

Week 06: Graph Data Structures and Search

Graph Definitions

Graphs

2/83

Many applications require

- a collection of *items* (i.e. a set)
- *relationships*/connections between items

Examples:

- maps: items are cities, connections are roads
- web: items are pages, connections are hyperlinks

Collection types you're familiar with

- lists ... linear sequence of items (week 4; COMP9021)
- trees ... branched hierarchy of items (COMP9021)

Graphs are more general ... allow arbitrary connections

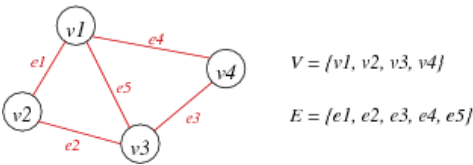
... Graphs

3/83

A graph $G = (V,E)$

- V is a set of *vertices*
- E is a set of *edges* (subset of $V \times V$)

Example:



... Graphs

4/83

A real example: Australian road distances

Distance	Adelaide	Brisbane	Canberra	Darwin	Melbourne	Perth	Sydney
Adelaide	-	2055	1390	3051	732	2716	1605

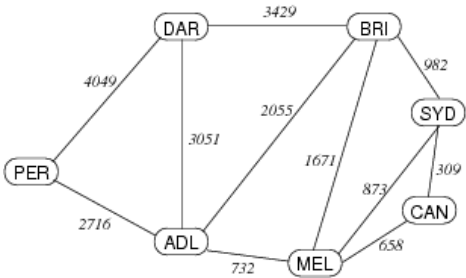
Brisbane	2055	-	1291	3429	1671	4771	982
Canberra	1390	1291	-	4441	658	4106	309
Darwin	3051	3429	4441	-	3783	4049	4411
Melbourne	732	1671	658	3783	-	3448	873
Perth	2716	4771	4106	4049	3448	-	3972
Sydney	1605	982	309	4411	873	3972	-

Notes: vertices are cities, edges are distance between cities, symmetric

... Graphs

5/83

Alternative representation of above:



... Graphs

6/83

Questions we might ask about a graph:

- is there a way to get from item A to item B?
- what is the best way to get from A to B?
- which items are connected?

Graph algorithms are generally more complex than tree/list ones:

- no implicit order of items
- graphs may contain cycles
- concrete representation is less obvious
- algorithm complexity depends on connection complexity

Properties of Graphs

7/83

Terminology: $|V|$ and $|E|$ (cardinality) normally written just as V and E .

A graph with V vertices has at most $V(V-1)/2$ edges.

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*
 - Example: web pages and hyperlinks

Knowing whether a graph is sparse or dense is important

- may affect choice of data structures to represent graph
- may affect choice of algorithms to process graph

Exercise #1: Number of Edges

8/83

The edges in a graph represent pairs of connected vertices. A graph with V has V^2 such pairs.

Consider $V = \{1,2,3,4,5\}$ with all possible pairs:

$$E = \{ (1,1), (1,2), (1,3), (1,4), (1,5), (2,1), (2,2), \dots, (4,5), (5,5) \}$$

Why do we say that the maximum #edges is $V(V-1)/2$?

... because

- (v,w) and (w,v) denote the same edge (in an undirected graph)
- we do not consider loops (v,v)

Graph Terminology

10/83

For an edge e that connects vertices v and w

- v and w are *adjacent* (neighbours)
- e is *incident* on both v and w

Degree of a vertex v

- number of edges incident on v

Synonyms:

- vertex = node, edge = arc = link (Note: some people use arc for *directed* edges)

... Graph Terminology

11/83

Path: a sequence of vertices where

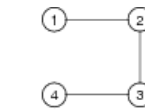
- each vertex has an edge to its predecessor

Cycle: a path where

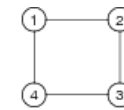
- last vertex in path is same as first vertex in path

Length of path or cycle:

- #edges



Path: 1-2, 2-3, 3-4



Cycle: 1-2, 2-3, 3-4, 4-1

... Graph Terminology

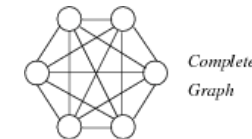
12/83

Connected graph

- there is a *path* from each vertex to every other vertex
- if a graph is not connected, it has ≥ 2 *connected components*

Complete graph K_V

- there is an *edge* from each vertex to every other vertex
- in a complete graph, $E = V(V-1)/2$



Complete Graph

... Graph Terminology

13/83

Tree: connected (sub)graph with no cycles

Spanning tree: tree containing all vertices

Clique: complete subgraph

Consider the following single graph:



This graph has 26 vertices, 33 edges, and 4 connected components

Note: The entire graph has no spanning tree; what is shown in green is a spanning tree of the third connected component

... Graph Terminology

14/83

A *spanning tree* of connected graph $G = (V, E)$

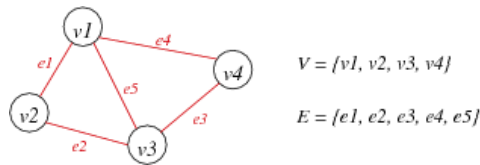
- is a subgraph of G containing all of V
- and is a single tree (connected, no cycles)

A *spanning forest* of non-connected graph $G = (V, E)$

- is a subgraph of G containing all of V
- and is a set of trees (not connected, no cycles),
 - with one tree for each *connected component*

Exercise #2: Graph Terminology

15/83



1. How many edges to remove to obtain a spanning tree?
2. How many different spanning trees?

1. 2
2. $\frac{5 \cdot 4}{2} - 2 = 8$ spanning trees (no spanning tree if we remove $\{e1, e2\}$ or $\{e3, e4\}$)

... Graph Terminology

17/83

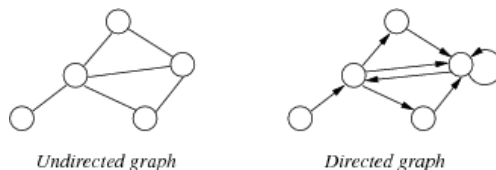
Undirected graph

- $edge(u, v) = edge(v, u)$, no self-loops (i.e. no $edge(v, v)$)

Directed graph

- $edge(u, v) \neq edge(v, u)$, can have self-loops (i.e. $edge(v, v)$)

Examples:



... Graph Terminology

Other types of graphs ...

Weighted graph

- each edge has an associated value (weight)
- e.g. road map (weights on edges are distances between cities)

Multi-graph

- allow multiple edges between two vertices
- e.g. function call graph ($f()$ calls $g()$ in several places)

Graph Data Structures

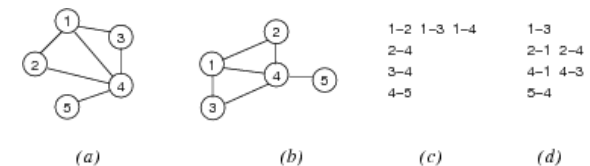
Graph Representations

20/83

Defining graphs:

- need some way of identifying vertices
- could give diagram showing edges and vertices
- could give a list of edges

E.g. four representations of the same graph:



... Graph Representations

21/83

We will discuss three different graph data structures:

1. Array of edges
2. Adjacency matrix
3. Adjacency list

Array-of-edges Representation

22/83

Edges are represented as an array of Edge values (= pairs of vertices)

- space efficient representation
- adding and deleting edges is slightly complex

- undirected: order of vertices in an Edge doesn't matter
- directed: order of vertices in an Edge encodes direction



For simplicity, we always assume vertices to be numbered $0 \dots V-1$

... Array-of-edges Representation

23/83

Graph initialisation

```
newGraph(V):
|   Input   number of nodes V
|   Output new empty graph
|
|   g.nV = V    // #vertices (numbered 0..V-1)
|   g.nE = 0    // #edges
|   allocate enough memory for g.edges[]
|   return g
```

How much is enough? ... No more than $V(V-1)/2$... Much less in practice (sparse graph)

... Array-of-edges Representation

24/83

Edge insertion

```
insertEdge(g, (v,w)):
|   Input graph g, edge (v,w)
|
|   g.edges[i]=(v,w)
|   g.edges[g.nE]=(v,w)
|   g.nE=g.nE+1
```

... Array-of-edges Representation

25/83

Edge removal

```
removeEdge(g, (v,w)):
|   Input graph g, edge (v,w)
|
|   i=0
|   while (v,w)≠g.edges[i] do
|       i=i+1
|   end while
|   g.edges[i]=g.edges[g.nE-1] // replace (v,w) by last edge in array
```

| g.nE=g.nE-1

Cost Analysis

26/83

Storage cost: $O(E)$

Cost of operations:

- initialisation: $O(1)$
- insert edge: $O(1)$ (assuming edge array has space)
- find/delete edge: $O(E)$ (need to find edge in edge array)

If array is full on insert

- allocate space for a bigger array, copy edges across $\Rightarrow O(E)$

If we maintain edges in order

- use binary search to insert/find edge $\Rightarrow O(\log E)$

Exercise #3: Array-of-edges Representation

27/83

Assuming an array-of-edges representation ...

Write an algorithm to output all edges of the graph

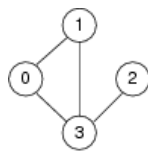
```
show(g):
|   Input graph g
|
|   for all i=0 to g.nE-1 do
|       print g.edges[i]
|   end for
```

Time complexity: $O(E)$

Adjacency Matrix Representation

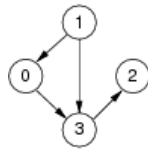
29/83

Edges represented by a $V \times V$ matrix



Undirected graph

A	0	1	2	3
0	0	1	0	1
1	1	0	0	1
2	0	0	0	1
3	1	1	1	0



Directed graph

A	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	0	0	0	0
3	0	0	1	0

... Adjacency Matrix Representation

30/83

Advantages

- easily implemented as 2-dimensional array
- can represent graphs, digraphs and weighted graphs
 - graphs: symmetric boolean matrix
 - digraphs: non-symmetric boolean matrix
 - weighted: non-symmetric matrix of weight values

Disadvantages:

- if few edges (sparse) \Rightarrow memory-inefficient

... Adjacency Matrix Representation

31/83

Graph initialisation

```
newGraph(V):
|   Input  number of nodes V
|   Output new empty graph
|
|   g.nV = V    // #vertices (numbered 0..V-1)
|   g.nE = 0    // #edges
|   allocate memory for g.edges[][]
|   for all i,j=0..V-1 do
|       g.edges[i][j]=0    // false
|   end for
|   return g
```

... Adjacency Matrix Representation

32/83

Edge insertion

```
insertEdge(g, (v,w)):
```

```
|   Input graph g, edge (v,w)
```

```
|   if g.edges[v][w]=0 then    // (v,w) not in graph
|       g.edges[v][w]=1        // set to true
|       g.edges[w][v]=1
|       g.nE=g.nE+1
|   end if
```

... Adjacency Matrix Representation

33/83

Edge removal

```
removeEdge(g, (v,w)):
```

```
|   Input graph g, edge (v,w)
```

```
|   if g.edges[v][w]≠0 then    // (v,w) in graph
|       g.edges[v][w]=0        // set to false
|       g.edges[w][v]=0
|       g.nE=g.nE-1
|   end if
```

Exercise #4: Show Graph

34/83

Assuming an adjacency matrix representation ...

Write an algorithm to output all edges of the graph (no duplicates!)

... Adjacency Matrix Representation

35/83

```
show(g):
|   Input graph g
|
|   for all i=0 to g.nV-2 do
|       for all j=i+1 to g.nV-1 do
|           if g.edges[i][j] then
|               print i-"-"j
|           end if
|       end for
|   end for
```

Time complexity: $O(V^2)$

Exercise #5:

36/83

Analyse storage cost and time complexity of adjacency matrix representation

Storage cost: $O(V^2)$

If the graph is sparse, most storage is wasted.

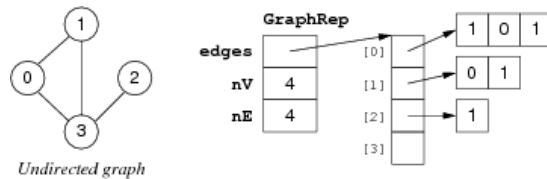
Cost of operations:

- initialisation: $O(V^2)$ (initialise $V \times V$ matrix)
- insert edge: $O(1)$ (set two cells in matrix)
- delete edge: $O(1)$ (unset two cells in matrix)

... Adjacency Matrix Representation

38/83

A storage optimisation: store only top-right part of matrix.



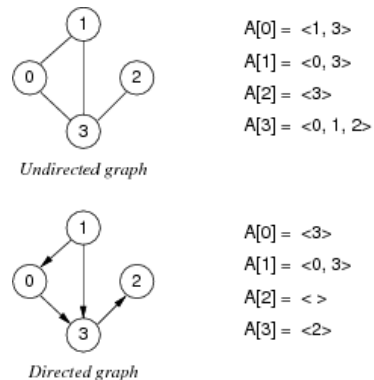
New storage cost: $V-1$ int ptrs + $V(V+1)/2$ ints (but still $O(V^2)$)

Requires us to always use edges (v,w) such that $v < w$.

Adjacency List Representation

39/83

For each vertex, store linked list of adjacent vertices:



... Adjacency List Representation

40/83

Advantages

- relatively easy to implement in languages like C
- can represent graphs and digraphs

- memory efficient if $E:V$ relatively small

Disadvantages:

- one graph has many possible representations (unless lists are ordered by same criterion e.g. ascending)

... Adjacency List Representation

41/83

Graph initialisation

```
newGraph(V):  
    Input number of nodes V  
    Output new empty graph  
  
    g.nV = V    // #vertices (numbered 0..V-1)  
    g.nE = 0    // #edges  
    allocate memory for g.edges[]  
    for all i=0..V-1 do  
        g.edges[i]=NULL    // empty list  
    end for  
    return g
```

... Adjacency List Representation

42/83

Edge insertion:

```
insertEdge(g, (v,w)):  
    Input graph g, edge (v,w)  
  
    insertLL(g.edges[v],w)  
    insertLL(g.edges[w],v)  
    g.nE=g.nE+1
```

... Adjacency List Representation

43/83

Edge removal:

```
removeEdge(g, (v,w)):  
    Input graph g, edge (v,w)  
  
    deleteLL(g.edges[v],w)  
    deleteLL(g.edges[w],v)  
    g.nE=g.nE-1
```

Exercise #6:

44/83

Analyse storage cost and time complexity of adjacency list representation

Storage cost: $O(V+E)$ (V list pointers, total of $2\cdot E$ list elements)

Cost of operations:

- initialisation: $O(V)$ (initialise V lists)
- insert edge: $O(I)$ (insert one vertex into list)
 - if you don't check for duplicates
- find/delete edge: $O(V)$ (need to find vertex in list)

If vertex lists are sorted

- insert requires search of list $\Rightarrow O(V)$
- delete always requires a search, regardless of list order

Comparison of Graph Representations

46/83

	array of edges	adjacency matrix	adjacency list
space usage	E	V^2	$V+E$
initialise	I	V^2	V
insert edge	I	I	I
find/delete edge	E	I	V

Other operations:

	array of edges	adjacency matrix	adjacency list
disconnected(v)?	E	V	I
isPath(x,y)?	$E\cdot\log V$	V^2	$V+E$
copy graph	E	V^2	E
destroy graph	I	V	E

Graph Abstract Data Type

Graph ADT

48/83

Data:

- set of edges, set of vertices

Operations:

- building: create graph, add edge

- deleting: remove edge, drop whole graph
- scanning: check if graph contains a given edge

Things to note:

- set of vertices is fixed when graph initialised
- we treat vertices as ints, but could be arbitrary Items

... Graph ADT

49/83

Graph ADT interface **Graph.h**

```
// graph representation is hidden
typedef struct GraphRep *Graph;

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

// edges are pairs of vertices (end-points)
typedef struct Edge { Vertex v; Vertex w; } Edge;

// operations on graphs
Graph newGraph(int V); // new graph with V vertices
void insertEdge(Graph, Edge);
void removeEdge(Graph, Edge);
bool adjacent(Graph, Vertex, Vertex); /* is there an edge
                                         between two vertices */

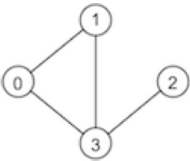
void freeGraph(Graph);
```

Exercise #7: Graph ADT Client

50/83

Write a program that uses the graph ADT to

- build the graph depicted below
- print all the nodes that are incident to vertex 1 in ascending order



```
#include <stdio.h>
#include "Graph.h"

#define NODES 4
#define NODE_OF_INTEREST 1

int main(void) {
```

```

Graph g = newGraph(NODES);

Edge e;
e.v = 0; e.w = 1; insertEdge(g,e);
e.v = 0; e.w = 3; insertEdge(g,e);
e.v = 1; e.w = 3; insertEdge(g,e);
e.v = 3; e.w = 2; insertEdge(g,e);

int v;
for (v = 0; v < NODES; v++) {
    if (adjacent(g, v, NODE_OF_INTEREST))
        printf("%d\n", v);
}

freeGraph(g);
return 0;
}

```

Graph ADT (Array of Edges)

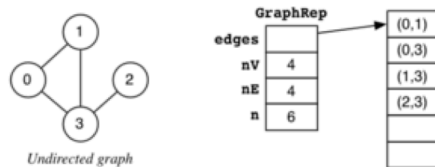
52/83

Implementation of GraphRep (array-of-edges representation)

```

typedef struct GraphRep {
    Edge *edges; // array of edges
    int nV;      // #vertices (numbered 0..nV-1)
    int nE;      // #edges
    int n;       // size of edge array
} GraphRep;

```



Graph ADT (Adjacency Matrix)

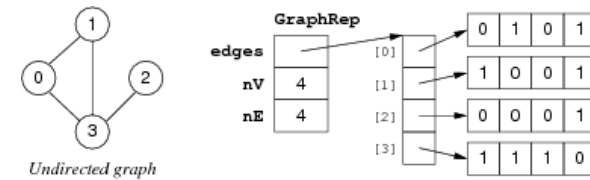
53/83

Implementation of GraphRep (adjacency-matrix representation)

```

typedef struct GraphRep {
    int **edges; // adjacency matrix
    int nV;      // #vertices
    int nE;      // #edges
} GraphRep;

```



... Graph ADT (Adjacency Matrix)

54/83

Implementation of graph initialisation (adjacency-matrix representation)

```

Graph newGraph(int V) {
    assert(V >= 0);
    int i;

    Graph g = malloc(sizeof(GraphRep));    assert(g != NULL);
    g->nV = V; g->nE = 0;

    // allocate memory for each row
    g->edges = malloc(V * sizeof(int *));    assert(g->edges != NULL);
    // allocate memory for each column and initialise with 0
    for (i = 0; i < V; i++) {
        g->edges[i] = calloc(V, sizeof(int)); assert(g->edges[i] != NULL);
    }
    return g;
}

```

standard library function **calloc(size_t nelems, size_t nbytes)**

- allocates a memory block of size `nelems*nbytes`
- and sets all bytes in that block to *zero*

... Graph ADT (Adjacency Matrix)

55/83

Implementation of edge insertion/removal (adjacency-matrix representation)

```

// check if vertex is valid in a graph
bool validV(Graph g, Vertex v) {
    return (g != NULL && v >= 0 && v < g->nV);
}

void insertEdge(Graph g, Edge e) {
    assert(g != NULL && validV(g,e.v) && validV(g,e.w));

    if (!g->edges[e.v][e.w]) { // edge e not in graph
        g->edges[e.v][e.w] = 1;
        g->edges[e.w][e.v] = 1;
        g->nE++;
    }
}

void removeEdge(Graph g, Edge e) {
    assert(g != NULL && validV(g,e.v) && validV(g,e.w));

    if (g->edges[e.v][e.w]) { // edge e in graph

```



```

    g->edges[e.v][e.w] = 0;
    g->edges[e.w][e.v] = 0;
    g->nE--;
}
}

```

Exercise #8: Checking Neighbours (i)

56/83

Assuming an adjacency-matrix representation ...

Implement a function to check whether two vertices are directly connected by an edge

```
bool adjacent(Graph g, Vertex x, Vertex y) { ... }
```

```

bool adjacent(Graph g, Vertex x, Vertex y) {
    assert(g != NULL && validV(g,x) && validV(g,y));

    return (g->edges[x][y] != 0);
}

```

Graph ADT (Adjacency List)

58/83

Implementation of GraphRep (adjacency-list representation)

```

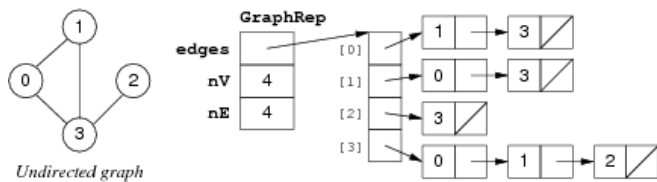
typedef struct GraphRep {
    Node **edges; // array of lists
    int    nV;    // #vertices
    int    nE;    // #edges
} GraphRep;

```

```

typedef struct Node {
    Vertex    v;
    struct Node *next;
} Node;

```



Exercise #9: Checking Neighbours (ii)

59/83

Assuming an adjacency list representation ...

Implement a function to check whether two vertices are directly connected by an edge

```
bool adjacent(Graph g, Vertex x, Vertex y) { ... }
```

```

bool adjacent(Graph g, Vertex x, Vertex y) {
    assert(g != NULL && validV(g,x));

    return inLL(g->edges[x], y);
}

```

inLL() checks if linked list contains an element

Graph Traversal

Finding a Path

62/83

Questions on paths:

- is there a path between two given vertices (*src*, *dest*)?
- what is the sequence of vertices from *src* to *dest*?

Approach to solving problem:

- examine vertices adjacent to *src*
- if any of them is *dest*, then done
- otherwise try vertices two edges from *src*
- repeat looking further and further from *src*

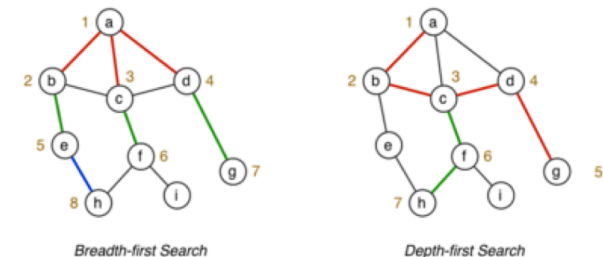
Two strategies for graph traversal/search: *depth-first*, *breadth-first*

- DFS follows one path to completion before considering others
- BFS "fans-out" from the starting vertex ("spreading" subgraph)

... Finding a Path

63/83

Comparison of BFS/DFS search for checking if there is a path from *a* to *h* ...



Both approaches ignore some edges by remembering previously visited vertices.

Depth-first Search

64/83

Depth-first search can be described recursively as

depthFirst(G,v):

1. mark v as visited
2. for each $(v,w) \in \text{edges}(G)$ do
if w has not been visited then
depthFirst(w)

The recursion induces *backtracking*

... Depth-first Search

65/83

Recursive DFS path checking

hasPath(G,src,dest):

```
Input graph G, vertices src,dest
Output true if there is a path from src to dest in G,
        false otherwise

return dfsPathCheck(G,src,dest)
```

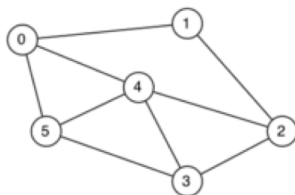
dfsPathCheck(G,v,dest):

```
mark v as visited
if v=dest then           // found dest
    return true
else
    for all (v,w) ∈ edges(G) do
        if w has not been visited then
            return dfsPathCheck(G,w,dest) // found path via w to dest
        end if
    end for
end if
return false           // no path from v to dest
```

Exercise #10: Depth-first Traversal (i)

66/83

Trace the execution of `dfsPathCheck(G,0,5)` on:



Consider neighbours in ascending order

Answer:

0 - 1 - 2 - 3 - 4 - 5

... Depth-first Search

68/83

Cost analysis:

- each vertex visited at most once \Rightarrow cost = $O(V)$
- visit all edges incident on visited vertices \Rightarrow cost = $O(E)$
 - assuming an adjacency list representation

Time complexity of DFS: $O(V+E)$ (adjacency list representation)

... Depth-first Search

69/83

Note how different graph data structures affect cost:

- array-of-edges representation
 - visit all edges incident on visited vertices \Rightarrow cost = $O(E^2)$
 - cost of DFS: $O(V+E^2)$
- adjacency-matrix representation
 - visit all edges incident on visited vertices \Rightarrow cost = $O(V^2)$
 - cost of DFS: $O(V^2)$

For *dense graphs* ... $E \cong V^2 \Rightarrow O(V+E) = O(V^2)$

For *sparse graphs* ... $E \cong V \Rightarrow O(V+E) = O(E)$

... Depth-first Search

70/83

Knowing whether a path exists can be useful

Knowing what the path is even more useful

\Rightarrow record the previously visited node as we search through the graph (so that we can then trace path through graph)

Make use of global variable:

- `visited[]` ... array to store previously visited node, for each node being visited

... Depth-first Search

71/83

`visited[]` // store previously visited node, for each vertex $0..nV-1$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Path: 6-5-1-0

... Depth-first Search

74/83

DFS can also be described non-recursively (via a *stack*):

```
hasPath(G,src,dest):
    Input graph G, vertices src,dest
    Output true if there is a path from src to dest in G,
            false otherwise

    push src onto new stack s
    found=false
    while not found and s is not empty do
        pop v from s
        mark v as visited
        if v=dest then
            found=true
        else
            for each (v,w)∈edges(G) such that w has not been visited
                push w onto s
            end for
        end if
    end while
    return found
```

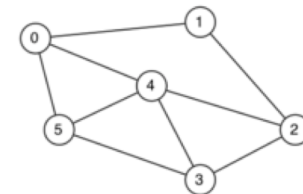
Uses standard stack operations (push, pop, check if empty)

Time complexity is the same: $O(V+E)$ (each vertex added to stack once, each element in vertex's adjacency list visited once)

Exercise #12: Depth-first Traversal (iii)

75/83

Show how the stack evolves when executing `findPathDFS(g,0,5)` on:



Push neighbours in *descending* order ... so they get popped in ascending order

```
findPath(G,src,dest):
    Input graph G, vertices src,dest

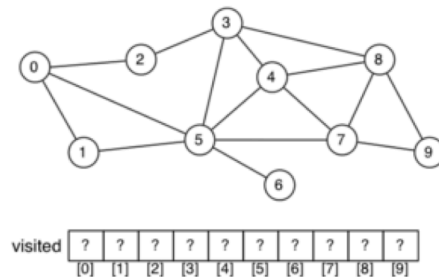
    for all vertices v∈G do
        visited[v]=-1
    end for
    visited[src]=src                // starting node of the path
    if dfsPathCheck(G,src,dest) then // show path in dest..src order
        v=dest
        while v≠src do
            print v "-"
            v=visited[v]
        end while
        print src
    end if

dfsPathCheck(G,v,dest):
    if v=dest then                // found edge from v to dest
        return true
    else
        for all (v,w)∈edges(G) do
            if visited[w]=-1 then
                visited[w]=v
                if dfsPathCheck(G,w,dest) then
                    return true    // found path via w to dest
                end if
            end if
        end for
    end if
    return false                // no path from v to dest
```

Exercise #11: Depth-first Traversal (ii)

72/83

Show the DFS order in which we visit vertices in this graph when searching for a path from 0 to 6:



Consider neighbours in ascending order

0	0	3	5	3	1	5	4	7	8
---	---	---	---	---	---	---	---	---	---

4

5

(empty) → 0 → 5 → 5 → 5 → 5 → 5 → 5

Breadth-first Search

77/83

Basic approach to breadth-first search (BFS):

- visit and mark current vertex
- visit all neighbours of current vertex
- then consider neighbours of neighbours

Notes:

- tricky to describe recursively
- a minor variation on non-recursive DFS search works
⇒ switch the *stack* for a *queue*

... Breadth-first Search

78/83

BFS algorithm (records visiting order, marks vertices as visited when put *on* queue):

visited[] // array of visiting orders, indexed by vertex 0..nV-1

```
findPathBFS(G,src,dest):
  Input graph G, vertices src,dest

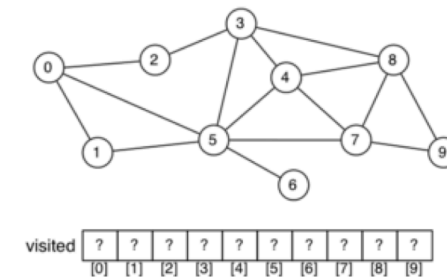
  for all vertices v∈G do
    visited[v]=-1
  end for
  enqueue src into new queue q
  visited[src]=src
  found=false
  while not found and q is not empty do
    dequeue v from q
    if v=dest then
      found=true
    else
      for each (v,w)∈edges(G) such that visited[w]=-1 do
        enqueue w into q
        visited[w]=v
      end for
    end if
  end while
  if found then
    display path in dest..src order
  end if
```

Uses standard queue operations (enqueue, dequeue, check if empty)

Exercise #13: Breadth-first Traversal

79/83

Show the BFS order in which we visit vertices in this graph when searching for a path from 0 to 6:



Consider neighbours in ascending order

0	0	0	2	5	0	5	5 (corrected)	3	-1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Path: 6-5-0

... Breadth-first Search

81/83

Time complexity of BFS: $O(V+E)$ (adjacency list representation, same as DFS)

BFS finds a "shortest" path

- based on minimum # edges between *src* and *dest*.
- stops with first-found path, if there are multiple ones

In many applications, edges are weighted and we want path

- based on minimum sum-of-weights along path *src* .. *dest*

We discuss weighted/directed graphs later.

Tips for Week 6 Problem Set

82/83

Main theme: *Graphs*

- Test your understanding of basic graph properties
- Exercise 2: Write a graph ADT client

- Compare the efficiency of different graph representations
 - *Exercise 5: Check your understanding of BFS and DFS*
 - Challenge exercise: find a solution, need not be efficient
-

Summary

83/83

- Graph terminology
 - vertices, edges, vertex degree, connected graph, tree
 - path, cycle, clique, spanning tree, spanning forest
 - Graph representations
 - array of edges
 - adjacency matrix
 - adjacency lists
 - Graph traversal
 - depth-first search (DFS)
 - breadth-first search (BFS)
 - Suggested reading (Sedgewick):
 - graph representations ... Ch.17.1-17.5
 - graph search ... Ch.18.1-18.3,18.7
-