Graph Definitions

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 3, COMP9021)
- trees … branched hierarchy of items (COMP9021)

Graphs are more general … allow arbitrary connections

A graph \( G = (V, E) \)
- \( V \) is a set of vertices
- \( E \) is a set of edges (subset of \( V \times V \))

Example:

A real example: Australian road distances

<table>
<thead>
<tr>
<th></th>
<th>Adelaide</th>
<th>Brisbane</th>
<th>Canberra</th>
<th>Darwin</th>
<th>Melbourne</th>
<th>Perth</th>
<th>Sydney</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adelaide</td>
<td>-</td>
<td>2055</td>
<td>1390</td>
<td>3051</td>
<td>732</td>
<td>2716</td>
<td>1605</td>
</tr>
</tbody>
</table>

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

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

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

A graph with \(V\) vertices has at most \(V(V-I)/2\) 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**

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:

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

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

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 $e$

Synonyms:

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

**Length of path or cycle:**

- #edges

**Connected graph**

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

**Complete graph $K_V$**

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

**Tree**: connected (sub)graph with no cycles

**Spanning tree**: tree containing all vertices

**Clique**: complete subgraph

Consider the following single graph:

This graph has 25 vertices, 32 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**
A spanning tree of a 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 a 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

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

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

... Graph Terminology

Undirected graph
- \( \text{edge}(u,v) = \text{edge}(v,u), \) no self-loops  (i.e. no \( \text{edge}(v,v) \))

Directed graph
- \( \text{edge}(u,v) \neq \text{edge}(v,u), \) can have self-loops  (i.e. \( \text{edge}(v,v) \))

Examples:

Graph Data Structures

We will discuss three different graph data structures:

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

Array-of-edges Representation

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..V-1.

---

**Array-of-edges Representation**

**Graph initialisation**

```plaintext
newGraph(V):
| Input      | number of nodes V |
| Output     | new empty graph   |

i=0
while i<g.nE ∧ (v,w)≠g.edges[i] do
  i=i+1
end while
if i<g.nE then                   // (v,w) found
  g.edges[i]=g.edges[g.nE-1] // replace by last edge in array
  g.nE=g.nE-1
end if
```

**Cost Analysis**

**Storage cost:** $O(E)$

**Cost of operations:**

- Initialisation: $O(1)$
- Insert edge: $O(E)$ (assuming edge array has space)
- 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 ⇒ still $O(E)$

If we maintain edges in order:

- Use binary search to find edge ⇒ $O(\log E)$

**Exercise #3: Array-of-edges Representation**

Assuming an array-of-edges representation ...

Write an algorithm to output all edges of the graph

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

---

**Adjacency Matrix Representation**

Edges represented by a $V \times V$ matrix.
Adjacency Matrix Representation

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

Graph initialisation

def 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

Edge insertion

def 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

Edge removal

def 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

Assuming an adjacency matrix representation ...

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

Exercise #5:

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

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

For each vertex, store linked list of adjacent vertices:

- $A[0] = \langle 1, 3 \rangle$
- $A[1] = \langle 0, 3 \rangle$
- $A[2] = \langle 0, 3 \rangle$
- $A[3] = \langle 0, 1, 2 \rangle$

... Adjacency List Representation

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

... Adjacency List Representation

Graph initialisation

```plaintext
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

Edge insertion:

```plaintext
insertEdge(g,(v,w)):
  Input graph g, edge (v,w)

  if ¬inLL(g.edges[v],w) then  // (v,w) not in graph
    insertLL(g.edges[v],w)
    insertLL(g.edges[w],v)
    g.nE=g.nE+1
  end if
```

... Adjacency List Representation

Edge removal:

```plaintext
removeEdge(g,(v,w)):
  Input graph g, edge (v,w)

  if inLL(g.edges[v],w) then  // (v,w) in graph
    deleteLL(g.edges[v],w)
    deleteLL(g.edges[w],v)
    g.nE=g.nE-1
  end if
```

Exercise #6:

Advantages
- relatively easy to implement in languages like C
- can represent graphs and digraphs
- memory efficient if $E:V$ relatively small
Analyse storage cost and time complexity of adjacency list representation

**Storage cost:** $O(E)$

**Cost of operations:**
- initialisation: $O(V)$ (initialise V lists)
- insert edge: $O(1)$ (insert one vertex into list)
- delete edge: $O(E)$ (need to find vertex in list)

If vertex lists are sorted
- insert requires search of list $\Rightarrow O(E)$
- delete always requires a search, regardless of list order

**Comparison of Graph Representations**

<table>
<thead>
<tr>
<th></th>
<th>array of edges</th>
<th>adjacency matrix</th>
<th>adjacency list</th>
</tr>
</thead>
<tbody>
<tr>
<td>space usage</td>
<td>$E$</td>
<td>$V^2$</td>
<td>$V+E$</td>
</tr>
<tr>
<td>initialise</td>
<td>$I$</td>
<td>$V^2$</td>
<td>$V$</td>
</tr>
<tr>
<td>insert edge</td>
<td>$E$</td>
<td>$I$</td>
<td>$I$</td>
</tr>
<tr>
<td>remove edge</td>
<td>$E$</td>
<td>$I$</td>
<td>$E$</td>
</tr>
</tbody>
</table>

Other operations:
- disconnected(v)? $E$
- isConnected(x,y)? $E \cdot \log V$
- isPath(x,y)? $V^2$
- copy graph $E$
- destroy graph $I$

**Graph Abstract Data Type**

**Graph ADT**

- 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 (Array of Edges)**

Implementation of GraphRep (array-of-edges representation)

```c
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 (Array of Edges)**
Implementation of graph initialisation (array-of-edges representation)

Graph newGraph(int V) {
    assert(V >= 0);
    Graph g = malloc(sizeof(GraphRep));   assert(g != NULL);
    g->nV = V; g->nE = 0;
    // allocate enough memory for edges
    g->n = Enough;
    g->edges = malloc(g->n*sizeof(Edge)); assert(g->edges != NULL);
    return g;
}

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

... Graph ADT (Array of Edges)

Implementation of edge insertion/removal (array-of-edges representation)

// check if two edges are equal
bool eq(Edge e1, Edge e2) {    
   return ( (e1.v == e2.v && e1.w == e2.w)  
   || (e1.v == e2.w && e1.w == e2.v) );
}

void insertEdge(Graph g, Edge e) {    // ensure that g exists and array of edges isn’t full
   assert(g != NULL && g->nE < g->n);
   int i = 0;
   while (i < g->nE && !eq(e,g->edges[i]))
      i++;
   if (i == g->nE)                     // edge e not found
      g->edges[g->nE++] = e;
}

void removeEdge(Graph g, Edge e) {    // ensure that g exists
   assert(g != NULL);
   int i = 0;
   while (i < g->nE && !eq(e,g->edges[i]))
      i++;
   if (i < g->nE)                      // edge e found
      g->edges[i] = g->edges[--g->nE];
}

... Exercise #7: Checking Neighbours (i)

Assuming an array-of-edges representation ...

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

bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL);
   Edge e;
   e.v = x; e.w = y;
   int i = 0;
   while (i < g->nE) {
      if (eq(e,g->edges[i]))    // edge found
         return true;
      i++;
   }
   return false;                 // edge not found
}

Graph ADT (Adjacency Matrix)

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;
}

undefined

... Graph ADT (Adjacency Matrix)

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

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

void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));
   if (!inLL(g->edges[e.v], e.w)) {    // edge e not in graph
      g->edges[e.v] = insertLL(g->edges[e.v], e.w);
      g->edges[e.w] = insertLL(g->edges[e.w], e.v);
      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--;
   }
}

bool adjacent(Graph g, Vertex x, Vertex y) {
   return (g->edges[x][y] != 0);
}

Exercise #8: Checking Neighbours (ii)

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) { … }
void removeEdge(Graph g, Edge e) {
    assert(g != NULL && validV(g,e.v) && validV(g,e.w));
    if (inLL(g->edges[e.v], e.w)) { // edge e in graph
        g->edges[e.v] = deleteLL(g->edges[e.v], e.w);
        g->edges[e.w] = deleteLL(g->edges[e.w], e.v);
        g->nE--;
    }
}

inLL, insertLL, deleteLL are standard linked list operations (as discussed in week 3)

Exercise #9: Checking Neighbours (iii)

Assuming an adjancency list representation ...

bool adjacent(Graph g, Vertex x, Vertex y) {
    return inLL(g->edges[x], y);
}

Exercise #10: Graph ADT Client

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;
}

Summary

- 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

Suggested reading:
- Sedgewick, Ch.17.1-17.5

Produced: 29 Aug 2017