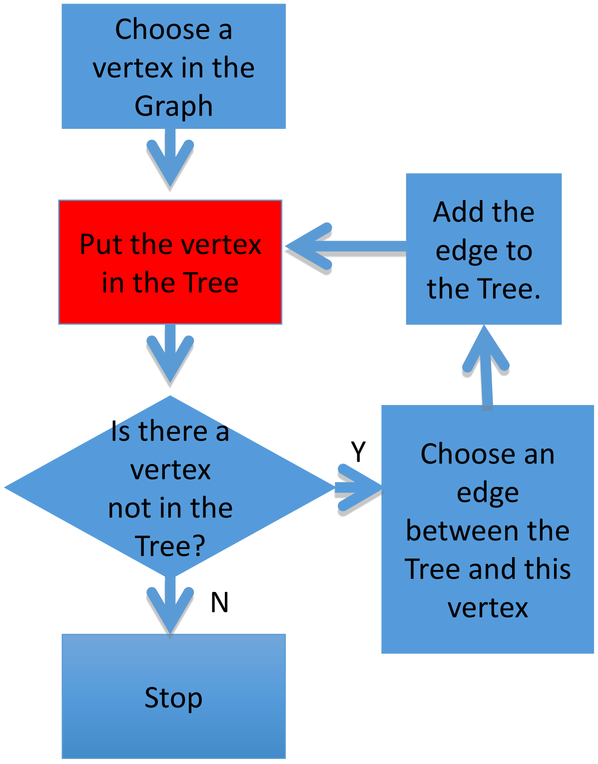
Original Graph – input to Prim's algorithm



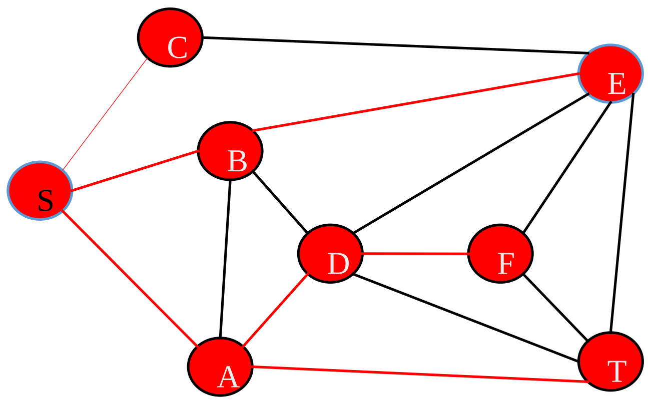
Tree created – output of Prim's algorithm



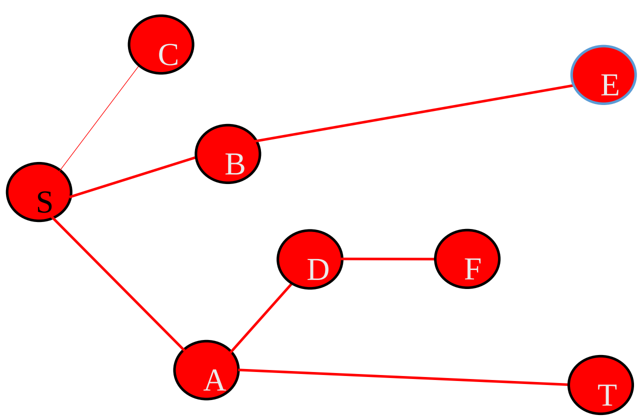
Prim's algorithm



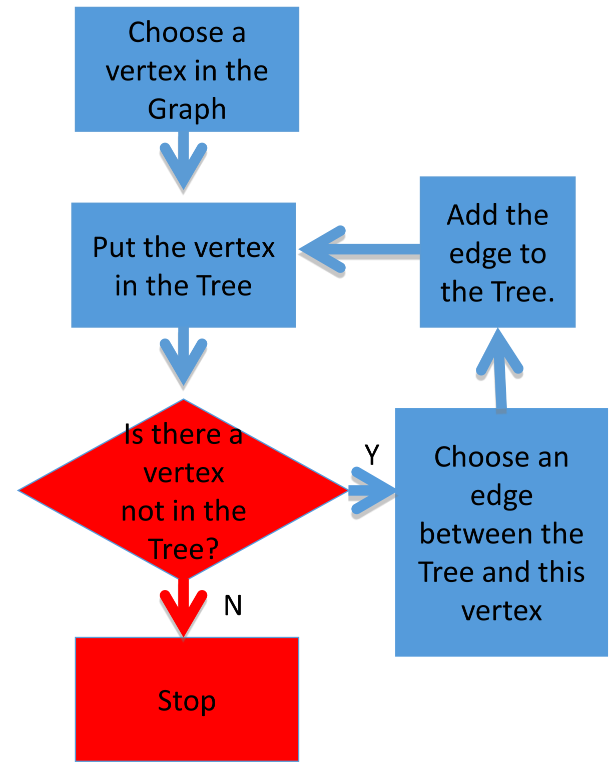
Original Graph – input to Prim’s algorithm



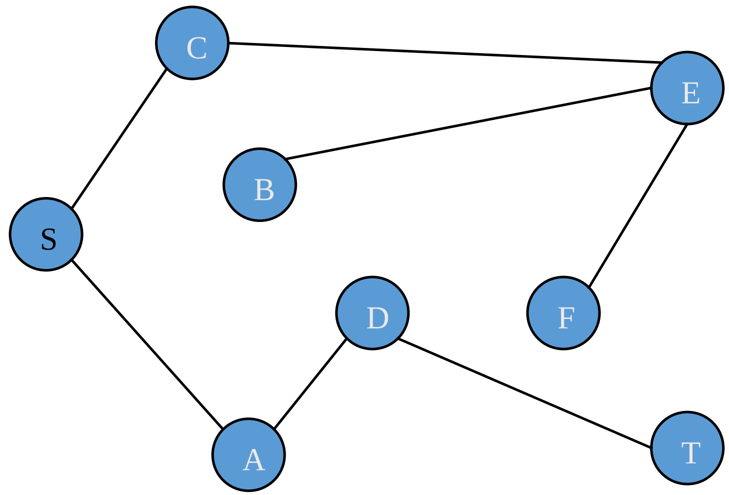
Tree created – output of Prim’s algorithm



Prim’s algorithm



Another Spanning Tree of a same graph



- If we assume that G is simple and connected then a spanning tree of G is a tree which:
  - Contains all the vertices of G (Spans G), and
  - The edges of the tree are edges of G (Subgraph of G).

Prim's algorithm in Python

```
def spanning_tree(graph):
    """Input : adjacency matrix of graph
       Output: adj. mat. of spanning tree of graph"""
```

Prim’s algorithm in Python

```
def spanning_tree(graph):
    """Input : adjacency matrix of graph
       Output: adj. mat. of spanning tree of graph"""
    n = len(graph)
    tree = empty_graph(n)
```

initialise result accumulation variable

have to implement function that creates n-by-n adjacency matrix with all entries zero

```
    return tree
```

Prim’s algorithm in Python

```
def spanning_tree(graph):
    """Input : adjacency matrix of graph
       Output: adj. mat. of spanning tree of graph"""
    n = len(graph)
    tree = empty_graph(n)
    conn = {0}
```

auxiliary accumulation variable that keeps track of already connected vertices

```
    return tree
```

Prim’s algorithm in Python

```
def spanning_tree(graph):
    """Input : adjacency matrix of graph
       Output: adj. mat. of spanning tree of graph"""
    n = len(graph)
    tree = empty_graph(n)
    conn = {0}
    while len(conn) < n:












    return tree
```

Prim’s algorithm in Python

```
def spanning_tree(graph):
    """Input : adjacency matrix of graph
       Output: adj. mat. of spanning tree of graph"""
    n = len(graph)
    tree = empty_graph(n)
    conn = {0}
    while len(conn) < n:

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









    return tree
```

iterate over all possible “extension edges”

Prim’s algorithm in Python

```
def spanning_tree(graph):
    """Input : adjacency matrix of graph
       Output: adj. mat. of spanning tree of graph"""
    n = len(graph)
    tree = empty_graph(n)
    conn = {0}
    while len(conn) < n:

        for i in conn:
            for j in range(n):
                if j not in conn and graph[i][j]==1:







    return tree
```

check if found an extension edge

Prim’s algorithm in Python

```
def spanning_tree(graph):
    """Input : adjacency matrix of graph
       Output: adj. mat. of spanning tree of graph"""
    n = len(graph)
    tree = empty_graph(n)
    conn = {0}
    while len(conn) < n:

        for i in conn:
            for j in range(n):
                if j not in conn and graph[i][j]==1:
                    tree[i][j] = 1
                    tree[j][i] = 1
                    conn = conn.add(j)

    return tree
```

add found edge to result adj. mat. and newly connected vertex to conn

Prim’s algorithm in Python

```
def spanning_tree(graph):
    """Input : adjacency matrix of graph
       Output: adj. mat. of spanning tree of graph"""
    n = len(graph)
    tree = empty_graph(n)
    conn = {0}
    while len(conn) < n:

        for i in conn:
            for j in range(n):
                if j not in conn and graph[i][j]==1:
                    tree[i][j] = 1
                    tree[j][i] = 1
                    conn = conn.add(j)

    return tree
```

now want to jump back to head of while-loop

Prim’s algorithm in Python

```
def spanning_tree(graph):
    """Input : adjacency matrix of graph
       Output: adj. mat. of spanning tree of graph"""
    n = len(graph)
    tree = empty_graph(n)
    conn = {0}
    while len(conn) < n:

        for i in conn:
            for j in range(n):
                if j not in conn and graph[i][j]==1:
                    tree[i][j] = 1
                    tree[j][i] = 1
                    conn = conn.add(j)

                    break


    return tree
```

single break statement only gets us back to start of first for-loop

Prim’s algorithm in Python

```
def spanning_tree(graph):
    """Input : adjacency matrix of graph
       Output: adj. mat. of spanning tree of graph"""
    n = len(graph)
    tree = empty_graph(n)
    conn = {0}
    while len(conn) < n:
        found = False
        for i in conn:
            for j in range(n):
                if j not in conn and graph[i][j]==1:
                    tree[i][j] = 1
                    tree[j][i] = 1
                    conn = conn.add(j)
                    found = True
                    break
            if found:
                break
    return tree
```

Decomposition...

...can be thought of from two perspectives:

1. Breaking down programs into sub-programs (components)
2. Breaking down problems into sub-problems

...is ***most useful if two views coincide***, i.e., sub-programs correspond to sub-problems

- structures thinking/attention for developing algorithms and *reasoning* about programs
- leads to *re-usable components* (because they solve a well-defined problem)

Our main tool for decomposition are functions!

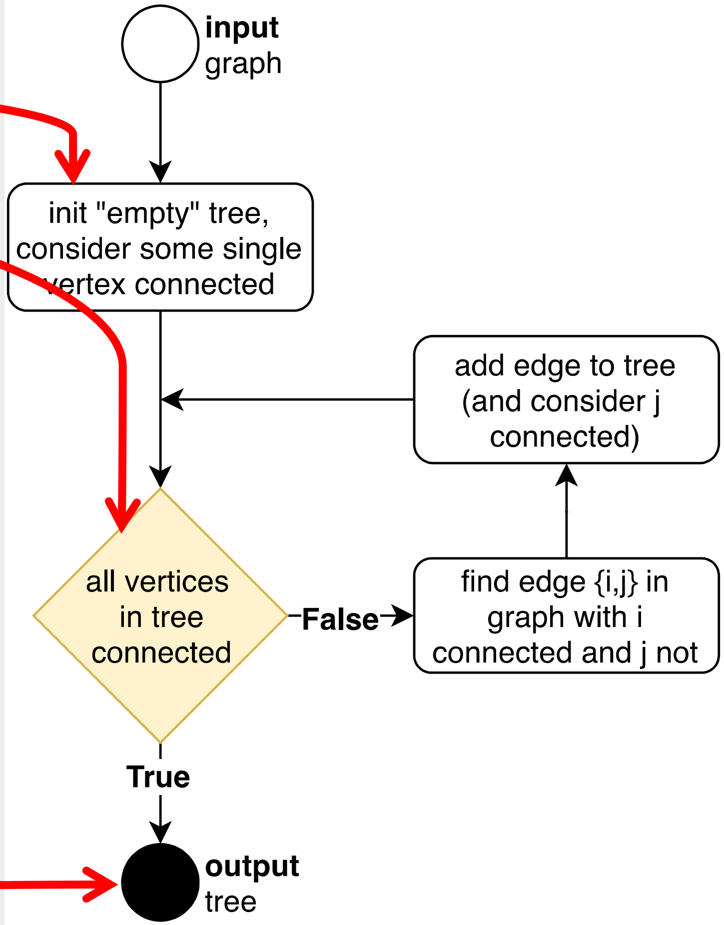
Prim's algorithm in Python – decomposition edition

MONASH College

```
def spanning_tree(graph):
    """Input : adjacency matrix of graph
       Output: adj. mat. of spanning tree of graph"""
    n = len(graph)
    tree = empty_graph(n)
    conn = {0}
    while len(conn) < n:
        found = False
        for i in conn:
            for j in range(n):
                if j not in conn and graph[i][j]==1:
                    tree[i][j] = 1
                    tree[j][i] = 1
                    conn = conn.add(j)
                    found = True
                    break
            if found:
                break
    return tree
```

How did simple flowchart turn into complicated code?

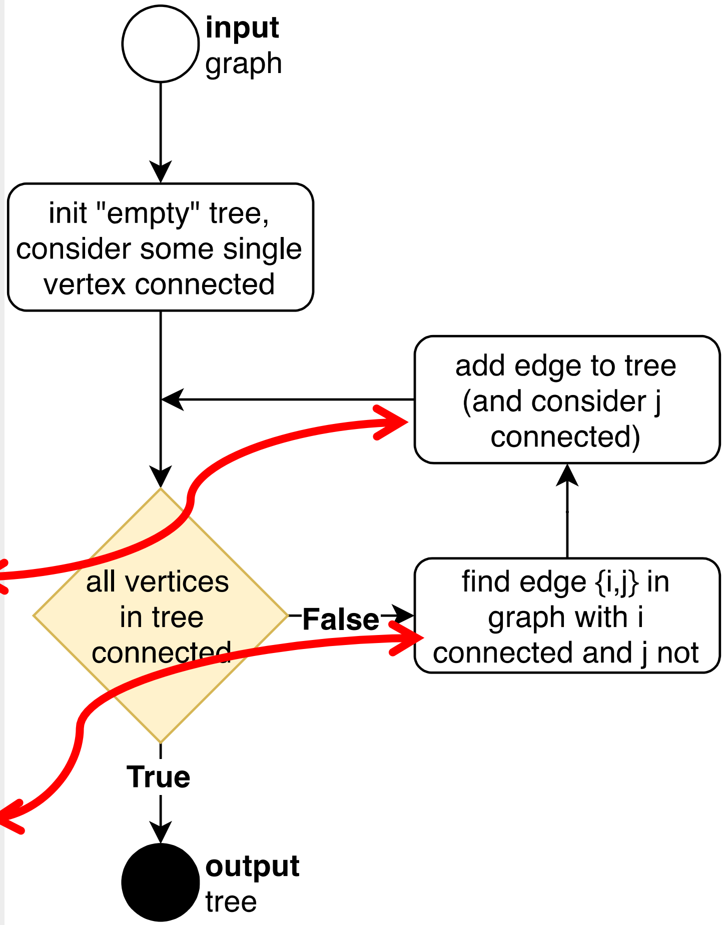
```
n = len(graph)
tree = empty_graph(n)
conn = {0}
while len(conn) < n:
    found = False
    for i in conn:
        for j in range(n):
            if j not in conn and \
            graph[i][j]==1:
                tree[i][j] = 1
                tree[j][i] = 1
                conn = conn.add(j)
                found = True
                break
            if found:
                break
return tree
```



Some lines can be cleanly mapped to high-level instructions

How did simple flowchart turn into complicated code?

```
n = len(graph)
tree = empty_graph(n)
conn = {0}
while len(conn) < n:
    found = False
    for i in conn:
        for j in range(n):
            if j not in conn and \
            graph[i][j]==1:
                tree[i][j] = 1
                tree[j][i] = 1
                conn = conn.add(j)
                found = True
                break
            if found:
                break
return tree
```



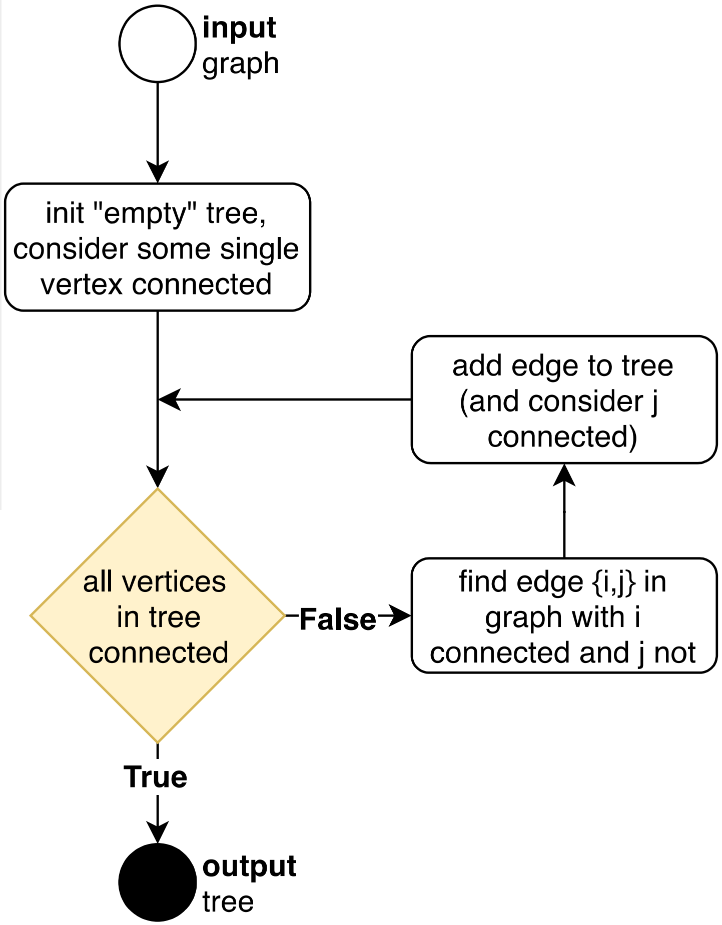
Identification of extension edge is not separated from its addition

Decomposition: factor out extension edge identification

```
n = len(graph)
tree = empty_graph(n)
conn = {0}
while len(conn) < n:




return tree
```



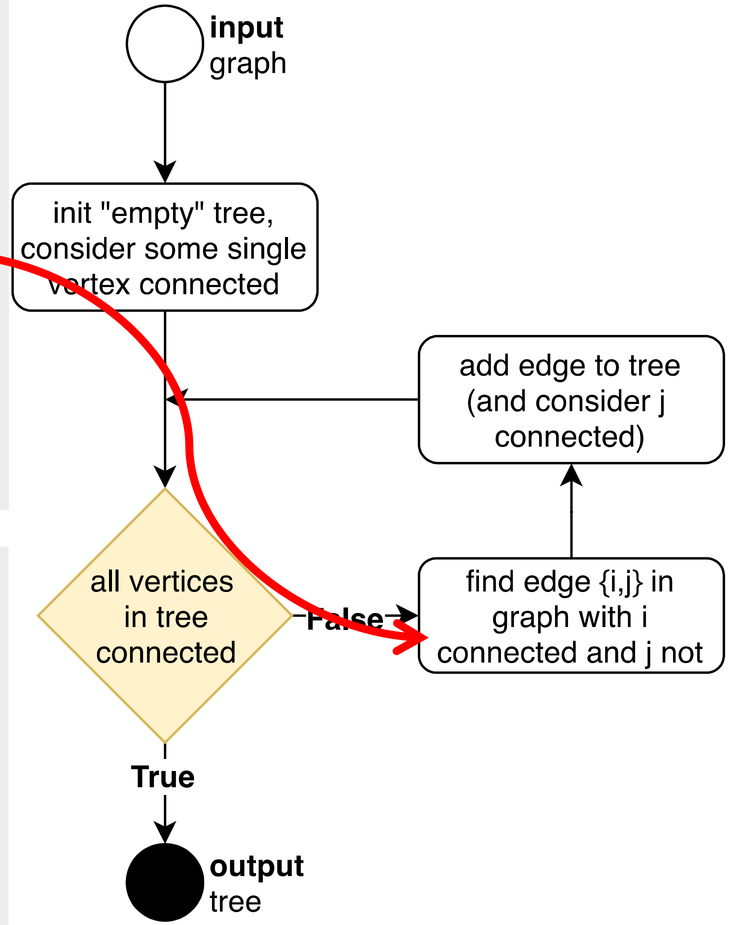
Decomposition: factor out extension edge identification

```
n = len(graph)
tree = empty_graph(n)
conn = {0}
while len(conn) < n:
    i, j = extension(conn, graph)



return tree
```

```
def extension(c, g):
    """I: connect. vertices, graph
       O: extension edge (i, j)"""
```



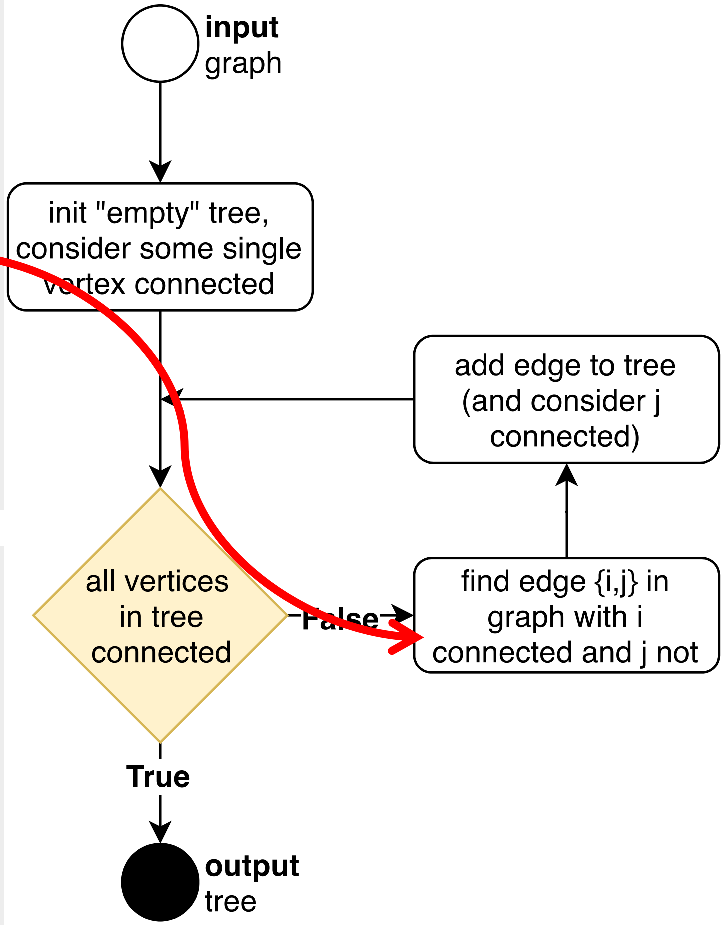
Decomposition: factor out extension edge identification

```
n = len(graph)
tree = empty_graph(n)
conn = {0}
while len(conn) < n:
    i, j = extension(conn, graph)




return tree
```

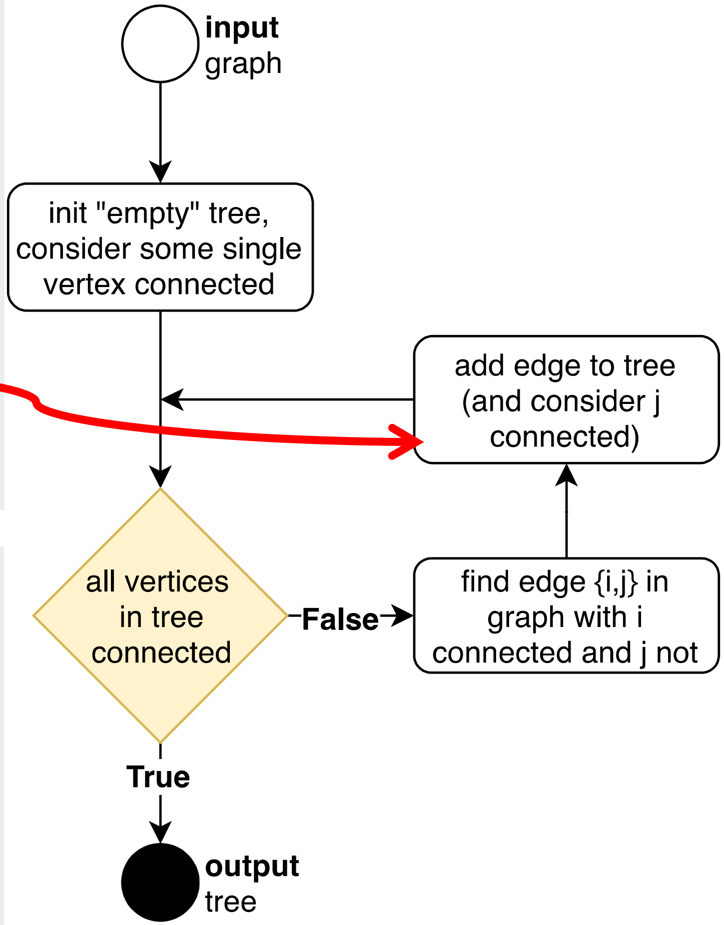
```
def extension(c, g):
    """I: connect. vertices, graph
       O: extension edge (i, j)"""
    n = len(g)
    for i in c:
        for j in range(n):
            if j not in c \
            and g[i][j]:
                return i, j
```



Choose a sub-problem and solve it

```
n = len(graph)
tree = empty_graph(n)
conn = {0}
while len(conn) < n:
    i, j = extension(conn, graph)
    tree[i][j] = 1
    tree[j][i] = 1
    conn.add(j)
return tree
```

```
def extension(c, g):
    """I: connect. vertices, graph
       O: extension edge (i, j)"""
    n = len(g)
    for i in c:
        for j in range(n):
            if j not in c \
            and g[i][j]:
                return i, j
```



Summary

- Graphs are an abstraction of relational data
- Adjacency matrices can be used to represent graphs in Python
- Trees are connected graphs without cycles
- Prim's algorithm finds spanning tree by iteratively adding extension edge to already connected subgraph
- Need to decompose programs to increase readability and simplify analysis