1 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 often use (u, v) because it looks a bit nicer (and we can always order the vertices arbitrarily). Unless otherwise stated, every graph we consider in this entire document is connected and has n vertices and m edges.

1.1 Breadth-First Search

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

Algorithm. We start at s, visit its neighbors, visit their neighbors (that haven't already been visited), and so on, until every vertex has been visited. Since this procedure finishes each "layer" (defined by distance from s) entirely before proceeding to the next, it is known as *breadth*-first search (BFS).

Algorithm 1 Breadth-First Search (BFS)

```
1: Initialize d[s] = 0 and d[u] = ∞ for all u ∈ V \ {s}.
2: Initialize a queue Q = (s).
3: while Q is not empty do
4:     Dequeue a vertex u from Q.
5:     for each neighbor v of u do                ▷ visit the unvisited neighbors of u
6:         if d[v] = ∞ then
7:             Set d[v] = d[u] + 1 and add v to Q.
8: return d
```

Theorem 1.1. *At the end of Algorithm 1, for every vertex v, $d[v]$ denotes the distance between the starting vertex s and 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 assume that $d[u] = k-1$ for every vertex u that has distance $k - 1$ from s. Since the algorithm processes vertices layer by layer, there is a point at which the vertices in Q are exactly the ones with distance $k - 1$ from s. At least one such vertex them is a neighbor of v, so we can let u denote the first such vertex dequeued. When u is removed from Q, v still has not been added, so the algorithm sets $d[v] = d[u] + 1 = k - 1 + 1 = k$, as desired. □

Notice that every vertex enters and leaves Q exactly once. Processing a vertex u requires $\deg(u)$ time and $\sum_{u \in V} \deg(u) = 2m$, so the overall running time is $O(m + n) = O(m)$.

1.2 Depth-First Search

Problem statement. Let $G = (V, E)$ be a directed graph. Our goal is to determine whether or not G contains a (directed) cycle; if so, we should return one such cycle.

Remark. Before solving the problem, we first describe the general Depth-First Search (DFS) algorithm. In particular, we show how it 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.

The DFS algorithm uses an "explore" procedure (Algorithm 2), which recursively traverses some path until it gets "stuck" because it has already visited all neighbors of the current vertex. It then simply follows 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 (away from the starting vertex) before turning around.

Algorithm 2 Explore (a vertex $x \in V$)

```
Mark x as visited, set pre[x] = t, and set t = t + 1.
for each edge (x, y) ∈ E do
    if y has not been marked as visited then
        Set p[y] = x and Explore (i.e., call this algorithm on) y.
Set post[x] = t and t = t + 1.
```

As shown above, DFS also maintains a global "clock" variable t, which is incremented whenever a vertex receives its pre or post value. Every vertex x receives its pre value when the exploration of x begins, and it receives its post value when the exploration ends. Also, when the exploration traverses an edge (x, y) we set $p[y] = x$, indicating that vertex y "came from" x, so x is the "parent" of y.

Algorithm 3 Depth-First Search (DFS)

```
Initialize all vertices as unmarked (i.e., unvisited) and t = 1.
for each vertex u ∈ V do
    if u has not been marked as visited then
        Set p[u] = u and call Algorithm 2 on u.
return pre, post, the parent pointers p
```

So the first vertex considered by DFS gets pre value 1, and the last vertex explored gets post value $2n$. Algorithm 2 is called on every vertex exactly once, and also examines every edge exactly once, so the overall running time of DFS is $O(m + n) = O(m)$. Before solving the cycle detection problem, we present the following definition.

Definition 1.2. *An edge (u, v) in G is a* back edge *if $\mathsf{pre}[v] < \mathsf{pre}[u] < \mathsf{post}[u] < \mathsf{post}[v]$.*

After running DFS, we have the pre and post values for all vertices, so we can easily find a back edge in linear time. Furthermore, every vertex has a parent, and if we orient the explored edges from parent to child, then we see that DFS constructs a forest of rooted trees. (A new tree is created whenever Algorithm 3 explores a new vertex.)

Theorem 1.3. *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 1.3 gives our final 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, so adding (u, v) to this path forms a cycle.

1.3 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, we assume that all of the edge weights in G are distinct. It follows, by a fairly straightforward exchange argument (similar to the proofs in this section), that G has exactly one MST. Thus, wwe will generally 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 1.4 (Cut Property)**.** *Let S be any subset of V (also known as a* cut*), and let e denote the lightest edge crossing S (i.e., among the edges with exactly one endpoint in S, e has minimum weight). The MST of G contains this edge e.*

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 1.5 (Cycle Property)**.** *Let C be any cycle in G, and let f denote the heaviest edge in C. The MST of G does not contain this edge f.*

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. □

To summarize: the cut property says that the lightest edge crossing any cut is in the MST, and the cycle property says that the heaviest edge in any cycle is not in the MST. As mentioned earlier, using these two properties, we can prove the correctness of multiple algorithms for MST.

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 (and only 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: (greedily) add the node v to S that minimizes the "attachment cost" defined as $\min_{(u,v):u \in S} w(u, v)$.[1]

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 (and only if) $(V, F \setminus \{e\})$ remains connected. (This is sort of 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; we leave the other two proofs as exercises.

Theorem 1.6. *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 immediately before e was added. Notice that $u \in S$, $v \in V \setminus S$, and no edge crossing S has been considered by the algorithm yet (because it would have been added). 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; it suffices to prove 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. □

1.4 Exercises

1. Recall that DFS returns two values $\mathsf{pre}[u]$ and $\mathsf{post}[u]$ for every vertex u of a directed graph. Consider any edge $e = (u, v)$; we saw what it means for e to be a back edge. What do you think should be the definition of a *tree* edge, *forward* edge, and *cross* edge? How is the situation different for undirected graphs?

2. 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]

3. 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.

[1] As we will see in the following section, this algorithm is similar to Dijkstra's algorithm.

[2] A *linear-time* algorithm is one whose running time is $O(m+n)$. If the graph is connected, then $n-1 \leq m$, so this bound can be simplified to $O(m)$.

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

(b) Give a linear-time algorithm to determine if G is bipartite, and 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).

4. 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.)

5. 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.

6. 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. (Hint: First run DFS on the *reverse* graph of G to obtain post values. Then, use these post values while running DFS (or BFS) on G.)

7. 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. (Hint: One algorithm involves running BFS twice.)

8. Let G be a graph; assume that G contains 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.

9. 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.

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

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

12. Consider the MST problem from Sec. 1.3.

    (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.

13. Consider the MST problem, but suppose some of the edge weights are negative. Describe how to modify the graph such that an MST in the new graph is also an MST of the original graph.

14. Suppose all of the edges of an undirected graph G have weight $w > 0$. Give a linear-time algorithm that returns an MST of G.

15. 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 "normal" MST is also optimal for this new problem.

16. Let T be a spanning tree of a graph G (which has unique 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 (only) cycle of $T \cup \{e\}$.