

Graphs: Shortest Paths

Neerja Mhaskar

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 4.4) and Prof. Janicki's course slides

Shortest paths in an edge-weighted digraph

- In this lecture, we study the shortest path problem, given an edge-weighted digraph.

edge-weighted digraph

4→5 0.35

5→4 0.35

4→7 0.37

5→7 0.28

7→5 0.28

5→1 0.32

0→4 0.38

0→2 0.26

7→3 0.39

1→3 0.29

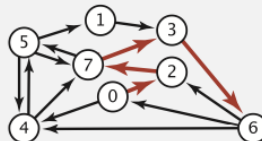
2→7 0.34

6→2 0.40

3→6 0.52

6→0 0.58

6→4 0.93



shortest path from 0 to 6

0→2 0.26

2→7 0.34

7→3 0.39

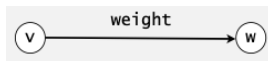
3→6 0.52

Shortest paths in an edge-weighted digraph problems

- 1 Given an edge-weighted digraph, find the shortest path from s to t .
- 2 Given an edge-weighted digraph, find the shortest paths from s to all other vertices (single source shortest paths).
- 3 It turns out that both time and space complexities of (1) and (2) are the same!
- 4 We cannot efficiently solve (1) without solving (2) in general case.

Weighted directed edge API

```
public class DirectedEdge  
  
    DirectedEdge(int v, int w, double weight)  weighted edge v→w  
  
    int from()                                vertex v  
  
    int to()                                  vertex w  
  
    double weight()                           weight of this edge  
  
    String toString()                         string representation
```



Idiom for processing an edge e : `int v = e.from(), w = e.to();`

Weighted directed edge: implementation in Java

Similar to Edge for undirected graphs, but a bit simpler.

```
public class DirectedEdge
{
    private final int v, w;
    private final double weight;

    public DirectedEdge(int v, int w, double weight)
    {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public int from()
    { return v; }

    public int to()
    { return w; }

    public int weight()
    { return weight; }
}
```

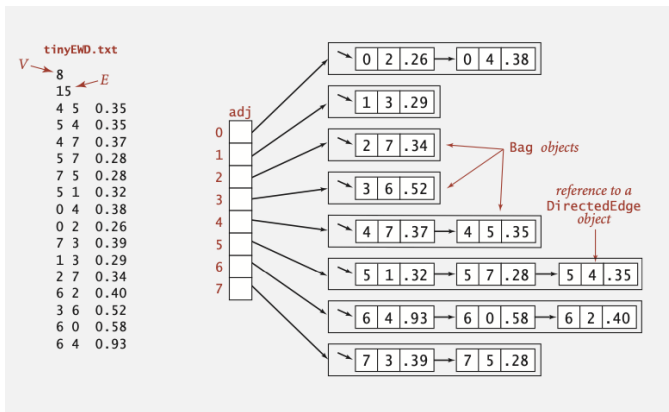
from() and to() replace
either() and other()

Edge-weighted digraph API

<code>public class EdgeWeightedDigraph</code>		
<code>EdgeWeightedDigraph(int V)</code>		<i>edge-weighted digraph with V vertices</i>
<code>EdgeWeightedDigraph(In in)</code>		<i>edge-weighted digraph from input stream</i>
<code>void addEdge(DirectedEdge e)</code>		<i>add weighted directed edge e</i>
<code>Iterable<DirectedEdge> adj(int v)</code>		<i>edges pointing from v</i>
<code>int V()</code>		<i>number of vertices</i>
<code>int E()</code>		<i>number of edges</i>
<code>Iterable<DirectedEdge> edges()</code>		<i>all edges</i>
<code>String toString()</code>		<i>string representation</i>

Conventions: Allow self-loops and parallel edges.

Edge-weighted digraph: adjacency-lists representation



Summary of properties and assumptions

Properties. Several important properties and assumptions made for this topic:

- **Paths are directed.** A shortest path must respect the direction of its edges.
- **Weights are not necessarily distances.** The edge weights might represent time or cost.
- **Not all vertices need be reachable.** If t is not reachable from s , there is no path at all, and therefore there is no shortest path from s to t .
- **Shortest paths are not necessarily unique.** There may be multiple paths of the lowest weight from one vertex to another; we are content to find any one of them.

Data structures for single-source shortest paths

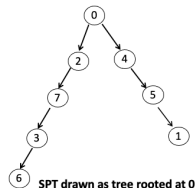
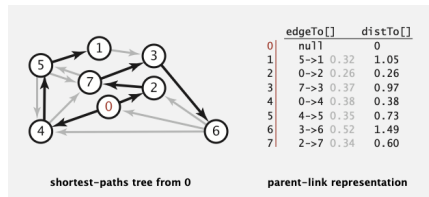
Given an edge-weighted digraph and a designated vertex s , a **shortest-paths tree (SPT)** for a source s is a subgraph containing s and all the vertices reachable from s that forms a directed tree rooted at s such that every tree path is a shortest path in the digraph.

A shortest-paths tree (SPT) solution exists. Why?

Data structures for single-source shortest paths

We can represent the SPT with two vertex-indexed arrays:

- $\text{distTo}[v]$ is length of shortest path from s to v (in the figures this weight is associated to the vertex v).
- $\text{edgeTo}[v]$ is last edge on shortest path from s to v .



Goal of all the algorithms is to compute **SPT**, that is, compute the $\text{edgeTo}[]$ and $\text{distTo}[]$ arrays.

Edge and Vertex relaxation

To compute SPT two techniques are useful:

- 1 Edge relaxation
- 2 Vertex relaxation

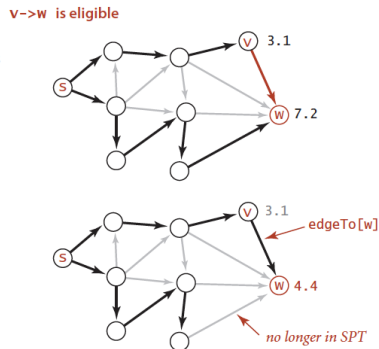
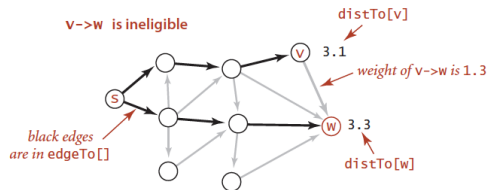
Edge relaxation - I

Edge relaxation idea:

- $distTo[v]$ is length of shortest path **known** from s to v .
- $distTo[w]$ is length of shortest **known** path from s to w .
- $edgeTo[w]$ is last edge on shortest **known** path from s to w .
- If $e = v \rightarrow w$ gives shorter path to w through v , update both $distTo[w]$ and $edgeTo[w]$ - in which case $v \rightarrow w$ is called **eligible** edge; otherwise, it is called an **ineligible** edge.

Edge relaxation - II

Example.



Edge relaxation (two cases)

Edge relaxation code

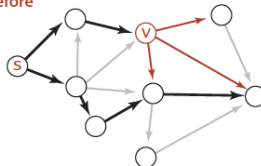
```
private void relax(DirectedEdge e)
{
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```

Edge relaxation

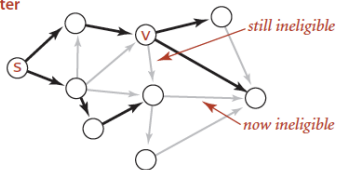
Vertex relaxation

- All the edges pointing **from** a given vertex are relaxed.
- A edge from a vertex whose $\text{distTo}[v]$ entry is finite to a vertex whose $\text{distTo}[]$ entry is infinite is eligible and will be added to $\text{edgeTo}[]$ if relaxed.

before



after



Vertex relaxation

Vertex relaxation code

```
private void relax(EdgeWeightedDigraph G, int v)
{
    for (DirectedEdge e : G.adj(v))
    {
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight())
        {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
        }
    }
}
```

Vertex relaxation

Shortest-paths optimality conditions I

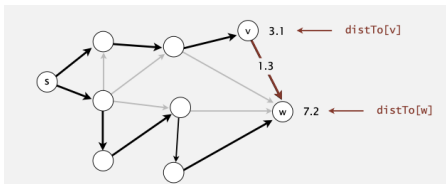
Proposition. Let G be an edge-weighted digraph.

Then $\text{distTo}[]$ are the shortest path distances from s (source) iff:

- 1 $\text{distTo}[s] = 0$.
- 2 For each vertex v , $\text{distTo}[v]$ is the length of some path from s to v .
- 3 For each edge $e = v \rightarrow w$, $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.

Proof. Pf. (\Rightarrow)

- Suppose that $\text{distTo}[w] > \text{distTo}[v] + e.\text{weight}()$ for some edge $e = v \rightarrow w$.
- Then, e gives a path from s to w (via v) of length less than $\text{distTo}[w]$ – a contradiction.




Shortest-paths optimality conditions II

Proposition. Let G be an edge-weighted digraph.
Then $\text{distTo}[]$ are the shortest path distances from s iff:


- ① $\text{distTo}[s] = 0$.
- ② For each vertex v , $\text{distTo}[v]$ is the length of some path from s to v .
- ③ For each edge $e = v \rightarrow w$, $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.

Proof. Pf. (\Leftarrow)

- Suppose that $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k = w$ is a shortest path from s to w .
- Then,

$\text{distTo}[v_1] \leq \text{distTo}[v_0] + e_1.\text{weight}()$		$e_i = i^{\text{th}} \text{ edge on shortest path from } s \text{ to } w$
$\text{distTo}[v_2] \leq \text{distTo}[v_1] + e_2.\text{weight}()$		
\dots		
$\text{distTo}[v_k] \leq \text{distTo}[v_{k-1}] + e_k.\text{weight}()$		
- Add inequalities; simplify; and substitute $\text{distTo}[v_0] = \text{distTo}[s] = 0$:

$$\text{distTo}[w] = \text{distTo}[v_k] \leq \underbrace{e_1.\text{weight}() + e_2.\text{weight}() + \dots + e_k.\text{weight}()}_{\text{weight of shortest path from } s \text{ to } w}$$
- Thus, $\text{distTo}[w]$ is the weight of shortest path to w . ■


weight of some path from s to w

Generic shortest-paths algorithm I

The previous proposition shows an equivalence between the global condition that the distances in $\text{distTo}[]$ are shortest-paths distances, and the local condition that we test to relax an edge.

Hence, the optimality conditions lead immediately to a generic algorithm that encompasses all of the shortest-paths algorithms that we consider. For now we consider nonnegative weights.

Generic algorithm (to compute SPT from s)

Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.

Repeat until optimality conditions are satisfied:

- Relax any edge.**
-

Generic shortest-paths algorithm II

Proposition. Generic algorithm computes SPT from s .

Pf sketch.

- The entry $\text{distTo}[v]$ is always the length of a simple path¹ from s to v .
- Each successful relaxation decreases $\text{distTo}[v]$ for some v .
- The entry $\text{distTo}[v]$ can decrease at most a finite number of times.

¹A directed path is simple if it has no repeated vertices.

Generic shortest-paths algorithm III

The generic algorithm does not specify in which order the edges are to be relaxed.

Efficient implementations. How to choose which edge to relax?

Ex 1. Dijkstra's algorithm (nonnegative weights).

Ex 2. Topological sort shortest path (SP) algorithm (no directed cycles).

Ex 3. Bellman-Ford algorithm (no negative cycles).

Dijkstra's algorithm (see demo)

Dijkstra's algorithm outline.

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $distTo[]$ value).
- Add the vertex closest to s (min.weight) to the SPT, and relax it; that is, relax all edges pointing from that it.

Dijkstra's algorithm: Implementation outline

The implementation maintains a **priority queue (pq)** containing vertices. It

- ① Initializes the elements of the array $distTo[]$ to positive infinity.
- ② Adds the source $s = 0$ to the priority queue pq and set $distTo[0] = 0$.
- ③ While pq is not empty, it deletes the min. priority/weighted vertex from pq and relaxes it; that is, relaxes all the edges from it.
 - ① while doing so it adds the the end points (say w) of these edges to the pq , if it is not already in pq ;
 - ② otherwise, it updates $distTo[w]$ and the weight of w in pq - only if the resulting weight is less.

Dijkstra's algorithm implementation: Observations

Observations:

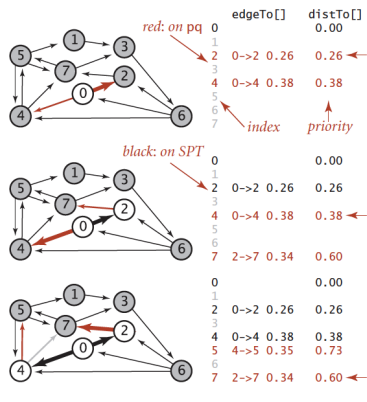
- For each vertex w on the priority queue, $distTo[w]$ = weight of a shortest path from s to w , that uses only intermediate vertices in the SPT and ends in the crossing edge $edgeTo[w]$.
- The $distTo[]$ entry for the vertex with the smallest priority is a shortest path weight, and therefore is the next to be relaxed. As a result, reachable vertices are relaxed in order of the weight of their shortest path from s .

Dijkstra's algorithm - Trace I

```

public DijkstraSP(EdgeWeightedDigraph G, int s)
{
    edgeTo = new DirectedEdge[G.V()];
    distTo = new double[G.V()];
    pq = new IndexMinPQ<Double>(G.V());
    for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;
    distTo[s] = 0.0;
    pq.insert(s, 0.0);
    while (!pq.isEmpty())
        relax(G, pq.delMin())
}

```

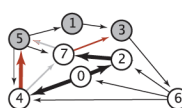


Dijkstra's algorithm - Trace II

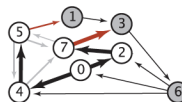
```

private void relax(EdgeWeightedDigraph G, int v)
{
    for(DirectedEdge e : G.adj(v))
    {
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight())
        {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) pq.change(w, distTo[w]);
            else pq.insert(w, distTo[w]);
        }
    }
}

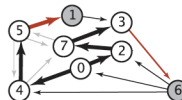
```



0		0.00
1		
2	0->2 0.26	0.26
3	7->3 0.37	0.97
4	0->4 0.38	0.38
5	4->5 0.35	0.73 ←
6		
7	2->7 0.34	0.60

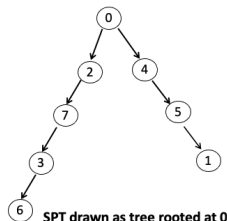
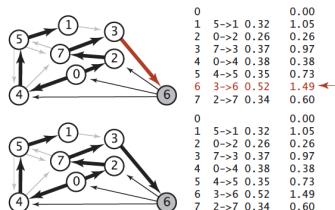


0		0.00
1	5->1 0.32	1.05 ←
2	0->2 0.26	0.26
3	7->3 0.37	0.97 ←
4	0->4 0.38	0.38
5	4->5 0.35	0.73
6		
7	2->7 0.34	0.60



0		0.00
1	5->1 0.32	1.05 ←
2	0->2 0.26	0.26
3	7->3 0.37	0.97
4	0->4 0.38	0.38
5	4->5 0.35	0.73
6	3->6 0.52	1.49 ←
7	2->7 0.34	0.60

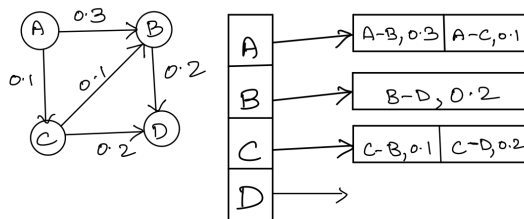
Dijkstra's algorithm - Trace III



This implementation of Dijkstra's algorithm grows the SPT by adding an edge at a time, always choosing the edge from a tree vertex to a non-tree vertex whose destination w is closest to s .

Moreover, vertices are added to the SPT in increasing order of their distance from the source, as indicated by the red arrows at the right edge of the diagram.

Shortest Path - Another example



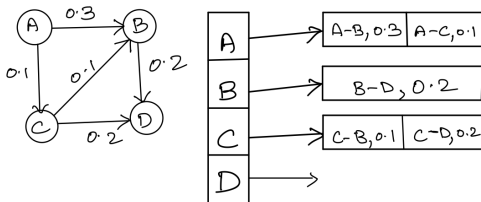
In the above graph with source = A:

Shortest path from A to B is: $A \rightarrow C \rightarrow B$ with path weight = $0.1 + 0.1 = 0.2$

Shortest path from A to C is: $A \rightarrow C$ with path weight = 0.1

Shortest path from A to D is: $A \rightarrow C \rightarrow D$ with path weight = $0.1 + 0.2 = 0.3$

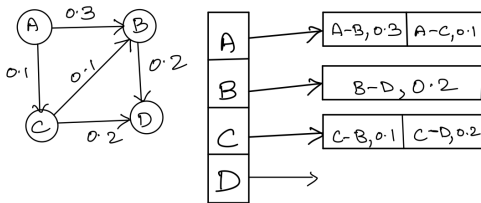
Shortest Path Dijkstra's algorithm - Another example - I



Dijkstra's algorithm maintain the below data structures to compute shortest paths from source to its reachable vertices:

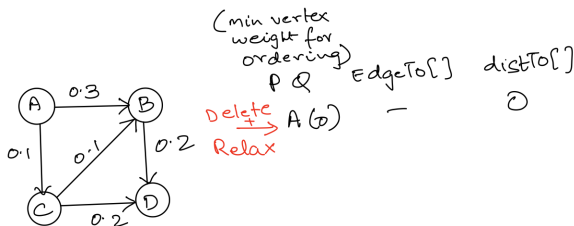
- *pq* to store vertices with priority = weight of the current shortest path from source to itself.
- *distTo[]* = vertex index array to store the weight of the shortest path from source to the respective vertex.
- *EdgeTo[]* = vertex index array to store the last edge connecting source to the vertex in the SPT.

Shortest Path Dijkstra's algorithm - Another example - II



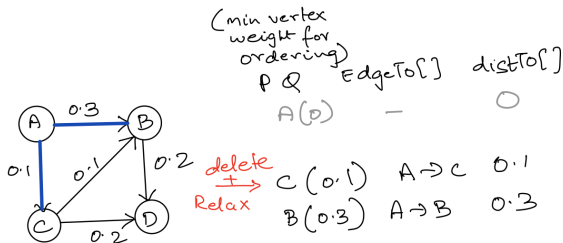
- 1 Dijkstra's algorithm, initially marks the elements of $distTo[]$ to positive infinity.
- 2 Then adds the source $A(0)$ to the priority queue pq and sets $distTo[0] = 0$.

Shortest Path Dijkstra's algorithm - Another example - III



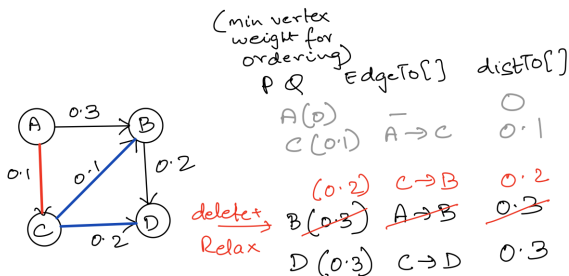
- 1 Delete the vertex in pq having min.priority [$pq.delMin()$] - in this example, it gives A.
- 2 Relax A: that is,
 - examine A's adjacency list.
 - update distances (if $distTo[w] > distTo[A] + e_{A \rightarrow w}.weight$) of vertices connected to A and already in pq ; otherwise, add them to pq , and updates $edgeTo[]$ and $distTo[]$ data structures accordingly.

Shortest Path Dijkstra's algorithm - Another example IV



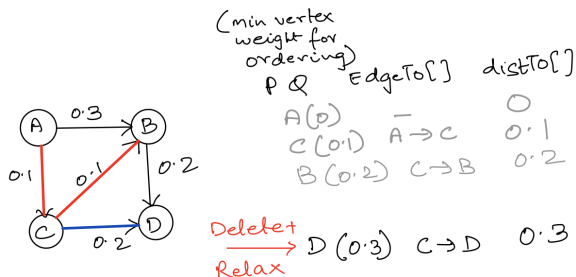
- 1 Deleting A from pq and relaxing it results in adding B, C to pq .
- 2 Blue arrows indicate edges under consideration.
- 3 Red arrows indicate edges in the SPT
- 4 Grey elements in $pq, edgeTo[], distTo[]$ indicate finalized/vertices added to SPT.

Shortest Path Dijkstra's algorithm - Another example V



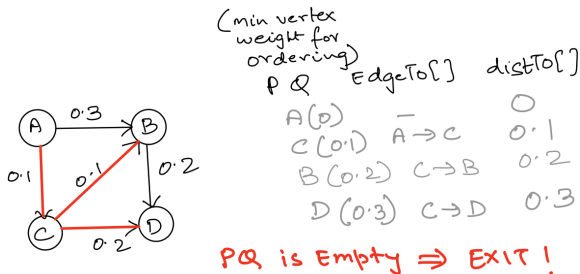
- 1 Delete the vertex in pq having min.priority [$pq.delMin()$] = C and relax it [results in updating B's values, adding D to pq , and updating their priority, $distTo[]$ and $edgeTo[]$ values].

Shortest Path Dijkstra's algorithm - Another example VI



- 1 Delete the vertex in pq having min.priority [$pq.delMin()$] = B and relax it [results in no change].

Shortest Path Dijkstra's algorithm - Another example VII



- 1 Delete the vertex in pq having min.priority [$pq.delMin()$] = D and relax it [results in no change].

Dijkstra's algorithm: correctness proof

Proposition. Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

Pf.

- Vertices are added to SPT (in increasing order of weights), after which they are relaxed; that is, all edges from it are relaxed.
- Also, when v is added to SPT, each edge $e = v \rightarrow w$ is relaxed exactly once, leaving $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$. This inequality holds until algorithm terminates because:
 - $\text{distTo}[w]$ cannot increase [$\text{distTo}[]$ values are monotone decreasing]
 - $\text{distTo}[v]$ will not change [we choose lowest $\text{distTo}[]$ value at each step (and edge weights are nonnegative)]
- Thus, upon termination, shortest-paths optimality conditions hold.

Dijkstra's algorithm: analysis

Dijkstra's algorithm (using a min.priority queue implemented as a binary heap) uses extra space proportional to V and time proportional to $E \log V$ (in the worst case) to compute the SPT rooted at a given source in an edge-weighted digraph with E edges and V vertices.

Proof. The PQ implementation: V insert, V delete-min, E decrease-key².

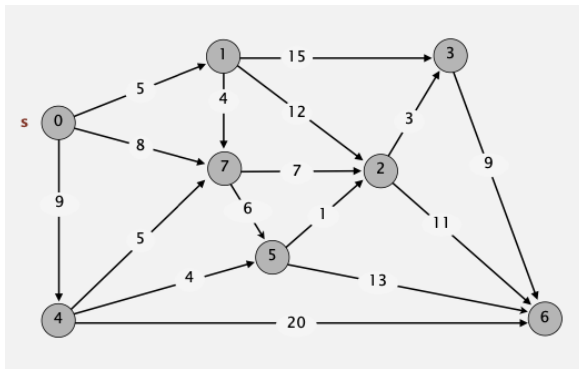
PQ implementation	insert	delete-min	decrease-key	total
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$

²Uses change operation on priority queue, which uses $\log V$ time

Acyclic edge-weighted digraphs

An **edge-weighted DAG** refers to an acyclic edge-weighted digraph.

Q. Suppose that we are given an edge-weighted DAG. Is it easier and faster to find shortest paths than in a general digraph?



A. Yes!

Topological sort shortest path algorithm - I

The topological sort shortest path (SP) algorithm :

- Solves the single-source problem in linear time $O(E + V)$
- Handles negative edge weights
- Solves related problems, such as finding longest paths.

In the course we will use Acyclic edge-weighted digraphs shortest path and Topological sort shortest path interchangeably.

Topological sort shortest path algorithm - II

Topological sort SP algorithm: combines vertex relaxation with topological sorting, and is outlined below:

- The algorithm, first initializes $\text{distTo}[s]$ to 0 and all other $\text{distTo}[]$ values to infinity,
- Then it relaxes the vertices, one by one, taking the vertices in topological sort order.

Note that, since we are processing vertices in an acyclic digraph in topological order, we never re-encounter a vertex that we have already relaxed.

Acyclic edge-weighted digraphs shortest path - Trace I

Topological sort SP algorithm

```

public AcyclicSP(EdgeWeightedDigraph G, int s)
{
    edgeTo = new DirectedEdge[G.V()];
    distTo = new double[G.V()];

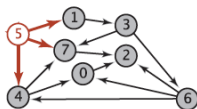
    for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;
    distTo[s] = 0.0;

    Topological top = new Topological(G);
    for (int v : top.order())
        relax(G, v);
}

```

topological sort

5 1 3 6 4 7 0 2



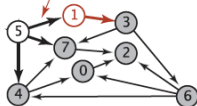
edgeTo[]

```

0
1 5->1
2
3
4 5->4
5
6
7 5->7

```

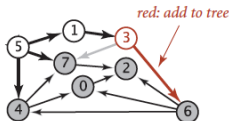
thick black: on tree



```

0
1 5->1
2
3 1->3
4 5->4
5
6
7 5->7

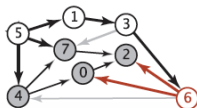
```



```

0
1 5->1
2
3 1->3
4 5->4
5
6 3->6
7 5->7

```

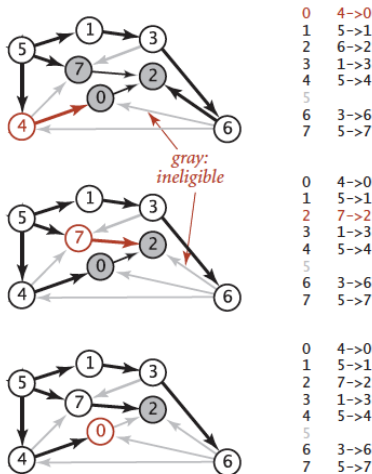


```

0 6->0
1 5->1
2 6->2
3 1->3
4 5->4
5
6 3->6
7 5->7

```

Acyclic edge-weighted digraphs shortest path - Trace II



Acyclic edge-weighted digraphs shortest path - Another Example - I

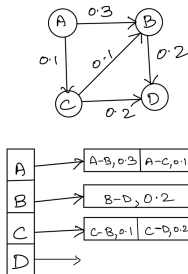


Figure 1: Sample graph G

Reverse Post order

```

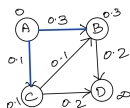
dfs(A)
  dfs(B)
    dfs(D)
      done D      D
    done B      B D
  dfs(C)
    check B
    check D
  done C      C B D
done A      A C B D
  
```

Figure 2: Reverse Postorder of $G = ACBD$

Acyclic edge-weighted digraphs shortest path - Another Example - II

Reverse Post order =

A C B D



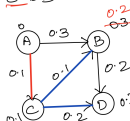
Relax A

	EdgeTo[]	distTo[]
A	—	0
B	A → B	0.3
C	A → C	0.1
D	—	∞

- We relax A; that is, update the *EdgeTo*[] and *distTo*[] values of B, C, and A to SPT.

Reverse Post order =

A C B D



Relax C

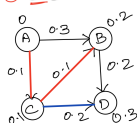
	EdgeTo[]	distTo[]
A	—	0
B	A → B	0.3 0.1 + 0.1 = 0.2
C	A → C	0.1
D	C → D	0.1 + 0.2 = 0.3

- The next vertex to be relaxed in reverse post order is C, therefore we add it to SPT along with the *edgeTo*[C] edge. Then we relax it.

Acyclic edge-weighted digraphs shortest path - Another Example - III

Reverse Post order =

A C B D



Relax B

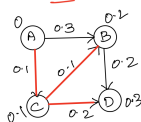
EdgeTo[] distTo[]

A	—	0
B	C → B	0.2
C	A → C	0.1
D	C → D	0.3

- The next vertex to be relaxed in reverse post order is *B*, therefore we add it to SPT along with the edgeTo[B] edge. Then we relax it [does not change anything].

Reverse Post order =

A C B D



Relax D

EdgeTo[] distTo[]

A	—	0
B	C → B	0.2
C	A → C	0.1
D	C → D	0.3

- The last vertex to be relaxed in reverse post order is *D*, therefore we add it to SPT along with the edgeTo[D] edge. Then we relax it [does change anything].

Shortest paths in edge-weighted DAGs: complexity

Topological sort SP algorithm computes SPT in any edge-weighted DAG in time proportional to $E + V$ – [edge weights can be negative.]

Proposition. Topological sort algorithm computes SPT in any edge-weighted DAG in time proportional to $E + V$.

edge weights
can be negative!

Pf.

- Each edge $e = v \rightarrow w$ is relaxed exactly once (when vertex v is relaxed), leaving $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.
- Inequality holds until algorithm terminates because:
 - $\text{distTo}[w]$ cannot increase ← $\text{distTo}[]$ values are monotone decreasing
 - $\text{distTo}[v]$ will not change ← because of topological order, no edge pointing to v will be relaxed after v is relaxed
- Thus, upon termination, shortest-paths optimality conditions hold. ■

Longest paths in edge-weighted DAGs

Formulate as a shortest paths problem in edge-weighted DAGs.

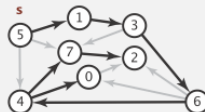
- Negate all weights.
- Find shortest paths.
- Negate weights in result.

longest paths input

5→4 0.35
4→7 0.37
5→7 0.28
5→1 0.32
4→0 0.38
0→2 0.26
3→7 0.39
1→3 0.29
7→2 0.34
6→2 0.40
3→6 0.52
6→0 0.58
6→4 0.93

shortest paths input

5→4 -0.35
4→7 -0.37
5→7 -0.28
5→1 -0.32
4→0 -0.38
0→2 -0.26
3→7 -0.39
1→3 -0.29
7→2 -0.34
6→2 -0.40
3→6 -0.52
6→0 -0.58
6→4 -0.93



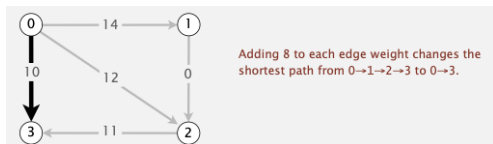
Key point. Topological sort algorithm works even with negative weights.

Shortest paths with negative weights: failed attempts

Dijkstra. Doesn't work with negative edge weights.



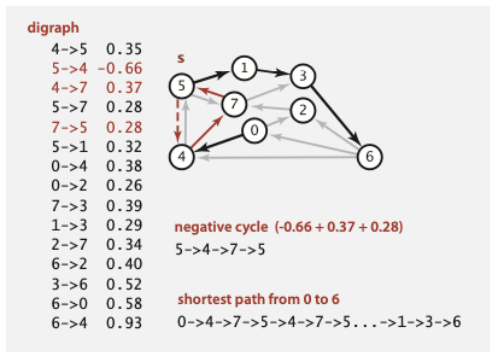
Re-weighting. Add a constant to every edge weight doesn't work.



Conclusion. Need a different algorithm.

Shortest paths with negative cycle

A **negative cycle** is a directed cycle whose sum of edge weights is negative.



The shortest path problem is meaningless with negative cycles (every time a cycle is added to the path, its weight decreases), and hence:

Proposition. An SPT exists iff no negative cycles.

Bellman-Ford algorithm - General idea

Bellman-Ford algorithm

Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.

Repeat V times:

- Relax each edge.**
-

Bellman-Ford algorithm - Queue Based

Queue-based Bellman-Ford. [for improved running time]

- We can easily determine a priori that numerous edges are not going to lead to a successful relaxation in any given pass: the only edges that could lead to a change in `distTo[]` are those leaving a vertex whose `distTo[]` value changed in the previous pass.
- To keep track of such vertices, the algorithm uses a FIFO queue. It contains the vertices to be relaxed.
- To avoid duplicates in the queue, a vertex-indexed boolean array `onQ[]` that indicates which vertices are on the queue is also maintained.

Queue-based Bellman-Ford: Algorithm Implementation

```
public BellmanFordSP(EdgeWeightedDigraph G, int s)
{
    distTo = new double[G.V()];
    edgeTo = new DirectedEdge[G.V()];
    onQ = new boolean[G.V()];
    queue = new Queue<Integer>();
    for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;
    distTo[s] = 0.0;
    queue.enqueue(s);
    onQ[s] = true;
    while (!queue.isEmpty() && !this.hasNegativeCycle())
    {
        int v = queue.dequeue();
        onQ[v] = false;
        relax(v);
    }
}
```

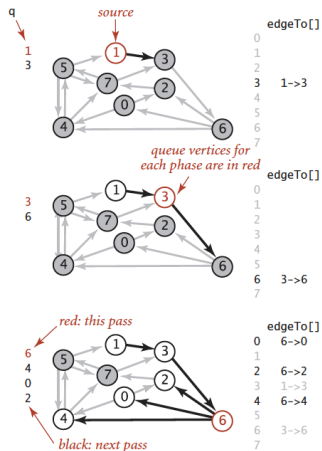
Queue-based Bellman-Ford: Trace I

tinyEWDn.txt

```

4->5 0.35
5->4 0.35
4->7 0.37
5->7 0.28
7->5 0.28
5->1 0.32
0->4 0.38
0->2 0.26
7->3 0.39
1->3 0.29
2->7 0.34
6->2 -1.20
3->6 0.52
6->0 -1.40
6->4 -1.25

```

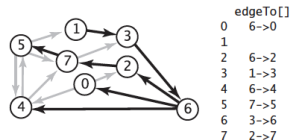
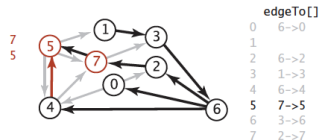
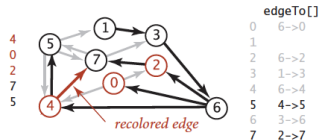


Queue-based Bellman-Ford: Trace II

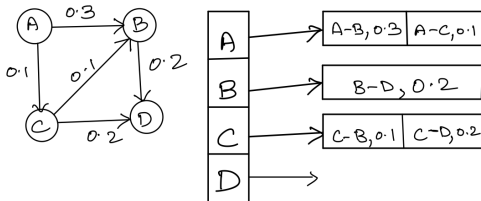
```

private void relax(EdgeWeightedDigraph G, int v)
{
    for (DirectedEdge e : G.adj(v))
    {
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight())
        {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (!onQ[w])
            {
                q.enqueue(w);
                onQ[w] = true;
            }
        }
    }
    if (cost++ % G.V() == 0)
        findNegativeCycle();
}

```



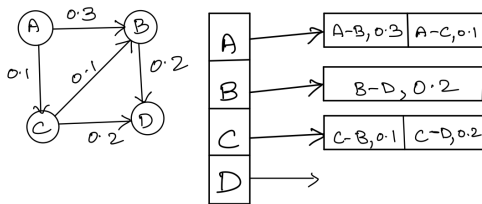
Queue-based Bellman-Ford: Another example I



Queue-based Bellman-Ford algorithm maintains the below data structures to compute shortest paths from source to its reachable vertices:

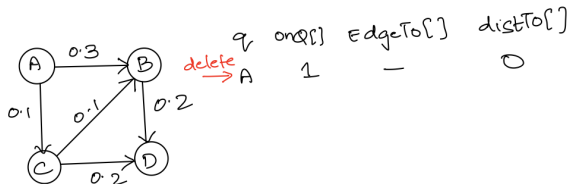
- q to store vertices whose weight is reduced during edge relaxation.
- $distTo[]$ = vertex indexed array to store the weight of the shortest path from source to the respective vertex.
- $EdgeTo[]$ = vertex indexed array to store the last edge connecting source to the vertex in the SPT.
- $onQ[]$ = vertex indexed boolean array to store if the vertex is on q . 0 means not on queue q , and 1 means the vertex is on queue q .

Queue-based Bellman-Ford: Another example - II



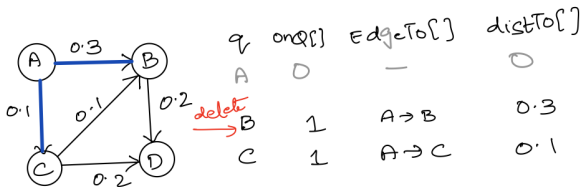
- 1 Queue-based Bellman-Ford algorithm, initially marks the elements of $distTo[]$ to positive infinity.
- 2 Then adds the source A to the FIFO queue q and sets $distTo[0] = 0$ and $onQ[0] = 1$.

Queue-based Bellman-Ford: Another example - III



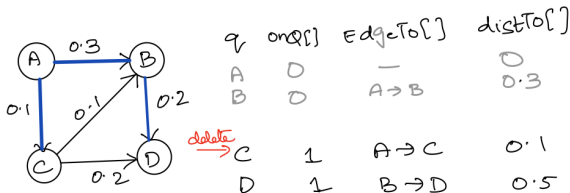
- 1 Delete the least recently added vertex from q [$q.dequeue()$] - in this example, it gives A .
- 2 Set $onQ[A] = 0$ and relax A ; that is,
 - examine A 's adjacency list.
 - update distances (if $distTo[w] > distTo[A] + e_{A \rightarrow w}.weight$) of vertices with an edge from A , and add only those vertices to q if they are not on queue q and whose weights have been decreased by relaxing A . Therefore, we add B, C to q .

Queue-based Bellman-Ford; Another example - IV



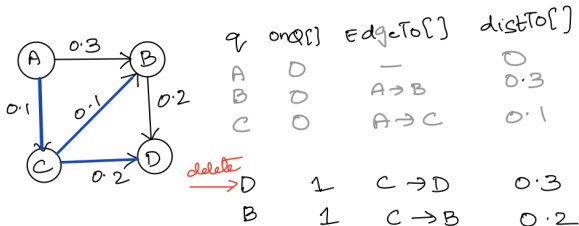
- 1 Blue arrows indicate edges under consideration.
- 2 Greyed elements indicate vertices NOT in queue.
- 3 B will be the next vertex dequeued from q .

Queue-based Bellman-Ford: Another example - V



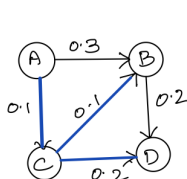
- 1 Delete B and relax it [results in adding D to q].
- 2 C will be the next vertex dequeued from q .

Queue-based Bellman-Ford: Another example - VI



- 1 Delete C and relax it [results in adding B back to q !].
- 2 Notice how some blue edges go back to being black ($A \rightarrow B, B \rightarrow D$) and new blue edges are added ($C \rightarrow B, C \rightarrow D$).
- 3 D will be the next vertex dequeued from q .

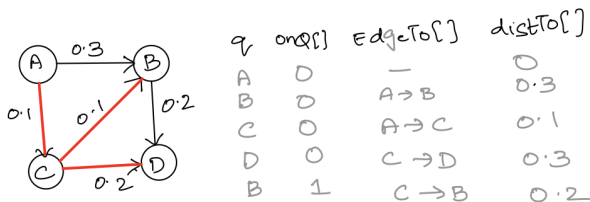
Queue-based Bellman-Ford: Another example - VII



q	$onQ[]$	$edgeTo[]$	$distTo[]$
A	0	—	0
B	0	A \rightarrow B	0.3
C	0	A \rightarrow C	0.1
D	0	C \rightarrow D	0.3
B	1	C \rightarrow B	0.2

- 1 Delete D and relax it [results in no changes in $edgeTo[]$ and $distTo[]$ data structures].
- 2 B will be the next vertex dequeued from q .

Queue-based Bellman-Ford: Another example - VIII



queue empty \Rightarrow EXIT!

- 1 Delete B and relax it [results in no changes in $edgeTo[]$ and $distTo[]$ data structures].
- 2 q is empty. Therefore, exit.
- 3 Red arrows indicate the final SPT.

Bellman-Ford algorithm: analysis

Proposition. Queue-based Bellman-Ford computes SPT in any edge-weighted digraph with no negative cycles in time proportional to $E \times V$ in the worst case.

However, it is much faster than that in practice [takes $(E + V)$ time].

The queue-based Bellman-Ford algorithm is an effective and efficient method for solving the shortest-paths problem that is widely used in practice, even for the case when edge weights are positive.