Lecture 13

Shortest Path: Dijkstra's

Single source shortest paths

- Done: BFS to find the minimum path length from **v** to **u** in $O(|E|+|V|)$

- Actually, can find the minimum path length from **v** to *every node*
  - Still $O(|E|+|V|)$
  - No faster way for a "distinguished" destination in the worst-case

- Now: Weighted graphs

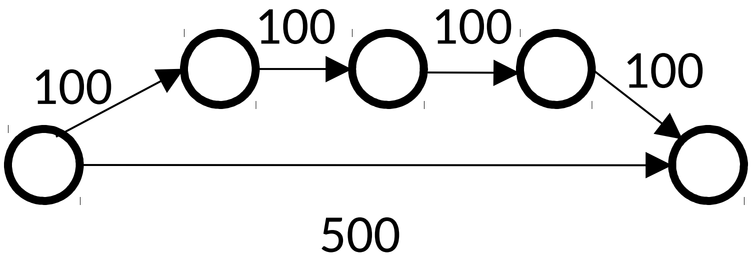
Given a weighted graph and node **v**,
find the minimum-cost path from **v** to every node

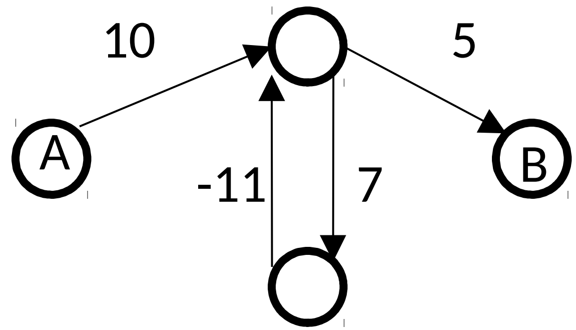
- As before, asymptotically no harder than for one destination
- Unlike before, BFS will not work -> only looks at path length.

Shortest Path: Applications

- Driving directions
- Cheap flight itineraries
- Network routing
- Critical paths in project management

Not as easy





Why BFS won't work: Shortest path may not have the fewest edges

- Annoying when this happens with costs of flights

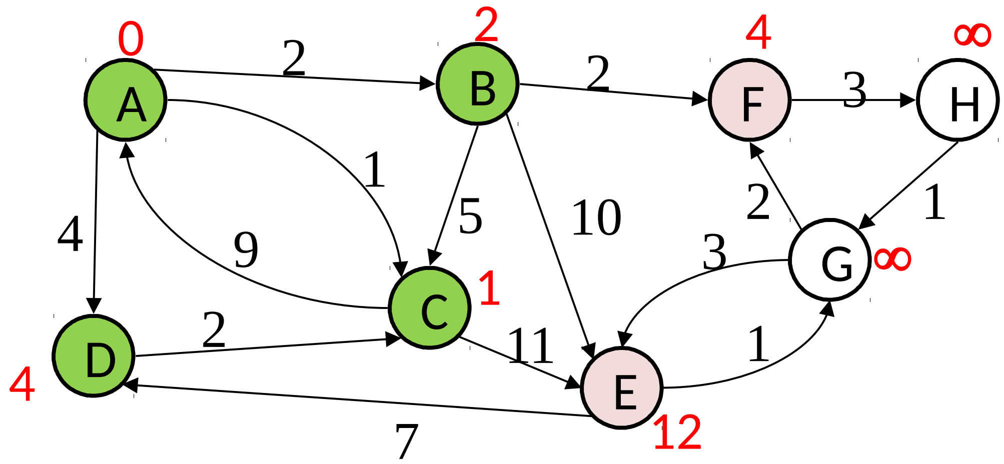
We will assume there are no negative weights

- *Problem is ill-defined* if there are negative-cost *cycles*
- *Today's algorithm is wrong* if *edges* can be negative
  - There are other, slower (but not terrible) algorithms

Dijkstra’s algorithm

- The idea: reminiscent of BFS, but adapted to handle weights
  - Grow the set of nodes whose shortest distance has been computed
  - Nodes not processed yet will have a “best distance so far”
  - A priority queue will turn out to be useful for efficiency

Dijkstra's Algorithm: Idea



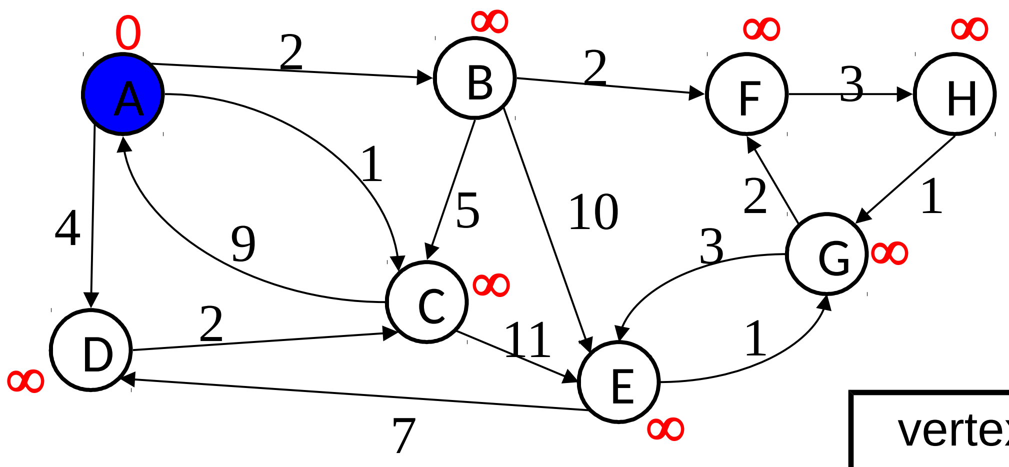
- Initially, start node has cost 0 and all other nodes have cost ∞
- At each step:
  - Pick closest unknown vertex **v**
  - Add it to the "cloud" of known vertices
  - Update distances for nodes with edges from **v**
- That's it! (But we need to prove it produces correct answers)

The Algorithm

1. For each node **`v`**, set **`v.cost = ∞`** and **`v.known = false`**
2. Set **`source.cost = 0`**
3. While there are unknown nodes in the graph
   a) Select the unknown node **`v`** with lowest cost
   b) Mark **`v`** as known
   c) For each edge **`(v,u)`** with weight **`w`**,

```
c1 = v.cost + w // cost of best path through v  to u
 c2 = u.cost  // cost of best path to u previously known
 if(c1 < c2){ // if the path through v is better
  u.cost = c1
    u.path = v // for computing actual paths
 }
```

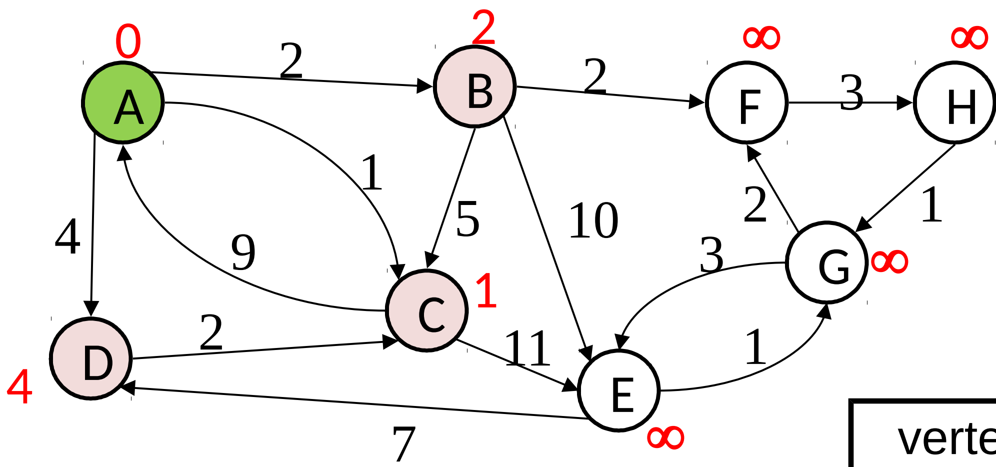
Example #1



Order Added to Known Set:

vertex	known?	cost	path
A		0	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
G		∞	
H		∞	

Example #1

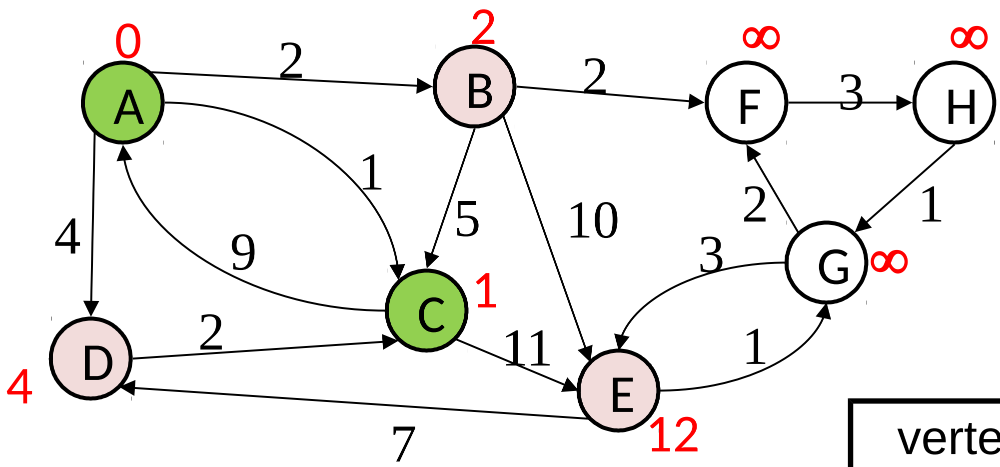


vertex	known?	cost	path
A	Y	0	
B		≤ 2	A
C		≤ 1	A
D		≤ 4	A
E		∞	
F		∞	
G		∞	
H		∞	

<u>Order Added to Known Set:</u>

A

Example #1

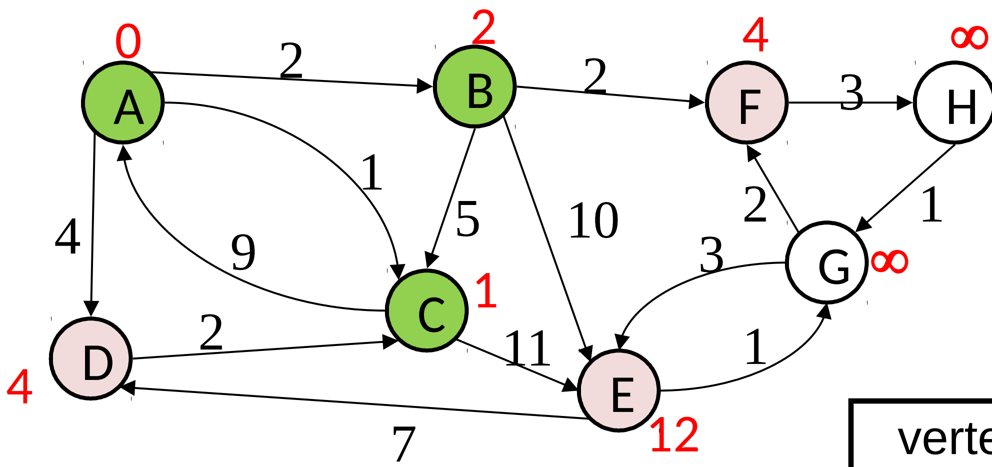


vertex	known?	cost	path
A	Y	0	
B		≤ 2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		∞	
G		∞	
H		∞	

<u>Order Added to Known Set:</u>

A, C

Example #1

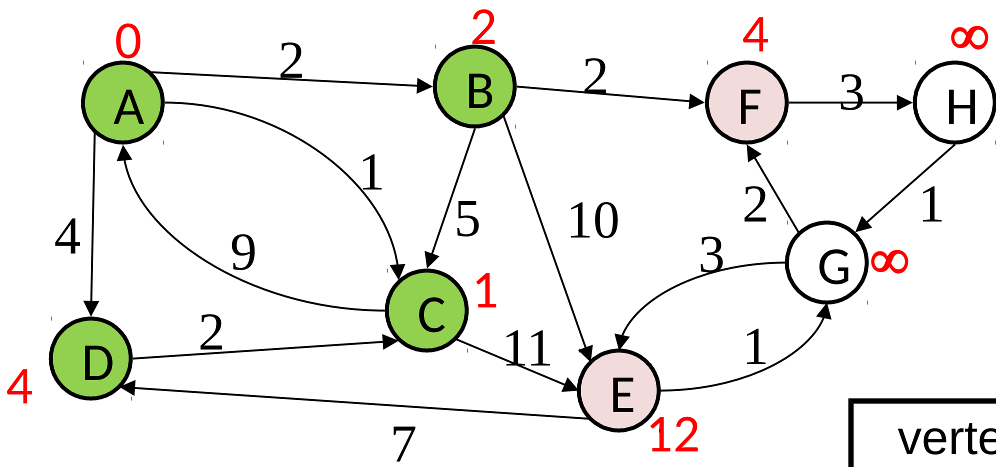


vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D		$\leq$ 4	A
E		$\leq$ 12	C
F		$\leq$ 4	B
G		∞	
H		∞	

<u>Order Added to Known Set:</u>

A, C, B

Example #1

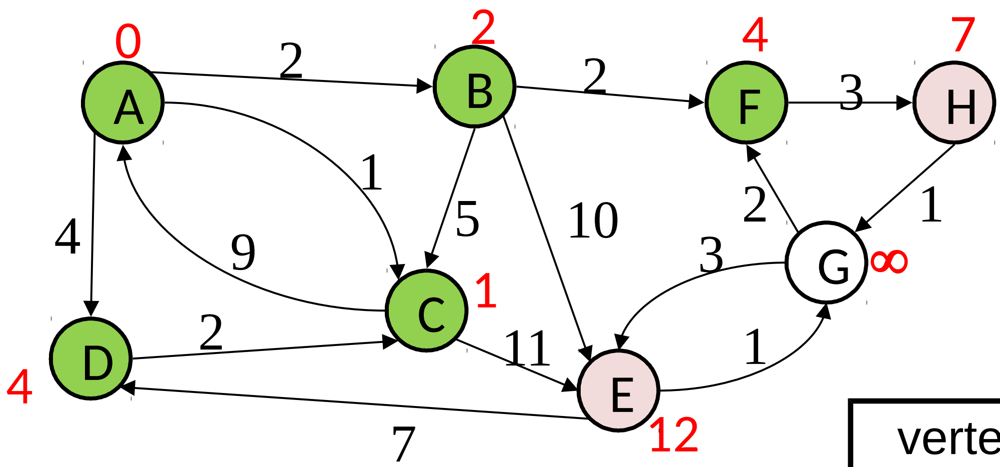


vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		$\leq$ 12	C
F		$\leq$ 4	B
G		∞	
H		∞	

<u>Order Added to Known Set:</u>

A, C, B, D

Example #1

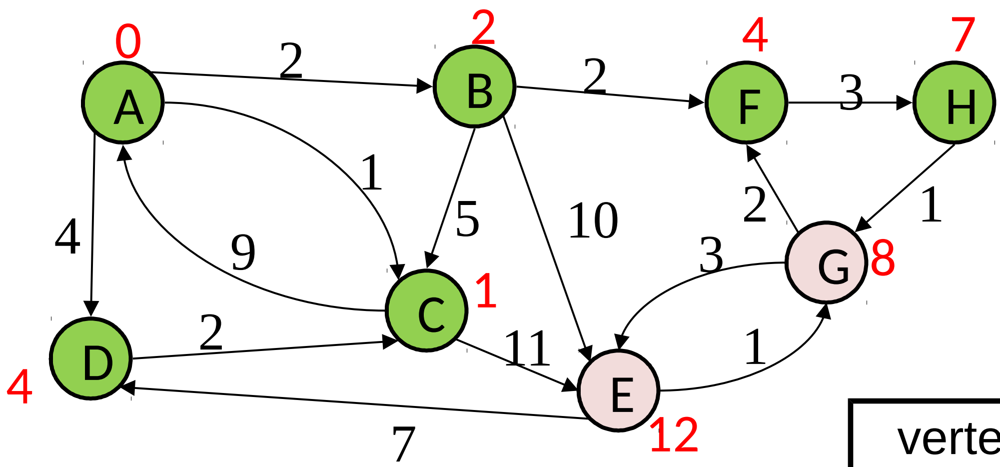


vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		∞	
H		≤ 7	F

Order Added to Known Set:

A, C, B, D, F

Example #1

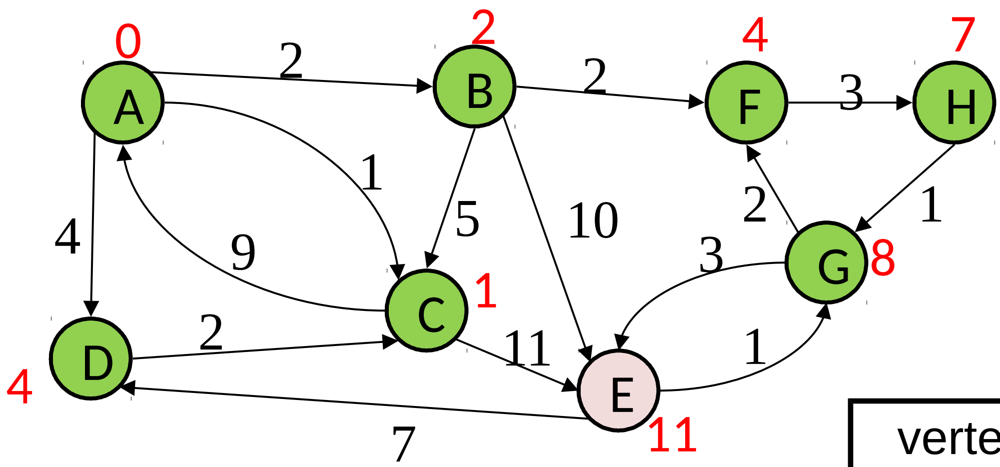


vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		$\leq$ 12	C
F	Y	4	B
G		$\leq$ 8	H
H	Y	7	F

<u>Order Added to Known Set:</u>

A, C, B, D, F, H

Example #1

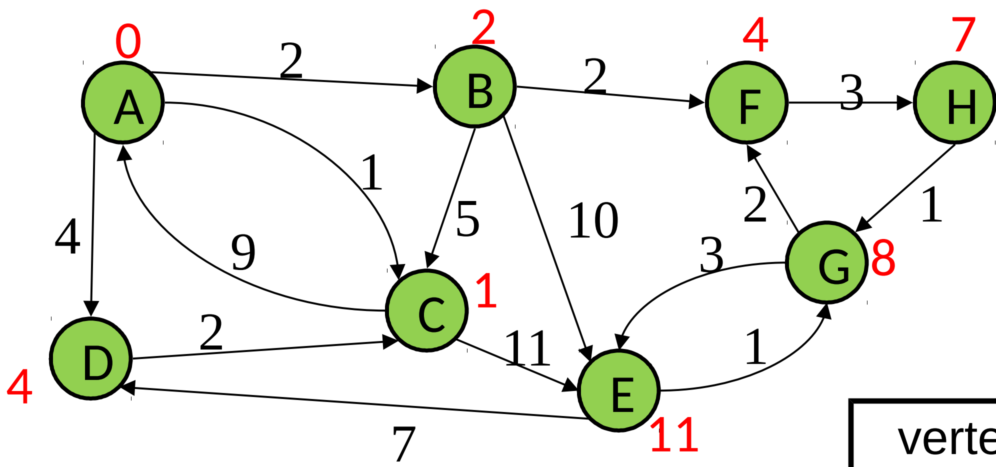


vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		$\leq$ 11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

<u>Order Added to Known Set:</u>

A, C, B, D, F, H, G

Example #1



vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Order Added to Known Set:

A, C, B, D, F, H, G, E

Features

- When a vertex is marked known,
  the cost of the shortest path to that node is known
  - The path is also known by following back-pointers

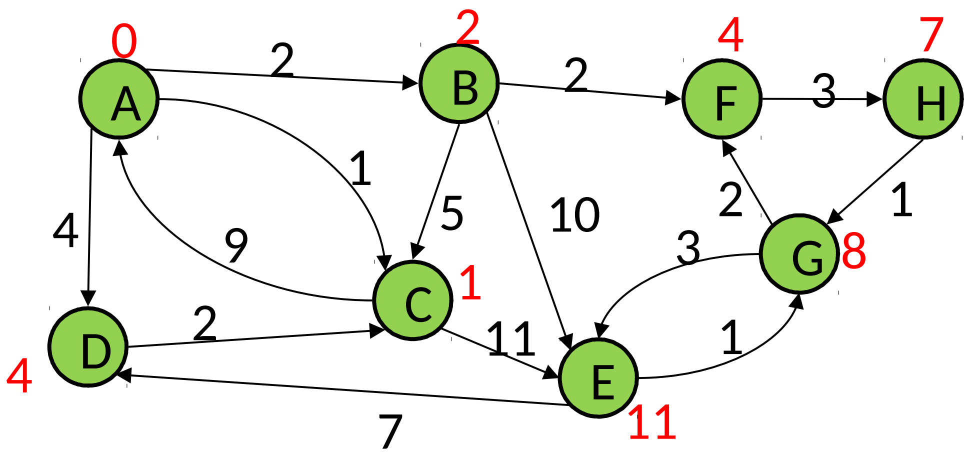
- While a vertex is still not known,
  another shorter path to it might still be found

Note: The “Order Added to Known Set” is not important

- A detail about how the algorithm works (client doesn’t care)
- Not used by the algorithm (implementation doesn’t care)
- It is sorted by path-cost, resolving ties in some way
  - Helps give intuition of why the algorithm works

Interpreting the Results

- Now that we're done, how do we get the path from, say, A to E?



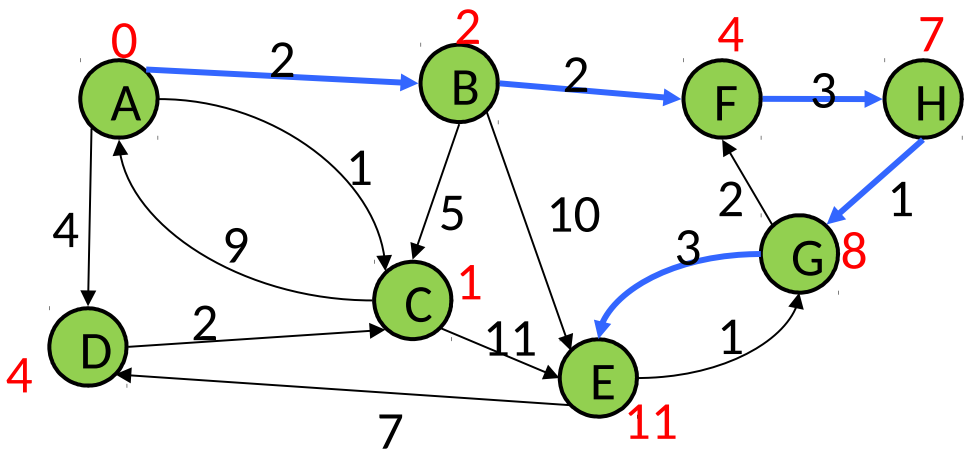
vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Order Added to Known Set:

A, C, B, D, F, H, G, E

Interpreting the Results

- Now that we're done, how do we get the path from, say, A to E?



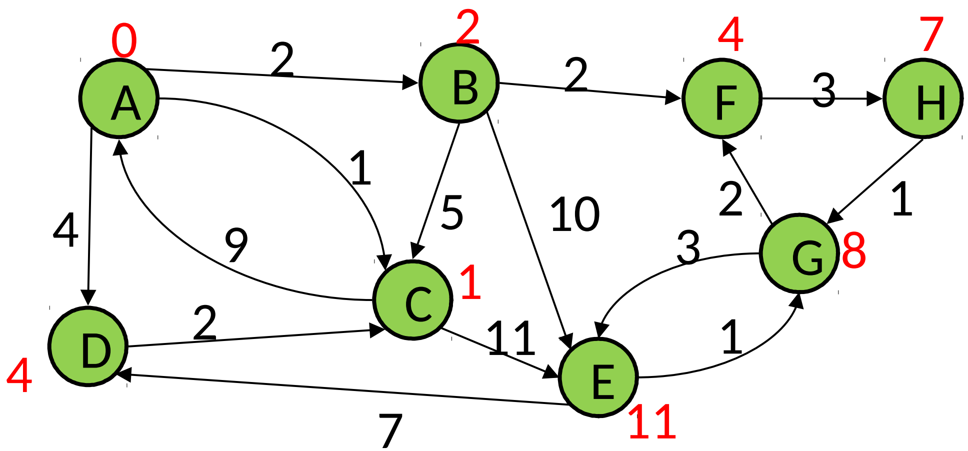
Order Added to Known Set:

A, C, B, D, F, H, G, E

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Stopping Short

- How would this have worked differently if we were only interested in:
  - The path from A to F?

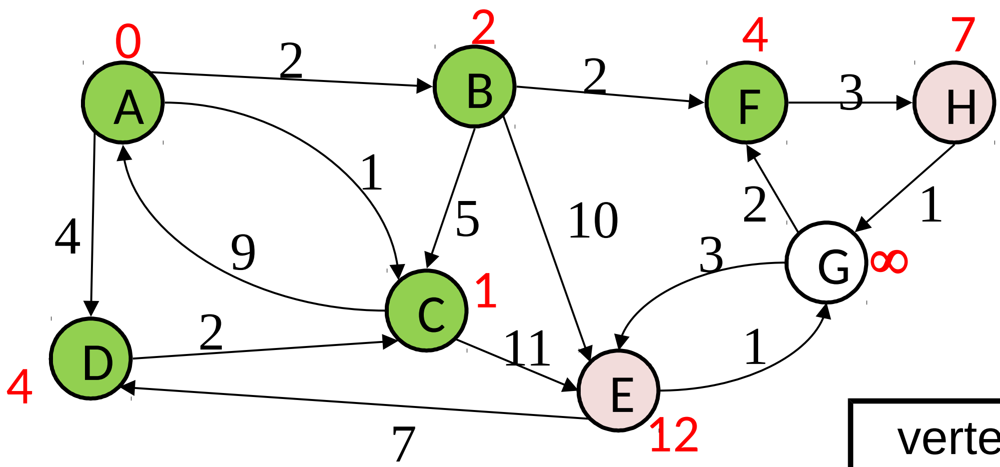


Order Added to Known Set:

A, C, B, D, F, H, G, E

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Stopping Short

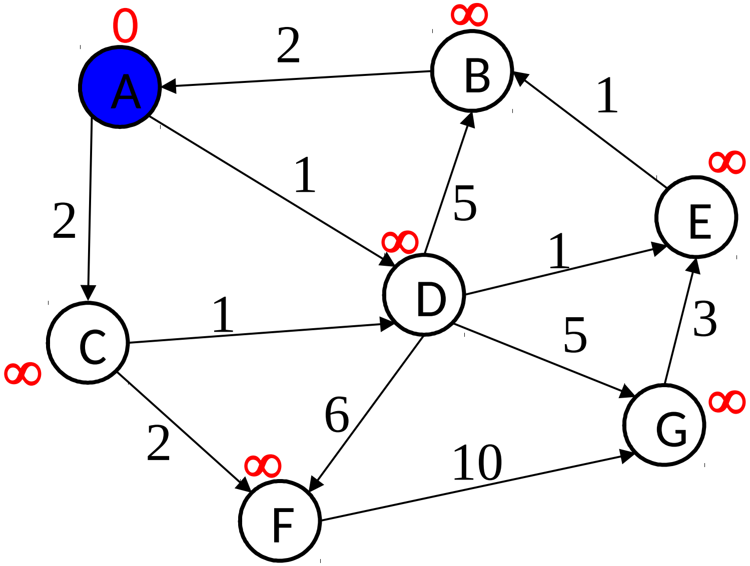


vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		$\leq$ 12	C
F	Y	4	B
G		∞	
H		$\leq$ 7	F

<u>Order Added to Known Set:</u>

A, C, B, D, F

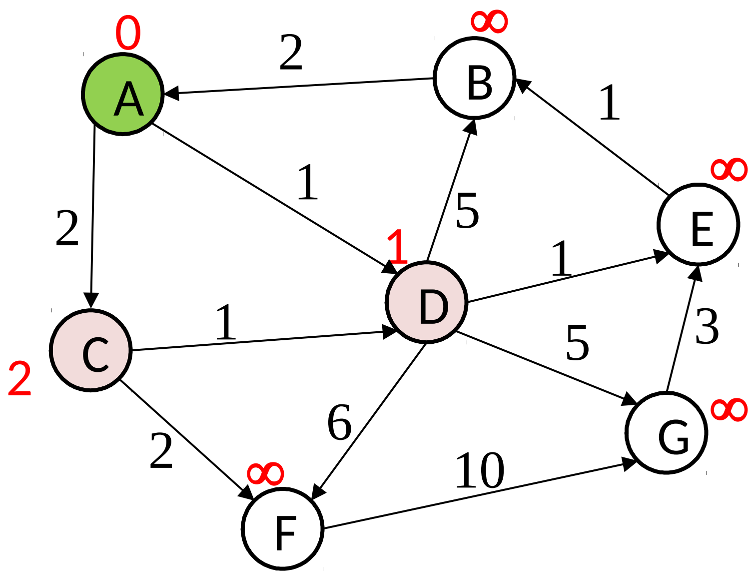
Example #2



Order Added to Known Set:

vertex	known?	cost	path
A		0	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
G		∞	

Example #2

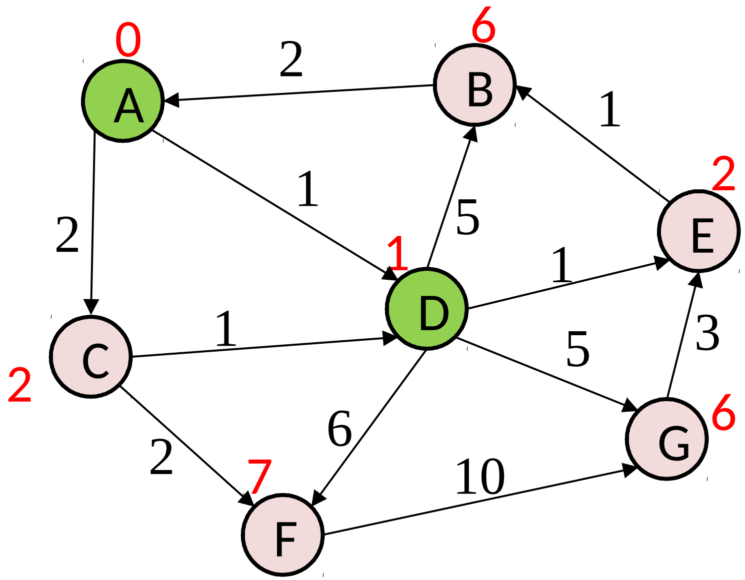


vertex	known?	cost	path
A	Y	0	
B		∞	
C		≤ 2	A
D		≤ 1	A
E		∞	
F		∞	
G		∞	

Order Added to Known Set:

A

Example #2

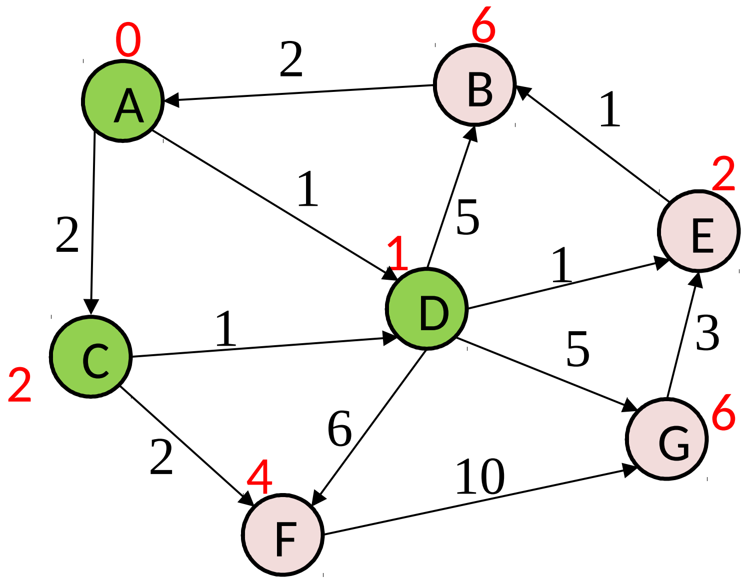


Order Added to Known Set:

A, D

vertex	known?	cost	path
A	Y	0	
B		≤ 6	D
C		≤ 2	A
D	Y	1	A
E		≤ 2	D
F		≤ 7	D
G		≤ 6	D

Example #2

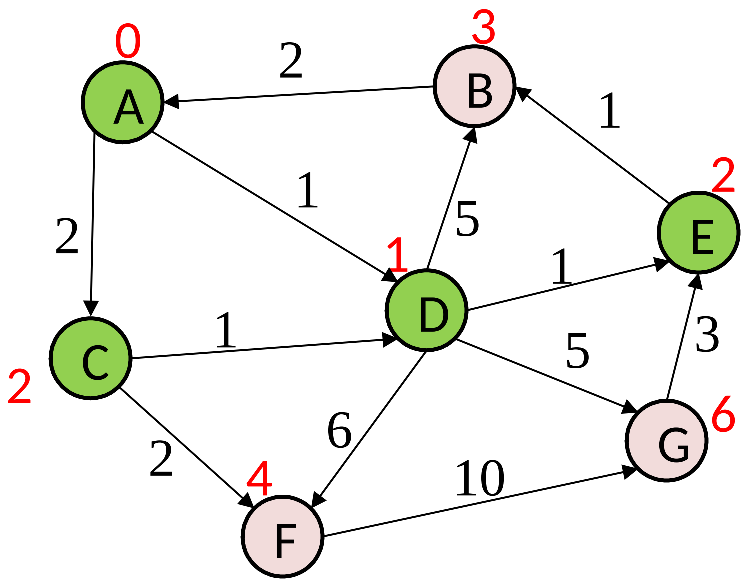


Order Added to Known Set:

A, D, C

vertex	known?	cost	path
A	Y	0	
B		≤ 6	D
C	Y	2	A
D	Y	1	A
E		≤ 2	D
F		≤ 4	C
G		≤ 6	D

Example #2

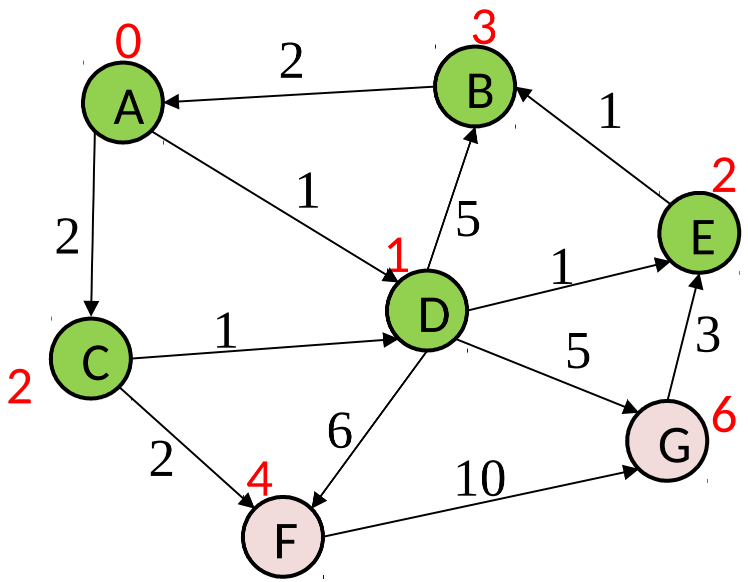


vertex	known?	cost	path
A	Y	0	
B		≤ 3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F		≤ 4	C
G		≤ 6	D

Order Added to Known Set:

A, D, C, E

Example #2

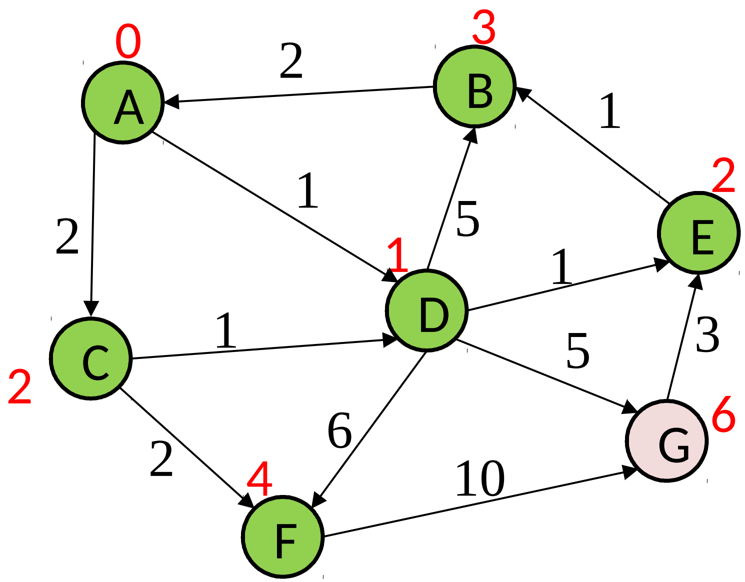


vertex	known?	cost	path
A	Y	0	
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F		≤ 4	C
G		≤ 6	D

Order Added to Known Set:

A, D, C, E, B

Example #2

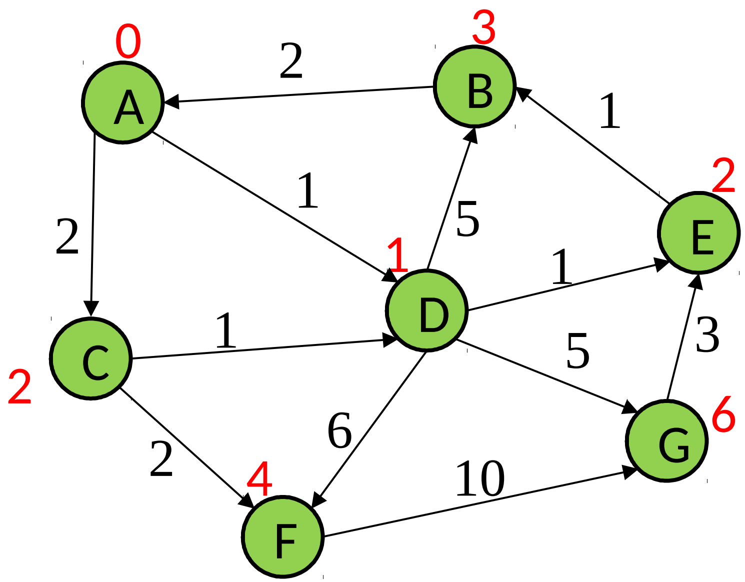


vertex	known?	cost	path
A	Y	0	
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F	Y	4	C
G		≤ 6	D

<u>Order Added to Known Set:</u>

A, D, C, E, B, F

Example #2

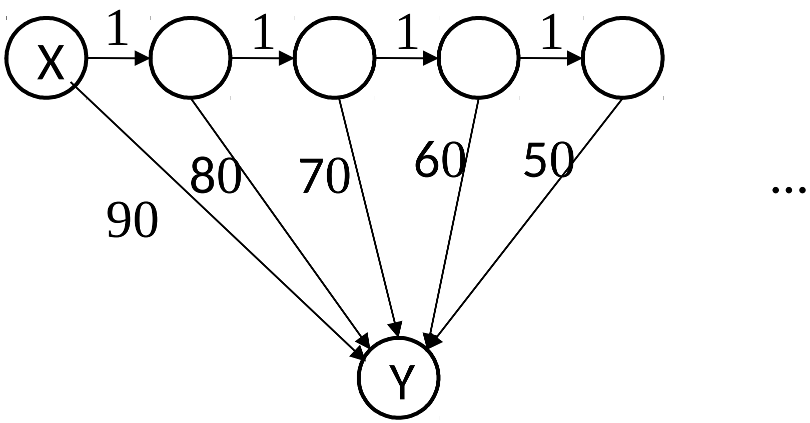


vertex	known?	cost	path
A	Y	0	
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F	Y	4	C
G	Y	6	D

Order Added to Known Set:

A, D, C, E, B, F, G

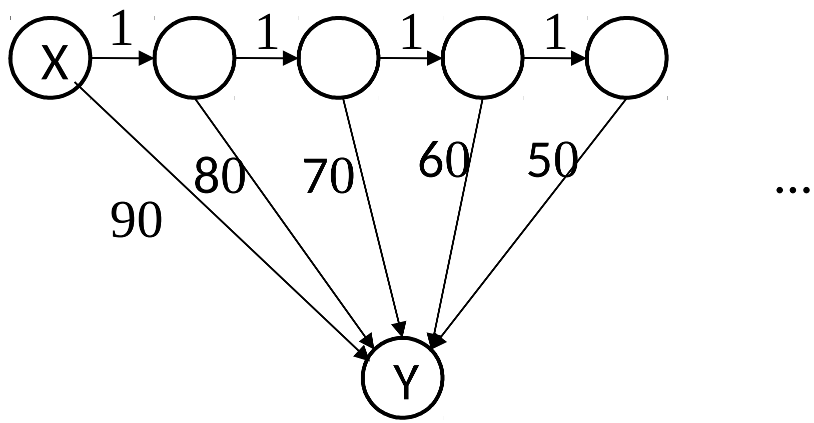
Example #3



As the algorithm runs, how will the best-cost-so-far for Y change?

Is this expensive?

Example #3



As the algorithm runs, how will the best-cost-so-far for Y change?

∞, 90, 81, 72, 63, 54, …

Is this expensive?

No, each *edge* is processed only once

A Greedy Algorithm

- *A greedy algorithm*:
  - At each step, irrevocably does what seems best at that step
    - A locally optimal step, not known to be globally optimal
  - Once a vertex is known, it is not revisited
    - Turns out to be globally optimal
- Dijkstra's algorithm
  - For single-source shortest paths in a weighted graph (directed or undirected) with no negative-weight edges

Correctness: Intuition

Rough intuition:

All the "known" vertices have the correct shortest path

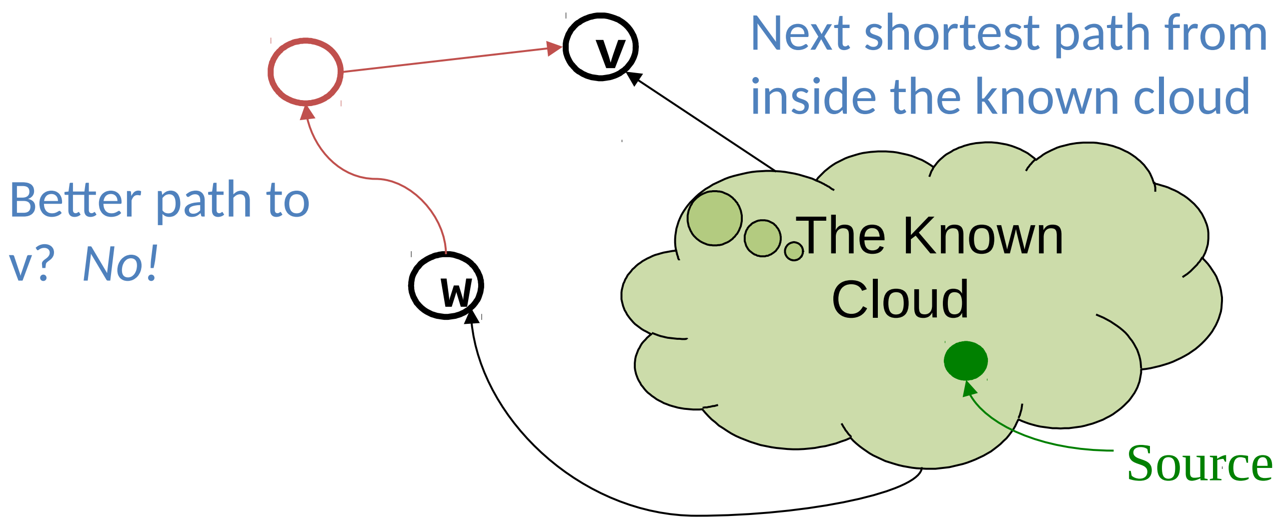
- True initially: shortest path to start node has cost 0
- If it stays true every time we mark a node "known", then **by induction** this holds and eventually everything is "known"

  (we'll get to talk about induction soon! ^^)

When we mark a vertex "known" we won't discover a shorter path later!

- This holds only because Dijkstra's algorithm picks the node with the next shortest path-so-far
- The proof is by contradiction…

Correctness: The Cloud (Rough Sketch)



Suppose v is the next node to be marked known ("added to the cloud")

- The best-known path to v must have only nodes "in the cloud"
  - Else we would have picked a node closer to the cloud than v
- Suppose the actual shortest path to v is different
  - It won't use only cloud nodes, or we would know about it
  - So it must use non-cloud nodes. Let w be the *first* non-cloud node on this path. The part of the path up to w is already known and must be shorter than the best-known path to v. So v would not have been picked. Contradiction.

Naïve asymptotic running time

Similar to topological sort, if we think of the algorithm as:

```
loop that runs based on # of vertices:
loop to find the next vertex to process:
process step cost based on # edges
```

Then the algorithm is $O(|V|^2)$ due to each iteration looking for the node to process next

- We solved it in Topological Sort with a queue of zero-degree nodes
- But here we need the next lowest-cost node, not necessarily the first node we put in our pending set. And costs can change as we process edges!

Improving asymptotic running time

Solution?

- A priority queue holding all unknown nodes, sorted by cost
- But must support `decreaseKey` operation
  - Must maintain a reference from each node to its current position in the priority queue
  - Conceptually simple, but can be a **pain** to code up

Efficiency, second approach

We'll use pseudocode to determine asymptotic run-time

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  build-heap with all nodes
  while(heap is not empty) {
    b = deleteMin()
    b.known = true
    for each edge (b,a) in G
     if(!a.known)
       if(b.cost + weight((b,a)) < a.cost){
         decreaseKey(a,"new cost – old cost")
         a.path = b
       }
}
```

Efficiency, second approach

We'll use pseudocode to determine asymptotic run-time

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  build-heap with all nodes
  while(heap is not empty) {
    b = deleteMin()
    b.known = true
    for each edge (b,a) in G
     if(!a.known)
       if(b.cost + weight((b,a)) < a.cost){
         decreaseKey(a,"new cost – old cost")
         a.path = b
       }
}
```

O(|V|)

O(|V|log|V|)

O(|E|log|V|)

O(|V|log|V|+|E|log|V|)