Graph Algorithms

Spanning trees

Recall that a spanning tree T of a graph G is a subgraph $T \subset G$ which happens to be a tree containing all vertices of G. We showed earlier that any connected graph G contains a spanning tree.
A weighted graph is a graph where each edge is assigned a weight (thought of as cost, like money, time, etc.). More rigorously, it is a graph $G = (V, E)$ together with a weight function $w : E \to \mathbb{R}$. A minimum weight spanning tree is a spanning tree T such that for **any** other spanning tree T' the total weight of T is less than the total weight of T', i.e.

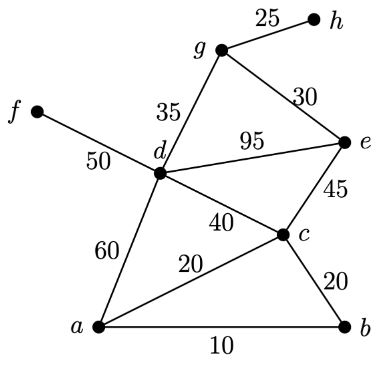
$$\sum_{e \in E_T} W(e) \leq \sum_{e \in E_{T'}} W(e)$$

Our goal is to find a minimum weight spanning tree for any connected graph. Here, we will describe three algorithms for finding a spanning tree of any connected graph. Note that a spanning tree is *not* unique. Indeed, even the same algorithm may come up with different trees based on arbitrary choices made along the way.

Algorithm 1 Kruskal's Algorithm

```
1: Mark an unmaked edge of smallest weight in the graph.
2: Select and mark a new edge among all unmarked edges of smallest weight that do not form a cycle
   with previously marked edges.
3: if the marked edges form a spanning tree, stop. Otherwise, go to step 2.
```

Example: Consider the following weighted graph.



One iteration of Kruskal's algorithm marks the following edges in order

1. ab
2. bc (note that ac is another valid choice leading to a different tree . . . try it!)
3. gh (ac has smaller weight but we can no longer select it as it will cause a cycle abca)
4. eg

5. dg
6. cd
7. df

We get a tree of total weight 210, which is the minimum weight possible for any spanning tree in this graph. How do we know?

Theorem 1. *Kruskal's algorithm produces a minimum weight spanning tree of any connected graph.*

Proof. Let G be a connected graph, and let T be the subgraph produced by running Kruskal's algorithm on G. We split the proof into multiple claims.

Claim 1: T is a tree.
By construction, we never mark an edge that creates a cycle so T is acyclic. Next, we argue that T is connected. If not, let $u, v \in V_T$ be two vertices with no path between them in T. Since G is connected, there is a path P in G from u to v. Thus, the path P contains edges $e_1, \ldots, e_k$ that are in G but not in T. Moreover, adding any edge e_i to T creates a cycle, since otherwise, it would have been marked and added to T by the algorithm. Let $C_1, \ldots, C_k$ be the cycles produced in T upon adding $e_1, \ldots, e_k$, respectively, and let P_i be the path in T we get upon deleting e_i from C_i. Now, replacing each e_i in P with a sub-path P_i, we get a trail in T connecting u to v, which is a contradiction. Thus, T is connected, and hence a tree. ■

Claim 2: T contains every vertex of G, i.e. T is a spanning tree.
Otherwise, assume v is a vertex of G but not in T. Since G is connected, there is a path P in G from v to some vertex, say u in T. Let us write $P = (v = v_0, v_1, v_2, \ldots, v_k = u)$. Since P contains both vertices in T and outside of T, there must be two consecutive vertices v_i and v_{i+1} in P with $v_i \notin V_T$ and $v_{i+1} \in V_T$. But then, Kruskal's algorithm could have safely marked the edge $v_i v_{i+1}$ since it cannot possibly form a cycle in T (as one vertex of the cycle would be not even in T), which is a contradiction. Thus, T spans all vertices of G. ■

Claim 3: T has minimum weight among all spanning trees of G.
Let e_i be the edge added by the algorithm at step i, so that the edges of T in the order they were added are $e_1, e_2, \ldots, e_{n-1}$, where n is the order of G. Suppose that T is not of minimum weight, and pick another spanning tree $\hat{T}$ of minimum weight, which agrees with T for the lonest initial segment, i.e. $\hat{T}$ is a minimum weight spanning tree with the largest k such that $e_1, \ldots, e_k \in E_{\hat{T}}$. Thus, no minimum-weight spanning tree contains all the edges $e_1, e_2, \ldots, e_{k+1}$. Since $\hat{T} \neq T$ and both have $n-1$ edges, we must have $k < n-1$, since otherwise, they contain exactly the same edges. Since $e_{k+1} \notin E_{\hat{T}}$, it must be the case that $\hat{T} + e_{k+1}$ contains a cycle C since otherwise, we get a tree with n vertices and n edges, which is not possible. Moreover, the cycle C must contain an edge not in T (otherwise, we get a cycle in T contradicting that it is a tree), say $\hat{e} \in E_{\hat{T}}$.
Next, let us consider a third tree T' where we delete the edge $\hat{e}$ from $\hat{T} + e_{k+1}$. The tree T' is a spanning tree since $\hat{e}$ was on a cycle. Moreover, the edge $\hat{e}$ does not form a cycle with the edges $e_1, e_2 \ldots, e_k$, since, if so, that cycle would be present in the tree $\hat{T}$. This means that in step $k+1$ of the algorithm, the edge $\hat{e}$ was available for marking but the algorithm chose e_{k+1} instead, so we conclude that $W(e_{k+1}) \leq W(\hat{e})$. It follows that the total weight of the spanning tree T' is less than or equal the total weight of the minimum-weight spanning tree $\hat{T}$, since they differ only by exchanging e_{k+1} and $\hat{e}$. So T' is also a minimum-weight spanning tree. But T' contains $e_1, e_2, \ldots, e_k, e_{k+1}$, contradicting the maximality of k in the choice of $\hat{T}$. Therefore, the tree T produced by Kruskal's algorithm must be a minimum weight spanning tree. □

Note that in intermediate steps of Kruskal's algorithm, the marked edges make a (possibly disconnected) forest. Prim's algorithm on the other hand is very similar, but insists on choosing edges that keep the graph connected at all times.

Algorithm 2 Prim's Algorithm

```
1: Mark an unmaked edge of smallest weight in the graph.
2: Select and mark a new edge among all unmarked edges of smallest weight that share exactly one
   vertex with currently marked edges and that do not form a cycle with marked edges.
3: if the marked edges form a spanning tree, stop. Otherwise, go to step 2.
```

Example: In the tree from the previous example, Prim's algorithm marks the following edges in order:

1. ab
2. ac (again, could also be bc producing a different tree)
3. cd
4. dg
5. gh
6. eg
7. df

Theorem 2. *Prim's algorithm always produces a minimum-weight spanning tree.*

Proof. Exercise! (Hint: modify the proof of correctness for Kruskal's algorithm.) □

The following algorithm was pointed out to me by a member of our class. I found it in the literature called "the reverse-delete algorithm" which is not as catchy, so I propose we call it Lukas's algorithm :)

Algorithm 3 Lukas's Algorithm

```
1: Start with the full graph $G$
2: Find a cycle of $G$ and delete an edge of maximum weight in that cycle.
3: if there are no cycles, left, stop. Otherwise, go to step 2.
```

Example: In the tree from the previous example, Lukas's algorithm deletes the following edges in order:

1. de
2. ad
3. ce
4. ac

Theorem 3. *Lukas's algorithm produces a minimum-weight spanning tree.*

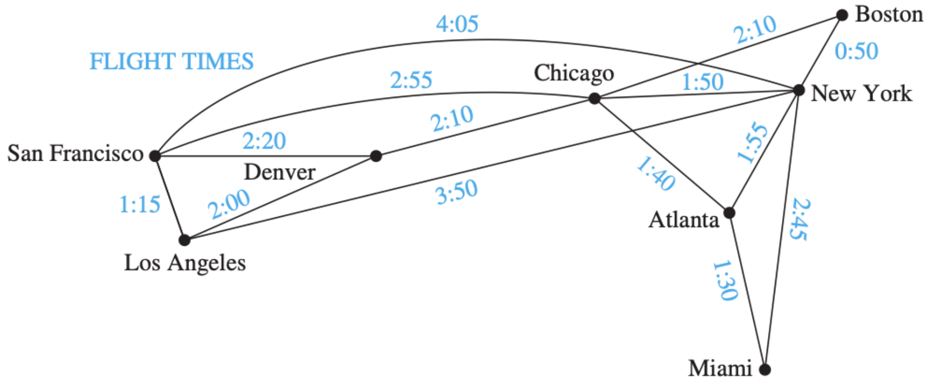
Proof. Exercise! (Hint: Split the proof into same claims as before. Proofs of claims 1 and 2 are now straightforward; now modify the proof of claim 3 in this "reversed" algorithm.) □

Shortest Path

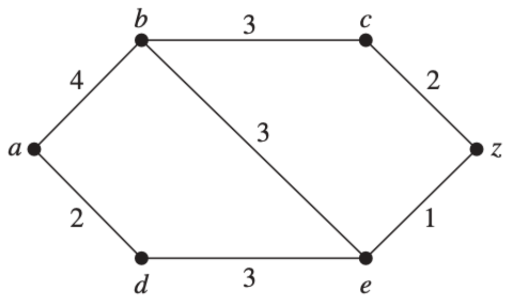
Definition: Given a weighted graph $G = (V, E)$ and two vertices $u, v \in V$, a *shortest path* from u to v is a path of minimum total path among all paths from u to v, i.e. it is a path P from u to v such that for any other such path P', we have

$$\sum_{e \in E_P} W(e) \leq \sum_{e \in E_{P'}} W(e),$$

where E_P and $E_{P'}$ denote the set of edges occuring in P and P', respectively.
The problem of finding a shortest path between two points is essential in many real-life applications, such as minimizing the time to communicate between two nodes in a computer network, or optimizing transportation routes. Our goal is to have an algorithm that can produce the shortest path between any two vertices of a weighted connected graph. For example, the following graph can be used to model fastest airline routes:



Let us consider an example first. Suppose we want the shortest path from a to z in the following graph:



We proceed to find shortest paths from a to each other vertex building up to z, and recording the most efficient distance from a so far as we go along the way.

- Start with a set $S = \{a\}$ and a labeling of $L(a) = 0$ (a has distance 0 to itself)..
- First consider the one-edge paths from a. These are ad of length 2 or ab of length 4. Thus, d is the closest vertex to a with distance 2 from a. So, we add d to our set to get $S = \{a, d\}$ and assign to d the label $L(d) = 2$.

- Next, consider edges with one endpoint from S and one endpoint outside of S. These are ab of length 4 and de of length 3. We combine these with previously examined shortest path to the endpoints in S to get two paths: ab of length 4 and ade of length 5. Thus, the second closest vertex to a is b with distance 4. So we add b to S to get $S = \{a, b, d\}$ and assign to b the label $L(b) = 4$.

- Now we find the third closest path to a. Again, consider edges with one endpoint in S and one endpoint not in S: these are bc, be, and de of lengths $3, 3$, and 3. Concatenating these with previous shortest path to the endpoints in S we get three paths: abc, abe, and ade of lengths $7, 7$, and 5, respectively. Thus, e is the third closest vertex to a with distance 5. We add e to S to get $S = \{a, b, d, e\}$ and label e by $L(e) = 5$.

- Continuing in this way, we find all edges with one endpoint in S and one endpoint not in S: these are bc, be, ez, which give us paths $abc, abe, adez$ of total length $7, 7, 6$, so z is the fourth closest vertex to a with distance 6 from a. We add z to S to get $S = \{a, b, d, e, z\}$ and label z by $L(z) = 6$.

- Since our goal was to find a shortest paths to z, we stop now that we have one.

Let us now write the general algorithm:

Algorithm 4 Dijkstra's Algorithm

```
1: Start with a graph $G$ with vertices $a = v_0, v_1, v_2, \ldots, v_n = z$ and length $W(v_i, v_j)$, where we take
   $W(v_i, v_j) = \infty$ if $v_iv_j \notin E_G$.
2: Initialize $L(v_0) = 0$ and $L(v_i) = \infty$ for all $i = 1, 2, \ldots, n$.
3: Initialize $S = \{a\}$
4: while $z \notin S$:
5:     Pick a vertex $u \notin S$ with $u \in N(S)$ with minimum $L(u)$.
6:     $S = S \cup u$
7:     for $v \notin S$ and $v \in N(u)$:
8:         if $L(u) + W(u, v) \leq L(v)$ then set $L(v) = L(u) + W(u, v)$.
9: return $L(z)$ = length of shortest path from $a$ to $z$.
```

Theorem 4. *Dijkstra's algorithm finds the shortest path between any two vertices in any connected weighted graph.*[1]

Proof. We will show that whenever a vertex v is added to S, its minimum distance from a is indeed $L(v)$. We proceed by (strong) induction on the size of S upon adding v.

Base case: If $|S| = 1$ then $v = a$ and $L(v) = 0$ and we are done.

Inductive step: assume the algorithm recorded for each vertex v previously added to S its shortest distance from a in the label $L(v)$, and suppose now that the algorithm adds a new vertex u to S, which was given the label $L(u)$.
We suppose towards a contradiction that $L(u)$ is *not* the shortest distance from a to u. Then, we have two cases: either the shortest path from a to u contains a vertex not in S or it does not.

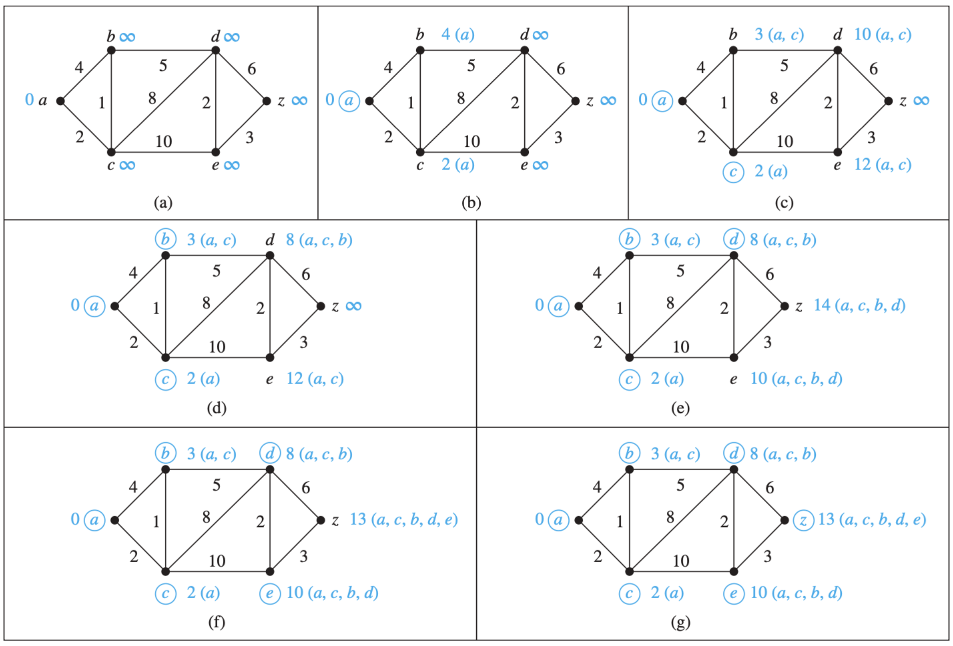
Case 1: The shortest path from a to u contains a vertex not in S. Let w be the *first* vertex not in S in the shortest path from a to u. Then, the shortest distance from a to u is $L(w)$+distance of shortest

[1]using $O(n^2)$ operations where n is the order of the graph.

path from w to u, which is at least a big as $L(w)$. Since u was picked by the algorithm to be added in S in the next step rather than w, it must be the case that $L(u) \leq L(w)$ since the algorithm picks the vertex with minimum label. But this is a contradiction since we just argued that the shortest path has length $L(w) +$ positive number $> L(u)$, and our assumption that this path has length less than $L(u)$. [2]

Case 2: The shortest path from a to u consists only of vertices in S. Let w be the vertex before u on the shortest path (i.e. second to last vertex). This means that $L(w) + W(w, v) < L(v)$. But this is a contradiction since upon adding w to S, it should have already set $L(v)$ to at most $L(w) + W(w, v)$. For all already visited vertices $v \in S$, the induction hypothesis already guarantees that $L(v)$ is the shortest distance and these values are unchanged upon adding a new vertex. This completes the induction step. $\square$

Example: Here is a step by step example of applying Dijkstra's algorithm to a graph to find the shortest path from a to z.



[2]Note: here it is essential to assume that the edges have positive weights, which is why we use the term length instead of weight. This assumption was not needed for the spanning tree algorithms discussed above.

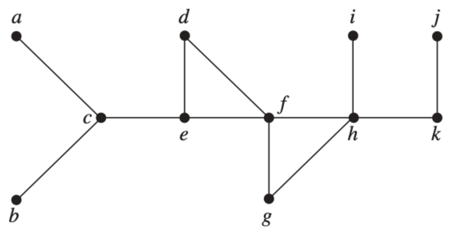
Graph Traversal

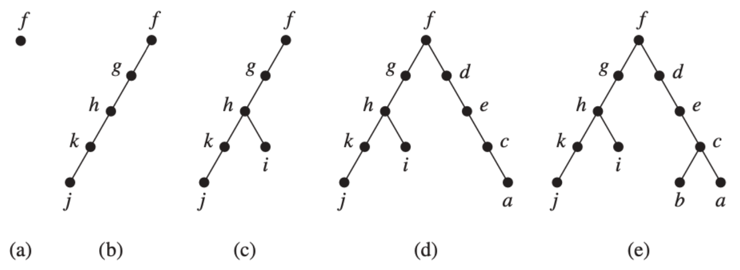
We now deal with unweighted graphs again. A graph traversal algorithm is a systematic method to visit every vertex of a graph. There are multiple such algorithms, each suitable to different applications. Here we will explore the two most commonly used graph traversal algorithms.

Depth-First Search (DFS)

The first algorithm we explore is the so-called "depth-first search". The philosophy of this algorithm is that we visit a vertex and keep going in one direction until we hit a dead end; then go back and explore a new direction starting the last turn we didn't take last time. You should imagine yourself as an adventurous and bold explorer going around exploring every path until its very end before giving up and trying a new path. This suggests an immediate application where we are trying to solve a maze by exploring where each turn leads us to the very end.

Example: Here is an example of a step-by-step application of a depth-first traversal of a graph starting from the vertex f (called the *root*).





Algorithm 5 Depth-First Search Algorithm (DFS)

```
Initialize $T$ = tree consisting only of the root vertex $v_0$.
procedure visit(vertex $v$ of $G$)
    for each vertex $w$ adjacent to $v$ and not yet in $T$
        Add the vertex $w$ and the edge $vw$ to $T$
        visit($w$) (note the recursion!)
visit($v_0$)
```

In the example above, the DFS visits vertices in the following order: $f, g, h, k, j, i, d, e, c, a, b$.

Note: The subgraph produced by DFS contains a unique path from the root to any given vertex, but that path need not be the shortest one; indeed, in the above example, the path in T from f to h has length 2, but the shortest such path has length 1 (here we are treating all edges to be of length 1 if you want to think of G as a weighted graph).

Theorem 5. *DFS produces a spanning tree of any connected graph starting from any root vertex. In particular, it traverses every vertex of the graph.*

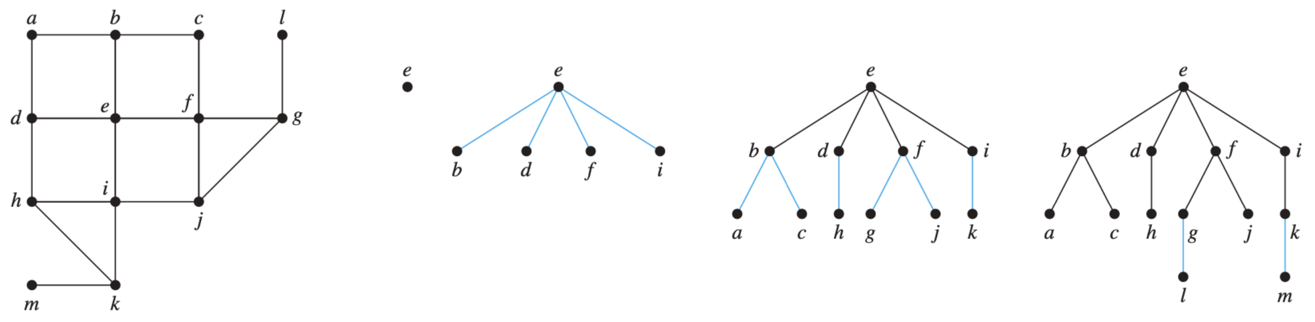
Proof. Let T be the subgraph obtained by applying DFS to the graph G with the root v_0. First, note that we only add edges starting from a vertex already in T, thus T is connected. Moreover, T starts with 1 vertex and 0 edges and adds one new edge for every new vertex; therefore, $|E_T| = |V_T| - 1$, so T is a tree.

To see that T contains every vertex, assume to the contrary that some vertex of v of G is not in T. If one of the neighbors u of v was previously visited, we would have visited v as well upon backtracking to u at some point (guaranteed by the condition of the for loop in step 3). So, none of the neighbors of v were visited, and none of the neighbors of the neighbors of v were visited, etc. etc. So, every vertex in the same connected component as v was not visited. But since G is connected, none of the vertices of G is in T, which contradicts that we started with a tree containing the root. □

Breadth-First Search (BFS)

Another philosophy of graph traversal is to visit *every* neighbor of the current neighbor before moving on the visit every neighbor of these neighbors and so on. This is called "breadth-first search". Picture a very hesitant and lazy explorer trying to find the treasure in a maze by staying as close as possible from their starting point; so they turn back whenever they see that the current path has more to it to explore other paths hoping they are shorter.

Example: Here is a BFS tree of the following graph starting from the root e.



Here the vertices are visited in the following order: $e, b, d, f, i, a, c, h, g, j, k, l, m$.
The above picture suggests the following intuition for BFS: imagine the edges of a graph G are made of stretchable strings and hold the graph from the root and let everything fall down, tightening all the edges as they fall.

Algorithm 6 Breadth-First Search Algorithm (BFS)

```
1: Initialize $T$ = tree consisting only of the root vertex $v_0$.
2: Initialize $L = (v_0)$ a list of visited but unprocessed vertices
3: while $L$ is not empty
4:     Remove the first vertex $v$ from $L$
5:     for each neighbor $w$ of $v$
6:         if $w$ not in $L$ and not in $T$
7:             Add $w$ to the end of the list $L$
8:             Add $w$ and the edge $vw$ to $T$.
```

Theorem 6. *BFS produces a spanning tree of any connected graph starting from any root vertex.*

Proof. The proof is similar to the proof for DFS and is left as an exercise to you. □

Note: BFS uses more memory than DFS since we need a list to keep track of vertices we visited but haven't yet explored. So, why use BFS then? Depending on your goal, the following theorem provides an answer to this question.

Theorem 7. *The unique path between the root vertex and any other vertex in the tree produced by BFS has the shortest length among all such path in G.*

Proof. Exercise. (Hint: Proceed by contradiction). □

Applications of DFS and BFS include:

- Solving and generating random mazes.
- Implementing web crawlers to explore links in a website.
- Used to find and delete unreachable objects in computer memory, which is known as garbage collection.
- Solving games like tic-tac-toe, Sudoku, and the 8-queen problem.
- Analyzing chess endgames.