Overview

- graph terminology
- data structures for storing graphs
- walk, path, circuit, cycle
- minimum spanning tree

(see Open Data Structures, Chapter 12 for further reading)
Graph Definition

- A graph $G = (V, E)$ consists of:
  - a set of **vertices**, $V$, representing objects in a set
  - a set of **edges**, $E \subseteq V \times V$.

- A vertex is usually represented by a point.
- An edge $(u, v)$ is usually represented by a line segment from $u$ to $v$.

![Graph Diagram](image-url)
Computer Networks: pairs of computers (vertices) joined by a network connection (edge).
Graph Applications

- **World Wide Web:** pairs of web pages (vertices) joined by a hyperlink (edge).
**Social networks:** pairs of users (vertices) joined by a friendship-relation (edge).
Graph Applications

- **Road networks:** pairs of locations (vertices) joined by a road (edge).
Graph Applications

- **Air map**: pairs of cities (vertices) joined by a direct flight (edge).
Undirected vs Directed Graphs

- In **undirected graphs**, there is no direction for edges.
- In **directed graphs**, also called **digraphs**, edges have one-way direction.
  - \((u, v)\) and \((v, u)\) are distinct possible edges between vertices \(v\) and \(u\).
Terminology

- An edge $e = (v, w)$ is **incident** on vertices $v$ and $w$.

- $v$ and $w$ are said to be **adjacent** or **neighbouring** vertices.

- An edge coming from a vertex $u$ into vertex $v$ is called an **in-edge** of $v$.

- Conversely, an edge going from vertex $v$ out to a vertex $u$ is described as an **out-edge** of $v$. 
Weighted Graphs

A numerical value may be assigned to every edge to form a **weighted graph**.
Weighted Graphs

- A numerical value may be assigned to every edge to form a **weighted graph**.
- Edge weight may represent:
  - distance
  - cost
  - speed
  - network traffic
Subgraph

- Given graphs $G = (V, E)$ and $H = (V', E')$, $H$ is a subgraph of $G$ if and only if $V'$ is a subset of $V$ and $E'$ is a subset of $E$.
  - If $V' = V$ then $H$ is a spanning subgraph of $G$.

Is $G$ a subgraph of $H$?
  - We have $V = \{1, 2, 3, 4\}$, $E = \{(1, 2), (2, 4), (1, 3), (3, 4), (2, 3)\}$
  - Also $V' = \{1, 2, 3\}$, and $E' = \{(1, 2), (2, 3)\}$. 
Subgraph

- Given graphs $G = (V, E)$ and $H = (V', E')$, $H$ is a subgraph of $G$ if and only if $V'$ is a subset of $V$ and $E'$ is a subset of $E$.
  - If $V' = V$ then $H$ is a spanning subgraph of $G$.

Is $G$ a subgraph of $H$?

- We have $V = \{1, 2, 3, 4\}$, $E = \{(1, 2), (2, 4), (1, 3), (3, 4), (2, 3)\}$
- Also $V' = \{1, 2, 3\}$, and $E' = \{(1, 2), (2, 3)\}$.
- $H$ is a subgraph of $G$ but since $V \neq V'$, it is not a spanning subgraph.
The **degree** of a vertex $v$ is the total number of edges incident upon $v$.

In case of a directed graph, the **in-degree** of $v$ is the number of in-edges at $v$, and the **out-degree** of $v$ is the number of out-edges at $v$.
The **degree** of a vertex $v$ is the total number of edges incident upon $v$.

In case of a directed graph, the **in-degree** of $v$ is the number of in-edges at $v$, and and the **out-degree** of $v$ is the number of out-edges at $v$.

- $v$ has degree 5, in-degree 2, and out-degree 3.
The **degree** of a vertex $v$ is the total number of edges incident upon $v$.

- In case of a directed graph, the **in-degree** of $v$ is the number of in-edges at $v$, and and the **out-degree** of $v$ is the number of out-edges at $v$.
  - $v$ has degree 5, in-degree 2, and out-degree 3.

The **maximum degree** of a graph $G$, denoted $\Delta(G)$, is defined as the maximum degree amongst all vertices $v \in V$. 

\[ G_1 \quad G_2 \]
The **degree** of a vertex $v$ is the total number of edges incident upon $v$.

In case of a directed graph, the **in-degree** of $v$ is the number of in-edges at $v$, and the **out-degree** of $v$ is the number of out-edges at $v$.

$v$ has degree 5, in-degree 2, and out-degree 3.

The **maximum degree** of a graph $G$, denoted $\Delta(G)$, is defined as the maximum degree amongst all vertices $v \in V$.

$G_1$ has maximum degree 3.
Degree

- The **degree** of a vertex $v$ is the total number of edges incident upon $v$.
  - In case of a directed graph, the **in-degree** of $v$ is the number of in-edges at $v$, and and the **out-degree** of $v$ is the number of out-edges at $v$.
    - $v$ has degree 5, in-degree 2, and out-degree 3.

- The **maximum degree** of a graph $G$, denoted $\Delta(G)$, is defined as the maximum degree amongst all vertices $v \in V$.
  - $G_1$ has maximum degree 3.

- All vertices in a **regular graph** have the same degree (e.g., $G_2$ is regular).
If a graph has $n$ vertices, what is the maximum number of edges it can have?

- This depends on whether self-loops (edges between a vertex and itself) are permitted and whether are directed.
If a graph has $n$ vertices, what is the maximum number of edges it can have?

- This depends on whether self-loops (edges between a vertex and itself) are permitted and whether are directed.
- If there is no self-loop and edges are not directed, there will be \( \binom{n}{2} = n(n-1)/2 \) possible edges.
Size of a Graph

- If a graph has $n$ vertices, what is the maximum number of edges it can have?
  - This depends on whether self-loops (edges between a vertex and itself) are permitted and whether are directed.
  - If there is no self-loop and edges are not directed, there will be $\binom{n}{2} = \frac{n(n-1)}{2}$ possible edges.
Size of a Graph

- If a graph has \( n \) vertices, what is the maximum number of edges it can have?
  - This depends on whether self-loops (edges between a vertex and itself) are permitted and whether are directed.
  - If there is no self-loop and edges are not directed, there will be \( \binom{n}{2} = \frac{n(n - 1)}{2} \) possible edges.
If a graph has \( n \) vertices, what is the maximum number of edges it can have?

- This depends on whether self-loops (edges between a vertex and itself) are permitted and whether are directed.
- If there is no self-loop and edges are not directed, there will be \( \binom{n}{2} = \frac{n(n-1)}{2} \) possible edges.

<table>
<thead>
<tr>
<th></th>
<th>undirected</th>
<th>directed</th>
</tr>
</thead>
<tbody>
<tr>
<td>self-loops</td>
<td>( n(n-1)/2 )</td>
<td>( n(n-1) )</td>
</tr>
<tr>
<td>not permitted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>self-loops</td>
<td>( n(n+1)/2 )</td>
<td>( n^2 )</td>
</tr>
<tr>
<td>permitted</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

COMP 2140 - Data Structures
Data Structures for Graphs

- How can we store the following graph in a data structure?

- The two common data structures for storing a graph are:

```
  V0  V1
   |   |
   V2 V3
   |   |
   V4 V5
```
How can we store the following graph in a data structure?

The two common data structures for storing a graph are:

- adjacency matrix
- adjacency list
Let \( G = (V, E) \) be a graph where \( V = \{ v_0, v_1, \ldots, v_{n-1} \} \).

The adjacency matrix of \( G \) is an \( n \times n \) matrix \( A \) such that

- \( A[i, j] = 1 \) if \( (v_i, v_j) \in E \).
- \( A[i, j] = 0 \) if \( (v_i, v_j) \notin E \).
The adjacency matrix of an undirected graph is **symmetric**.

The adjacency matrix of a directed graph may not be asymmetric.
We represent a weighted graph by storing the weight of edge \((v_i, v_j)\) at \(A[i, j]\).

- We assume all weights are non-zero

\[
A = \begin{bmatrix}
0 & 10 & 0 & 0 & 0 & 0 \\
10 & 0 & 16 & 0 & 0 & 0 \\
0 & 16 & 0 & 5 & 7 & 0 \\
0 & 0 & 5 & 0 & 32 & 0 \\
0 & 0 & 7 & 32 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
public class AdjacencyMatrix {
    private int numVert;
    private int[][] A;  // adjacency matrix

    public AdjacencyMatrix(int newSize) {
        numVert = newSize;
        A = new int[numVert][numVert];
        for (int i = 0 ; i < numVert ; i++)
            for (int j = 0 ; j < numVert ; j++)
                A[i][j] = 0;
    }

    public void addEdge(int u, int v) {
        addEdge(u, v, 1);
    }

    public void addEdge(int u, int v, int weight) {
        if (0 <= u && u < numVert && 0 <= v && v < numVert)
            A[u][v] = weight;
    }
}
Adjacency Matrix Summary

- Let $n$ denote the number of vertices and $m$ be the number of edges.
- **Storing** the matrix takes
Adjacency Matrix Summary

- Let $n$ denote the number of vertices and $m$ be the number of edges.
- **Storing** the matrix takes $O(n^2)$.
- We can check whether an edge (edge-search) between $v_i$ and $v_j$ exists in
Adjacency Matrix Summary

- Let $n$ denote the number of vertices and $m$ be the number of edges.
- **Storing** the matrix takes $O(n^2)$.
- We can check whether an edge (edge-search) between $v_i$ and $v_j$ exists in $O(1)$ time (just check the index $a[i][j]$).
  - Similar time for adding an edge (just set the value of $a[i][j]$ to 1 or another number to indicate weight).

Let \( n \) denote the number of vertices and \( m \) be the number of edges.

**Storing** the matrix takes \( O(n^2) \).

We can check whether an edge (edge-search) between \( v_i \) and \( v_j \) exists in \( O(1) \) time (just check the index \( a[i][j] \)).

- Similar time for adding an edge (just set the value of \( a[i][j] \) to 1 or another number to indicate weight).

We can compute the **indegree** of a vertex \( v_i \) in time...
Let $n$ denote the number of vertices and $m$ be the number of edges.

Storing the matrix takes $O(n^2)$.

We can check whether an edge (edge-search) between $v_i$ and $v_j$ exists in $O(1)$ time (just check the index $a[i][j]$).

Similar time for adding an edge (just set the value of $a[i][j]$ to 1 or another number to indicate weight).

We can compute the indegree of a vertex $v_i$ in time $O(n)$ (just scan the $i$’th column and count non-zero elements).

We can compute the outdegree of a vertex $v_i$ in time
Adjacency Matrix Summary

- Let $n$ denote the number of vertices and $m$ be the number of edges.
- **Storing** the matrix takes $O(n^2)$.
- We can check whether an edge (edge-search) between $v_i$ and $v_j$ exists in $O(1)$ time (just check the index $a[i][j]$).
  - Similar time for adding an edge (just set the value of $a[i][j]$ to 1 or another number to indicate weight).
- We can compute the **indegree** of a vertex $v_i$ in time $O(n)$ (just scan the $i$'th column and count non-zero elements).
- We can compute the **outdegree** of a vertex $v_i$ in time $O(n)$ (just scan the $i$'th row and count non-zero elements).
Adjacency List

- An adjacency matrix requires $O(n^2)$ space, where $n = |V|$.
- For a sparse matrix (when $m$ is small relative to $n$), a data structure that uses less space may be useful.
- **Adjacency List**: use a linked list for each vertex.
Adjacency List in Java

```java
public class Node {
    public int endpoint;
    public int weight;
    public Node next;
    public Node(int newEnd, int newWeight, Node newNext) {
        endpoint = newEnd;
        weight = newWeight;
        next = newNext;
    }
}

public class AdjacencyList {
    private Node[] list;
    private int numVert;
    public AdjacencyList(int newSize) {
        numVert = newSize;
        list = new Node[numVert];
    }
    public void addEdge(int u, int v) {
        addEdge(u, v, 1);
    }
    public void addEdge(int u, int v, int weight) {
        if (0 <= u && u < numVert)
            A[u] = new Node(v, weight, A[u]);
    }
}
```
public class Node {
    public int endpoint;
    public int weight;
    public Node next;
    public Node(int newEnd, int newWeight, Node newNext) {
        endpoint = newEnd;
        weight = newWeight;
        next = newNext;
    }
}

public class AdjacencyList {
    private Node[] list;
    private int numVert;
    public AdjacencyList(int newSize) {
        numVert = newSize;
        list = new Node[numVert];
    }

    public void addEdge(int u, int v) {
        addEdge(u, v, 1);
    }

    public void addEdge(int u, int v, int weight) {
        if (0 <= u && u < numVert)
            A[u] = new Node(v, weight, A[u]);
    }
}
Adjacency List Summary

- An adjacency list requires a space of $O(m + n)$, where $n = |V|$ and $m = |E|$.

There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).

Checking for an edge $(v_i, v_j)$ takes $O(\Delta(G))$; recall that $\Delta(G)$ is the max degree and is at most $n - 1$.

Adding an edge takes the same time of $O(\Delta(G))$: method `addEdge(u, v)` should check whether edge $(u, v)$ is already in the linked-list $A[u]$ to avoid inserting an edge multiple times.

Degree queries:
- Computing the out-degree of $v_i$ takes $O(\Delta(G))$; just scan the list of $v_i$ and report its length.
- Computing the in-degree of $v_i$ takes $O(m + n)$; we need to go through all nodes.
Adjacency List Summary

- An adjacency list requires a space of $O(m + n)$ space, where $n = |V|$ and $m = |E|$.
  - There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).

- Checking for an edge $(v_i, v_j)$ takes $O(\Delta(G))$.
Adjacency List Summary

- An adjacency list requires a **space** of $O(m + n)$ space, where $n = |V|$ and $m = |E|$.
  - There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).
- Checking for an edge $(v_i, v_j)$ takes $O(\Delta(G))$; recall that $\Delta(G)$ is the max degree and is at most $n - 1$.
  - we just need to scan the list associated with one of the vertices.
Adjacency List Summary

- An adjacency list requires a space of $O(m + n)$ space, where $n = |V|$ and $m = |E|$.
  - There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).
- Checking for an edge $(v_i, v_j)$ takes $O(\Delta(G))$; recall that $\Delta(G)$ is the max degree and is at most $n - 1$.
  - we just need to scan the list associated with one of the vertices.
  - adding an edge takes the same time of $O(\Delta(G))$: method `addEdge(u, v)` should check whether edge $(u, v)$ is already in the linked-list $A[u]$ to avoid inserting an edge multiple times.
An adjacency list requires a space of \( O(m + n) \) space, where \( n = |V| \) and \( m = |E| \).

- There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).

Checking for an edge \((v_i, v_j)\) takes \( O(\Delta(G)) \); recall that \( \Delta(G) \) is the max degree and is at most \( n - 1 \).

- we just need to scan the list associated with one of the vertices.
- adding an edge takes the same time of \( O(\Delta(G)) \): method `addEdge(u, v)` should check whether edge \((u, v)\) is already in the linked-list \( A[u] \) to avoid inserting an edge multiple times.
Adjacency List Summary

- An adjacency list requires a space of $O(m + n)$ space, where $n = |V|$ and $m = |E|$.
  - There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).
- Checking for an edge $(v_i, v_j)$ takes $O(\Delta(G))$; recall that $\Delta(G)$ is the max degree and is at most $n - 1$.
  - we just need to scan the list associated with one of the vertices.
  - adding an edge takes the same time of $O(\Delta(G))$: method addEdge($u, v$) should check whether edge ($u, v$) is already in the linked-list $A[u]$ to avoid inserting an edge multiple times.
- Degree queries:
  - Computing the out-degree of $v_i$ takes
Adjacency List Summary

- An adjacency list requires a **space** of $O(m + n)$ space, where $n = |V|$ and $m = |E|$.
  - There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).
- Checking for an edge $(v_i, v_j)$ takes $O(\Delta(G))$; recall that $\Delta(G)$ is the max degree and is at most $n − 1$.
  - we just need to scan the list associated with one of the vertices.
  - adding an edge takes the same time of $O(\Delta(G))$: method `addEdge(u, v)` should check whether edge $(u, v)$ is already in the linked-list $A[u]$ to avoid inserting an edge multiple times.
- Degree queries:
  - Computing the out-degree of $v_i$ takes $O(\Delta(G))$; just scan the list of $v_i$ and report its length.
An adjacency list requires a space of $O(m + n)$ space, where $n = |V|$ and $m = |E|$.

- There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).

Checking for an edge $(v_i, v_j)$ takes $O(\Delta(G))$; recall that $\Delta(G)$ is the max degree and is at most $n - 1$.

- we just need to scan the list associated with one of the vertices.
- adding an edge takes the same time of $O(\Delta(G))$: method `addEdge(u, v)` should check whether edge $(u, v)$ is already in the linked-list $A[u]$ to avoid inserting an edge multiple times.

Degree queries:

- Computing the out-degree of $v_i$ takes $O(\Delta(G))$; just scan the list of $v_i$ and report its length.
- Computing the in-degree of $v_i$ takes
Adjacency List Summary

- An adjacency list requires a space of \( O(m + n) \) space, where \( n = |V| \) and \( m = |E| \).
  - There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).
- Checking for an edge \((v_i, v_j)\) takes \( O(\Delta(G)) \); recall that \( \Delta(G) \) is the max degree and is at most \( n - 1 \).
  - we just need to scan the list associated with one of the vertices.
  - adding an edge takes the same time of \( O(\Delta(G)) \): method `addEdge(u, v)` should check whether edge \((u, v)\) is already in the linked-list \( A[u] \) to avoid inserting an edge multiple times.
- Degree queries:
  - Computing the out-degree of \( v_i \) takes \( O(\Delta(G)) \); just scan the list of \( v_i \) and report its length.
  - Computing the in-degree of \( v_i \) takes \( O(m + n) \); we need to go through all nodes.
Adjacency Matrix vs. Adjacency List

- Recall that $n$ denotes the number of vertices and $m$ denotes the number of edges.
- In general, we use adjacency matrices for dense graphs (with many edges) and adjacency lists for sparse graphs (with relatively a few number of edges).

<table>
<thead>
<tr>
<th></th>
<th>adjacency matrix</th>
<th>adjacency list</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n + m)$</td>
</tr>
<tr>
<td>edge search</td>
<td>$\Theta(1)$</td>
<td>$\Theta(\Delta(G))$</td>
</tr>
<tr>
<td>compute out-degree of $v$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\Delta(G))$</td>
</tr>
<tr>
<td>compute in-degree of $v$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n + m)$</td>
</tr>
</tbody>
</table>
Graph Embedding

- If we are given a pre-determined fixed positioning for the vertices of a graph $G$, then $G$ is an embedded graph.
- Any fixed positioning of the vertices of $G$ is described as an embedding of $G$.
- If there exists a two-dimensional embedding of graph $G$ in the plane in which none of the edges of $G$ cross, then $G$ is planar.
Graph Embedding

- If we are given a pre-determined fixed positioning for the vertices of a graph $G$, then $G$ is an **embedded** graph.
- Any fixed positioning of the vertices of $G$ is described as an **embedding** of $G$.
- If there exists a two-dimensional embedding of graph $G$ in the plane in which none of the edges of $G$ cross, then $G$ is planar.
**Walks, Paths, Circuits, and Cycles**

- **Walk**: A walk from vertex \( v \) to vertex \( w \) is a finite sequence of adjacent vertices of \( G \).
  - \( 2,5,1,2,5,4 \) is a walk.

- **Path**: A path from \( v \) to \( w \) is a walk from \( v \) to \( w \) that does not contain any repeated edges.
  - \( 1,2,4,5 \) is a path (and also a walk).

- **Circuit**: A circuit is a path that begins and ends on the same vertex.
  - \( 1,5,2,4,3,2,1 \) is a circuit (also a path and a walk).

- **Cycle**: A cycle is a circuit that does not contain any repeated vertices.
  - \( 1,2,3,4,5,1 \) is a cycle (and also a circuit, a path, and a walk).
A walk from vertex $v$ to vertex $w$ is a finite sequence of adjacent vertices of $G$. - $2,5,1,2,5,4$ is a walk.

A path from $v$ to $w$ is a walk from $v$ to $w$ that does not contain any repeated edges. - $1,2,4,5$ is a path (and also a walk).
Walks, Paths, Circuits, and Cycles

- A **walk** from vertex $v$ to vertex $w$ is a finite sequence of adjacent vertices of $G$.
  - $2,5,1,2,5,4$ is a walk.

- A **path** from $v$ to $w$ is a walk from $v$ to $w$ that does not contain any repeated edges.
  - $1,2,4,5$ is a path (and also a walk).

- A **circuit** is a path that begins and ends on the same vertex.
  - $1,5,2,4,3,2,1$ is a circuit (also a path and a walk).
A **walk** from vertex \( v \) to vertex \( w \) is a finite sequence of adjacent vertices of \( G \).

A **path** from \( v \) to \( w \) is a walk from \( v \) to \( w \) that does not contain any repeated edges.

A **circuit** is a path that begins and ends on the same vertex.

A **cycle** is a circuit that does not contain any repeated vertices.

- 2,5,1,2,5,4 is a walk.
- 1,2,4,5 is a path (and also a walk).
- 1,5,2,4,3,2,1 is a circuit (also a path and a walk).
- 1,2,3,4,5,1 is a cycle (and also a circuit, a path, and a walk).
A walk from vertex $v$ to vertex $w$ is a finite sequence of adjacent vertices of $G$.

A path from $v$ to $w$ is a walk from $v$ to $w$ that does not contain any repeated edges.

A circuit is a path that begins and ends on the same vertex.

A cycle is a circuit that does not contain any repeated vertices.

A $k$-cycle is a cycle of length $k$.

- $2,5,1,2,5,4$ is a walk.
- $1,2,4,5$ is a path (and also a walk).
- $1,5,2,4,3,2,1$ is a circuit (also a path and a walk).
- $1,2,3,4,5,1$ is a cycle (and also a circuit, a path, and a walk).
More Terminology

- The **length** of a walk, path, circuit, or cycle is the number of edges in the sequence.
- The **distance** between vertices $v$ and $w$ is the length of the shortest path from $v$ to $w$.
- The **diameter** of graph $G$ is the maximum distance between any two vertices $v, w$ in $G$. 
More Terminology

- The **length** of a walk, path, circuit, or cycle is the number of edges in the sequence.
- The **distance** between vertices $v$ and $w$ is the length of the shortest path from $v$ to $w$.
- The **diameter** of graph $G$ is the maximum distance between any two vertices $v, w$ in $G$.

- $b$ and $c$ have distance $d$. 
More Terminology

- The **length** of a walk, path, circuit, or cycle is the number of edges in the sequence.
- The **distance** between vertices $v$ and $w$ is the length of the shortest path from $v$ to $w$.
- The **diameter** of graph $G$ is the maximum distance between any two vertices $v, w$ in $G$.

- $b$ and $c$ have distance 1.
More Terminology

- The **length** of a walk, path, circuit, or cycle is the number of edges in the sequence.
- The **distance** between vertices $v$ and $w$ is the length of the shortest path from $v$ to $w$.
- The **diameter** of graph $G$ is the maximum distance between any two vertices $v, w$ in $G$.

- $b$ and $c$ have distance 1.
- $a$ and $d$ have distance 5.
More Terminology

- The **length** of a walk, path, circuit, or cycle is the number of edges in the sequence.
- The **distance** between vertices $v$ and $w$ is the length of the shortest path from $v$ to $w$.
- The **diameter** of graph $G$ is the maximum distance between any two vertices $v, w$ in $G$.

- $b$ and $c$ have distance 1.
- $a$ and $d$ have distance 5.
More Terminology

- The **length** of a walk, path, circuit, or cycle is the number of edges in the sequence.
- The **distance** between vertices \( v \) and \( w \) is the length of the shortest path from \( v \) to \( w \).
- The **diameter** of graph \( G \) is the maximum distance between any two vertices \( v, w \) in \( G \).

- \( b \) and \( c \) have distance 1.
- \( a \) and \( d \) have distance 5.
- \( G \) has diameter 2.
Connected Graphs

- Two vertices $v$ and $w$ are **connected** iff there is a path from $v$ to $w$.
- Graph $G$ is connected iff any two vertices, $v$, $w$ in $G$ are connected.

$G_1$ is connected and $G_2$ is not connected.
Connected Graphs

- Two vertices $v$ and $w$ are connected iff there is a path from $v$ to $w$.
- Graph $G$ is connected iff any two vertices, $v$, $w$ in $G$ are connected.
- Here $G_1$ is connected and $G_2$ is not connected.
Bipartite Graphs

A graph $G = (V, E)$ is **bipartite** if there exists a partition of its vertices, $V = V_1 \cup V_2$, such that:

- $V_1 \cap V_2 = \emptyset$, and
- every edge $(v_1, v_2) \in E$ has one endpoint in each partition: $v_1 \in V_1$ and $v_2 \in V_2$ or $v_1 \in V_2$ and $v_2 \in V_1$. 

![Diagram of a bipartite graph with two partitions and an example graph](image)
Trees

- An undirected graph $T$ is a **tree** if $T$ is connected and $T$ does not contain any cycles.
  - In a **rooted tree**, one vertex is distinguished from the others and called the root.
- An undirected graph $F$ is a **forest** if $F$ does not contain any cycles. $F$ is a set of trees.
A **spanning tree** for a graph $G$ is a spanning subgraph of $G$ that is a tree.

- Every connected graph has a spanning tree.
- Any spanning tree for a graph $G = (V, E)$ has $|V|$ vertices.
- Any spanning tree for a graph $G = (V, E)$ has $|V| - 1$ edges.
Spanning Trees Application

- Your employer has a contract to provide high-speed internet to an island.
- Each client must be connected to the network while minimizing the total cost of building the network.
- You are provided cost estimates for various possible links in the network.
A minimum spanning tree of a weighted graph $G$ is a spanning tree of $G$ that has the least possible total weight compared to all other spanning trees of $G$. 
A **minimum spanning tree** of a weighted graph $G$ is a spanning tree of $G$ that has the least possible total weight compared to all other spanning trees of $G$.

- If two or more edges have equal weight in a graph $G$, then $G$ may have more than one unique minimum spanning tree.
More Minimum Spanning Tree Example

- It is not always easy to derive a minimum spanning tree 'with eyes'.

![Minimum Spanning Tree](image)
More Minimum Spanning Tree Example

- It is not always easy to derive a minimum spanning tree ‘with eyes’.
- Two efficient algorithms for finding a minimum spanning tree:
  - Kruskal’s algorithm
  - Prim’s algorithm
Kruskal’s MST algorithm

- Initialize $T$ to be $\emptyset$.
- Sort edges in the non-decreasing of their weights and process them one by one.
- If an edge $e$ does not form a cycle in MST, add it to MST.
Kruskal's MST algorithm

- Initialize $T$ to be $\emptyset$.
- Sort edges in the non-decreasing of their weights and process them one by one.
- If an edge $e$ does not form a cycle in MST, add it to MST.

The time complexity of the Kruskal's algorithm is defined by the sorting of edges. Kruskal's algorithm takes $O(m \log m)$ for a graph of $m$ edges.

Note that $O(m \log m) = O(m \log n)$ (why?)
Kruskal’s MST algorithm

- Initialize $T$ to be $\emptyset$.
- Sort edges in the non-decreasing of their weights and process them one by one.
- If an edge $e$ does not form a cycle in MST, add it to MST.
  - Maintain MST’s connected component as disjoint sets of vertices
  - $e$ does not form a cycle iff its endpoints are in different components

The time complexity of the Kruskal’s algorithm is defined by the sorting of edges: $O(m \log m)$ for a graph of $m$ edges.

Note that $O(m \log m) = O(m \log n)$ (why?)
Kruskal’s MST algorithm

- Initialize $T$ to be $\emptyset$.
- Sort edges in the non-decreasing of their weights and process them one by one.
- If an edge $e$ does not form a cycle in MST, add it to MST.
  - Maintain MST’s connected component as disjoint sets of vertices
  - $e$ does not form a cycle iff its endpoints are in different components

![Graph with weights and MST edges highlighted]
Kruskal’s MST algorithm

- Initialize $T$ to be $\emptyset$.
- Sort edges in the non-decreasing of their weights and process them one by one.
- If an edge $e$ does not form a cycle in MST, add it to MST.
  - Maintain MST’s connected component as disjoint sets of vertices
  - $e$ does not form a cycle iff its endpoints are in different components
Kruskal’s MST algorithm

- Initialize $T$ to be $\emptyset$.
- Sort edges in the non-decreasing of their weights and process them one by one.
- If an edge $e$ does not form a cycle in MST, add it to MST.
  - Maintain MST’s connected component as disjoint sets of vertices
  - $e$ does not form a cycle iff its endpoints are in different components

The time complexity of the Kruskal’s algorithm is defined by the sorting of edges.
Kruskal’s algorithm takes $O(m \log n)$ for a graph of $m$ edges.

Note that $O(m \log m) = O(m \log n)$ (why?)
Kruskal’s MST algorithm

- Initialize $T$ to be $\emptyset$.
- Sort edges in the non-decreasing of their weights and process them one by one.
- If an edge $e$ does not form a cycle in MST, add it to MST.
  - Maintain MST’s connected component as disjoint sets of vertices
  - $e$ does not form a cycle iff its endpoints are in different components

The time complexity of the Kruskal’s algorithm is defined by the sorting of edges. Kruskal’s algorithm takes $O(m \log m)$ for a graph of $m$ edges.

Note that $O(m \log m) = O(m \log n)$ (why?)
Kruskal’s MST algorithm

- Initialize $T$ to be $\emptyset$.
- Sort edges in the non-decreasing of their weights and process them one by one.
- If an edge $e$ does not form a cycle in MST, add it to MST.
  - Maintain MST’s connected component as disjoint sets of vertices
  - $e$ does not form a cycle iff its endpoints are in different components
Kruskal’s MST algorithm

- Initialize $T$ to be $\emptyset$.
- Sort edges in the non-decreasing of their weights and process them one by one.
- If an edge $e$ does not form a cycle in MST, add it to MST.
  - Maintain MST’s connected component as disjoint sets of vertices
  - $e$ does not form a cycle iff its endpoints are in different components

The time complexity of the Kruskal’s algorithm is defined by the sorting of edges.

Kruskal’s algorithm takes $O(m \log m)$ for a graph of $m$ edges.

Note that $O(m \log m) = O(m \log n)$ (why?)
Kruskal’s MST algorithm

- Initialize \( T \) to be \( \Phi \).
- Sort edges in the non-decreasing of their weights and process them one by one.
- If an edge \( e \) does not form a cycle in MST, add it to MST.
  - Maintain MST’s connected component as disjoint sets of vertices
  - \( e \) does not form a cycle iff its endpoints are in different components
- The time complexity of the Kruskal’s algorithm is defined by the sorting of edges
Kruskal’s MST algorithm

- Initialize \( T \) to be \( \Phi \).
- Sort edges in the non-decreasing of their weights and process them one by one.
- If an edge \( e \) does not form a cycle in MST, add it to MST.
  - Maintain MST’s connected component as disjoint sets of vertices
  - \( e \) does not form a cycle iff its endpoints are in different components
- The time complexity of the Kruskal’s algorithm is defined by the sorting of edges
  - Kruskal’s algorithm takes \( O(m \log m) \) for a graph of \( m \) edges.
Kruskal’s MST algorithm

- Initialize $T$ to be $\emptyset$.
- Sort edges in the non-decreasing of their weights and process them one by one.
- If an edge $e$ does not form a cycle in MST, add it to MST.
  - Maintain MST’s connected component as disjoint sets of vertices
  - $e$ does not form a cycle iff its endpoints are in different components
- The time complexity of the Kruskal’s algorithm is defined by the sorting of edges
  - **Kruskal’s algorithm takes $O(m \log m)$ for a graph of $m$ edges.**
    - Note that $O(m \log m) = O(m \log n)$ (why?)
Prim’s algorithm

- Initialize: let $T = \{ \text{an edge in the graph with minimum weight} \}$
- Repeat $n - 2$ times:
  - $e = \text{an edge in } G \text{ of minimum weight that has one endpoint in } T$
    and one endpoint outside $T$
  - $T = T = \{ e \}$
Prim’s algorithm

- Initialize: let $T = \{ \text{an edge in the graph with minimum weight} \}$
- Repeat $n - 2$ times:
  - $e = \text{an edge in } G \text{ of minimum weight that has one endpoint in } T$ and one endpoint outside $T$
  - $T = T = \{ e \}$
Prim’s algorithm

- Initialize: let $T = \{ \text{an edge in the graph with minimum weight} \}$
- Repeat $n - 2$ times:
  - $e = \text{an edge in } G \text{ of minimum weight that has one endpoint in } T$
  - and one endpoint outside $T$
  - $T = T = \{ e \}$
Prim’s algorithm

- Initialize: let $T = \{ \text{an edge in the graph with minimum weight} \}$
- Repeat $n - 2$ times:
  - $e = \text{an edge in } G \text{ of minimum weight that has one endpoint in } T$ and one endpoint outside $T$
  - $T = T = \{e\}$
Prim’s algorithm

- Initialize: let $T = \{ \text{an edge in the graph with minimum weight} \}$
- Repeat $n - 2$ times:
  - $e = \text{an edge in } G \text{ of minimum weight that has one endpoint in } T \text{ and one endpoint outside } T$
  - $T = T \cup \{ e \}$
Prim’s algorithm

- Initialize: let \( T = \{ \text{an edge in the graph with minimum weight} \} \)
- Repeat \( n - 2 \) times:
  - \( e = \text{an edge in } G \text{ of minimum weight that has one endpoint in } T \) and one endpoint outside \( T \)
  - \( T = T = \{ e \} \)
Prim’s algorithm

- Initialize: let $T = \{\text{an edge in the graph with minimum weight}\}$
- Repeat $n - 2$ times:
  - $e = \text{an edge in } G \text{ of minimum weight that has one endpoint in } T$ and one endpoint outside $T$
  - $T = T + \{e\}$
Prim’s algorithm

- Initialize: let $T = \{\text{an edge in the graph with minimum weight}\}$
- Repeat $n - 2$ times:
  - $e = \text{an edge in } G \text{ of minimum weight that has one endpoint in } T$
    and one endpoint outside $T$
  - $T = T = \{e\}$
Prim’s algorithm

- Initialize: let \( T = \{ \text{an edge in the graph with minimum weight} \} \)
- Repeat \( n - 2 \) times:
  - \( e = \text{an edge in } G \text{ of minimum weight that has one endpoint in } T \) and one endpoint outside \( T \)
  - \( T = T = \{ e \} \)
Prim’s algorithm

- Initialize: let \( T = \{ \text{an edge in the graph with minimum weight} \} \)

- Repeat \( n - 2 \) times:
  - \( e = \text{an edge in } G \text{ of minimum weight that has one endpoint in } T \) and one endpoint outside \( T \)
  - \( T = T = \{ e \} \)
Prim’s algorithm Implementation

How to implement the Prim’s algorithm?

- Let $T$ be the $\{e\}$ where $e$ is the edge with min-weight
- Insert edges incident to endpoints of $e$ to an initially empty min-heap $H$
- Repeatedly extractMax (to get the next edge $e'$), and insert edges incident to endpoints of $e'$ to $H$. 
How to implement the Prim’s algorithm?

- Let $T$ be the $\{e\}$ where $e$ is the edge with min-weight
- Insert edges incident to endpoints of $e$ to an initially empty min-heap $H$
- Repeatedly extractMax (to get the next edge $e'$), and insert edges incident to endpoints of $e'$ to $H$. 
Prim’s algorithm Implementation

How to implement the Prim’s algorithm?

- Let $T$ be the \{e\} where $e$ is the edge with min-weight
- Insert edges incident to endpoints of $e$ to an initially empty min-heap $H$
- Repeatedly extractMax (to get the next edge $e'$), and insert edges incident to endpoints of $e'$ to $H$.  

![Graph example with prim's algorithm visualization]
How to implement the Prim’s algorithm?

- Let $T$ be the $\{e\}$ where $e$ is the edge with min-weight
- Insert edges incident to endpoints of $e$ to an initially empty min-heap $H$
- Repeatedly extractMax (to get the next edge $e'$), and insert edges incident to endpoints of $e'$ to $H$. 

![Graph with weights and Prim's algorithm implementation diagram]
How to implement the Prim’s algorithm?

- Let $T$ be the \{e\} where $e$ is the edge with min-weight
- Insert edges incident to endpoints of $e$ to an initially empty min-heap $H$
- Repeatedly extractMax (to get the next edge $e'$), and insert edges incident to endpoints of $e'$ to $H$. 
**Prim’s algorithm Implementation**

- How to implement the Prim’s algorithm?
  - Let $T$ be the $\{e\}$ where $e$ is the edge with min-weight
  - Insert edges incident to endpoints of $e$ to an initially empty **min-heap** $H$
  - Repeatedly **extractMax** (to get the next edge $e’$), and insert edges incident to endpoints of $e’$ to $H$. 

---

![Graph diagram](image)
How to implement the Prim’s algorithm?

- Let $T$ be the $\{e\}$ where $e$ is the edge with min-weight
- Insert edges incident to endpoints of $e$ to an initially empty min-heap $H$
- Repeatedly extractMax (to get the next edge $e'$), and insert edges incident to endpoints of $e'$ to $H$. 
How to implement the Prim’s algorithm?

- Let $T$ be the $\{e\}$ where $e$ is the edge with min-weight
- Insert edges incident to endpoints of $e$ to an initially empty min-heap $H$
- Repeatedly extractMax (to get the next edge $e'$), and insert edges incident to endpoints of $e'$ to $H$. 
How to implement the Prim’s algorithm?

- Let \( T \) be the \( \{e\} \) where \( e \) is the edge with min-weight.
- Insert edges incident to endpoints of \( e \) to an initially empty min-heap \( H \).
- Repeatedly extractMax (to get the next edge \( e' \)), and insert edges incident to endpoints of \( e' \) to \( H \).
Prim’s algorithm Implementation

- How to implement the Prim’s algorithm?
  - Let $T$ be the \{e\} where $e$ is the edge with min-weight
  - Insert edges incident to endpoints of $e$ to an initially empty min-heap $H$
  - Repeatedly \texttt{extractMax} (to get the next edge $e'$), and insert edges incident to endpoints of $e'$ to $H$. 

![Graph](image-url)
Prim’s algorithm Implementation

How to implement the Prim’s algorithm?

- Let $T$ be the $\{e\}$ where $e$ is the edge with min-weight
- Insert edges incident to endpoints of $e$ to an initially empty min-heap $H$
- Repeatedly extractMax (to get the next edge $e'$), and insert edges incident to endpoints of $e'$ to $H$. 
Prim’s algorithm Implementation

How to implement the Prim’s algorithm?

- Let $T$ be the $\{e\}$ where $e$ is the edge with min-weight
- Insert edges incident to endpoints of $e$ to an initially empty min-heap $H$
- Repeatedly extractMax (to get the next edge $e'$), and insert edges incident to endpoints of $e'$ to $H$. 
Prim’s Algorithm Running Time

- Each edge is inserted at most once and deleted at most once from the heap.

- At any given time, there are at most $m = |E| = O(n^2)$ edges in the heap.
  - Insert and ExtractMax take $O(\log m) = O(\log(n^2)) = O(\log n)$ time.

- For all edges, we incur a cost of at most $O(m \log n)$. 

Theorem

Both Kruskal and Prim algorithms for finding minimum spanning tree take $O(m \log n)$ for a graph with $n$ vertices and $m$ edges.
Prim’s Algorithm Running Time

- Each edge is inserted at most once and deleted at most once from the heap.
- At any given time, there are at most $m = |E| = O(n^2)$ edges in the heap.
  - Insert and ExtractMax take $O(\log m) = O(\log(n^2)) = O(\log n)$ time.
- For all edges, we incur a cost of at most $O(m \log n)$.

**Theorem**

*Both Kruskal and Prim algorithms for finding minimum spanning tree take $O(m \log n)$ for a graph with $n$ vertices and $m$ edges.*
Graph Coloring

- A graph is **coloured** if each vertex has been assigned a colour such that adjacent vertices have different colours.
Graph Coloring

- A graph is **coloured** if each vertex has been assigned a colour such that adjacent vertices have different colours.
- Clearly, we can color a graph using $n$ colors.
A graph is **coloured** if each vertex has been assigned a colour such that adjacent vertices have different colours.

Clearly, we can color a graph using \( n \) colors.

But it is desirable to use as few colors as possible.
Graph Coloring Example

Four students are taking the following courses:

- Keith: chemistry, English
- Ron: physics, English
- Mick: Spanish, computer science, math
- Charlie: Spanish, English

What is the minimum number of time slots required to schedule final exams such that no student has two simultaneous exams?
Graph Coloring Example

Four students are taking the following courses:

- Keith: chemistry, English
- Ron: physics, English
- Mick: Spanish, computer science, math
- Charlie: Spanish, English

What is the minimum number of time slots required to schedule final exams such that no student has two simultaneous exams?

Create a graph. Let courses be vertices. Add edge \((u, v)\) if some student is taking both course \(u\) and course \(v\).
Graph Coloring Example

- Four students are taking the following courses:
  - Keith: chemistry, English
  - Ron: physics, English
  - Mick: Spanish, computer science, math
  - Charlie: Spanish, English

- What is the minimum number of time slots required to schedule final exams such that no student has two simultaneous exams?
- Create a graph. Let courses be vertices. Add edge \((u, v)\) if some student is taking both course \(u\) and course \(v\).
- Colour this graph with as few colours as possible: here 3 colours is required \(\rightarrow\) at least 3 timeslots are needed.
Graph Coloring in Planar Graphs

- **Four Colour Theorem**: Every planar graph can be coloured with 4 colors.

![Graph Coloring Example](image)
Graph Coloring in Planar Graphs

- **Four Colour Theorem**: Every planar graph can be coloured with 4 colors.
  - Alternatively, a map can be colored using 4 colors.
Graph Coloring in Planar Graphs

- **Four Colour Theorem**: Every planar graph can be coloured with 4 colors.
  - Alternatively, a map can be colored using 4 colors.
  - It took about a century to prove it.
  - The first major theorem proved by a computer.
Graph Coloring in Bipartite Graphs

- How many colors we need to color a bipartite graph?

![Graph Coloring in Bipartite Graphs](image)
How many colors we need to color a bipartite graph?

Only 2 colors are sufficient!
Graph Coloring Summary

- Graph colouring is a difficult problem.

- In fact, given an arbitrary $n$-vertex graph as input, no algorithm is known that can colour the vertices of the graph with the minimum number of colours in running time that is polynomial with respect to $n$.

- That is, given an arbitrary graph $G = (V, E)$ as input, no fixed $k$ is known such that there exists an algorithm for colouring graph $G$ in time $O(n^k)$, where $n = |V|$.

- Graph colouring is an example of a problem that is NP-complete.
Graph Isomorphism

Let \( G \) and \( G' \) be graphs with vertex sets \( V \) and \( V' \) and edge sets \( E \) and \( E' \). \( G \) is **isomorphic** to \( G' \) iff there exists a bijective function \( f : V \rightarrow V' \) such that:

\[
\forall u, v \in V, (u, v) \in E \leftrightarrow (f(u), f(v)) \in E'
\]
Graph Isomorphism

Are these graphs isomorphic?

\[
\begin{align*}
&f(8) = i, \\
&f(d) = 9, \\
&f(a) = 1, \\
&f(b) = 5, \\
&f(c) = 7, \\
&f(e) = 2, \\
&f(f) = 6, \\
&f(g) = 3, \\
&f(h) = 4;
\end{align*}
\]
Graph Isomorphism

Are these graphs isomorphic?

yes: $f(8) = i, f(d) = 9, f(a) = 1, f(b) = 5, f(c) = 7, f(e) = 2, f(f) = 6, f(g) = 3, f(h) = 4;$
The Ending

Observation

You should aim for the stars - and hopefully avoid ending up in the clouds! Roxanne McKee

Template for final will be posted. If any thing in the slides is not clear, ask me to explain it on Piazza.
Observation

You should aim for the stars - and hopefully avoid ending up in the clouds!  

Roxanne McKee

- Template for final will be posted. If any thing in the slides is not clear, ask me to explain it on Piazza.

- Your feedback is appreciated; if something can be improved (which is 100 percent the case), let me know.
The Ending

Observation

You should aim for the stars - and hopefully avoid ending up in the clouds!  

Roxanne McKee

- Template for final will be posted. If any thing in the slides is not clear, ask me to explain it on Piazza.
- Your feedback is appreciated; if something can be improved (which is 100 percent the case), let me know.
- I hope to see you next year; possibly in COMP 3170.