Graph Traversals

A traversal is a systematic procedure for exploring a graph by examining all of its vertices and edges. Graph traversal algorithms are key to answering many fundamental questions about graphs involving the notion of reachability.

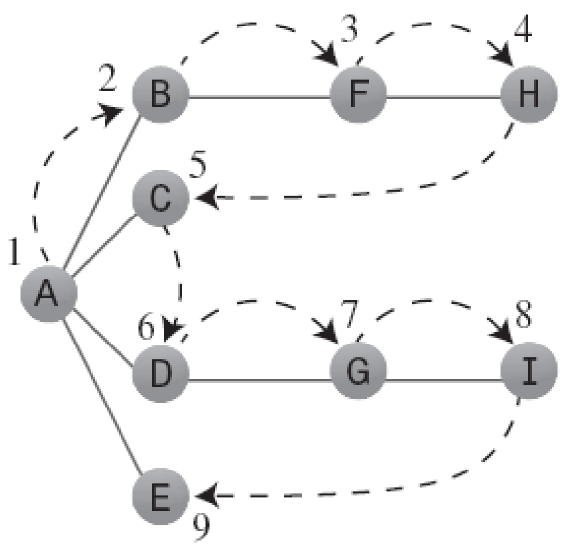
If the size of the graph is finite, DFS would likely find a outlier (larger distance between root and goal) element faster where BFS would find a closer element faster. Except in the case where DFS chooses the path of the shallow element. A DFS doesn't necessarily find the shortest path to a node, while breadth-first search does.

Depth First Search (DFS)

It can be said that the depth-first search algorithm likes to get as far away from the starting point as quickly as possible and returns only when it reaches a dead end. If you use the term depth to mean the distance from the starting point, you can see where the name depth-first search comes from. A big advantage of DFS is that it has much lower memory requirements than BFS, because it’s not required to store all of the child pointers at each level.

DFS for a graph is similar to Depth First Traversal of a tree. The difference is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array for each vertex.

Note: The implementation(GraphDFS.java) prints only vertices that are reachable from a given vertex. To print all vertices of a graph, we need to call DFS for every vertex.

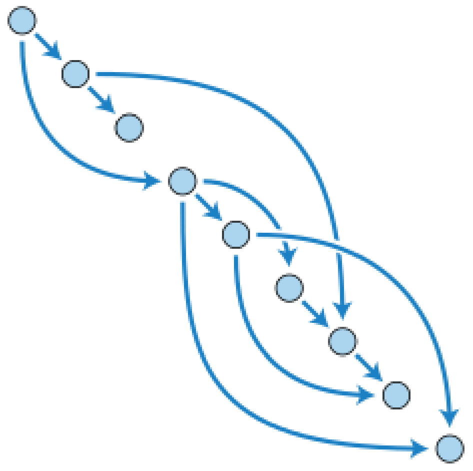


Depth-first search

Applications of DFS in graph

Following are the problems that use DFS as a building block:

1. ***Topological Sorting*** is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, a topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex v, u comes before v in the ordering. Topological ordering for this example is the sequence which does not violate the prerequisite requirement. A topological ordering is possible if and only if the graph has no directed cycles, i.e. if it is a directed acyclic graph (DAG).

   Topological ordering implementation:
   - *Using DFS*
   - *Kahn's Algorithm*
     The approach is based on the below fact that - A DAG has at least one vertex with in-degree 0 and one vertex with out-degree 0.

The main idea is that we maintain a stack or a queue and start off by adding all vertices with no incoming edges and then run either depth first search (in the case of using a stack) or breadth first search (in the case of using a queue) with the modification that we delete edges to adjacent vertices, and only process them when they have no more incoming edges to them.

Pseudo-code (with stack - DFS)

```
V : set of all vertices
adjacent : adjacency list
S : stack of vertices, initialized to be empty

kahnTopologicalSort(V):
result : queue of vertices, initialized to be empty

for each vertex x in V:
    if x has no incoming edges:
    S.push(x)

while (!S.isEmpty()):
    current = S.pop()
    result.enqueue(current)

    for each vertex x in adjacent[current]:
    deleteEdge(current, x)

    if (x does not have any more incoming edges):
        S.push(x)

return result
```

Unlike the method with the queue, we do not need to maintain the set `visited` because we are only concerned with exploring vertices as deep as possible which can be added to the result for a valid topological ordering. When we delete the edge going from the current vertex to each adjacent vertex x, then intuitively, if x still has incoming edges, say from some other vertex y, then it follows that we cannot add x to the result because y must go before x. But if x does not have anymore incoming edges, then we can add it to the result.

Pseudo-code (with queue - BFS)

```
V : set of all vertices
adjacent : adjacency list
visited : set containing all of the visited vertices
Q : queue of vertices, initialized to be empty

kahnsTopologicalSort(V):
result : queue of vertices, initialized to be empty

for each vertex x in V:
   if x has no incoming edges:
   Q.enqueue(x)

while (!Q.isEmpty()):
   current = Q.dequeue()
   result.enqueue(current)

   for each vertex x in adjacent[current]:
   if (!visited.contains(x)):
       visited.add(x)
       Q.enqueue(x)

return result
```

The queue method is very similar to regular breadth first search, and unlike the stack method, we do not need to delete edges or check if adjacent vertices still have incoming edges, simply because the breadth first search mechanism already guarantees that by the time we get to explore a certain vertex x, all of the vertices from its incoming edges have already been visited. We see this intuitively because by definition of breadth first search, we process vertices on the same level before going deeper into adjacent vertices.

Related: **Hamiltonian path** (or traceable path) is a path in an undirected or directed graph that visits each vertex exactly once. Hamiltonian path in a DAG exists if and only if there is unique topological sorting.

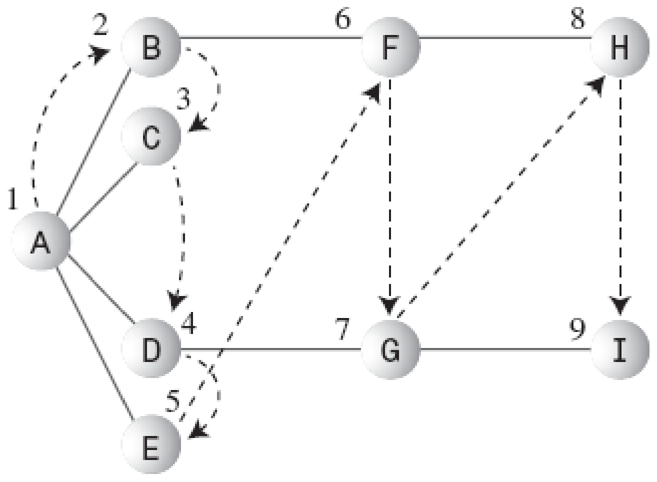
2. Solving puzzles with only one solution, such as mazes.

Running time of DFS

DFS is called at most once on each vertex (since it gets marked as visited), and therefore every edge is examined at most twice for an undirected graph, once from each of its end vertices, and at most once in a directed graph, from its origin vertex.
If we let $n_s \leq n$ be the number of vertices reachable from a vertex s, and $m_s \leq m$ be the number of incident edges to those vertices, a DFS starting at s runs in $O(n_s + m_s)$ time. Hence worst case complexity is $O(n + m)$ for both directed and un-directed graphs.

Breadth First Search (BFS)

In the breadth-first search, the algorithm likes to stay as close as possible to the starting point. It visits all the vertices adjacent to the starting vertex, and only then goes further afield. This kind of search is implemented using a queue instead of a stack.



Breadth-first search

Applications of BFS

1. *Peer to Peer Networks:* In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
2. *Crawlers in Search Engines:* Web Crawlers builds index using Breadth First Traversal starting at source page. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of the built tree can be controlled based on need.
3. *Social Networking Websites:* In social networks, we can find people within a given distance `k` from a person using Breadth First Search till `k` levels.
4. *GPS Navigation systems:* BFS is used to find all neighboring locations from a given start location.
5. *Broadcasting in Network:* In networks, a broadcasted packet follows BFS to reach all nodes.
6. *In Garbage Collection:* Breadth First Search is used in copying garbage collection using Cheney’s algorithm. Breadth First Search is preferred over Depth First Search because of better locality of reference.
7. The *Ford–Fulkerson method or Ford–Fulkerson algorithm (FFA)* is a greedy algorithm that computes the maximum flow in a flow network. Either BFS or DFS can be used to find the maximum flow. BFS is preferred as it reduces worst case time complexity to $O(VE^2)$.

Running time of BFS

Let G be a graph with n vertices and m edges represented with the adjacency list structure. A BFS traversal of G takes $O(n+m)$ time.

Applications that can use either DFS or BFS

1. *Shortest Path and Minimum Spanning Tree:*
   In an unweighted graph, the *shortest path* is the path with least number of edges. With Breadth First, we always reach a vertex from given source using the minimum number of edges.

   Both traversal can produce minimum spanning tree or shortest path tree from a given vertex.

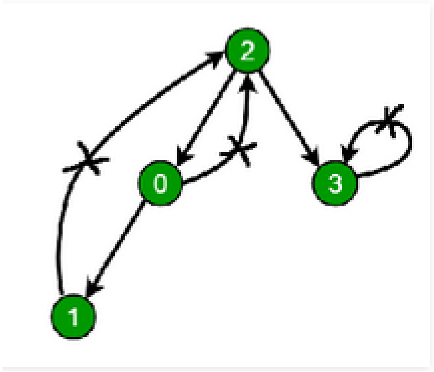
   Minimum spanning tree: A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.
   Shortest-path tree: Given a connected, undirected graph G, a *shortest-path tree* rooted at vertex v is a spanning tree T of G, such that the path distance from root v to any other vertex u in T is the shortest path distance from v to u in G.
2. Test if a *graph is bipartite*
3. *Path Finding* We can either use BFS or DFS Traversal.
   Algorithm to find a path between two given vertices u and z.
   a) Call DFS(G, u) or BFS(G, u) with u as the start vertex.
   b) Use a stack S (in DFS) or a queue Q (in BFS) to keep track of the path between the start vertex and the current vertex.
   c) As soon as destination vertex z is encountered, return the path as the contents of the stack.
4. *Finding Strongly Connected Components* of a graph
   A directed graph is called *strongly connected* if there is a path from each vertex in the graph to every other vertex.
5. *Cycle detection in graph:* A graph has cycle if and only if we see a back edge (already-visited vertex) during DFS traversal.

   While in *undirected graph* back edges are edges from the current vertex to an-already-visited vertex.

   In *directed graph* the definition for back edge is different. A back edge in a directed graph is an edge from current vertex to a vertex still in the recursion stack. (the DFS for this vertex has started but not yet finished)

Problems and solution approaches

- Detect Cycle in Directed Graph

- Using DFS
  DFS for a *connected graph* produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestor in the tree produced by DFS. In the following graph, there are 3 back edges, marked with a cross sign.
  

  For a *disconnected graph*, we get the DFS forest as output. To detect cycle, we can check for a cycle in individual trees by checking back edges.
  To detect a back edge, we can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree. The edge that connects current vertex to the vertex in the recursion stack is a back edge.

  - Using colors
  - Using BFS
- Detect Cycle in Un-Directed Graph
  - Using DFS link 1, link 2
    During DFS, for any current vertex `x` (currently visiting vertex) if there an adjacent vertex `y` is present which is already visited and `y` is not a direct parent of `x` then there is a cycle in graph. The assumption of this approach is that there are no parallel edges between any two vertices.
    Why not parent:
    Let's assume, vertex `x` and `y` and we have edge between them. Now do DFS from `x`, once you reach to `y`, will do the DFS from `y` and adjacent vertex is `x` and since its already visited so there should be cycle but actually there is no cycle. That is why we will ignore visited vertex if it is parent of current vertex.

  - Using BFS
  - Disjoint Set (Or Union-Find) - link 1, link 2
- Clone an Un-directed Graph
- Find the strongly connected components of the directed graph

- Kosaraju's Algorithm { video link }
  Algorithm:
  - Create a order of vertices by finish time in decreasing order.
  - Reverse the graph.
  - Do a DFS on reverse graph by finish time of vertex and created strongly connected components.
    Runtime complexity - *O*(V + E)
    Space complexity - *O*(V)

- Tarjan’s Algorithm
  - Strongly connected component

```
input: graph G = (V, E)
output: set of strongly connected components (sets of vertices)

index := 0
S := empty stack
for each v in V do
    if (v.index is undefined) then
    strongconnect(v)
    end if
end for

function strongconnect(v)
    // Set the depth index for v to the smallest unused index
    v.index := index
    v.lowlink := index
    index := index + 1
    S.push(v)
    v.onStack := true

    // Consider successors of v
    for each (v, w) in E do
    if (w.index is undefined) then
        // Successor w has not yet been visited; recurse on it
        strongconnect(w)
        v.lowlink  := min(v.lowlink, w.lowlink)
    else if (w.onStack) then
        // Successor w is in stack S and hence in the current SCC
        // If w is not on stack, then (v, w) is a cross-edge in the DFS tree and must be ignored
    v.lowlink  := min(v.lowlink, w.index)
    end if
    end for

    // If v is a root node, pop the stack and generate an SCC
    if (v.lowlink = v.index) then
    start a new strongly connected component
    repeat
        w := S.pop()
        w.onStack := false
        add w to current strongly connected component
    while (w != v)
    output the current strongly connected component
    end if
end function
```

- Get the articulation point / get bi-connected components { video link }

```
GetArticulationPoints(i, d)
visited[i] = true
depth[i] = d
low[i] = d
childCount = 0
isArticulation = false
for each ni in adj[i]
    if not visited[ni]
        parent[ni] = i
        GetArticulationPoints(ni, d + 1)
        childCount = childCount + 1
        if low[ni] >= depth[i]
            isArticulation = true
        low[i] = Min(low[i], low[ni])
    else if ni != parent[i]
        low[i] = Min(low[i], depth[ni])
if (parent[i] != null and isArticulation) or (parent[i] == null and childCount > 1)
    Output i as articulation point
```

The child and parent denote the relations in the DFS tree, not the original graph.
If any one of following condition meets then vertex is articulation point.

1. If vertex is root of DFS and has at least 2 independent children.(By independent it means they are not connected to each other except via this vertex). This condition is needed because if we started from corner vertex it will meet condition 2 but still is not an articulation point. To filter out those vertices we need this condition.
2. It is not root of DFS and if [visitedTime of vertex] <= [lowTime of any adjacent vertex] then its articulation point. For a given edge $(u, v) \Rightarrow low[v] \geq dfn(u)$

   Time complexity is *O*(E + V)
   Space complexity is *O*(V)

- Graph Coloring (can be both vertex or edge coloring)
  - *m-color decision problem* (can the graph be colored with m colors?)
    Approaches to solve - backtracking, greedy
  - *m-color optimization problem* (minimum how many colors required to color the graph?)

    The vertex-coloring problem seeks to assign a label (or color) to each vertex of a graph such that no edge links any two vertices of the same color. If `d` is the maximum degree of a vertex in the given graph than the graph can be colored with `(d + 1)` colors.

References

Tarjan's strong connected components wikipedia