Dijkstra & Shortest Paths

CSE 332 Summer 2021

Instructor: Kristofer Wong

Teaching Assistants:

Alena Dickmann	Arya GJ	Finn Johnson
Joon Chong	Kimi Locke	Peyton Rapo
Rahul Misal	Winston Jodjana	

Announcements

- Regrades

- Resume review & guest lecture next week

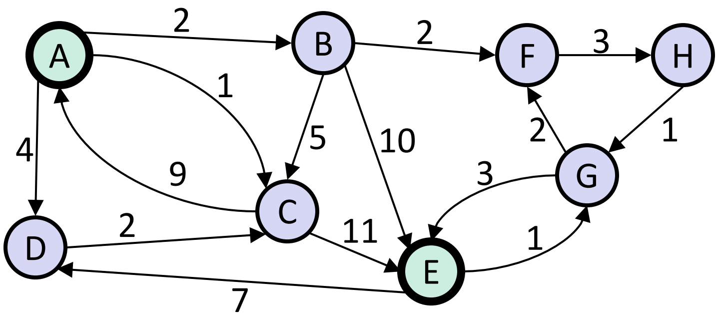
Lecture Outline

- **Shortest Paths!**

- Dijkstra's Algorithm
  - Introduction
  - Correctness Proof
  - Runtime

Exercise (No Gradescope today!)

- Without using any formal algorithms, find the shortest path from A to E
  - … assuming this graph is unweighted
  - … assuming this graph is weighted



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*
  - Worst-case runtime for single-destination is no faster than worst-case runtime for all-destinations
  - Still $O(|E|+|V|)$

```
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)
}
```

Shortest Path: Applications

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

Wait, these are all <u>weighted</u> graphs!

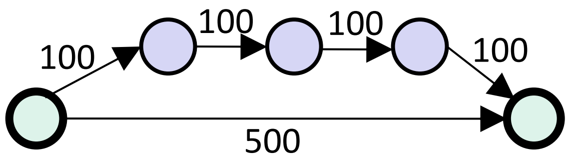
Single-Source Shortest Paths … *for Weighted Graphs*

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

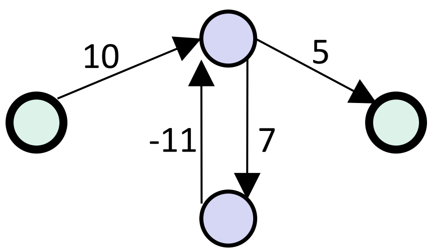
- As before:
  - All-destinations is asymptotically no harder than single-destination
- Unlike before:
  - BFS will not work

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

- Shortest Paths!

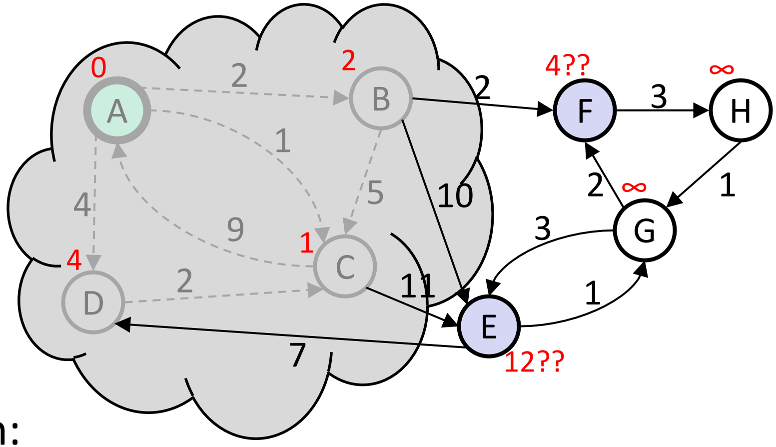
- Dijkstra's Algorithm
  - **Introduction**
  - Correctness Proof
  - Runtime

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 ∞
- 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

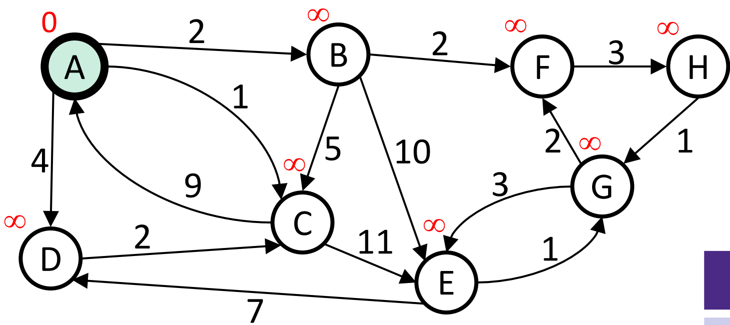
```
dijkstra(Graph g, Vertex start) {
  foreach vertex v in g:
    v.distance = ∞
    v.known = false
  start.distance = 0

  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

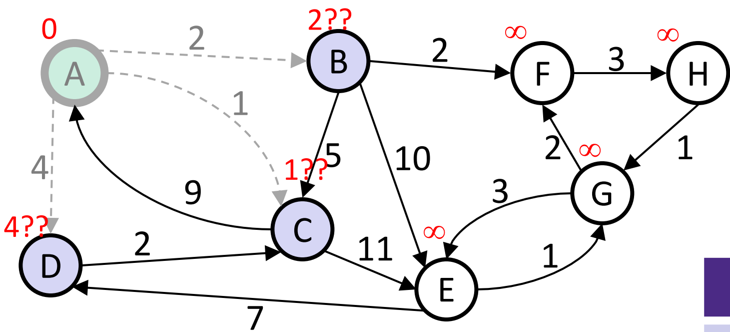
Dijkstra's Algorithm: Example #1



Order Added to Known Set:

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

Dijkstra's Algorithm: Example #1

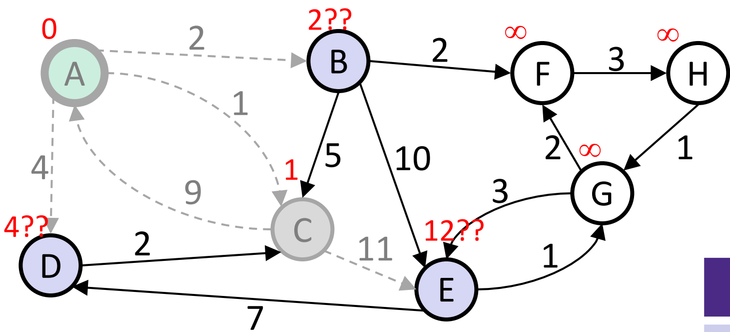


Vertex	Known?	Distance	Previous
A	Y	0	/
B		**≤ 2**	**A**
C		**≤ 1**	**A**
D		**≤ 4**	**A**
E		∞	
F		∞	
G		∞	
H		∞	

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

A

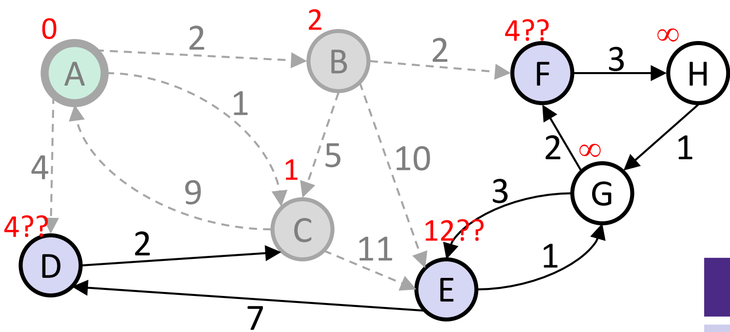
Dijkstra's Algorithm: Example #1



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		∞	

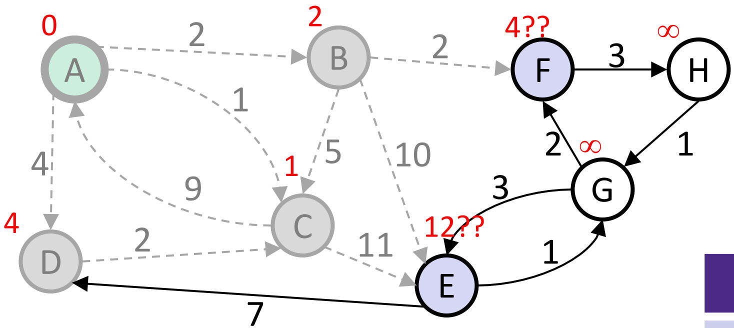
Dijkstra's Algorithm: Example #1



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

Order Added to Known Set:
A, C, B

Dijkstra's Algorithm: Example #1

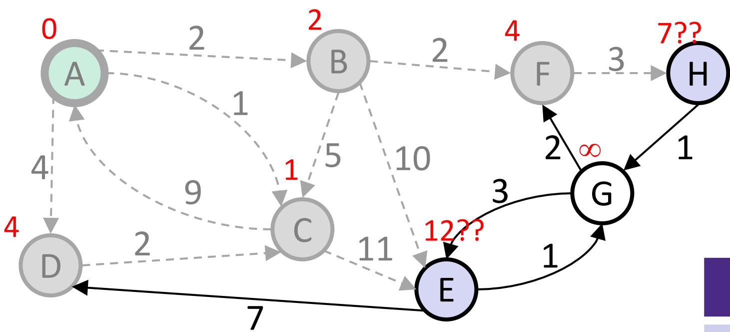


Vertex	Known?	Distance	Previous
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F		≤ 4	B
G		∞	
H		∞	

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

A, C, B, D

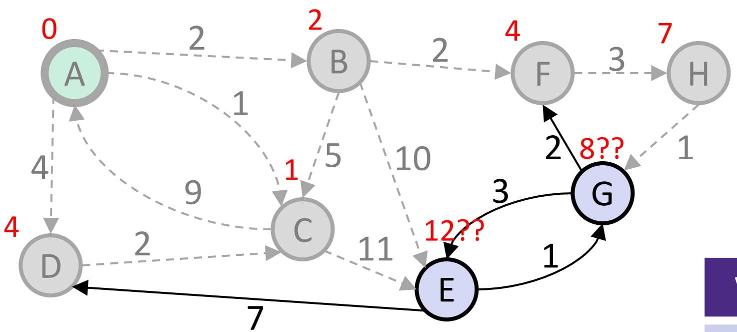
Dijkstra's Algorithm: Example #1



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**	**F**

Dijkstra's Algorithm: Example #1

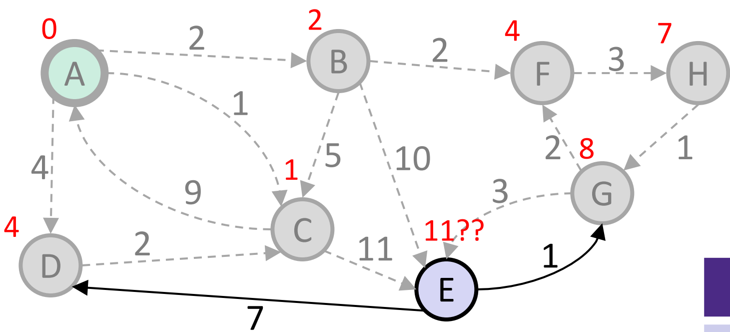


Vertex	Known?	Distance	Previous
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		**≤ 8**	**H**
H	Y	7	F

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

A, C, B, D, F, H

Dijkstra's Algorithm: Example #1

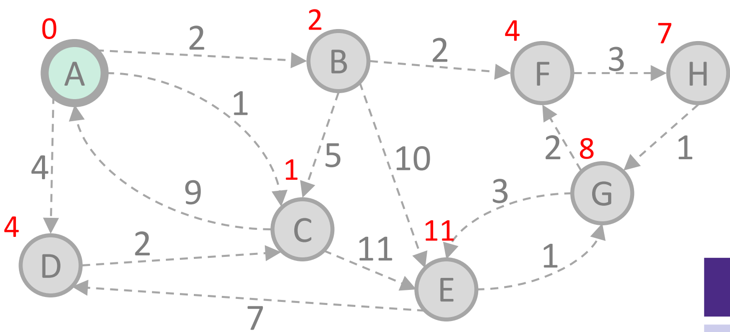


Order Added to Known Set:

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	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Dijkstra's Algorithm: Example #1

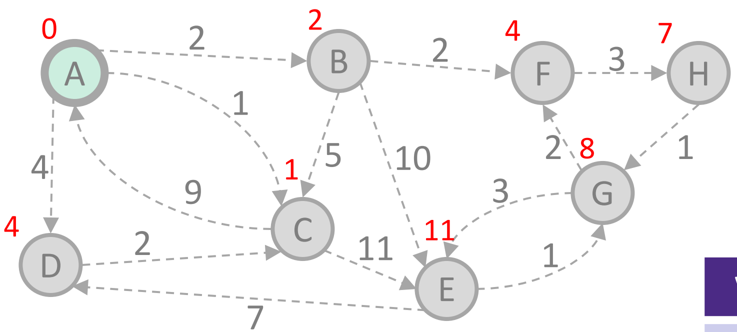


TADA!!!

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

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

Dijkstra's Algorithm: Interpreting the Results

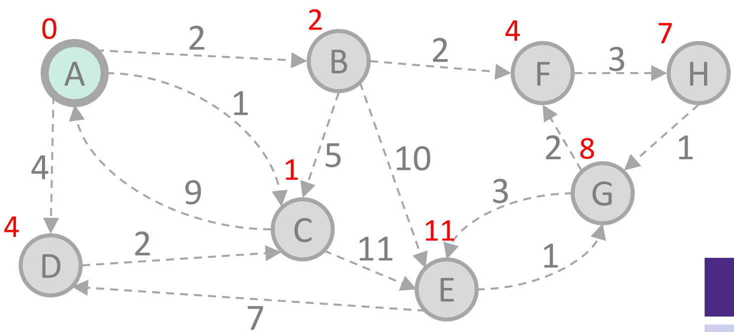


- Now that we're done, how do we get the path from A to E?

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

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

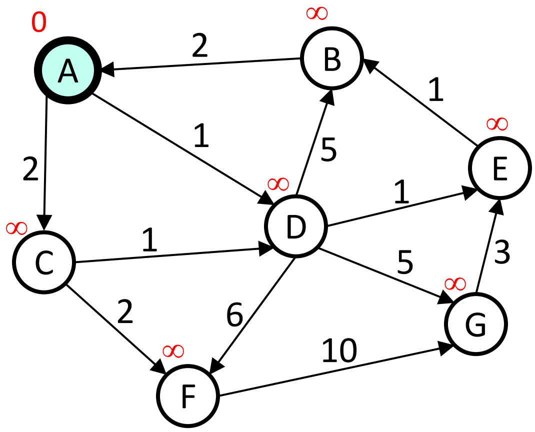
Review: Important Features

- 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"

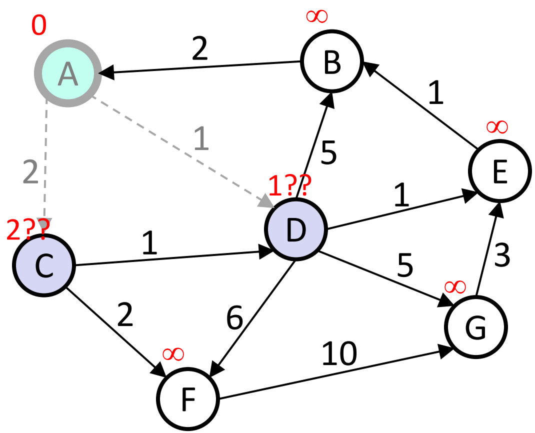
Dijkstra's Algorithm: Example #2



Order Added to Known Set:

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

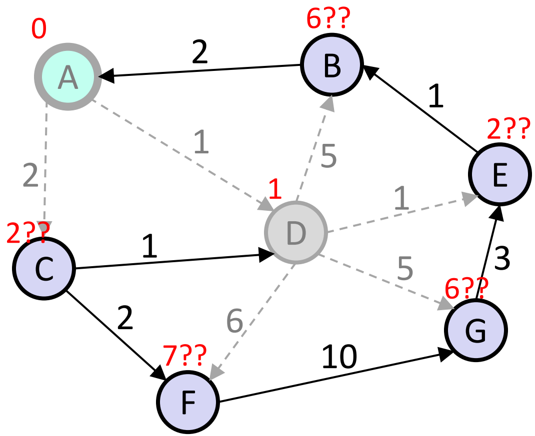
Dijkstra's Algorithm: Example #2



Order Added to Known Set:
A

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

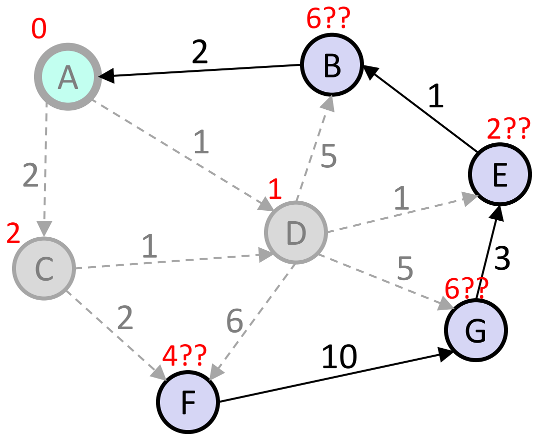
Dijkstra's Algorithm: Example #2



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**

Order Added to Known Set:
A, D

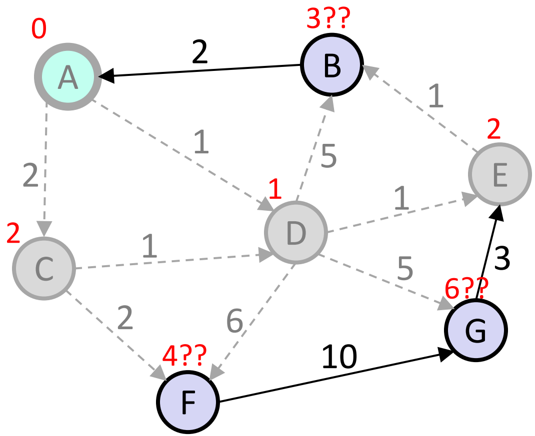
Dijkstra’s Algorithm: Example #2



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

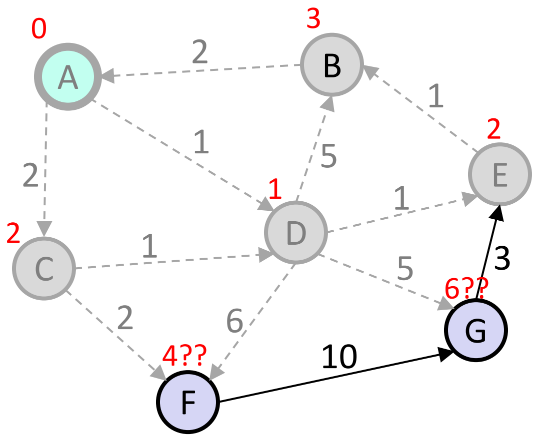
Dijkstra's Algorithm: Example #2



Vertex	Known?	Distance	Previous
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

Dijkstra's Algorithm: Example #2

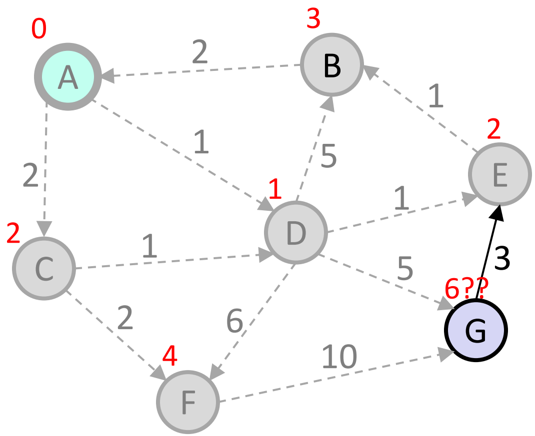


Order Added to Known Set:

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

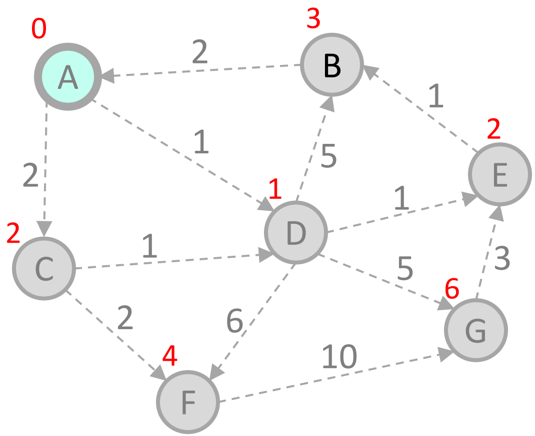
Dijkstra's Algorithm: Example #2



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

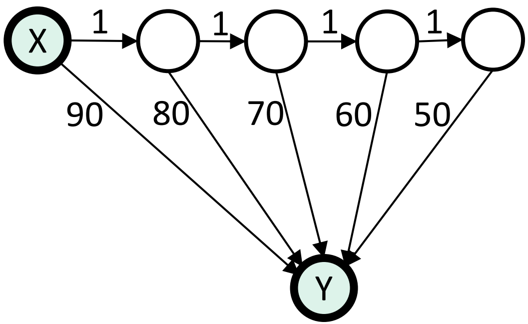


🐷🐷 WOOHOO!!! 🐷🐷

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

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

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

- Shortest Paths!

- Dijkstra's Algorithm
  - Introduction
  - **Correctness Proof**
  - Runtime

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

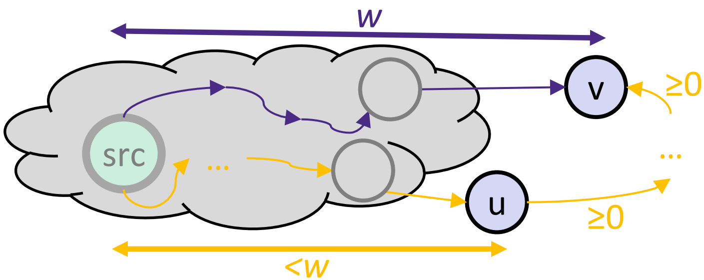
- Analyze its efficiency
  - And improve it by using a data structure we learned earlier!

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 – `weight(u, v)` – must be ≥0 *(no negative weights)*
    - Thus, the total weight of the path from src to u must be <w *(otherwise* `weight(src,u) + weight(u,v) > w` *and this path wouldn't be shorter)*
    - But if `weight(src,u) < w`, then v would not have been picked

CONTRADICTION!!!

Lecture Outline

- Shortest Paths!
- Dijkstra's Algorithm
  - Introduction
  - Correctness Proof
  - **Runtime**

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|)

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| ∈ O(|V|)
    - So second approach (heap) is better? O(|E|log|V|)
  - In a dense graph, |E| ∈ Θ(|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