Greedy Algorithms: Dijkstra’s Shortest Path Algorithm

Let $G(V, E, w)$ be an edge weighted graph, where $w : E \rightarrow \mathbb{R}^+$. Let $s, t$ be two vertices in $G$ (think of $s$ as a source, $t$ as a terminal), and suppose you were asked to compute a shortest (i.e. cheapest) path between $s$ and $t$. Notice that $G$ could possibly have more than one shortest path between $s$ and $t$. Consider the graph below for instance: Both $P = s, a, t$ and $Q = s, b, t$ are cheapest paths from $s$ to $t$.

One way to solve this problem is to compute all $st$ paths in $G$, and choose the cheapest. Notice that if the graph is unweighted (think of this as all the edges having equal weights), then starting BFS from $s$ would solve this problem. Maybe we can adjust BFS to take into account the weights.

Recall in BFS vertices are pushed into a queue when visited. When visiting the neighbours of a vertex $u$, the ordering in which the neighbours enter the queue is arbitrary. If now the goal is to compute the cheapest path, then one way to modify BFS would be to push the cheapest neighbours first. By cheapest, we mean with shortest distance.

“Modified BFS”: Consider using a priority queue instead of a queue to collect unvisited vertices. Set the priority to be the shortest distance so far. This is precisely the idea behind Dijkstra’s algorithm.

Example: Consider the graph below for instance. Suppose we want to compute the cheapest path from $s = A$ to $t = F$.

The table below keeps track of the distances computed at every iteration from the source $A$ to the every vertex in the graph. Read the table as follows: In the fist iteration, the distance from the source to itself is 0, and $\infty$ to any other vertex. At every iteration, choose the cheapest available vertex and try to build the next cheapest path from said vertex. Repeat the process until all vertices have been visited.
<table>
<thead>
<tr>
<th>i</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>$S_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>$\infty$</td>
<td>7</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>14</td>
<td>7</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>7</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>7</td>
<td>$\infty$</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>7</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>7</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>7</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Notice from the table above that at every iteration $i$, the set $S_i$ is the set of edges that could potentially lead to a shortest path. If $e = (u, v)$ is added to $S_i$, with $u \in S_i, v \notin S_i$, then the current value of $d[v]$ is a shortest $sv$ path. Formally the algorithm is as follows:

Algorithm 1 Dijkstra’s Shortest Path Algorithm

**Input:** An edge weighted connected graph $G(V, E, w)$ where $w : E \rightarrow \mathbb{R}^+$ and two vertices $s, t$.

**Output:** A path from $s$ to $t$ with minimum total cost (shortest path)

1. $S = \emptyset$.
2. Initialize empty priority queue.
3. foreach $v \in V$ do
   4. $p[v] = \text{NIL}$ \hspace{1cm} $\triangleright$ predecessor of $v$ in shortest $s, v$ path so far
   5. $d[v] = \infty$ \hspace{1cm} $\triangleright$ priority of $v = \text{min distance } s, v \text{ so far.}$
   6. enqueue($v$) \hspace{1cm} $\triangleright$ With priority $d[v] = \infty$.
4. end for
5. $d[s] = 0$
6. Update queue order of $s$
7. while queue is not empty do \hspace{1cm} $\triangleright$ Main Loop
8. $v = \text{dequeue element with min priority } d[]$
9. if $p[v] = \text{NIL}$ then
10. $S = S \cup \{(p[v], v)\}$
11. end if
12. foreach edge $(u, v)$ do
13. if $u$ is in the queue and $d[v] + w(v, u) < d[u]$ then
14. $p[u] = v$
15. $d[u] = d[v] + w(v, u)$
16. Update queue order of $u$
17. end if
18. end for
19. end while
20. return $S$

**Proof of Correctness:**

We will argue on the $S_i$’s. In particular, at every iteration, we generate subsets of edges $S_1, S_2, \ldots, S_n$. We say that $S_i$ can be extended to some collection of shortest paths $S_i^*$ (this is just a tree of shortest paths) using only edges that do not have both endpoints in $S_i$. That is, we only add edges to $S_i$ with at least one endpoint in the queue.

**Loop Invariant**

$$S_i \text{ is promising, and } \forall u \in S_i, \forall v \notin S_i : d[u] = \text{dist}[s, v] \leq \text{dist}[s, u] \leq d[v]$$
Where \( \text{dist}[s, u] \) is the minimum cost of all paths from \( s \) to \( u \).

**Proof.** By induction on \( i \).

**Base Case:** \( S_0 = \emptyset \) is trivially promising.

**Induction Hypothesis:** For some \( i \), suppose \( S_i \) can be extended to some shortest paths tree \( S_i^* \), using only edges without both endpoints in \( S_i \), and that:

\[
d[u] = \text{dist}[s, u] \leq \text{dist}[s, v] \leq d[v], \text{ for all } u \in S_i, v \notin S_i
\]

**Induction Step:** Consider \( S_{i+1} = S_i \cup \{(u, v)\} \) with \( u \in S_i, v \) outside \( S_i \). We have 2 possible cases:

1. \( (u, v) \in S_i^* \):
   - If \( (u, v) \in S_i^* \) then \( S_i^* \) extends \( S_{i+1} \) and \( \text{dist}[s, v] = \text{dist}[s, u] + w(u, v) \) since \( (u, v) \in S_i^* \). Thus we have:
     \[
     d[v] = d[u] + w(u, v) = \text{dist}[s, u] + w(u, v) = \text{dist}[s, v]
     \]
   - Moreover, since \( d[v] \) was the cheapest of all \( d[\cdot] \) values for vertices outside \( S_i \), it follows that \( \forall x \in S_{i+1} \) and \( \forall y \notin S_{i+1} : d[x] = \text{dist}[s, x] \leq d[s, y] \leq d[y] \).

2. \( (u, v) \notin S_i^* \):
   - Then consider the path \( P \) in \( S_i^* \) from \( s \) to \( v \). \( P \) is optimal. Let \( (z, v) \) be the last edge on this path.
   - **If \( z \) were outside \( S_i \), then let \( (x, y) \) be the first edge on \( P \) with \( x \in S_i, y \notin S_i \) (why does such an edge exist?). We have \( d[y] \leq d[x] + w(x, y) \) (because \( d[y] \) is the smallest value of \( d[t] + w(t, y), \forall e = (t, y), t \in S_i \).) We thus have:
     \[
     d[y] \leq d[x] + w(x, y) = \text{dist}[s, x] + w(x, y) < \text{dist}[s, x] + w(x, y) + \text{dist}[y, v] \quad \text{(Since } w : E \rightarrow \mathbb{R}^+) \quad \text{(Since } w : E \rightarrow \mathbb{R}^+) \quad \text{(Since } w : E \rightarrow \mathbb{R}^+) \quad \text{(Since } w : E \rightarrow \mathbb{R}^+)
     \]
     \[
     = \text{dist}[s, v] \leq d[v] \quad \text{(Since } w : E \rightarrow \mathbb{R}^+) \quad \text{(Since } w : E \rightarrow \mathbb{R}^+) \quad \text{(Since } w : E \rightarrow \mathbb{R}^+) \quad \text{(Since } w : E \rightarrow \mathbb{R}^+)
     \]
   - Thus we have:
     \[
     d[y] < d[v]
     \]
   - But this contradicts the fact that \( d[v] \) is the smallest \( d[\cdot] \) value for all vertices outside \( S_i \).
   - Therefore \( z \) must be in \( S_i \). This implies that \( \text{dist}[s, v] = \text{dist}[s, z] + w(z, v) = d[z] + w(z, v) \).
   - Since \( d[v] \) is the minimum of all \( d[x] + w(x, v) \forall x \in S_i \), it follows that:
     \[
     d[v] \leq d[z] + w(z, v) = \text{dist}[s, v] \quad \text{(i.e. it is the shortest distance already.)}
     \]
   - So we can let \( S_{i+1}^* = S_i^* \setminus \{(z, v)\} \cup \{(u, v)\} \) and after the update to \( d[a] \) for all \( e = (v, a) \) with \( a \notin S_i \), we still have that \( d[x] = \text{dist}[s, x] \leq \text{dist}[s, y] \leq d[y] \), for all \( x \in S_i, y \notin S_i \).

Hence the loop invariant holds. When the algorithm terminates, we have: \( d[u] = \text{dist}[s, u], \forall u \in V \).