Single-Source Shortest Paths (Ch. 24)

The single-source shortest path problem (SSSP)

**input:** a graph $G = (V, E)$ with edge weights, and a specific source node $s$.

**goal:** find a minimum weight (shortest) path from $s$ to every other node in $V$ (sometimes we only want the cost of these paths, sometimes we want the path itself because path determines efficient reachability). Notice here we are concerned about the entire path length.

Weights in SSSP algorithms can include distances, times, hops, cost, etc. These algorithms are used in many routing applications.

Note: BFS finds the shortest paths for the special case when all edge weights are 1. Running time = $O(V + E)$

The result of a SSSP algorithm can be viewed as a tree rooted at $s$, containing a shortest (wrt weight) path from $s$ to all other nodes.

Negative-weight edges

Some graphs that are inputs to an SSSP algorithm may contain negative edge weights.

How might such edges occur?

Suppose vertices in $G$ represent cities and the weights of edges in $G$ represent how much money it costs to go from one city to another. If someone is willing to pay us to go from, say JFK to ORD, then the "cost" of the edge (JFK, ORD) would be negative.

We will see an algorithm that is correct if negative-weight edges can occur and one that is not correct in this situation.

Single-Source Shortest Path Algorithms

Chapter 24 presents 3 different SSSP algorithms, we will study two of them:

1. Dijkstra’s algorithm: Finds SSSPs in weighted, directed graph when all edge weights are non-negative.
2. Bellman-Ford Algorithm: General case in which edge weights may be negative. Detects negative-weight cycles.
**Procedures used by SSSP Algorithms**

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>for each vertex ( v \in G )</td>
</tr>
<tr>
<td>2.</td>
<td>( d[v] = \infty )</td>
</tr>
<tr>
<td>3.</td>
<td>( \pi[v] = \text{NIL} )</td>
</tr>
<tr>
<td>4.</td>
<td>( d[s] = 0 )</td>
</tr>
</tbody>
</table>

Attribute \( d[v] \) is an upper bound on the weight of the shortest path from source \( s \) to \( v \), or a "shortest path estimate".

**Relax** \((u,v,w)\)

1. if \( d[v] > d[u] + w(u,v) \)
2. \( d[v] = d[u] + w(u,v) \)
3. \( \pi[v] = u \)

Relax procedure lowers shortest path estimates and changes predecessors. Decreases the value of a key.

**Dijkstra’s SSSP Algorithm**

The labeling and mechanics of Dijkstra's algorithm are like Prim's.

Both construct an expanding subtree of vertices by selecting the next vertex from the priority queue of the remaining vertices.

However, the two algorithms solve different problems.

Dijkstra's algorithm compares entire, multi-hop path lengths and therefore must incrementally add edge weights, while Prim's algorithm compares the edge weights of only 1-hop paths, with no summation along a path.

**Running Time of Dijkstra’s SSSP Alg**

**Algorithm SSSP-Dijkstra (G, s)**

1. Initialize-Single-Source(G,s)
2. \( S = \emptyset \)
3. \( Q = V[G] \) // all nodes enqueued
4. while \( Q \neq \emptyset \)
5. \( u = \text{Extract-Min}(Q) \)
6. \( S = S \cup \{u\} \)
7. for each vertex \( v \in \text{Adj}[u] \)
8. \( \text{Relax}(u,v,w) \)

**Steps 1-3:**

**Steps 4-8:**

Steps 7-8: E iterations overall
(looks at each edge once)
Suppose \( \text{Relax} \) takes \( O(Y_v) \) time.

Total: \( O(VX^2 + EY) \)
Running Time of Dijkstra’s SSSP Alg

If G is dense (i.e., $O(V^2)$ edges):
Asymptotically speaking, there is no point in being clever about Extract-Min. Store each $d[v]$ in the $v$th entry of an array. Each insert and decrease-key takes $O(1)$ time. Extract-Min takes $O(V)$ time (why?)
Total time: $O(V^2 + E) = O(V^2)$

If G is sparse (i.e., $o(V^2)$ edges):
Try to minimize Extract-Min, use binary heap (heapsort) so time for Extract-Min & Decrease-Key = $O(lgV)$
Total time: $O(VlgV + ElgV) = O((V + E) lgV)$

Correctness of Dijkstra’s SSSP Alg

Theorem: Dijkstra’s algorithm finds shortest paths from a designated source vertex to all other vertices in G.

Proof Idea: The proof argues that the shortest path to a node must include only nodes that have already been extracted from Q and added to the set $S$. It uses contradiction and the fact that there are no negative edge weights allowed to show that, if there is a shorter path to a particular node $v$ that does not include all nodes in $S$, then other vertices along that path would have been extracted from Q (and added to S) before $v$.

Bellman-Ford SSSP Algorithm

- Computes single-source shortest paths even when some edges have negative weight.
- Can detect if there are any negative-weight cycles in the graph.

The algorithm has 2 parts:
Part 1: Computing shortest paths tree:
- $|V| - 1$ iterations.
- Iteration $i$ computes the shortest path from $s$ using paths of up to $i$ edges.
Part 2: Checking for negative-weight cycles.
Bellman-Ford SSSP Algorithm

```
Bellman-Ford (G, w, s)
// Part I – find shortest paths
1. Initialize-Single-Source(G,s)
2. for i = 1 to |V| - 1
3. for each edge (u, v) ∈ E
4. Relax(u,v,w)
// Part II – check for negative weight cycles
5. for each edge (u, v) ∈ E
6. if d[v] > d[u] + wt(u, v)
7. return false
8. return true
```

Boolean value returned is false if there is a negative weight cycle in G and true otherwise.

Correctness of Bellman-Ford Algorithm

Let \( \delta(s,v) \) be the actual shortest path distance from \( s \) to \( v \)

• Theorem
  Suppose there are no negative-weight cycles in G. After \(|V| - 1\) iterations of the for loop, \( d[v] = \delta(s,v) \) for all vertices \( v \) that are reachable from \( s \).

  • Proof:
    - If \( v \) is reachable from \( s \), then there is a path from \( s \) to \( v \)
    - Say \( s = u_0, u_1, u_2, ..., u_k = v \), where \( k < |V| \).
    - There are \( k \) edges in this path.
    - After the first iteration, \( u_0, u_1 \) is a shortest path; after the second pass, \( u_0, u_1, u_2 \) is a shortest path; after \( k \) passes, \( u_0, u_1, u_2, ..., u_k \) is a shortest path. \( \blacksquare \)

Complexity of Bellman-Ford Algorithm

- Initialization = \( O(|V|) \)
- decrease-key (in Relax) is called \( (|V| - 1) \cdot |E| \) times
- Test for negative-weight cycle = \( O(|E|) \)

- Total: \( O(|V||E|) \) -- so more expensive than Dijkstra's, but also more general, since it works for graphs with negative edge weights.