Shortest Path Algorithms

- Shortest path problem
  - Dijkastra Algorithm
  - Belman-Ford algorithm

Shortest Paths Problem

Problem Statement: Find the minimum-weight path from a given source vertex s to another vertex v in a given weighted directed graph G(V,E)

– Shortest-path having minimum weight

– Weight of path as sum of weights of its constituent edges

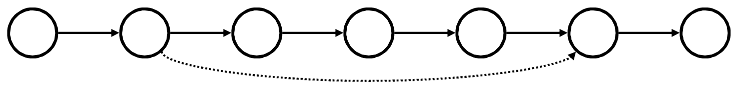
Example: *A Road map ,Railway map*

- **Flavors**:
  - **Single source shortest paths problem** → Finds shortest path from given source S to each vertex *V*
  - Single destination shortest paths problems →Find shortest path to a given destination D from each vertex V
  - Single pair shortest path problem → Find shortest path from U to V for given vertex U and V
  - All pairs shortest paths problem→ Find shortest path from U to V for every pair of vertices U and V

Shortest Path Properties

Optimal substructure of a shortest path:

shortest path between two vertices contains other shortest paths within it

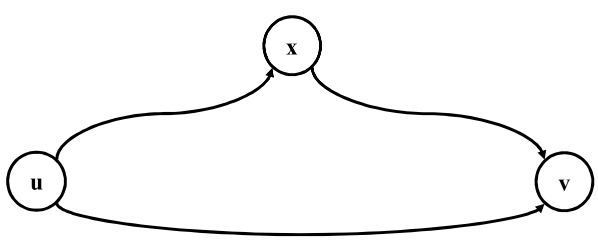


suppose some subpath is not a shortest path

- then there must exist a shorter subpath
- could substitutes the shorter subpath for a shorter path
- but then overall path is not shortest path which contradicts

Shortest Path Properties

- Let $\delta(u,v) \rightarrow$ the weight of the shortest path from u to v
- Shortest paths satisfy the *triangle inequality*: $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$



This path is no longer than any other path

Shortest Paths Problem

Certain Constraints :

- The graph cannot contain any *negative weight cycles*
  - as there would be no minimum path
  - since it could simply continue to follow the negative weight cycle producing a path weight of infinity($-\infty$)
- solution cannot have any *positive weight cycles*
  - as the cycle could simply be removed giving a lower weight path
- solution can be assumed to have no zero weight cycles
  - as they would not affect the minimum value
- under these restrictions, the shortest paths must be *acyclic*
  - with $\leq |V|$ distinct vertices ➔ $\leq |V| - 1$ edges in each path

Initialization

INITIALIZE-SINGLE-SOURCE(G, s)

1 **for** each vertex $v \in V[G]$
2 **do** $d[v] \leftarrow \infty$
3 $\pi[v] \leftarrow$ NIL
4 $d[s] \leftarrow 0$

-- *d[v]* shortest path estimate :-- initially distance from source to node is ∞
-- maintain for each vertex a predecessor ***π[v]***
--predecessor either another vertex or NIL

Relaxation

- a key technique in shortest path algorithms
  - *edge relaxation* determine whether going through edge (*u*,*v*) reduces the distance to *v*
    - if so update ***π[v]*** and ***d[v]***
  - It lowers the weight upper-bound to a vertex if new edge is lower than current estimate
  - Current estimate *d[v]* is shortest path explored so far from source s to *v*
  - Specifically, ***for all v, maintain upper bound d[v] on δ(s,v)***

RELAX(u, v, w)

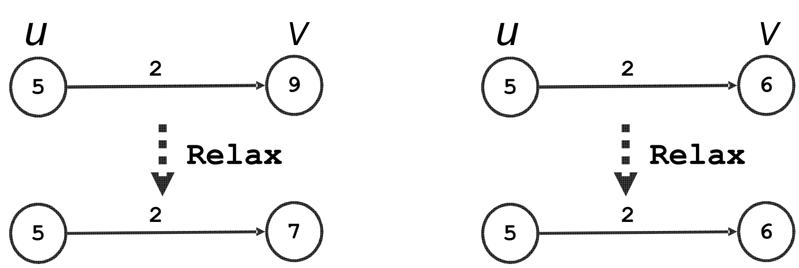
if $d[v] > d[u] + w(u, v)$

  then $d[v] \leftarrow d[u] + w(u, v)$

    $\pi[v] \leftarrow u$

Edge Relaxation

- Tests whether we can improve shortest path to v found so far by going through u , if so update *d[v]* and *π[v]*
- Relaxation may decrease value of shortest path estimate and update ***v***'s predecessor field



Bellman-Ford Algorithm

works with negative weight edges

```
BELLMAN-FORD(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  for i ← 1 to |V[G]| − 1
      do for each edge (u, v) ∈ E[G]
             do RELAX(u, v, w)
  for each edge (u, v) ∈ E[G]
      do if d[v] > d[u] + w(u, v)
            then return FALSE
  return TRUE
```

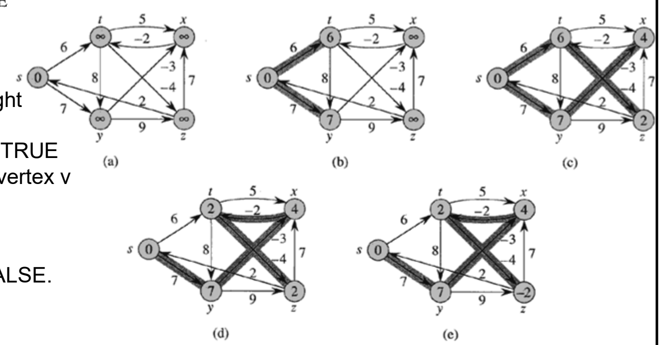
path to any reachable vertex can be found by starting at the vertex and following the π 's back to the source.

- When no negative edge weight cycle reachable from s
  - → Bellman-Ford returns TRUE
  - → d[v] = δ(s,v) for every vertex v
- If a negative edge weight cycle reachable from s
  - → Bellman-Ford returns FALSE.

Time complexity: Θ(|V| |E|)

This algorithm also detect any negative weight cycles

- such that there is no solution



The execution of the Bellman-Ford algorithm. The source is vertex s. The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

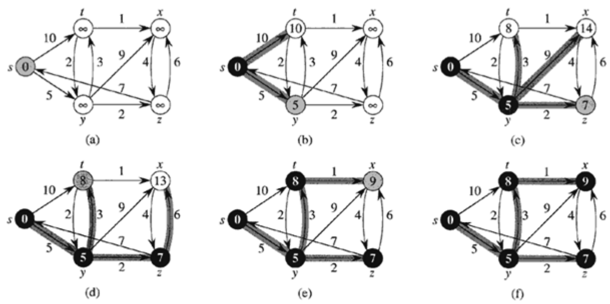
Dijkstra's Algorithm

- If no negative edge weights, we can beat BF
- Similar to breadth-first search
  - use predecessor π and distance *d* fields for each vertex as BFS
  - Grow a tree gradually, advancing from vertices taken from a queue
- Also similar to Prim's algorithm for MST
  - Use a priority queue keyed on d[v]

Dijkstra's Algorithm

// all edge weights are assumed non-negative

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← ∅              /// contains vertices of final shortest-path weights from s
3  Q ← V[G]           // Initialize priority queue Q
4  while Q ≠ ∅
5    do u ← EXTRACT-MIN(Q)   // Extract new vertex
6       S ← S ∪ {u}
7       for each vertex v ∈ Adj[u]
8           do RELAX(u, v, w)     ///Perform relaxation for each vertex v adjacent to u
```



The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d and π values shown in part (f) are the final values.