Algorithms

Lecture : Graph Exploration - DFS, BFS, DAG & SCC

Imdad ullah Khan

Contents

1 Graph Exploration

The problem that we're interested in the following

Problem : Given a graph $G = \{V, E\}$ and a node $v \in V$ we want to figure out all the nodes that are reachable from v.

Solution: Consider the following algorithm for this problem. We maintain a list of nodes that we have visited. And a *todolist* of nodes that we have to explore (nodes whose neighbors we might not have looked at yet). Initially no node is visited and the todolist just contains the given source node v. At every iteration we remove one of the nodes from our todolist, if it is not visited then we mark it as visited and put all of it's neighbors in the todolist. We keep doing this until the todolist is empty. After that the program terminates.

Algorithm Explore(s)

```
$visited \leftarrow$ zeros($n$)
Insert($todo, s$)
while $todo \neq \emptyset$ do
    $u \leftarrow$ Remove($todo$)
    $visited[u] \leftarrow 1$
    for $v \in N(u)$ do
        if $visited[v] = 0$ then
            Insert($todo, v$)
```

We haven't specified some details here, such as how the algorithm chooses which node to remove from the todolist. So this isn't some particular algorithm, but only a general approach for exhaustively searching a graph, i.e any algorithm which goes through each node in a graph will look something like the above algorithm.

Note that if a node u is reachable from a node v then either it must be a neighbor of v or a neighbor of a neighbour of v and so on. This gives us the following recursive definition of reachability

Definition 1. *Given a graph $G = \{V, E\}$ and two nodes $u, v \in V$, u is reachable from v if u is a neighbor of v or u is reachable from one of the neighbors of v*

This definition allows us to construct a recursive algorithm for the problem. To *explore* a given node v we will mark v as visited and recursively explore neighbors of v. We will do this in a depth first fashion. That is suppose the neighbors of v are visited in the order $u_1, u_2, \ldots, u_k$. Then the neighbors of u_1 will be explored before $u_2, u_3, \ldots, u_k$ are explored. To ensure that we don't keep going around in loops and that the algorithm terminates we will only explore those nodes that haven't yet been visited.

Algorithm Recursive Explore

```
function Explore($G$,$s$)
    $visited[s] \leftarrow 1$
    for $u \in N(s)$ do
        if $visited[u] = 0$ then
            Explore($G$,$u$)
```

Note that in this algorithm there is no todolist. Where did that go? How are keeping track of all the nodes that we have yet to explore? The answer to these questions is that we *are* still keeping a track of the nodes that we still have to explore but now it's being done implicitly through the call stack instead of us having to explicitly maintain a todolist.

Proof of Correctness : We want to prove that the algorithm stated above does what it's intended to do. That is

1. If u is not reachable from the given vertex then $visited[u] = 0$
2. If u is reachable from the given vertex then $visited[u] = 1$

1) is left to the reader as an exercise
We can prove 2) by using induction on the path length. If the path length between u and v is one than that means that u is a neighbor of v and we call explore on all neighbors of v, so $visited[u] = 1$. Suppose the algorithm gives the correct output for any vertex u which is connected to v by a path of length k. We will refer to the set of vertices that are connected to v by a path length of k as S_k. Then any vertex which is connected to u by a path length of $k + 1$ will be a neighbor of one of the vertices in S_k. Since all vertices in S_k are visited Explore must've been called on all of their neighbors, which means visited of all vertices connected to u by a path of length $k + 1$ will be 1 too.

2 Depth First Search

The explore algorithm visits all nodes in *one* connected component of a graph. What if we want to visit *all* nodes in the graph. For that we modify our algorithm in the following way. Once we've explored one connected component we will call explore on any one of the unvisited nodes and we will keep repeating until all nodes have been marked as visited. This called Depth First Search. The pseudocode for the algorithm is given below.

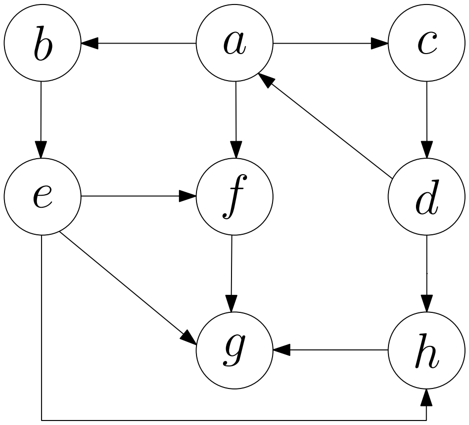
```
Algorithm DFS($G$)
  $visited \leftarrow$ ZEROS($n$)
  for all $v \in V$ do
      if $visited[v] = 0$ then
          EXPLORE($G, v$)
```

Analysis : In this algorithm Explore is called on each vertex of the graph exactly once. Inside the explore subroutine there's a for loop in which some constant amount of work is done for each neighbor of the given vertex (and a constant amount of work is done outside the for loop). That is, for each vertex v of the graph we do $(\mathcal{O}(1) + deg(v))$ work. Recall the handshaking lemma that says

$$\sum_{v \in V} deg(v) = 2|E|$$

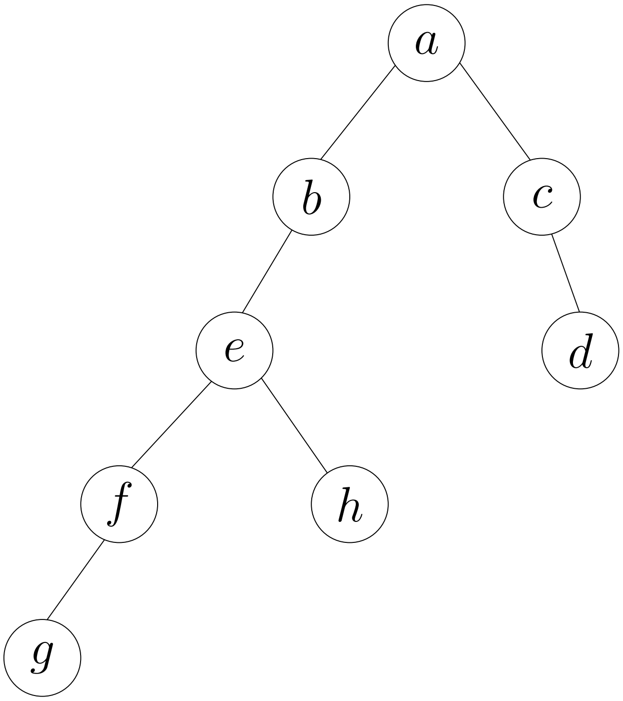
So we get a runtime of $\mathcal{O}(|V| + |E|)$.

Consider the following connected graph



If we run our DFS algorithm on this graph starting from node a and explore neighbours in alphabetical order then we would visit all the nodes in the following order a, b, e, f, g, h, c, d and the set edges we would use would be $E_{dfs} = \{(a, b), (b, e), (e, f)(f, g), (e, h), (a, c), (c, d)\}$. The set of vertices V of the graph and the edges used in dfs (E_{dfs}) together form a tree, called the depth first search tree of the graph G.

Note that if the graph G wasn't connected then we would get a forest instead of a single tree.



Based on the DFS of the graph, we can classify it's edges into four categories

- Tree edges - Edges that are used in the DFS
- Back edges - Edges from a node to it's ancestor
- Forward edges - Edges from a node to a non-child descendent
- Cross edges - All other edges are called cross edges

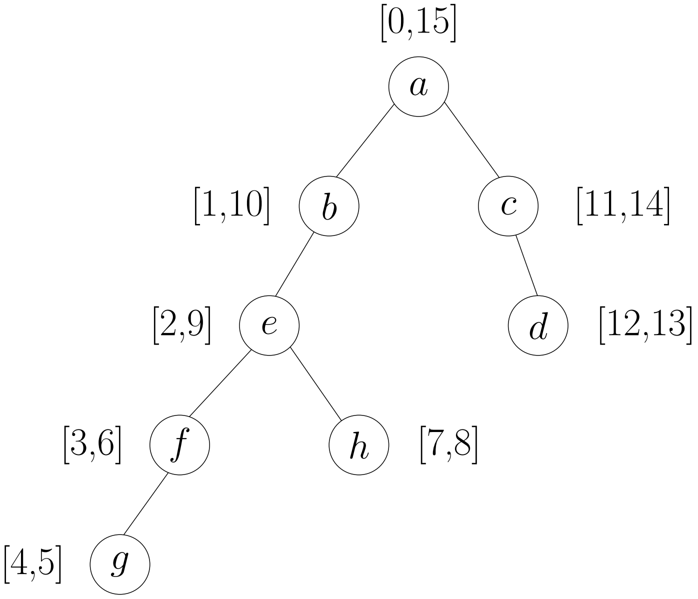
2.1 DFS with Start/Finish Times

Suppose we modify our explore algorithm a little to do some book keeping. We keep a global counter that's 0 initially and at every time we visit a node or leave a node we increment it by one. We keep two arrays *start_time* and *end_time*. When we visit a node u we put the current value of the counter in *start_time*$[u]$ and once we've finished exploring u we put the current value of the counter in *end_time*$[u]$. The pseudocode for this is given below.

Algorithm DFS(G) using Recursive Explore with Start/Finish Times

```
$visited \leftarrow$ ZEROS($n$)                    ▷ Initialize the visited array to n zeros
$s \leftarrow$ ZEROS($n$)                          ▷ Initialize the start_times to zeros
$f \leftarrow$ ZEROS($n$)                          ▷ Initialize the finish_times to zeros
$time \leftarrow 1$
for all $v \in V$ do
    if $visited[v] = 0$ then
        EXPLORE($G, v$)
function EXPLORE($G$,$v$)
    $visited[v] \leftarrow 1$
    $s[v] \leftarrow time$                         ▷ Update the start_time of v
    $time \leftarrow time + 1$                     ▷ Increment time
    for $u \in N(v)$ do
        if $visited[u] = 0$ then
            EXPLORE($G$,$u$)
    $f[v] \leftarrow time$                         ▷ Update the end_time of v
    $time \leftarrow time + 1$
```

This algorithm gives us the following starting and ending time of all the nodes in the graph.



Now suppose we don't have this tree, we haven't run DFS on the graph and someone just gives us these two vectors, the starting times and the ending times of all the vertices in the graph that this algorithm would have outputed. And you're told all the edges in the original graph. Using this information how can you determine which edges are tree edges and which are back edges, cross edges etc ?

Note that if there's an edge between u and v and $[start(u), end(u)] = [a, b]$ and $[start(v), end(v)] = [c, d]$ and $c > a$ and $d < b$ then v must be a descendant of u (Try to figure out how can you distinguish between tree edges and forward edges). And similarly we can tell about back edges. Also note that if the start-end of u and v are non overlapping then the edge must be a cross edge. Consult the problem set (and its solution) to learn more about what information we get from the start-time and end-time of vertices.

3 Directed Acyclic Graph

Definition 2. *A Directed Acyclic Graph (DAG) is a directed graph that has no cycles.*

Note that there could be an undirected cycle in a DAG, but there must not be any directed cycles.

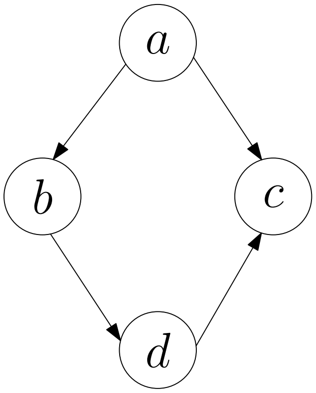
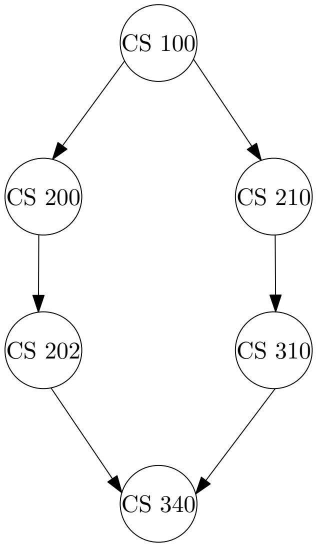


Figure 1: Example of a DAG

DAG's allow us to model all sorts of situations regarding dependencies. For example consider the following graph that models course pre-requisites.



This graph tells us that anyone can take CS100 but in order for someone to take CS200 or CS210 they must first take CS100 and so on. In general we model the dependency of u on v by having an edge from v to u. A valid order in which you can take these courses is called the topological ordering of the DAG.

3.1 Topological Sort

Definition 3. *Topological Sort of a DAG is a linear ordering of its vertices such that for every directed edge uv in the DAG from vertex u to vertex v, u comes before v in the ordering.*

Note that there might be many valid topological sorts for a DAG.

For example in the DAG for the CS courses $CS100, CS200, CS202, CS210, CS310, CS340$ and $CS100, CS200, CS210, CS202, CS310, CS340$ are both valid topological orderings.

So how do we find a topological ordering of a DAG? Note that if a node u has no incoming edge then it doesn't depend on anything. And conversely if a node has an incoming edge then it does depend on something.

Definition 4. *A node in a directed graph is said to be a source if it has no incoming edges, i.e. it's in-degree is* 0*.*

Definition 5. *A node in a directed graph is said to be a sink if it has no outgoing edge, i.e. it's out-degree is* 0.

In Figure 1, vertex a is a source and vertex c is a sink, while vertices b and d are neither.

Does every directed graph has a source and/or a sink? The answer is no, for instance the directed graph in Figure 2 has no source and no sink, as every vertex has in-degree and out-degree 1.

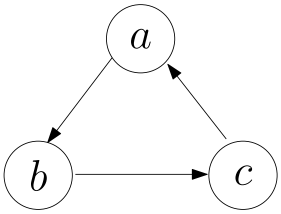


Figure 2: A directed graph with no source and no sink

Coming back to topological sorting, any node which is not a source can't possibly be the first node in the topological ordering because it depends on something else. So in order for a topological ordering to exist there must be a source in every DAG. And indeed there is.

Theorem 1. *Every directed acyclic graph has a source and a sink*

Proof. Suppose on the contrary that G doesn't have a sink vertex. Then pick a node u and mark it visited. Then pick any node v such that $(u, v) \in E$ and mark v visited. (v must exist because u is not a sink). Similarly now pick a neighbor of v that is not visited and mark it visited. Again this neighbor must exist because v is not a sink and v can't be adjacent to any of the already visited nodes since that would mean there's a directed cycle in the graph. And we can go on picking neighbors of neighbors and marking them as visited and we would never stop because there's no sink in the DAG. But the number of vertices in the DAG is finite. Hence we must stop at some time. This is a contradiction. So there must be a sink vertex in the graph. □

To prove that every DAG must have a source, reverse all the edges of the graph. The resulting graph is still a DAG. (Because if there was a directed cycle $u, u_1, u_2, \ldots, v$ in this graph, then there must have been a directed cycle $v, \ldots, u_1, u$ in the original graph). Now since this is a DAG, it must have a sink vertex (we just proved this). Which means the original graph must have had a source vertex.

This theorem gives us the following algorithm for topological sorting. We also define a function findSource(V, E), which takes in a set of vertices and a set of edges and returns a source vertex:

Algorithm findSource(V, E)

```
function findSource(V,E)
    $inDeg[1, \dots, |V|] = 0$                ▷ Initialize in-degree of every vertex to 0
    for edge $e = (x, y) \in E$ do
        $inDeg[y] + +$                        ▷ Increment in-degree for vertex $y$ in edge $e$
    for $i = 1, \dots, |V|$ do
        if $inDeg[i] == 0$ then return $i$
    return 0                                  ▷ If there is no source vertex, return 0
```

Algorithm Topological Sort

```
$sorted \leftarrow [\ ]$
$numNodes \leftarrow |V|$
while not $sorted.length == numNodes$ do
    $v \leftarrow$ findSource$(V, E)$            ▷ Source guranteed to exist by theorem 1
    $sorted \leftarrow [sorted, v]$             ▷ Append source to sorted
    $V \leftarrow V \setminus v$                ▷ Remove v from V
    $E \leftarrow E \setminus \{(v, u) | u \in E\}$   ▷ Remove all edges in E that originate from v
```

3.2 Topological Sort based on finish time

A simpler algorithm for sorting a DAG topologically, depends on the finishing time for each vertex v when DFS is called upon its graph G.

Algorithm Topological Sort based on finish time

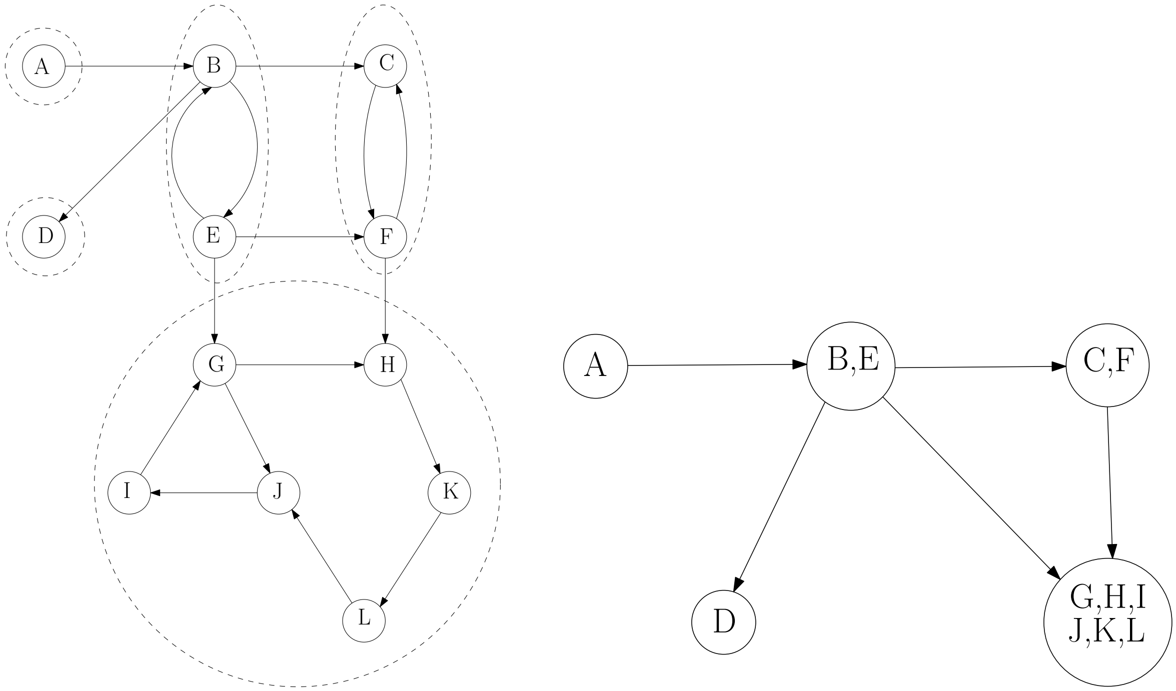
```
$sorted \leftarrow []$                       ▷ $sorted$ is a linked list
DFS$(G)$          ▷ run DFS$(G)$ to compute finishing times $v.f$ for each vertex $v$
as each vertex is finished, insert to front of $sorted$
return $sorted$
```

We provide the proof of correctness for this algorithm below:

Proof. SUppose that DFS is run on a given DAG $G = (V, E)$ to determine finishing times for its vertices. It can be shown that for any pair of distinct vertices $u, v \in V$, if G contains an edge from u to v, then $v.f < u.f$. Consider any edge (u, v) explored by DFS(G). When this edge is explored, v must either have finished, or has not been explored yet. If it were otherwise, then this would mean that (u, v) is a back edge, contradicting the fact that the graph is a DAG. If v has not been explored yet, then it becomes a descendant of u and so $v.f < u.f$. If v has already been visited, then $v.f$ will be set. Since u is still being explored, $u.f$ has not been set, but once it is, we will be sure that $v.f < u.f$. Hence for any edge (u, v) in the DAG, we will have $v.f < u.f$, proving that the vertices will be in topological order. □

4 Strongly Connected Components

It can be of interest to determine the strongly components of a directed graph, and to see how these components are connected to each other. As discussed previously, strong connectivity is defined as there being at least one path from a to b and b to a in a directed graph, therefore a strongly connected component of a directed graph G is a subset S of its vertices such that each vertex $x \in S$ is reachable from every other vertex $y \in S$ and the set S is maximal, i.e. no other vertex can be added to S. As an illustratory example, lets take the following graph:



(a) A directed graph

(b) The SCC-graph

Figure 3: A directed graph and its SCC-graph

As shown in Figure 2(a), the directed graph can be divided into 5 strongly connected components, indicated by the dashed circles. If we were to collapse these components into vertices, we would get what is called a *SCC-graph*, with each vertex representing a strongly connected component, as shown in Figure 2(b). An edge from vertices a and b in a SCC-graph indicate that there is atleast one edge that connects a vertex in the component indicated by a to a vertex in the component indicated by b. This graph is commonly referred to as a *strongly-connected component graph* G^{SCC}.

We outline an algorithm for finding out the strongly connected components of a graph, one that uses the transpose of a graph. The transpose of a directed graph is simply the same graph with the directions of its edges reversed: (Taken from Introduction to Algorithms, 3rd Edition by Thomas Cormen et. al)

Algorithm Strongly-Connected-Components(G)

```
1. call DFS($G$) to compute finishing times $u.f$ for each vertex $u$
2. compute $G^T$
3. call DFS($G^T$) but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4. output vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component.
```

The algorithm depends on a key property of the component graph $G^{SCC} = (V^{SCC}, E^{SCC})$. G has strongly connected components $C_1, C_2, \ldots, C_k$. The vertex set $V^{SCC} = \{v_1, v_2, \ldots, v_k\}$, where v_i is a vertex representing C_i. There is an edge $(v_i, v_j) \in E^{SCC}$ if G contains a directed edge (x, y) for some $x \in C_i$ and some $y \in C_j$. Hence, by contracting all edges whose incident vertices are in the same strongly connected component of G, we get G^{SCC}. The key property is defined by the following theorem.

Theorem 2. *The strongly connected component graph of any graph G is a DAG.*

Proof. Suppose there is a cycle $C = C_1, C_2, \ldots, C_p$, where each C_i is a SCC of G. By definition of this component graph, there is an edge from $(x, y) \in E(G)$, such that $x \in C_1$ and $y \in C_2$. Since every vertex in $x' \in C_1$ can reach x and y can reach every vertex $y' \in C_2$, using this (x, y) edge we get that every vertex $x' \in C_1$ can reach every vertex $y' \in C_2$.

Transitively, we get that every vertex $x' \in C_1$ can reach every vertex $p' \in C_p$. Now using the fact that (C_p, C_1) is an edge in the components graph, by similar argument we get that every vertex in C_p can reach every vertex in C_1. Combining this with the previous fact, we get that C_1 and C_p should be the same SCC (not two different components). The same hold for any pair, so a cycle containing several SCC's would merge all of them into one SCC. □

Theorem 3. *If DFS is run on a graph G with two strongly connected components C and C', and there lies an edge from a node in C to a node in C', then the largest $fin - time$ will be for a vertex in C.*

Proof. Case 1: We start DFS on some vertex $u \in C$. Then the largest finish time will be for u, as all other vertices in C and C' are reachable from this u. We proved this fact above.

Case 2: We start DFS on some vertex $v \in C'$. Then DFS is called on some node of C once C' is completely explored and $fin - time$ of every node in C' is established. Because no vertex in C is reachable from any vertex in C'. Now we will call DFS from some vertex u of C, since all vertices in $C \cup C'$ are reachable from u, those in C' are already visited (hence will give us cross edges) and those in C will be visited and finished before u gets its finish-time, hence it will be the largest. □

Theorem 4. *The algorithm gives us the correct strongly connected components for the graoh input given.*

Proof. We prove this by induction. The inductive hypothesis is that the first k trees produced in line 3 of the algorithm are strongly connected components. For $k = 0$, this is trivial. Assuming this to be true then for the first k trees, we seek to prove it for the $(k+1)$st tree. Let the root of this tree be u, and let this vertex u be in strongly connected component C. In line 3, we select a root based on its finish time being greater than all of the currently unexplored nodes i.e. $u.f = f(C) > f(C')$ for any strongly connected component C' yet to be found. With this, all other vertices of C are now descendants of u in the depth-first search tree. Since we are performing the search on G^T, any edges that leave C must be to strongly connected components that were found before C. Hence there would be no vertex that is the descendant of u in the depth-first search of G^T and is in a strongly connected component other than C. Hence the $(k + 1)$st tree forms exactly one strongly connected component. □

5 Breadth First Search

Breadth First Search (BFS) is another exhaustive search algorithm for graphs. The difference between depth first and breadth first search is in the order in which they visit the nodes. Breadth First Search on a node v first visits all the neighbours of v then the neighbours of neighbours of v and so on. That is,

At iteration 0 we visit v

At iteration i we visit $\{u : visited[u] = 0 \wedge \exists v \in L_{i-1}, (v, u) \in E\}$

(where L_{i-1} is the set of vertices visited at iteration $i - 1$)

Depth first search, on the other hand, would have picked a neighbour u of v, visited everything that was reachable from u before moving on to the other neighbours of v.

The pseudocode for BFS is given below

Algorithm BFS(G)

```
$visited \leftarrow$ ZEROS($n$)
for $v \in V$ do
    if $visited[v] = 0$ then
        bfsExplore($v$)
```

Algorithm bfsExplore(s)

```
$visited[v] \leftarrow 1$
Enqueue($\mathcal{Q}, s$)
while $\mathcal{Q} \neq \emptyset$ do
    $v \leftarrow$ Dequeue($\mathcal{Q}$)
    for $u \in N(v)$ do
        if $visited[u] = 0$ then
            $visited[u] \leftarrow 1$
            Enqueue($\mathcal{Q}, u$)
```

You can prove the correctness of the algorithm similar to how we proved the correctness of DFS, i.e by induction on path length.

Runtime Analysis : If the graph is connected then calling BFS on a vertex would take $\mathbb{O}(|V| + |E|)$ time. The analysis can be done in the exact same way as we did for DFS. Each node is visited exactly once. This gives us the $|V|$ term. And for each vertex v we run a loop $deg(v)$ times. This gives the $|E|$ term.

To get a visual sense of how BFS and DFS differ, consider the following graph and it's DFS tree and BFS tree (BFS tree is defined similar to how we defined a DFS tree)

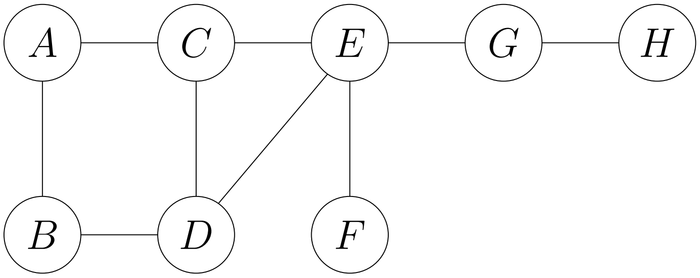
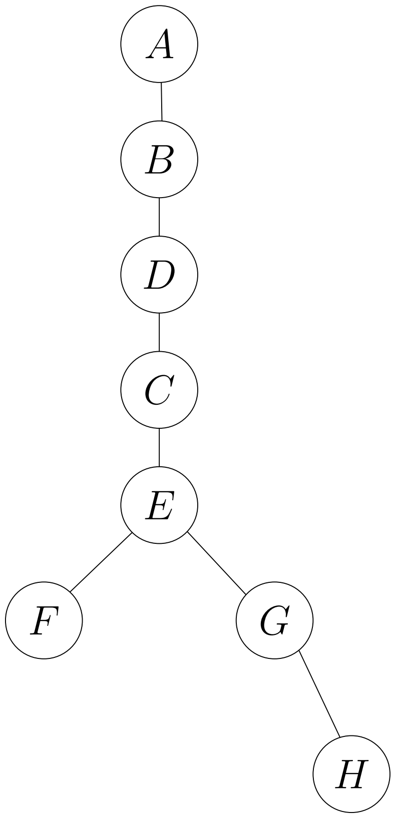
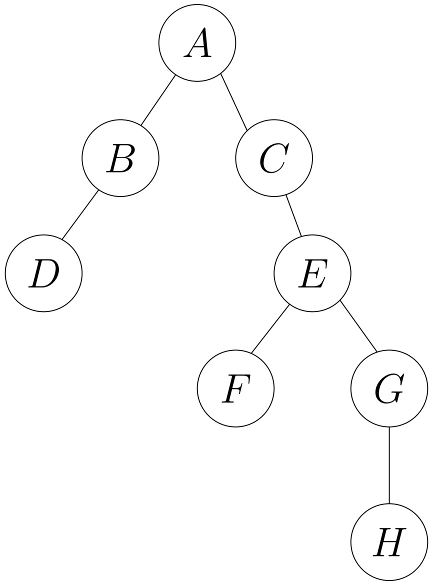


Figure 4: Graph 1



(a) DFS Tree of Graph 1



(b) BFS Tree of Graph 1

5.1 Single-Source Shortest Paths

We defined paths from vertex s to vertex t (or path from s to t if the graph is directed). In many applications we are interested in finding the paths between two vertices that are of minimum length. Such paths are called shortest paths. Distance from s to t is defined to be the length of shortest path from s to t. Note that shortest paths need not be unique. The single-source shortest paths problem is as follows. Given a source vertex s we want to find the shortest paths from s to every other vertex in the graph.

For this we note that when do BFS on s everything at level k of the BFS tree must have distance at most k from the s (because BFS gives a path to it of length k)
Also everything at level k has distance at least k from s. (You can prove this using induction and sub-path optimality. The proof isn't included here because this is homework question)
This gives us the following theorem

Theorem 5. *At level k there are exactly those vertices that are at distance k away from the source vertex*

This means if we want to get the shortest paths from s to all other vertices we can just run BFS on s. And if we want to get the shortest paths between all pairs of vertices we can run BFS once for every vertex.

We can visually see the distances from the BFS tree, but we aren't making any tree when we run the BFS algorithm. We're also not storing any distances, but we can modify the algorithm slightly to do that. The pseudo code for that is given below.

Algorithm Breadth First Search for distances

```
function BFS(s)
    $visited[1..n] \gets$ ZEROS($n$)
    $distance[1..n] \gets$ ZEROS($n$)                    ▷ Initialize the distances to 0
    $todo\_queue \gets [s]$
    $distance[s] = 0$                                   ▷ Distance from $s$ to itself is 0
    while not IS_EMPTY($todo\_queue$) do
        $v \gets todo\_queue.dequeue()$
        $visited[v] \gets 1$
        for all neighbours $u$ of $v$ do
            if not $visited[u]$ then
                $todo\_queue.enqueue(u)$
                $distance[u] \gets distance[v] + 1$     ▷ $(v, u) \in E \to d(s, u) = d(s, v) + 1$
```

5.2 Properties of the BFS Tree

If we run BFS on an undirected graph then the BFS tree can't have any forward or back edges. It can have cross edges but with the following restriction

Theorem 6. *All edges must be b/w same level vertices or consecutive levels*

There can't be any back edges because suppose there was a back edge between a vertex u at level k and a vertex v at level $k - i$ (where $i > 1$). Then there exists a path of length $k - i + 1$ from s to u (go from s to v using the path in the BFS tree of length $k - i$. Then go from v to u). This means that the $d(s, u) = (k - i + 1) < k$. But u is at level k and we proved that everything at level k must have a distance of k from the source vertex. This means the edge we just considered can't exist. Forward edges and back edges in an undirected graph are the same thing. So there can't be any forward edges either. The restriction on cross edges can be arrived at using reasoning similar to what we've used for back edges.