2 Basic Graph Algorithms

A graph $G = (V, E)$ consists of a set of vertices and edges. If the graph is directed, then each edge is of the form (u, v); otherwise, an edge between u and v is technically $\{u, v\}$, but people sometimes still use (u, v), maybe because it looks a bit nicer.

2.1 Breadth-First Search

Problem statement. Let $G = (V, E)$ be a directed graph and s be a vertex of G. Our goal is to find the shortest path from s to all other vertices in G.

Remark. We'll cheat a bit: instead of finding the shortest path from s to all other vertices, we'll find the *length* of the shortest path (i.e., distance) from s to all other vertices. Once we understand how to find distances, recovering the paths themselves is fairly straightforward. This concept appears throughout these notes.

Algorithm. We'll populate an array d, where $d[u]$ will eventually contain the distance from s to u, starting with $d[s] = 0$. Then, for each out-neighbor u of s, set $d[u] = 1$. Then, for each out-neighbor v of those out-neighbors (that aren't out-neighbors of s), set $d[v] = 2$. In general, the algorithm proceeds out from s in these layers using a queue to properly order the vertices. Since it finishes each layer before moving onto the next, it is known as *breadth*-first search (BFS).

Algorithm 6 Breadth-First Search (BFS)

Input: Graph $G = (V, E)$, $s \in V$

```
d[s] ← 0, d[u] ← ∞ for each u ∈ V \ {s}
Create a queue Q = (s)
while Q is not empty do
    Dequeue a vertex u from Q
    for each out-neighbor v of u do
        if d[v] = ∞ then
            d[v] ← d[u] + 1, add v to Q
return d
```

Theorem 2.1. *At the end of BFS, for every vertex v, $d[v]$ is the distance from the starting vertex s to v.*

Proof. Let k denote the distance between s and v; we proceed by induction on k. If $k = 0$, then $s = v$ and $d[s] = 0$, so we are done. Otherwise, by induction, we can assume $d[u] = k-1$ for every vertex u that has distance $k-1$ from s. At some point during the algorithm, these vertices are exactly the ones in Q. At least one such vertex has v as an out-neighbor; let u denote the first such vertex removed from Q. When u gets removed from Q, v still has not been added, so the algorithm sets $d[v] = d[u] + 1 = k - 1 + 1 = k$, as desired. □

Theorem 2.2. *BFS runs in $O(m + n)$ (i.e., linear) time.*

Proof. The initialization of d and Q take $O(n)$ time. Every vertex gets added to Q at most once, so we execute the main loop at most n times. It might seem like processing a vertex u involves $O(n)$ operations (since u could have roughly n neighbors), making the total running time $O(n^2)$. This is true, but we can analyze this more carefully: processing u only requires $\deg(u)$ time, and $\sum_{u \in V} \deg(u) = 2m$, so the overall running time is $O(n+m)$. This is $O(m)$ if the graph is connected, since in that case, $n \leq m + 1$.

Also, we generally assume that the graph is given in adjacency list format, so for a graph with n vertices and m edges, the size of the input is roughly $m + n$. Any algorithm whose running time is proportional to the length of the input is a linear-time algorithm. □

To find the shortest path from s to all other vertices, we can maintain a "parent array" p as we run BFS. In particular, whenever we set $d[v]$ to be $d[u] + 1$, we also set $p[v] = u$. This indicates that on the shortest path from s to v, u is the vertex before v. By tracing these pointers p backwards, we can recover the shortest paths.

2.2 Depth-First Search

Problem statement. Let $G = (V, E)$ be a directed graph. Our goal is return a directed cycle in G or report that none exists.

Remark. Lets start by looking at how Depth-First Search (DFS) labels every vertex u with two positive integers: $\mathsf{pre}[u]$ and $\mathsf{post}[u]$. These values allow us to solve multiple problems, including the cycle detection problem stated above.

DFS uses an "explore" procedure (Algorithm 7), which recursively traverses some path until it gets "stuck" because it has already visited all neighbors of the current vertex. It then traverses another path until it gets stuck again. This process repeats until all vertices have been visited (by possibly restarting the process from another vertex). The algorithm gets its name because it goes as "deep" as possible (along some path) before turning around.

Algorithm 7 Explore

Input: Graph $G = (V, E)$, $u \in V$

1: Mark u as visited, $\mathsf{pre}[u] \leftarrow t$, $t \leftarrow t + 1$
2: **for** each out-neighbor v of u **do**
3: **if** v has not been marked as visited **then**
4: $p[v] \leftarrow u$, Explore(G, v)
5: $\mathsf{post}[u] \leftarrow t, t \leftarrow t + 1$

DFS also maintains a global "clock" variable t, which is incremented whenever a vertex receives its pre or post value. Every vertex u receives its pre value when the exploration of u begins, and it receives its post value when the exploration ends. Also, when the exploration traverses an edge (u, v) we set $p[v] = u$, indicating that vertex v "came from" u, so u is the "parent" of v. If s is the first vertex explored then $\mathsf{pre}[s] = 1$, and if v is the last vertex to finish exploring then $\mathsf{post}[v] = 2n$. Every vertex gets explored at most once, and every edge participates in Line 2 at most once, so the running time of DFS is $O(m + n)$.

Algorithm 8 Depth-First Search (DFS)

Input: A graph $G = (V, E)$

```
Initialize all vertices as unmarked, t ← 1
for each vertex u ∈ V do
    if u has not been marked as visited then
        p[u] ← u, Explore(G, u)
return pre, post, p
```

Definition 2.3. *An edge (u, v) in G is a* back edge *if* $\mathsf{pre}[v] < \mathsf{pre}[u] < \mathsf{post}[u] < \mathsf{post}[v]$.

Theorem 2.4. *A directed graph has a cycle if and only if DFS creates a back edge.*

Proof. If (u, v) is a back edge, then while v was being explored, the algorithm discovered u. Since the algorithm only follows edges in the graph, this means there is a path from v to u in the graph. This path, combined with the edge (u, v), is a cycle.

Conversely, suppose the graph has a cycle C. Let v denote the first vertex of C explored by DFS, and let u denote the vertex preceding v in C. Then while v is being explored, u will be discovered, which will lead to (u, v) being a back edge. □

The proof of Theorem 2.4 gives our $O(m + n)$-time algorithm for the cycle detection problem. If DFS does not create any back edges, then the graph is acyclic. Otherwise, if (u, v) is a back edge, then there must be a path from v to u traversed by DFS, which together with (u, v) forms a cycle.

Classifying edges. After running DFS on a directed graph G, we can naturally partition the edges as follows:

- An edge (u, v) is a *tree edge* if it was directly traversed by DFS (i.e., $p[v] = u$). Let T denote the subgraph of G containing its vertices and tree edges.
- An edge (u, v) is a *forward edge* if there is a path from u to v in T.
- An edge (u, v) is a *back edge* if there is a path containing from v to u in T. (This is equivalent to the definition given above.)
- An edge (u, v) is a *cross edge* if there is no (directed) path from u to v or from v to u in T (i.e., u and v do not have an ancestor/descendant relationship).

2.3 Minimum Spanning Tree

Problem statement. Let $G = (V, E)$ be an undirected graph where each edge e has weight $w_e > 0$. Our goal is to compute $F \subseteq E$ such that (V, F) is connected and the total weight of edges in F is minimized. Any subset of E satisfying these two properties is a *minimum spanning tree* (MST) of G.

Remark. For simplicity, unless stated otherwise, we assume that all of the edge weights in G are distinct. It follows (by an exchange argument similar to the proofs in this section) that G has exactly one MST. Thus, we can refer to *the* (rather than *an*) MST of G.

Algorithm. Instead of giving a single algorithm, we first prove two facts about MSTs known as the cut property and the cycle property. Using these two facts, we can prove the correctness of three natural greedy algorithms for the MST problem.

Lemma 2.5 (Cut Property)**.** *For any subset S of V (S is called a* cut*), the lightest edge crossing S (i.e., having exactly one endpoint in S) is in the MST of G.*

Proof. Let T denote the MST of G, and for contradiction, suppose there exists a cut S such that the lightest edge e that crosses S is not in T. Notice that $T \cup \{e\}$ must contain a cycle C that includes e. Since e crosses S, some other edge $f \in C$ must cross S as well, and since e is the lightest edge crossing S, we must have $w(e) < w(f)$. Thus, adding e and removing f from T yields a spanning tree with less total weight than T, and this contradicts our assumption that T is the MST. □

Lemma 2.6 (Cycle Property)**.** *For any cycle C in G, the heaviest edge in C is not in the MST of G.*

Proof. The proof is similar to that of the cut property. Let T denote the MST of G, and for contradiction, suppose there exists a cycle C whose heaviest edge f belongs to T. Removing f from T disconnects T into two trees, and also defines a cut of G. Some other edge e of C must cross this cut, and since f is the heaviest in C, we have $w(e) < w(f)$. Thus, adding e and removing f from T yields a spanning tree with less total weight than T, and this contradicts our assumption that T is the MST. □

We are now ready to state three algorithms for the MST problem:

1. **Kruskal's algorithm.** Sort the edges in order of increasing weight. Then iterate through the edges in this order, adding each edge e to F (initially empty) if $(V, F \cup \{e\})$ remains acyclic.

2. **Prim's algorithm.** Let s be an arbitrary vertex and initialize a set $S = \{s\}$. Repeat the following $n - 1$ times: add the lightest edge e crossing S to the solution (initially empty), and add the endpoint of e not in S to S.

3. **Reverse-Delete.** Sort the edges in order of decreasing weight. Then iterate through the edges in this order, removing each edge e from F (initially all of E) if $(V, F \setminus \{e\})$ remains connected. (This a "backward" version of Kruskal's algorithm.)

Given the cut and cycle properties, the proofs of correctness for all three algorithms are fairly short and straightforward. Below, we prove the correctness of Kruskal's algorithm.

Theorem 2.7. *At the end of Kruskal's algorithm, F is a minimum spanning tree.*

Proof. Consider any edge $e = \{u, v\}$ added to F, and let S denote the set of vertices connected to u via F immediately before e was added. Notice that $u \in S$ and $v \in V \setminus S$. Furthermore, no edge crossing S has been considered by the algorithm yet, because it would have been added, which would have caused both of its endpoints to be in S. Thus, e is the lightest edge crossing S, so by the cut property, e is in the MST.

Now we know F is a subset of the MST; we still need to show that F is spanning. For contradiction, suppose that there exists a cut $S \subset V$ such that no edge crossing S is in F. But the algorithm would have added the lightest edge crossing S, so we are done. □

For each of the three algorithms, there is a relatively straightforward implementation that runs in $O(m^2)$ time; we leave the details as an exercise.

2.4 Exercises

2.1. Let G be an undirected graph on n vertices, where $n \geq 2$. Is it possible, depending on the starting vertex and tiebreaking rule, that at some point during BFS, all n vertices are simultaneously in the queue?

2.2. Recall that the running time of BFS is $O(m+n)$, where m and n are the number of edges and vertices, respectively, in the input graph. In a complete graph, $m = n(n-1)/2$, which makes the running time $\Omega(n^2)$. In light of this case, why do we still say that BFS runs always runs in linear time?

2.3. Let G be a directed path graph on n vertices. If we run BFS on G starting at the first vertex, what is the d value of each vertex? After we run DFS on G starting at the first vertex, what are the pre and post values of each vertex?

2.4. Let T be an MST of a graph G. Suppose we increase the weight of every edge in G by 1. Prove that T is still an MST of G, or give a counterexample.

2.5. Suppose every edge in an undirected graph G has the same weight w. Give a linear-time algorithm that returns an MST of G.

2.6. Give an $O(n)$-time algorithm that finds a cycle in an undirected graph G, or reports that none exists.

2.7. Show that Kruskal's, Prim's, and the Reverse-Delete algorithm can be implemented to run in $O(m^2)$ time.

2.8. Give the analogous versions of Definition 2.3 for tree, forward, and cross edges. Note that the definitions might overlap.

2.9. Suppose we run DFS on an undirected graph. How does the edge classification system (tree/forward/back/cross) change, if at all?

2.10. Give an undirected graph G with distinct edge weights such that Kruskal's and Prim's algorithms, when run on G, return their solutions in different orders.

2.11. Let $G = (V, E)$ be an undirected graph with n vertices. Show that if G satisfies any two of the following properties, then it also satisfies the third: (1) G is connected, (2) G is acyclic, (3) G has $n-1$ edges.

2.12. Prove that an undirected graph is a tree if and only if there exists a unique path between each pair of vertices.

2.13. Prove that a graph has exactly one minimum spanning tree if all edge weights are unique, and show that the converse is not true.

2.14. Consider the MST problem on a graph containing n vertices.

(a) Prove the correctness of Prim's algorithm, and show that it can be implemented to run in $O(n^2)$ time.

(b) Prove the correctness of the Reverse-Delete algorithm.

2.15. Let G be a directed graph and s be a starting vertex. Give a linear-time algorithm that returns the number of shortest paths between s and all other vertices.

2.16. An undirected graph $G = (V, E)$ is *bipartite* if there exists a partition of V into two nonempty groups such that no edge of E has one endpoint in each group.

(a) Prove that G is bipartite if and only if G contains no odd cycle.

(b) Give a linear-time algorithm to determine if an input graph G is bipartite. If so, the algorithm should return a valid partition of V (i.e., the algorithm should label each vertex using one of two possible labels). If not, the algorithm should return an odd cycle in G.

2.17. Let G be a directed acyclic graph (DAG). Use DFS to give a linear-time algorithm that orders the vertices such that for every edge (u, v), u appears before v in the ordering. (Such an ordering is known as a *topological sort* of G.)

2.18. Let G be a connected undirected graph. An edge e is a *bridge* if the graph $(V, E \setminus \{e\})$ is disconnected.

(a) Give an $O(m^2)$-time algorithm to determine if G has a bridge, and if so, the algorithm should return one.

(b) Assume G is bridgeless. Give a linear-time algorithm that orients every edge (i.e., turns it into a directed edge) in a way such that the resulting (directed) graph is strongly connected.

2.19. Let G be a graph containing a cycle. Our goal is to return the shortest cycle in G. Give an $O(m^2)$-time algorithm assuming G is undirected, and give an $O(mn)$-time algorithm assuming G is directed.

2.20. Let G be a directed graph. We say that vertices u and v are in the same *strongly connected component* (SCC) if they are strongly connected, i.e., there exists a path from u to v and a path from v to u. Consider the graph G^S whose vertices are the SCC's of G and there exists an edge from A to B if (and only if) there exists at least one edge (u, v) in G with $u \in A$ and $v \in B$.

(a) Show that G^S is a DAG.

(b) Give a linear-time algorithm to construct G^S using G.

2.21. Recall that the *distance* between two vertices is the length of the shortest path between them. The *diameter* of a graph is the maximum distance between any two vertices.

(a) Give an $O(mn)$-time algorithm that computes the diameter of a graph.

(b) Give a linear-time algorithm that computes the diameter of a tree.

2.22. Consider the following variant of the MST problem: our goal is to find a spanning tree such that the weight of the heaviest edge in the tree is minimized. In other words, instead of minimizing the sum, we are minimizing the maximum. Prove that a standard MST is also optimal for this new problem.

2.23. Let T be a spanning tree of a graph G, which has distinct edge weights. Prove that T is the MST of G if and only if T satisfies the following property: for every edge $e \in E \setminus T$, e is the heaviest edge in the cycle in $T \cup \{e\}$.