

Week 5: Graph Data Structures

Graph Definitions

Graphs

2/106

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 (last week; COMP9021)
- trees ... branched hierarchy of items (COMP9021)

Graphs are more general ... allow arbitrary connections

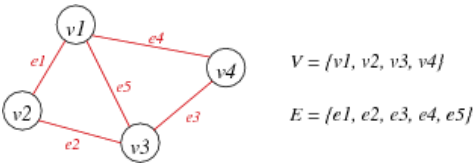
... Graphs

3/106

A graph $G = (V, E)$

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

Example:



... Graphs

4/106

A real example: Australian road distances

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

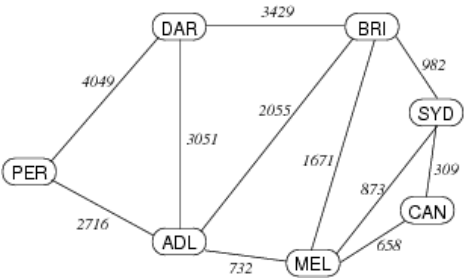
Brisbane	2055	-	1291	3429	1671	4771	982
Canberra	1390	1291	-	4441	658	4106	309
Darwin	3051	3429	4441	-	3783	4049	4411
Melbourne	732	1671	658	3783	-	3448	873
Perth	2716	4771	4106	4049	3448	-	3972
Sydney	1605	982	309	4411	873	3972	-

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

... Graphs

5/106

Alternative representation of above:



... Graphs

6/106

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

7/106

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

8/106

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

10/106

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

11/106

Path: a sequence of vertices where

- each vertex has an edge to its predecessor

Simple path: a path where

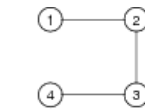
- all vertices and edges are different

Cycle: a path

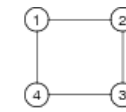
- that is simple except last vertex = first vertex

Length of path or cycle:

- #edges



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



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

... Graph Terminology

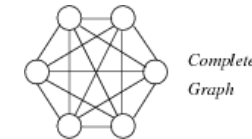
12/106

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

13/106

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

14/106

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

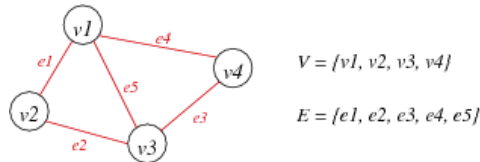
- is a subgraph of G containing all of V
- and is a single tree (connected, no cycles)

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

15/106



$$V = \{v1, v2, v3, v4\}$$

$$E = \{e1, e2, e3, e4, e5\}$$

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

17/106

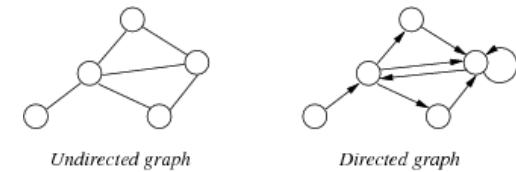
Undirected graph

- $edge(u, v) = edge(v, u)$, 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:



Undirected graph

Directed graph

... Graph Terminology

18/106

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

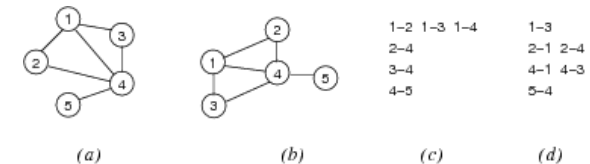
Graph Representations

20/106

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

21/106

We will discuss three different graph data structures:

1. Array of edges
2. Adjacency matrix

Array-of-edges Representation

22/106

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

23/106

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

24/106

Edge insertion

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

... Array-of-edges Representation

25/106

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

26/106

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

27/106

Assuming an array-of-edges representation ...

Write an algorithm to output all edges of the graph

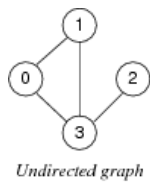
```
show(g):
|   Input   graph g
|
|   for all i=0 to g.nE-1 do
|       print g.edges[i]
|   end for
```

Time complexity: $O(E)$

Adjacency Matrix Representation

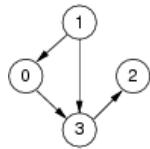
29/106

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

30/106

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

31/106

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

32/106

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

33/106

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

34/106

Assuming an adjacency matrix representation ...

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

... Adjacency Matrix Representation

35/106

```

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:

36/106

Analyse storage cost and time complexity of adjacency matrix representation

Storage cost: $O(V^2)$

If the graph is sparse, most storage is wasted.

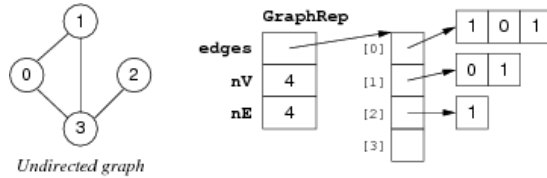
Cost of operations:

- 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

38/106

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



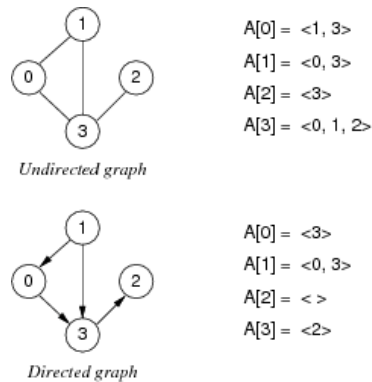
New storage cost: $V-1$ int ptrs + $V(V+1)/2$ ints (but still $O(V^2)$)

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

Adjacency List Representation

39/106

For each vertex, store linked list of adjacent vertices:



... Adjacency List Representation

40/106

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

41/106

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

42/106

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

43/106

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:

44/106

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

46/106

	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

48/106

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

49/106

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;

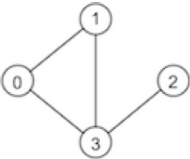
// operations on graphs
Graph newGraph(int V); // new graph with V vertices
void insertEdge(Graph, Edge);
void removeEdge(Graph, Edge);
bool adjacent(Graph, Vertex, Vertex); /* is there an edge
                                         between two vertices */
void freeGraph(Graph);
```

Exercise #7: Graph ADT Client

50/106

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)

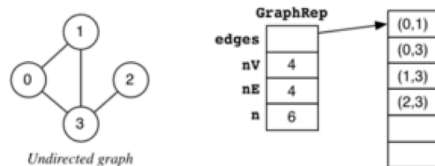
52/106

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)

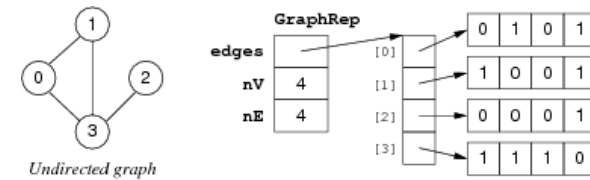
53/106

Implementation of GraphRep (adjacency-matrix representation)

```

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

```



... Graph ADT (Adjacency Matrix)

54/106

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)

55/106

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)

56/106

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)

58/106

Implementation of GraphRep (adjacency-list representation)

```

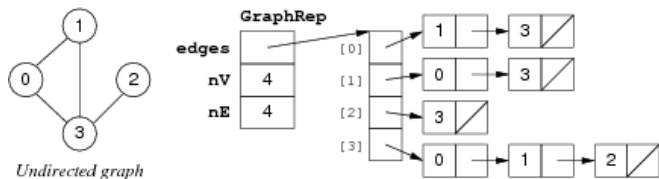
typedef struct GraphRep {
    Node **edges; // array of lists
    int    nV;    // #vertices
    int    nE;    // #edges
} GraphRep;

```

```

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

```



Exercise #9: Checking Neighbours (ii)

59/106

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

Problems on Graphs

Problems on Graphs

62/106

What kind of problems do we want to solve on/via graphs?

- is the graph fully-connected?
- can we remove an edge and keep it fully-connected?
- is one vertex reachable starting from some other vertex?
- which vertices are reachable from v ? (transitive closure)
- is there a cycle that passes through all vertices? (circuit)
- what is the cheapest cost path from v to w ?
- is there a tree that links all vertices? (spanning tree)
- what is the minimum spanning tree?
- what is the maximal flow through a graph?
- ...
- can a graph be drawn in a plane with no crossing edges? (planar graphs)
- are two graphs "equivalent"? (isomorphism)

Graph Algorithms

63/106

In this course we examine algorithms for

- graph traversal (simple graphs)
- reachability (directed graphs)
- minimum spanning trees (weighted graphs)
- shortest path (weighted graphs)
- maximum flow (weighted graphs)

Graph Traversal

Finding a Path

65/106

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

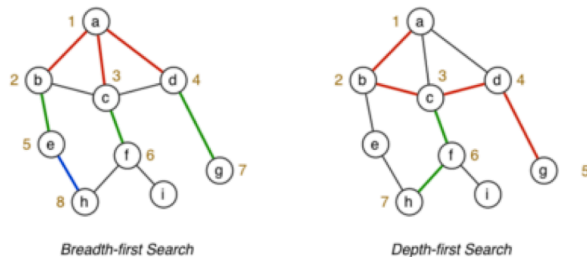
Two strategies for graph traversal/search: *depth-first*, *breadth-first*

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

... Finding a Path

66/106

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

67/106

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

68/106

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

mark all vertices in G as unvisited

return **dfsPathCheck($G, src, dest$)**

dfsPathCheck($G, v, dest$):

mark v as visited

if $v=dest$ **then** // found dest

return true

else

for all $(v, w) \in \text{edges}(G)$ **do**

if w has not been visited **then**

return **dfsPathCheck($G, w, dest$)** // found path via w to dest

end if

end for

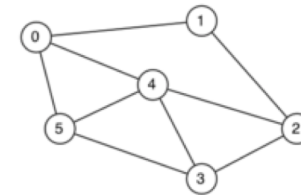
end if

return false // no path from v to dest

Exercise #10: Depth-first Traversal (i)

69/106

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



Consider neighbours in ascending order

Answer:

0 - 1 - 2 - 3 - 4 - 5

... Depth-first Search

71/106

Cost analysis:

- all vertices marked as unvisited, 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)

- the larger of V, E determines the complexity

... Depth-first Search

72/106

Note how different graph data structures affect cost:

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

73/106

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

74/106

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

```
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
```

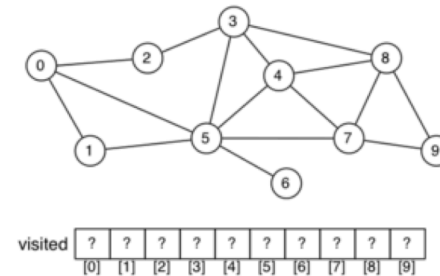
```
dfsPathCheck(G,v,dest):  
    if v=dest then                // found edge from v to dest  
        return true
```

```
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)

75/106

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

Path: 6-5-1-0

... Depth-first Search

77/106

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  
  
    mark all vertices in G as unvisited  
    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

```

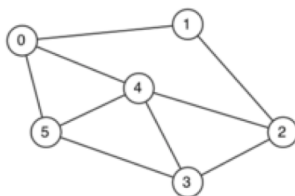
Uses standard stack operations (push, pop, check if empty)

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)

78/106

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



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

				4	5		
			3	5	5	5	
	1	2	4	4	4	4	
	4	4	4	4	4	4	
(empty)	→ 0	→ 5	→ 5	→ 5	→ 5	→ 5	→ 5

Breadth-first Search

80/106

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

81/106

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

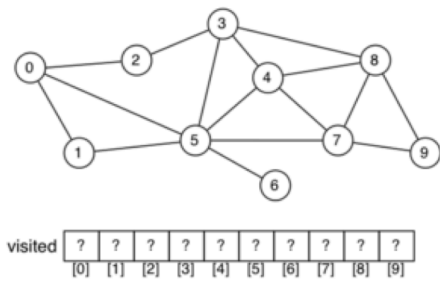
```

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

Exercise #13: Breadth-first Traversal

82/106

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

Path: 6-5-0

... Breadth-first Search

84/106

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*

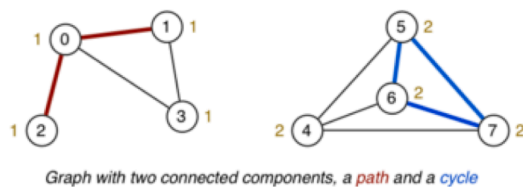
We discuss weighted/directed graphs later.

Other DFS Examples

85/106

Other problems to solve via DFS graph search

- checking for the existence of a cycle
- determining which connected component each vertex is in



Exercise #14: Buggy Cycle Check

86/106

A graph has a cycle if

- it has a path of length > 1
- with start vertex *src* = end vertex *dest*
- and without using any edge more than once

We are not required to give the path, just indicate its presence.

The following DFS cycle check has two bugs. Find them.

```
hasCycle(G):
|   Input  graph G
|   Output true if G has a cycle, false otherwise
|
|   choose any vertex v ∈ G
|   return dfsCycleCheck(G, v)
|
dfsCycleCheck(G, v):
|   mark v as visited
|   for each (v, w) ∈ edges(G) do
|   |   if w has been visited then // found cycle
|   |       return true
|   |   else if dfsCycleCheck(G, w) then
|   |       return true
|   end for
|   return false // no cycle at v
```

1. Only one connected component is checked.
2. The loop

for each (v, w) ∈ edges(G) do

should exclude the neighbour of v from which you just came, so as to prevent a single edge *w-v* from being classified as a cycle.

Computing Connected Components

88/106

Problems:

- how many connected subgraphs are there?
- are two vertices in the same connected subgraph?

Both of the above can be solved if we can

- build an array, one element for each vertex V
- indicating which connected component V is in

- componentOf [] ... array [0..nV-1] of component IDs

... Computing Connected Components

89/106

Algorithm to assign vertices to connected components:

```
components(G):
  Input graph G

  for all vertices v∈G do
    componentOf[v] = -1
  end for
  compID = 0
  for all vertices v∈G do
    if componentOf[v] = -1 then
      dfsComponents(G, v, compID)
      compID = compID + 1
    end if
  end for

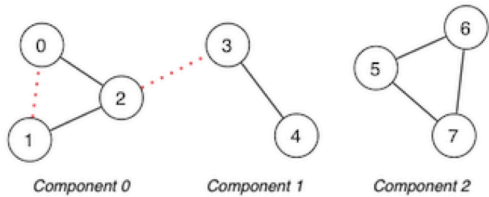
dfsComponents(G, v, id):
  componentOf[v] = id
  for all vertices w adjacent to v do
    if componentOf[w] = -1 then
      dfsComponents(G, w, id)
    end if
  end for
```

Exercise #15: Connected components

90/106

Trace the execution of the algorithm

1. on the graph shown below
2. on the same graph but with the dotted edges added



Consider neighbours in ascending order

1.

-1	-1	-1	-1	-1	-1	-1	-1
0	-1	-1	-1	-1	-1	-1	-1
0	-1	0	-1	-1	-1	-1	-1
0	0	0	-1	-1	-1	-1	-1
0	0	0	1	-1	-1	-1	-1
...							
0	0	0	1	1	2	2	2

2.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
-1	-1	-1	-1	-1	-1	-1	-1
0	-1	-1	-1	-1	-1	-1	-1
0	0	-1	-1	-1	-1	-1	-1
0	0	0	-1	-1	-1	-1	-1
...							
0	0	0	0	0	1	1	1

Hamiltonian and Euler Paths

Hamiltonian Path and Circuit

93/106

Hamiltonian path problem:

- find a path connecting two vertices v, w in graph G
- such that the path includes each *vertex* exactly once

If $v = w$, then we have a *Hamiltonian circuit*

Simple to state, but difficult to solve (*NP*-complete)

Many real-world applications require you to visit all vertices of a graph:

- Travelling salesman
- Bus routes
- ...

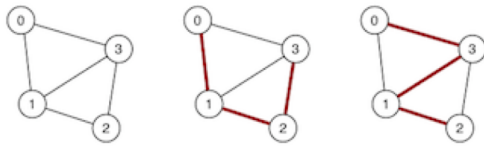
Problem named after Irish mathematician, physicist and astronomer Sir William Rowan Hamilton (1805 — 1865)

... Hamiltonian Path and Circuit

94/106

Graph and two possible Hamiltonian paths:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]



... Hamiltonian Path and Circuit

95/106

Approach:

- generate all possible simple paths (using e.g. DFS)
- keep a counter of vertices visited in current path
- stop when find a path containing V vertices

Can be expressed via a recursive DFS algorithm

- similar to simple path finding approach, except
 - keeps track of path length; succeeds if length = v
 - resets "visited" marker after unsuccessful path

... Hamiltonian Path and Circuit

96/106

Algorithm for finding Hamiltonian path:

visited[] // array [0.. $nV-1$] to keep track of visited vertices

```
hasHamiltonianPath(G,src,dest):
  for all vertices v∈G do
    visited[v]=false
  end for
  return hamiltonR(G,src,dest,#vertices(G)-1)
```

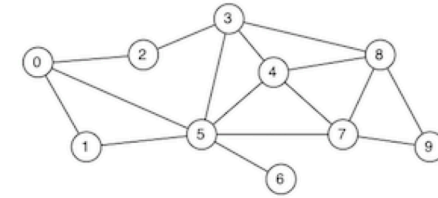
```
hamiltonR(G,v,dest,d):
  Input G      graph
         v      current vertex considered
         dest   destination vertex
         d      distance "remaining" until path found

  if v=dest then
    if d=0 then return true else return false
  else
    mark v as visited
    for each unvisited neighbour w of v in G do
      if hamiltonR(G,w,dest,d-1) then
        return true
      end if
    end for
  end if
  mark v as unvisited           // reset visited mark
  return false
```

Exercise #16: Hamiltonian Path

97/106

Trace the execution of the algorithm when searching for a Hamiltonian path from 1 to 6:



Consider neighbours in ascending order

1-0-2-3-4-5-6	$d \neq 0$
1-0-2-3-4-5-7-8-9	no unvisited neighbour
1-0-2-3-4-5-7-9-8	no unvisited neighbour
1-0-2-3-4-7-5-6	$d \neq 0$
1-0-2-3-4-7-8-9	no unvisited neighbour
1-0-2-3-4-7-9-8	no unvisited neighbour
1-0-2-3-4-8-7-5-6	$d \neq 0$
1-0-2-3-4-8-7-9	no unvisited neighbour
1-0-2-3-4-8-9-7-5-6	✓

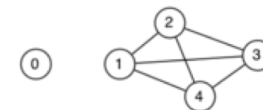
Repeat on your own with $src=0$ and $dest=6$

... Hamiltonian Path and Circuit

99/106

Analysis: worst case requires $(V-1)!$ paths to be examined

Consider a graph with isolated vertex and the rest fully-connected



Checking $hasHamiltonianPath(g,x,0)$ for any x

- requires us to consider every possible path
- e.g 1-2-3-4, 1-2-4-3, 1-3-2-4, 1-3-4-2, 1-4-2-3, ...
- starting from any x , there are $3!$ paths $\Rightarrow 4!$ total paths
- there is no path of length 5 in these $(V-1)!$ possibilities

There is no known simpler algorithm for this task $\Rightarrow NP$ -hard.

Note, however, that the above case could be solved in constant time if we had a fast check for 0 and x being in the same connected component

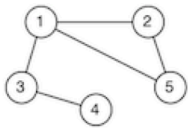
Euler Path and Circuit

100/106

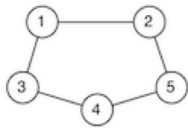
Euler path problem:

- find a path connecting two vertices v,w in graph G
- such that the path includes each *edge* exactly once
(note: the path does not have to be simple \Rightarrow can visit vertices more than once)

If $v = w$, then we have an *Euler circuit*



Euler Path: 4-3-1-5-2-1



Euler Circuit: 1-2-5-4-3-1

Many real-world applications require you to visit all edges of a graph:

- Postman
- Garbage pickup
- ...

Problem named after Swiss mathematician, physicist, astronomer, logician and engineer Leonhard Euler (1707 — 1783)

... Euler Path and Circuit

101/106

One possible "brute-force" approach:

- check for each path if it's an Euler path
- would result in factorial time performance

Can develop a better algorithm by exploiting:

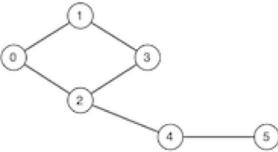
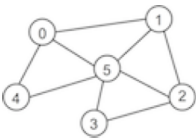
Theorem. A graph has an Euler circuit if and only if it is connected and all vertices have even degree

Theorem. A graph has a non-circuitous Euler path if and only if it is connected and exactly two vertices have odd degree

Exercise #17: Euler Paths and Circuits

102/106

Which of these two graphs have an Euler path? an Euler circuit?



No Euler circuit

Only the second graph has an Euler path, e.g. 2-0-1-3-2-4-5

... Euler Path and Circuit

104/106

Assume the existence of `degree(g, v)` (degree of a vertex, cf. problem set week 5)

Algorithm to check whether a graph has an Euler path:

```
hasEulerPath(G,src,dest):
    Input graph G, vertices src,dest
    Output true if G has Euler path from src to dest
           false otherwise

    if src≠dest then           // non-circuitous path
        if degree(G,src) or degree(G,dest) is even then
            return false
        end if
    else if degree(G,src) is odd then // circuit
        return false
    end if
    for all vertices v∈G do
        if v≠src and v≠dest and degree(G,v) is odd then
            return false
        end if
    end for
    return true
```

... Euler Path and Circuit

105/106

Analysis of `hasEulerPath` algorithm:

- assume that connectivity is already checked
- assume that degree is available via $O(1)$ lookup
- single loop over all vertices $\Rightarrow O(V)$

If degree requires iteration over vertices

- cost to compute degree of a single vertex is $O(V)$
- overall cost is $O(V^2)$

⇒ problem tractable, even for large graphs (unlike Hamiltonian path problem)

For the keen, a linear-time (in the number of edges, E) algorithm to compute an Euler path is described in [Sedgewick] Ch.17.7.

Summary

106/106

- 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)
 - cycle check, connected components
 - Hamiltonian paths/circuits, Euler paths/circuits
- Suggested reading (Sedgewick):
 - graph representations ... Ch. 17.1-17.5
 - Hamiltonian/Euler paths ... Ch. 17.7
 - graph search ... Ch. 18.1-18.3, 18.7