Graph Theory I

"Everything is a graph...
and a graph is everything."
– Benjamin Burton

Graph theory is, in simple terms, *the study of relationships between things.*

1 Introduction & terminology

A **graph** is a set of **vertices** ("vertex" singular, "nodes" for short) and a set of corresponding **edges**. Each edge describes a connection/link/relationship between a pair of vertices. Vertices are drawn as circles and edges are drawn as lines between those circles.

In an **undirected graph**, each edge is bidirectional – if an edge exists between u and v, then u is adjacent to v and v is also adjacent to u. A social network is an example of an undirected graph, because if I am friends with you than you are friends with me (hopefully).

In a **directed graph** ("digraph" for short), edges are unidirectional – if an edge exists between u and v, u is adjacent to v, but v is not adjacent to u (unless another edge from v to u also exists). Edges are drawn as lines with arrows at their endpoints to indicate direction. We can represent the results of a tennis tournament as a directed graph, where each vertex represents a tennis player and an edge from player u to v means "u has beaten v in the tournament". This is *not* an undirected graph; just because Roger Federer beat me in the tournament does not imply that I have also beaten him.

Terminology

- Vertex u is **adjacent** to vertex v if and only if an edge exists from u to v. The set of vertices adjacent to u are the **neighbours** of u. If u is adjacent to v then v is a neighbour of u.
- In complexity notation, V denotes the number of vertices and E denotes the number of edges. The time complexity of a graph algorithm looks like $O(E \log V)$ or $O(V^3)$.
- A **path** between vertices u and v is a sequence of *distinct* vertices $u, a_1, \ldots, a_m, v$ so that u is adjacent to a_1, a_m is adjacent to v and a_k is adjacent to a_{k+1} for $k = 1..m-1$.
- Vertex u is **connected** to vertex v if a path exists from u to v.
- A graph is **connected** if all pairs of vertices in the graph are connected to each other. In other words, a graph is connected if you can get from any vertex to any other vertex by following edges.

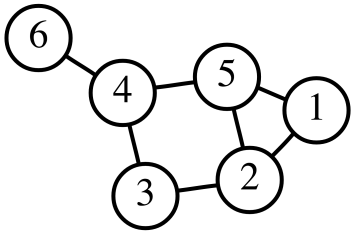


Fig. 1.1: an undirected graph

- Vertex 4 is adjacent to vertices 6, 5, 3.
- The neighbours of vertex 4 are the vertices $\{6, 5, 3\}$.
- Vertex 4 is connected to vertex 2 as we can follow edges from vertex 4 to reach vertex 2 (e.g. $4 \to 5 \to 2$ or $4 \to 3 \to 2$).

- A **subgraph** of G is a graph obtained by deleting edges or vertices (or both) from G.
- A **connected component** of G is a subgraph of G consisting of a vertex v in G as well as all vertices connected to v, and all edges between pairs of these vertices.
- A **cycle** is a sequence of vertices $a_1, \ldots, a_m$ for which a_k is adjacent to a_{k+1} for $k = 1 \ldots m-1$ and furthermore a_m is adjacent to a_1.
- The **degree** of a vertex v is the total number of edges incident to (entering or leaving) v. For a directed graph, the **in-degree** and **out-degree** of v is the number of edges into and out of v respectively.

Some practice questions

Exercise 1. 10 informatics students find themselves at December camp. Every student shakes hands exactly once with every other student. Consider the graph in which each vertex is a student and an edge between two vertices means those two students shake hands at December camp.

1. Is this graph directed or undirected?
2. If all the students are mutual strangers before the camp (informatics students are notoriously introverted), how many edges does the graph contain?

Exercise 2. Consider a graph representing a particular game of chess. Each vertex represents a state of the board. u is adjacent to v if and only at some point in the game, the board was in the state represented by u, then a player made a move, then the board was in the state represented by v.

1. Is this graph directed or undirected?
2. If the chess game concludes in n moves,

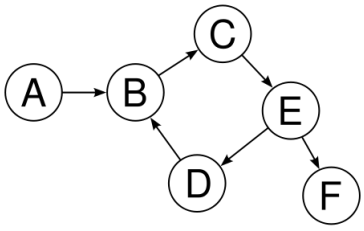


Fig. 1.2: a directed graph ("digraph")

- A is adjacent to B but B is not adjacent to A.
- The neighbours of E are $\{F, D\}$.
- C is connected to D and D is connected to C.
- D is connected to F (because we can follow $D \to B \to C \to E \to F$) but F is *not* connected to D.

(a) What is the most number of vertices that the graph can have? What is the most number of edges?

(b) (**) What is the least number of vertices that the graph can have? What is the least number of edges?

3. (**) Can there exist a pair of vertices u, v so that u is adjacent to v and v is adjacent to u?

2 Representing adjacency

Before we can begin manipulating the graph, we must first have a way of defining the graph and storing it in memory. Suppose the V vertices of a graph are numbered $1..V$ (if a numbering of vertices is not explicitly given in a problem, we usually assign a numbering anyway for convenience). To represent the graph in computer code, we need a data structure that tells us information about which vertices are adjacent to each other. There are 3 traditional ways of representing adjacency: **edge lists**, **adjacency matrices** and **adjacency lists**. Each its advantages and disadvantages.

Edge list

The edge list representation simply stores a list (array or linked list) of all the graph's E edges. Each edge is notated as (i, j) representing an edge from vertex i to vertex j. An edge list representation of the graph in fig. 1.1 is: $\langle (1,2), (1,5), (2,3), (2,5), (3,4), (4,5), (4,6) \rangle$.

i, j adjacent? $O(E)$. We do a linear scan for (i, j). If the graph is undirected, we also scan for (j, i).

Edge insertion/deletion: $O(1)/O(E)$ respectively. We insert by simply appending a new (i, j) pair to the list. We delete using standard array/linked list deletion methods.

Neighbours of i: $O(E)$. We linear scan for all edges of the form $(\square, i)$ or $(i, \square)$.

Memory usage: $O(E)$.

Adjacency matrix

The **adjacency matrix** for the graph is the $N \times N$ matrix whose entry in row i column j is the number of edges from the ith vertex to the jth vertex.[1]

In an undirected graph, if the ith vertex is adjacent to jth vertex than the jth vertex is also adjacent to the ith vertex. The adjacency matrix of an undirected graph is thus **symmetric**: the entry in (i, j) is equal to the entry in (j, i).

We can store an adjacency matrix simply as a 2-dimensional integer array `int adj[V][V];`[2]

The adjacency list for the graph in fig. 1.1 follows (note the symmetry along the top-left-to-bottom-right diagonal!)

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

i, j adjacent? $O(1)$. We simply verify that `adj[i][j] > 0`.

Edge insertion/deletion: $O(1)$. We insert and delete the edge (i, j) by incrementing and decrementing `adj[i][j]` respectively.

Neighbours of i: $O(V)$. We linear scan through all entries along the row for vertex i.

Memory usage: $O(V^2)$.

Although operations on an adjacency matrix are very fast, they require $O(V^2)$ memory. This makes adjacency matrices effectively useless when $V \gg 10^5$.

Adjacency list

The **adjacency list** is the linked list of the neighbours of each node, for every node. In C++, we can safely use vectors in place of linked lists. The traditional declaration of an adjacency list in C++, then, is `vector<int> adj[V];` (vectors are implemented as dynamic arrays – ask your data structures lecturer).

The adjacency list for the graph in fig. 1.1 is given below.

Vertex	Neighbours
1	$\langle 2, 5 \rangle$
2	$\langle 1, 3, 5 \rangle$
3	$\langle 2, 4 \rangle$
4	$\langle 3, 5, 6 \rangle$
5	$\langle 1, 2, 4 \rangle$
6	$\langle 4 \rangle$

[1] In the case of a **loop** on the ith vertex (an edge from the ith vertex back to the ith vertex), by convention we count the edge twice and so we set entry (i, i) in the adjacency matrix to 2.

[2] In informatics, it is rare that we encounter graphs featuring multiple edges between the same pairs of nodes, or edges from a node back to itself. Most adjacency matrices hence contain only the entries 0 or 1. In this case, you can use C++'s `bool adj[V][V];`

Algorithm 1 basic framework of recursive DFS

```
bool processed[V] = {false, false, ...};

void dfs(int v) {
   // process v here

   processed[v] = true;
   for (each u adjacent to v) {
      if (!processed[u]) {
         dfs(u);
      }
   }
}
```

i, j adjacent? $O(V)$. We verify that j is in `adj[i]`. This is proportional to the number of neighbours of i, which is usually much smaller than V.

Edge insertion/deletion: $O(1)/O(V)$ respectively. We insert/delete the element j from `adj[i]`.

Neighbours of i: $O(1)$. Simply return `adj[i]`.

Memory usage: $O(E)$. The number of neighbours of each node is the same as the number of edges incident to that node.

Adjacency lists provide a suitable compromise between the adjacency matrix's speed and the edge list's memory usage.

3 Graph search

Depth-first search

The **depth-first search** algorithm is our basic tool for answering queries regarding connectivity of the graph. It takes as input some vertex v and processes all nodes connected to v.

DFS runs in $O(V + E)$ – every vertex is processed exactly once, and every edge is considered at most twice (once in each direction).

DFS has some glaring problems:

- The order in which vertices are processed is "depth-first", which is not a useful search order in most cases.
- When $V \gg 10^5$ we risk exceeding the recursive depth limit.

Breadth-first search

BFS solves these issues at the cost of a more complex algorithm. As the name suggests, BFS processes vertices in a *breadth-first order*. Imagine a "radius of influence" that begins at v and expands evenly in all directions. We process vertices as soon as they enter our radius of influence.

Algorithm 2 basic framework of BFS

```
1  void bfs(int v) {
2     bool enqueued[V] = {false, false, ...};
3     queue<int> q; // this is a STL type in <queue> library
4
5     q.push(v);
6     enqueued[v] = true;
7
8     while (!q.empty()) {
9        int cur = q.front();
10       q.pop();
11
12       // process cur here
13
14       for (each u adjacent to cur) {
15          if (!enqueued[u]) {
16             enqueued[u] = true;
17             q.push(u);
18          }
19       }
20    }
21 }
```

BFS also runs in $O(V + E)$.

The breadth-first ordering ensures that we process vertices in the order of their distance from v; so *the number of 'layers' processed to reach a vertex u is the length of the shortest path $v \to u$!* This is *not* the case with DFS. So in addition to answering queries about connectivity, BFS also allows us to solve problems about shortest paths between vertices.

If we replace the queue with a stack, we get a DFS. This DFS is not recursive and won't have recursive depth issues, but if you've coded up a DFS algorithm with a stack, you may as well replace the stack with a queue and use a BFS instead. There is rarely a good reason to use DFS over BFS.[3]

Backtracing (BFS)

Determining the length of a shortest path is not always sufficient. Often, we want a shortest path itself (i.e. the sequence of vertices) from u to v. To do this, we use an informal technique called backtracing (backtracing is not to be confused with backtracking, an unrelated technique in graph search). The backtrace associates to each vertex v its 'parent' vertex – the vertex whose edge we relaxed to find v.

Although it seems more intuitive to associate to each vertex the vertex that *follows* it in the BFS's search ordering, BFS may discover multiple 'child' vertices for a single 'parent' (e.g. shortest paths to w and v might both pass through t, but a shortest path to w and a shortest path to v will only pass through one 'parent': t). So we track the ancestral relationship rather than the successor relationship.

To reconstruct the shortest path, we begin at v and use the backtrace to "trace back" our steps until we find ourselves at u. The vertices we encounter along this trace are the vertices along a shortest path.

Backtracing is best understood through code. See algorithm 3.

4 Special graphs

Linked list

The linked list is a basic graph that you have encountered already.

Tree

A tree is a connected graph with $V - 1$ edges. Trees have several interesting properties; for instance, in a tree there is exactly one path between any pair of vertices, and trees never contain cycles. A tree is **rooted** if there is a unique vertex (the **root**) from which the tree is 'suspended'.

Family trees are an example of rooted trees. In the former, the root is the oldest ancestor in the family. In the latter, the root is the winner of the tournament.

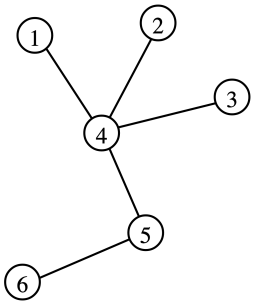
In a rooted tree, the **parent** of vertex v is the first vertex we encounter when we walk from v to the root. v's **children** are the vertices adjacent to v excluding v's parent. The **ancestors** of v are all vertices encountered on the walk from v to the root, and the **descendants** of v are all the vertices 'downstream' from v (away from the root). A **binary tree** is a rooted tree whose nodes have 0, 1 or 2 children.

Trees (and binary trees in particular) form the basis of many advanced data structures.

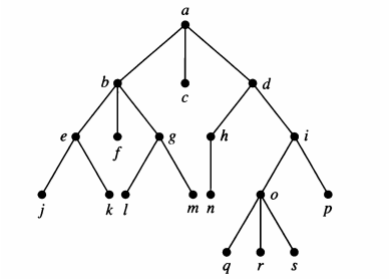
[3] There are a few situations when the recursive structure of a DFS is useful, e.g. Kosaraju's algorithm for strongly-connected components (April camp graph theory).

Algorithm 3 BFS with backtracing and path reconstruction

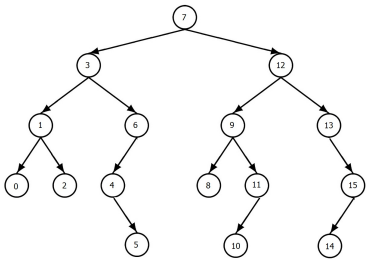
```
// returns list of vertices along a shortest path u -> v
vector<int> shortestPath(int u, int v) {
   bool enqueued[V] = {false, false, ...};
   int backtrace[V]; // backtrace[w] = parent of w in search structure
   queue<int> q;

   q.push(u);
   enqueued[u] = true;

   while (!q.empty()) {
      int cur = q.front();
      q.pop();

      for (each w adjacent to cur) {
         if (!enqueued[w]) {
            // w's parent is cur
            backtrace[w] = cur
            enqueued[w] = true;
            q.push(w);
         }
      }
   }

   if (!enqueued[v]) {
      // we never encountered v. no shortest path.
   } else {
      vector<int> path;
      // we reconstruct the path backwards, then reverse it
      int lastVertex = v;
      while (lastVertex != u) {
         path.push(lastVertex);
         lastVertex = backtrace[lastVertex];
      }
      path.push(u);
      path.reverse();
      return path;
   }
}
```



(a) an unrooted tree



(b) a rooted tree with root a



(c) a binary tree with root 7

Fig. 4.1: trees