Point-to-Point Shortest Path Algorithms

Arnab Ganguly, Assistant Professor
Department of Computer Science, University of Wisconsin – Whitewater
Algorithms in the Real World (CS 738)

1 Shortest Path Algorithms: An Overview

We look at the problem of computing the shortest path from a source vertex s to a target vertex t. Let $v_1, v_2, \ldots, v_k$ be the only vertices that have an outgoing edge to t. Then, any path from s to t must travel via one of these k vertices; in particular, the shortest path must also travel through one of these k vertices. Let $D(x, y)$ represent the length of the shortest path from x to y, and for any edge from u to v, let $W(u, v)$ represent its weight. The main intuition is that a shortest path from s to t satisfies the following equation:

$$D(s,t) = \min\{D(s, v_1) + W(v_1, t), D(s, v_2) + W(v_2, t), \ldots, D(s, v_k) + W(v_k, t)\}$$

In other words, the length of the shortest path from s to t can be obtained by adding the weight of the edge (v_i, t), $1 \leq i \leq k$, to the shortest path from s to v_i and then picking the minimum.

Recall from data structure days that in acyclic graphs, v_1 through v_k are ahead in topological order compared to t. That allowed us to process vertices in topological order. Alas, general graphs may not have a topological order, as they may have a cycle. Worse still, even the concept of shortest path is somewhat ambiguous for a graph with arbitrary edge weights. Visualize a graph which has a cycle and all edge weights are negative. Now, if a path from s to t goes via this cycle, then we can simply loop around the cycle infinitely many times to bring the length of the shortest path down to $-\infty$. Practically, this makes little sense.

Summarizing, in a general graph with arbitrary weights, what we are really looking for is a shortest path that visits a vertex at most once. Unfortunately, this problem is very hard. In fact, it is one of the NP-hard problems, which loosely translates to it being extremely unlikely to even have a polynomial time solution (in the number of vertices), let alone an efficient one.[1] So, if we want a decent solution, assumptions are necessary.[2]

2 Dijkstra's Algorithm

Dijkstra's algorithm assumes: **all edge weights are non-negative.**[3]

This non-negative assumption, however, gives us a crucial way of dissecting the problem. We observe that if v is a vertex on the shortest path from s to t, then $D(s, v) \leq D(s, t)$, i.e., the length of the shortest path from s to any v is at most the length of the shortest path from s to t.

[1]The problem is closely associated with the Hamiltonian path problem, which is stated as follows: *is there a path that visits each vertex exactly once?* This is an NP-complete problem, which very loosely means that it is unlikely that one can answer this question in polynomial time. If you have heard of the famous $P = NP$ conjecture, know that it can be reformulated as "*is there a polynomial time algorithm for the Hamiltonian path problem?*" If you have not heard of the conjecture, I encourage you to look it up.

[2]Well at least until someone proves that $P = NP$.

[3]Pretty realistic assumption, as graphs with negative edge weights are rare.

The main idea is **process vertices in non-decreasing order of their shortest path distance from** s. We already know that Breadth-First Search (BFS) does something of this sort, and it is no surprise that Dijkstra's algorithm can be viewed as BFS, albeit with some modifications.

We keep track of the shortest path distances from the source vertex using an array $dist[\,]$ having length equal to the number of vertices. At any point, $dist(v)$ stores the distance of the shortest path from s to v that has been found so far. Initially, $dist(s) = 0$ and $dist(v) = \infty$ for all $v \neq s$.

We use *closed* to keep track of the vertices to which shortest path has already been found.At any point, if v is in *closed*, then $dist(v)$ records the length of the shortest path from s to v.

A vertex v is in *open* if we have found a path to v, but we are not sure if it is the shortest path or not. Specifically, a vertex v is in *open* if v is not *closed* and $dist(v) \neq \infty$.

We use $parent[\,]$ array (having length equal to the number of vertices), where $parent(v)$ is the vertex preceding v in the shortest path from s to v that has been found so far. Thus, if v is in *closed* and $parent(v) = u$, then u is the vertex that precedes v on the shortest path from s to v.

Dijkstra's Algorithm

```
• Initially $dist(s) = 0$ and $dist(v) = \infty$, for all $v \neq s$
• Initialize: $open = \{s\}$ and $closed = \varnothing$
• While there exists an open vertex, execute:
    – Let $u$ be an open vertex with the minimum $dist$ value.
    – If $u = t$, then stop.
    – remove $u$ from open and add $u$ to $closed$
    – For each adjacent vertex $v$ of $u$
        ∗ If $v$ is closed, then continue;
        ∗ Let $len = dist(u) + W(u, v)$
        ∗ If $len < dist(v)$, then
            · $dist(v) = len$
            · $parent(v) = u$
            · if $v$ is not in $open$, add $v$ to $open$
```

Constructing a Shortest Path from s to v

```
• Initialize $path = \langle v \rangle$ and $x = v$
• While $parent(x) \neq \varnothing$, execute:
    – add $parent(x)$ at the front of $path$
    – set $x = parent(x)$
```

Constructing a Shortest Path Tree

```
• Initially, simply add all the vertices without adding any edges
• For each added vertex $v$, add an edge from $parent(v)$ to $v$
```

2.1 Simulation

The table below simulates the shortest path algorithm on the graph in the following figure.

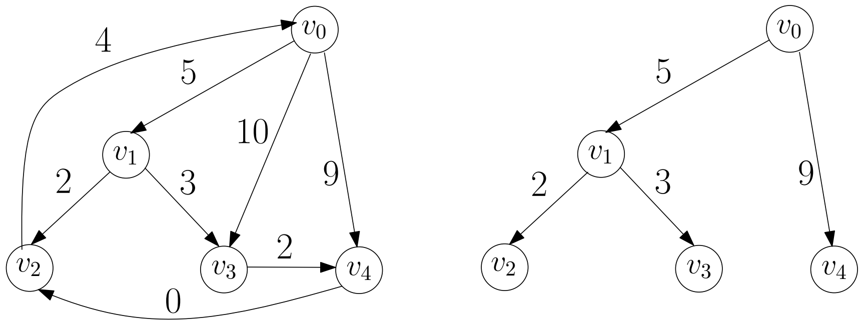

Figure 1: A graph (left) and its shortest path tree (right)

	dist					open	closed	parent				
	v_0	v_1	v_2	v_3	v_4			v_0	v_1	v_2	v_3	v_4
At Start:	0	∞	∞	∞	∞	$\{v_0\}$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
Min open vertex: v_0 Relaxing v_0	0	5	∞	10	9	$\{v_1, v_3, v_4\}$	$\{v_0\}$	$\varnothing$	v_0	$\varnothing$	v_0	v_0
Min open vertex: v_1 Relaxing v_1	0	5	7	8	9	$\{v_3, v_4, v_2\}$	$\{v_0, v_1\}$	$\varnothing$	v_0	v_1	v_1	v_0
Min open vertex: v_2 Relaxing v_2	0	5	7	8	9	$\{v_3, v_4\}$	$\{v_0, v_1, v_2\}$	$\varnothing$	v_0	v_1	v_1	v_0
Min open vertex: v_3 Relaxing v_3	0	5	7	8	9	$\{v_4\}$	$\{v_0, v_1, v_2, v_3\}$	$\varnothing$	v_0	v_1	v_1	v_0
Min open vertex: v_4 Relaxing v_4	0	5	7	8	9	$\varnothing$	$\{v_0, v_1, v_2, v_3, v_4\}$	$\varnothing$	v_0	v_1	v_1	v_0

2.2 Correctness Proof

First it is obvious that the algorithm terminates because once a vertex is closed, it is never opened again; hence, each is relaxed exactly once. We prove the correctness by induction on the size of *closed*. Specifically, we show that when the k^{th} vertex, say v_k, is closed by the algorithm, then shortest path from s to v_k has been found. Note that the base case (i.e., $k = 1$) holds because $s = v_1$ is the first vertex closed and obviously $D(s, s) = 0$. By induction hypothesis, assume that the shortest path from s to v_k has been found, and the algorithm is about to close v_{k+1}, i.e., the $(k + 1)^{th}$ vertex. We complete the proof by contradiction.

In particular, assume that the algorithm is incorrect at this point, i.e., it fails to find the shortest path to v_{k+1}; then, obviously there is a shorter path P from s to v_{k+1} that uses an open vertex. Let v' be the first open vertex on $P = \langle s = v_1, v_2, \ldots, v_r, v', \ldots, v_{k+1}\rangle$, where $r \leq k$; note that $v'_1 \neq v_{k+1}$. By the algorithm, $dist(v') \leq dist(v_r) + W(v_r, v') = D(s, v_r) + W(v_r, v')$. Also, since P is a shorter path, we have $D(s, v_r) + W(v_r, v') + D(v', k) < dist(v_{k+1})$. But then, $dist(v') < dist(v_{k+1})$ and the algorithm should have selected v' from open, a contradiction. Hence, v_{k+1} is the correct choice, which completes the proof.

2.3 Complexity

Let V and E be the number of vertices and edges in the graph respectively. Assume that $t_{update} = \Omega(1)$ is the time needed to update the distance label of a vertex in *open*, $t_{min} = \Omega(1)$ is the time needed to select the vertex with the minimum distance label in *open*, and t_{insert} is the time needed to insert a vertex into *open* Whether a vertex is closed or not can be easily detected in $O(1)$ by maintaining a boolean array.

Note that each vertex is added to *open* at most once. Since each edge is relaxed at most once, the total number of updates to *open* or *dist/parent* arrays is at most E. Hence, the complexity is $O\big(E * t_{update} + V * (t_{min} + t_{insert})\big)$.

By implementing the *open* set of vertices as a priority queue, we obtain $t_{update} = t_{insert} = t_{min} = \Theta(\log V)$. This leads to a complexity of $\Theta\big((E+V)\log V\big)$; as in most graphs, $E = \Omega(V)$, the complexity reduces to $\Theta(E \log V)$.

If graphs are dense, i.e., $E = \Omega(V^2/\log V)$, then we implement open as an array of length V, which results in $t_{update} = t_{insert} = \Theta(1)$ and $t_{min} = \Theta(V)$. Thus, we get the complexity $O(V^2)$.[4]

2.4 The Caveats

The main problem with Dijkstra's algorithm is that it discovers vertices which are never going to be on the desired shortest path from s to t. Say you want to travel from Chicago to Denver (around 1000 miles), and use Dijkstra's algorithm. Now, this algorithm will discover New York before it closes Denver simply because New York is closer to Chicago at around 800 miles; this is because Dijkstra closes vertices in increasing order of shortest path length from the source. This is bad because New York is in the opposite direction, and we should have completely (or at least mostly) discarded cities to the east of Chicago. In fact, Dijkstra's algorithm will close any and all cities within a 1000 mile radius of Chicago before it finds Denver. The main idea with P2P shortest path algorithms is to prune this search space.

3 Bi-directional Algorithm

Continuing with our previous example, say we run Dijkstra in both directions, one from Denver and the other from Chicago, and we will terminate once both the searches closes the same vertex in both directions. Say, we close Lincoln, which is roughly 500 miles from both Denver and Chicago; then, we will find the shortest path from Chicago to Denver. Interestingly note that in this case the two Dijkstra instances will discover vertices within a 500 mile radius, which is likely to contain fewer cities than the 1000 mile radius around Chicago.[5]

In a nutshell, here lies the main idea with the bi-directional algorithm:

- Alternate Dijkstra's algorithm in forward and reverse direction (starting from s in the original graph and starting from t in the reverse graph). In both directions, maintain the *distance* arrays and the *open* sets.
- When a a vertex label is *set* in one direction, and we see that the label of the same vertex is also set in the reverse direction, it means we have found a path from s to t. Record the

[4]Note that when $E = \Theta(V^2)$, the array based implementation of open is better than a priority queue based implementation; the former has a complexity of $\Theta(V^2)$ and the latter $\Theta(V^2 \log V)$. Typically, in most real cases, graphs are sparse, i.e., $E = \Theta(V)$.

[5]Because $(r_1 + r_2)^2 > r_1^2 + r_2^2$. So, heuristically thinking, if cities are evenly distributed, then the r_1 and r_2 radius circles will cover less cities than the $(r_1 + r_2)$ radius circle.

length of this path by taking the sum of the *distance* array entries for the two vertices and update a variable μ if μ is larger (initially, $\mu = \infty$).

- Terminate once you see that the sum of minimum distances in the two *open* sets exceed μ.

Bi-directional Algorithm

```
• Initially $dist(s) = 0$ and $dist(v) = \infty$, for all $v \neq s$
• Initially $dist_R(t) = 0$ and $dist_R(v) = \infty$, for all $v \neq t$
• Initialize: $open = \{s\}$ and $open_R = \{t\}$
• Initialize: $closed = \varnothing$ and $closed_R = \varnothing$
• Set $\mu = \infty$
• While there exists a vertex in both $open$ and $open_R$, execute:
    1. Let $min$ and $min_R$ be the minimum distances in $open$ and $open_R$ respectively.
    2. If $min + min_R \geq \mu$, then return $\mu$.
    3. Remove a minimum vertex $u$ from open (i.e., a vertex with $dist(u) = min$).
       If $u = t$, then return $min$, else add $u$ to $closed$.
    4. For each adjacent vertex $v$ of $u$
        – If $v$ is in $closed$, then continue;
        – Let $len = dist(u) + W(u, v)$
        – If $len < dist(v)$, then
            ∗ $dist(v) = len$
            ∗ if $v$ is not in $open$, add $v$ to $open$
            ∗ if ($dist_R(v) \neq \infty$ and $len + dist_R(v) < \mu$), set $\mu = len + dist_R(v)$;
    5. Remove a minimum vertex $u_R$ from open (i.e., a vertex with $dist_R(u_R) = min_R$).
       If $u_R = s$, then return $min_R$, else add $u_R$ to $closed_R$.
    6. For each adjacent vertex $v_R$ of $u_R$
        – If $v_R$ is in $closed_R$, then continue;
        – Let $len = dist_R(u_R) + W_R(u_R, v_R)$
        – If $len < dist_R(v_R)$, then
            ∗ $dist_R(v_R) = len$
            ∗ if $v_R$ is not in $open_R$, add $v_R$ to $open_R$
            ∗ if ($dist(v_R) \neq \infty$ and $len + dist(v_R) < \mu$), set $\mu = len + dist(v_R)$;
• return $\mu$
```

Although the worst case complexity is still that of Dijkstra's, in most practical scenarios, this is going to be much faster. Imagine a graph whose maximum outdegree is b, and the graph has a diameter (i.e., the longest hop distance between any two vertices) is $d = \log_b V$. Then, the bi-directional search can terminate after potentially closing $2b^{d/2} = 2\sqrt{V}$ vertices, i.e., after relaxing $b\sqrt{V}$ edges. If b is not too high compared to V (which is usually the case in practical scenarios), then it is pretty likely that this will outperform Dijkstra.

4 A* Algorithm

A* is yet another "goal directed" search algorithm. Here, the idea is that we attach to each vertex v a heuristic estimate $H(v,t)$ of what the shortest path from that vertex v to the target vertex t is. Now, instead of selecting the vertex from open that has the minimum distance label, we select a vertex that minimizes the sum of the distance label and the heuristic estimate. As long as the heuristic estimate is a lower bound on the actual shortest path distance, it is guaranteed that A* will find the shortest path.

Going back to our example, say we have an estimate that the shortest path from New York to Denver is of length 1200. Then, the A* algorithm is no longer going to close New York before it closes Denver (as the sum of the actual shortest distance from Chicago to New York and the heuristic estimate from New York to Denver is 2000, which is higher than the actual distance from Chicago to Denver).

A* Algorithm

```
• Initially $dist(s) = 0$ and $dist(v) = \infty$, for all $v \neq s$
• Initialize: $open = \{s\}$
• While there exists an open vertex, execute:
    – Let $u$ be an open vertex with the minimum $dist(u) + H(u,t)$ value.
    – If $u = t$, then stop.
    – For each adjacent vertex $v$ of $u$
        ∗ Let $len = dist(u) + W(u,v)$
        ∗ If $len < dist(v)$, then
            · $dist(v) = len$
            · if $v$ is not in $open$, add $v$ to $open$
```

4.1 Heuristics Definition

Admissible Heuristic. A heuristic is said to be admissible if for every vertex v, $H(v,t) \leq D(v,t)$, i.e., for every vertex the heuristic estimate of the shortest path from v to t is a lower bound on the shortest path distance from v to t.

Consistent Heuristic. A heuristic is said to be consistent if for any vertex v, we have $H(v,v) = 0$, and for every edge (u,v), we have $W(u,v) + H(v,t) \geq H(u,t)$.

4.2 Properties of Heuristics

If a heuristic is admissible, then A* will find the shortest path. For the proof, check the paper "A Formal Basis for the Heuristic Determination of Minimum Cost Paths".

If a heuristic is consistent, then it is also admissible. Let $P = \langle s = v_1, v_2, v_3, \ldots, v_k = t \rangle$ be a shortest path from s to t. Since the heuristic is consistent, we get the following:

$$\begin{aligned} W(v_1, v_2) + H(v_2, t) &\geq H(v_1, t) \\ W(v_2, v_3) + H(v_3, t) &\geq H(v_2, t) \\ \vdots \quad\quad \vdots \quad\quad & \vdots \quad\quad \vdots \\ W(v_{k-2}, v_{k-1}) + H(v_{k-1}, t) &\geq H(v_{k-2}, t) \\ W(v_{k-1}, v_k) + H(v_k, t) &\geq H(v_{k-1}, t) \end{aligned}$$

From the above, we get

$$\begin{aligned} & H(v_1, t) \leq W(v_1, v_2) + W(v_2, v_3) + \cdots + W(v_{k-1}, v_k) + H(v_k, t) \\ \implies & H(v_1, t) \leq D(v_1, v_k) = D(v_1, t) \end{aligned}$$

For an admissible (but not consistent) heuristic, the A* algorithm may add a vertex to *open* more than once. Thus, A* may update the distance estimate of a vertex even after it is has been removed from *open*. Hence, for non-consistent admissible heuristics, we CANNOT use the concept of *closed* as we have been doing for Dijkstra's and Bi-directional algorithm.

If a heuristic is consistent, then A* adds a vertex to *open* at most once, i.e., the shortest path to a vertex is found when it is removed from open. We show that when the k^{th} vertex, say v_k, is removed from *open*, then shortest path from s to v_k has been found. We prove this by induction. Note that the statement holds for $k = 1$ because $s = v_1$ is the first vertex removed and obviously $D(s, s) = 0$. Assume that the shortest path from s to v_k has been found, and the algorithm is about to remove v_{k+1}, i.e., the $(k+1)^{th}$ vertex. If the algorithm does not find the shortest path to v_{k+1}, then obviously the shortest path P from s to v_{k+1} uses a vertex (other than v_{k+1}) currently in open. Let v' be the first open vertex on $P = \langle s = v_1, v_2, \ldots, v_r, v', \ldots, v_{k+1} \rangle$, where $r \leq k$; note that $v'_1 \neq v_{k+1}$. Clearly,

$$dist(v') = D(s, v_r) + W(v_r, v') = D(s, v')$$

Since the algorithm selected v_{k+1}, we have

$$dist(v') + H(v', t) \geq dist(v_{k+1}) + H(v_{k+1}, t)$$

Also, since P is a shorter path, we have

$$\begin{aligned} & D(s, v_r) + W(v_r, v') + D(v', v_{k+1}) < dist(v_{k+1}) \\ \implies & dist(v') + D(v', v_{k+1}) < dist(v_{k+1}) \\ \implies & dist(v') + D(v', v_{k+1}) < dist(v') + H(v', t) - H(v_{k+1}, t) \\ \implies & D(v', v_{k+1}) < H(v', t) - H(v_{k+1}, t) \\ \implies & D(v', v_{k+1}) + H(v_{k+1}, t) < H(v', t) \\ \implies & D(v', v_t) < H(v', t) \end{aligned}$$

But the last inequality violates admissibility criteria; hence, the proof follows by contradiction.

Hence, for consistent heuristics, we CAN again use the concept of *closed* as we have been doing for Dijkstra's and Bi-directional algorithm.

Monotone Property. If there exists two consistent heuristics H_1 and H_2, such that at every node v, H_1 dominates H_2, i.e., $H_1(v,t) \geq H_2(v,t)$, then any node expanded by the A* algorithm using H_1 is also expanded by H_2.

This means it is better to choose a heuristic that has a tighter estimate on the length of the shortest path as it will expand no more vertices than a heuristic with a looser estimate.

4.3 Euclidean Heuristics

The most obvious (consistent) heursitic measure of the shortest-path distance is the straight-line distance between the two points. In other words, $H(v,t) = \sqrt{(x_v - x_t)^2 + (y_v - y_t)^2}$, where v and t are represented respectively by the points (x_v, y_v) and (x_t, y_t).

4.4 Using Landmarks: ALT algorithm

Although Euclidean heuristic will reduce the number of edges relaxed, we seek to obtain a tighter estimate. Landmark is one these techniques.

Pre-processing Step: We choose a subset of vertices $\mathcal{L}$ and call them landmarks. Then, for each vertex $\ell \in \mathcal{L}$, we pre-compute and store the shortest path lengths to and from every other vertex in the graph. That is for every $\ell \in \mathcal{L}$ and every $v \in V$, we store $D(\ell, v)$ and $D(v, \ell)$ explicitly.

Heuristic Computation: Suppose we want to estimate $H(v,t)$. Note that by the definition of shortest paths, for any landmark $\ell \in \mathcal{L}$, we have

$$D(v,t) + D(t,\ell) \geq D(v,\ell) \implies D(v,t) \geq D(v,\ell) - D(t,\ell)$$
$$D(\ell,v) + D(v,t) \geq D(\ell,t) \implies D(v,t) \geq D(\ell,t) - D(\ell,v)$$

Thus, $D(v,\ell) - D(t,\ell)$ and $D(\ell,t) - D(\ell,v)$ form a lower bound on the shortest path distance $D(v,t)$ from v to t. Hence, we choose:

$$H(v,t) = max_{\ell \in \mathcal{L}} \left\{ max\big(D(v,\ell) - D(t,\ell), D(\ell,t) - D(\ell,v)\big) \right\}$$

Note that the heuristic is consistent.

4.4.1 Random Landmarks

The first obvious candidate is to choose the set of landmarks randomly.

4.4.2 Grid-based Method

The random landmark selection often does not pan out well. So, we need to devise an intelligent way of selecting landmarks. Here, we shall see a strategy which works well, but for more clever strategies you may want to check the papers cited at the beginning.

For this strategy we assume that the number of landmarks L is a perfect a square, i.e., $L = m * m$ where m is a positive integer. We divide the graph into $m \times m$ grid, where the left and right boundaries of the grid are the minimum and maximum longitudes, and the bottom and top boundaries of the grid are the minimum and maximum latitudes. For each square in the grid, we find out the latitude longitude intersection right at the cente of the square, and then pick the vertex that is closest to this centre w.r.t Eucledian distance.