FIT2004: Supplementary notes for lecture 8

Graphs: Representation and Traversal

DANIEL ANDERSON [1]

Graphs are a simple way of modelling sets of objects and encoding relationships between those objects. Despite their simplicity on the surface, graphs have an enormous number of applications in a wide range of fields. Graph problems are ubiquitous not only in computer science, but in modelling complex processes and situations in all disciplines of science and business. We will begin our study of graphs with the two most fundamental ideas that we will need to absorb, which are representation: how do we actually model and store a graph, and traversal: how do we examine the contents of a graph to determine its underlying properties.

Summary: Graph representation and traversal

In this lecture, we cover:

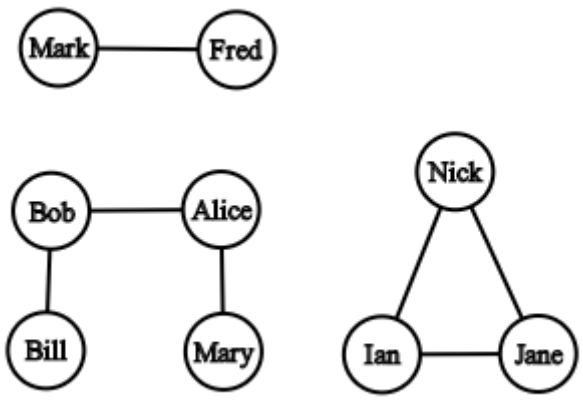
- Modelling real-world problems with graphs and formal descriptions of graphs
- Representation and storage techniques for graphs
- Graph traversal algorithms with applications

Recommended Resources: Graph representation and traversal

- CLRS, Introduction to Algorithms, Sections 22.1, 22.2, 22.3
- Weiss, Data Structures and Algorithm Analysis, Section 12.4.1
- `http://users.monash.edu/~lloyd/tildeAlgDS/Graph/`
- `https://visualgo.net/en/dfsbfs`

Modelling with Graphs

At the most basic level, graphs tell us about relationships between pairs of objects. Graphs that model objects with particular attributes are often referred to as *networks*, although the terms can usually be used interchangeably, so don't worry about the distinction. For example, consider a social network where friends connect with each other. We can represent this information in a graph, where each person is represented by a *node* or *vertex* of the graph, and a connection between friends is denoted by an *edge* or *arc* connecting them.



A network representing a set of people who have connections with each other.

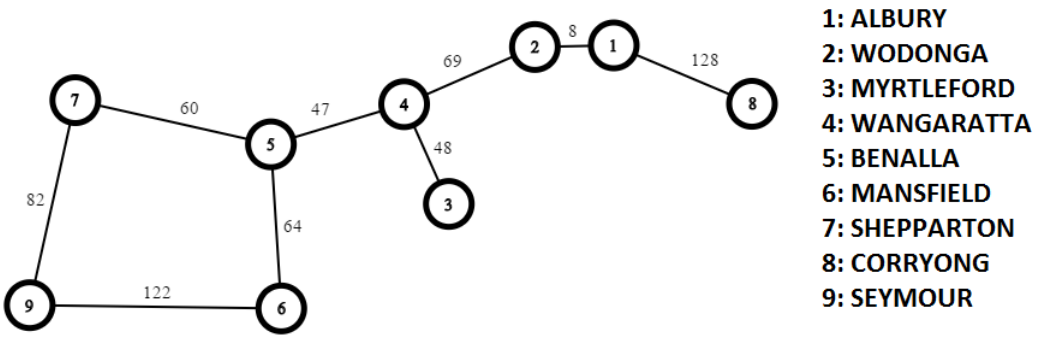
[1] Faculty of Information Technology, Monash University: `daniel.anderson@monash.edu`. These notes are based on the lecture slides developed by Arun Konagurthu for FIT2004.

Given this graph, several interesting questions might immediately come to mind:

- What are the groups of mutual friends? ie. friends that have a friend in common, or a friend-of-a-friend etc. (These are called the *connected components* of the graph)
- What is the largest degree of separation between two people in the network? (This would correspond to the longest *distance* between any people in the graph)
- Are there any people in the network whom if removed would cause a pair of mutual friends to no longer be mutual friends? (Such a vertex in a graph is called an *articulation point* or *cut point*)

Graph and network algorithms can help us solve each of these problems.

Graphs can also be *weighted*. As another example, consider a road map of towns in the country, where each town is represented by a vertex, and roads between towns are represented by edges with *weights* corresponding to the distance between them.

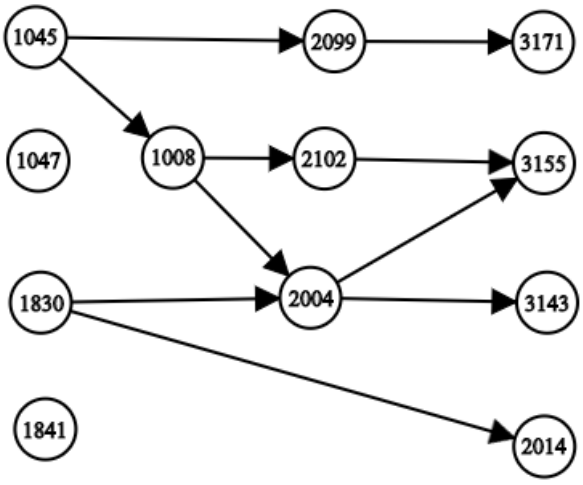


A network representing a road map of country towns.

Several interesting questions can be asked about such a graph:

- Given a pair of towns, what is the shortest route connecting them? (This is called the *shortest path problem*)
- What is the overall shortest subset of roads that we could keep, removing all others while still keeping every town connected? (This is called the *minimum spanning tree problem*)

Edges in a graph can also be directed, that is the relation only holds in one direction but not both. For example, consider your courses that you are taking for your degree, which have prerequisites that must be completed first. We can model this with a graph, in which a course has a directed edge to another course if the first course is a prerequisite of the second course and hence must be completed first.



A network representing a course progression for a Computer Science degree.

Some more interesting questions now arise:

- Are there any cycles in the course graph? If so, the degree is impossible to complete! Ooops.
- If there aren't any cycles, what is a valid order to complete the courses in that satisfies all of the prerequisites? (Such an order is called a *topological ordering* of the graph.)
- If there aren't any cycles, what is the longest chain of prerequisites? ie. assuming we could handle as many courses at a time as we wanted, how many semesters would it take us to complete the degree? (This is called the *critical path problem*)

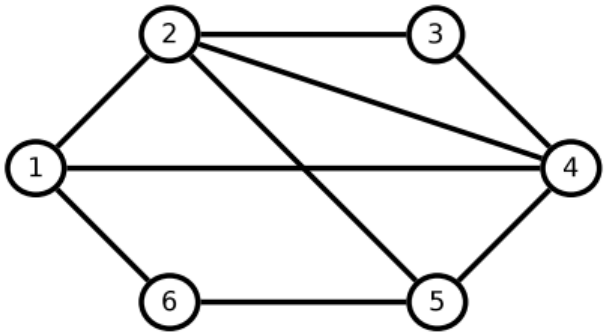
Once again, graph and network algorithms can help us solve all of these and many more problems!

Formal graph notation

Formally, a graph G is defined by a pair of sets V and E, where

- V is the the set of vertices / nodes.
- E is the set of edges / arcs, where an edge e connects two nodes u and v

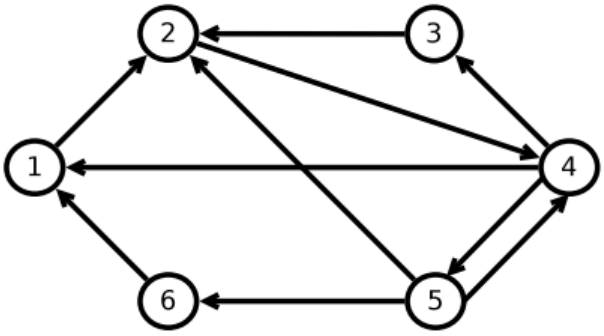
The number of vertices in a graph is usually either denoted by $|V|$, or as n or N. Sometimes when the distinction is obvious and the writer is lazy, people will simply write V to refer to the number of vertices. Similarly, we denote the number of edges by $|E|$, m or M, or occasionally when being lazy and not totally accurate, by E.



An unweighted, undirected graph

Direction

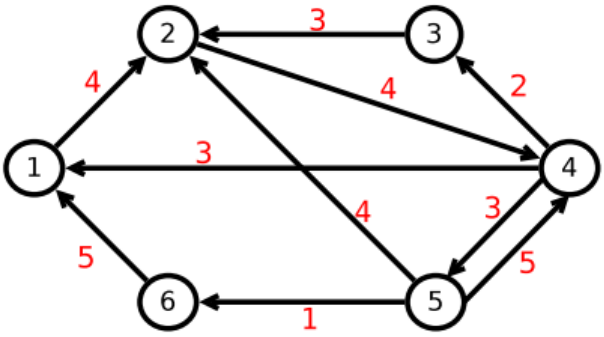
The edges of a graph may be *directed* or *undirected.* In an undirected graph, the edges (u, v) and (v, u) are equivalent, but in a directed graph, they represent different things (for example, a one-way street in a road map, or a course prerequisite, which are not the same if you reverse them!)



An unweighted, directed graph

Weights

The edges may also be *unweighted* or *weighted,* in which case they have an associated quantity, which may represent for example, the distance between two towns in a road map, the bandwidth of a connection in an internet network, the amount of money that it would cost to connect two houses via fibre-optic cable, or anything else you can imagine. We will usually denote the weight of the edge (u, v) by $w(u, v)$.



A weighted, directed graph

Multigraphs and loops

Depending on the particular problem, it may be allowed or disallowed for a graph to contain multiple edges between the same pair of vertices. A graph that does contain multiple edges between the same pair of vertices is usually referred to as a *multigraph.*

Similarly, for a particular purpose, a graph may or may not be allowed to contain edges that connect a vertex to itself. Such edges are usually referred to as *loops.* A graph that contains no loops or multiple edges between the same pair of vertices is called a *simple graph.*

Representation and Storage of Graphs

There are several options that we may choose from when it comes to actually storing a graph for use in a computer program. The choice that we make will depend on the particular variety of graph and on the problem that we plan on solving.

Considerations - Density of the graph

The biggest factor that will influence our decision of how to store our graph will be its *density,* which informally means does it have few edges or lots of edges. If we allow multiple edges between vertices, our graph can have an unbounded number of edges, so assume for now that our graphs do not have multiple edges.

- In a directed graph, the maximum number of possible edges that we can have is therefore $|V|^2$ if we allow loops, or $|V|(|V| - 1)$ otherwise.
- In an undirected graph, the maximum number of possible edges is $\binom{|V|+1}{2}$ if we allow loops, or $\binom{|V|}{2}$ otherwise.

We say that a graph is *dense* if $|E| \approx |V|^2$, that is informally, the graph has a lot of edges. Conversely, we call a graph *sparse* if $|E| \ll |V|^2$, ie. the graph has a relatively small number of edges.

Representation strategy – Adjacency Matrix

One way to represent a graph is using an *adjacency matrix.* The adjacency matrix of a graph $G = (V, E)$ is a matrix A of size $|V| \times |V|$. The space requirement of storing an adjacency matrix is therefore $O(|V|^2)$. When

the edges are unweighted, the entries of the adjacency matrix are defined by

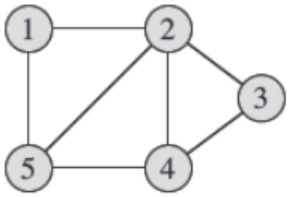
$$a_{i,j} = \begin{cases} 1, & \text{if there is an edge } (i,j) \\ 0, & \text{otherwise} \end{cases}.$$

In the case of multigraphs, adjacency matrices can be generalised such that $a_{i,j} = k$ where k is the number of edges between vertices i and j.

In a weighted graph, the adjacency matrix stores the weights of the edges, such that

$$a_{i,j} = \begin{cases} w(i,j), & \text{if there is an edge } (i,j) \\ 0 \text{ or } \infty, & \text{otherwise} \end{cases}.$$

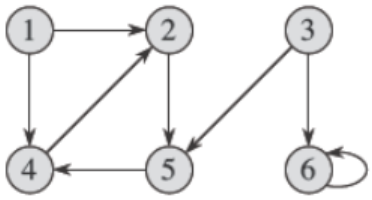
The choice of whether to use 0 or ∞ as an indicator that a particular edge is absent will depend completely on the problem at hand. The choice must be made such that the inclusion of the absent edge with the given weight does not change the solution to the problem being considered. Alternatively, if this is not feasible, one could store two matrices, one indicating adjacency without weights, and one storing the weights.



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

An example of an adjacency matrix for an unweighted, undirected graph. Source: CLRS

In an undirected graph, the adjacency matrix will clearly be symmetric, so if we wish to, we can save memory by only storing the upper triangular entries. Given an adjacency matrix, checking whether two vertices have an edge between them can be done in constant $O(1)$ time by simply checking the corresponding entry in the adjacency matrix.



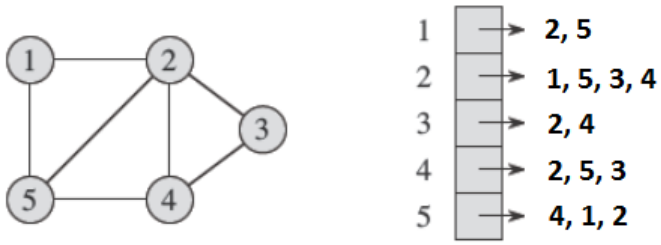
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

An example of an adjacency matrix for an unweighted, directed graph. Source: CLRS

Representation strategy – Adjacency List

Adjacency matrices are good for representing dense graphs since they use a fixed amount of memory that is proportional to the total number of possible edges. However, when a graph is sparse, the corresponding adjacency matrix will contain a very large number of 0 entries, using the same $O(|V|^2)$ space regardless of the

actual number of edges in the graph. In this case, a strong alternative is to store the graph in the form of an *adjacency list*. An adjacency list is simply a list of lists, where each list corresponding to a particular vertex stores the vertices that the given vertex is adjacent to. In a weighted graph, the adjacency list will store the weights of the edges alongside the vertices corresponding to those edges.



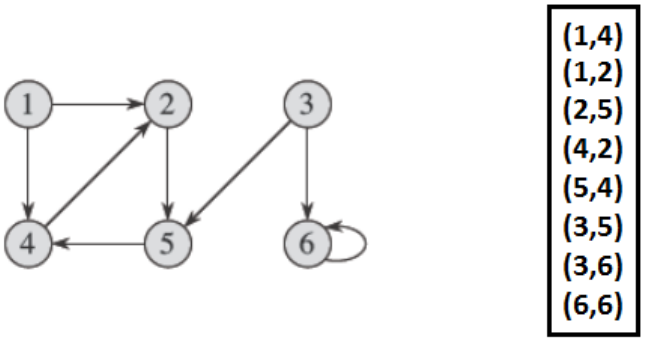
An example of an adjacency list for an unweighted, undirected graph. Source: CLRS

Since an adjacency list only uses memory for edges that actually exist in the graph, the space required for this representation is $O(|E|)$, which is a huge improvement over adjacency matrices for sparse graphs, and about the same for dense graphs.

Unlike with adjacency matrices, we can no longer check whether two given vertices are adjacent in constant time using an adjacency list. Adjacency lists are however, particularly convenient when we only need to iterate over the existing edges since we will never need to examine a non-existent edge. To make a slight trade-off of space, we could instead store our adjacency list for each vertex in a balanced binary search tree, which would allow for $O(\log(|E|))$ adjacency checks while retaining the ability to iterate efficiently. Such a strategy would use more memory however and is not particularly useful for most applications.

Representation strategy – Edge List

Although used only rarely, occasionally we will opt for the *edge list* strategy. In this case, we simply store a list that contains all of the edges of the graph in no particular order.



An example of an edge list for an unweighted, directed graph. Source: CLRS

The edge list, like the adjacency list only uses $O(|E|)$ space, but affords us neither the ability to check adjacency quickly or iterate over the adjacent vertices of a particular vertex efficiently. Their primary use is in Kruskal's minimum spanning tree algorithm where we need to sort all of the edges in descending order of weight, which can not be done effectively with the other two representation strategies.

Graph Traversal and Applications

A huge number of graph algorithms are built on a few small key ideas. The simplest and most wide reaching archetype of graph algorithm are the depth-first and breadth-first traversals. *Traversing* a graph simply means to explore it and figure out some of its properties.

Depth-first search

A depth-first traversal as the name implies, searches a graph by following a path as deep as possible before reaching a dead end and backtracking up the path, continuing the search from edges that have yet to be explored. In a depth-first search traversal, we maintain a flag on each node that indicates whether or not we have visited that node yet. When a node is visited, we mark its flag as such in order to avoid visiting the same node multiple times and causing infinite repetition. Due to its nature, a depth-first search can be implemented very succinctly using recursion. We will assume that the graph is represented as an adjacency list `E[1..N]`, where `E[u]` contains the list of vertices adjacent to `u`.

Algorithm: Generic Depth-First Search

```
1:  function TRAVERSE(G = (V[1..N], E[1..N]))
2:      Set visited[1..N] = False
3:      for u = 1 to N do
4:          if not visited[u] then
5:              dfs(u)
6:          end if
7:      end for
8:  end function
9:
10: function DFS(u)
11:     visited[u] = True
12:     for each vertex v ∈ E[u] do
13:         if not visited[v] then
14:             dfs(v)
15:         end if
16:     end for
17: end function
```

To save on repetitively passing large numbers of arguments, we will assume for any of the following algorithms that the contents of the graph `G`, the `visited` array and any other variables that are being computed are visible to the function `dfs`.

The skeleton above comprises a simple depth-first traversal that is the basis of a huge number of graph algorithms with wide-ranging applications. Since the depth-first search algorithm visits every vertex only once and examines every edge at most once, its time complexity is $O(|V| + |E|)$.

Some applications of depth-first search

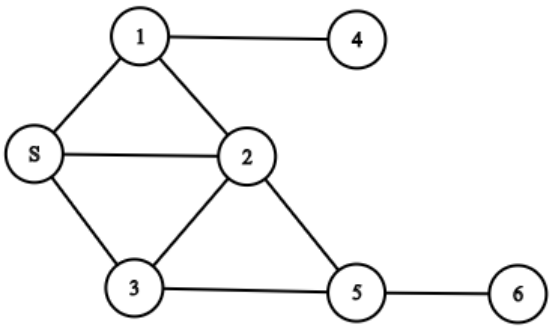
- Finding the connected components of a graph – the connected components of a graph are the maximal connected subgraphs, that is, they are the subsets of vertices that are mutually connected, directly or indirectly.
- Two-colouring a graph, or deciding that the graph can not be two-coloured – a graph can be two-coloured if every vertex can be assigned a colour, white or black, such that no two neighbouring vertices share the same colour. This is equivalent to the graph being bipartite.
- Finding a cycle in a directed graph – a cycle is a connected sequence of vertices that begins and ends at the same vertex.
- Finding a topological ordering of a directed, acyclic graph (DAG) – a topological ordering is an ordering of the vertices of DAG such that all vertices `v` that are reachable from the vertex `u` come after it in the topological ordering, in other words, they are orderings that satisfy prerequisite relationships.
- Finding bridges and cut-points of a graph – these are edges and vertices that if removed from the graph would disconnect some connected component, in a sense they are "non-redundant" connections.

Breadth-first search

Breadth-first search, like depth-first search traverses a graph one vertex at a time, never visiting a vertex more than once, until all vertices have been visited. The only difference between the two is the order in which the

vertices are visited. Indeed, many problems that can be solved with a depth-first search can also be solved with a breadth-first search as an alternative, but not always.

While a depth-first search explores paths as deep as possible immediately, a breadth-first search visits nearby vertices first and slowly spans out to further away vertices. Formally, breadth-first search always visits every vertex that is a distance k from the starting point before visiting any vertices that are a distance $k+1$. For this reason, breadth-first search can unfortunately not be written recursively and hence is slightly more difficult to implement. In order to visit vertices that are nearby sooner, breadth-first search maintains a queue of vertices that need to be visited, appending progressively further vertices to the end of the queue as it traverses.



A graph G whose vertices are labelled in BFS order (with ties broken from the top of the image downward.) Note that every vertex that is at a distance 1 from `s` is visited before those of distance 2 etc.

Note that due to typical applications, it is most common to use breadth-first search to explore a graph G that is known to be connected, and hence our pseudocode does not perform multiple searches like we did in the depth-first search and instead begins from a single given source node s. This is completely a matter of convention, of course you can perform breadth-first search from multiple components if you so desire. You may also breadth-first search from multiple starting locations simultaneously if required.

Algorithm: Generic Breadth-First Search

```
1: function BFS(G = (V[1..N], E[1..N]), s)
2:     Set visited[1..N] = False
3:     Set inqueue[1..N] = False
4:     Set queue = Queue()
5:     queue.push(s)
6:     while queue is not empty do
7:         u = queue.pop()
8:         visited[u] = True
9:         inqueue[u] = False
10:        for each vertex v ∈ E[u] do
11:            if not visited[v] and not inqueue[v] then
12:                queue.push(v)
13:                inqueue[v] = True
14:            end if
15:        end for
16:    end while
17: end function
```

Finally, just like the depth-first search, breadth-first search also visits each vertex at most once and examines each edge at most once, hence its time complexity is $O(|V| + |E|)$ (provided that we are using a queue data structure that supports $O(1)$ push and pop.)

Application of BFS – Single-source shortest paths in an unweighted graph

Due to the fact that it visits vertices in order of nearby first, far away last, breadth-first search can be used for finding the shortest paths in an unweighted graph from a given starting vertex `s`. We will track information about shortest paths by maintaining two arrays, `dist[1..N]` and `pred[1..N]`, where `dist[u]` denotes the

distance of the vertex `u` from our starting vertex `s`, and `pred[u]` denotes the vertex that proceeded `u` on the shortest path from `s`. By maintaining just these two arrays, we will be able to reconstruct the shortest paths from `s` to every other vertex in the graph.

Algorithm: Single-source Shortest Paths in an Unweighted Graph

```
1: function BFS(G = (V[1..N], E[1..N]), s)
2:     Set visited[1..N] = False
3:     Set inqueue[1..N] = False
4:     Set dist[1..N] = ∞
5:     Set pred[1..N] = 0
6:     Set queue = Queue()
7:     queue.push(s)
8:     dist[s] = 0
9:     while queue is not empty do
10:        u = queue.pop()
11:        visited[u] = True
12:        inqueue[u] = False
13:        for each vertex v ∈ E[u] do
14:            if not visited[v] and not inqueue[v] then
15:                dist[v] = dist[u] + 1
16:                pred[v] = u
17:                queue.push(v)
18:                inqueue[v] = True
19:            end if
20:        end for
21:    end while
22: end function
```

Note that this algorithm does not only compute the shortest path between one pair of vertices, it computes the shortest paths from the source vertex `s` to **every other** reachable vertex in the graph. In general, the former problem is usually asymptotically the same difficulty as the later, so we almost always focus on solving the more general problem, as it subsumes the former without adding extra difficulty. There are however, some techniques specifically for finding individual shortest paths in extremely large search spaces such as the meet-in-the-middle approach, which can reduce an exponential search time to a smaller but still exponential search time.

Once we have the shortest path information, we can reconstruct the sequence of vertices that make up the shortest path from `s` to the vertex `u` by backtracking through the `pred` array until we reach `s`, at which point we will have built the entire path (in reverse, so we can just reverse it at the end.)

Algorithm: Reconstruct Shortest Path

```
1: function GET_PATH(u, pred[1..N])
2:     Set v = u
3:     Set path = [v]
4:     while pred[v] ≠ 0 do
5:         path.append(pred[v])
6:         v = pred[v]
7:     end while
8:     return reverse(path)
9: end function
```

A shortest path can never contain repeated vertices, hence one has length at most $|V|$, making the time complexity of the `get_path` function $O(|V|)$.