

# Dijkstra's Algorithm

CSE 332 Summer 2020

**Instructor:** Richard Jiang

## **Teaching Assistants:**

Hamsa Shankar Kristin Li Winston Jodjana

Maggie Jiang Hans Zhang Michael Duan

Jeffery Tian Annie Mao

*Lecture Q&A: [pollev.com/332summer](https://pollev.com/332summer)*

*Lecture clarifications: [tinyurl.com/332-08-12A](https://tinyurl.com/332-08-12A)*

# Announcements

- ❖ Guest lecture this Friday, don't miss it!
- ❖ Look out for checkpoint feedback soon...
  - This will help with your parallelism style

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-12A](https://tinyurl.com/332-08-12A)*

# Lecture Outline

## ❖ Traversals

- Trees and Graphs: Level-order / Breadth-first
- Graphs: Depth-first
- Conclusion

## ❖ Shortest Paths!

## ❖ Dijkstra's Algorithm

- Introduction
- Correctness Proof
- Runtime

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-12A](https://tinyurl.com/332-08-12A)*

# Tree and Graph Traversals

- ❖ Can answer reachability with a *tree traversal* or *graph traversal*
  - Iterates over every node in a graph in some defined ordering
  - “Processes” or “visits” its contents
- ❖ There are several types of tree traversals
  - Level Order Traversal aka Breadth-First Traversal
  - Depth-First Traversal
    - Pre-order Traversal
    - In-order Traversal
    - Post-order Traversal

# Tree/Graph Traversal: High-level Algorithm

## ❖ High-level Algorithm:

### ■ Initialization:

- Create an empty data structure to track “remaining work”
- Mark start as visited

### ■ While we still have work, follow the nodes:

- Get a node
- Visit/process that node
- Update its neighbors (eg, add to “remaining work” if it’s not already there)

## ❖ *Memorize this 5-step pattern!*

```
traverseGraph(Node start) {
    pending = emptySet();
    pending.add(start)
    mark start as visited

    while (!pending.empty()) {
        next = pending.remove()
        process(next)
        foreach u adjacent to next
            if (!u.marked)
                mark u
            pending.add(u)
    }
}
```

# Tree/Graph Traversal: Running Time

- ❖ Assuming  $\text{add}()$  and  $\text{remove}()$  are  $O(1)$ , traversal is  $O(|E|)$ 
  - Remember: we default to using an adjacency list

# Tree/Graph Traversal: Order

- ❖ The order we process() depends *entirely* on how pending.add() and pending.remove() are implemented
  - Queue:
    - Tree: Level-order traversal
    - Graph: Breadth-first graph search (BFS)
  - Stack:
    - Tree: Depth-first search (3 flavors!)
    - Graph: Depth-first graph search (DFS)
  
- ❖ DFS and BFS are “big ideas” in computer science
  - Depth: explore one part before exploring other unexplored parts
  - Breadth: explore parts closer to the start before exploring farther parts

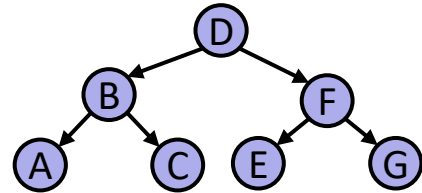
# Lecture Outline

- ❖ Traversals
  - **Trees and Graphs: Level-order / Breadth-first**
  - Graphs: Depth-first
  - Conclusion
  
- ❖ Shortest Paths!
  
- ❖ Dijkstra's Algorithm
  - Introduction
  - Correctness Proof
  - Runtime

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-12A](https://tinyurl.com/332-08-12A)*

# Trees: Level-Order

- ❖ Process top-to-bottom, left-to-right
  - Like reading in English
  - Goes “broad” instead of “deep”
- ❖ Resembles how we converted our binary heap (ie, a complete tree) to its array representation



# Graphs: Breadth-First

- ❖ Note: we call the need-to-explore vertices “the fringe”

```
BFS(Node start) {
    q.enqueue(start)
    mark start as visited

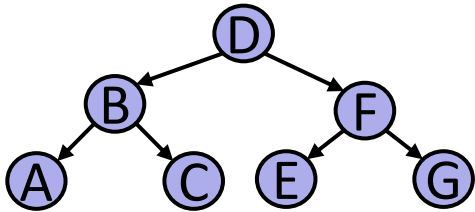
    while (!q.empty())
        next = q.dequeue()
        process(next)
        foreach u adjacent to next
            if (!u.marked)
                mark u
                q.enqueue(u)
}
```

# Lecture Outline

- ❖ Traversals
  - Trees and Graphs: Level-order / Breadth-first
  - **Graphs: Depth-first**
  - Conclusion
  
- ❖ Shortest Paths!
  
- ❖ Dijkstra's Algorithm
  - Introduction
  - Correctness Proof
  - Runtime

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-12A](https://tinyurl.com/332-08-12A)*

# Trees and Graphs: Depth-First



```
DFSIterative(Node start) {  
    s.push(start)  
    mark start as visited  
  
    while (!s.empty())  
        next = s.pop()  
        process(next)  
        foreach u adjacent to next  
            if (!u.marked)  
                mark u  
                s.push(u)  
}
```

# Lecture Outline

- ❖ Traversals
  - Trees and Graphs: Level-order / Breadth-first
  - Graphs: Depth-first
  - **Conclusion**
  
- ❖ Shortest Paths!
  
- ❖ Dijkstra's Algorithm
  - Introduction
  - Correctness Proof
  - Runtime

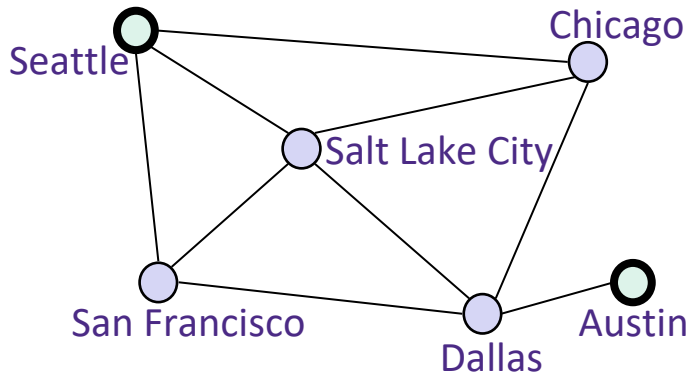
*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-12A](https://tinyurl.com/332-08-12A)*

# Saving the Path

- ❖ These graph traversals can answer the “reachability question”:
  - “Is there a path from node  $x$  to node  $y$ ?”
- ❖ But what if we want to output the actual path or its length?
  - Eg, getting driving directions vs knowing it’s possible to get there
- ❖ Modifications:
  - Instead of just “marking” a node, store the path’s previous node
    - ie: when processing  $u$ , if we add  $v$  to the “remaining work” set  $v.\text{prev}$  to  $u$
  - When you reach the goal, follow  $\text{prev}$  fields backwards to start
    - (don’t forget to reverse the answer)
  - Path length:
    - Same idea, but also store integer distance at each node

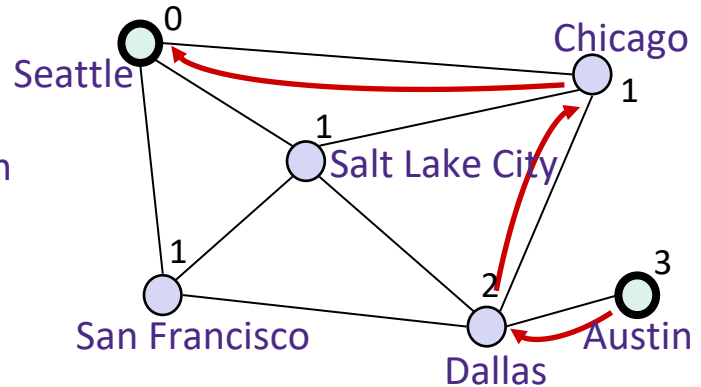
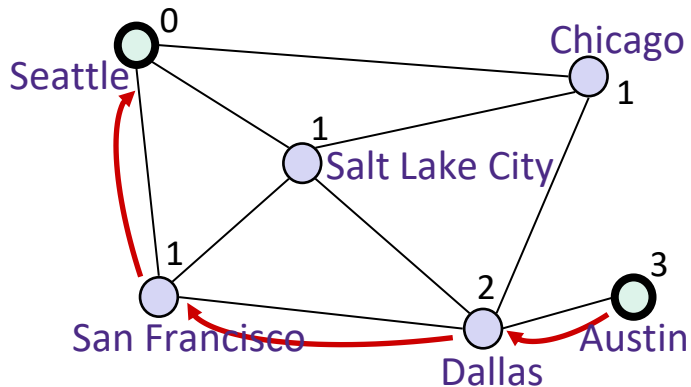
# Saving the Path: Example using BFS (1 of 2)

- ❖ Find the shortest path from Seattle to Austin
  - Remember marked nodes are not re-enqueued
  - Shortest paths may not be unique



## Saving the Path: Example using BFS (2 of 2)

- ❖ Find the shortest path from Seattle to Austin
  - Remember marked nodes are not re-enqueued
  - Shortest paths may not be unique



# DFS/BFS Comparison

## ❖ Breadth-first search:

- Always finds shortest paths, i.e., finds “optimal solutions”
  - Better for “what is the shortest path from x to y?”
- But queue may hold up to  $O(|V|)$  nodes
  - Eg, at the bottom level of perfect binary tree, queue contains  $|V|/2$  nodes

## ❖ Depth-first search:

- Can use less space when finding a path
  - If longest path in the graph is  $p$  and highest out-degree is  $d$  then stack never has more than  $d \cdot p$  elements

# It Doesn't Have to be Either/Or

- ❖ A third approach: Iterative deepening (IDDFS):
  - Try DFS, but don't allow recursion more than  $K$  levels deep
  - If fails to find a solution, increment  $K$  and start the entire search over
- ❖ Like BFS, finds shortest paths. Like DFS, less space

# Summary

- ❖ Two very different “standard” graph representations
  - Must understand tradeoffs to choose between adj list and adj matrix
- ❖ TopoSort finds a total ordering in a DAG representing a partial ordering
  - Runtime for TopoSort was dependent on graph representation and a helper data structure!
- ❖ We can traverse both trees and graphs
  - Depth-first-style tree traversals have 3 flavors (named by when the processing happens)
  - Breadth-first-style tree traversals are called “level-order”
  - Graphs can have “pre-” and “post-” style traversals, but ordering is less important than in trees

# Lecture Outline

## ❖ Traversals

- Trees and Graphs: Level-order / Breadth-first
- Graphs: Depth-first
- Conclusion

## ❖ Shortest Paths!

## ❖ Dijkstra's Algorithm

- Introduction
- Correctness Proof
- Runtime

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-12A](https://tinyurl.com/332-08-12A)*

# Single-Source Shortest Paths

- ❖ We've seen BFS finds the minimum path length from  $v$  to  $u$ 
  - Runtime:  $O(|E|+|V|)$
- ❖ Actually, BFS finds the min path length from  $v$  to *every vertex*
  - Still  $O(|E|+|V|)$
  - Worst-case runtime for single-destination is no faster than worst-case runtime for all-destinations

# Shortest Path: Applications

- ❖ Network routing
- ❖ Driving directions
- ❖ Cheap flight tickets
- ❖ Critical paths in project management (see textbook)
- ❖ ...

*Wait, these are all weighted graphs!*

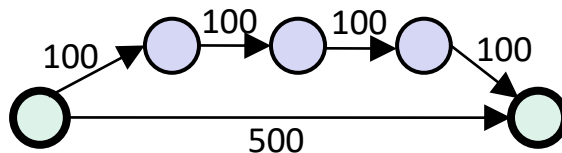
## Single-Source Shortest Paths ... *for Weighted Graphs*

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

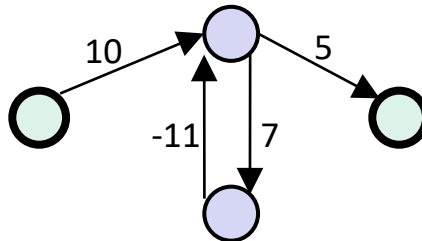
- ❖ As before:
  - All-destinations is asymptotically no harder than single-destination
- ❖ Unlike before:
  - BFS will not work

# BFS for Weighted Graphs

- ❖ BFS doesn't work! Shortest path may not have fewest edges
  - Eg: cost of flight. May be cheaper to fly through a hub than fly direct

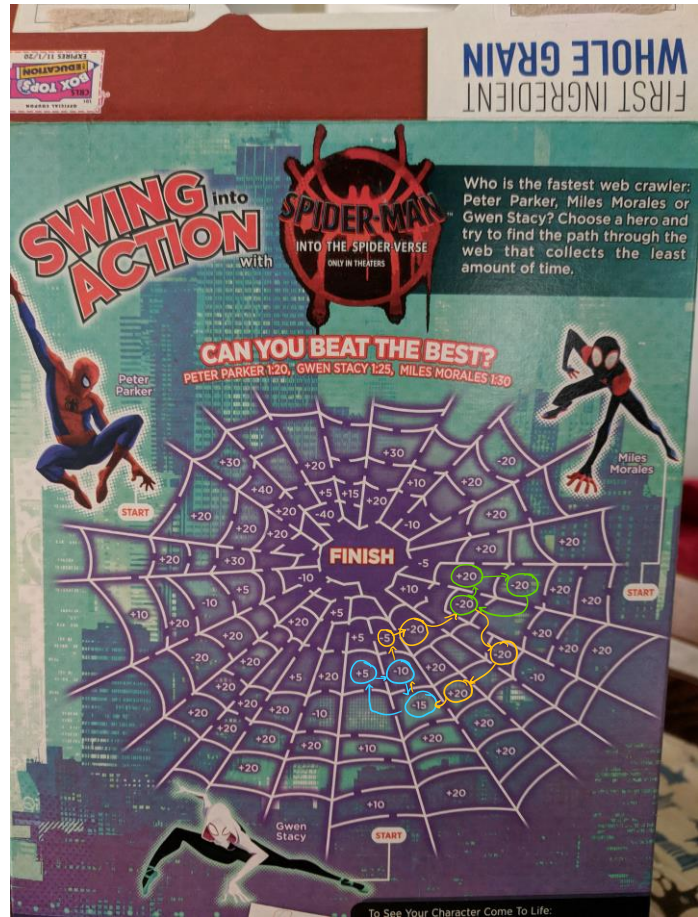


- ❖ We will assume there are *no negative edge weights*
  - Entire problem is *ill-defined* if there are negative-cost *cycles*
  - Today's algorithm is *wrong* if there are negative-cost *edges*



# Negative Cycles vs Negative Edges

- ❖ *Negative cycles*: no algorithm can find a finite optimal path
  - You can always decrease the distance by going through the negative cycle a few more times
  
- ❖ *Negative edges*: Dijkstra's can't guarantee correctness
  - But other algorithms might



# Lecture Outline

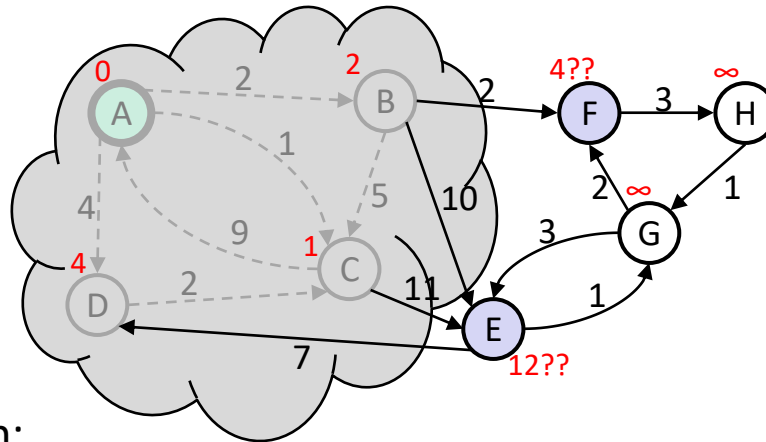
- ❖ Traversals
  - Trees and Graphs: Level-order / Breadth-first
  - Graphs: Depth-first
  - Conclusion
  
- ❖ Shortest Paths!
  
- ❖ Dijkstra's Algorithm
  - **Introduction**
  - Correctness Proof
  - Runtime

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-12A](https://tinyurl.com/332-08-12A)*

# Dijkstra's Algorithm

- ❖ Named after its inventor, Edsger Dijkstra (1930-2002)
  - Truly one of the “founders” of computer science
  - 1972 Turing Award
  - This algorithm is just *one* of his many contributions!
  - Example quote: “Computer science is no more about computers than astronomy is about telescopes”
- ❖ The idea: reminiscent of BFS, but adapted to handle weights
  - Grow the set of nodes whose shortest distance has been computed
  - Nodes not in the set will have a “best distance so far”

# Dijkstra's Algorithm: Idea



## ❖ Initialization:

- Start vertex has distance **0**; all other vertices have distance  $\infty$

## ❖ At each step:

- Pick closest unknown vertex  $v$
- Add it to the “cloud” of known vertices
- Update distances for nodes with edges from  $v$

# Dijkstra's Algorithm: Pseudocode

```
dijkstra(Graph g, Vertex start) {
  foreach vertex v in g:
    v.distance =  $\infty$ 
    v.known = false
  start.distance = 0

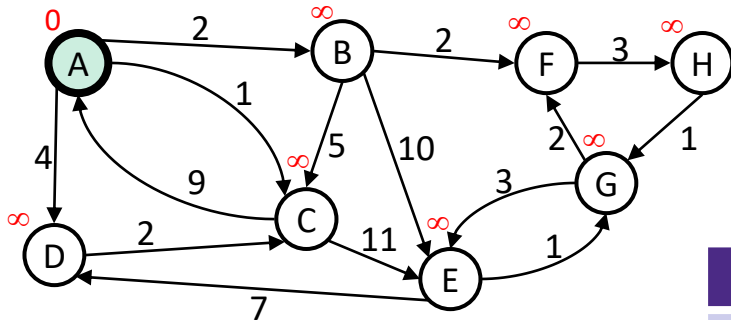
  while there are vertices in g that are not known:
    select vertex v with lowest cost
    v.known = true
    foreach edge (v, u) with weight w:
      d1 = v.distance + w // best path through v to u
      d2 = u.distance // previous best path to u
      if (d1 < d2) // if this is a better path to u
        u.distance = d1
        u.previous = v // backtracking info to
                        // recreate path
}
```

*Remember our 5-step pattern for a graph traversal?*

# Dijkstra's Algorithm: Important Features

- ❖ Once a vertex is marked known, its shortest path is known
  - Can reconstruct path by following back-pointers (“previous” fields)
- ❖ While a vertex is not known, another shorter path might be found

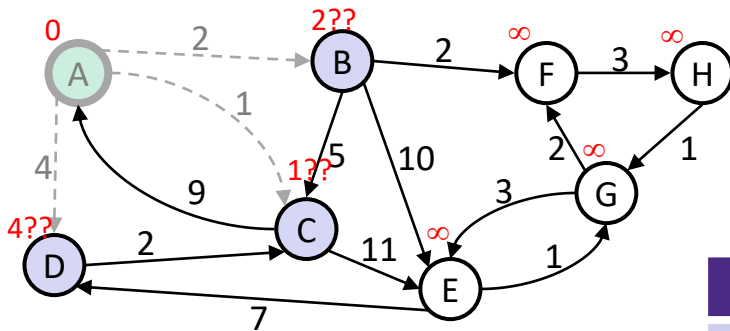
# Dijkstra's Algorithm: Example #1



Order Added to Known Set:

Vertex	Known?	Distance	Previous
A		0	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
G		∞	
H		∞	

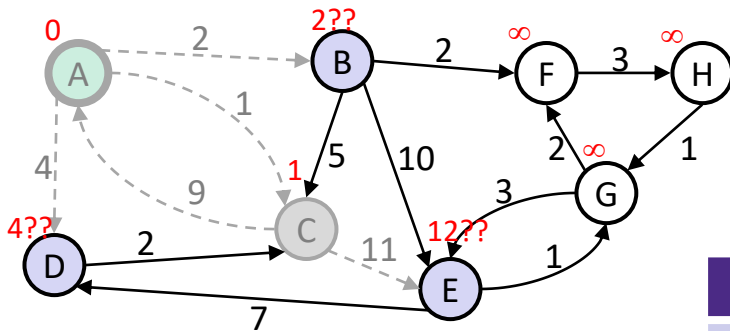
# Dijkstra's Algorithm: Example #1



Order Added to Known Set:  
A

Vertex	Known?	Distance	Previous
A	Y	0	/
B		$\leq 2$	A
C		$\leq 1$	A
D		$\leq 4$	A
E		$\infty$	
F		$\infty$	
G		$\infty$	
H		$\infty$	

# Dijkstra's Algorithm: Example #1

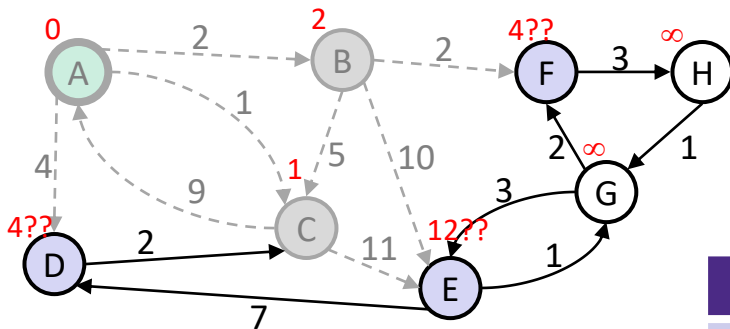


Order Added to Known Set:

A, C

Vertex	Known?	Distance	Previous
A	Y	0	/
B		≤ 2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		∞	
G		∞	
H		∞	

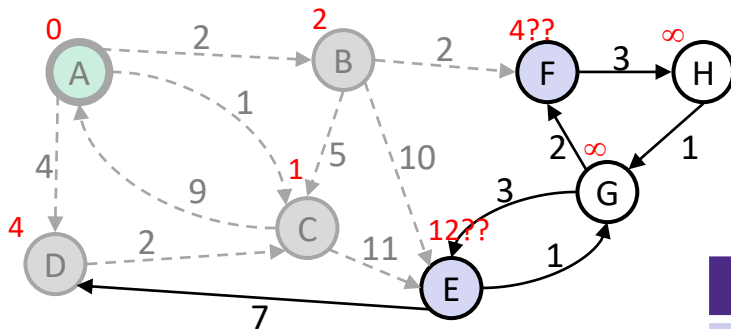
# Dijkstra's Algorithm: Example #1



Order Added to Known Set:  
A, C, B

Vertex	Known?	Distance	Previous
A	Y	0	/
B	Y	2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		≤ 4	<b>B</b>
G		∞	
H		∞	

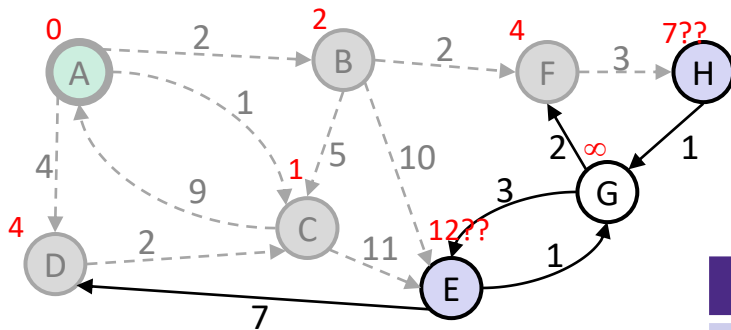
# Dijkstra's Algorithm: Example #1



Order Added to Known Set:  
A, C, B, D

Vertex	Known?	Distance	Previous
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F		≤ 4	B
G		∞	
H		∞	

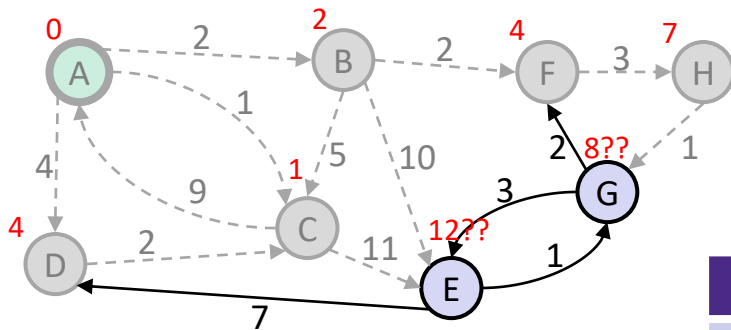
# Dijkstra's Algorithm: Example #1



Order Added to Known Set:  
A, C, B, D, F

Vertex	Known?	Distance	Previous
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	<b>F</b>

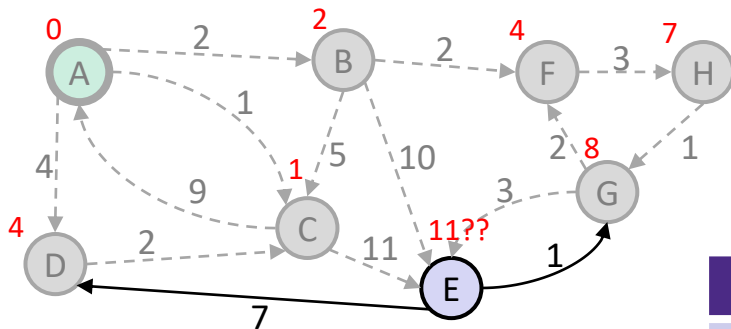
# Dijkstra's Algorithm: Example #1



Order Added to Known Set:  
A, C, B, D, F, H

Vertex	Known?	Distance	Previous
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		≤ 8	H
H	Y	7	F

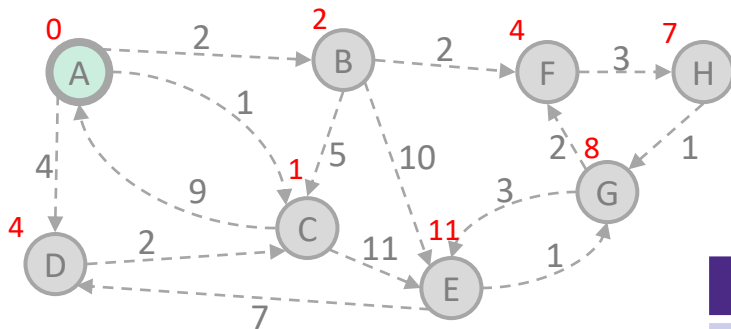
# Dijkstra's Algorithm: Example #1



Order Added to Known Set:  
A, C, B, D, F, H, G

Vertex	Known?	Distance	Previous
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		$\leq 11$	<b>G</b>
F	Y	4	B
G	Y	8	H
H	Y	7	F

# Dijkstra's Algorithm: Example #1



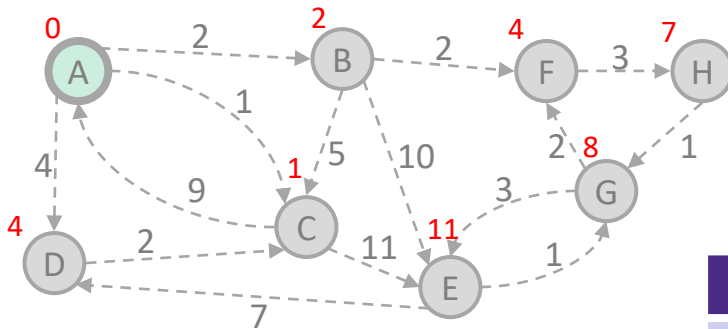
  **TADA!!!**  

Order Added to Known Set:

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

Vertex	Known?	Distance	Previous
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

# Dijkstra's Algorithm: Interpreting the Results



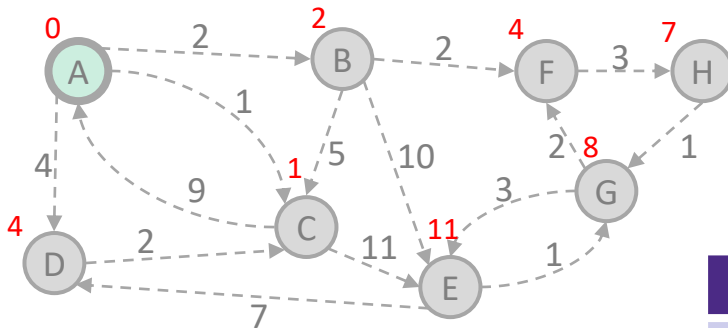
- ❖ Now that we're done, how do we get the path from A to E?

Order Added to Known Set:

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

Vertex	Known?	Distance	Previous
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

# Dijkstra's Algorithm: Stopping Short



- ❖ Would this have been different if we only wanted:
  - The path from A to G?
  - The path from A to D?

Order Added to Known Set:

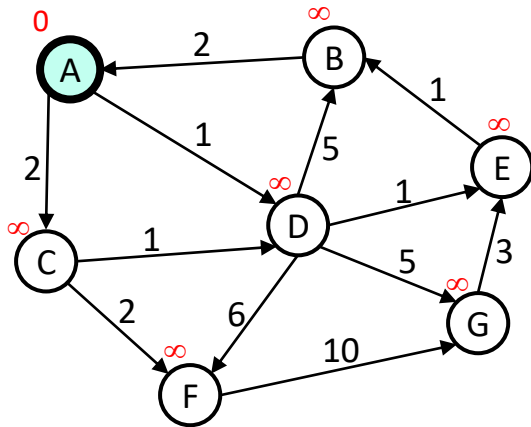
A, C, B, D, F, H, G, E

Vertex	Known?	Distance	Previous
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

# Review: Important Features

- ❖ Once a vertex is marked known, its shortest path is known
  - Can reconstruct path by following back-pointers (“previous” fields)
- ❖ While a vertex is not known, another shorter path might be found
- ❖ The “Order Added to Known Set” is unimportant
  - 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-distance; ties are resolved “somehow”

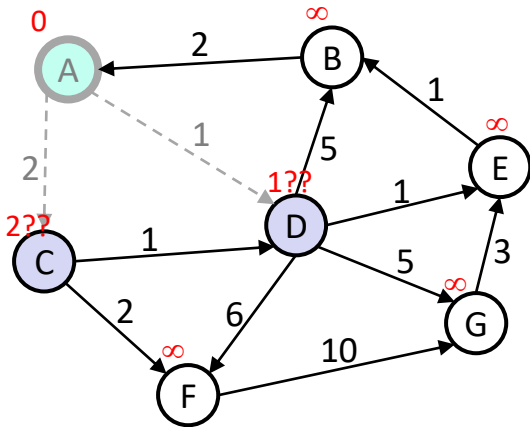
## Dijkstra's Algorithm: Example #2



Order Added to Known Set:

Vertex	Known?	Distance	Previous
A		0	
B		$\infty$	
C		$\infty$	
D		$\infty$	
E		$\infty$	
F		$\infty$	
G		$\infty$	

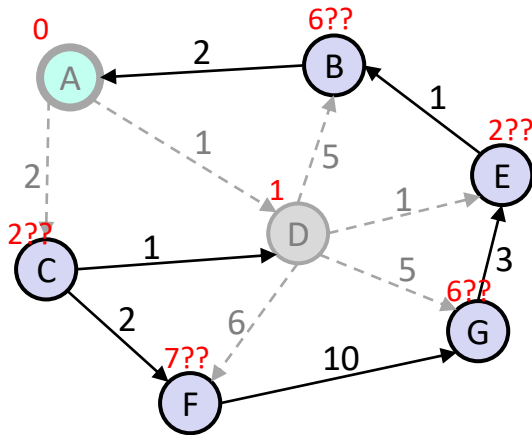
# Dijkstra's Algorithm: Example #2



Order Added to Known Set:  
A

Vertex	Known?	Distance	Previous
A	Y	0	/
B		$\infty$	
C		$\leq 2$	A
D		$\leq 1$	A
E		$\infty$	
F		$\infty$	
G		$\infty$	

# Dijkstra's Algorithm: Example #2

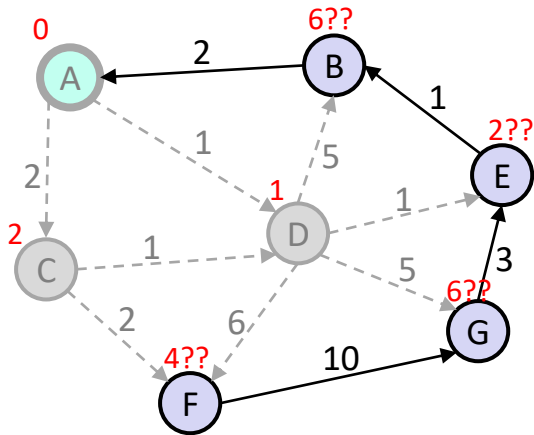


Order Added to Known Set:

A, D

Vertex	Known?	Distance	Previous
A	Y	0	/
B		$\leq 6$	D
C		$\leq 2$	A
D	Y	1	A
E		$\leq 2$	D
F		$\leq 7$	D
G		$\leq 6$	D

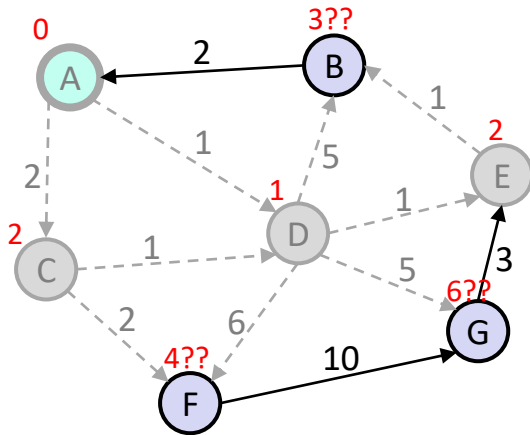
# Dijkstra's Algorithm: Example #2



Order Added to Known Set:  
A, D, C

Vertex	Known?	Distance	Previous
A	Y	0	/
B		≤ 6	D
C	Y	2	A
D	Y	1	A
E		≤ 2	D
F		≤ 4	C
G		≤ 6	D

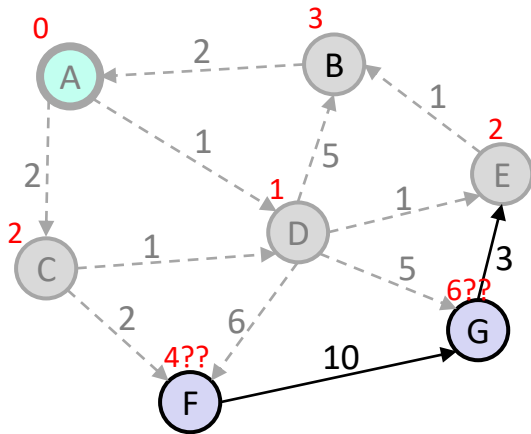
# Dijkstra's Algorithm: Example #2



Order Added to Known Set:  
A, D, C, E

Vertex	Known?	Distance	Previous
A	Y	0	/
B		$\leq 3$	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F		$\leq 4$	C
G		$\leq 6$	D

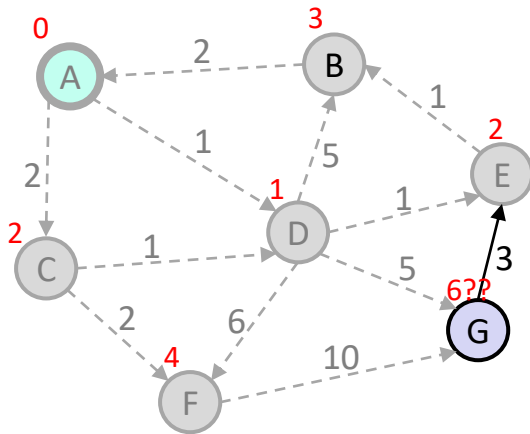
# Dijkstra's Algorithm: Example #2



Order Added to Known Set:  
A, D, C, E, B

Vertex	Known?	Distance	Previous
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

# Dijkstra's Algorithm: Example #2

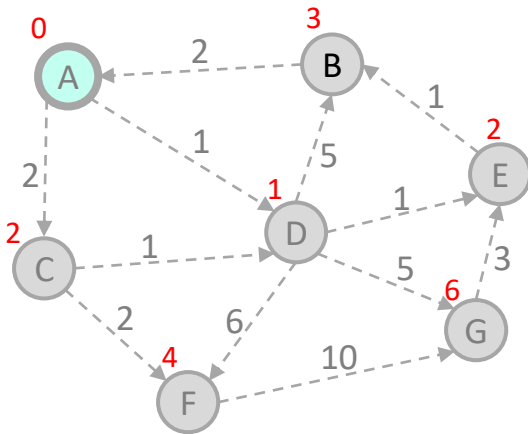


Order Added to Known Set:

A, D, C, E, B, F

Vertex	Known?	Distance	Previous
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

# Dijkstra's Algorithm: Example #2



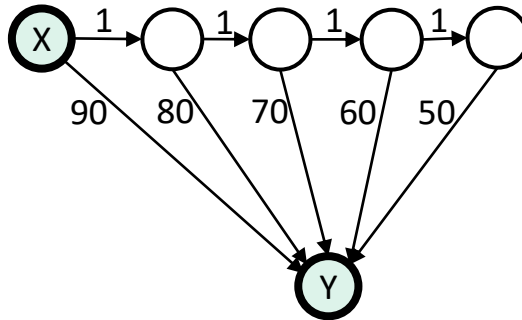
🐷🐷 WOOHOO!!! 🐷🐷

Order Added to Known Set:

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

Vertex	Known?	Distance	Previous
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

## Dijkstra's Algorithm: Example #3



- ❖ How will the best-pathlen-so-far for Y proceed?
  - *90, 81, 72, 63, 54, ...*
- ❖ Is this expensive?
  - *No, each edge is processed only once*

# Lecture Outline

- ❖ Traversals
  - Trees and Graphs: Level-order / Breadth-first
  - Graphs: Depth-first
  - Conclusion
  
- ❖ Shortest Paths!
  
- ❖ Dijkstra's Algorithm
  - Introduction
  - **Correctness Proof**
  - Runtime

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-12A](https://tinyurl.com/332-08-12A)*

# A Greedy Algorithm

- ❖ Dijkstra's Algorithm
  - Single-source shortest paths in a weighted graph (directed or undirected) with no negative-weight edges
  
- ❖ Dijkstra's is an example of a *greedy algorithm*:
  - At each step, *irrevocably* does what seems best *at that step*
    - Makes locally optimal decision; decision isn't necessarily globally optimal
  - Once a vertex is known, it is not revisited
    - Turns out, the decision is globally optimal!

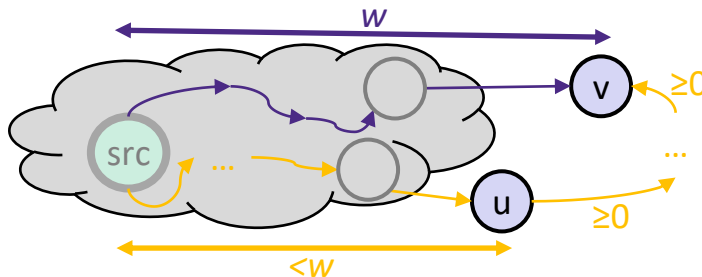
# Where Are We?

- ❖ What should we do after learning an algorithm?
- ❖ Prove it is correct
  - Not obvious!
  - We will sketch the key ideas
- ❖ Analyze its efficiency
  - And improve it by using a data structure we learned earlier!

# Correctness: Intuition

- ❖ *Statement*: all “known” vertices have the correct shortest path
  - True initially: shortest path to start vertex has cost 0
  - If the new vertex marked “known” also has the correct shortest path, then by induction this statement holds
  - Thus, when the algorithm terminates (ie, everything is “known”), we will have the correct shortest path to every vertex
- ❖ *Key fact we need*: when we mark a vertex “known”, we won’t discover a shorter path later!
  - This holds only because Dijkstra’s algorithm picks the vertex with the next shortest path-so-far
  - The proof of this fact is by contradiction ...

# Correctness: Rough Idea



- ❖ Let  $v$  be the next vertex marked known (“added to the cloud”)
  - The *best-known path* to  $v$  only contains nodes “in the cloud” and has weight  $w$ 
    - (we used Dijkstra’s to select this path, and we only know about paths through the cloud to a vertex in the fringe)
  - Assume the *actual shortest path* to  $v$  is different
    - It must use at least one non-cloud vertex (*otherwise we’d know about it*)
    - Let  $u$  be the *first* non-cloud vertex on this path
    - The path weight from  $u$  to  $v$  –  $\text{weight}(u, v)$  – must be  $\geq 0$  (*no negative weights*)
    - Thus, the total weight of the path from  $\text{src}$  to  $u$  must be  $< w$  (*otherwise  $\text{weight}(\text{src}, u) + \text{weight}(u, v) > w$  and this path wouldn’t be shorter*)
    - But if  $\text{weight}(\text{src}, u) < w$ , then  $v$  would not have been picked

**CONTRADICTION!!!**

# Lecture Outline

- ❖ Traversals
  - Trees and Graphs: Level-order / Breadth-first
  - Graphs: Depth-first
  - Conclusion
  
- ❖ Shortest Paths!
  
- ❖ Dijkstra's Algorithm
  - Introduction
  - Correctness Proof
  - **Runtime**

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-12A](https://tinyurl.com/332-08-12A)*

# Runtime, First Approach

```
dijkstra(Graph g, Vertex start) {  
  foreach vertex v in g:  
    v.distance = ∞  
    v.known = false  
  start.distance = 0  
  
  while there are vertices in g  
  that are not known:  
    select vertex v with lowest cost  
    v.known = true  
    foreach unknown edge (v, u) in g:  
      d1 = v.distance + g.weight(v, u)  
      d2 = u.distance  
      if (d1 < d2)  
        u.distance = d1  
        u.previous = v  
}
```

}  $O(|V|)$

}  $O(|V|^2)$

}  $O(|E|)$   
*(notice each edge is  
processed only once)*

**Total:**  $O(|V|^2 + |E|)$

# Improving Asymptotic Runtime

- ❖ *Current runtime:*  $O(|V|^2 + |E|) \in O(|V|^2)$
  
- ❖ We had a similar “problem” with toposort being  $O(|V|^2 + |E|)$ 
  - Caused by each iteration looking for the next vertex to process
  - Solved it with a queue of zero-degree vertex!
  - But here we need:
    - The lowest-cost vertex
    - Ability to change costs, since they can change as we process edges
  
- ❖ Solution?
  - A priority queue holding all unknown vertex sorted by cost
  - Must support **decreaseKey** operation
    - Conceptually simple, but a pain to code up

# Runtime, Second Approach

```

dijkstra(Graph g, Vertex start) {
  foreach vertex v in g:
    v.distance = ∞
  start.distance = 0
  heap = buildHeap(g.vertices)

  while (! heap.empty()):
    v = heap.deleteMin()
    foreach unknown edge (v, u) in g:
      d1 = v.distance + g.weight(v, u)
      d2 = u.distance
      if (d1 < d2)
        heap.decreaseKey(u, d1)
        u.previous = v
}

```

}  $O(|V|)$

}  $O(|V|)$

}  $O(|V| \log |V|)$

}  $O(|E|)$   
*(each edge processed once)*

}  $O(|E| \log |V|)$   
*( $|E|$  decreaseKey() calls)*

Total:  $O(|V| \log |V| + |E| \log |V|)$  60

# Runtime as a Function of Density

- ❖ *First approach (linear scan):*  $O(|V|^2 + |E|)$
- ❖ *Second approach (heap):*  $O(|V|\log|V| + |E|\log|V|)$
  
- ❖ So which is better?
  - In a sparse graph,  $|E| \in O(|V|)$ 
    - So second approach (heap) is better?  $O(|E|\log|V|)$
  - In a dense graph,  $|E| \in \Theta(|V|^2)$ 
    - So first approach (linear scan) is better?  $O(|E|)$
  
- ❖ But: remember these are worst-case and asymptotic
  - Heap might have worse constant factors
  - Maybe `decreaseKey` is cheap, making  $|E|\log|V|$  more like  $|E|$ 
    - It's called rarely, or vertices don't percolate far