EBERHARD KARLS UNIVERSITÄT TÜBINGEN

FACULTY OF HUMANITIES

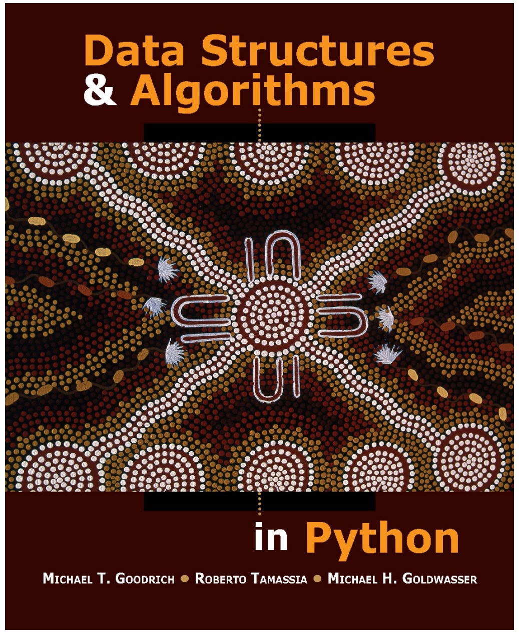
Department of General and Computational Linguistics

Shortest Paths

Data Structures and Algorithms for CL III, WS 2019-2020

Corina Dima

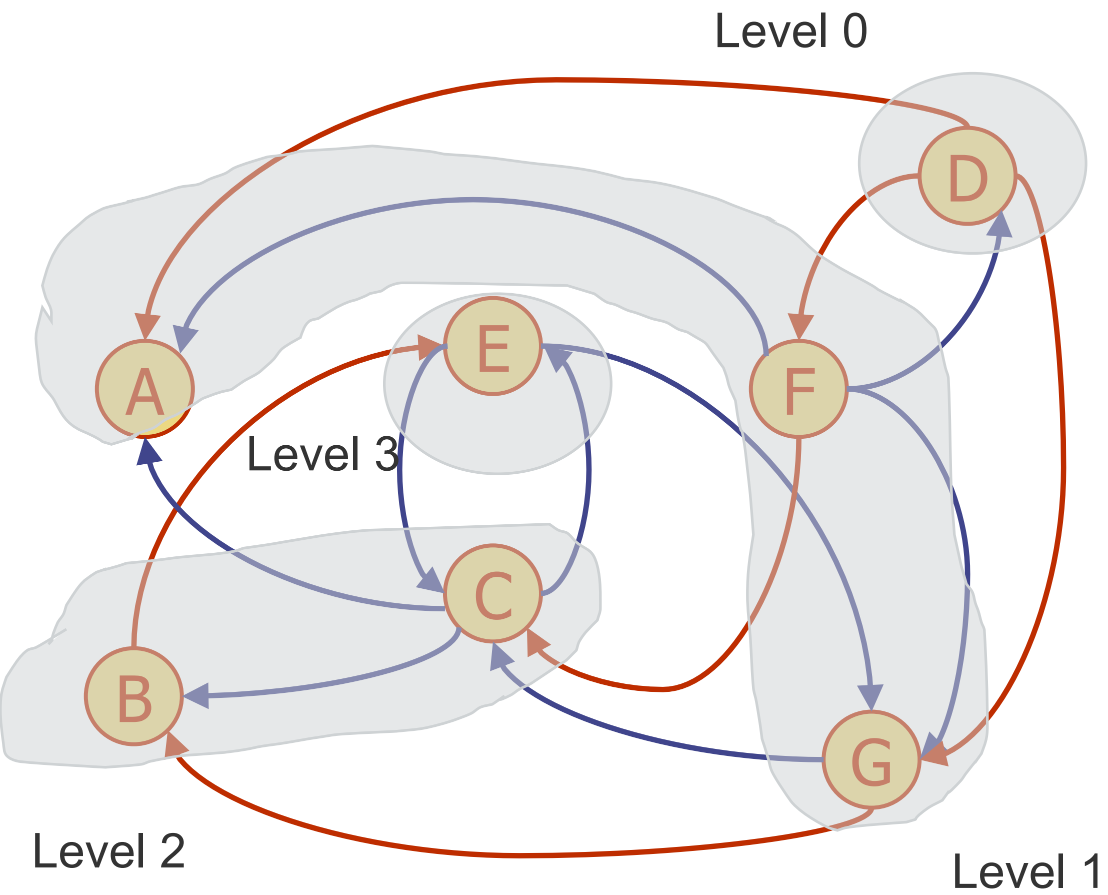
corina.dima@uni-tuebingen.de


Data Structures
& Algorithms
in Python
MICHAEL T. GOODRICH ● ROBERTO TAMASSIA ● MICHAEL H. GOLDWASSER

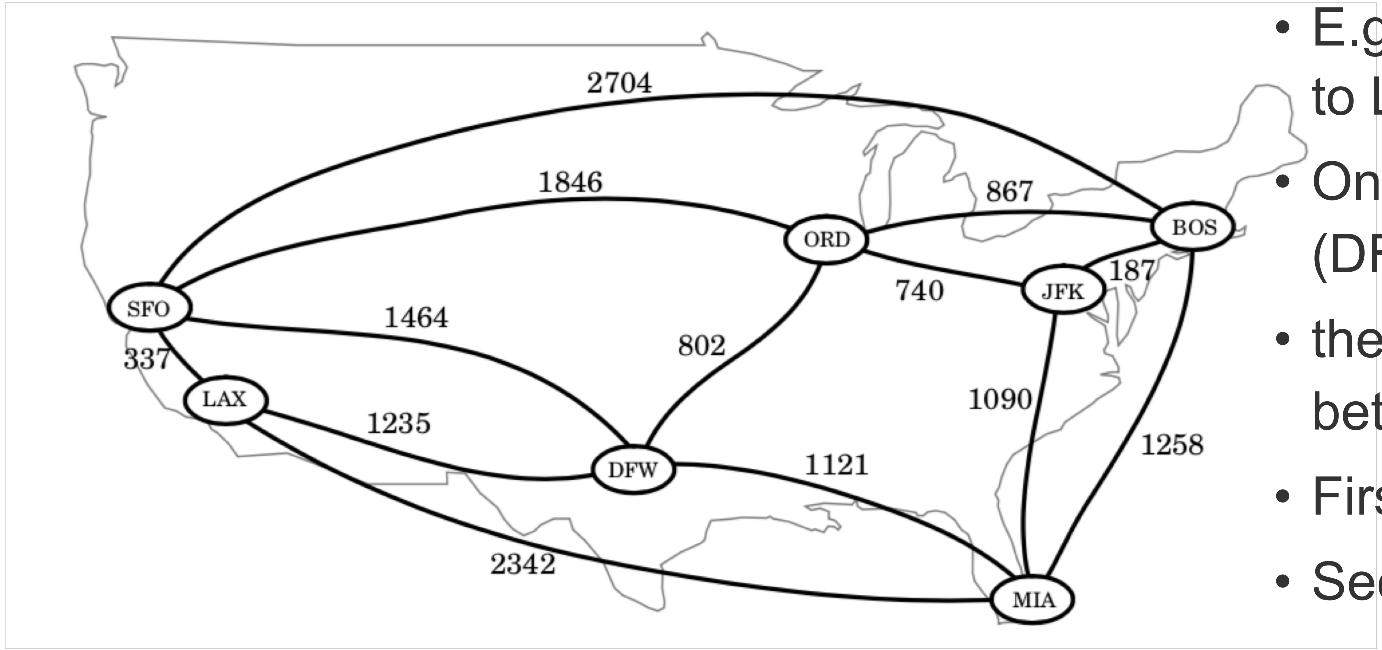
14.6 Shortest Paths

Shortest Paths

- A BFS traversal can be used to find the shortest path from some starting vertex to every other vertex in a connected graph
    - E.g. shortest path from vertex D to to E is 3 edges
- If any edge is as good as other – this approach works



Shortest Paths (cont'd)



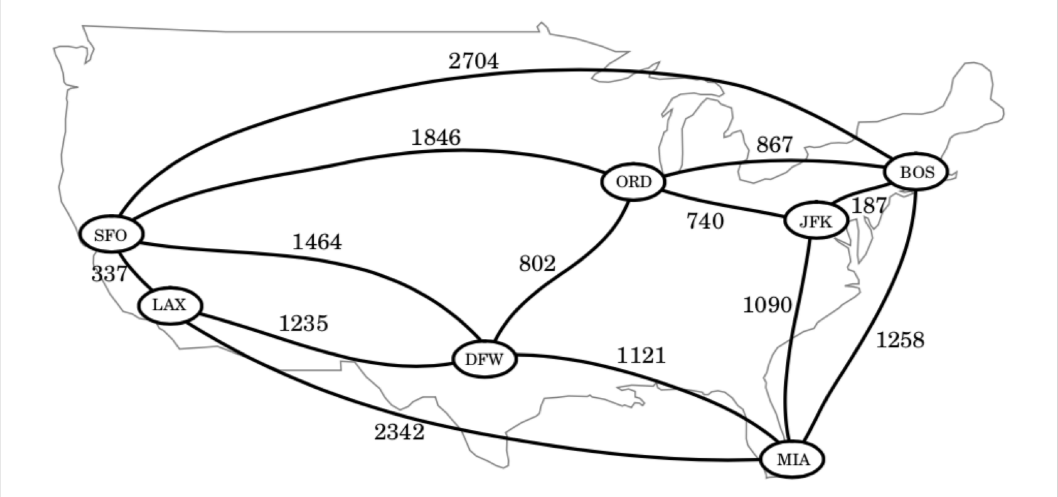
- But there are also cases when some edges might be preferred to others
- E.g. two flights going from New York (JFK) to Los Angeles (LAX)
- One visits Chicago (ORD) and Dallas (DFW) in between
- the other only visits Miami (MIA) in between
- First path: $740 + 802 + 1235 = 2777$ miles
- Second path: $1090 + 2342 = 3432$ miles
- The first path is the minimum weight path in the graph from JFK to LAX

Weighted Graphs

- A weighted graph is a graph that has a numeric (integer, float) label $w(e)$ associated with each edge e, called the weight (or cost) of the edge
- For $e = (u, v)$ we use the notation $w(u, v) = w(e)$
- The length (or weight) of a path P is the sum of the weights on the edges of P
- If $P = \big((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\big)$, then the length of P, $w(P)$ is

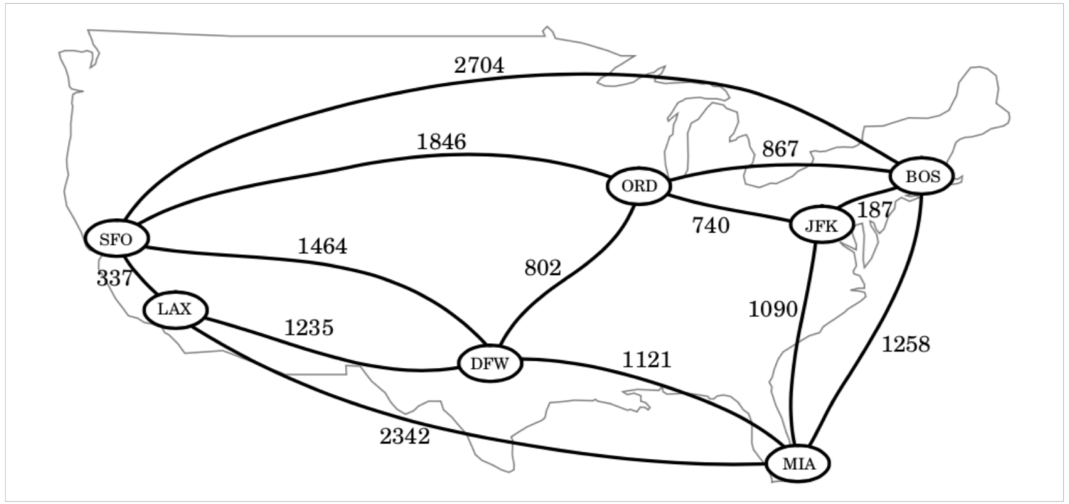
$$w(P) = \sum_{i=0}^{k-1} w(v_i\,, v_{i+1})$$

- The distance from a vertex u to a vertex v in G, $d(u, v)$, is the length of a minimum-length path (shortest path) from u to v, if such a path exists
  - The distance from JFK to LAX is 2777 miles



Weighted Graphs (cont'd)

- By convention, $d(u, v) = \infty$ if there is no path from u to v in the graph
- If there is a negative-weight cycle in the graph, then $d(u, v)$ might not be defined
  - If someone is paying us to fly from New York (JFK) to Chicago (ORD), then the weight of the path is -740
  - If they also pay us to go back to ORD, then we have a negative cycle, where each pass through the cycle modifies the length of any other path by -1480
  - Any path in this graph could always be made "cheaper" by going through the negative cycle (an infinite number of times)
- So when modelling shortest paths we should be careful not to introduce negative weight cycles



Shortest Path Problem

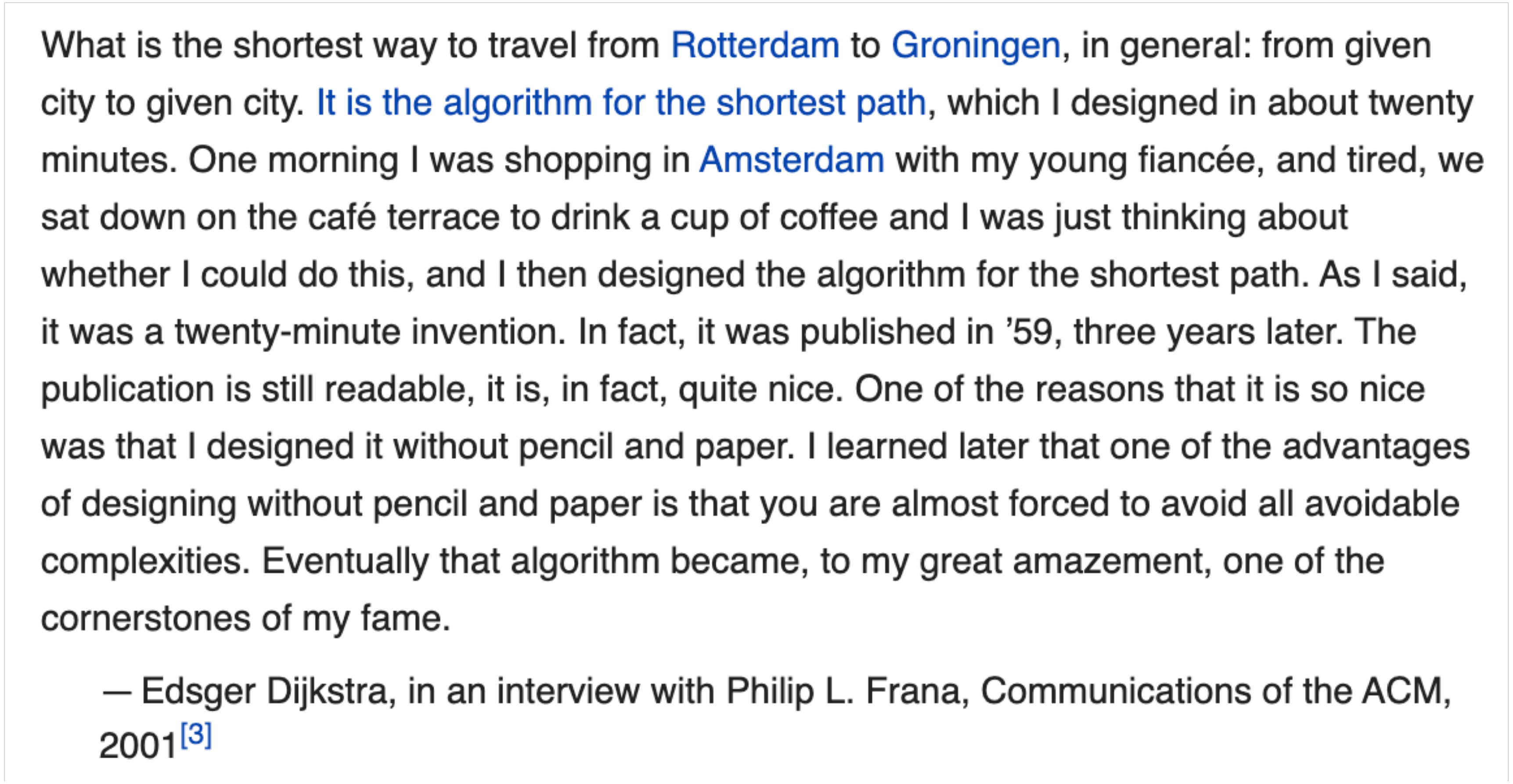
- Given a weighted digraph $\vec{G}$, find the shortest path from some vertex s to each other vertex of $\vec{G}$, viewing the weights on the edges as distances
- Also called the single-source shortest path problem
- Explore several algorithms for finding the shortest path, that make different assumptions

I. All edge weights are non-negative (but there might be cycles)

II. The digraph doesn't have cycles (but there might be negative edges)

III. The digraph can have both cycles and negative weights, but no negative cycles

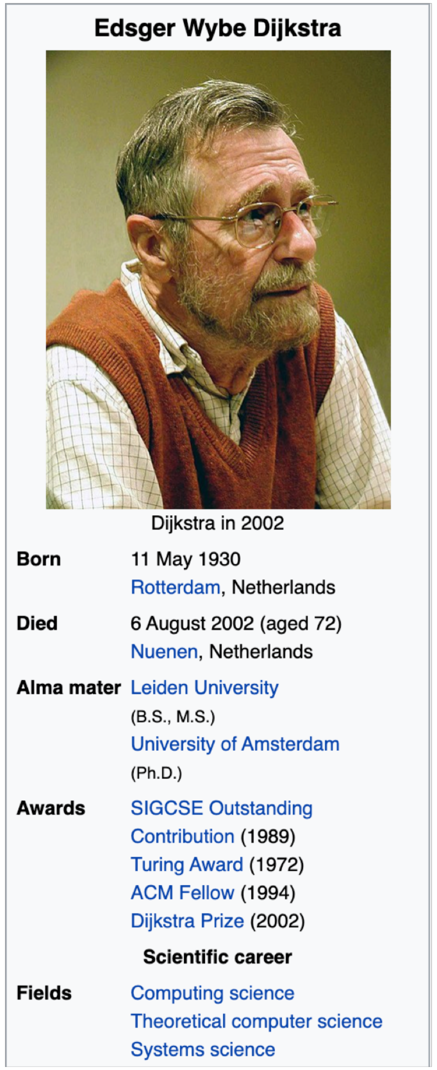
Dijkstra’s Algorithm



What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, which I designed in about twenty minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention. In fact, it was published in ’59, three years later. The publication is still readable, it is, in fact, quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities. Eventually that algorithm became, to my great amazement, one of the cornerstones of my fame.

— Edsger Dijkstra, in an interview with Philip L. Frana, Communications of the ACM, 2001[3]

https://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf



Edsger Wybe Dijkstra

Dijkstra in 2002

Born	11 May 1930 Rotterdam, Netherlands
Died	6 August 2002 (aged 72) Nuenen, Netherlands
Alma mater	Leiden University (B.S., M.S.) University of Amsterdam (Ph.D.)
Awards	SIGCSE Outstanding Contribution (1989) Turing Award (1972) ACM Fellow (1994) Dijkstra Prize (2002)
Scientific career	
Fields	Computing science Theoretical computer science Systems science

Dijkstra’s Algorithm

- Use a greedy method to solve the shortest path problem
- Intuitively, perform a “weighted” BFS from a start vertex s
  - The weights on the edges incident to s provide the means to decide which edge should be followed next
- The algorithm iteratively grows a “cloud” of vertices starting at s
  - The vertices join the “cloud” in the order of their distance from s
  - At each iteration, the next vertex to be chosen is the vertex that is closest to s
  - The algorithm terminates when there are no more vertices to be added to the “cloud”, or when the remaining vertices are not connected to the vertices in the “cloud”
  - At this point we have a shortest path from s to each vertex of that graph G that is reachable from s

Edsger Wybe Dijkstra

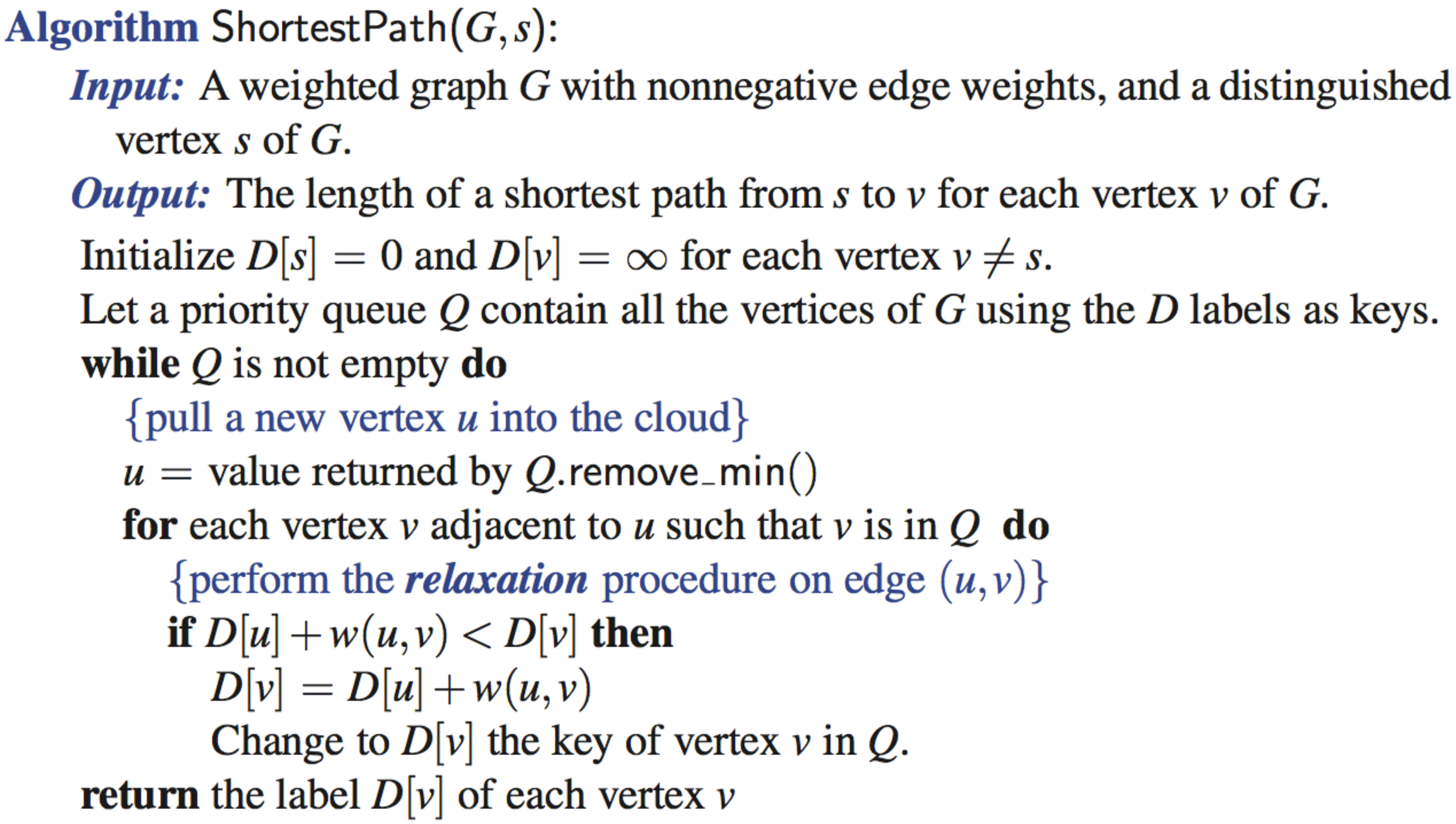
Dijkstra in 2002

Born	11 May 1930 Rotterdam, Netherlands
Died	6 August 2002 (aged 72) Nuenen, Netherlands
Alma mater	Leiden University (B.S., M.S.) University of Amsterdam (Ph.D.)
Awards	SIGCSE Outstanding Contribution (1989) Turing Award (1972) ACM Fellow (1994) Dijkstra Prize (2002)

Scientific career

Fields	Computing science Theoretical computer science Systems science

Dijkstra’s Algorithm - Pseudocode



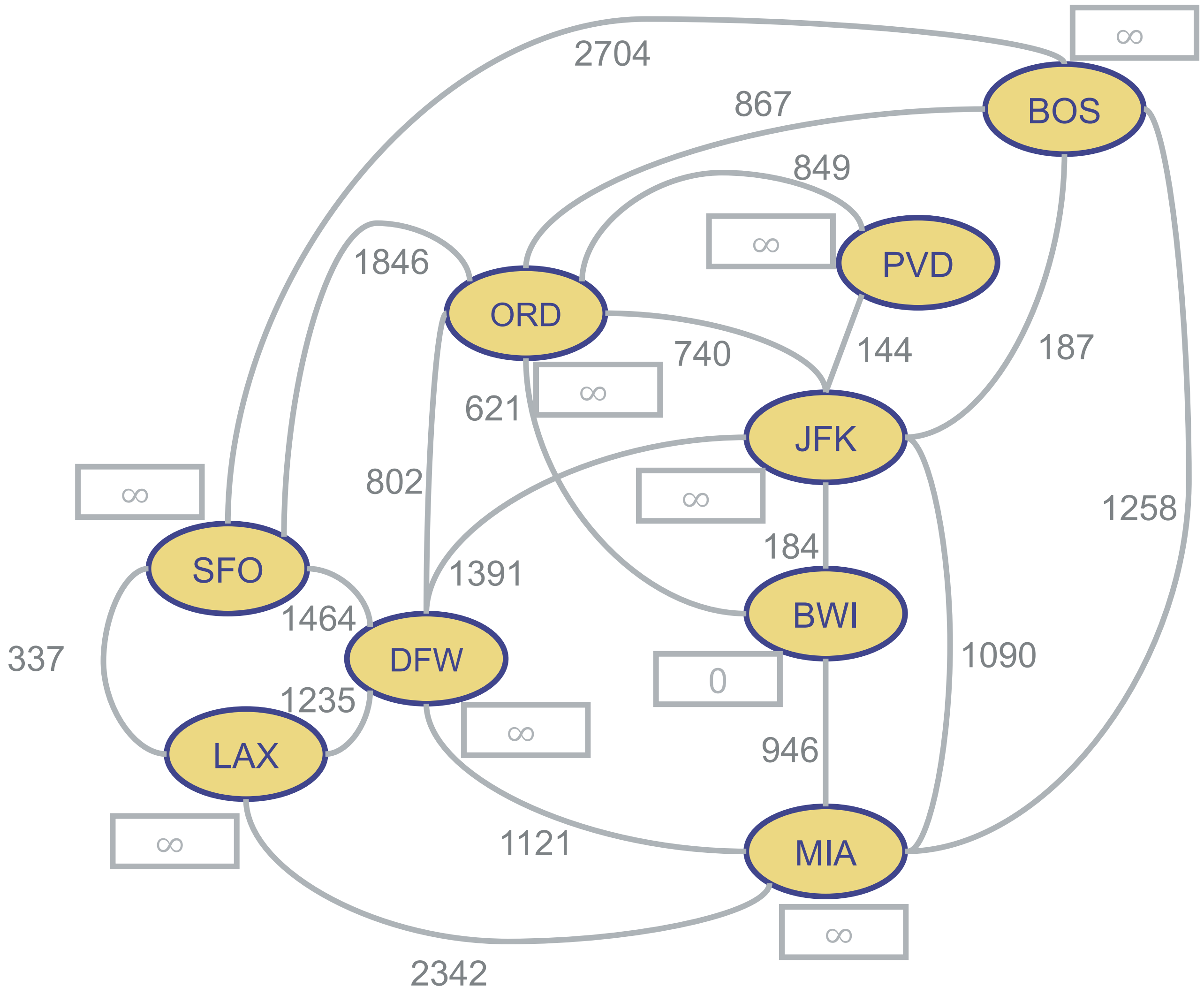
```
Algorithm ShortestPath($G, s$):
    Input: A weighted graph $G$ with nonnegative edge weights, and a distinguished
      vertex $s$ of $G$.
    Output: The length of a shortest path from $s$ to $v$ for each vertex $v$ of $G$.
  Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
  Let a priority queue $Q$ contain all the vertices of $G$ using the $D$ labels as keys.
  while $Q$ is not empty do
    {pull a new vertex $u$ into the cloud}
    $u$ = value returned by $Q$.remove_min()
    for each vertex $v$ adjacent to $u$ such that $v$ is in $Q$ do
      {perform the relaxation procedure on edge $(u, v)$}
      if $D[u] + w(u, v) < D[v]$ then
        $D[v] = D[u] + w(u, v)$
        Change to $D[v]$ the key of vertex $v$ in $Q$.
  return the label $D[v]$ of each vertex $v$
```

Edge Relaxation

- $D[v]$ approximates the distance from s to each vertex v in the graph G
- $D[v]$ will always store the length of the best path found so far from s to v
- Initially, $D[s] = 0$ and $D[v] = \infty$ for every v of G other than s
- C is the set of “cloud” vertices – initially empty
- At each iteration:
  - Select the vertex u not in C such that $D[u]$ is the smallest distance of all remaining vertices (so the closest vertex from the vertices that are not yet in the “cloud”)
  - Add u to C, the “cloud”
  - Update $D[v]$ for every vertex v adjacent to u and is outside of C, since it might be a better way to get to v via u; this update is called edge relaxation:

$$\text{if } D[u] + w(u,v) < D(v) \text{ then}$$
$$D[v] = D[u] + w(u,v)$$

Dijkstra’s Algorithm - Example



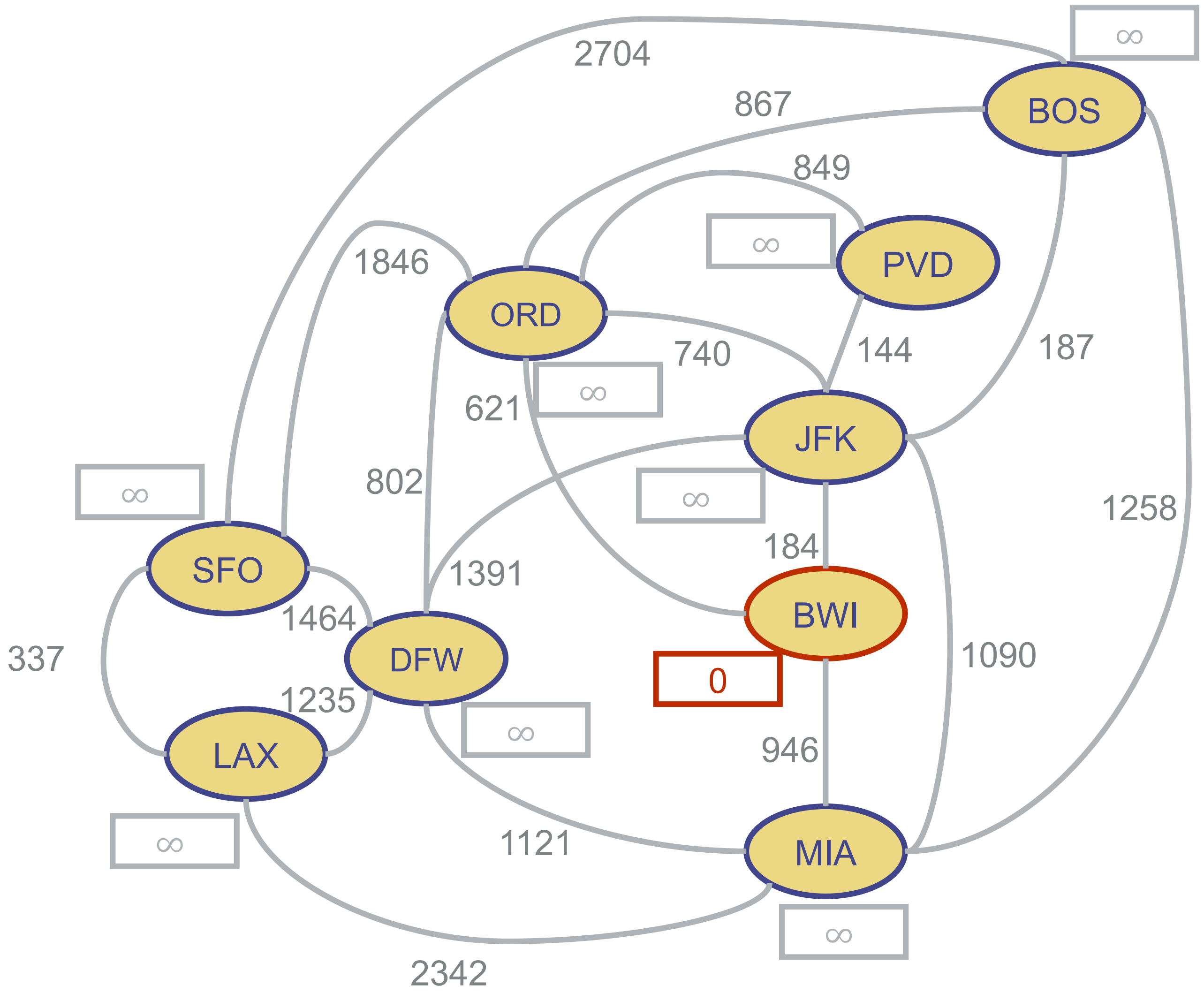
PQ		C (“cloud”)
BOS	∞	
PVD	∞	
JFK	∞	
BWI	0	
MIA	∞	
ORD	∞	
DFW	∞	
SFO	∞	
LAX	∞	

↑

$D[v]$

- Start vertex is BWI, the only one with length 0

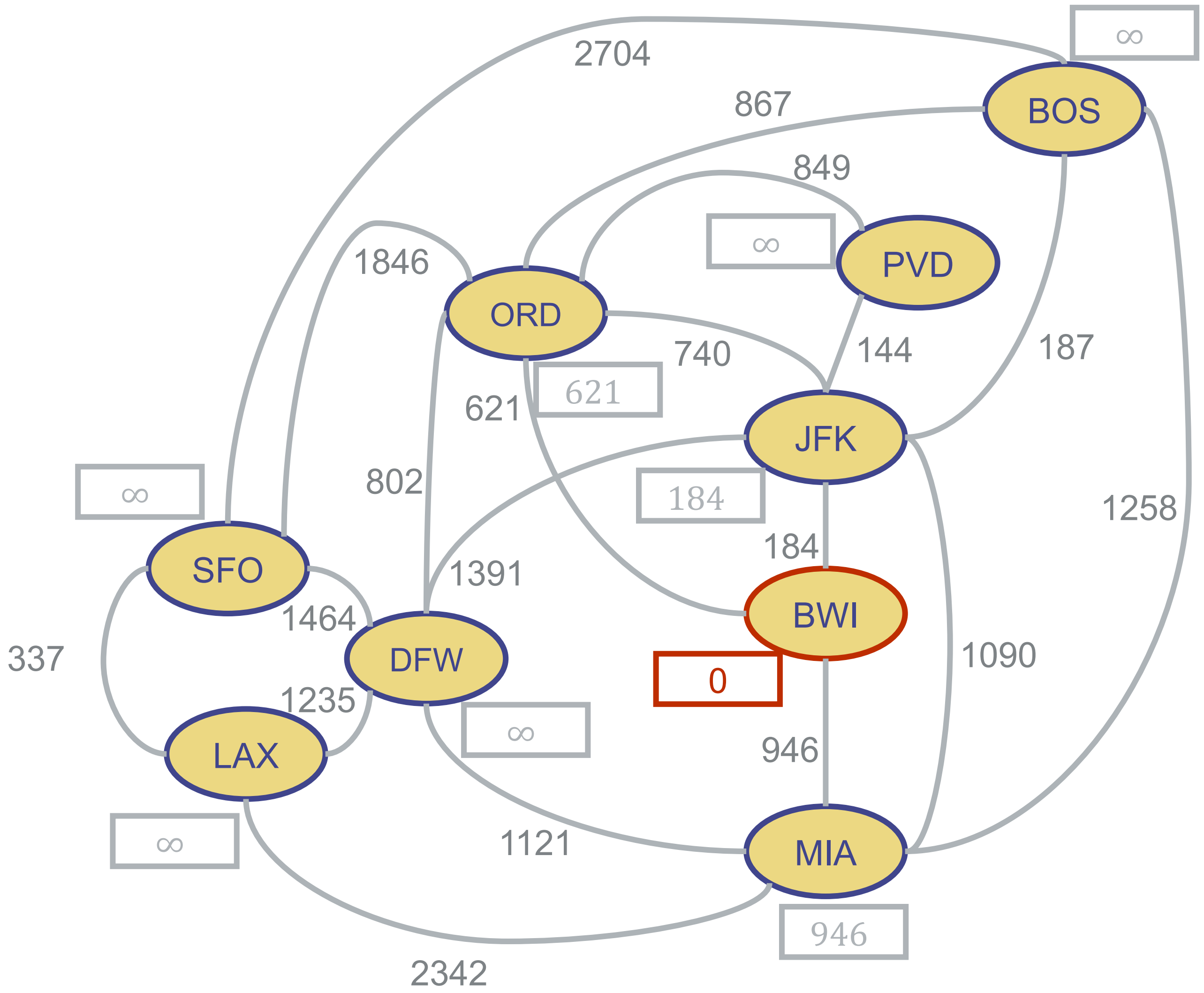
Dijkstra’s Algorithm - Example



PQ		C (“cloud”)	
BOS	∞	BWI	0
PVD	∞		
JFK	∞		
MIA	∞		
ORD	∞		
DFW	∞		
SFO	∞		
LAX	∞		

- Remove minimum length vertex, BWI, from priority queue, and add it to cloud
- Vertices in the “cloud” are marked red

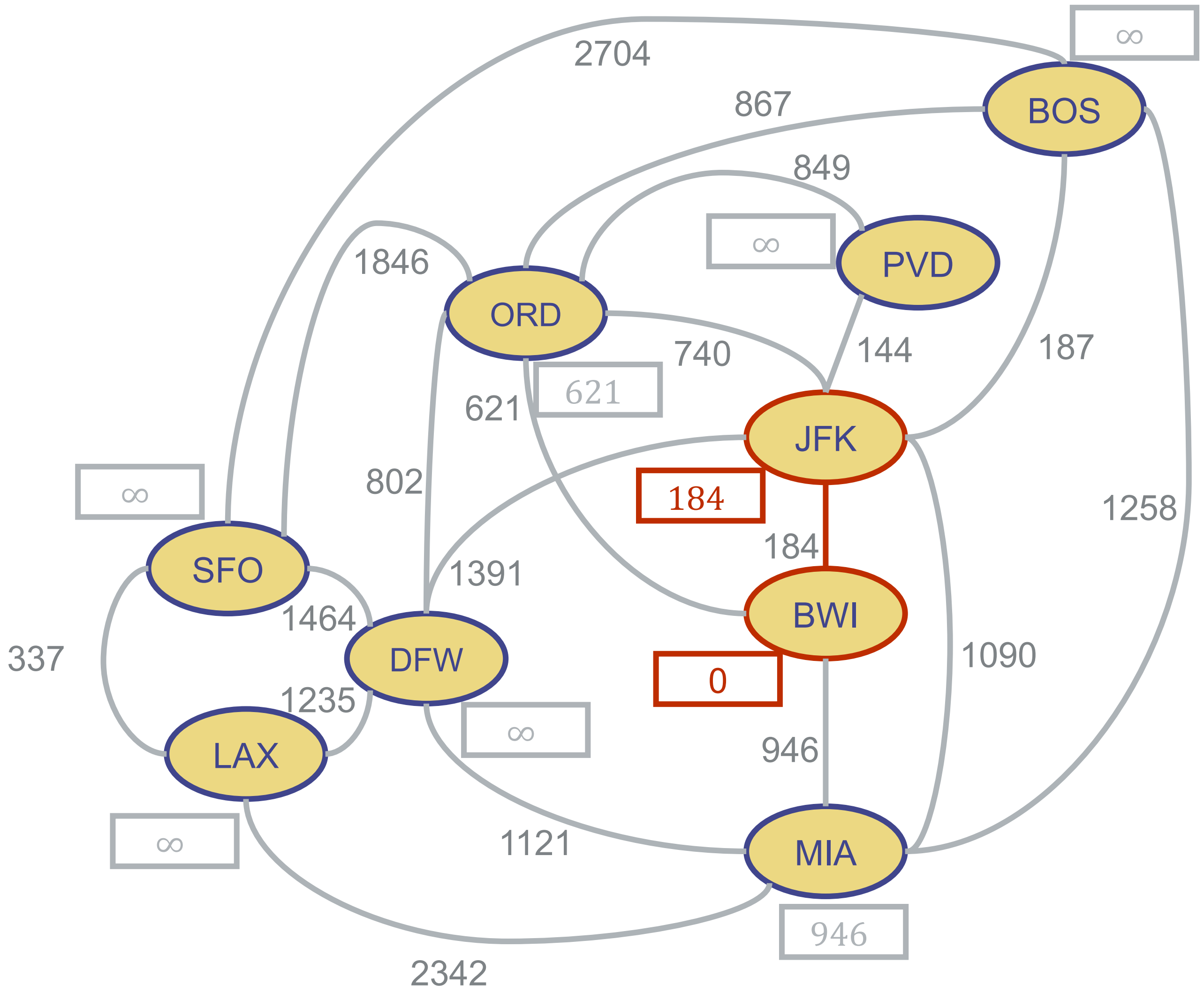
Dijkstra’s Algorithm - Example



PQ		C (“cloud”)	
BOS	∞	BWI	0
PVD	∞		
JFK	184		
MIA	946		
ORD	621		
DFW	∞		
SFO	∞		
LAX	∞		

- Update the lengths of the paths from BWI to all the vertices adjacent to BWI which are not in the cloud
- Edge relaxation for JFK, ORD and MIA
  - $0 + 184 < \infty$, update JFK path
  - $0 + 946 < \infty$, update MIA path
  - $0 + 621 < \infty$, update ORD path

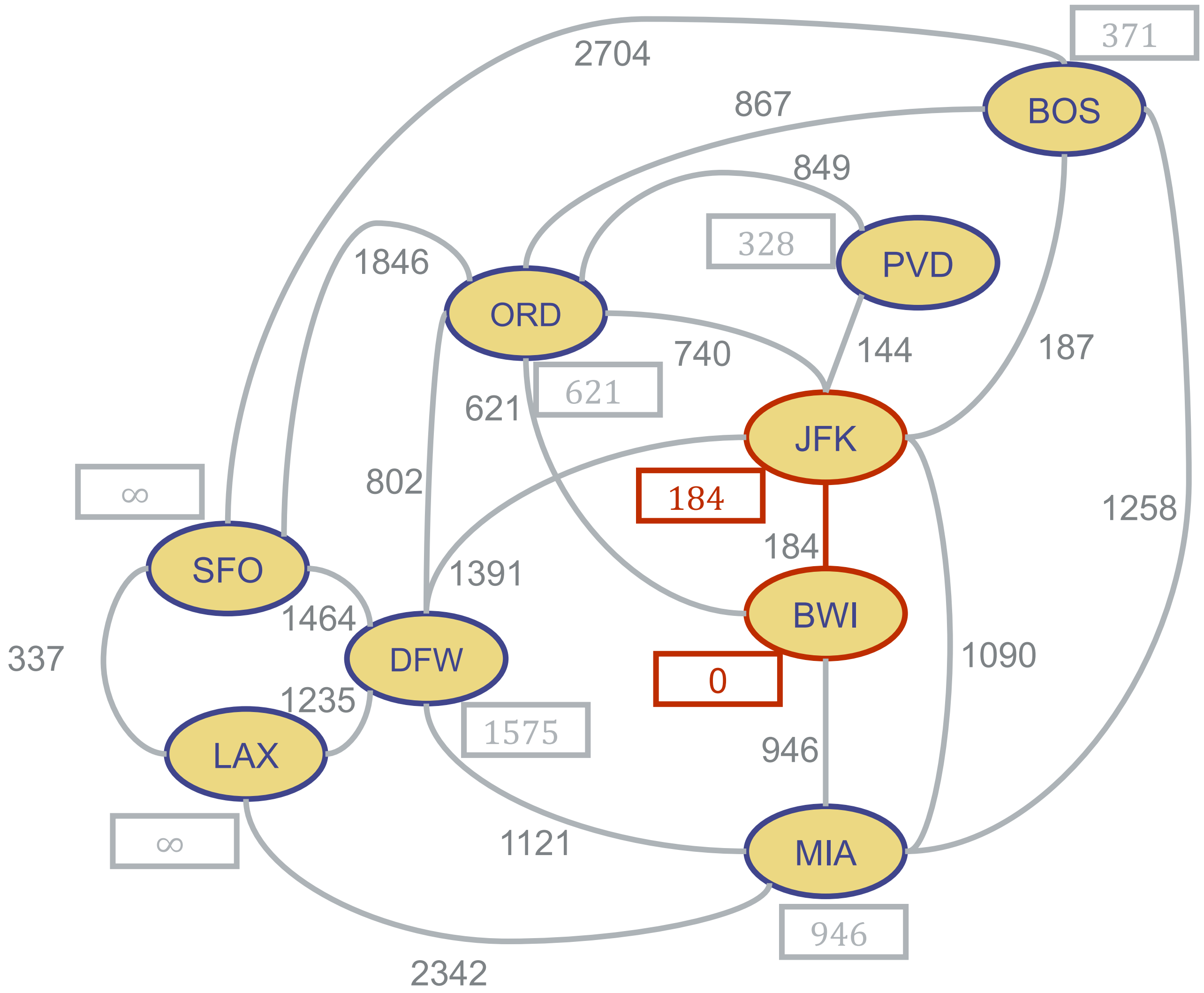
Dijkstra’s Algorithm - Example



PQ		C (“cloud”)	
BOS	∞	BWI	0
PVD	∞	JFK	184
MIA	946		
ORD	621		
DFW	∞		
SFO	∞		
LAX	∞		

- Remove next vertex with minimum path from PQ – JFK, and add it to cloud

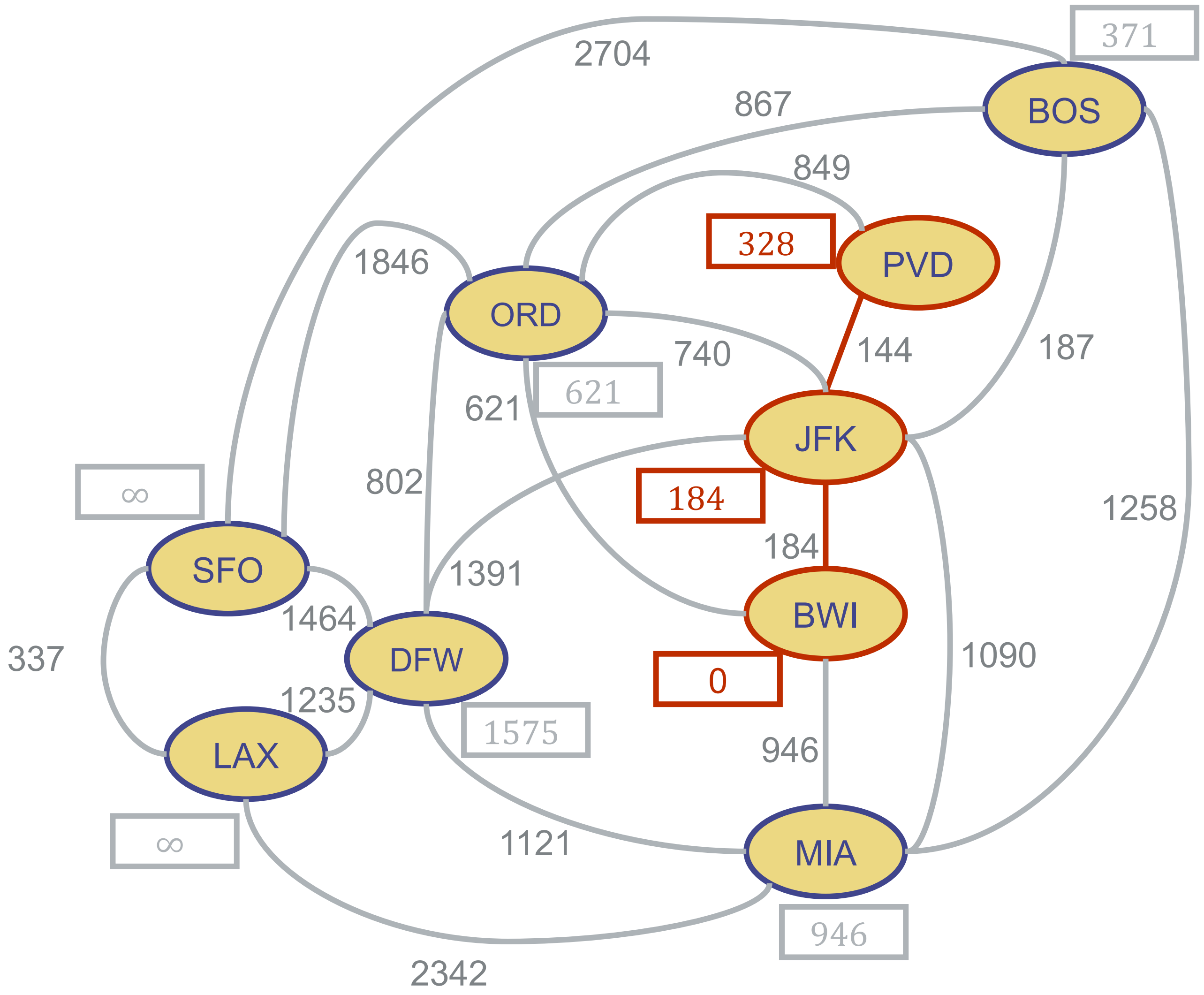
Dijkstra’s Algorithm - Example



PQ		C (“cloud”)	
BOS	371	BWI	0
PVD	328	JFK	184
MIA	946		
ORD	621		
DFW	1575		
SFO	∞		
LAX	∞		

- Update the lengths of the paths from BWI to all the vertices adjacent to JFK which are not in the cloud
- Edge relaxation for BOS, PVD, ORD, DFW and MIA
    - $184 + 187 < \infty$, update BOS path
    - $184 + 144 < \infty$, update PVD path
    - $184 + 1090 > 946$, keep existing MIA path
    - $184 + 740 > 621$, keep existing ORD path
    - $184 + 1391 < \infty$, update DFW path

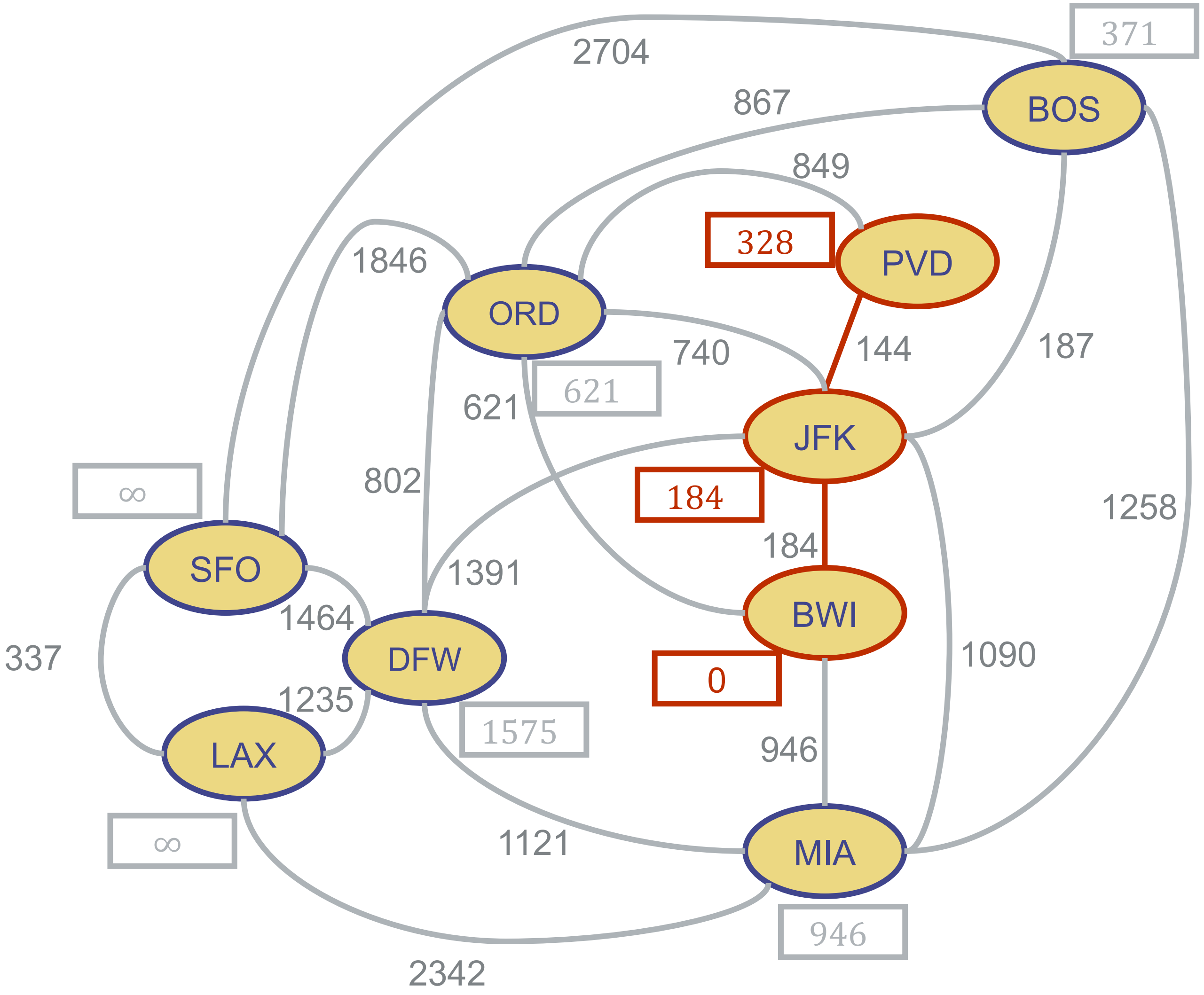
Dijkstra’s Algorithm - Example



PQ		C (“cloud”)	
BOS	371	BWI	0
MIA	946	JFK	184
ORD	621	PVD	328
DFW	1575		
SFO	∞		
LAX	∞		

- Remove next vertex with minimum path from PQ – PVD, and add it to cloud

Dijkstra’s Algorithm - Example

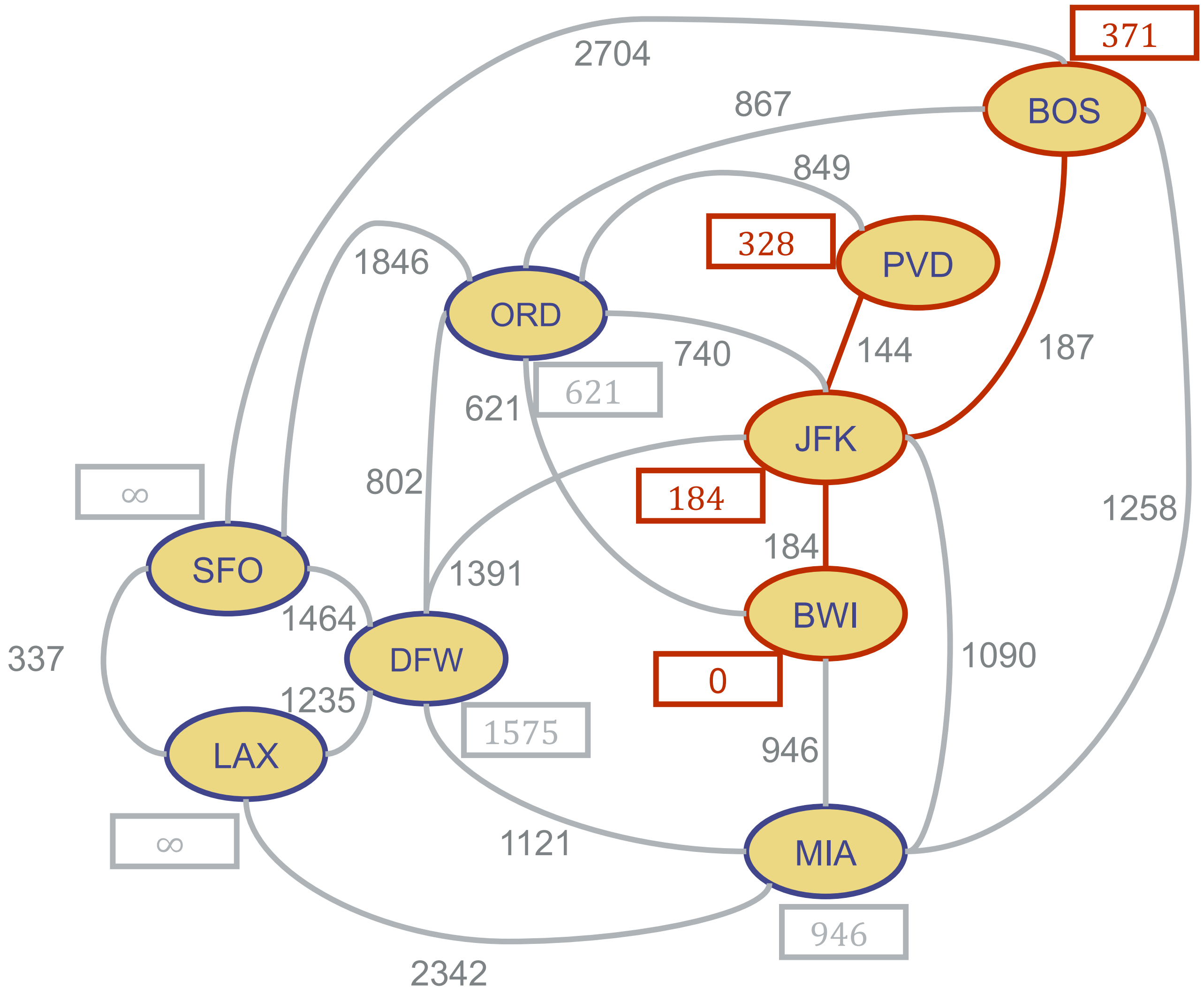


PQ		C (“cloud”)	
BOS	371	BWI	0
MIA	946	JFK	184
ORD	621	PVD	328
DFW	1575		
SFO	∞		
LAX	∞		

- Update the lengths of the paths from BWI to all the vertices adjacent to PVD which are not in the cloud
- Edge relaxation for ORD
    - $328 + 849 > 621$, keep existing ORD path

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Dijkstra's Algorithm - Example

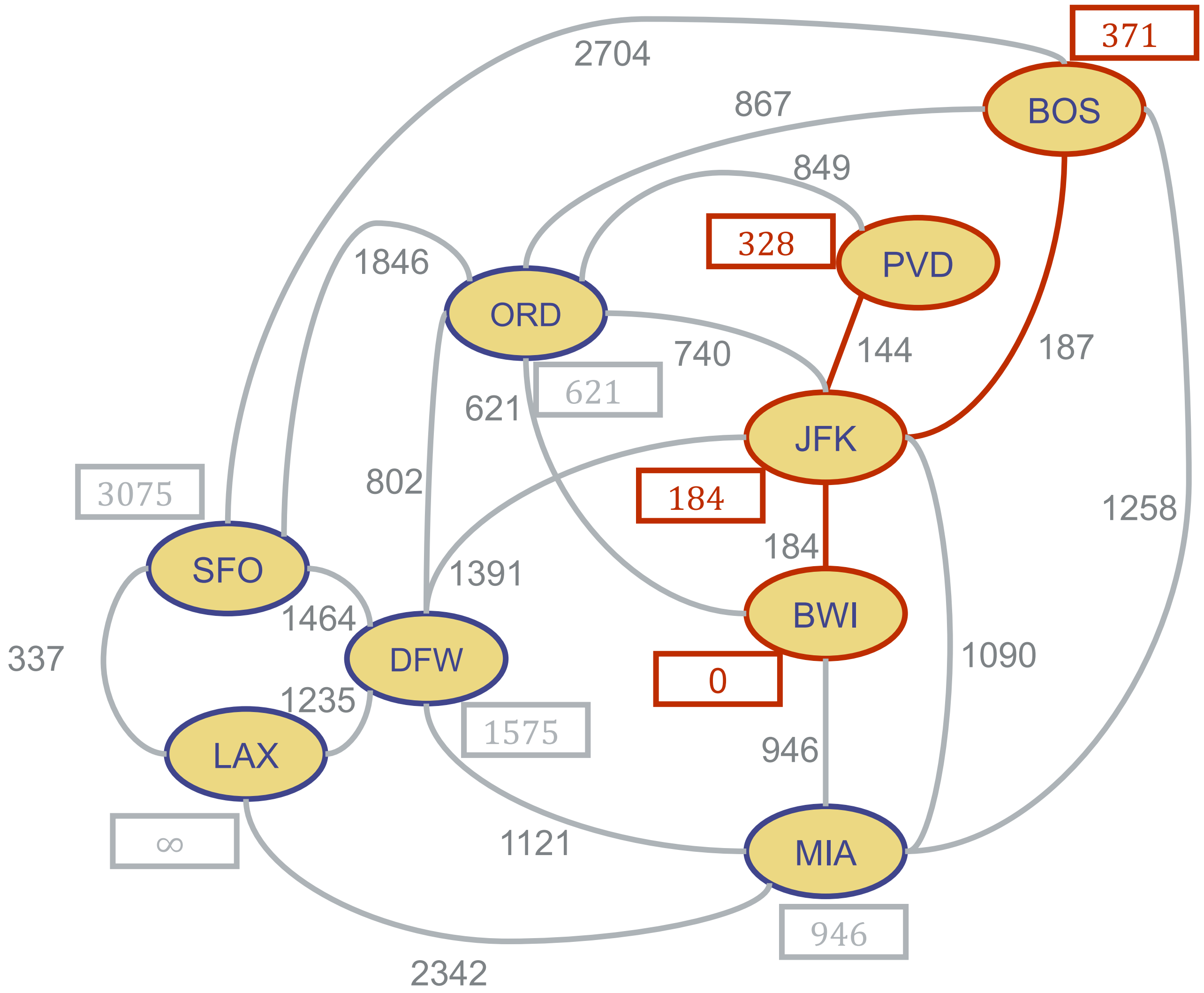


PQ		C ("cloud")	
MIA	946	BWI	0
ORD	621	JFK	184
DFW	1575	PVD	328
SFO	∞	BOS	371
LAX	∞		

- Remove next vertex with minimum path from PQ – BOS, and add it to cloud

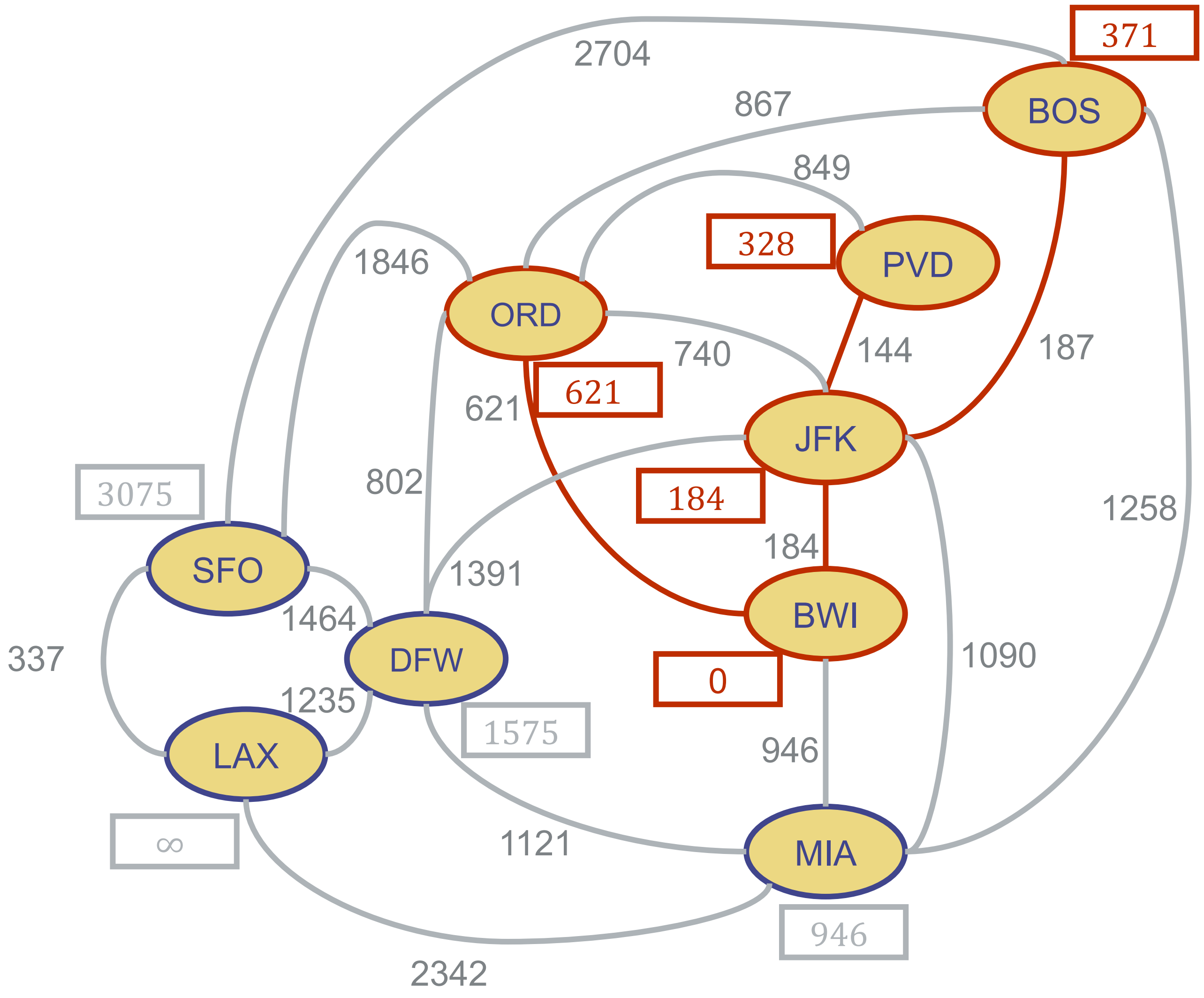
Dijkstra’s Algorithm - Example

PQ		*C* (“cloud”)	
MIA	946	BWI	0
ORD	621	JFK	184
DFW	1575	PVD	328
SFO	3075	BOS	371
LAX	∞		



- Update the lengths of the paths from BWI to all the vertices adjacent to BOS which are not in the cloud
- Edge relaxation for ORD, SFO and MIA
  - $371 + 867 > 621$, keep existing ORD path
  - $371 + 2704 < \infty$, update SFO path
  - $371 + 1258 > 946$, keep existing MIA path

Dijkstra’s Algorithm - Example

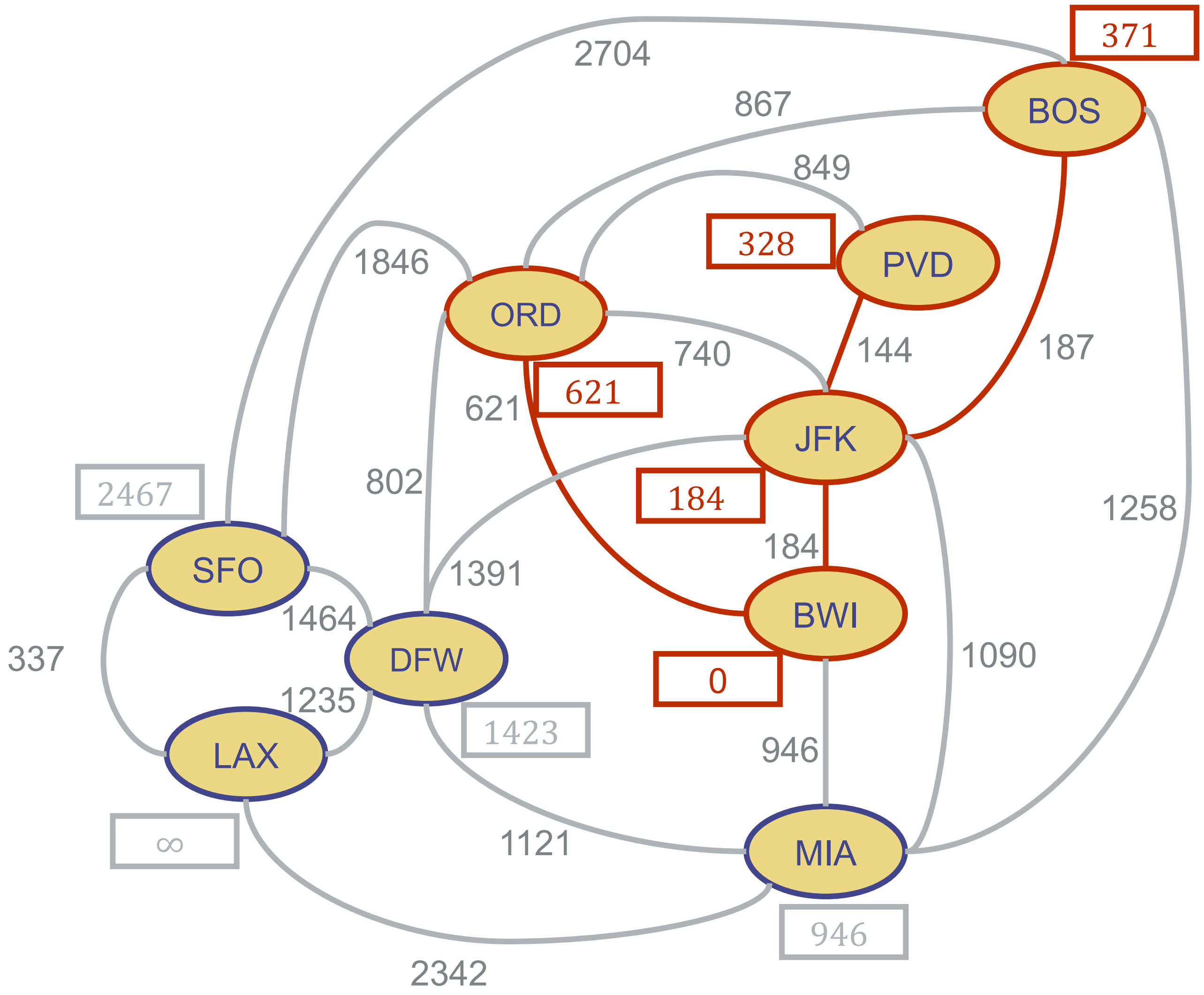


PQ		***C*** **(“cloud”)**	
MIA	946	BWI	0
DFW	1575	JFK	184
SFO	3075	PVD	328
LAX	∞	BOS	371
		ORD	621

- Remove next vertex with minimum path from PQ – ORD, and add it to cloud

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

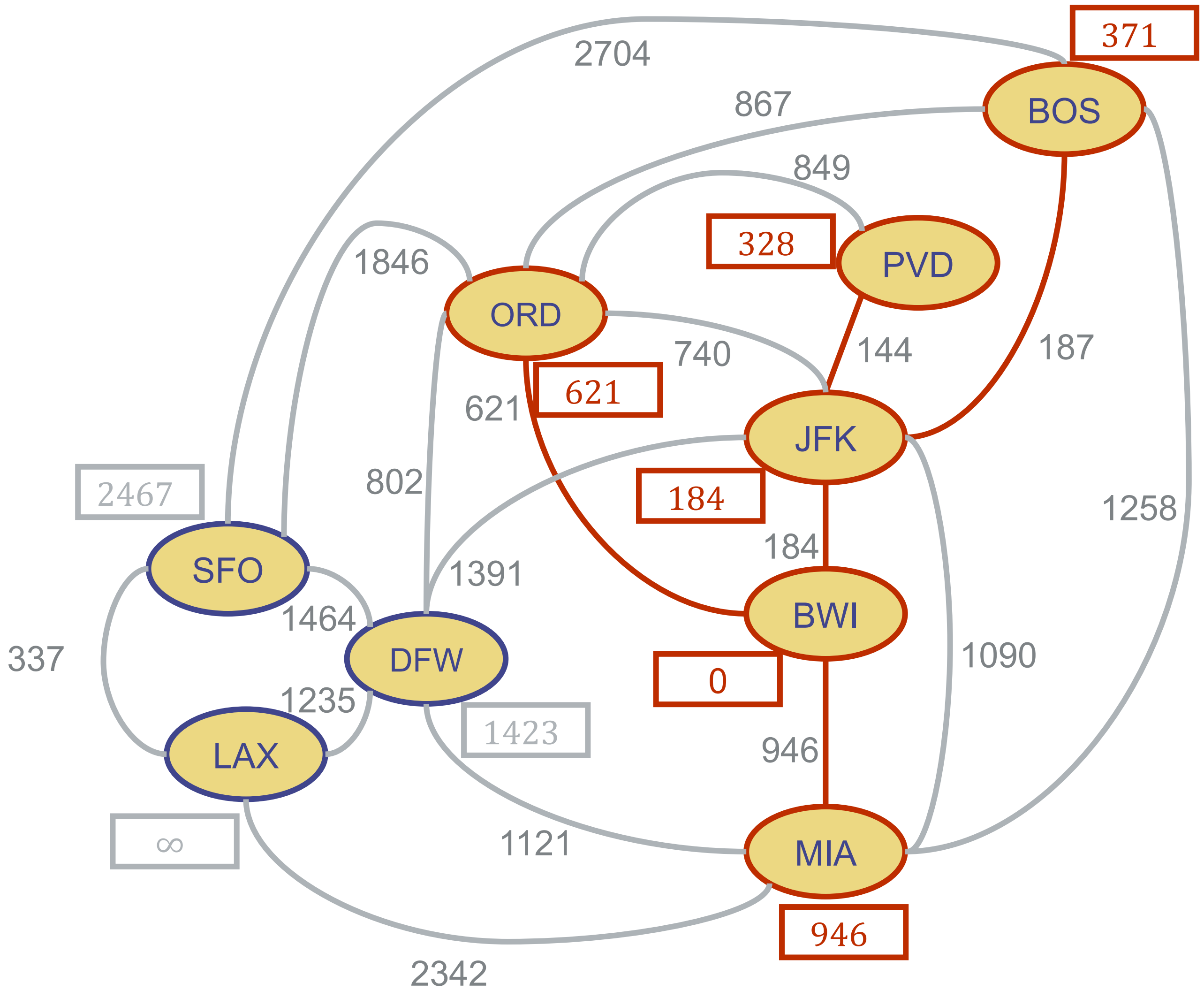
Dijkstra’s Algorithm - Example



PQ		C (“cloud”)	
MIA	946	BWI	0
DFW	1423	JFK	184
SFO	2467	PVD	328
LAX	∞	BOS	371
		ORD	621

- Update the lengths of the paths from BWI to all the vertices adjacent to ORD which are not in the cloud
- Edge relaxation for SFO and DFW
  - $621 + 1846 < 3075$, update SFO path
  - $621 + 802 < 1575$, update DFW path

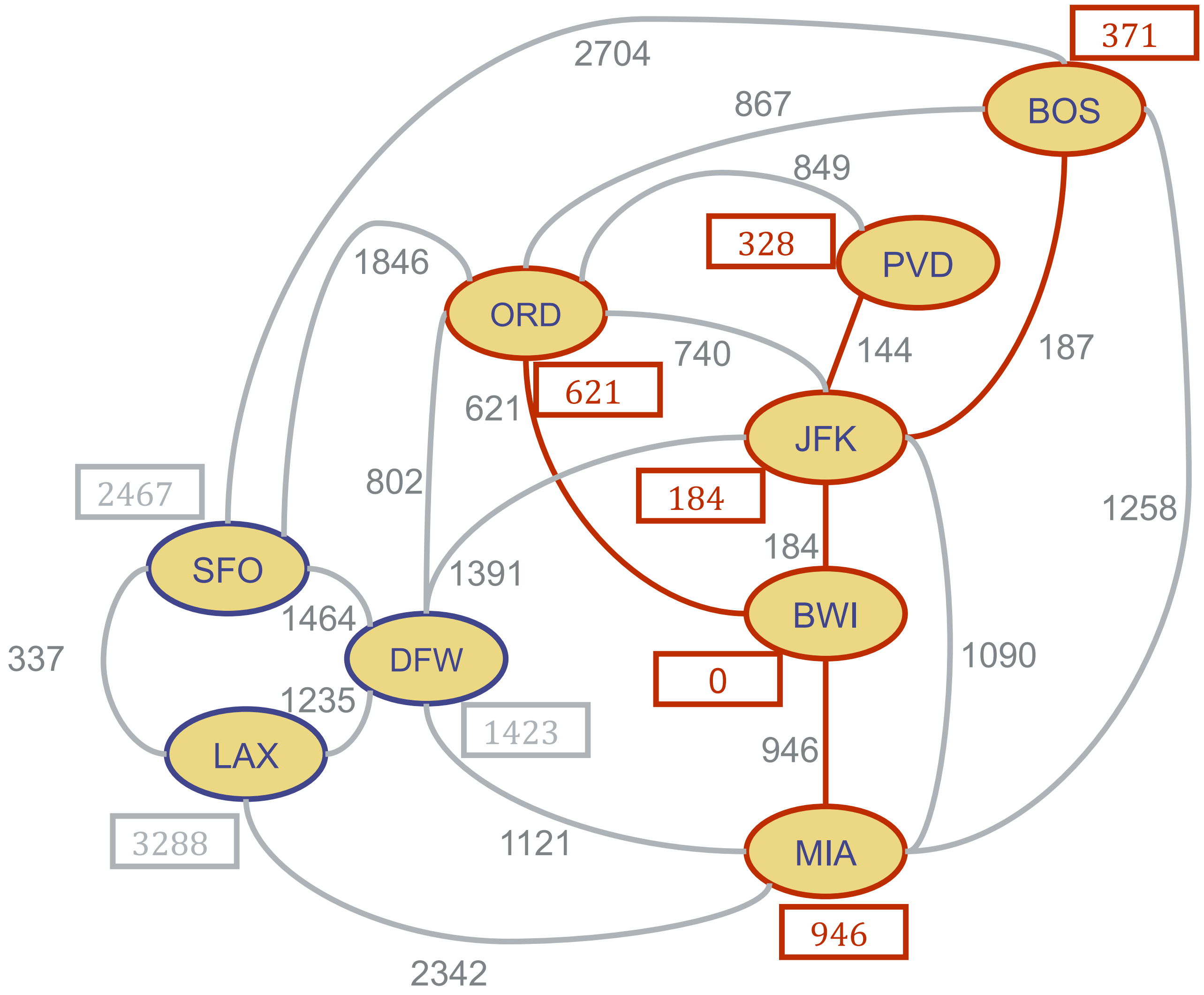
Dijkstra's Algorithm - Example



PQ		C ("cloud")	
DFW	1423	BWI	0
SFO	2467	JFK	184
LAX	∞	PVD	328
		BOS	371
		ORD	621
		MIA	946

- Remove next vertex with minimum path from PQ – MIA, and add it to cloud

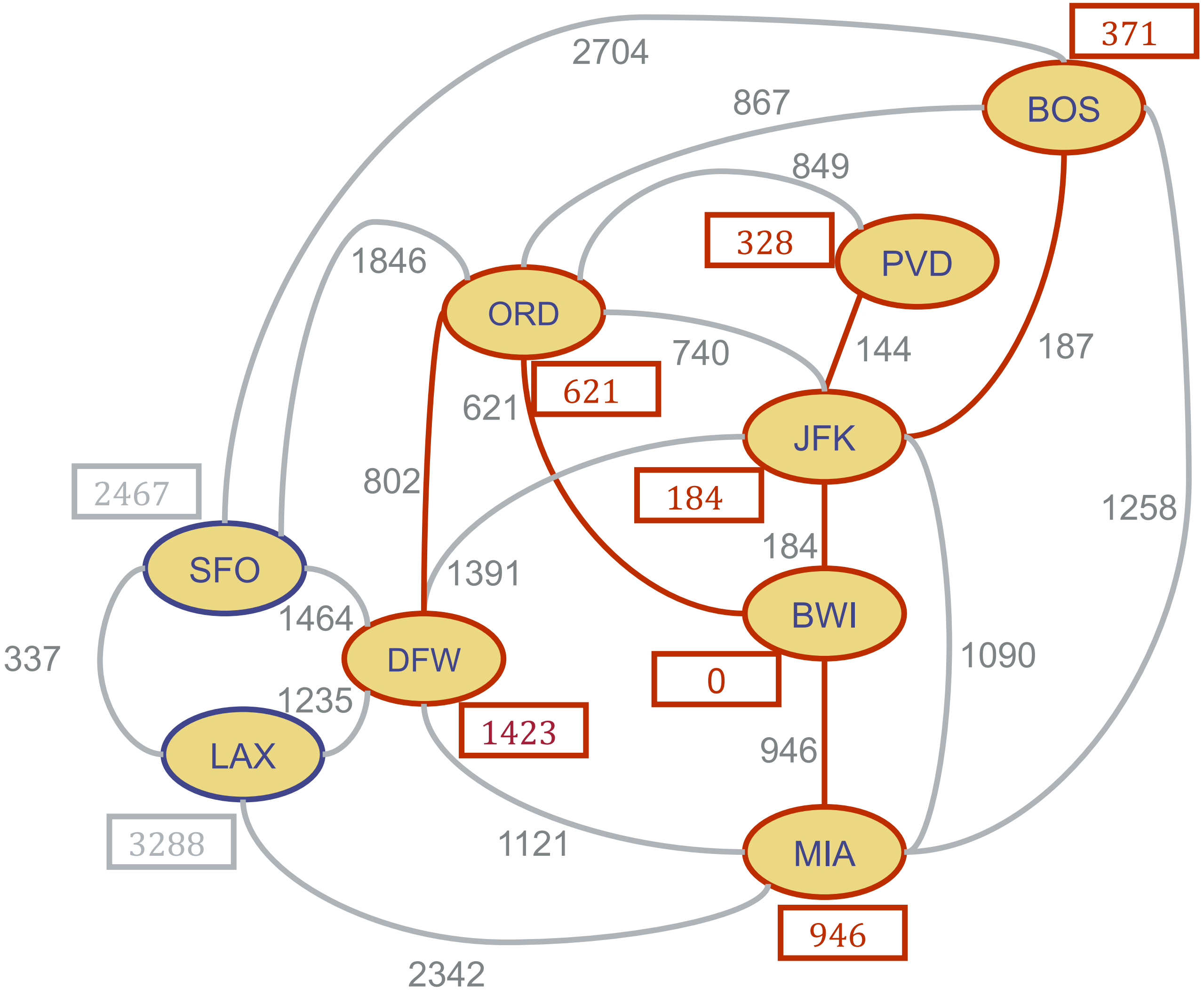
Dijkstra’s Algorithm - Example



PQ		C (“cloud”)	
DFW	1423	BWI	0
SFO	2467	JFK	184
LAX	3288	PVD	328
		BOS	371
		ORD	621
		MIA	946

- Update the lengths of the paths from BWI to all the vertices adjacent to MIA which are not in the cloud
- Edge relaxation for LAX and DFW
  - $946 + 2342 < \infty$, update LAX path
  - $946 + 1121 > 1423$, keep existing DFW path

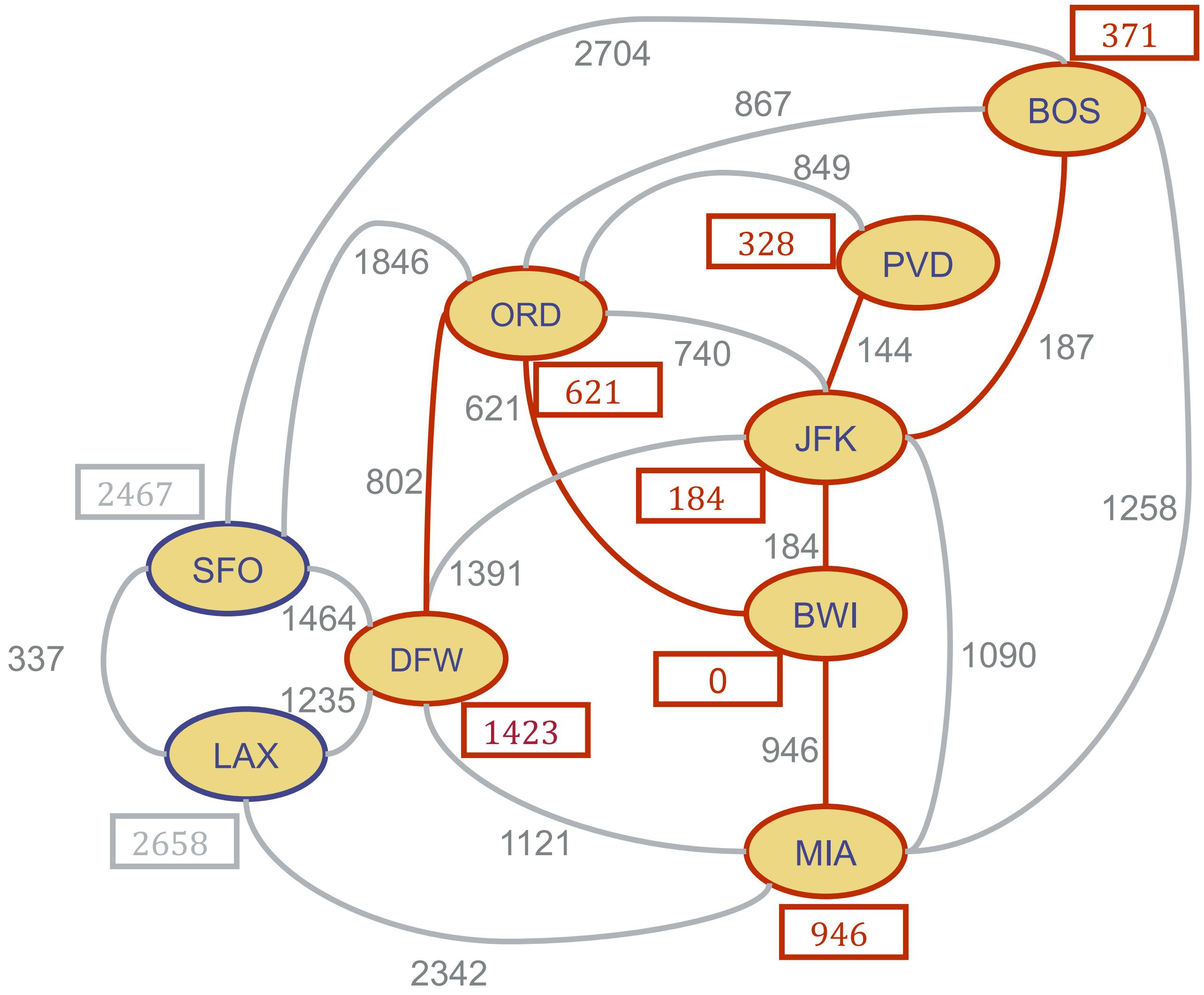
Dijkstra’s Algorithm - Example



PQ		C (“cloud”)	
SFO	2467	BWI	0
LAX	3288	JFK	184
		PVD	328
		BOS	371
		ORD	621
		MIA	946
		DFW	1423

- Remove next vertex with minimum path from PQ – DFW, and add it to cloud

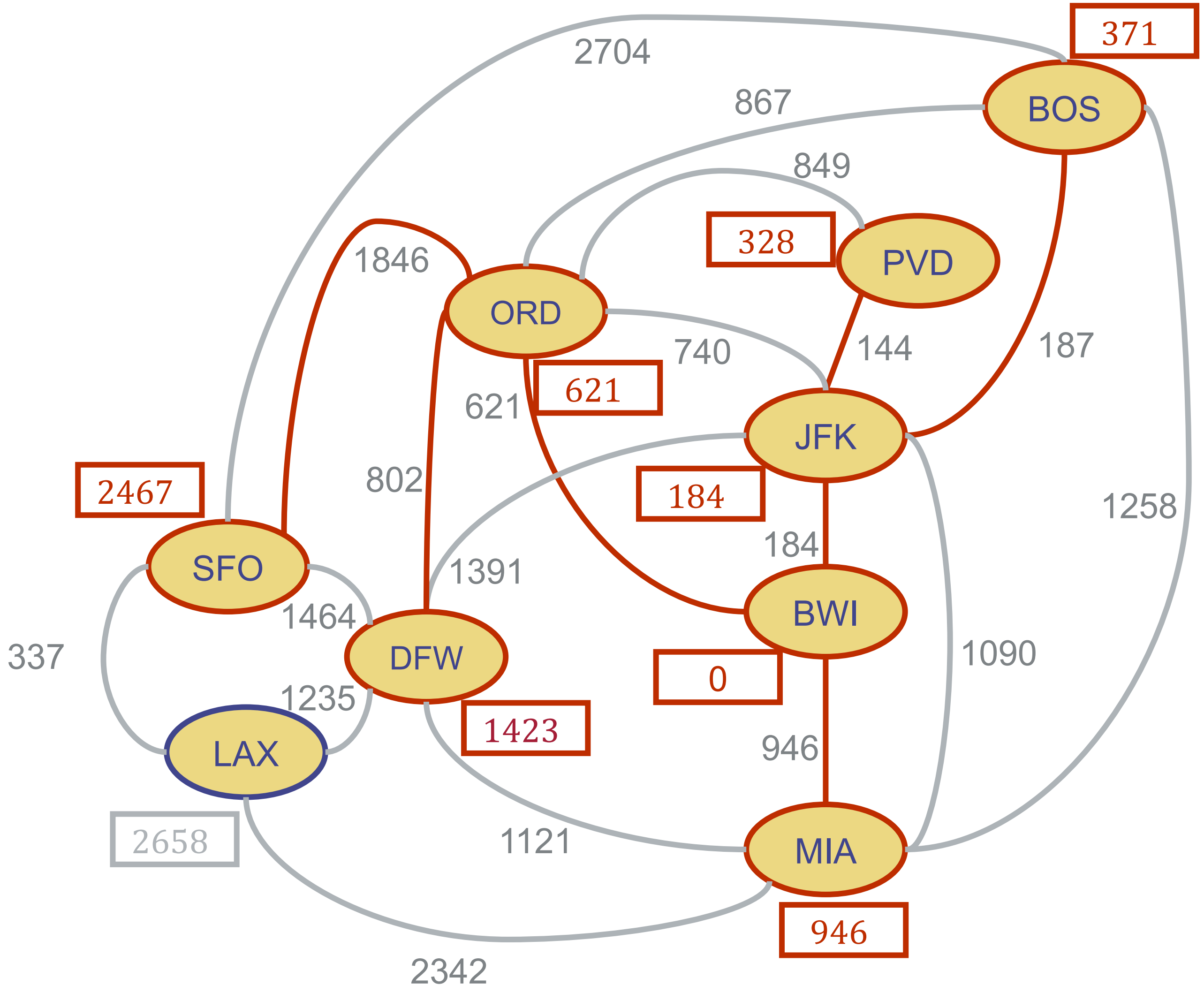
Dijkstra’s Algorithm - Example



PQ		C (“cloud”)	
SFO	2467	BWI	0
LAX	2658	JFK	184
		PVD	328
		BOS	371
		ORD	621
		MIA	946
		DFW	1423

- Update the lengths of the paths from BWI to all the vertices adjacent to DFW which are not in the cloud
- Edge relaxation for LAX and SFO
    - $1423 + 1235 < 3288$, update LAX path
    - $1423 + 1464 > 2467$, keep existing SFO path

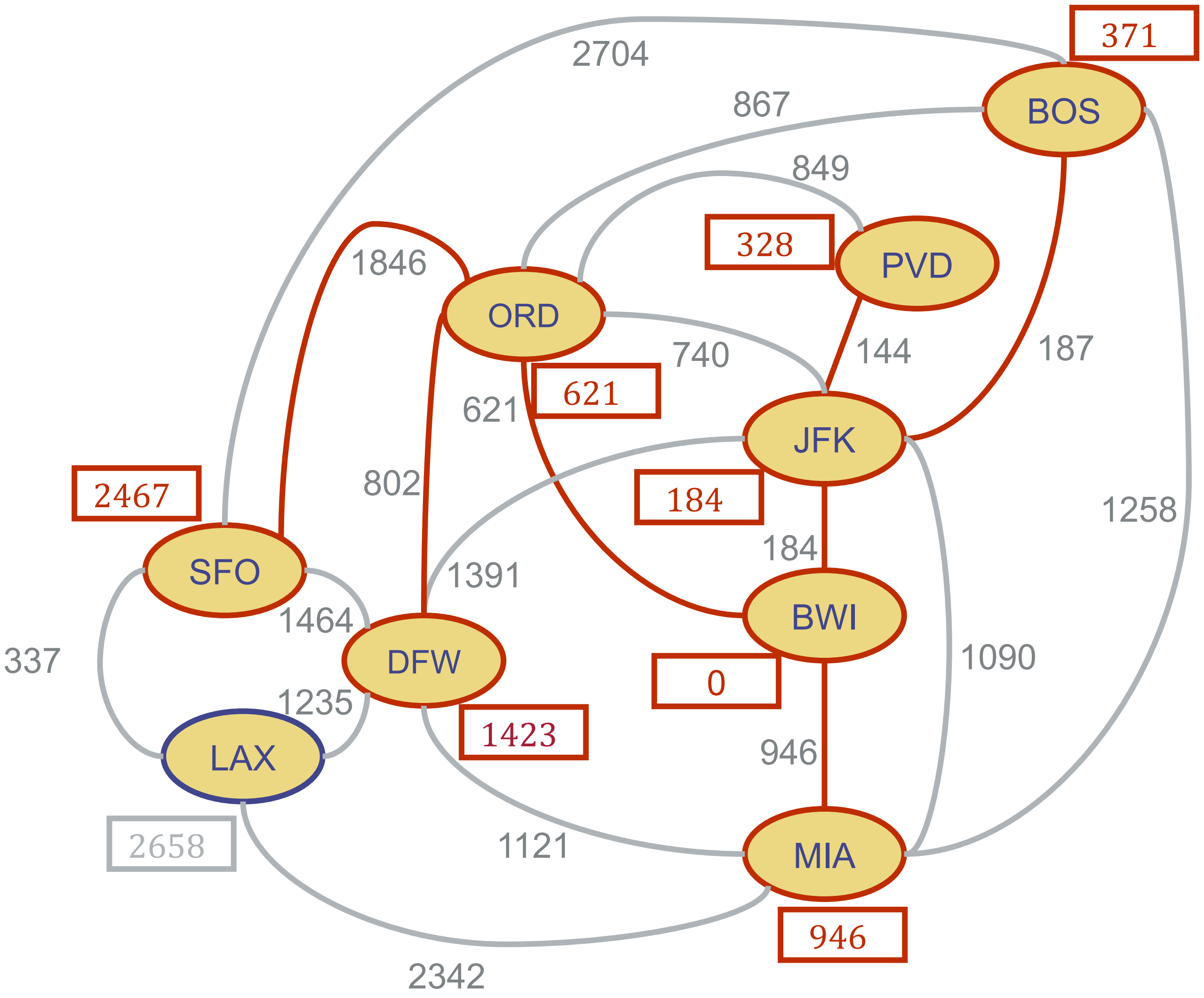
Dijkstra’s Algorithm - Example



PQ		C (“cloud”)	
LAX	2658	BWI	0
		JFK	184
		PVD	328
		BOS	371
		ORD	621
		MIA	946
		DFW	1423
		SFO	2467

- Remove next vertex with minimum path from PQ – SFO, and add it to cloud

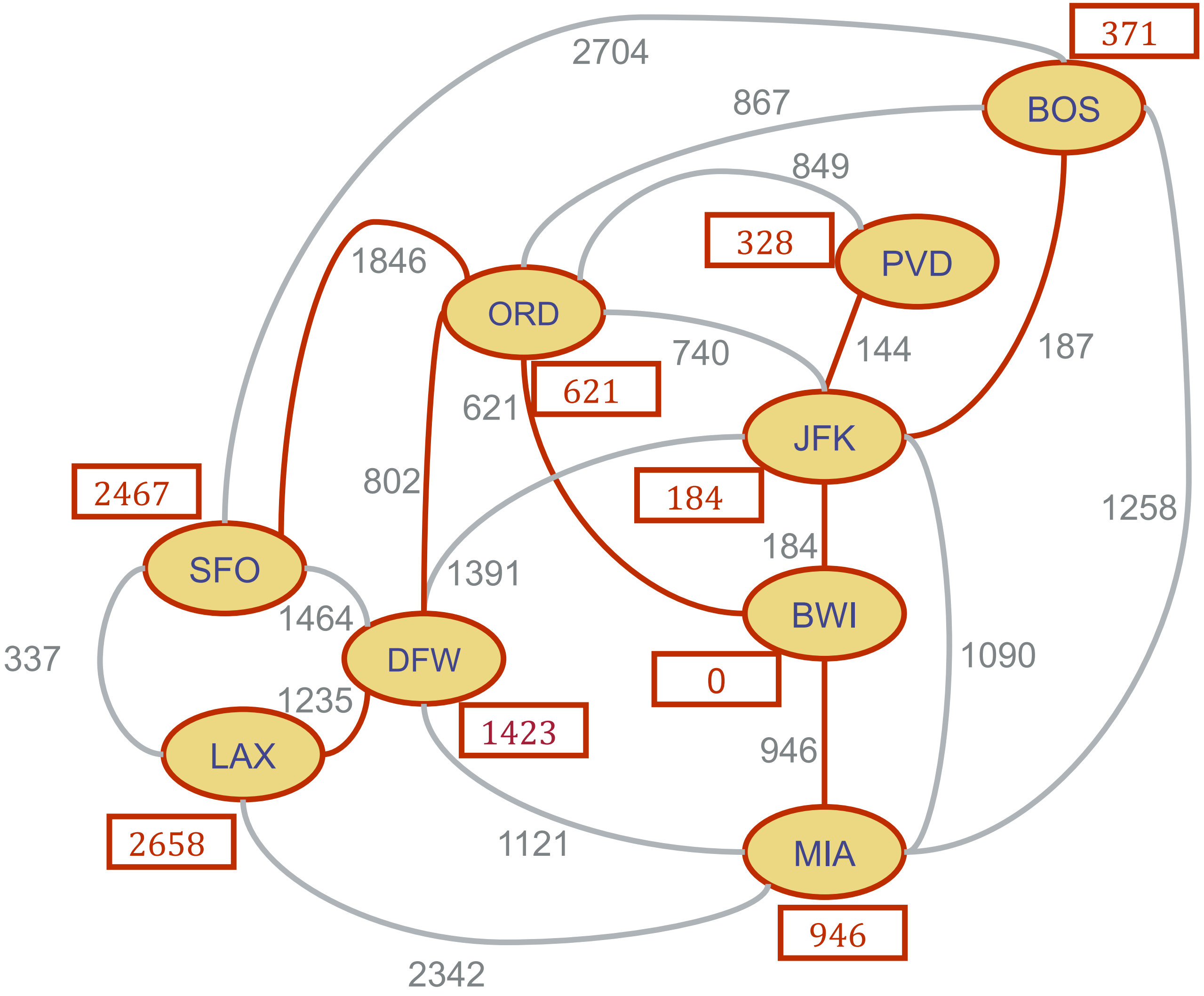
Dijkstra’s Algorithm - Example



PQ		C (“cloud”)	
LAX	2658	BWI	0
		JFK	184
		PVD	328
		BOS	371
		ORD	621
		MIA	946
		DFW	1423
		SFO	2467

- Update the lengths of the paths from BWI to all the vertices adjacent to SFO which are not in the cloud
- Edge relaxation for LAX
  - $2467 + 337 > 2658$, keep existing LAX path

Dijkstra’s Algorithm - Example

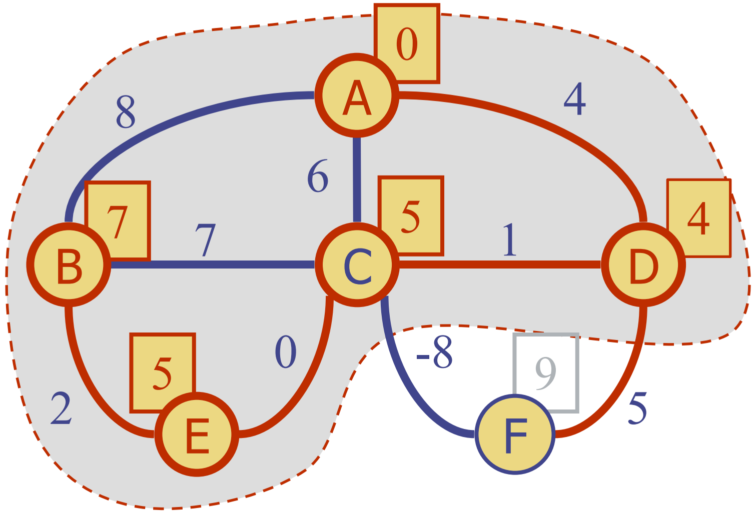


PQ	C (“cloud”)	
	BWI	0
	JFK	184
	PVD	328
	BOS	371
	ORD	621
	MIA	946
	DFW	1423
	SFO	2467
	LAX	2658

- Remove next vertex with minimum path from PQ – LAX, and add it to cloud
- No edges to relax – all vertices are in the “cloud”
- PQ empty. STOP.

Why Does Dijkstra's Algorithm Require Nonnegative Weights?

- Dijkstra's algorithm is based on the greedy method, and adds vertices by increasing distance
- If a node with a negative edge were to be added late to the "cloud", it would invalidate the distances (shortest paths) for vertices already in the cloud
  - E.g. the true distance of C is 1, but it is already added to the "cloud" with $d(C) = 5$

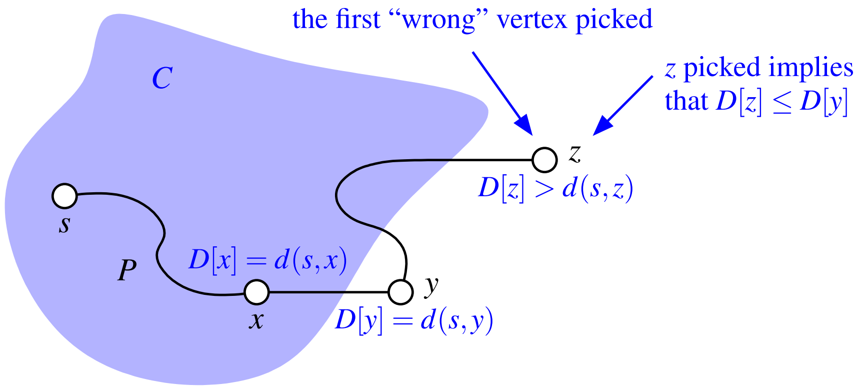


Why Does Dijkstra’s Algorithm Work?

- At the moment when a vertex u is added to the “cloud”, its label $D[u]$ stores the correct length of a shortest path from s to u
    - This is true only if there are no negative-weight edges in G, otherwise the greedy method does not work correctly
- Thus, when the algorithm terminates it will have computed the shortest-path distance from s for every vertex of G

Why Does Dijkstra's Algorithm Work?

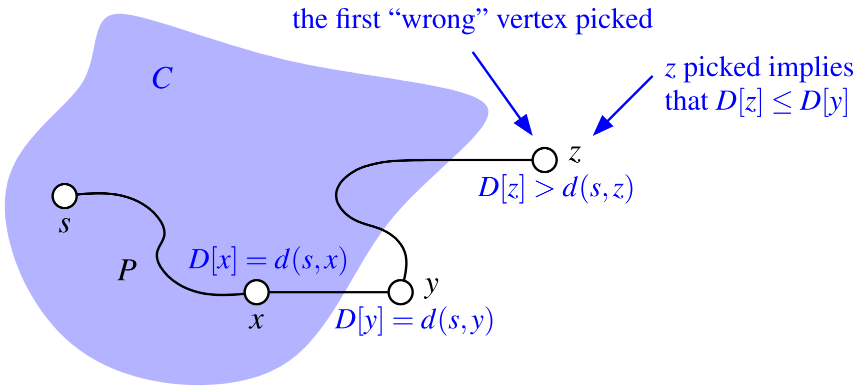
- Proposition. In Dijkstra's algorithm, whenever a vertex v is added to the cloud, the label $D[v]$ is equal to $d(s,v)$, the length of the shortest path from s to v.
- Justification. Suppose that $D[v] > d(s,v)$ for some vertex v.
  - Let z be the first vertex added to the cloud C, such that $D[z] > d(s,z)$
  - There must be a shorter path P from s to z (otherwise, $d(s,z) = \infty = D[z]$)
  - Consider the moment when z is added to C:
    - Let y be the first vertex of P (going from s to z) that is not in C
    - Let x be the predecessor of y in the path P
    - x is already in C, and $D[x] = d(s,x)$, since we assumed that z was the first vertex where $D[z] > d(s,z)$



the first "wrong" vertex picked
z picked implies that $D[z] \le D[y]$
C
z
$D[z] > d(s,z)$
s
P
$D[x] = d(s,x)$
y
x
$D[y] = d(s,y)$

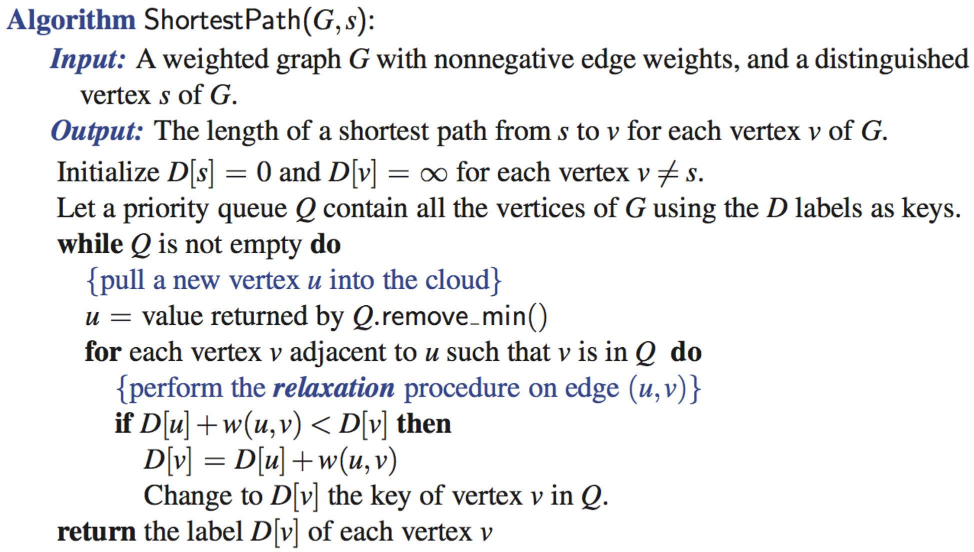
Why Does Dijkstra's Algorithm Work?

- When x was added to the C, we relaxed the edge leading to y: we tested (and possibly updated) $D[y]$ such that $D[y] \leq D[x] + w(x, y) = d(s, x) + w(x, y)$
- Since y is the next vertex on the shortest path from s to z, $D[y] = d(s, y)$
- Now we are adding z to the path, so $D[z] \leq D[y]$ (because we are always adding to the vertex with min path that is not yet in the "cloud" to C)
- A subpath of a shortest path is itself a shortest path, so y must be on the shortest path from s to z, therefore $d(s, y) + d(y, z) = d(s, z)$
- Also, $d(y, z) \geq 0$, since we cannot have negative weights
- Therefore $D[z] \leq D[y] = d(s, y) \leq d(s, y) + d(y, z) = d(s, z)$
- Contradiction! We had supposed that z was the first vertex added to C such that $D[z] > d(s, z)$


the first "wrong" vertex picked
z picked implies that $D[z] \leq D[y]$
C
z
$D[z] > d(s,z)$
s
P
$D[x] = d(s,x)$
y
x
$D[y] = d(s,y)$

Running Time of Dijkstra’s Algorithm

- Assume that the graph G has n vertices and m edges, and is implemented using an adjacency list/adjacency map
- This allows us to step through the incident edges of a vertex in time proportional to their number
- The for loop takes therefore $O(m)$ time
- Priority queue operations
  - Assume PQ implemented using a heap
  - Each vertex is inserted once and removed once from the priority queue, each insertion and removal takes $O(\log n)$ time
  - The key of a vertex in the PQ is modified at most $\deg(v)$ times, where each key change takes $O(\log n)$ time
- In this case, Dijkstra’s algorithm runs in $O\big((n+m)\log n\big)$ time – or $O(n^2 \log n)$



```
Algorithm ShortestPath($G, s$):
  Input: A weighted graph $G$ with nonnegative edge weights, and a distinguished
    vertex $s$ of $G$.
  Output: The length of a shortest path from $s$ to $v$ for each vertex $v$ of $G$.
  Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
  Let a priority queue $Q$ contain all the vertices of $G$ using the $D$ labels as keys.
  while $Q$ is not empty do
    {pull a new vertex $u$ into the cloud}
    $u$ = value returned by $Q$.remove_min()
    for each vertex $v$ adjacent to $u$ such that $v$ is in $Q$ do
      {perform the relaxation procedure on edge $(u, v)$}
      if $D[u] + w(u, v) < D[v]$ then
        $D[v] = D[u] + w(u, v)$
        Change to $D[v]$ the key of vertex $v$ in $Q$.
  return the label $D[v]$ of each vertex $v$
```

Running Time of Dijkstra's Algorithm – variant 2

- Assume that the graph G has n vertices and m edges, and is implemented using an adjacency list/adjacency map
- Assume a priority queue implemented using an unsorted list
    - This means that it takes $O(n)$ time to extract the minimum element
    - But it allows for fast key updates in $O(1)$ time, provided the PQ supports location-aware entities
- In this case, Dijkstra's algorithm runs in $O(n^2)$ time

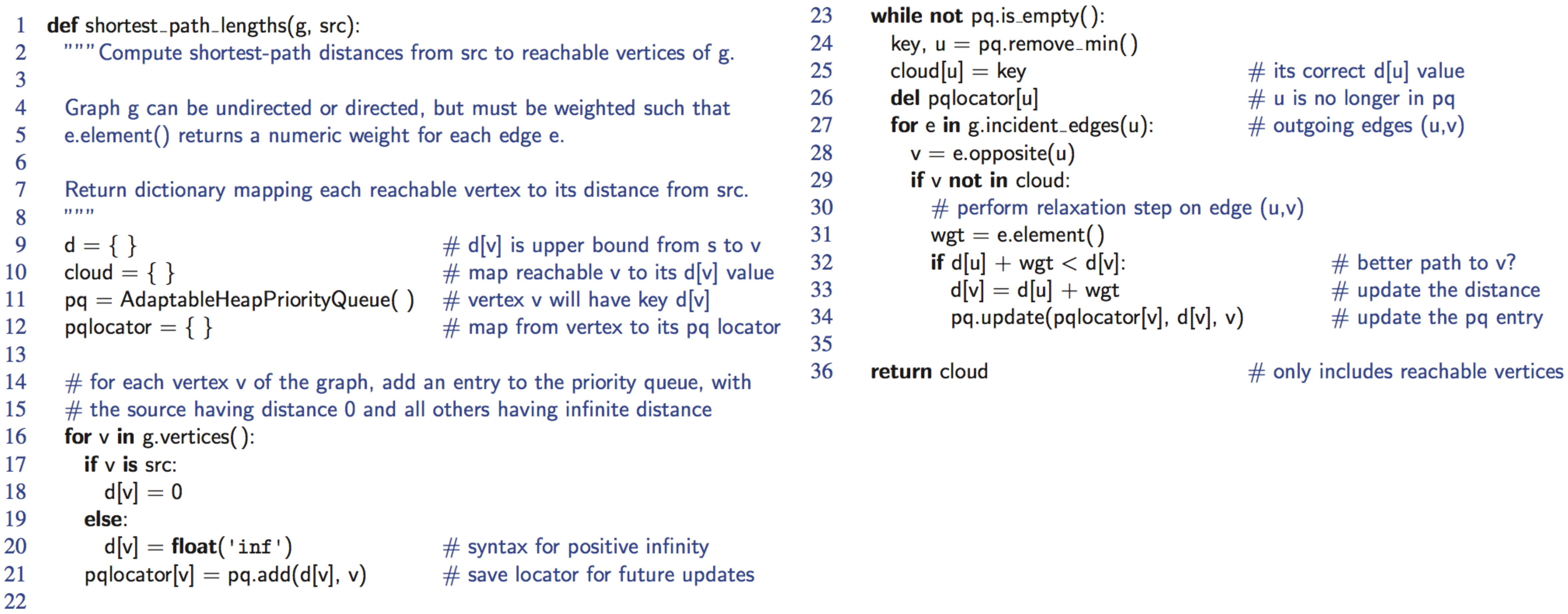
- The heap implementation is preferable when the number of edges in the graph is small – when

$$m < n^2 / \log n$$

- The list implementation is preferable when the number of edges is large – when

$$m > n^2 / \log n$$

Dijkstra’s Algorithm – Python Implementation

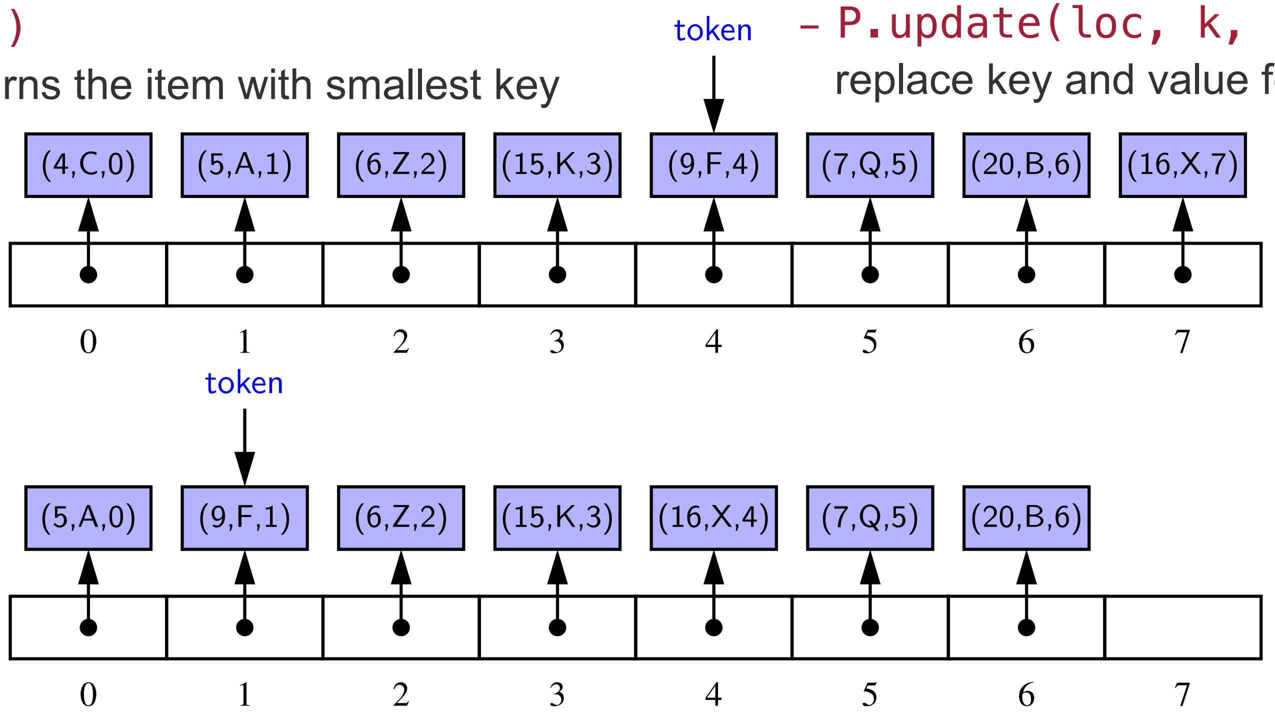


```
def shortest_path_lengths(g, src):
  """Compute shortest-path distances from src to reachable vertices of g.

  Graph g can be undirected or directed, but must be weighted such that
  e.element() returns a numeric weight for each edge e.

  Return dictionary mapping each reachable vertex to its distance from src.
  """
  d = { }                          # d[v] is upper bound from s to v
  cloud = { }                      # map reachable v to its d[v] value
  pq = AdaptableHeapPriorityQueue( ) # vertex v will have key d[v]
  pqlocator = { }                  # map from vertex to its pq locator

  # for each vertex v of the graph, add an entry to the priority queue, with
  # the source having distance 0 and all others having infinite distance
  for v in g.vertices( ):
    if v is src:
      d[v] = 0
    else:
      d[v] = float('inf')          # syntax for positive infinity
    pqlocator[v] = pq.add(d[v], v) # save locator for future updates

  while not pq.is_empty( ):
    key, u = pq.remove_min( )
    cloud[u] = key                 # its correct d[u] value
    del pqlocator[u]               # u is no longer in pq
    for e in g.incident_edges(u):  # outgoing edges (u,v)
      v = e.opposite(u)
      if v not in cloud:
        # perform relaxation step on edge (u,v)
        wgt = e.element( )
        if d[u] + wgt < d[v]:              # better path to v?
          d[v] = d[u] + wgt                # update the distance
          pq.update(pqlocator[v], d[v], v) # update the pq entry

  return cloud                     # only includes reachable vertices
```

Adaptable Priority Queue

- Remember the methods supported by the priority queue ADT, for a priority queue P:
  - `P.add(k, x)`
    inserts an item with key k and value x
  - `P.min()`
    returns, but does not remove the item with the smallest key
  - `P.remove_min()`
    removes and returns the item with smallest key

- The adaptable priority queue, needs to support efficient an efficient `update()` operation
- Need mechanisms to find the node with a particular element without searching through the entire collection
- Use a locator as a parameter when invoking the update method
  - `P.update(loc, k, v)`
    replace key and value for the item identified by locator

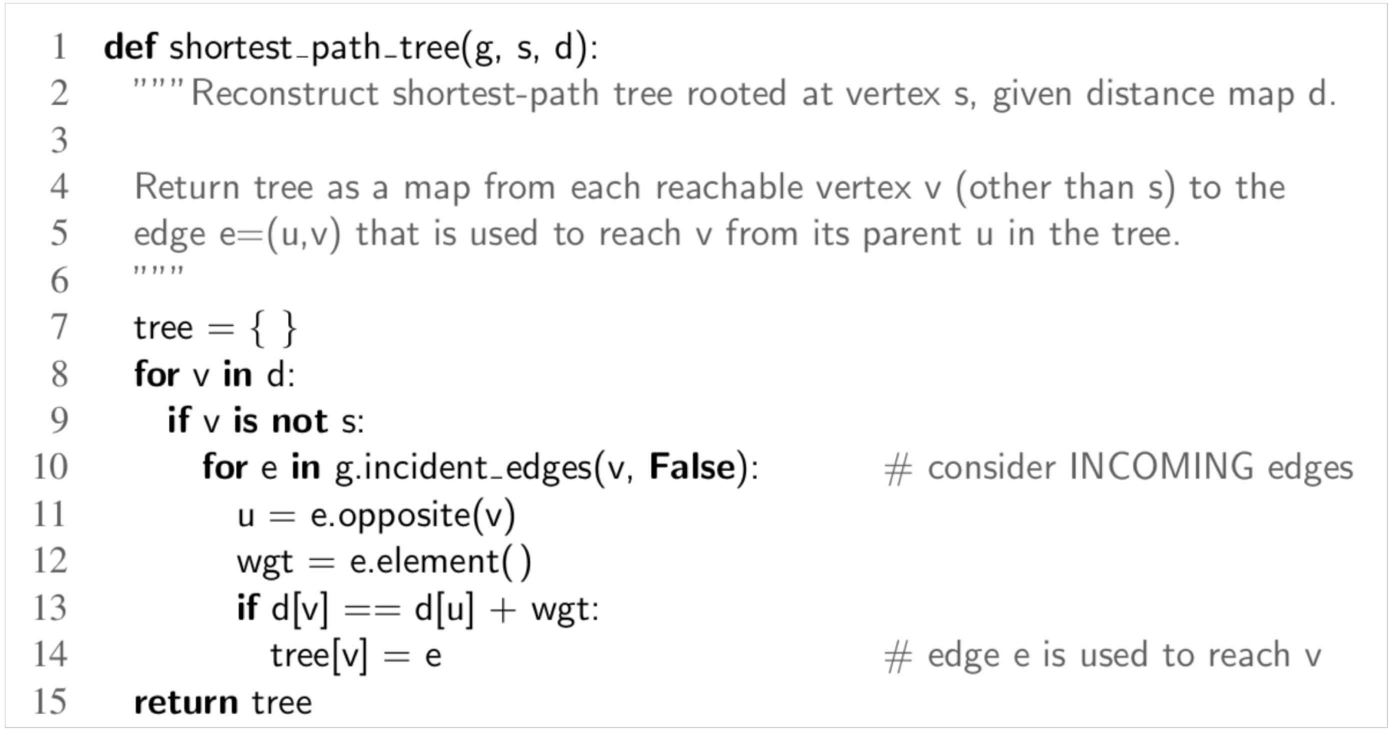


PQ after `remove_min()`

Reconstructing the Shortest-Path Tree

- Dijkstra's algorithm computes the length of the shortest path from the source vertex s to any other reachable vertex v
- It does not provide the actual shortest path – in terms of vertices and edges
- The collection of all the shortest paths from a source vertex s can be compactly represented by a shortest-path tree
- It can be constructed using the $d[v]$ values obtained during Dijkstra's algorithm
- Idea: like with the DFS and BFS trees, map each vertex $v \neq s$ to a parent u such that u is the vertex immediately before v on a shortest path from s to v
- If u is the vertex just before v on the shortest path from s to v, then $d[u] + w(u, v) = d[v]$

Reconstructing the Shortest-Path Tree (cont’d)



```
def shortest_path_tree(g, s, d):
  """Reconstruct shortest-path tree rooted at vertex s, given distance map d.

  Return tree as a map from each reachable vertex v (other than s) to the
  edge e=(u,v) that is used to reach v from its parent u in the tree.
  """
  tree = { }
  for v in d:
    if v is not s:
      for e in g.incident_edges(v, False):          # consider INCOMING edges
        u = e.opposite(v)
        wgt = e.element( )
        if d[v] == d[u] + wgt:
          tree[v] = e                               # edge e is used to reach v
  return tree
```

- Reconstruct the tree by testing all the incoming edges for each vertex v
- If $d[v] = d[u] + weight$, then save the edge e as being the edge used to reach v from its parent u
- Running time is $O(n + m)$

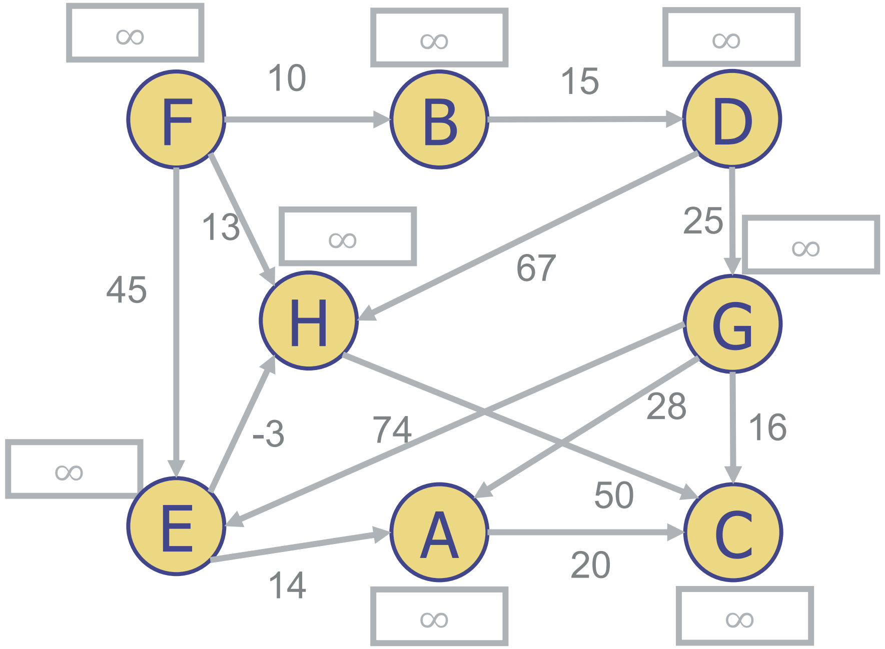
Single-Source Shortest Path Problem in a DAG

Single-Source Shortest Path Problem in a DAG

- The single-source shortest path problem can be solved more efficiently if we know that we are dealing with a directed acyclic graph
- Algorithm based on topological order, simpler and faster than Dijkstra’s algorithm
- It will work even if negative-weight edges are part of the DAG
- Idea:
    - Initialize $D[s] = 0$, $D[v] = \infty$ for every vertex in G
    - Relax the edges one by one, taking the vertices in topological order
- Running time: $O(n + m)$

Single-Source Shortest Path Problem in a DAG - Example

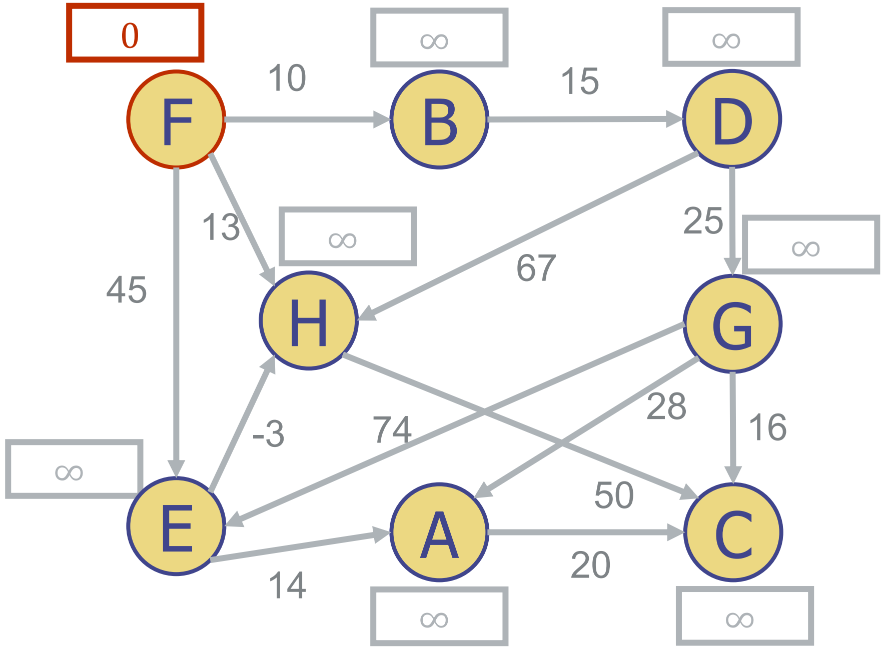
topo
F
B
D
G
E
H
A
C



$D[v]$		C (“cloud”)
A	∞	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	
G	∞	
H	∞	

Single-Source Shortest Path Problem in a DAG - Example

topo
B
D
G
E
H
A
C

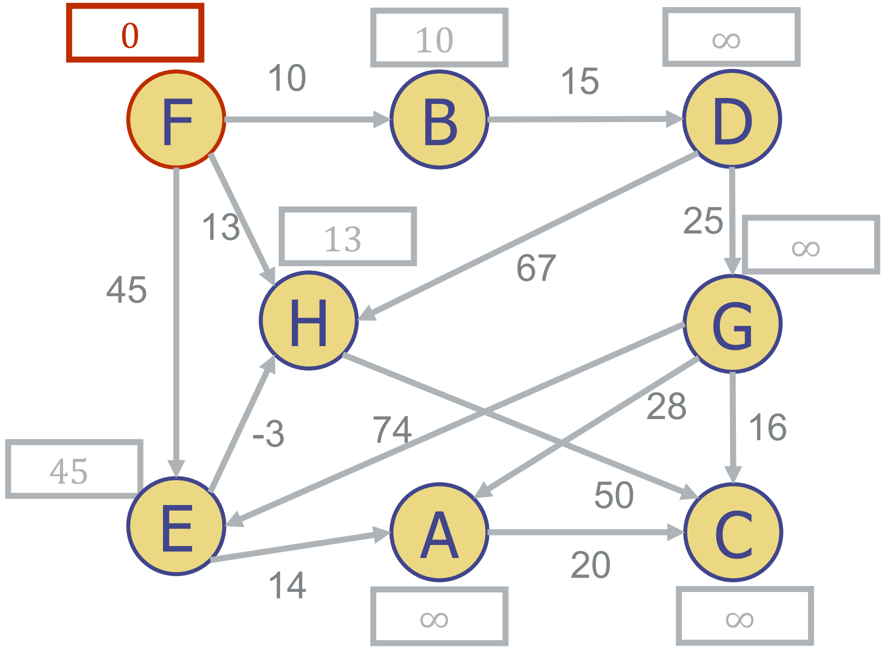


$D[v]$		C (“cloud”)	
A	∞	F	0
B	∞		
C	∞		
D	∞		
E	∞		
F	0		
G	∞		
H	∞		

- Start vertex is F, the first vertex in the topological sort
- Remove F from the topological sort, and add it to cloud
- Vertices in the “cloud” are marked red

Single-Source Shortest Path Problem in a DAG - Example

topo
B
D
G
E
H
A
C

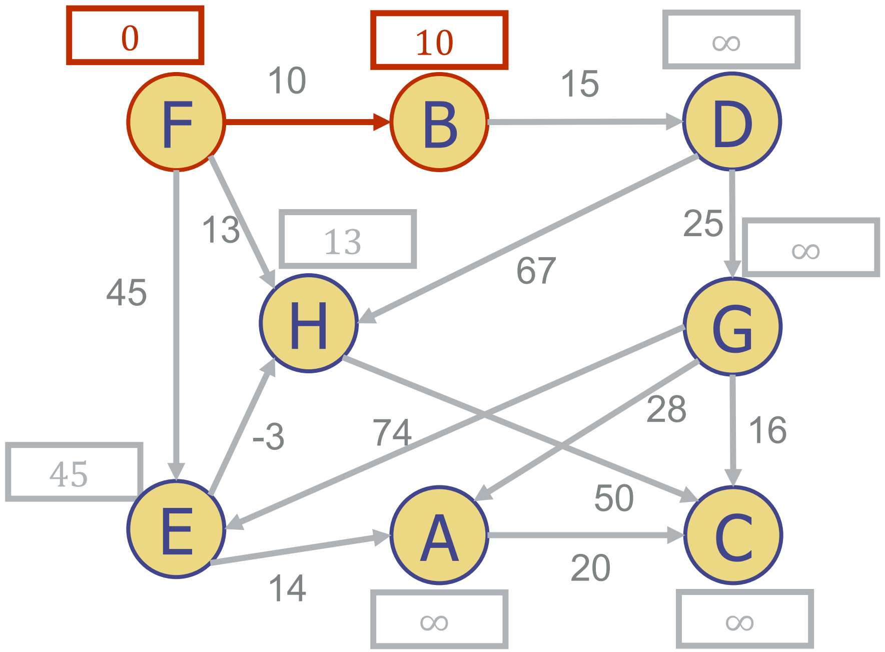


$D[v]$		C (“cloud”)	
A	∞	F	0
B	10		
C	∞		
D	∞		
E	45		
F	0		
G	∞		
H	13		

- Update the lengths of the paths from F to all the vertices adjacent to F
- Edge relaxation for B, H and E
  - $0 + 10 < \infty$, update B path
  - $0 + 13 < \infty$, update H path
  - $0 + 45 < \infty$, update E path

Single-Source Shortest Path Problem in a DAG - Example

topo
D
G
E
H
A
C

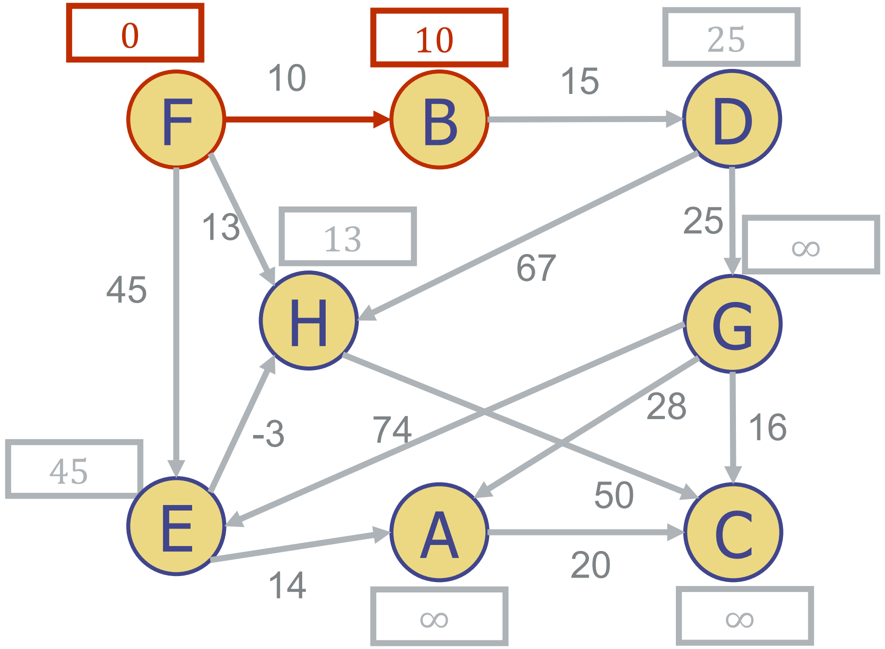


$D[v]$		C (“cloud”)	
A	∞	F	0
B	10	B	10
C	∞		
D	∞		
E	45		
F	0		
G	∞		
H	13		

- Remove next vertex from topological sort - B, and add it to cloud

Single-Source Shortest Path Problem in a DAG - Example

topo
D
G
E
H
A
C

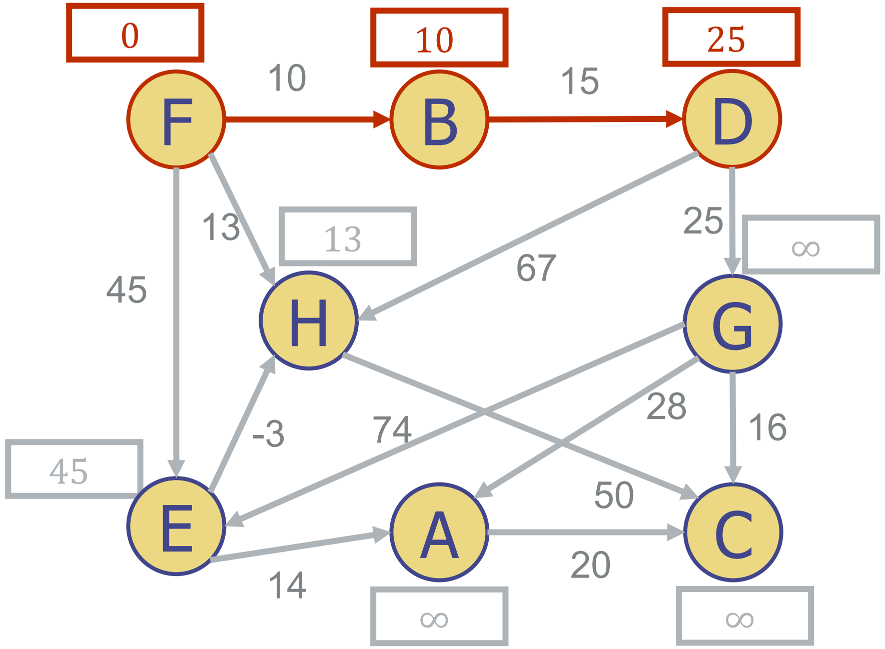


$D[v]$		C ("cloud")	
A	∞	F	0
B	10	B	10
C	∞		
D	25		
E	45		
F	0		
G	∞		
H	13		

- Update the lengths of the paths from B to all the vertices adjacent to B
- Edge relaxation for D
  - $10 + 15 < \infty$, update D path

Single-Source Shortest Path Problem in a DAG - Example

topo
G
E
H
A
C

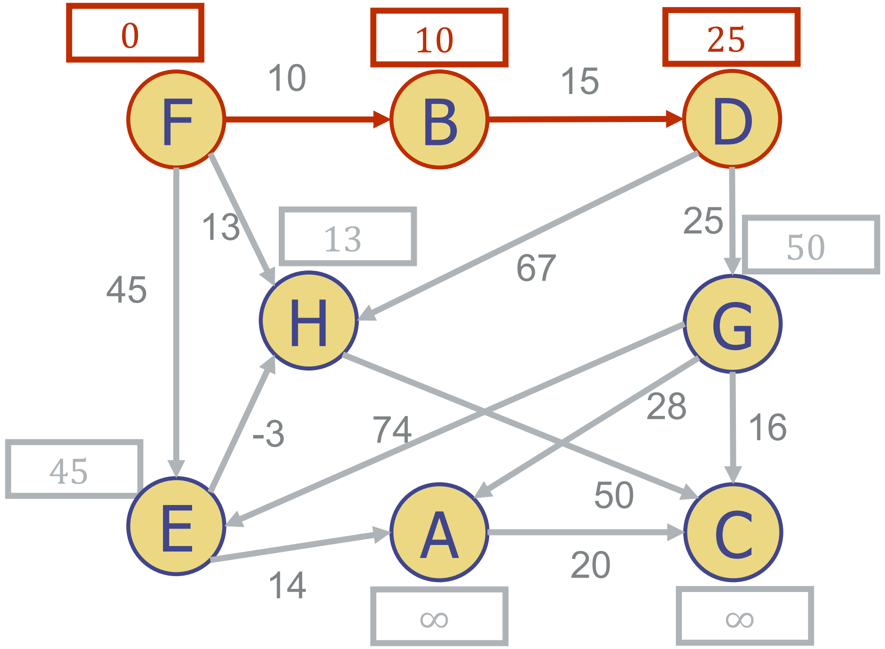


$D[v]$		C ("cloud")	
A	∞	F	0
B	10	B	10
C	∞	D	25
D	25		
E	45		
F	0		
G	∞		
H	13		

- Remove next vertex from topological sort - D, and add it to cloud

Single-Source Shortest Path Problem in a DAG - Example

topo
G
E
H
A
C

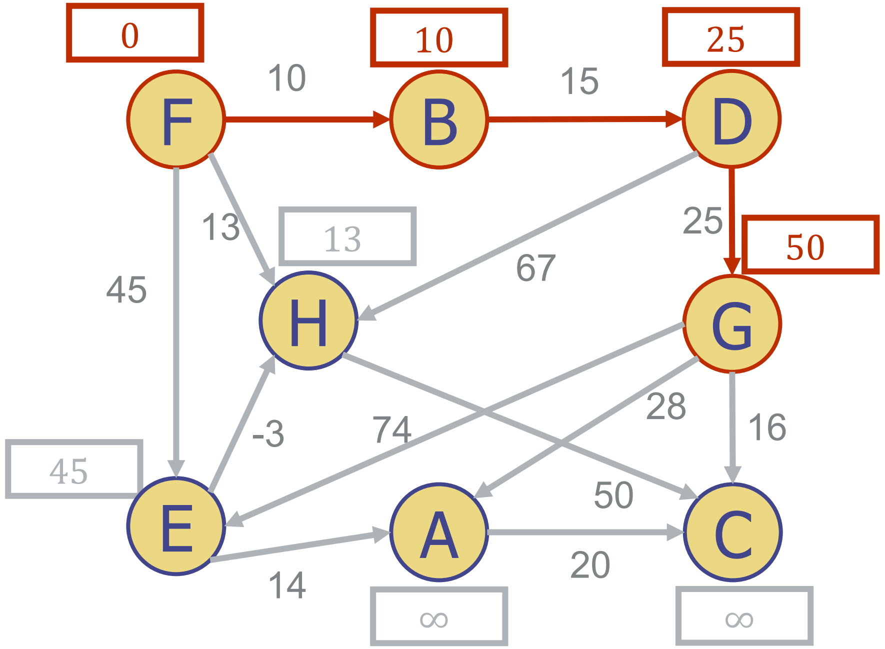


$D[v]$		C (“cloud”)	
A	∞	F	0
B	10	B	10
C	∞	D	25
D	25		
E	45		
F	0		
G	50		
H	13		

- Update the lengths of the paths from D to all the vertices adjacent to D
- Edge relaxation for G and H
  - $25 + 25 < \infty$, update G path
  - $25 + 67 > 13$, keep existing H path

Single-Source Shortest Path Problem in a DAG - Example

topo
E
H
A
C

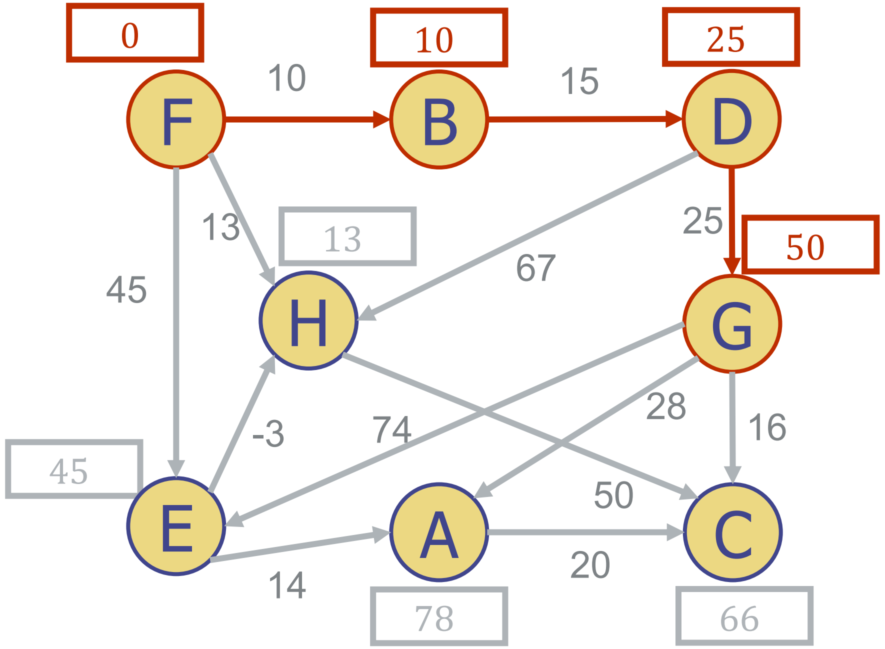


$D[v]$		C ("cloud")	
A	∞	F	0
B	10	B	10
C	∞	D	25
D	25	G	50
E	45		
F	0		
G	50		
H	13		

- Remove next vertex from topological sort - G, and add it to cloud

Single-Source Shortest Path Problem in a DAG - Example

topo
E
H
A
C

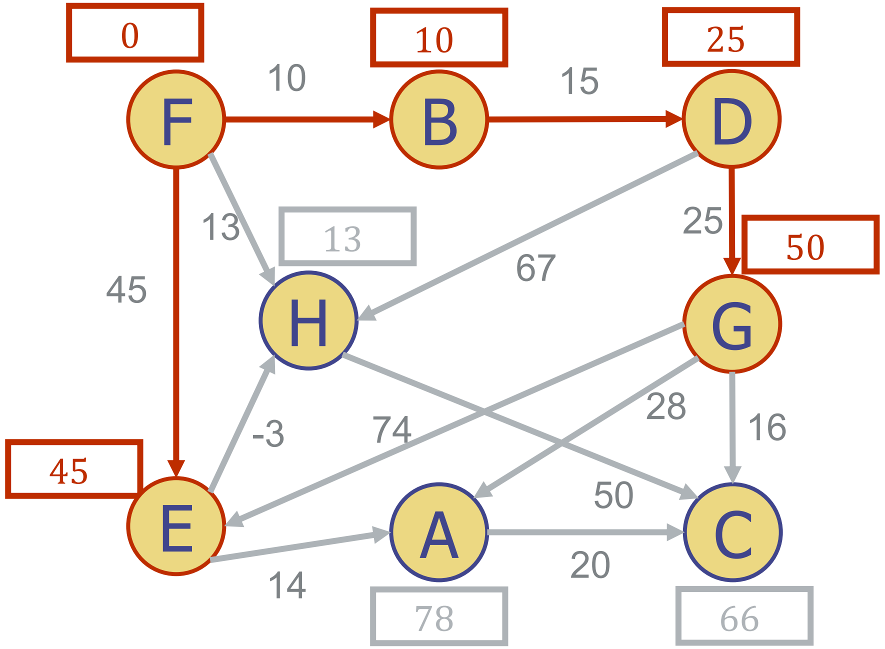


$D[v]$		C ("cloud")	
A	78	F	0
B	10	B	10
C	66	D	25
D	25	G	50
E	45		
F	0		
G	50		
H	13		

- Update the lengths of the paths from G to all the vertices adjacent to G
- Edge relaxation for E, A and C
  - $50 + 74 > 45$, keep existing E path
  - $50 + 28 < \infty$, update A path
  - $50 + 16 < \infty$, update C path

Single-Source Shortest Path Problem in a DAG - Example

topo
H
A
C

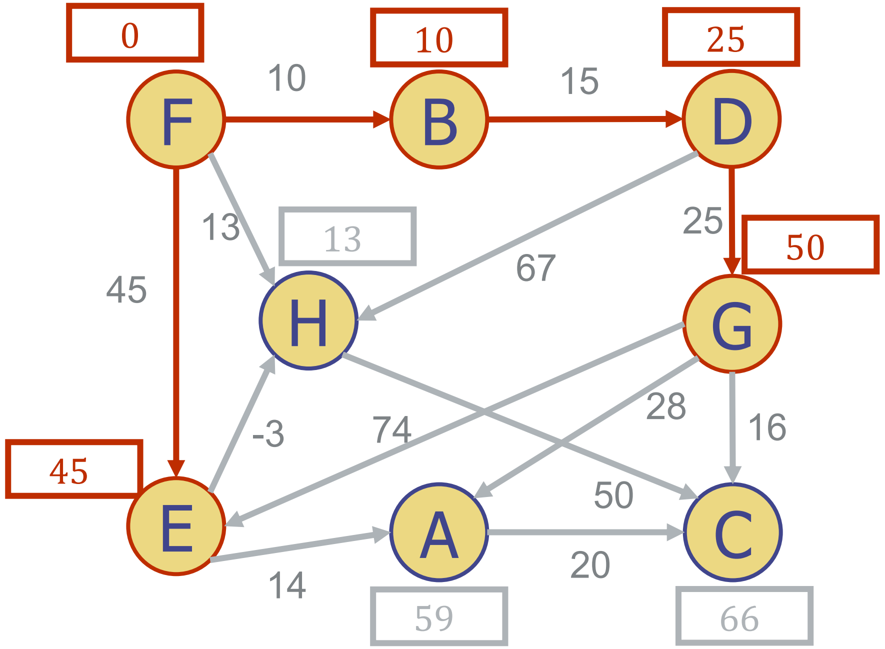


$D[v]$		C ("cloud")	
A	78	F	0
B	10	B	10
C	66	D	25
D	25	G	50
E	45	E	45
F	0		
G	50		
H	13		

- Remove next vertex from topological sort - E, and add it to cloud

Single-Source Shortest Path Problem in a DAG - Example

topo
H
A
C

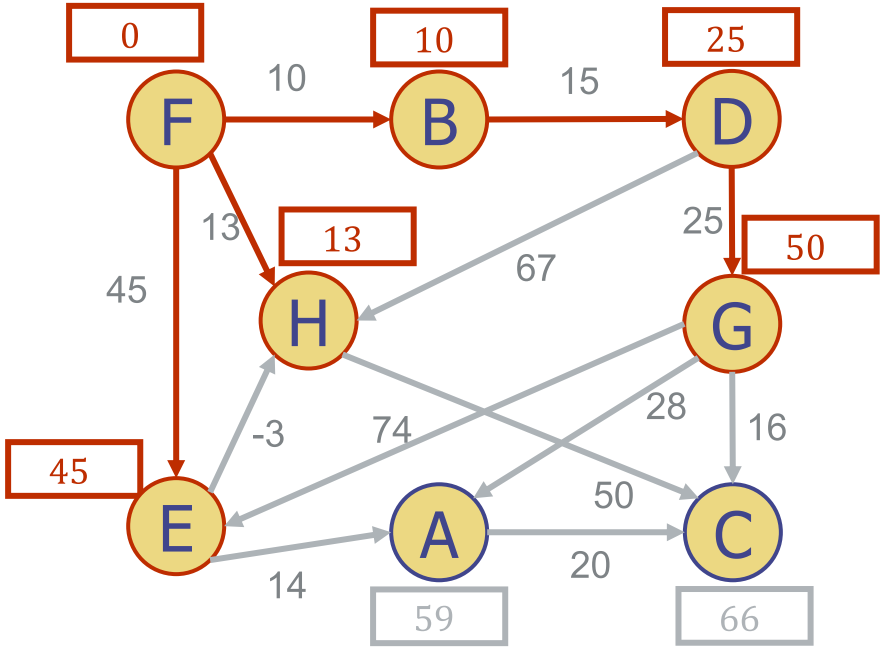


$D[v]$		C ("cloud")	
A	59	F	0
B	10	B	10
C	66	D	25
D	25	G	50
E	45	E	45
F	0		
G	50		
H	13		

- Update the lengths of the paths from E to all the vertices adjacent to E
- Edge relaxation for H and A
  - $45 - 3 > 13$, keep existing H path
  - $45 + 14 < 78$, update A path

Single-Source Shortest Path Problem in a DAG - Example

topo
A
C

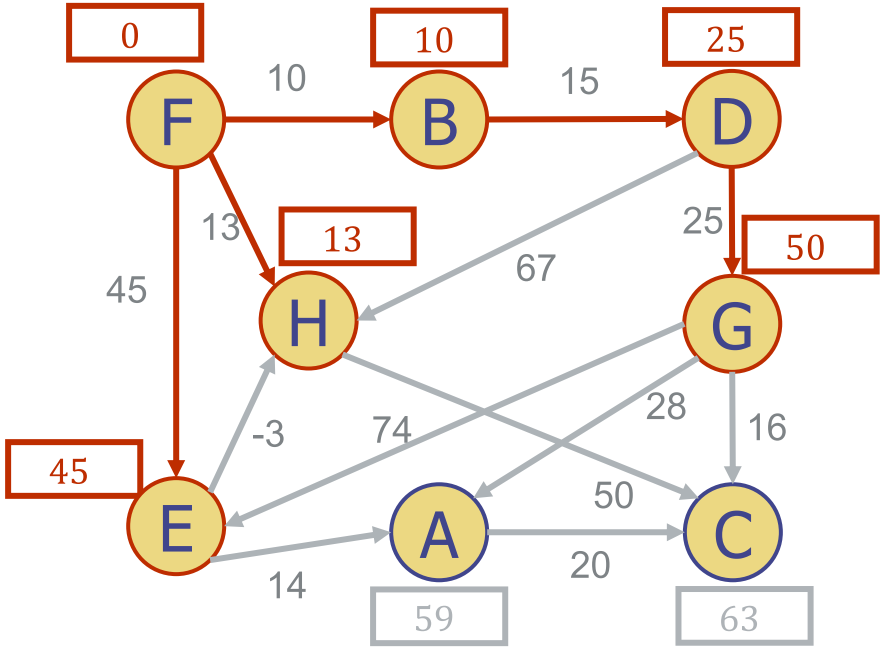


$D[v]$		C ("cloud")	
A	59	F	0
B	10	B	10
C	66	D	25
D	25	G	50
E	45	E	45
F	0	H	13
G	50		
H	13		

- Remove next vertex from topological sort - H, and add it to cloud

Single-Source Shortest Path Problem in a DAG - Example

topo
A
C

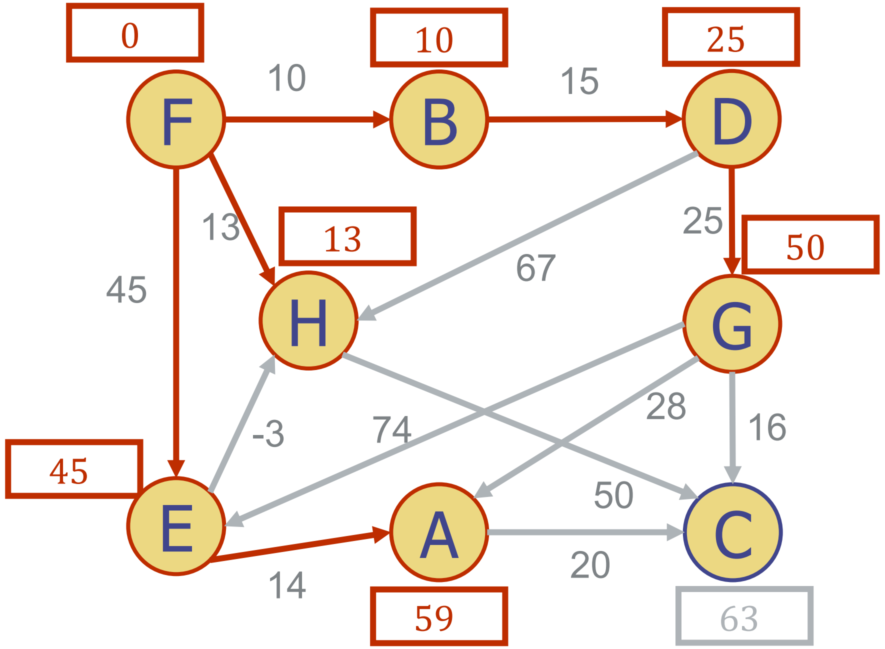


$D[v]$		C ("cloud")	
A	59	F	0
B	10	B	10
C	63	D	25
D	25	G	50
E	45	E	45
F	0	H	13
G	50		
H	13		

- Update the lengths of the paths from H to all the vertices adjacent to H
- Edge relaxation for C
  - $13 + 50 < 66$, update C path

Single-Source Shortest Path Problem in a DAG - Example

topo
C

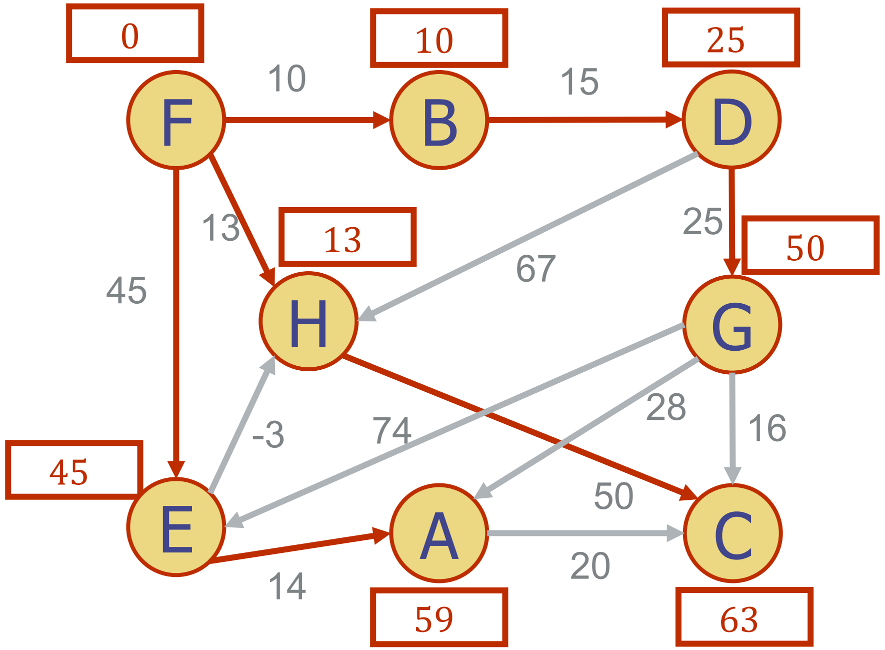


$D[v]$		C ("cloud")	
A	59	F	0
B	10	B	10
C	63	D	25
D	25	G	50
E	45	E	45
F	0	H	13
G	50	A	59
H	13		

- Remove next vertex from topological sort - A, and add it to cloud

Single-Source Shortest Path Problem in a DAG - Example

topo

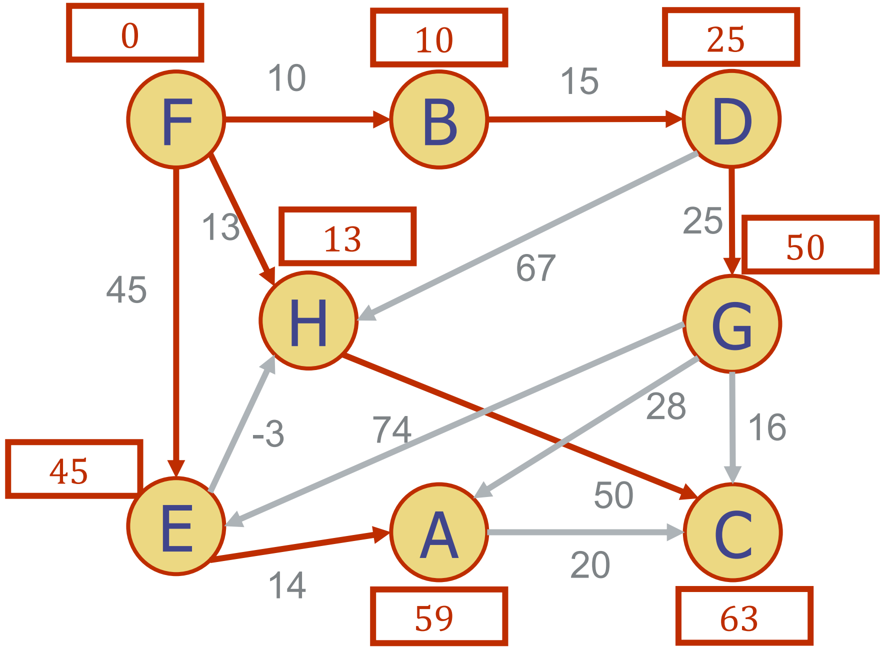


$D[v]$		C (“cloud”)	
A	59	F	0
B	10	B	10
C	63	D	25
D	25	G	50
E	45	E	45
F	0	H	13
G	50	A	59
H	13	C	63

- Remove next vertex from topological sort - C, and add it to cloud

Single-Source Shortest Path Problem in a DAG - Example

topo



$D[v]$		C ("cloud")	
A	59	F	0
B	10	B	10
C	63	D	25
D	25	G	50
E	45	E	45
F	0	H	13
G	50	A	59
H	13	C	63

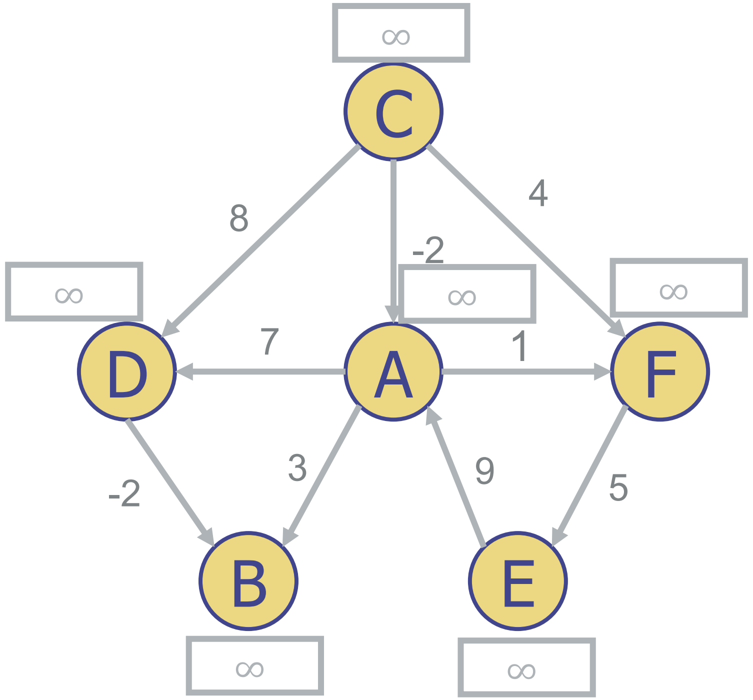
- C has no outgoing edges, so no edges to relax
- Topo empty, STOP.

Single-Source Shortest Path Problem in a Directed Graph without negative cycles

Single-Source Shortest Path Problem in a Directed Graph without negative cycles

- The single-source shortest path problem can also be solved more generally in a digraph $\vec{G}$ with n vertices and m edges
  - Even if some edges have negative weights
  - Even if there are cycles
  - The only condition is that the graph has no negative cycles
- The Bellman-Ford algorithm
  - Initialize $D[s] = 0$, $D[v] = \infty$ for every vertex in $\vec{G}$
  - Make n passes over the edges
    - At each pass, relax all the edges
- Running time $O(n \cdot m)$

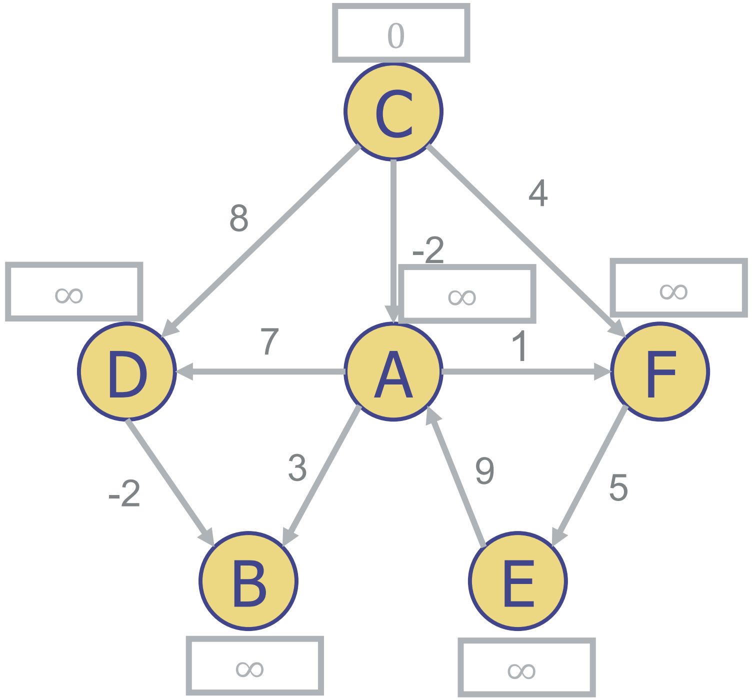
Bellman-Ford Algorithm - Example



$D[v]$	
A	∞
B	∞
C	∞
D	∞
E	∞
F	∞

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

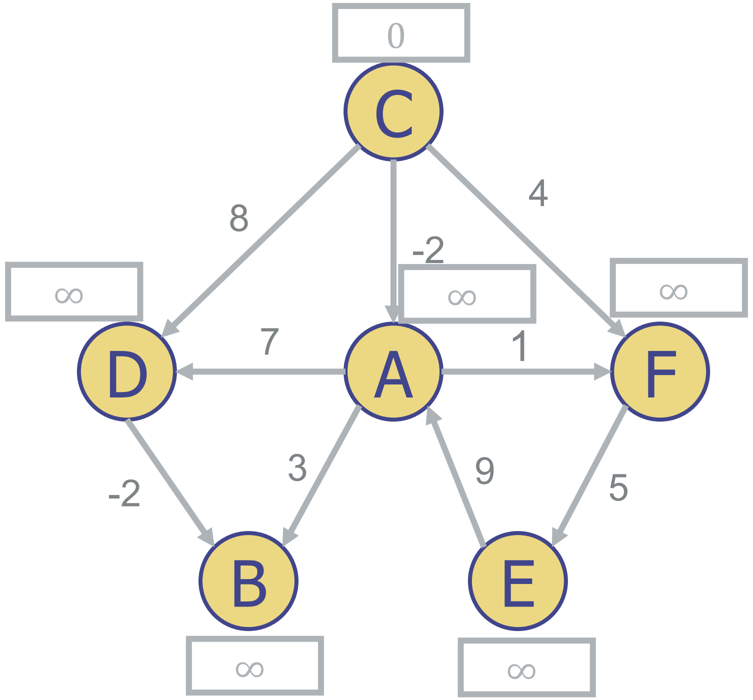


$D[v]$	
A	∞
B	∞
C	0
D	∞
E	∞
F	∞

Start from vertex C. Initially, the distance to C is 0, all the other distances are ∞.

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

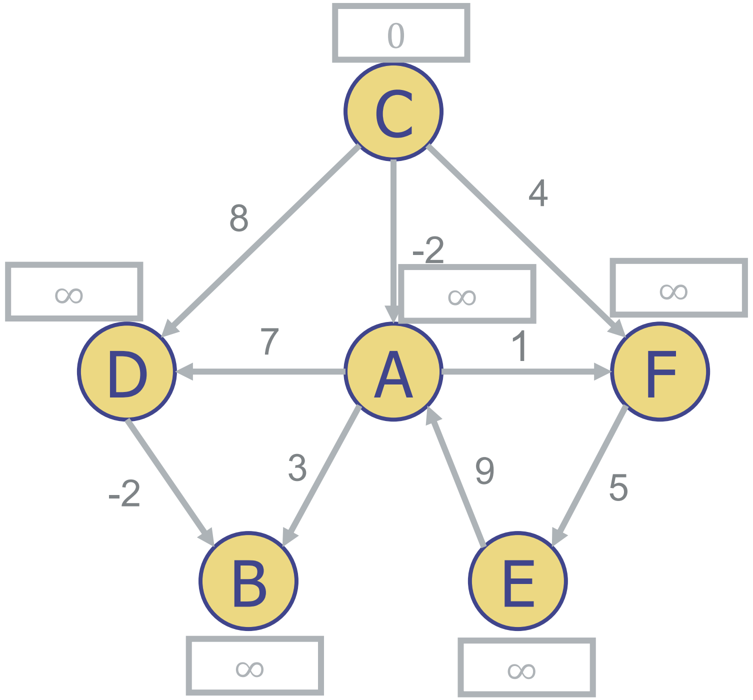


$D[v]$	
A	∞
B	∞
C	0
D	∞
E	∞
F	∞

Do the first pass, relax all edges in the given order.

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example



$D[v]$	
A	∞
B	∞
C	0
D	∞
E	∞
F	∞

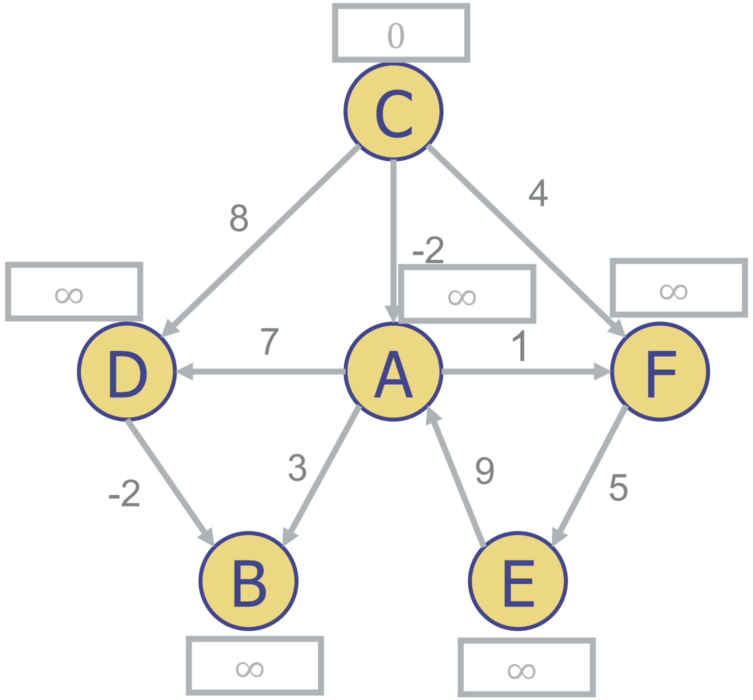
(A,B) $\infty + 3 > \infty$, no change

↓

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Bellman-Ford Algorithm - Example

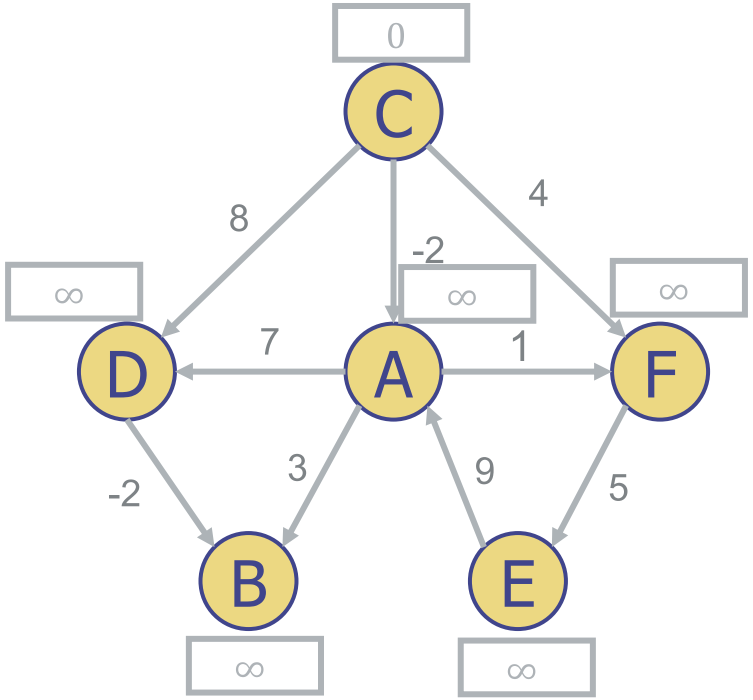


$D[v]$	
A	∞
B	∞
C	0
D	∞
E	∞
F	∞

(A,D) ∞ + 7 > ∞, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

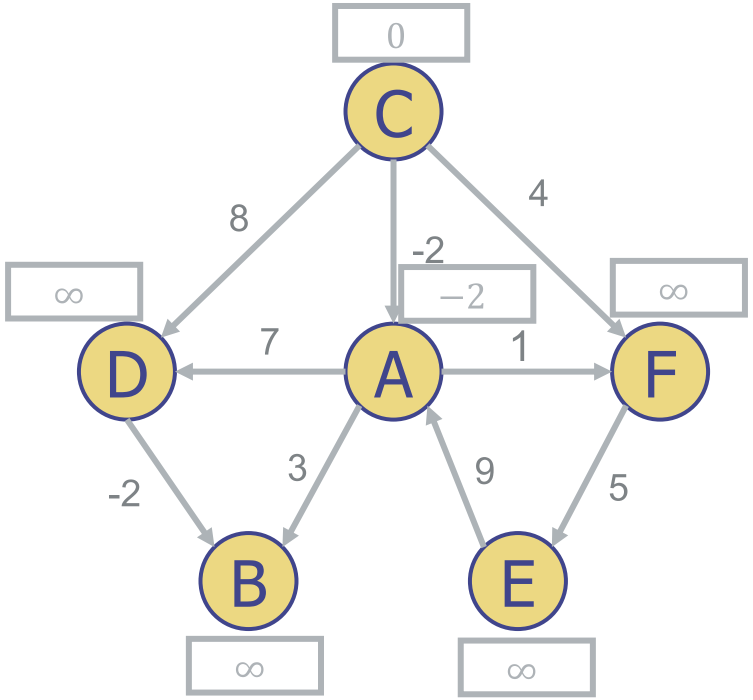


$D[v]$	
A	∞
B	∞
C	0
D	∞
E	∞
F	∞

(A,F) $\infty + 1 > \infty$, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example



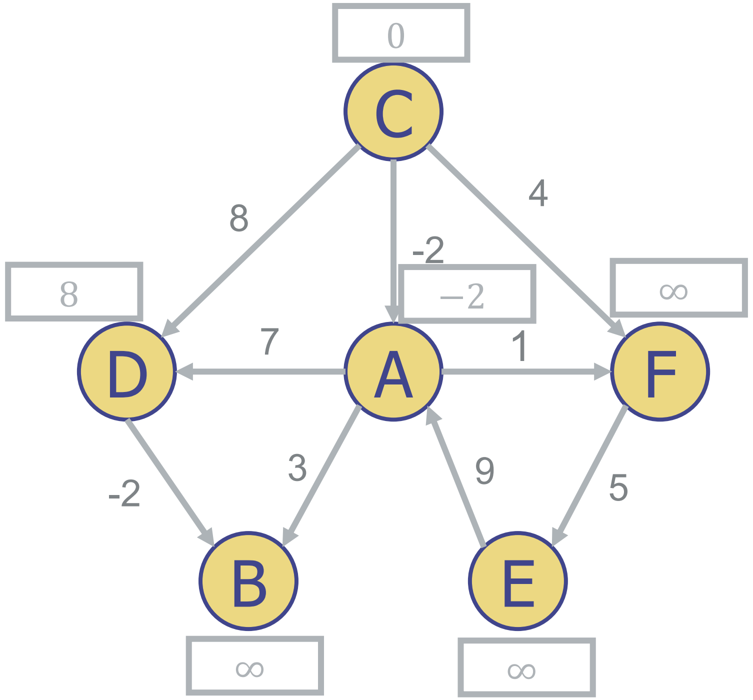
$D[v]$	
A	-2
B	∞
C	0
D	∞
E	∞
F	∞

(C,A) $0 - 2 < \infty$, update $D[A]$

↓

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

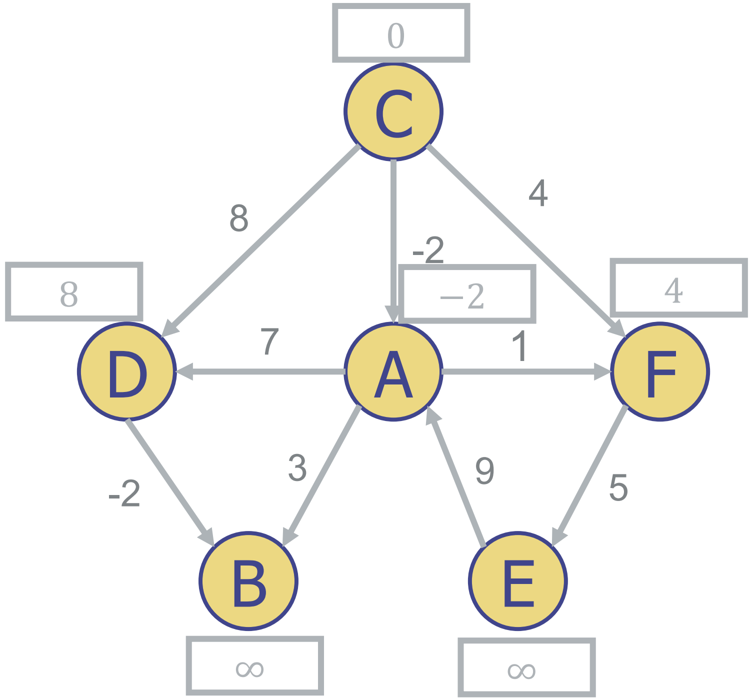


$D[v]$	
A	-2
B	∞
C	0
D	8
E	∞
F	∞

(C,D) $0 + 8 < \infty$, update $D[D]$

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example



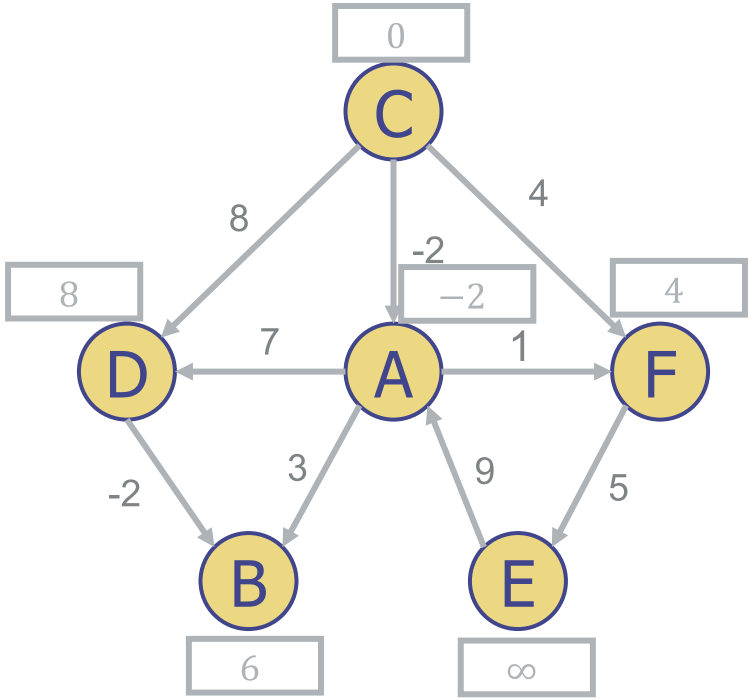
$D[v]$	
A	-2
B	∞
C	0
D	8
E	∞
F	4

(C,F) $0 + 4 < \infty$, update $D[F]$

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Bellman-Ford Algorithm - Example



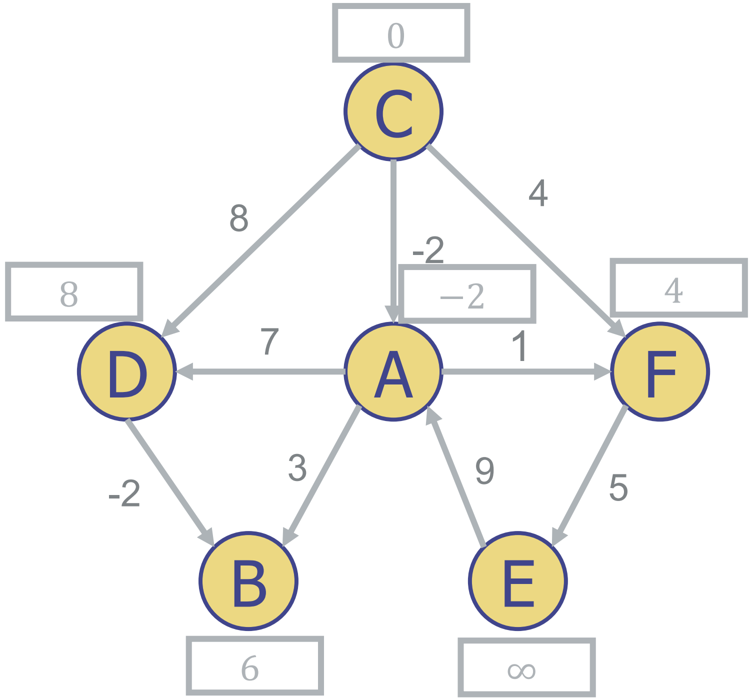
$D[v]$	
A	-2
B	6
C	0
D	8
E	∞
F	4

(D,B) $8-2 < \infty$, update $D[B]$

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Bellman-Ford Algorithm - Example

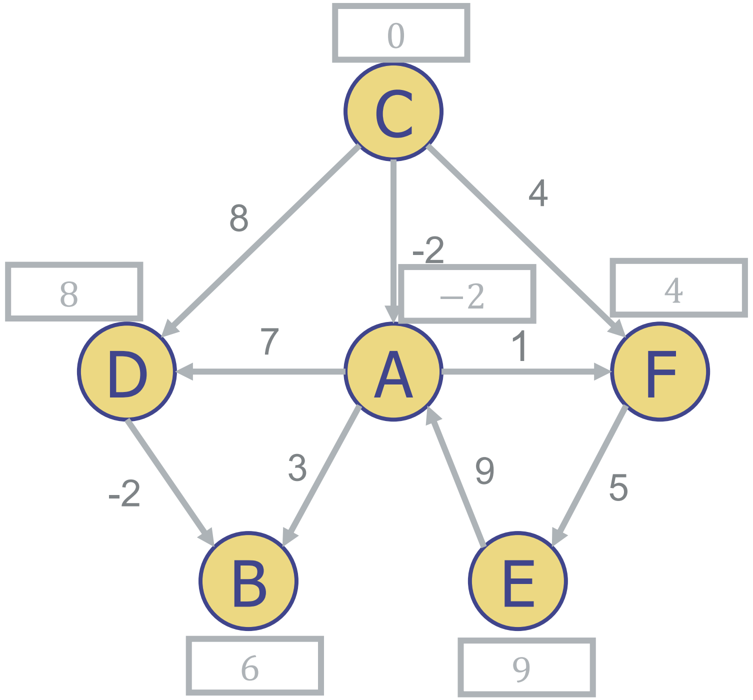


$D[v]$	
A	-2
B	6
C	0
D	8
E	∞
F	4

(E,A) $\infty + 9 > \infty$, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example



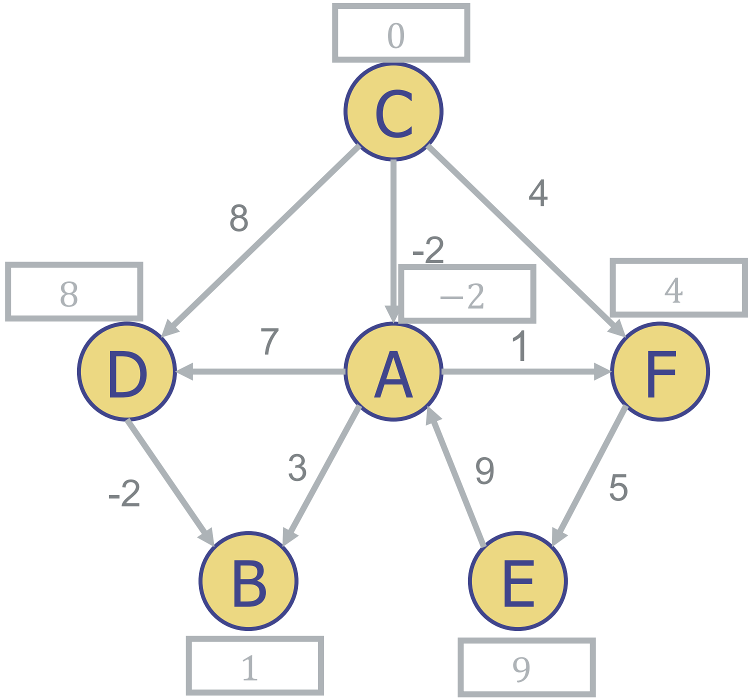
$D[v]$	
A	-2
B	6
C	0
D	8
E	9
F	4

(F,E) $4+5<\ \infty$, update $D[E]$
First pass ended, distances to 5 vertices updated. Start second pass.

↓

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example



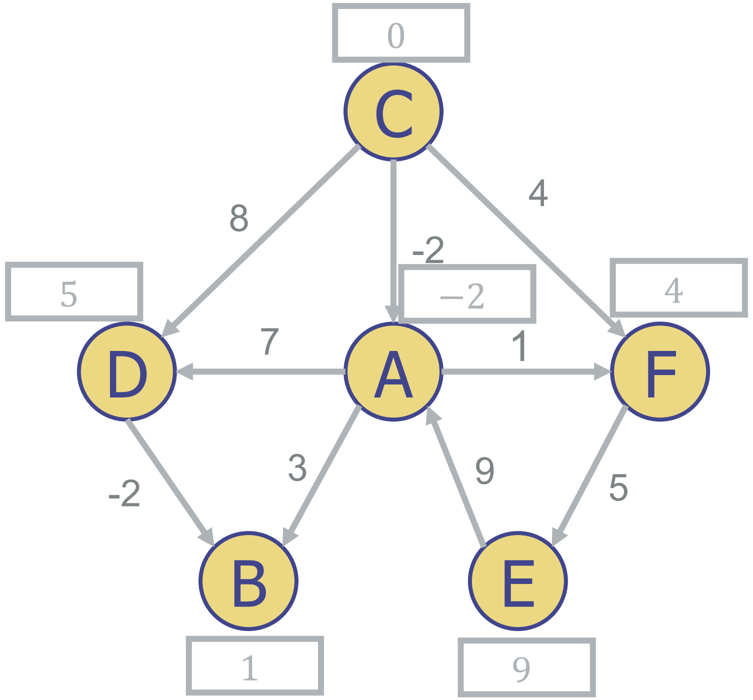
$D[v]$	
A	-2
B	1
C	0
D	8
E	9
F	4

(A,B) $-2 + 3 < 6$, update $D[B]$

↓

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

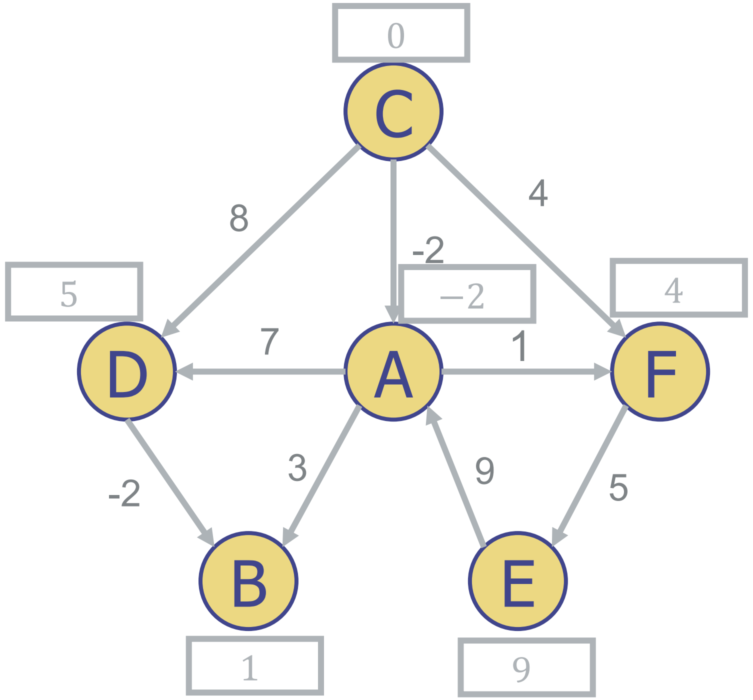


$D[v]$	
A	-2
B	1
C	0
D	5
E	9
F	4

(A,D) $-2 + 7 < 8$, update $D[D]$

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example



$D[v]$	
A	-2
B	1
C	0
D	5
E	9
F	-1

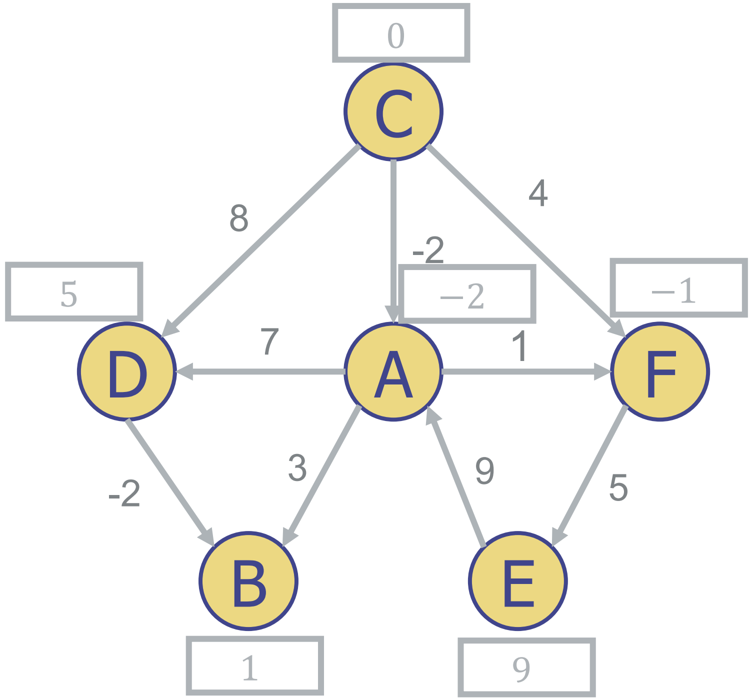
(A,F) $-2 + 1 < 4$, update $D[F]$

↓

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Bellman-Ford Algorithm - Example

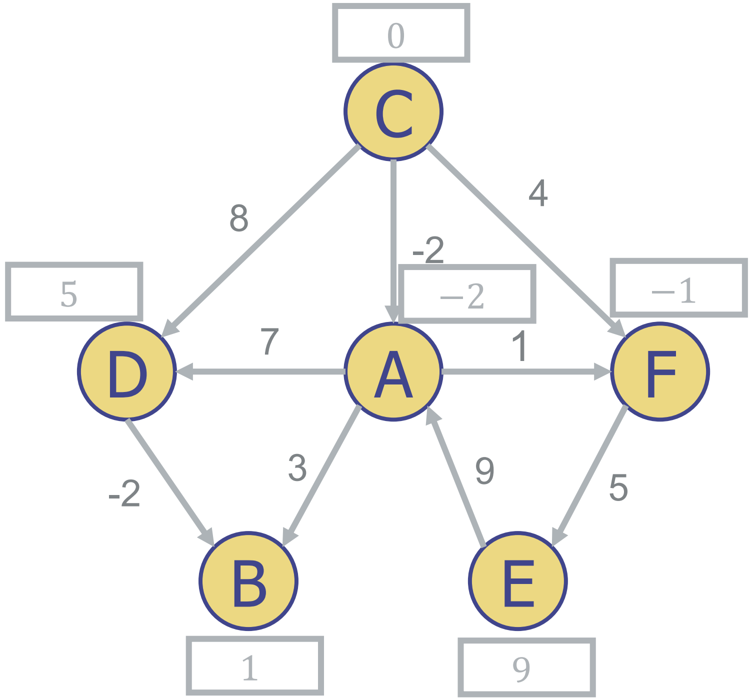


$D[v]$	
A	-2
B	1
C	0
D	5
E	9
F	-1

(C,A) $0-2=-2$, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

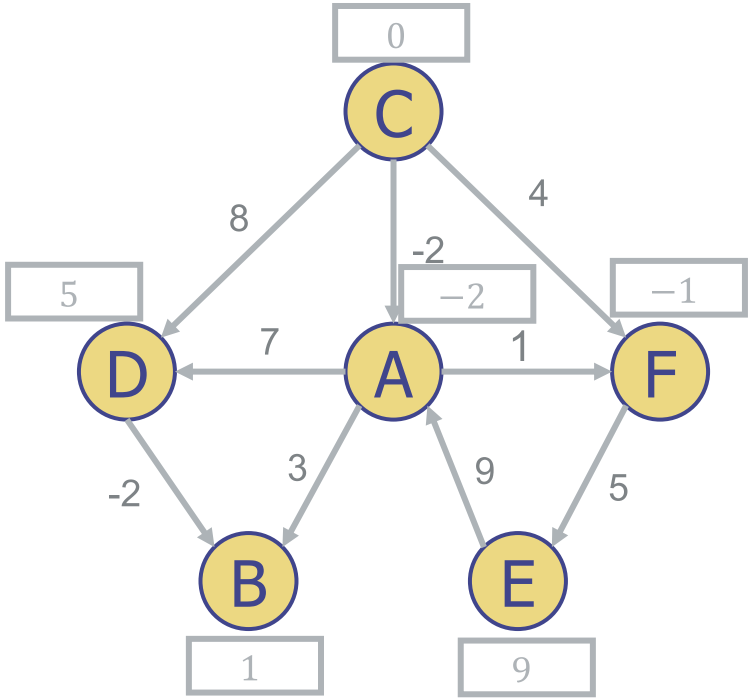


$D[v]$	
A	-2
B	1
C	0
D	5
E	9
F	-1

(C,D) $0 + 8$ > 5, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

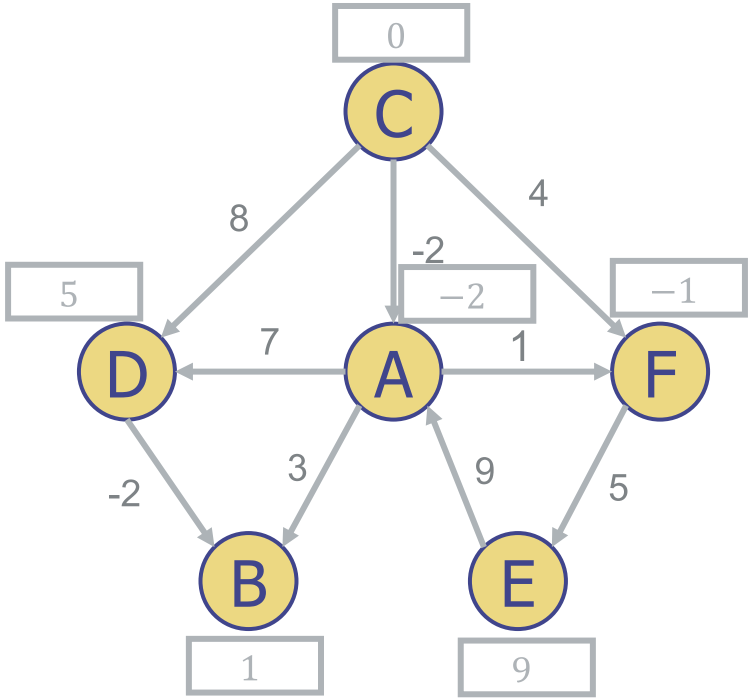


$D[v]$	
A	-2
B	1
C	0
D	5
E	9
F	-1

(C,F) $0 + 4 > -1$, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

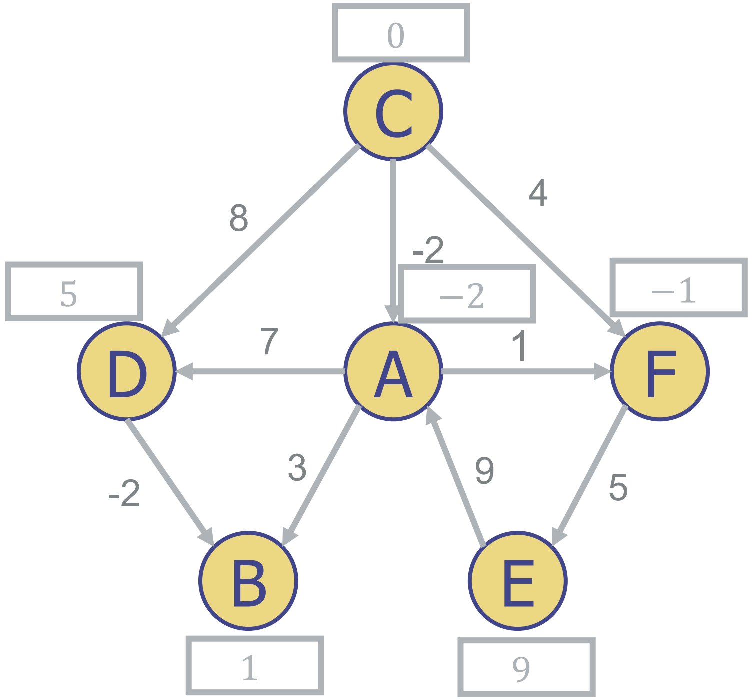


$D[v]$	
A	-2
B	1
C	0
D	5
E	9
F	-1

(D,B) $5-2\ >\ 1$, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

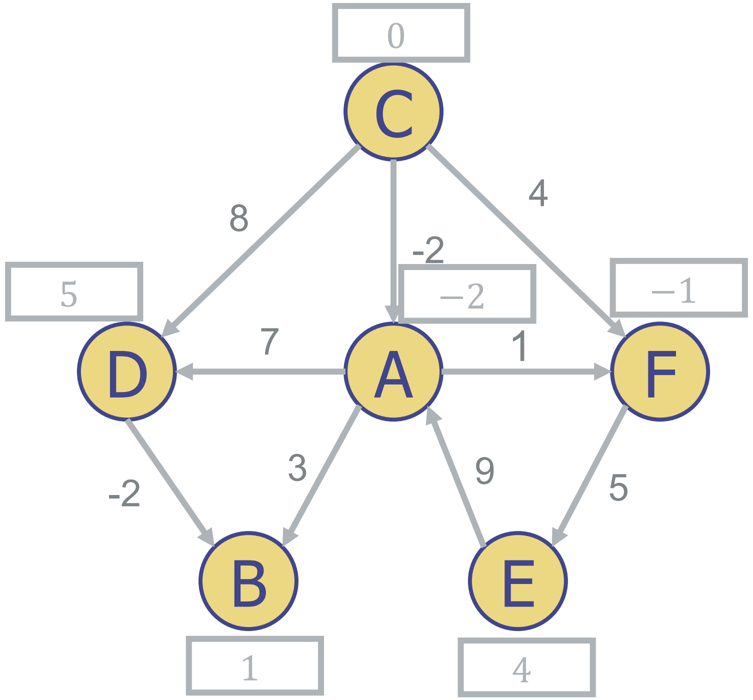


$D[v]$	
A	-2
B	1
C	0
D	5
E	9
F	-1

(E,A) $9 + 9 \ > -2$, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

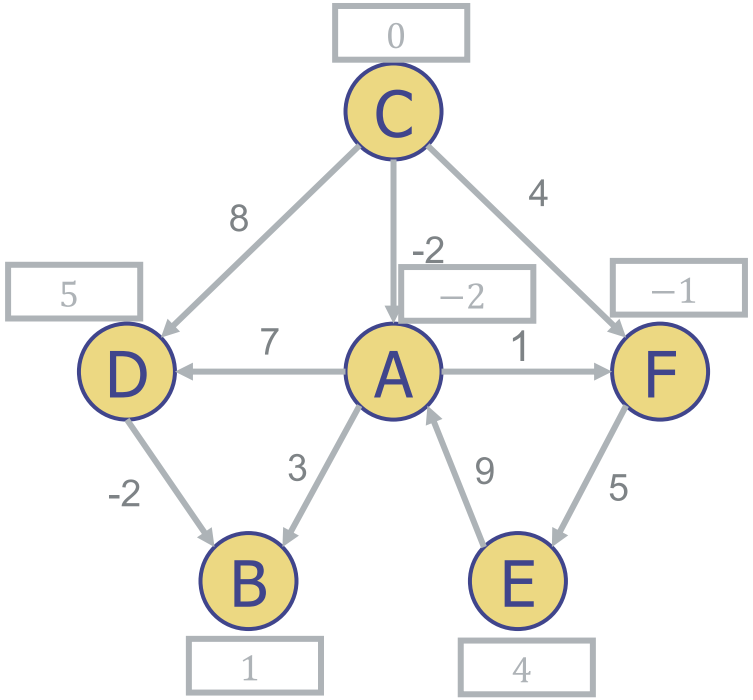


$D[v]$	
A	-2
B	1
C	0
D	5
E	4
F	-1

(F,E) $-1 + 5 < 9$, update $D[E]$
Second pass done, distances to 4 vertices updated. Start third pass.

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

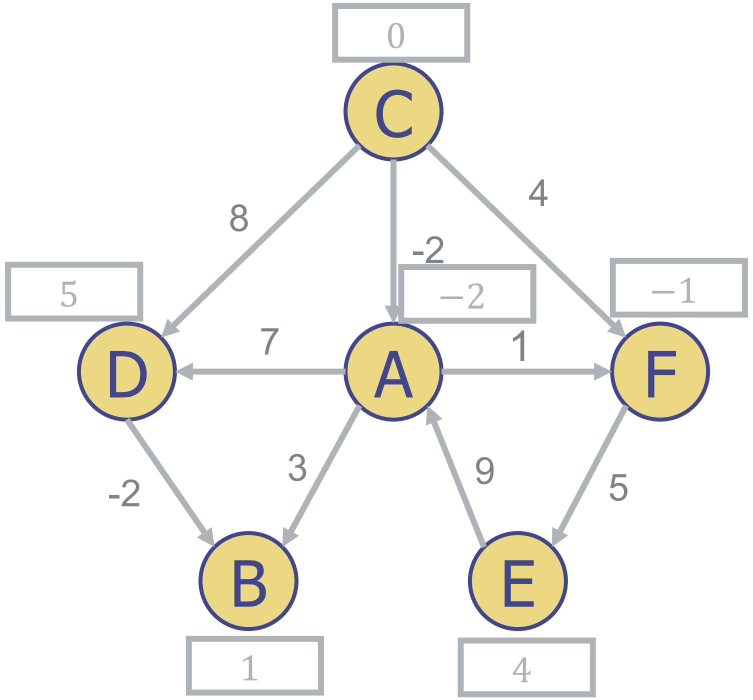


$D[v]$	
A	-2
B	1
C	0
D	5
E	4
F	-1

(A,B) $-2+3=1$, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

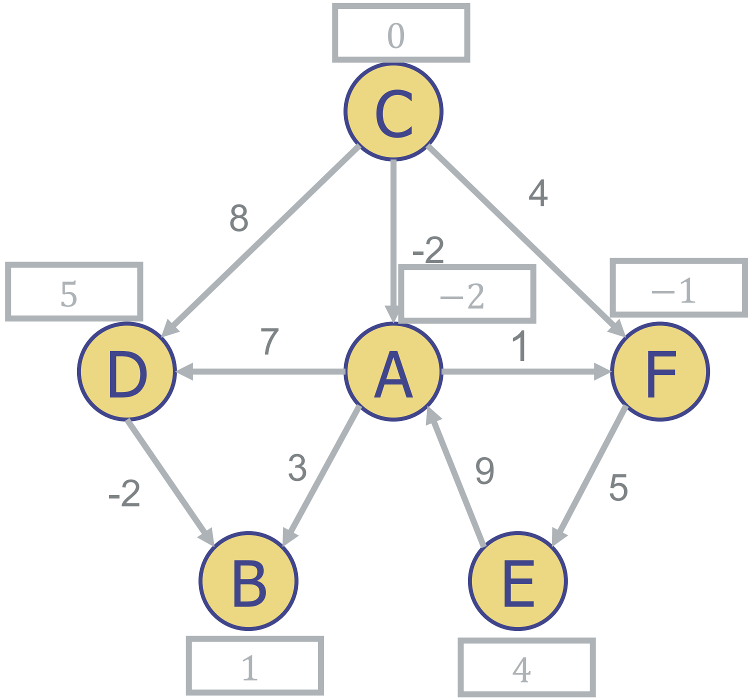


$D[v]$	
A	-2
B	1
C	0
D	5
E	4
F	-1

(A,D) $-2 + 7 = 5$, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

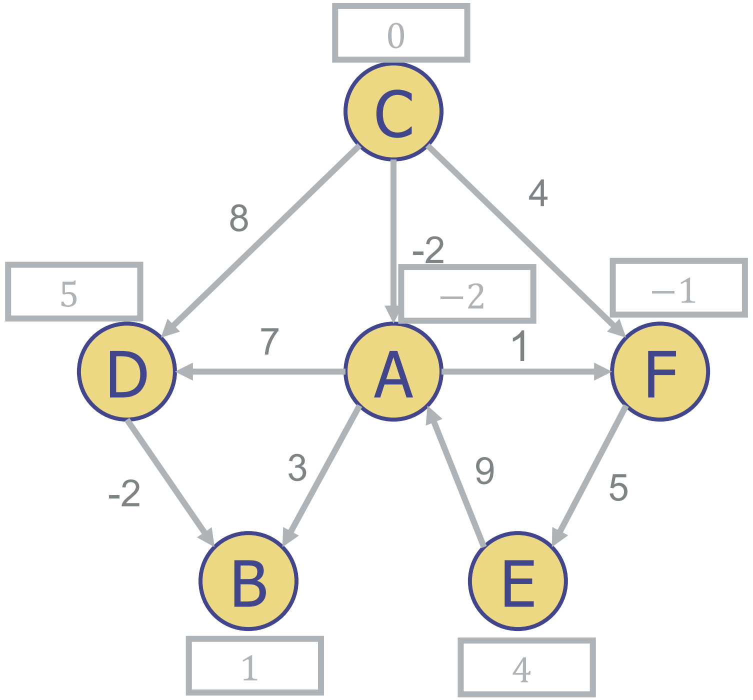


$D[v]$	
A	-2
B	1
C	0
D	5
E	4
F	-1

(A,F) $-2 + 1 = -1$, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

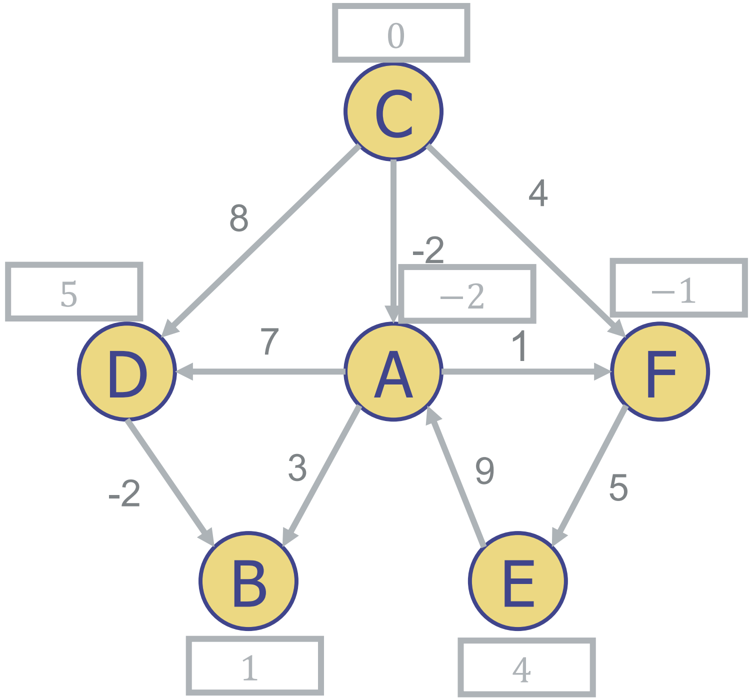


$D[v]$	
A	-2
B	1
C	0
D	5
E	4
F	-1

(C,A) $0-2=-2$, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

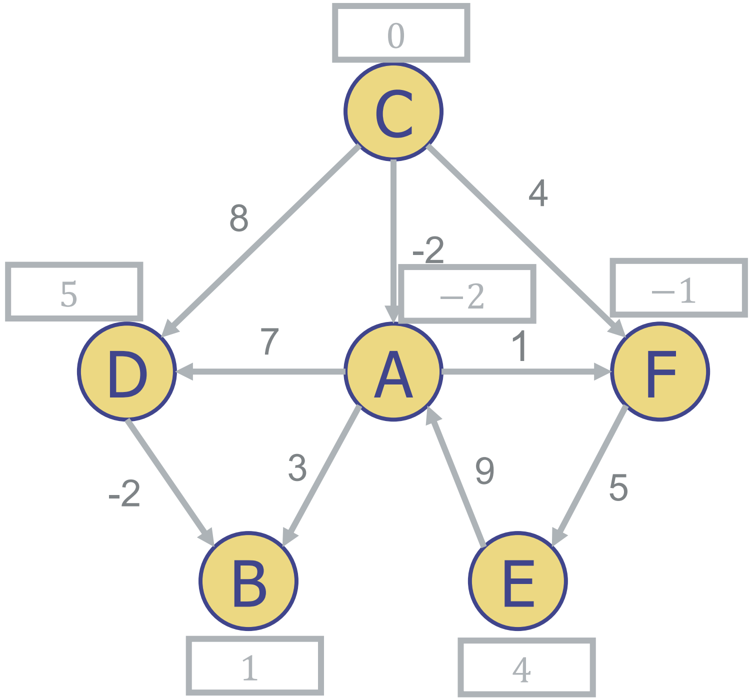


$D[v]$	
A	-2
B	1
C	0
D	5
E	4
F	-1

(C,D) $0 + 8 > 5$, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

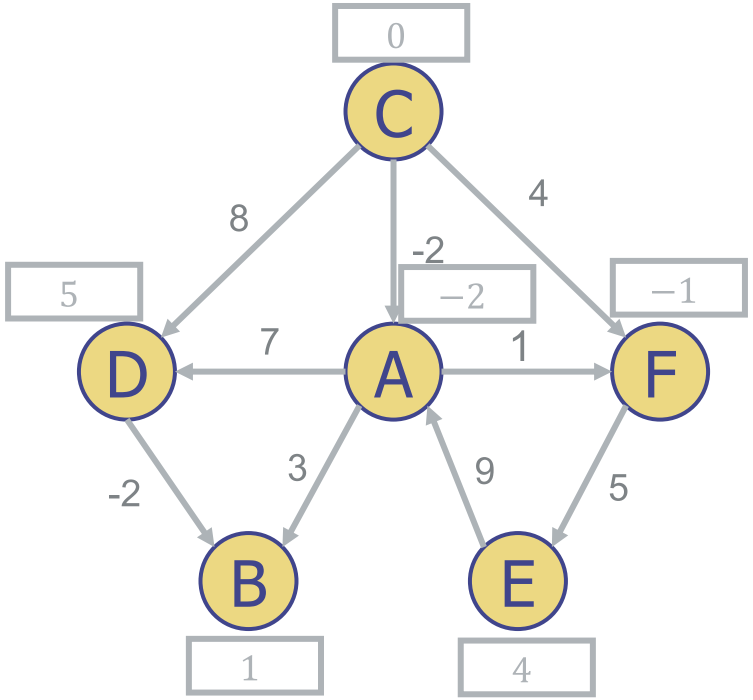


$D[v]$	
A	-2
B	1
C	0
D	5
E	4
F	-1

(C,F) $0 + 4 > -1$, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) ↓ (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

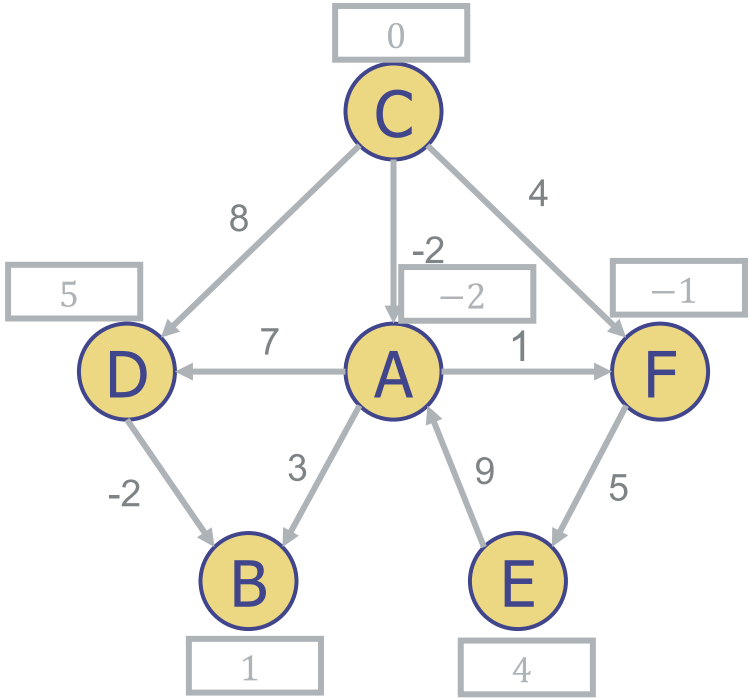


$D[v]$	
A	-2
B	1
C	0
D	5
E	4
F	-1

(D,B) $5 - 2 > 1$, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example

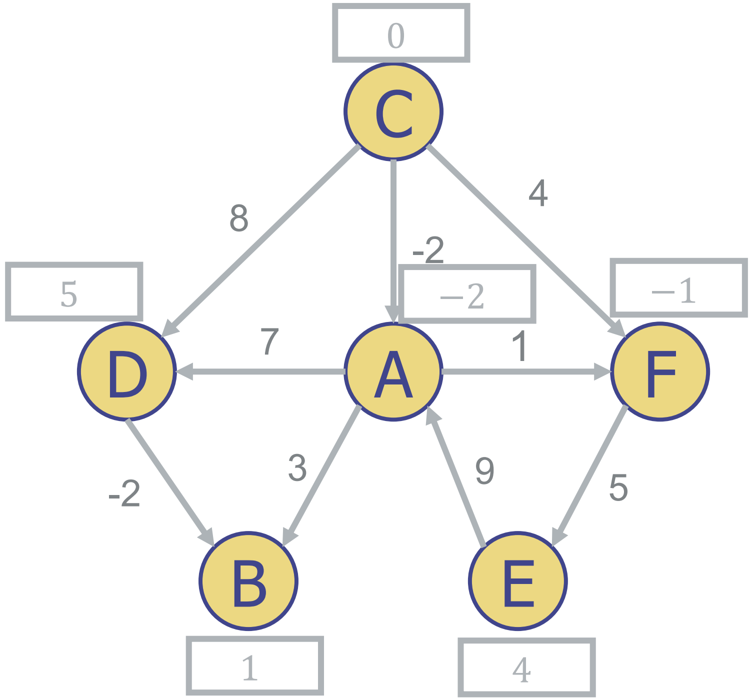


$D[v]$	
A	-2
B	1
C	0
D	5
E	4
F	-1

(E,A) $4 + 9 > -2$, no change

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Example



$D[v]$	
A	-2
B	1
C	0
D	5
E	4
F	-1

(F,E) $-1 + 5 = 4$, no change. Third pass ended with no changes. Fourth, fifth and sixth pass will also bring no changes. STOP.

Edges (A,B) (A,D) (A,F) (C,A) (C,D) (C,F) (D,B) (E,A) (F,E)

Bellman-Ford Algorithm - Note

- The algorithm can be improved by keeping track of the vertices that have changed – only those vertices will be able to generate further changes, e.g. by using a queue

Thank you.