Graph Theory

Contents-

Breadth First Search

To find the shortest path from a source to a destination.
Data Structure Required- Queue
Arrays- Path, Visited

Steps-

1. Add source to queue
2. Iteration starts, if queue is empty then stop
3. Remove an element from queue
4. If removed element is destination then stop
5. Traverse adjacent vertices of the removed element
6. If adjacent vertex is not visited then
    a. Set its visited flag to true
    b. Update path(set path[adjacent vertex] to index of removed element)
    c. If it is destination then stop
    d. Else add it to the queue

Dijkstra's Algorithm

To find the shortest path from a source to a destination in a weighted graph. It is a vertex based algorithm and very much identical to BFS.
Data Structure Required- Min Heap
Arrays- Path, Visited, Weight

Steps-

1. Add source vertex to heap and set its weight as 0.
2. Iteration starts, if the heap is empty, STOP.
3. Remove an element from the heap and set its visited flag to true
4. If removed element is destination, STOP.
5. Traverse adjacent vertices of removed element
6. If adjacent vertex is source OR already visited Do Nothing.
7. Else, calculate {(weight in weight array of element removed from heap)+(weight of adjacent vertex in graph)} THEN compare it with (weight of adjacent vertex in weight array),

        a. If the sum is equal or greater Do nothing
        b. If sum is lesser
            i. Update weight in weight array of adjacent vertex.(set it to sum)
            ii. Update path (set path[adjacent vertex]=removed element)
8. Update adjacent vertex in min heap

Note-
1. Min heap elements are arranged according to their weights in weight array
2. To update heap
    a. If element exists in heap
        i. Remove element
        ii. Add updated element
    b. Just add the element

Depth First Search

To find the shortest path between two nodes.
Generally most DFS implementations don't guarantee the shortest path, but in the implementation below a depth array is used to figure out the shortest path.
It is an edge-based algorithm.
Data Structure Required- Stack
Arrays- Path, Visited, Depth

Steps-

1. Push source to stack and set its depth as 0
2. Iteration starts, is stack is empty STOP.
3. Pop an element from stack
4. If destination is visited AND depth of popped element is greater than depth of destination then Do Nothing.
5. If popped element is source AND is visited Do Nothing
6. Traverse adjacent vertices of popped element
7. If adjacent vertex is visited
    a. If (depth of popped element)+1 is LESS than (depth of adjacent vertex)
        i. Update depth of adjacent vertex to (depth of popped element)+1

```
            ii. Update path (path[adjacent vertex]=popped element)
8. Else (adjacent vertex is not visited)
    a. Update path, depth and visited flag of adjacent vertex and push it into stack
9. After iteration ends,
    a. If destination is visited then backtrack the path
    b. Else, no path
```

Bellman-Ford Algorithm

To compute the shortest path from a source to all the vertices in a weighted graph
Data Structures Required- None
Arrays- Path, Weight, isWeightUpdated

Steps-

1. Set weight of source to 0,as isWeightUpdated to true.
2. Do (number of nodes) - 1 times.
    a. Do for each element in graph
        i. If weight of element not updated,Do nothing
        ii. Traverse adjacent vertices of each element
            1. Sum= (weight of each element from weight array)+(weight of adjacent vertex from graph)
            2. If weight of adjacent vertex not yet updated OR sum<(weight of adjacent vertex from weight array)

3. Update path, weight, isWeightUpdated of adjacent vertex

After this process ends, it is crucial to determine if the graph has a negative cycle. For that, we need to run the relaxation process(Step 2) once again, only this time when we encounter ‘sum’ to less than weight of adjacent vertex in weight array, we conclude that the graph has a negative cycle.

Binary Weighted Graph

To find the shortest path in a binary weighted graph
Data Structure Required- Doubly Ended Queue

Steps-

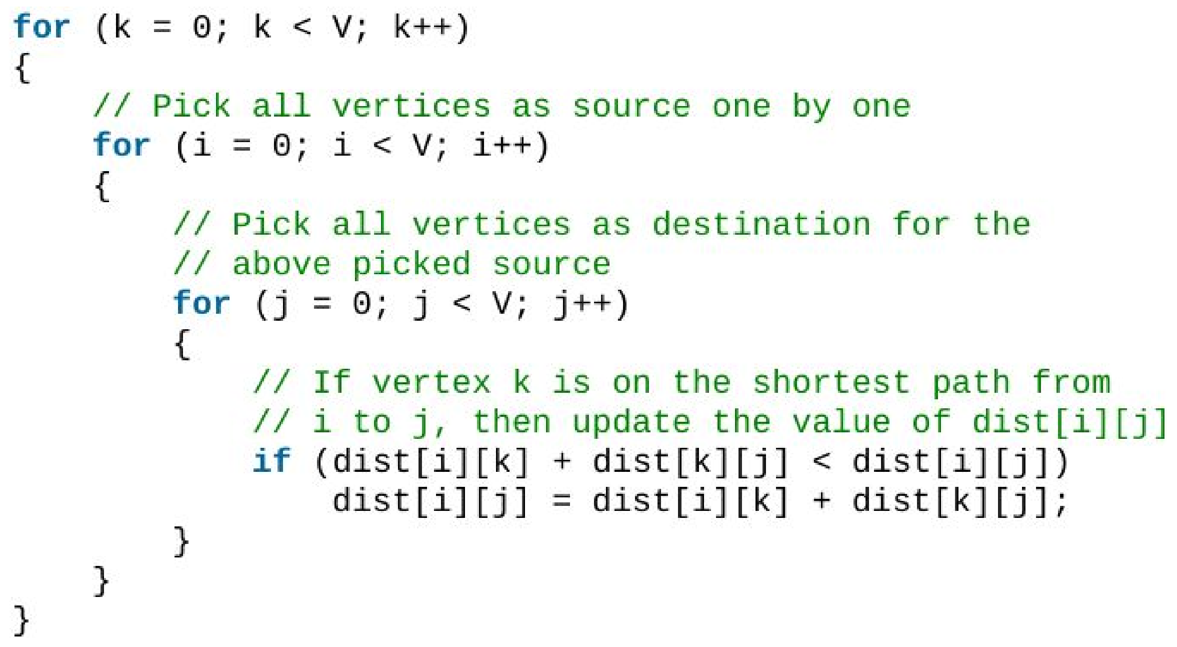
1. Add source to queue and set its weight 0.
2. Iteration starts, if queue is empty then STOP.
3. Remove an element from the front of the queue and set its visited as true.
4. Traverse its adjacent vertices
5. If adjacent vertex is already visited Do nothing
6. Sum= (weight of removed element from weight array)+(weight of adjacent vertex from graph)
7. If weight of adjacent vertex is not updated OR sum is less than the weight of adjacent vertex in weight array.
    a. Update isWeightUpdated, path, weight of adjacent vertex in weight array
    b. If weight of adjacent vertex is lower among the binary weights
        i. Push it to the front of the queue.
    c. Else,
        i. Push it to the back of the queue.

Floyd Warshall Algorithm

To find the shortest distance between all pairs of vertices

Just see the code for yourself-
https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/

Concept-
Let’s say we know the distance from a → b ,b → c . We may or may not know the distance from a → c
If ((a → b)+ (b → c)) > (a → c) then update a → c



```
for (k = 0; k < V; k++)
{
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++)
    {
        // Pick all vertices as destination for the
        // above picked source
        for (j = 0; j < V; j++)
        {
            // If vertex k is on the shortest path from
            // i to j, then update the value of dist[i][j]
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
        }
    }
}
```

Shortest path between all pairs can be found by maintaining a separate path matrix.

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.
Use: When inter-dependent entities are required to be processed such that the entity that is the dependency of some other entity(s) is processed before them.

In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack.
Then just print the contents of the stack in the end.

Another way would be to implement DFS as shown previously in this document and just maintain a separate stack(stk2) in which you push every element that is pushed in stk1.

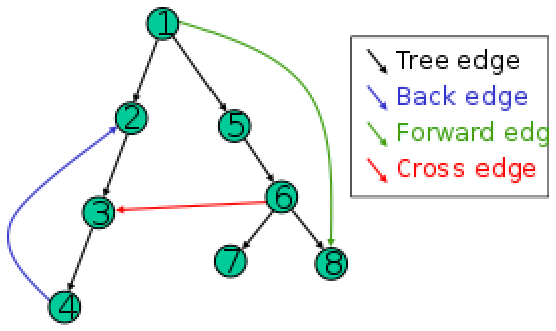
Mother vertex is a vertex such that all the vertices in a graph can be reached from this vertex. There may exist multiple mother vertex in a graph.
A vertex in graph is said to be finished when its DFS call is completed i.e. all its adjacent nodes are visited.

FACT: If there exist mother vertex (or vertices), then one of the mother vertices is the last finished vertex in DFS. (Or a mother vertex has the maximum finish time in DFS traversal).

Based on the fact above,
Method for finding mother vertex-
https://www.geeksforgeeks.org/find-a-mother-vertex-in-a-graph/

1. Find the last finishing node
2. For the last finishing node as source try to reach all the other nodes in graph through dfs/bfs, if you are successful then this node is a mother vertex ,else there is no mother vertex in the graph.

Detecting a cycle in graph is a fairly simple task.
Back edge is an edge that leads back to an ancestor vertex or itself.



If a graph has a back edge then it is cyclic.
To detect a back edge we need to keep track of vertices that are currently being processed(are in recursion stack). If we try to insert a vertex that is already in the recursion stack then a cycle is detected.