Graph Algorithms: BFS and DFS

This lecture is our first in a series on *Graph Algorithms*. We begin by studying methods of *searching* a graph. In order to follow these lectures, it is important that you have a basic knowledge of graph definitions, including adjacency lists and matrix representations. References are included on the class webpage.

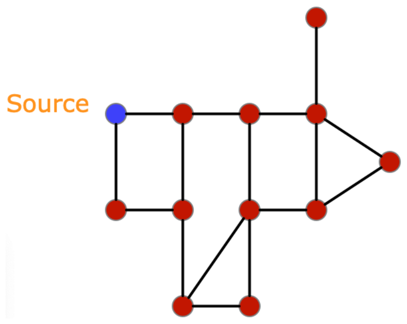
In this series of lectures, we use the usual notation for a graph, G, which consists of a set of vertices, V, and a set of edges, E. Each edge is represented as $(u, v) \in E$ where u, v are vertices in V. Furthermore, following the notation of CLRS, we assume that a vertex v stores the neighbors in and adjacency list: $Adj\,[v]$.

1 Breadth-first search

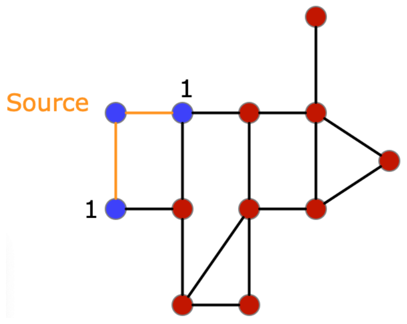
We begin with one of the simplest graph searching algorithms. The breadth-first search algorithm searches through the graph G starting at node **s**, typically referred to as the *source*. From s, the breadth-first search eventually discovers *all* other vertices in the graph that are *reachable* from s. The BFS search algorithm has the following properties:

- BFS discovers all vertices **reachable from** s.
- It computes the **shortest distance** from s to every vertex in G
- It produces a "**breadth-first search tree**" with root s that contains all reachable vertices from s.
- The algorithm works by exploring all vertices at distance k from s before discovering any vertices at distance $k + 1$

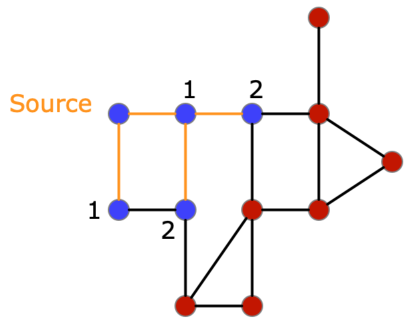
Before presenting the technical details of the algorithm, we begin with a demonstration of the search process of BFS. The graph below begins with all vertices as "undiscovered" (marked in red). BFS starts the search procedure at a given source node, s. This source node is typically part of the input. This is the first node that is *visited* (marked in blue in the figure below) by BFS.



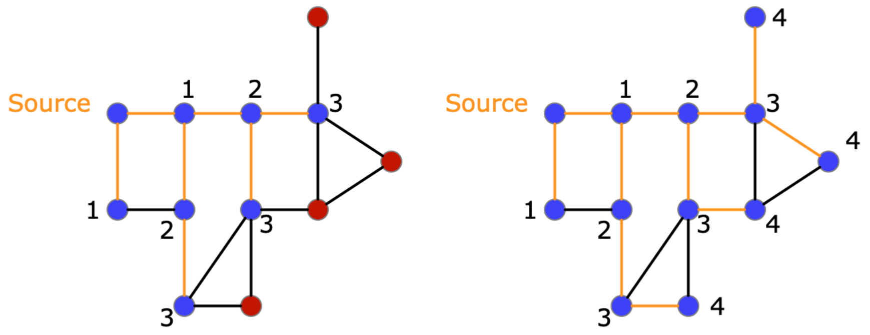
Next, BFS visits all vertices that are at a **distance of one** from the source. Recall that we measure distance as the *number of edges from s to the vertex*.



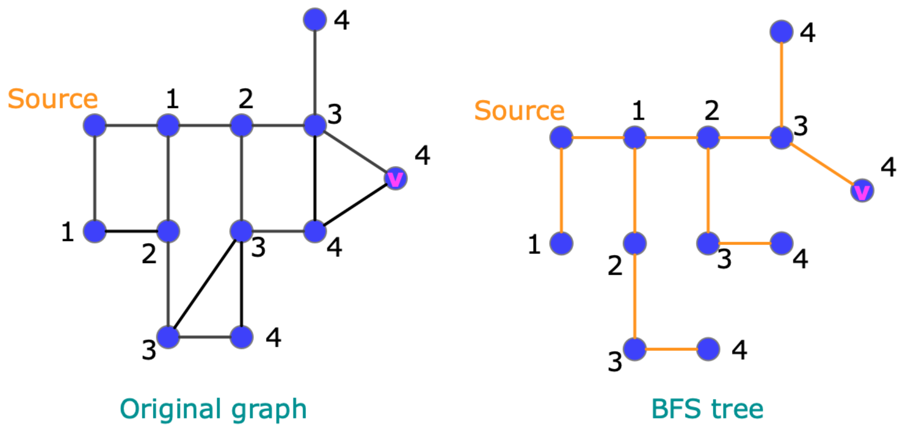
Next, BFS visits all nodes at minimum distance 2 from s, as seen below.



This search process continues until all vertices are visited.



Whenever a new node is discovered along a particular edge in the graph, that edge is added to the **BFS tree** (shown as orange edges). This tree is like a "trace" of the BFS search process. We shall see in the next section that this tree is not necessarily unique. Depending on which vertices were visited along which path, different BFS trees may result.



The shortest distance from the source to vertex v is given by the path of length 4 shown in the tree on the right. Notice that there are certainly *other* paths in G that lead from s to v, however the path in the BFS tree is guaranteed to have the shortest length.

This important property is stated below:

The **shortest path** from s to any vertex v in G is represented by a path the BFS tree

1.1 Additional tools required by the BFS algorithm:

In this section we look at the actual details of the BFS algorithm. Let's begin by looking at the data structures and graph attributes that are needed. The BFS algorithm processes the vertices of G by marking them as "visited" once they are discovered. Therefore we will need some way to keep track of this property. Secondly, we need a data structure that maintains a list of the vertices that should be searched next - and that structure should **ensure that the vertices are visited in order of distance from** s. We also need to keep track of which edges are part of the final BFS tree, and what the final shortest distance is from the source to each vertex. The tools necessary for each of these requirements are explained below:

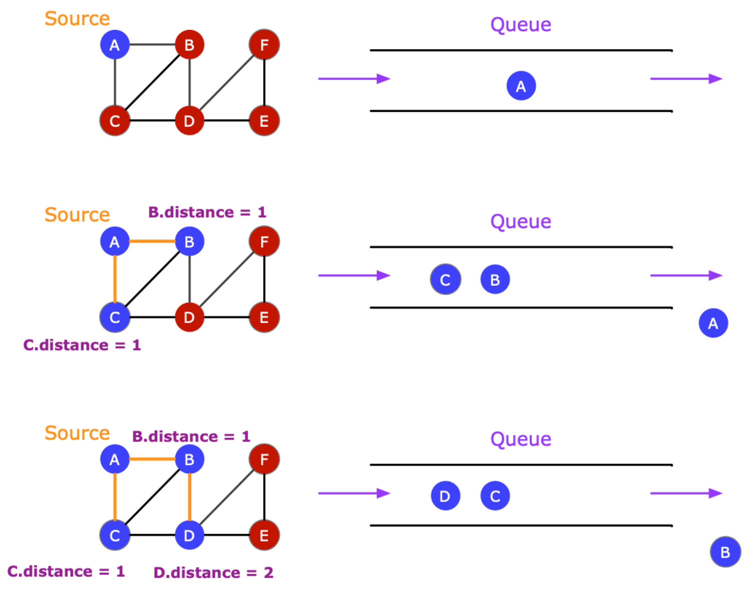
1. Recording when a vertex has been visited:

Each vertex v contains an attribute $v.visited$ which is a boolean variable that indicates if this vertex has been visited. There are variations of this concept in different versions of BFS. You may come across the use of colors, or other boolean variables. In most cases, the idea is the same: we need a key associated to vertex v that stores whether or not that vertex has been visited. This attribute of the vertex v will be set to *true* when that vertex is discovered by the BFS algorithm:

v.visited =false
v.visited =true

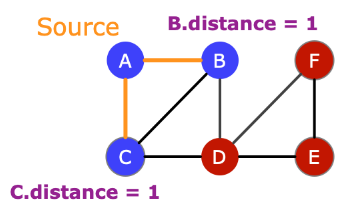
2. How to keep track of which vertices to visit next:

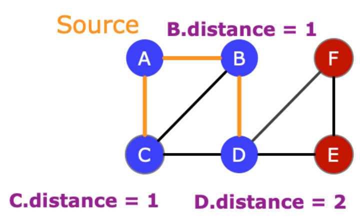
The BFS algorithm uses a QUEUE, Q, to maintain a list of the vertices to be visited. When a vertex v is visited by BFS, the *unvisited* neighbors of v are added to the queue. The vertices are processed one at a time by removing them ($Dequeue$) from the queue. An example of this process is shown below. In this example, vertex A is removed from the queue and the neighbors of A, vertices B and C, are marked as visited and added to the queue. At the next stage, vertex B is dequeued, and its unvisited neighbor (vertex d) is marked as visited and added to the queue. The properties of the queue are such that we *enqueue* on one end and *dequeue* on the other. Therefore the vertices are added to the queue in **order of their distance from** s.



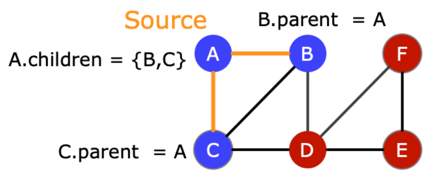
3. Recording the distance from s to v

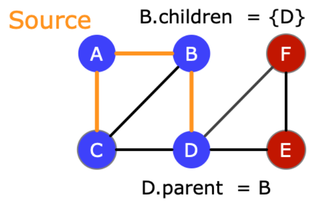
The BFS procedure not only visits all the nodes in G that are reachable from s, but it also records the minimum distance form s to any vertex v. This is done by using an additional attribute called $v.distance$ which represents the minimum path length from s to vertex v.





4. Building the breadth-first search tree during the search:
In order to store the structure of the BST tree, we need a method to record which vertices are discovered along which path. This is done using the attributes $v.children$ and $v.parent$. When a vertex v is removed from the queue, his *unvisited* neighbors are added to the queue (as described above) and then we add these neighbors as **children of vertex** v. In the example below left, when vertex A is removed from the queue, the unvisited neighbors B and C are added to $A.children$. The parent pointers are set as: $B.parent = A$ and $C.parent = A$. The next vertex to be removed from the queue is vertex B (below right). The unvisited neighbor D is added to $B.children$, and we set $D.parent = B$.





In summary then, each vertex uses the additional attributes which will be necessary for BFS.

$$v.distance, \quad v.parent, \quad v.children, \quad v.visited$$

We are now ready to describe the BFS algorithm using the above mentioned attributes, along with the Queue Q.

1.2 The BFS algorithm

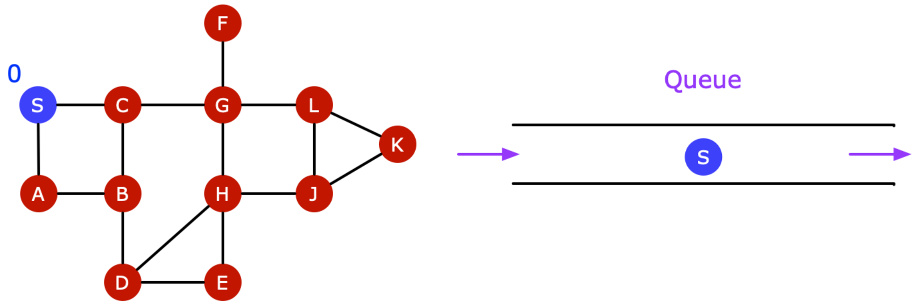
Input: The input to BFS is a graph G, consisting of a list of vertices V and a list of edges E. Recall that the neighbors of a vertex v are stored in $Adj\,[v]$. In the example below, $Adj\,[s] = \{a, c\}$. The algorithm also takes as input a source vertex s.

The Queue operations: The algorithm uses a queue Q, and the operation $Enqueue(Q, v)$ in order to add a vertex to the queue, and the operation $Dequeue(Q)$ to remove a vertex from the queue.

BFS(G,s)

Step 1: For all vertices in G, initialize them as unvisited: for all $v \in V$ set $v.visited = false$. Initialize the parent pointers to NIL for each vertex. Initialize the list $v.children$ to NIL for each vertex.

Step 2: Node s is the starting node, set $s.visited = true$ and $s.distance = 0$. Add s to Q.

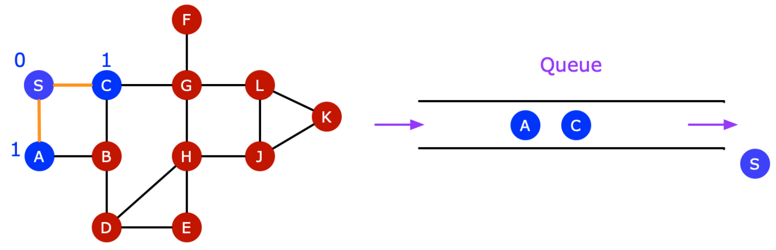


Step 3: Now we are ready to carry out the search. Vertices are removed from Q one at a time. When a vertex is visited, its unvisited neighbors are added to Q.

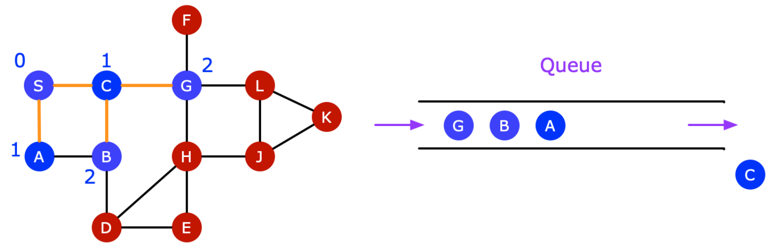
```
while $Q \neq empty$
    u = DEQUEUE(Q)
    for each $v$ in $Adj[u]$
        if $v.visited = false$
            $v.visited = true$
            $v.distance = u.distance + 1$
            $v.parent = u$
            Add $v$ to $u.children$
            $ENQUEUE(Q, v)$
```

We trace through the execution of the above algorithm using the example shown in step 2.

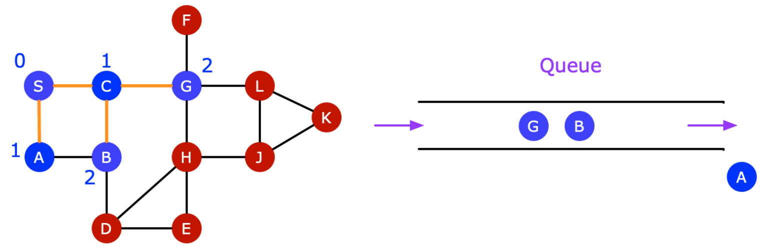
Vertex S is removed from Q. Neighbors C and A are added to Q and marked as visited. This distances are set to $s.distance + 1$, and their parent pointers are set to s. Furthermore, both A and C are added to $s.children$.



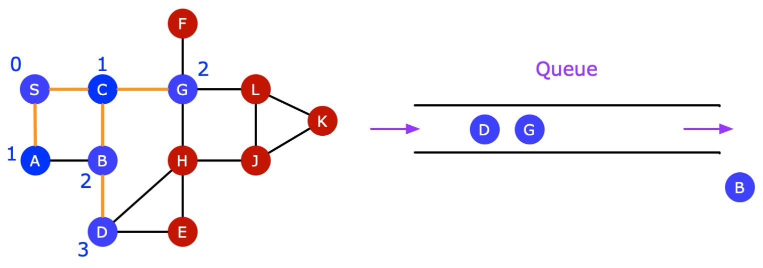
Vertex C is removed from Q. Neighbors B and G are added to Q and marked as visited. This distances are set to $C.distance + 1 = 1 + 1 = 2$, and their parent pointers are set to C. Furthermore, both B and G are added to $C.children$.



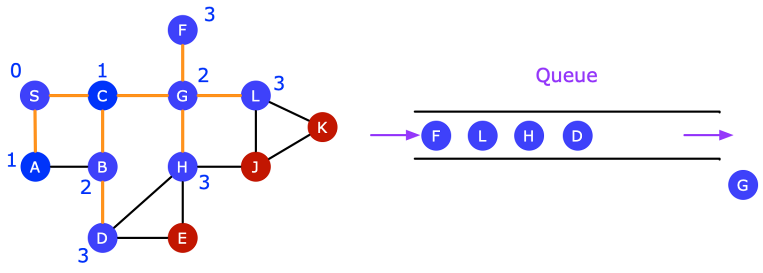
Vertex A is removed from Q. There are no unvisited neighbors.



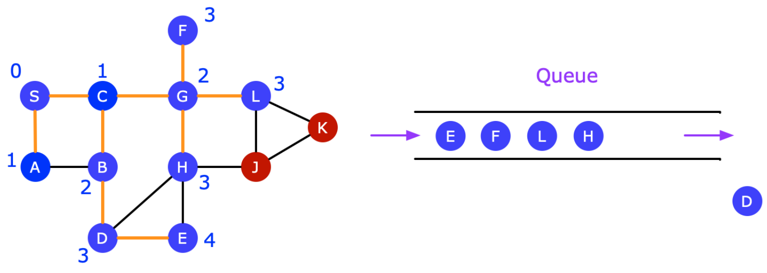
Vertex B is removed from Q. Neighbor D is marked as visited and added to Q. We set $D.distance = B.distance + 1 = 2 + 1 = 3$, and $D.parent = B$, and $B.children = \{D\}$.



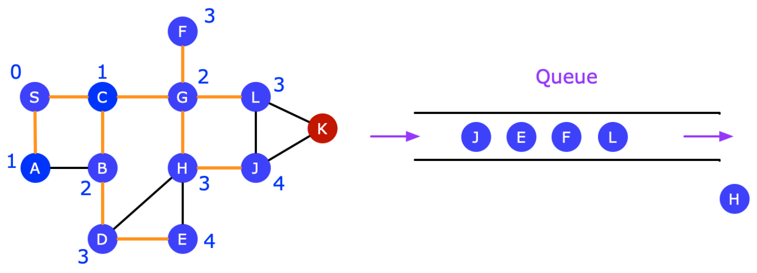
Vertex G is removed from Q. Neighbors H, L, F are marked as visited and added to Q. We set each distance to $G.distance + 1 = 2 + 1 = 3$, and we set each parent to G. Finally $G.children = \{H, L, F\}$.



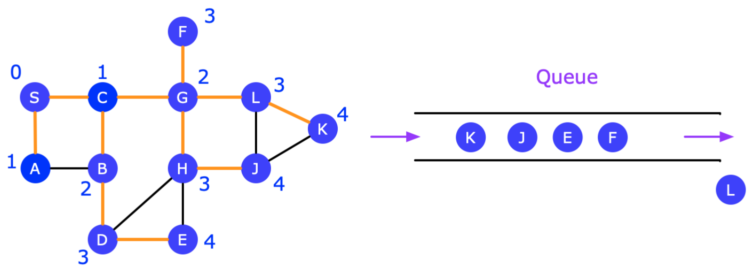
Vertex D is dequeued. Neighbor E is marked as visited and enqueued. We set $E.distance = D.distance + 1 = 3 + 1 = 4$ and $E.parent = D$ and $D.children = \{E\}$.



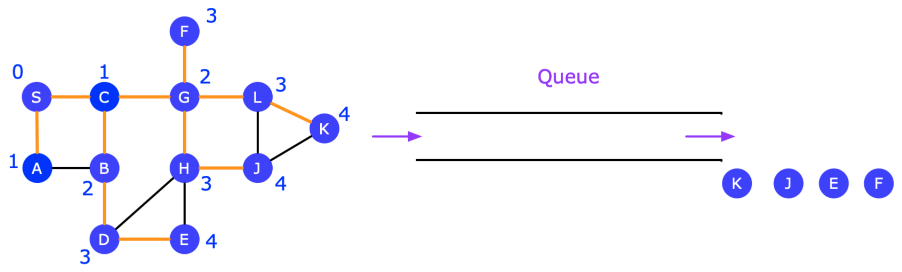
Vertex H is dequeued. Neighbor J is marked as visited and enqueued. We set $J.distance = H.distance + 1 = 3 + 1 = 4$, and $H.children = \{J\}$ and $J.parent = H$.



Vertex L is dequeued. Neighbor K is enqueued. We set $K.distance = L.distance + 1 = 3 + 1 = 4$, and $L.children = \{K\}$ and $K.parent = L$.



Finally, the remaining vertices F, E, J and K are dequeued one by one. None of them have any unvisited neighbors. The main while loop of BFS terminates. Each vertex has the correct shortest distance from s shorted in $v.distance$, and the BFS tree is complete.



Runtime:
The initialization of BFS in Step 1 runs in time $O(V)$. During the execution of BFS, each vertex is placed in the queue once. Therefore the runtime of adding and removing to Q over the entire execution of BFS is $O(V)$. Secondly, the for loop above is executed for each vertex u in $Adj\,[v]$ for every vertex v. Since each edge is stored *twice* in the adjacency list, the body of the for loop is actually executed *twice* for each edge in the tree. However, the run time of the for loop is still $O(E)$. The overall runtime of BFS is therefore $O(V + E)$.

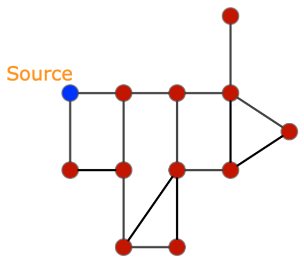
In the next section we look at a different search algorithm, one that doesn't branch out in the same way BFS does, but instead opts to search "deeper" into the graph G before branching.

2 Depth-first search

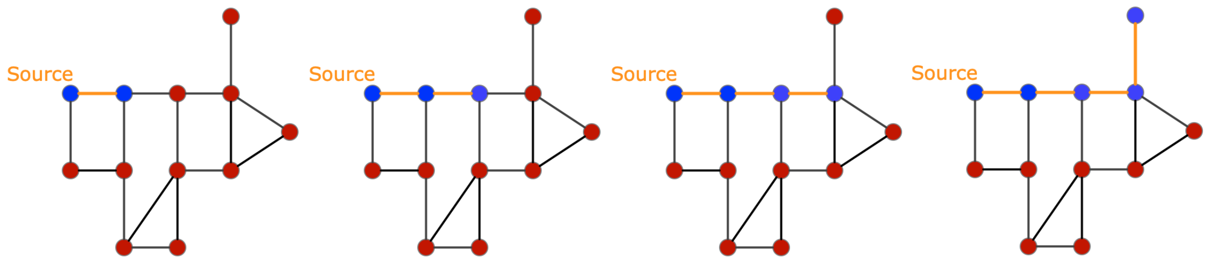
2.1 Overview

The depth-first approach is to search at "as far as possible" in the graph before branching out. Imagine searching along a path exploring unvisited vertices as far as possible. When there are no more unvisited vertices to discover, the search backtracks and looks for paths from the last visited vertex. As with BFS, the DFS algorithm can run on both undirected and directed graphs.

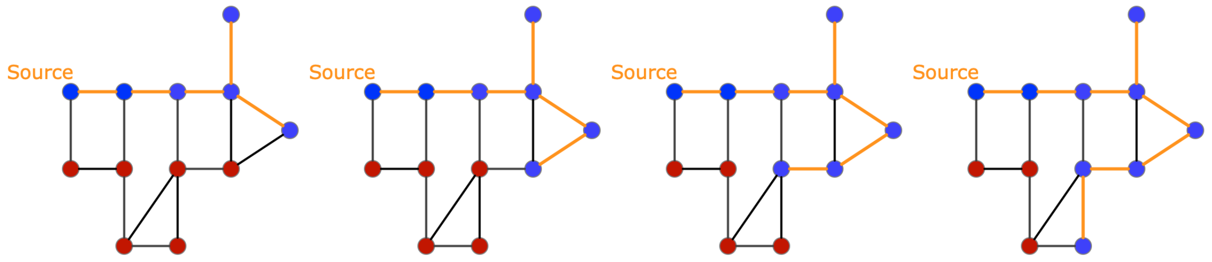
DFS takes as input both a graph G and a source vertex s, and similarly uses the attribute $v.visited$. Initially, only the source vertex is marked as visited (in blue below):



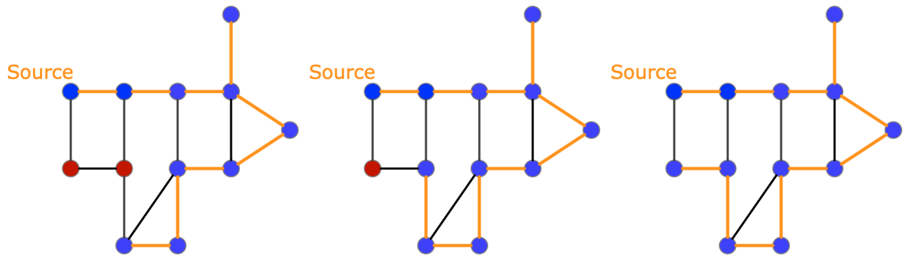
Next, DFS searches along a neighbor of the current vertex, continuing in this way as far as possible:



At this point the search "backtracks" and looks for a previous neighbor which leads to an undiscovered node. The search then continues along edges to new unvisited neighbors:



Eventually all vertices are visited and DFS terminates:



Analogously, the path traced out by this DFS search is referred to as a **DFS tree** (shown in orange). Notice that DFS tree in this example is very different from the BFS tree.

2.2 The DFS Algorithm

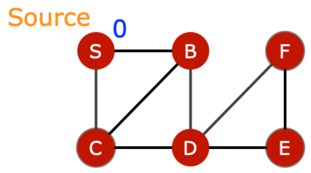
The DFS algorithm requires an attribute $v.visited$ in order to keep track of which vertices have been visited. The proper execution of the algorithm does not actually need anything else! However, as we shall see in the next lecture, some other attributes may be useful once DFS has completd. Therefore we typically use the following attributes:

- The attribute $v.visited$ is used to mark whether or not the vertex has been visited in the search
- The attribute $v.parent$ is used to store the parent node in the DFS tree
- The attribute $v.distance$ is used to store the distance from s to v in the DFS tree. Note that this is **not** the minimum distance in the graph G.

The algorithm does not need an external data structure such as a queue or stack, because it can be written recursively. It is the recursive algorithm that ensures that the search continues along edges to new vertices as long as possible. The easiest way to trace through the algorithm is to keep track of the recursive calls. We demonstrate the process on the small undirected graph below.

DFS(s)

Step 1: For all vertices in G, initialize them as unvisited: for all $v \in V$ set $v.visited = false$. Initialize the parent pointers to NIL for each vertex. Set $s.distance = 0$.

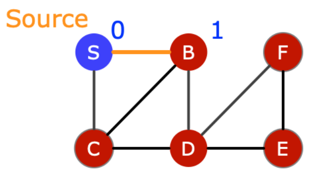


Step 2: Call the recursive algorithm DFS-visit(s). This is a call to the recursive algorithm below:

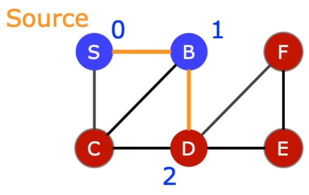
```
DFS-visit(u)
        Mark node $u$ as visited: $u.visited = true$
        For each $v \in Adj[u]$
            If $v.visited = false$
                $v.parent = u$
                $v.distance = u.distance + 1$
                DFS-visit(v)
```

We now illustrate the execution of the above recursive algorithm. The first call is to DFS-visit(s), using the source vertex, s:

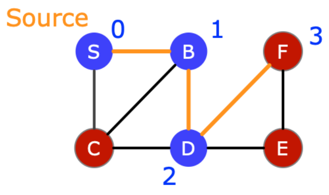
$\boxed{\text{DFS-visit(s):}}$ The call to DFS-visit(s) marks vertex s as visited, and the neighbors of s are processed by the for loop. Since neighbor B is not visited, its parent is set to s, it's distance is set to $s.distance + 1 = 0 + 1 = 1$ and a recursive call is made to DFS-visit(B). Note that the *next* neighbor of s, namely C, will not be verified until DFS-visit(B) terminates.



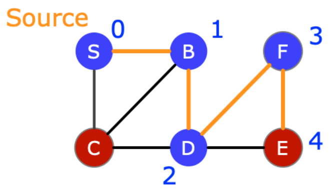
$\boxed{\text{DFS-visit(B):}}$ Vertex B is marked as visited, and suppose the first unvisited neighbor in the for loop is D. The parent of D is set as B, and $D.distance = 2$. Next, a recursive call is made to DFS-visit(D).



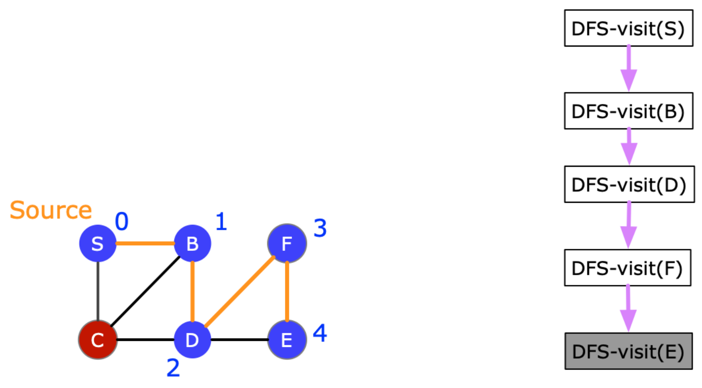
$\boxed{\text{DFS-visit(D):}}$ Vertex D is marked as visited, and suppose the first unvisited neighbor in the for loop is F. The parent of F is set as D, and $F.distance = 3$. Next, a recursive call is made to DFS-visit(F).



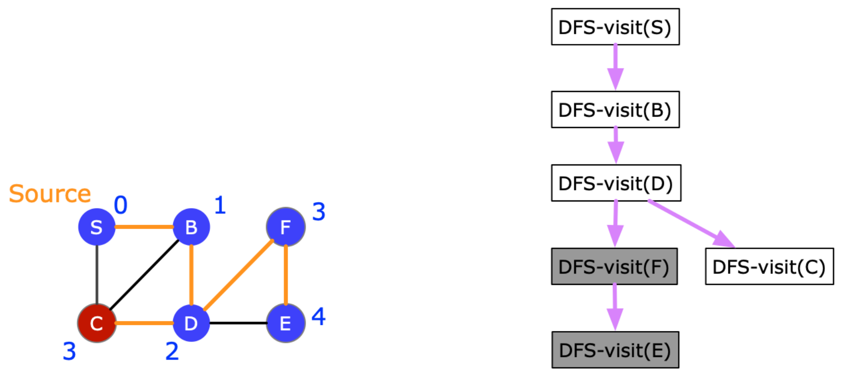
$\boxed{\text{DFS-visit(F):}}$ Vertex F is marked as visited, and the only unvisited neighbor is E. The parent of E is set as F, and $E.distance = 4$. Next, a recursive call is made to DFS-visit(E).



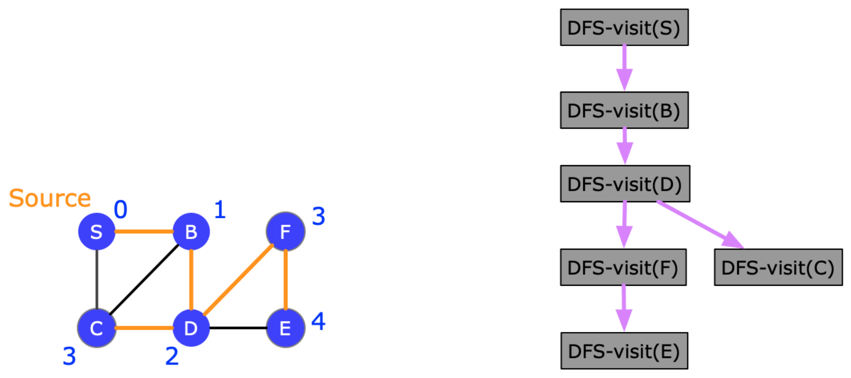
$\boxed{\text{DFS-visit(E):}}$ Vertex E is marked as visited. There are no unvisited neighbors, so no recursive calls are made. The call to DFS-visit(E) terminates. The current stage of the recursive process is shown in the right below:



At this point in the recursion tree, the next vertex to be processed is any other unvisited neighbors of F. Again, there are none, so DFS-visit(F) terminates. The next vertex to be processed is any unvisited neighbors of D. Neighbor C of vertex D is unvisited. The parent of C is set as D and $C.distance = 3$. Next, a recursive call is made to DFS-visit(C).



$\boxed{\text{DFS-visit(C):}}$ Vertex C is marked as visited. Since there are no unvisited neighbors, the call to DFS-visit(C) terminates. At this point in the recursion tree, the next vertex to be processed is any other unvisited neighbors of D. There are none. So the next vertex is any unvisited neighbors of B. Again, there are none. Finally, since vertex S also has no unvisited neighbors, the exectution of all levels of the recursion tree terminates and DFS-visit(s) is complete.



Runtime:

The DFS algorithm marks each vertex as visited only once. For each vertex, the algorithm carries out a for loop for each neighbor of v. Over all vertices in the graph, this is equivalent to doing a constant amount of work for each edge in the tree. The total runtime is then $\Theta(V + E)$.

2.3 Running DFS on the entire graph G

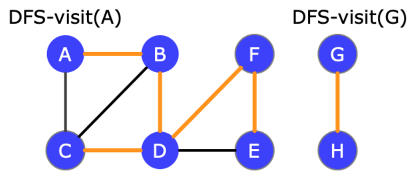
The DFS(s) algorithm only discovers nodes in the graph that are connected to the original node s (by an undirected path for undirected graphs, or a directed path for directed graphs). The DFS(s) algorithm may terminate before the entire graph G has been discovered. In order to visit each node of the graph, we simply restart DFS as long as there are more unvisited nodes in G. The algorithm below does not take a source vertex as input, and instead calls DFS-visit(v) for any unvisited node v. This is repeated until all vertices of the graph are visited.

DFS(G)

Step 1: Initialize all vertices in G to unvisited. Initialize all parent pointers to NIL, and all distances to 0.

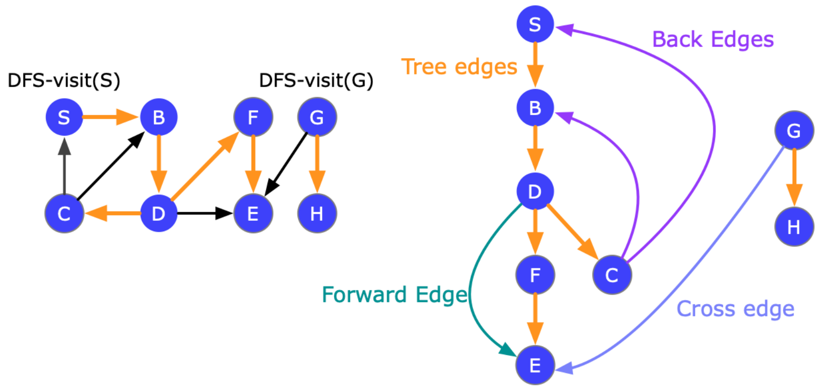
Step 2: Now carry out DFS-visit as long as there are still unvisited vertices:

```
for all $v \in V$
    if $v.visited = false$
        DFS-visit(v)
```



2.4 DFS on directed graphs

The DFS algorithm on a directed graph works in the same way. The resulting DFS tree is stored in the parent pointers which are set during the DFS algorithm. The result of calling DFS(G) may be a single tree, or a set of trees called a *DFS forest*. The example below shows the result of DFS(G) on the directed graph. The DFS forest is drawn on the right. The edges belonging to the DFS trees are shown in orange. The remaining edges of the graph G that are not part of the DFS trees can be classified as either Back edges, Forward Edges or Cross edges.



In summary, the completion of DFS(G) on a directed graph results in the following 4 types of edges:

- Tree Edges: edges that are in G and also part of the DFS tree
- Back edges: non-tree edges $(u, v) \in E$ that connect u to an ancestor v in the tree
- Forward edges: non-tree edges (u, v) that connect a vertex u to a descendant v in the DFS tree.
- Cross edges: all other edges