Graph Operations and Analysis

Graph Operations and Analysis of Implementations

As with any data structure, the graph needs to support certain operations. We can then choose a preferred implementation based on the operations required.

The main operations are:

- Supporting Operations
  - Count number of vertices
  - Return/Enumerate neighbours of vertex v
  - Count edges
  - Check for edge between vertices u and v
- Traversing a graph
  - Path Reconstruction
  - Find out if vertex u can be reached from vertex v
  - Find shortest path between vertex u and vertex v
  - Identify and count number of components in a graph
  - Topological Sort
- Find a minimum spanning tree
  - Where the sum of edge weights are the minimum and all vertices are reachable from one another
- SSSP
  - Covered in next section

Analysis of Supporting Operations

1. Count number of vertices

The efficiency depends on the implementation:

- Adjacency Matrix: Simply the number of rows for the array. O(1) if the size is stored as a separate variable. O(1) if using ArrayList.size()
- Adjacency List: Simply the number of rows for the array. O(1) if size is stored as a separate variable. O(1) if using ArrayList.size()
- Edge List: O(E) to iterate through all E edges. However, it may be an impossible operation if there are vertices isolated i.e. no edges connecting it

2. Enumerating neighbours of vertex v

The efficiency depends on the implementation:

- Adjacency Matrix: O(V) to iterate through all V items in row v. Scan AdjMatrix[v][j], $\forall j \in [0..V-1]$
- Adjacency List: O(k) as we scan AdjList[v], where k is the number of neighbours of vertex **v** (output-sensitive algorithm)
- Edge List: O(E log E + k) = O(E log V + k) as we need to first sort the edges (E log V), and then binary search (log E) and scan through the k neighbours (k)

3. Count edges

The efficiency depends on the implementation:

- Adjacency Matrix: $O(V^2)$ to count all non-zero entries
- Adjacency List: O(V) as we sum the length of all V lists

- Edge List: O(1) as we just return size of array. May need to divide by two if undirected edges are stored twice

4. Check for edge between vertices u and v

The efficiency depends on the implementation:

- Adjacency Matrix: O(1) as we check if AdjMatrix[u][v] = 0
- Adjacency List: O(k) as we scan through k neighbours of u to see if v is present
- Edge List: O(E) as we search through all E edges. If the edges are sorted beforehand, possibly O(log E).

Graph Traversal

Unlike a tree which has a root, a graph has no root. Hence, we simply start from a vertex which we call source. We also do not get to choose between a left and right child, and simply travel via edges to adjacent vertices.

One thing to be aware of is that there may be trivial and non-trivial cycles, which may result in us blindly going in a loop if no proper algorithm is employed. This is why we have the Breadth First Search and the Depth First Search algorithms.

Breadth First Search

Starting from a vertex s, we visit vertices of the graph G in a breadth-first manner. This means that we will visit all vertices that are one edge away before visiting vertices which are two edges away and so on.

- To maintain the order of visitation, we will need to use a queue, which initially contains only s
- To know whether we have already visited a specific vertex and avoid cycling, we can use an array of size V, with visited[v] = 0 when unvisited and visited[v] = 1 when visited
- To keep track of the path taken, we can have an array p of size V, where p[v] holds the predecessor or parent of v

The resultant predecessor array p will hold a BFS spanning tree of G. If the graph is not a connected graph, then p will contain the spanning tree of the component.

Using a Queue and two Arrays

This is the pseudo code for BFS:

```
// Initialisation
for all v in V
      visited[v] ← 0
      p[v] ← -1
Q ← {s} // start from s
visited[s] ← 1

// Execution
while Q is not empty
      u ← Q.dequeue()
      for all v adjacent to u // order of neighbor
            if visited[v] = 0 // influences BFS
                  visited[v] ← true // visitation sequence
                  p[v]←u
                  Q.enqueue(v)
```

We can then use the information in visited and p to solve any problems.

BFS Analysis

The initialization will always take O(V) time as it iterates through V vertices to initialize the values. However, if an Edge List is used, then this initialization may be impossible, as we may not be able to identify all vertices using an edge list. Hence, an Adjacency Matrix or Adjacency List is required.

If an Adjacency Matrix is used, the execution will take $O(V^2)$ time as the outer loop performs V times (since each vertex can only be queued once and all will be dequeued eventually), while the inner loop that checks for vertices adjacent to the current vertex will also run V times, due to the implementation of an Adjacency Matrix.

If an Adjacency List is used, the execution will take O(V+E) time as all V vertices are queued once, and for each vertex, its k edges are examined, where the sum of all k is E. Overall, the time complexity is thus O(V+E).

Depth First Search

Starting from vertex s, we visit the vertices of graph G in a depth-first manner. This means that we will keep going down a certain path until we encounter no new vertices to explore, before backtracking and exploring all possible paths while doing so.

- To maintain the order of visitation, we can use a stack or recursion, which is an implicit stack
- The other elements such as the arrays are the same as for BFS

The resultant predecessor array p will hold a DFS spanning tree of G. If the graph is not a connected graph, then p will contain the spanning tree of the component.

Using Recursion and two Arrays:

This is the pseudo code for DFS:

```
// initialisation
for all v in V
      visited[v] ← 0
      p[v] ← -1
DFSrec(s) // start the recursive call from s

DFSrec(u)
      visited[u] ← 1 // to avoid cycle
      for all v adjacent to u // order of neighbor
            if visited[v] = 0 // influences DFS
                  p[v] ← u // visitation sequence
                  DFSrec(v) // recursive (implicit stack)
```

We can then use the information in visited and p to solve any problems.

DFS Analysis

The initialization will always take O(V) time as it iterates through V vertices to initialize the values. However, if an Edge List is used, then this initialization may be impossible, as we may not be able to identify all vertices using an edge list. Hence, an Adjacency Matrix or Adjacency List is required.

If an Adjacency Matrix is used, the execution will take $O(V^2)$ time as the function is performed V times (since each vertex can only be visited once), while the loop that checks for vertices adjacent to the current vertex will also run V times, due to the implementation of an Adjacency Matrix.

If an Adjacency List is used, the execution will take O(V+E) time as all V vertices are visited once, and for each vertex, its k edges are examined, where the sum of all k is E. Overall, the time complexity is thus O(V+E).

Application of Graph Traversal

After we have our visited and p, how do we go about using it?

1. Path Reconstruction

Using the p array returned, we can output the spanning tree created by BFS or DFS. There are two ways to do so:

Iterative Path Reconstruction

We simply start from the final vertex, then output itself while looping back to its predecessor. We stop once we hit the source vertex.

```
// iterative version (will produce reversed output)
Output "(Reversed) Path:"
i ← t // start from end of path: suppose vertex t
while i != s
      Output i
      i ← p[i] // go back to predecessor of i
Output s
```

The output order is unfortunately in reverse order, due to the nature of while loops.

Recursive Path Reconstruction

Over here, we can achieve what we could not using iteration thanks to Tail Recursion.

```
// in main method
// recursive version (normal path)
Output "Path:"
backtrack(t); // start from end of path (vertex t)

void backtrack(u)
      if (u == -1) // recall: predecessor of s is -1
            stop
      backtrack(p[u]) // go back to predecessor of u
      Output u // recursion like this reverses the order
```

2. Reachability Test

We can easily test whether vertex v is reachable from vertex u. First, we run DFS or BFS on vertex u, then we check if visited[v] = 1. If it's equal to 1, then it's reachable, else it's unreachable.

3. Find shortest path between two vertices

For finding the shortest path from one vertex to another, **BFS must be used**. This is because DFS may take a longer path to reach a vertex than necessary.

However, there is one main condition to take note:

- The graph must be unweighted OR all edges must have the same weight
    - BFS can only return the least number of edges required to be travelled from vertex u to v
    - Weight is not considered

Using the O(V+E) BFS algorithm, we can run BFS from source u, and construct the shortest path from u to v using the p array after BFS finishes. The cost of the shortest path would be the number of edges in the path x weight of an edge.

4. Identifying and Counting Components

A component is a subgraph with at least one vertex, or if there are more than one vertex, where the vertices are connected to each other by at least one path.

We can use both BFS and DFS to do this. We simply iterate through the list of vertices with a global visited array. If the current vertex is not visited, we will increase the component count by one and perform DFS or BFS on it.

```
componentCount ← 0
for all v in V
      visited[v] ← 0
for all v in V
      if visited[v] == 0
            componentCount ← componentCount + 1
            DFSrec(v)// BFS from v is also OK
```

Overall complexity is still O(V+E), as we will still only visit every vertex once throughout this whole process.

5. Bipartite Graph Identification

Note that for any given node in a bipartite graph, if it were to be put into one set, then every node it is linked to by an edge must be put into the other set. Thus we can perform a modified BFS/DFS on the given graph such that as we perform the search, we set the current node being visited to the opposing parity as the node visited before it. We then check whether any of the nodes it is linked to has the same parity. If they do, then the graph cannot be bipartite, else continue the search.

```
Bipartite(G = (V,E))
      for all v in V do
            visited[v] ← 0
      end for
      flag ← true
      for all v in V do
            if visited[v] == 0 then
                  call dfs(v,1)
            end if
      end for
      return flag
dfs(u,c)
      if flag == false then
            return //Just terminate recursion
      end if
      visited[u] ← c
      for each node w such that (u, w) in E do
            if visited[w] == 0 then
                  dfs(w,-1 * c) // alternate c from 1 to -1 to 1...
            else if visited[w] == c then
                  flag ← false
                  return // just terminate recursion
            end if
      end for
```

6. Topological Sort

A topological sort of a Directed Acyclic Graph (DAG) is a linear ordering of its vertices in which each vertex comes before all vertices to which it has outbound edges.

Claim: Every DAG has a topological ordering
Proof:

Lemma 1: If G is a DAG, it has a vertex with no incoming edges

Proof by contradiction:

- Assume every node in G has an incoming edge
- Pick a node V and follow one of its incoming edge backwards e.g (U,V) which will visit U
- Do the same thing with U, and keep repeating this process
- Since every node has an incoming edge, at some point you will visit a node W 2 times. Stop at this point
- Every vertex encountered between successive visits to W will form a cycle (contradiction that G is a DAG)

Lemma 2: If G is a DAG, then it has a topological ordering

Constructive proof:

- Pick node V with no incoming edge (must exist according to previous lemma) remove V from G and number it 1
- G-{V} must still be a DAG since removing V cannot create a cycle
- Pick the next node with no incoming edge W and number it 2
- Repeat the above with increasing numbering until G is empty
- For any node it cannot have incoming edges from nodes with a higher numbering
- Thus ordering the nodes from lowest to highest number will result in a topological ordering

There are two ways to do topological ordering – Kahn's algorithm and DFS.

Kahn's Algorithm

Kahn's algorithm uses a modified version of BFS to give us a valid topological sort. This is done through the use of the following modifications:

- Replace the visited array with an indeg array that keeps track of the in-degree of every vertex in the DAG
- Use an ArrayList toposort to record the vertices

The concept is to constantly pull out the vertex with no incoming edges. This is only possible as a DAG has no cycles, aka there are no edges going in reverse order, which prevents some vertices from being removed due to their "cyclic dependency".

The pseudo code is as follows:

```
// initialisation
for all v in V
      indeg[v] ← 0 // O(V)
      p[v] ← -1
for each edge (u,v) // get in-degree of vertices
      indeg[v] ← indeg[v] + 1 // O(E)
for all v' where indeg[v'] = 0
      Q ← {v'} // enqueue v', O(V) if totally disconnected

// execution - O(V+E)
while Q is not empty
      u ← Q.dequeue()
      append u to back of toposort
      for all v adjacent to u // order of neighbor
            if indeg[v] > 0
                  indeg[v] ← indeg[v] - 1
            if indeg[v] = 0 // add to queue
                  p[v] ← u
                  Q.enqueue(v)
```

Here's the rough breakdown of what's happening:

- We first initialize an array of in-degrees and predecessors
- Running through the edges, we increase the in-degree of edges on the receiving end
- Unlike the usual BFS where we start from a source vertex, we start from all vertices with an in-degree of zero. We enqueue them.
- We dequeue the vertices and process their edges, decreasing all of their neighbours' in-degrees by one.
- If the neighbours now have zero in-degree, we queue them as well, as well as record their predecessor.
- Repeat until all vertices have been processed, which is when the queue is empty.

Kahn's Algorithm Analysis

The overall algorithm runs in O(V+E) time, as the initialization takes around O(max(V, E)), while the main execution is O(V+E) from BFS. We need to add V and E as they are generally independent.

We are also able to toposort all components since the initialization queues all vertices with in-degree of 0 in.

DFS Topological Ordering

The process of using DFS for topological sort is to output the vertices in reverse topological ordering, then reversing it at the end. The reason why this works is because due to the recursive/stack nature of DFS, if we process a vertex during the tail part of recursion, all vertices that it could possibly travel to that is later than itself in the topological ordering have already been visited. As such, if we append it to the topological sort array at that point, it will be in reverse topological order. It is somewhat similar to visiting from the "leaf" vertices of the DFS tree upwards to the root(s).

This is the pseudo code:

```
// in the main method
for all v in V
      visited[v] ← 0
      p[v] ← -1
clear toposort
for all v in V
  if visited[v] == 0
      DFSrec(v) // start the recursive call from s
reverse toposort and output it

// execution of DFS
DFSrec(u)
      visited[u] ← 1 // to avoid cycle
      for all v adjacent to u // order of neighbor
            if visited[v] = 0 // influences DFS
                  p[v] ← u // visitation sequence
                  DFSrec(v) // recursive (implicit stack)
      append u to the back of toposort // "post-order"
```

The main function helps us to catch all components as well, similar to how component counting works.

To better visualize it:

- Suppose we have visited all neighbours of 0 recursively with DFS
- Toposort list = [[list of vertices reachable from 0], vertex 0]
    - Suppose we have visited all neighbours of 1 recursively with DFS
    - Toposort list = [[[list of vertices reachable from 1], vertex 1], vertex 0]
    - and so on...
- We will eventually have the reversed toposort list

	BFS	DFS
	Queue	Recursion/Stack
Fastest Time Complexity	O(V+E)	O(V+E)
Path Reconstruction	O(V+E) – BFS Spanning Tree	O(V+E) – DFS Spanning Tree
Reachability Test	O(V+E) – check visited array	O(V+E) – check visited array
Shortest Path for Unweighted Graph	O(V+E) – BFS from source and path reconstruction	Not Applicable
Counting Components	O(V+E)	O(V+E)
Topological Sort	O(V+E) - Kahn's Algorithm with modified BFS using in-degree array	O(V+E) – Reverse toposort
SSSP	Can be applied for unweighted graphs Can be applied for trees	Can be applied for trees

Additional Proving Questions

Claim: A general graph is stored in an edge list. There is no way to run DFS on this graph faster than O(VE), because for every vertex, we need O(E) time to determine its neighbours.

Proof:
False. If we sort the edge list by the starting vertex followed by the second vertex, we can potentially reduce any searching time to be O(log E) for binary search. The sorting itself will be O(E log E). This combined with the processing of N vertices results in around O(Max(V log E/E log E)).

We can also translate an Edge List into an Adjacency List, which takes O(E) time. Then we perform BFS/DFS with Adjacency List, which takes O(V+E) time.

Claim: A connected graph has at least E = V - 1 edges. Prove or disprove it.

Proof:
True, as V-2 edges will not allow all vertices to be connected. V-1 allows us to form a tree.

Claim: A single linked list has V vertices and E-1 directed edges. These edges go from vertex i to i + 1, where 0 <= i < (V - 1). Therefore, there is only one topological sort. Prove or disprove it.

True, since the edges only point in a single increasing order. This results in only one order of sorting. There is only one vertex with no incoming edge, and one with no outgoing edge. Hence the topological sort must start with that vertex and end with the latter.

Backtracking DFS to find all possible Topological Sorts

The backtracking DFS is to mainly generate all possible orderings. The pseudo code is as follows:

```
DFS_BT(u, count, permutation) {
      visited[u] = 1 // to avoid cycle
      if count == V // have created a permutation of a V vertices
            check if permutation is valid topological ordering
            if valid output the permutation otherwise do nothing
      for all v adjacent to u // scan all neighbors of u
            if (visited[v] == 0)
                  DFS_BT(v, count+1, permutation+",v")
      visited[u] = 0 // let vertex u be reusable later
}
```

To check if the permutation is a valid topological ordering:

1. Have an integer array order and a Boolean array visited1 which is initialized to false.
2. For a permutation, go through the vertices from left to right (if using a "," separated string, we can easily tokenize the vertex numbers and convert back to integers and store in an array) and set order[v] = position of v in the permutation for each vertex v.
3. Go through the vertices again from left to right and for each vertex v, process all outgoing edges (v,u). If visited1[u] == true and order[u] < order[v], this is not a valid permutation (backward/leftward pointing edge). After processing outgoing edges, visited1[v] = true.
4. If no backward pointing edge in step 3 this is a valid topological ordering.