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

Collections of Related Things

... Related Nodes? (I)

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.

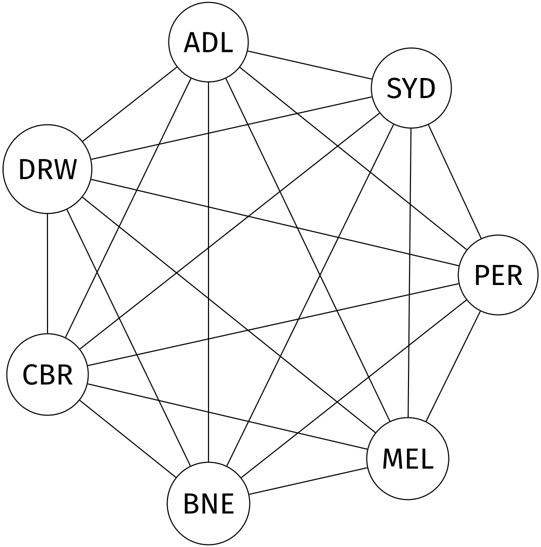
Questions we could answer with a graph:

- 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"?

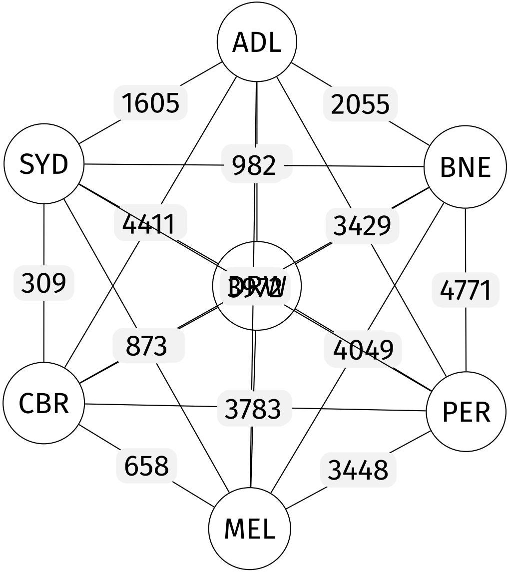
Road Distances

	ADL	BNE	CBR	DRW	MEL	PER	SYD
ADL	—	2055	1390	3051	732	2716	1605
BNE	2055	—	1291	3429	1671	4771	982
CBR	1390	1291	—	4441	658	4106	309
DRW	3051	3429	4441	—	3783	4049	4411
MEL	732	1671	658	3783	—	3448	873
PER	2716	4771	4106	4049	3448	—	3972
SYD	1605	982	309	4411	873	3972	—

Road Distances

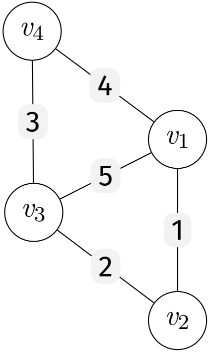


Road Distances



A graph G is a set of vertices V and edges E.

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

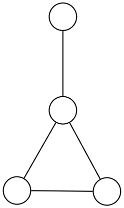


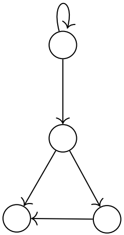
$$V = \{v_1, v_2, v_3, v_4\}$$

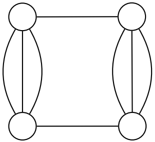
$$E = \left\{ \begin{array}{lll} e_1 & := & (v_1, v_2), \\ e_2 & := & (v_2, v_3), \\ e_3 & := & (v_3, v_4), \\ e_4 & := & (v_1, v_4), \\ e_5 & := & (v_1, v_3) \end{array} \right\}$$

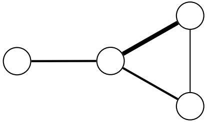
Graphs

Types of Graphs


undirected


directed


multigraph


weighted

Directed Graphs

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 ($v \to v$)

Multigraphs and Weighted Graphs

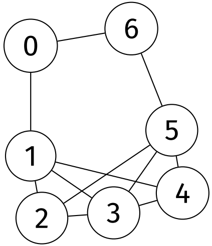
Multi-Graphs...
allow multiple edges between two vertices
(e.g., callgraphs; maps)

Weighted Graphs...
each edge has an associated weight
(e.g., maps; networks)

Simple Graphs

At this point,
we'll only consider simple graphs:

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



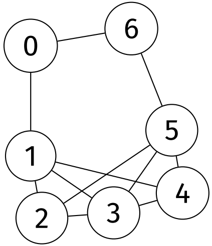
$|V| = 7; |E| = 11.$

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

Simple Graphs

At this point,
we'll only consider simple graphs:

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



$|V| = 7; |E| = 11.$

How many edges can a
7-vertex simple graph have?
$7 \times (7-1)/2 = 21$

Graph Terminology (I)

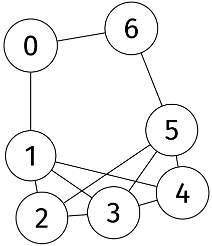
Note: $|V|$ and $|E|$ is normally written as V and E for simplicity.

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.

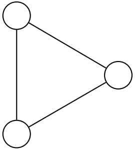


$V = 7$; $E = 11$.

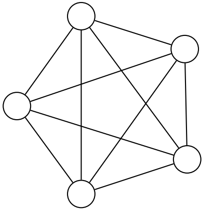
Graph Terminology (II)

A complete graph is a graph where
every vertex is connected to all other vertices:

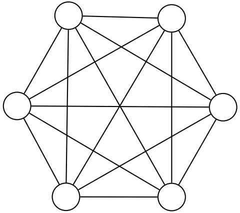
$$E = (V \times (V - 1))/2$$



K_3



K_5

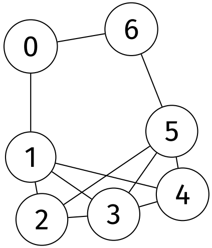


K_6

Graph Terminology (III)

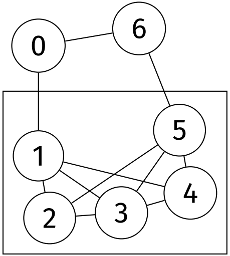
Two vertices v and w are adjacent if an edge $e := (v, w)$ connects them; we say e is incident on v and w

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



Graph Terminology (IV)

A subgraph is a subset of vertices and associated edges

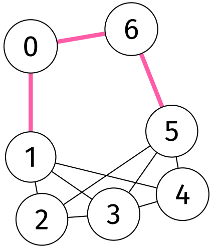


Graph Terminology (V)

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.

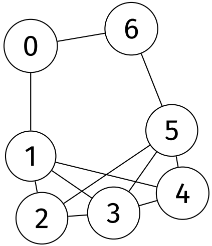


Graph Terminology (VI)

A connected graph
has a path from every vertex
to every other vertex

A connected graph
with no cycles is a tree.

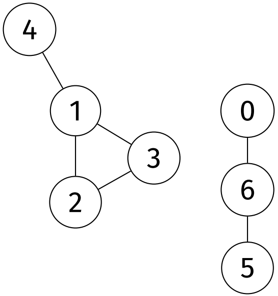
A tree has exactly one path
between each pair of vertices.



(not a tree)

Graph Terminology (VII)

A graph that is not connected consists of a set of **connected components**: maximally connected subgraphs

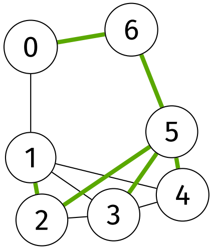


Graph Terminology
(VIII)

A spanning tree of a graph
is a subgraph that
contains all its vertices
and is a single tree

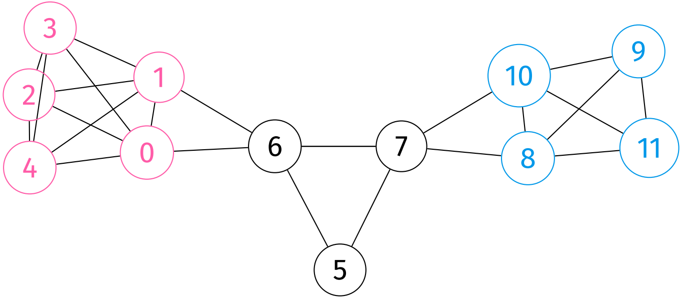
A spanning forest of a graph
is a subgraph that
contains all its vertices
and is a set of trees

There isn't necessarily *only* one
spanning tree/forest for a graph.



Graph Terminology
(IX)

A clique is a complete subgraph.



Graph ADT

Graph ADT

What do we need to represent?
What operations do we need to support?

Graph ADT

What do we need to represent?

A graph G is a set of vertices $V := \{v_1, \cdots, v_n\}$,
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.

A Graph ADT

"Graph.h" - Operations to Create/Destroy

```
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);
```

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 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

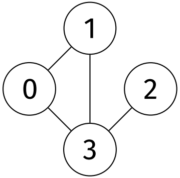
Adjacency List
Edges defined by entries in array of V lists

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

We'll consider these representations for *unweighted*, *undirected* graphs.

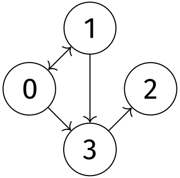
Adjacency Matrix

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



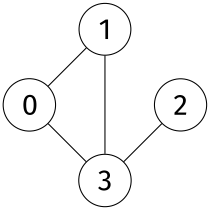
$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

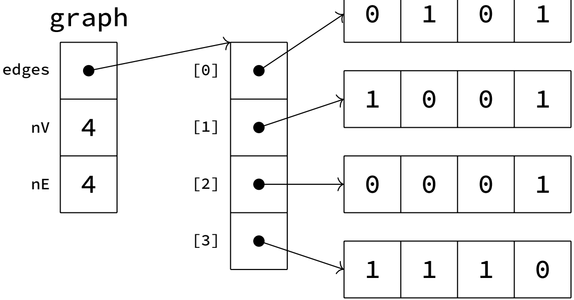
directed

Adjacency Matrix

Implementation in C

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





Adjacency Matrix

Advantages and Disadvantages

Advantages

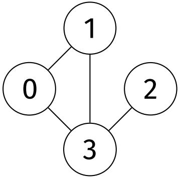
- Easy to implement!
  two-dimensional array of `bool`/`int`/`double`/...
- Works for:
  graphs! digraphs!
  weighted graphs!
- Efficient!
  $O(1)$ edge-insert, edge-delete
  $O(1)$ is-adjacent

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



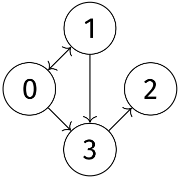
A[0] = <1, 3>

A[1] = <0, 3>

A[2] = <3>

A[3] = <0, 1, 2>

undirected



A[0] = <1, 3>

A[1] = <0, 3>

A[2] = <>

A[3] = <2>

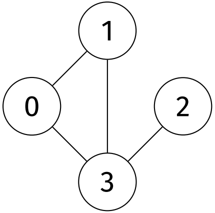
directed

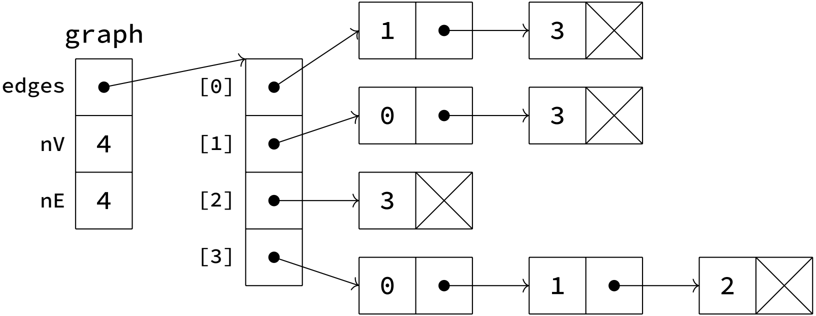
Adjacency List

Implementation in C

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

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





Adjacency List

Advantages and Disadvantages

Advantages

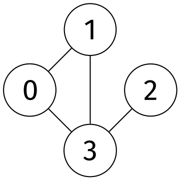
- Relatively easy to implement!
- Works for:
  graphs! digraphs!
  weighted graphs!
- Space-efficient!
  if graph has fewer edges
  $O(V + E)$ memory usage

Disadvantages

- Inefficient!
  $O(V)$ edge-insert, edge-delete
  $O(V)$ is-adjacent
  (matters less for sparse graphs)

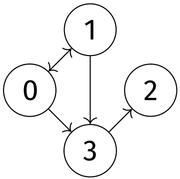
Array of Edges

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),
]
```

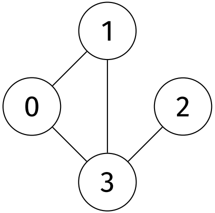
directed

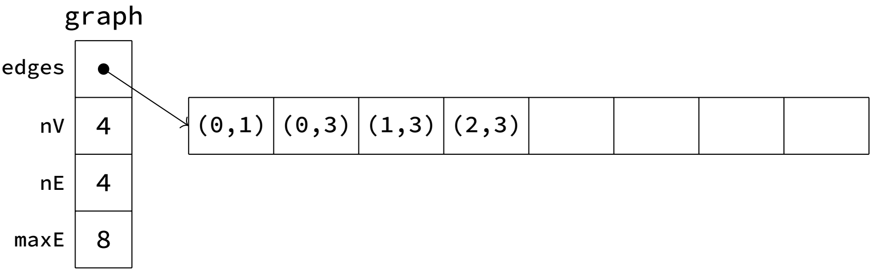
Array of Edges

Implementation in C

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

struct edge {
    Vertex v;
    Vertex w;
};
```





Array of Edges

Advantages and Disadvantages

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

	Adjacency Matrix	Adjacency List	Array of Edges
Space usage	$O(V^2)$	$O(V+E)$	$O(E)$
Create	$O(V^2)$	$O(V)$	$O(1)$
Destroy	$O(V)$	$O(V+E)$	$O(1)$
Insert edge	$O(1)$	$O(V)$	$O(E)$
Remove edge	$O(1)$	$O(V)$	$O(E)$
Is adjacent	$O(1)$	$O(V)$	$O(E)$*
Degree	$O(V)$	$O(V)$	$O(E)$*

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

Feedback

https://forms.office.com/r/aPF09YHZ3X