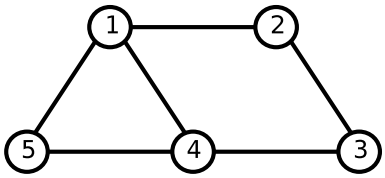
Basic Graph Algorithms

1 Representations of Graphs

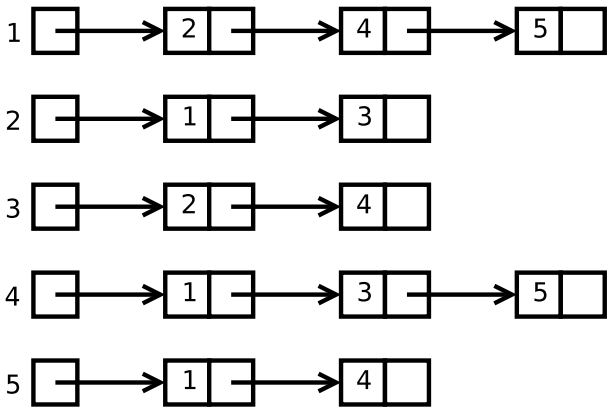
There are two standard ways to represent a graph $G(V, E)$ where V is the set of vertices and E is the set of edges.

- adjacency list representation
- adjacency matrix

Say we have a graph with five vertices and six edges as follows.



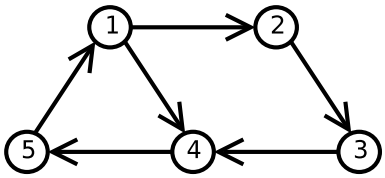
In an adjacency list representation, we will have a linked-list for each vertex. From list 2, for example, we can tell that vertex 2 has edges to both vertices 1 and 3.



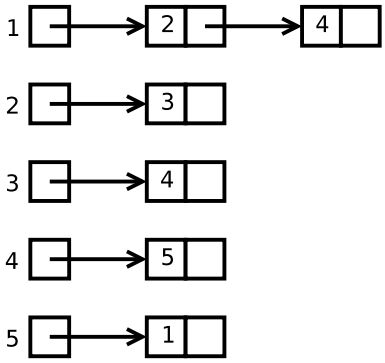
The adjacency matrix representation for the same graph is as follows. From row 2 of the matrix, we find that there are 1s in columns 1 and 3. This indicates that there are edges from vertex 2 to vertices 1 and 3.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

These representation techniques can also be used for **directed graphs**. Consider the following directed graph.



The adjacency list and adjacency matrix representations follow.



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

2 Basic Search Techniques for Graphs

2.1 Classification of edges

A search done on a graph G results in some solution forest (set of trees) consisting of a subset of edges of G. The edges of G can be classified as follows:

- Tree edges – edges in the solution forest.
- Back edges – edges of the original graph that connects a vertex to its ancestor in a solution tree.
- Forward edges – edges not in the solution forest that connect a vertex to a decendent in a solution tree.
- Cross edges – all other edges. They can go between vertices in the same solution tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different solution trees.

2.2 Breadth First Search

Breadth-first search (BFS) is a search technique that expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.

To keep track of progress, BFS colors each vertex WHITE, GRAY, or BLACK. All vertices start out WHITE and may later become GRAY and then BLACK.

```
BFS(G, s)
1   for each vertex u ∈ V[G] − {s}
2       do color[u] ← WHITE
3          d[u] ← ∞
4          p[u] ← NIL
5   color[s] ← GRAY
6   d[s] ← 0
7   p[s] ← NIL
8   Q ← {s}  /* Q always contains the set of GRAY vertices */
9   while Q ≠ ∅
10         do u ← head[Q]
11            for each v ∈ adj[u]
12                do if color[v] = WHITE
13                      then color[v] ← GRAY
14                           d[v] ← d[u] + 1
15                           p[v] ← u
16                           ENQUEUE(Q, v)
17            DEQUEUE(Q, u)
18            color[u] ← BLACK
```

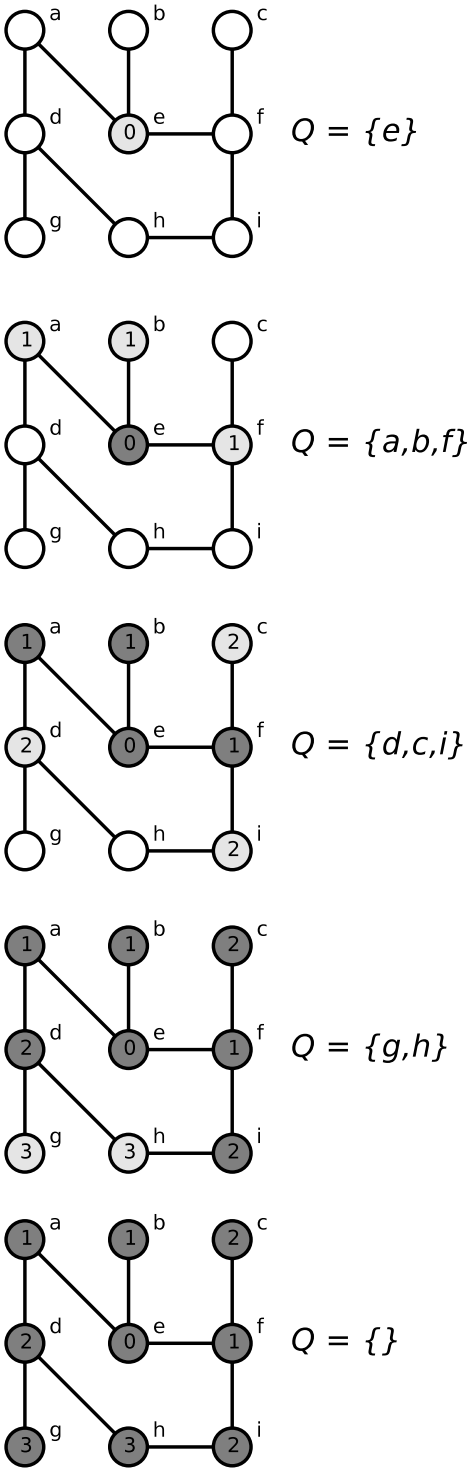


Figure 1: BFS example on an undirected graph with 9 vertices called with *source* = *e*.

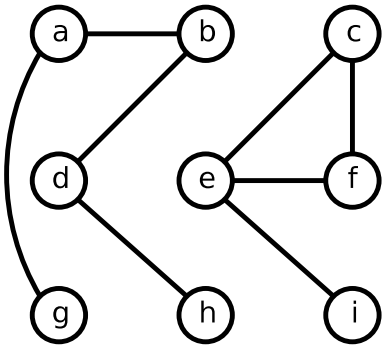
2.3 Depth First Search

The strategy followed by DFS is to search *deeper* in the graph whenever possible. In DFS, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v's edges have been explored, the search backtracks to explore edges leaving the vertex from which v was discovered. $d(u)$ is the start time of visiting u and $f(u)$ is the finish time of having explored all the neighboring vertices of u.

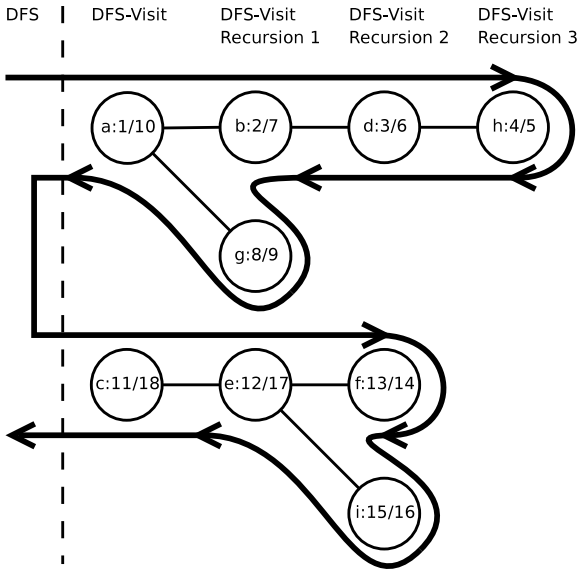
```
DFS(G)
1   for each vertex u ∈ V[G] − {s}
2       do color[u] ← WHITE
3          p[u] ← NIL
4   count ← 0
5   for each u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-Visit(u)
```

```
DFS-Visit(u)
1   color[u] ← GRAY
2   d[u] ← count
3   count ← count + 1
4   for each v ∈ adj[u]
5       do if color[v] = WHITE
6             then p[v] ← u
7                  DFS-Visit(v)
8   color[u] ← BLACK
9   f[u] ← count
10   count ← count + 1
```

Consider the following undirected graph. In DFS, all vertices are initialised to WHITE, then each one is visited if they are white. Assuming that $V[G] = \{a, b, c, d, e, f, g, h, i\}$, DFS-Visit is first invoked on a. Within DFS-Visit, a is coloured GRAY, then the vertices adjacent to a are visited recursively if they are WHITE. When all visits are completed, a is coloured BLACK (note that DFS still works if we leave it as GRAY). This results in the visiting order $\{a, b, d, h, g, c, e, f, i\}$.



The following diagram shows the DFS trees created. The top row shows at which level of recursion (if any) a vertex was visited by DFS-Visit. The bold line traces the visiting order of the DFS and DFS-Visit algorithms. The start and finish times are illustrated using the "$d[u]/f[u]$" notation; where "time" is basically a counter that gets incremented as DFS discovers or leaves a vertex.



2.3.1 Properties of DFS

1. In any DFS of a (directed or undirected) graph $G(V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

   - the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint,
   - the interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is a descendant of v in the DFS tree, or

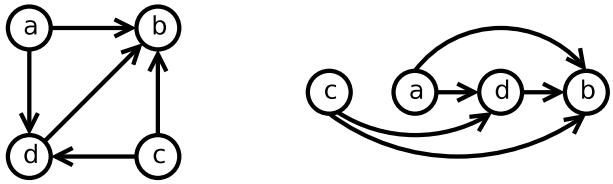


Figure 2: Topological Sorting. $c \leq a \leq d \leq b$.

- the interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$, and v is a descendant of u in the DFS tree.

For instance, in the previous example, $d[c] = 11$, and $f[c] = 18$. This contains all of $\{e, f, i\}$ because $d[c] < d[\{e, f, i\}]$ and $f[\{e, f, i\}] < f[c]$.

2. In a DFS forest of a directed or undirected graph $G(V, E)$, vertex v is a descendant of vertex u if and only if at the time $d[u]$ that the search discovers u, vertex v can be reached from u along a path consisting entirely of white vertices.

3. In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

3 Topological Sorting

A topological sort of a Directed Acyclic Graph (DAG) $G = (V, E)$ is a linear ordering of vertices in the DAG such that if G contains a directed edge $\langle u, v \rangle$, then u appears before v in the ordering (see Figure 2). A directed graph is acyclic if and only if a depth-first search on the graph yields no back edges.

Topological-Sort(G) can be determined by sorting the finish times $f[v]$ of each vertex $v \in V[G]$ in decreasing order. One way to implement this is to modify DFS-Visit to place each finished vertex into the front of a linked-list. The steps of this approach are as follows:

1. Initialise an empty linked list.

2. Execute DFS(G). As each vertex is coloured BLACK, prepend it to the front of the linked list.

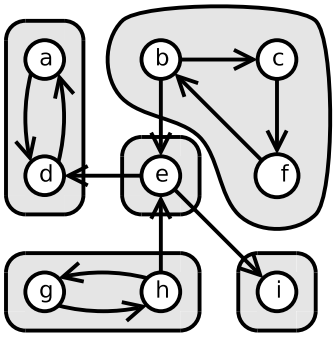
3. Return the linked list. The rank of each node is its position in the linked list started from the head of the list.

The running time is $O(m+n)$, where m is the number of edges and n is the number of vertices in G.

4 Strongly Connected Components

The strongly connected components (SCC) of a directed graph $G(V, E)$ is a maximal set of vertices $U \subseteq V$ such that for every pair of vertices u and v in U, u and v are mutually reachable.

Take the following graph of nine vertices for example. The strongly connected components in this graph are shaded in grey bubbles. Every vertex in each bubble is reachable by every other vertex in the same bubble.



STRONGLY-CONNECTED-COMPONENTS(G)
1 *finish-order* ← finish order of DFS(G)
2 G^T ← transpose of G
3 *dfs-forest* ← DFS trees from DFS(G^T)
. visited in reverse *finish-order*.
4 **return** *dfs-forest*
. /* *each tree in dfs-forest is a strongly connected component.* */

The running time of the algorithm is $O(m+n)$. Steps 1 and 3 take $O(m+n)$ time. Step 2 requires $O(m+n)$ time, and Step 4 takes $O(n)$ time. Thus, the algorithm takes $O(m+n)$ time.