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

```
Dijkstra(G, s)
1    for each vertex $v \in V_G$
2        $dist[v] = \infty$
3        $parent[v] = \text{NIL}$
4    $dist[s] = 0$
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            if $dist[v] > dist[u] + w(u, v)$
10               $dist[v] = dist[u] + w(u, v)$
11               $parent[v] = u$
```

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?

*Solution.* There is no "cheapest" path in a graph with a negative-weight cycle. Any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle.

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

*Solution.* Add the following code to the end of the code seen above to detect negative-weight cycles:

```plaintext
for each Edge $(u, v)$ with weight $w$ in $E$

    if $dist[u] + w < dist[v]$

    error "Graph cannot contain a negative-weight cycle."
```

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

*Solution.* A greedy algorithm makes the best choice that is currently available. Dijkstra's algorithm follows this paradigm by using a priority queue structure that, when polled, always produces the node with the shortest distance from the source node.

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

*Solution.* Dijkstra's algorithm produces the shortest paths to all nodes in the graph from a single source. In order to find all shortest paths (i.e., the shortest path between any pair of nodes in the graph), you can simply run Dijkstra's from each node in the graph, for a resulting running time of $O(V(|E| + |V|) \log V)$.

**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?

*Solution.* Dijkstra's algorithm produces the shortest paths to all nodes in the graph from a single source. If you are only interested in finding the shortest path from $s$ to $t$, you can stop the algorithm once $t$ is removed from the priority queue. The algorithm would still run in $O(E \log V + V \log V)$ because in the worst case, $t$ may be the last node to be removed off the min-heap.

**Problem 6.** Find the shortest path between vertices $E$ and $G$. 

---

2
Solution. Dijkstra’s algorithm produces the following state:

<table>
<thead>
<tr>
<th>Node</th>
<th>Distance from E</th>
<th>Node</th>
<th>Parent node</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>A</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>9</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>17</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>E</td>
<td>NULL</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>F</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>22</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>13</td>
<td>H</td>
<td>I</td>
</tr>
<tr>
<td>I</td>
<td>12</td>
<td>I</td>
<td>F</td>
</tr>
<tr>
<td>J</td>
<td>20</td>
<td>J</td>
<td>I</td>
</tr>
</tbody>
</table>

We can use the mapping from nodes to parent nodes to find the shortest path from $E$ to $G$, which is $E \rightarrow B \rightarrow F \rightarrow I \rightarrow H \rightarrow G$.

Problem 7. True or false: Dijkstra’s algorithm will not terminate if run on a graph with negative edge weights.

Solution. False. The algorithm will terminate, but it will return a wrong answer.

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.

Solution. True. Any linear transformation on the weights will not affect the calculation of shortest paths.
You can think of it as unit-conversion. For instance, if you converted weights from expression in miles to kilometers, that would not affect the relative ordering of shortest paths.

**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**

Prim’s 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.

[1]: https://en.wikipedia.org/wiki/Ackermann_function#Inverse
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 $O(E\alpha(V))$. 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.

**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?

*Solution.* It is not guaranteed. Whenever a $\text{Union}$ is performed, one of the subsets will always change its indicator value. It is not possible to prevent this for all vertices, as the nature of the methods requires indicators to change, but it is possible to modify the algorithm such that one particular indicator is maintained.

We can add a condition around our set precedence to state that if the $\text{Union}$ of two sets is performed, and one of the sets has our desired constant indicator, then force the other set to become a child of the first. Note that this will create inefficiencies, as we can no longer guarantee that the heights of our trees only increase in cases of equal height.

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

*Solution.* Perform a DFS on the graph. As the search progresses, do the following whenever a new edge $(u, v)$ is encountered:

1. Compare the result of $\text{Find}$ on both $u$ and $v$. If they are in the same component, a cycle exists. Otherwise:
2. Perform $\text{Union}(u, v)$.

By the end of the algorithm, if no node was encountered twice within the same component, we may be sure that there are no cycles. Note that this is not quite as efficient as the node-coloring method, but may be useful in some restrictive circumstances.

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

*Solution.* Yes. The algorithm will still select the lowest weight edges in order (more negative edges first).
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.

Solution. If $G$ only has positive weights, then this claim holds. Proof by contradiction: assume the claim does not hold. Let us say we are using Kruskal’s algorithm (similar argument can be used for Prim’s). Consider the first edge where the algorithms running on $G$ and $G'$, diverged. Then we must have that the algorithm selected some edge $e'$ within $G'$ instead of $e$ within $G$. Then $w(e) < w(e')$ but $w(e)^2 > w(e')^2$. Note that this is not possible for positive integers, and thus we have a contradiction. However, this claim is possible for negative integers! Thus, if $G$ had negative weights, our answer would change to no, $T$ is not necessarily an MST in $G'$.

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.

Solution. Our optimal algorithm would be to run DFS and only keep track of the tree edges (so we don’t introduce any cycles). Notice at any step we can choose any edge, since the edge weights are all equal. The running time is there $O(E + V)$.

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.

Solution. If we can be sure that our MST is invalid because of this change, it follows that the modified edge $e$ cannot exist in any possible MST of the new graph. Let us remove $e$ from our MST. This has now separated the MST into two components, and we no longer have a tree. We may fix this by adding a particular edge.

The cut between the two components must have some minimal edge. This is the edge that we would have encountered in our MST creation in the new graph, as it will be the first of the edges in the cut to be added. Since we are given that the $e$ cannot be in our tree, the minimal edge must be distinct from $e$. Thus we may add this edge, after which we will again have a valid MST of a single component.

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.

Solution. If we can be sure that our MST is invalid because of this change, it follows that the modified edge $e$ exists in all possible MSTs of the new graph. Let us add $e$ to our MST. This has now created a cycle, and we no longer have a tree. We may fix this by removing a particular edge.

The cycle must have some maximal edge. This is the edge that we would not have encountered in our MST creation in the new graph, as it will be the last of the edges in the cycle to be added. Since we are given that the new edge is necessary, the maximal edge must be distinct from $e$. Thus we may remove this edge, after which we will again have a valid MST.