Shortest Paths

**Definition 1.** A shortest path from vertex \( s \) to vertex \( t \) is a directed path from \( s \) to \( t \) with the property that no other such path has a lower total edge weight.

**Definition 2.** Edge relaxation: Relaxing an edge \((u, v)\) is the process of checking whether one can find a path to \( v \) with a smaller cost than the current path to \( v \) by taking the shortest path to \( u \) and then the edge \((u, v)\).

**Dijkstra’s Algorithm**

Dijkstra’s algorithm finds the shortest path between two given vertices in a weighted graph, assuming that the graph’s edge weights are non-negative. The running time of the algorithm is \( O(E \log V + V \log V) \) when the graph is implemented using adjacency lists. With a special transformation (use of Fibonacci heaps) this can be reduced to \( O(E + V \log V) \), which is the fastest version of this algorithm. The pseudo code for the algorithm is given below.

**Pseudocode**

\[
\text{Dijkstra}(G, s)
\]

1. for each vertex \( v \in V_G \)
2. \( \text{dist}[v] = \infty \)
3. \( \text{parent}[v] = \text{NIL} \)
4. \( \text{dist}[s] = 0 \)
5. \( Q = V_G \)
6. while \( Q \neq \emptyset \)
7. \( u = \text{Extract-Min}(Q) \)
8. for each vertex \( v \in G.\text{Adj}[u] \)
9. \( \text{if dist}[v] > \text{dist}[u] + w(u, v) \)
10. \( \text{dist}[v] = \text{dist}[u] + w(u, v) \)
11. \( \text{parent}[v] = u \)

**Bellman-Ford Algorithm**

The Bellman-Ford algorithm finds the shortest path from a source vertex to all other vertices, even on a graph with negative edge weights. The running time of the algorithm is \( O(VE) \) when the graph is implemented using adjacency lists. Intuitively, Bellman-Ford is a more generalized version of Dijkstra’s algorithm. Both use edge relaxation, but instead of greedily choosing the vertex with the smallest distance estimate and performing the relaxation on all of its outgoing edges like in Dijkstra, Bellman-Ford relaxes all the edges \( |V| - 1 \) times. The pseudo code for the algorithm is given below.
Pseudocode

**Bellman-Ford** \((G, s)\)

1. for each vertex \(v \in V_G\)
2. \(dist[v] = \infty\)
3. \(parent[v] = \text{NIL}\)
4. \(dist[s] = 0\)
5. for \(i\) from 1 to \(|V| - 1\)
6. for each edge \((u, v)\) with weight \(w\) in \(E\)
7. if \(dist[u] + w < dist[v]\)
8. \(dist[v] = dist[u] + w\)
9. \(parent[v] = u\)

Questions

**Problem 1.** Bellman-Ford fails on graphs with negative-weight cycles. Why is this the case?

**Problem 2.** How could you fix Bellman-Ford (as shown above) to detect negative-weight cycles and throw an error in that case?

**Problem 3.** Dijkstra’s algorithm is a *greedy algorithm*. What does this mean?

**Problem 4.** How could you modify the algorithm to find all shortest paths?

**Problem 5.** How could you modify the algorithm to stop once it’s found the shortest path to a particular node? Does this affect the asymptotic running time of the algorithm?

**Problem 6.** Find the shortest path between vertices \(E\) and \(G\).
Problem 7. True or false: Dijkstra’s algorithm will not terminate if run on a graph with negative edge weights.

Problem 8. True or false: If we double the weights of all the edges in a graph, then Dijskstra’s algorithm will produce the same shortest path.

Minimum Spanning Trees

Definition 3. A minimum spanning tree $T$ of $G$ is a spanning tree of $G$ with the property that the sum of the weights of every edge in $T$ is smaller than the sum of the weights in any other spanning tree of $G$.

Prim’s Algorithm

Prims algorithm finds a minimum spanning tree for a connected weighted graph. The greedy algorithm can be summarized in the following way:

- Initialize a tree with a single vertex, chosen arbitrarily from the graph.
- Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
- Repeat the previous step (until all vertices are in the tree).
union-find

The Union-Find data structure is an efficient way to maintain disjoint sets in a group of elements. It has the following methods:

- **Union(u, v)**: Joins the set containing u and the subset containing v (performing set union, or \( \cup \)).
- **Find(v)**: Reports the representative id of the set containing v.

Each subset is organized as a tree. Unions are performed by making the parent pointer of one set’s root point to another’s root. The former should be the tree of lesser rank (the shorter tree); the latter should be the tree of greater rank (the taller tree). Maintaining the rank of each subtree may be done during the algorithm without incurring asymptotically more time. A Find may simply be performed by following the parent pointers up the tree until the root is found.

Since both these operations will take time proportionate to the height of the tree representations, it is in our best interest to reduce those heights whenever possible. To this end, when performing a Find, redirect the parent pointers of all nodes encountered to point to the root (an optimization called path compression). This flattens the tree and reduces subsequent operations' running times.

The amortized analysis of union-find is complex, but we can take for granted that it can perform union and find in \( O(\alpha(V)) \) (where \( \alpha \) is the very slowly increasing inverse Ackermann function).\(^1\)

Kruskal’s Algorithm

This undirected minimum spanning tree algorithm can be described as follows:

- Start off with all vertices as standalone trees within a forest.
- Consider the next minimum edge from the graph.
- Add the edge to the forest if it connects two disjoint trees, otherwise discard it.
- Continue this until every vertex has been considered, and return the tree formed.

The premise is simple, but there are two complex operations that are taking place: union — the merging of two subsets into one, and find — determining if two subsets are connected. The fastest data structure for these operations is the union-find data structure, described above. Using the union-find data structure, the run time of Kruskal’s with sorted input is \( O(\alpha(V)) \). If the input is unsorted, sorting the edges by weight becomes the bottleneck operation, making our running time \( O(E \log(V)) \).

Kruskal’s vs. Prim’s

- Both are algorithms to find the minimum spanning tree.
- Prim starts with a single vertex, and grows a tree from this vertex.
- Kruskal starts with every vertex as a separate tree, and combines them to form a single tree.
- Complexity of eager Prim (with a binomial heap): \( O((E + V) \log V) \), complexity of Kruskal (with union-find with path compression): \( O(E \alpha(V)) \).
- In Kruskal, checking to see if adding an edge will create a cycle can be slow. Thus, Kruskal’s algorithm works better when there are fewer edges to vertices.
- Prim’s algorithm works better for dense graphs with more edges than vertices.
- Since Prim’s algorithm “grows” the tree by adding vertices, it always has a partial tree. If you only need a partial solution, use Prim.

\(^1\)https://en.wikipedia.org/wiki/Ackermann_function#Inverse

4
Questions

Problem 9. Is it guaranteed that a call to $\text{Find}(v)$ will always return the same result throughout the algorithm? If not, is it possible to modify the algorithm such that it does?

Problem 10. How could one use Union-Find to detect cycles in an undirected graph?

Problem 11. Does Kruskal’s algorithm work on a graph with negative weights?

Problem 12. Say we have some MST, $T$, in a positively weighted graph $G$. Construct a graph $G'$ where for any weight $w(e)$ for edge $e$ in $G$, we have weights $(w(e))^2$ in $G'$. Does $T$ still remain an MST in $G'$? Prove your answer. Now if $G$ also had negative weights, would your answer change from the previous part? Prove your answer.

Problem 13. Imagine we have a graph $G$ where all edge weights are equal. Design an algorithm to efficiently find an MST of $G$. Analyze the running time.

Problem 14. Suppose that we have found an MST $T$ of a graph $G$, but soon after, we are told that an edge in $T$ has a higher weight than we at first thought, and as such our MST is now invalid. Is it guaranteed that we can fix our tree by removing an edge and adding a different one? If so, explain how. If not, provide a counterexample.

Problem 15. Suppose that we have found an MST $T$ of a graph $G$, but soon after, we are told that an edge not in $T$ has a lower weight than we at first thought, and as such our MST is now invalid. Is it guaranteed that we can fix our tree by removing an edge and adding a different one? If so, explain how. If not, provide a counterexample.