Single Source Shortest Paths

Shortest Path Problem

The input to a shortest path problem is a weighted directed graph $G = (V, E)$, with a weight funciton $w : E \rightarrow R$, mapping edges to real-valued weights.

A path p of length k is defined as an ordered set of vertices $(v_0, v_1, ...v_k)$. When an agent is traversing the graph from source v_0 to destination v_k, the agent would have to travel through the points in p in the specified order.

The weight $w(p)$ of a path is the sum of weights of the constituent edges.

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

We define the shortest path weight function $\delta(u, v)$ for a path from u to v to be given by

$$\delta(u, v) = \begin{cases} min\{w(p) : u - v\} & \text{if there is a path from u to v} \\ \infty & otherwise \end{cases}$$

Edge weights can represent metrics other than distances, such as time, cost, penalties, loss or any other quantitiy that accumulates linearly along a path that you want to minimize on.

The Breadth First Search Algorithm seen earlier, is a example of a simple shortest path algorithm that works for unweighted graphs.

Properties of a Shortest Path

Shortest Path algorithms rely on the property that a shortest path between two vertices contains other shortest paths within it. This is the **optimal substructure** of a shortest path.

Shortest Paths **cannot contain negative weight cycles** that are reachable from the source and the destination vertices. If there is a negative weight cycle in the path, then the agent may choose to travel along the negative weight cycle repeatedly to reduce the total weight of the path, and this can go on ad-infinitum.

Shortest Path algorithms typically assume that the that all the edge weights in the shortest path algorithm are non-negative. This nessecarily avoids the existence of negative edge weights.

The previous property can be generalized further to tell the shortest path **cannot contain a cycle at all.** Assume that a path has a cycle, and like previously described, the agent could traverse the cycle n times in order to increase the weight of the total path. The minimum weight of the path occurs when the agent traverses the cycle zero times, which is equivalent to the cycle not being in the path at all.

Shortest paths, and more generally shortest path trees, are **not nessecarily unique**. You can have a weighted directed graph with two shortest path trees for the same root.

Common Functions

Initialize Single Source

```
INITIALIZE-SINGLE-SOURCE
for vertex v in G.v
    v.d = infinity
    v.pi = NIL
s.d = 0
```

After calling the initial single source, we've added two properties to each of the vertices:

1. **The Distance from Source**: Before we make any calculations, we set it to be infinity. We reduce the value as we go forward.
2. **Priori**: When making a path betweek the source and the current vertex, the vertex that you come from to the current vertex. If there is no path, it will be nil.

The last line of the function sets the source vertex to have a distance of 0 from the source.

Relax

```
RELAX(u,v,w)
if v.d > u.d + w(u,v):
    v.d = u.d + w(u,v)
    v.pi = u
```

The process of relaxing an edge is basically just checking if the path going through the vertex u, improves the shortest path to vertex v found so far. By using the relax function, we're tightening the upper bound on the shortest distance between s adn v.

We just check if the distance of v from the source is less than the distance of u from the source, plus the weight of the edge (u, v). If that is the case, it's faster to reach v through u than through the existing route. Thus, the priori of v is set to be u, and the distance is set to be distance to u + $w(u, v)$.

Properties Coming from the Relax Function

We represent the shortest distance between the source s and the destination v with the function $\delta(s, v)$.

1. Triangle Inequality:

   For any edge (u, v), we have $\delta(s, u) \leq \delta(s, v) + w(u, v)$

2. Upper bound Property:

   For any iteration, $v.d \geq \delta(s, v)$. The value of $v.d$ is always an upper bound for $\delta(s, v)$

3. No-Path property:

   If there is no path between s and v, then $\delta(s, v) = \infty$

Djikstra's Algorithm

Djikstra's algorithm is a single source shortest path algorithm. It solves the Single Source Shortest Path problem for a weighted, directed graph $G = (V, E)$, given that all the edge weights are non-negative.

Djikstra's algorithm works by maintaining a set S of vertices whose shortest paths have been determined. The algorithm repeatedly selects a vertex $u \in V - S$ with the miniimum shortest path estimate. u is added to the set S, and the edges leaving u are all relaxed.

Pseudocode

In the Algorithm itsef, we're storing the set $Q = V - S$ using a priority queue, ordered based on the parameter $v.d$. EXTRACT-MIN removes the minimum element from the queue. UPDATE-PRIORITY-QUEUE updates the order of the elements in the queue based on the updated values of the elements.

```
DJIKSTRA(G,w, s)
1. INITIALIZE-SINGLE-SOURCE(G,s)
2. S = empty array
3. Q = empty array
4. for each vertex u in G.V
5.     INSERT(Q,u)
6. while Q != empty array:
7.    u = EXTRACT-MIN(Q)
8.    S = S + {u}
9.    for each vertex v in G.adj(u):
10.       RELAX(u,v,w)
11.       UPDATE-PRIORITY-QUEUE(Q)
```

Line 1-3 are just initializations.

Line 4 and 5 populates Q with the elements of the vertex set. Since the queue is ordered by $v.d$, the first element in the queue would be the source itself, and the remaining elements ordered arbitrarily.

In line 7 we choose the queue element with the minimal size in the variable u, and it's appended to the set S. In lines 9,10 and 11 we relax the vertices adjacent to u, and we update Q with the new values after the relaxation.

Time complexity

The initialization steps till line 3 is constant time. Line 4 and 5 are constructing a priority queue, which takes time on the order of $O(\log V)$ with the max heapify algorithm.

In line 7, you're doing EXTRACT-MIN(Q), which returns the value of the minimum element, and removes it from the queue. The EXTRACT-MIN code calls the max heapify algorithm, so this has a time complexity of $O(\log V)$. This happens once per vertex, which gives us a running time for this section to be $O(V \log V)$.

Following Relaxation, we are updating the values of all the vertices, and the priority queue has to be updated again. For this, we update the priority queue by using the max heapify algorithm, which takes $O(\log V)$. This happens once per edge, which gives us a running time for this section to be $O(E \log V)$.

Both combined, the total running time of the algorithm to be $O(E \log V + V \log V)$

Bellman Ford Algorithm

Bellman Ford Algorithm sovles the Single Source Shortest Path problem for a general case, where the edge weights may be negative as well. The Algorithm checks if there are any cycles of negative weights that are reachable from the source. If there is, then it reports that no solution exists, otherwise it returns a solution for the algorithm.

Pseudocode

The procedure relaxes edges repeatedly, progressively decreasing the upper bound $v.d$ until it achieves $\delta(v, s)$. The algorithm returns True only if there's no negative weight cycles. If any are found, it returns False, and terminates the routine.

```
BELLMAN-FORD(G,w,s)
1.INATIALIZE-SINGLE-SOURCE(G,s)
2.for i = 1 to |G.V| - 1
3.    for each edge (u,v) in G.E
4.        RELAX(u,v,w)
5.for each edge (u,v) in G.E
6.    if v.d > u.d + w(u,v)
7.        return False
8.return True
```

After initializing the $v.d$ and $v.pi$ in line 1, it iteratites over the whole set of edges for $|V| - 1$ times. Each iteration, it goes over the whole set of edges, and relaxes them, updating the estimates of the $v.d$. After $|V| - 1$ iterations, it has acheived the $\delta(s, v)$ as the value of $v.d$.

Lines 5-8 check for negative weight cycles. Essentially, it's checking if the triangle inequality is satisfied. If it's not satisfied, it's a sign of there being negative weight cycles, which we do not want. And thus it returns a False.

If all edges satisfy the triangle inequality, it returns True.

Time Complexity

The Bellman Ford Algorithm runs in Time complexity $O(V^2 - E)$

Single Source Shortest Paths in Directed Acyclic Graphs

In this section, we introduce another restriction to the weighted directed graphs. They must remain acyclic. We'll see that if the edges of a weighted DAG are relaxed according to a topological sort of it's vertices, then it takes $\Theta(V + E)$ running time to compute the shortest path from a source to the vertices.

The algorithm begins by topologicaly sorting the given DAG, which imposes a linear ordering of it's vertices. If the DAG contains a path from the vertex u to the vertex v, then u preceeds v in the topological sort.

Otherwise, there is not going to be a path. DAG-SHORTEST-PATH makes only one pass over the vertices sorted topologically. As it proceeds over a vertex, it relaxes each edge that leaves the vertex.

```
DAG-SHORTEST-PATH(G,w,s)
1.topologically sort DAG
2.INITIALIZE-SINGLE-SOURCES(G,s)
3.for each vertex u in G.V taken in topologically sorted orc
4.  for each vertex v in G.Adj[u]
5.    RELAX(u,v,w)
```

The topological sort takes a running time of $\theta(V + E)$. Initialize single source takes a running time of $\theta(V)$.

The relaxation happens once per edge, so that takes a running time of $\theta(E)$. This gives us a total running time of the algorithm to be $O(V + E)$