Shortest Paths

March $19^{th}/21^{st}$

Outline For Today

1. SSSP in DAGs: DP Algorithm
2. SSSP without Negative Edges: Dijkstra's Greedy Algorithm
3. SSSP with Negative Edges: Bellman Ford DP Algorithm
4. All pairs Shortest Paths: Floyd Warshall DP Algorithm

Shortest Paths Problems

- Input is G(V, E) with edge weights
- Shortest Paths from a single source s to all/one destination in DAGs (DP Solution)
- Shortest Paths from a single source s to all/one destination in general graphs with no negative edge weights (Dijkstra: Greedy)
- Shortest Paths from all sources to all dests (Floyd Warshall: DP).

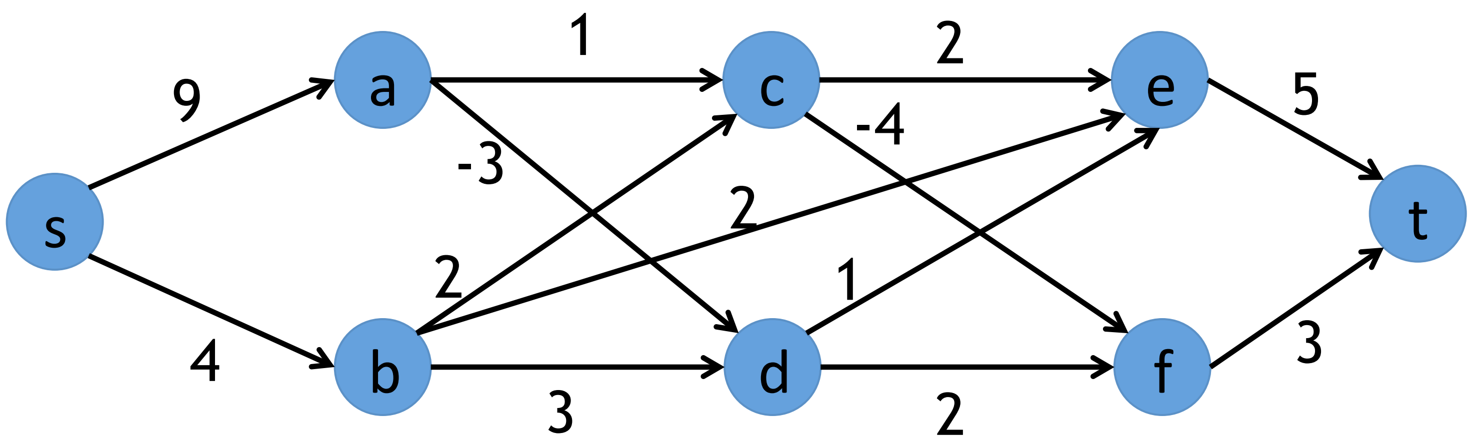
Outline For Today

1. SSSP in DAGs: DP Algorithm
2. SSSP without Negative Edges: Dijkstra's Greedy Algorithm
3. SSSP with Negative Edges: Bellman Ford DP Algorithm
4. All pairs Shortest Paths: Floyd Warshall DP Algorithm

Shortest Paths In DAGs

- Input: weighted DAG G(V, E) with arbitrary edge weights and source s
  - Note edge weights can be negative
- Output: shortest paths from s to all vertices

Shortest Paths In DAGs



Q: Shortest path (distance) from s to e?

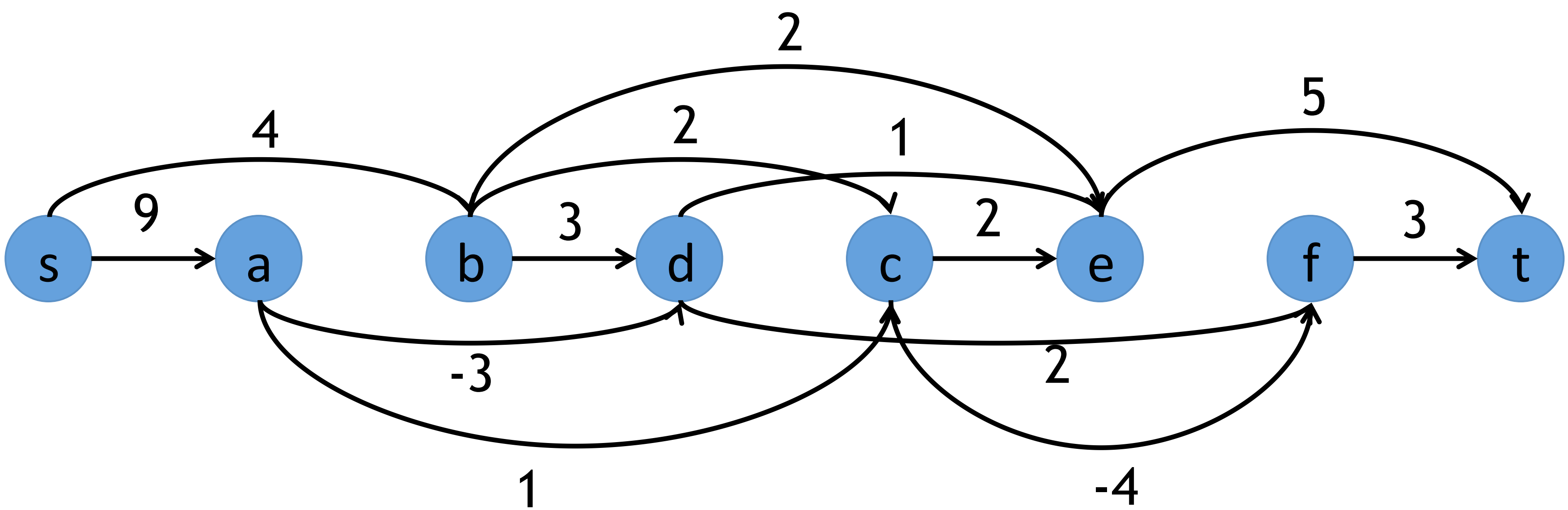
A: 6: s->b->e

Let's think of a DP solution.

Defining Subproblems

- Recall Linear IS:
  - Line graph was naturally ordered from left to right.
  - Subproblems could be defined as prefix graphs.
- Recall Sequence Alignment:
  - X, Y strands were naturally ordered strings.
  - Subproblems could be defined as prefix strings.
- Trick: Use the Topological ordering of G and solve shortest paths for "prefix graphs" again.

Defining Subproblems

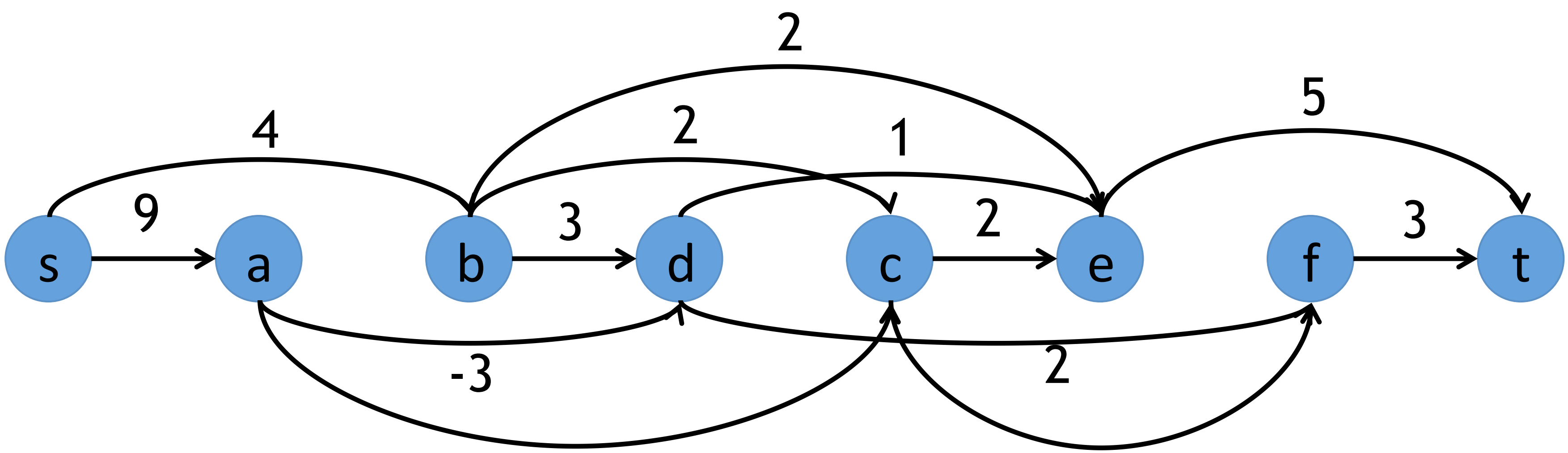


Let's solve a larger subproblem in terms of smaller subproblems.
For example distance to vertex e:

$$SD(e) = \min \begin{cases} SD(d) + w(d, e) \\ SD(c) + w(c, e) \\ SD(b) + w(b, e) \end{cases}$$

Idea: think of the last edge in path $s \rightsquigarrow e$

In General:



$$SD(v) = \min_{(u,v) \in E} \left\{ SD(u) + w(u, v) \right.$$

B/c G is a DAG, we can find shortest paths from left to right!

SSSP DAG DP

```
procedure ssspDAG(DAG G(V, E)):
  topologically sort G
   let SD[s] = 0; SD[v] = +∞
   for v ∈ G in topologically sorted order:
     SD[v] = min_(u,v)∈ E SD[u] + w(u, v)
return D
```

O(m + n)

$$\sum_{u \in V} \deg(u) = m$$

O(n + m):

We loop over each vertex exactly once.

We look at each edge exactly once

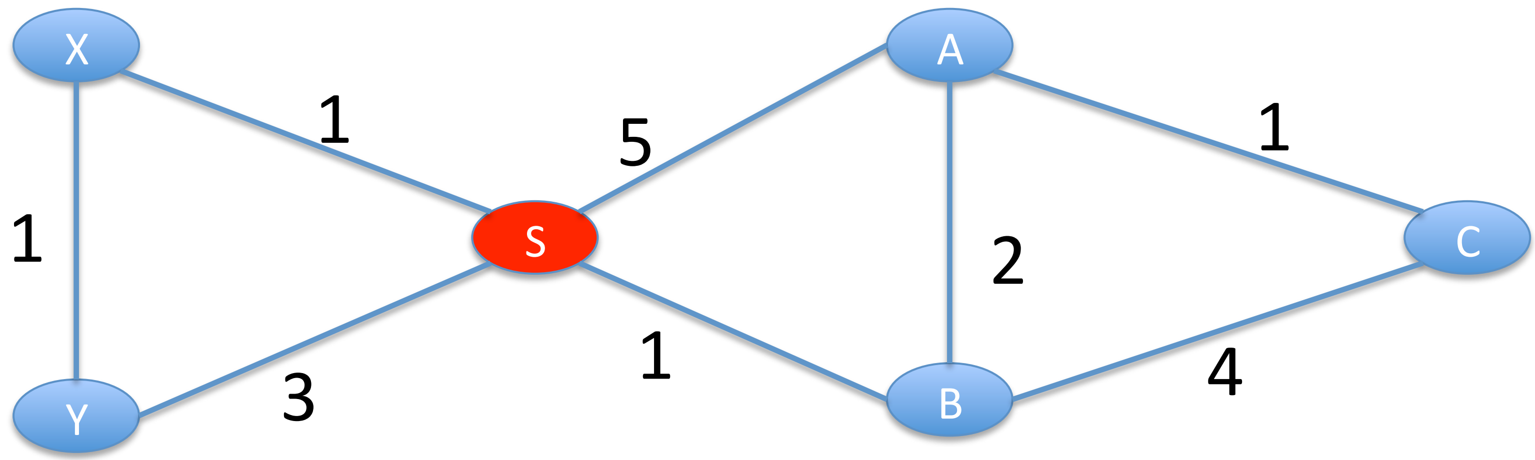
Total Runtime: O(n + m)

Outline For Today

1. SSSP in DAGs: DP Algorithm
2. SSSP without Negative Edges: Dijkstra's Greedy Algorithm
3. SSSP with Negative Edges: Bellman Ford DP Algorithm
4. All pairs Shortest Paths: Floyd Warshall DP Algorithm

SSSP In General Graphs Without Neg. Edges

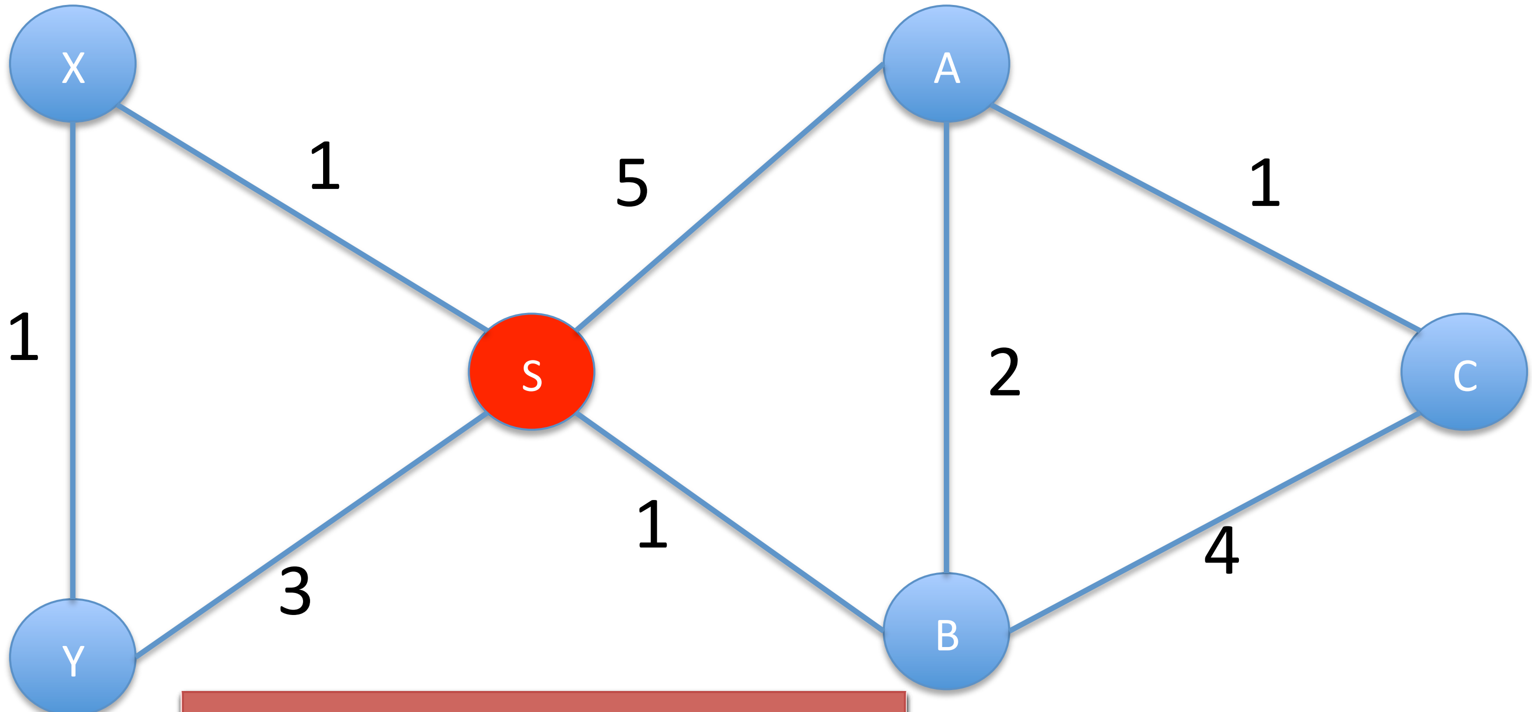
- Input: A directed/undirected graph G(V, E):
  - n nodes (one is the source), m edges (u,v) and costs $c_{u,v}$



- Output: For each node v in the graph, shortest s-v path.
- Assumption 1: Graph is connected (all s-v paths exist)
- Assumption 2: Edge costs are non-negative, i.e., w(u, v) ≥ 0

Shortest Path Example

INPUT

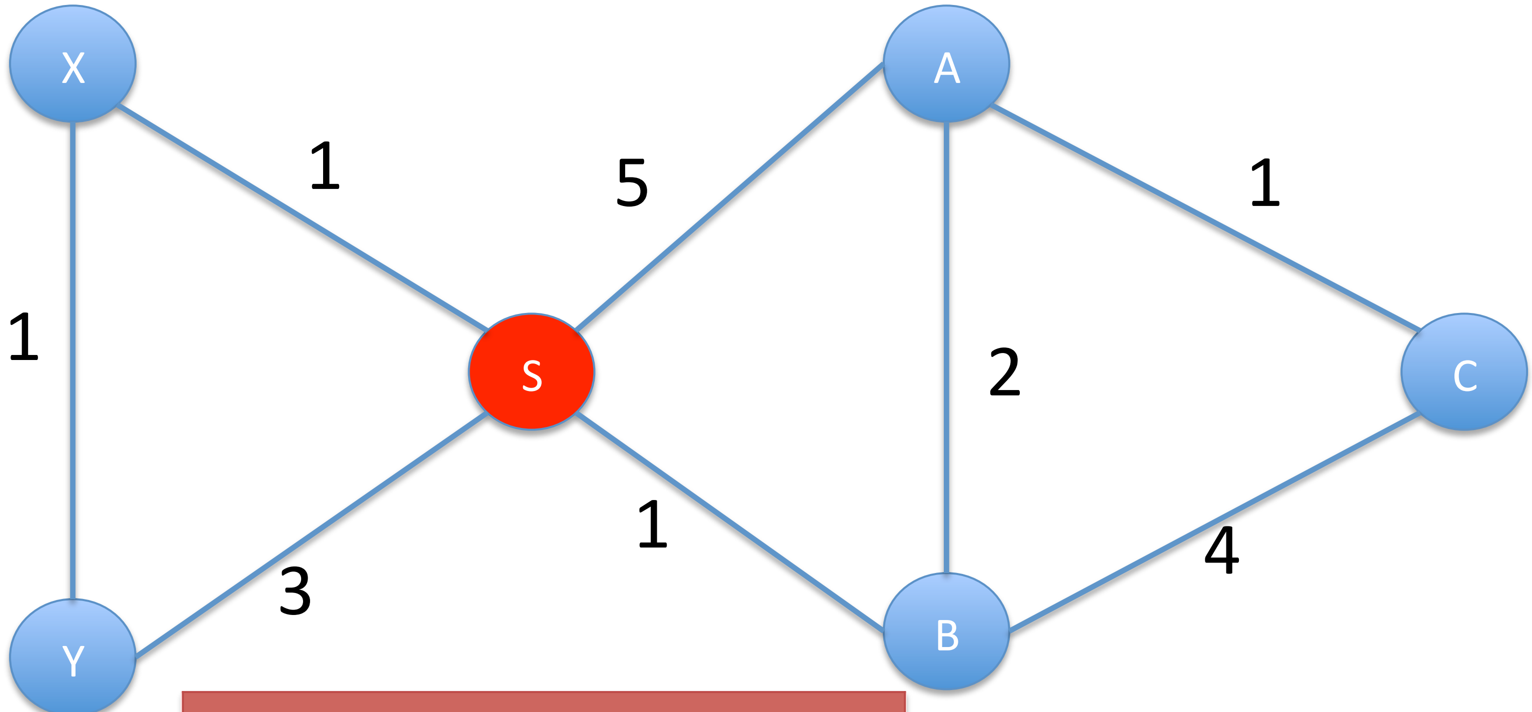


OUTPUT

Dst	Path	Distance
X	S->X	1
Y	S->X->Y	2
A	S->B->A	3
B	S->B	1
C	S->B->A->C	4

Shortest Path Example

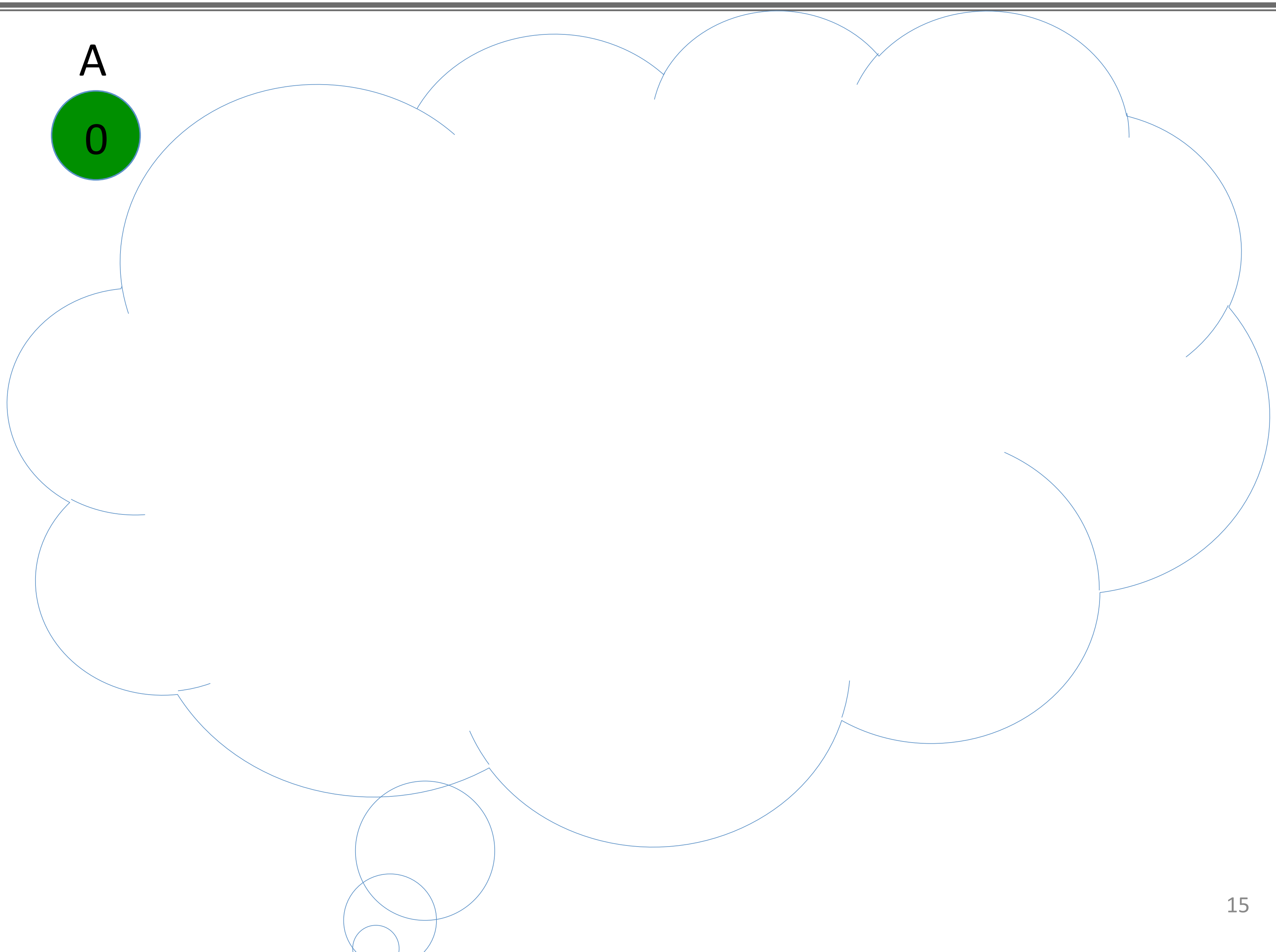
INPUT



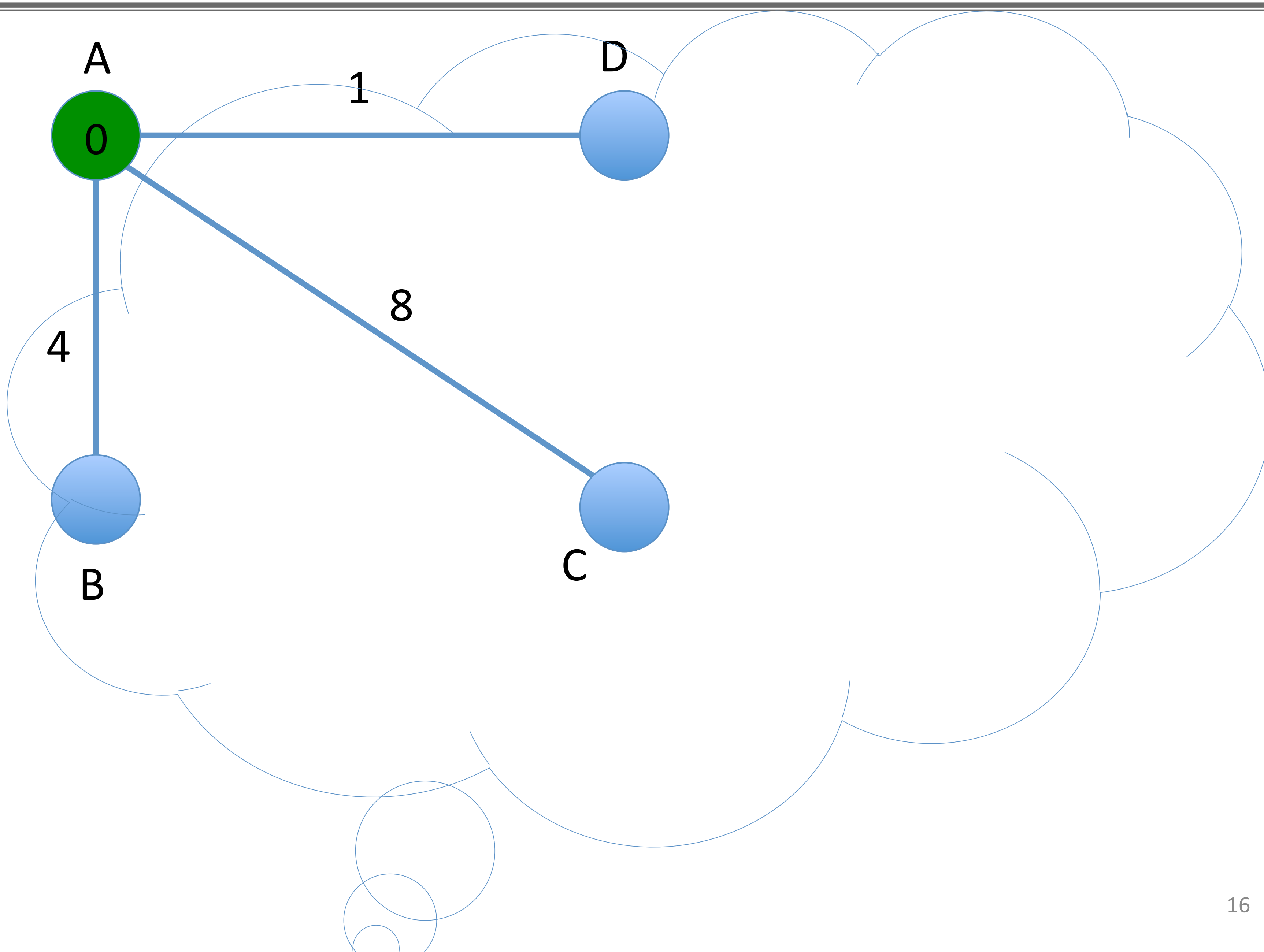
OUTPUT

Dst	Path	Distance
X	S->X	1
Y	S->X->Y	2
A	S->B->A	3
B	S->B	1
C	S->B->A->C	4

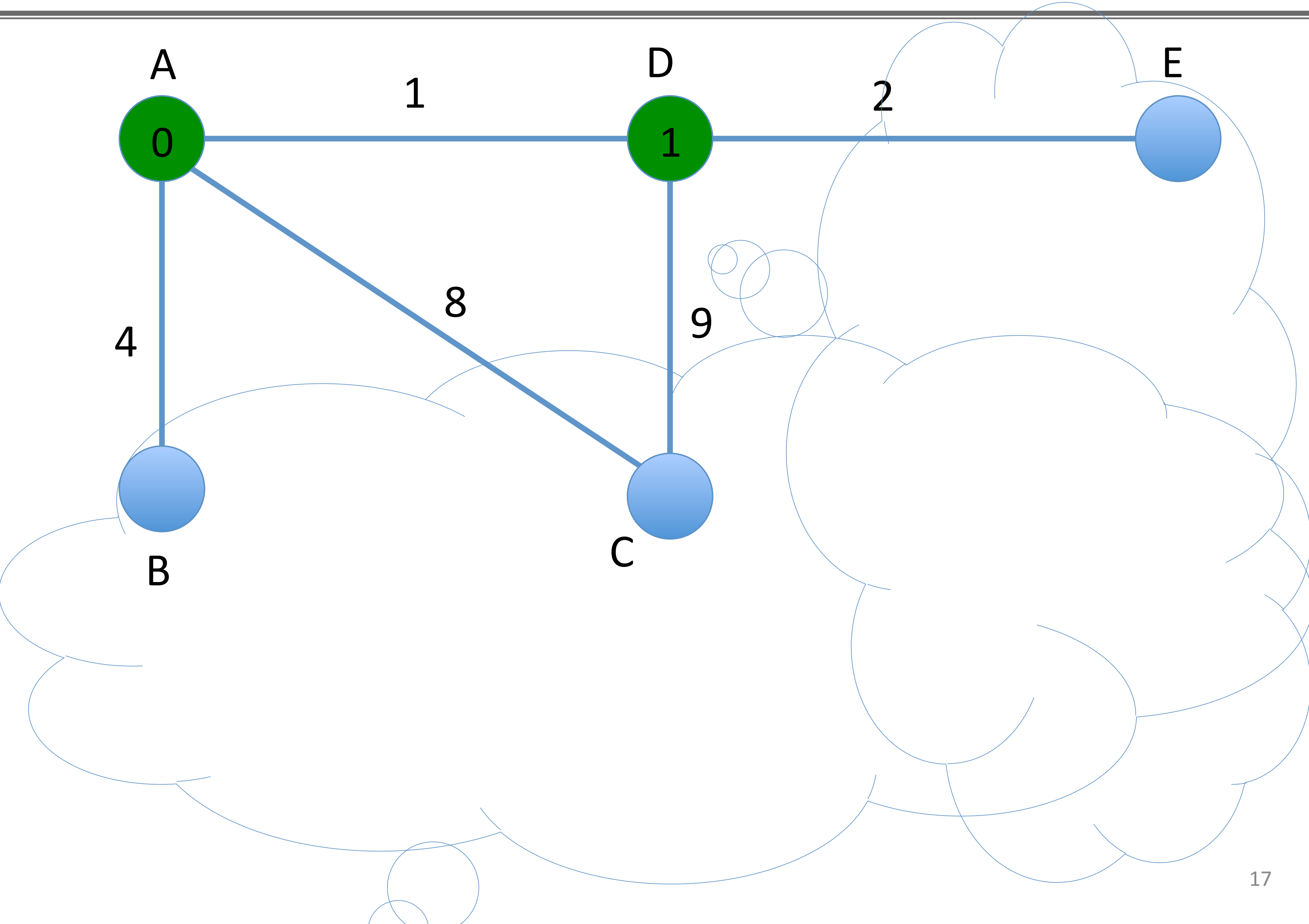
Dijkstra's Algorithm



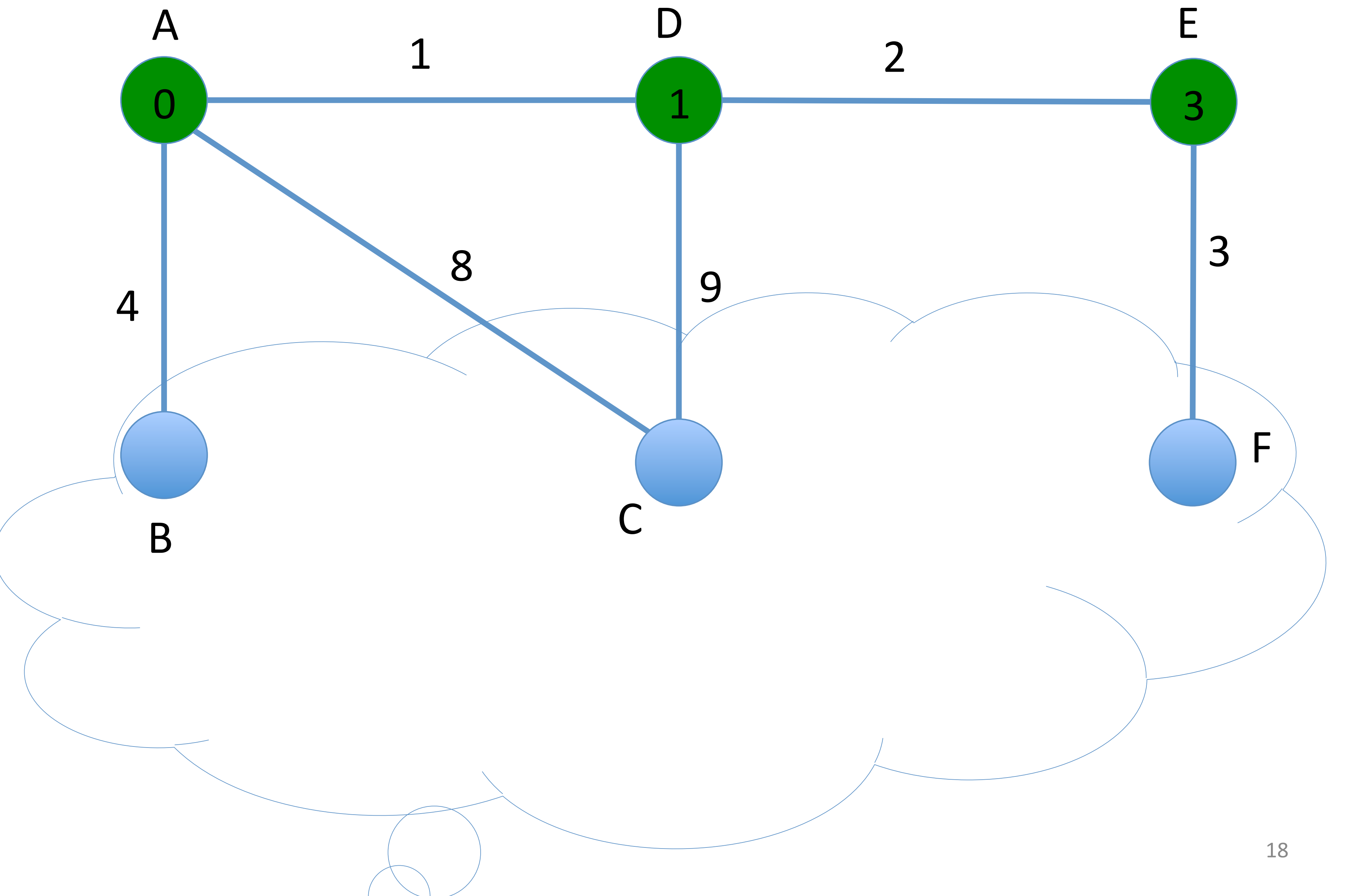
Dijkstra’s Algorithm



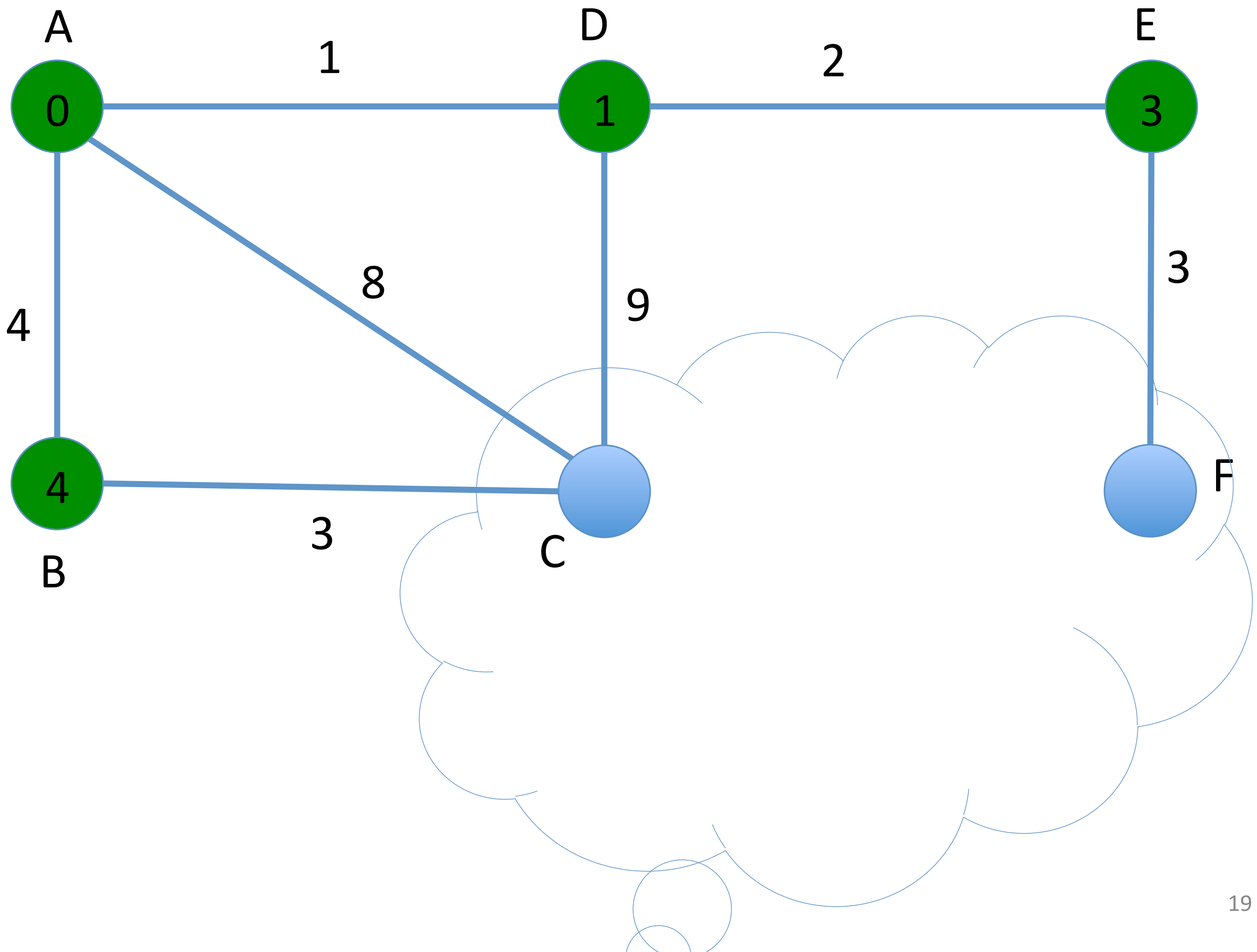
Dijkstra's Algorithm



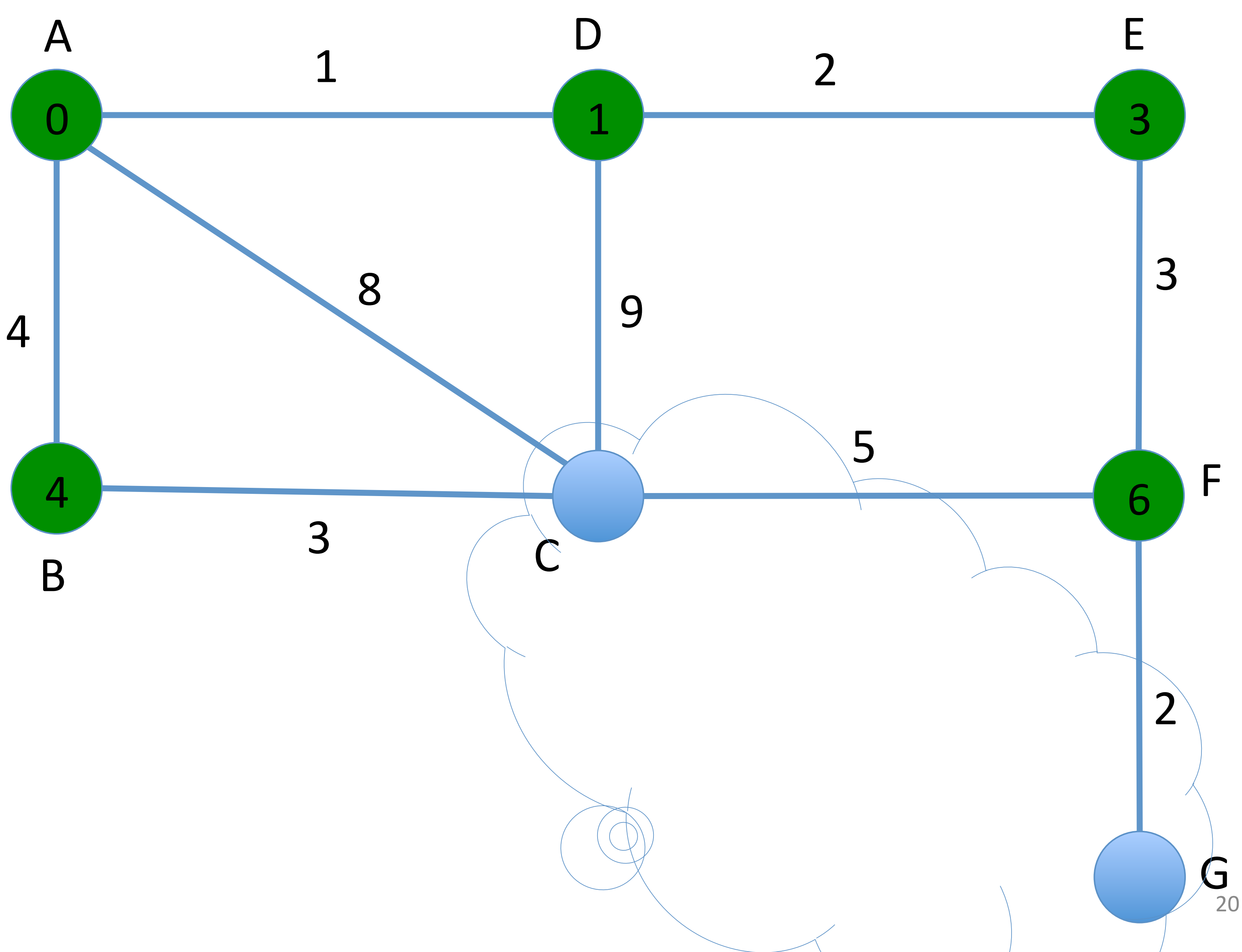
Dijkstra's Algorithm



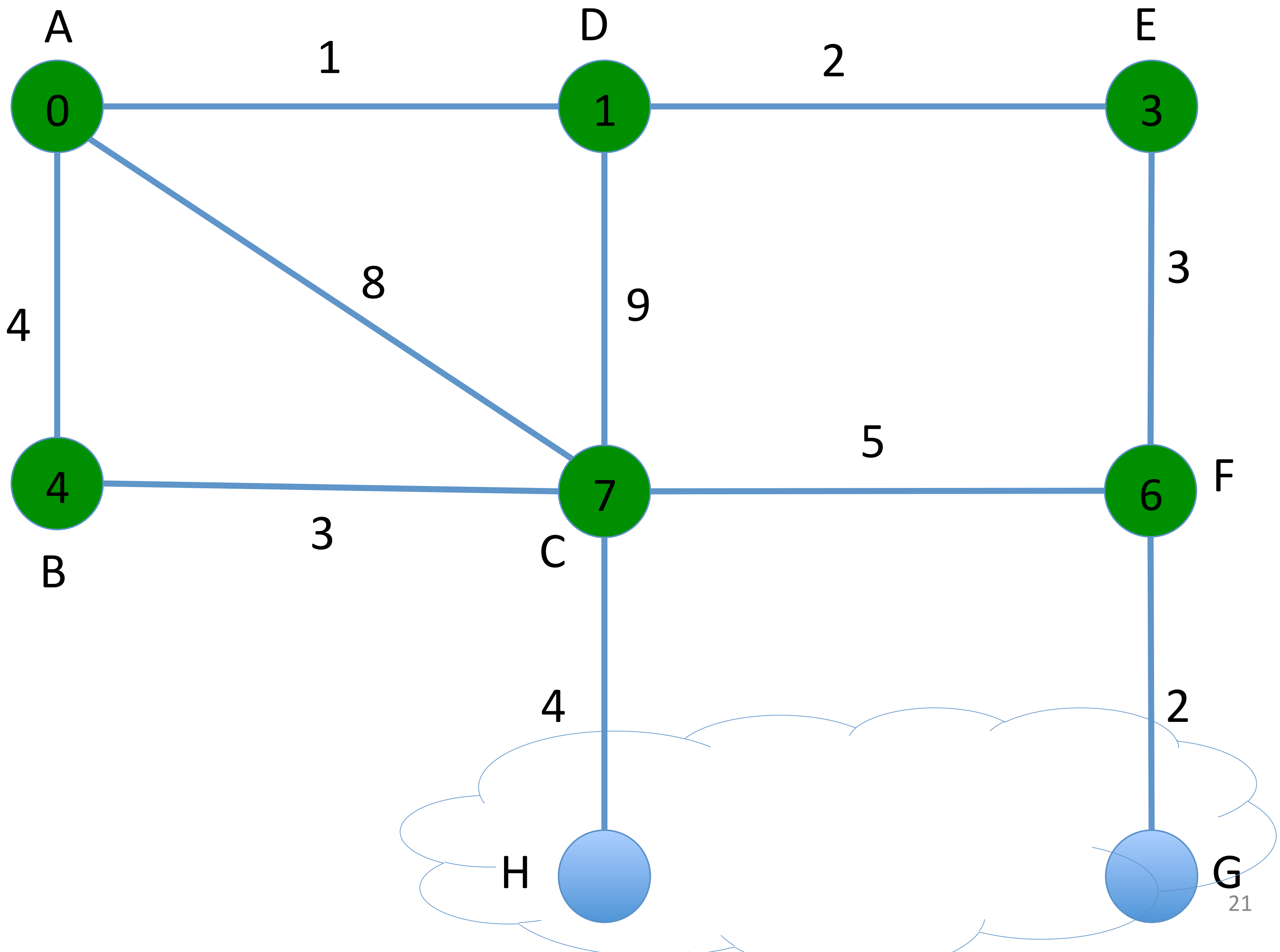
Dijkstra's Algorithm



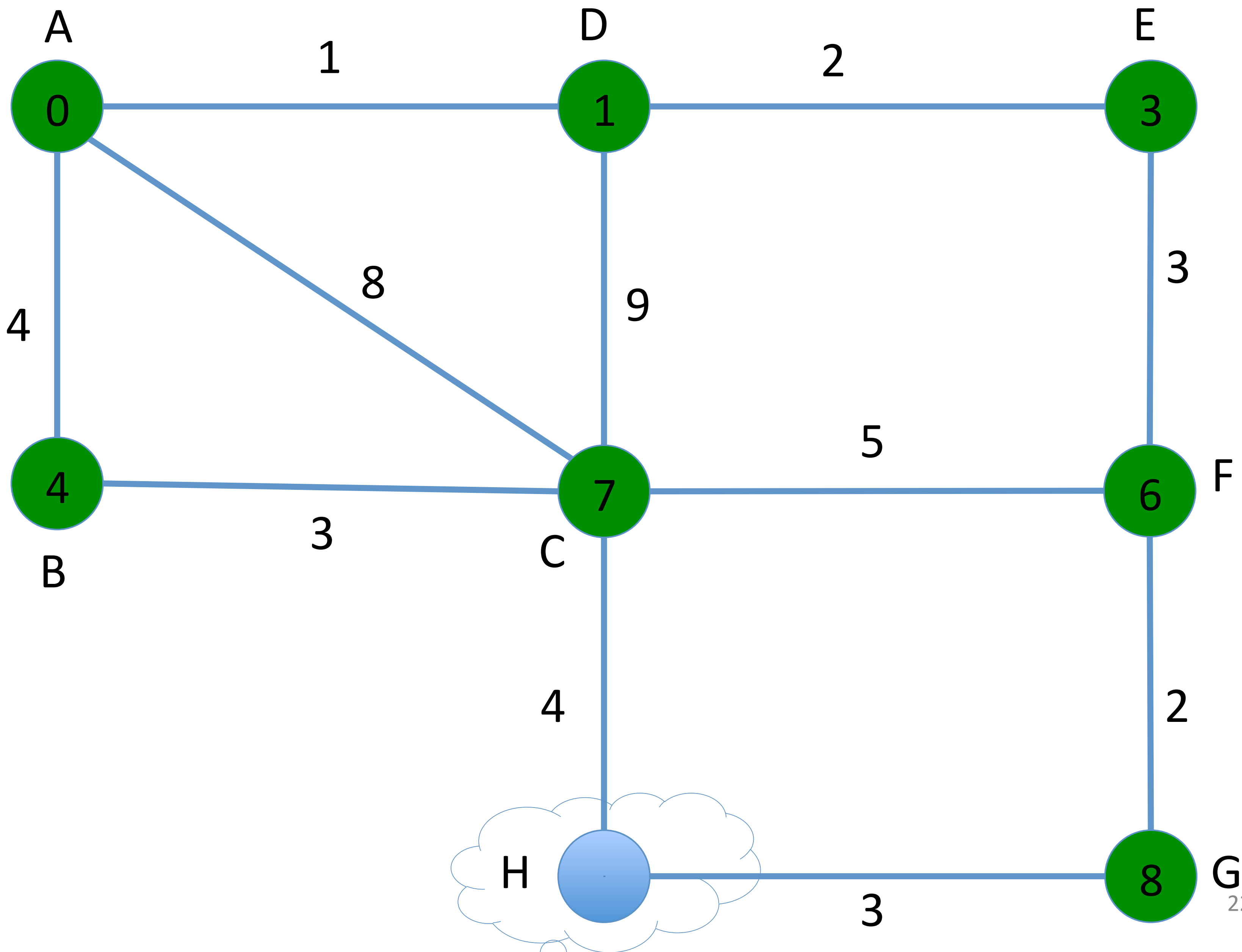
Dijkstra’s Algorithm



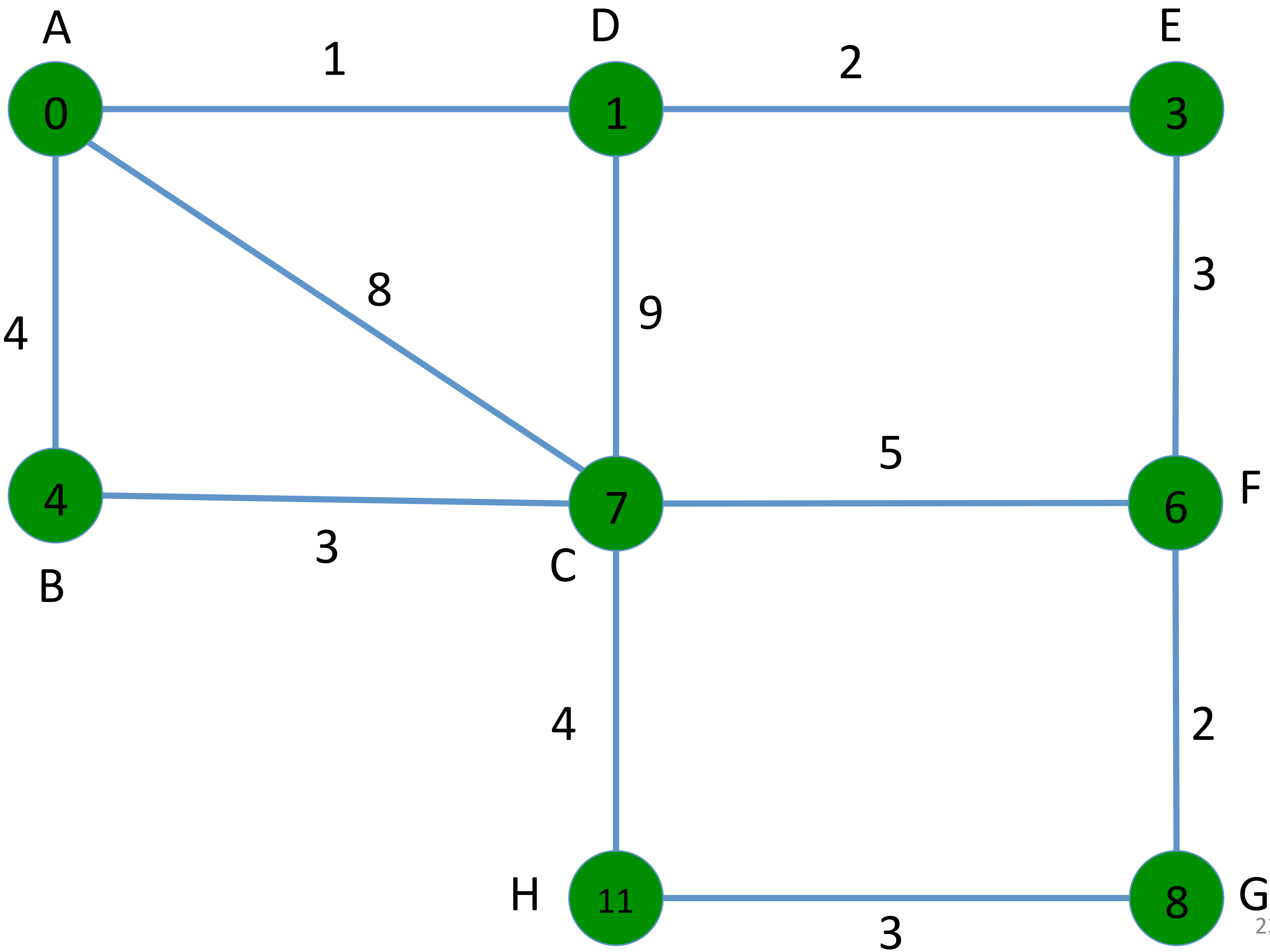
Dijkstra’s Algorithm



Dijkstra's Algorithm



Dijkstra's Algorithm



Dijkstra's Algorithm

```
procedure dijkstra(G(V,E),s, weights w(u, v)) :
   L = {s}; R=V-{s}
   shortestDistL is an array initialized to null
   parent is an array initialized to null
   distSoFarR = priority queue of size n
   distSoFarR[s] = 0; distSoFarR[v] = +∞ for other v
   for i = 1 to n-1:
     let v* = extract-min from distSoFarR
     remove v* from R and add to L
     shortestDistL[v*] = distSoFarR[v*]
     for each (v*, w) s.t. w∈R:
       decrement distSoFarR[w] =
         min{distSoFarR[w], shortestDistL[v*] + w(v*, w)
         if distSoFar[w] decreased: set parent[w] = v*
return shortestDist
```

O(log(n))

O(log(n))

Run time: O(mlog(n))

Dijkstra's Correctness (1)

Induction on the # of iterations

Inductive Claim: at each iteration i:

$\forall v \in L$, shortestDist[v] is correct (same for parent[v])

$\forall v \in R$, shortestDist[v] is shortest (s, v) path contained in L (except last edge)

Base Case: L only contains s and shortestDist[s] is 0 and true.

IH: Assume both claims hold for first k v's in L (i.e., for iteration k)

Let v* be the picked vertex from R in iteration k + 1.

(i.e. distSoFar[v*] was the minimum over all vertices in R)

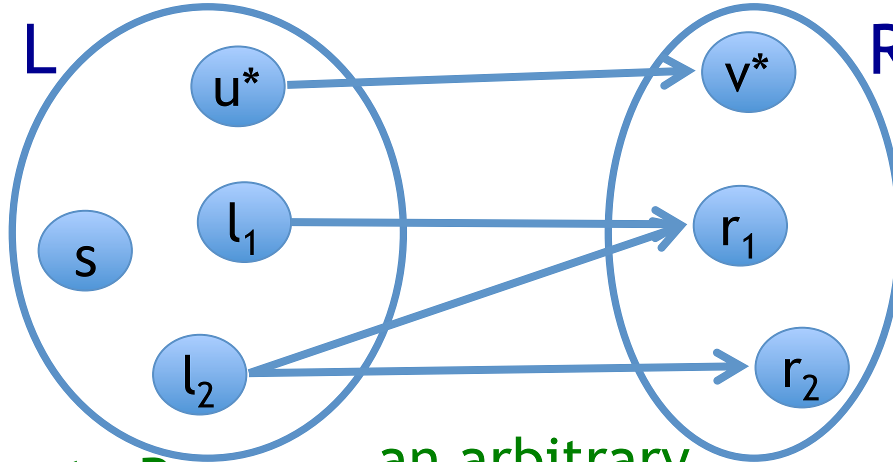
And let (u*, v*) be the edge that minimized v*'s distSoFar.

So Dijkstra's path P is:

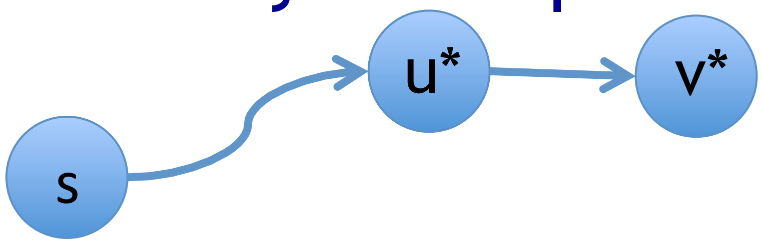
Claim: P is the shortest path from s to v*!

Dijkstra's Correctness (2)

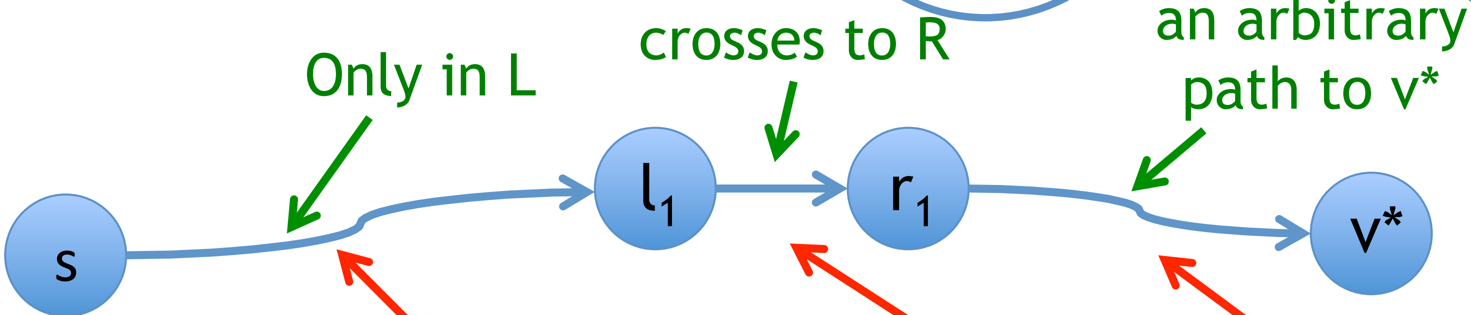
Consider any other path P`



P



P`



$$cost(P`) = \underbrace{(\geq shortestDist[l_1]) + w(l_1, r_1)}_{} + (\geq 0\ cost)$$

$$\geq distSoFar[r_1] \geq distSoFar[v^*] = cost(P)$$

Q.E.D

Outline For Today

1. SSSP in DAGs: DP Algorithm
2. SSSP without Negative Edges: Dijkstra's Greedy Algorithm
3. SSSP with Negative Edges: Bellman Ford DP Algorithm
4. All pairs Shortest Paths: Floyd Warshall DP Algorithm

Pros/Cons of Dijkstra's Algorithm

Pros: O(mlogn) super fast & simple algorithm

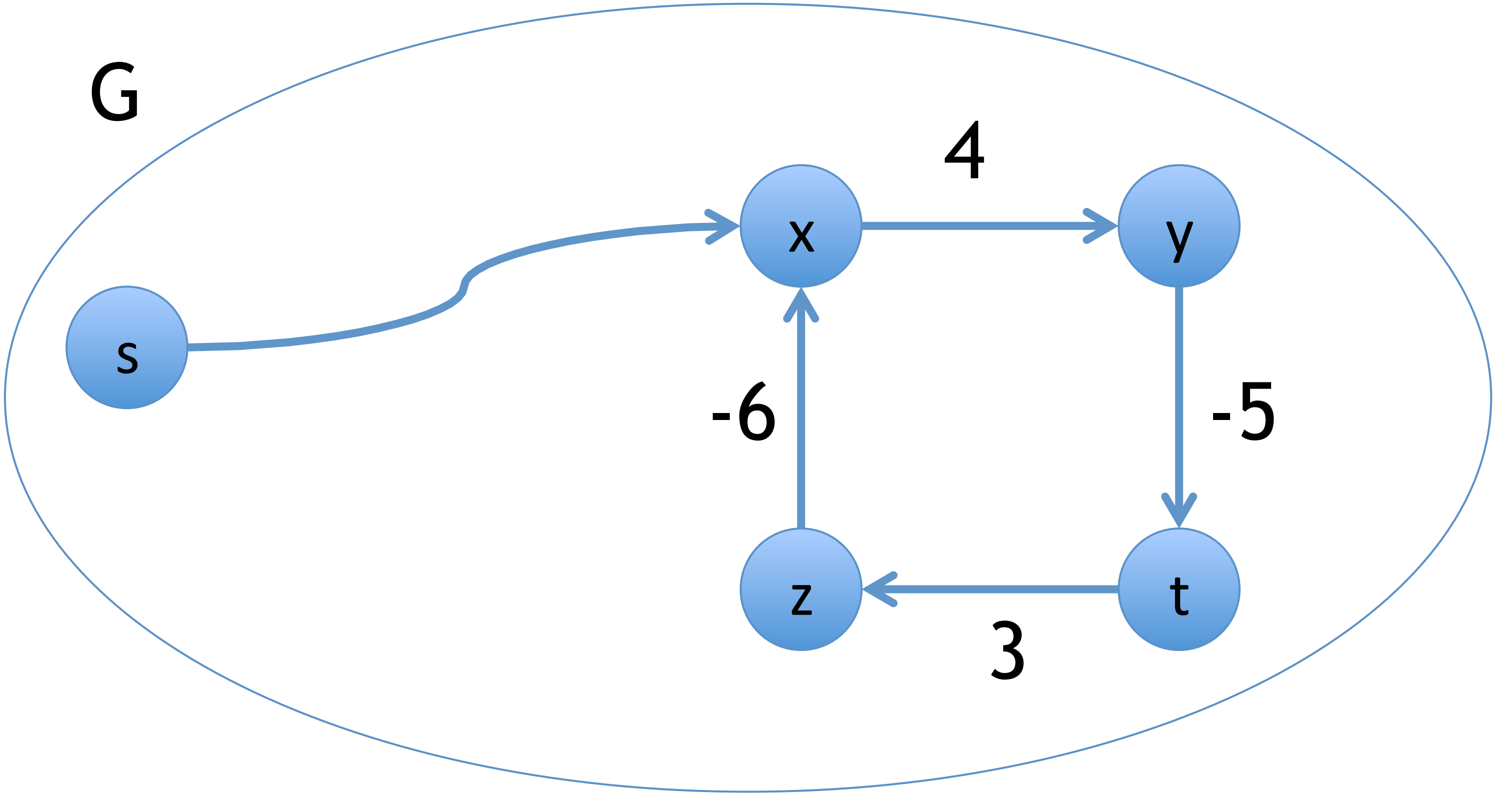
Cons:

1. Works only if $c_e \geq 0$
   - Sometimes need negative weights, e.g. (finance)
2. Not parallelizable:
   - Looks very "serial"

Bellman-Ford addresses both of these drawbacks

Preliminary: Negative Weight Cycles

Question: How to define shortest paths when G has negative weights cycles?

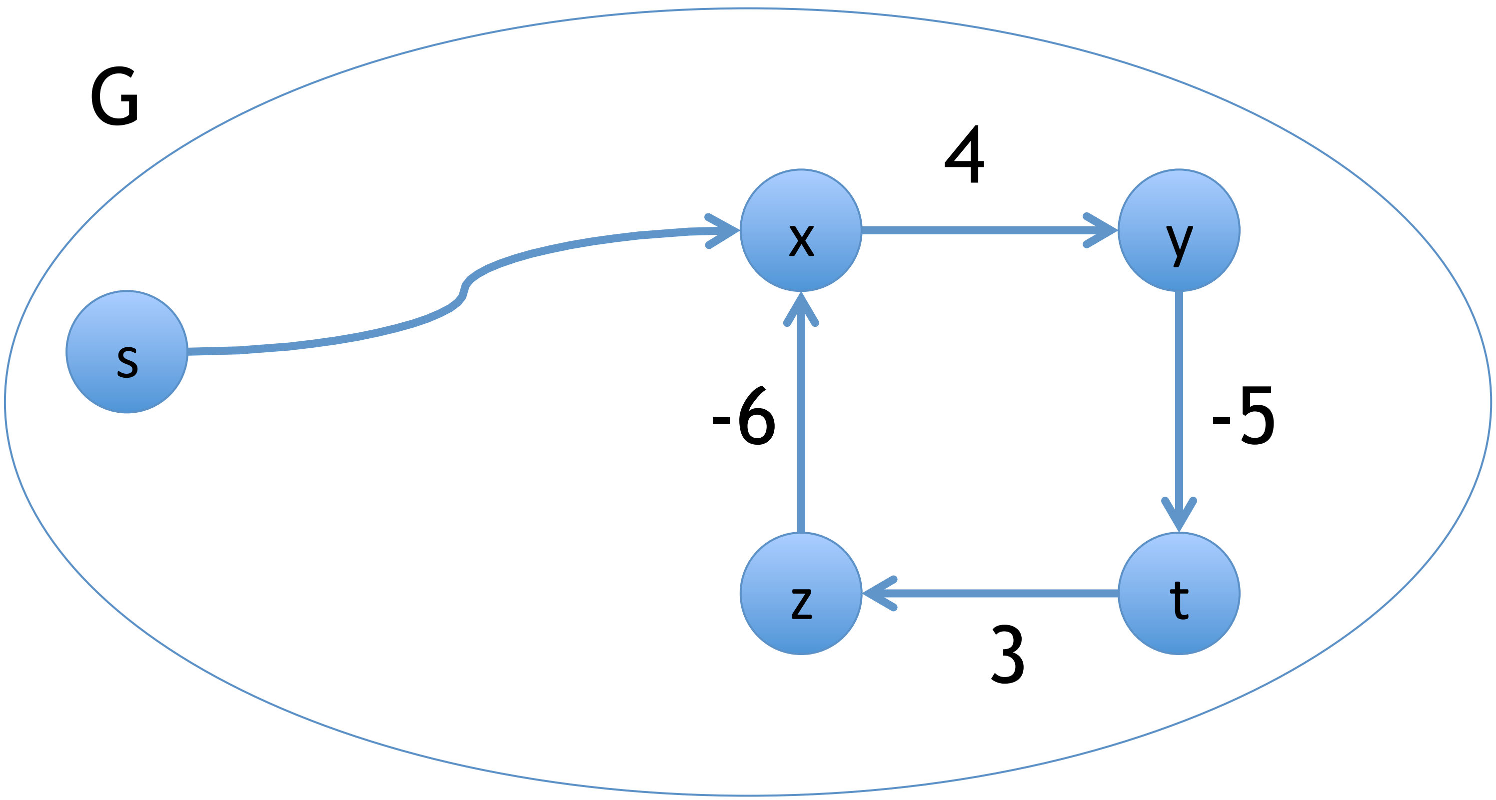


Possible Shortest s⤳v Path Definition 1

Shortest path from s to v with cycles allowed.

Problem: Can loop forever in a negative cycle.

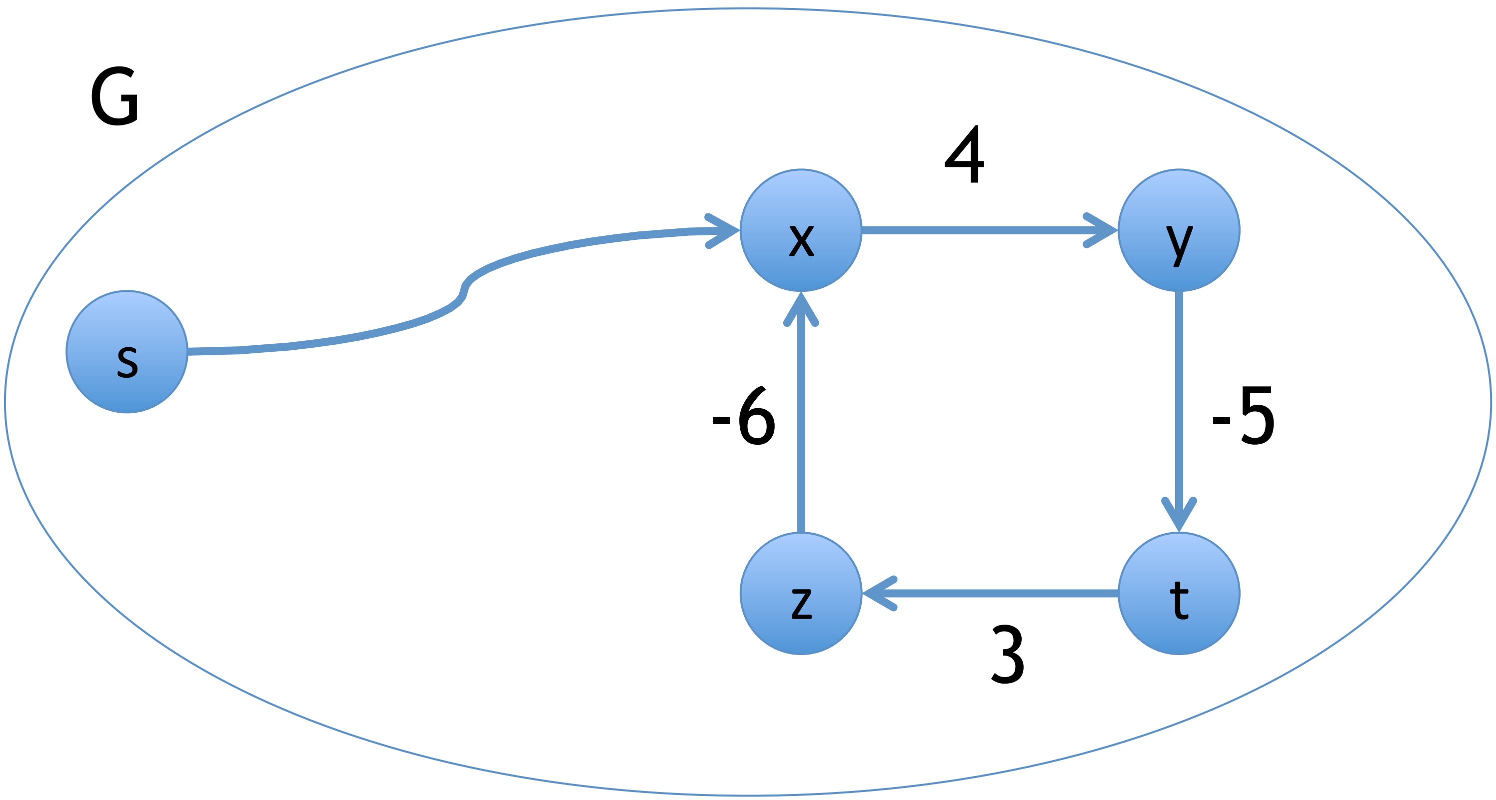
So there is no “shortest path”!



Possible Shortest s⤳v Path Definition 2

Shortest path from s to v, cycles NOT allowed.

Problem: Now well-defined. But NP-hard. Don't expect a "fast" algorithm solving it exactly.



Solution: Assume No Negative-Weight Cycles

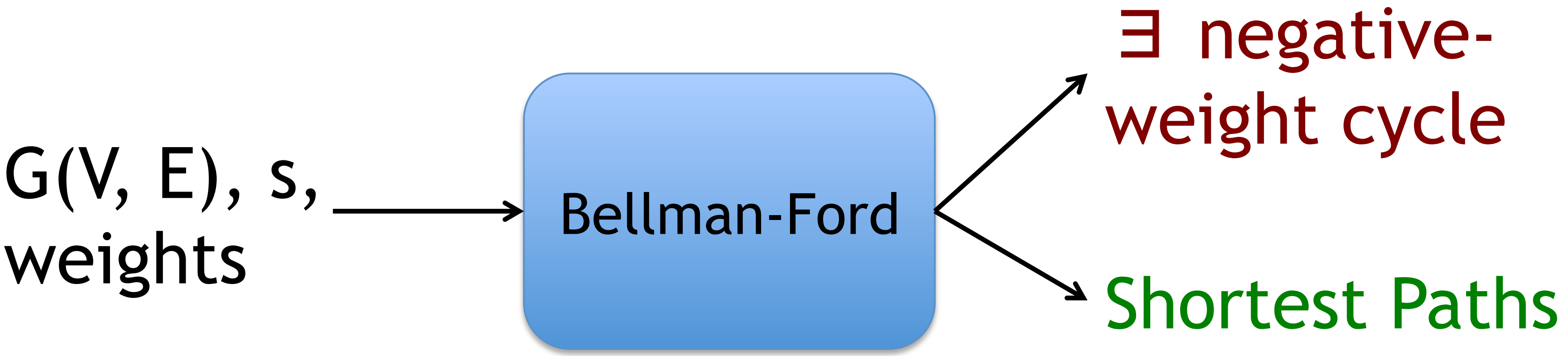
Q: Now, can shortest paths contain cycles?

A: No. Assume P is shortest path from s to v, with a cycle.

P= $s \leadsto x \leadsto x \leadsto v$. Then since $x \leadsto x$ is a cycle, and we assumed no negative weight cycles, we could get P`= $s \leadsto x \leadsto v$, and get a shorter path.

Upshot: Bellman-Ford's Properties

Note: Bellman-Ford will be able to detect if there is a negative weight cycle!



Both outputs computed in reasonable amount of time.

Challenge of A DP Approach

Need to identify some sub-problems.

- Linear IS:
  - Line graph was naturally ordered from left to right.
  - Subproblems could be defined as prefix graphs.
- Sequence Alignment:
  - X, Y strands were naturally ordered strings.
  - Subproblems could be defined as prefix strings.

Shortest Paths' Input G Has No Natural Ordering

High-level Idea Of Subproblems

But the Output Is Paths & Paths Are Sequential!

Trick: Impose an Ordering Not On G but on Paths.

Larger Paths Will Be Derived By Appending New Edges To The Ends Of Smaller (Shorter) Paths.

Subproblems

Input: $G(V, E)$ no negative cycles, s, c_e arbitrary weights.

Output: $\forall$v, global shortest paths from s to v.

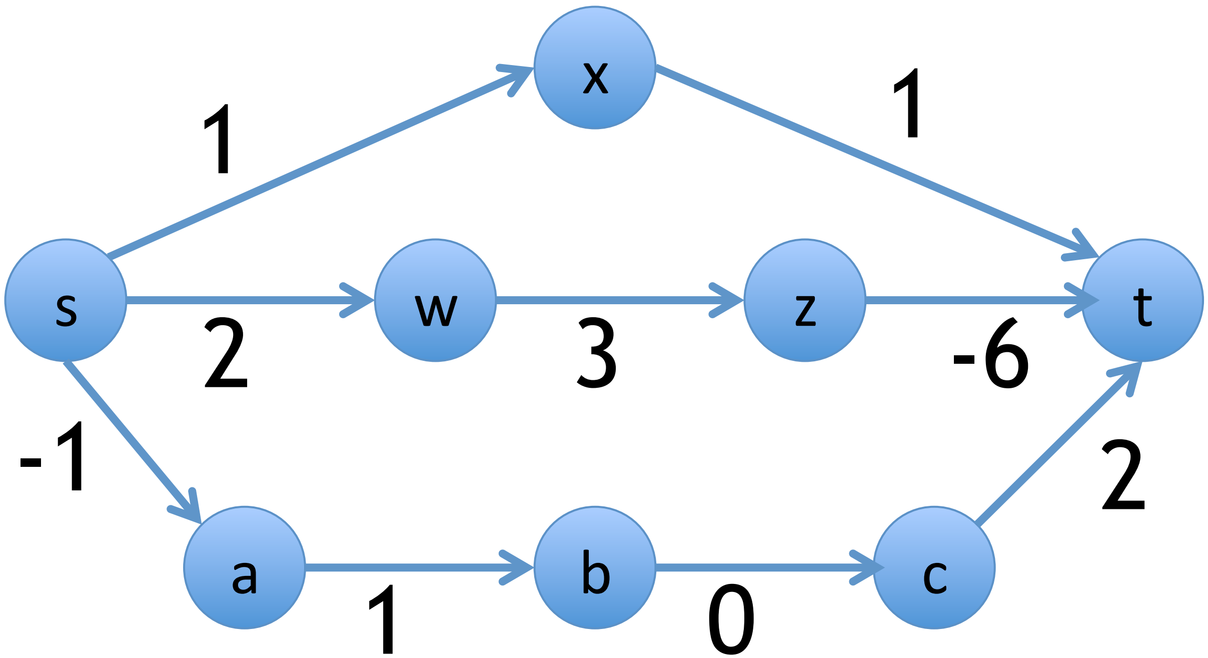
Q: Max possible # hops (or # edges) on the shortest paths?

A: n-1 (**since there are no negative cycles**)

$P_{(v,\ i)}$ = *Shortest path from s to v with at most i edges (& no cycles).*

Example

Let $P_{(v,\ i)}$ be the shortest s-v path with ≤ i edges.

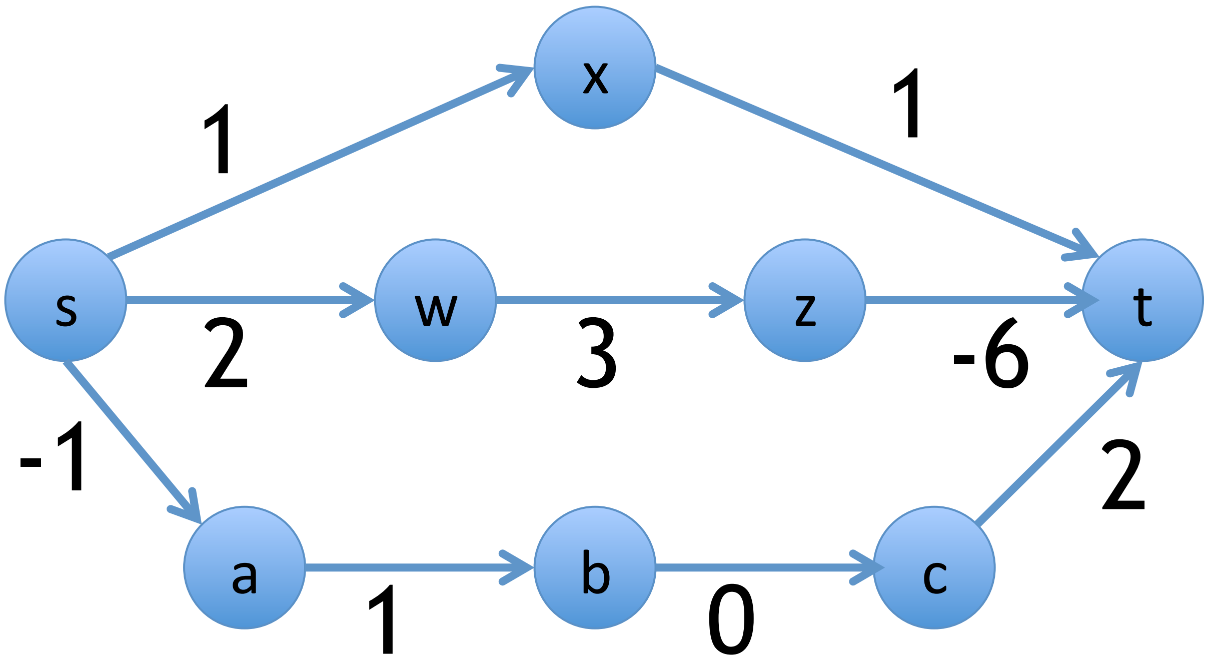


Q: $P_{(t,1)}$?

A: Does not exist (assume such paths have ∞ weights.)

Example

Let $P_{(v,\ i)}$ be the shortest s-v path with $\leq$ i edges.

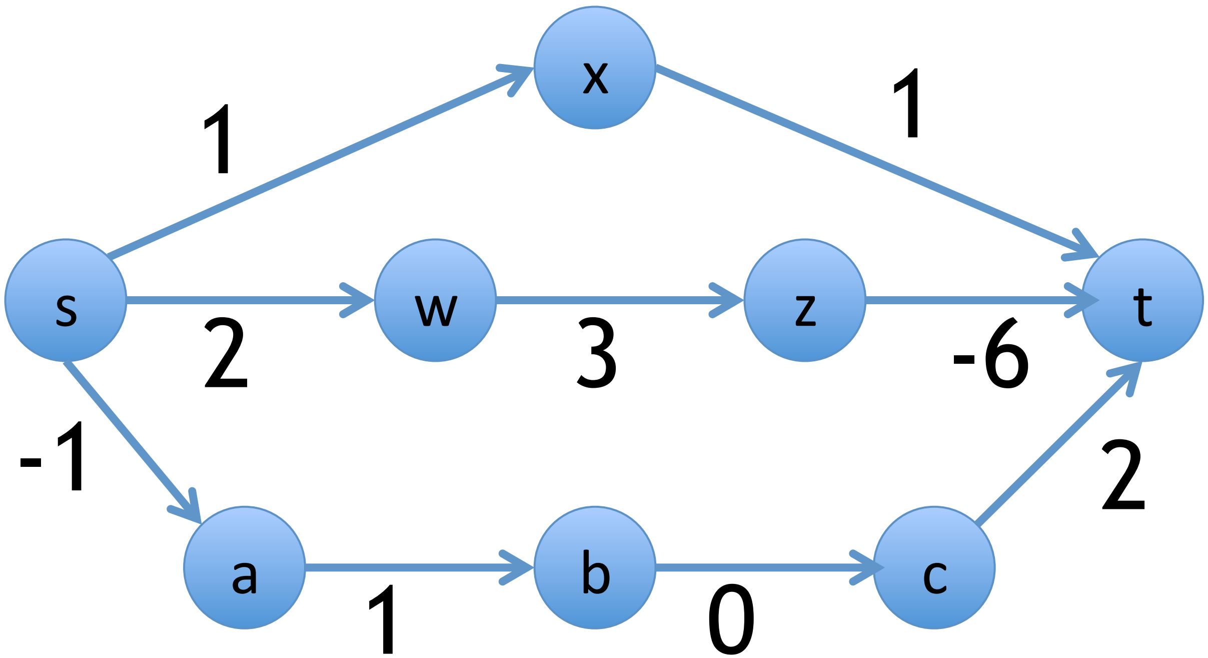


Q: $P_{(t,2)}$?

A: s->x->t with weight 2.

Example

Let $P_{(v,\ i)}$ be the shortest s-v path with ≤ i edges.

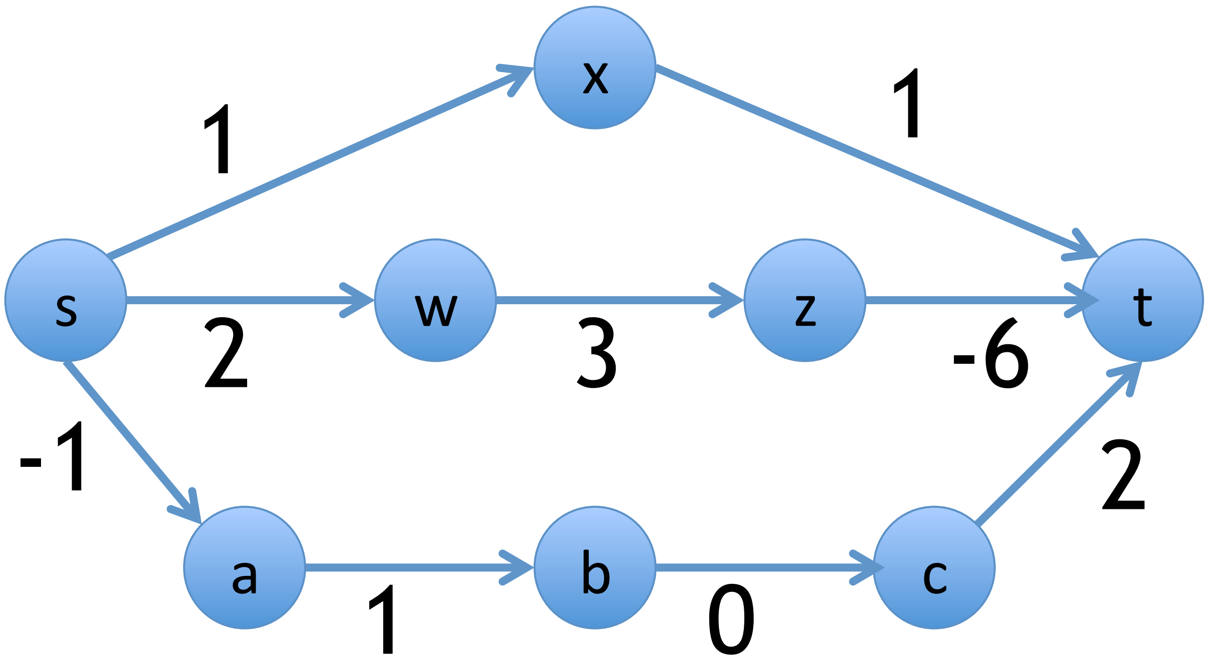


Q: $P_{(t,3)}$?

A: s->w->z->t with weight -1.

Example

Let $P_{(v,\, i)}$ be the shortest s-v path with ≤ i edges.



Q: $P_{(t,4)}$?

A: s->w->z->t with weight -1.

Solving $P_{(v,i)}$ In Terms of "Smaller" Subproblems

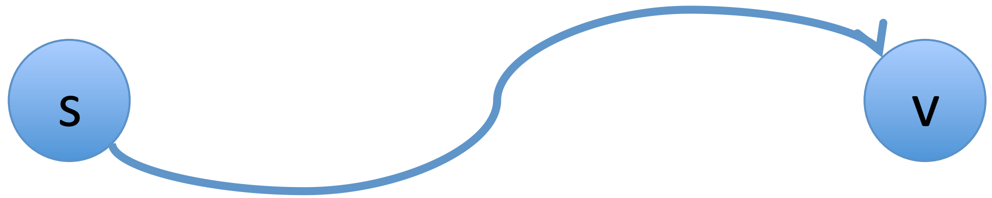
Let $P = P_{(v,\ i)}$ be the shortest s-v path with ≤ i edges

Note: For some v, an s-v path with ≤ i edges may not exist. Assume v has such a path.

A Claim that does not require a proof:

$|P| \leq i\text{-}1$ OR $|P| = i$

Case 1: $|P=P_{(v,\ i)}| \leq i\text{-}1$

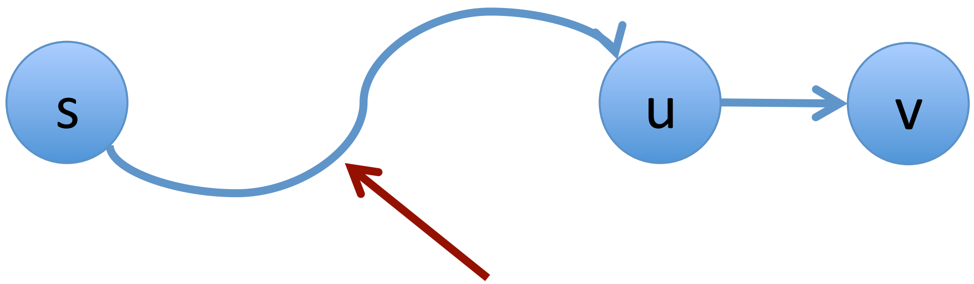


Q: What can we assert about $P_{(v,\ i-1)}$?

A: $P_{(v,\ i-1)} = P_{(v,\ i)}$ (by contradiction)

($P_{(v,\ i)}$ is also the shortest s⤳v path with at most i-1 edges)

Case 2: $|P=P_{(v,\ i)}|\ = i$



P` => |s⤳u| = i-1

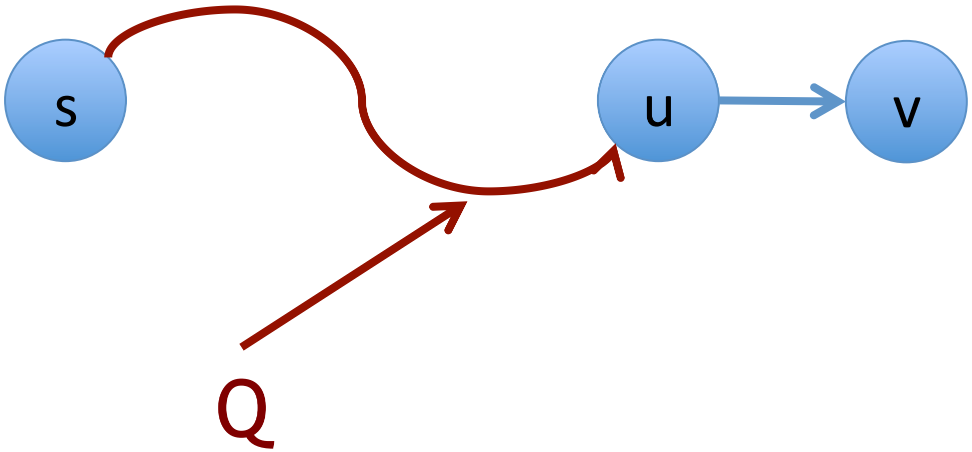
Q: What can we assert about P`?

Claim: P` = $P_{(u,\ i\text{-}1)}$

(P` is shortest s⤳u path in with ≤ i-1 edges)

Proof that $P` = P_{(u, i-1)}$

Assume $\exists$ a better $s \rightsquigarrow u$ path Q with $\leq$ i-1 edges



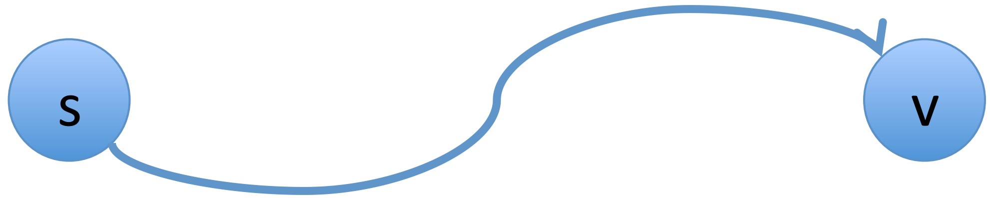
Q had $\leq$ i-1 edges, then $Q \cup (u,v)$ has $\leq$ i edges.

$cost(Q) < cost(P`)$, $cost(Q \cup (u,v)) < cost(P)$.

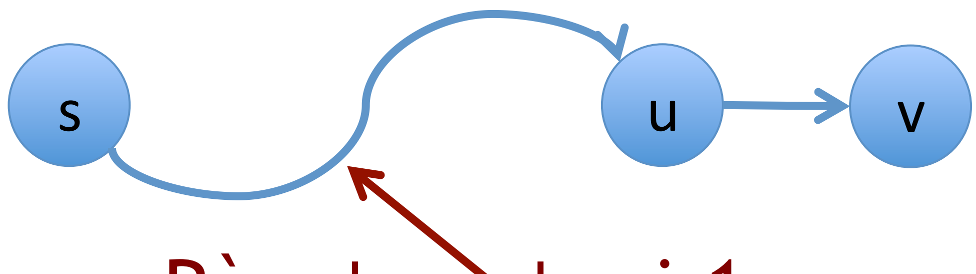
Q.E.D

Summary of the 2 Cases

Case 1: $|P_{(v,\ i)}| \leq i\text{-}1 \Rightarrow P_{(v,\ i\text{-}1)} = P_{(v,\ i)}$



Case 2: $|P_{(v,\ i)}| = i \Rightarrow P` = P_{(u,\ i\text{-}1)}$



$P` \Rightarrow |s \rightsquigarrow u| = i\text{-}1$

Our Subproblems Agains

$\forall$ v, and for i={1, …, n}

$P_{(v,\ i)}$: shortest s $\rightsquigarrow$ v path with $\leq$ i edges (or null)

$L_{(v,\ i)}$: $w(P_{(v,\ i)})$ (and $+\infty$ for null paths)

$$L_{(v,\ i)} = \min \begin{cases} L_{(v,\ i-1)} \\ \min_{u:\ \exists (u,v)\in E} : L_{(u,\ i-1)} + c_{(u,v)} \end{cases}$$

Bellman-Ford Algorithm

$L_{(v,\ i)}$: $w(P_{(v,\ i)})$

Let A be an nxn 2D array.

A[i][v] = shortest path to vertex v with ≤ i edges.

```
procedure Bellman-Ford(G(V,E), weights C):
  Base Cases: A[0][s] =
```

Bellman-Ford Algorithm

$L_{(v,\ i)}$: $w(P_{(v,\ i)})$

Let A be an nxn 2D array.

A[i][v] = shortest path to vertex v with ≤ i edges.

```
procedure Bellman-Ford(G(V,E), weights C):
  Base Cases: A[0][s] = 0
              A[0][j] =
```

Bellman-Ford Algorithm

$L_{(v,\ i)}$: $w(P_{(v,\ i)})$

Let A be an nxn 2D array.

A[i][v] = shortest path to vertex v with ≤ i edges.

```
procedure Bellman-Ford(G(V,E), weights C):
  Base Cases: A[0][s] = 0
              A[0][j] = +∞ where j ≠ s
  for i = 1, …, n-1:
    for v ∈ V:
      A[i][v] = min {A[i-1][v]
                     min_(u,v)∈E A[i-1][u]+c_(u,v)
```

Correctness of BF

By induction on i and correctness of the recurrence for $L_{(v, i)}$ (exercise)

Runtime of BF

entries in A is n^2.

Q: How much time for computing each A[i][v]?

A: in-deg(v)

For each i, total work for all A[i][v] entries is: $\sum_{v \in V} in-\deg(v)$

Total Runtime: O(nm)

```
…
 for i = 1, …, n-1:
    for v ∈ V:
       A[i][v] = min {A[i-1][v]
                       min_(u,v)∈E A[i-1][u]+c_(u,v)
```

Runtime Optimization: Stopping Early

Suppose for some i ≤ n:

$A[i][v] = A[i-1][v] \ \forall\ v.$

Q: What does this mean?

A: Values will not change in any later iteration

=> We can stop!

Values only depend on the previous iteration!

```
…
 for i = 1, …, n-1:
   for v ∈ V:
     A[i][v] = min {A[i-1][v]
                    min_{(u,v)∈E} A[i-1][u]+c_{(u,v)}
```

Negative Cycle Checking

Consider any graph G(V, E) with arbitrary edge weights.

=>There may be negative cycles.

Claim: If BF stabilizes at some iteration $i > 0$, then

G has no negative cycles.

(i.e., negative cycles implies BF never stabilizes!)

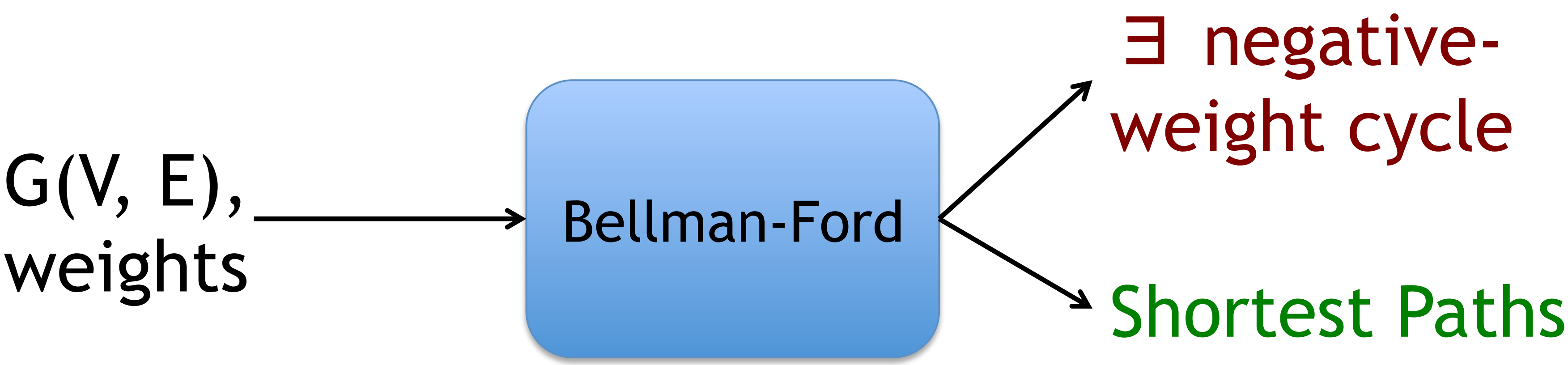
How To Check For Cycles If Claim is True

Run BF just one extra iteration!

Check if A[n][v] = A[n-1][v] for all v.

If so, no negative cycles, o.w. there is a negative cycle.

Running n iterations is the general form of BF:



Proof of Claim:
BF Stabilizes => G has no negative cycles

Assume BF has stabilized in iteration i.

Notation: d(v) = A[i][v] = A[i-1][v] (by above assumption)

$$\boxed{A[i][v]} = \min \{\boxed{A[i-1][v]}$$
$$\min_{(u,v)\in E} \boxed{A[i-1][u]} + c_{(u,v)}$$

$$d(v) = \min \{d(v)$$
$$\min_{(u,v)\in E} d(u) + c_{(u,v)}$$

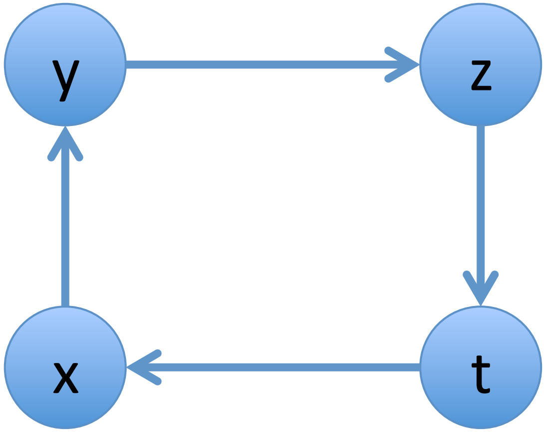
$$d(v) \leq d(u) + c_{(u,v)}$$

Let's argue that every cycle C has non-negative weight...

Proof of Claim (continued)

$d(v) \leq d(u) + c_{(u,v)}$

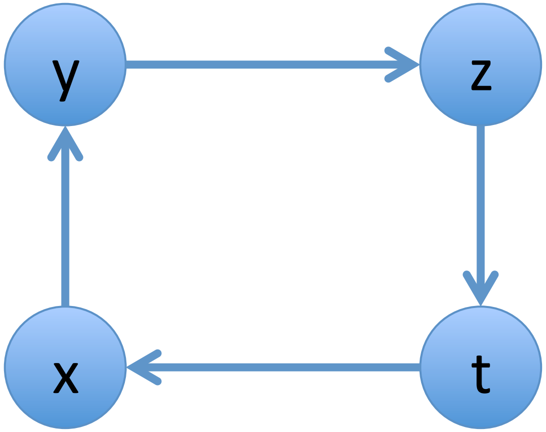
Fix a cycle C:



Proof of Claim (continued)

$d(v) \le d(u) + c_{(u,v)} \Rightarrow d(v) - d(u) \le c_{(u,v)}$

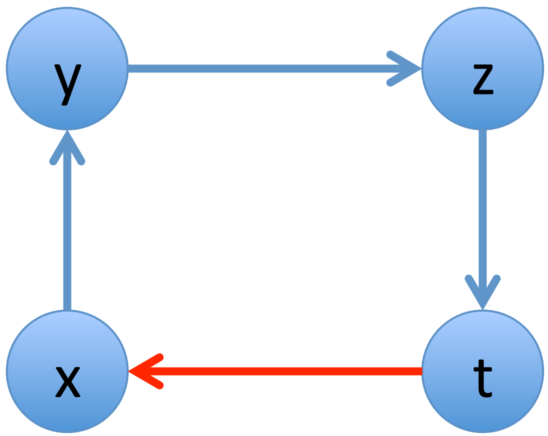
Fix a cycle C:



Proof of Claim (continued)

$d(v) \leq d(u) + c_{(u,v)} \Rightarrow d(v) - d(u) \leq c_{(u,v)}$

Fix a cycle C:

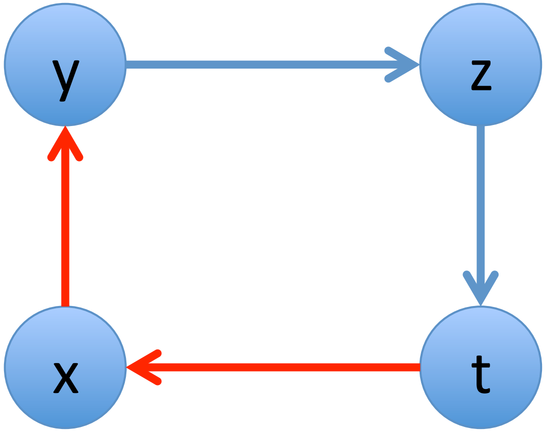


$d(x) - d(t) \leq c_{(t,x)}$

Proof of Claim (continued)

$d(v) \leq d(u) + c_{(u,v)} \Rightarrow d(v) - d(u) \leq c_{(u,v)}$

Fix a cycle C:



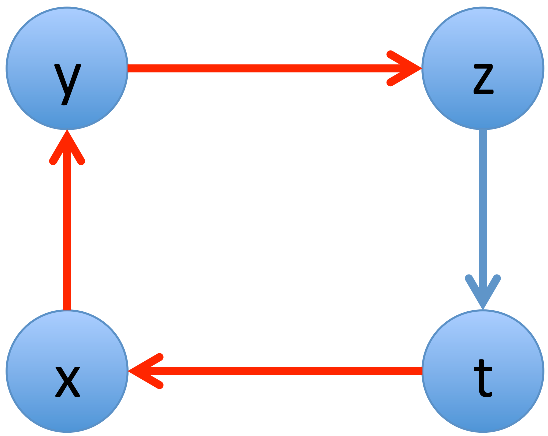
$d(x) - d(t) \leq c_{(t,x)}$

$d(y) - d(x) \leq c_{(x,y)}$

Proof of Claim (continued)

$d(v) \leq d(u) + c_{(u,v)}$ => $d(v) - d(u) \leq c_{(u,v)}$

Fix a cycle C:



$d(x) - d(t) \leq c_{(t,x)}$

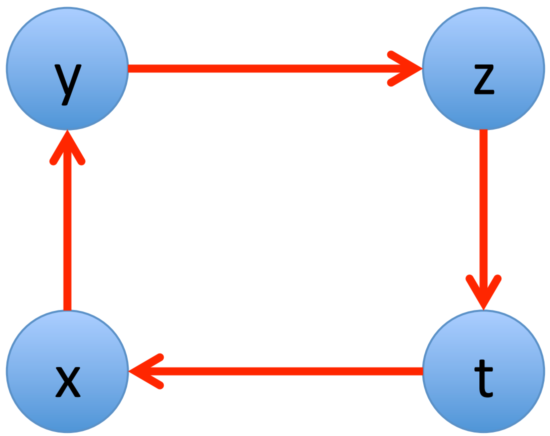
$d(y) - d(x) \leq c_{(x,y)}$

$d(z) - d(y) \leq c_{(y,z)}$

Proof of Claim (continued)

$d(v) \leq d(u) + c_{(u,v)}$ => $d(v) - d(u) \leq c_{(u,v)}$

Fix a cycle C:



$$
\begin{array}{r}
d(x) - d(t) \leq c_{(t,x)} \\
d(y) - d(x) \leq c_{(x,y)} \\
d(z) - d(y) \leq c_{(y,z)} \\
d(t) - d(z) \leq c_{(z,t)} \\
\hline
0 \leq w(C)
\end{array}
$$

Same algebra and result for any cycle (exercise).

Q.E.D.

Space Optimization (1)

Only need A[i-1][v]'s to compute A[i][v]s.

$\Rightarrow$ Only need O(n) space; i.e., O(1) per vertex.

Q: By throwing things out, what do we lose in general?

A: Reconstruction of the actual paths.

But with only O(n) more space, we can actually reconstruct the paths!

```
…
 for i = 1, …, n-1:
    for v ∈ V:
       A[i][v] = min {A[i-1][v]
                         min_(u,v)∈E A[i-1][u]+c_(u,v)
```

Space Optimization (1)

Fix: Each v stores a predecessor pointer (initially null)

Whenever A[i][v] is updated to A[i-1][u]+$c_{(u,v)}$, we set the Pred[v] to u.

Claim: At termination, tracing pointers back from v yields the shortest s-v path.

(Details in the book, by induction on i)

…

```
for i = 1, …, n-1:
   for v ∈ V:
      A[i][v] = min {A[i-1][v]
                     min(u,v)∈E A[i-1][u]+c(u,v)
```

Summary of BF

Runtime: O(nm), not as fast as Dijkstra's O(mlogn).

But works with negative weight edges.

And is distributable/parallelizable.

Might see its distributed version last lecture of class.

Outline For Today

1. SSSP in DAGs: DP Algorithm
2. SSSP without Negative Edges: Dijkstra's Greedy Algorithm
3. SSSP with Negative Edges: Bellman Ford DP Algorithm
4. All pairs Shortest Paths: Floyd Warshall DP Algorithm

All-Pairs Shortest Paths (APSP)

Input: Directed $G(V, E)$, arbitrary edge weights.

Output: $\forall$ u, v, d(u, v): shortest (u, v) path in G.

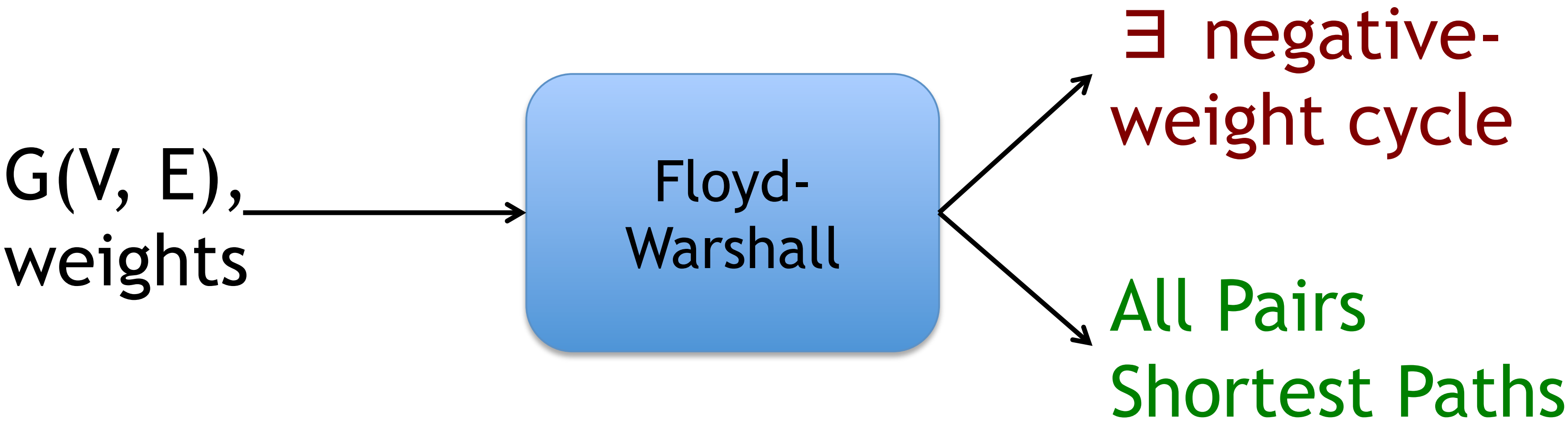
(no fixed source s)

Q: What's a lower-bound to solve APSP?

A: $O(n^2)$ b/c there are $O(n^2)$ outputs

Upshot: Floyd-Warshall's Properties

Note: Floyd-Warshall will be able to detect if there is a negative weight cycle!



Both outputs computed in asymptotically the same amount of time.

Floyd-Warshall Idea

Linear IS: input graph naturally ordered sequentially

Seq. Alignment: strings naturally ordered sequentially

SSSP in DAGs: topological ordering

FW imposes sequentiality on the vertices

$\Rightarrow$ order vertices from 1 to n

$\Rightarrow$ only use the first i vertices in each subproblem

(Same idea works for SSSP, but not very efficient)

Floyd-Warshall Subproblems

$V = \{1, \ldots, n\}$, ordered completely arbitrarily

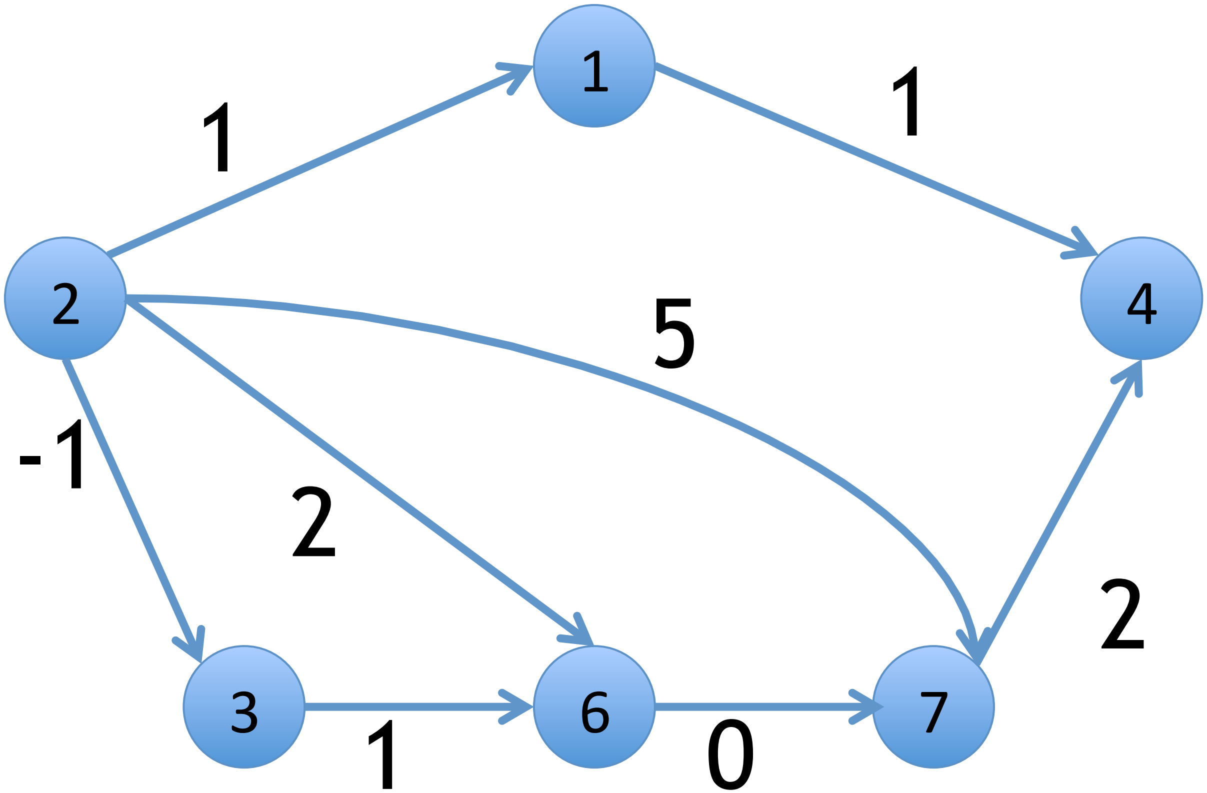
$V^k = \{1, \ldots, k\}$

Original Problem: $\forall (u, v)$ shortest u, v path.

We need to define the subproblems.

Subproblem $P_{(i, j, k)}$ = shortest i, j path that uses only V^k as intermediate nodes (excluding i and j).

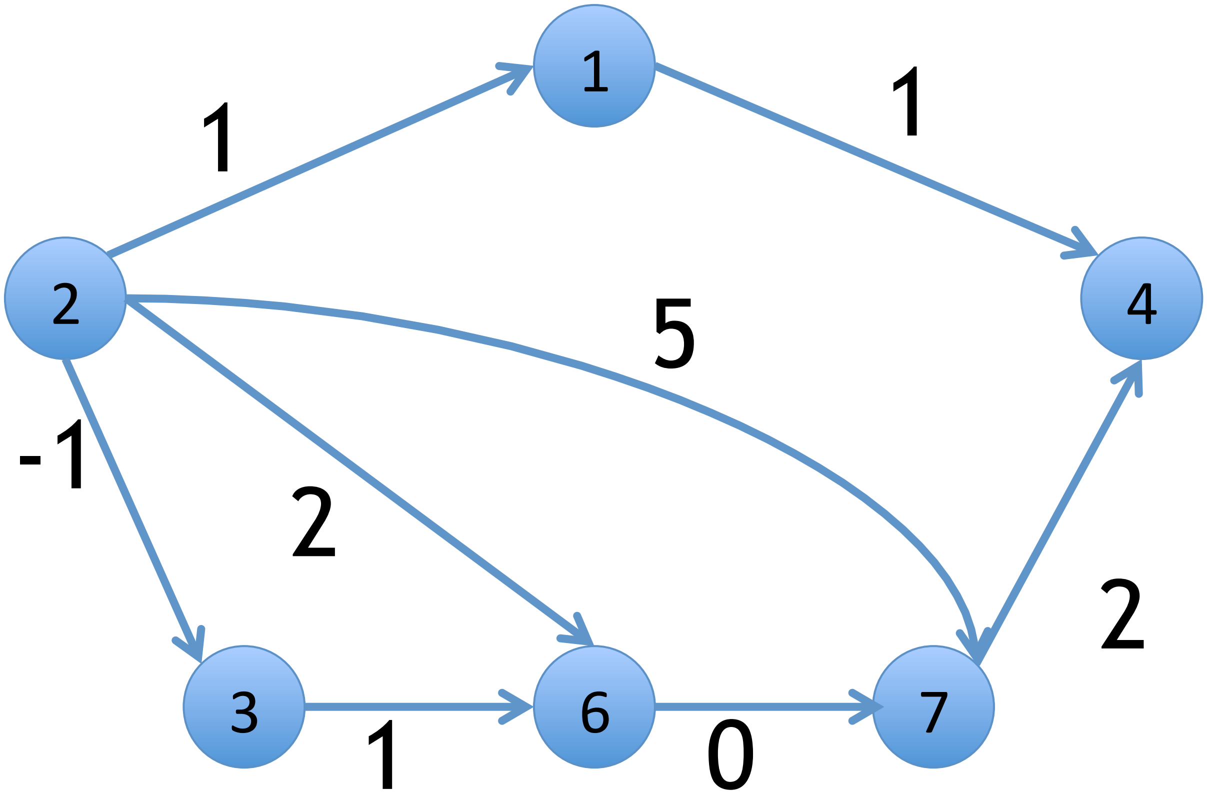
Floyd-Warshall Subproblems



Q: $P_{(6,4,1)}$?

A: null (weight of $+\infty$)

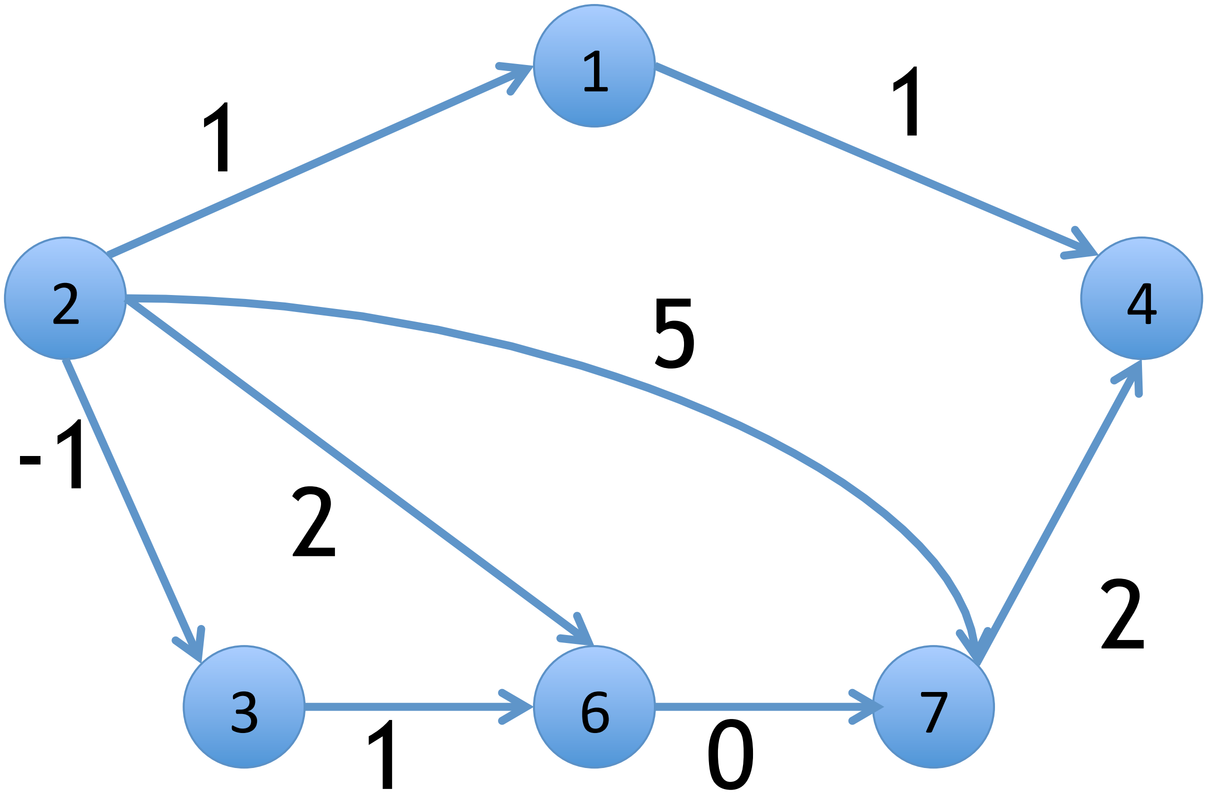
Floyd-Warshall Subproblems



Q: $P_{(2,4,1)}$?

A: 2->1->4 (weight of 2)

Floyd-Warshall Subproblems

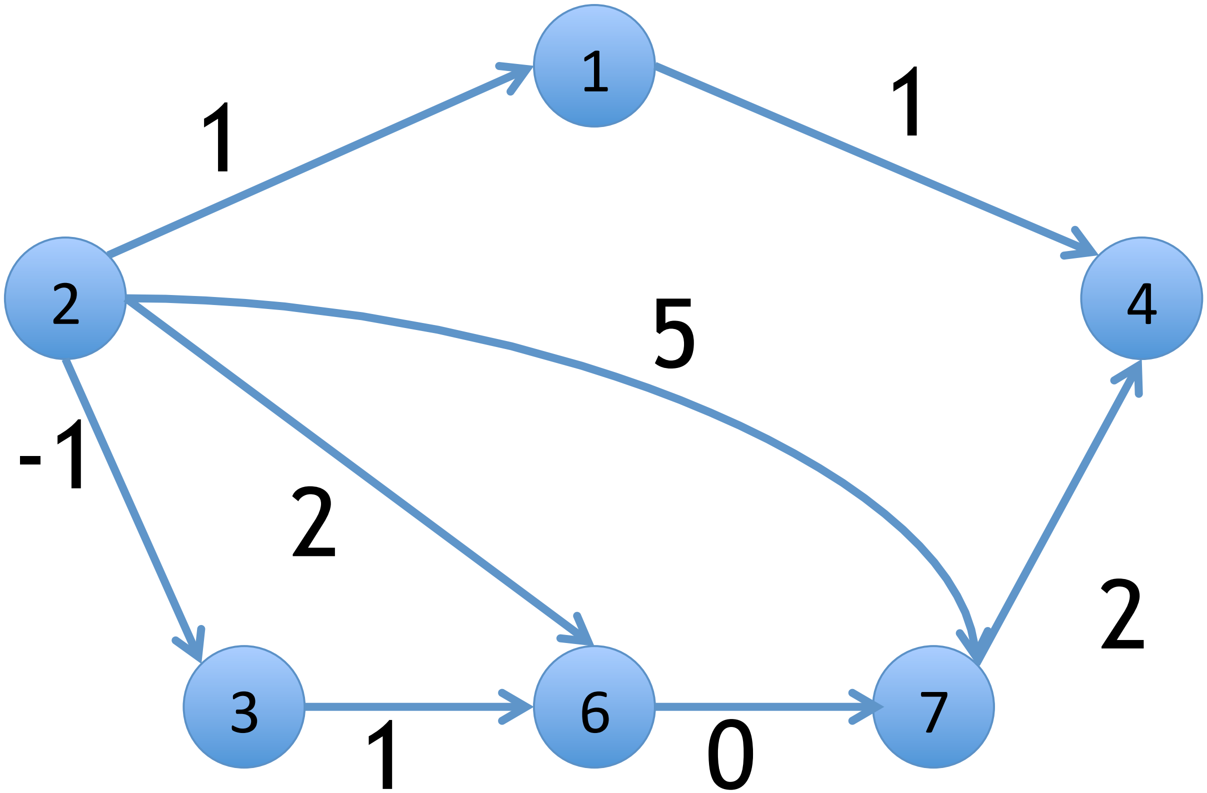


Q: $P_{(2,6,0)}$?

A: 2-6 (weight of 2)

(no intermediate nodes needed)

Floyd-Warshall Subproblems

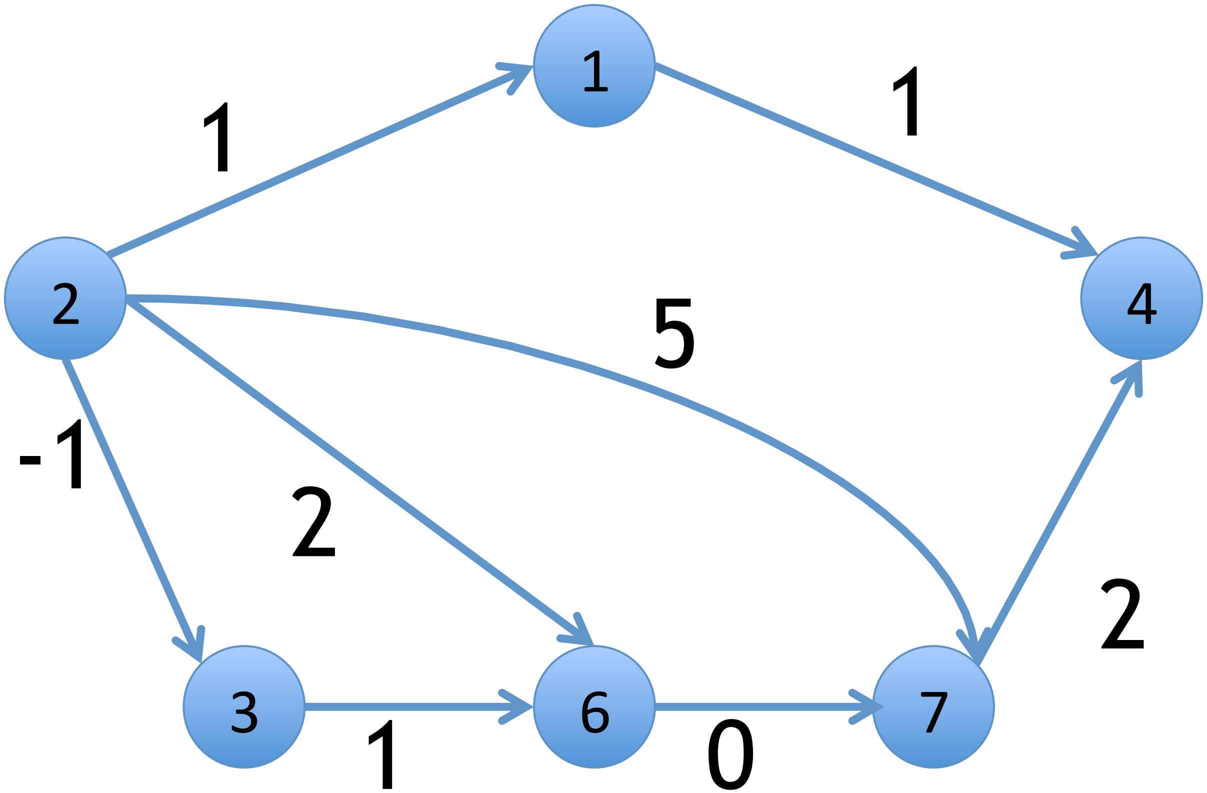


Q: $P_{(2,6,1)}$?

A: 2-6 (still weight of 2)

(without intermediate nodes)

Floyd-Warshall Subproblems

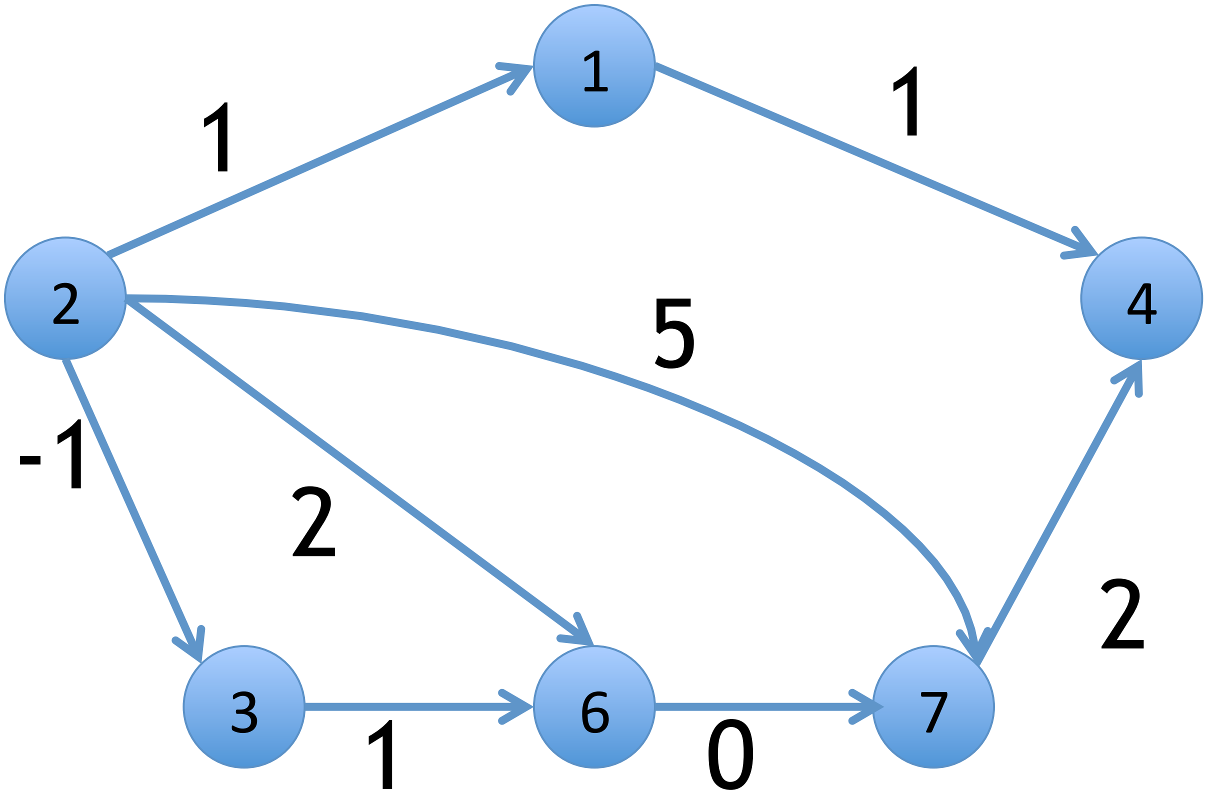


Q: $P_{(2,6,2)}$?

A: 2-6 (still weight of 2)

(without intermediate nodes)

Floyd-Warshall Subproblems

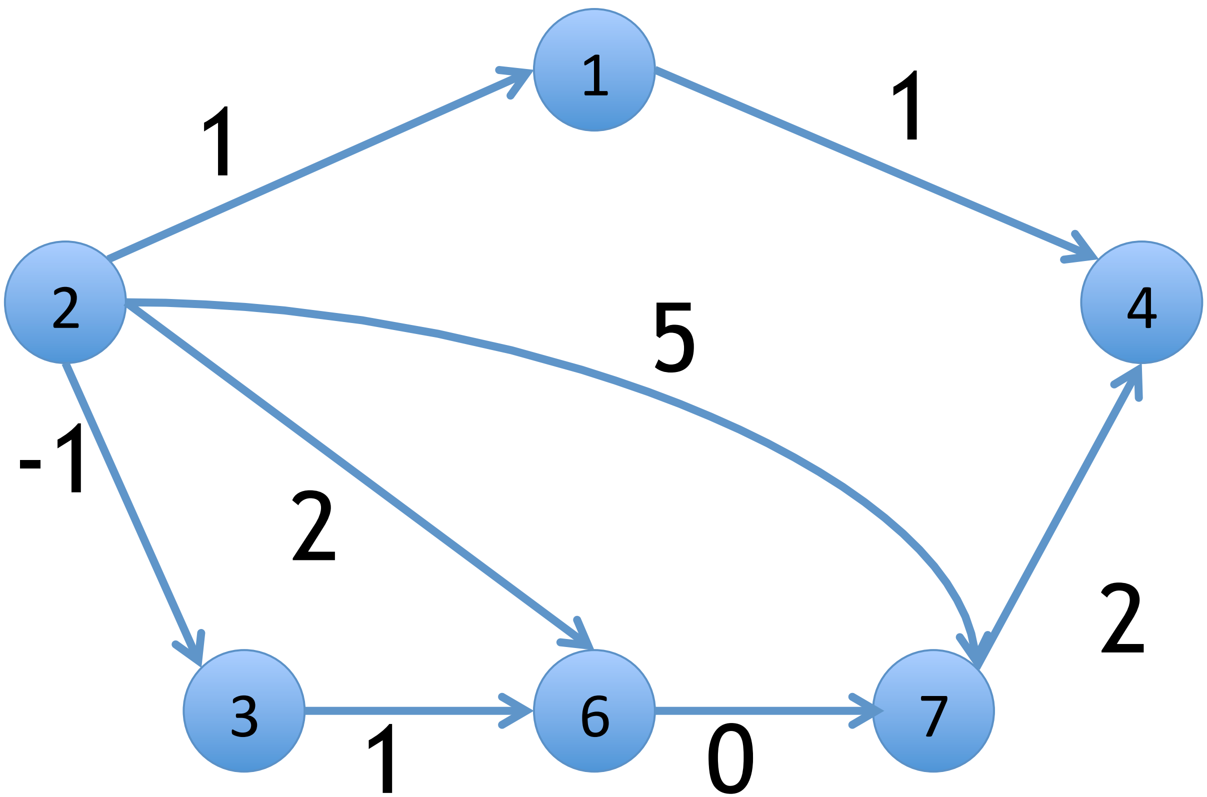


Q: $P_{(2,6,3)}$?

A: 2->3->6 (weight of 0)

(now with intermediate node 3)

Floyd-Warshall Subproblems

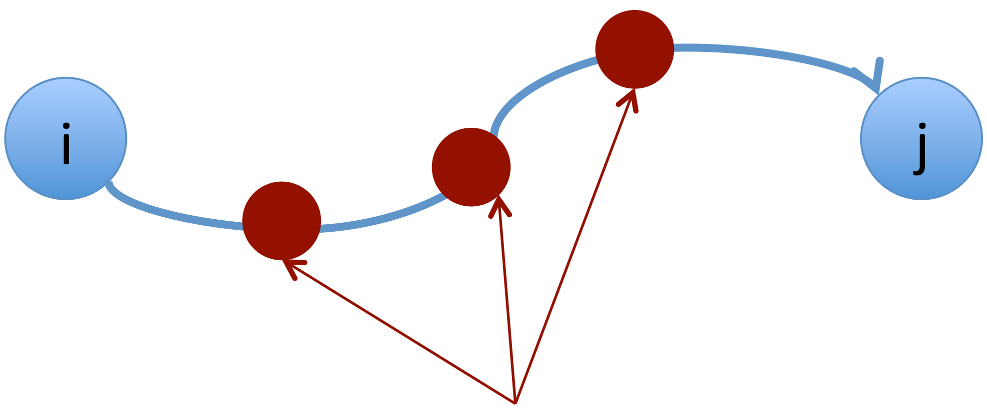


Final shortest $i \rightsquigarrow j$ path is $P_{(i,\, j,\, n)}$ when we're allowed to use any vertices as intermediate nodes.

Claim That Doesn't Require A Proof

Fix source i, and destination j. Consider $P_{(i,\ j,\ k)}$:

$$k \notin P_{(i,\ j,\ k)} \text{ OR } k \in P_{(i,\ j,\ k)}$$

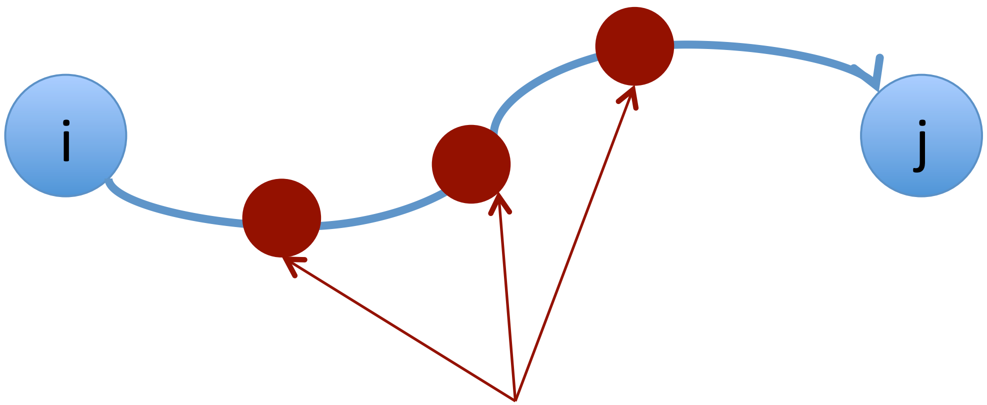


(either one of the intermediate vertices is k or it's not)

Case 1: $k \notin P_{(i,\ j,\ k)}$

Then all internal nodes are from 1,…,k-1.

Q: What can we assert about $P_{(i,\ j,\ k)}$?



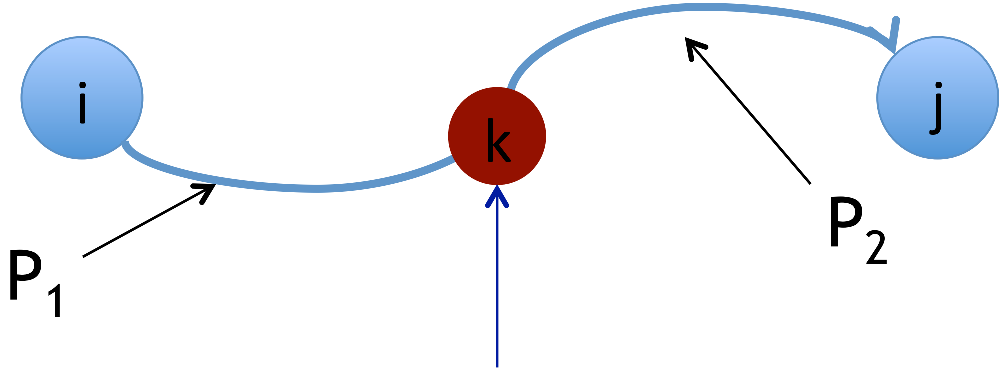
all intermediate vertices are from V^{k-1}

A: $P_{(i,\ j,\ k)} = P_{(i,\ j,\ k-1)}$

(proof by contradiction)

Case 2: $k \in P_{(i,\ j,\ k)}$

Q: What can we assert about P_1 and P_2?



one (and only one) of the int. nodes is k. (why only one?)

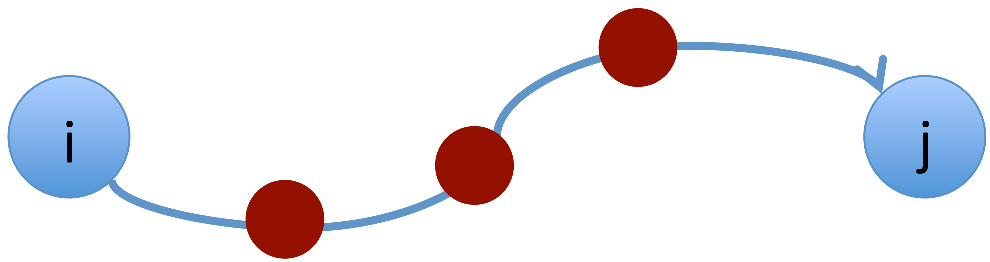
A1: P_1 & P_2 only contain int. nodes 1,…,k-1

A2: $P_1 = P_{(i,k,k-1)}$ & $P_2 = P_{(k,j,k-1)}$

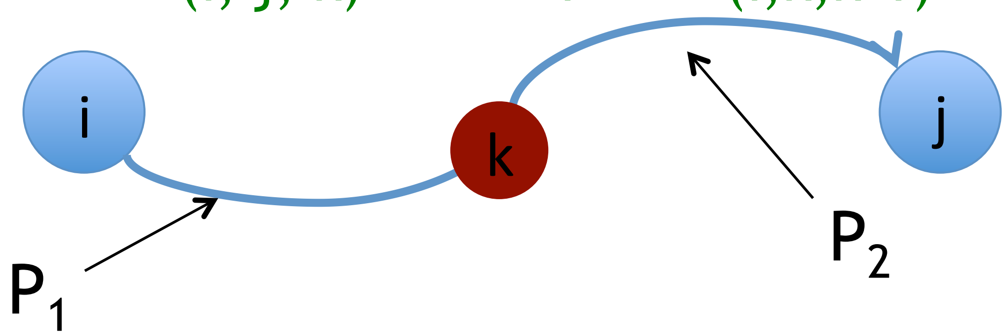
(proof by contradiction)

Summary of the 2 Cases

Case 1: $k \notin P_{(i,\ j,\ k)}$ => $P_{(i,\ j,\ k)} = P_{(i,\ j,\ k-1)}$



Case 2: $k \in P_{(i,\ j,\ k)}$ => $P_1 = P_{(i,k,k-1)}$ & $P_2 = P_{(k,j,k-1)}$



Recurrence for Larger Subproblems

$\forall$ i, j, k and where i,j,k={1, …, n}

$P_{(i,\ j,\ k)}$: shortest i $\rightsquigarrow$ j path with all intermediate nodes from V^k={1,…,k} (or null)

$L_{(i,\ j,\ k)}$: w($P_{(i,\ j,\ k)}$) (and +∞ for null paths)

$$L_{(i,\ j,\ k)} = \min \begin{cases} L_{(i,\ j,\ k-1)} \\ L_{(i,\ k,\ k-1)} + L_{(k,\ j,\ k-1)} \end{cases}$$

With appropriate base cases.

Floyd-Warshall Algorithm

Let A be an nxnxn 3D array.

A[i][j][k] = shortest i⇝j path with V^k as intermediate nodes

```
procedure Floyd-Warshall(G(V,E), weights C):
  Base Cases: A[i][i][0]
```

Floyd-Warshall Algorithm

Let A be an nxnxn 3D array.

A[i][j][k] = shortest i⇝j path with V^k as intermediate nodes

```
procedure Floyd-Warshall(G(V,E), weights C):
  Base Cases: A[i][i][0] = 0
              A[i][j][0] =
```

Floyd-Warshall Algorithm

Let A be an nxnxn 3D array.

A[i][j][k] = shortest i⤳j path with V^k as intermediate nodes

```
procedure Floyd-Warshall(G(V,E), weights C):
 Base Cases: A[i][i][0] = 0
             A[i][j][0] = C_{i,j} if (i,j) ∈ E
                          +∞ if (i,j) ∉ E
 for k = 1, …, n:
   for i = 1, …, n:
     for j = 1, …, n:
        A[i][j][k] = min {A[i][j][k-1],
                      A[i][k][k-1] + A[k][j][k-1]}
```

Correctness & Runtime

Correctness: induction on i,j,k & correctness of recurrence

Runtime: $O(n^3)$ (b/c n^3 subproblems, $O(1)$ for each one)

```
procedure Floyd-Warshall(G(V,E), weights C):
 Base Cases: A[i][i][0] = 0
             A[i][j][0] = C_{i,j} if (i,j) ∈ E
                          +∞ if (i,j) ∉ E
 for k = 1, …, n:
   for i = 1, …, n:
     for j = 1, …, n:
       A[i][j][k] = min {A[i][j][k-1],
                  A[i][k][k-1] + A[k][j][k-1]}
```

Detecting Negative Cycles

Just check the A[i][i][n] for each i!

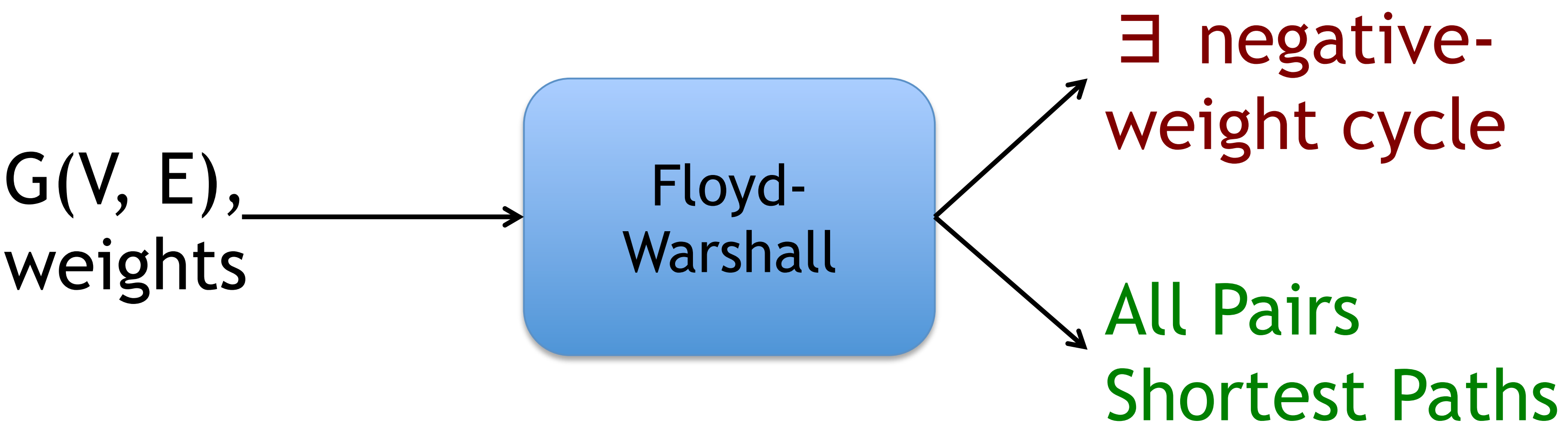
Let C be a negative cycle with l the largest

ID vertex on C

=>for any vertex j on C, A[j][j][l] ≤ 0

=> therefore A[j][j][n] will be negative

As Promised


G(V, E),
weights
Floyd-
Warshall
∃ negative-
weight cycle
All Pairs
Shortest Paths

Path Reconstruction

Keep successors for each i j path in an array S[i][j].

Initially, S[i][j] = null or j if (i,j) exists.

If A[i][j][k] = A[i][k][k-1] + A[k][j][k-1]

then update S[i][j] to S[i][k].

E.g: Suppose at termination S[i][j] = w.

Then we look at S[w][j] = z

Then we look at S[z][j] … until we hit j.

SSSP DAG, Dijkstra, FW

	SSSP DAG	Dijkstra	Bellman-Ford	FW
Single-Source /All Pairs	Single-Source	Single-Source	Single Source	All Pairs
Run-time	$O(n + m)$	$O(m\log(n))$	$O(mn)$	$O(n^3)$
Negative Edges	Yes	No	Yes	Yes
Negative Cycles	No	No	No, but can detect	No, but can detect

Next Week: Intractability, P vs NP &

What to Do for NP-hard Problems?

Especially don’t miss the first lecture!