Graphs: Breadth First Search

**Graph Traversals:** There are a number of approaches used for solving problems on graphs. One of the most important approaches is based on the notion of systematically visiting all the vertices and edges of a graph. The reason for this is that these traversals impose a type of tree structure (or generally a forest) on the graph, and trees are usually much easier to reason about than general graphs.

**Breadth-first search:** Given a graph $G = (V, E)$, breadth-first search starts at some source vertex $s$ and "discovers" which vertices are reachable from $s$. Define the distance between a vertex $v$ and $s$ to be the minimum number of edges on a path from $s$ to $v$. Breadth-first search discovers vertices in increasing order of distance, and hence can be used as an algorithm for computing shortest paths. At any given time there is a “frontier” of vertices that have been discovered, but not yet processed. Breadth-first search is named because it visits vertices across the entire “breadth” of this frontier.

Initially all vertices (except the source) are colored white, meaning that they are undiscovered. When a vertex has first been discovered, it is colored gray (and is part of the frontier). When a gray vertex is processed, then it becomes black.

The search makes use of a queue, a first-in first-out list, where elements are removed in the same order they are inserted. The first item in the queue (the next to be removed) is called the head of the queue. We will also maintain arrays $\text{color}[u]$ which holds the color of vertex $u$ (either white, gray or black), $\text{pred}[u]$ which points to the predecessor of $u$ (i.e. the vertex who first discovered $u$), and $d[u]$, the distance from $s$ to $u$. Only the color is really needed for the search (in fact it is only necessary to know whether a node is nonwhite). We include all this information, because some applications of BFS use this additional information.

Observe that the predecessor pointers of the BFS search define an inverted tree (an acyclic directed graph in which the source is the root, and every other node has a unique path to the root). If we reverse these edges we get a rooted unordered tree called a BFS tree for $G$. (Note that there are many potential BFS trees for a given graph, depending on where the search starts, and in what order vertices are placed on the queue.) These edges of $G$ are called tree edges and the remaining edges of $G$ are called cross edges.

It is not hard to prove that if $G$ is an undirected graph, then cross edges always go between two nodes that are at most one level apart in the BFS tree. (Can you see why this must be true?) Below is a sketch of a proof that on termination, $d[v]$ is equal to the distance from $s$ to $v$. (See the CLRS for a detailed proof.)

**Theorem:** Let $\delta(s, v)$ denote the length (number of edges) on the shortest path from $s$ to $v$. Then, on termination of the BFS procedure, $d[v] = \delta(s, v)$.

**Proof:** (Sketch) The proof is by induction on the length of the shortest path. Let $u$ be the predecessor of $v$ on some shortest path from $s$ to $v$, and among all such vertices the first to be processed by the BFS. Thus, $\delta(s, v) = \delta(s, u) + 1$. When $u$ is processed, we have (by induction) $d[u] = \delta(s, u)$. Since $v$ is a neighbor of $u$, we set $d[v] = d[u] + 1$. Thus we have

$$d[v] = d[u] + 1 = \delta(s, u) + 1 = \delta(s, v).$$
BFS(G,s) {
    for each u in V { // initialization
        color[u] = white
        d[u] = infinity
        pred[u] = null
    }
    color[s] = gray // initialize source s
    d[s] = 0
    Q = {s} // put s in the queue
    while (Q is nonempty) {
        u = Q.Dequeue() // u is the next to visit
        for each v in Adj[u] {
            if (color[v] == white) { // if neighbor v undiscovered
                color[v] = gray // ...mark it discovered
                d[v] = d[u]+1 // ...set its distance
                pred[v] = u // ...and its predecessor
                Q.Enqueue(v) // ...put it in the queue
            }
        }
        color[u] = black // we are done with u
    }
}

Fig. 1: Breadth-first search: Example.
as desired.

**Analysis:** The running time analysis of BFS is similar to the running time analysis of many graph traversal algorithms. As done in CLR \( V = |V| \) and \( E = |E| \). Observe that the initialization portion requires \( \Theta(V) \) time. The real meat is in the traversal loop. Since we never visit a vertex twice, the number of times we go through the while loop is at most \( V \) (exactly \( V \) assuming each vertex is reachable from the source). The number of iterations through the inner for loop is proportional to \( \text{deg}(u) + 1 \). (The +1 is because even if \( \text{deg}(u) = 0 \), we need to spend a constant amount of time to set up the loop.) Summing up over all vertices we have the running time

\[
T(V) = V + \sum_{u \in V} (\text{deg}(u) + 1) = V + \sum_{u \in V} \text{deg}(u) + V = 2V + 2E \in \Theta(V + E).
\]

The analysis is essentially the same for directed graphs.

**Depth-First Search**

**Depth-First Search:** The next traversal algorithm that we will study is called depth-first search, and it has the nice property that nontree edges have a good deal of mathematical structure.

Consider the problem of searching a castle for treasure. To solve it you might use the following strategy. As you enter a room of the castle, paint some graffiti on the wall to remind yourself that you were already there. Successively travel from room to room as long as you come to a place you haven’t already been. When you return to the same room, try a different door leaving the room (assuming it goes somewhere you haven’t already been). When all doors have been tried in a given room, then backtrack.

Notice that this algorithm is described recursively. In particular, when you enter a new room, you are beginning a new search. This is the general idea behind depth-first search.

**Depth-First Search Algorithm:** We assume we are given an directed graph \( G = (V, E) \). The same algorithm works for undirected graphs (but the resulting structure imposed on the graph is different).

We use four auxiliary arrays. As before we maintain a color for each vertex: white means undiscovered, gray means discovered but not finished processing, and black means finished. As before we also store predecessor pointers, pointing back to the vertex that discovered a given vertex. We will also associate two numbers with each vertex. These are time stamps. When we first discover a vertex \( u \) store a counter in \( d[u] \) and when we are finished processing a vertex we store a counter in \( f[u] \). The purpose of the time stamps will be explained later. (Note: Do not confuse the discovery time \( d[u] \) with the distance \( d[u] \) from BFS.) The algorithm is shown in code block below, and illustrated in Fig. 2. As with BFS, DFS induces a tree structure. We will discuss this tree structure further below.

**Analysis:** The running time of DFS is \( \Theta(V + E) \). This is somewhat harder to see than the BFS analysis, because the recursive nature of the algorithm obscures things. Normally, recurrences are good ways to analyze recursively defined algorithms, but it is not true here, because there is no good notion of “size” that we can attach to each recursive call.

First observe that if we ignore the time spent in the recursive calls, the main DFS procedure runs in \( O(V) \) time. Observe that each vertex is visited exactly once in the search, and hence the call \( \text{DFSVisit()} \) is made exactly once for each vertex. We can just analyze each one individually and add up their running times. Ignoring the time spent in the recursive calls, we can see that each vertex \( u \) can be processed in \( O(1 + \text{outdeg}(u)) \) time. Thus the total time used in the procedure is

\[
T(V) = V + \sum_{u \in V} (\text{outdeg}(u) + 1) = V + \sum_{u \in V} \text{outdeg}(u) + V = 2V + E \in \Theta(V + E).
\]

A similar analysis holds if we consider DFS for undirected graphs.
Depth-First Search

DFS(G) { // main program
  for each u in V { // initialization
    color[u] = white;
    pred[u] = null;
  }
  time = 0;
  for each u in V
    if (color[u] == white) // found an undiscovered vertex
      DFSVisit(u); // start a new search here
}

DFSVisit(u) { // start a search at u
  color[u] = gray; // mark u visited
  d[u] = ++time;
  for each v in Adj(u) do
    if (color[v] == white) { // if neighbor v undiscovered
      pred[v] = u; // ...set predecessor pointer
      DFSVisit(v); // ...visit v
    }
  color[u] = black; // we’re done with u
  f[u] = ++time;
}

Fig. 2: Depth-First search tree.
Tree structure: DFS naturally imposes a tree structure (actually a collection of trees, or a forest) on the structure of the graph. This is just the recursion tree, where the edge \((u, v)\) arises when processing vertex \(u\) we call \(\text{DFSVisit}(v)\) for some neighbor \(v\). For directed graphs the other edges of the graph can be classified as follows:

Back edges: \((u, v)\) where \(v\) is a (not necessarily proper) ancestor of \(u\) in the tree. (Thus, a self-loop is considered to be a back edge).

Forward edges: \((u, v)\) where \(v\) is a proper descendant of \(u\) in the tree.

Cross edges: \((u, v)\) where \(u\) and \(v\) are not ancestors or descendents of one another (in fact, the edge may go between different trees of the forest).

It is not difficult to classify the edges of a DFS tree by analyzing the values of colors of the vertices and/or considering the time stamps. This is left as an exercise.

With undirected graphs, there are some important differences in the structure of the DFS tree. First, there is really no distinction between forward and back edges. So, by convention, they are all called back edges by convention. Furthermore, it can be shown that there can be no cross edges. (Can you see why not?)

Time-stamp structure: There is also a nice structure to the time stamps. In CLR this is referred to as the parenthesis structure. In particular, the following are easy to observe.

**Lemma:** (Parenthesis Lemma) Given a digraph \(G = (V, E)\), and any DFS tree for \(G\) and any two vertices \(u, v \in V\).
- \(u\) is a descendent of \(v\) if and only if \([d[u], f[u]] \subseteq [d[v], f[v]]\).
- \(u\) is an ancestor of \(v\) if and only if \([d[u], f[u]] \supseteq [d[v], f[v]]\).
- \(u\) is unrelated to \(v\) if and only if \([d[u], f[u]]\) and \([d[v], f[v]]\) are disjoint.

**Cycles:** The time stamps given by DFS allow us to determine a number of things about a graph or digraph. For example, suppose you are given a graph or digraph. You run DFS. You can determine whether the graph contains any cycles very easily. We do this with the help of the following two lemmas.

**Lemma:** Given a digraph \(G = (V, E)\), consider any DFS forest of \(G\), and consider any edge \((u, v) \in E\). If this edge is a tree, forward, or cross edge, then \(f[u] > f[v]\). If the edge is a back edge then \(f[u] \leq f[v]\).

**Proof:** For tree, forward, and back edges, the proof follows directly from the parenthesis lemma. (E.g. for a forward edge \((u, v)\), \(v\) is a descendent of \(u\), and so \(v\)'s start-finish interval is contained within \(u\)'s, implying that \(v\) has an earlier finish time.) For a cross edge \((u, v)\) we know that the two time intervals are disjoint. When we were processing \(u\), \(v\) was not white (otherwise \((u, v)\) would be a tree edge), implying that \(v\) was started before \(u\). Because the intervals are disjoint, \(v\) must have also finished before \(u\).
Lemma: Consider a digraph \( G = (V, E) \) and any DFS forest for \( G \). \( G \) has a cycle if and only the DFS forest has a back edge.

Proof: (\( \Leftarrow \)) If there is a back edge \((u, v)\), then \( v \) is an ancestor of \( u \), and by following tree edges from \( v \) to \( u \) we get a cycle.

(\( \Rightarrow \)) We show the contrapositive. Suppose there are no back edges. By the lemma above, each of the remaining types of edges, tree, forward, and cross all have the property that they go from vertices with higher finishing time to vertices with lower finishing time. Thus along any path, finish times decrease monotonically, implying there can be no cycle.

Beware: No back edges means no cycles. But you should not infer that there is some simple relationship between the number of back edges and the number of cycles. For example, a DFS tree may only have a single back edge, and there may anywhere from one up to an exponential number of simple cycles in the graph.

A similar theorem applies to undirected graphs, and is not hard to prove.

**Topological Sort and Strong Components**

**Directed Acyclic Graph:** A directed acyclic graph is often called a DAG for short DAG’s arise in many applications where there are precedence or ordering constraints. For example, if there are a series of tasks to be performed, and certain tasks must precede other tasks (e.g. in construction you have to build the first floor before you build the second floor, but you can do the electrical wiring while you install the windows). In general a precedence constraint graph is a DAG in which vertices are tasks and the edge \((u, v)\) means that task \( u \) must be completed before task \( v \) begins.

A topological sort of a DAG is a linear ordering of the vertices of the DAG such that for each edge \((u, v)\), \( u \) appears before \( v \) in the ordering. Note that in general, there may be many legal topological orders for a given DAG.

To compute a topological ordering is actually very easy, given DFS. By the previous lemma, for every edge \((u, v)\) in a DAG, the finish time of \( u \) is greater than the finish time of \( v \). Thus, it suffices to output the vertices in reverse order of finishing time. To do this we run a (stripped down) DFS, and when each vertex is finished we add it to the front of a linked list. The final linked list order will be the final topological order. This is given below.

```plaintext
TopSort(G) {
    for each (u in V) color[u] = white; // initialize
    L = new linked_list; // L is an empty linked list
    for each (u in V)
        if (color[u] == white) TopVisit(u);
    return L; // L gives final order
}

TopVisit(u) {
    // start a search at u
    color[u] = gray; // mark u visited
    for each (v in Adj(u))
        if (color[v] == white) TopVisit(v);
    Append u to the front of L; // on finishing u add to list
}
```

This is typical example of DFS is used in applications. Observe that the structure is essentially the same as the basic DFS procedure, but we only include the elements of DFS that are needed for this application.
As an example we consider the DAG presented in CLRS for Professor Bumstead’s order of dressing. Bumstead lists the precedences in the order in which he puts on his clothes in the morning. We do our depth-first search in a different order from the one given in CLRS, and so we get a different final ordering. However both orderings are legitimate, given the precedence constraints. As with depth-first search, the running time of topological sort is $\Theta(V + E)$.

![Topological sort diagram]

**Final order: socks, shirt, tie, shorts, pants, shoes, belt, jacket**

**Fig. 4: Topological sort.**

**Strong Components:** Next we consider a very important connectivity problem with digraphs. When digraphs are used in communication and transportation networks, people want to know that there networks are complete in the sense that from any location it is possible to reach any other location in the digraph. A digraph is strongly connected if for every pair of vertices, $u, v \in V$, $u$ can reach $v$ and vice versa.

We would like to write an algorithm that determines whether a digraph is strongly connected. In fact we will solve a generalization of this problem, of computing the strongly connected components (or strong components for short) of a digraph. In particular, we partition the vertices of the digraph into subsets such that the induced subgraph of each subset is strongly connected. (These subsets should be as large as possible, and still have this property.) More formally, we say that two vertices $u$ and $v$ are mutually reachable if $u$ and reach $v$ and vice versa. It is easy to see that mutual reachability is an equivalence relation. This equivalence relation partitions the vertices into equivalence classes of mutually reachable vertices, and these are the strong components.

Observe that if we merge the vertices in each strong component into a single super vertex, and joint two supervertices $(A, B)$ if and only if there are vertices $u \in A$ and $v \in B$ such that $(u, v) \in E$, then the resulting digraph, called the component digraph, is necessarily acyclic. (Can you see why?) Thus, we may be accurately refer to it as the component DAG.

![Component DAG diagram]

**Fig. 5: Strong Components.**

The algorithm that we will present is an algorithm designer’s “dream” (and an algorithm student’s nightmare).
It is amazingly simple and efficient, but it is so clever that it is very difficult to even see how it works. We will give some of the intuition that leads to the algorithm, but will not prove the algorithm’s correctness formally. See CLRS for a formal proof.

**Strong Components and DFS:** By way of motivation, consider the DFS of the digraph shown in the following figure (left). By definition of DFS, when you enter a strong component, every vertex in the component is reachable, so the DFS does not terminate until all the vertices in the component have been visited. Thus all the vertices in a strong component must appear in the same tree of the DFS forest. Observe that in the figure each strong component is just a subtree of the DFS forest. Is it always true for any DFS? Unfortunately the answer is no. In general, many strong components may appear in the same DFS tree. (See the DFS on the right for a counterexample.) Does there always exist a way to order the DFS such that it is true? Fortunately, the answer is yes.

Suppose that you knew the component DAG in advance. (This is ridiculous, because you would need to know the strong components, and that is the problem we are trying to solve. But humor me for a moment.) Further suppose that you computed a reversed topological order on the component digraph. That is, \((u, v)\) is an edge in the component digraph, then \(v\) comes before \(u\) in this reversed order (not after as it would in a normal topological ordering). Now, run DFS, but every time you need a new vertex to start the search from, select the next available vertex according to this reverse topological order of the component digraph.

Here is an informal justification. Clearly once the DFS starts within a given strong component, it must visit every vertex within the component (and possibly some others) before finishing. If we do not start in reverse topological, then the search may “leak out” into other strong components, and put them in the same DFS tree. For example, in the figure below right, when the search is started at vertex \(a\), not only does it visit its component with \(b\) and \(c\), but the it also visits the other components as well. However, by visiting components in reverse topological order of the component tree, each search cannot “leak out” into other components, because other components would have already have been visited earlier in the search.

![Fig. 6: Two depth-first searches.](image)

This leaves us with the intuition that if we could somehow order the DFS, so that it hits the strong components according to a reverse topological order, then we would have an easy algorithm for computing strong components. However, we do not know what the component DAG looks like. (After all, we are trying to solve the strong component problem in the first place). The “trick” behind the strong component algorithm is that we can find an ordering of the vertices that has essentially the necessary property, without actually computing the component DAG.
The Plumber's Algorithm: I call this algorithm the plumber's algorithm (because it avoids leaks). Unfortunately it is quite difficult to understand why this algorithm works. I will present the algorithm, and refer you to CLRS for the complete proof. First recall that $G^R$ (what CLRS calls $G^T$) is the digraph with the same vertex set as $G$ but in which all edges have been reversed in direction. Given an adjacency list for $G$, it is possible to compute $G^R$ in $\Theta(V + E)$ time. (I'll leave this as an exercise.)

Observe that the strongly connected components are not affected by reversing all the digraph’s edges. If $u$ and $v$ are mutually reachable in $G$, then certainly this is still true in $G^R$. All that changes is that the component DAG is completely reversed. The ordering trick is to order the vertices of $G$ according to their finish times in a DFS. Then visit the nodes of $G^R$ in decreasing order of finish times. All the steps of the algorithm are quite easy to implement, and all operate in $\Theta(V + E)$ time. Here is the algorithm.

```c
StrongComp(G) {
    Run DFS(G), computing finish times f[u] for each vertex u;
    Compute R = Reverse(G), reversing all edges of G;
    Sort the vertices of R (by CountingSort) in decreasing order of f[u];
    Run DFS(R) using this order;
    Each DFS tree is a strong component;
}
```

Correctness: Why visit vertices in decreasing order of finish times? Why use the reversal digraph? It is difficult to justify these elements formally. Here is some intuition, though. Recall that the main intent is to visit the strong components in a reverse topological order. The question is how to order the vertices so that this is true. Recall from the topological sorting algorithm, that in a DAG, finish times occur in reverse topological order (i.e., the first vertex in the topological order is the one with the highest finish time). So, if we wanted to visit the components in reverse topological order, this suggests that we should visit the vertices in increasing order of finish time, starting with the lowest finishing time. This is a good starting idea, but it turns out that it doesn’t work. The reason is that there are many vertices in each strong component, and they all have different finish times. For example, in the figure above observe that in the first DFS (on the left) the lowest finish time (of 4) is achieved by vertex e, and its strong component is first, not last, in topological order.

It is tempting to give up in frustration at this point. But there is something to notice about the finish times. If we consider the maximum finish time in each component, then these are related to the topological order of the component DAG. In particular, given any strong component $C$, define $f(C)$ to be the maximum finish time
among all vertices in this component.

\[ f(C) = \max_{u \in C} f[u]. \]

**Lemma:** Consider a digraph \( G = (V, E) \) and let \( C \) and \( C' \) be two distinct strong components. If there is an \((u, v)\) of \( G \) such that \( u \in C \) and \( v \in C' \), then \( f(C) > f(C') \).

See the book for a complete proof. Here is a quick sketch. If the DFS visits \( C \) first, then the DFS will leak into \( C' \) (along edge \((u, v)\) or some other edge), and then will visit everything in \( C' \) before finally returning to \( C \). Thus, some vertex of \( C \) will finish later than every vertex of \( C' \). On the other hand, suppose that \( C' \) is visited first. Because there is an edge from \( C \) to \( C' \), we know from the definition of the component DAG that there cannot be a path from \( C' \) to \( C \). So \( C' \) will completely finish before we even start \( C \). Thus all the finish times of \( C \) will be larger than the finish times of \( C' \).

For example, in the previous figure, the maximum finish times for each component are 18 (for \( \{a, b, c\} \)), 17 (for \( \{d, e\} \)), and 12 (for \( \{f, g, h, i\} \)). The order \( 18, 17, 12 \) is a valid topological order for the component digraph.

This is a big help. It tells us that if we run DFS and compute finish times, and then run a new DFS in decreasing order of finish times, we will visit the components in topological order. The problem is that this is not what we wanted. We wanted a reverse topological order for the component DAG. So, the final trick is to reverse the digraph, by forming \( G^R \). This does not change the strong components, but it reverses the edges of the component graph, and so reverses the topological order, which is exactly what we wanted. In conclusion we have:

**Theorem:** Consider a digraph \( G \) on which DFS has been run. Sort the vertices by decreasing order of finish time. Then a DFS of the reversed digraph \( G^R \), visits the strong components according to a reversed topological order of the component DAG of \( G^R \).