COMP2521 24T1
Graphs (I)
Introduction to Graphs

Kevin Luxa
cs2521@cse.unsw.edu.au

graph fundamentals
graph adt
graph representations
Graph Fundamentals
Up to this point, we’ve seen a few collection types...

- **lists**: a *linear* sequence of items
  each node is connected to its next node

- **trees**: a *branched* hierarchy of items
  each node is connected to its child node(s)

what if we want something more general?
each node is connected to arbitrarily many nodes
Many applications need to model relationships between items.

... on a map: cities, connected by roads
... on the Web: pages, connected by hyperlinks
... in a game: states, connected by legal moves
... in a social network: people, connected by friendships
... in scheduling: tasks, connected by constraints
... in circuits: components, connected by traces
... in networking: computers, connected by cables
... in programs: functions, connected by calls
... etc. etc. etc.
A graph is a data structure consisting of:
- A set of vertices \( V \)
  - Also called nodes
- A set of edges \( E \) between pairs of vertices

\[
V = \{v_1, v_2, v_3, v_4\}
\]

\[
E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_3, v_4)\}
\]
Vertices are distinguished by a unique identifier.

- In this course, usually an integer between 0 and $|V| - 1$

Edges may be (optionally) directed, weighted and/or labelled.
Example: Australian cities and roads

The diagram represents a graph with cities as nodes (Perth (PER), Adelaide (ADL), Melbourne (MEL), Darwin (DAR), Brisbane (BRI), Sydney (SYD), and Canberra (CAN)) and roads as edges with distances indicated.
Questions we could answer with a graph:

- Is there a way to get from $A$ to $B$?
- What is the best way to get from $A$ to $B$?
- In general, what vertices can we reach from $A$?
- Is there a path that lets me visit all vertices?
- Can we form a tree linking all vertices?
- Are two graphs “equivalent”?

Graph problems are generally more complex to solve than linked list problems:

- Items are not ordered
- Graphs may contain cycles
- Concrete representation is more complex
Graphs can be a combination of these types:

- undirected or directed
- unweighted or weighted
- without loops or with loops
- non-multigraph or multigraph

... and others ...
In an undirected graph, edges do not have direction.
In a directed graph or digraph, each edge has a direction.

undirected graph
directed graph
In a **weighted graph**, each edge has an associated weight. For example, road maps, networks.

![unweighted graph](image1)

![weighted graph](image2)
A loop is an edge from a vertex to itself.

Depending on the context, a graph may or may not be allowed to have loops.
In a **multigraph**, multiple edges are allowed between two vertices. For example, call graphs, maps.

Multigraphs will not be considered in this course.
A **simple graph** is an undirected graph with no loops and no multiple edges.

For now, we will only consider simple graphs.
Question:

For a simple graph with $V$ vertices, what is the maximum possible number of edges?
Question:
For a simple graph with $V$ vertices, what is the maximum possible number of edges?

\[ E = 0 \]

\[ E = V(V-1)/2 \]

Note on notation:
The number of vertices $|V|$ and the number of edges $|E|$ are normally written as $V$ and $E$ for simplicity.
Two vertices $v$ and $w$ are adjacent if an edge $e := (v, w)$ connects them; we say $e$ is incident on $v$ and $w$.

The degree of a vertex $v$ ($\deg(v)$) is the number of edges incident on $v$. 

\begin{align*}
\deg(0) &= 2 \\
\deg(1) &= 3 \\
\deg(2) &= 2 \\
\deg(3) &= 1 \\
\deg(4) &= 4
\end{align*}
The ratio $E:V$ can vary considerably.

If $E$ is closer to $V^2$, the graph is **dense**.
If $E$ is closer to $V$, the graph is **sparse**.

Knowing whether a graph is dense or sparse will affect our choice of representation and algorithms.
A **path** is a sequence of vertices where each vertex has an edge to the next in the sequence.

A path is **simple** if it has no repeating vertices.

A **cycle** is a path where only the first and last vertices are the same:
- 0-1-2-0
- 1-2-3-1
- 0-1-3-2-0
A **complete** graph is a graph where every vertex is connected to every other vertex via an edge.

In a complete graph, \( E = \frac{1}{2} V(V - 1) \).

\[ K_3 \quad K_5 \quad K_6 \]
A connected graph is a graph where there is a path from every vertex to every other vertex.

Connected graph

Disconnected graph
A **tree** is a connected graph with no cycles.

A tree has exactly one path between each pair of vertices.
A subgraph of a graph $G$ is a graph that contains a subset of the vertices of $G$ and a subset of the edges between these vertices.
A connected component is a maximally connected subgraph.

A connected graph has one connected component — the graph itself. A disconnected graph has two or more connected components.
A spanning tree of a graph $G$ is a subgraph that contains all the vertices of $G$ and is a single tree.

Spanning trees only exist for connected graphs.
A **spanning forest** of a graph $G$ is a subgraph that contains all the vertices of $G$ and contains one tree for each connected component.
A **clique** is a complete subgraph.

A clique is non-trivial if it has 3 or more vertices.
Graph ADT
What do we need to represent?
What operations do we need to support?
What do we need to represent?
A set of vertices $V := \{v_1, \ldots, v_n\}$
A set of edges $E := \{(v, w) \mid v, w \in V\}$

What operations do we need to support?
create/destroy graph
add/remove edges
get #vertices, #edges
check if an edge exists
Graph ADT Operations

**create/destroy**
- create a graph
- free memory allocated to graph

**query**
- get number of vertices
- get number of edges
- check if an edge exists

**update**
- add edge
- remove edge

We will extend this ADT with more complex operations later.
typedef struct graph *Graph;

// vertices denoted by integers 0..V-1
typedef int Vertex;

/** Creates a new graph with nV vertices */
Graph GraphNew(int nV);

/** Frees all memory allocated to a graph */
void GraphFree(Graph g);
A Graph ADT

"Graph.h" - Operations to Query

/** Returns the number of vertices in a graph */
int GraphNumVertices(Graph g);

/** Returns the number of edges in a graph */
int GraphNumEdges(Graph g);

/** Returns true if there is an edge between the given vertices and false otherwise */
bool GraphIsAdjacent(Graph g, Vertex v, Vertex w);
/** Inserts an edge into a graph */
void GraphInsertEdge(Graph g, Vertex v, Vertex w);

/** Removes an edge from a graph */
void GraphRemoveEdge(Graph g, Vertex v, Vertex w);
Graph Representations
Graph Representations

3 main graph representations:

**Adjacency Matrix**
Edges defined by presence value in $V \times V$ matrix

**Adjacency List**
Edges defined by entries in array of $V$ lists

**Array of Edges**
Explicit representation of edges as $(v, w)$ pairs

We’ll consider these representations for *unweighted, undirected* graphs.
A $V \times V$ matrix; each cell represents an edge.

\[
\begin{bmatrix}
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
1 & 1 & 1 & 0
\end{bmatrix}
\]

undirected

\[
\begin{bmatrix}
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\]

directed
Graphs
Graph ADT
Graph Reps
Adjacency Matrix
Adjacency List
Array of Edges
Summary

```
struct graph {
    int nV;
    int nE;
    bool **edges;
};
```

```
graph
```

```
edges
nV  4
nE  4
```

```
[0] 0 1 0 1
[1] 1 0 0 1
[2] 0 0 0 1
[3] 1 1 1 0
```

```
0 1 2 3
```

```
0->1, 0->2, 1->2, 1->3
```

```
0 1 2 3
```

```
0->1, 0->2, 1->2, 1->3
```

```
0 1 2 3
```

```
0->1, 0->2, 1->2, 1->3
```

```
0 1 2 3
```

```
0->1, 0->2, 1->2, 1->3
```

```
0 1 2 3
```

```
0->1, 0->2, 1->2, 1->3
```

```
0 1 2 3
```

```
0->1, 0->2, 1->2, 1->3
```

```
0 1 2 3
```
Advantages

- Efficient edge insertion/deletion and adjacency check ($O(1)$)

Disadvantages

- Huge memory usage ($O(V^2)$)
  - sparse graph ⇒ wasted space!
  - undirected graph ⇒ wasted space!
### Adjacency List

**Array of V lists**

- **Undirected**
  - $A[0] = <1, 3>$
  - $A[1] = <0, 3>$
  - $A[3] = <0, 1, 2>$

- **Directed**
  - $A[0] = <1, 3>$
  - $A[1] = <0, 3>$
struct graph {
    int nV;
    int nE;
    struct adjNode **edges;
};

struct adjNode {
    Vertex v;
    struct adjNode *next;
};

Graphs
Graph ADT
Graph Reps
Adjacency Matrix
Adjacency List
Array of Edges
Summary
Adjacency List

Advantages and Disadvantages

Advantages

- Space-efficient for sparse graphs
- $O(V + E)$ memory usage

Disadvantages

- Inefficient edge insertion/deletion ($O(V)$)
  (matters less for sparse graphs)
Edges represented by an array of edge structs (pairs of vertices)

Undirected:

\[ A = [ (0, 1), (0, 3), (1, 3), (2, 3), ] \]

Directed:

\[ A = [ (0, 1), (0, 3), (1, 0), (1, 3), (3, 2), ] \]
struct graph {
    int nV;
    int nE;
    int maxE;
    struct edge *edges;
};

struct edge {
    Vertex v;
    Vertex w;
};

```c
struct graph {
    int nV;
    int nE;
    int maxE;
    struct edge *edges;
};

struct edge {
    Vertex v;
    Vertex w;
};

graph
edges
nV 4
nE 4
maxE 8

(0,1) (0,3) (1,3) (2,3)

Array of Edges
Implementation in C
Array of Edges

Advantages and Disadvantages

Advantages

Very space-efficient for sparse graphs where $E < V$

Disadvantages

Inefficient edge insertion/deletion ($O(E)$)
## Summary of Graph Representations

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List</th>
<th>Array of Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space usage</strong></td>
<td>$O(V^2)$</td>
<td>$O(V + E)$</td>
<td>$O(E)$</td>
</tr>
<tr>
<td><strong>Create</strong></td>
<td>$O(V^2)$</td>
<td>$O(V)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>Destroy</strong></td>
<td>$O(V)$</td>
<td>$O(V + E)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>Insert edge</strong></td>
<td>$O(1)$</td>
<td>$O(V)$</td>
<td>$O(E)$</td>
</tr>
<tr>
<td><strong>Remove edge</strong></td>
<td>$O(1)$</td>
<td>$O(V)$</td>
<td>$O(E)$</td>
</tr>
<tr>
<td><strong>Is adjacent</strong></td>
<td>$O(1)$</td>
<td>$O(V)$</td>
<td>$O(E)^*$</td>
</tr>
<tr>
<td><strong>Degree</strong></td>
<td>$O(V)$</td>
<td>$O(V)$</td>
<td>$O(E)^*$</td>
</tr>
</tbody>
</table>

* Can be $O(\log E)$ if the array is ordered and both directions of each edge are stored in an undirected graph.