Dijkstra’s Shortest-Path Algorithm

Singe-source shortest path Problem
Given a weighted graph $G = (V, E, W)$ and a source vertex $s$, find a shortest path from $s$ to each vertex $v$. 
Growing a shortest-path tree

- Start at source vertex $s$ and “branches out” by selecting edges that lead to new vertices
- For each vertex $z$ in the fringe, there is at least one tree vertex $v$ such that $vz$ is an edge of $G$. why?
  (otherwise how can $z$ be in the fringe)
- Choose $v$ such that $d(s,v) + W(vz)$ is minimized

**Theorem 8.6** Let $G = (V,E,W)$. $V'$ is a subset of $V$, and $V'$ contains the source $s$. Let $d(s,y)$ be the shortest distance in $G$ from $s$ to $y$, for each $y$ in $V'$. If edge $yz$ is chosen to minimize $d(s,y) + W(yz)$ over all edges with $y$ in $V'$ and $z$ in $V-V'$, then the path consisting of a shortest path from $s$ to $y$ followed by the edge $yz$ is a shortest path from $s$ to $z$ in $V$.

**Proof:** For any other path $P'$ from $s$ to $z$, we have

$$W(P) = d(s,y) + W(yz) \leq d(s,r) + W(rt) \leq W(P')$$

That is why vertex $z$ is chosen by the algorithm to expand $V'$. 
Dijkstra(G,w,s)
1. Initialize all vertices as unseen
2. Start the tree with the specified source vertex s; reclassify it as tree
3. Define $d(s,s) = 0$
4. Reclassify all vertices adjacent to s as fringe
5. while there are fringe vertices
6. select a tree vertex t and a fringe vertex v
    such that $d(s,t) + W(tv)$ is minimum
7. reclassify v as tree; add edge tv to the tree
8. define $d(s,v) = d(s,t) + W(tv)$
9. reclassify all unseen vertices adjacent to v as fringe

---

$G$

**Fringe**

The tree so far

d(A,B) + W(BC) = 6

d(A,A) + W(AG) = 5

d(A,A) + W(AF) = 9

d(A,C) + W(CD) = 8

d(A,A) + W(AF) = 9

d(A,G) + W(GI) = 7

d(A,G) + W(GH) = 10
Dijkstra Algorithm

Graph G, weights w, source s

Dijkstra(G,w,s)
1. for each v ∈ V[G]
2. d[v] ← ∞
3. P[v] ← nil
4. d[s] ← 0
5. S ← empty
6. Q ← V[G]
7. while Q is not empty
8. u ← Extract-Min(Q)
9. S ← S ∪ \{u\}
10. for each v ∈ Adj[u]
11. if d[v] > d[u] + w(u,v)
12. then d[v] ← d[u] + w(u,v)
13. P[v] ← u
Dijkstra Algorithm

Time analysis
- Initialization of priority queue (as a binary heap): $O(V)$
- Extract-min is called $|V|$ times
  - Each Extract-Min takes $O(\lg V)$ time
  - For each adjacent vertex, update its distance (with Decrease-Key operation: $O(\lg v)$)

Total running time: $O((V+E)\lg(V))$

---

Dijkstra Algorithm

Running time with different Q implementations
- Array
  - Extract-Min: $O(V)$
  - Decrease-Key: $O(1)$
  - Total: $O(V^2)$
- Binary heap
  - Extract-Min: $O(\lg V)$
  - Decrease-Key: $O(\lg V)$
  - Total: $O(E \lg V)$
- Fibonacci heap
  - Extract-Min: $O(\lg V)$
  - Decrease-Key (amortized): $O(1)$
  - Total: $O(V \lg V + E)$
For all pairs in a graph,
- Is there a path from \( u \) to \( v \)?
- What is the shortest path from \( u \) to \( v \)?

(F is the weight matrix of graph \( G = (V,E,W) \).

\[
W_{ij} = W(v_iv_j)
\]
\[
= \infty \quad \text{if } v_iv_j \notin E \text{ and } i \neq j
\]
\[
= \min(0, W(v_iv_j)) \quad \text{if } v_iv_i \in E
\]
\[
= 0 \quad \text{if } v_iv_i \notin E
\]

D with entry \( d_{ij} \) = the shortest-path distance from \( v_i \) to \( v_j \).

**Lemma** If a shortest path from \( v_i \) to \( v_j \) goes through an intermediate vertex \( v_k \), then the segments of that path from \( v_i \) to \( v_k \) and from \( v_k \) to \( v_j \) are themselves shortest paths.

**Floyd algorithm**

W is the weight matrix of graph \( G = (V,E,W) \).
Path $p$ is the shortest path from $i$ to $j$. Path $p_1$, the portion of path $p$ from $i$ to $k$, is the shortest path from $i$ to $k$. Suppose $p'_1$ is the shortest path from $i$ to $k$, then $p'_1$ and $p_2$ would make a path with shorter distance than path $p$.

Let $k$ be the highest-indexed intermediate vertex in path $p$.

The highest-indexed intermediate vertex in path $p_1$ must have an index lower than $k$, that is, all the intermediate vertices of $p_1$ are from subset $\{v_1, \ldots, v_{k-1}\}$. The same holds for path $p_2$.

Let $d_{ij}^{(k)}$ be the weight of a shortest path from $i$ to $j$ with all intermediate vertices in the set $\{v_1, \ldots, v_k\}$. We shall have

$$d_{ij}^{(k)} = \begin{cases} 
\min(d_{ij}^{(k-1)} + d_{kj}^{(k-1)}) & \text{for } k \geq 1 \\
= w_{ij} & \text{for } k = 0
\end{cases}$$

Floyd-APSP($W$, $D$)

1. $D \leftarrow W$
2. for $k \leftarrow 1$ to $n$
3. for $i \leftarrow 1$ to $n$
4. for $j \leftarrow 1$ to $n$
5. $D[i][j] \leftarrow \min(D[i][j], D[j][k] + D[k][j])$
6. return $D$

Time analysis:

- $\Theta(n^3)$
Transitive closure of a binary relation

- Binary relation on a set $S$ is a subset of $S \times S$. If $(s_i, s_j) \in A$, we say $s_i$ is $A$-related to $s_j$ and use notation $s_i A s_j$.
- Transitive closure of $A$ is a binary relation, denoted as $R$, such that, $s_i R s_j$ if and only if there is a path from $s_i$ to $s_j$ in graph $G=(S,A)$.
- Transitive closure is also called reachability relation. $R$ matrix is a $n \times n$ matrix
  
  $$r_{ij} = \begin{cases} 
  1 & \text{if there is a path from } s_i \text{ to } s_j \\
  0 & \text{otherwise}
  \end{cases}$$
- Find a connected component of a graph
  - Undirected graph
    - Corresponds to a depth-first search tree
  - Directed graph
    - A depth-first search tree does not guarantee to give a connected component
- A connected component ≠ Transitive closure
  - For each vertex $u$, do DPS starting $u$, update $r_{uv}$ for all reachable vertex $v$. Running time: $O(V^2 + VE)$

---

### Warshall Algorithm

- Define $t^{(k)}_{ij} = 1$ if there exists a path in graph $G$ from vertex $i$ to vertex $j$ with all intermediate vertices in the set $\{1, 2, \ldots, k\}$, and 0 otherwise.

  \[
  t^{(0)}_{ij} = \begin{cases} 
  1 & \text{if } i = j \text{ or } (i, j) \in E \\
  0 & \text{if } i \neq j \text{ and } (i, j) \notin E 
  \end{cases} 
  \]

  \[
  t^{(k)}_{ij} = t^{(k-1)}_{ij} \lor (t^{(k-1)}_{ik} \land t^{(k-1)}_{kj}) 
  \]
Transitive-Closure(G)
1. for i ← 1 to n
2. for j ← 1 to n
3. if i=j or (i,j) ∈ E[G]
   then t^(0)_(ij) ← 1
   else t^(0)_(ij) ← 0
4. for k ← 1 to n
   for i ← 1 to n
   for j ← 1 to n
   t^(k)_(ij) ← t^(k-1)_(ij) ∨ ( t^(k-1)_(ik) ∧ t^(k-1)_(kj) )
9. return T^(n)