Shortest paths

- Shortest paths problem
- Dijkstra's algorithm
- Bellman-Ford algorithm
- DAG shortest paths

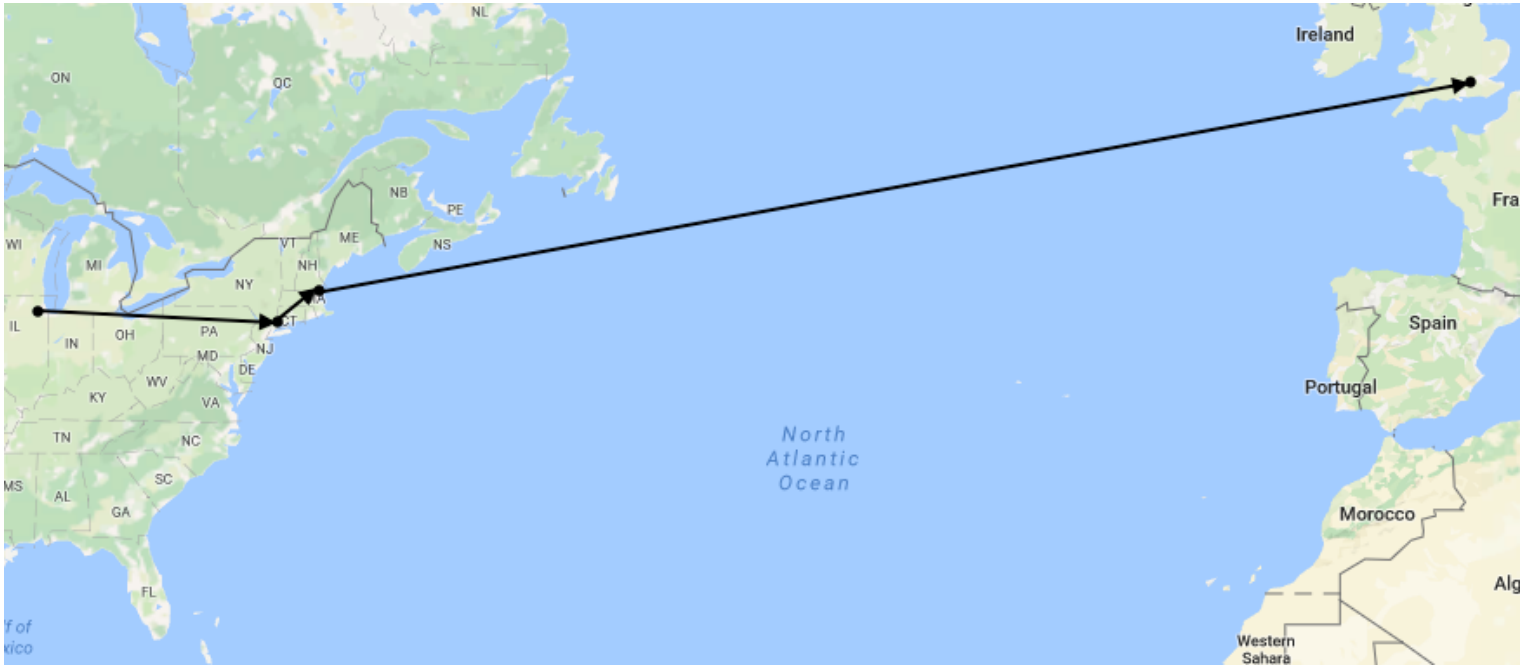
Shortest paths

BFS treats all edges as having the same length and counts # of edges to determine the length of a path

Not true in many applications where shortest paths are to be found: length of each edge is important

Example: If you want to fly from Chicago to London, then you are interested in knowing whether there is a direct flight or if you have to change planes once or if you have to change planes twice; but you are also interested in knowing the cost of each fight

If there is a way of flying from Chicago to London by changing planes twice, like this:



But there is another route in which you only need to change planes once (Chicago to Tokyo to London):

It is less expensive in terms of # times to change planes but more expensive in terms of time or flight cost

The way we handle problems of this type is by a *weighted graph*

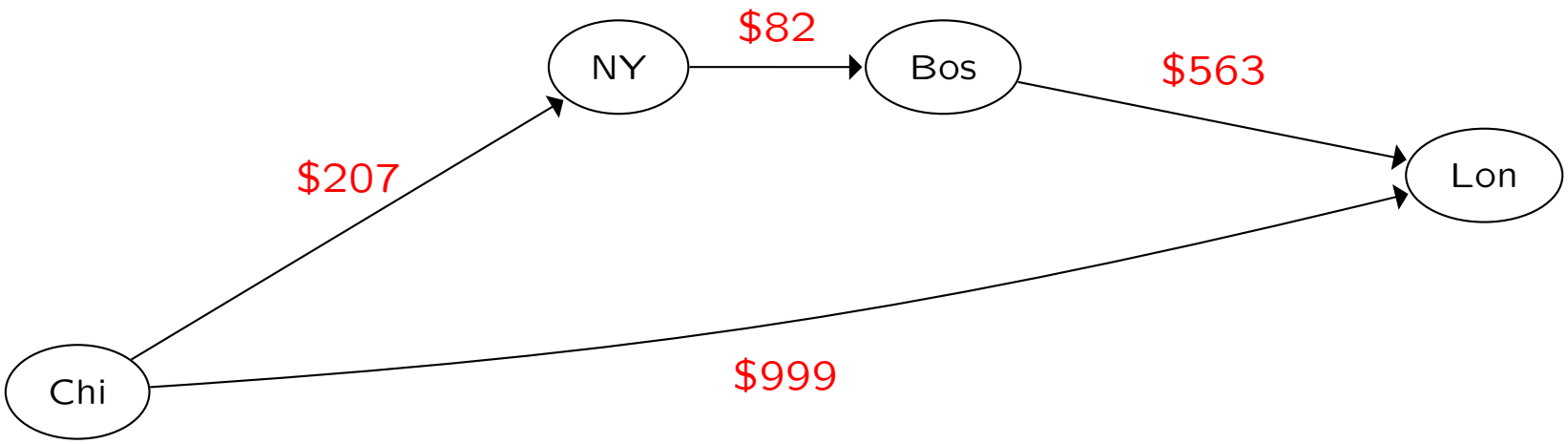
Weighted graph: a directed graph $G = (V, E)$ with a weight function w:

$$G = (V, E, w)$$

where $w : E \to \mathbb{R}$ is a function which associates a real number $w(u, v)$ to each edge $(u, v) \in E$

Edge weights $w(u, v)$ can correspond to physical lengths, time, money, or any other quantity to be conserved

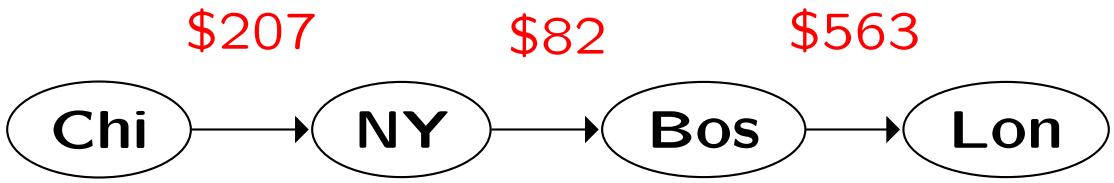
Example: Edge weights (red) correspond to costs of individual flights (British Airways: 11/7/16)



Weight of path $p = v_1 \to v_2 \to \cdots \to v_k$ is:

$$w(p) = w(v_1, v_2) + \cdots + w(v_{k-1}, v_k) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

Example:



$$w(p) = \$207 + \$82 + \$563 = \$852$$

What is a shortest path?

Shortest path from $u \in V$ to $v \in V$ is:

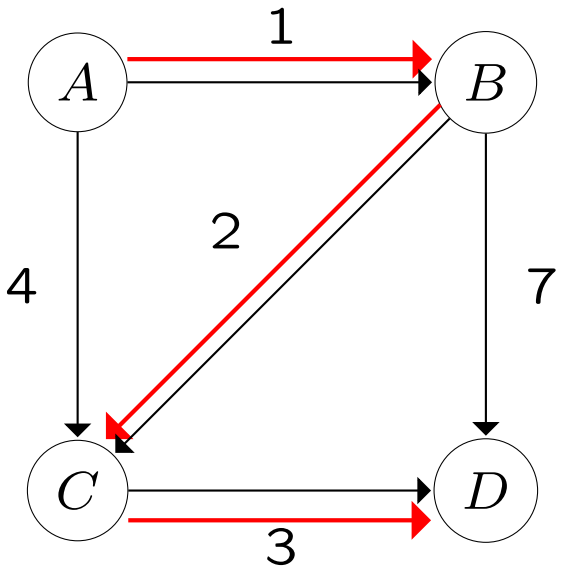
a path p of minimum weight $w(p)$ from u to v
(among all paths from u to v)

Shortest path weight $\delta(u, v)$:

weight of such a path is shortest path weight:

Def : $\delta(u, v) = \min_p \{w(p) : p \text{ is a path from } u \text{ to } v\}$

Example:



shortest path (red) is

$A \to B \to C \to D$

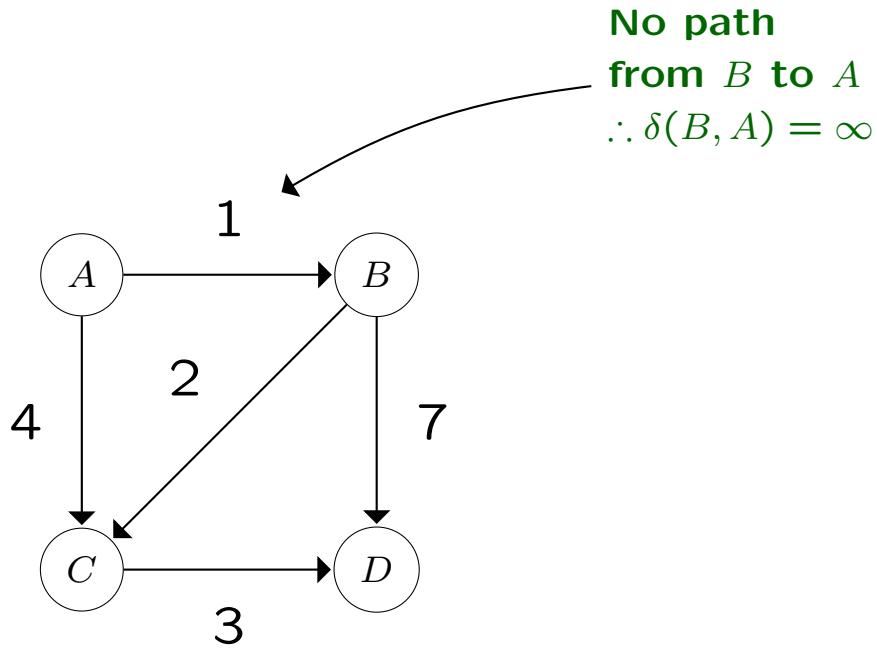
$\delta(A, D) = 6$

When do shortest paths not exist?

- Case 1: If no path from u to v, then there is no shortest path from u to v

Def : $\delta(u, v) = \infty$ if no path from u to v

Example:

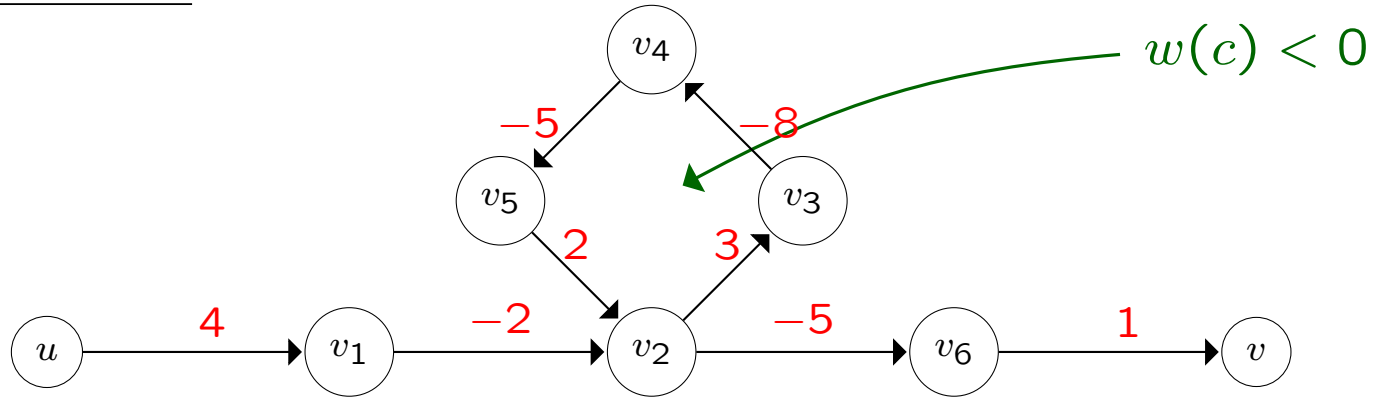


- Case 2: There may be a path from u to v, but no shortest path

negative-weight cycle in graph $\Rightarrow$ some shortest paths may not exist

negative-weight cycle: $c = v_i \to \cdots \to v_j \to v_i$ has $w(c) < 0$

Example:



$\delta(u, v) = -\infty$

Argument: As long as there is a negative-weight cycle reachable from u that can also reach v, can always get a shorter path by going around cycle again

Def: $\delta(u, v) = -\infty$ if negative-weight cycle along any path from u to v

Brute force algorithm:

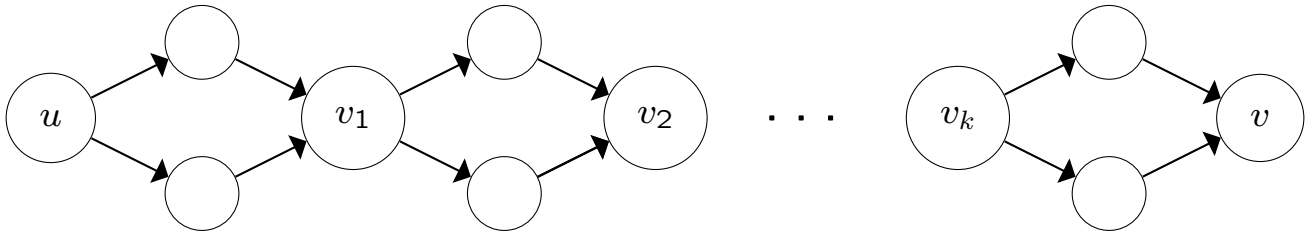
shortest-path(G, u, v)// G in Adj list format; $u, v \in V$

1 **for** each path p from u to v
2 compute $w(p)$
3 **return** p with minimum $w(p)$

Problems with this algorithm?

- infinite time if $\delta(u, v) = -\infty$
- if $\delta(u, v)$ finite, exponential # of paths can exist

Example:



How many different paths from u to v in this graph?

Single-source shortest-paths problem

Input: directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, and source vertex $s \in V$

Goal: compute $\delta(s, v)$ for all $v \in V$;
compute shortest-path tree containing a shortest path from s to each $v \in V$
(represented by storing $\pi[v]$ for each $v \in V$)

What is an efficient algorithm for this problem?

Dijkstra's algorithm:

- Generalization of BFS to handle weighted graphs
- Only allows nonnegative edge weights
- Finds shortest path weights from source s to all $v \in V$
- Actual paths can be easily reconstructed
- Uses priority queue Q keyed by $d[v]$
  (BFS uses FIFO queue)

Dijkstra:

Input: (G, w, s)

- $G = (V, E)$, directed, adj list format
- weight function $w : E \to \mathbb{R}$ $(\forall e \in E)(w(e) \geq 0)$
- $s \in V$ source vertex

Output:

- $d[v]$: distance = weight of a shortest path from s to v
- $\pi[v]$: parent/predecessor of v on shortest paths tree

Guarantee: Path found from s to each vertex v is a shortest path

Note: it is possible that there are several shortest paths. Dijkstra will not represent all shortest paths but will represent a shortest path between source and vertex v

Idea of algorithm:

Maintain a current cost to every vertex:

$$d[v] : \text{current min cost of reaching vertex } v$$

Imagine that we are searching the web for flights from Chicago to London. Find we can fly for $728, so write it down. Next search shows we can fly for $987, so discard that. Next find we can fly for $466, so write it down. You keep reducing. So you maintain keys, and only change to those keys is always to go down. Never increase a key.

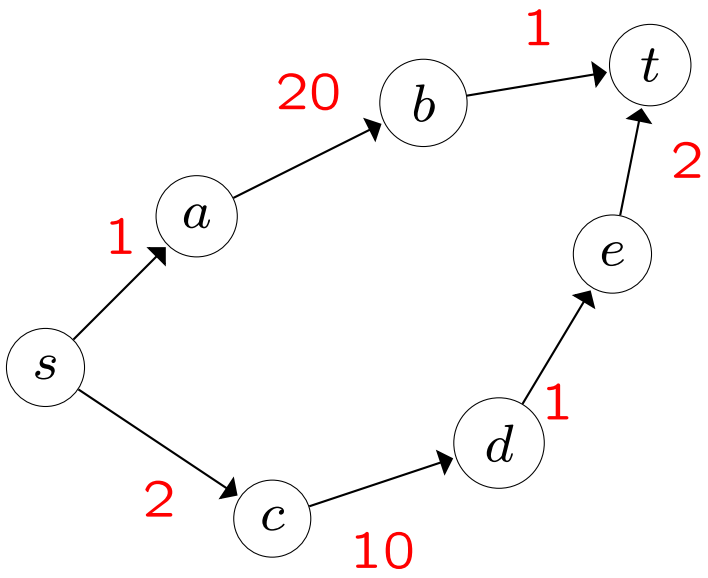
Therefore this algorithm is going to use the decrease-key operation in some data structure that we set up separately.

What else do we need? A parent/predecessor vertex:

$$\pi[v] : \text{parent of } v$$

Suppose we are searching for a path from source s to target t. We find a path (s, a, b, t) that is reasonably inexpensive: $w(p) = 22$. We make b the parent of t.

Later we come across another path (s, c, d, e, t) that is even less expensive $w(p) = 15$. We update the parent: $\pi[t] = e$.



The parents keep changing. Each time we update, each time we find there is a less expensive way to get somewhere, the parent is changed.

At some point we have to know that the number associated with the vertex, $d[v]$, is the min cost and that its value is never going to change. The vertex is finalized.

Code for Dijkstra's algorithm

```
Dijkstra(G, w, s)// G in Adj list format
1   Initialize-Single-Source(G, s) // d[v] = ∞ except d[s] = 0
2   S = ∅
3   Q = V // priority queue, keyed by d[v]
4   while Q ≠ ∅
5         u = Extract-Min(Q)
6         S = S ∪ {u}
7         for each vertex v ∈ Adj[u]
8               Relax(u, v, w)// implicit Decrease-key op on v in Q
```

Initialize-Single-Source(G, s)

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

Relax(u, v, w)

1 **if** $d[v] > d[u] + w(u, v)$
2 $\quad d[v] = d[u] + w(u, v)$
3 $\quad \pi[v] = u$

There are two separate lists:

- S: set of vertices whose shortest paths from s have been finalized

- Q: list of vertices prioritized by $d[v]$ values

What does the algorithm do?

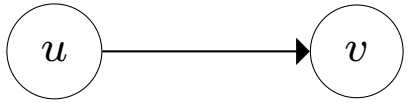
It takes the minimum current-cost vertex from Q
Initially only the source s has finite cost $d[s] = 0$
Call this vertex u

Now add u to S
This changes the status of u: i.e., its current cost $d[u]$ is its final cost: it will not be changed any more

Now we look at the neighbors of u
Any neighbors with an infinite cost will be updated to a finite cost
What is the finite cost going to be?

Relaxing an edge (u, v)

Consider edge (u, v):

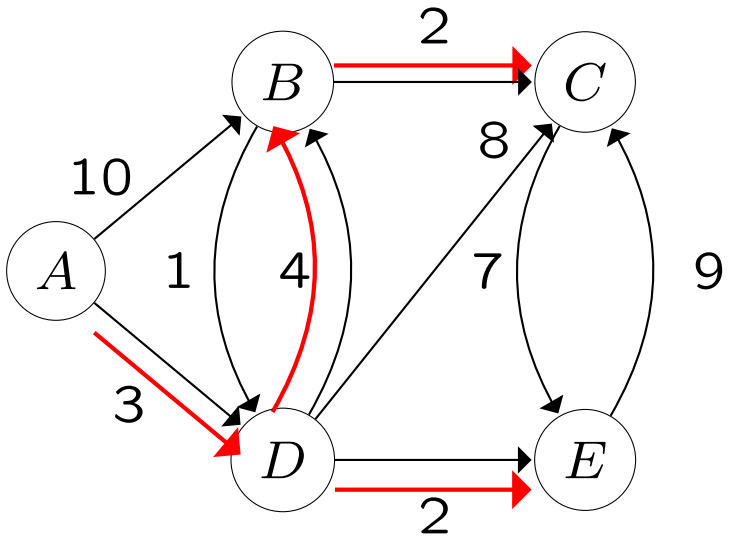


If $d[v] > d[u] + w(u, v)$, set $d[v] = d[u] + w(u, v)$, i.e., $d[v]$ is decreased by going through u and taking the edge (u, v)

Setting $d[v]$ updates Q (Decrease-Key)

This is what the algorithm does: it keeps updating neighbors and removing vertices one at a time, and eventually the set of vertices Q will become empty

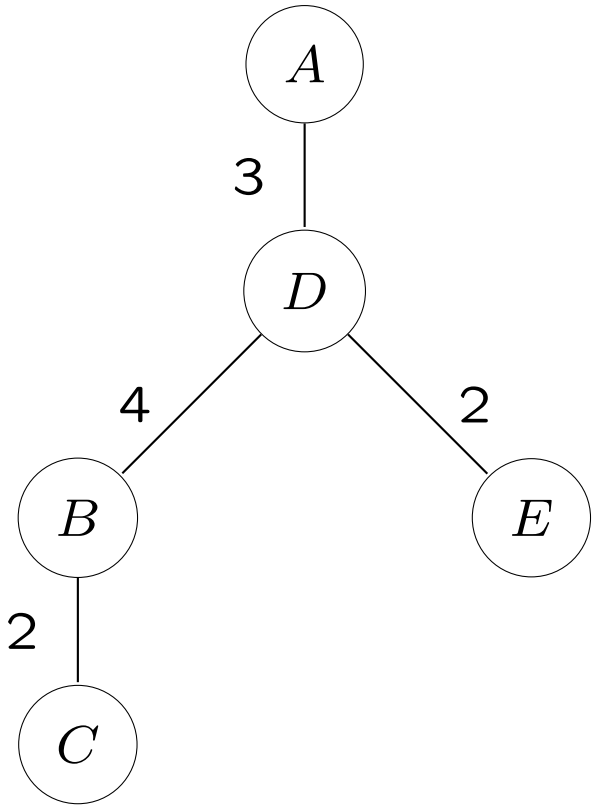
Example:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	∞	3	∞
		7	11		5
		7	11		
			9		

	A	B	C	D	E
d	0	7	9	3	5
π	N	D	B	A	D

Shortest paths tree:



Analysis

Consider different priority queue Q implementations:

- Array

A priority queue can be implemented as an unordered array of key values for the vertices of the graph

Initially, these values are set to ∞

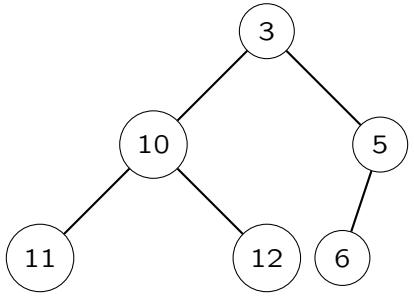
Decrease-Key is $O(1)$ because it only involves adjusting a key value

Extract-Min requires a $O(V)$ time scan of the array

- Binary Min-heap

Key values are stored in a *complete* binary tree, i.e., a binary tree in which each level is filled from left to right and must be full before the next level is started

A special ordering constraint is enforced: the key value of any node of the binary tree $\leq$ key values of its children; the root always contains the minimum element:



To Extract-Min, return the root value. To remove this element from the min-heap, take the last node, i.e., node in the rightmost position in bottom row, and place it at the root. If it is larger than either child, exchange it with the smaller child and repeat. The number of exchanges is at most the height of the tree, which is $O(\lg V)$ when there are $|V|$ elements.

To Decrease-Key, decrease the value of the element at its current position. If it is smaller than its parent, exchange the two and repeat. Again, this takes $O(\lg V)$ time.

Running time:

- $|V|$ executions of Extract-Min (each vertex deleted exactly once)
- $|E|$ executions of Decrease-Key (each edge relaxed exactly once)
- $|Q| \leq |V|$

implementation	Extr-min	Decr-Key	Dijkstra
unsorted array	$O(V)$	$O(1)$	$O(V^2 + E) = O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O((V + E) \lg V)$

Which implementation is preferable?

- For all graphs, $|E| \leq |V|^2$
- If graph is dense ($\Omega(V^2)$), array implementation is faster
- If graph is sparse ($|E| = O(V)$), binary heap is faster

Binary heap implementation is fast:

- nearly linear time: $O((V + E) \lg V)$
- One pass through edges; need logarithmic time to pick next edge to relax

Correctness

Theorem: When u extracted from Q, $d[u] = \delta(s, u)$

Proof: By induction on extraction order

Basis step: $d[s] = 0 = \delta(s, s)$ b/c no negative cycles (b/c no negative edge weights)

Let $S =$ vertices already extracted from Q

By inductive hypothesis, $d[v] = \delta(s, v)$ for all $v \in S$

Consider shortest path p from s to u: $w(p) = \delta(s, u)$

Consider first edge (x, y) where $x \in S$ and $y \notin S$: exists because $s \in S$ and $u \notin S$

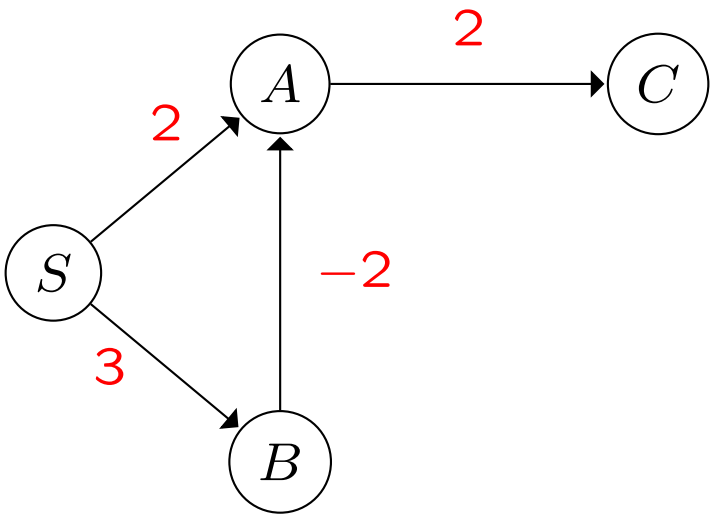
When x was extracted from Q, $d[x] = \delta(s, x)$ and we relaxed (x, y)

$\Rightarrow d[y] \leq \delta(s, x) + w(x, y) \leq w(p) \leq d[u]$ (since $w(p) = \delta(s, u)$ and $\delta(s, u) \leq d[u]$)

but u was chosen to have min d-value $\Rightarrow d[u] = d[y] = \delta(s, u) = \delta(s, y)$ q.e.d.

Corollary: at end of Dijkstra's algorithm, $d[u] = \delta(s, u)$ for all $u \in V$

Example where Dijkstra's algorithm fails



Q:	S	A	B	C
	0	∞	∞	∞
		2	3	∞
			3	4
		1		4

	S	A	B	C
d	0	1	3	4
π	N	B	S	A

What happened? $\delta(S, C) = 3$ with path $S \to B \to A \to C$, but Dijkstra outputs $d[C] = 4$ along path $S \to A \to C$

Observe: A is extracted from Q with a value $d[A] > \delta(S, A) = 1$

When B is selected and (B, A) relaxed, $d[A]$ is recalculated for A and $d[A]$ is updated to the correct value. But the edge (A, C) (the only edge to C) had already been relaxed, so $d[C]$ was never updated.

Moral: Need to relax edges more than once if allow negative edges

Bellman Ford algorithm

Now describe an algorithm for single-source shortest paths problem when the graph has negative weights

As with Dijkstra's algorithm, maintain a variable $d[v]$ which represents our current best estimate as to what the shortest path weight is from the source s to v:

- $d[v]$: current best estimate of shortest path weight from s to v
- Initially $d[s] = 0$; $d[v] = \infty$ for all $v \neq s$
- Idea: Build a tree of shortest paths by relaxing all edges of graph in turn

Recall Moral: Edges need to be relaxed more than once for the end result to be a shortest paths tree

Belllman-Ford algorithm simply relaxes all the edges (in arbitrary order), then relaxes them all again, and repeats this step until all edges have been relaxed $|V|-1$ times

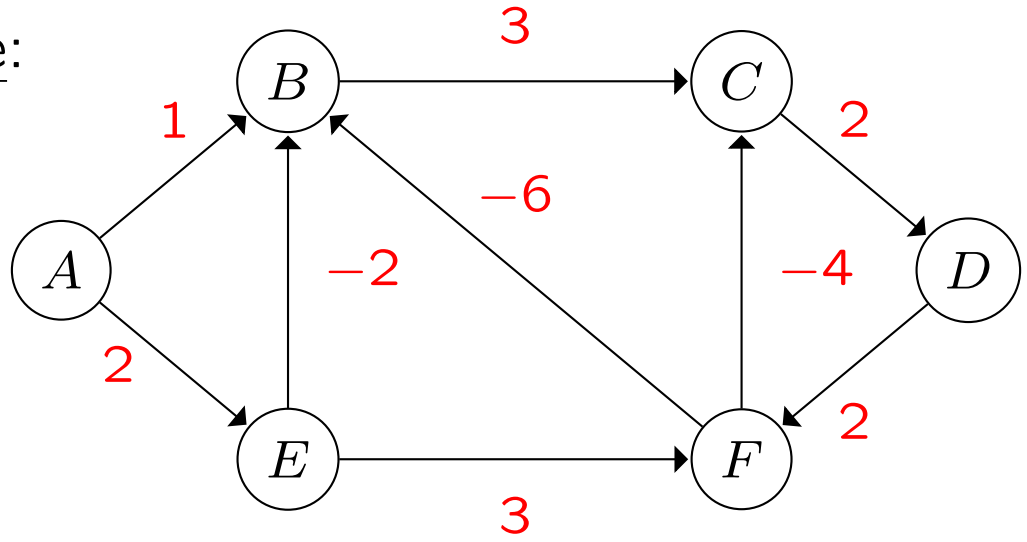
```
Bellman-Ford(G, w, s)
1   Initialize-Single-Source(G, s)
2   for i = 1 to |V| − 1
3         for each edge (u, v) ∈ E
4               Relax(u, v, w)
5   for each edge (u, v) ∈ E
6         if d[v] > d[u] + w(u, v)
7               return FALSE // report that negative cycle exists
8   return TRUE
```

Running time?

Three sections of code:

- Initialize d, which will converge to the shortest path value δ, and π
- Relaxation: $|V| - 1$ times, do the relaxation step for each edge
- Test whether got a solution (gets solution if and only if no negative-weight cycles)

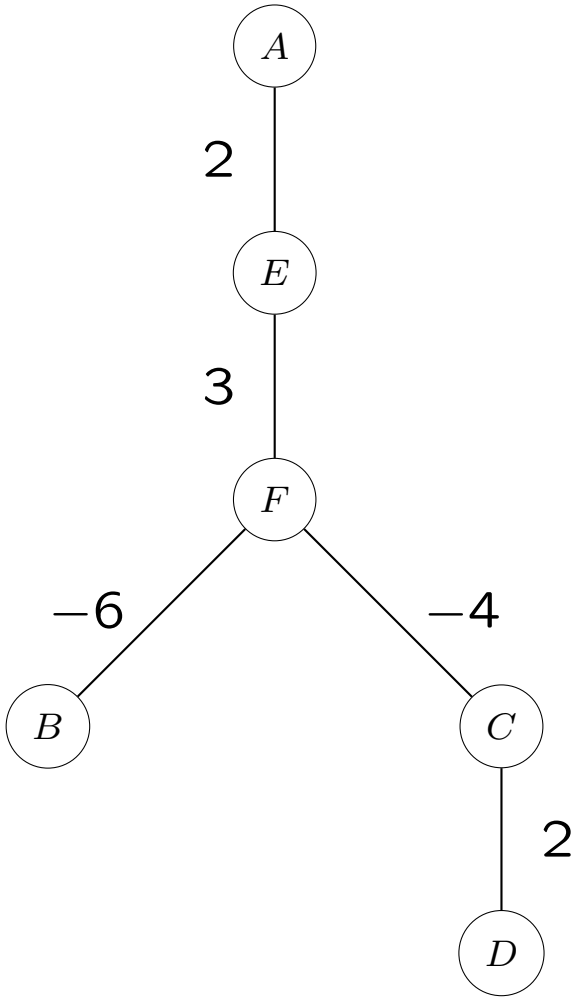
Example:



	A	B	C	D	E	F
d	0	∞	∞	∞	∞	∞
		1	4	6	2	8
		0	1			5
		-1				
				3		

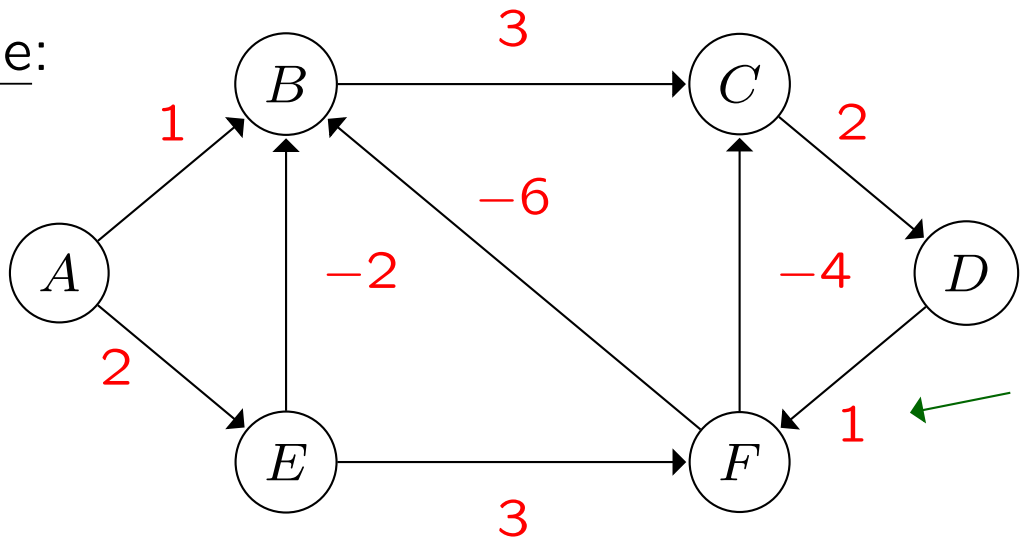
	A	B	C	D	E	F
π	N	A	B	C	A	D
		E	F			E
		F				

Shortest paths tree:



What happens if there is a negative-weight cycle?

Example:



	A	B	C	D	E	F
d	0	∞	∞	∞	∞	∞
		1	4	6	2	7
		0	1			5
		-1				
		-2	0	3		4
		-3	-1	2		3

	A	B	C	D	E	F
π	N	A	B	C	A	D
		E	F			E
		F				
						D

Correctness: why does this algorithm work?

Key invariant: d-values are always either overestimates or exactly correct

Start at ∞: only way ever change d is by relaxing along an edge:

- Relax(u, v, w)
- $d[v] = \min\{d[v], d[u] + w(u, v)\}$

This Relax (update) operation is simply an expression of the fact that the distance to v cannot possibly be more than the distance to u, plus $w(u, v)$. It has the following properties:

1. It gives the correct distance to v when u is the 2nd-to-last vertex in the shortest path to v and $d[u]$ is correctly set.

2. It will never make $d[v]$ too small. $\therefore$ extra Relax's cannot hurt

Look at a shortest path from s to t:

$$s \to v_1 \to v_2 \to \cdots \to u_k \to t$$

This path can have at most $|V| - 1$ edges (Why?)

- Initially $d[s]$ is correctly set, assuming no negative weight cycles

- If sequence of Relax's includes (s, v_1), (v_1, v_2), ..., (v_k, t) *in that order*, then by property 1 above the distance to t will be correctly computed.

If we do not know all the shortest paths beforehand, how can we be sure to update/Relax the right edges in the right order? Easy: Relax them all, $|V| - 1$ times

Like other DP problems, this algorithm calculates the shortest paths in a bottom-up manner

- It 1st calculates the shortest path weights for the shortest paths with at most 2 edges, and so on
- After the ith iteration of the outer loop, the shortest path with at most i edges are calculated
- There can be a maximum of $|V| - 1$ edges in any simple path, so outer loop runs $|V| - 1$ times

Shortest paths in directed acyclic graphs (DAGs)

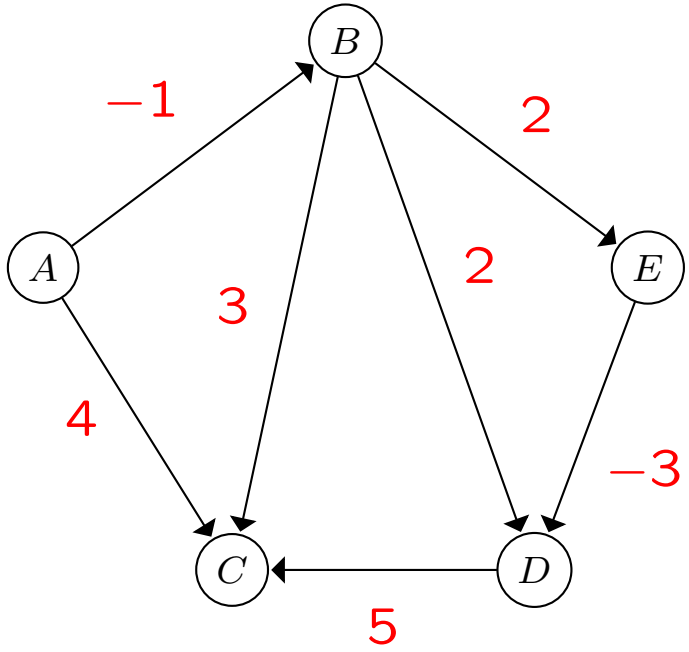
Two subclasses of graphs automatically exclude possibility of negative cycles:

- Graphs without negative edges
- Graphs without cycles

We know how to handle the former efficiently

Now see how the single-source shortest path problem can be solved on DAGs

Example: Relax edges in order: $AB, AC, BC, BD, BE, ED, DC$

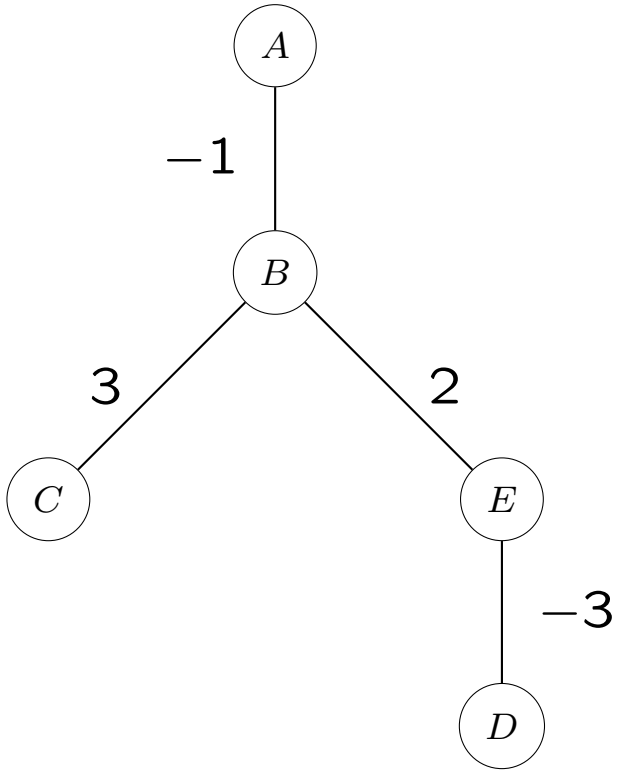


	A	B	C	D	E
d	0	∞	∞	∞	∞
		-1	4	1	1
			2	-2	

	A	B	C	D	E
π	N	A	A	B	B
		E	B	E	

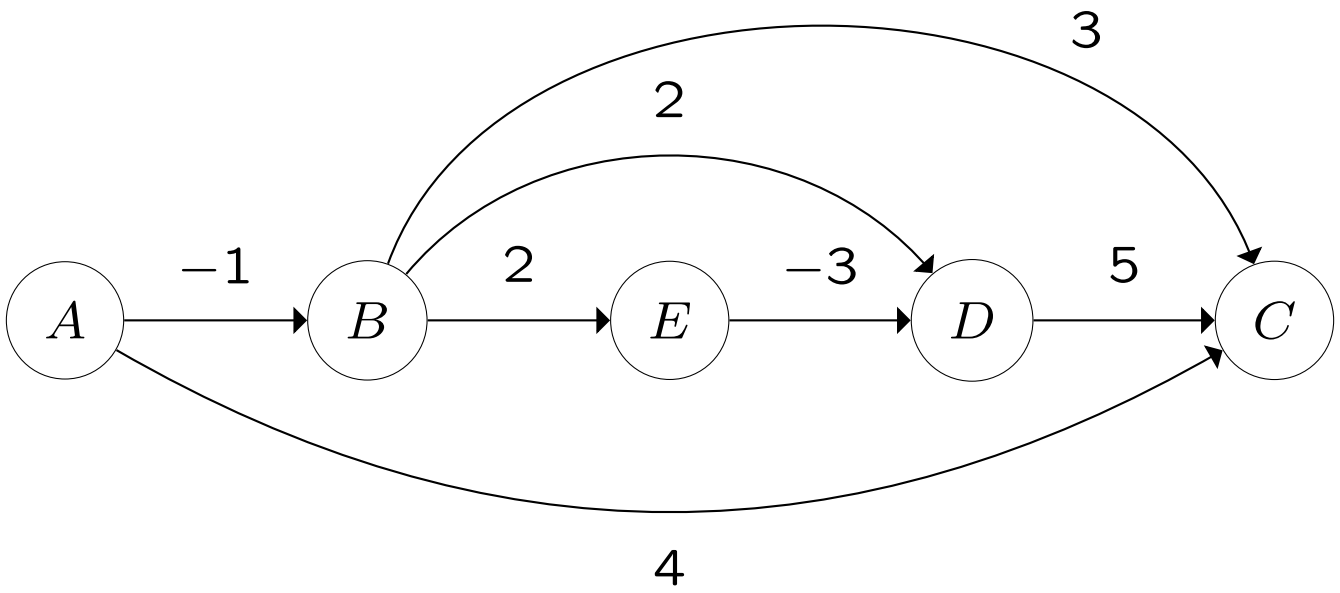
Done after one iteration!

Shortest paths tree:



How did we get the edge order: $AB, AC, BC, BD, BE, ED, DC$?

Topological sort!



DAG-Shortest-Paths(G, w, s)

```
1   for each vertex v ∈ V
2       d[v] = ∞
3       π[v] = NIL
4   d[s] = 0
5   topologically sort vertices V
6   // now (u, v) ∈ E ⇒ rank(u) < rank(v) in V
7   for each vertex u ∈ V (in topologically sorted order)
8       for each v ∈ Adj[u]
9           Relax(u, v, w)
```

Running time: $O(V + E)$

Note: This algorithm does not require the edges to have nonnegative weight

Correctness

Theorem: In a DAG, this algorithm sets $d[u] = \delta(s, u)$ for all $u \in V$.

Proof: By induction, $d[u] = \delta(s, u)$ when we encounter u in the outer loop

Basis step: $d[s] = 0$ correct (no cycles) (other base cases have $d[u] = \infty$)

Inductive step: When we encounter u, already processed all previous vertices, including all vertices with edges into u

By inductive hypothesis, these vertices had correct d-values when we relaxed the edges into u.

Since all edges into u have already been relaxed, the edge into u on a shortest path has been relaxed.

$\therefore d[u] = \delta(s, u)$, as required. q.e.d.