# Lecture 1: The Basics of Graph Theory

Algorithms and Data Structures - 24/25

Dr. Amitabh Trehan

amitabh.trehan@durham.ac.uk

**Based on the slides of ADS-21/22 by Dr. George Mertzios*

# Contents for today's lecture

- Graphs and types of graphs;
- Graph models;
- Basic terminology;
- Classes of graphs;
- Examples and exercises.

# Formal definitions

## Definition

A graph $G$ is a pair $(V(G), E(G))$, where $V(G)$ is a nonempty set of vertices (or nodes) and $E(G)$ is a set of unordered pairs $\{u, v\}$ with $u, v \in V(G)$ and $u \neq v$, called the edges of $G$.

- $V(G)$ can be infinite, but all our graphs here will be finite.
- If no confusion can arise, we write $uv$ instead of $\{u, v\}$.
- If the graph $G$ is clear from the context, we write $V$ and $E$ instead of $V(G)$ and $E(G)$.

- It often helps to draw graphs:
  - represent each vertex by a point, and
  - each edge by a line or curve connecting the corresponding points;
  - only endpoints of lines/curves matter, not the exact shape.

# Types of graphs

Possible variations in definition:

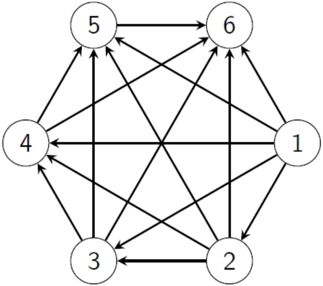
- directed graphs or digraphs — edges can have directions.
  - the Web graph: vertices are webpages and edges are hyperlinks.
  - the precedence graph: vertices are program statements, edges reflect execution order.
  - the influence graph: vertices are people in the group, edges mean "influences"
- multi-graphs — multiple edges are allowed between two vertices.
  - the air link graph: several different airlines can fly between two towns.
- pseudo-graphs — edges of the form $uu$, called loops, are allowed.
  - region pseudo-graph in computer graphics: Vertices are connected regions, edges mean "can get from one to the other by crossing a fence".
- vertex- or edge-weighted graphs — vertices and/or edges can have weights
  - the road map graph: weights on edges are distances.

By default, all our graphs are simple undirected or simple directed graphs (sometimes edge-weighted too), i.e. no multiple edges, no loops.

# Types of graphs

Case study example: sport tournament

- vertices are teams
- directed edge from $x$ to $y$: team $x$ wins over team $y$




- team 1: absolute winner ("Unconquerable!")
- team 6: absolute loser

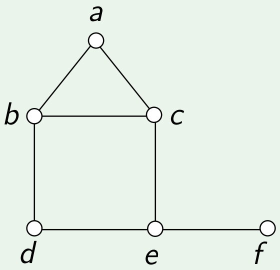
Q: does always an absolute winner / loser exist?

# Terminology

## Definitions

Let $G$ be a graph and $uv$ an edge in it. Then

- $u$ and $v$ are called endpoints of the edge $uv$
- $u$ and $v$ are called neighbours or adjacent vertices
- $uv$ is said to be incident to $u$ (and to $v$)
- if $vw$ is also an edge (where $w \neq u$) then $uv$ and $vw$ are called adjacent.



# More terminology

## Definitions

Let $G = (V, E)$ be a graph. The neighbourhood of a vertex $v \in V$, notation $N(v)$, is the set of neighbours of $v$, i.e., $N(v) = \{ u \in V \mid uv \in E \}$.

The degree of a vertex $v \in V$, notation $deg(v)$, is the number of neighbours of $v$, i.e. $deg(v) = |N(v)|$.

With $\delta(G)$ or $\delta$ we denote the smallest degree in $G$, and with $\Delta(G)$ or $\Delta$ the largest degree.

A vertex with degree 0 will be called an isolated vertex.

A vertex with degree 1 an end vertex or a pendant vertex.

## Definition

A subgraph $G' = (V', E')$ of $G = (V, E)$ is a graph with $V' \subseteq V$ and $E' \subseteq E$. This subgraph is called proper if $G' \neq G$ and spanning if $V' = V$.

It is called induced subgraph if $E'$ contains all edges of $E$ between vertices of $V'$, i.e. it is obtained by just removing from $G$ all vertices of $V \setminus V'$ (and their edges).

# First theorem in Graph Theory

Can you guess the relationship between the sum of the degrees of the vertices of a graph $G$ and the number of edges of $G$?

**Theorem (Handshaking Lemma)**

*Let $G = (V, E)$ be a graph. Then $\sum_{v \in V} deg(v) = 2|E|$.*

How to prove this?

**Proof.**

Every edge has two endpoints and contributes one to each of their degrees, so contributes two to the sum of the degrees of all the vertices of $V$. $\square$

This simple relationship can be useful for proving non-existence of graphs with certain properties.

# The most basic graph classes

Some graphs appear so often that they got special names or even special dedicated symbols.

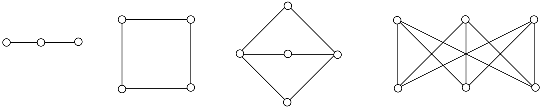


Figure: Special graph classes

All four of these graphs can be described as a $K_{p,q}$: a graph consisting of two disjoint vertex sets on $p$ and on $q$ vertices, and all possible edges between these two vertex sets (and no other edges). So, $K_{p,q}$ has $p \cdot q$ edges.

## Definition

$K_{p,q}$ is called a complete bipartite graph. Any subgraph of $K_{p,q}$ is called a bipartite graph.
So a graph is bipartite if and only if we can partition its vertex set to two vertex sets such that every edge has one endpoint in each set.

Bipartite graphs play an eminent role in scheduling and assignment problems.

# Lecture 2: Paths, Cycles, Connectivity

Dr. Amitabh Trehan

amitabh.trehan@durham.ac.uk

**Based on the slides of ADS-21/22 by Dr. George Mertzios*

# Contents for today's lecture

- Paths and directed paths;
- The shortest path problem;
- Connectivity and connected components;
- Eulerian and Hamiltonian cycles;
- Examples and exercises.

# Walks, paths, cycles, and distances

- A walk in a graph $G$ is a sequence of edges $v_0v_1, v_1v_2, v_2v_3, \ldots, v_{n-1}v_n$. In this case we also say that $v_0, v_1, \ldots, v_n$ is a walk in $G$.
- A walk $v_0, v_1, \ldots, v_n$ in $G$ is a path if all $v_i$'s are distinct. In this case we also say that $v_0, v_1, \ldots, v_n$ is a path in $G$.
- A walk $v_0, v_1, \ldots, v_n$ with $v_0 = v_n$ is called a circuit or closed walk.
- A closed walk is a cycle (or simple circuit) if all $v_i$'s in it are distinct except $v_0 = v_n$.
- If $G$ is a directed graph then the directed paths and directed cycles are defined in a natural way, with each edge being directed from $v_i$ to $v_{i+1}$.

- The length of a path or a cycle is the number of edges in it.
- The distance between vertices $u$ and $v$ in a graph, denoted $dist(u, v)$, is the length of a shortest path from $u$ to $v$ if such a path exists, and $\infty$ otherwise.
- The diameter of a graph is the largest distance between two vertices in it

# The Erdös-Bacon number

## Definition

For any person, their Erdös-Bacon number is the sum of their Erdös number and their Bacon number.

There are not many people with a small Erdös-Bacon number.

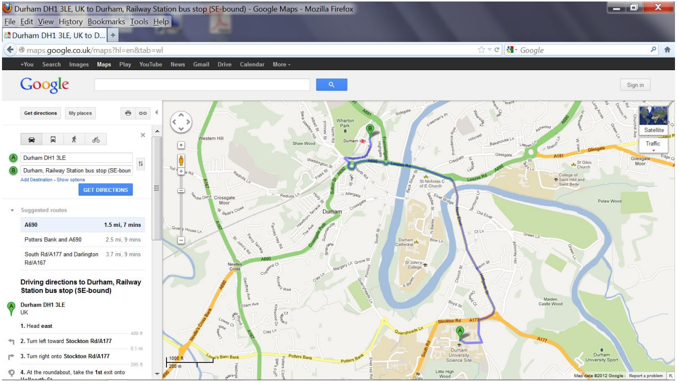
For example, the following people have Erdös-Bacon number 7 or less.

- Btw, my Erdös-Bacon number is $\infty$, but I have a colleague who has a co-author (Hubie Chen) with Erdös-Bacon number 5 (3+2)

# Shortest-path problems

In a graph (possibly with edge weights, the problem of computing a path from a given vertex $u$ ("source") to a given vertex $v$ ("target") with the smallest total length (or weight) is known as the shortest-path problem.

We all often (use software applications that) compute such paths. Example?



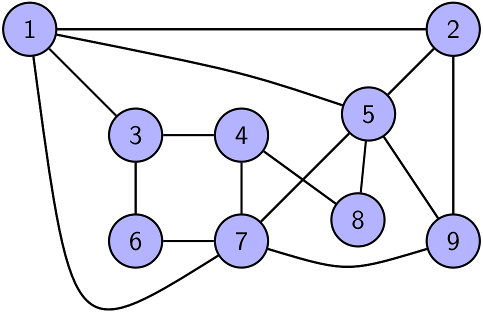

We will learn about algorithms for the (unweighted) problem in a few lectures.

# Connectivity

## Definition

A graph $G = (V, E)$ is called connected if, between every pair of vertices $u, v$, there exists at least one path in $G$.

A connected component of $G$ is a maximal connected subgraph of $G$.

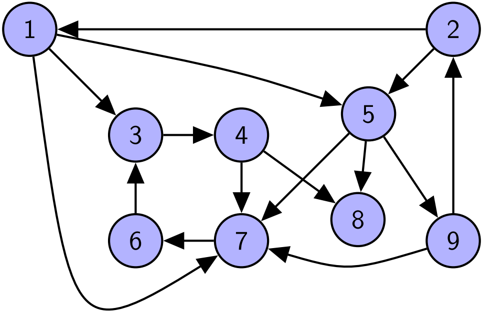



- Is this graph connected?
- What about this graph?
- How many connected components does this graph have?

# Strong connectivity

## Definition

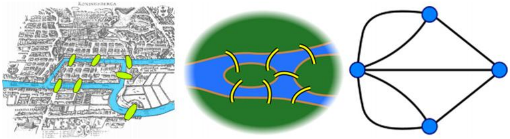
A directed graph $G$ is called (weakly) connected if the graph obtained from $G$ by forgetting directions is connected.
A directed graph is called strongly connected if any two distinct vertices are connected by directed paths in both directions.
A strongly connected component (or simply strong component) of a digraph $G$ is a maximal strongly connected subgraph of $G$.



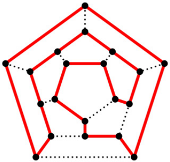

- Is this graph strongly connected?

# Special circuits/cycles in graphs

- Can we travel along the edges of a given graph $G$ so that we start and finish at the same vertex and traverse each edge exactly once?
    - Such a circuit in $G$ is called a Eulerian circuit, after Leonhard Euler (1707-83).



- Can we travel along the edges of a given graph so that we start and finish at the same vertex and visit each vertex exactly once?
    - Such a cycle is called a Hamiltonian cycle, after William Hamilton (1805-65).



- Detecting one of these two types of circuits is easy, while detecting the other is not easy at all. Which is which?

# Travelling Salesman Problem (TSP)

The (famous) TSP is the following problem:

- A salesman should visit cities $c_1, c_2, \ldots, c_n$ in some order, visiting each city exactly once and returning to the starting point
- A (positive integer) cost $d(i, j)$ of travel between each pair $(c_i, c_j)$ is known.
- Goal: find an optimal (i.e. cheapest) route for the salesman.

Given a graph $G$ with set $V$ of vertices ($|V| = n$) and set $E$ of edges,

- for each vertex $v$, create a city $c_v$;
- for each pair of distinct $u, v \in V$, set $d(c_u, c_v) = 1$ if $uv \in E$ and $d(c_u, c_v) = 2$ otherwise.

Then detecting a Hamiltonian cycle in $G$ can be viewed as TSP:

- if $G$ has a Hamiltonian cycle then the cycle is a route of cost exactly $n$.
- if there is a route of cost $n$ then it can't use pairs with cost 2 and so goes through edges of $G$ and hence is a Hamiltonian cycle.

# Lecture 3: Trees and Isomorphism

Dr. Amitabh Trehan

amitabh.trehan@durham.ac.uk

# Trees

We now turn to a special graph class that has many applications in many areas.

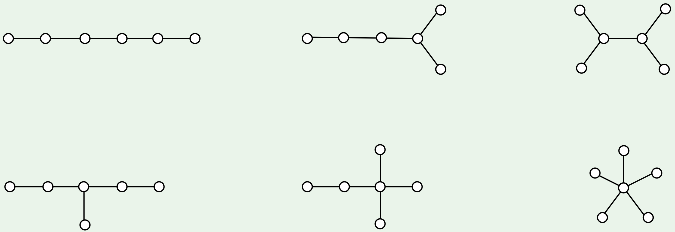
## Definitions

A forest is an acyclic graph, i.e. graph without cycles.

A tree is a connected forest, i.e. a connected acyclic graph.

## Examples

The different trees on 6 vertices are shown below.



We can also consider this as a forest on 36 vertices.

# Spanning trees

A subgraph $G' = (V', E')$ of a graph $G = (V, E)$ is spanning if $V' = V$.

## Theorem

*Every connected graph contains a spanning tree (a spanning subgraph that is a tree).*

## An algorithmic proof.

Let $G$ be a connected graph.

- If $G$ contains no cycles, it is a tree, and hence a spanning tree of itself.
- If $G$ contains a cycle, we can remove one edge from the cycle.
- The new graph is still connected. (Why?)
- Repeating this, we can destroy all cycles and end up with a spanning tree. □

How many repetitions do we need for the above algorithm?

It follows that trees are the smallest connected structures.

Finding minimum-weight spanning trees in edge-weighted graphs is an important task in practice: we will learn fast algorithms for it in a few lectures.

# Leaves in trees

A leaf in a tree is a vertex of degree 1.

## Lemma

*Every tree on at least two vertices contains a leaf.*

## Proof.

By contradiction:

- Assuming that every vertex has degree 0 or at least 2, we will show that the graph is not a tree.
- If a vertex has degree 0, then: the graph (which contains at least two vertices) is not connected, hence not a tree.
- If every vertex has degree at least 2: just start at a vertex, go to one of its neighbours, from there go to another neighbour, etc.
- Since the vertex set is finite, at some stage we encounter a vertex we have already visited.
- This implies that the graph contains a cycle, so is not a tree, contradiction. □

Can a tree have exactly one leaf?

# Edges of trees

How many edges does a tree on $n$ vertices have?

## Theorem

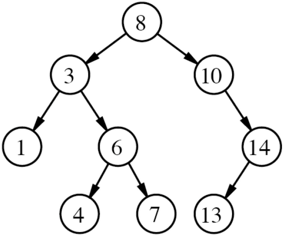
*A connected graph on $n$ vertices is a tree iff it has $n-1$ edges.*

## Proof.

$(\Rightarrow)$. Show, by induction on $n$, that a tree on $n$ vertices has $n-1$ edges.

- For small $n$ the lemma holds: a tree on one vertex has no edges; a tree on two vertices has one edge.
- Suppose each tree on $n-1$ vertices has $n-2$ edges (induction hypothesis).
- Take a tree $T$ on $n$ vertices, for some $n \geq 3$.
- $T$ contains a leaf $v$. Consider the graph $T-v$, it has one vertex less and one edge less than $T$.
- $T-v$ is still connected and (still) acyclic.
- $T-v$ is a tree with $n-1$ vertices, by induction hypothesis it has $n-2$ edges.
- $T$ has one edge more, so $n-1$ edges.

# Rooted trees, children and parents




## Definitions

Let $v$ be a vertex in a rooted tree $T$.

- The neighbours of $v$ in the next level are called the children of $v$.
- the (unique) neighbour of $v$ in the previous level (if $v$ is not the root) is called the parent of $v$.
- If $v$ has no children then it is called a leaf of $T$;
- If $v$ has children, then it is an internal vertex.

# Every tree is a bipartite graph

## Theorem

*Every tree is a bipartite graph.*

## Proof.

We give a direct proof. We can use the known result on unique paths in a tree $T$ to define a bipartition of its vertex set $V(T)$.

- Choose any vertex $v$ and put this vertex in the set $V_1$.
- For every vertex $u \neq v$, there is a unique path from $v$ to $u$ in $T$, consider the length of this path.
- If the length is odd, put $u$ in $V_2$; otherwise put $u$ in $V_1$.
- We have to show that this is a valid bipartition.
- $V_1$ and $V_2$ are disjoint and together make up $V(T)$. (Why?)
- Every edge has end vertices in both $V_1$ and $V_2$. (Why?)
- This completes the proof.

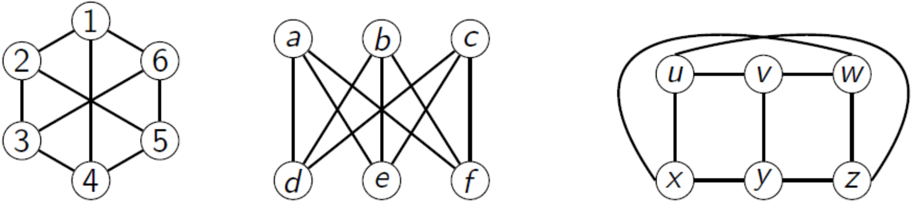
□

# Graph isomorphism

## Definition

Two graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there exists a bijective function $f : V \to V'$ such that for every $u, v \in V$ we have:
$uv \in E$ if and only if $f(u)f(v) \in E'$. Then we write: $G \cong G'$.

Example: which of these graphs are isomorphic?



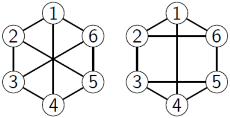

bijective function $f$ for the first and second graph:
$1 \mapsto a,\ 2 \mapsto d,\ 3 \mapsto b,\ 4 \mapsto e,\ 5 \mapsto c,\ 6 \mapsto f$
bijective function $f$ for the second and third graph:
$a \mapsto x,\ b \mapsto v,\ c \mapsto z,\ d \mapsto u,\ e \mapsto w,\ f \mapsto y$

# Graph isomorphism

If $G$ and $G'$ are isomorphic, they shared all their structural characteristics, e.g.:

- number of vertices and edges, degree sequence, (strong) connectivity
- Euler circuit, Hamiltonian Circuit, chromatic number, size of largest independent set, . . .

But none of these alone determines isomorphism:




Not isomorphic:

- although same number of vertices / edges, degree sequence
- Euler circuit, Hamiltonian Circuit: yes
- chromatic number, size of largest independent set: no

All these (and all other characteristics):

- can only be used to show non-isomorphism

# Lecture 4a: Breadth-First Search

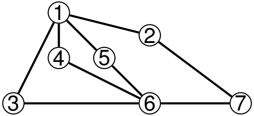
Amitabh Trehan

amitabh.trehan@durham.ac.uk

**Based on the slides of ADS-21/22 by Dr. George Mertzios*

# Graphs: Representations

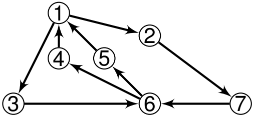
Undirected




1 : 2, 3, 4, 5
2 : 1, 7
3 : 1, 6
4 : 1, 6
5 : 1, 6
6 : 3, 4, 5, 7
7 : 2, 6

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

Directed




1 : 2, 3
2 : 7
3 : 6
4 : 1
5 : 1
6 : 4, 5
7 : 6

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

For each representation ($n$ nodes, $m$ edges; assume undirected graph):

- How much space do we need to store it?
- How long does it take to initialize an empty graph?
- How long does it take to make a copy?
- How long does it take to insert an edge?
- How long does it take to list the vertices adjacent to a vertex $u$?
- How long does it take to find out if the edge $(u, v)$ belongs to $G$?

| | Space | Init | Copy | Insert | List Nbrs | Search e |
|---|---|---|---|---|---|---|
| Edge Array | $m$ | 1 | $m$ | 1 | $m$ | $m$ |
| Adj Matrix | $n^2$ | $n^2$ | $n^2$ | 1 | $n$ | 1 |
| Adj List | $n + m$ | $n$ | $m$ | 1 | $n$ | $deg(u) = O(n)$ |

# Breadth-First Search (Graph Traversal)

- Input: a graph $G = (V, E)$ and a source vertex $s$.
- Aim: to find the distance from $s$ to each of the other vertices in the graph.

  Example: Suppose you want to find the “distance” between you and a specific person $x$ on Facebook. How can you do that?

- Idea: send out a wave from $s$.
  - The wave first hits vertices at distance 1
  - Then the wave hits vertices at distance 2
  - and so on

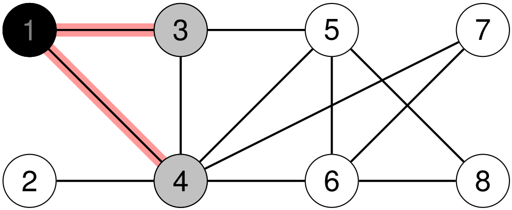
# Breadth-First Search

- BFS maintains a queue that contains vertices that have been discovered but are waiting to be processed.
- BFS colours the vertices:
  - White indicates that a vertex is undiscovered
  - Grey indicates that a vertex is discovered but unprocessed
  - Black indicates that a vertex has been processed.
- The algorithm maintains an array $d$ (distance)
  - $d[s] = 0$, where $s$ is the source vertex;
  - if we discover a new vertex $v$ while processing $u$, we set $d[v] = d[u] + 1$.

# Example

$Q = 3, 4$

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $d$ | 0 | | 1 | 1 | | | | |




- Initialization: source vertex grey, others are white; distance to source is 0; add source to the queue
- while the queue is not empty
  - remove the first vertex $v$ from the queue (why the first one?)
  - add white neighbours of $v$ to queue and colour them grey; distance is 1 greater than to $v$
  - colour $v$ black

# Lecture 4b:Depth-First Search

Amitabh Trehan

amitabh.trehan@durham.ac.uk

**Based on the slides of ADS-21/22 by Dr. George Mertzios*

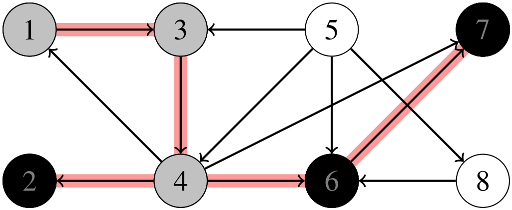
# Depth-first search

- Like BFS, Depth-first search explores the graph (but does not find distances to the source).
- In contrast to BFS, when a vertex is discovered it is immediately explored.
- Two timestamps are recorded for each vertex, $d$ and $f$; the discovery and finish times. We can also record predecessors again.
- Again colours are used:
  - white for undiscovered,
  - grey for discovered but not finished, and
  - black for finished.

# Example

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $d$ | 1 | 4 | 2 | 3 | | 6 | 7 | |
| $f$ | | 5 | | | | 9 | 8 | |

time= 9



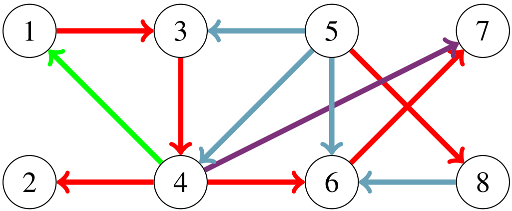

- Initialize: source vertex grey, others white; source discovered at time 1.
- Repeat:
  - Increment the time.
  - If there is a white neighbour of the current vertex, then it is coloured grey and its discovery time noted and it becomes current.
  - Else colour the current vertex black, note its finish time and return to its predecessor (or jump to an undiscovered vertex), or stop.

# Classification of the edges

Once we have obtained a DFS-forest for a graph $G$, we can classify the edges of $G$.

- Tree edges are those edges in the DFS-forest.
- Back edges are edges that join a vertex to an ancestor.
- Forward edges are edges not in the tree that join a vertex to its descendant.
- Cross edges: all other edges.

# Example

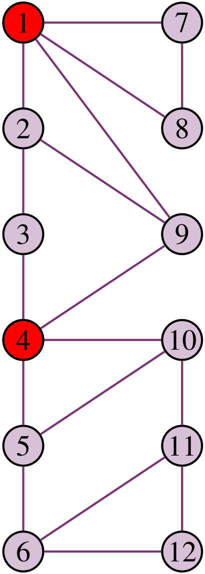



Tree edges

Forward edges

Back edges

Cross edges




- Every edge in an undirected graph is either a tree edge or a back edge.
- A graph is connected if each pair of vertices is joined by a path.
- A cycle is a sequence of edges that start and end at the same vertex.
- An articulation point is a vertex whose removal disconnects the graph.

# Lecture 5a: Minimum Spanning Trees

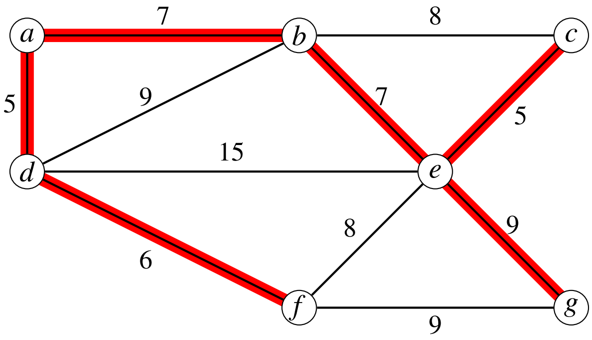
Amitabh Trehan

amitabh.trehan@durham.ac.uk

**Based on the slides of ADS-21/22 by Dr. George Mertzios*

# Connecting the vertices

Input: a graph $G = (V, E)$ with a weight (or a cost) $w(u, v)$ for each edge $(u, v)$.



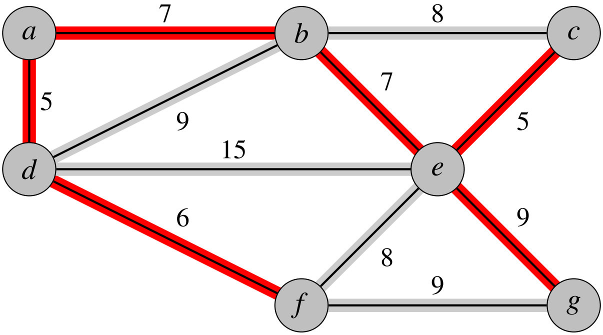

Objective: Choose a subset of the edges that connects the vertices. Find the solution that costs the least.

# Kruskal's Algorithm

1. Sort the edges by weight.
2. Let $A = \emptyset$.
3. Consider edges in increasing order of weight. For each edge $e$, add $e$ to $A$ unless this would create a cycle.

(Running time is $O(E \log V)$.)

# Kruskal's algorithm



# Kruskal's Algorithm: simple implementation

$V$ is the set of vertices, $E$ is the set of edges
$A$ is the empty set (will add edges until it is MST)
sort $E$
**while** $E$ is not empty **do**
    choose $e$ in $E$ with min cost
    **if** $A + e$ contains no cycle **then**
        add $e$ to $A$
    **end if**
**end while**
return A

- Sorting initially takes time $O(E \log E) = O(E \log V)$.
- Iterate through while loop once for each edge.
- Need to check every time for a cycle using, for example, depth-first search — takes time $O(V + E)$.
- Running time $O(E \log V) + O(E(V + E))$

# The Union-Find Data Structure

- Kruskal's algorithm, like many other algorithms in Computer Science, requires a dynamic partition of an $n$-element set $S$ into a collection of disjoint subsets $S_1, S_2, \ldots, S_k$.
- After being initialised as a collection of $n$ one element subsets, we perform union and find operations on the collection.
- The union operation joins two subsets into a single set. The number of such operations is bounded by $n - 1$, since we have $n$ elements in total.

# An example

- As an example, let $S = \{1, 2, 3, 4, 5, 6\}$.
- Then makeset(i) creates the set $\{i\}$ and applying this six times gives:

$$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}$$

- Performing union(1, 4) and union(5, 2) yields:

$$\{1, 4\}, \{5, 2\}, \{3\}, \{6\}$$

- If followed by union(4, 5) and union(3, 6), it yields:

$$\{1, 4, 5, 2\}, \{3, 6\}$$

# Back to Kruskal's Algorithm

- How does the Union-Find data structure help with Kruskal's algorithm?
- Store each vertex as a separate integer (i.e. makeset($x$))
- Each time we want to add an edge $(i, j)$ to the MST, we need to determine if adding edge $(i, j)$ to the MST would create a cycle
- To do this, we simply need to determine if find($i$) = find($j$)
- If so, both vertices $i$ and $j$ are in the same subset and we can't add an edge between them without creating a cycle
- If we can add an edge to the graph, then we perform union($i, j$) to update the data structure

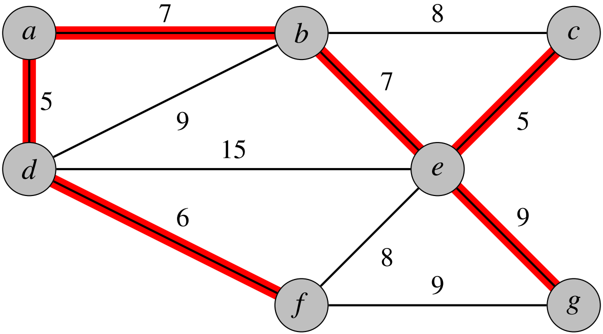
# Prim's Algorithm

1. Let $U = \{u\}$ where $u$ is some vertex chosen arbitrarily.
2. Let $A = \emptyset$.
3. Until $U$ contains all vertices: find the least-weight edge $e$ that joins a vertex $v$ in $U$ to a vertex $w$ not in $U$ and add $e$ to $A$ and $w$ to $U$.

(Running time is $O(V \log V + E)$.)

# Prim's algorithm

Start at $b$:



# Prim's Algorithm: improved implementation

$V$ is the set of vertices
$E$ is the set of edges
$U = \{u\}$
$A$ is the empty set (will add edges until it is MST)
**for** each vertex $v$ except $u$ **do**
    $B(v)$ is the least-weight edge from $v$ to $U$
**end for**
**while** $U \neq V$ **do**
    choose $v$ with minimum cost $B(v)$
    $A = A + e$
    $U = U + v$
    update $B$
**end while**
return A

# Prim's Algorithm

Implement the array using a Priority Queue (using a heap, for example).

- To initialize, all edges considered.
- Iterate through While loop once for each vertex.
- Extracting the minimum cost edge and performing updates take $O(\log V)$ time.
- Running time $O(V \log V + E)$.

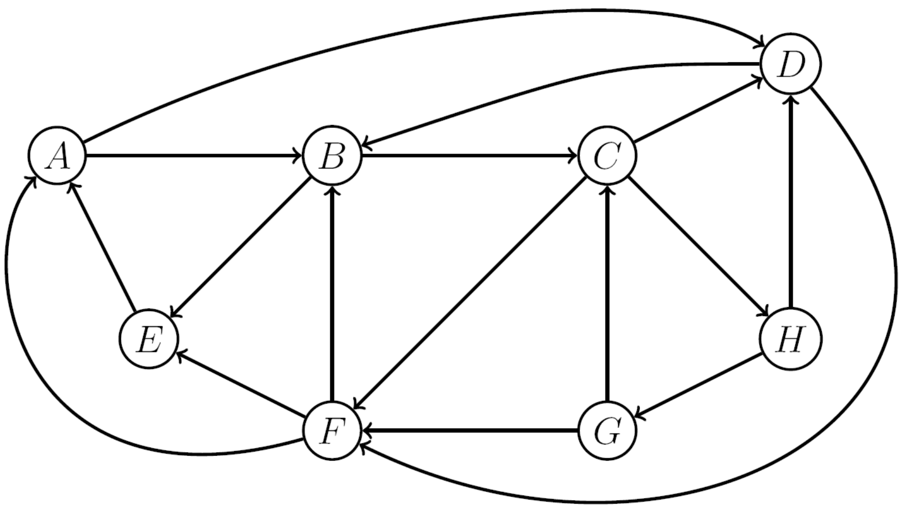
## Section D Graph algorithms (Dr Amitabh Trehan)

### Question 4

(a) From a depth-first search of a directed graph $G$, a depth-first tree $T$ is obtained that contains the edges traversed during the search. Describe how the edges of $G$ can be classified with respect to $T$ as tree, forward, back or cross edges. **[4 Marks]**

**Solution:** *Knowledge*
If an edge is in $T$ it is a tree edge [1 mark]. If an edge joins a vertex to an ancestor (in $T$) it is a back edge [1 mark]; to a descendant a forward edge [1 mark]; other edges are cross edges [1 mark].

(b) Suppose a depth-first search is run on the directed graph below with vertex $A$ as the source. Classify each edge of the graph as a tree, forward, back or cross edge. **[6 Marks]**




**Solution:** *Knowledge,Comprehension,Application*
(1 mark lost for each mistake) tree edges: AB BC CD DF FE CH HG, back edges: EA FA DB FB GC DB, forward edges: AD BE CF, cross edges: HD GF, [6 marks]

(c) Assume it takes constant time to update arrays and colour vertices. Prove the upper bound on the running time for depth-first search is $\mathcal{O}(V + E)$. **[4 Marks]**

**continued**

**Solution:** *Knowledge,Application,Analysis*

Initialisation takes $\mathcal{O}(V)$. [1 mark]

Time spent on updating arrays and colouring vertices is constant for each vertex so $\mathcal{O}(V)$. [1 mark]

Each vertex in each adjacency list is considered once so total is $\mathcal{O}(E)$. [1 mark]

Summing gives $\mathcal{O}(V + E)$. [1 mark]

**this question is continued on the next page**

(d) Here is an attempt to use breadth-first search (BFS) to create an algorithm to decide whether or not, in an undirected graph $G$, there is a cycle that contains a specified pair of vertices $u$ and $v$ and has at most $k$ edges.

i. Use BFS to find a shortest path $P$ from $u$ to $v$ in $G$.

ii. From $G$, delete the edges of the path $P$ to create a new graph $H$.

iii. Use BFS to find a shortest path $Q$ from $v$ to $u$ in $H$.

iv. Let $p$ be the number of edges in $P$. Let $q$ be the number of edges in $Q$. If $p + q \leq k$, then the output is $\textsc{Yes}$; otherwise it is $\textsc{No}$

Explain why this algorithm is not correct. **[4 Marks]**

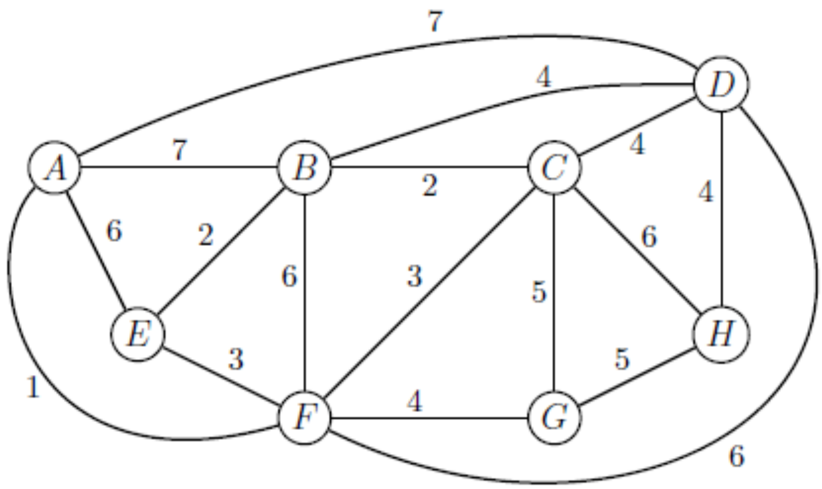
**Solution:** *Knowledge,Application,Analysis*
(Other lines of reasoning possible.) The two paths $P$ and $Q$ might share internal vertices [1 mark]. Thus the union of $P$ and $Q$ is not necessarily a cycle [1 mark] as is implicit in the logic of the iv. [2 marks]. (Graph might not even contain such a cycle (could give easy example with a cutvertex between $u$ and $v$.).

(e) Describe briefly (you do not need to write any pseudocode) a correct algorithm that uses BFS to find the shortest cycle in a directed graph (with no conditions on which vertices the cycle contains). Show that your algorithm has running time $\mathcal{O}(\mathrm{V}^3)$. **[7 Marks]**

**Solution:** *Knowledge,Application,Analysis,Synthesis*
Run BFS from each vertex $u$. [1 mark] At each step check whether there is a back edge to $u$ [1 mark]. The first time one is found this is the shortest cycle including $u$ so stop this instance of BFS [1 mark]. Having found the shortest cycle through each vertex, the shortest cycle in the graph must have been found. [2 marks] BFS is $\mathcal{O}(V^2)$ and it is being used $V$ times so algorithm is $\mathcal{O}(V^3)$. [2 marks]

**END OF PAPER**

Find a minimum spanning tree (MST) of the graph below using Prim's algorithm. State the edges of the tree in the order in which they are added to the MST. **[5 Marks]**




**Knowledge, Comprehension, Application**
various possible answers including: pick $A$ as the first vertex and then edges selected, in order, are $AF, CF, BC, BE, BD, DH, FG$. [5 marks] (1 mark off for each error).

Suppose a depth-first search is run on the directed graph on vertex set $V = \{1, \ldots, 8\}$ given by the adjacency matrix below, with vertex 1 as the source. Classify each edge of the graph as a tree, forward, back or cross edge. **[6 Marks]**

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

**Knowledge, Comprehension, Application**

(knowledge, comprehension, application) tree edges are 17, 76, 18, 84, 42, 23, 35; back edges are 41, 48, 51, 61, 67; forward edges are 25; cross edges are 27, 36, 46, 47 [6 marks]. (1 mark lost for each error)

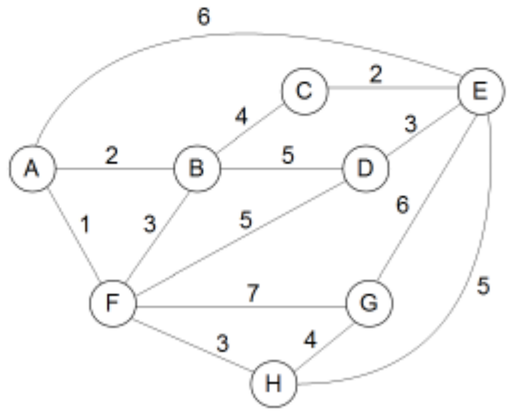
Let $G$ be an undirected graph with weights on the edges. Let $T_1$ and $T_2$ be two distinct minimum spanning trees of $G$. Show that there is an edge $e_1$ in $T_1$ but not $T_2$ and an edge $e_2$ in $T_2$ but not $T_1$ such that $\{T_1 - e_1\} \cup e_2$ is also an MST. **[5 Marks]**

**Knowledge, Comprehension, Application**

Let $e_2$ be the least weight edge in one of $T_1$ and $T_2$ but not the other [1 mark]. Let us suppose it is in $T_2$ Then $T_1 \cup e_2$ contains a cycle [1 mark]. Let $e_1$ be the least weight edge of the cycle not in $T_2$. So $\{T_1 - e_1\} \cup e_2$ is a tree [1 mark] and by the choice of $e_2$ the weight of $e_1$ is at least the weight of $e_2$ [1 mark] so the weight of $\{T_1 - e_1\} \cup e_2$ is at most the weight of $T_1$ and so is also an MST [1 mark].

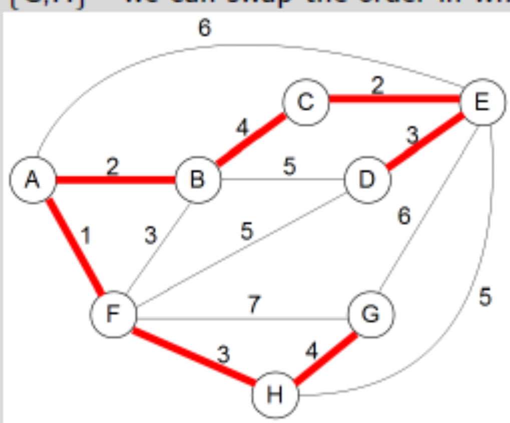
Find a minimum spanning tree of the graph below using Kruskal's algorithm. State the edges of the tree in the order they are added to the MST.

**[5 Marks]**




**Solution:** *Knowledge, Application*

Add the edges in an order such as: {A,F}, {A,B}, {C,E}, {F,H}, {D,E}, {B,C}, {G,H} - we can swap the order in which edges with the same score are added.




2 marks for edge order, 3 for MST. Subtract 1 mark for each mistake.

# Examination Paper

| Examination Session: | Year: | Exam Code: |
|---|---|---|
| May/June | 2024 | COMP1081-WE01 |

**Title: Algorithms and Data Structures**

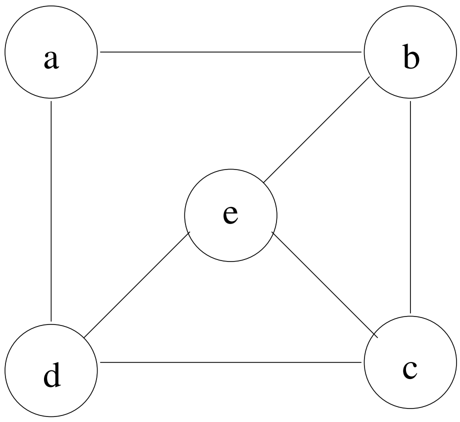
| | | |
|---|---|---|
| Time Allowed: | 2 hours | |
| Additional Material provided: | None | |
| Materials Permitted: | None | |
| Calculators Permitted: | Yes | Models Permitted: Casio fx-83GT range and Casio fx-85GT range |
| Visiting Students may use dictionaries: | Yes | |

| **Instructions to Candidates:** | Answer ALL questions.<br>Students must use the Computer Science answer booklet. |
|---|---|

## Section D Graph algorithms (Dr Amitabh Trehan)

### Question 4

(a) Consider the following graph:

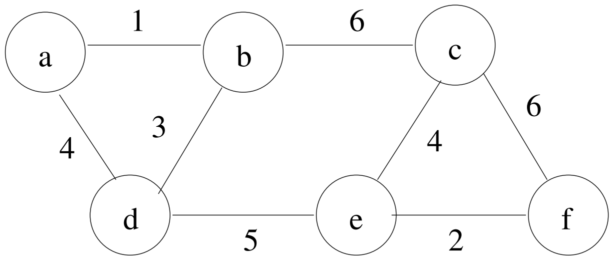



Graph 1

i. State, with justification, if Graph 1 is a **simple graph**. **[2 Marks]**

ii. Graph 1 does not have an Euler cycle. Give the definition of an Euler cycle (or Euler circuit) and give a set of edges which could be added to Graph 1 so that it remains simple and it does have an Euler cycle. **[4 Marks]**

(b) Consider the following graph:




Graph 2

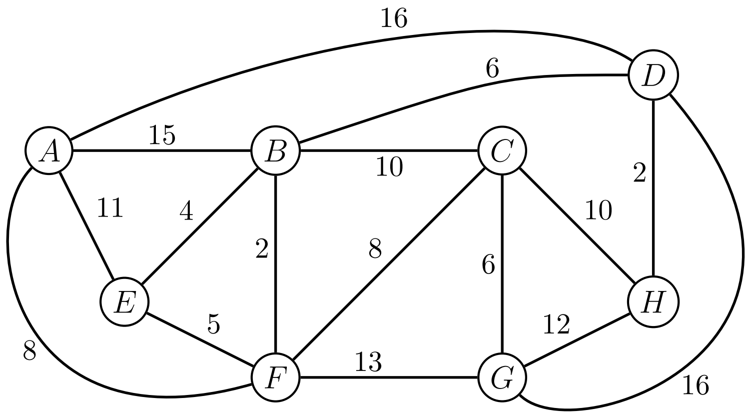
For the following two sets of edges, state with justification whether they are spanning trees and/or minimum spanning trees for Graph 2:

$$E_1 = \{(a, d), (a, b), (b, c), (c, f), (e, f), (e, c)\}$$
$$E_2 = \{(a, b), (e, f), (b, d), (d, e), (c, e)\}$$

**[6 Marks]**

**this question is continued on the next page**

(c) Find a minimum spanning tree of the graph below (Graph 3) using Prim's algorithm starting from node $F$. List the edges of this tree in the order in which they are added to the tree. What is the weight of the constructed tree?




Graph 3

**[6 Marks]**

(d) Let $G = (V, E)$ be an undirected graph such that every edge $e \in E$ has a positive weight $w_e$, and let $T$ be a minimum spanning tree of $G$. Now suppose that we replace every weight $w_e$ by its square $w_e^2$, thereby creating a new instance of the problem with the same graph $G$ but with the new weights. Is $T$ always a minimum spanning tree in this new instance or not? Justify your answer. **[7 Marks]**

## END OF PAPER