Discrete Mathematics

Topic 06 — Graphs and Networks

Lecture 02 — A Survey of Graph Algoroithms

Dr Kieran Murphy

Department of Computing and Mathematics,
SETU (Waterford).
(kieran.murphy@setu.ie)

Autumn Semester, 2022

Outline

- Representing graphs using matrices
- Constructing walks, and tours in graphs
- Trees and Minimum Spanning Trees (MST)
- Colouring graphs

Outline

Handshaking Lemma

Lemma 1

In any graph the sum of all the vertex-degrees is twice the number of edges, i.e.,

$$\sum_{v \in V(G)} \deg(v) = 2|E(G)|$$

A few comments regarding notation:

- $E(G)$ represents the set of edges of G.
- $|E(G)|$ is the cardinality (number of elements of) $E(G)$, i.e., number of edges of G.
- $\sum_{v \in V(G)}$ means "add over each v in the set $V(G)$", i.e., add over each vertex in the vertex set of G.
- Left side equals sum of degrees of all vertices, right side equals twice the number of edges.

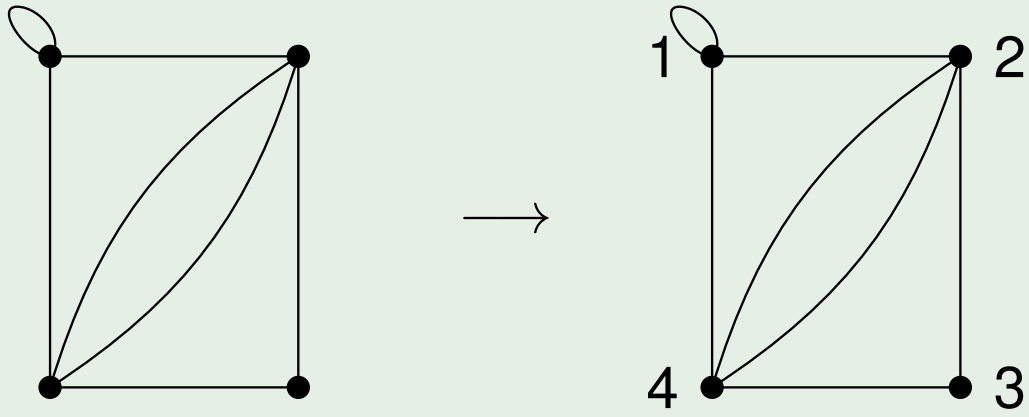
Representation of Graphs using Adjacency Matrices I

Matrix* representation of graph enables efficient representation of small, dense (have lots of edges) graphs.

Definition 2 (Adjacency Matrix)

If graph, G, has vertices labelled $\{1, 2, \ldots, n\}$ and has m edges. Then, the adjacency matrix, A, is the $n \times n$ matrix whose ij-th entry is the number of edges joining vertex i and vertex j. (Note self-loops contribute 2.)

Example 3

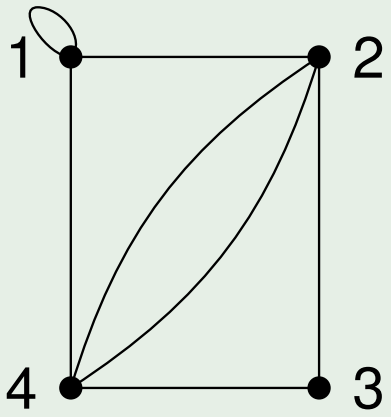


$$A = \begin{array}{c} \\ \text{from node} \\ \end{array} \begin{array}{c} \begin{array}{cccc} \text{to node} & & & \\ 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 2 & 1 & 0 & 1 \\ 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{l} 1 \\ 2 \\ 3 \\ 4 \end{array}$$

*A matrix is a rectangular table of numerical values, where the ij entry corresponds to the value in row i and column j – think Roman Catholic.

Representation of Graphs using Adjacency Matrices II

Example 4



$$A = \begin{array}{c} \\ \text{from node} \end{array} \begin{array}{c} \begin{array}{cccc} \text{to node} & & & \\ 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 2 & 1 & 0 & 1 \\ 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}$$

- The sum of the elements along row i of the adjacency matrix of a graph is the degree of vertex i.
- The adjacency matrix is symmetric (bottom left corner is same as top right corner).

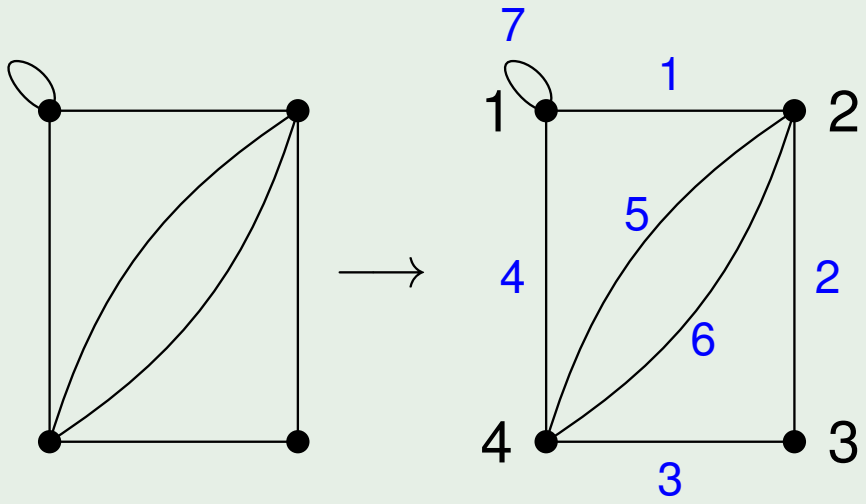
Representation of Graphs using Incidence Matrices I

An alternative representation is based on labelling the edges . . .

Definition 5 (Incidence Matrix)

If graph, G, has vertices labelled $\{1, 2, \ldots, n\}$ and has m edges. Then the incidence matrix, M, is the $n \times m$ matrix whose ij-th entry is 1 if vertex i is incident to edge j, and 0 otherwise.

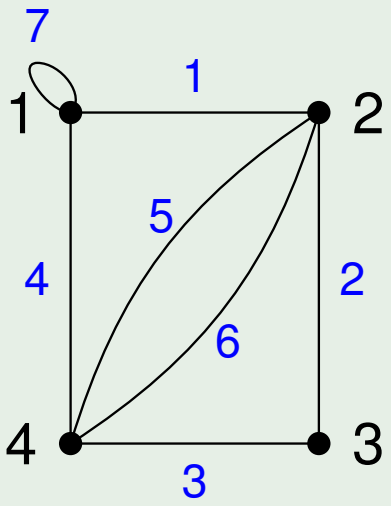
Example 6



$$M = \begin{array}{c} \\ \text{node } 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccccc} \multicolumn{7}{c}{\text{edge label}} \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 2 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Representation of Graphs using Incidence Matrices II

Example 7



$$M = \begin{array}{c} \\ \text{node} \\ \end{array} \begin{array}{c} \text{edge label} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 2 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \\ \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \end{array} \begin{array}{c} \\ \\ 1 \\ 2 \\ 3 \\ 4 \end{array}$$

- The sum down every column is 2 — an edge has two end points.
- Self-loops appear as a 2.
- Parallel edges result in duplicate columns (see column 5 and 6).

Review Exercises 1 (Representing Graphs using.Matrices)

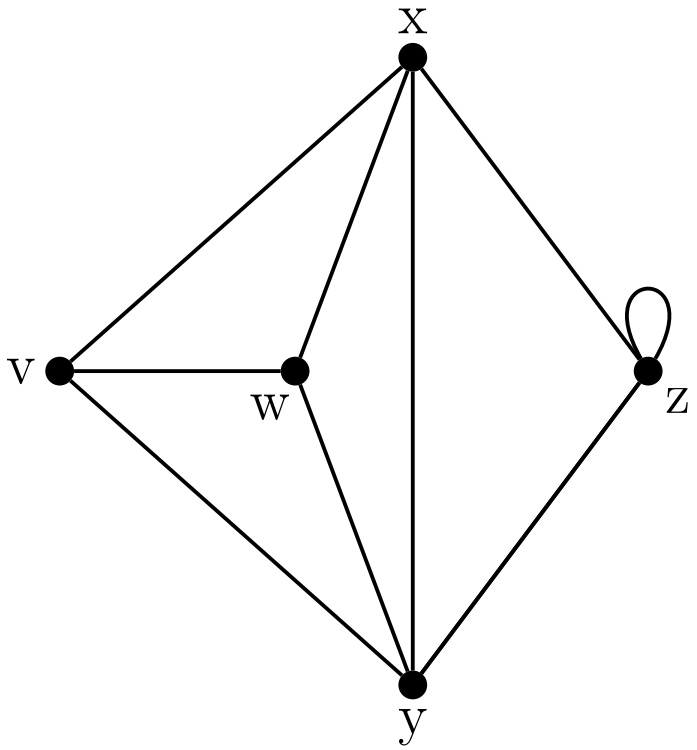
Outline

Walks

Definition 8 (Walk)

Given a graph, G, a walk in G is a finite sequence of edges ofthe form, v_0v_1, v_1v_2, $\ldots$, $v_{m-1}v_m$, in which any two consecutive edges are adjacent or identical. This walk is also denoted by $v_0 \to v_1 \to \cdots \to v_m$.

- A walk determines a sequence of vertices from the initial vertex, v_0, to the final vertex, v_m.
- The number of edges in a walk is called its length.
- For example, in the graph on the right, $v \to w \to x \to y \to z \to z \to y \to w$ is a walk of length 7 from v to w.



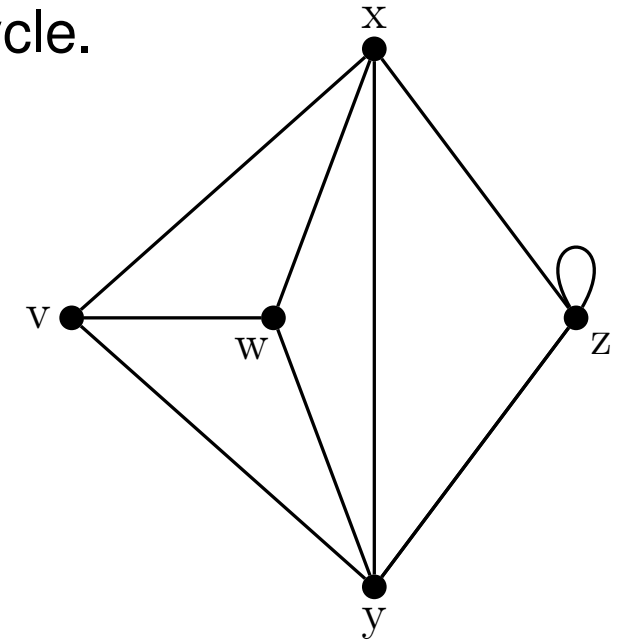
Skip

Trails, Paths and Cycles

Definition 9 (Trail, Path, Cycle)

A walk, $v_0 \to v_1 \to \cdots \to v_m$, in which all the edges are distinct is a trail.
If in addition, the vertices $v_0, v_1, \ldots, v_m$ are distinct (except, possibly $v_0 = v_m$) then the trial is a path.
A path or trail is closed if $v_0 = v_m$ and a closed path containing at least one edge is a cycle.

- Note that a loop or pair of multiple edges is a cycle.
- A cycle of length 3 is called a triangle.
- For example, in the graph on the right,
  - $v \to w \to x \to y \to z \to z \to x$ is a trail
  - $v \to w \to x \to y \to z$ is a path
  - $v \to w \to x \to y \to z \to x \to v$ is a closed trail
  - $v \to w \to x \to y \to v$ is a cycle



Skip

Review Exercises 2 (Paths and Walks)

Question 1:
In a Peterson graph, find

(a) a trail of length 5;

(b) a path of length 9;

(c) cycles of length 5, 6, 8, and 9;

(d) cutsets with 3, 4, and 5 edges.

Question 2:
The **girth** of a graph is the length of its shortest cycle. Write down the girth of each of the following graphs.

(a) K_9 **(b)** $K_{5,7}$ **(c)** C_8 **(d)** W_8

Outline

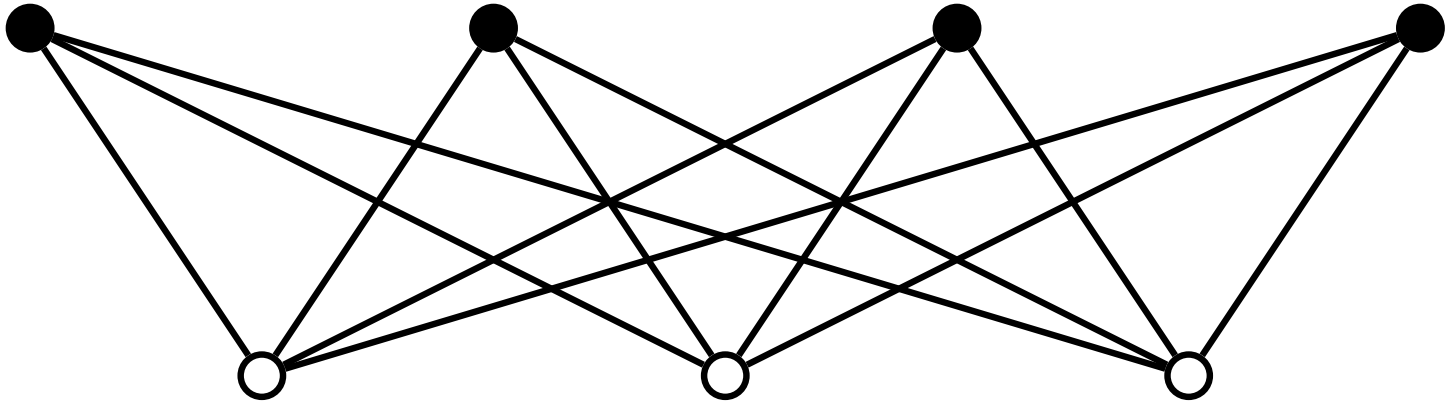
Cycles in Bipartite Graphs

While in arbitrary graphs of n vertices cycles can exist of length 2 to n, for bipartite graphs a cycle length must be even.

Theorem 10

A graph G is a bipartite graph iff each cycle of G is of even length.

- You don't have to prove this result but think why it must be true.



Bounds on Number of Edges

In a connected simple graph, the number of edges increases with the number of cycles. In particular, in a connected graph of n vertices with the minimum number of edges, $n-1$, there are no cycles, and as the numbers of edges increase up to the maximum of $n(n-1)/2$ the number of cycles is greater than $n!$.

Theorem 11

Let G be a simple graph on n vertices. If G has k components, then the number, m, of edges of G satisfies

$$n-k \leq m \leq (n-k)(n-k+1)/2$$

- Recall that in a general graph, i.e., with parallel edges, there is no upper bound on the number of edges.

Corollary 12

Any simple graph with n vertices and more than $(n-2)(n-1)/2$ *edges is connected.*

Proof: Put $k = 2$ in Theorem 11.

Disconnecting Sets

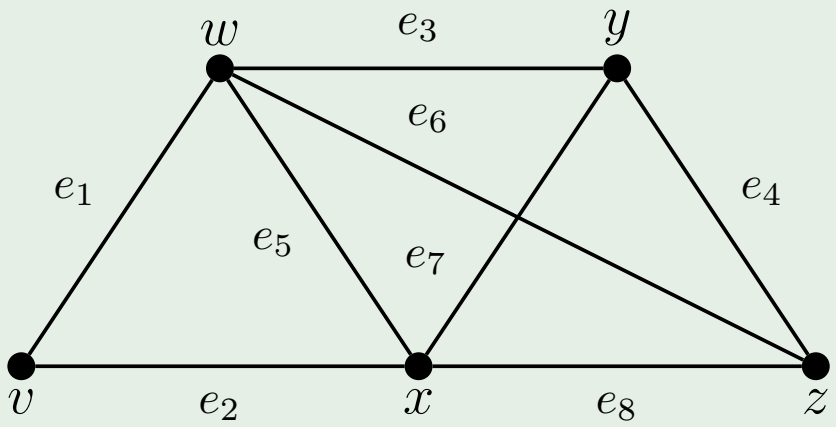
Often, of interest is the number of edges or vertices that need to be removed in order to disconnect a connected graph.

Definition 13 (Disconnecting Set)

A disconnecting set in a connected graph, G, is a set of edges whose removal disconnects G.

Example 14

The sets $\{e_1, e_2, e_5\}$ and $\{e_3, e_6, e_7, e_8\}$ are both disconnecting sets of G.



Cutset

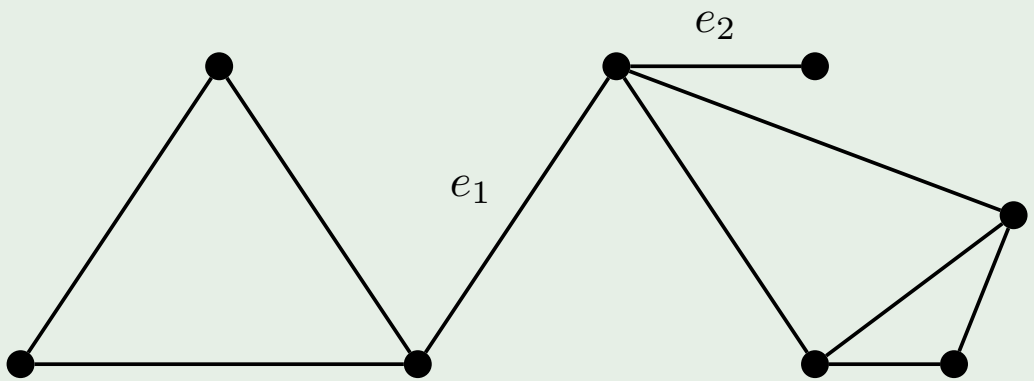
Definition 15 (Cutset, Bridge)

A cutset is a disconnecting set, no proper subset of which is a disconnecting set.
If a cutset has only one edge, e, it is called a bridge.

- The removal of a cutset from a connected graph always results in a graph with exactly two components.

Example 16

The following graph has two bridges, e_1 and e_2.



Edge Connectivity, $\lambda(G)$

The minimum number of edges needed to be removed in order to disconnect a graph is

Definition 17 (Edge Connectivity)

If G is connected, its edge connectivity, $\lambda(G)$, is the size of the smallest cutset in G.

- If G contains a bridge then $\lambda(G) = 1$.
- A graph, G, is said to be k-edge connected if $\lambda(G) \geq k$.

Separating Sets

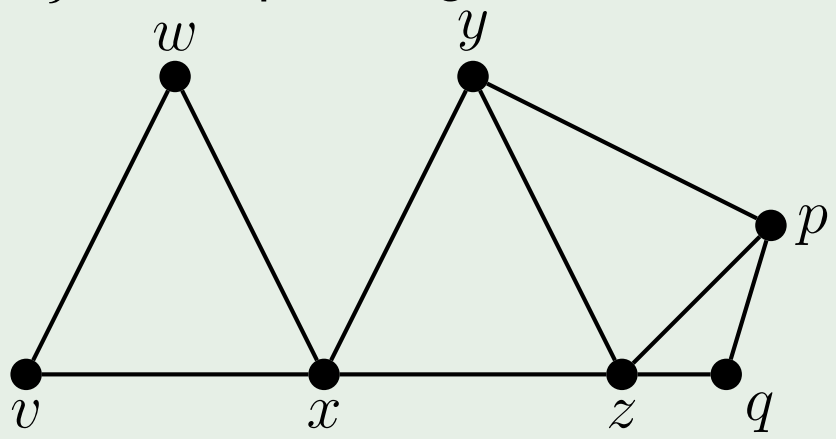
Analogous concepts exist for the removal of vertices rather than edges.

Definition 18 (Separating Set, cut-vertex)

A separating set in a connected graph, G, is a set of vertices whose removal disconnects G.
A separating set consists of only one vertex is a cut-vertex.

Example 19

The set $\{x, w\}$ is a separating set, and x is a cut-vertex of G.



Vertex Connectivity, $\kappa(G)$

The minimum number of vertices needed to be removed in order to disconnect a graph is

Definition 20 (Vertex Connectivity)

If G is connected, its (vertex) connectivity, $\kappa(G)$, is the size of the smallest separating set in G.

- If G contains a cut-vertex then $\kappa(G) = 1$.
- A graph, G, is said to be k-connected if $\kappa(G) \geq k$.
- It can be proved that if G is any connected graph, then

$$\kappa(G) \leq \lambda(G) \leq \delta(G)$$

where $\delta(G)$ is the minimum vertex degree in G.

Review Exercises 3 (How Connected is a Graph?)

Question 1:
Write down $\kappa(G)$ and $\lambda(G)$ for each of the following graphs.

(a) The cycle graph, C_6.

(b) The wheel graph, W_6.

(c) The complete bipartite graph, $K_{4,7}$.

Question 2:
Show that, if G is a connected graph with minimum degree k, then $\lambda(G) \leq k$.

Question 3:
Draw a graph G with minimum degree k for which $\kappa(G) < \lambda(G) < k$.

Outline

Eulerian Graphs

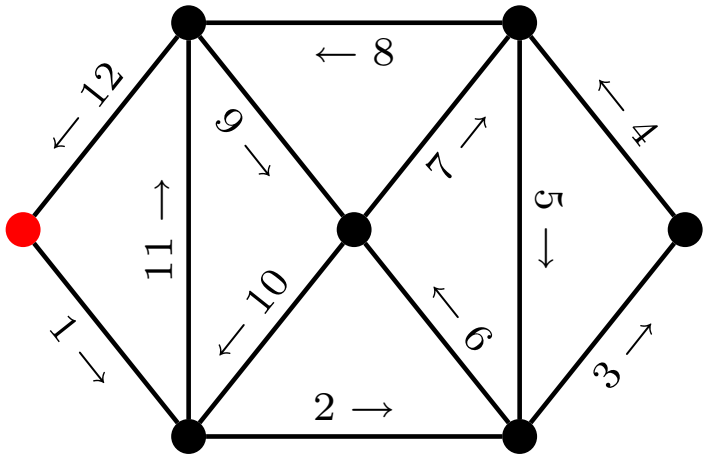
Definition 21 (Eulerian graphs, semi-Eulerian graphs)

A (connected) graph, G, is Eulerian if there exists a closed trail containing every edge of G. Such a trail is an Eulerian trail or Euler tour.

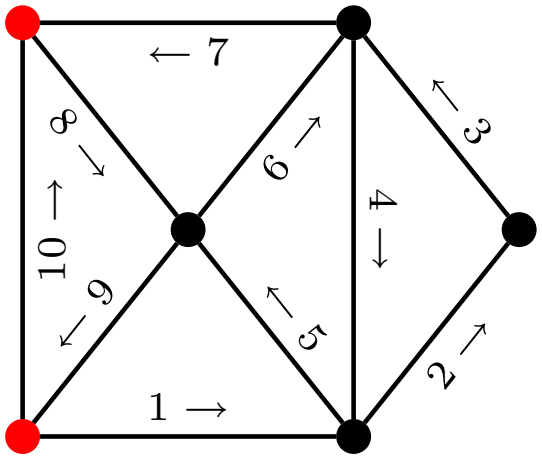
A non-Eulerian graph, G, is semi-Eulerian if there exists a trail containing every edge of G. Such a trail is an semi-Eulerian trail.

- In general, a trail that passes through every vertex is called a tour.
- Unlike a walk, in a trail edges are not repeated. Hence in a Eulerian tour each edge is traversed once and once only.
- Also, unlike a path, in a trail vertices may be repeated. Hence vertices may be repeated in a Eulerian tour.

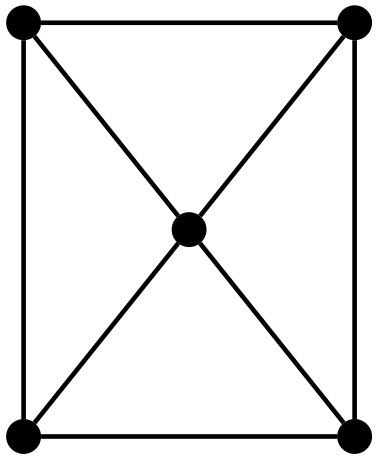
Example



Eulerian



Semi-Eulerian



Non-Eulerian

Given a graph, G, of interest is

- Does an Euler tour exist, i.e., is G Eulerian?
  — Necessary and sufficient conditions are known.
- How to find/construct an Euler tour?
  — Fleury's algorithm (linear time).

Necessary & Sufficient Conditions for Eulerian Graphs

The main result in this section, due to Euler, provides both necessary and sufficient conditions for the existence of an Euler tour.

Theorem 22 (Euler, 1736)

A connected graph, G, is Eulerian if and only if the degree of each vertex of G is even.

Based on Euler's theorem the following result can also be obtained.

Corollary 23

A connected graph is semi-Eulerian if and only if it has exactly two vertices of odd degree.

- Note that in a semi-Eulerian graph, any semi-Eulerian path must have one vertex of odd degree as its initial vertex and the other as its final vertex.
- Also, by the handshaking lemma, a graph cannot have an odd number of vertices of odd degree.

Construction of Euler Tours — Fluery's algorithm

The construction of Euler tours is possible in linear time, via implementation of the following result.

Theorem 24 (Fluery's algorithm)

Let G be a Eulerian graph. Then the following construction is always possible and produces an Eulerian tour of G.
Start at any vertex, u, and traverse the edges in an arbitrary manner, subject only to the following rules:

1. *Erase the edges as they are traversed, and if any isolated vertices result, erase them also.*
2. *At each stage, use a bridge only if there is no alternative.*

Review Exercises 4 (Eulerian Graphs)

Question 1:
Which of the following are Eulerian? semi-Eulerian?

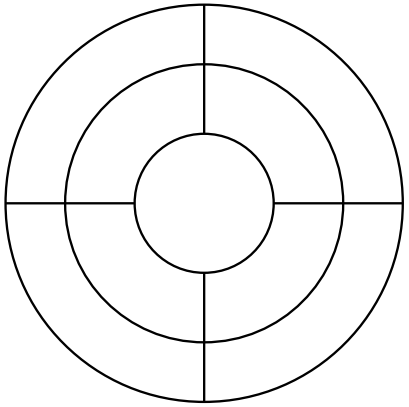
(a) The complete graph, K_5.

(b) The complete bipartite graph, $K_{2,3}$.

(c) The Peterson graph.

Question 2:
Let G be a connected graph with k (> 0) vertices of odd degree.

(a) How many continuous pen-strokes are needed to draw the diagram without repeating any line?



Question 3:
An Eulerian graph is **randomly traceable** from a vertex, v, if whenever we start from v and traverse the graph in an arbitrary way never using any edge twice, we eventually obtain an Eulerian trail.

Outline

Hamiltonian Graphs

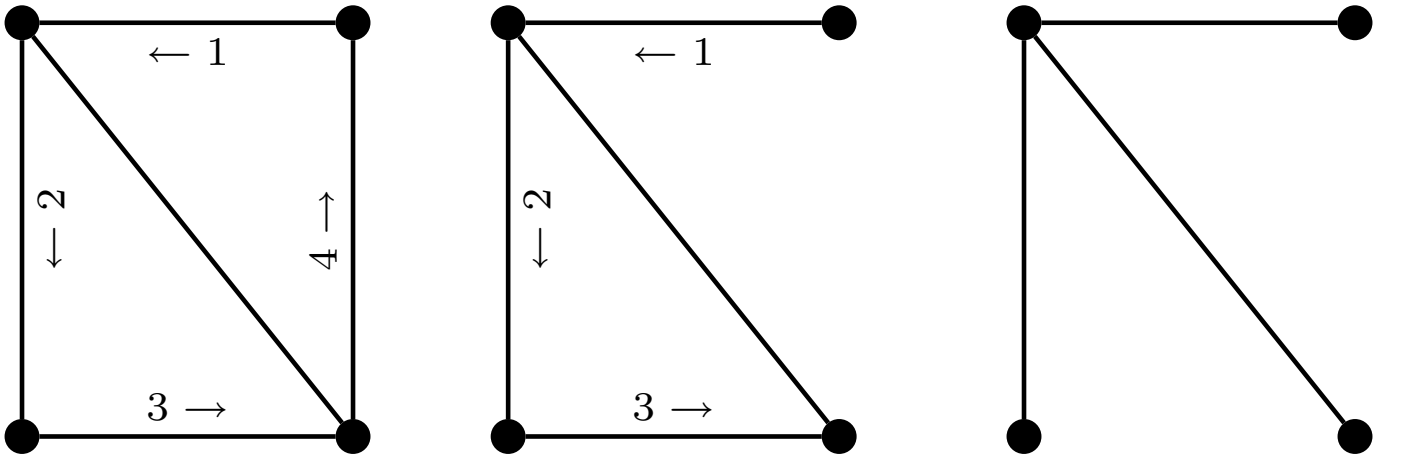
Definition 25 (Hamiltonian Graph)

A (connected) graph, G, is Hamiltonian if there exists a closed trail that passes exactly once through each vertex of G. Such a trail is an Hamiltonian cycle or Hamiltonian tour.

A non-Hamiltonian graph, G, is semi-Hamiltonian if there exists a (open) path passing through every vertex.

- Technical point: A Hamiltonian cycle is a cycle except for the null graph N_1.
- While Hamiltonian tours appear to similar to Eulerian tours, passing through each vertex once rather than through each edge, Hamiltonian tours are significantly harder than Eulerian, in terms of conditions for existence, and construction.

Example



Hamiltonian Semi-Hamiltonian Non-Hamiltonian

Given a graph, *G*, of interest is

- Does an Hamiltonian tour exist, i.e., is *G* Hamiltonian?
  — Some necessary and some sufficient conditions are known.
- How to find/construct a Hamiltonian tour?
  — No known sub-exponential algorithm.

Necessary Conditions for Hamiltonian Graphs I

Unfortunely, unlike the case for Eulerian graphs, necessary and sufficient conditions for Hamiltonian graphs are not known. The best simple result is due to Dirac (1952) which was later generalised by Ore (1960) as follows.

Theorem 26 (Ore, 1960)

If G is a simple graph with n (≥ 3) vertices, and if

$$\deg(v) + \deg(w) \geq n$$

for each pair of vertices, then G is Hamiltonian.

and, there is a slightly stronger result in the special case of $w = v$

Corollary 27 (Dirac, 1952)

If G is a simple graph with n (≥ 3) vertices, and if

$$\deg(v) > n/2$$

for each vertex v, then G is Hamiltonian.

Review Exercises 5 (Hamiltonian Graphs)

Question 1:

(a) For which values of n is K_n Hamiltonian?

(b) Which complete bipartite graphs are Hamiltonian?

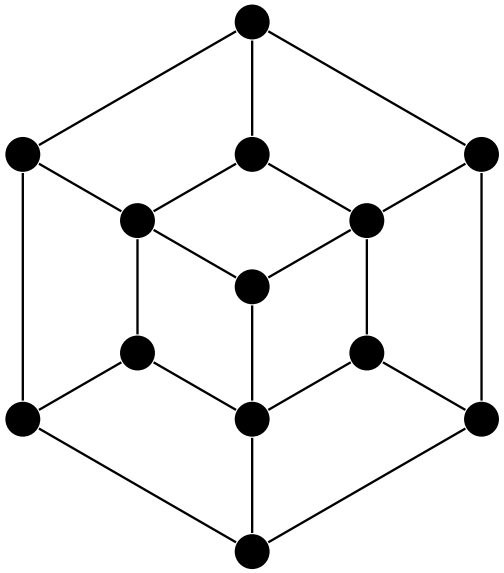
(c) For which values of n is the wheel W_n Hamiltonian?

(d) For which values of n is the k-cube Q_k Hamiltonian?

Question 2:

(a) Prove that, if G is a bipartite graph with an odd number of vertices then G is non-Hamiltonian.

(b) Deduce that the graph bellow is non-Hamiltonian.



(c) Show that if n is odd, it is not possible for a knight to visit all the squares of an $n \times n$ chessboard exactly once by knight's moves and return to the starting point.

Outline

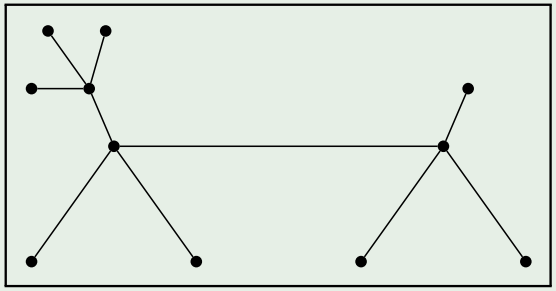
Trees and Forests

Definition 28 (Forest, Tree)

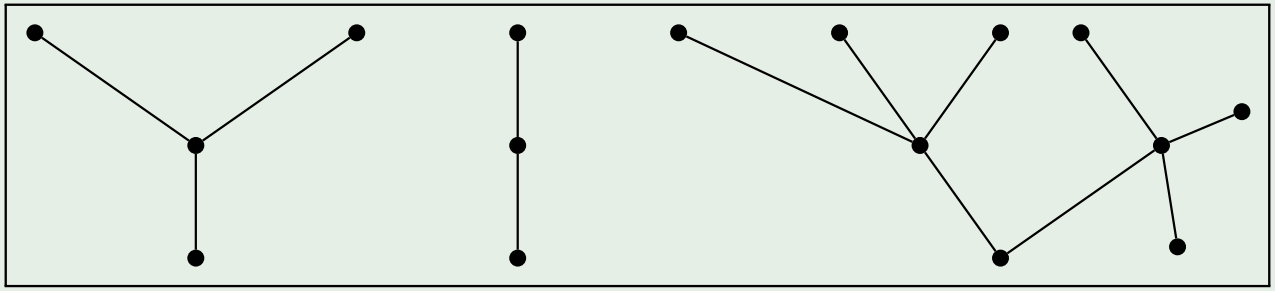
A forest is a graph that contains no cycles, and a connected forest is a tree.

- A tree is is the simplest non-trivial type of graph.
- Principal concept for searching through graphs.
- Underlying data structure in mutable sorted collections.

Example 29



Tree



Forest

Properties of Trees

Theorem 30

Let T be a graph with n vertices. Then the following are equivalent:

(i) *T is a tree.*

(ii) *T contains no cycles and has $n - 1$ edges.*

(iii) *T is connected and has $n - 1$ edges.*

(iv) *T is connected and each edge is a bridge.*

(v) *Any two vertices of T are connected by exactly one path.*

(vi) *T contains no cycles, but the addition of any new edges creates exactly one cycle.*

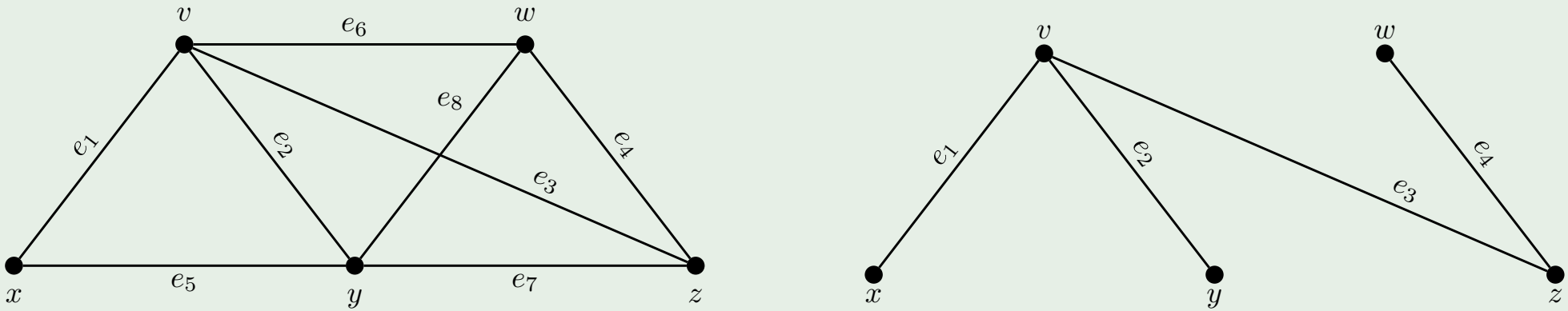
Spanning Trees

Given any connected graph, G, we can choose a cycle and removes any one of its edges, and the resulting graph remains connected. If we repeat the process until no cycle remains we obtain a spanning tree of G.

Definition 31 (Spanning Tree, Spanning Forest)

A spanning tree of a connected graph G is a tree that connects all the vertices of G.
More generally, if G is not connected we obtain a spanning forest.

Example 32



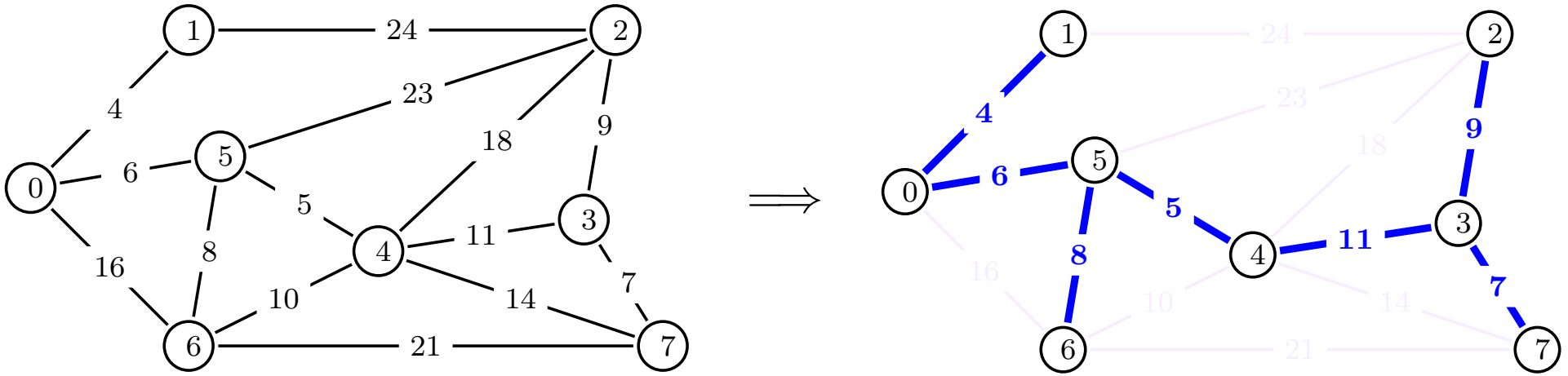
A simple graph and one possible spanning tree.

Minimum Spanning Tree

Problem 33 (Minimum Spanning Tree (MST))

Given connected graph G with positive edge weights, find a minimum weight set of edges that connects all of the vertices.

Example



Cayley's Theorem (1889)

There are n^{n-2} spanning trees on the complete graph on n vertices $\Rightarrow$ Can't solve MST problem by brute force.

Applications of MST

- Network design.
  - Telephone, electrical (Otakar Boruvka, 1926), hydraulic, TV cable, computer, road.
- Cluster analysis
  - Analyzing fungal spore spatial patterns
  - Microarray gene expression data clustering
  - Finding clusters of quasars and Seyfert galaxies
- Approximation algorithms for NP-hard problems
  - Travelling salesperson problem
- Indirect applications
  - Max bottleneck paths
  - LDPC codes for error correction
  - Learning salient features for real-time face verification
  - Reducing data storage in sequencing amino acids in a protein
  - Model locality of particle interactions in turbulent fluid flows
  - Autoconfig protocol for Ethernet bridging to avoid cycles in a network.

The Minimum Spanning Tree Theorem

The following theorem is the basis of the two algorithms covered in this lecture for construction of minimum spanning trees.

Theorem 34 (MST Theorem)

Given graph, G, with vertex set V and edge iset E. Let U be a proper subset of V, i.e., $U \subset V$.
If the edge (u, v) has the lowest cost among edges such that $u \in U$ and $v \in V - U$, then there exists an MST that contains the edge (u, v).

- This result allows us to develop algorithms based on picking individual edges, $e = (u, v)$, and by testing whether e is the cheapest edges we decide to keep it or not.

Algorithms for Constructing Minimum Spanning Trees

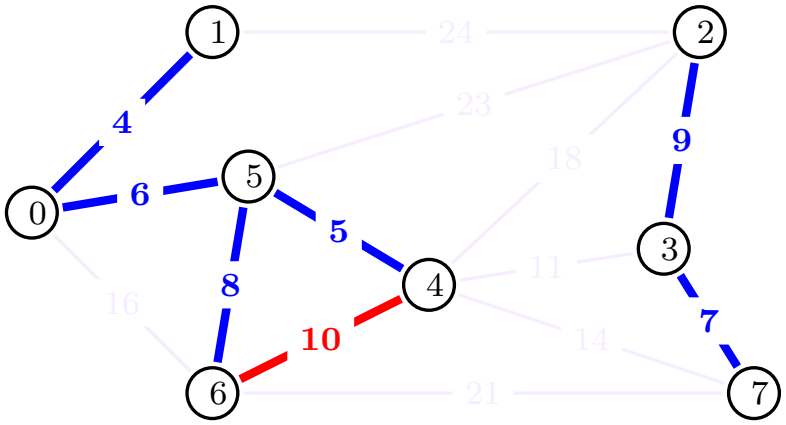
The following greedy algorithms have been used to solve the minimum spanning tree problem.

- Start with an empty forest (one vertex) and grow trees, adding edges in non-decreasing weight order, and skipping edges that create a cycle. Stop when the forest merges to form a single spanning tree —- Kruskal's algorithm.
- Start with an empty tree (one vertex) and grow, by adding edges in non-decreasing weight order, and skipping edges that create a cycle. Stop when tree spans the graph — Prim's algorithm.
- Start with the graph and shrink graph by removing edges in non-increasing order ensuring that resulting graph is still connected. Stop when there are no cycles.

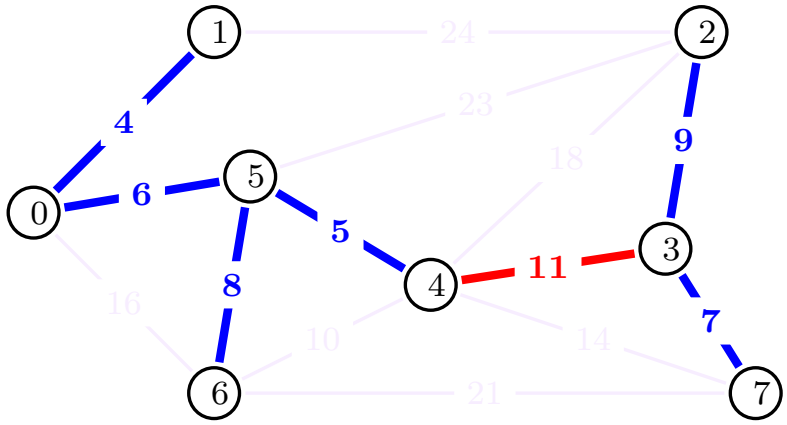
Kruskal's Algorithm

Kruskal's Algorithm (1956)

1. Initialise forest $F = \{\}$
2. Consider edges in non-descending order of weight.
   1. If adding edge e to forest F does not create a cycle, then add it. Otherwise, discard e.
   2. If more than one edge with same weight then select any edge.



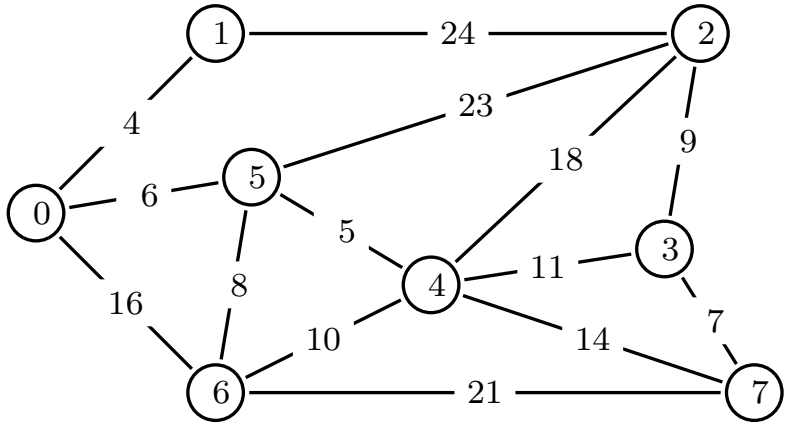
Adding edge 4-6 creates cycle
($\Rightarrow$ discard edge)



Adding edge 3-4 connects two components
($\Rightarrow$ edge is in MST)

Example

Graph



Input

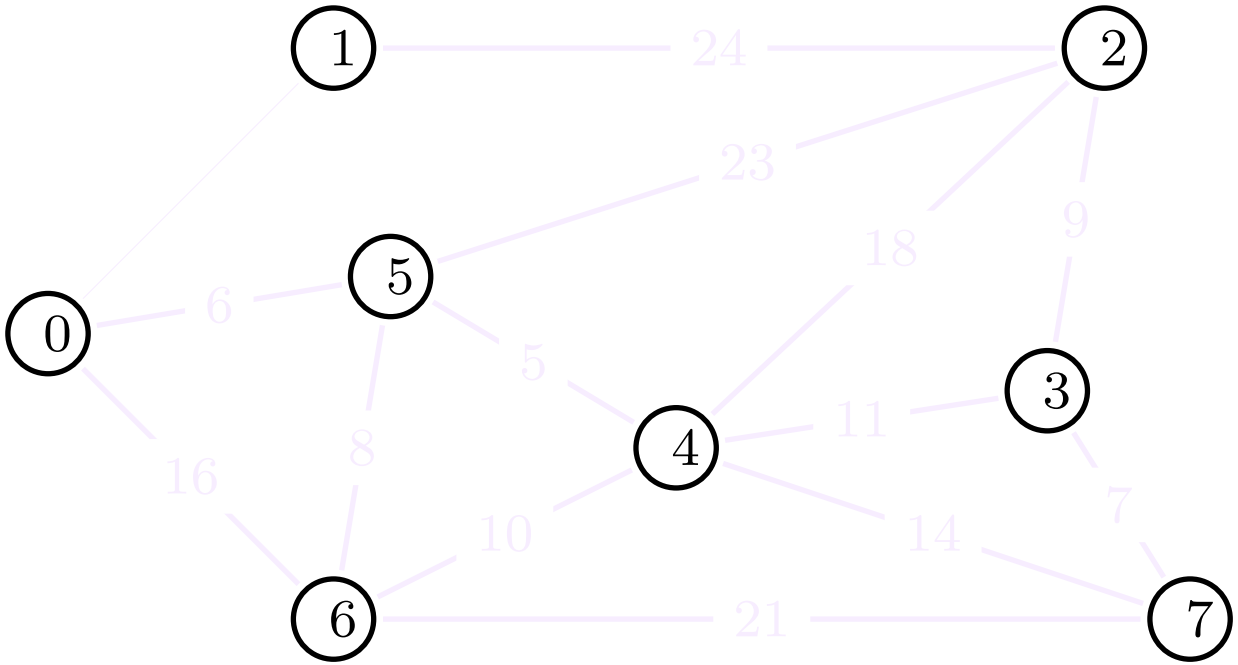
Graph (8, 14)

0:	6(16)	1(4)	5(6)	
1:	2(24)	0(4)		
2:	1(24)	4(18)	5(23)	3(9)
3:	2(9)	4(11)	7(7)	
4:	2(18)	6(10)	7(14)	3(11) 5(5)
5:	2(23)	0(6)	4(5)	6(8)
6:	0(16)	4(10)	7(21)	5(8)
7:	4(14)	6(21)	3(7)	

Output

```
Cost of MST: 50
MST: [0-1 (4), 4-5 (5), 0-5 (6), 3-7 (7), 5-6 (8), 2-3 (9), 3-4 (11)]
```

Kruskal Example



weight | m | k

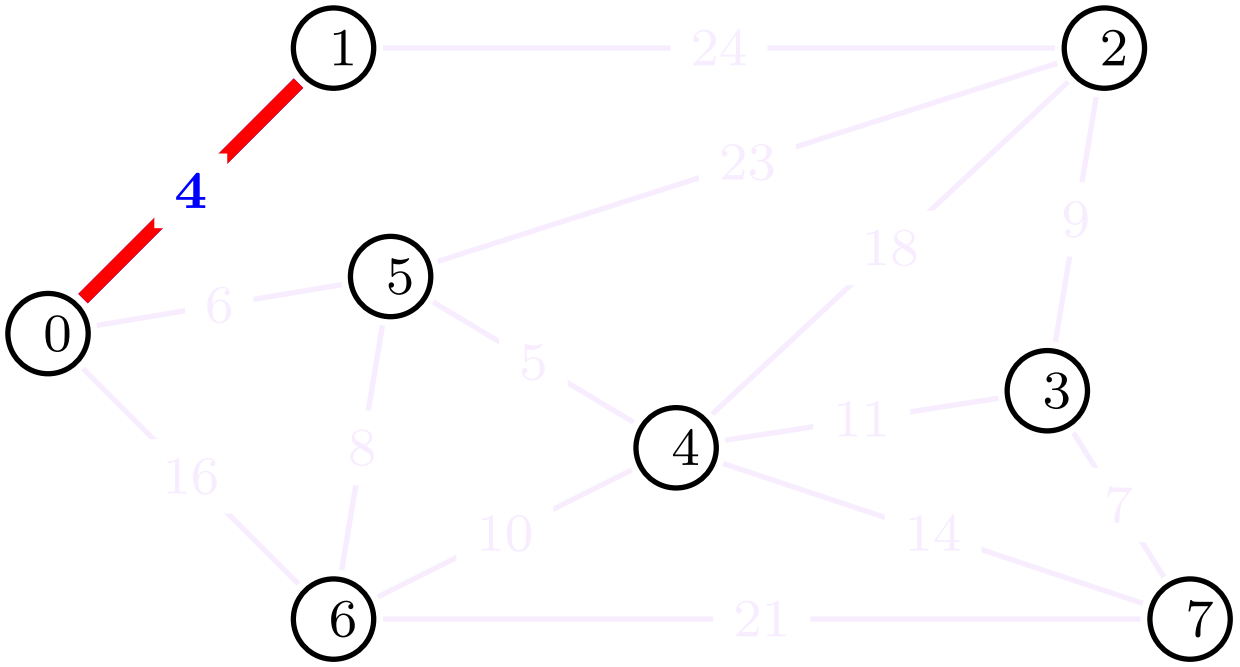
0

mst

Start with total weight of zero and no edges added to MST.

≪< 1 2 3 4 5 6 7 8 9 >≫

Kruskal Example



weight	m	k
4	0	1

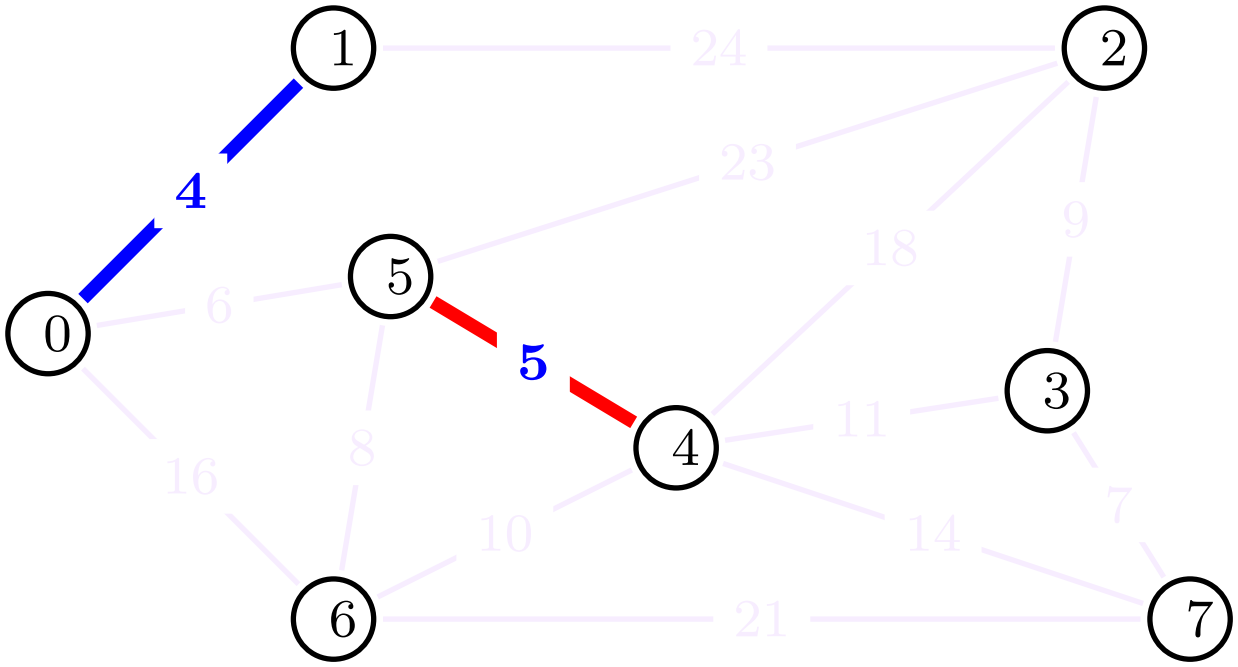
mst

Edge(0,1)	4

Find (next) edge with lowest weight, $(0, 1)$ and add it unless it causes a cycle. It doesn't so add it.

≪< 1 2 3 4 5 6 7 8 9 >≫

Kruskal Example



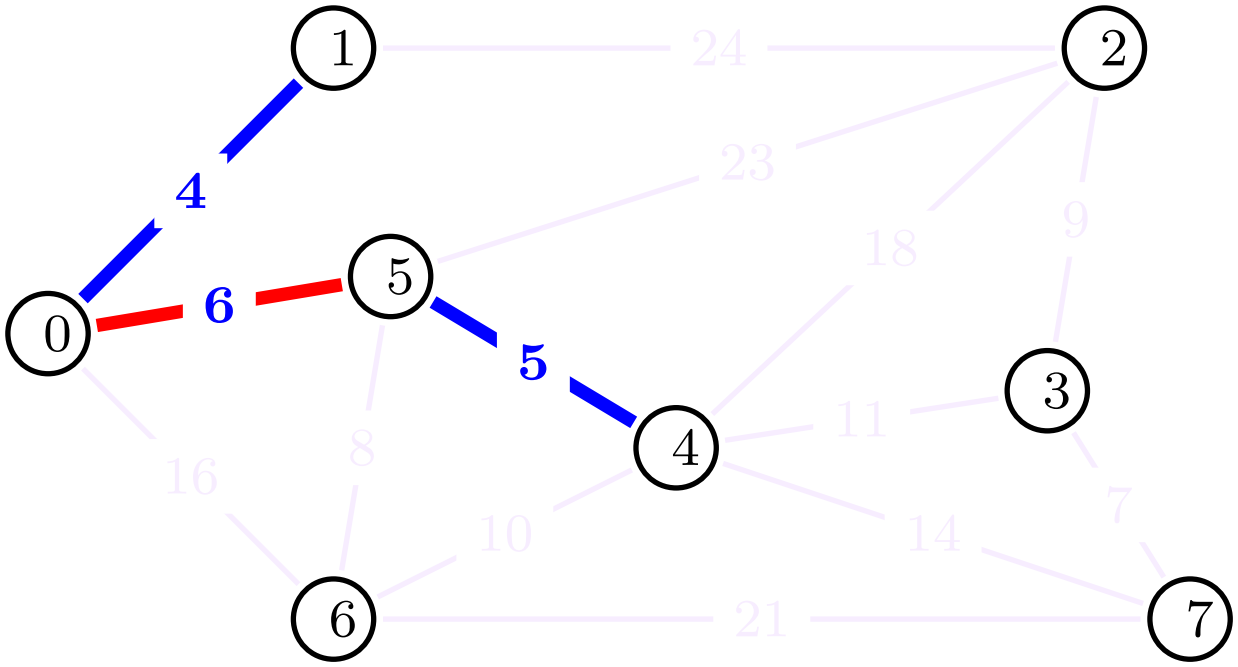
weight	m	k
9	1	2

mst

Edge(0,1)	4
Edge(4,5)	5

Find (next) edge with lowest weight, $(4, 5)$ and add it unless it causes a cycle. It doesn't so add it.

Kruskal Example



weight	m	k
15	2	3

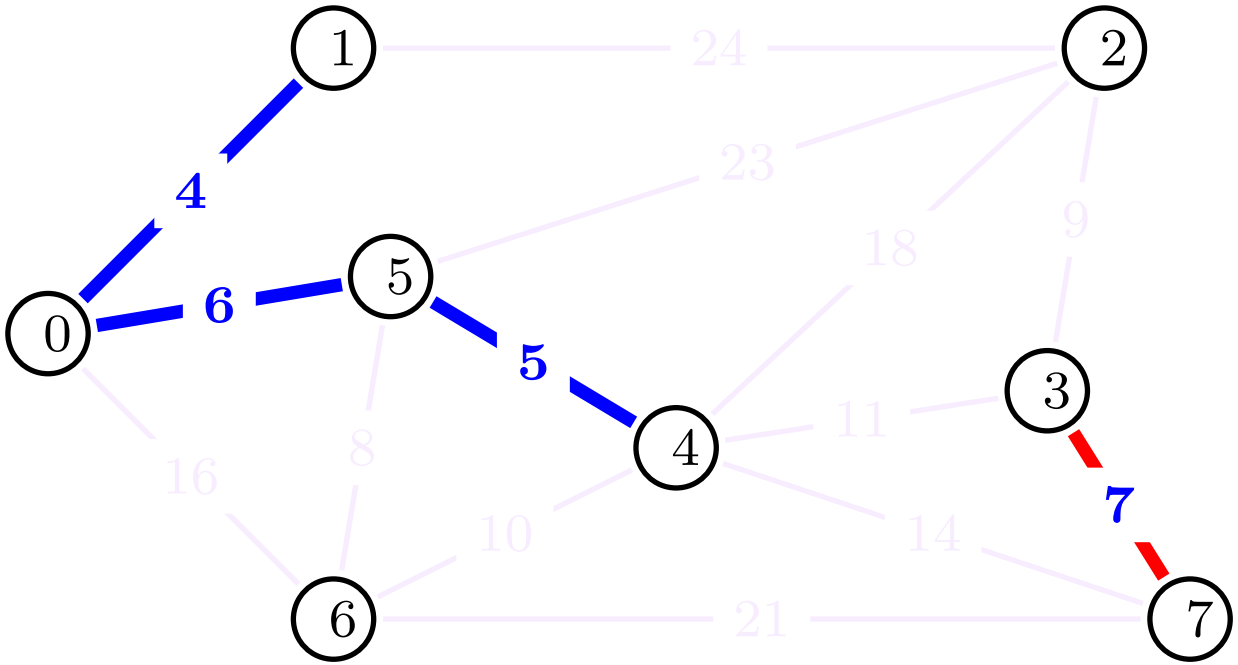
mst

Edge(0,1)	4
Edge(4,5)	5
Edge(0,5)	6

Find (next) edge with lowest weight, $(0, 5)$ and add it unless it causes a cycle. It doesn't so add it.

≪< 1 2 3 4 5 6 7 8 9 >≫

Kruskal Example



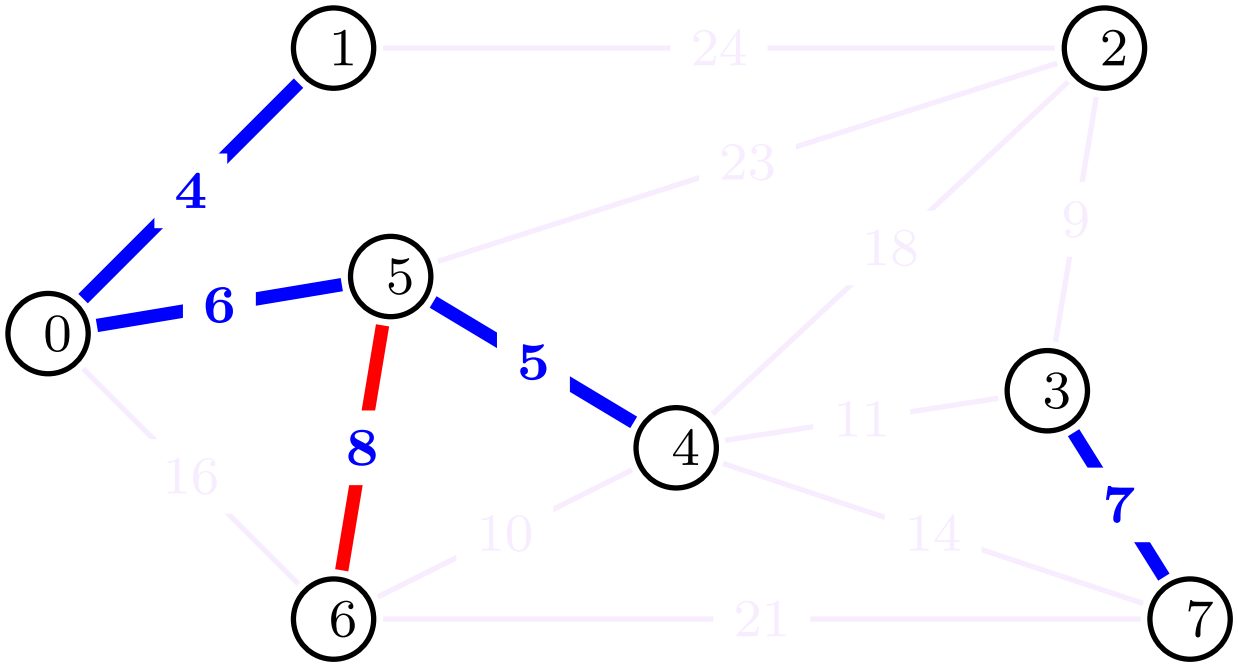
weight	m	k
22	3	4

mst

Edge(0,1)	4
Edge(4,5)	5
Edge(0,5)	6
Edge(3,7)	7

Find (next) edge with lowest weight, $(3, 7)$ and add it unless it causes a cycle. It doesn't so add it.

Kruskal Example



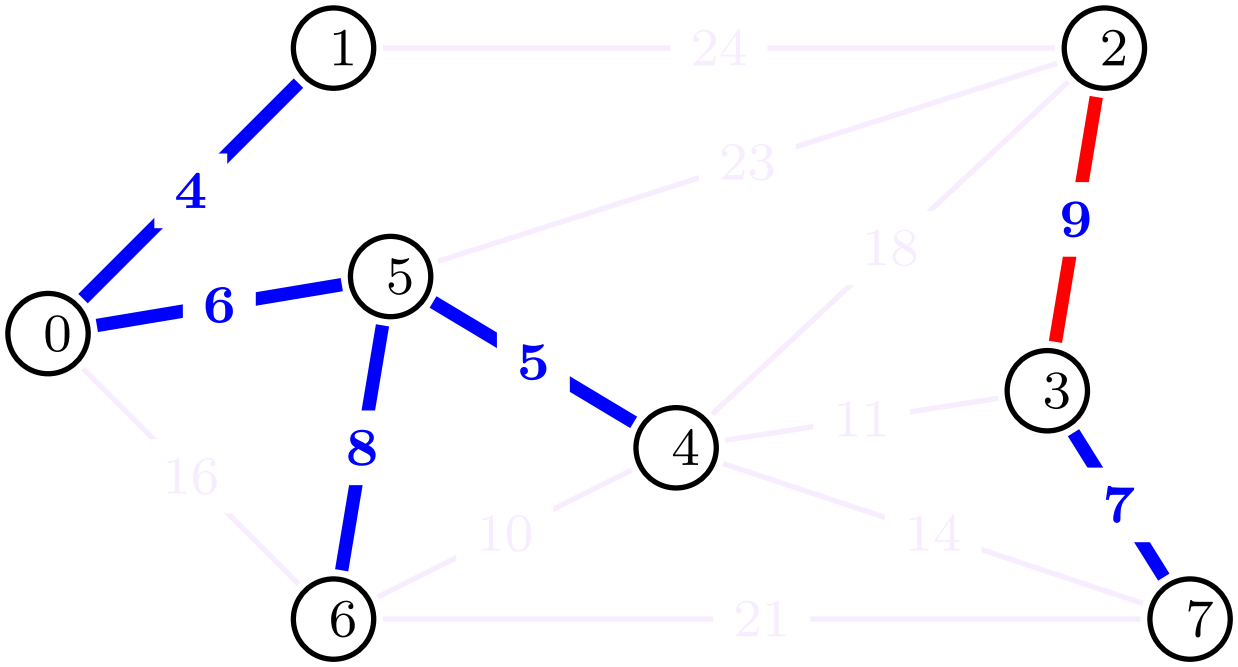
weight	m	k
30	4	5

mst

Edge(0,1)	4
Edge(4,5)	5
Edge(0,5)	6
Edge(3,7)	7
Edge(5,6)	8

Find (next) edge with lowest weight, $(5, 6)$ and add it unless it causes a cycle. It doesn't so add it.

Kruskal Example



weight	m	k
39	5	6

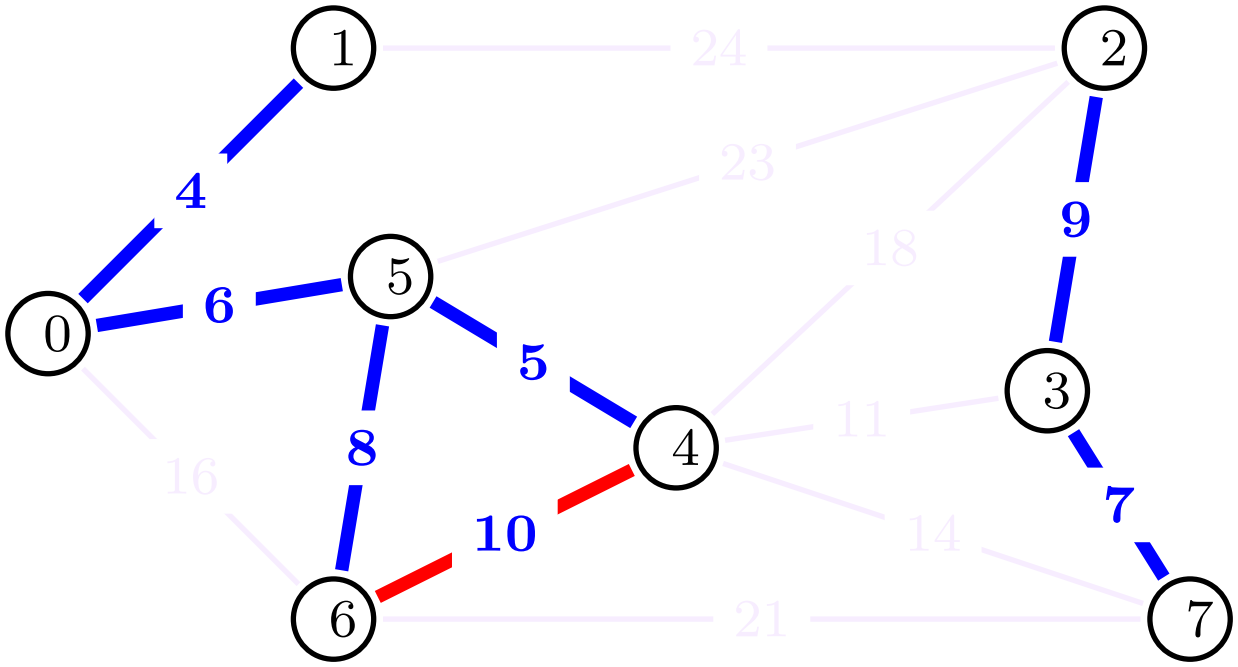
mst

Edge(0,1)	4
Edge(4,5)	5
Edge(0,5)	6
Edge(3,7)	7
Edge(5,6)	8
Edge(2,3)	9

Find (next) edge with lowest weight, $(2, 3)$ and add it unless it causes a cycle. It doesn't so add it.

≪< 1 2 3 4 5 6 7 8 9 >≫

Kruskal Example



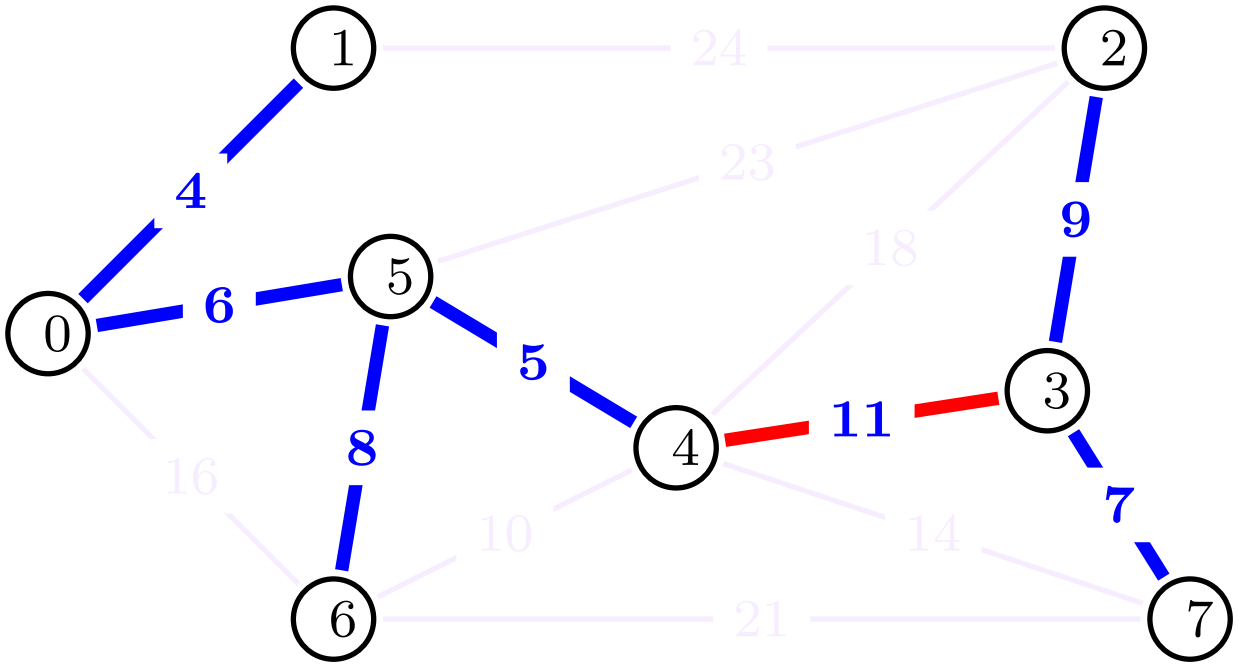
weight	m	k
39	6	6

mst

Edge(0,1)	4
Edge(4,5)	5
Edge(0,5)	6
Edge(3,7)	7
Edge(5,6)	8
Edge(2,3)	9

Find (next) edge with lowest weight, $(4, 6)$ and add it unless it causes a cycle. It does so skip it.

Kruskal Example



weight	m	k
50	7	7

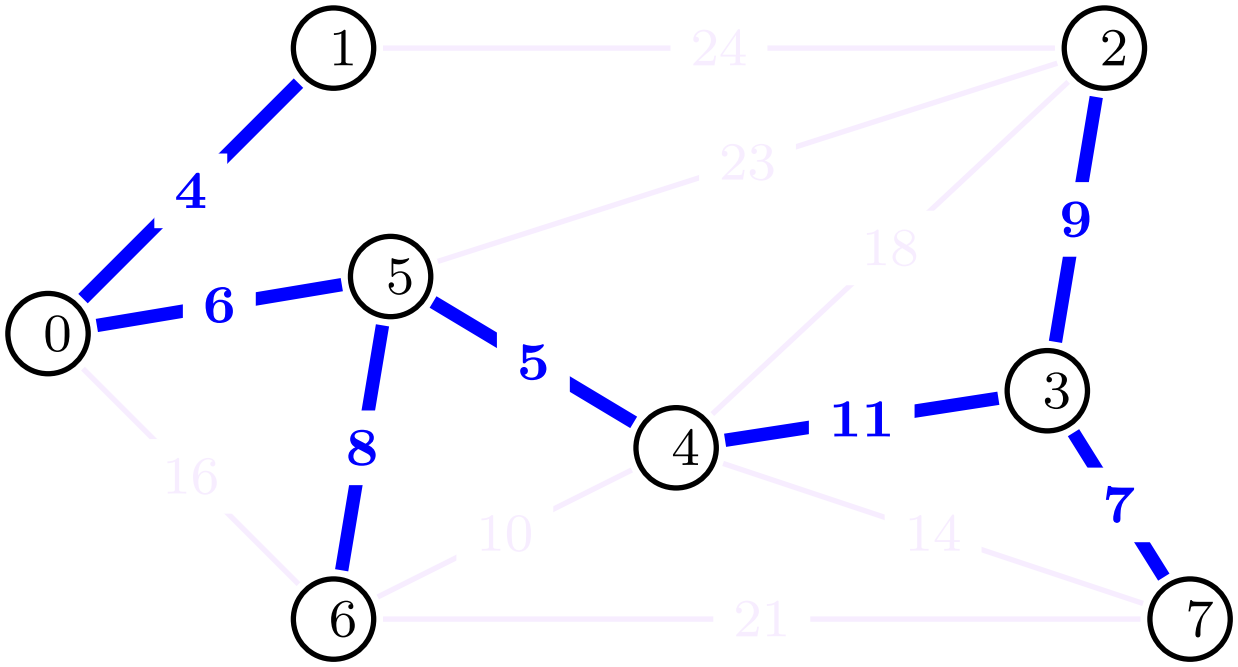
mst

Edge(0,1)	4
Edge(4,5)	5
Edge(0,5)	6
Edge(3,7)	7
Edge(5,6)	8
Edge(2,3)	9
Edge(3,4)	11

Find (next) edge with lowest weight, $(3, 4)$ and add it unless it causes a cycle. It doesn't so add it.

≪< 1 2 3 4 5 6 7 8 9 >≫

Kruskal Example



weight	m	k
50	7	7

mst

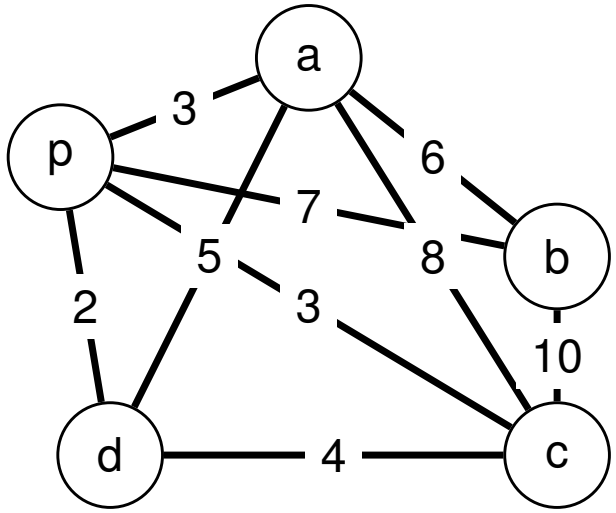
Edge(0,1)	4
Edge(4,5)	5
Edge(0,5)	6
Edge(3,7)	7
Edge(5,6)	8
Edge(2,3)	9
Edge(3,4)	11

We have added $n - 1 = 7$ edges connecting all $n = 8$ vertices, so can stop.

Review Exercises 6 (Minimum Spanning Tree (MST))

Question 1:

A company is considering building a gas pipeline to connect 4 wells (*a*, *b*, *c* and *d*) to a process plant *p*. The possible pipelines that they can construct and their costs (in millions of euro) are shown in the accompanying graph.



What pipelines do you suggest be built and what is the total cost of your suggested pipeline network?

Question 2:

Apply Kruskal's algorithm to determine a minimum cost spanning tree for the graph with the following cost matrix. How many such trees are there?

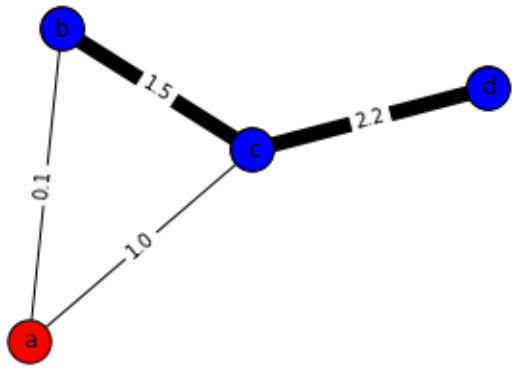
	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
A	0	12	0	14	11	0	17	8
B	12	0	9	0	12	15	10	9
C	0	9	0	18	14	31	0	9
D	14	0	18	0	0	6	23	14
E	11	12	14	0	0	15	16	0
F	0	15	31	6	15	0	8	16
G	17	10	0	23	16	8	0	22
H	8	9	9	14	0	16	22	0

Outline

NetworkX — A quick example

quick_example.py

```
3  import networkx as nx
4
5  G = nx.Graph()
6  G.add_edge('a', 'b', weight=0.1)
7  G.add_edge('b', 'c', weight=1.5)
8  G.add_edge('a', 'c', weight=1.0)
9  G.add_edge('c', 'd', weight=2.2)
10
11 nx.draw(G)
12
13 print('Nodes = ', G.nodes())
14 print('Edges = ', G.edges())
15 print('Path = ', nx.shortest_path(G, 'b', 'd'))
```




```
Nodes =  ['a', 'c', 'b', 'd']
Edges =  [('a', 'c'), ('a', 'b'), ('c', 'b'), ('c', 'd')]
Path =  ['b', 'c', 'd']
```

NetworkX Overview

Features

- Node-centric view of network.
- NetworkX defines no custom node objects or edge objects.
- Nodes can be any hashable object, while edges are tuples with optional edge data (stored in dictionary).
- Any Python object is allowed as edge data and it is assigned and stored in a Python dictionary (default empty).
- Nearly 100% python — some rendering capabilities are based on external tools (graphviz).
- Extensive set of native readable and writable formats.

Use cases

- Focus on computational network modelling.
- Prototyping new algorithms or models.
- Suitable for medium sized problems (1M/10M nodes/edges).
  - Most of the core algorithms rely on extremely fast legacy code.
  - But poor use of memory/threads needed for larger problems.

Importing the library

NetworkX

```
>>> import networkx as nx
```

- We could use "`from network import *`" but then searching for functions is more difficult, since can't use the code completion features or use `dir` function.

matplotlib

```
>>> import matplotlib.pyplot as plt
```

- Only needed for rendering graphs.
- Use IPython magic `%matplotlib inline` to embed images.

numpy

```
>>> import numpy as np
```

- Only needed when explicitly dealing with matrix representation.

Creating graph and adding nodes

Construction of empty graph

```
>>> g = nx.Graph()
```

- There are different Graph classes for undirected and directed networks.
- NetworkX includes many graph generator functions and facilities to read and write graphs in many formats.

Adding/Removing Nodes

- One node at a time:
  ```
  >>> g.add_node(1)
  ```
- A list of nodes
  ```
  >>> g.add_nodes_from([8,15])
  ```
- A container of nodes
  ```
  >>> h = nx.path_graph(10)
  >>> g.add_nodes_from(h)
  ```
- You can remove any node of the graph (error if not exist)
  ```
  >>> g.remove_node(2)
  ```

Nodes

A node can be any hashable object such as strings, numbers, files, functions, and more.

- Create an empty graph . . .

```
>>> g = nx.Graph()
```

- Import the math library . . .

```
>>> import math
```

- Add the function math.cos as a node to graph g . . .

```
>>> g.add_node(math.cos)
```

- Create a file handle . . .

```
>>> fh = open('tmp.txt', 'w')
```

- Add the file handle as a node to the graph . . .

```
>>> g.add_node(fh)
```

- List graph nodes . . .

```
>>> print (g.nodes())
[<open file 'tmp.txt', mode 'w' at 0x10fd01f60>, <built-in function cos>]
```

Edges

Adding a single edge

```
>>> g.add_edge(1,2)
```

or create an (edge) tuple and unpack it using *

```
>>> e = (1,2)
>>> g.add(*e)
```

Adding list of edges

```
>>> g.add_edges_from([(1,2), (1,3)])
```

Adding from a container of edges

```
>>> h = nx.path_graph(10)
>>> g.add_edges_from(h.edges())
```

Removing edges

Can remove a single edge using g.remove_edge or a list of edges using g.remove_edges_from. Exception raised if edge does not exist.

Accessing Nodes and Edges

```
>>> g = nx.Graph()
>>> g.add_edges_from([(1 ,2) ,(1 ,3)])
>>> g.add_node('a')
```

Graph order and size

```
>>> g.number_of_nodes()          # also g.order()
4
>>> g.number_of_edges()          # also g.size()
2
```

Node and Edge lists

```
>>> g.nodes()
['a', 1, 2, 3]
>>> g.edges()
[(1, 2), (1, 3)]
```

Adjacency

```
>>> g.neighbors(1)
[2, 3]
>>> g.degree(1)
2
```

Accessing Node and Edge Properties

```
>>> g = nx.Graph()
```

Node Properties

Any NetworkX graph behaves like a Python dictionary with nodes as primary keys . . .

```
>>> g.add_node(1, time='10am', day='Fri')
>>> g.node[1]
{'day': 'Fri', 'time': '10am'}
>>> g.node[1]['time']
'10am'
```

Edge properties

Similar to node, but the special edge attribute 'weight' should always be numeric and holds values used by algorithms requiring weighted edges.

```
>>> g.add_edge(1, 2, weight=4.0, status='stable')
>>> g[1][2]
{'capacity': '42', 'weight': 4.0}
```

Node and Edge Iterators

Many applications require iteration over nodes or over edges — simple in NetworkX.

```
>>> g = nx.Graph()
>>> g.add_edge(1, 2, weight=1.5)
>>> g.add_edge(2, 3, weight=2.5)
```

Node Iterator

```
>>> for node in g.nodes():
        print (node, g.degree(node))
1 1
2 2
3 1
```

Edge Iterator

```
>>> for u, v, d in g.edges(data=True):
        print (u, v, d['weight'])
1 2 1.5
2 3 2.5
```