COMP2521 23T3
Graphs (I)

Kevin Luxa
cs2521@cse.unsw.edu.au

graph fundamentals
graph representations
Graph Fundamentals
Collections of Related Things

Up to this point, we’ve seen a few collection types...

- **lists**: a linear sequence of items
  each node knows about its next node
- **trees**: a branched hierarchy of items
  each node knows about its child node(s)

what if we want something more general?
...each node knows about its related 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.
Questions we could answer with a graph:

- what items are connected? how?
- are the items fully connected?
- is there a way to get from $A$ to $B$?
  what’s the best way? what’s the cheapest way?
- in general, what can we reach from $A$?
- is there a path that lets me visit all items?
- can we form a tree linking all vertices?
- are two graphs “equivalent”?
<table>
<thead>
<tr>
<th></th>
<th>ADL</th>
<th>BNE</th>
<th>CBR</th>
<th>DRW</th>
<th>MEL</th>
<th>PER</th>
<th>SYD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADL</td>
<td>—</td>
<td>2055</td>
<td>1390</td>
<td>3051</td>
<td>732</td>
<td>2716</td>
<td>1605</td>
</tr>
<tr>
<td>BNE</td>
<td>2055</td>
<td>—</td>
<td>1291</td>
<td>3429</td>
<td>1671</td>
<td>4771</td>
<td>982</td>
</tr>
<tr>
<td>CBR</td>
<td>1390</td>
<td>1291</td>
<td>—</td>
<td>4441</td>
<td>658</td>
<td>4106</td>
<td>309</td>
</tr>
<tr>
<td>DRW</td>
<td>3051</td>
<td>3429</td>
<td>4441</td>
<td>—</td>
<td>3783</td>
<td>4049</td>
<td>4411</td>
</tr>
<tr>
<td>MEL</td>
<td>732</td>
<td>1671</td>
<td>658</td>
<td>3783</td>
<td>—</td>
<td>3448</td>
<td>873</td>
</tr>
<tr>
<td>PER</td>
<td>2716</td>
<td>4771</td>
<td>4106</td>
<td>4049</td>
<td>3448</td>
<td>—</td>
<td>3972</td>
</tr>
<tr>
<td>SYD</td>
<td>1605</td>
<td>982</td>
<td>309</td>
<td>4411</td>
<td>873</td>
<td>3972</td>
<td>—</td>
</tr>
</tbody>
</table>
Road Distances
Road Distances

Graphs
Types of Graphs
Graph Terminology
Graph ADT
Graph Rep.

ADL
SYD
BRIS
CBR
MEL
PER
BNE
A graph $G$ is a set of vertices $V$ and edges $E$.

$E := \{(v, w) \mid v, w \in V, (v, w) \in V \times V\}$

$V = \{v_1, v_2, v_3, v_4\}$

$E = \{e_1 := (v_1, v_2), e_2 := (v_2, v_3), e_3 := (v_3, v_4), e_4 := (v_1, v_4), e_5 := (v_1, v_3)\}$
Types of Graphs

- Undirected
- Directed
- Multigraph
- Weighted
If edges in a graph are directed, the graph is a directed graph or digraph.

\((v, w) \in E\) does not imply \((w, v) \in E\).

A digraph with \(V\) vertices can have at most \(V^2\) edges. Digraphs can have self loops \((v \rightarrow v)\).
Multigraphs and Weighted Graphs

**Multi-Graphs...**
allow multiple edges between two vertices
(e.g., callgraphs; maps)

**Weighted Graphs...**
each edge has an associated weight
(e.g., maps; networks)
At this point, we’ll only consider simple graphs:

- a set of vertices
- a set of undirected edges
- no self loops
- no parallel edges

How many edges can a 7-vertex simple graph have?

\[ |V| = 7; |E| = 11. \]
At this point, we’ll only consider simple graphs:

- a set of vertices
- a set of undirected edges
- no self loops
- no parallel edges

How many edges can a 7-vertex simple graph have?

\[ 7 \times (7 - 1)/2 = 21 \]
Graph Terminology

Note: $|V|$ and $|E|$ is normally written as $V$ and $E$ for simplicity.

For a simple graph:

$$E \leq (V \times (V - 1))/2$$

- if $E$ closer to $V^2$, dense
- if $E$ closer to $V$, sparse

These properties affect our choice of representation and algorithms.

$V = 7; E = 11$. 
A complete graph is a graph where every vertex is connected to all other vertices:

\[ E = \left( V \times (V - 1) \right) / 2 \]
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 \( v (\text{deg}(v)) \) is the number of edges incident on \( v \).
A subgraph is a subset of vertices and associated edges.
A path is a sequence of vertices and edges... 1, 0, 6, 5

A path is simple if it has no repeating vertices.

A path is a cycle if it is simple except for its first and last vertex, which are the same.
A **connected graph** has a path from every vertex to every other vertex.

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

A tree has exactly one path between each pair of vertices.
A graph that is not connected consists of a set of connected components: maximally connected subgraphs.
A **spanning tree** of a graph is a subgraph that contains all its vertices and is a single tree.

A **spanning forest** of a graph is a subgraph that contains all its vertices and is a set of trees.

There isn’t necessarily *only* one spanning tree/forest for a graph.
A clique is a complete subgraph.
Graph ADT
What do we need to represent?
What operations do we need to support?
What do we need to represent?

A graph $G$ is a set of vertices $V := \{v_1, \ldots, v_n\}$, and a set of edges $E := \{(v, w) \mid v, w \in V; (v, w) \in V \times V\}$.

Directed graphs: $(v, w) \neq (w, v)$.

Weighted graphs: $E := \{(v, w, \sigma)\}$.

What operations do we need to support?

create/destroy graph;
add/remove vertices, edges;
get #vertices, #edges;
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

**manipulate**
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 memory allocated to a graph */
void GraphFree(Graph g);
/** 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 given vertices and false otherwise */
bool GraphIsAdjacent(Graph g, Vertex v, Vertex w);
A Graph ADT

"Graph.h" - Operations to Manipulate

/** 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.

**Adjacency Matrix**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

undirected

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

directed
struct graph {
    int nV;
    int nE;
    bool **edges;
};
Advantages

- Easy to implement! two-dimensional array of bool/int/double/...
- Works for: graphs! digraphs! weighted graphs!
- Efficient! \(O(1)\) edge-insert, edge-delete \(O(1)\) is-adjacent

Disadvantages

- Huge space overheads! \(V^2\) cells of some type sparse graph \(\Rightarrow\) wasted space!
  undirected graph \(\Rightarrow\) wasted space!
- Inefficient! \(O(V^2)\) initialisation
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>
A[2] = <>
```
**Adjacency List**

Implementation in C

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

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

```
graph
edges
nV 4
nE 4
```

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

```
1 0
3 0
1 2
```
Advantages

- Relatively easy to implement!
- Works for: graphs! digraphs! weighted graphs!
- Space-efficient! if graph has fewer edges $O(V + E)$ memory usage

Disadvantages

- Inefficient! $O(V)$ edge-insert, edge-delete $O(V)$ is-adjacent (matters less for sparse graphs)
Edges represented by an array of edge structs (pairs of vertices)

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

undirected

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

directed
Graphs
Graph ADT
Graph Rep.
Adjacency Matrix
Adjacency List
Array of Edges

Array of Edges
Implementation in C

```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)
```
Advantages

• Works for: graphs! digraphs! weighted graphs!
• Very space-efficient! especially for sparse graphs where $E < V$

Disadvantages

• Inefficient! $O(E)$ edge-insert, edge-delete
### 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>Space usage</td>
<td>$O(V^2)$</td>
<td>$O(V + E)$</td>
<td>$O(E)$</td>
</tr>
<tr>
<td>Create</td>
<td>$O(V^2)$</td>
<td>$O(V)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Destroy</td>
<td>$O(V)$</td>
<td>$O(V + E)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insert edge</td>
<td>$O(1)$</td>
<td>$O(V)$</td>
<td>$O(E)$</td>
</tr>
<tr>
<td>Remove edge</td>
<td>$O(1)$</td>
<td>$O(V)$</td>
<td>$O(E)$</td>
</tr>
<tr>
<td>Is adjacent</td>
<td>$O(1)$</td>
<td>$O(V)$</td>
<td>$O(E)^*$</td>
</tr>
<tr>
<td>Degree</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.
https://forms.office.com/r/aPF09YHZ3X