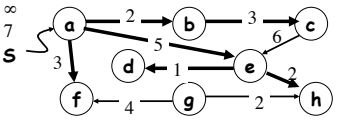
Single-Source Shortest Paths (Ch. 24)

The single-source shortest path problem (SSSP)
input: a digraph G = (V, E) with edge weights, and a specific *source node s.*
goal: find a minimum weight (shortest) cumulative path from s to every other node in V

s → b = 2
s → c = 5
s → d = 6
s → e = 5
s → f = 3
s → g = ∞
s → h = 7



Single-Source Shortest Paths

Weights in SSSP algorithms can include distances, time spans, hops, length of wire, 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)

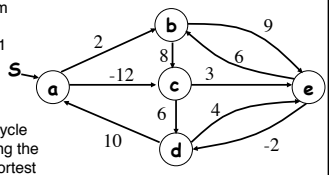
The result of a SSSP algorithm can be viewed as a tree rooted at s, containing a shortest (wrt weight) path from s to all other nodes.

Negative-weight cycles

Some graphs may have negative-weight cycles and these are a problem for some SSSP algorithms.

What is the shortest path from a to d and what is the cost?
• path a-c-e-d = -12+3-2 = -11
• path a-c-e-d-a-c-e-d = -12+3-2+(10-12+3-2)= -13

If we keep going around the cycle (d-a-c-e-d), we keep shortening the weight of the path. So the shortest path has weight -∞



To avoid this problem, some algorithms require that the graph has no negative weight cycles, otherwise the solution does not exist.

Single-Source Shortest Paths

Question: Can a shortest path contain a positive-weight cycle?

No.

Suppose we have a shortest path $p = \langle v_0, v_1, ..., v_k \rangle$ and $c = \langle v_i, v_{i+1}, ..., v_j \rangle$ is a positive-weight cycle on p so that $v_i = v_j$ and $w(c) > 0$. Then the path (obtained from splicing out c, $p' = \langle v_0, v_1, ..., v_i, v_{j+1}, v_{j+2}, ...v_k \rangle$) has $w(p') = w(p) - w(c) < w(p)$. So p can't be a shortest path.

Therefore, we can assume, wlog, that shortest paths have no cycles.

Single-Source Shortest Path Algorithms

Chapter 24 presents 3 different SSSP algorithms:

- **Bellman-Ford Algorithm: General case in which edge weights may be negative. Detects negative-weight cycles.**
- **SSSP algorithm for DAGs with negative-weight edges (but no negative-weight cycles by virtue of being acyclic).**
- **Dijkstra's algorithm: Applies to weighted, directed graph when all edge weights are nonnegative.**

Procedures used by SSSP Algorithms

Initialize-Single-Source (G,s)
1. for each vertex v ∈ G
2. v.d = ∞
3. v.π = NIL
4. s.d = 0

Attribute v.d 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(u,v))
1. if v.d > u.d + w(u,v)
2. v.d = u.d + w(u,v)
3. v.π = u

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

Edge relaxation

For all v, `dist[v]` is the length of some path from s to v.

Relaxation along edge e from v to w.
- `dist[v]` is length of some path from s to v
- `dist[w]` is length of some path from s to w
- if v-w gives a shorter path to w through v, update `dist[w]` and `pred[w]`

```
if (dist[w] > dist[v] + e.weight())
{
    dist[w] = dist[v] + e.weight();
    pred[w] = e;
}
```

Relaxation sets `dist[w]` to the length of a shorter path from s to w (if v-w gives one)

Dijkstra's SSSP Algorithm

Algorithm SSSP-Dijkstra (G, s)
1. Initialize-Single-Source(G, s)
2. S = ∅
3. Q = V[G] // all vertices put into PQ
4. **while** PQ ≠ ∅
5. u = PQ.Extract-Min()
6. S = S ∪ {u}
7. **for** each outgoing neighbor v of u
8. Relax (u, v, w(u,v))

Nodes in S represent the nodes in the set of shortest paths.

Tree nodes are nodes extracted from PQ (added to S, the set of shortest paths).

Fringe nodes (aka nodes in S') are nodes in PQ with v.d < ∞

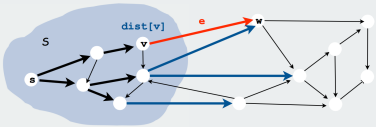
Unseen nodes are nodes in PQ with v.d = ∞

Dijkstra's algorithm

S: set of vertices for which the shortest path length from s is known.

Invariant: for v in S, `dist[v]` is the length of the shortest path from s to v.

Initialize S to s, `dist[s]` to 0, `dist[v]` to ∞ for all other v
Repeat until S contains all vertices connected to s
- find e with v in S and w in S' that minimizes `dist[v] + e.weight()`
- relax along that edge
- add w to S



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 path lengths and therefore must ADD edge weights, while Prim's algorithm compares the edge weights as they are found, with no addition of those weights to find path weights.

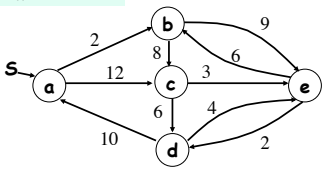
Dijkstra's SSSP Algorithm

Algorithm SSSP-Dijkstra (G, s)
1. Initialize-Single-Source(G, s)
2. S = ∅
3. Q = V[G]
4. **while** Q ≠ ∅
5. u = Q.Extract-Min()
6. S = S ∪ {u}
7. **for** each outgoing neighbor v of u
8. Relax(u, v, w(u, v))

Trace execution of Dijkstra's algorithm on graph below.

S is the set of edges in the shortest path

Relax (u, v, w(u,v))
1. if v.d > u.d + w(u,v)
2. v.d = u.d + w(u,v)
3. v.π = u



iteration 0:

Q	a	b	c	d	e
v.d	0	∞	∞	∞	∞
v.π	⊘	⊘	⊘	⊘	⊘

iteration 1: Q = Q – {a}

Q	~~a~~	b	c	d	e
v.d	0	2	12	∞	∞
v.π	⊘	a	a	⊘	⊘

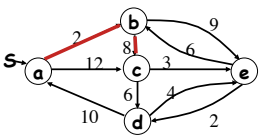
iteration 2: Q = Q – {b}

Q	~~a~~	~~b~~	c	d	e
v.d	0	2	10	∞	11
v.π	⊘	a	b	⊘	b

iteration 3: Q = Q – {c}

Q	~~a~~	~~b~~	~~c~~	d	e
v.d	0	2	10	16	11
v.π	⊘	a	b	c	b

Algorithm SSSP-Dijkstra (G, s)
1. Initialize-Single-Source(G, s)
2. S = ∅
3. Q = V[G]
4. **while** Q ≠ ∅
5. u = Q.Extract-Min()
6. S = S ∪ {u}
7. **for** each outgoing neighbor v of u
8. Relax(u, v, w(u, v))



iteration 4: Q = Q – {e}

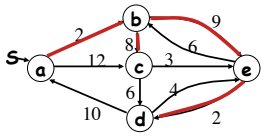
Q	~~a~~	~~b~~	~~c~~	d	~~e~~
v.d	0	2	10	13	11
v.π	⊘	a	b	e	b

iteration 5: Q = Q – {d}

Q	~~a~~	~~b~~	~~c~~	~~d~~	~~e~~
v.d	0	2	10	13	11
v.π	⊘	a	b	e	b

DONE

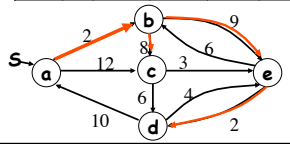
Algorithm SSSP-Dijkstra (G, s)
1. Initialize-Single-Source(G, s)
2. S = ∅
3. Q = V[G]
4. **while** Q ≠ ∅
5. u = Q.Extract-Min()
6. S = S ∪ {u}
7. **for** each outgoing neighbor v of u
8. Relax(u, v, w(u, v))



Trace table of Dijkstra's SSSP Algorithm

Note: blank means node has been extracted from priority queue

iter	u	a.d	b.d	c.d	d.d	e.d
0	-	0	∞	∞	∞	∞
1	a		2	12	∞	∞
2	b			10	∞	11
3	c				16	11
4	e				13	
5	d					



a.π ⊘
b.π a
c.π b
d.π e
e.π b

15

Running Time of Dijkstra's SSSP Alg

Algorithm SSSP-Dijkstra (G, s)
1. Initialize-Single-Source(G, s)
2. S = ∅
3. Q = V[G]
4. **while** Q ≠ ∅
5. u = Q.Extract-Min()
6. S = S ∪ {u}
7. **for** each outgoing neighbor v of u
8. Relax(u, v, w(u, v))

Steps 1-3: O(V) time
Steps 4-8:
V iterations
Suppose Extract-Min takes $O(X_Q)$ time

Steps 5-8: E iterations overall (looks at each edge once)
Suppose Relax takes $O(Y_Q)$ time.

Total: $O(VX_Q + EY_Q)$

Running Time of Dijkstra's SSSP Alg

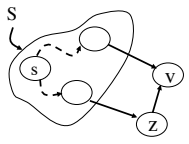
If G is dense (i.e., $\theta(V^2)$ edges):
Asymptotically speaking, there is no point in saving time using Extract-Min. Store each v.d in the v^{th} entry of an array. Each insert and decrease-key takes O(1) time. Extract-Min takes O(V) time (why?)
Total time: $O(V^2 + E) = O(V^2)$

If G is sparse (i.e., $o(V^2)$ edges):
Try to minimize time with Extract-Min, using min-heap:
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 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 no negative edge weights are 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.**
- **Processes edges in the same order on each iteration.**

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)
 1. Initialize-Single-Source(G,w,s)
 2. for i = 1 to |V| - 1
 3.     for each edge (u,v) ∈ E
 4.         Relax(u, v, w(u, v))
 5. for each edge (u, v) ∈ E
 6.     if v.d  > u.d + w(u, v)
 7.         return false
 8. return true
```

Relax (u, v, w(u,v))
1. if v.d > u.d + w(u,v)
2. v.d = u.d + w(u,v)
3. v.π = u

Boolean value returned indicates whether or not there is a negative weight cycle in G. True indicates there are no negative weight cycles.

Complexity of Bellman-Ford Algorithm

- **Initialization = O(V)**
- **Decrease-key is called (|V| - 1) · |E| times**
- **Test for negative-weight cycle = O(E)**

- **Total: O(VE) -- so more expensive than Dijkstra's, but also more general, since it detects graphs with negative weight cycles and finds single-source shortest path on graphs with negative-weight edges.**

Alternate topological sort

- **The in-degree of vertex *u* is the number of incoming edges incident on *u*. The out-degree of vertex u is the number of outgoing edges incident on *u*.**

input: directed acyclic graph (DAG)
output: ordering of nodes s.t. if (u,v) ∈ E, then u comes before v in ordering

```
Topological-Sort (G)
1.  while V is not empty
2.      remove a vertex u of in-degree 0 and all its outgoing edges.
3.      insert u at the tail of a linked list of vertices.
```

How could we implement this to run in O(V + E) time if G is represented with an adjacency list?

Property of a DAG

- **Why does the previous algorithm work?**

Claim: a DAG G must have some vertex with no incoming edges. Why?

Suppose, in contradiction, that every vertex in G has at least one incoming edge. Choose a vertex v_0. Trace the edge incoming at v_0 to its source, v_1. Since v_1 must have an incoming edge, we can follow that edge to its source, v_2. If we continue backtracking in this fashion, since there are a finite number of vertices, we will eventually return to a previously visited vertex. At this point, we will have discovered a cycle, which is a contradiction to our assumption that G is a DAG. Therefore, a DAG has at least one vertex with no incoming edge (a similar argument holds for outgoing edges).

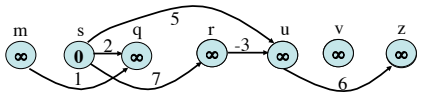
SSSPs in DAGs

Note: There can't be negative-weight cycles in G because G is a DAG.

```
DAG-Shortest-Paths (G, s)
1. Topological-Sort(G)
2. Initialize-Single-Source (G,s)
3. for each vertex u, taken in topologically sorted order
4.     for each vertex v adjacent to u
5.         v.d = min(v.d, u.d + wt(u,v))
```

SSSPs in DAGs

- **In the DAG-Shortest-Paths algorithm the vertices are added to the shortest paths tree in topological order. For a given vertex v, every incoming edge at v will be relaxed before the shortest path distance is set at v (including negative weight edges that lower the overall cost of the path). Therefore the shortest paths can be found in a DAG with negative weight edges in time no greater than the time for DFS = O(V + E).**



List the shortest path distance from s to every other node in the DAG shown above.