Graphs

Computing 2 COMP1927 16x1
Sedgewick Part 5: Chapter 17
WHAT ARE GRAPHS

- 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 we've seen so far
  - Lists…linear sequence of items
  - trees ... branched hierarchy of items
  - These are both special cases of graphs.
- Graphs are more general ... allow arbitrary connections.
DEFINITION OF A GRAPH

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

Example:

![Graph Diagram]

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

$E = \{e1, e2, e3, e4, e5\}$
## Other Graph Application Examples

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
<td>Telephones, Computers</td>
<td>cables</td>
</tr>
<tr>
<td>Games</td>
<td>Board positions</td>
<td>Legal moves</td>
</tr>
<tr>
<td>Social networks</td>
<td>People</td>
<td>Friendships</td>
</tr>
<tr>
<td>Scheduling</td>
<td>Tasks</td>
<td>Precedence Constraints</td>
</tr>
<tr>
<td>Circuits</td>
<td>Gates, Registers, Processors</td>
<td>Wires</td>
</tr>
<tr>
<td>Transport</td>
<td>Intersections/ airports</td>
<td>Roads, flights</td>
</tr>
</tbody>
</table>
A REAL EXAMPLE:
AUSTRALIAN ROAD DISTANCES

<table>
<thead>
<tr>
<th>Dist</th>
<th>Adel</th>
<th>Bris</th>
<th>Can</th>
<th>Dar</th>
<th>Melb</th>
<th>Perth</th>
<th>Syd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adel</td>
<td>-</td>
<td>2055</td>
<td>1390</td>
<td>3051</td>
<td>732</td>
<td>2716</td>
<td>1605</td>
</tr>
<tr>
<td>Bris</td>
<td>2055</td>
<td>-</td>
<td>1291</td>
<td>3429</td>
<td>1671</td>
<td>4771</td>
<td>982</td>
</tr>
<tr>
<td>Can</td>
<td>1390</td>
<td>1291</td>
<td>-</td>
<td>4441</td>
<td>658</td>
<td>4106</td>
<td>309</td>
</tr>
<tr>
<td>Dar</td>
<td>3051</td>
<td>3429</td>
<td>4441</td>
<td>-</td>
<td>3783</td>
<td>4049</td>
<td>4411</td>
</tr>
<tr>
<td>Melb</td>
<td>732</td>
<td>1671</td>
<td>658</td>
<td>3783</td>
<td>-</td>
<td>3448</td>
<td>873</td>
</tr>
<tr>
<td>Perth</td>
<td>2716</td>
<td>4771</td>
<td>4106</td>
<td>4049</td>
<td>3448</td>
<td>-</td>
<td>3972</td>
</tr>
<tr>
<td>Syd</td>
<td>1605</td>
<td>982</td>
<td>309</td>
<td>4411</td>
<td>873</td>
<td>3972</td>
<td>-</td>
</tr>
</tbody>
</table>
A real graph example:

- Alternative representation of Australian roads:
**Graphs**

- Questions we might ask about a graph
  - is there a way to get from item A to item B?
  - what’s the best way?
  - which items are connected?
- **Graph algorithms** are in general significantly more difficult than list or tree processing
  - no implicit order of the items
  - graphs can contain cycles
  - concrete representation is less obvious
  - complexity of algorithms depend connection complexity
**SIMPLE GRAPHS**

Depending on the application, graphs can have different properties:

- undirected
- directed
- multigraph
- weighted

At this point, we will only consider **simple graphs** which are characterised by:

- a set of vertices, and
- a set of undirected edges that connect pairs of vertices
  - no self loops
  - no parallel edges
**Simple Graph: Vertices and Edges**

*In our example graph:*
- \( V \) (number of vertices): 7
  - *From 0 to 6*
  - A 7-vertex graph
- \( E \) (number of edges): 11

How many edges can a 7-vertex simple graph have?
- \( 7 \times (7-1)/2 = 21 \)
SIMPLE GRAPH: VERTICES AND EDGES

- $E \leq V^*(V-1)/2$
  - If $E$ is closer to $V^2$ the graph is dense
  - If $E$ is closer to $V$ the graph is sparse
  - If $E$ is 0 we have a set

- These properties may affect
  - choice of data structures to represent the graph and
  - the algorithms used
**Graphs: Terminology**

- The **degree** of a vertex is the number of edges from the vertex.
- A **complete graph** is a graph where every vertex is connected to all the other vertices.
  - \( E = \frac{V(V-1)}{2} \)
  - The degree of every vertex is \( V-1 \)
**Graph Terminology**

- **adjacent**: two vertices, $v$ and $w$ are adjacent if there is an edge, $e$, between them.
- $e$ is **incident** on both $v$ and $w$.
**Graph Terminology**

**subgraph**: a subset of vertices with their associated edges
GRAPH TERMINOLOGY: PATHS

a path: a sequence of vertices where each one is connected to its predecessor 1,0,6,5

a graph is a tree if there is exactly one path between each pair of vertices

a path is simple if it doesn’t have any repeating vertices

a path is a cycle if it is simple apart from its first and last vertex
GRAPH TERMINOLOGY

- A graph is a **connected graph**, if there is a path from every vertex to every other vertex in the graph.
A graph that is not connected consists of a set of connected components, which are maximally connected subgraphs.
GRAPH TERMINOLOGY

- A spanning tree of a graph is a subgraph that contains all the vertices and is a single tree
A spanning forest of a graph is a subgraph that contains all its vertices and is a set of trees.
CLIQUES

- Clique: complete subgraph
  - Clique containing vertices {A, G, H, J, K, M}
  - Another clique containing vertices {D, E, F, L}
Graph Terminology

- Hamilton path
  - A simple path that connects two vertices that visits every vertex in the graph exactly once.
  - If the path is from a vertex back to itself it is called a Hamilton tour.
EXERCISE:
Does this have a Hamilton path?
**Graph Terminology**

- **Euler path**
  - A path that connects two given vertices using each edge in the path exactly once.
  - If the path is from a vertex back to itself it is an Euler tour.
EXERCISE: DOES THIS HAVE AN EULER PATH?

- A graph has an Euler tour if and only if it is connected and all vertices are of even degree.
- A graph has an Euler path if and only if it is connected and exactly 2 vertices are of odd degree.
**Directed Graphs**

- If the edges in a graph are directed, the graph is called a **directed graph or digraph**
  - A digraph with $V$ vertices can have at most $V^2$ edges
    - Can have self loops
    - $\text{edge}(u,v) \neq \text{edge}(v,u)$
  - A digraph is a tree if there is one vertex which is connected to all other vertices, and there is at most one path between any two vertices

- Unless specified, we assume graphs are undirected in this course.
UNDIRECTED VS DIRECTED GRAPHS

Undirected graph

Directed graph
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)
  - eg. Transport – may be able to get to new location by bus or train or ferry etc…
**DEFINING GRAPHS**

- need some way of identifying vertices and their connections
- Below are 4 representations of the **same** graph
Graph ADT

- **Data:**
  - set of edges,
  - set of vertices

- **Operations:**
  - building: create graph, create edge, add edge
  - deleting: remove edge, drop whole graph
  - scanning: get edges, copy, show

- **Notes:** In our graphs
  - set of vertices is fixed when graph initialised
  - we treat vertices as ints, but could be Items
ADT INTERFACE FOR GRAPHS

- Vertices and Edges

typedef int Vertex;

// edge representation
typedef struct edge {
    Vertex v;
    Vertex w;
} Edge;

// edge construction
Edge mkEdge (Vertex v, Vertex w);
ADT Interface or Graphs

- **Graph basics:**
  ```c
  // graph handle
typedef struct GraphRep *Graph;

  // create a new graph
  Graph graphInit (int noOfVertices);
  int validV(Graph g, Vertex v); // validity check
  ```

- **Graph inspection and manipulation:**
  ```c
  void insertEdge (Graph g, Edge e);
  void removeEdge(Graph g, Edge e);
  Edge * edges (Graph g, int * nE);
  int isAdjacent(Graph g, Vertex v, Vertex w);
  int numV(Graph g);
  int numE(Graph g);
  ```

- **Whole graph operations:**
  ```c
  Graph GRAPHcopy (Graph g);
  void GRAPHdestroy (Graph g);
  ```
Adjacency matrix representation

- Edges represented by a VxV matrix

**Undirected graph**

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Directed graph**

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```
ADJACENCY MATRIX REPRESENTATION

Advantages
- easily implemented in C 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, memory-inefficient
**Adjacency Matrix Implementation**

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

*Undirected graph*
**Adjacency Matrix Storage Optimisation**

- **Storage cost:**
  - $V \text{ int ptrs} + V^2 \text{ ints}$ If the graph is sparse, most storage is wasted.

- **A storage optimisation:**
  - If undirected, store only top-right part of matrix.
  - New storage cost: $V-1 \text{ int ptrs} + V(V+1)/2 \text{ ints}$ (but still $O(V^2)$)
  - Requires us to always use edges $(v,w)$ such that $v < w$.

---

Undirected graph

```
1
\|\|\|\|\|
0 1 2 3
```

### GraphRep

```
<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>edges</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nV</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nE</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
1 0 1
0 1
1
```
Cost of operations on adjacency matrix

- 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)

See code for the implementation of these functions and their cost
- int isAdjacent(Graph g, Vertex v, Vertex w);
- Vertex * adjacentVertices(Graph g, Vertex v, int * nV);

Exercise: write the functions and find the cost for
- Edge * edges (Graph g, int * nE);
**Adjacency List Representation**

- For each vertex, store linked list of adjacent vertices:

  **Undirected graph**

  0 -> 1, 2, 3
  1 -> 0, 2
  2 -> 0, 1
  3 -> 1

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

  **Directed graph**

  0 -> 1
  1 -> 0, 2
  2 -> 0
  3 -> 0, 1

  \[ A[0] = <3> \]
  \[ A[1] = <0, 3> \]
  \[ A[2] = <> \]
  \[ A[3] = <2> \]
**Adjacency List Representation**

- **Advantages**
  - relatively easy to implement in 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 Matrix Implementation**

typedef struct vNode *VList;
struct vNode { Vertex v; VList next; };
typedef struct GraphRep {
    int nV; // #vertices
    int nE; // #edges
    VList *edges; // array of lists
} GraphRep;

Undirected graph
Costs of Operations on Adjacency Lists

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

- If vertex lists are sorted insert requires search of list
  $\Rightarrow O(V)$

- If we do not want to allow parallel edges it is $O(V)$

- delete always requires a search, regardless of list order
COSTS OF OPERATIONS ON ADJACENCY LISTS

- See code for the implementation of these functions and their cost
  - int isAdjacent(Graph g, Vertex v, Vertex w);
  - Vertex * adjacentVertices(Graph g, Vertex v, int * nV);

- Exercise: write the functions and find the cost for
  - Edge * edges (Graph g, int * nE);
## Comparison of Different Graph Representations

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Matrix</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$V^2$</td>
<td>$V + E$</td>
</tr>
<tr>
<td>Initialise empty</td>
<td>$V^2$</td>
<td>$V$</td>
</tr>
<tr>
<td>Copy</td>
<td>$V^2$</td>
<td>$E$</td>
</tr>
<tr>
<td>Destroy</td>
<td>$V$</td>
<td>$E$</td>
</tr>
<tr>
<td>Insert edge</td>
<td>1</td>
<td>$V$</td>
</tr>
<tr>
<td>Find/Remove edge</td>
<td>1</td>
<td>$V$</td>
</tr>
<tr>
<td>Is v isolated?</td>
<td>$V$</td>
<td>1</td>
</tr>
<tr>
<td>Is adjacent</td>
<td>1</td>
<td>$V$</td>
</tr>
</tbody>
</table>