Single-Source Shortest Path

Analysis of Algorithms

Shortest Path Applications

http://en.wikipedia.org/wiki/Seam_carving

- Map routing
- Seam carving
- Robot navigation
- Texture mapping
- Typesetting in TeX
- Urban traffic planning
- Optimal pipelining of VLSI chip
- Telemarketer operator scheduling
- Routing of telecommunications messages
- Network routing protocols (OSPF, BGP, RIP)
- Exploiting arbitrage opportunities in currency exchange
- Optimal truck routing through given traffic congestion pattern

Single-Source Shortest Path

- Single-source shortest-path algorithms find the series of edges between two vertices that has the smallest total weight
- A minimum spanning tree algorithm won’t work for this because it would skip an edge of larger weight and include many edges with smaller weights that could result in a longer path than the single edge

Single-Source Shortest Path

- Initialize distTo[source] = 0
- Initialize distTo[v] = ∞ for all other vertices, v
- Optimality condition:
  - For each edge (u, v), distTo[v] ≤ distTo[u] + w(u, v)
- To achieve the optimal condition, repeat until satisfied:
  - Relax an edge
  - Relaxing an edge means getting “closer to optimal” on each iteration

Edge Relaxation

- "Relaxing" an edge:
  - If an edge (u, v) with weight w gives a shorter path from the source to v through u, then update the distTo[v] and set the parent (predecessor) of v to u:

```
RELAX(u, v, W):
    If distTo[v] > distTo[u] + w[u, v]
        distTo[v] := distTo[u] + w[u, v]
        parent[v] := u
```

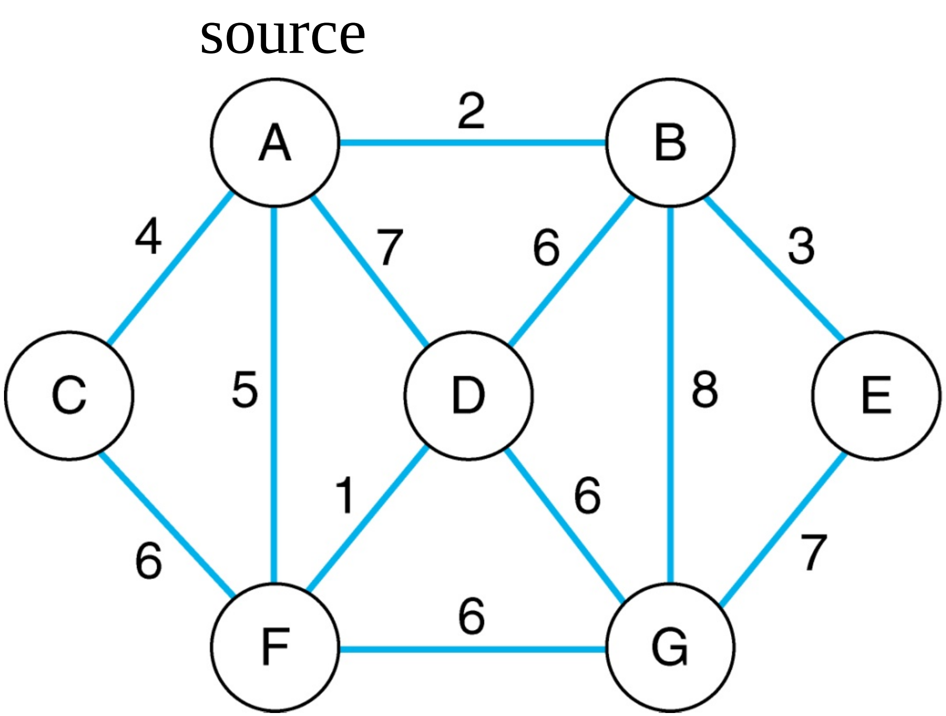
Dijkstra’s Algorithm

- Dijkstra’s algorithm is similar to the Prim MST algorithm, but instead of just looking at a ***single*** shortest edge from a vertex to a vertex in the fringe, we look at the ***overall*** shortest path from the start vertex to a vertex in the fringe
- Note: In order for Dijkstra’s method to work, all weights must be non-negative

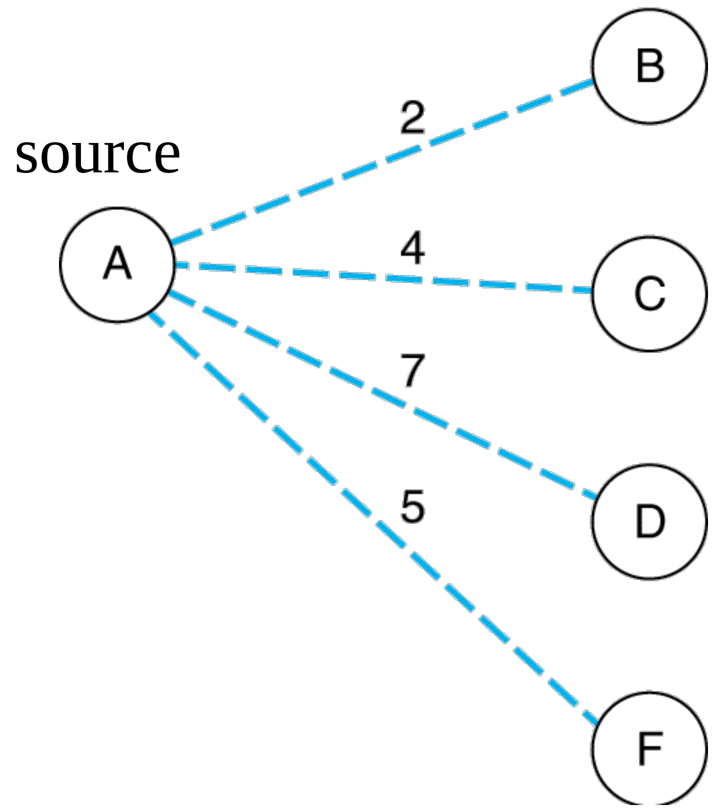
Dijkstra's Algorithm

```
DIJKSTRA(Graph, source):
    Initialize distance to every vertex to ∞
    Initialize distance to source to 0
    Initialize shortest path set S to empty
    Insert all vertices into priority queue, PQ

    while PQ is not empty:
        u := extract the vertex with the min value in the PQ
        Insert vertex u into set S
        for each vertex v adjacent to u:
            RELAX(u, v)
            update the priority of v
```

Dijkstra Example

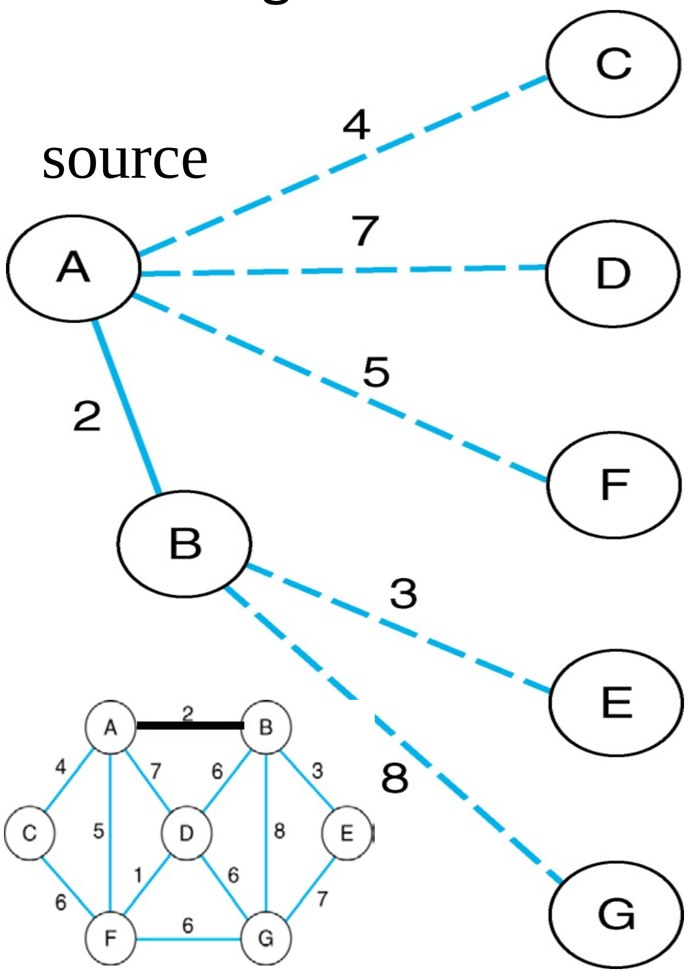


Initial fringe:

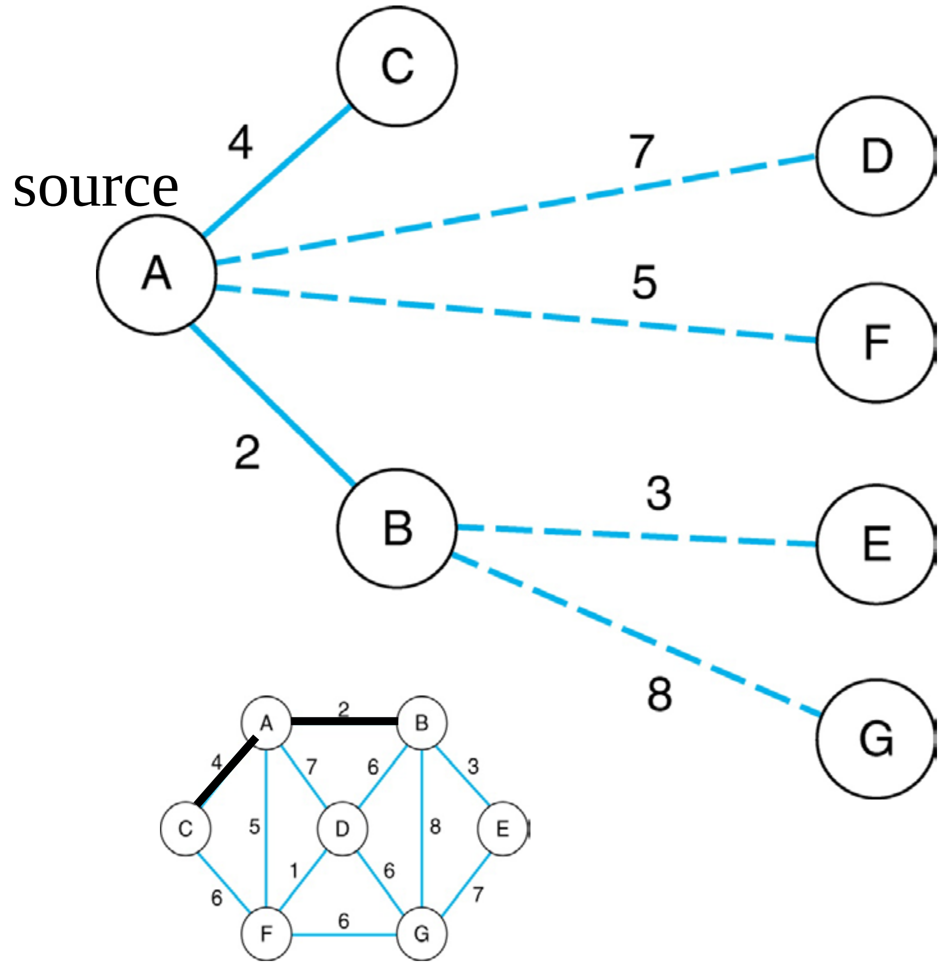


Dijkstra Example

Select edge A-B:

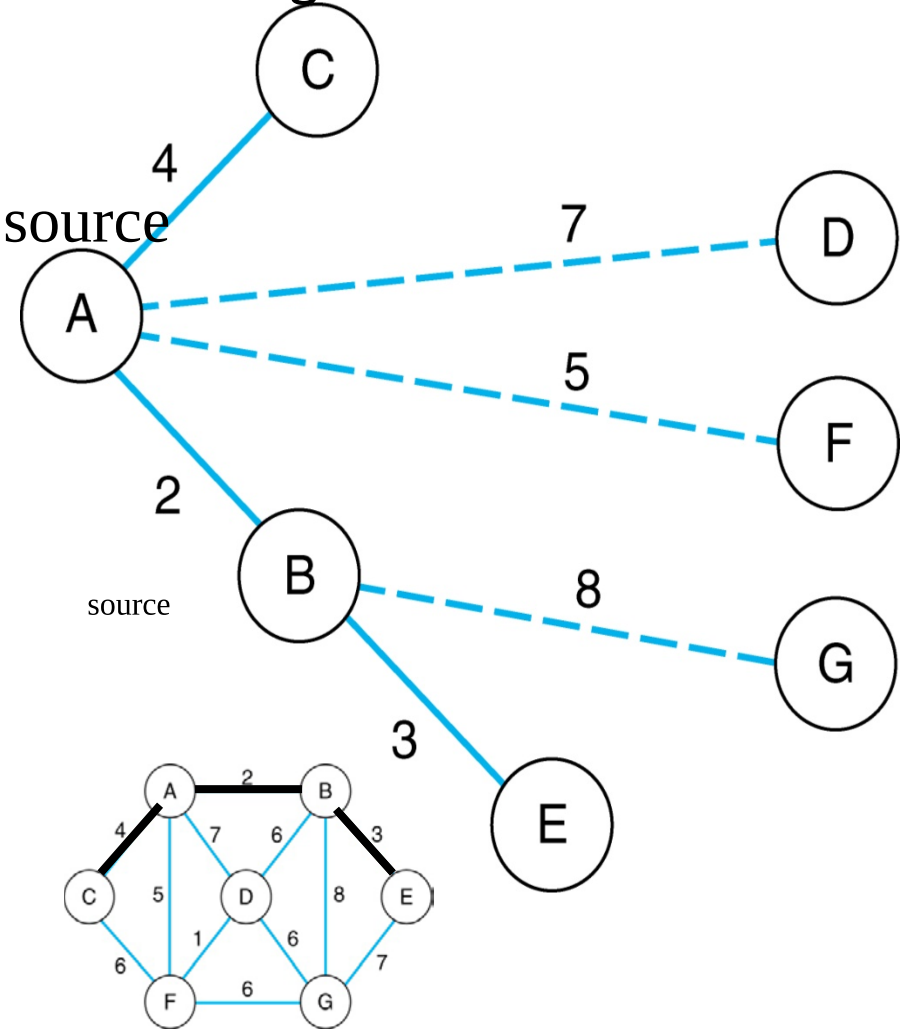


Select edge A-C:

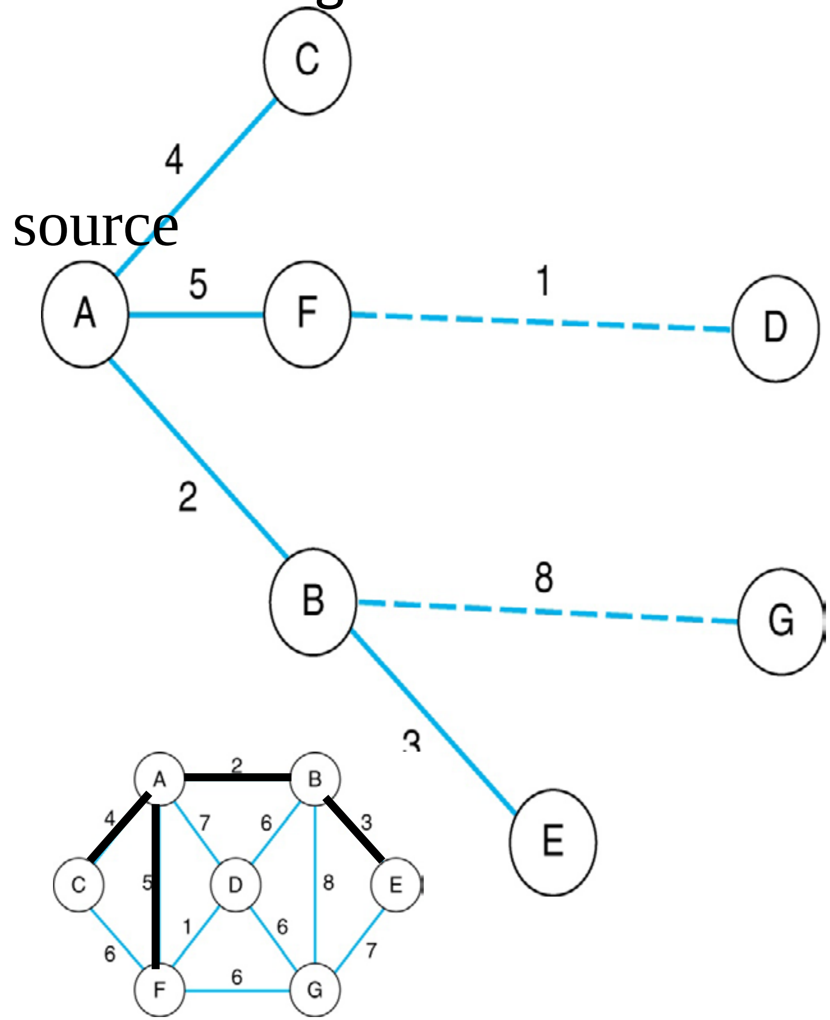


Dijkstra Example

Select edge B-E:

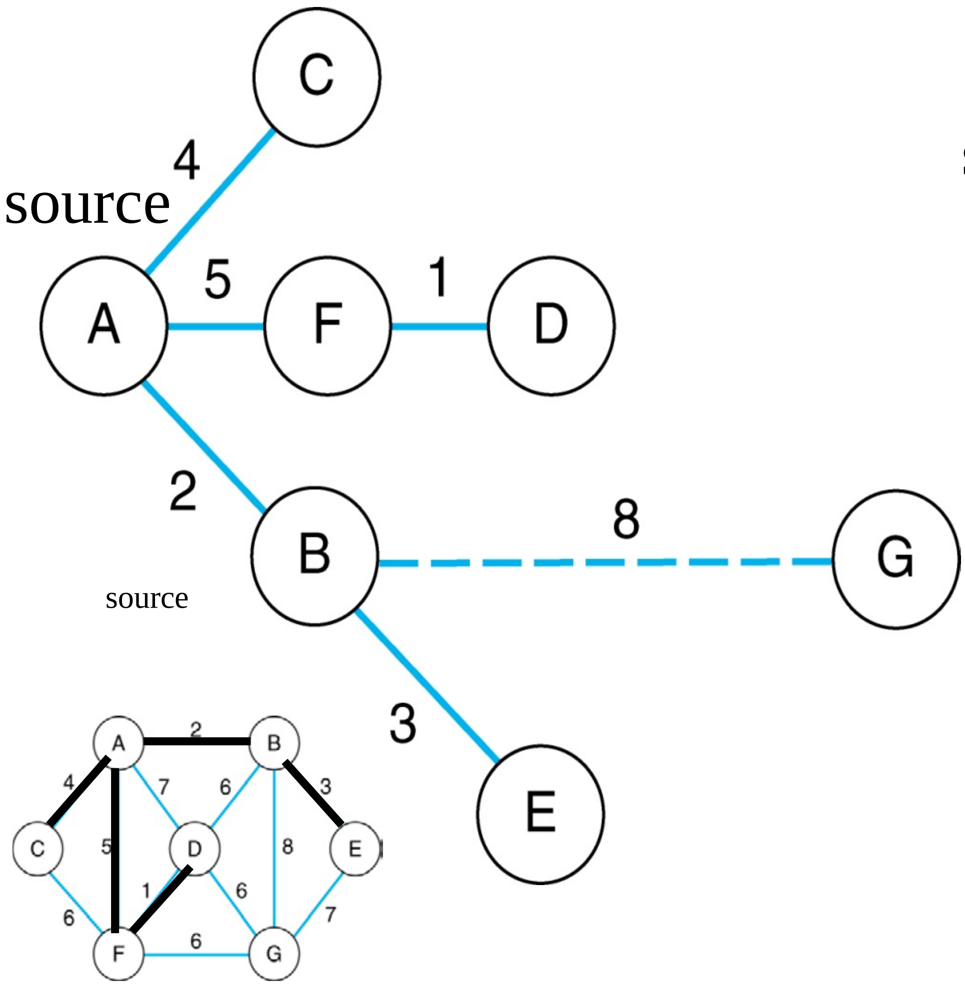


Select edge A-F:

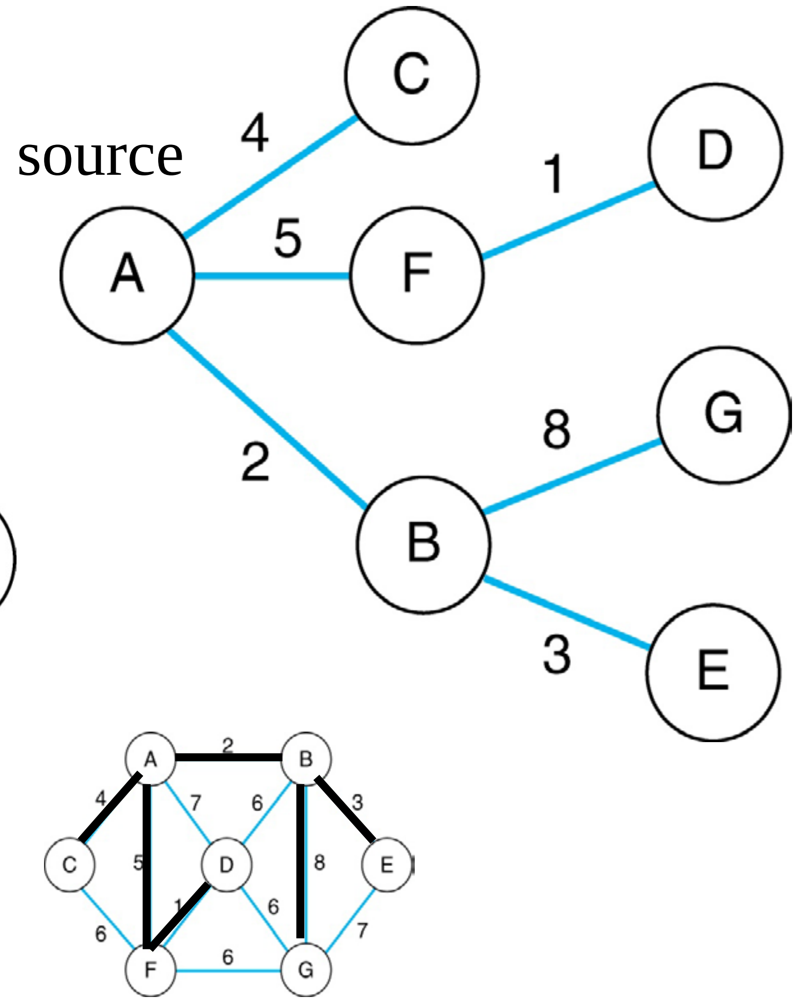


Dijkstra Example

Select edge F-D:



Select edge B-G:



Dijkstra and Prim

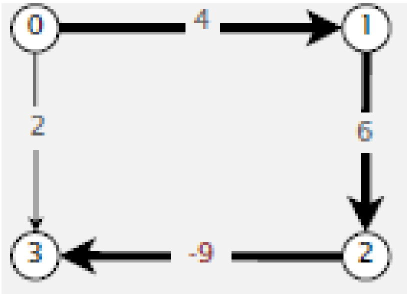
- Dijkstra's shortest path algorithm is essentially the same as Prim's minimum spanning tree algorithm
- The main distinction between the two is the rule that is used to choose next vertex for the tree
  - Prim: Choose the closest vertex (smallest weight) ***to any vertex*** in the minimum spanning tree so far
  - Dijkstra's: Choose the closest vertex (smallest weight) ***from the source vertex***
  - Note: DFS and BFS are also in this family of algorithms

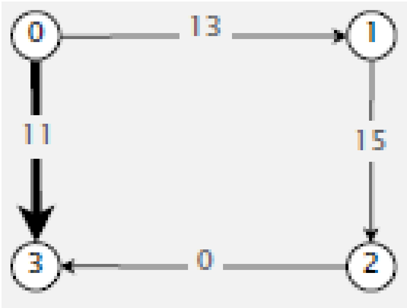
Analysis of Dijkstra's Algorithm

- Algorithm:
  - While the PQ is not empty, return and remove the "best" vertex (the one closest to the source), and update the priorities of all the neighbors of that best vertex
  - The overall runtime depends on implementation:
    - Using a simple array or linked list causes the runtime to be proportional to $N^2 + M \approx N^2$ (best for dense graph)
    - Using a binary heap causes the total runtime to be proportional to $N \log N + M \log N \approx M \log N$ (best for sparse graph)

Negative Weights

- Dijkstra does not work with negative weights
  - Dijkstra selects vertex 3 immediately after 0, but shortest path from 0 to 3 is 0 → 1 → 2 → 3
- What about re-weighting the edges?
  - Add a constant to every edge weight to make all edges positive doesn't work either
  - Adding 9 to each edge weight causes Dijkstra to again incorrectly select vertex 3
- Conclusion: We need a different algorithm for negative weights



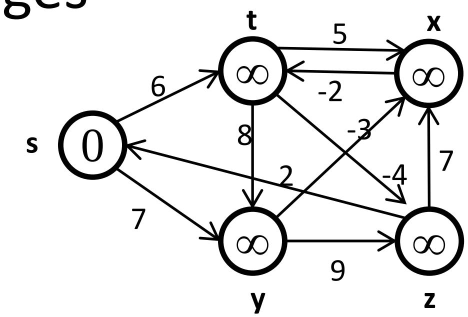


Bellman-Ford Algorithm

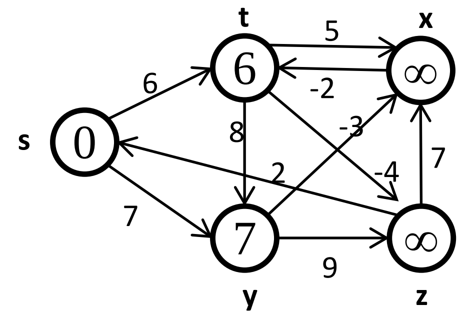
```
BELLMAN-FORD(Graph, source):
    Initialize distance to every vertex to ∞
    Initialize distance to source to 0

    for i := 1 to N-1
        for each edge (u, v)
            RELAX(u, v)

    for each edge (u, v)
        if distTo[v] > distTo[u] + w[u, v]
            return false

    return true
```

Bellman-Ford Example

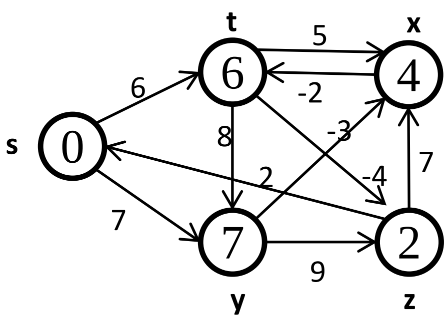
Each pass relaxes the edges in some arbitrary order:
(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)



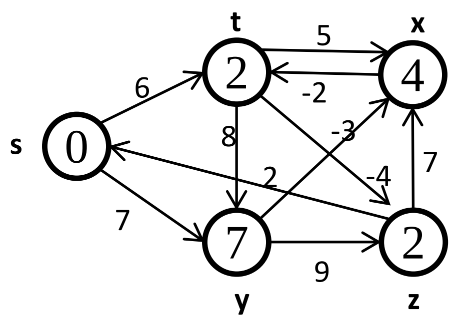
Start



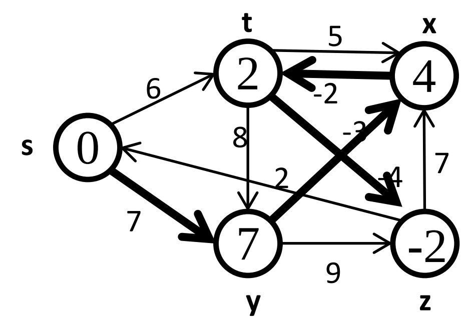
After Pass 1



After Pass 2



After Pass 3



After Pass 4

Bellman-Ford Java Code

```
public class BellmanFordSP
{
   private double[] distTo;
   private DirectedEdge[] edgeTo;
   private boolean[] onQ;
   private Queue<Integer> queue;

   public BellmanFordSPT(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 < V; v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;

      queue.enqueue(s);
      while (!queue.isEmpty())
      {
         int v = queue.dequeue();
         onQ[v] = false;
         for (DirectedEdge e : G.adj(v))
            relax(e);
      }
   }
}
```

queue of vertices whose `distTo[]` value changes

```
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;
      if (!onQ[w])
      {
         queue.enqueue(w);
         onQ[w] = true;
      }
   }
}
```

Analysis of Bellman-Ford

- Weights can be negative, but the graph cannot have negative-weight cycles!
- Bellman-Ford will detect a negative-weight cycle
  - Run the algorithm one more iteration: if the shortest path returned is ***less than*** the shortest path from the previous iteration, then return false (no solution exists because of a negative-weight cycle)
  - Else return true (the path returned is the shortest path solution)
- Runtime
  - N-1 passes, each pass looks at M edges
  - Thus, the total runtime is proportional to ***N·M***

Analysis of Bellman-Ford

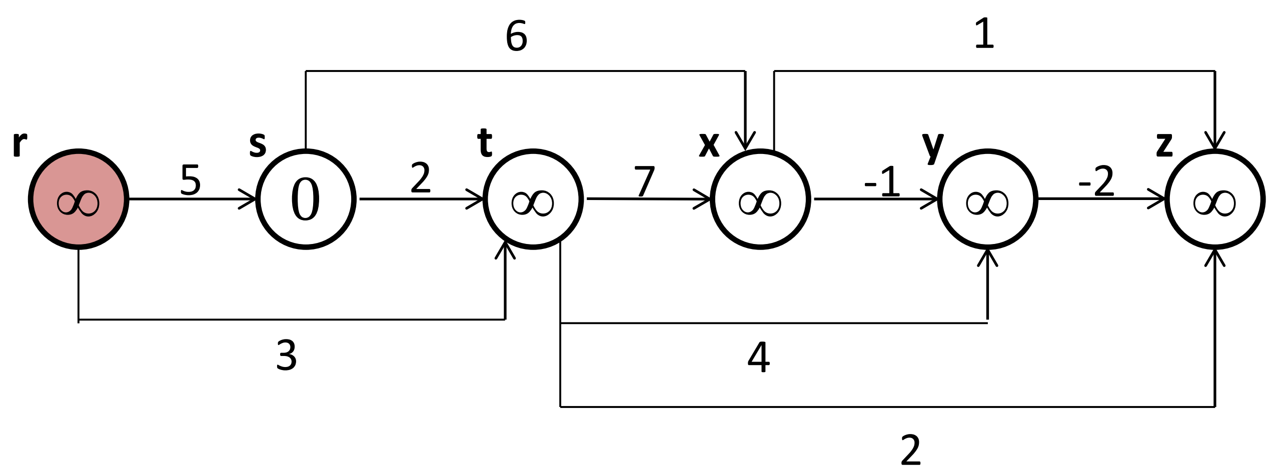
- Bellman-Ford is naturally distributed, whereas Dijkstra is naturally local
- Can be used for a network routing protocol
  - Change from a source-driven algorithm to a destination-driven algorithm by just reversing the direction of the edges in Bellman-Ford
  - Change to a “push-based” algorithm: as soon as a vertex *v* discovers it’s shortest path to the destination, *v* notifies all of its neighbors
    - This works well even in an asynchronous network

Acyclic Shortest Path Algorithm

- Suppose an edge-weighted digraph has no directed cycles (i.e., it is a weighted DAG)
- Consider the vertices in topological order
- Relax all edges pointing from that vertex

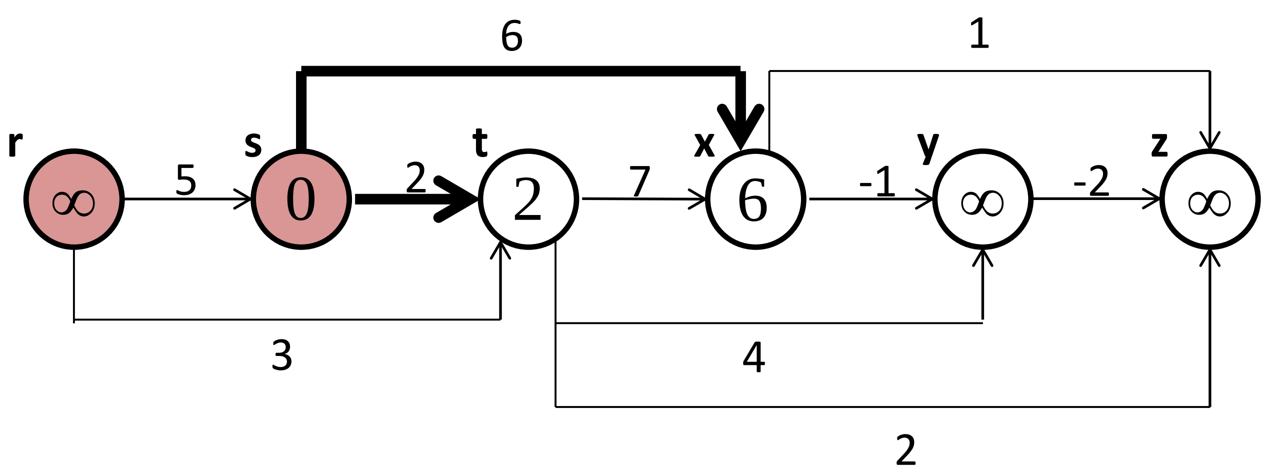
```
DAG-SHORTEST-PATHS(G, source):
    Topologically sort the vertices of G
    Initialize distance to every vertex to ∞
    Initialize distance to source to 0

    for each vertex u taken in topological order
        for each vertex v adjacent to u
            RELAX(u, v)
```

Acyclic Shortest Path Algorithm



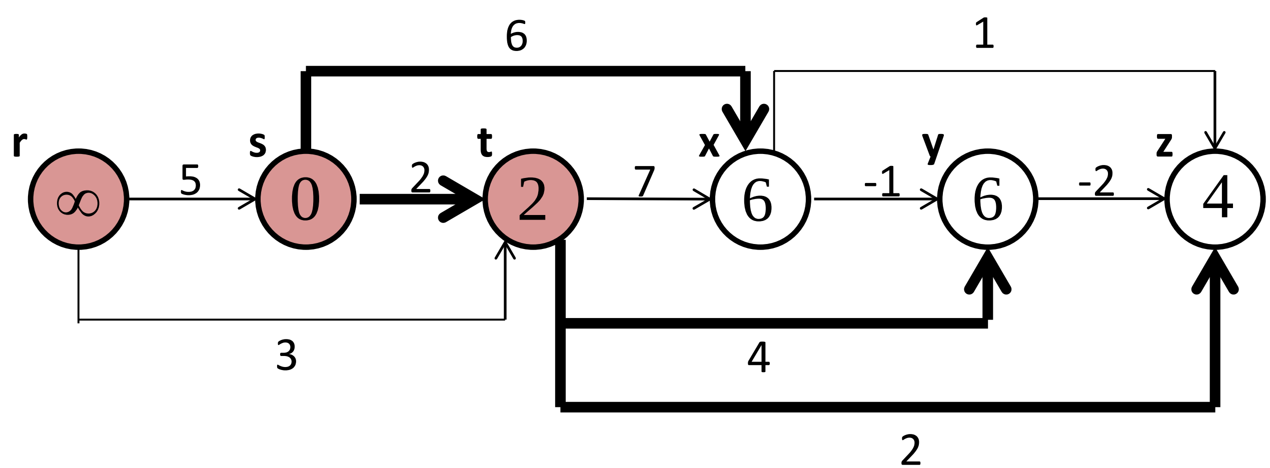
First, topologically sort the vertices (assume source is s). This figure shows after the first iteration of the for loop. The colored vertex, r, was used as *u* in this iteration.

Acyclic Shortest Path Algorithm



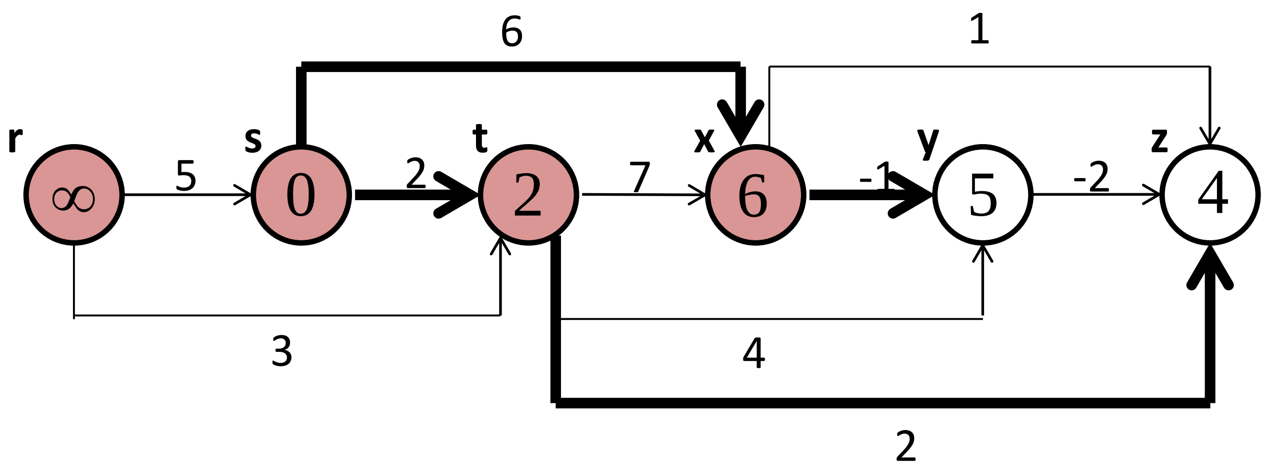
After the second iteration of the for loop.
The colored vertex, s, was used as *u* in this iteration.
The bold edges indicate the shortest path from source.

Acyclic Shortest Path Algorithm



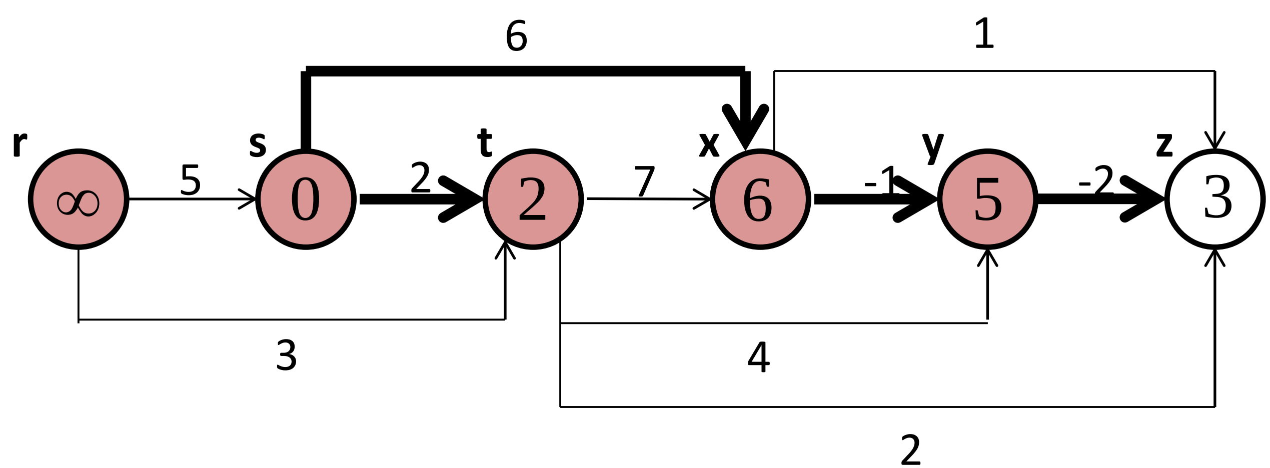
After the third iteration of the for loop.
The colored vertex, t, was used as *u* in this iteration.
The bold edges indicate the shortest path from source.

Acyclic Shortest Path Algorithm



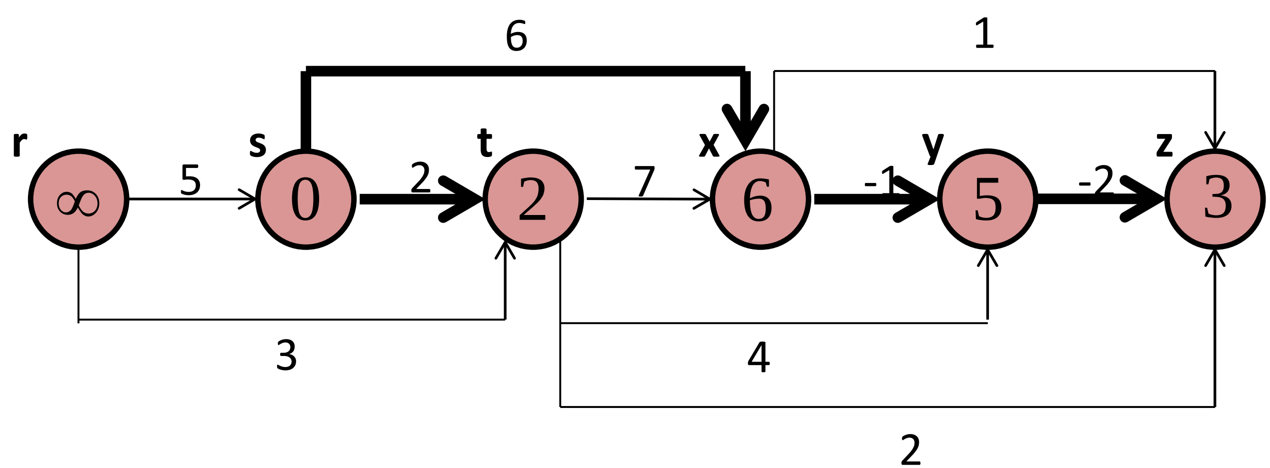
After the fourth iteration of the for loop.
The colored vertex, x, was used as *u* in this iteration.
The bold edges indicate the shortest path from source.

Acyclic Shortest Path Algorithm



After the fifth iteration of the for loop.
The colored vertex, y, was used as *u* in this iteration.
The bold edges indicate the shortest path from source.

Acyclic Shortest Path Algorithm



After the sixth iteration of the for loop (final values).
The colored vertex, z, was used as *u* in this iteration.
The bold edges indicate the shortest path from source.

Analysis of Acyclic SP

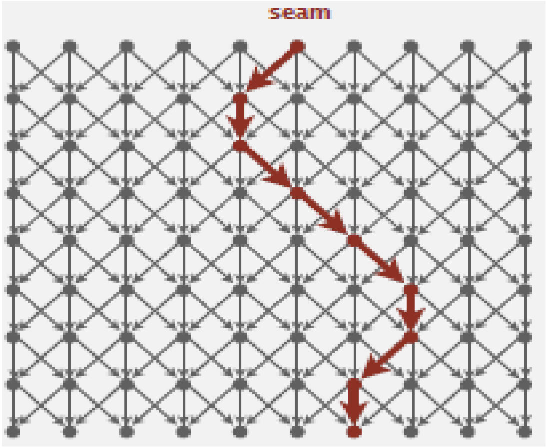
- Topological sort computes a shortest path tree in any edge weighted DAG in time proportional to ***M + N*** (edge weights can be negative!)
  - Each edge is relaxed exactly once (when v is relaxed), leaving distTo[v] ≤ distTo[u] + w(u, v), so total runtime of acyclic SP is $M + N + M \approx M + N$
  - Inequality holds until algorithm terminates:
    - distTo[v] cannot increase because distTo values are monotonically decreasing
    - distTo[u] will not change; no edge pointing to u will be relaxed after u is relaxed because of topological order

Application of Acyclic SP

- Seam carving (Avidan and Shamir): Resize an image for displaying without distortion on a cellphone or web browser
- Enable the user to see the whole image without distortion while scrolling
- Uses DAG shortest path algorithm to find the “shortest path” of pixels through the image (the path that has the lowest energy)
  - The shortest path is almost a column, but not exactly a column

Content-Aware Resizing

- To find vertical seam, create a DAG of pixels:
  - Vertex = pixel; edge = from pixel to 3 downward neighbors
  - Weight of edge = “energy” (difference in gray levels) of neighboring pixels
  - Seam = shortest path (lowest energy) from top to bottom



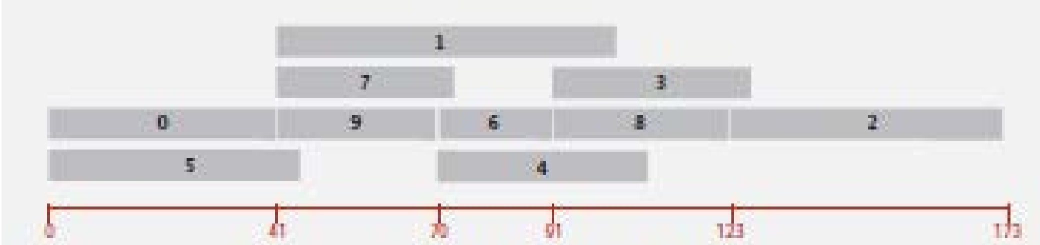
Acyclic Longest Path Algorithm

- The (acyclic) longest path is called the ***critical path***
- Formulate as an acyclic shortest path problem:
  - Negate all initial weights and run the acyclic shortest path (SP) algorithm as is, or
  - Run acyclic SP, replacing ∞ with $-\infty$ in the initialize procedure and $>$ with $<$ in the relax procedure
- Recall that topological sort algorithm works even with negative weights

Application of Acyclic LP

- Goal: Given a set of jobs with durations and precedence constraints, find the *minimum* amount of time required for all jobs to complete (i.e., find the bottleneck)
  - Some jobs must be done before others, and some jobs may be performed simultaneously

job	*duration*	*must complete before*
0	41.0	1 7 9
1	51.0	2
2	50.0	
3	36.0	
4	38.0	
5	45.0	
6	21.0	3 8
7	32.0	3 8
8	32.0	2
9	29.0	4 6

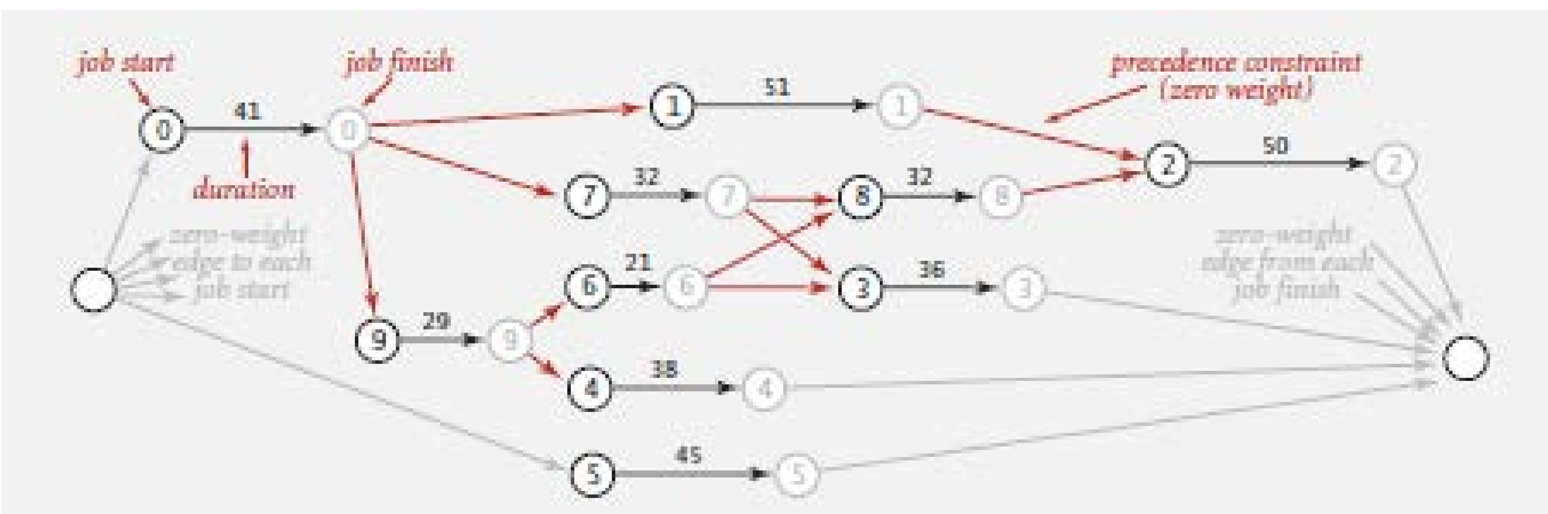


Parallel job scheduling solution

Application of Acyclic LP

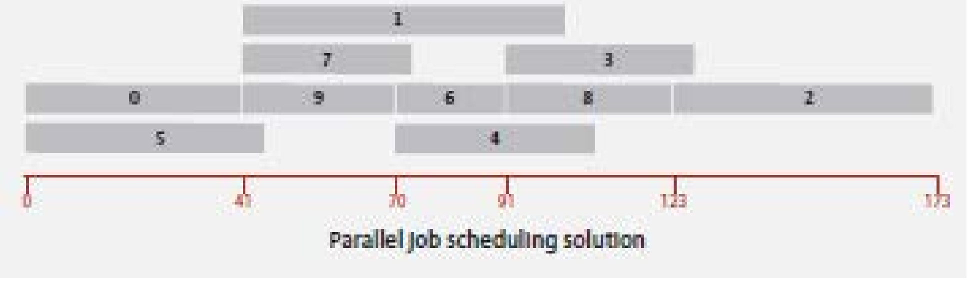
- Create a weighted DAG with source and sink vertices
- Have two vertices (start and finish) for each job
- Have three edges for each job:
  - source to start (0 weight)
  - start to finish (weighted by duration of job)
  - finish to sink (0 weight)
- Have one edge for each precedence constraint (0 weight)

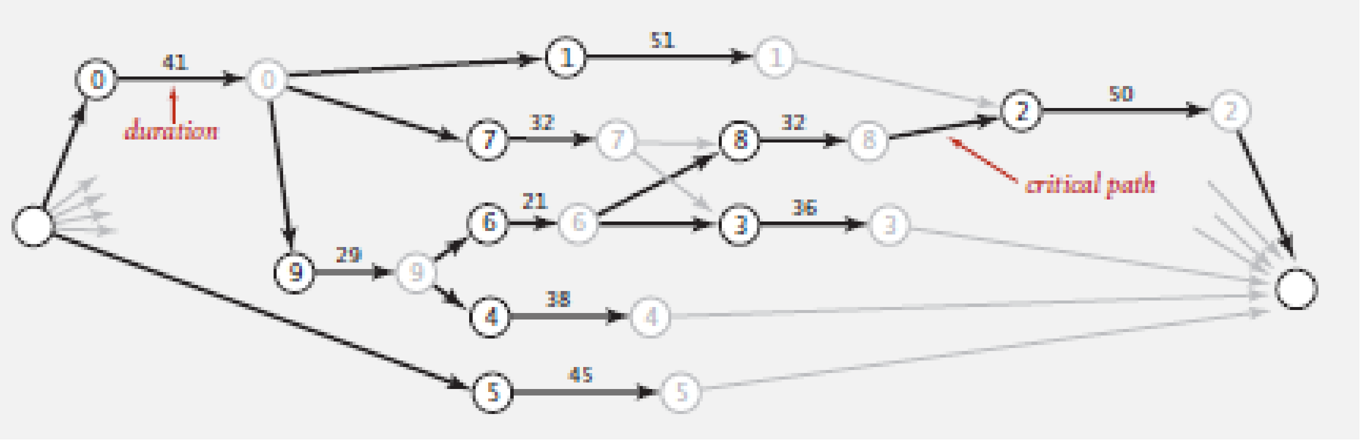
job	duration	must complete before
0	41.0	1 7 9
1	51.0	2
2	50.0	
3	36.0	
4	38.0	
5	45.0	
6	21.0	3 8
7	32.0	3 8
8	32.0	2
9	29.0	4 6



Application of Acyclic LP

- Now run the “modified” acyclic SP algorithm to get acyclic LP
- The acyclic longest path from the source to the destination is equal to the overall minimum completion time (the bottleneck)





Difference Constraints

- Goal: optimize a linear function subject to a set of linear inequalities
  - Given an M x N matrix $\boldsymbol{A}$, an M-vector $\boldsymbol{b}$, we wish to find a vector $\boldsymbol{x}$ of N elements that maximizes an objective function, subject to the M constraints given by $\boldsymbol{Ax} \leq \boldsymbol{b}$
  - This problem can be reduced to finding the shortest paths from a single source

Difference Constraints

For example, find the 5-element vector $\boldsymbol{x}$ that satisfies:

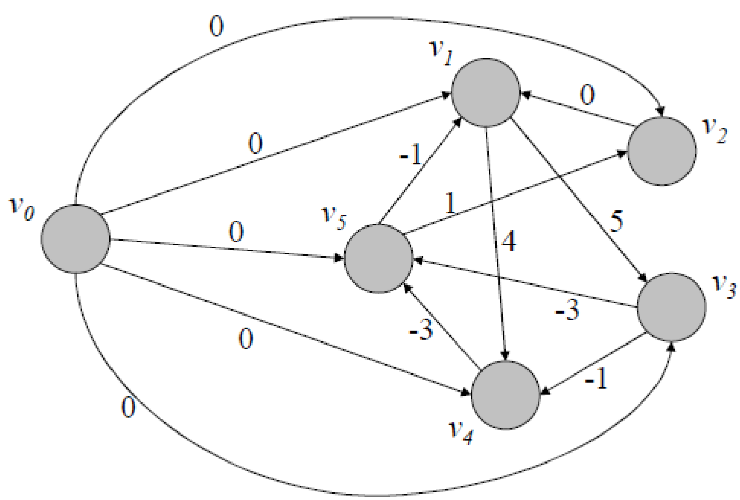
$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

This problem is equivalent to finding values for the unknowns x_1, x_2, x_3, x_4, x_5 satisfying these 8 difference constraints:

$$\begin{aligned} x_1 - x_2 &\leq 0 \\ x_1 - x_5 &\leq -1 \\ x_2 - x_5 &\leq 1 \\ x_3 - x_1 &\leq 5 \\ x_4 - x_1 &\leq 4 \\ x_4 - x_3 &\leq -1 \\ x_5 - x_3 &\leq -3 \\ x_5 - x_4 &\leq -3 \end{aligned}$$

Difference Constraints

Create a *constraint graph* with an additional vertex v_0 to guarantee that the graph has a vertex which can reach all other vertices. Include a vertex v_i for each unknown x_i. The edge set contains an edge for each difference constraint. Then run the Bellman-Ford algorithm from v_0.



$$
\begin{aligned}
x_1 - x_2 &\le 0 \\
x_1 - x_5 &\le -1 \\
x_2 - x_5 &\le 1 \\
x_3 - x_1 &\le 5 \\
x_4 - x_1 &\le 4 \\
x_4 - x_3 &\le -1 \\
x_5 - x_3 &\le -3 \\
x_5 - x_4 &\le -3
\end{aligned}
$$

One feasible solution to this problem is x = (-5, -3, 0, -1, -4).