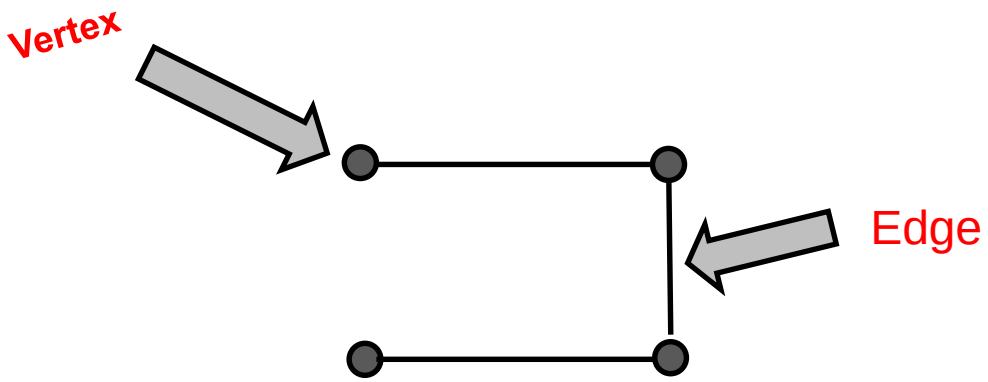
UNIVERSITY OF
LEICESTER

CO1107
Data Structure

What is Graph (Formal Notation)

- A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges. [1]
- A graph G = (V, E) is defined using a set of **vertices V** and a set of **edges E**.
- An edge e is represented as e = (u, v) where u and v are two vertices



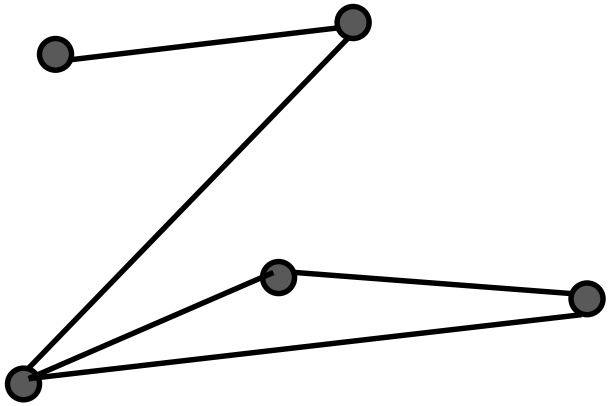
[1] https://www.tutorialspoint.com/python_data_structure/python_graphs.htm

Real world examples of Graph

- the London tube map (what are the nodes? what are the edges?)
- a “friendship” relation between students in a yeargroup
- the motorway map of the UK
- Almost anything can be viewed as a graph; a graph is a very general data structure.

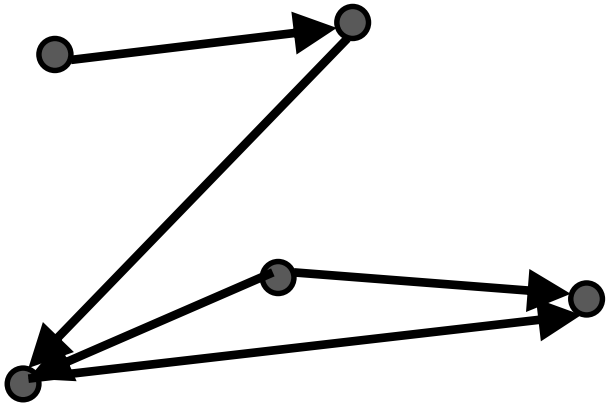
Different Types of Graph

- Undirected Graph



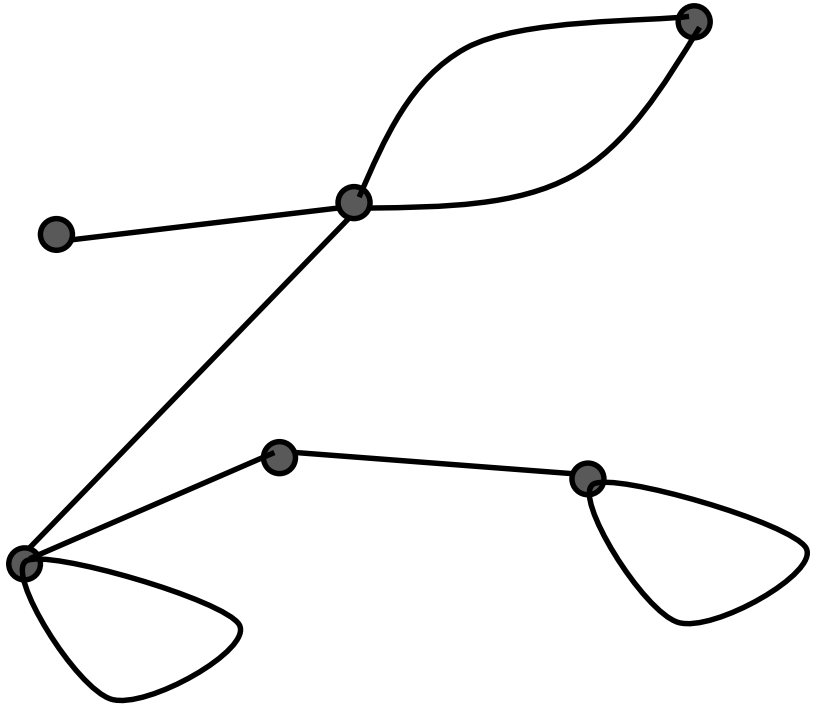
Different Types of Graph

- Directed Graph



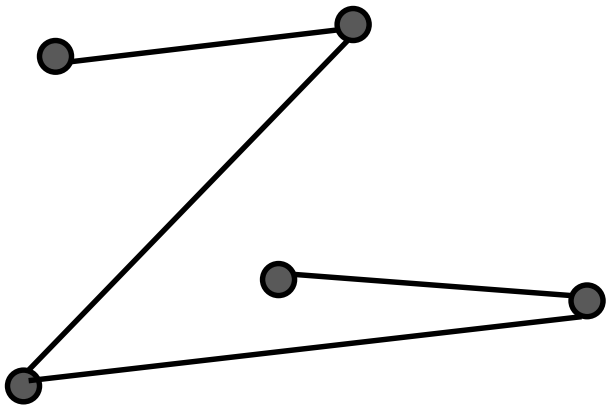
Different Types of Graph

- Non-simple Graph



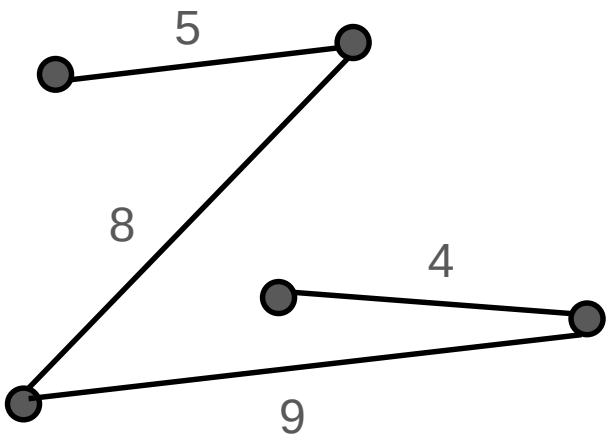
Different Types of Graph

- Simple Graph



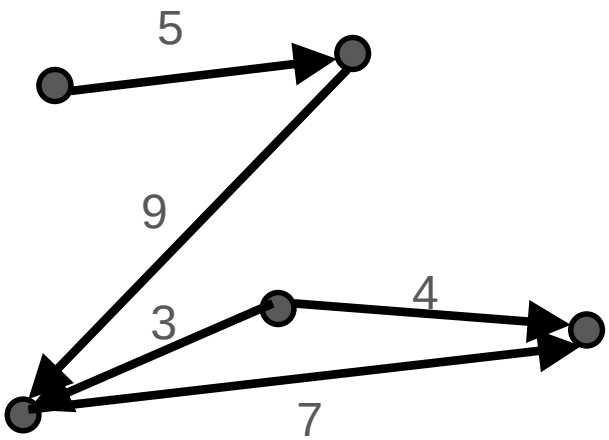
Different Types of Graph

- Undirected Weighted Graph



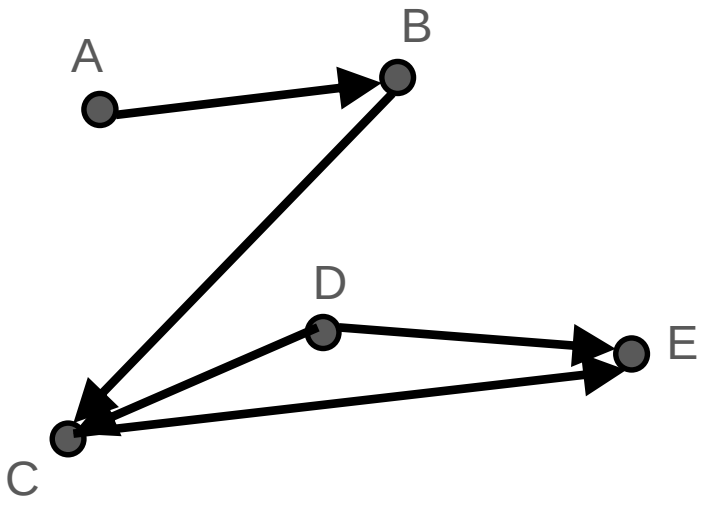
Different Types of Graph

- Directed Weighted Graph



Different Types of Graph

- Labelled Graph



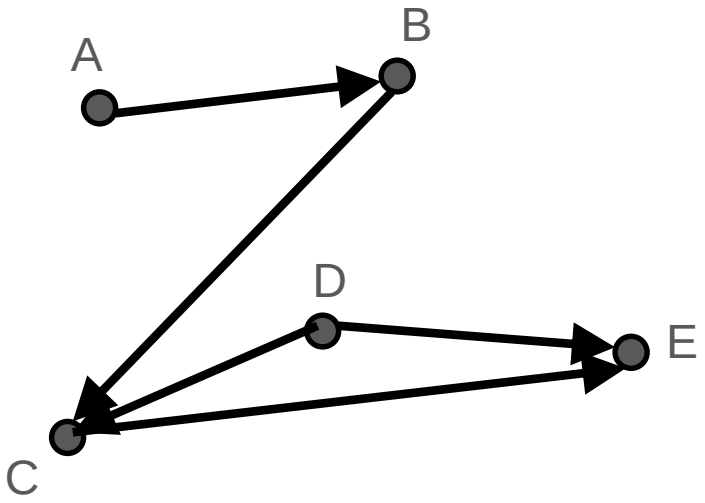
Some Graph Representation

- **Adjacency Matrix**
  - A table where each cell is a possible edge
  - The indices of that cell tell us which vertices it refers to
  - 0 represents no edge
  - 1 represents edge
  - If the graph is weighted the value becomes the weight

Some Graph Representation

- **Adjacency Matrix for Directed Graph**

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	0	0
C	0	0	0	0	1
D	0	0	1	0	1
E	0	0	0	0	0



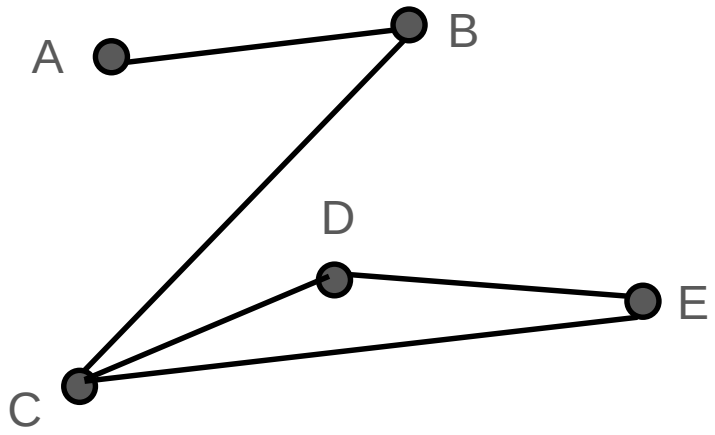
In Python:

```
[
 [0,1,0,0,0]
 [0,0,1,0,0]
 [0,0,0,0,1]
 [0,0,1,0,1]
 [0,0,0,0,0]
]
```

Some Graph Representation

- **Adjacency Matrix for Un-Directed Graph**

	A	B	C	D	E
A	0	1	0	0	0
B	1	0	1	0	0
C	0	1	0	1	1
D	0	0	1	0	1
E	0	0	1	1	0



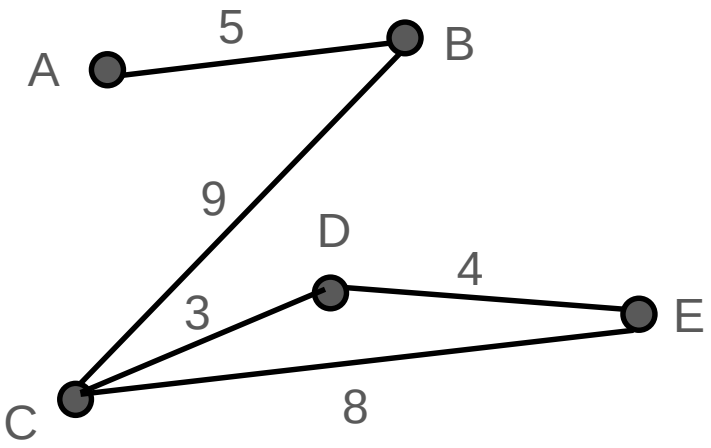
In Python:

```
[
 [0,1,0,0,0]
 [1,0,1,0,0]
 [0,1,0,1,1]
 [0,0,1,0,1]
 [0,0,1,1,0]
]
```

Some Graph Representation

- **Adjacency Matrix for Weighted Graph**

	A	B	C	D	E
A	0	5	0	0	0
B	5	0	9	0	0
C	0	9	0	3	8
D	0	0	3	0	4
E	0	0	8	4	0



In Python:

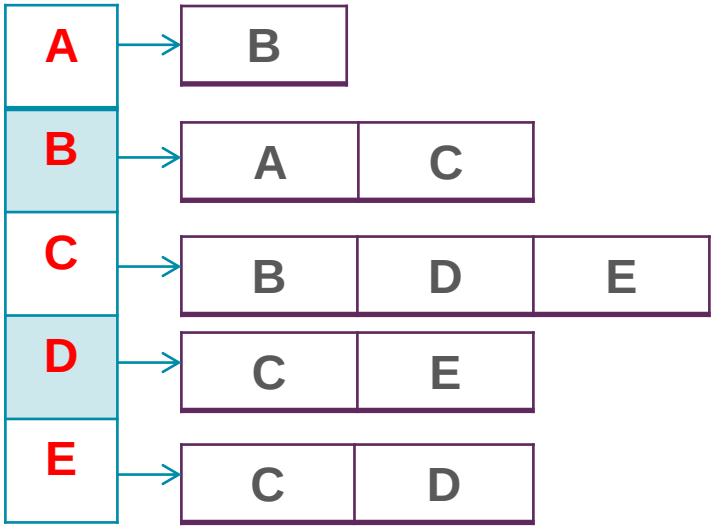
```
[
 [0,5,0,0,0]
 [5,0,9,0,0]
 [0,9,0,3,8]
 [0,0,3,0,4]
 [0,0,8,4,0]
]
```

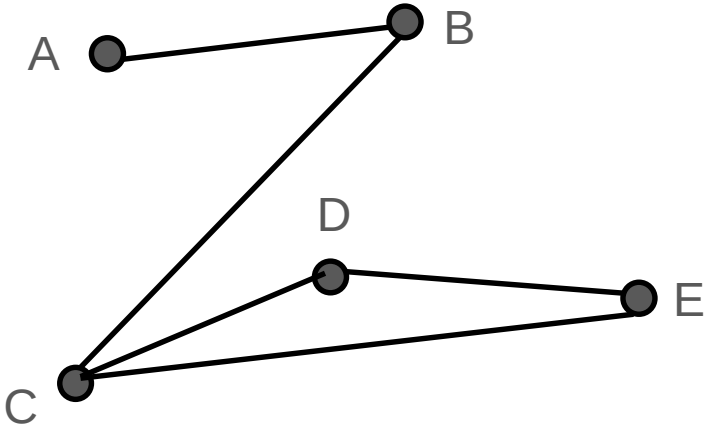
Some Graph Representation

- **Adjacency List:**
  - A table where each inner list holds the vertices that are adjacent to a given vertex.

Some Graph Representation

- **Adjacency List**

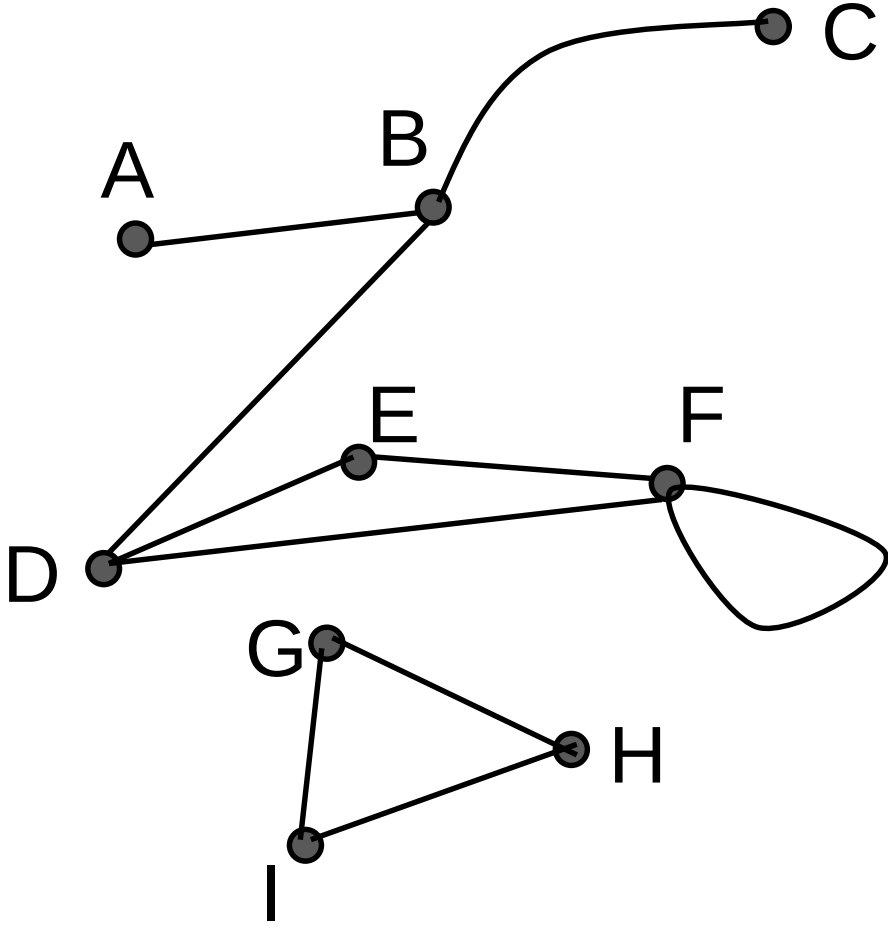




Some Graph Representation

Adjacency List

A -> B
B -> A C D
C -> B
D -> B E F
E -> D F
F -> D E F
G -> H I
H -> G I
I -> G H



Class Activity

- Consider un-directed graph G which include the following edges:

Start vertex	End vertex	Cost
1	3	10
2	3	7
2	4	5
2	5	2
3	6	20

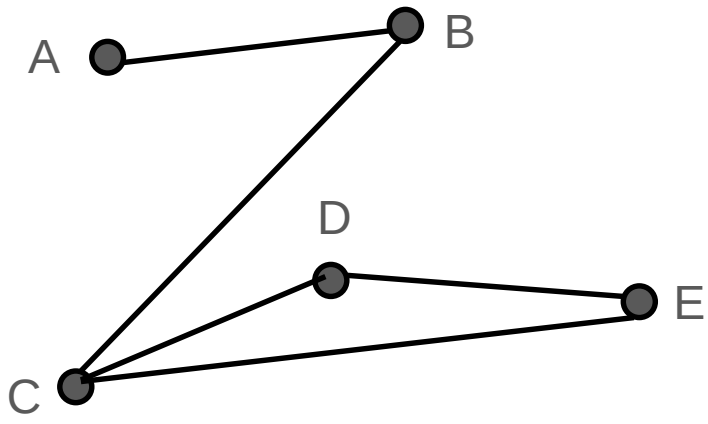
- A) Draw the graph that corresponds to this set of edges including the weights.
- B) Produce an adjacency list that represents this graph; please put weights in brackets

- 1-> 3(10)
- 2-> 3(7), 4(5), 5(2)
- 3->6(20)

Basic Graph Representation in Python

- Given the graph below, we can use Python dictionary data type to present the graph.

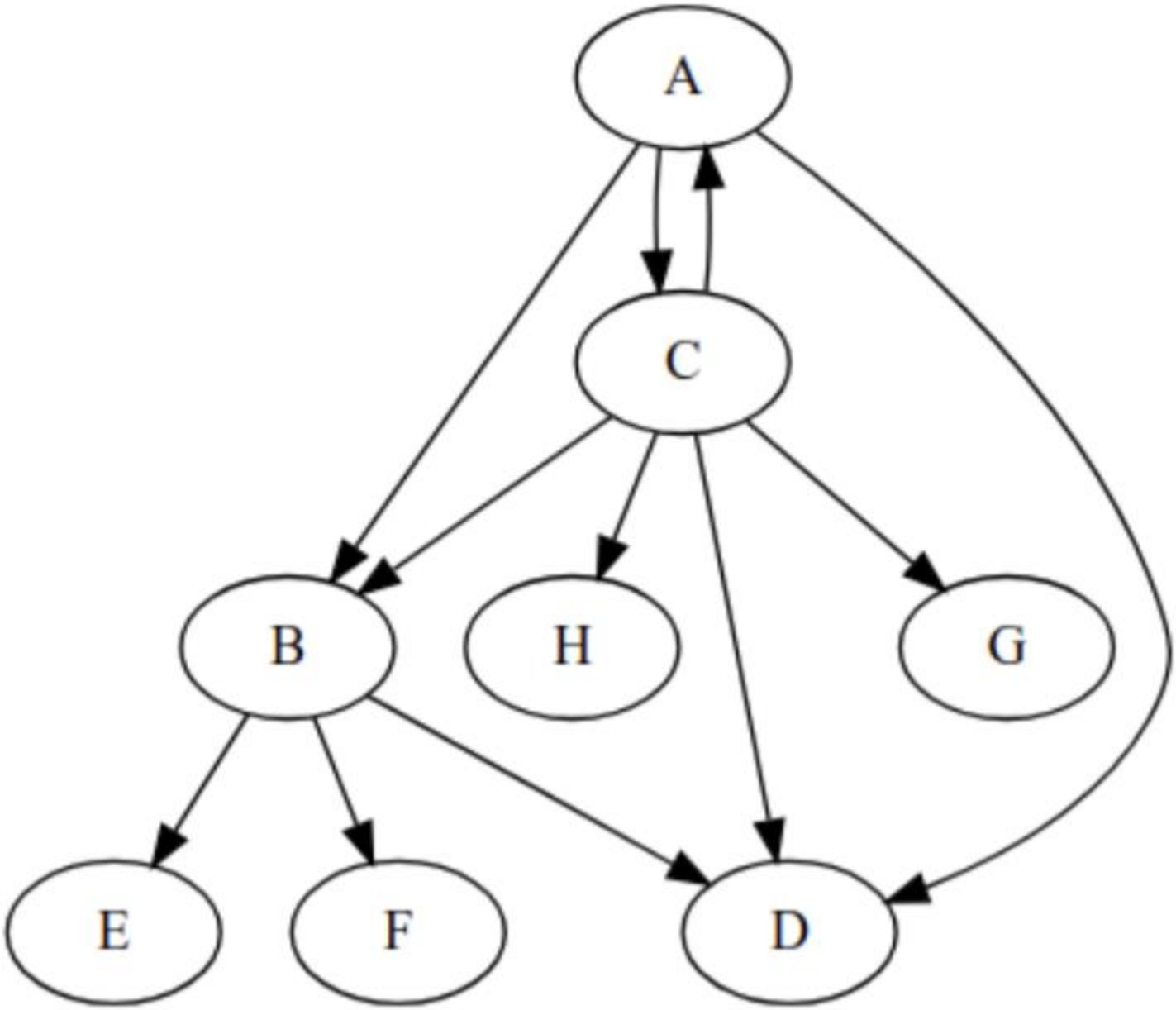
```
graph = { "A" : ["B"],
          "B" : ["A", "C"],
          "C" : ["B","D", "E"],
          "D" : ["C","E"],
          "E" : ["C","D"]
        }
print(graph)
```



Basic Algorithm: Are two nodes are connected?

- Start at one node (the “source” or “start” node)
- Look at all the neighbours
- Look at all the neighbours of the neighbours
- Look at all the neighbours of the neighbours of the neighbours, etc.
- Stop if you reach the other node (the “destination” or “target” node), in which case, the two nodes ARE connected.
- Otherwise, stop if there are no neighbours left that you haven’t already looked at (in which case, there is no way you can get from one node to the other, and the nodes are NOT connected).
- Indeed this is a sort of breadth-first(BFS) search of a graph.
- BFS traverses a graph breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Watch lecture capture to see how BFS work for the graph



Breadth-first Search?

Breadth-first search, pseudocode

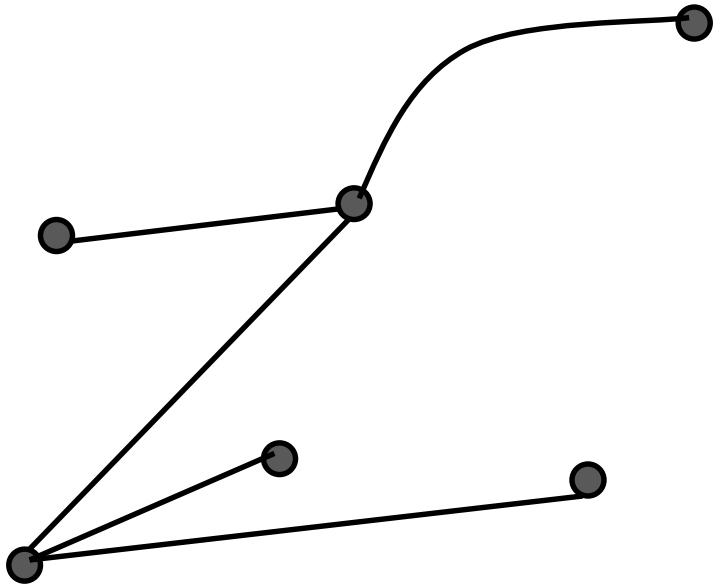
```
G = ...   # the graph
n = ...   # the start node
visited = <the empty set>  # the set of visited nodes
todo = [n]  # the list of nodes we need to visit
while <todo is not empty>:
    m = todo[0]  # m is first item in todo
    todo = todo[1:]  # remove first item from todo
    print(m)
    <add m to visited>
    for x in neighbours(m):
        if x not in visited:
            if x not in todo:
                todo.append(x)
```

How to check two nodes are connected or no?

- Start at the source node.
- Perform a breadth-first search.
- If you reach the target, the two nodes are connected.
- Otherwise, the two nodes aren’t connected.
- **The lab exercise:** write this algorithm in Python. [Hint: use the breadth-first search code, but tweak it slightly. . . You need to return either True, or False.]

Quick Intro about Tree

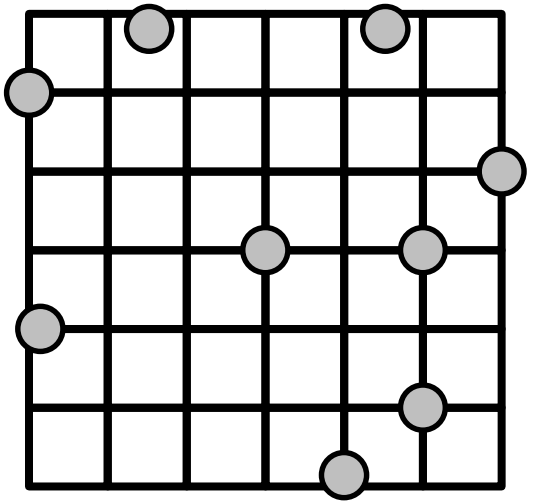
- Trees are graph which are
  - Simple
  - Connected
  - And have no cycle



- We will know more about Tree next week.

Class Discussion : Electrical Problem

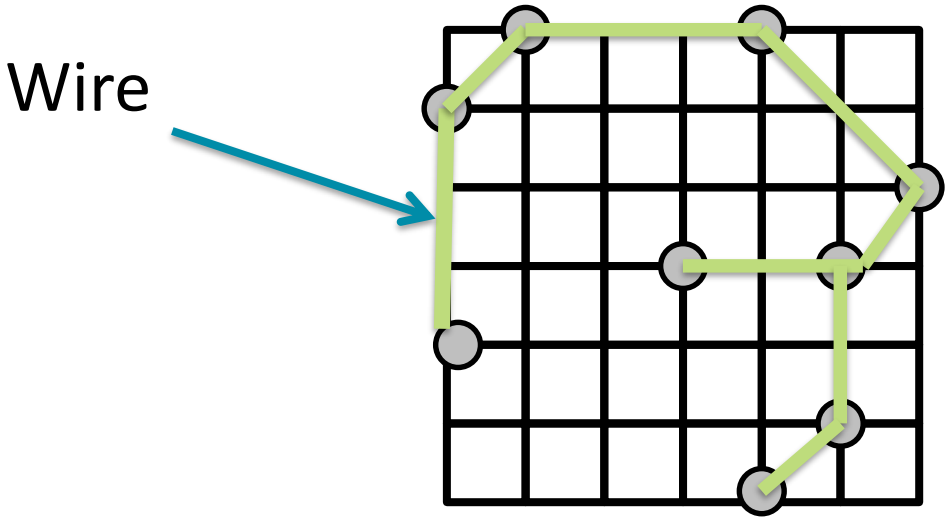
- Suppose an electrical engineer, has terminals that need to be connected with wires.
- Describe an algorithm so that the **least amount of wire is used**.



The Problem Can be Represented as:

- Consider the graph consisting of the following:
  - Vertices corresponding to the terminals
  - Edges joining each pair of vertices
  - Weight on each edge corresponding to the length of the edge.
- Find a **spanning tree** of this graph with minimum total weight.
- A spanning tree of graph G is a tree which:
  - ✓ Contains all the vertices of G, and
  - ✓ Contains a subset of the edges in G (or all if it's already a tree).

Possible Solution

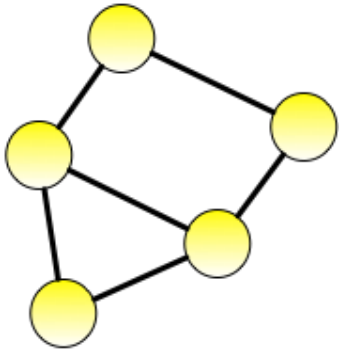


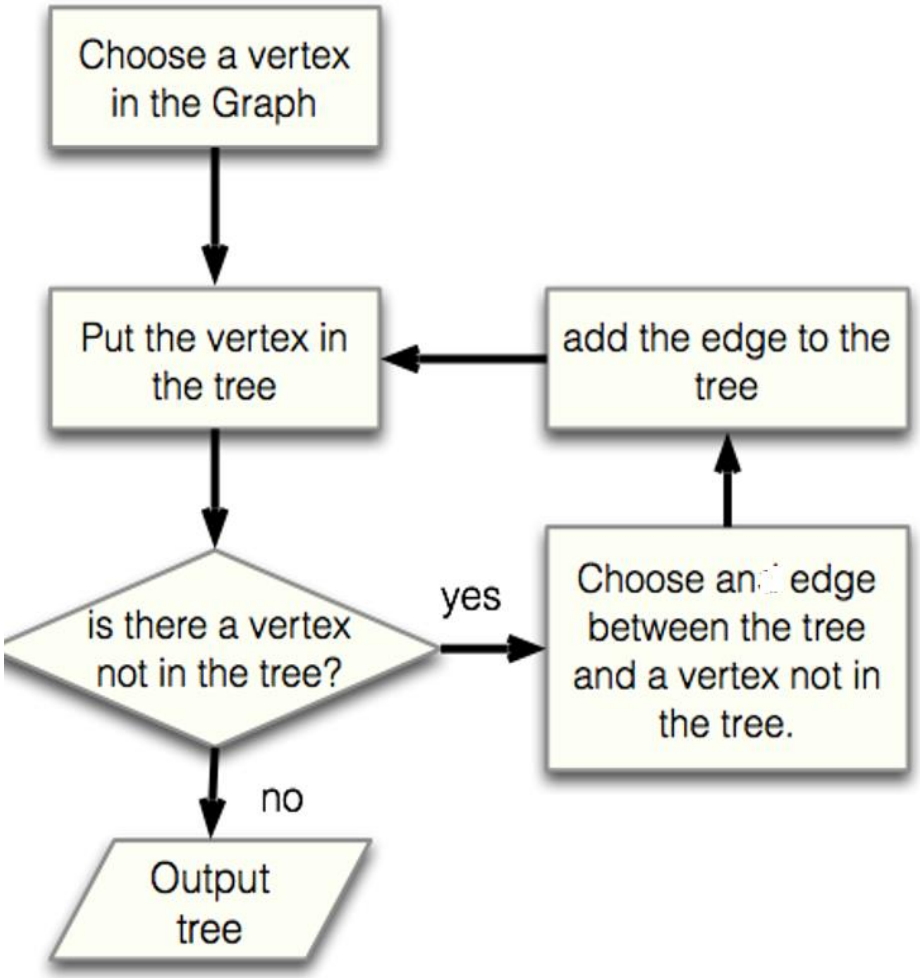
Prim's algorithm to find Spanning Tree

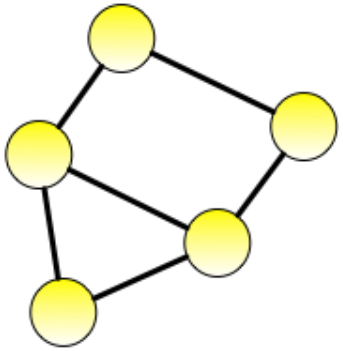
Finds a spanning tree of a graph G

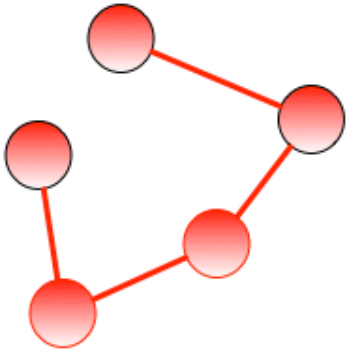
Input: A graph G, simple and connected

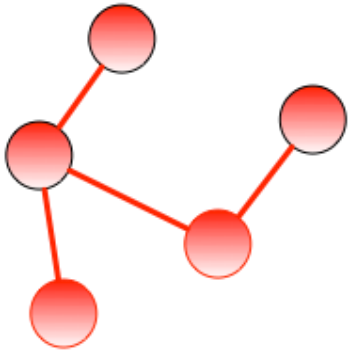
Output: A spanning tree of G

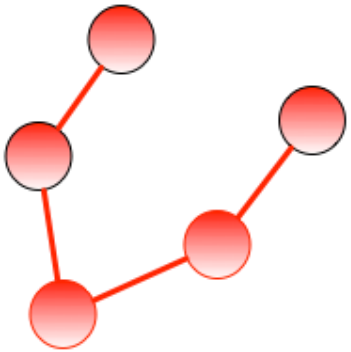


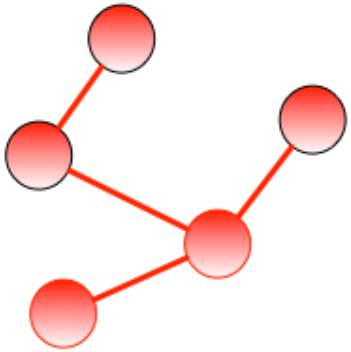


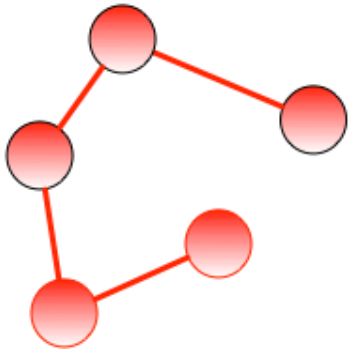


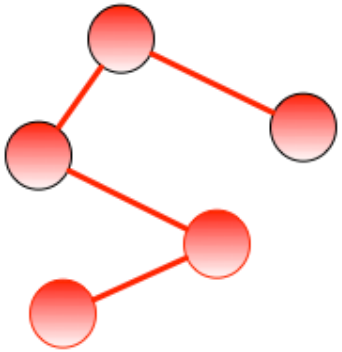










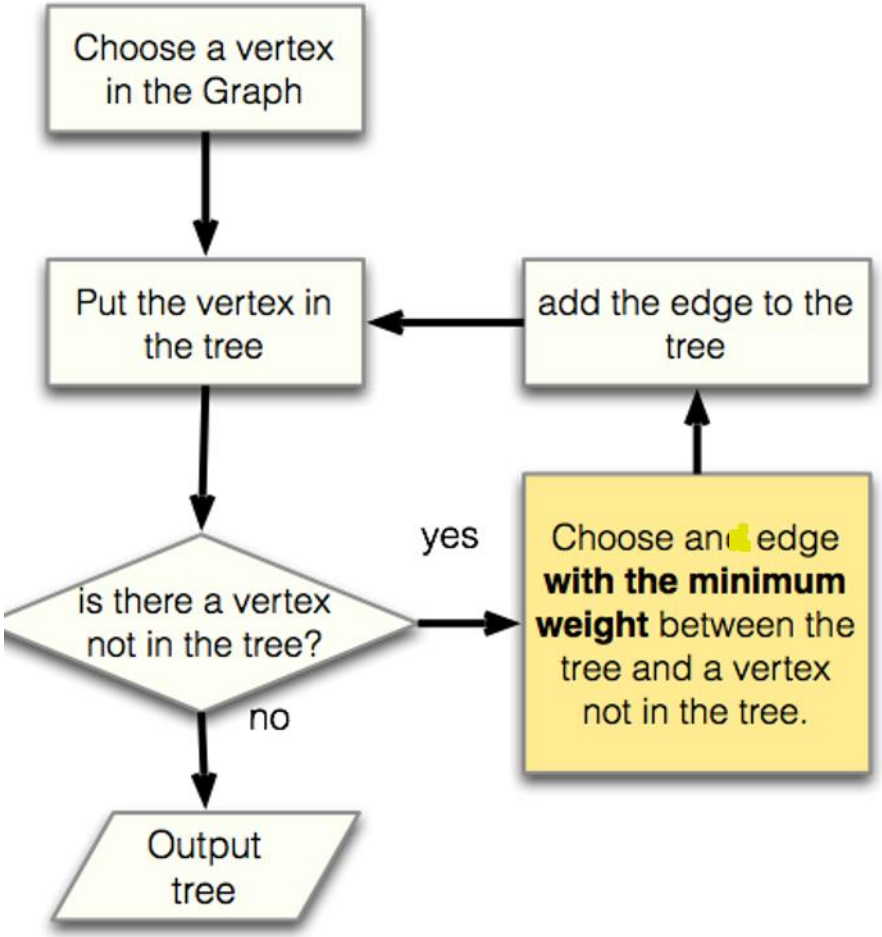


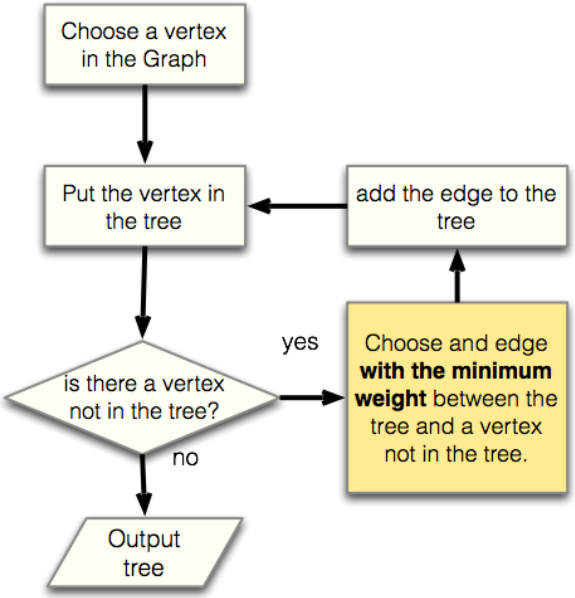
Prim's algorithm to find a Minimum Spanning Tree

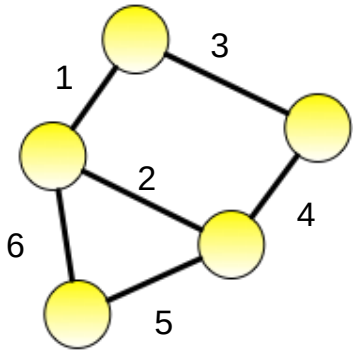
Finds a spanning tree of a graph G

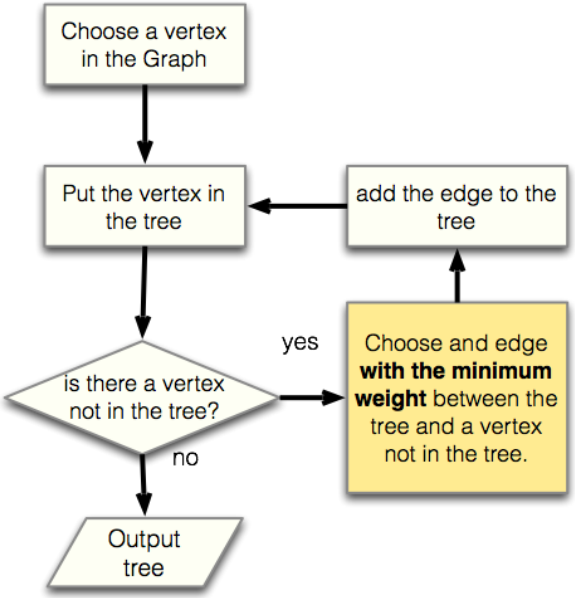
Input: A graph G, simple and connected

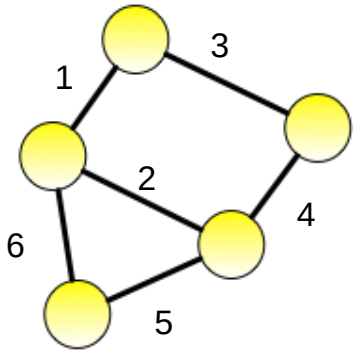
Output: A **minimum** spanning tree of G

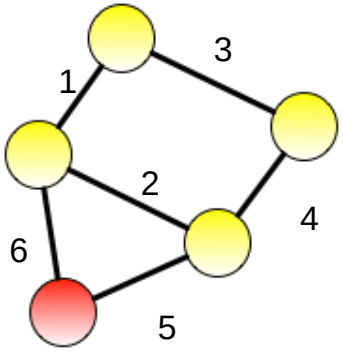


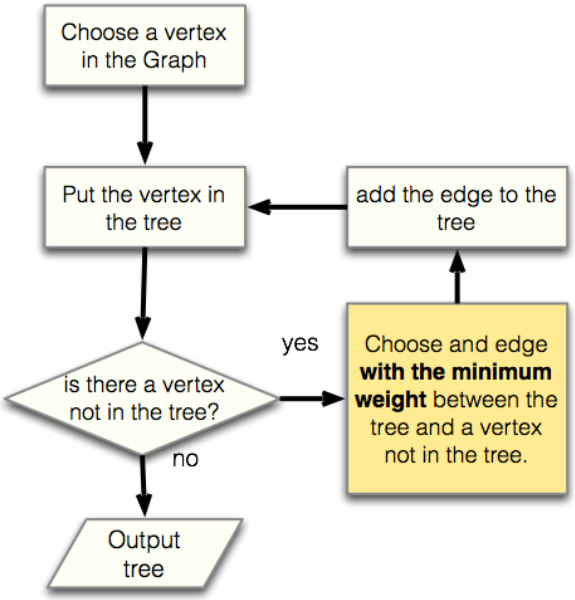


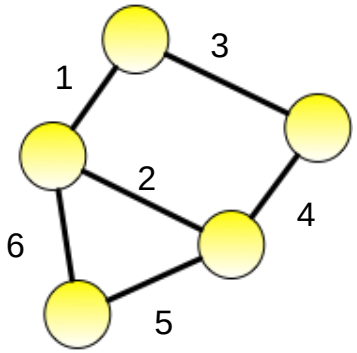


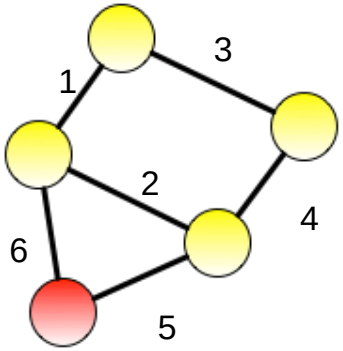


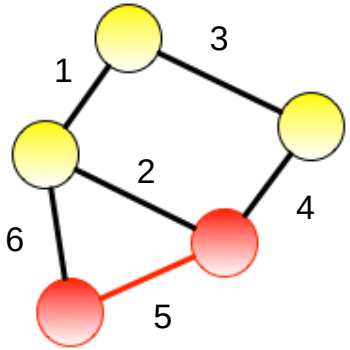


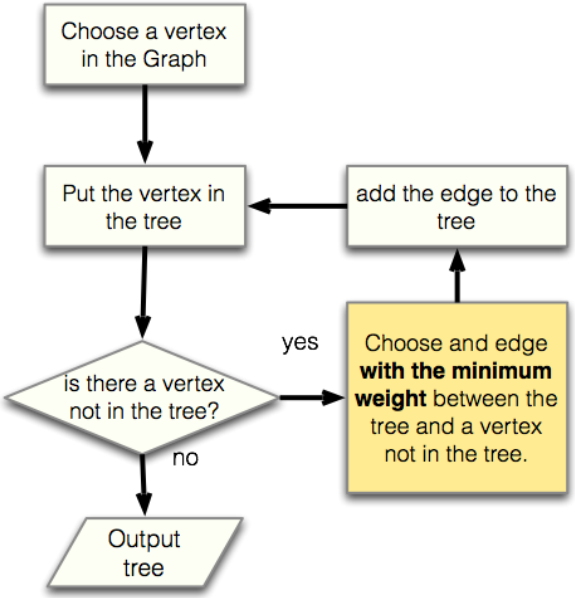


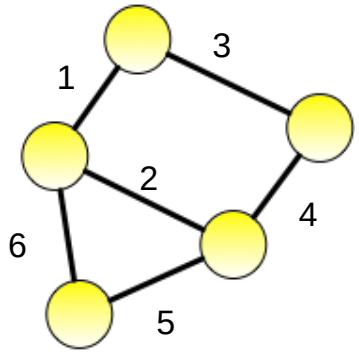


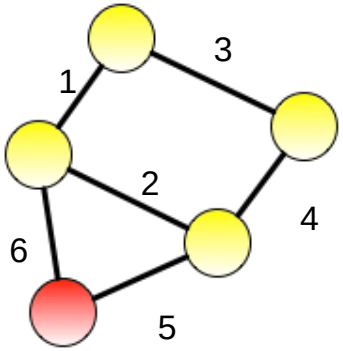


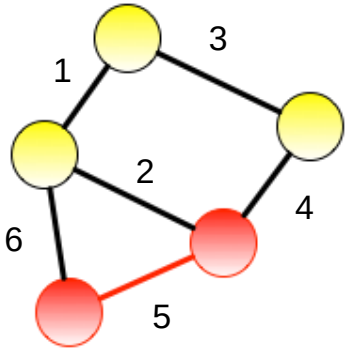


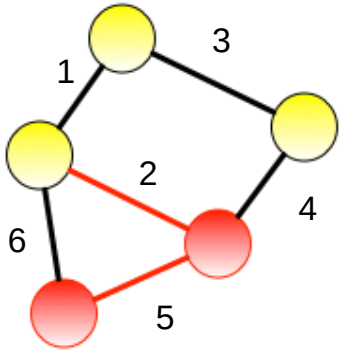


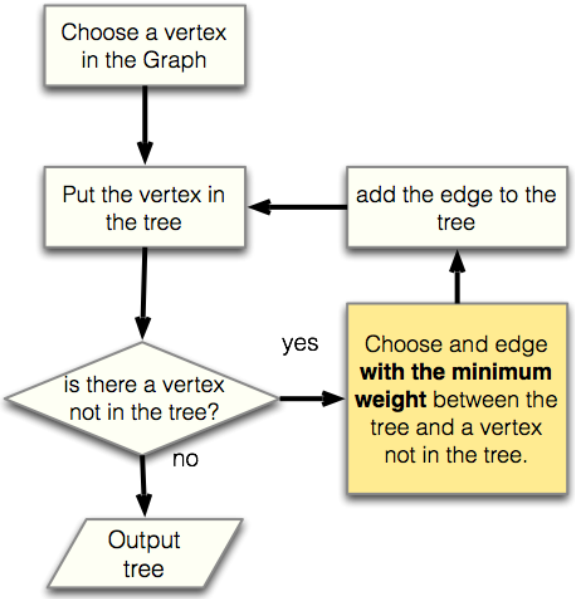


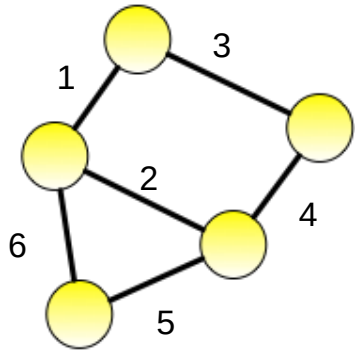


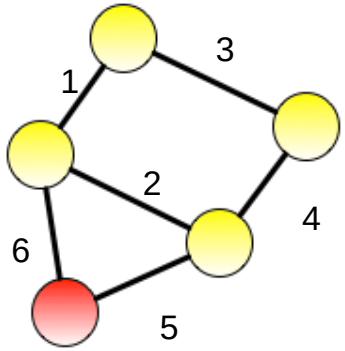


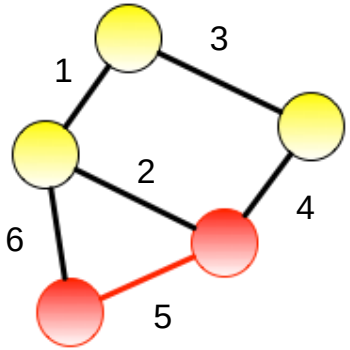


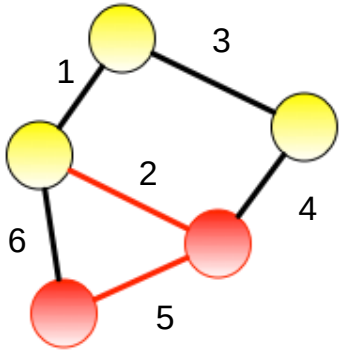


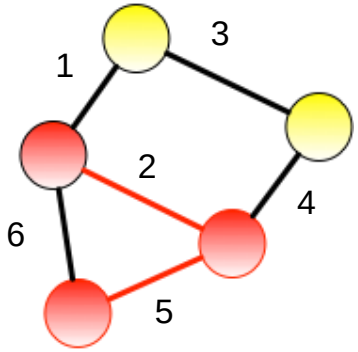


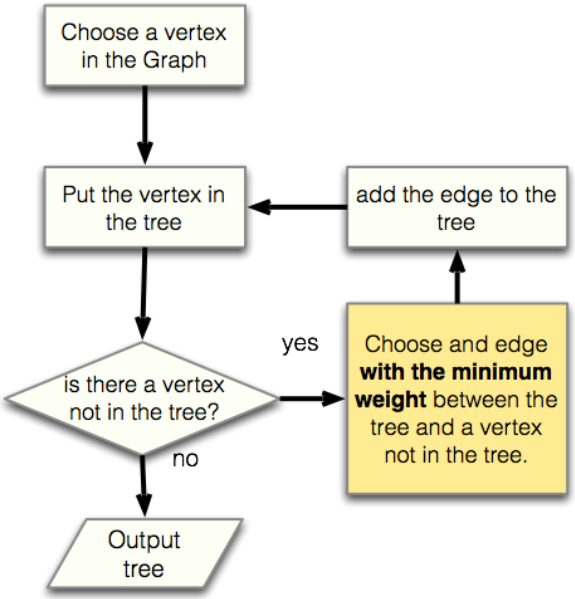


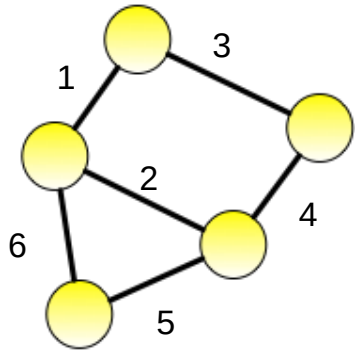


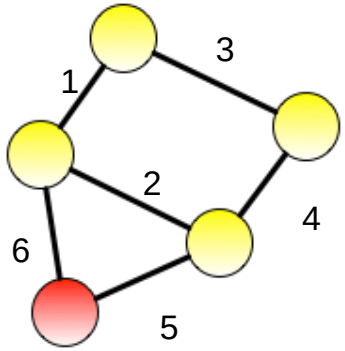


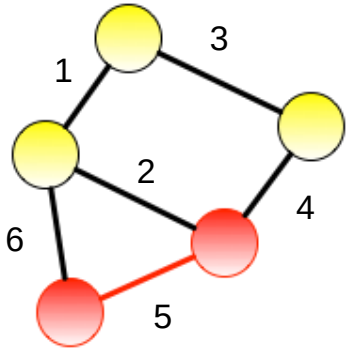


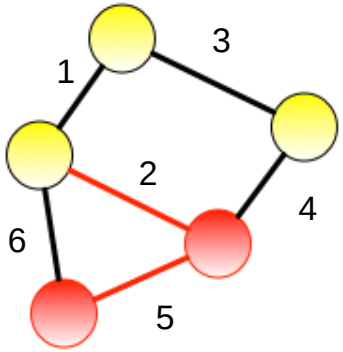


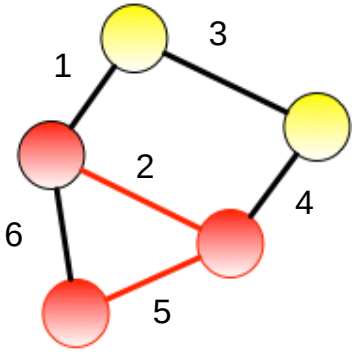


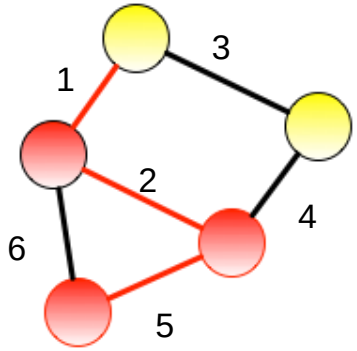


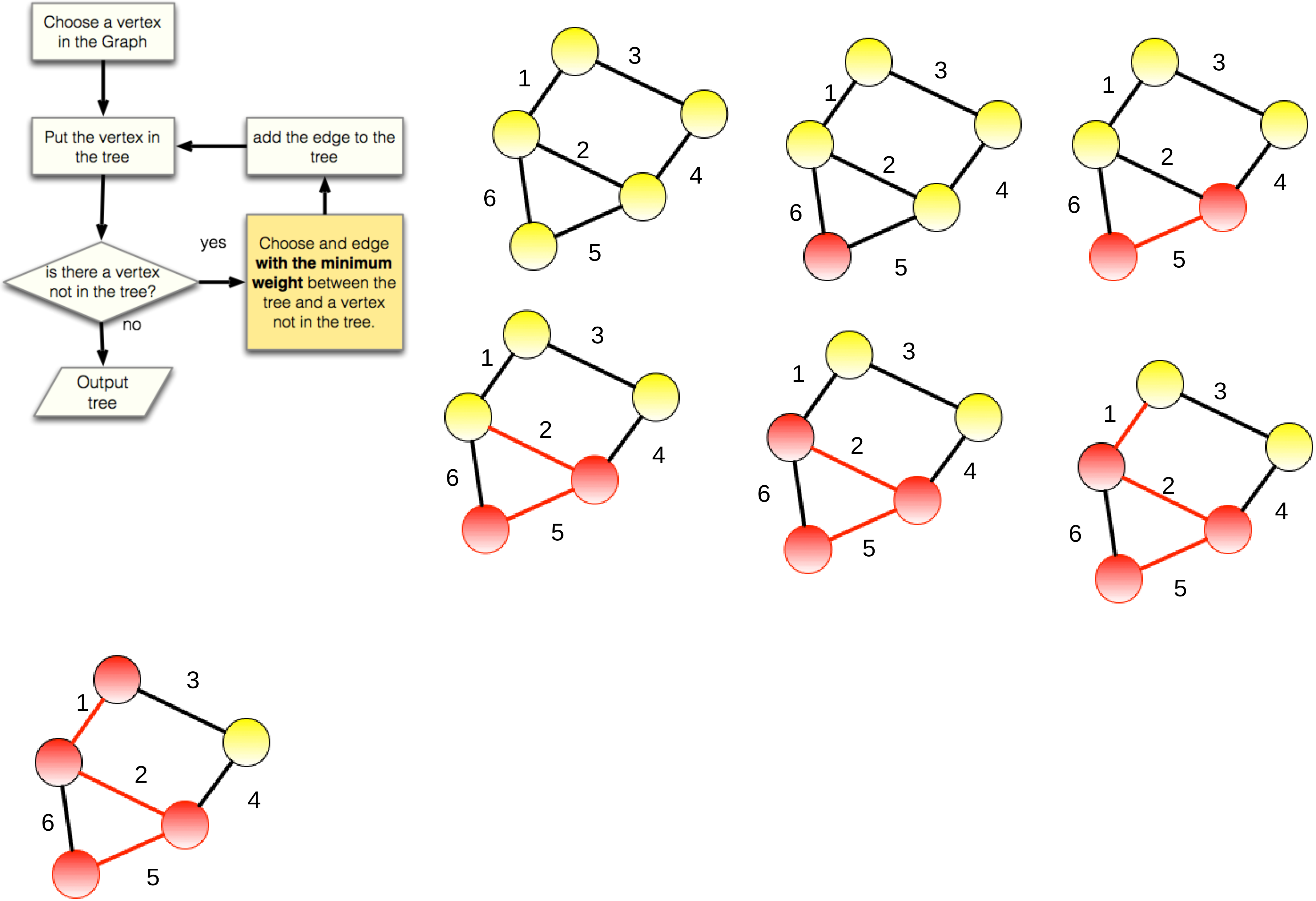


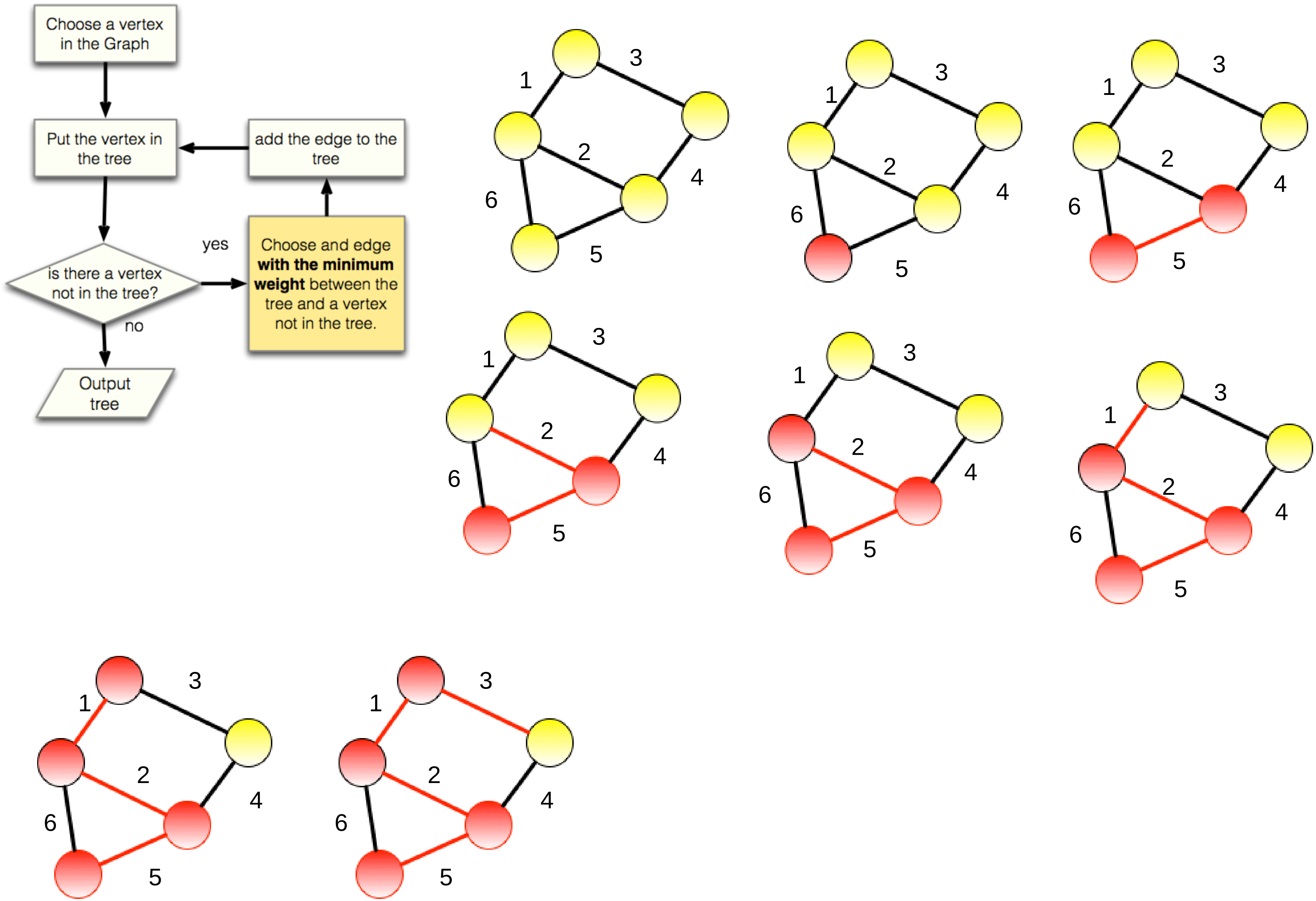


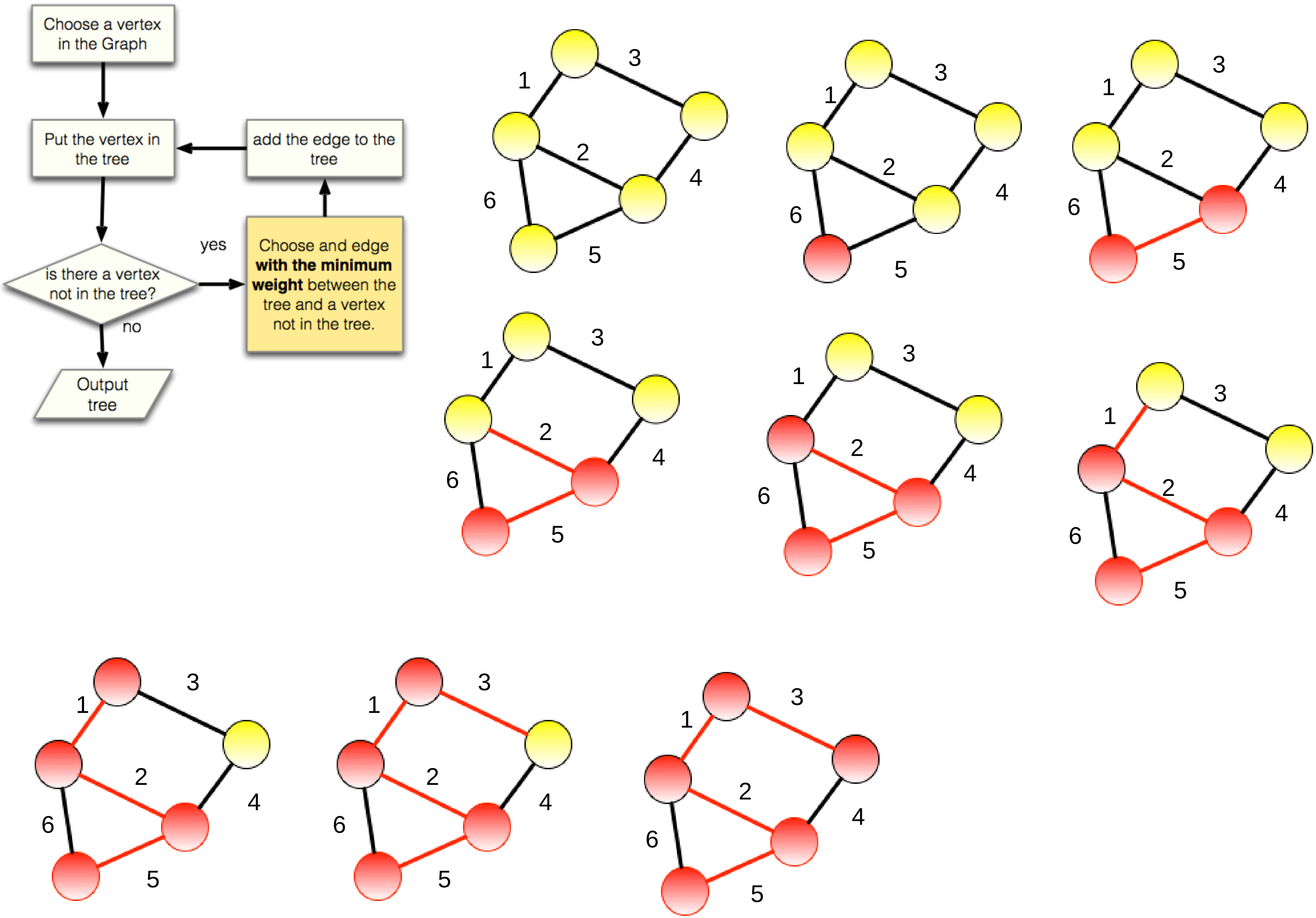












UNIVERSITY OF LEICESTER