FIT1045: Algorithms and Programming Fundamentals in Python

Lecture 9

Graphs and Spanning Trees

https://insidesmallbusiness.com.au/growth-development/five-tips-to-enhance-professional-connections

Objectives

- Learn about graphs, trees, and spanning trees
- Prim's algorithm (simplified version) for finding spanning trees
- Simplify problems and algorithm by decomposition

Covered learning outcomes:

1 – Translate between problem descriptions and program designs with appropriate input/output representations

2 – Choose and implement appropriate problem solving strategies

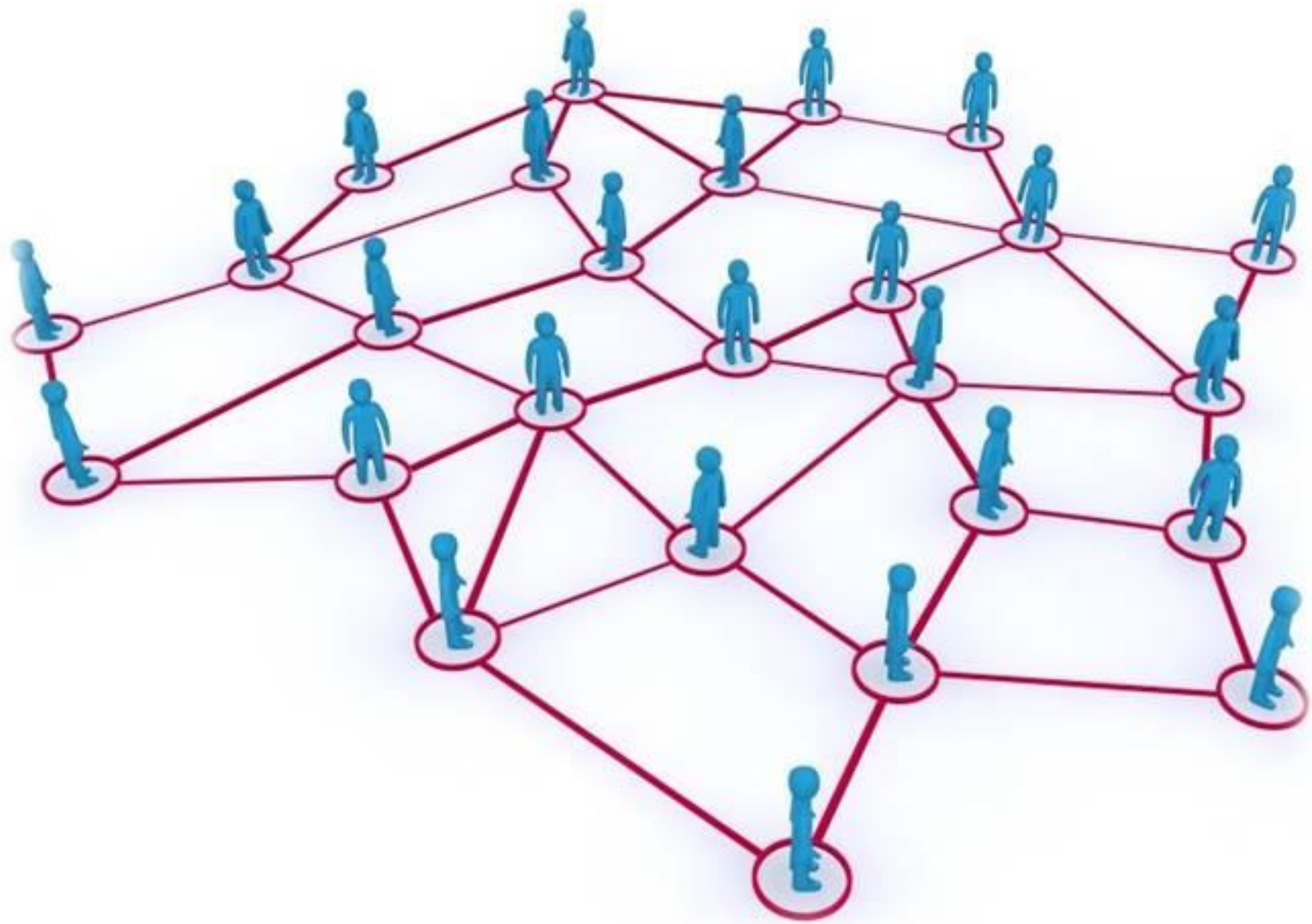
4 – Decompose problems into simpler problems and reduce unknown to known problems

Concrete goal: build mazes

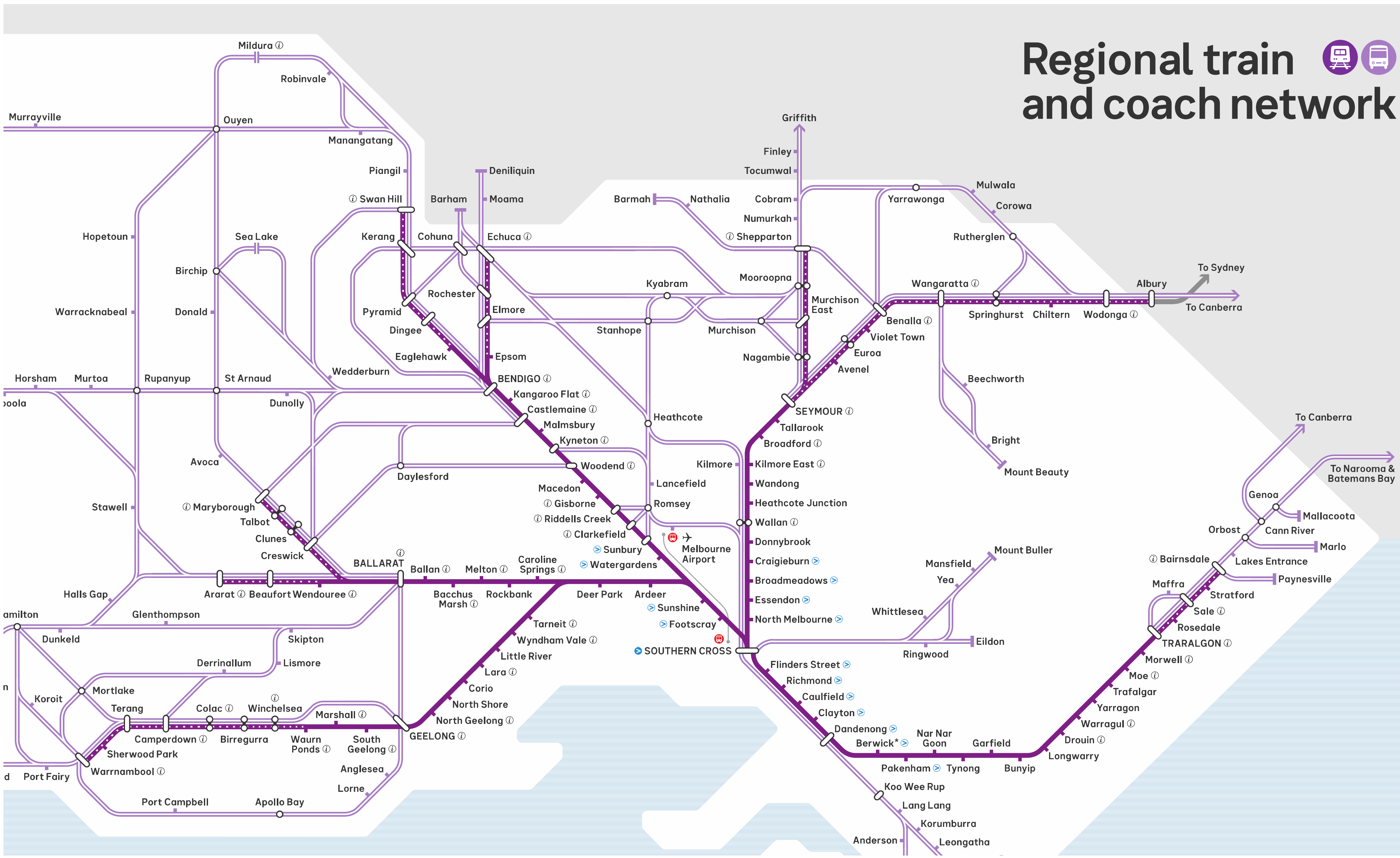
Where am I?

1. Graphs
2. Trees and Spanning Trees
3. Prims algorithm (simplified)
4. *Problem decomposition (if time left)*

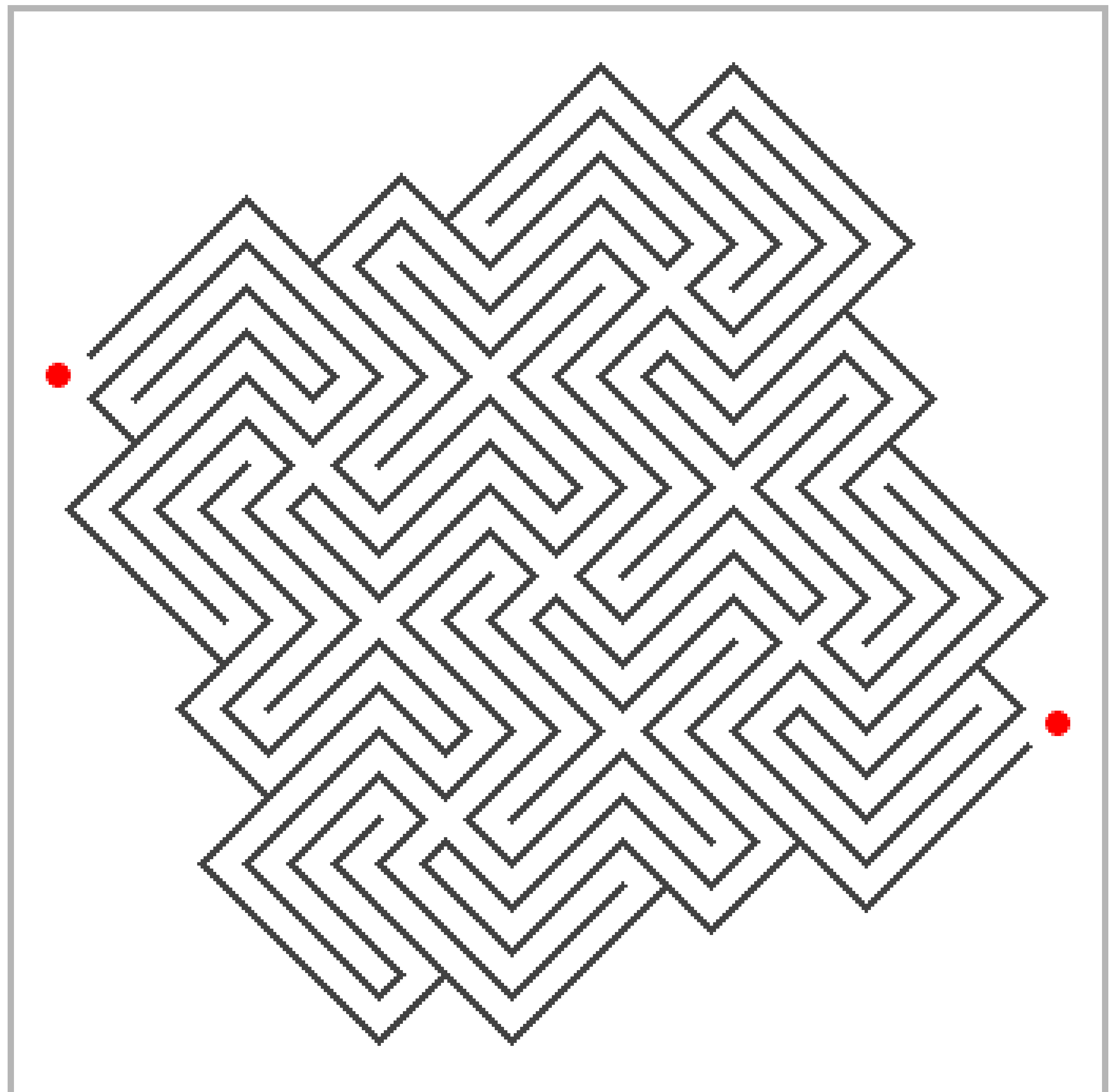
How to represent relational data?



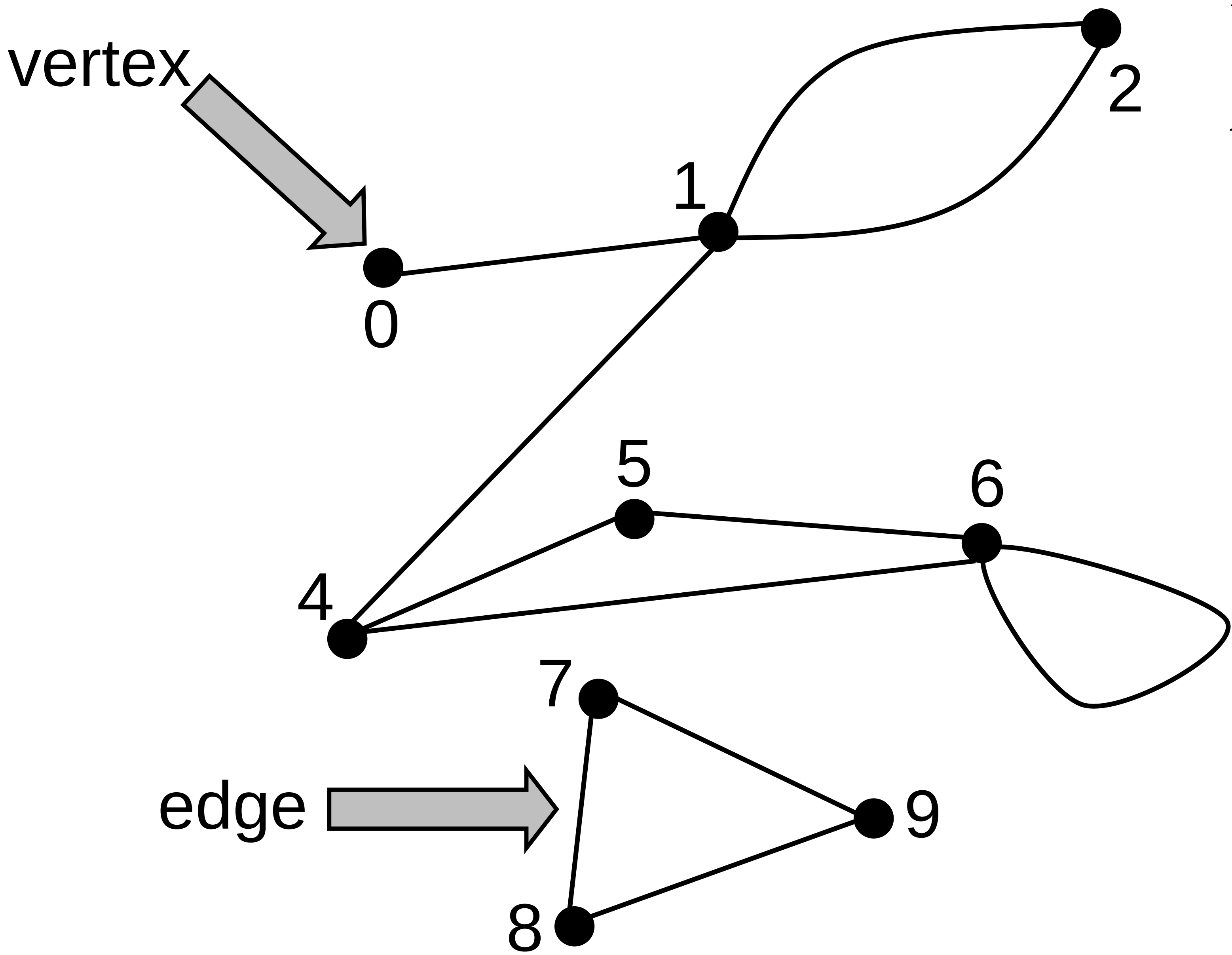
How to represent relational data?



How to represent relational data?



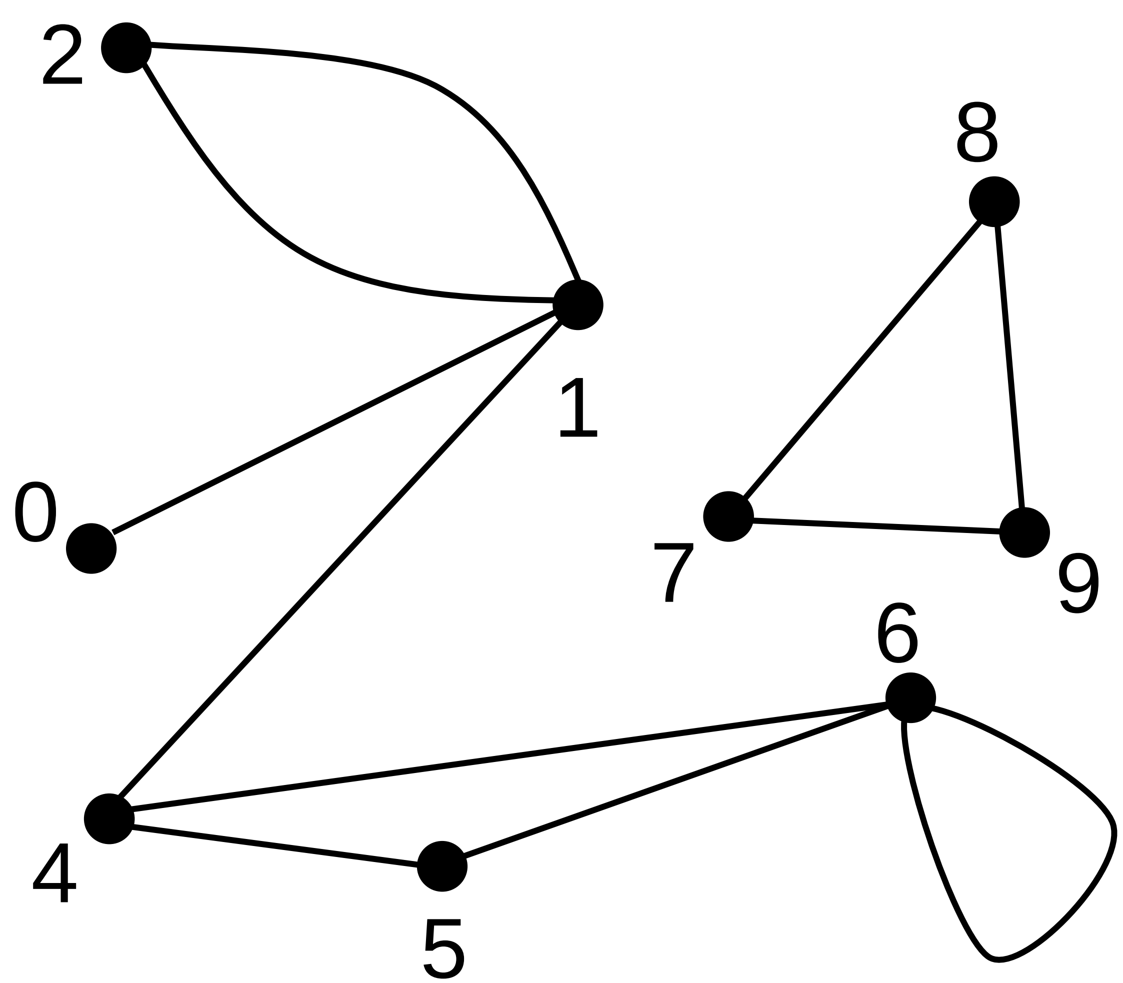
Abstraction of such data: *graphs*



$V = \{0, \dots, 9\}$
$E = \{(0,1), (1,2), (1,2),$
$(1,4), (4,5), (4,6),$
$(5,6), (6,6), (7,8),$
$(7,9), (8,9)\}$

Definition [Levitin, p28]:
Graph $G = (V, E)$ is a set of *vertices* $V = \{v_1, \dots, v_n\}$ *and* collection of *edges* $E = \{e_1, \dots, e_m\}$ where $e_i = (v, w), v, w \in V$

Many ways to draw the same graph!

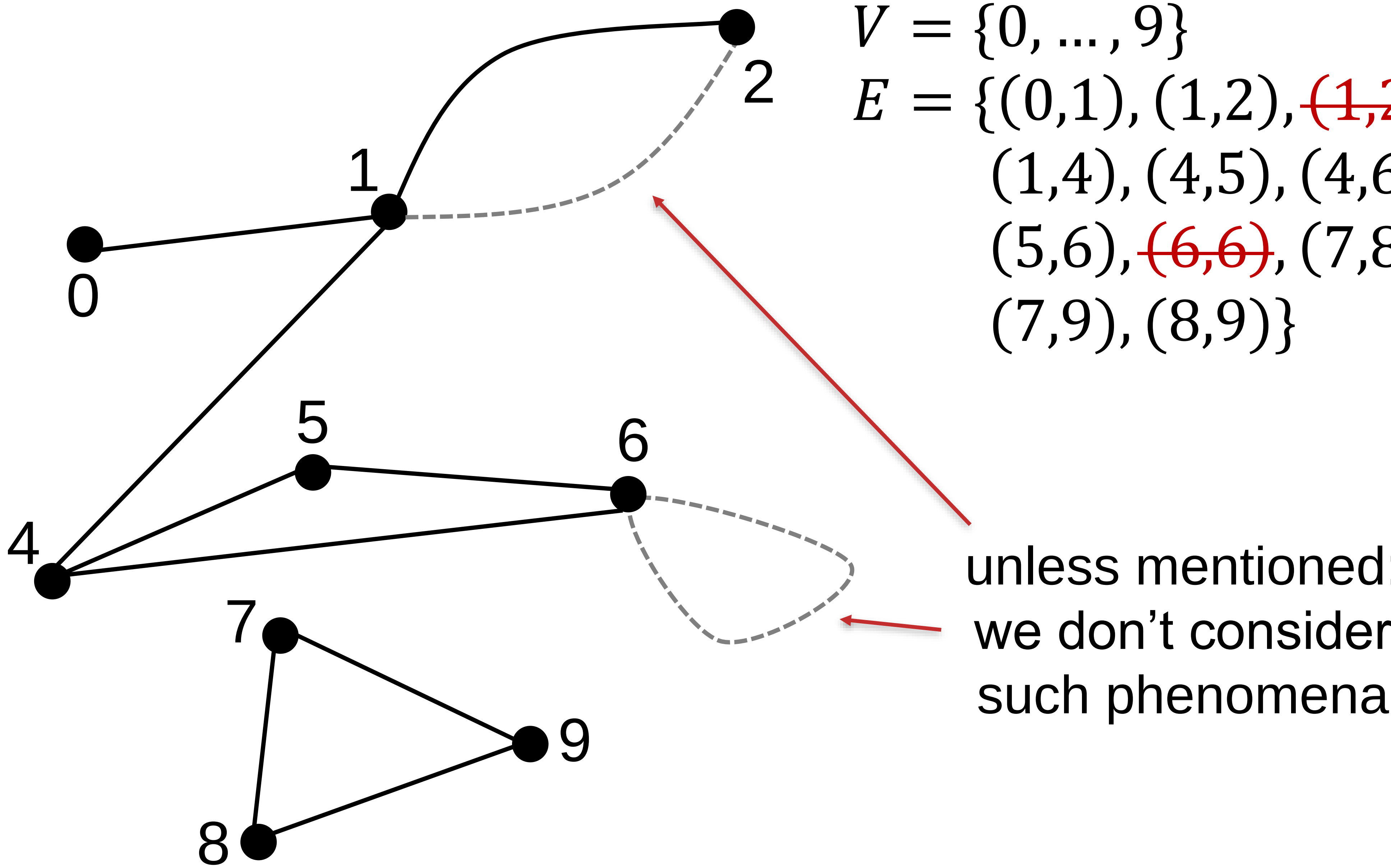


$V = \{0, \dots, 9\}$
$E = \{(0,1), (1,2), (1,2), (1,4), (4,5), (4,6), (5,6), (6,6), (7,8), (7,9), (8,9)\}$

same graph as before

Definition [Levitin, p28]:
Graph $G = (V, E)$ is a set of *vertices* $V = \{v_1, \dots, v_n\}$ *and* collection of *edges* $E = \{e_1, \dots, e_m\}$ where $e_i = (v, w), v, w \in V$

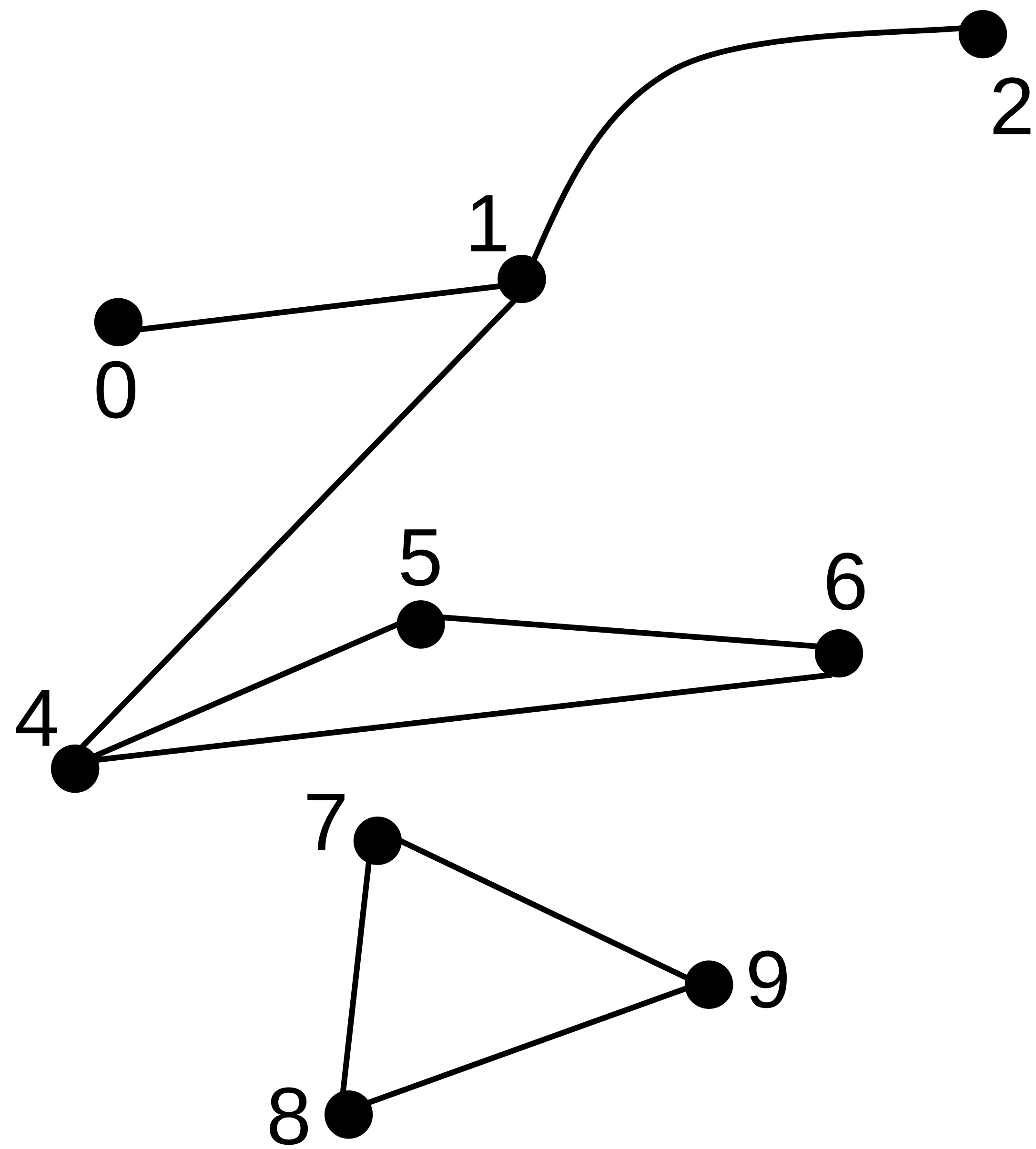
We focus on "simple" graphs



$V = \{0, \dots, 9\}$
$E = \{(0,1), (1,2), \cancel{(1,2)}, (1,4), (4,5), (4,6), (5,6), \cancel{(6,6)}, (7,8), (7,9), (8,9)\}$

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

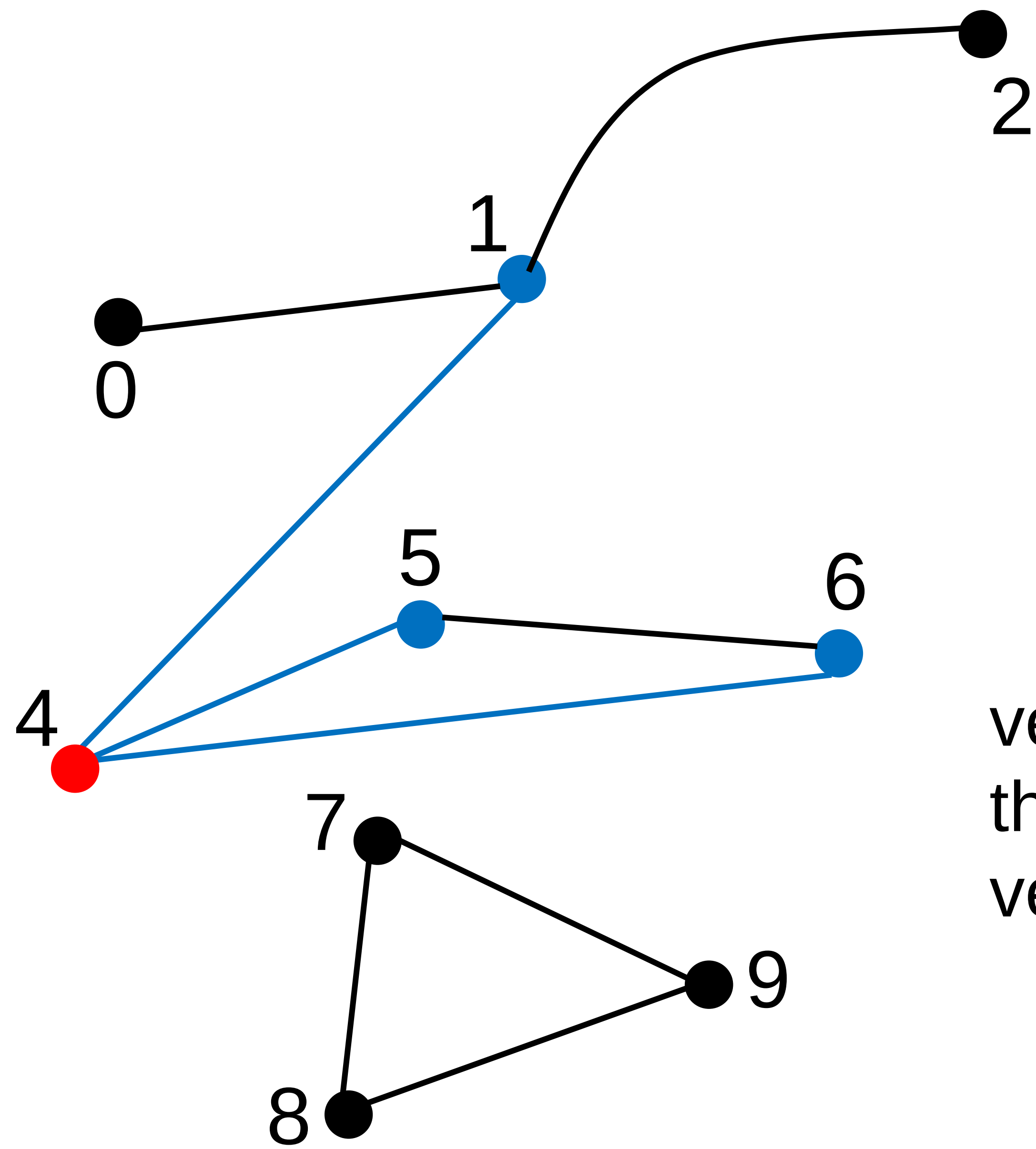
Adjacency and neighbours



$V = \{0, \dots, 9\}$
$E = \{(0,1), (1,2),$
$(1,4), (4,5), (4,6),$
$(5,6), (7,8),$
$(7,9), (8,9)\}$

Definition:
A vertex j is called adjacent to a vertex i (or a neighbour of i) if there is an edge $(i, j) \in E$.

Adjacency and neighbours

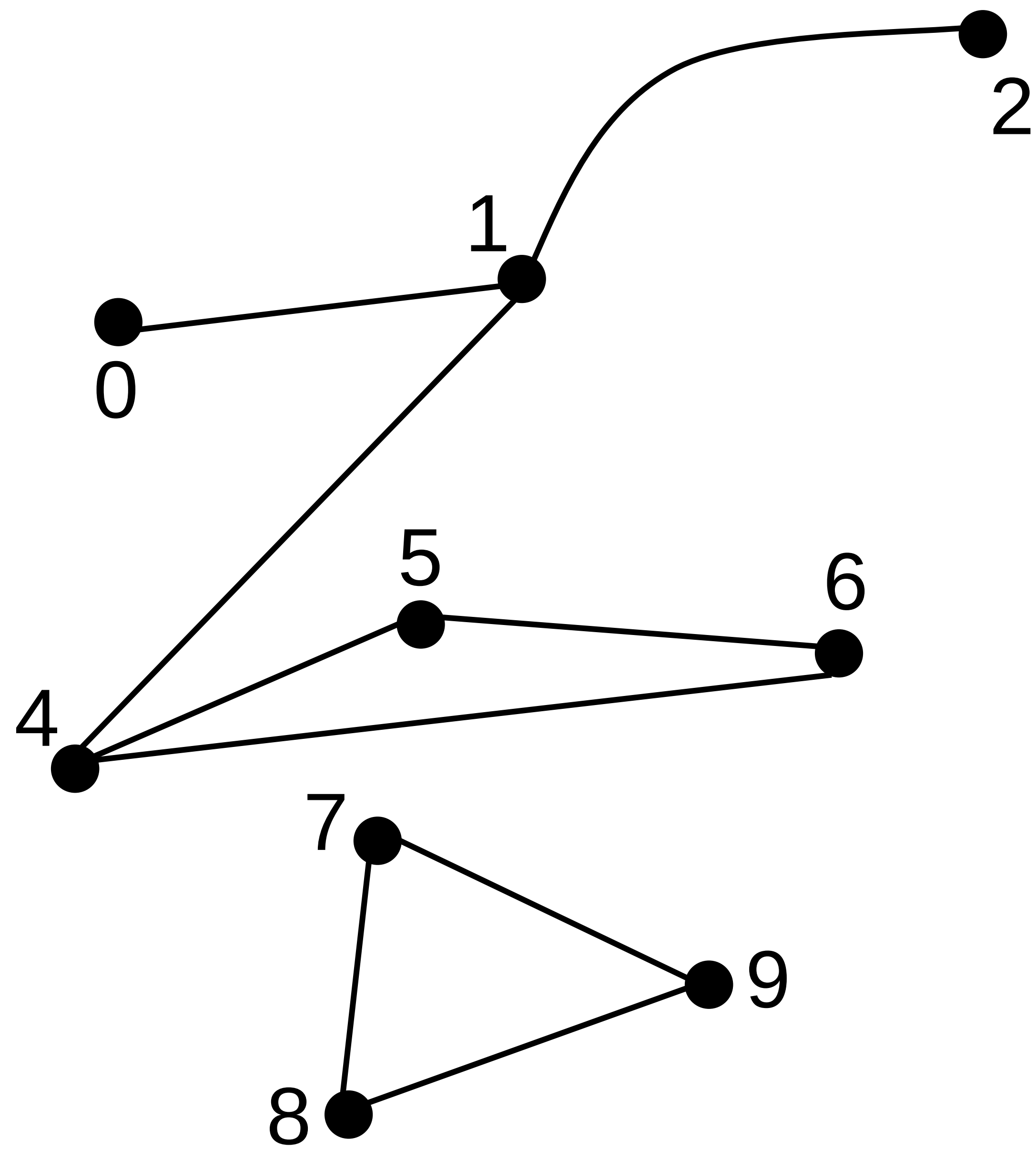


$V = \{0, \dots, 9\}$
$E = \{(0,1), (1,2),$
$(1,4), (4,5), (4,6),$
$(5,6), (7,8),$
$(7,9), (8,9)\}$

vertices $\{1,5,6\}$ are the neighbours of vertex 4

Definition:
A vertex j is called adjacent to a vertex i (or a neighbour of i) if there is an edge $(i, j) \in E$.

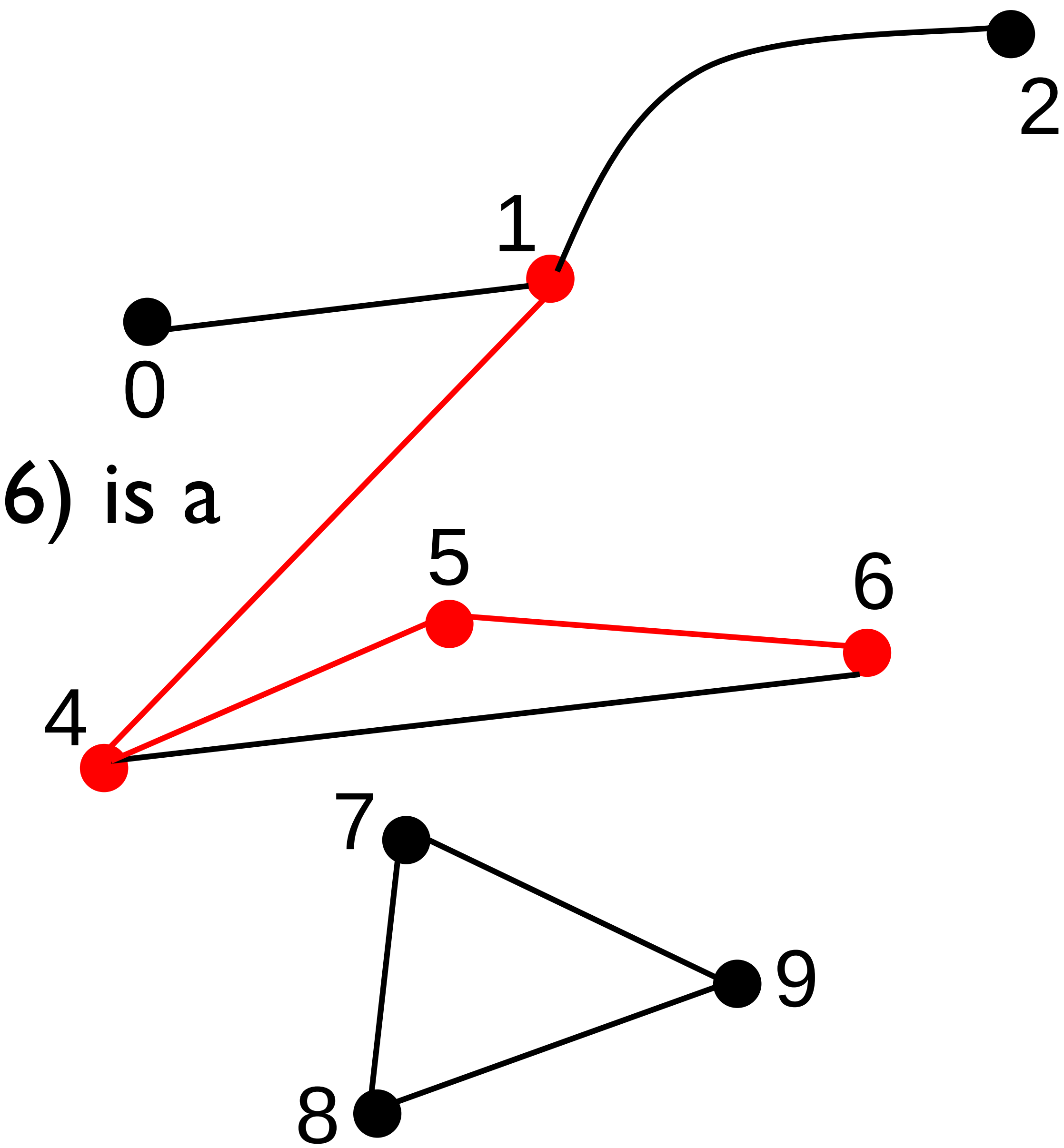
Paths, cycles, and connectivity



$V = \{0, \ldots, 9\}$
$E = \{(0,1), (1,2),$
$(1,4), (4,5), (4,6),$
$(5,6), (7,8),$
$(7,9), (8,9)\}$

Definition:
A ***path*** is a non-self-intersecting sequence of vertices such that there is an edge between consecutive vertices in the sequence.

Paths, cycles, and connectivity



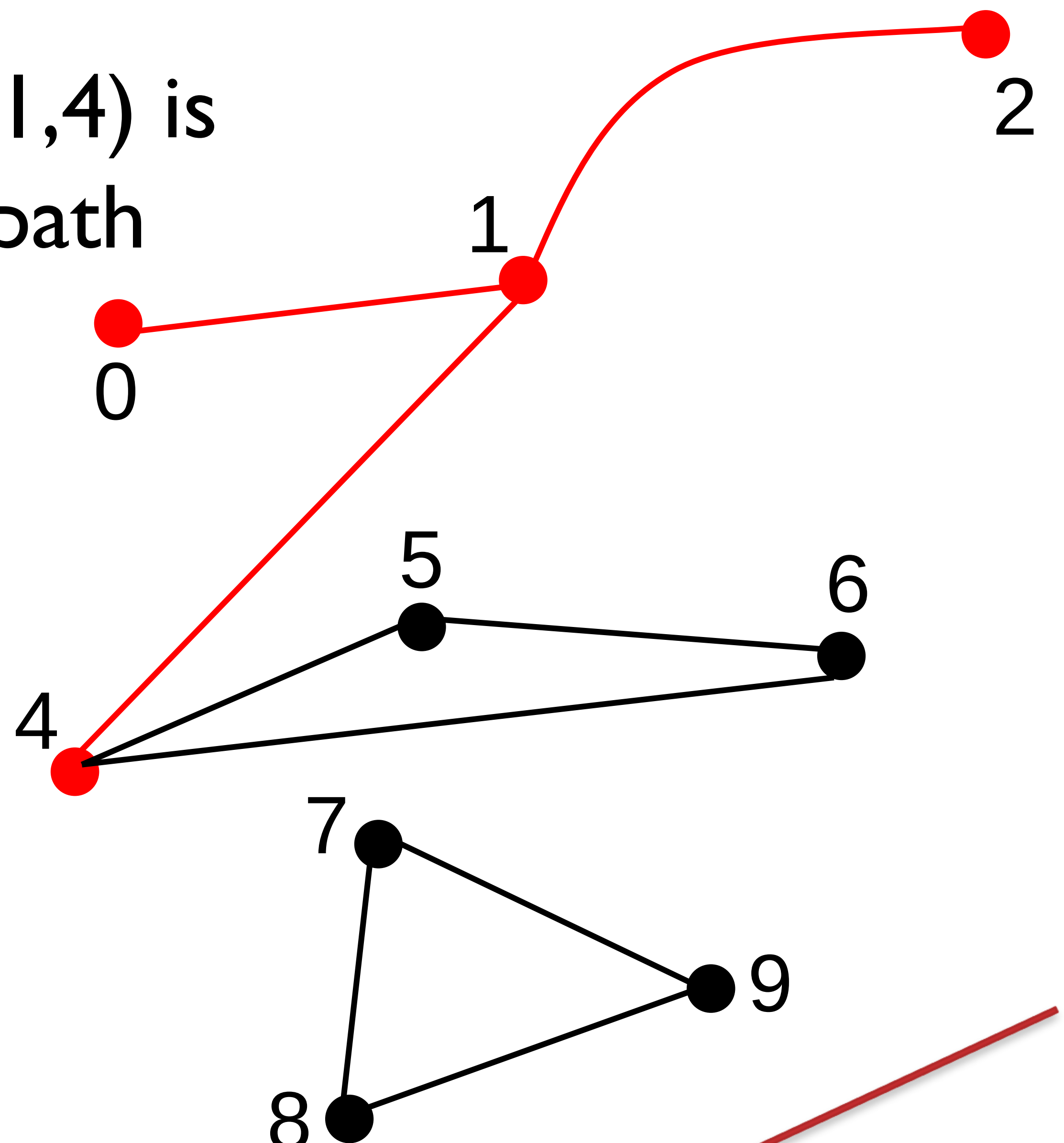
$V = \{0, \ldots, 9\}$
$E = \{(0,1), (1,2),$
$(1,4), (4,5), (4,6),$
$(5,6), (7,8),$
$(7,9), (8,9)\}$

(1, 4, 5, 6) is a path

Definition:
A ***path*** is a non-self-intersecting sequence of vertices such that there is an edge between consecutive vertices in the sequence.

Paths, cycles, and connectivity

(0,1,2,1,4) is not a path

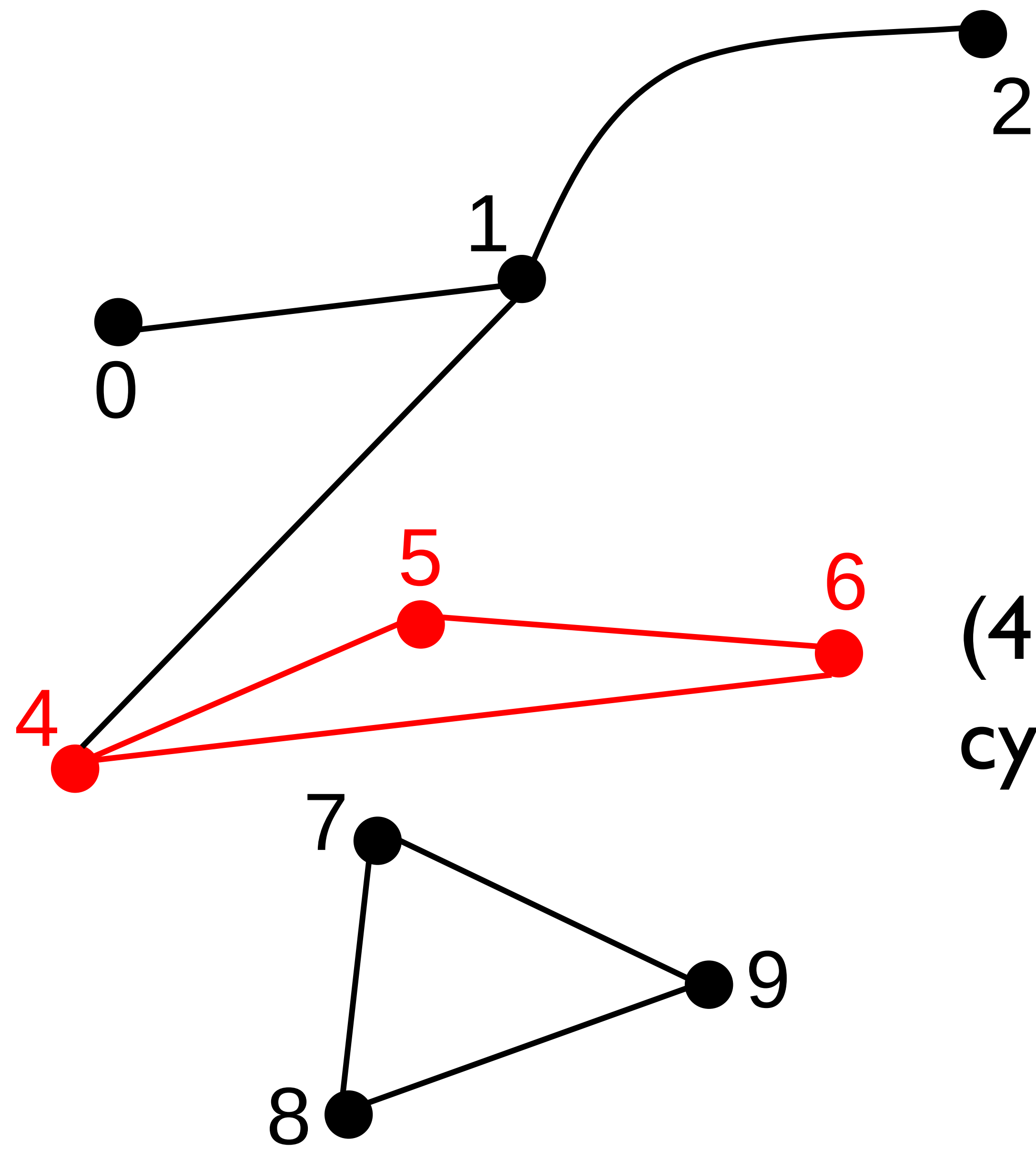


$V = \{0, \ldots, 9\}$
$E = \{(0,1), (1,2), (1,4), (4,5), (4,6), (5,6), (7,8), (7,9), (8,9)\}$

no two vertices in the sequence can be identical (except first and last)

Definition:
A ***path*** is a non-self-intersecting sequence of vertices such that there is an edge between consecutive vertices in the sequence.

Paths, cycles, and connectivity

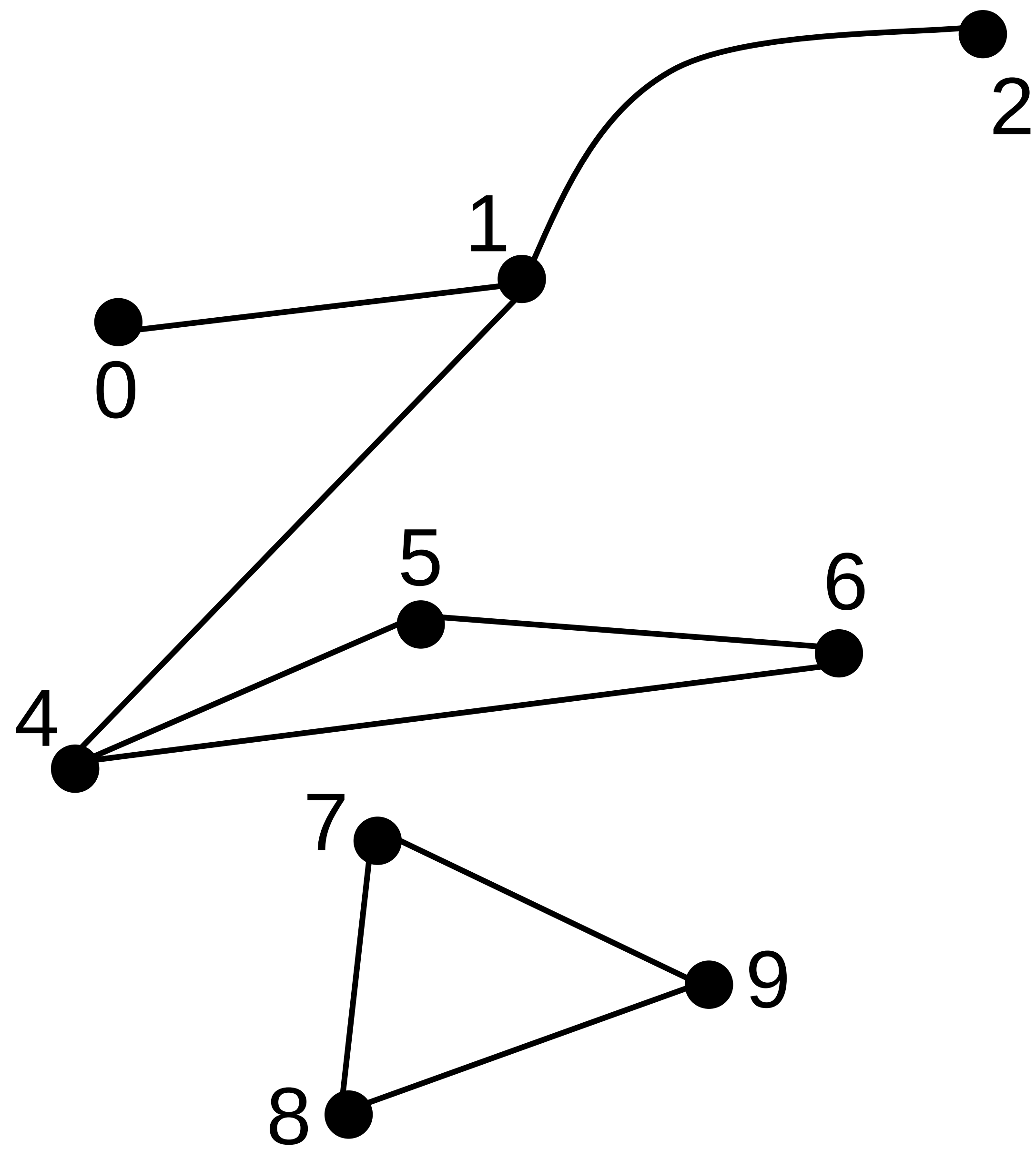


$V = \{0, \ldots, 9\}$
$E = \{(0,1), (1,2),$
$(1,4), (4,5), (4,6),$
$(5,6), (7,8),$
$(7,9), (8,9)\}$

(4,5,6,4) is a cycle

Definition:
A ***cycle*** is a path with same start and end vertex.

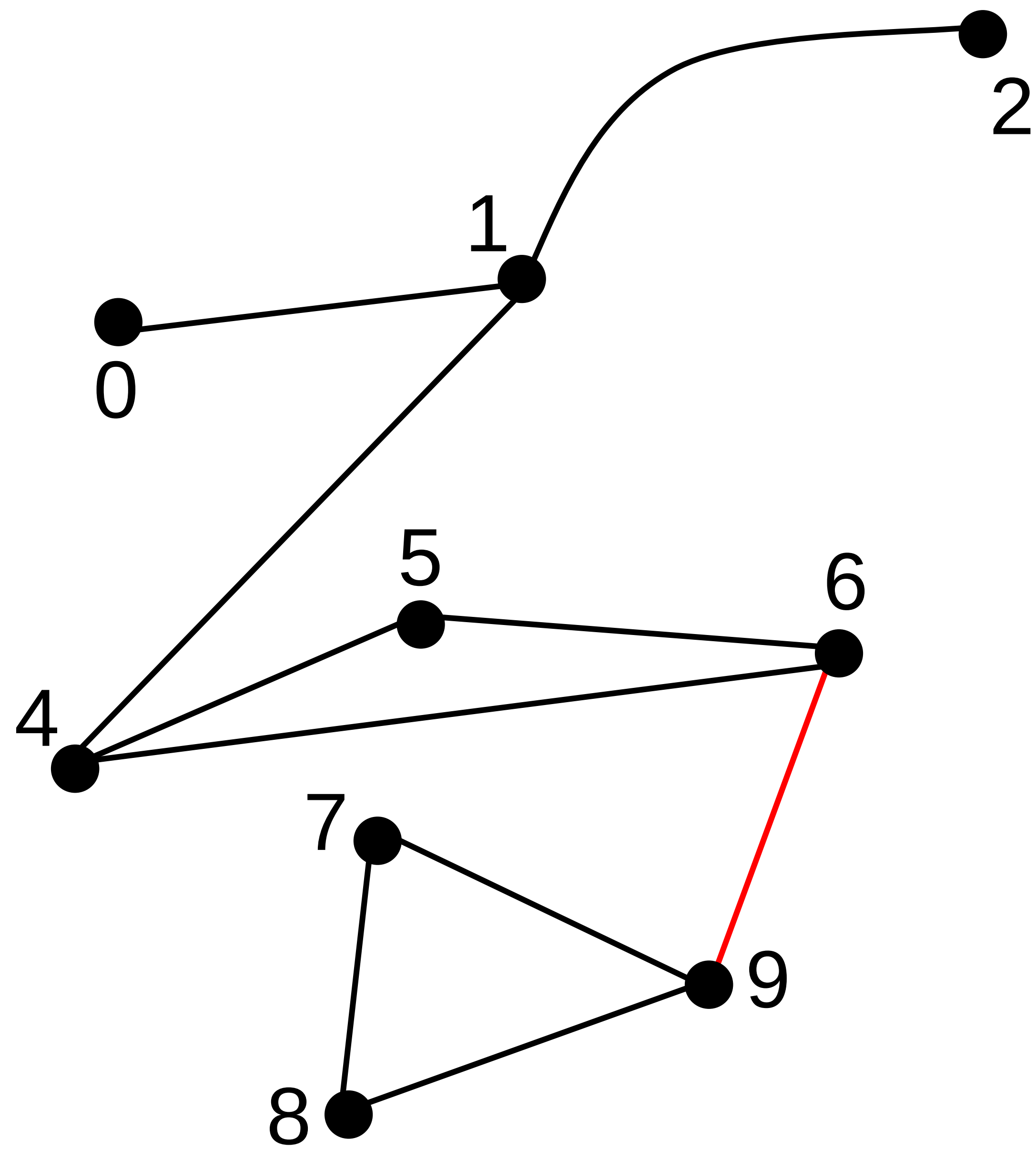
Paths, cycles, and connectivity



$V = \{0, \dots, 9\}$
$E = \{(0,1), (1,2), (1,4), (4,5), (4,6), (5,6), (7,8), (7,9), (8,9)\}$

Definition:
Graph in which there is a path between any two vertices is called ***connected***.

Paths, cycles, and connectivity

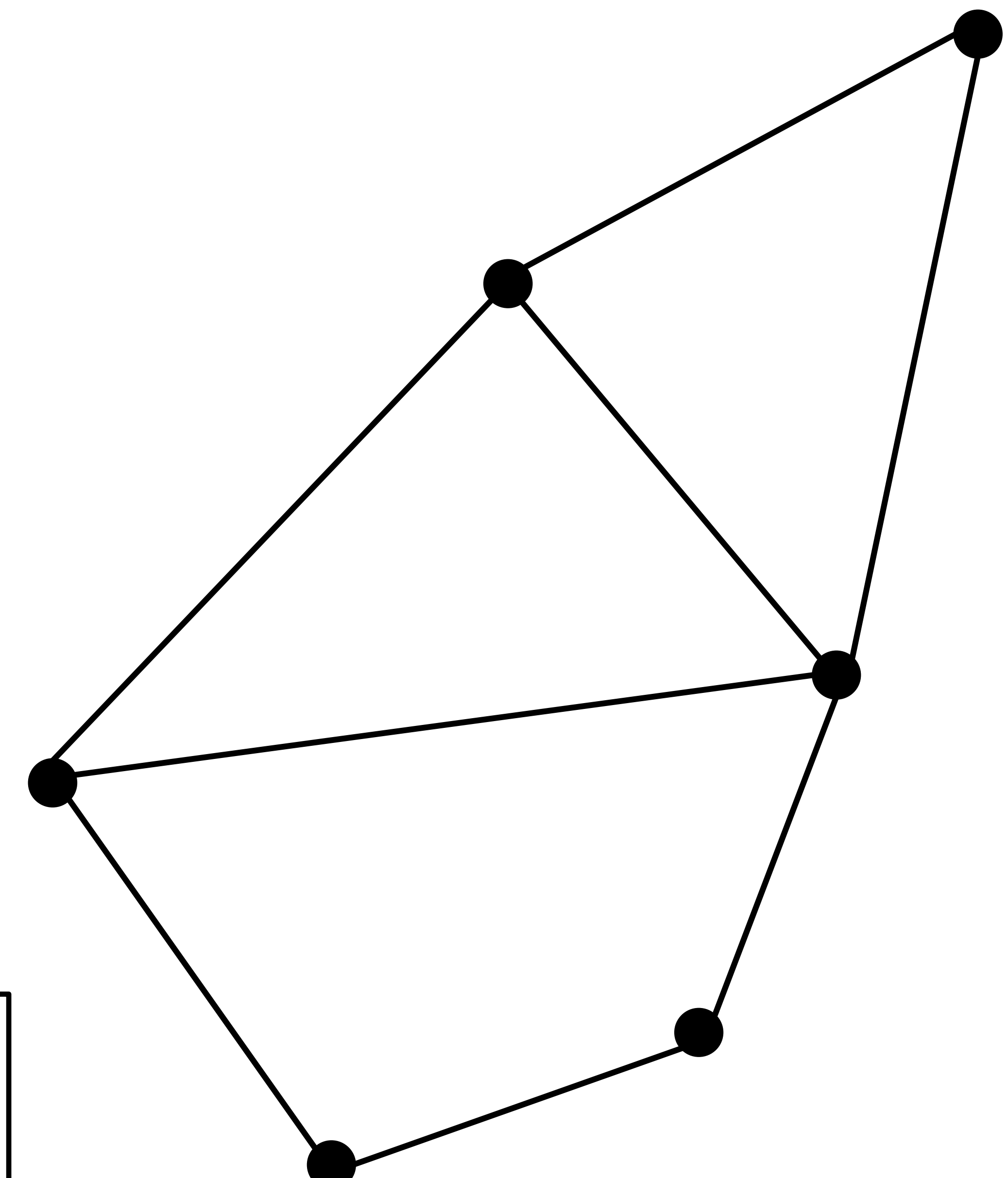


$V = \{0, \dots, 9\}$
$E = \{(0,1), (1,2), (1,4), (4,5), (4,6), (5,6), (7,8), (7,9), (8,9), (6,9)\}$

Definition:
Graph in which there is a path between any two vertices is called ***connected***.

Let's practice some graph notions

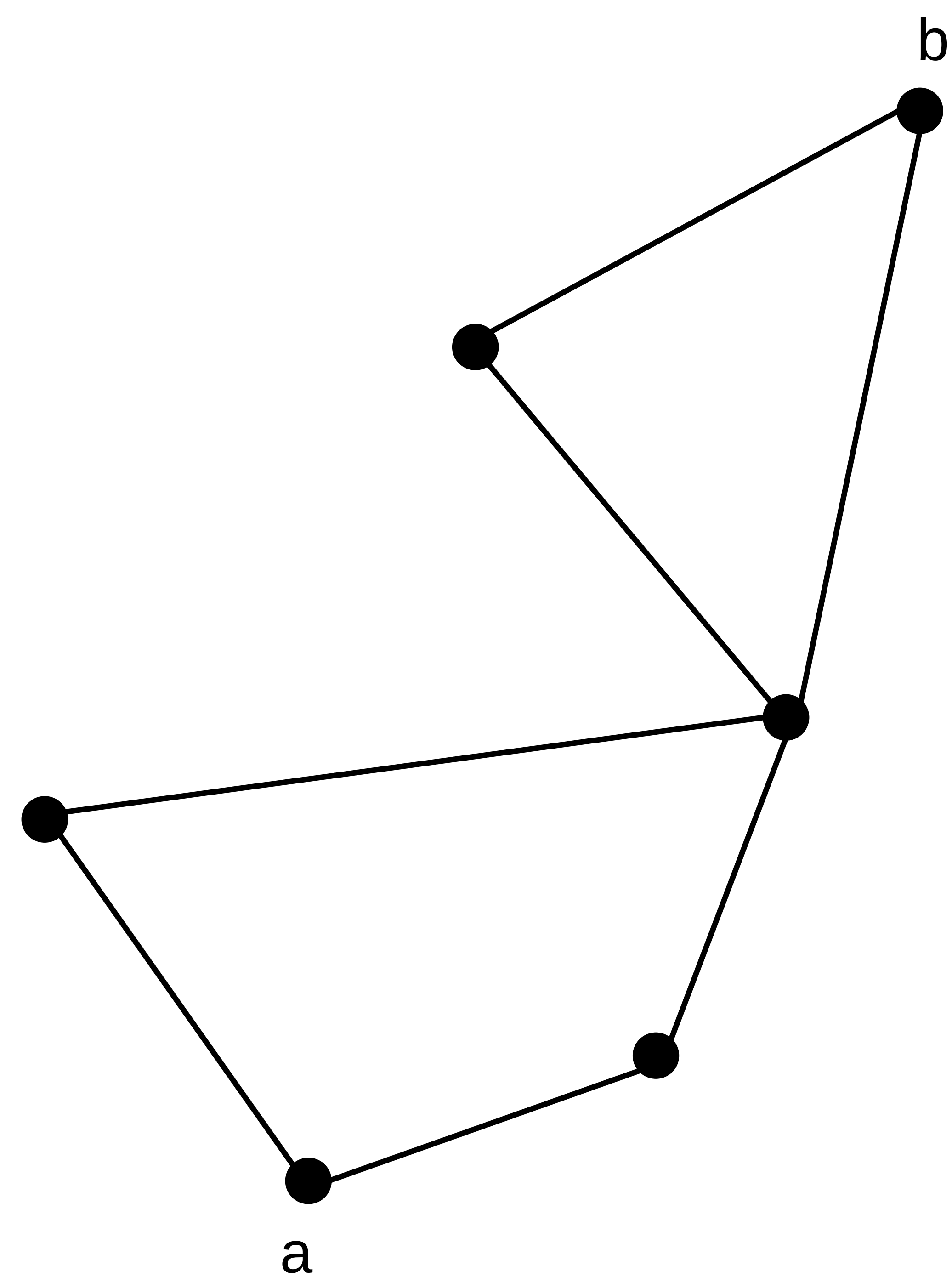
How many cycles does this graph have?



Quiz time (https://flux.qa)
Clayton: AXXULH
Malaysia: LWERDE

Let's practice some graph notions

How many paths are there from a to b?

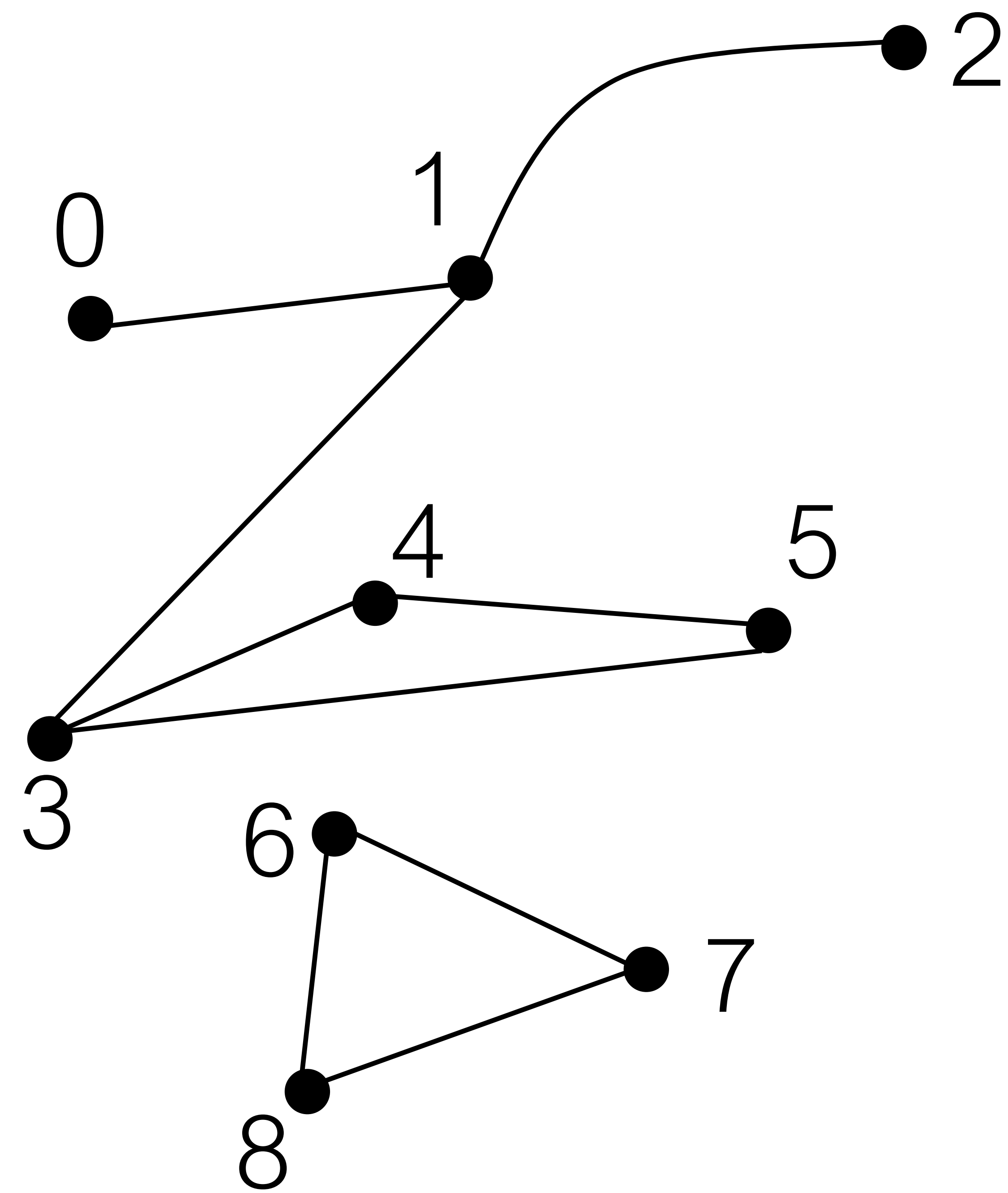


Quiz time (https://flux.qa)
Clayton: AXXULH
Malaysia: LWERDE

Representation as *adjacency matrix*

- table with one column and one row per vertex
- cell i, j represents existence of edge between vertices i and j (1- edge, 0 - no edge)

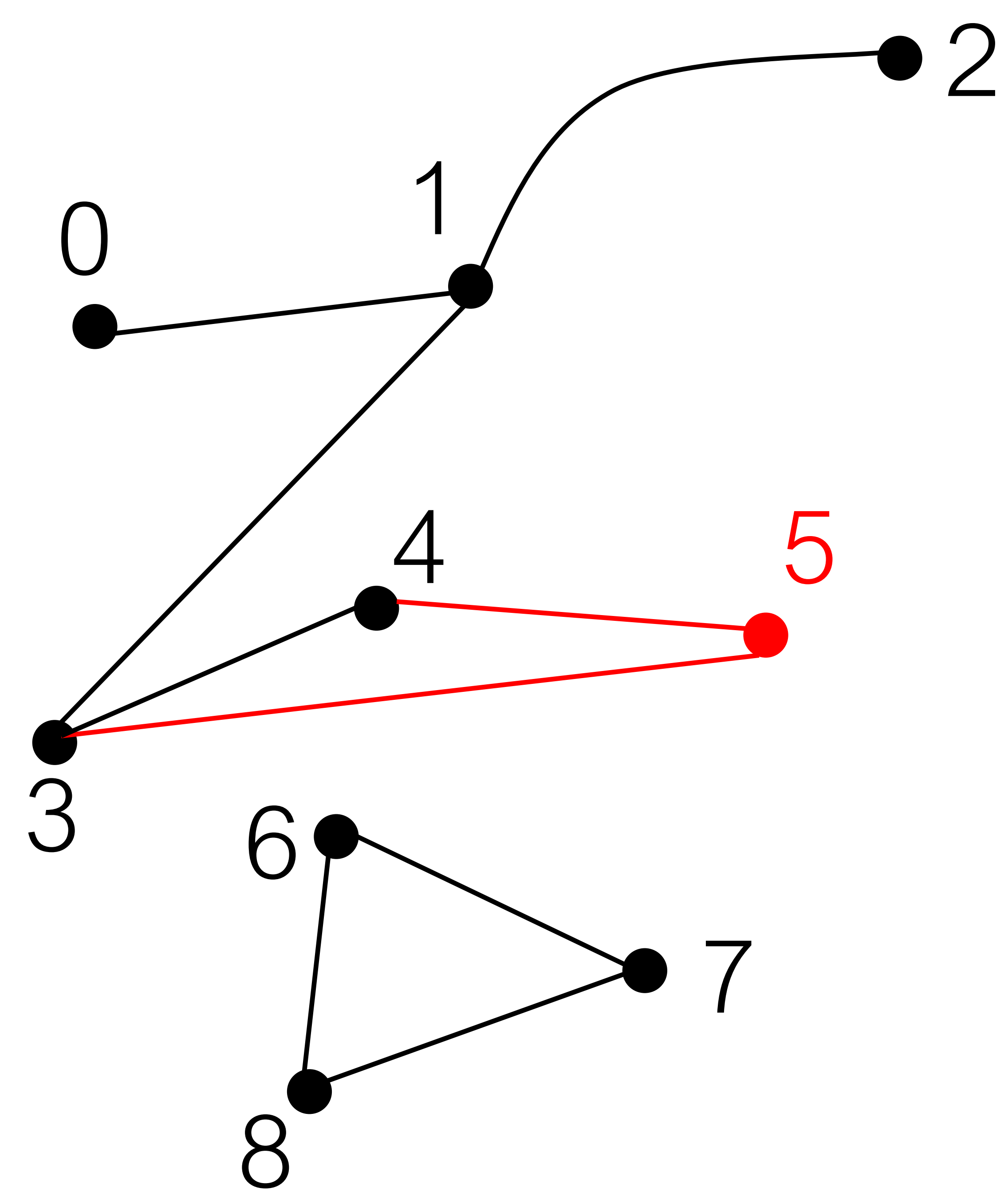
	0	1	2	3	4	5	6	7	8
0	0	1	0	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0
3	0	1	0	0	1	1	0	0	0
4	0	0	0	1	0	1	0	0	0
5	0	0	0	1	1	0	0	0	0
6	0	0	0	0	0	0	0	1	1
7	0	0	0	0	0	0	1	0	1
8	0	0	0	0	0	0	1	1	0



Representation as *adjacency matrix*

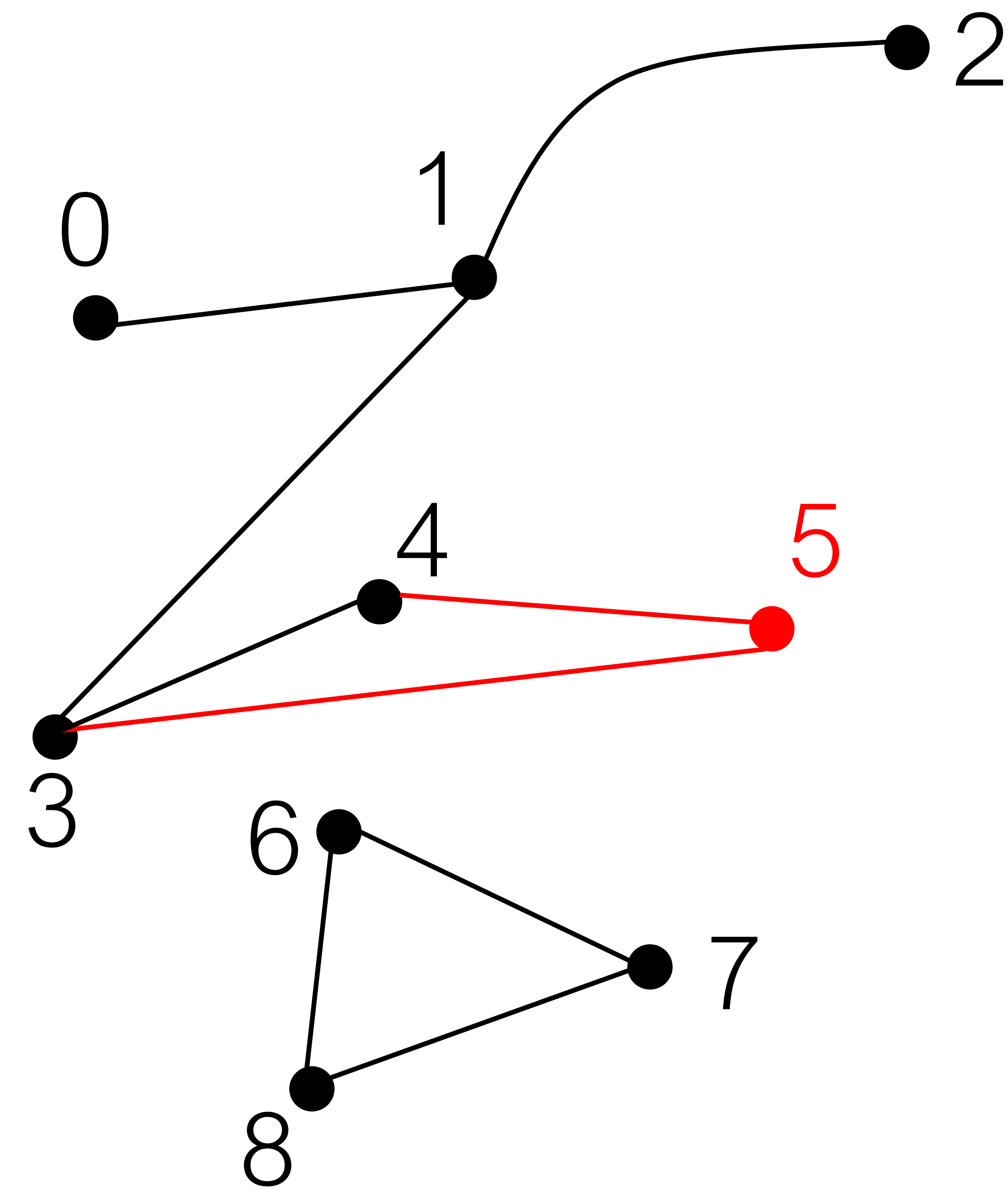
- table with one column and one row per vertex
- cell i, j represents existence of edge between vertices i and j (1- edge, 0 - no edge)

	0	1	2	3	4	5	6	7	8
0	0	1	0	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0
3	0	1	0	0	1	1	0	0	0
4	0	0	0	1	0	1	0	0	0
5	0	0	0	1	1	0	0	0	0
6	0	0	0	0	0	0	0	1	1
7	0	0	0	0	0	0	1	0	1
8	0	0	0	0	0	0	1	1	0



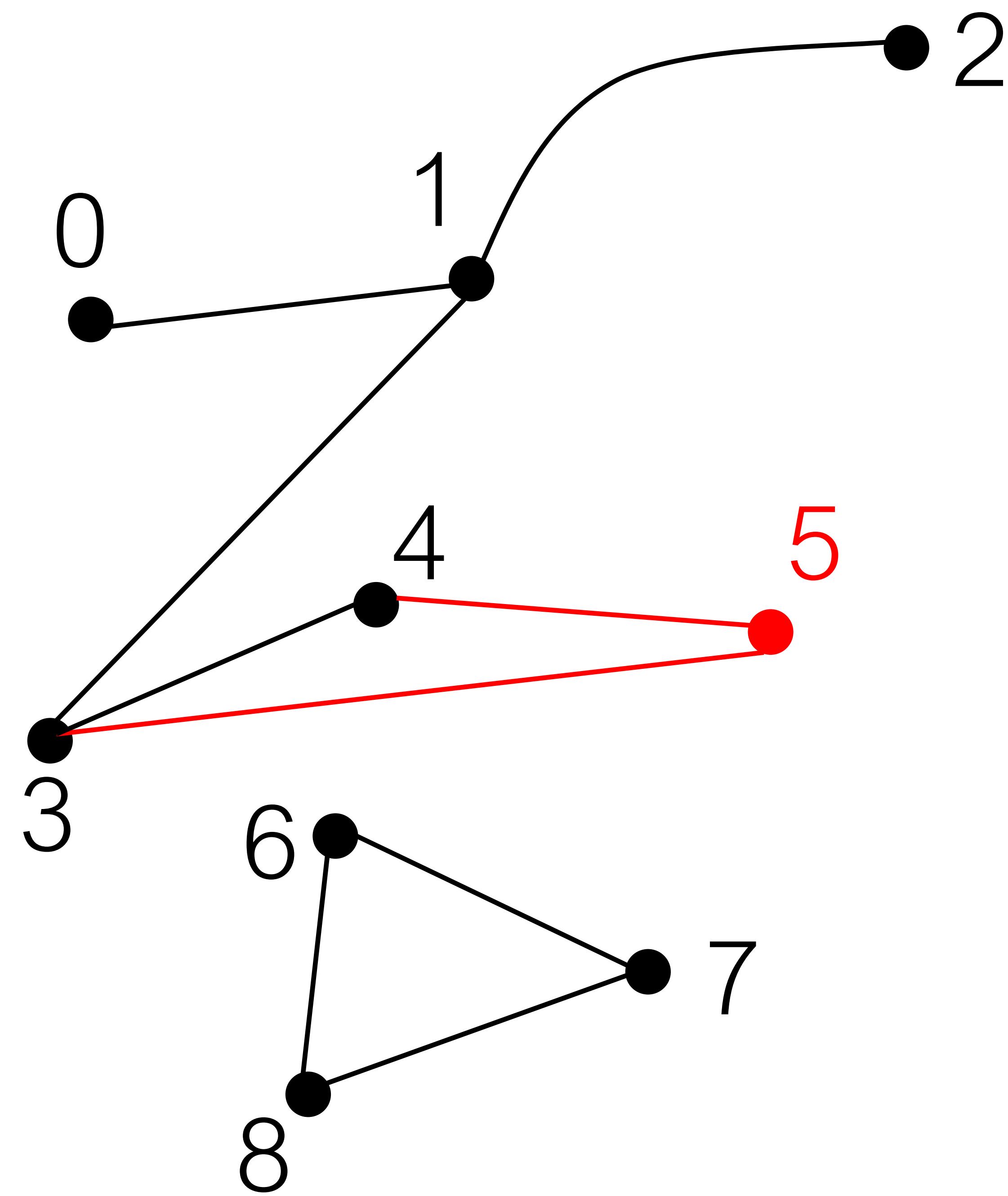
Representation as *adjacency matrix*

```
graph = \
    [[0,1,0,0,0,0,0,0,0],
     [1,0,1,1,0,0,0,0,0],
     [0,1,0,0,0,0,0,0,0],
     [0,1,0,0,1,1,0,0,0],
     [0,0,0,1,0,1,0,0,0],
     [0,0,0,1,1,0,0,0,0],
     [0,0,0,0,0,0,0,1,1],
     [0,0,0,0,0,0,1,0,1],
     [0,0,0,0,0,0,1,1,0]]
```



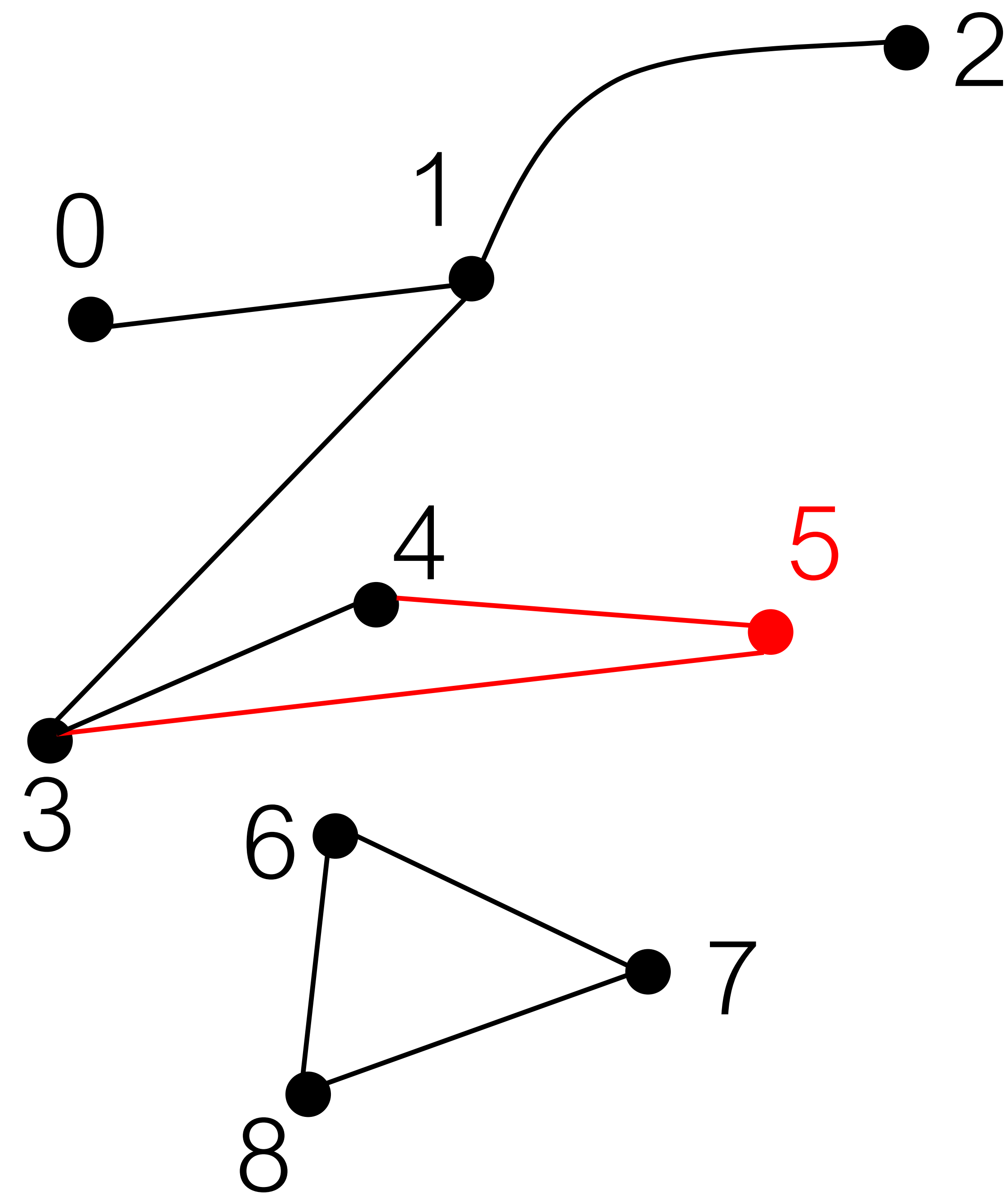
Representation as *adjacency matrix*

```
graph = \
    [[0,1,0,0,0,0,0,0,0],
     [1,0,1,1,0,0,0,0,0],
     [0,1,0,0,0,0,0,0,0],
     [0,1,0,0,1,1,0,0,0],
     [0,0,0,1,0,1,0,0,0],
     [0,0,0,1,1,0,0,0,0],
     [0,0,0,0,0,0,0,1,1],
     [0,0,0,0,0,0,1,0,1],
     [0,0,0,0,0,0,1,1,0]]

def neighbours(i, g):
    """I: vertex i, graph g
       O: neighbours of i
       For example:
       >>> neighbours(5, graph)
       [3, 4]
    """


    ???
```



Representation as *adjacency matrix*

```
graph = \
    [[0,1,0,0,0,0,0,0,0],
     [1,0,1,1,0,0,0,0,0],
     [0,1,0,0,0,0,0,0,0],
     [0,1,0,0,1,1,0,0,0],
     [0,0,0,1,0,1,0,0,0],
     [0,0,0,1,1,0,0,0,0],
     [0,0,0,0,0,0,0,1,1],
     [0,0,0,0,0,0,1,0,1],
     [0,0,0,0,0,0,1,1,0]]

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



Where am I?

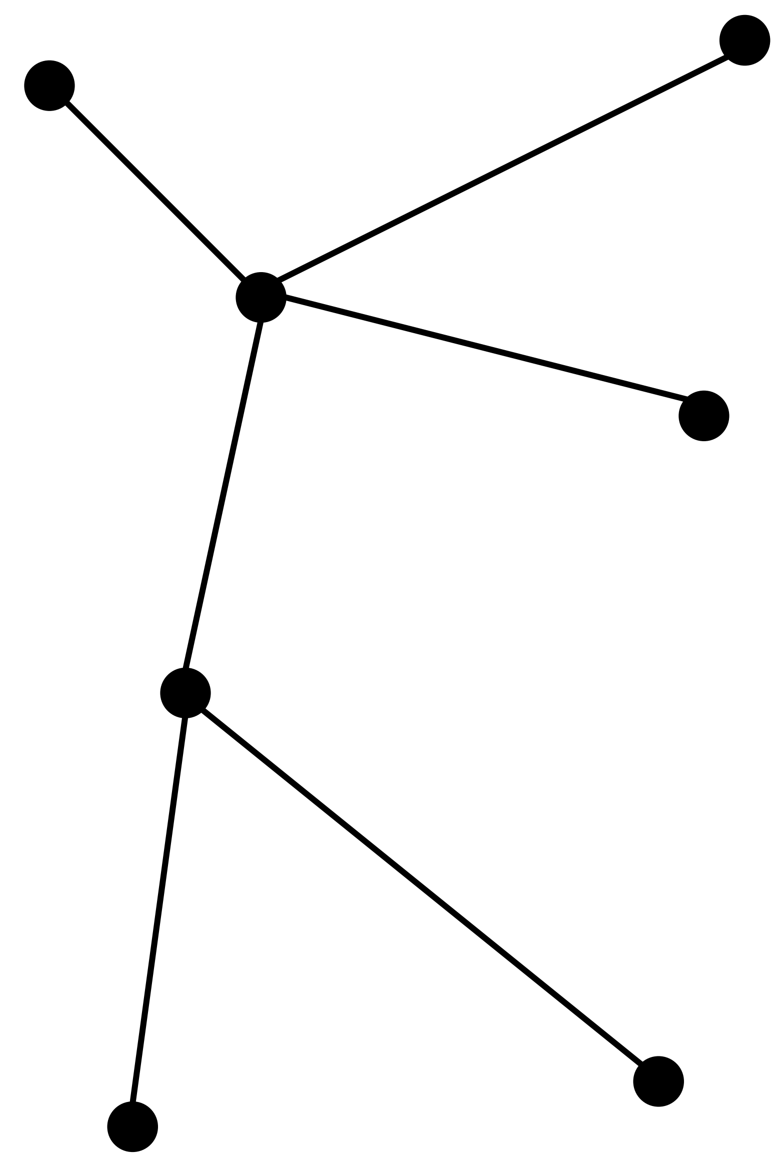
1. Graphs
2. Trees and Spanning Trees
3. Prims algorithm (simplified)
4. *Problem decomposition (if time left)*

Trees

Definition

A simple graph $T = (V, E)$ is called a ***tree*** if it is

- connected and
- has no cycles.



Quiz time (https://flux.qa)
Clayton: AXXULH
Malaysia: LWERDE

Trees

Definition

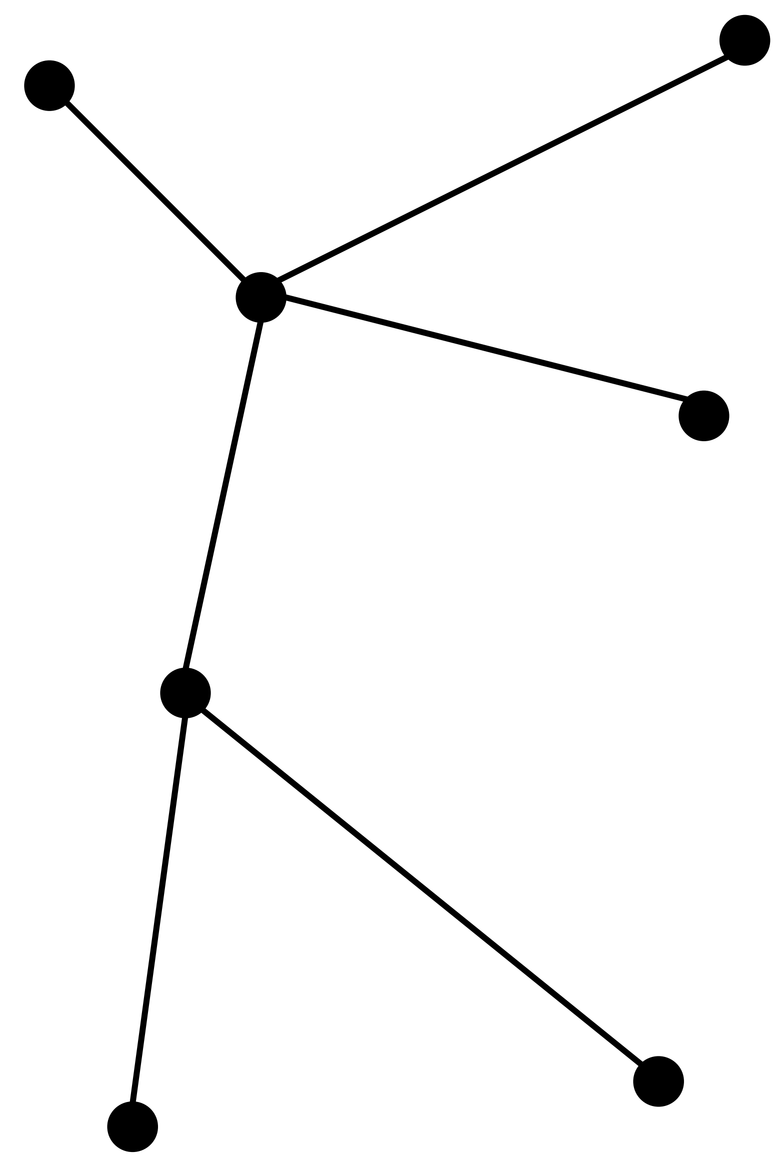
A simple graph $T = (V, E)$ is called a ***tree*** if it is

- connected and
- has no cycles.

Properties

A tree

- is *minimally connected*, i.e., removing any edge makes graph disconnected
- contains *unique path* between any two vertices $v, w \in E$



Trees

Definition

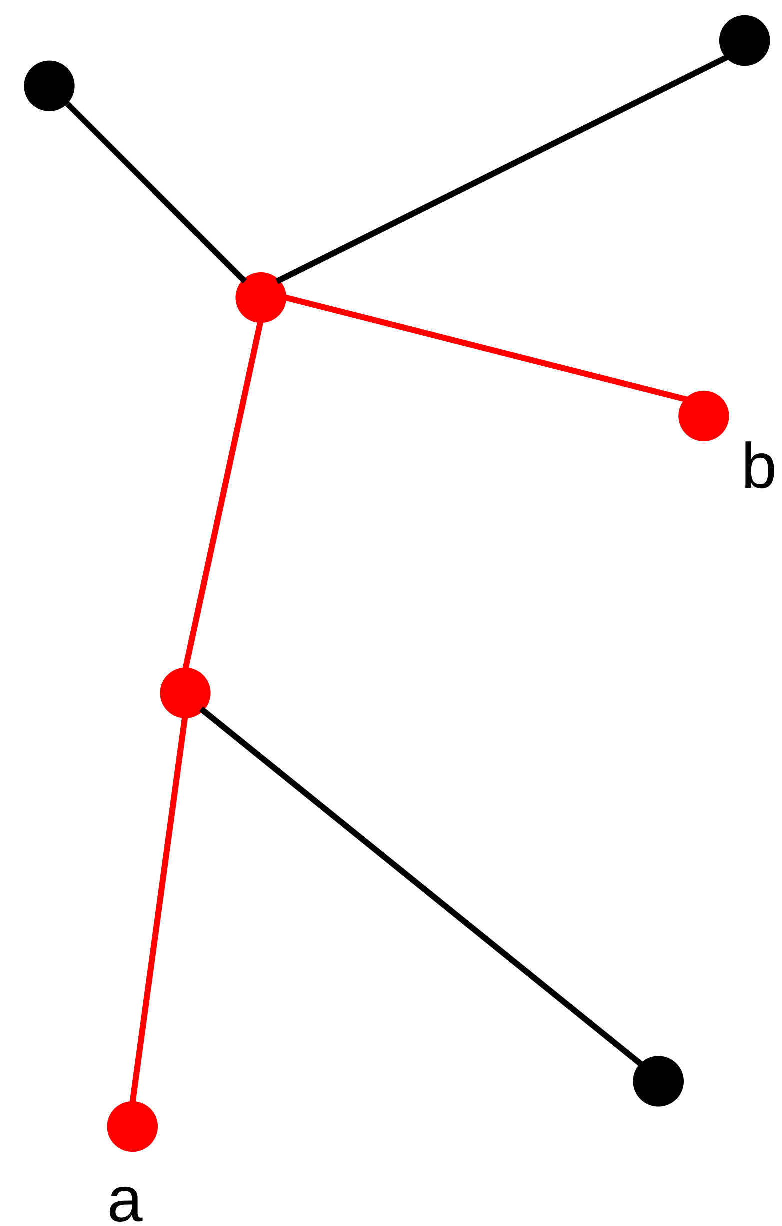
A simple graph $T = (V, E)$ is called a ***tree*** if it is

- connected and
- has no cycles.

Properties

A tree

- is *minimally connected*, i.e., removing any edge makes graph disconnected
- contains *unique path* between any two vertices $v, w \in E$



Trees

Definition

A simple graph $T = (V, E)$ is called a ***tree*** if it is

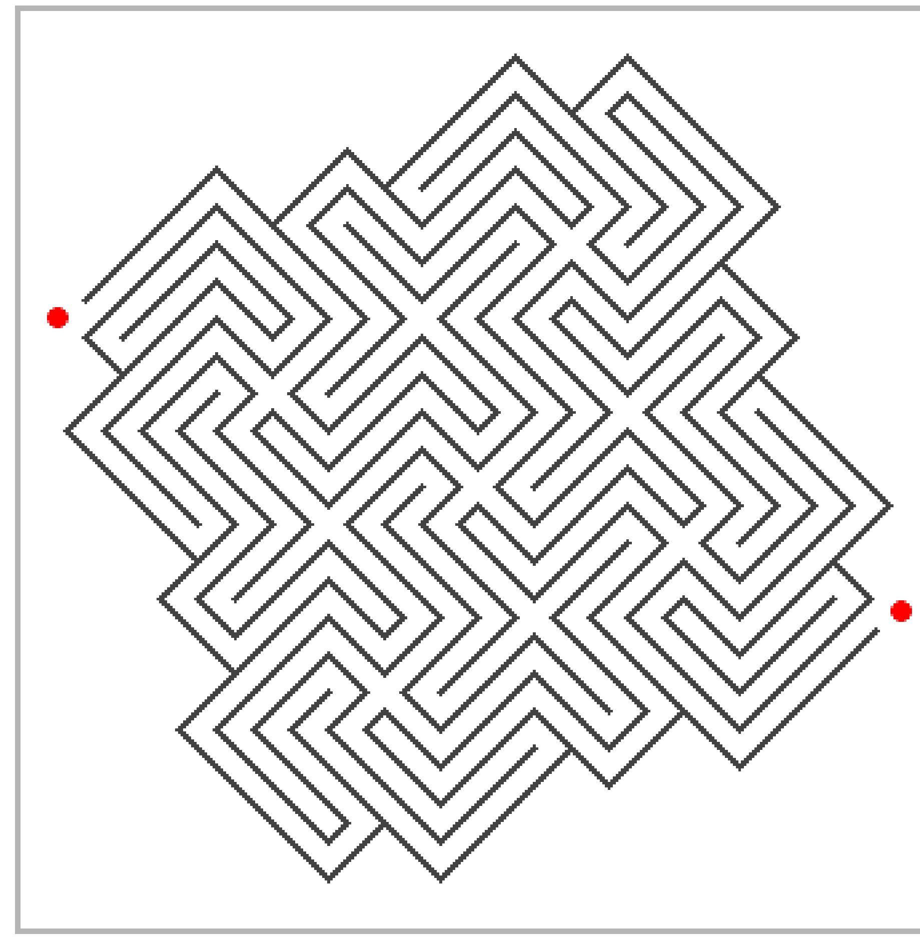
- connected and
- has no cycles.

Properties

A tree

- is *minimally connected*, i.e., removing any edge makes graph disconnected
- contains *unique path* between any two vertices $v, w \in E$

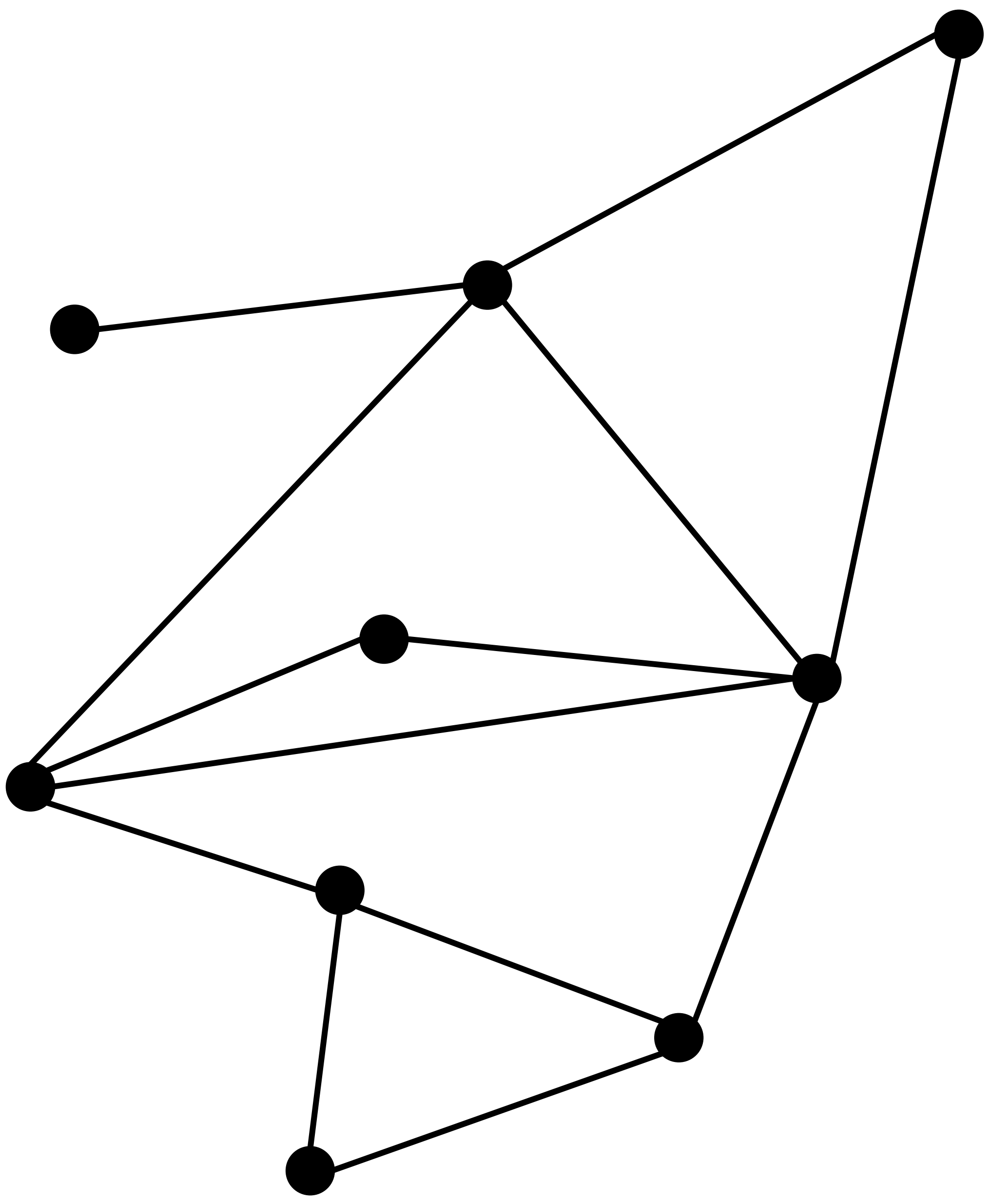
Trees are a lot like mazes, if you do not allow loops in the maze



Spanning tree of a graph

Assume $G = (V, E)$ is simple and connected graph.

A ***spanning tree*** of G is a subgraph $T = (V, F)$ of G that

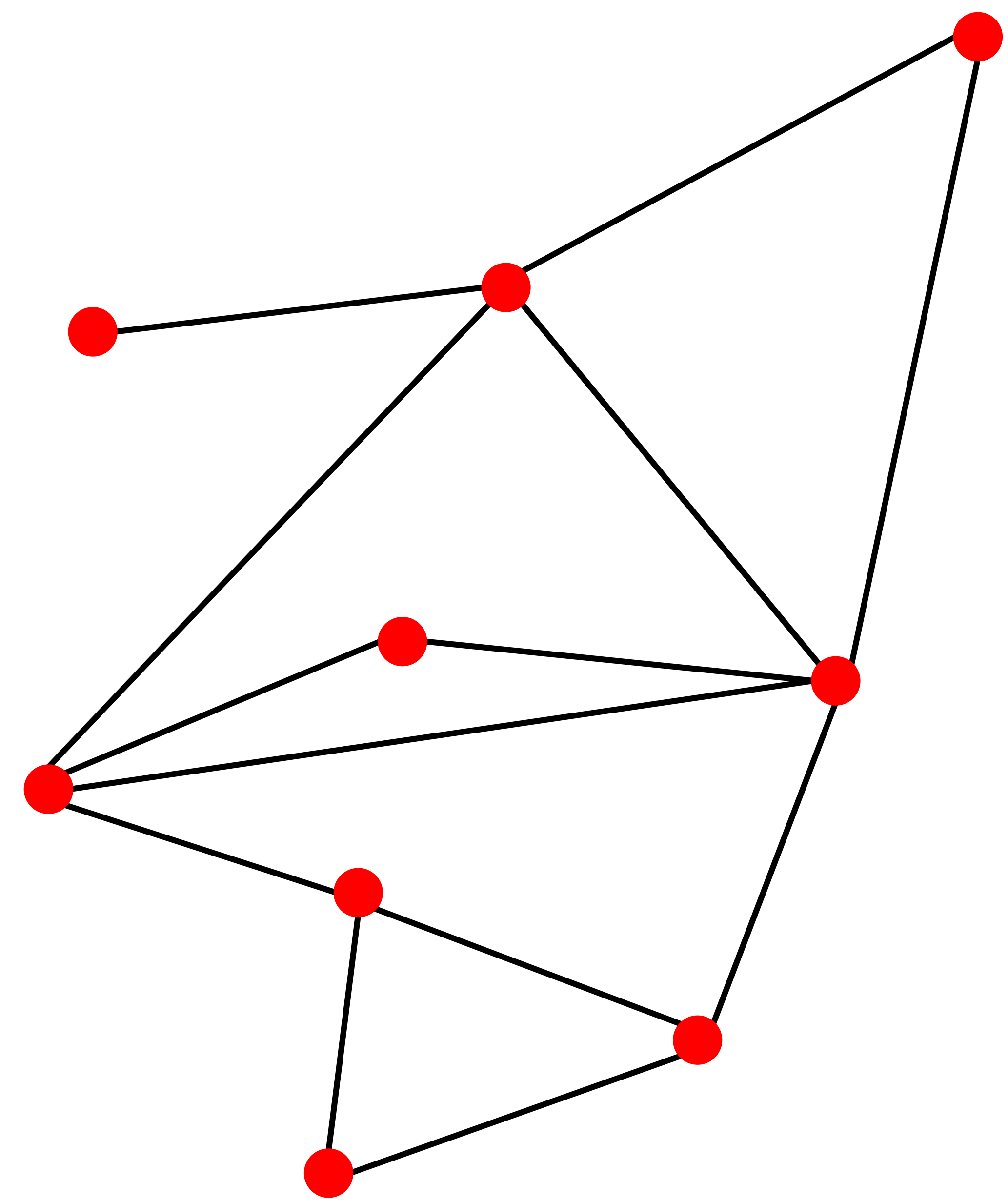


Spanning tree of a graph

Assume $G = (V, E)$ is simple and connected graph.

A ***spanning tree*** of G is a subgraph $T = (V, F)$ of G that

- contains all vertices of G and

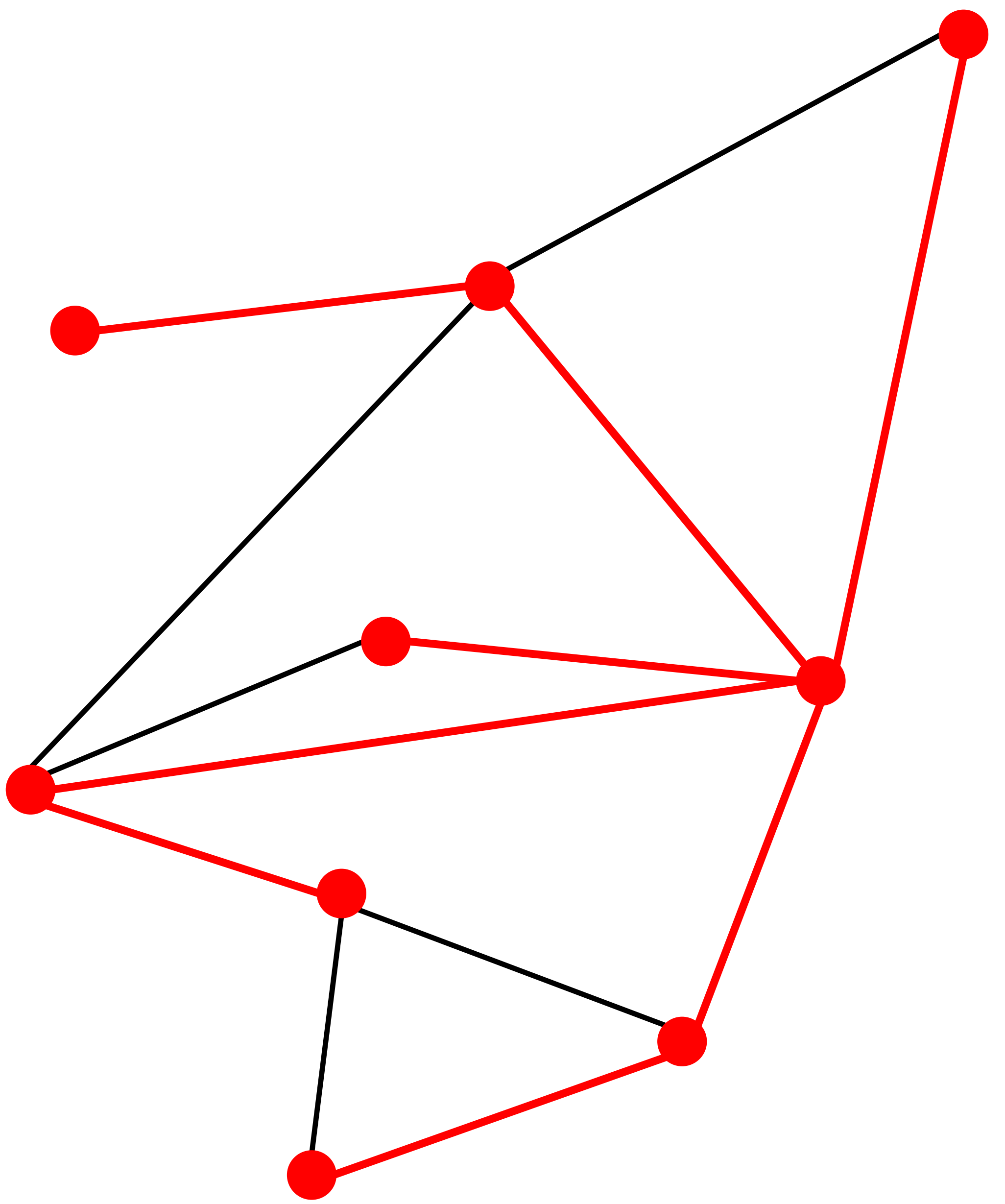


Spanning tree of a graph

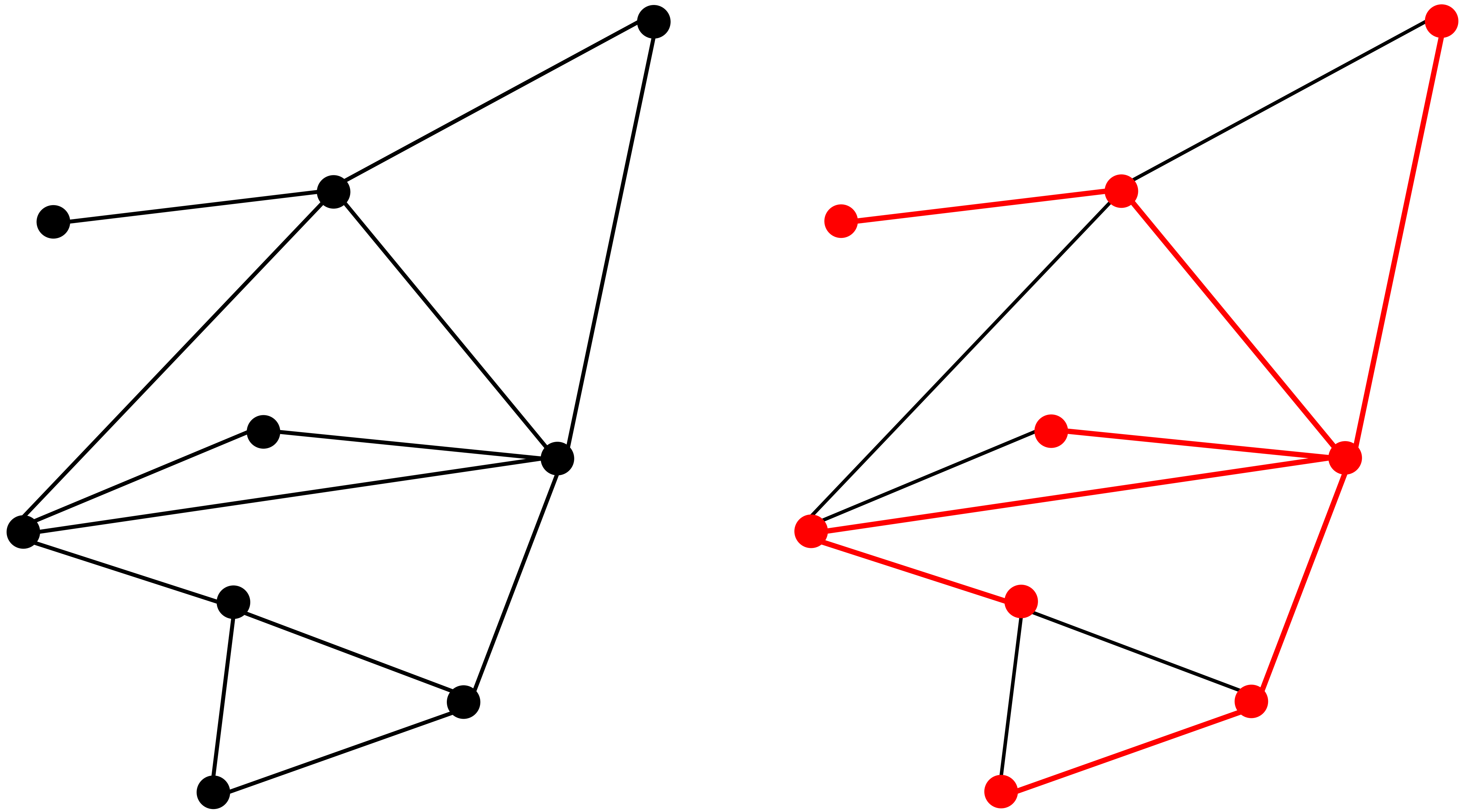
Assume $G = (V, E)$ is simple and connected graph.

A ***spanning tree*** of G is a subgraph $T = (V, F)$ of G that

- contains all vertices of G and
- that is a tree (i.e., connected and cycle free)



Spanning tree of a graph

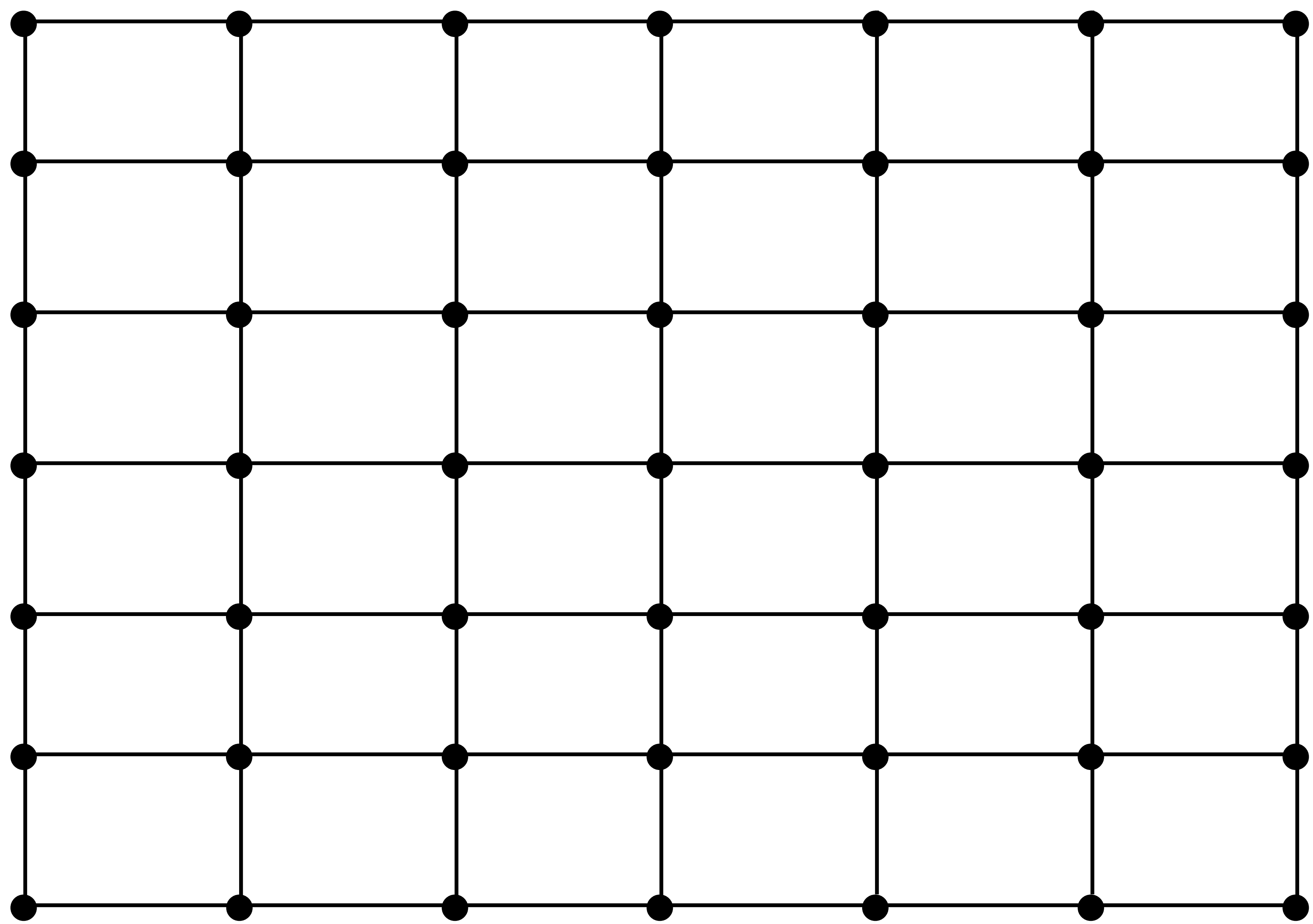


<u>Spanning Tree Problem</u>

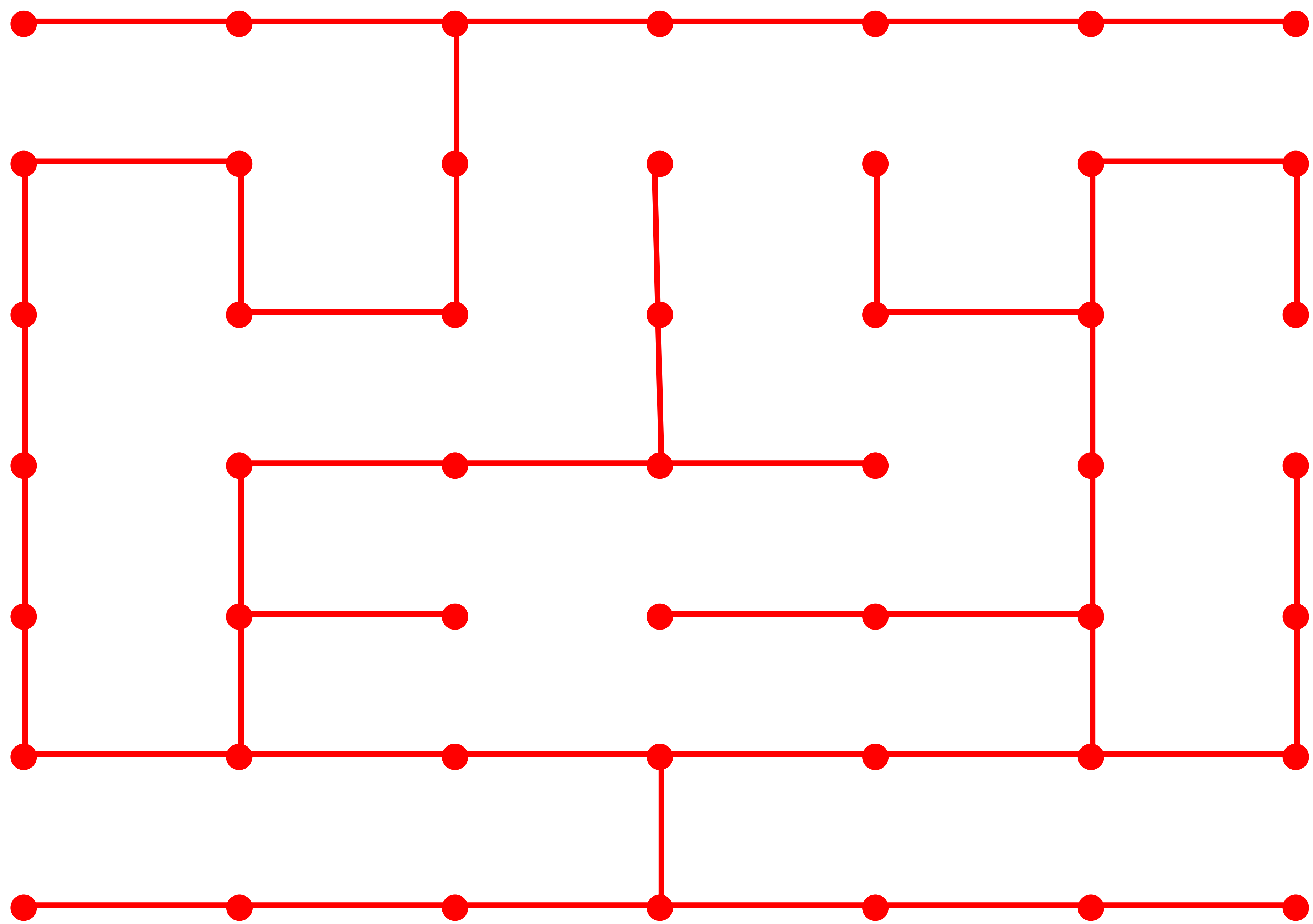
Input: adjacency matrix `graph` of connected graph

Output: adjacency matrix of spanning tree `tree` of `graph`

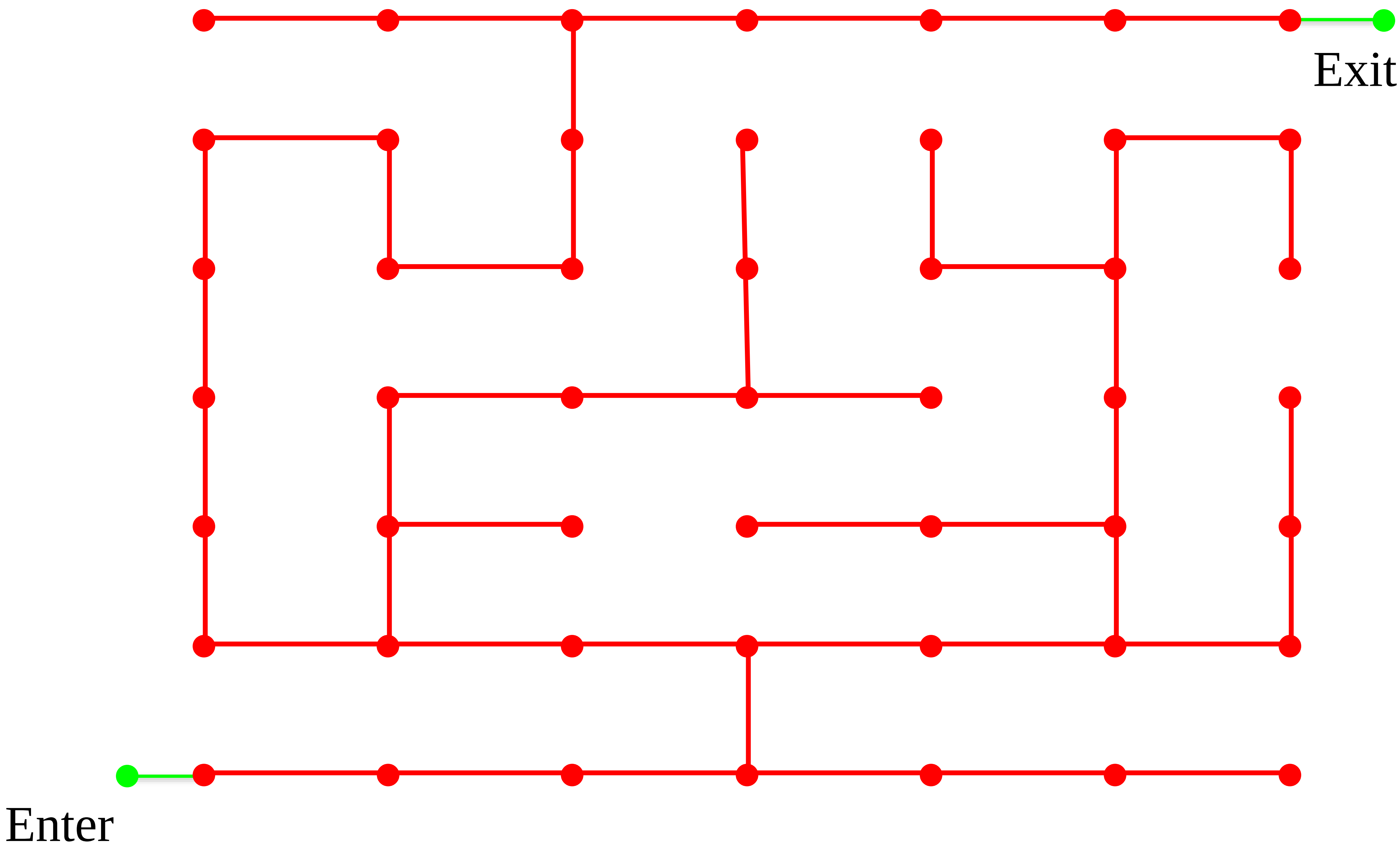
Algorithm for maze construction: start with a regular grid graph…



… then find a spanning tree



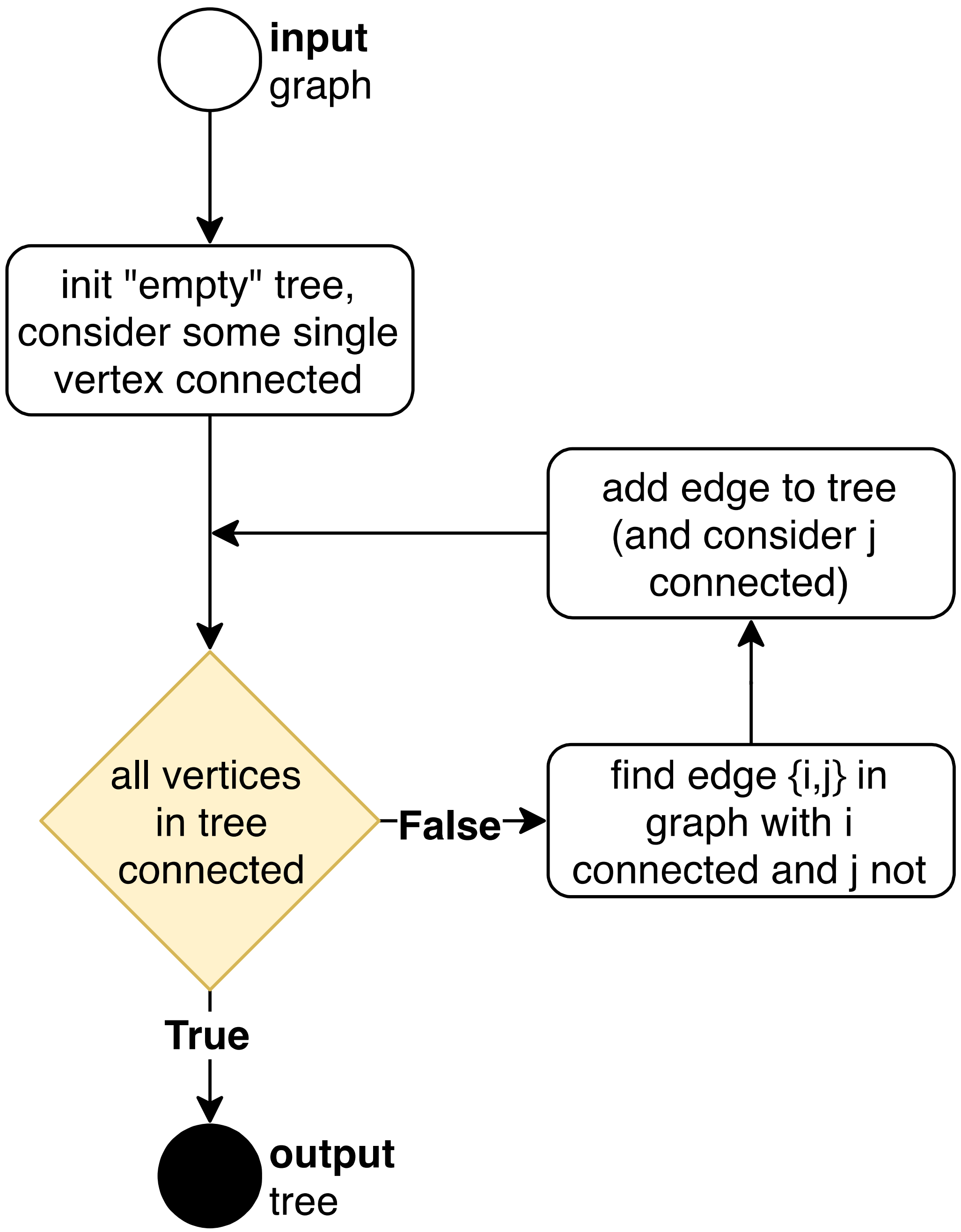
…and add entrance and exit



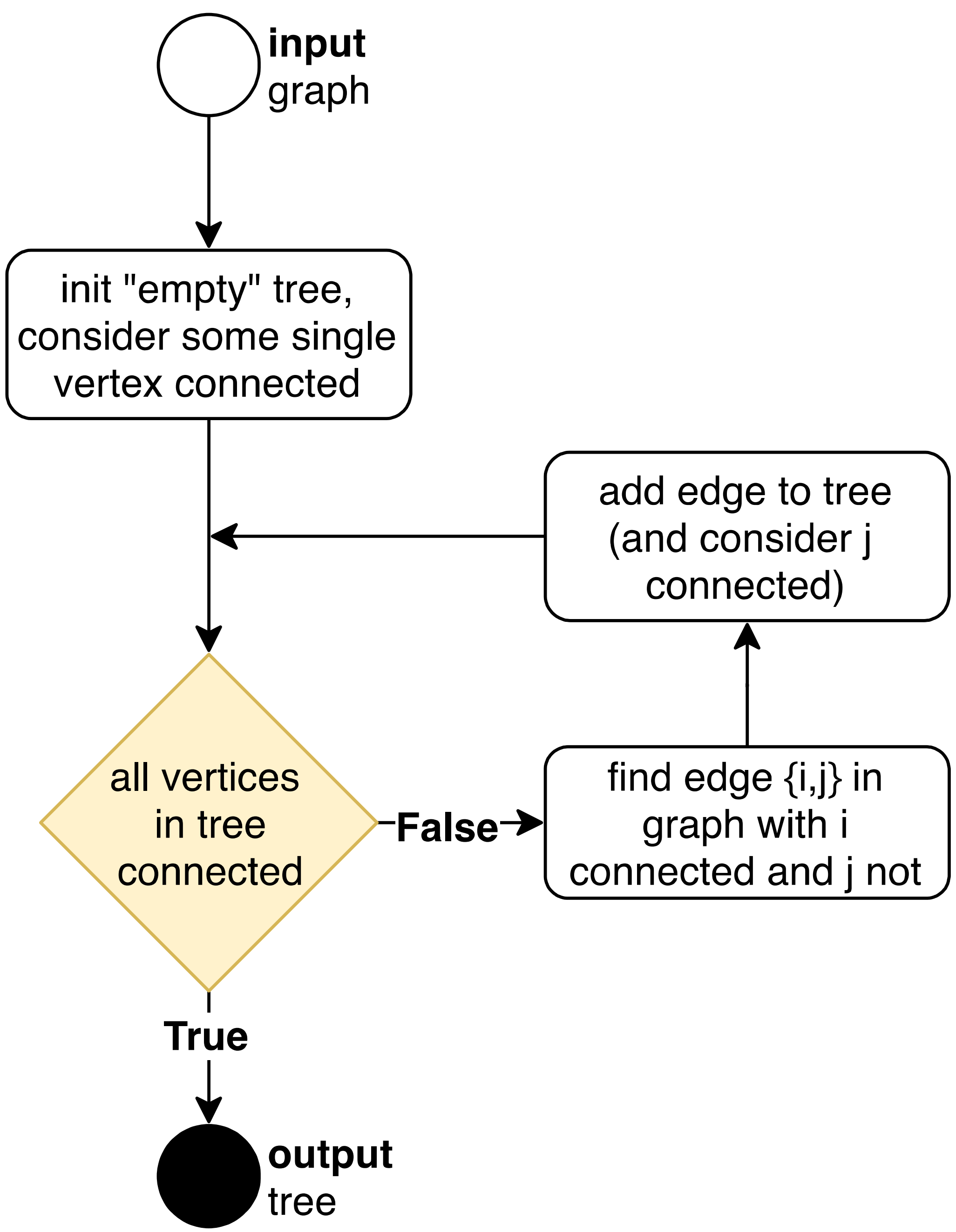
Where am I?

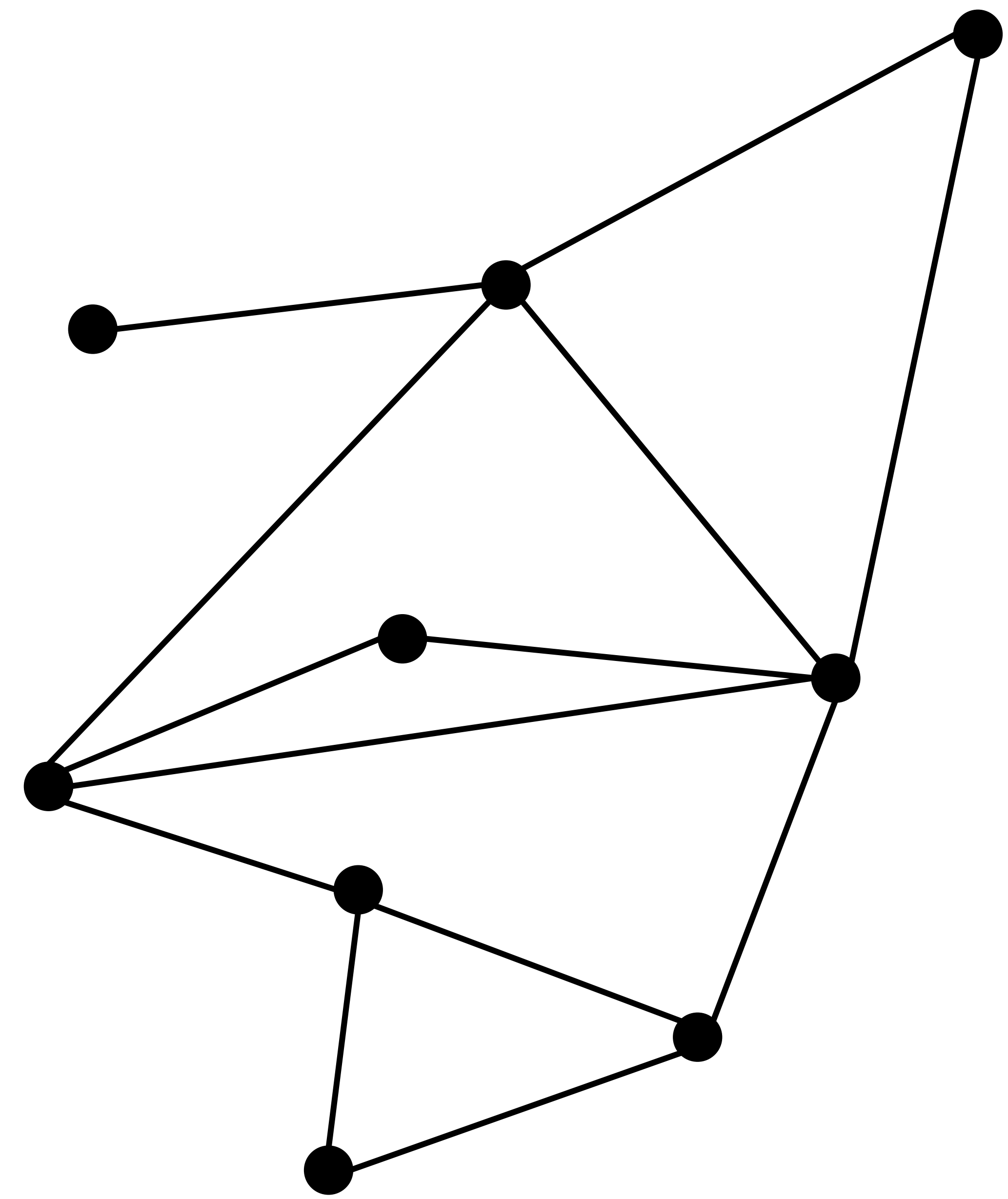
1. Graphs
2. Trees and Spanning Trees
3. Prims algorithm (simplified)
4. *Problem decomposition (if time left)*

Prim's algorithm for finding a spanning tree

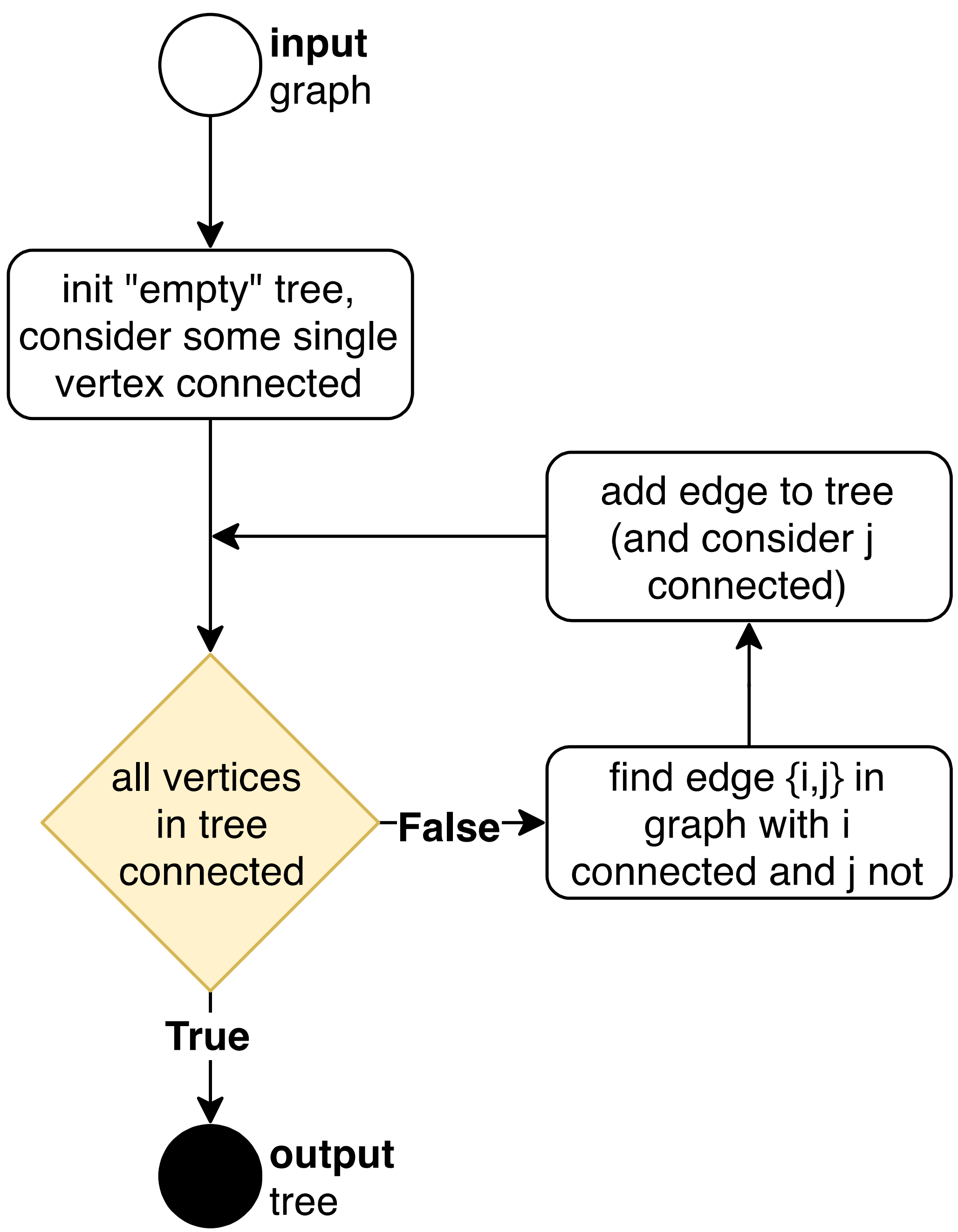


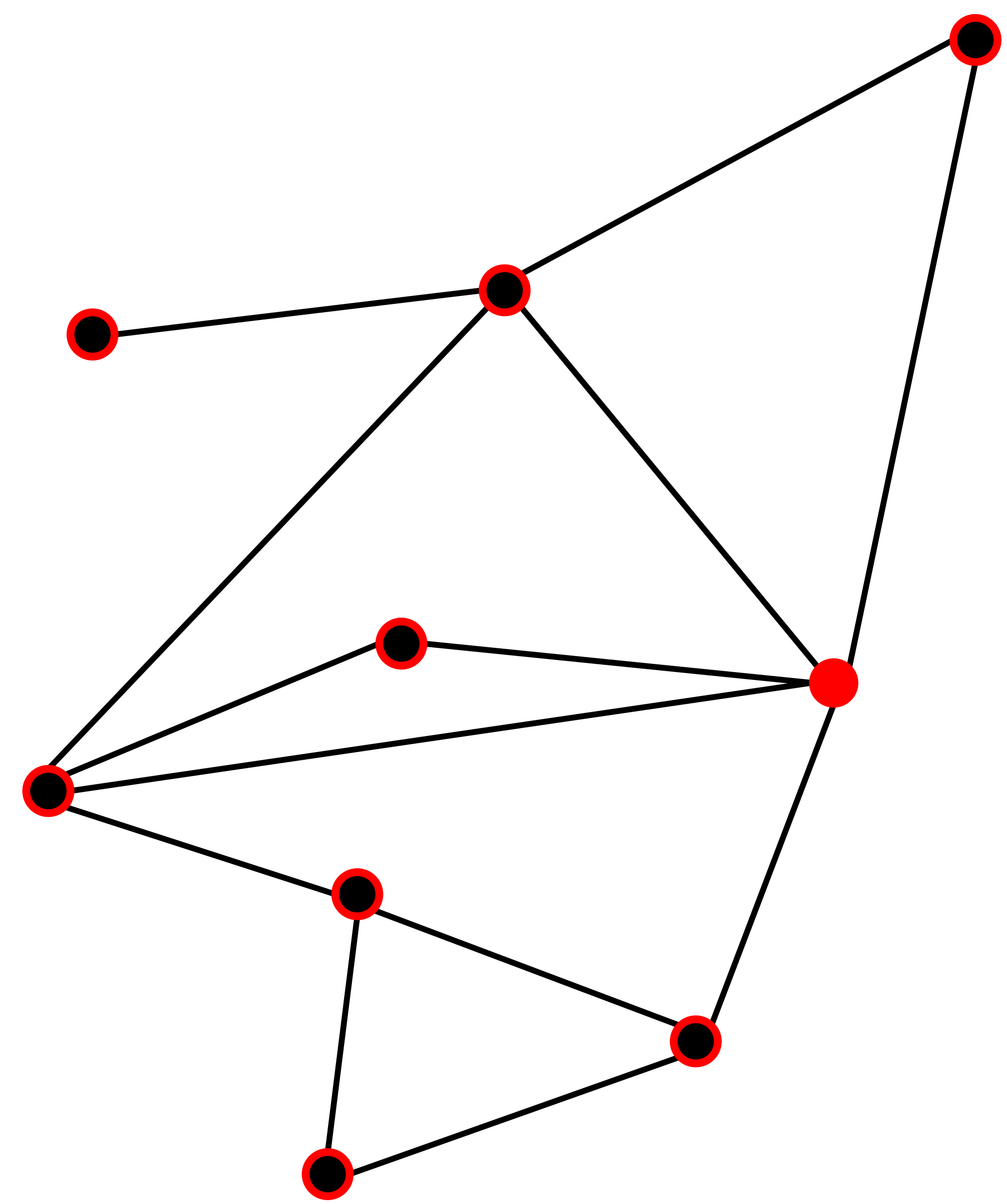
Prim's algorithm for finding a spanning tree



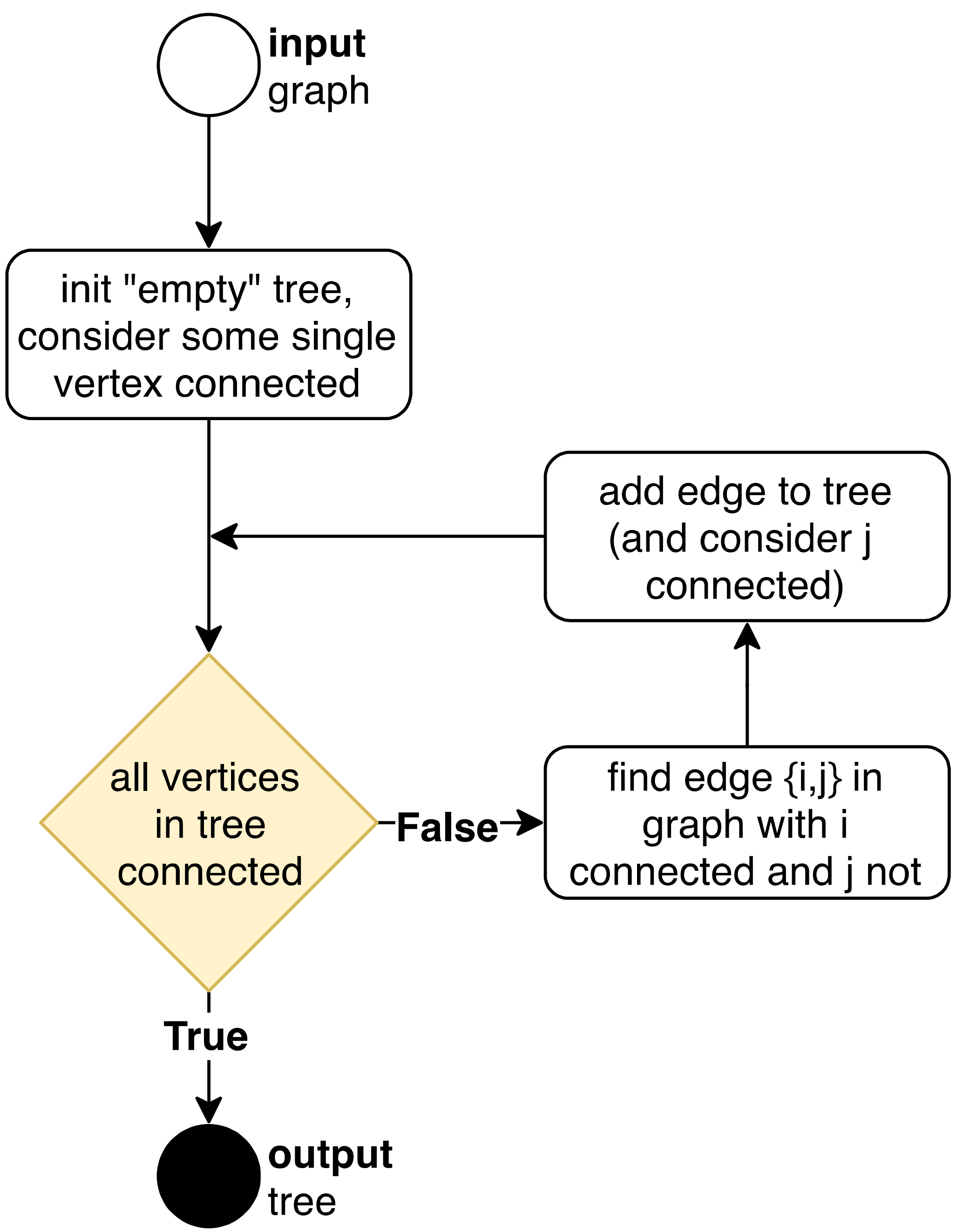


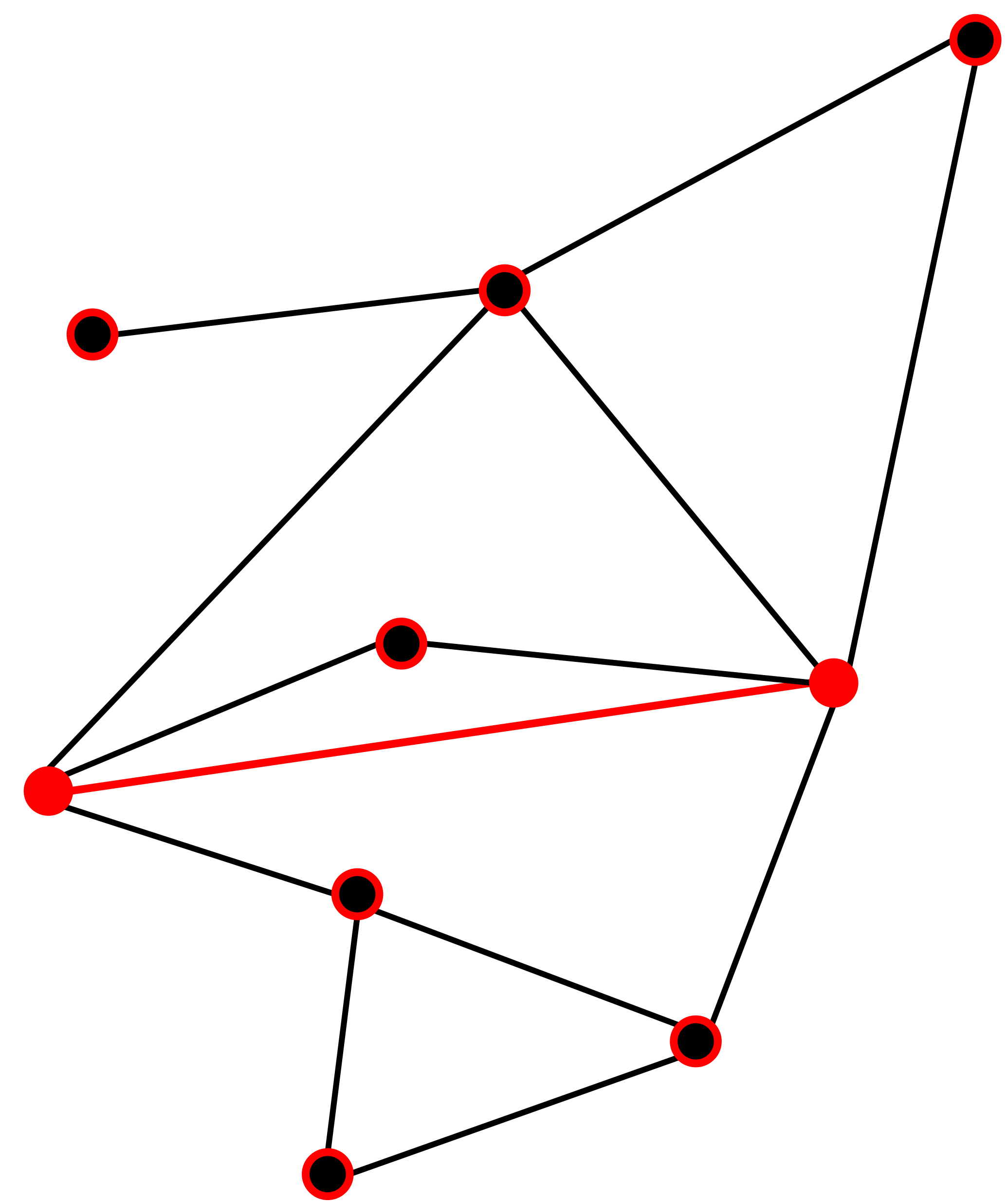
Prim's algorithm for finding a spanning tree



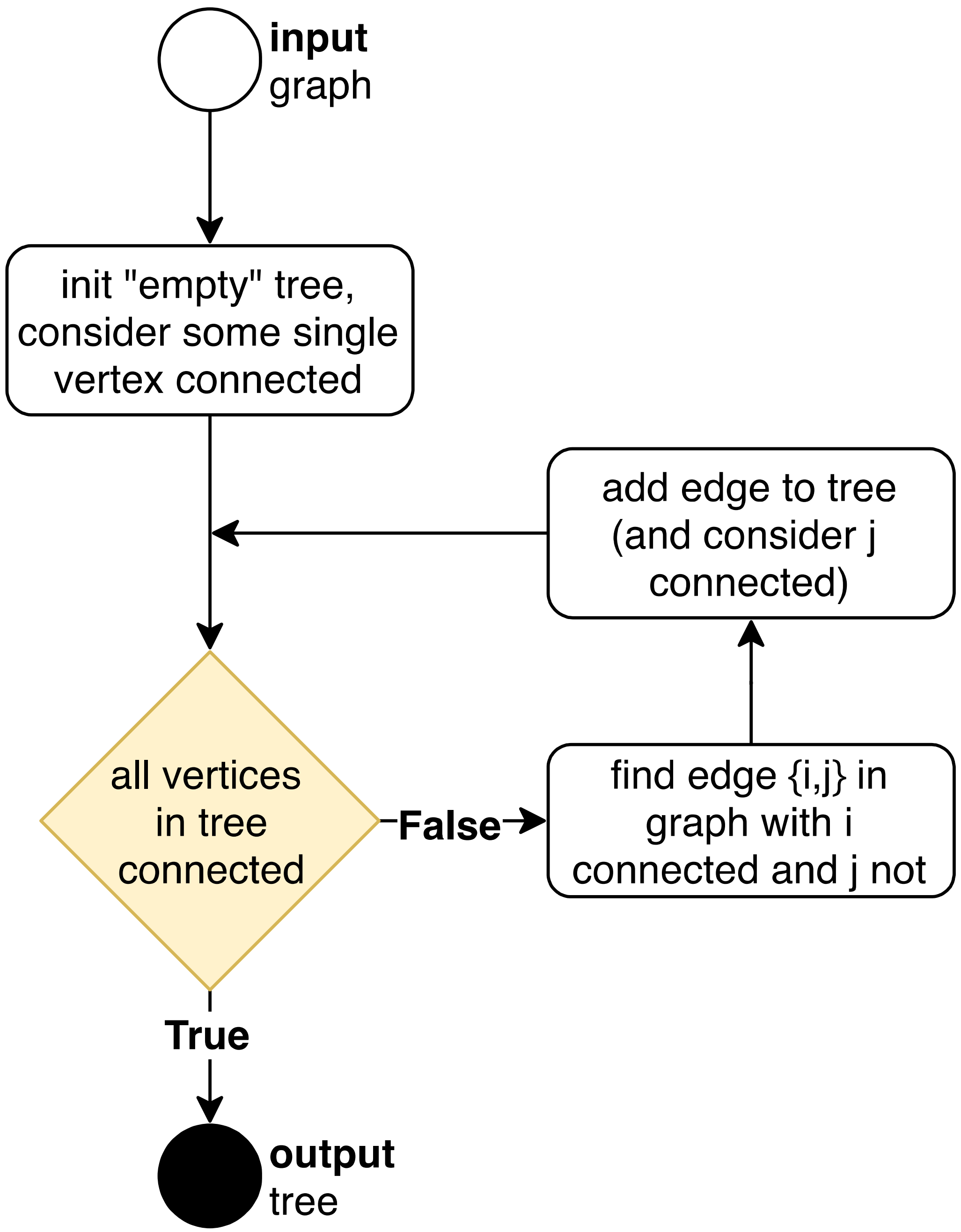


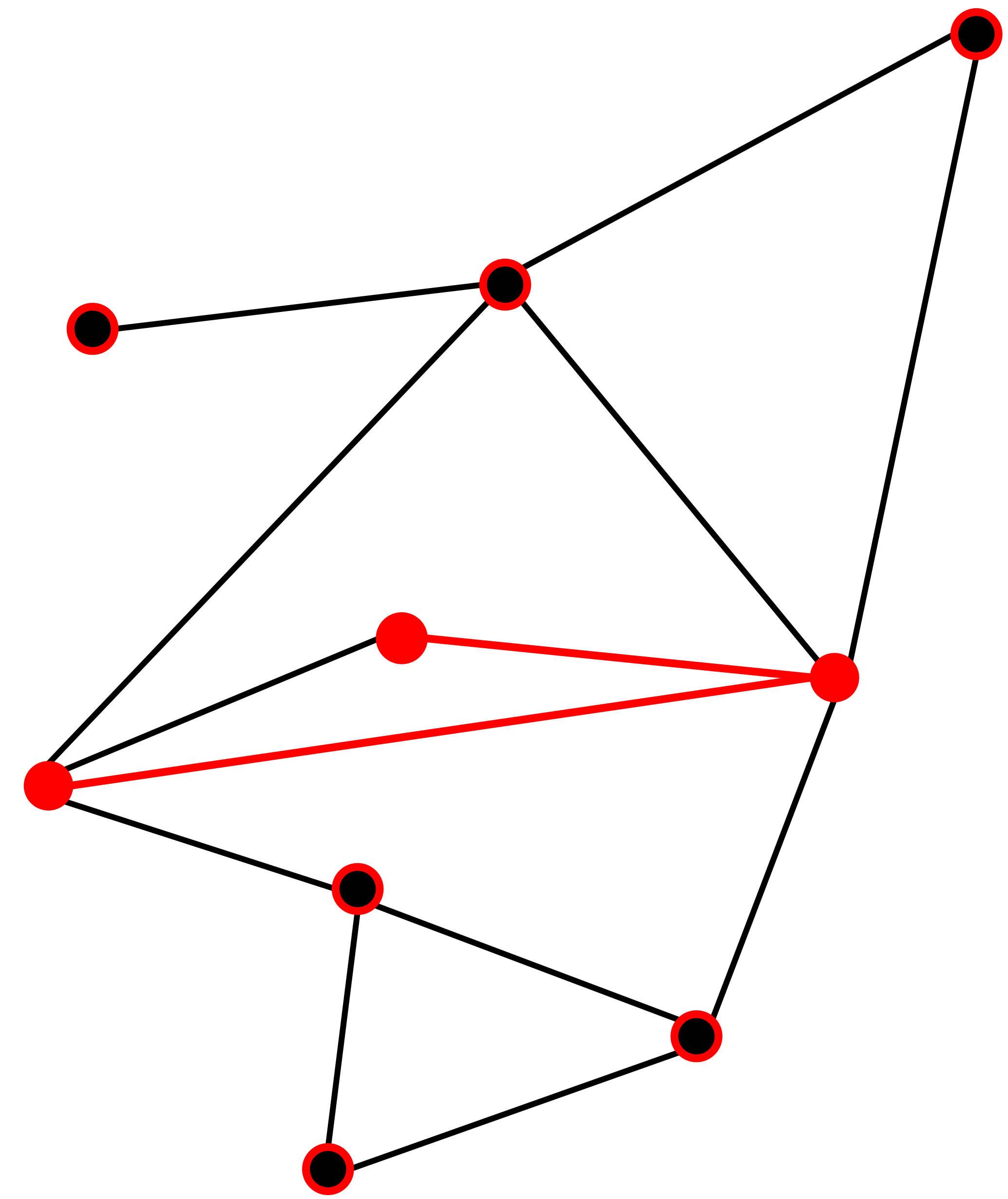
Prim's algorithm for finding a spanning tree



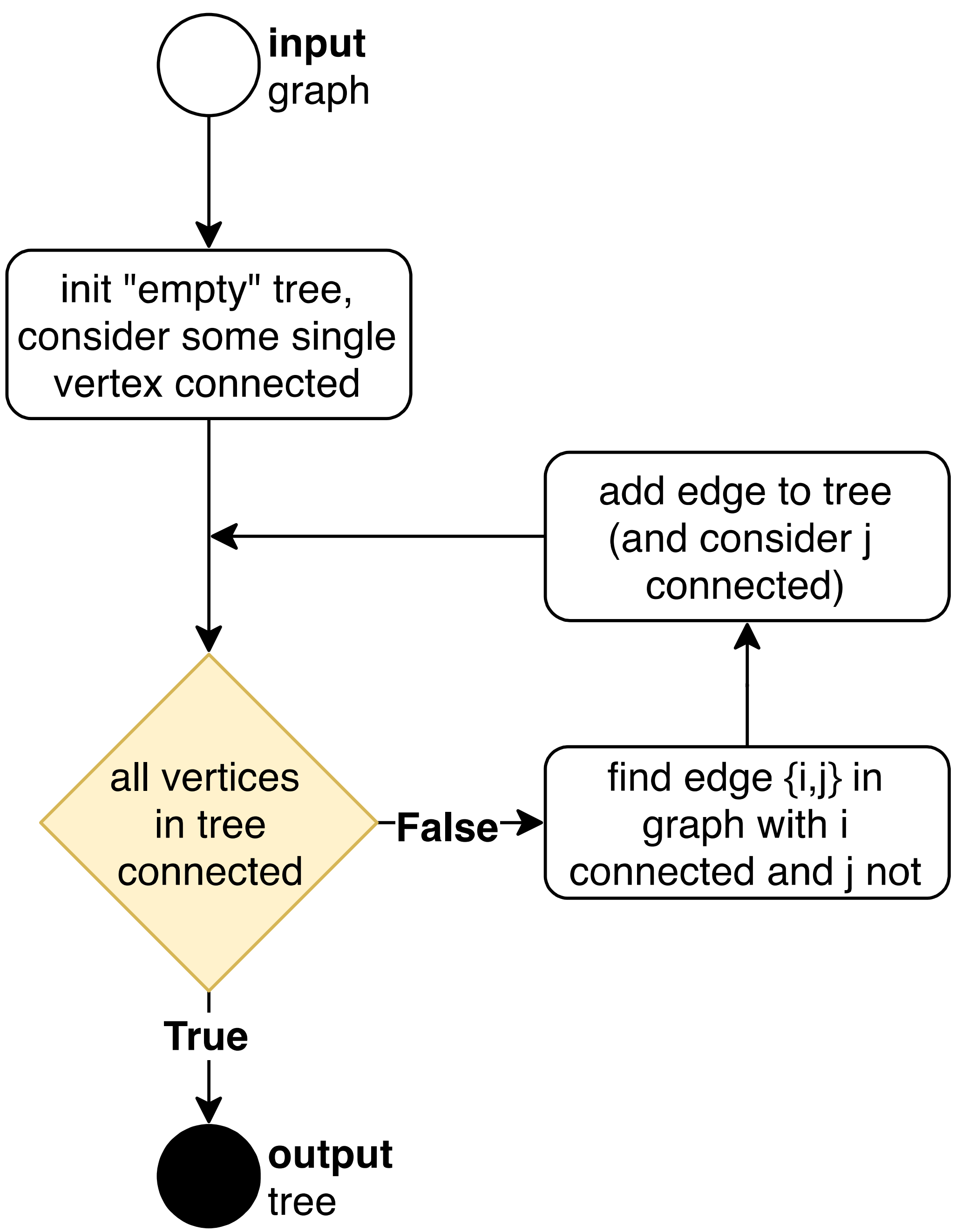


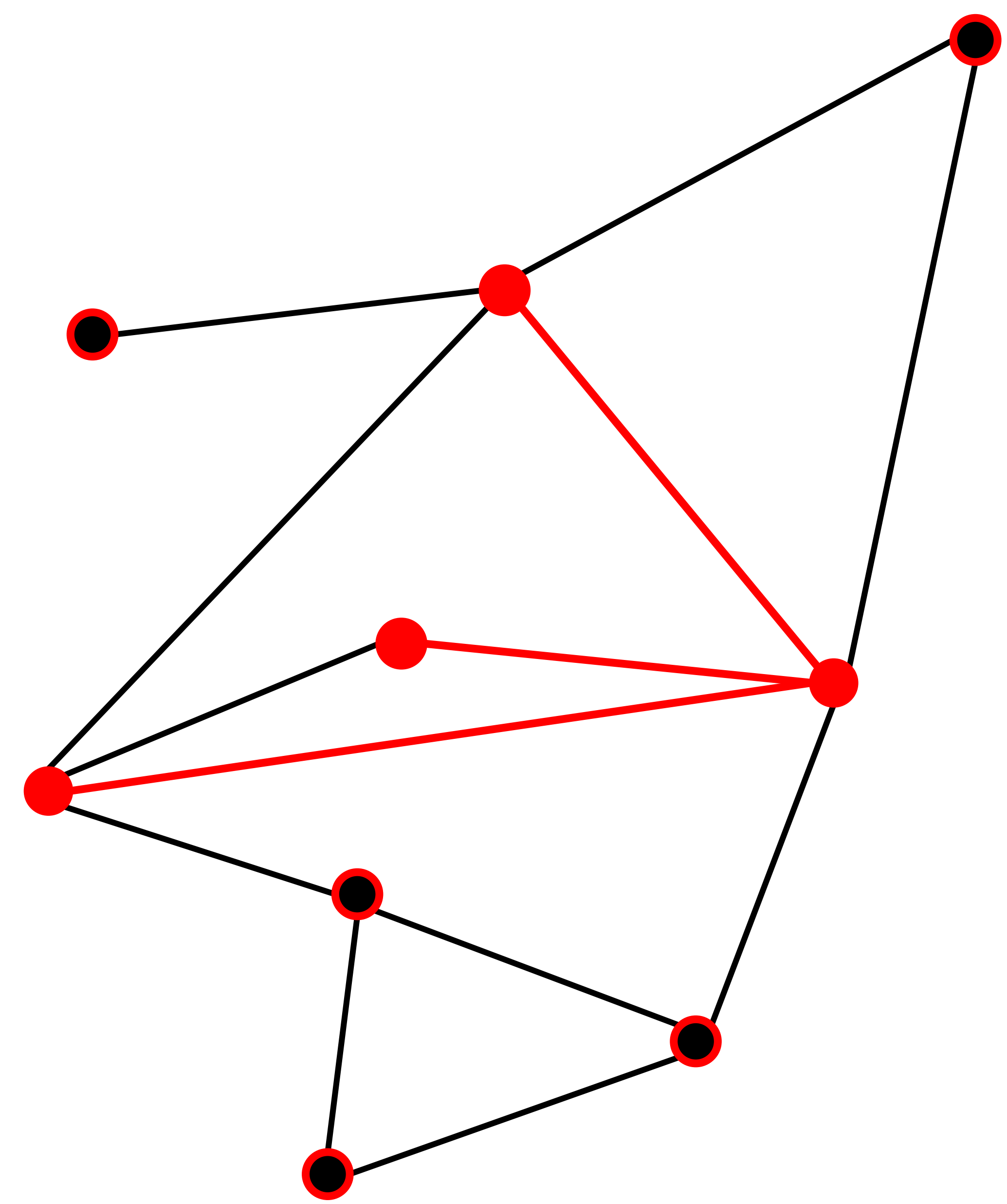
Prim’s algorithm for finding a spanning tree



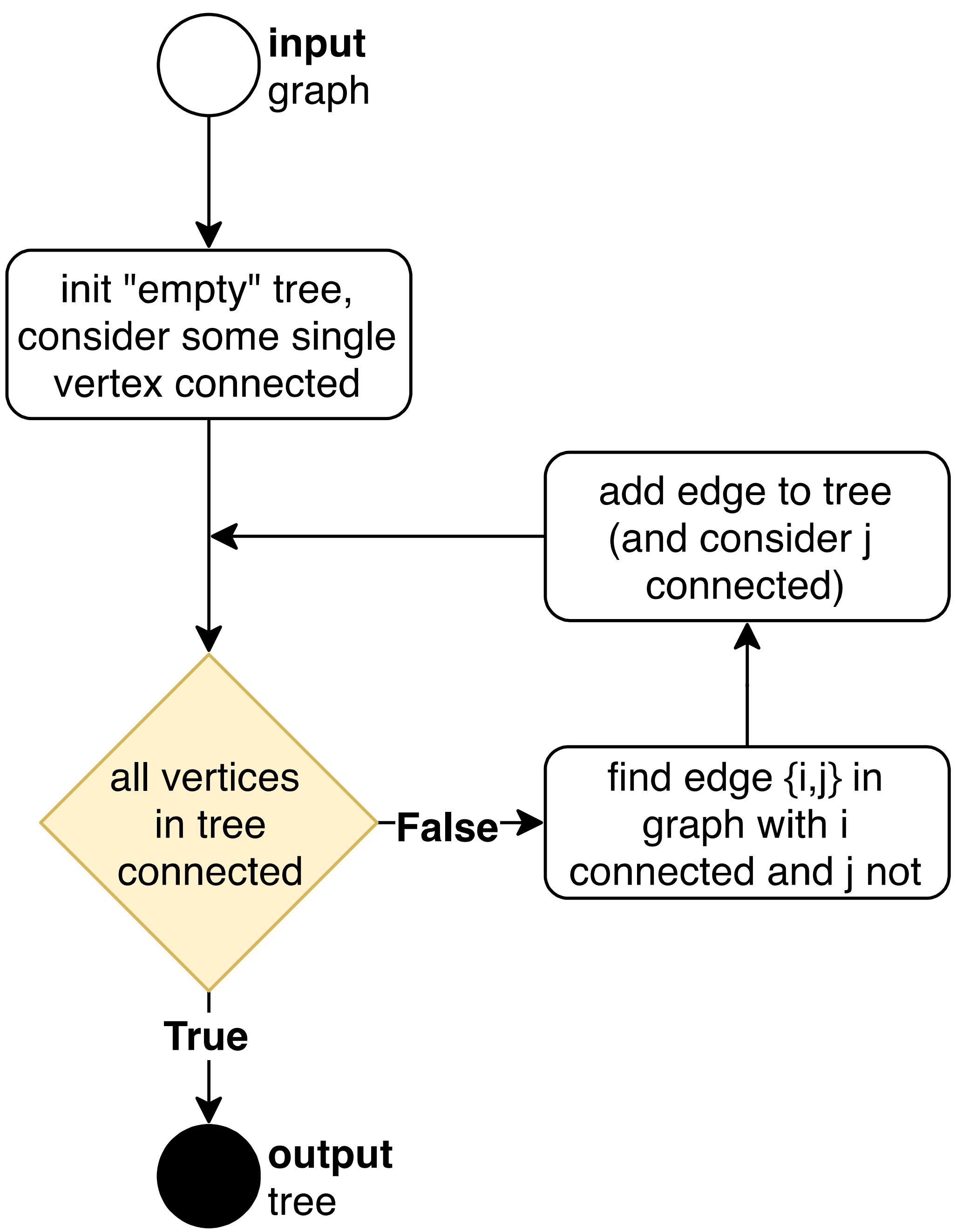


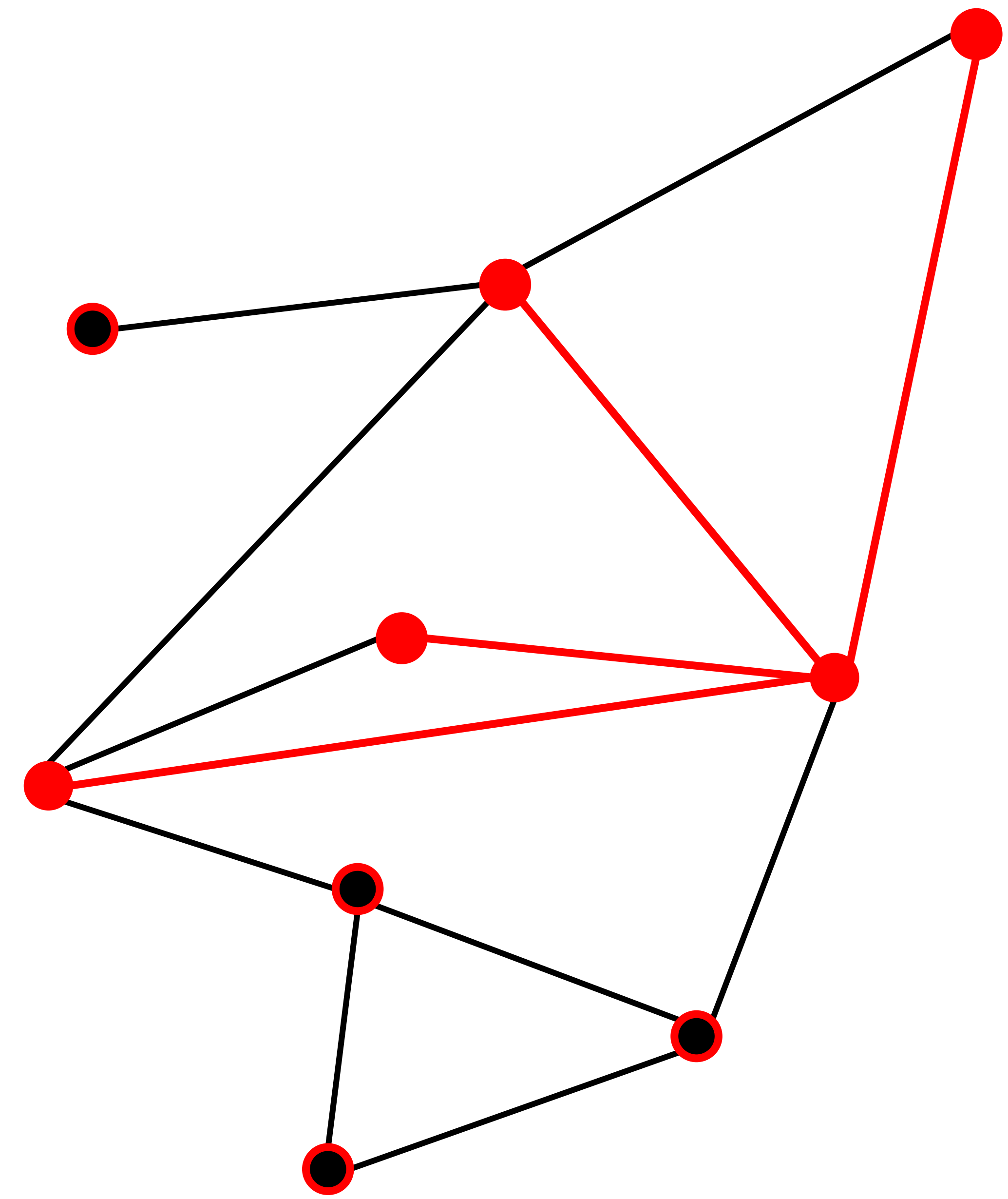
Prim's algorithm for finding a spanning tree



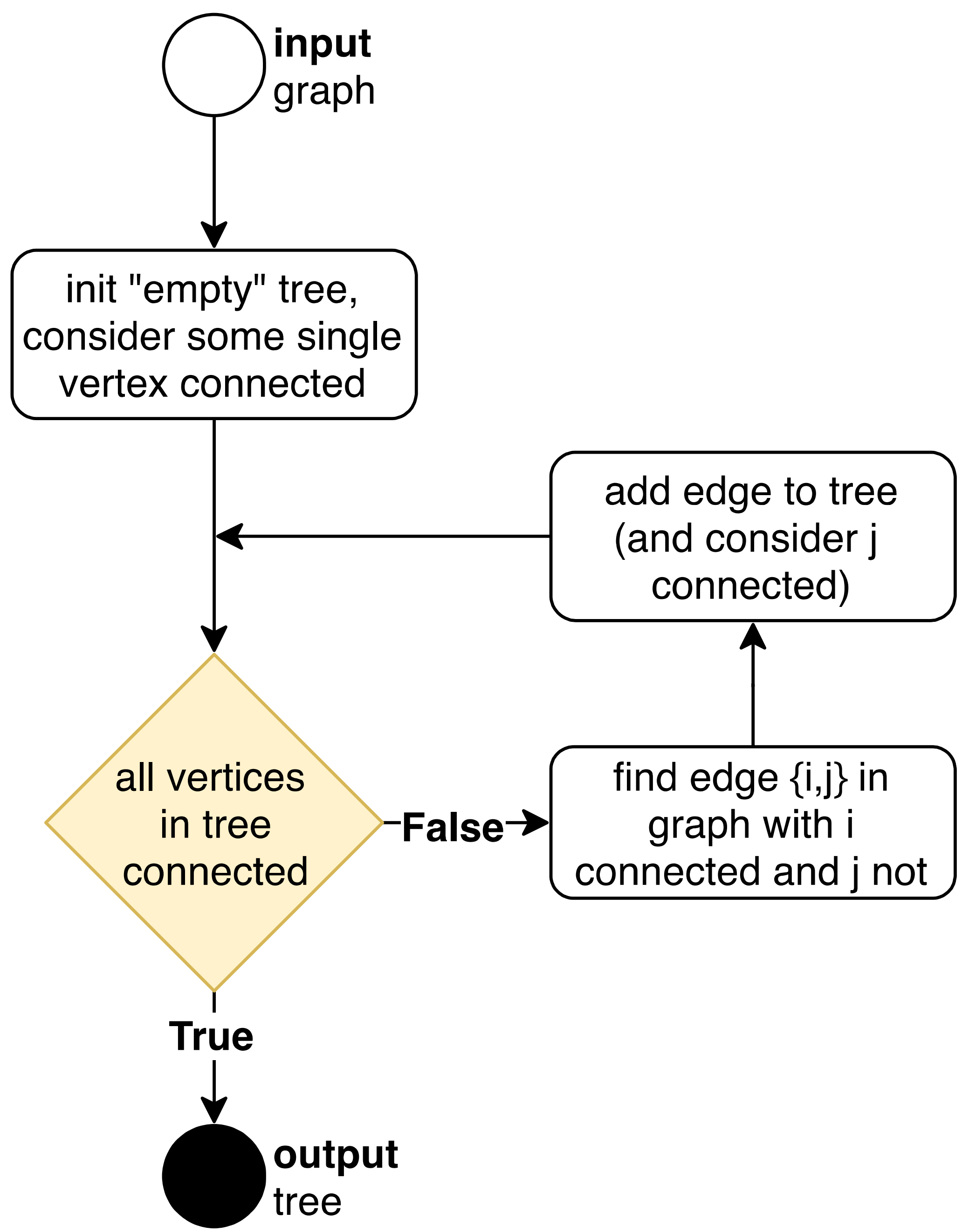


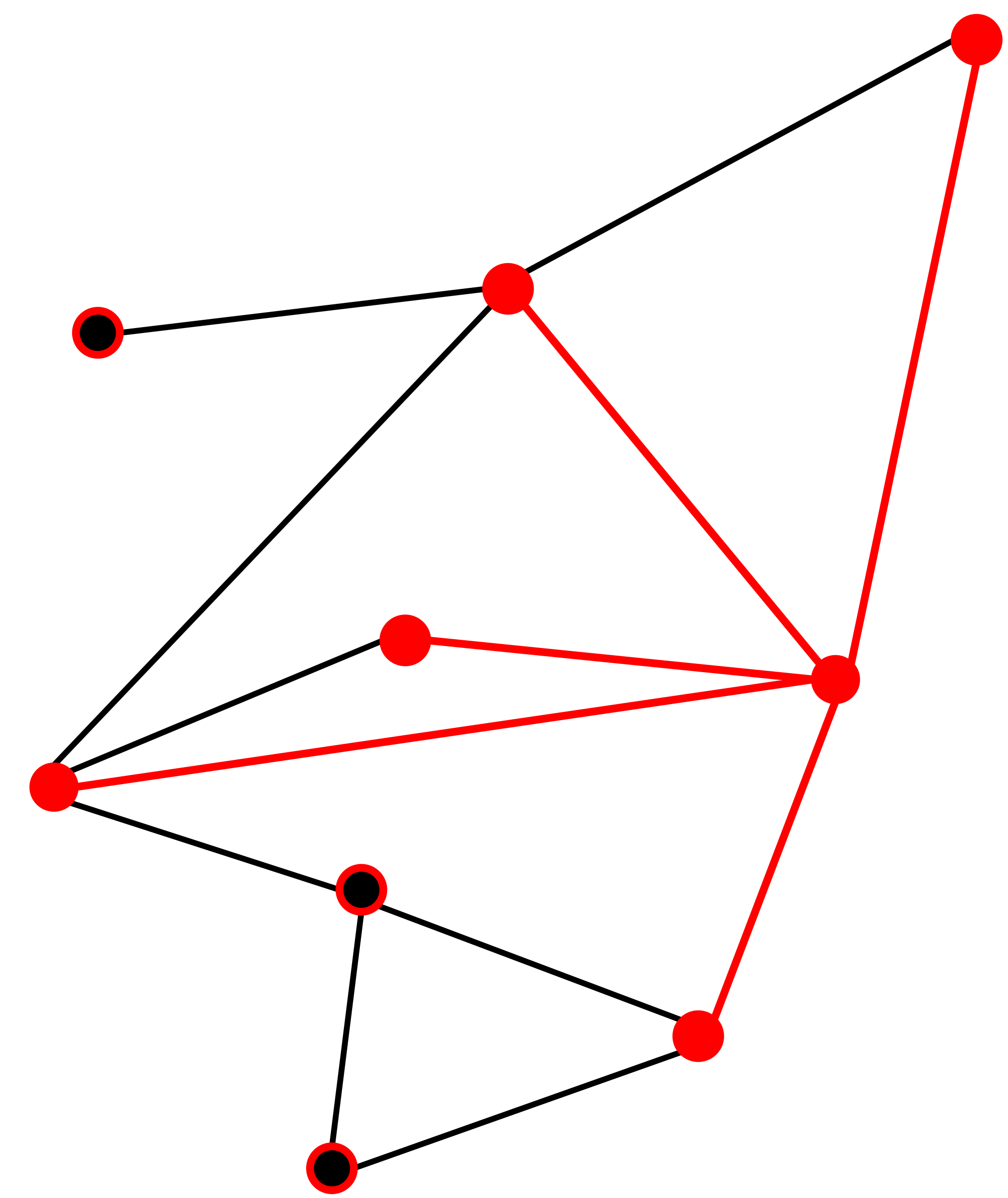
Prim's algorithm for finding a spanning tree



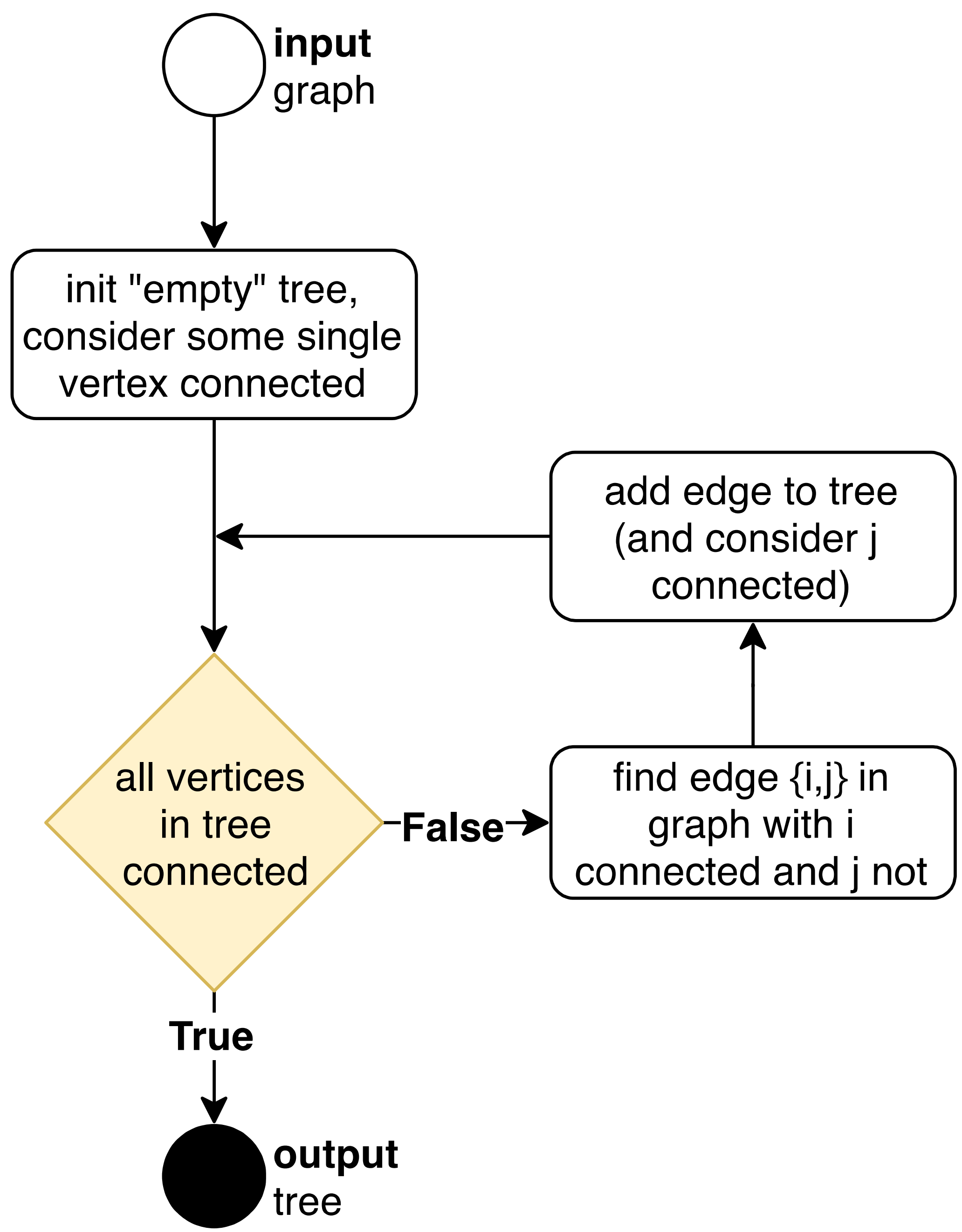


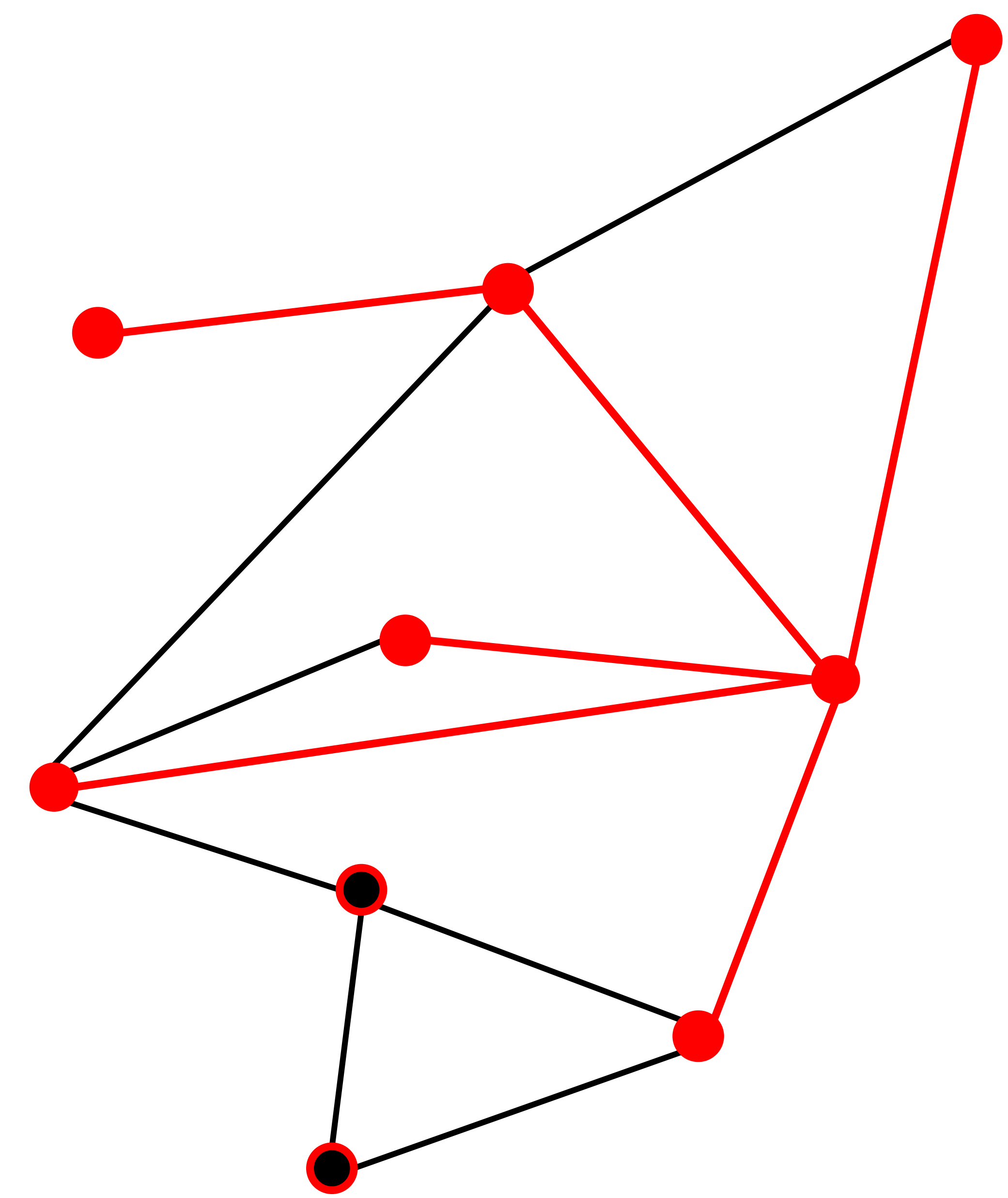
Prim's algorithm for finding a spanning tree



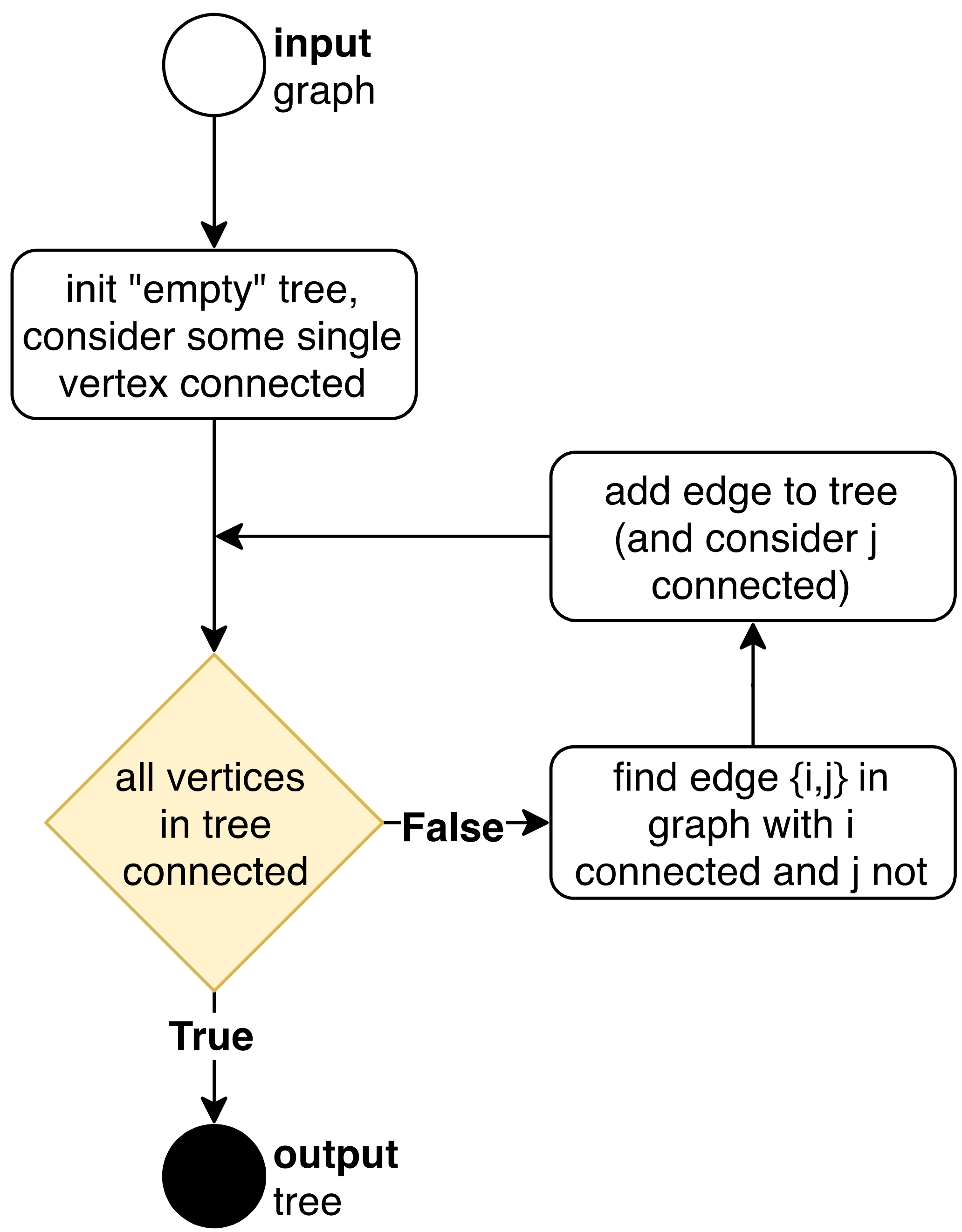


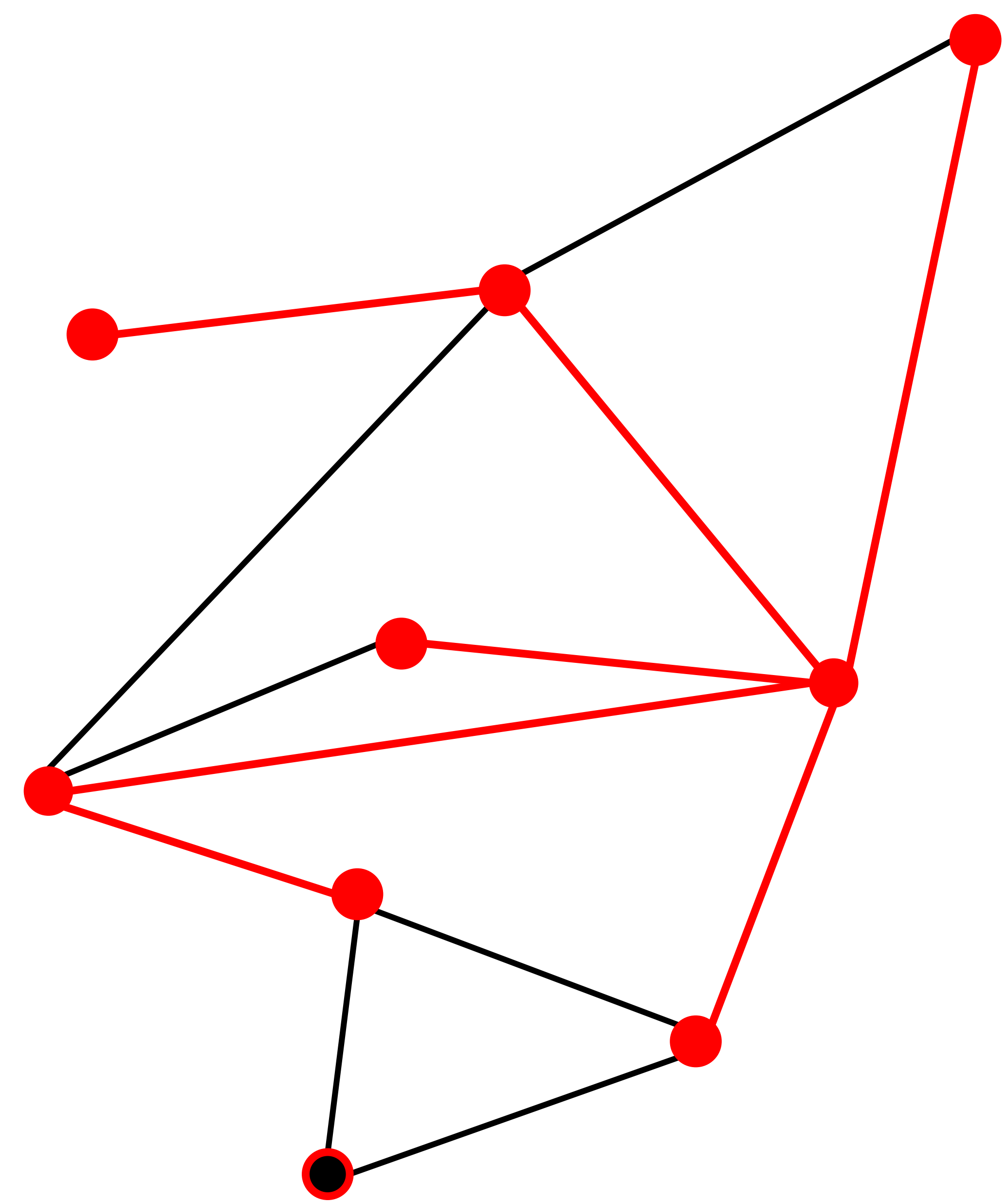
Prim’s algorithm for finding a spanning tree



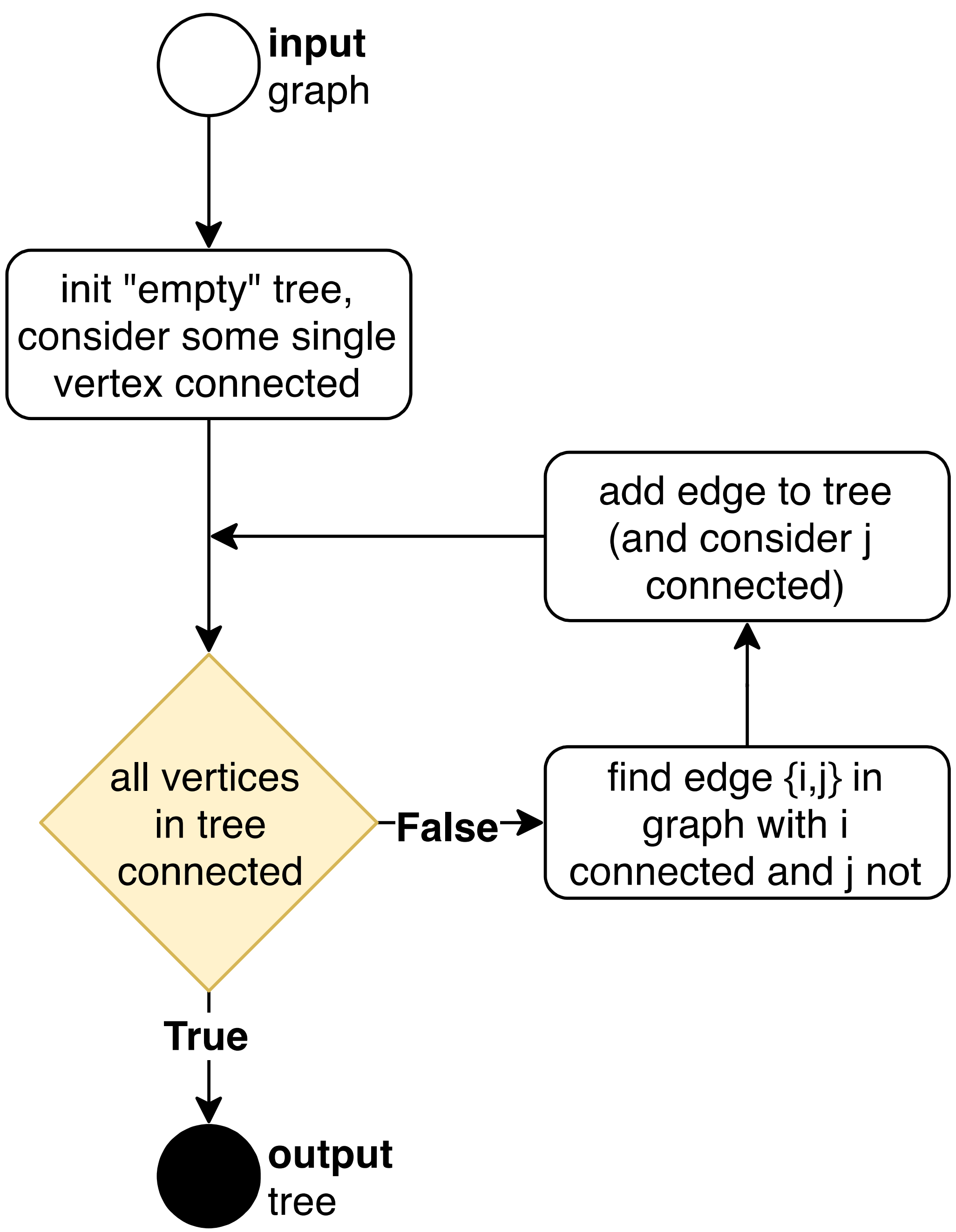


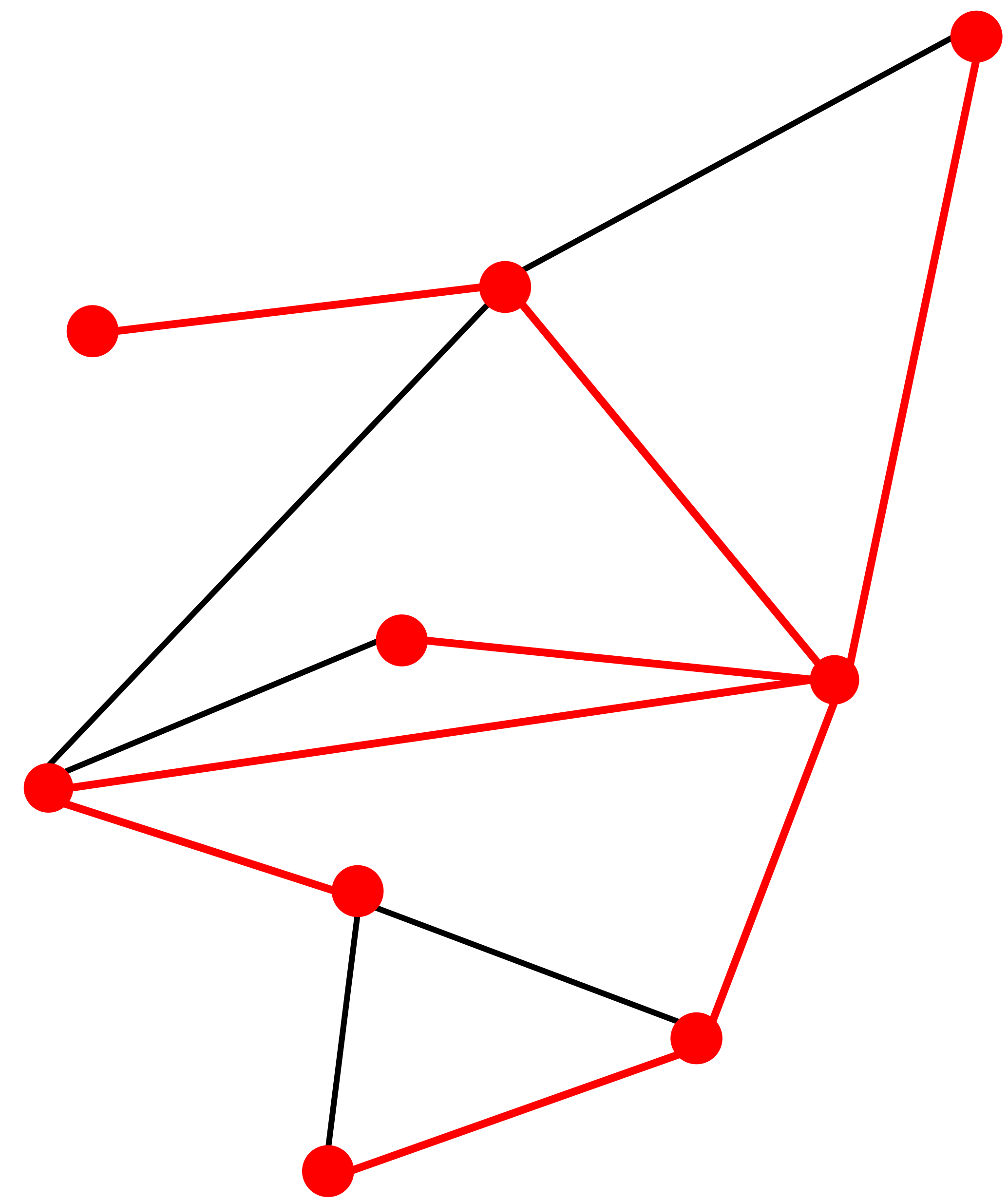
Prim’s algorithm for finding a spanning tree



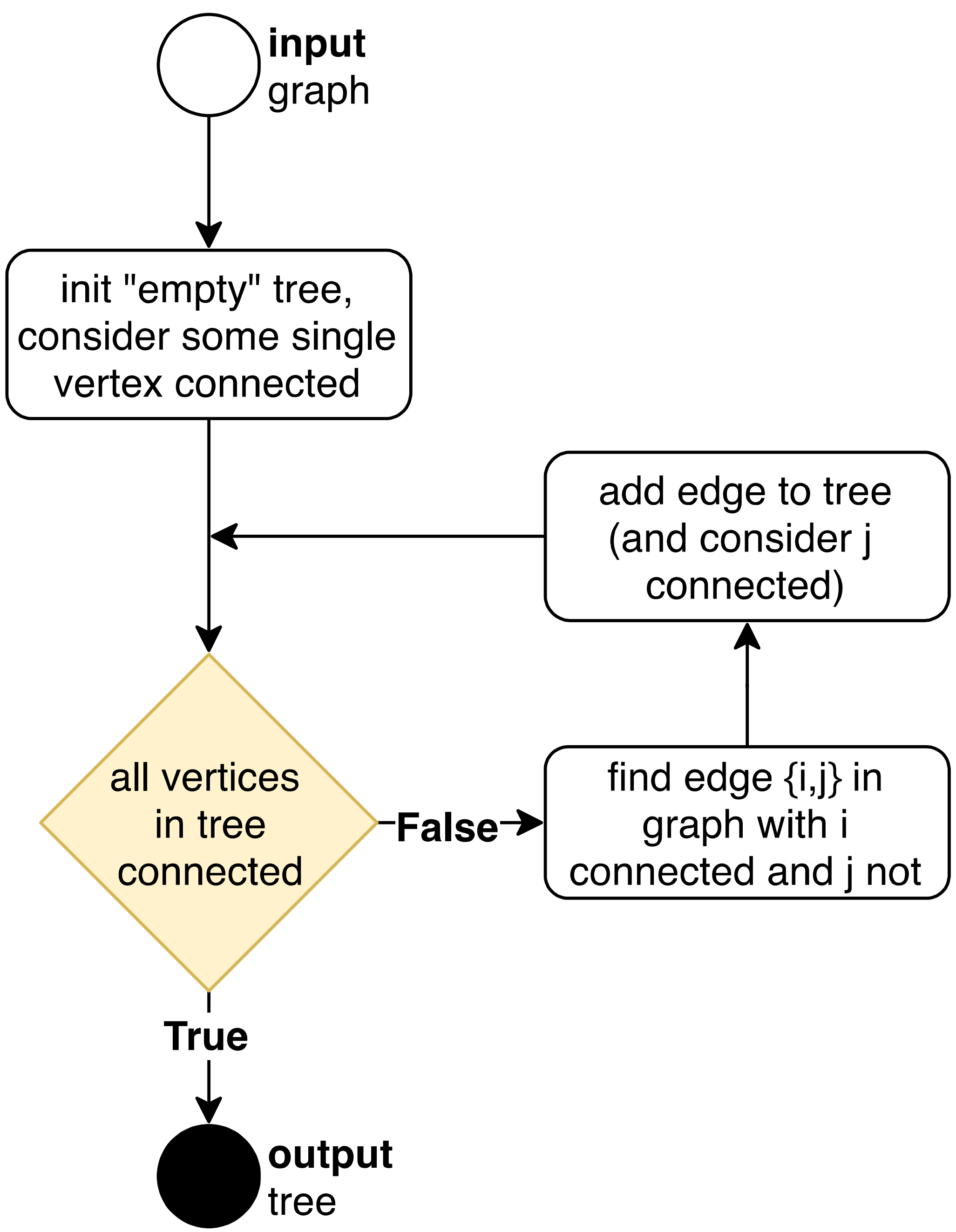


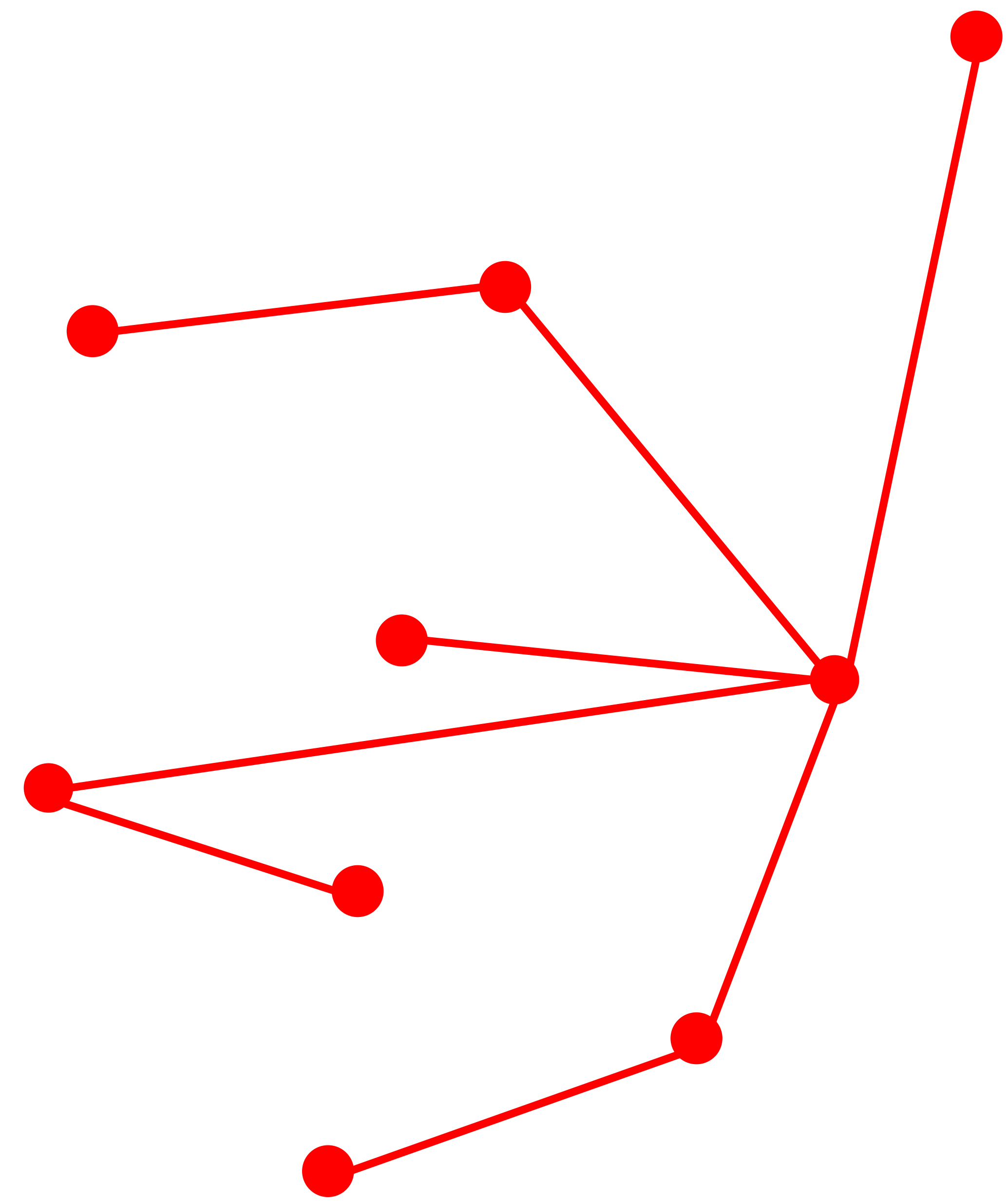
Prim's algorithm for finding a spanning tree





Prim's algorithm for finding a spanning tree





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)

    return tree
```

initialise result accumulation variable

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

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
```

Where am I?

1. Graphs
2. Trees and Spanning Trees
3. Prims algorithm (simplified)
4. *Problem decomposition (if time left)*

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)

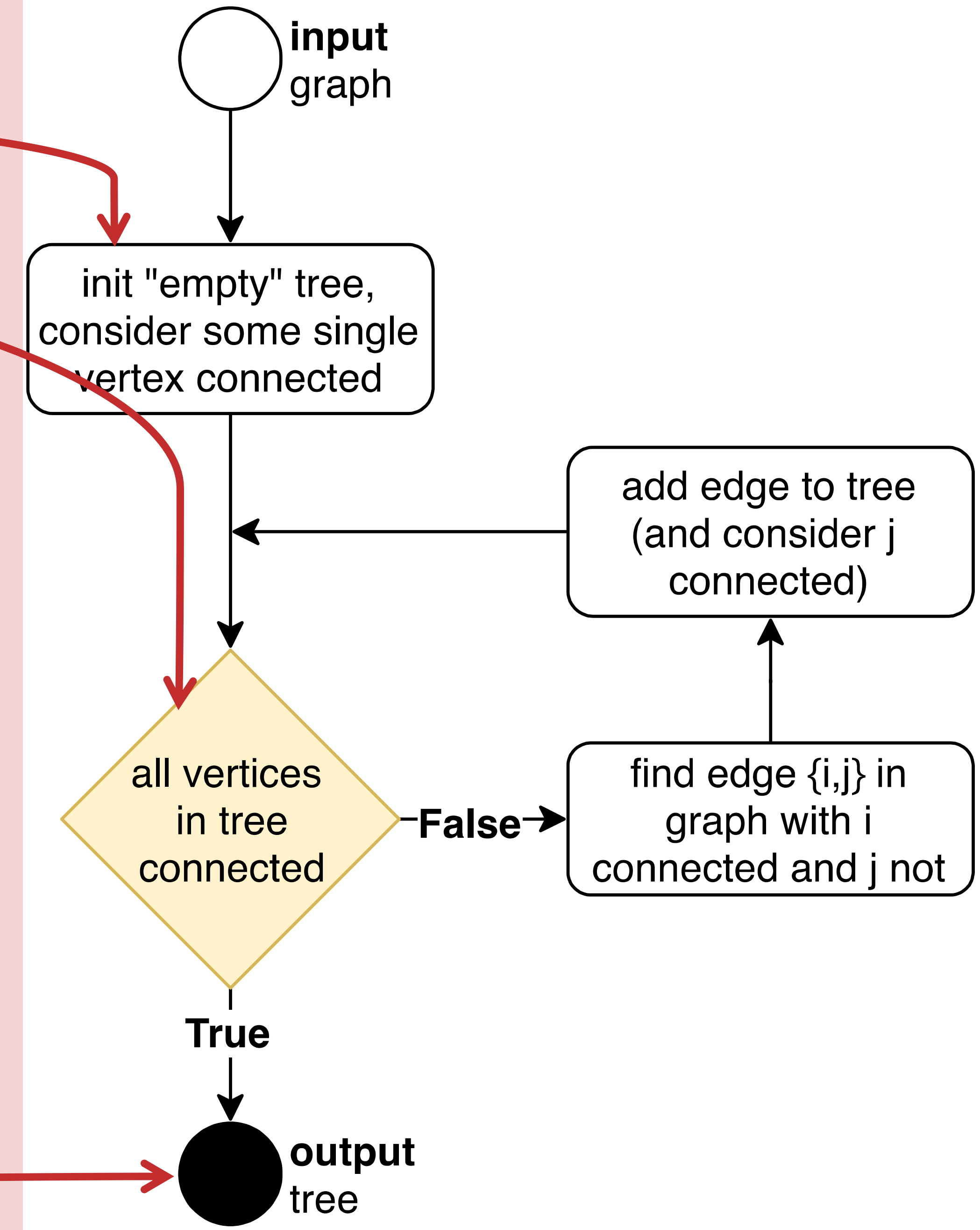
Our main tool for decomposition are functions!

Prim’s algorithm in Python – decomposition edition

```
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?

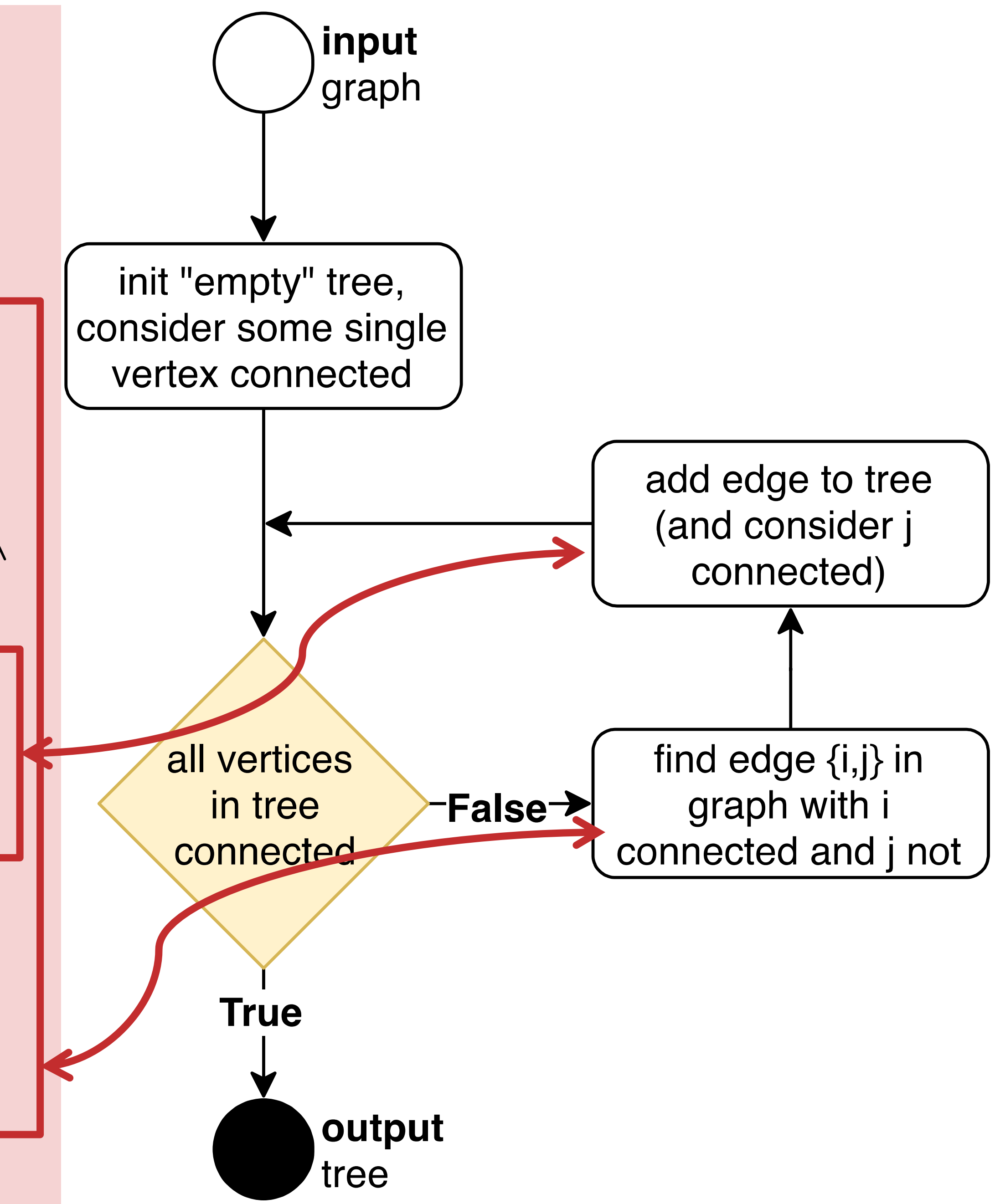
```
n = len(graph)
tree = empty_graph(n)
conn = {0}
while len(conn) < n:
    found = False
    for i in conn:
        for j in range(n):
            if j not in conn and \
            graph[i][j]==1:
                tree[i][j] = 1
                tree[j][i] = 1
                conn = conn.add(j)
   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

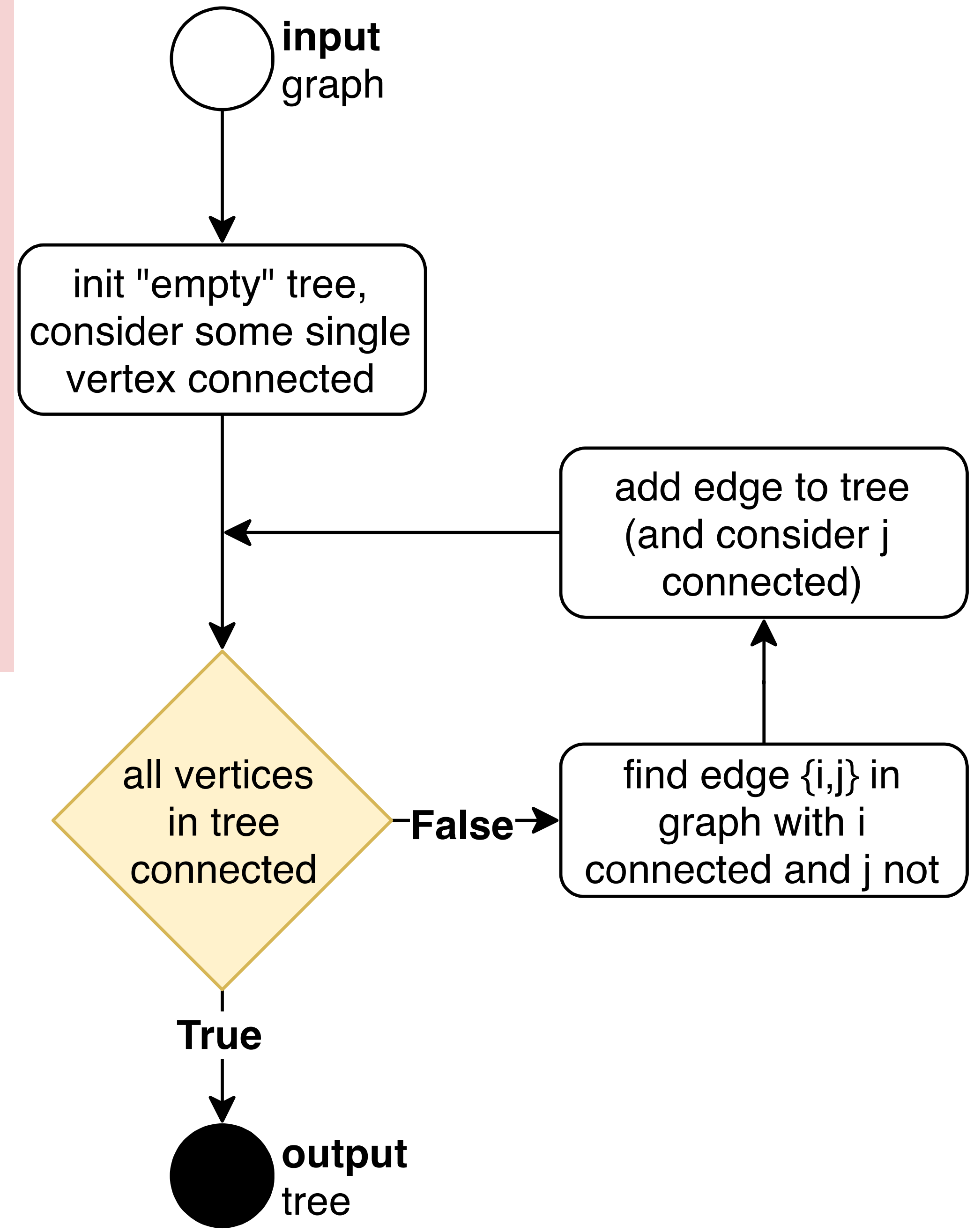
```
n = len(graph)
tree = empty_graph(n)
conn = {0}
while len(conn) < n:
    found = False
    for i in conn:
        for j in range(n):
            if j not in conn and \
            graph[i][j]==1:
                tree[i][j] = 1
                tree[j][i] = 1
                conn = conn.add(j)
   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



Identification of extension edge is not separated from its addition

Decomposition: factor out extension edge identification

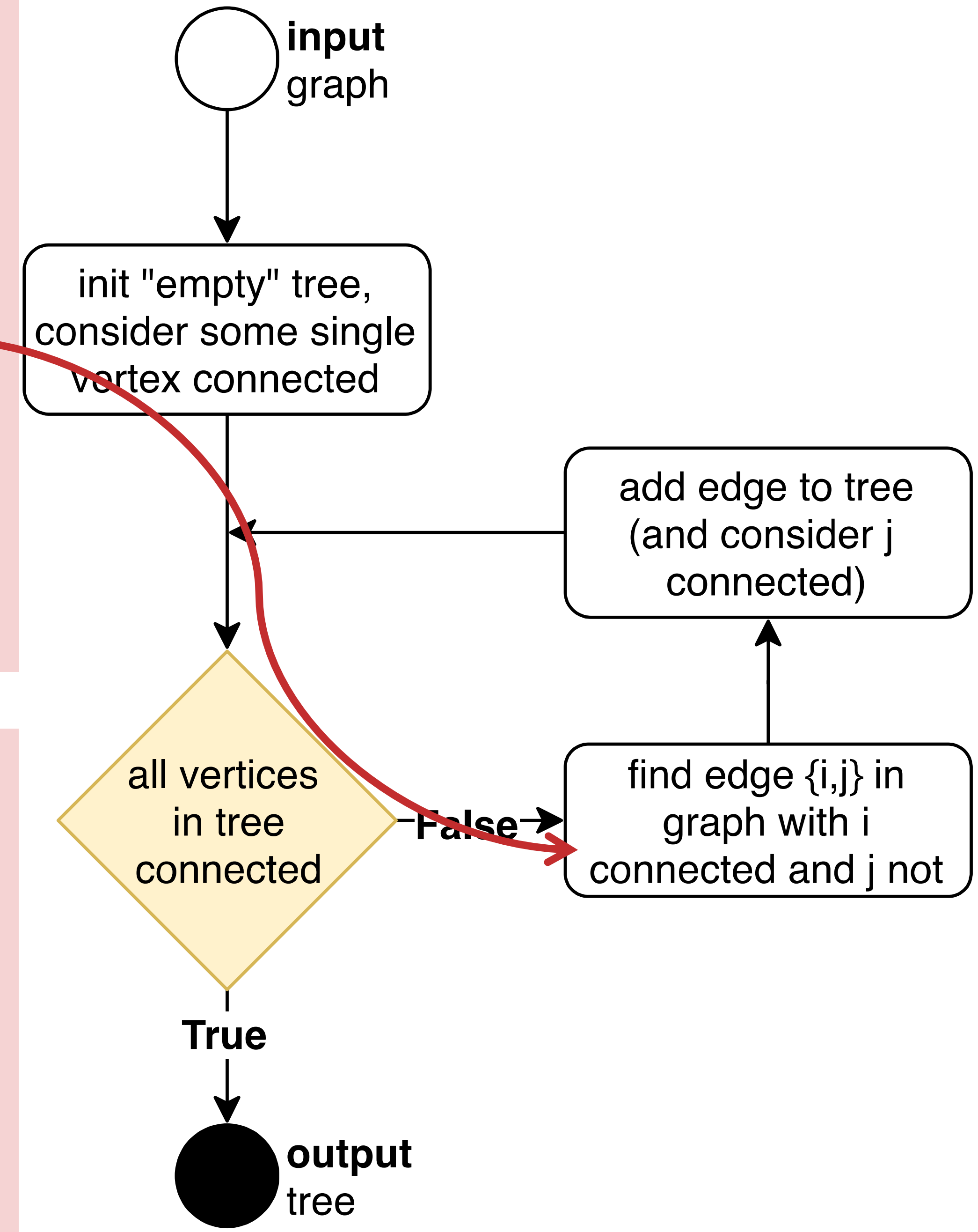
```
n = len(graph)
tree = empty_graph(n)
conn = {0}
while len(conn) < n:




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
```

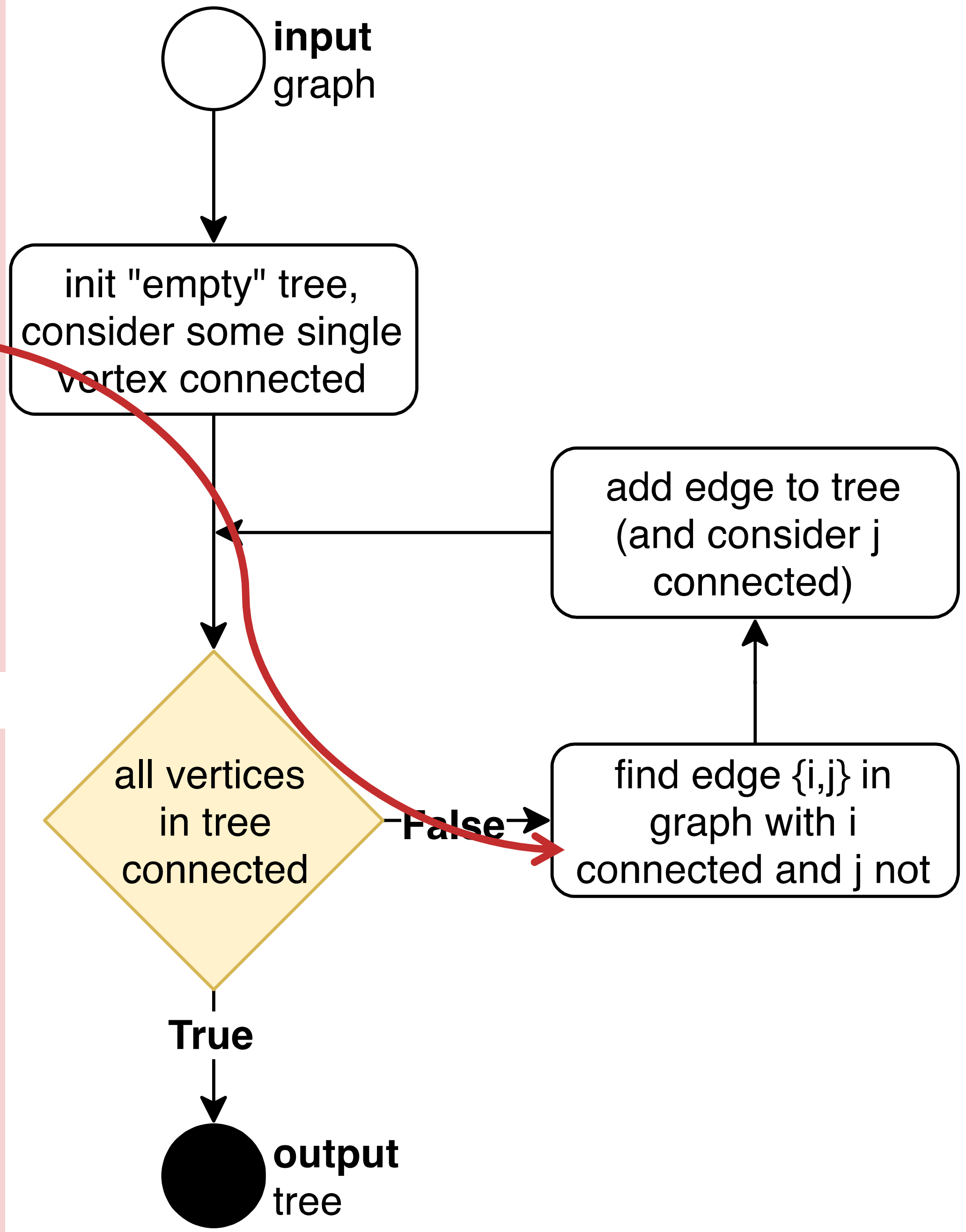
```
def extension(c, g):
    """I: connect. vertices, graph
       O: extension edge (i, j)"""
```



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
```

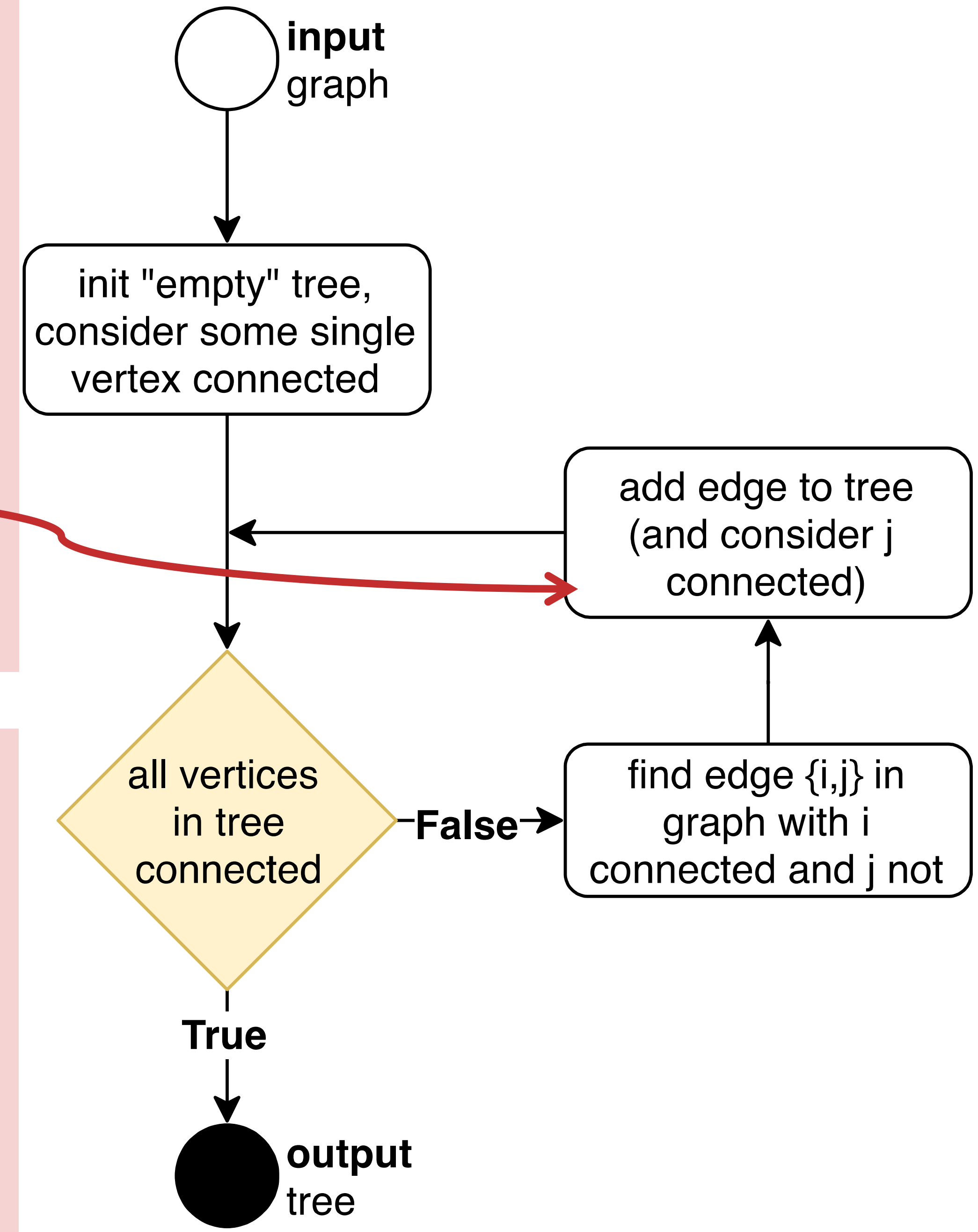
```
def extension(c, g):
    """I: connect. vertices, graph
       O: extension edge (i, j)"""
    n = len(g)
    for i in c:
        for j in range(n):
            if j not in c \
            and g[i][j]:
                return i, j
```



Choose a sub-problem and solve it

```python
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
```

```python
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

Before Next Lecture

Re-program examples from the lecture

Coming Up

- Invariants