1 Running Time of Dijkstra

**General Running Time:** Initialization: $O(|V|)$, Main loop: Every vertex requires exactly one \texttt{Extract-Max} (we can skip the \texttt{Inserts} if we assume we insert all vertices when they have equal keys of $\infty$ and therefore can insert them in any order). Each edge can require up to one \texttt{Decrease-Key} and it is possible to come up with a case in which every edge does require a \texttt{Decrease-Key}. Therefore, in terms of these operations the running time is $O(|V|)T_{\text{Extract-Min}} + O(|E|)T_{\text{Decrease-Key}}$.

We would like \texttt{Extract-Min} and \texttt{Decrease-Key} to all be constant time... but as long as $T_{\text{Decrease-Key}} = o(|V|)$ and $T_{\text{Extract-Min}} = o(|E|)$, this is still probably better than Bellman-Ford!

<table>
<thead>
<tr>
<th>Data Structures</th>
<th>$T_{\text{Extract-Min}}$</th>
<th>$T_{\text{Decrease-Key}}$</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
</tr>
<tr>
<td>binary heap</td>
<td>$O(\log</td>
<td>V</td>
<td>)$</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>$O(\log</td>
<td>V</td>
<td>)$ amortized</td>
</tr>
</tbody>
</table>

**Fibonacci Heaps:** Are not part of 6.006. Chapter 20 of CLRS talks about them if you are interested.

2 Bounded Integer Edge Weights

Assume edge weights are non-negative integers bounded by $C$. Use an array of length $|V|C + 1$ for priority queue where vertices with paths of length $i$ are stored in bucket $i$.

Why does this work? Because all paths are less than $(|V| - 1)C \Rightarrow |V|C - C + 1 \leq |V|C + 1$ possible path values.

When you assign a path length to a previously unassigned node, you put it in its correct bucket. “$\infty$” lives in $|V|C$.

**Algorithms:** \texttt{Decrease-Key($v$, $k$)} just moves vertex $v$ from bucket $\text{key}[v]$ to bucket $k$. 

---

**Contents**

1 Running Time of Dijkstra 1

2 Bounded Integer Edge Weights 1

3 Single source, Single Target (SPP) 2

  3.1 Early Termination: ................................................................. 2

  3.2 Bi-directional Search: .......................................................... 2

  3.3 Potentials ................................................................. 4

---

1 Running Time of Dijkstra

**General Running Time:** Initialization: $O(|V|)$, Main loop: Every vertex requires exactly one \texttt{Extract-Max} (we can skip the \texttt{Inserts} if we assume we insert all vertices when they have equal keys of $\infty$ and therefore can insert them in any order). Each edge can require up to one \texttt{Decrease-Key} and it is possible to come up with a case in which every edge does require a \texttt{Decrease-Key}. Therefore, in terms of these operations the running time is $O(|V|)T_{\text{Extract-Min}} + O(|E|)T_{\text{Decrease-Key}}$.

We would like \texttt{Extract-Min} and \texttt{Decrease-Key} to all be constant time... but as long as $T_{\text{Decrease-Key}} = o(|V|)$ and $T_{\text{Extract-Min}} = o(|E|)$, this is still probably better than Bellman-Ford!

<table>
<thead>
<tr>
<th>Data Structures</th>
<th>$T_{\text{Extract-Min}}$</th>
<th>$T_{\text{Decrease-Key}}$</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
</tr>
<tr>
<td>binary heap</td>
<td>$O(\log</td>
<td>V</td>
<td>)$</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>$O(\log</td>
<td>V</td>
<td>)$ amortized</td>
</tr>
</tbody>
</table>

**Fibonacci Heaps:** Are not part of 6.006. Chapter 20 of CLRS talks about them if you are interested.
**Extract-Max**

1. global curr-bucket // Initialize to zero at start of whole algorithm
2. while (curr-bucket is empty)
3.    curr-bucket ← curr-bucket +1
4. return First value in curr-bucket

**Correctness of Extract-Max:** By induction. Do it yourself.

**Running Time of Decrease-Key and Extract-Min:** Clearly Decrease-Key is $O(1)$. In addition, Extract-Max amortizes to $O(1)$: over the course of the whole algorithm we see every bucket once. So, over the algorithm, the time of all calls to Extract-Max sums to $O(|V|C)$. We extract each vertex once and only once so we make $|V|$ calls to Extract-Max. Therefore, each call amortizes to $O(|V|/O(|V|) = O(1))$.

**Running Time:** $O(|V| + |E|)$ It’s linear!

**Circular Queues:** You can use an array of only length $C$ rather than length $|V|C + 1$ where path length $i$ is stored in bucket $i \mod C$. The correctness of this relies on the fact that the first vertex with path length $kC+i$ is not inserted into the queue until we have extracted the last vertex with path length $(k-1)C+i$. Just argue that since every edge weight is less than $C$, you cannot relax an edge with weight less than $(k-1)C+i$ and get an edge with weight $kC+i$. Do it formally yourself.

3 Single source, Single Target (SPP)

If you have a negative weight and cycles, you can’t do better than Bellman-Ford since you never know where a negative weight cycle might be lurking... You can’t stop at $t$ when you reach it.

So assume non-negative weights.

3.1 Early Termination:

We could stop Dijkstra when we hit $t$. In the worst case, this is no better, but it can help.

3.2 Bi-directional Search:

Search from $s$ and $t$ and hope you meet in the middle!

Alternate foward from $s$ with backwards from $t$. $d_f$ is forward distances, $d_b$ is backwards distance.

Termination: Vertex $w$ has been popped off queue in both forwards and backwards search. BUT $\delta(s,t) \neq d_f[w] + d_b[w]$. Rather $\delta(s,t) = \min_{u \in V} (d_f[u] + d_b[u])$.

**Lemma A:** $\delta(s,t) \leq \min_{u \in V} (d_f[u] + d_b[u])$
Proof: Let a shortest path from $s$ to $t$ be $(u_1, u_2), (u_2, u_3), \ldots, (u_{n-1}, u_n)$ where $u_1 = s$ and $u_n = t$. For all shortest paths, we must have at least one edge weight wrong.

$$
\min_{u \in V} (d_f[u] + d_b[u]) \geq \min_{u \in V} (\delta(s, u) + \delta(u, t))
$$

by the upper bound theorem

$$
\min_{u \in V} (\delta(s, u) + \delta(u, t) \geq \delta(s, t))
$$

by the triangle inequality.

Lemma B: Let $v$ be the vertex that has been popped off the queue in both the forward and backwards searches. Let $p = (u_1, u_2), \ldots, (u_{n-1}, u_n)$ be a shortest path from $s$ to $t$ ($u_1 = s$ and $u_n = t$). Let $u_f$ be the vertex on this path with the largest $d_f$ value that has come out of the forward queue and let $u_b$ be the vertex on this path with the largest $d_b$ value that has come out of the backwards queue. Then either $u_f = u_b$ or $u_f = u_{b-1}$.

Proof: By the properties of Dijkstra, we must have that $d_f[u_f] = \delta(s, u_f)$ and $d_b[u_b] = \delta(u_b, t)$. Moreover, since $u_f$ is the largest vertex to have come out of the forwards queue and $u_b$ is the largest vertex to come out of the backwards queue and $p$ is a shortest path, the following must hold:

1. $\delta(s, u_f) \leq \delta(s, v)$
2. $\delta(s, u_f) + w(u_f, u_{f+1}) \geq \delta(s, v)$
3. $\delta(u_b, t) \leq \delta(v, t)$
4. $w(u_{b-1}, u_b) + \delta(u_b, t) \geq \delta(v, t)$
5. $\delta(s, u_f) + \delta(u_b, t) + \sum_{i=f}^{b-1} w(u_i, u_{i+1}) \leq \delta(s, v) + \delta(v, t)$

Combining equations 2 and 4

$$
\delta(s, u_f) + \delta(u_b, t) + w(u_f, u_{f+1}) + w(u_{b-1}, u_b) \geq \delta(s, v) + \delta(v, t).
$$

In order that equation 5 hold, therefore

$$
\sum_{i=f}^{b-1} w(u_i, u_{i+1}) \leq w(u_f, u_{f+1}) + w(u_{b-1}, u_b)
$$

and we must have either $u_f = u_b$ or $u_f = u_{b-1}$.

Correctness: When a vertex has been popped off the queue of both the forwards and the backwards search, $\delta(s, t) = \min_{u \in V} (d_f[u] + d_b[u])$.

Proof: We show that there is some vertex $q$ such that $\delta(s, t) = d_f[q] + d_b[q]$. The correctness then follows from Lemma A.

Let $p = (u_1, u_2), \ldots, (u_{n-1}, u_n)$ be a shortest path from $s$ to $t$. Let $u_f$ be the vertex on this path with the largest $d_f$ that has been popped off the forward queue and let $u_b$ be vertex on this path with the largest $d_b$ that has been popped off the queue. Then, by the proof of Dijkstra’s algorithm, we have relaxed $(u_1, u_2), \ldots, (u_{f-1}, u_f), (u_f, u_{f+1})$ in order (we did the last relaxation when we pushed $u_f$ onto the queue). Similarly, we have relaxed $(u_n, u_{n-1}), \ldots, (u_b, u_{b-1})$ in order (in the backwards search). By Lemma B,
$u_f = u_b$ or $u_f = u_{b-1}$. In the first case, consider vertex $u_f$. We have by the path relaxation property that $d_f[u_f] = \delta(s, u_f)$ and $d_b[u_f] = \delta(u_f, t)$. Since $p$ is a shortest path, $\delta(s, t) = \delta(s, u_f) + \delta(u_f, t) = d_f[u_f] + d_b[u_f]$. Similarly, in the second case, consider the vertex $u_{f+1}$. By the path relaxation property, $d_f[u_{f+1}] = \delta(s, u_{f+1})$, $d_b[u_{f+1}] = \delta(u_b, t)$ and $\delta(s, t) = \delta(s, u_{f+1}) + \delta(u_b, t) = d_f[u_{f+1}] + d_b[u_b]$.

3.3 Potentials

Potential function is function of target $t$ and vertex $v$, denoted $\lambda_t(v)$.

We modify weights using this potential:

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

**Feasibility:** $\forall u, v \in V$, $w(u, v) - \lambda_t(u) + \lambda_t(v) \geq 0$. So we can still use Dijkstra!

**Landmarks:** Choose a landmark $l$. For each $u \in V$, pre-compute $\delta(u, l)$. Then

$$\lambda_t^{(l)}(u) = \delta(u, l) - \delta(t, l) \quad (6)$$

is a feasible potential. This is basically just saying, a good estimate is that the route from $u$ to $l$ probably gets near $t$.

**Proof of Feasibility:** Show that

$$w^*(u, v) = w(u, v) - \lambda_t^{(l)}(u) + \lambda_t^{(l)}(v) \geq 0$$

You’ll do this on problem set 5.

You can also do this with a whole set of potentials $L$. For each $l \in L$ and $u \in V$, compute $\lambda_t^{(l)}(u)$. Then you can use the potential function

$$\lambda_t(u) = \max_{l \in L} \lambda_t^{(l)}(u)$$

or once you know the target, you can pick a landmark close to the target. Again, you’ll prove the feasibility of this in problem set 5.

**Euclidean distance:** Assume we are working with a graph set on a Euclidean plane and the weight of an edge, $w(u, v)$, is somehow related to the Euclidean distance between the vertices, $l(u, v)$. This is the case, for example, when $u$ and $v$ lie on a map and the weight is the driving distance between them. The absolute shortest possible distance from $u$ to $v$ is $l(u, v)$ (“as the crow flies”) but being constrained to stay on a road may make the distance longer. However, it is unlikely that the driving distance between two points two miles apart will be greater than the driving distance between two points 50 miles apart so the weights are related to the Euclidean distance.

In this case, a feasible potential is

$$\lambda_t(u) = \frac{l(u, t)}{v_{\text{max}}}$$

where

$$v_{\text{max}} = \max_{u, v} \left( \frac{l(u, v)}{w(u, v)} \right)$$.
Proof of Feasibility:

\[ w^*(u, v) = w(u, v) - \lambda_t(u) + \lambda_t(v) \]
\[ = w(u, v) - \frac{l(u, t) - l(v, t)}{v_{\text{max}}} \]
\[ \geq w(u, v) - \frac{l(u, v)}{v_{\text{max}}} \]
\[ = w(u, v) - l(u, v) \min_{q, r \in V} \left( \frac{w(q, r)}{l(q, r)} \right) \]
\[ \geq w(u, v) - l(u, v) \left( \frac{w(u, v)}{l(u, v)} \right) \]
\[ = 0 \]

Note that the triangle inequality gives \( l(t, v) \geq l(t, u) + l(u, v) \). Rearranging shows \( l(u, t) - l(v, t) \leq l(u, v) \), which we have used.

Connection to A* Search: In A* search, heap is ordered according to function

\[ f(u) = g(u) + h(u) \]

\( g(u) = d[u] \), the current distance from the starting vertex \( s \) to \( n \). \( h(u) \) is a heuristic function giving an estimate of the distance from \( u \) to \( t \) (note that \( h(u) = 0 \) gives Dijkstra’s algorithm). A heuristic is admissible if \( h(u) \leq \delta(u, t) \). A search with an admissible heuristic is guaranteed to find the shortest path.

Given a potential function \( \lambda_t(u) \), doing Dijkstra’s algorithm 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 \( d^*[u] \) (distance using starred weights) in Dijkstra’s algorithm was found using path \( (v_1, v_2), \ldots, (v_{n-1}, v_n) \):

\[ d^*[u] = \sum_{i=1}^{n-1} w^*(u_i, u_{i+1}) \]
\[ = \sum_{i=1}^{n-1} w(u_i, u_{i+1}) - \sum_{i=1}^{n-1} \lambda_t(u_i) + \sum_{j=2}^{n} \lambda_t(u_j) \]
\[ = d[u] - \lambda_t(s) + \lambda_t(u) \]
\[ = g(u) + h(u) \quad (7) \]

A very similar calculation shows that if \( \lambda_t \) is a feasible potential then the corresponding heuristic function is admissible if \( \lambda_t(t) \geq 0 \).