Lecture 18: Graphs

COSC242: Algorithms and Data Structures

Brendan McCane

Department of Computer Science, University of Otago

Definition

A graph, G, consists of a set of vertices, V, and a set of edges, E. Vertices are specified by unique labels, and an edge is an ordered pair of vertex labels.

For example:

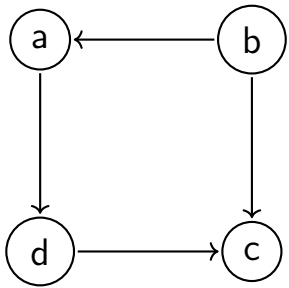
$$G = (V, E)$$
$$V = \{a, b, c, d\} \quad E = \{(b, a), (b, c), (a, d), (d, c)\}$$

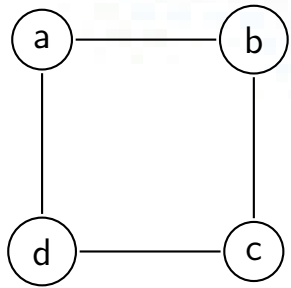
A graph may be directed or undirected. The definition above suffices for both, or we can explicitly define all edges in an undirected graph:

$$E = \{(b, a), (a, b), (b, c), (c, b), (a, d), (d, a), (d, c), (c, d)\}$$

Visual representation

Graphs are often represented visually. For example, the above graph, with directed edges on the left, and undirected on the right, looks like:





Applications

Here are a few of the applications of graphs:

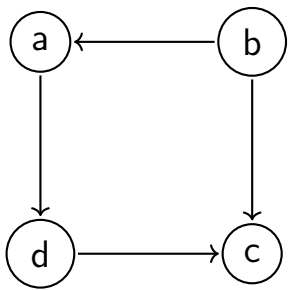
- the web, internet and all computer networks are graphs
- network routing
- web search (e.g. via pagerank)
- a map can be represented as a graph
- shortest route algorithms are graph algorithms
- automatically timetabling classes to minimise clashes
- social networks are graphs
- your brain is a graph.
- molecules can be modeled as graphs and used to simulate physical processes
- biological systems can be modeled as graphs (predator, prey, etc)
- projects can be modeled as graphs. Graph algorithms are used to determine critical paths or critical actions in a project

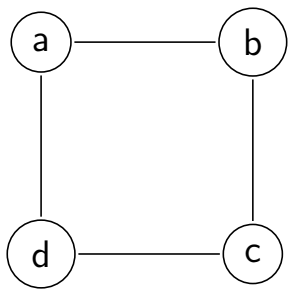
Implementation: adjacency matrix

Let $G = (V, E)$ with $V = \{a_1, a_2, \ldots, a_n\}$.

An *adjacency matrix* for G is an $n \times n$ Boolean array A, such that $A[i, j] = 1$ if there is a (directed) edge from j to k, else $A[j, k] = 0$.

For our example graphs we get (to be done in class):

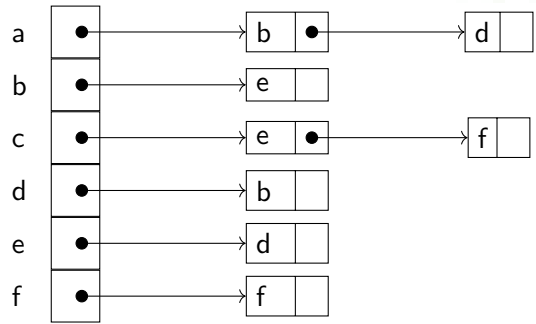




Implementation: adjacency list

Let $G = (V, E)$ with $V = \{a_1, a_2, \ldots, a_n\}$.

An adjacency list is an array, A, of length $|V| = n$ such that every $A[j]$ is a linked list containing all the vertices a_k such that there is an edge from a_j to a_k.



Comparing Implementations

Suppose G is a graph with n vertices in V. Which is better, an adjacency matrix or adjacency list?

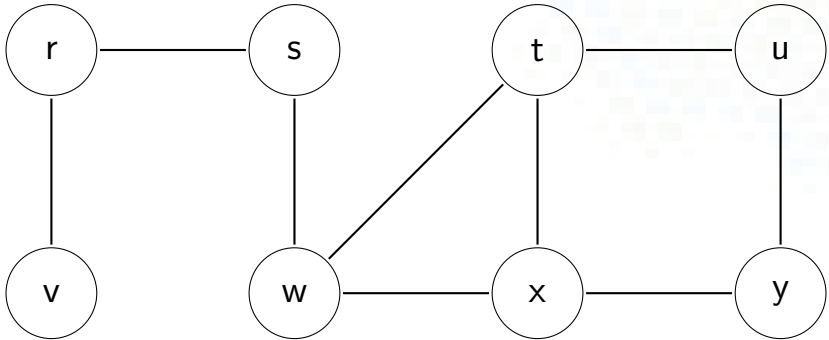
The space requirements of the adjacency matrix are high — if the graph is dense this space is utilised, if the graph is sparse an adjacency list would be more economical. (Dense means the number of edges in E is close to n^2, sparse means n or fewer edges.)

It also depends on the operations we want to use. Apart from insertion and deletion, the three most common operations on graphs are:

1. Given two vertices j and k, determine whether there is an edge connecting them.
2. Given vertex j, find all vertices adjacent to j.
3. Given a vertex j as starting point, traverse the graph.

The first operation is supported best by the adjacency matrix, the second and third by adjacency lists. Why?

Breadth-first search example



BFS overview

Given a graph and a source vertex s, BFS finds the shortest paths from s to all the other vertices.

Every vertex starts off coloured white, then is coloured grey when it is discovered for the first time, and finally is coloured black when its adjacency list has been fully examined (i.e. when all the vertices adjacent to it have been coloured grey).

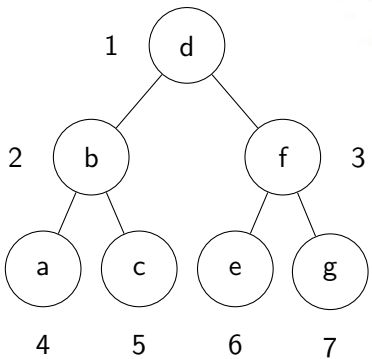
For every vertex u, we compute the distance from the source vertex s to u and store it in a variable $d[u]$. Initially the value of the variable $d[u]$ is set to a default value, say, ∞. We could simplify the algorithm to do without colours (instead of testing whether colour[u] = white we can test whether $d[u] = \infty$).

BFS algorithm

```
1:  procedure BFS(G, s)
2:      for each vertex u different from source s do
3:          colour[u] ← white; d[u] ← ∞
4:      end for
5:      colour[s] ← gray; d[u] ← 0; Q ← ∅; Enqueue(Q, s)
6:      while Q ≠ ∅ do
7:          u ← Dequeue(Q)
8:          for v ∈ Adj[u] do
9:              if colour[v] = white then
10:                 colour[v] ← gray; d[v] ← d[u] + 1; Enqueue(Q, v)
11:             end if
12:         end for
13:         colour[u] ← black
14:     end while
15: end procedure
```


BFS and binary trees

When applied to a binary tree, BFS gives a new kind of traversal — a level-order traversal.



Depth-first Search

The idea of depth-first search (DFS) is that one goes deeper whenever possible, i.e. we explore the edges from the most recently discovered vertex.

DFS leads to backtracking — when all the edges leaving a vertex v have been explored, we backtrack to the vertex u which was v's parent (i.e. from which v was discovered).

As in BFS, vertices start off white, are made grey when first discovered, and are made black when finished, i.e. their adjacency list has been examined completely.

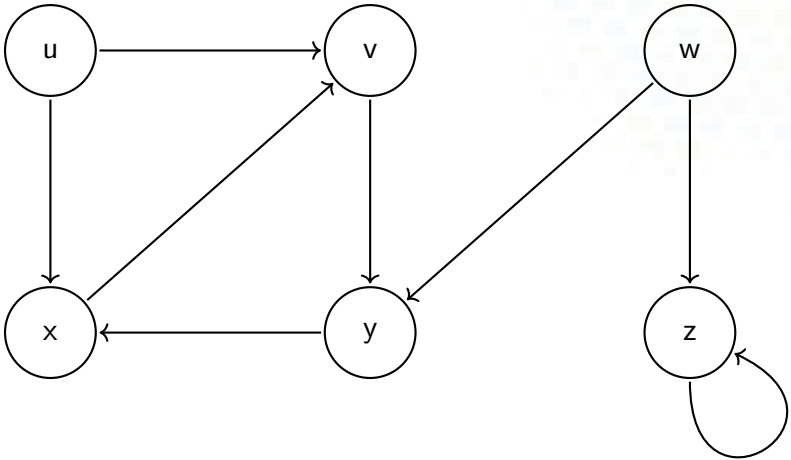
DFS Algorithm

$time$ is a global variable in the following. Let $G = (V, E)$ be the graph of interest.

```
1:  procedure DFS(G)
2:      for each u ∈ V do
3:          colour[u] ← white
4:      end for
5:      time ← 0
6:      for each u ∈ V do
7:          if colour[u]=white then
8:              DFS_visit(u)
9:          end if
10:     end for
11: end procedure
```

```
1:  procedure DFS_VISIT(u)
2:      colour[u] ← grey
3:      time ← time + 1
4:      d[u] ← time
5:      for each vertex v ∈ adj[u] do
6:          if colour[v] = white then
7:              DFS_visit(v)
8:          end if
9:      end for
10:     colour[u] ← black
11:     time ← time + 1
12:     f[u] ← time
13: end procedure
```


DFS example



Relevant parts of the textbook

Graphs and graph algorithms are discussed in Chapters 22-26.

The introductory material is discussed in Chapter 22.