Single-Pair Shortest-Path using Dijkstra

Sometimes, instead of finding the shortest paths from one source vertex \( s \) to every other vertex \( v \), we are only interested in a shortest path from one source vertex \( s \) to one target vertex \( t \). We can modify Dijkstra to handle this problem by terminating as soon as \( t \) is extracted from the priority queue. The pseudocode for the modified Dijkstra is included below:

\[
\begin{align*}
&\text{Initialize()} \\
&Q \leftarrow V[G] \\
&\text{while } Q \neq \emptyset \\
&\quad \text{do } u \leftarrow \text{EXTRACT\_MIN}(Q) \\
&\quad \text{if } u == t: \text{break} \\
&\quad \text{for each vertex } v \in \text{Adj}[u] \\
&\quad \quad \text{do } \text{RELAX}(u, v, w)
\end{align*}
\]

There are multiple ways to speed this up even further.

Bidirectional Search (Highly branched graphs)

Suppose we have a graph \( G \) in which the number of vertices distance \( d \) away from any given vertex is roughly \( b^d \) (in other words, our graph has a branching factor of \( b \)). An example of a graph like this is the Rubik’s cube state graph. If we have \( n \) total vertices in this graph, then we can roughly estimate the diameter of the graph to be \( \log_b n \) if we assume that the last layer outwards dominates the number of nodes. In these highly branching situations, bidirectional search can be used to speed up the Dijkstra modification to solve the single-pair shortest-path problem. Notice that the speedup does not change worst-case behavior, but reduces the number of visited vertices in practice.

The algorithm

Bidirectional search runs a forward Dijkstra’s search from \( s \) in parallel with a reverse Dijkstra’s search from \( t \) and stops once the two searches meet (see below for more details on the stopping condition).

Stopping condition

Wrong idea 1
Stop as soon as the two searches connect at an edge \((p, q)\). Then reconstruct the path from \(s\) to \(p\) to \(q\) to \(t\). To get the weight of this path, simply add the weights of \(p.s\) (the distance from \(s\) to \(p\)), \(q.t\) (the distance from \(t\) to \(q\)), and \(w(p, q)\).

This would be incorrect in cases where \(w(p, q)\) is very large. Just because the edge was discovered in both searches does not mean that the edge is very efficient.

Wrong idea 2
Stop as soon as the same node \(m\) has been extracted from both \(Q_f\) the forward search queue and \(Q_b\) the backward search queue. Then reconstruct the path from \(s\) to \(m\) to \(t\). To get the weight of this path, simply add the weights of \(m.s\) (the distance from \(s\) to \(m\)) and \(m.t\) (the distance from \(t\) to \(m\)).

This termination condition guarantees that a shortest path has been found (the frontiers met with one connecting vertex), but the actual shortest path might not contain \(m\). See example in Figure 1, where \(w\) is the first node to be extracted from both queues, but the actual shortest path does not contain \(w\).

Correct idea
Instead, we maintain a variable \(\mu\) to represent the weight of the current best seen path between \(s\) and \(t\). We do the following check every time we find an edge between the forward exploration and reverse exploration:

1. Update \(\mu\) to be \(\min(\mu, p.s + w(p, q) + q.t)\).
2. Find the minimum, \(l_s\) of \(v.s\) over every unexpanded node \(v\) in the forward queue
3. Find the minimum, \(l_t\) of \(u.t\) over every unexpanded node \(u\) in the reverse queue
4. If \(l_s + l_t \geq \mu\), then all remaining paths on the queue cannot possibly reduce the shortest path \(\mu\) any further, so stop.

Correctness Assume that there exists a path \(p\) from \(s\) to \(t\) with weight smaller than \(\mu\). Let \((u, v)\) be an edge on path \(p\) s.t. \(\delta(s, u) < l_s\) and \(\delta(v, t) < l_t\). This means that both \(u\) and \(v\) have been extracted from their corresponding queues. Without loss of generality, assume that \(v\) was processed after \(u\). Then when \(v\) was extracted, \(\mu\) was updated to a value at most the weight of path \(p\). That is, \(\mu <= \delta(s, u) + w(u, v) + \delta(v, t)\). However, this contradicts our assumption that the weight of \(p\) is smaller than \(\mu\).

Performance Assume that a highly branching graph has a branching factor \(b\), and the diameter is roughly \(\log_b n\). The forward and backward searches can potentially meet in the middle of this diameter, each exploring a depth of \(\frac{\log_b n}{2}\). Thus the number of vertices explored by each search direction is approximately \(b\) to the power of depth, which is \(n^{\frac{1}{2}}\). Thus in the best case, bidirectional search only explores \(O(\sqrt{n})\) vertices.
Figure 1: Forward and Backward Search and Termination.
**Goal-Directed Search and $A^*$**

A problem with using DIJKSTRA for the single-pair shortest-path problem is that it sometimes wastes a lot of time exploring paths that obviously lead us away from the target $t$. If we change the edge weights to direct DIJKSTRA towards the target, but without changing the identity of the shortest paths, then we can potentially save time in practice. With the changed weights, paths leading towards locations near the target should have lower weights, so that DIJKSTRA naturally goes "down-hill" towards the target. To achieve this, we introduce potential functions.

**Potentials**

A potential function is a function of target $t$ and vertex $v$, denoted $\lambda_t(v)$. We can modify weights using this potential:

$$w^*(u, v) \leftarrow w(u, v) - \lambda_t(u) + \lambda_t(v)$$

We call a potential function **feasible** if $w(u, v) - \lambda_t(u) + \lambda_t(v) \geq 0, \forall (u, v) \in E$. So if we pick potential functions that are feasible, the new $w^*(u, v)$ will remain non-negative, meaning that we can still use Dijkstra on this new graph with reduced weights!

Notice that, for any path $p$ with weight $w(p)$ from $s$ to $t$ in the old graph, its weight in the new graph will become $w(p) - \lambda_t(s) + \lambda_t(t)$. That is to say, all paths from $s$ to $t$ have their weights modified by the same amount. Thus, what was a shortest path in the old graph would remain a shortest path in the new graph. This can be proved by expanding a path $p$ to its vertices, $v_1, v_2, \ldots, v_{n-1}, v_n$. Then, the sum of the path’s edges would be

$$\sum_{i=1}^{n-1} w(v_i, v_{i+1}) - \lambda_t(v_i) + \lambda_t(v_{i+1})$$

$$= w(v_1, v_2) - \lambda_t(v_1) + \lambda_t(v_2) + w(v_2, v_3) - \lambda_t(v_2) + \lambda_t(v_3) + \ldots$$

$$+w(v_{n-2}, v_{n-1}) - \lambda_t(v_{n-2}) + \lambda_t(v_{n-1}) + w(v_{n-1}, v_n) - \lambda_t(v_{n-1}) + \lambda_t(v_n)$$

$$= w(p) - \lambda_t(v_1) + \lambda_t(v_n)$$

where the last equality is because the previous summation is a telescoping series that cancels out most of the terms.

**Landmarks**

Landmarks are a common approach for defining potential functions. Choose a landmark $l$. For each $u \in V$, pre-compute $\delta(u, l)$. Then $\lambda_l^t(u) = \delta(u, l) - \delta(t, l)$ can be proved to be a feasible potential:
\[ w^*(u, v) = w(u, v) - \lambda_t^{(i)}(u) + \lambda_t^{(j)}(v) \]
\[ = w(u, v) - \delta(u, l) + \delta(t, l) + \delta(v, l) - \delta(t, l) \]
\[ = w(u, v) - \delta(u, l) + \delta(v, l) \geq 0 \text{ by the } \Delta \text{-inequality} \]

Sometimes we pick a small set of landmarks, and set \( \lambda_t(u) = \max_{l \in L} \lambda_t^{(i)}(u) \), which can also be proved to be feasible.

A* search is a modification to Dijkstra for solving the single-pair shortest-path problem. We use the potential function to change the ordering of the vertices in the priority queue so that vertices that look closer to the target are popped first from the queue.

To do this, the priority queue is keyed by

\[ f(u) = u.d + \lambda_t(u) - \lambda_t(s) \]

instead of just \( u.d \). That is, when extracting paths from the priority queue, we prefer paths that are both short and promising (close to the target).

**Heuristic function**

We can generalize the idea of adding an estimate to change the ordering of the priority queue. We define \( h(u) \) to be a heuristic function that estimates the remaining distance to the target. Then, the priority queue is keyed by

\[ f(u) = u.d + h(u). \]

In the previous case, we defined \( h(u) = \lambda_t(u) - \lambda_t(s) \), but there are many other heuristics that we could use.

A heuristic is admissible if \( h(u) \leq \delta(u, t) \).

**Proof of Correctness**

We can then prove that doing an A* search is equivalent to doing Dijkstra with a modified graph. Given a potential function \( \lambda_t(u) \), doing Dijkstra with the weights \( w^* \) is equivalent to doing an A* search using the heuristic function \( h(u) = \lambda_t(u) - \lambda_t(s) \). Assume our current \( u.d^* \) in Dijkstra was found using path \( (s, v_1), (v_1, v_2), ..., (v_{j-1}, v_j), (v_j, u) \). Then, we find that the summation becomes a telescoping series:
\[ u.d^* = w^*(s, v_1) + w^*(v_1, v_2) + \ldots + w^*(v_j, u) \]
\[ = w(s, v_1) - \lambda_t(s) + \lambda_t(v_1) + w(v_1, v_2) - \lambda_t(v_1) + \lambda_t(v_2) + \ldots \]
\[ + w(v_j, u) - \lambda_t(v_j) + \lambda_t(u) \]
\[ = u.d - \lambda_t(s) + \lambda_t(u) \]
\[ = u.d + h(u) \]

We proved earlier that all paths from \( s \) to \( t \) will be modified by the same amount if \textsc{Dijkstra} is done with this modified graph, so that means that the same shortest path will be found by an \( A^* \) search.