COMP2521 19T0

Week 3, Thursday: Graphic Content (II)!

Jashank Jeremy

jashank.jeremy@unsw.edu.au

graph representation
graph search

Graph Representation

What do we need to represent?

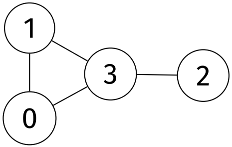
A graph G is a set of vertices $V := \{v_1, \cdots, v_n\}$,
and a set of edges $E := \{(v, w) \mid v, w \in V;\ (v, w) \in V \times V\}$.
Directed graphs: $(v, w) \neq (w, v)$.
Weighted graphs: $E := \{(v, w, \sigma)\}$.
Multigraphs: E is a list, not a set.

What operations do we need to support?

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

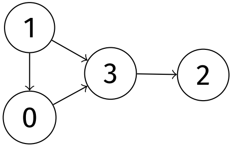
Adjacency Matrices

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



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

directed

Adjacency Matrices

Advantages and Disadvantages

Advantages

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

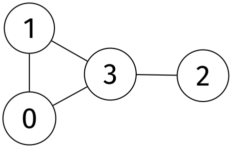
Disadvantages

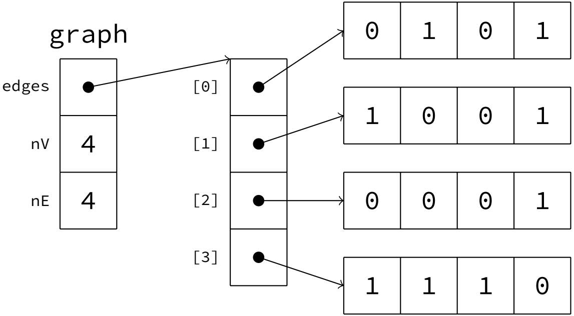
- Huge space overheads!
  V^2 cells of some type
  sparse graph $\Rightarrow$ wasted space!
  undirectd graph $\Rightarrow$ wasted space!
- Inefficient!
  $O(V^2)$ initialisation
  $O(V^2)$ vertex-insert/-delete

Adjacency Matrices

Implementation in C

```
struct graph {
    size_t nV, nE;
    bool **matrix;
};
```





Exercise: Time Complexity

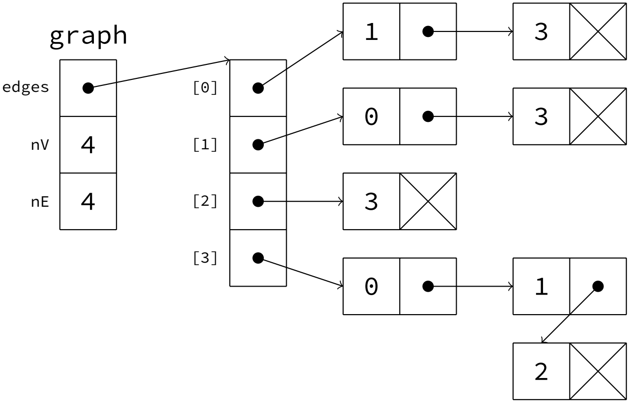
Given an adjacency matrix representation,
find the time complexity, and implement, these functions

- `bool graph_adjacent (Graph g, vertex v, vertex w);`
  ... returns true if vertices v and w are connected, false otherwise
- `size_t graph_degree (Graph g, vertex v);`
  ... return the degree of a vertex v

Adjacency Lists

Implementation in C

```
typedef
    struct adjnode
    adjnode;
struct graph {
    size_t nV, nE;
    adjnode **edges;
};
struct adjnode {
    vertex w;
    adjnode *next;
};
```



Comparison

Adjacency Lists vs Adjacency Matrices

- Space: matrix: V^2; adjlist: $V + E$
- Initialise: matrix: V^2, adjlist: V
- Destroy: matrix: V, adjlist: E
- Insert edge: matrix 1, adjlist: V
- Find/remove edge: matrix: 1, adjlist: V
- is isolated? matrix: V, adjlist: 1
- Degree: matrix: V, adjlist: E
- is adjacent? matrix: 1, adjlist: V

Designing an ADT

What do we need to represent?
What operations do we need to support?
What behaviours are we trying to model?
How do we interact with other types?

A Graph ADT

<graph.h> — Create, Destroy

```
typedef struct graph *Graph;
```

A Graph ADT

<graph.h> — Create, Destroy

```
typedef struct graph *Graph;

/** A concrete edge type. */
typedef struct edge { vertex v, w; weight n; } edge;
```

A Graph ADT

<graph.h> — Create, Destroy

```
typedef struct graph *Graph;

/** A concrete edge type. */
typedef struct edge { vertex v, w; weight n; } edge;

/** Create a new instance of a Graph. */
Graph graph_new (
    size_t max_edges,       /**< maximum value hint */
    size_t max_vertices,    /**< maximum value hint */
    bool directed,          /**< true if a digraph */
    bool weighted           /**< true if edges have weight */
);

/** Deallocate resources used by a Graph. */
void graph_drop (Graph g);
```

A Graph ADT

<graph.h> — Simple Facts

```
/** Get the number of vertices in this Graph. */
size_t graph_num_vertices (Graph g);

/** Get the number of edges in this Graph. */
size_t graph_num_edges (Graph g);
```

A Graph ADT

<graph.h> — Simple Facts

```
/** Get the number of vertices in this Graph. */
size_t graph_num_vertices (Graph g);

/** Get the number of edges in this Graph. */
size_t graph_num_edges (Graph g);

/** Is this graph directed? */
bool graph_directed_p (Graph g);

/** Is this graph weighted? */
bool graph_weighted_p (Graph g);
```

A Graph ADT

<graph.h> — Add/Remove

```
/** Add vertex with index `v' to the Graph.
 * If the vertex already exists, a no-op returning false. */
bool graph_vertex_add (Graph g, vertex v);
```

A Graph ADT

<graph.h> — Add/Remove

```
/** Add vertex with index `v' to the Graph.
 * If the vertex already exists, a no-op returning false. */
bool graph_vertex_add (Graph g, vertex v);

/** Add edge `e', from `v' to `w' with weight `n', to the Graph.
 * If the edge already exists, a no-op returning false. */
bool graph_edge_add (Graph g, edge e);
```

A Graph ADT

<graph.h> — Add/Remove

```
/** Add vertex with index `v' to the Graph.
 * If the vertex already exists, a no-op returning false. */
bool graph_vertex_add (Graph g, vertex v);

/** Add edge `e', from `v' to `w' with weight `n', to the Graph.
 * If the edge already exists, a no-op returning false. */
bool graph_edge_add (Graph g, edge e);

/** Remove edge `e' between `v' and `w' from the Graph. */
void graph_edge_remove (Graph g, edge e);
```

A Graph ADT

<graph.h> — Add/Remove

```
/** Add vertex with index `v' to the Graph.
 * If the vertex already exists, a no-op returning false. */
bool graph_vertex_add (Graph g, vertex v);

/** Add edge `e', from `v' to `w' with weight `n', to the Graph.
 * If the edge already exists, a no-op returning false. */
bool graph_edge_add (Graph g, edge e);

/** Remove edge `e' between `v' and `w' from the Graph. */
void graph_edge_remove (Graph g, edge e);

/** Remove vertex `v' from the Graph. */
void graph_vertex_remove (Graph g, vertex v);
```

A Graph ADT

<graph.h> — Answering Questions

```
/** Does this Graph have this vertex? */
bool graph_has_vertex_p (Graph g, vertex v);
```

A Graph ADT

<graph.h> — Answering Questions

```
/** Does this Graph have this vertex? */
bool graph_has_vertex_p (Graph g, vertex v);

/** What is the degree of this vertex on this Graph? */
size_t graph_vertex_degree (Graph g, vertex v);
```

A Graph ADT

<graph.h> — Answering Questions

```
/** Does this Graph have this vertex? */
bool graph_has_vertex_p (Graph g, vertex v);

/** What is the degree of this vertex on this Graph? */
size_t graph_vertex_degree (Graph g, vertex v);

/** Does this Graph have this edge? */
bool graph_has_edge_p (Graph g, edge e);
```

Graph Search

Searching on Graphs

We learn properties of a graph by
systematically examining
each of its edges and vertices —

… to compute the degree of all vertices,
we visit each vertex, and count its edges

… for path-related properties
we move from vertex to vertex along edges
choosing edges as we go

we implement general graph-search algorithms
which can solve a wide range of graph problems

Simple Path Search

PROBLEM

does a path exist between vertices v and w?

Simple Path Search

PROBLEM

does a path exist between vertices v and w?

- examine vertices adjacent to v;
- if any of them is w, we're done!
- otherwise, check from all of the adjacent vertices
  … rinse and repeat moving away from v

Simple Path Search

PROBLEM

does a path exist between vertices v and w?

- examine vertices adjacent to v;
- if any of them is w, we're done!
- otherwise, check from all of the adjacent vertices
  ... rinse and repeat moving away from v

What order do we visit nodes in?
'Breadth-first' (BFS): adjacent nodes first
'Depth-first' (DFS): longest paths first
Dijkstra: lowest-cost paths first
'Greedy Best-First' (GBFS): shortest-heuristic-distance
A*: lowest-cost *and* shortest-heuristic-distance

Simple Path Search

Path searches on graphs tend to follow a simple pattern:

- create a structure that will tell us what next
- add the starting node to that structure
- while that structure isn't empty:
  - get the next vertex from that structure;
  - mark that vertex as visited; and
  - add its neighbours to the structure

Simple Path Search

Path searches on graphs tend to follow a simple pattern:

- create a structure that will tell us what next
- add the starting node to that structure
- while that structure isn't empty:
  - get the next vertex from that structure;
  - mark that vertex as visited; and
  - add its neighbours to the structure

What data structure should we use?

BFS:

DFS:

Simple Path Search

Path searches on graphs tend to follow a simple pattern:

- create a structure that will tell us what next
- add the starting node to that structure
- while that structure isn't empty:
  - get the next vertex from that structure;
  - mark that vertex as visited; and
  - add its neighbours to the structure

What data structure should we use?

BFS: a queue!

DFS:

Simple Path Search

Path searches on graphs tend to follow a simple pattern:

- create a structure that will tell us what next
- add the starting node to that structure
- while that structure isn't empty:
  - get the next vertex from that structure;
  - mark that vertex as visited; and
  - add its neighbours to the structure

What data structure should we use?
BFS: a queue!
DFS: a stack!

Depth-First Search

`count`	number of vertices traversed so far
`pre[]`	order in which vertices were visited (for ‘pre-order’)
`st[]`	predecessor of each vertex (for ‘spanning tree’)

Depth-First Search

`count` number of vertices traversed so far

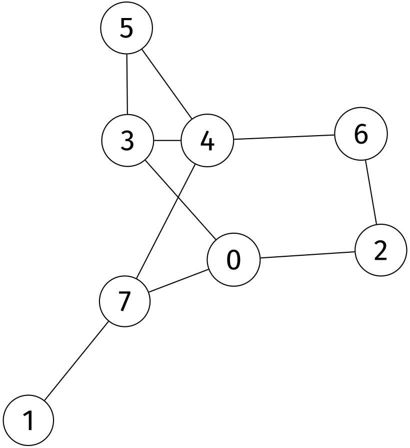
`pre[]` order in which vertices were visited (for ‘pre-order’)

`st[]` predecessor of each vertex (for ‘spanning tree’)

the edges traversed in all graph walks form a spanning tree, which has —

- has edges corresponding to call-tree of recursive function
- is the original graph sans cycles/alternate paths
- (in general, a spanning tree has all vertices
  and a minimal set of edges to produce a connected graph;
  no loops, cycles, parallel edges)

Depth-First Search



Depth-First Search

... on an unconnected graph

If a graph is not connected,
DFS will produce
a spanning forest

An edge connecting a vertex with
an ancestor in the DFS tree
that is not its parent is a back edge

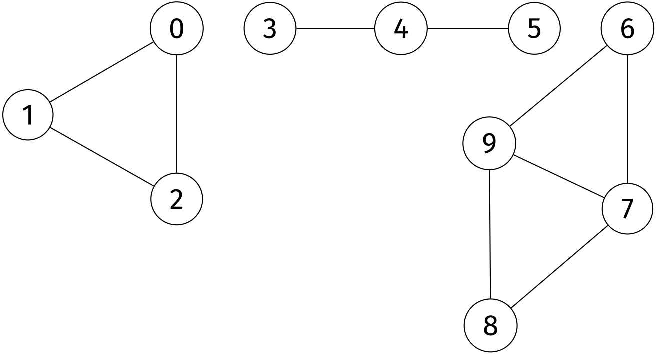
Depth-First Search

... recursively, using the call stack (I)

```
void dfsR (Graph g, edge e) {
    // ... set up `pre' array of `g->nV' items set to -1
    // ... set up `st' array of `g->nV' items set to -1
    // ... set up `count' = 0
    pre[w] = count++;
    st[w] = e.v;
    vertex w = e.w;
    for (vertex i = 0; i < g->nV; i++)
        if (g->edges[w][i] && pre[i] == -1)
            dfsR (g, (edge){.v = w, .w = i});
}
```

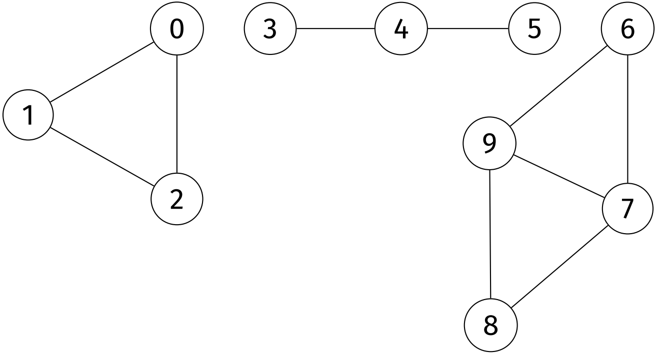
Depth-First Search

... recursively, using the call stack (II)



Depth-First Search

... recursively, using the call stack (II)



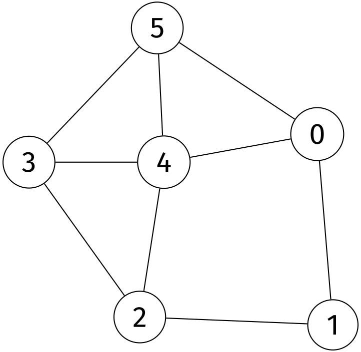
How can we ensure that all vertices are visited?

Depth-First Search

... recursively, using the call stack (III)

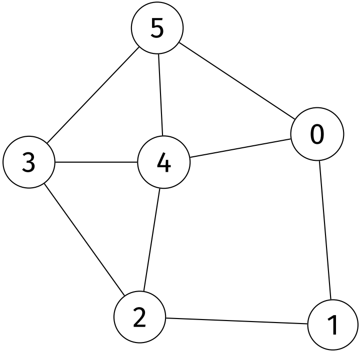
```
void dfs (Graph g)
{
    count = 0;
    pre = calloc (g->nV * sizeof (int));
    st = calloc (g->nV * sizeof (int));
    for (vertex v = 0; v < g->nV; v++)
        pre[v] = st[v] = 1;
    for (vertex v = 0; v < g->nV; v++)
        if (pre[v] == -1)
            dfsR (g, (edge){.v = v, .w = v});
}
```

Depth-First Search



Let’s do a DFS!

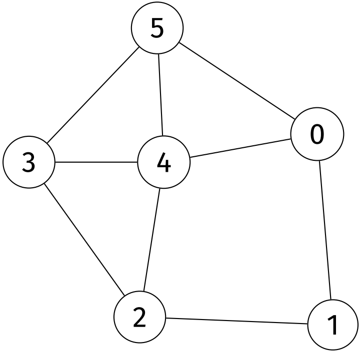
Depth-First Search



Let's do a DFS!

Does a path exist from $0 \cdots 5$?

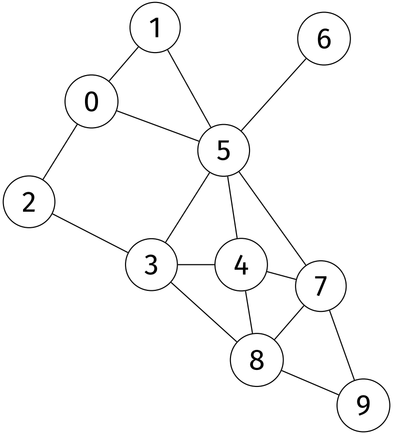
Depth-First Search



Let's do a DFS!

Does a path exist from $0 \cdots 5$?
Yes: $0, 1, 2, 3, 4, 5$.

Depth-First Search



Let's do a DFS!
What do `pre[]` and `st[]` look like?

Depth-First Search

... iteratively

```
void dfs (Graph g, edge e)
{
    // ... set up `pre' array of `g->nV' items set to -1
    // ... set up `st' array of `g->nV' items set to -1
    // ... set up `count' = 0
    Stack s = stack_new ();
    stack_push (s, e);
    while (stack_size (s) > 0) {
        e = stack_pop (s);
        if (pre[e.w] != -1) continue;
        pre[e.w] = count++; st[e.w] = e.v;
        for (int i = 0; i < g->nV; i++)
            if (has_edge (e.w, i) && pre[i] == -1)
                stack_push (s, (edge){.v = e.w, .w = i });
    }
}
```

Breadth-First Search

... iteratively

```
void bfs (Graph g, edge e)
{
    // ... set up `pre' array of `g->nV' items set to -1
    // ... set up `st' array of `g->nV' items set to -1
    // ... set up `count' = 0
    Queue q = queue_new ();
    queue_en (q, e);
    while (queue_size (q) > 0) {
        e = queue_de (q);
        if (pre[e.w] != -1) continue;
        pre[e.w] = count++; st[e.w] = e.v;
        for (int i = 0; i < g->nV; i++)
            if (has_edge (e.w, i) && pre[i] == -1)
                queue_en (q, (edge){.v = e.w, .w = i });
    }
}
```