Dijkstra's Algorithm

The most used shortest paths algorithm for weighted directed graphs. It is sneakily the most intuitive, even though its implementation is more complicated than that of DAG-SP and BELLMAN-FORD. Imagine we drop a huge colony of ants onto the source vertex s and each goes along a single possible edge from s. Whenever an ant arrives at a vertex, it splits into several different ants each following a different edge going outwards.

Each ant walks at the same speed, so when an ant reaches a vertex for the first time, we know that it will have followed the shortest path to the vertex. We then mark a vertex the first time an ant arrives with the distance the ant travels, $\delta(s, u)$. Any other ant arriving at the same location can be obliterated, because we only really care about the first ant that arrives. This ensures that we do not check worthless paths.

To turn this into an actual algorithm, we maintain a frontier of visited vertices. Each of these vertices in the frontier will have the invariant that they have been visited by ants (their shortest path has been determined from s). We then pick the closest vertex to s and relax its edges, thereby expanding the frontier. From here, we get our familiar algorithm:

DIJKSTRA(G,W,S)

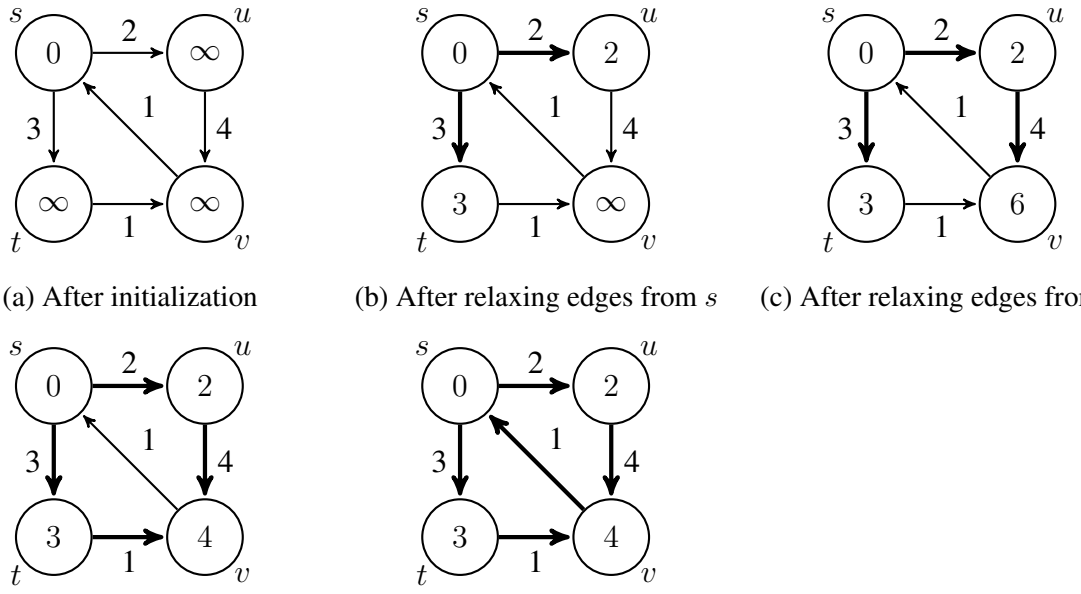
1 Initialize (G, s) $\triangleright$ uses priority queue Q
2 $S \leftarrow \phi$
3 $Q \leftarrow V[G]$ $\triangleright$ Insert into Q
4 **while** $Q \neq \phi$
5 **do** $u \leftarrow$ EXTRACT-MIN(Q) $\triangleright$ deletes u from Q
6 $S = S \cup \{u\}$
7 **for** each vertex $v \epsilon$ Adj$[u]$
8 **do** RELAX(u, v, w) $\leftarrow$ this is an implicit DECREASE_ KEY operation

In Figure 2, we see a sample execution of DIJKSTRA. Recall that the priority queue is used to store the shortest distances (seen so far) from the starting node s to each node (seen so far). Note that DECREASE_ KEY in Dijkstra's algorithm is used to change the shortest distance to a node if it is changed in the relaxation step.

Priority Queues

An important part of DIJKSTRA is the use of the priority queue, a data structure which allows us to extract the current minimum element very quickly, and also to decrease the value of any element in the data structure. This sounds awfully similar to heaps, which we learned about in this class. One of the biggest uses of heaps is actually for DIJKSTRA! (And other very similar graph algorithms).

In DIJKSTRA, we call EXTRACT-MIN once for every vertex in the graph, and for every vertex we relax its outgoing edges once, such that the total number of relaxations we do is $O(E)$. Thus,



(a) After initialization (b) After relaxing edges from s (c) After relaxing edges from u

(d) After relaxing edges from t (e) After relaxing edges from v

Figure 1: Sample execution of DIJKSTRA. The numbers in the nodes correspond to the current distance estimates, and bold edges indicate that that edge has been relaxed.

the overall runtime for DIJKSTRA is given by

$$O(V \cdot T_{\text{EXTRACT-MIN}} + E \cdot T_{\text{DECREASE-KEY}}).$$

We recall that the most common implementation of the heap is using an array, which gives us the following complexity:

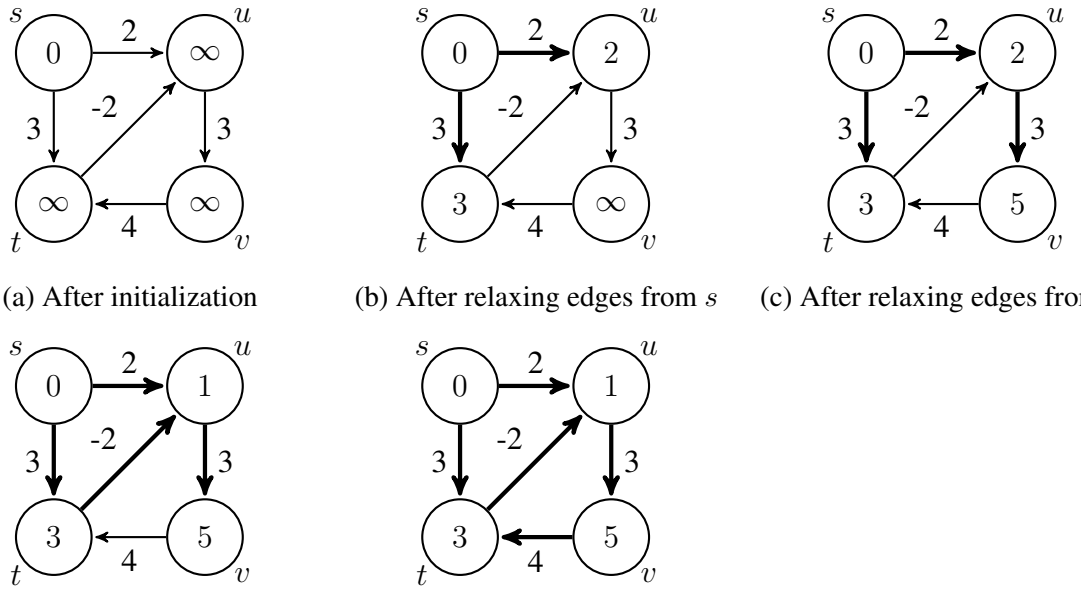
- EXTRACT-MIN: $\Theta(\log V)$
- DECREASE-KEY: $\Theta(\log V)$

The runtime for DIJKSTRA using a binary min heap would then be $O(V \log V + E \log V)$.

How could we speed this up? Well turns out, binary heaps are not the most efficient heaps. Fibonacci heaps (covered in 6.854) operate in $\Theta(\log V)$ for EXTRACT-MIN and $\Theta(1)$ for DECREASE-KEY, in amortized time. The runtime of DIJKSTRA will then be $O(V \log V + E)$. We know this is an optimal implementation of a priority queue because of the $\Omega(n \log n)$ sorting bound.

Negative Weight Edges

DIJKSTRA is only guaranteed to work on graphs with non-negative edge weights. If a graph contains a negative weight edge, then DIJKSTRA may produce an incorrect distance for one or



(a) After initialization (b) After relaxing edges from s (c) After relaxing edges from u

(d) After relaxing edges from t (e) After relaxing edges from v

Figure 2: Sample execution of DIJKSTRA on a graph with negative weight edges. The numbers in the nodes correspond to the current distance estimates, and bold edges indicate that that edge has been relaxed. We can see that after the algorithm has terminated, the distance to node v is incorrect (it should be 4).

more of the nodes. Note that the edge does not need to introduce a negative weight cycle in order to break DIJKSTRA. In Figure 3, we see a sample execution of DIJKSTRA on a graph with a single negative weight edge, but no negative weight cycles.

Intuitive Proof for Dijkstra's Algorithm Using BFS

We can provide a very broad, high-level intuitive proof for Dijkstra's Algorithm using BFS. Consider a graph $G(V, E)$ with positive edge lengths. We "expand" the graph by replacing every edge $(u, v) \in E$ with weight w with $w + 1$ vertices and path consisting of unit length edges through them. If we perform this transformation on all edges in E, then we obtain a new unweighted graph. Running BFS on this new graph simulates running Dijkstra's algorithm on the original graph. We can see that Dijkstra's algorithm "fast-forwards" the BFS to only those vertices that we care about. Thus, a vertex connected by a path of smaller weight to the starting vertex will be reached by the BFS on the "expanded" graph sooner than a vertex connected by a path of larger weight. Therefore, it makes sense to continue the BFS from that vertex first since it would be the first reached by BFS.

Note that this intuitive explanation does not account for non-integer edge weights. To see the

formal proof of Dijkstra's algorithm and its correctness, please refer to Chaper 24.3.

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:

```
Initialize()
Q ← V[G]
while Q ≠ φ
        do u ← EXTRACT_MIN(Q)
        if u == t: break
        for each vertex v ϵ Adj[u]
            do RELAX(u, v, w)
```

There are multiple ways to speed this up even further.

Bidirectional Search

Suppose we have a graph G in which the number of vertices distance d away from any given vertex is given by the function $f(d)$. For instance, for the Rubik's cube state graph, we may have $f(d) = c^d$ for some constant c. 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 μ 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 μ 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 μ any further, so stop.

Correctness Assume that there exists a path p from s to t with weight smaller than μ. 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, μ 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 μ.

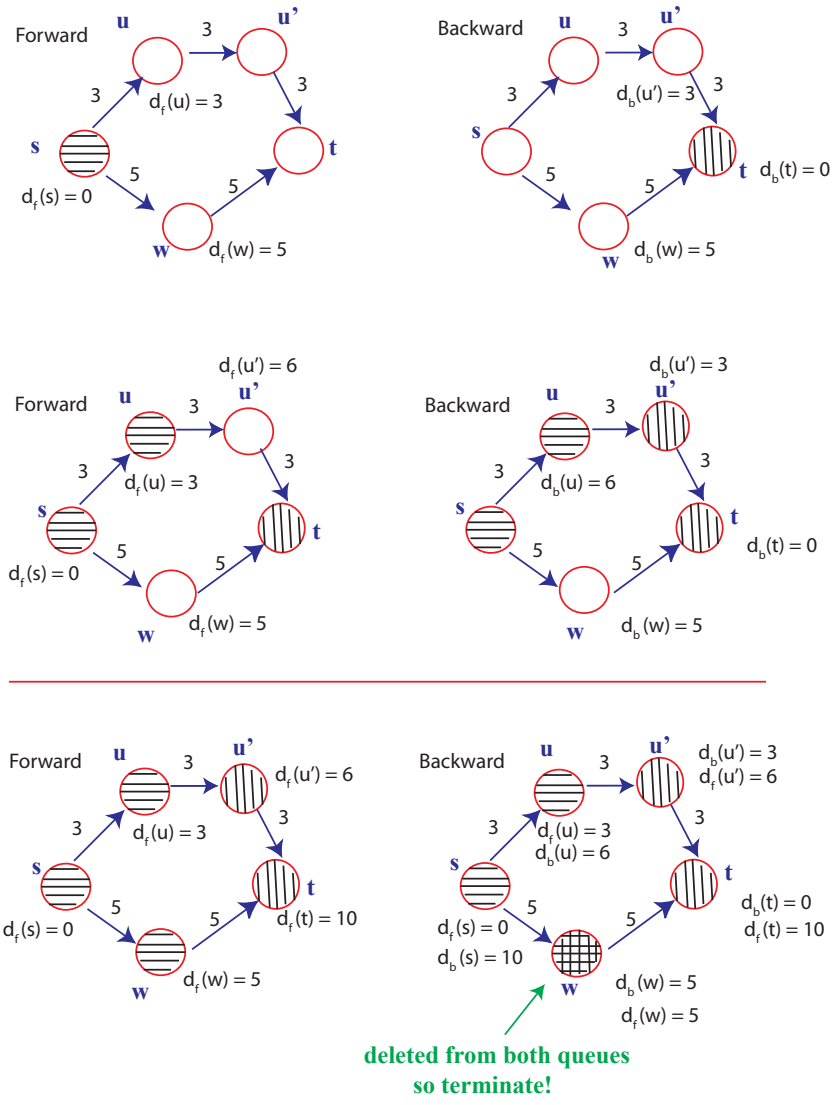


Figure 3: Forward and Backward Search and Termination.

Performance We define $f(r)$ to be the number of vertices within distance r of s or t. If we run normal Dijkstra, we would expect to visit $f(d)$ vertices. By contrast, with bidirectional Dijkstra, we could expect the forward and backward Dijkstras to meet somewhere in the middle, visiting $f(\frac{d}{2})$ vertices each for a total of $2 * f(\frac{d}{2})$ vertices.

For geometric data, where $f(r) = r^c$ for some c, we have that normal Dijkstra visits d^c vertices while bidirectional Dijkstra visits $2 * (\frac{d}{2})^c$ vertices, reducing the number of vertices visited by a factor of 2^{c-1}. For data like that found in games or puzzles, such as exploring the possible moves a knight can make on a chessboard, we may have $f(r) = c^r$, in which case bidirectional Dijkstra explores $2 * c^{\frac{d}{2}}$ vertices, for a square root speed-up over normal Dijkstra, which visits c^d vertices.

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 $\lambda_t^{(l)}(u) = \delta(u, l) - \delta(t, l)$ can be proved to be a feasible potential:

$$\begin{aligned} w^*(u,v) &= w(u,v) - \lambda_t^{(l)}(u) + \lambda_t^{(l)}(v) \\ &= w(u,v) - \delta(u,l) + \delta(t,l) + \delta(v,l) - \delta(t,l) \\ &= w(u,v) - \delta(u,l) + \delta(v,l) \geq 0 \quad \text{by the } \Delta \text{ -inequality} \end{aligned}$$

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

A*

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 $h(u) \leq \delta(u, t)$. A^* with an admissible heuristic is guaranteed to find the shortest path.

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:

$$\begin{aligned} u.d^* &= w^*(s, v_1) + w^*(v_1, v_2) + ... + w^*(v_j, u) \\ &= w(s, v_1) - \lambda_t(s) + \lambda_t(v_1) + w(v_1, v_2) - \lambda_t(v_1) + \lambda_t(v_2) + ... \\ &\quad + w(v_j, u) - \lambda_t(v_j) + \lambda_t(u) \\ &= u.d - \lambda_t(s) + \lambda_t(u) \\ &= u.d + h(u) \end{aligned}$$

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

Graph Transformation and Other Shortest Path Example Problems

Fall 2009 Quiz 2, Problem 5. Consider a road network modelled as a weighted undirected graph G with positive edge weights where edges represent roads connecting cities in G. However, some roads are known to be very rough, and while traversing from city s to t we never want to take a route that takes more than a single rough road. Assume a boolean attribute $r[e]$ for each edge r which indicates if e is rough or not. Give an efficient algorithm to compute the shortest distance between two cities s and t that doesn't traverse more than a single rough road. (Hint: Transform G and use a standard shortest-path algorithm as a black-box.)

Solution: For each vertex $v \in G$, construct a pair of vertices v_{smooth} and v_{rough}. For each smooth edge $e = (u, v)$, add a directed edge from u_{smooth} to v_{smooth} and a directed edge from u_{rough} to v_{rough}. For each rough edge $e = (u, v)$, add a directed edge from u_{smooth} to v_{rough}. Then generate all shortest paths rooted at s_{smooth}, and pick the shorter of $d[s_{\text{smooth}}, t_{\text{smooth}}]$ and $d[s_{\text{smooth}}, t_{\text{rough}}]$.

Spring 2011 Final, Problem 6. Consider a connected weighted directed graph $G = (V, E, w)$. Define the ***fatness*** of a path P to be the maximum weight of any edge in P. Give an efficient algorithm that, given such a graph and two vertices $u, v \in V$, finds the minimum possible fatness of a path from u to v in G.

Solution: To solve this problem, it is sufficient to update the standard RELAX method so that instead of summing edge weights, we take the maximum of the two edge weights. Then we can use Dijkstra's to compute shortest paths.