Dijkstra's Algorithm (cont.); Minimum Spanning Trees

CSE 332 Spring 2021

Instructor: Hannah C. Tang

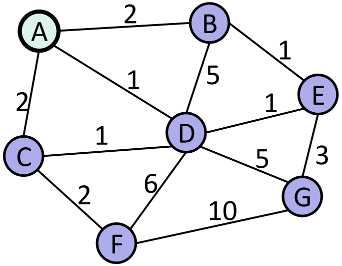
Teaching Assistants:

Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

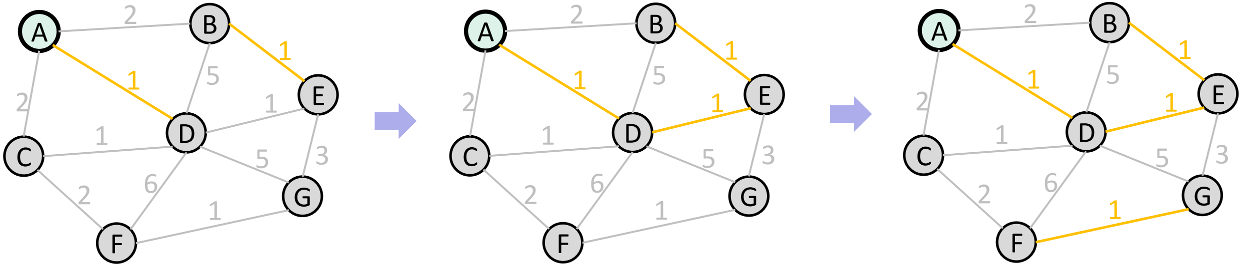
gradescope

gradescope.com/courses/256241

- Run Dijkstra's on this graph
  - (this is example #2 from the previous lecture)



- Consider the following thought experiment:
  - Dijkstra's iterates over the *unknown vertices*, adding them to the "*known cloud*"
  - You decide to invent your own shortest-paths algorithm that iterates over *unknown edges* and adds them *if they don't create a cycle*
  - Does your algorithm correctly find all the shortest paths from the start vertex?

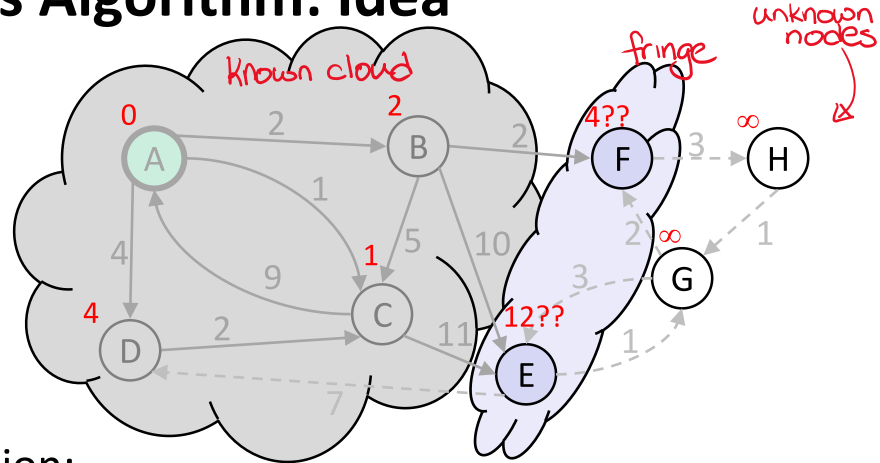


Lecture Outline

- Dijkstra's Algorithm
  - **Review**
  - For Reading: Correctness and Runtime

- Minimum Spanning Tree
  - Introduction
  - Prim's Algorithm
  - Kruskal's Algorithm, sorta

Dijkstra's Algorithm: Idea

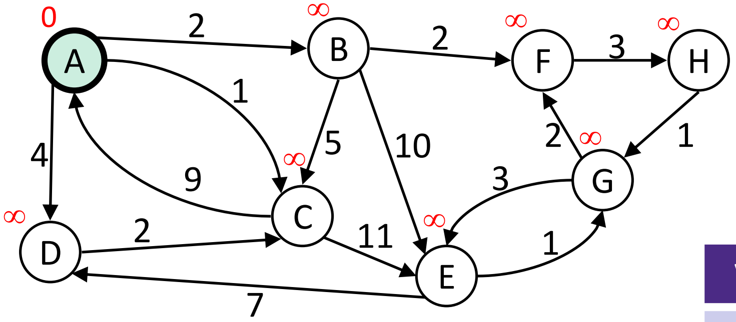


- Initialization:
  - Start vertex has distance **0**; all other vertices have distance ∞

1. Initialize aux data structure
2. Have vertices in data struct?
    3. Get vertex from data struct
    4. Visit/process vertex
    5. Update vertex's neighbors

- At each step:
  - Pick closest unknown vertex v
  - Add it to the "cloud" of known vertices
  - Update distances for vertices with edges from v

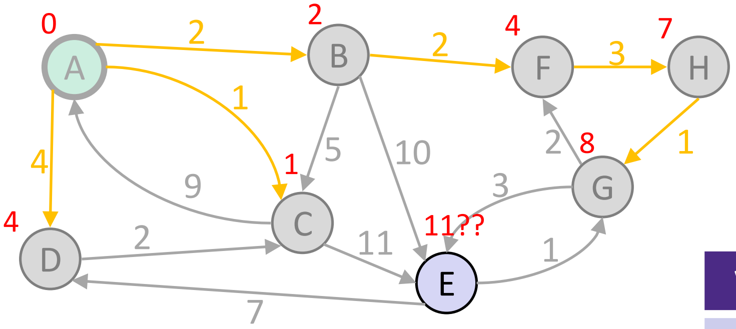
Dijkstra's Algorithm: Example #1



Order Added to Known Set:

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

Dijkstra's Algorithm: Interpreting the Results

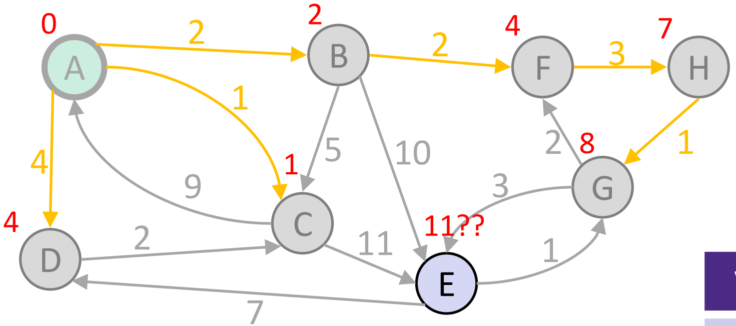


- Now that we're done, how do we get the path from A to E?

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?

<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

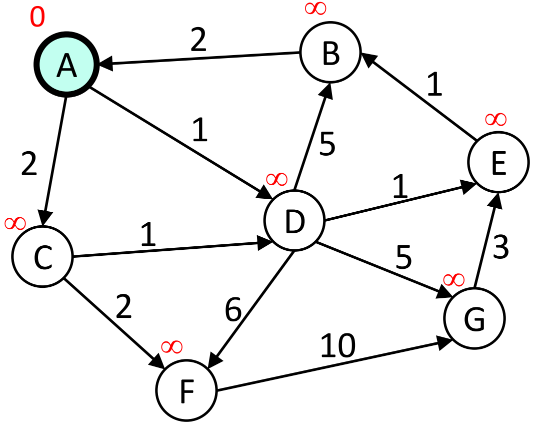
- The "Order Added to Known Set" is unimportant
  - A detail about how the algorithm works *(client doesn't care)*
  - Not used by the algorithm *(implementation doesn't care)*
  - It is sorted by path-distance; ties are resolved "somehow"

Dijkstra’s is Greedy

- 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!

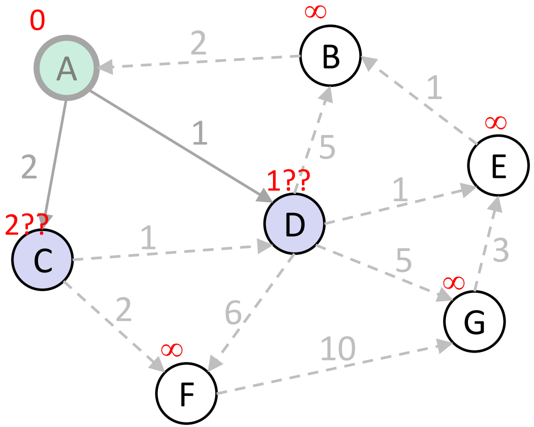
Dijkstra’s Algorithm: Example #2



Order Added to Known Set:

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

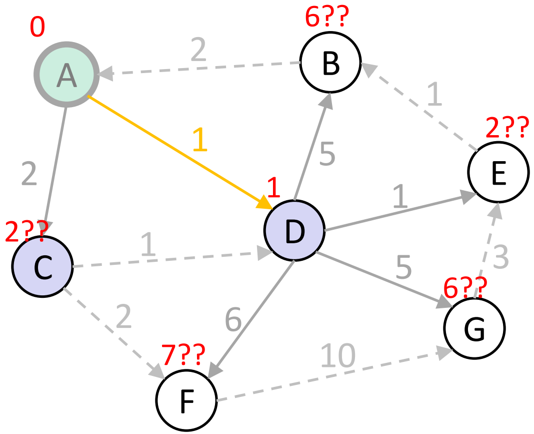
Dijkstra's Algorithm: Example #2



Order Added to Known Set:
A

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

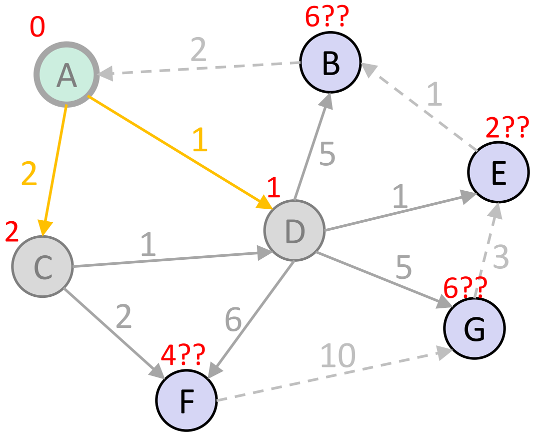
Dijkstra's Algorithm: Example #2



Order Added to Known Set:
A, D

Vertex	Known?	Distance	Previous
A	Y	0	/
B		**≤ 6**	**D**
C		≤ 2	A
D	Y	1	A
E		**≤ 2**	**D**
F		**≤ 7**	**D**
G		**≤ 6**	**D**

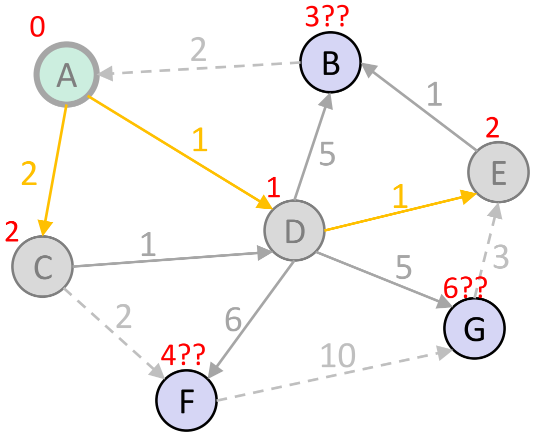
Dijkstra's Algorithm: Example #2



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

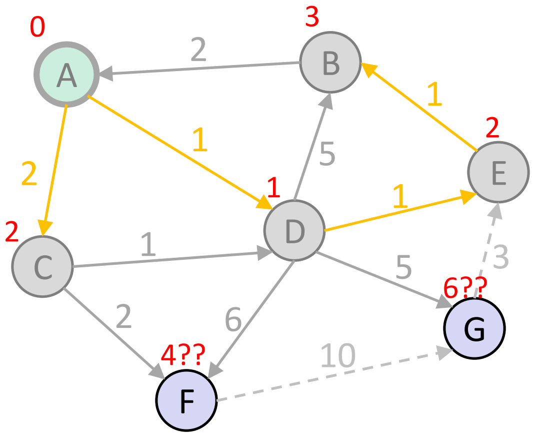
Dijkstra’s Algorithm: Example #2



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

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

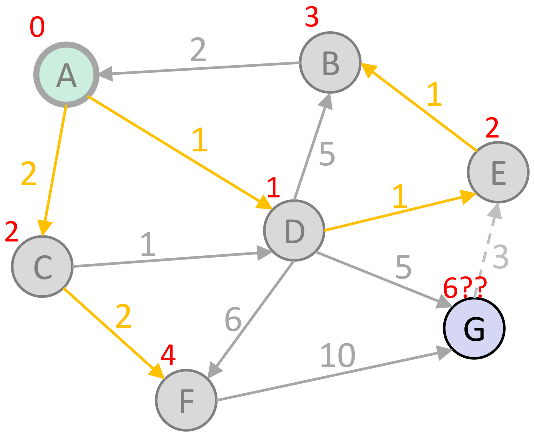
Dijkstra's Algorithm: Example #2



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

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

Dijkstra's Algorithm: Example #2

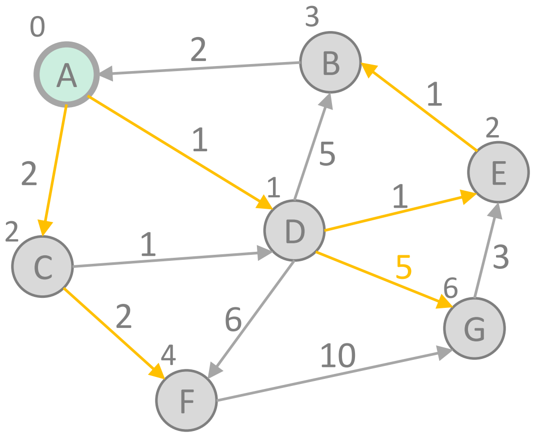


Order Added to Known Set:

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

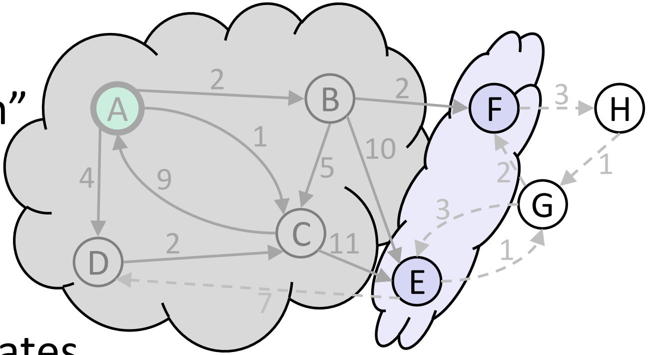
Lecture Outline

- Dijkstra’s Algorithm
    - Review
    - **For Reading: Correctness and Runtime**

- Minimum Spanning Tree
    - Introduction
    - Prim’s Algorithm
    - Kruskal’s Algorithm, sorta

Correctness: Intuition (1 of 2)

- *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

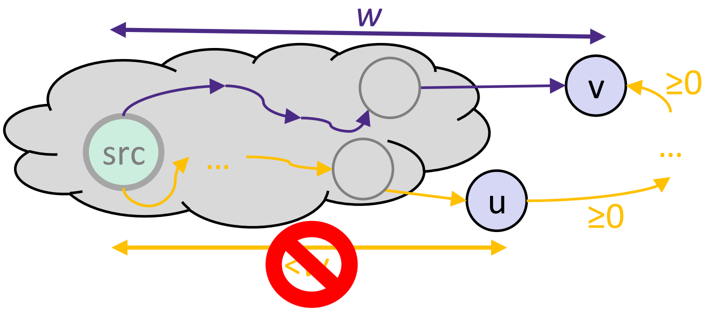


Correctness: Intuition (2 of 2)

- *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* …

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

  while there are unknown vertices:
    v = lowest cost unknown vertex
    v.known = true
    foreach unknown v.neighbor
    with weight w:
      d1 = v.distance + w
      d2 = u.distance
      if (d1 < d2):
        u.distance = d1
        u.previous = v
}
```

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!!!

Runtime, First Approach

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

  while there are unknown vertices:
    v = lowest cost unknown vertex
    v.known = true
    foreach unknown v.neighbor
    with weight w:
      d1 = v.distance + w
      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 vertices!
  - But here we need:
    - The lowest-cost vertex
    - Ability to change costs, since they can change as we process edges

- Solution?
  - A priority queue of unknown vertices, using distance-from-src as priority
  - Must support **`decreaseKey`** operation find + percolate up: $O(\log V)$
    - 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 v.neighbor
    with weight w:
      d1 = v.distance + w
      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

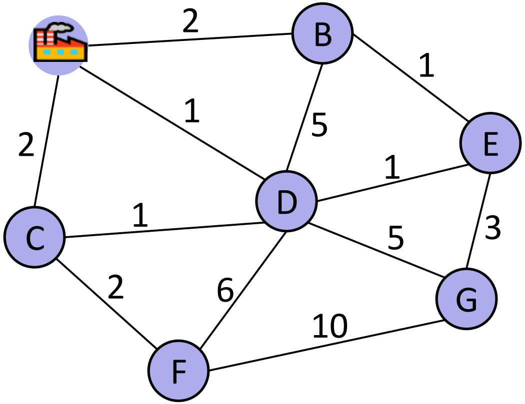
Lecture Outline

- Dijkstra’s Algorithm
  - Review
  - For Reading: Correctness and Runtime

- Minimum Spanning Tree
  - **Introduction**
  - Prim’s Algorithm
  - Kruskal’s Algorithm, sorta

Problem Statement

- Your friend at the electric company needs to connect all these cities to the power plant
- She knows the cost to lay wires between any pair of cities and wants the cheapest way to ensure electricity gets to every city

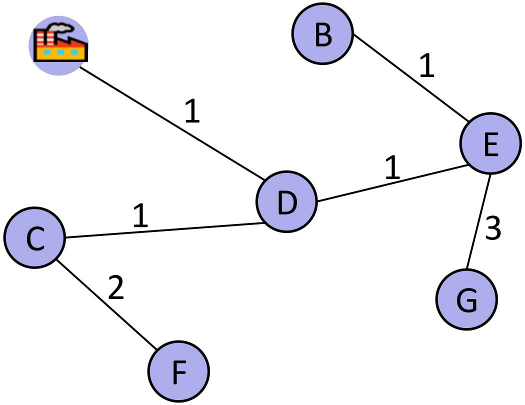


- Assume:
  - The graph is connected and undirected
  - *(In general, edge weights can be negative; just not in this example)*

Solution Statement

- We need a set of edges such that:
  - Every vertex touches at least one edge (“the edges **span** the graph”)
  - The graph using just those edges is **connected**
  - The total weight of these edges is **minimized**

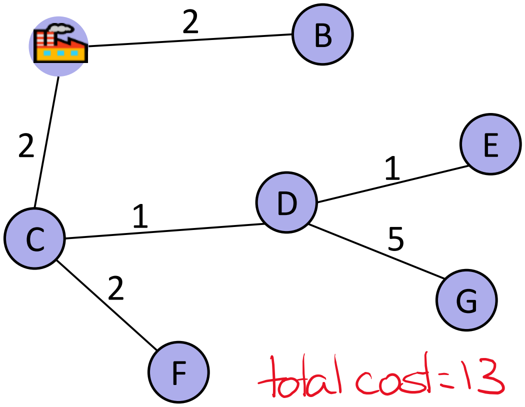
- *Claim*: The set of edges we pick never forms a cycle. Why?
  - V-1 edges is the exact number of edges to connect all vertices
  - Taking away 1 edge breaks connectiveness
  - Adding 1 edge makes a cycle



Solution Statement (v2)

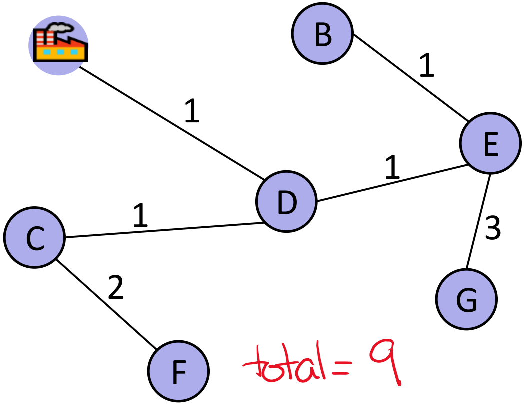
- We need a ~~set of edges such that~~ Minimum Spanning Tree:
  - Every vertex touches at least one edge ("the edges **span** the graph")
  - The graph using just those edges is **connected**
  - The total weight of these edges is **minimized**

spanning tree



total cost = 13

minimum spanning tree



total = 9

Minimum Spanning Trees

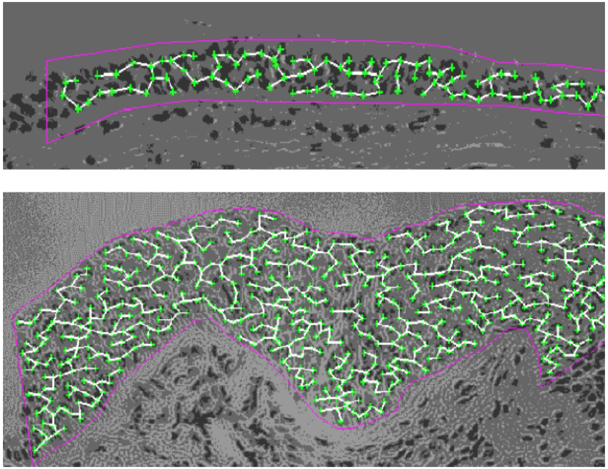
- Given an undirected graph G = (V,E), a minimum spanning tree is a graph G' = (V, E') such that:
  - E' is a subset of E
  - |E'| = |V| - 1
  - G' is connected
  - $\sum_{(u,v)\in E'} c_{uv}$ is minimal

Applications of MSTs

- Handwriting recognition
  - http://dspace.mit.edu/bitstream/handle/1721.1/16727/43551593-MIT.pdf;sequence=2

Figure 4-3: A typical minimum spanning tree

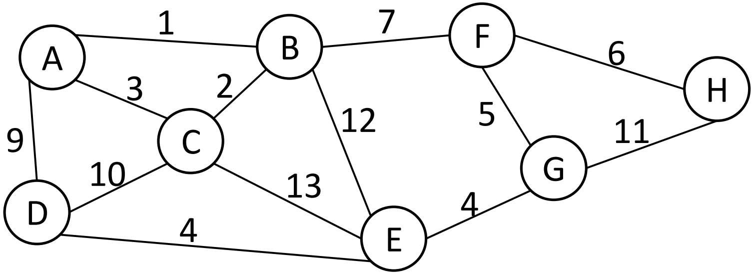
- Medical imaging
  - e.g. arrangement of nuclei in cancer cells



For more, see: http://www.ics.uci.edu/~eppstein/gina/mst.html gradescope

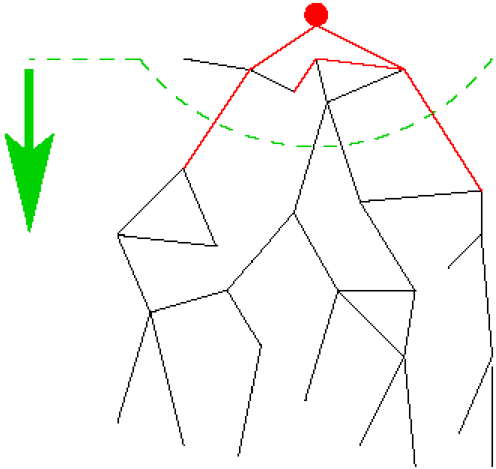
gradescope.com/courses/256241

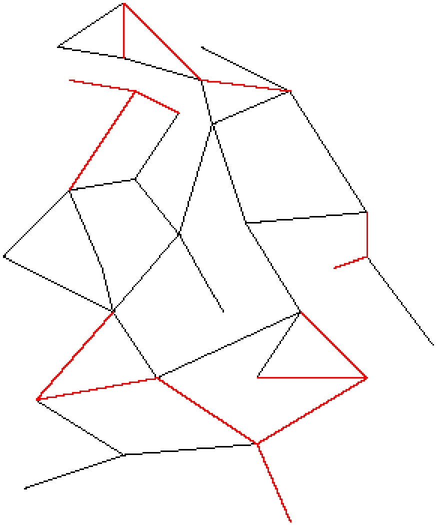
- Find the MST for this graph:



- Important: pay attention to your MST-finding process. What are you iterating through (eg: vertices? edges?)? In what order are you iterating through those objects?

MST Algorithms: Two Different Approaches





Prim's Algorithm

Almost identical to Dijkstra's
Start with one node, grow greedily

Kruskals's Algorithm

Completely different!
Start with a *forest* of MSTs, union them together
(Need a new data structure for this)

Lecture Outline

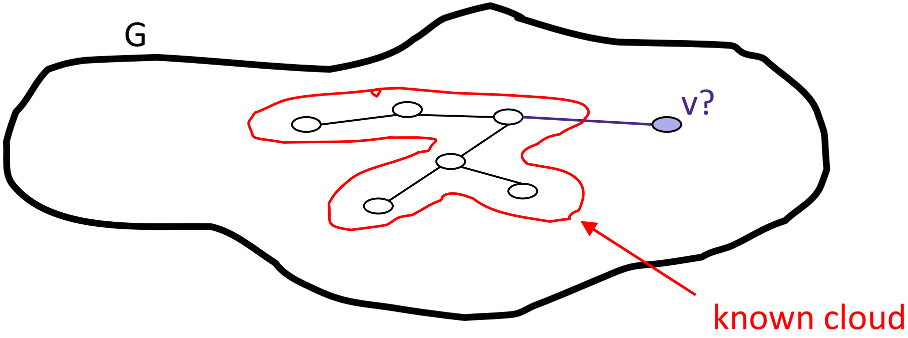
- Dijkstra's Algorithm
  - Review
  - For Reading: Correctness and Runtime

- Minimum Spanning Tree
  - Introduction
  - **Prim's Algorithm**
  - Kruskal's Algorithm, sorta

Prim's Algorithm**

- *Intuition*: a vertex-based greedy algorithm
    - Builds MST by greedily adding vertices

- *Summary*: Grow a single tree by picking a vertex from the fringe that has the smallest cost
    - Unlike Dijkstra's, cost is the *edge weight* into the known set



*** This algorithm was developed in 1930 by Votěch Jarník, then independently rediscovered by Robert Prim in 1957 and then Dijkstra in 1959. It's also known as Jarník's, Prim-Jarník, or DJP*

Prim's Algorithm: Pseudocode

```
prim(Graph g) {
  foreach vertex v in g:
    v.distance = ∞
  start = g.getSomeArbitraryVertex()
  start.distance = 0
  heap = buildHeap(g.vertices)

  mst = {}
  while (! heap.empty()):
    v = heap.deleteMin()
    if mst.hasVertex(v):
      continue;
    mst.addEdge(v, v.previous)
    foreach edge (v, u) in v.neighbors():
      d1 = v.distance
      d2 = u.distance
      if (d1 < d2):
        heap.decreaseKey(u, d1)
        u.previous = v
}
```

1. Initialize aux data structure
2. Have vertices in data struct?
   3. Get vertex from data struct
   4. Visit/process vertex
   5. Update vertex's neighbors

Prim's Algorithm vs. Dijkstra's Algorithm (1 of 2)

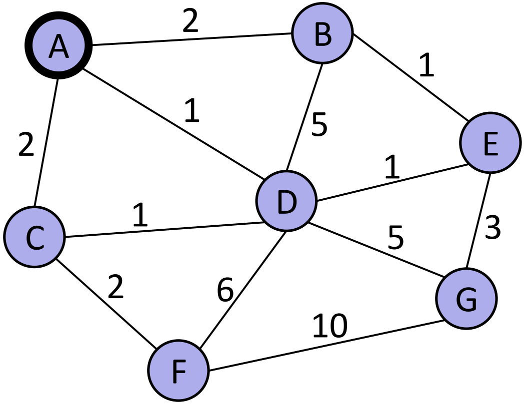
- Dijkstra's picks an unknown vertex with smallest *distance to the source*
  - ie, path weights
- Prim's picks an unknown vertex with smallest *distance to the known set*
  - i.e., edge weights
- Some differences in the initialization, but otherwise identical

Prim’s Algorithm vs. Dijkstra’s Algorithm (2 of 2)

```
prim(Graph g) {
  foreach vertex v in g:
    v.distance = ∞
  start = g.someArbitraryVertex()
  start.distance = 0
  heap = buildHeap(g.vertices)

  mst = {}
  while (! heap.empty()):
    v = heap.deleteMin()
    if mst.hasVertex(v):
      continue;
    mst.addEdge(v, v.previous)
    foreach edge (v, u)
    in v.neighbors():
      d1 = v.distance
      d2 = u.distance
      if (d1 < d2):
        heap.decreaseKey(u, d1)
        u.previous = v
}
```

```
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 v.neighbor
    with weight w:
      d1 = v.distance + w
      d2 = u.distance
      if (d1 < d2):
        heap.decreaseKey(u, d1)
        u.previous = v
}
```

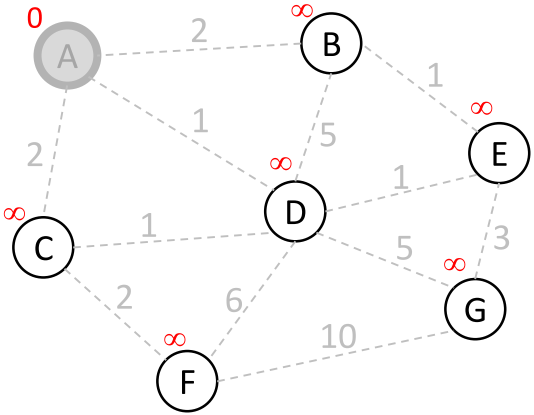
Prim's Algorithm: Example



Order Added to Known Set:

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

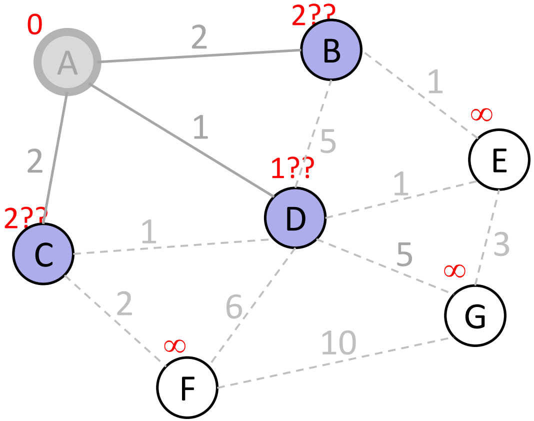
Prim's Algorithm: Example



Order Added to Known Set:
A

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

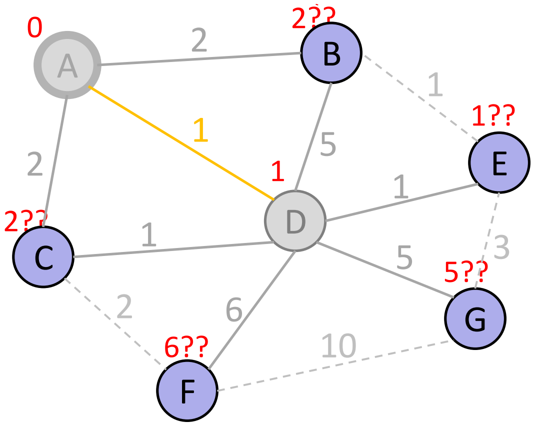
Prim’s Algorithm: Example



Order Added to Known Set:
A

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

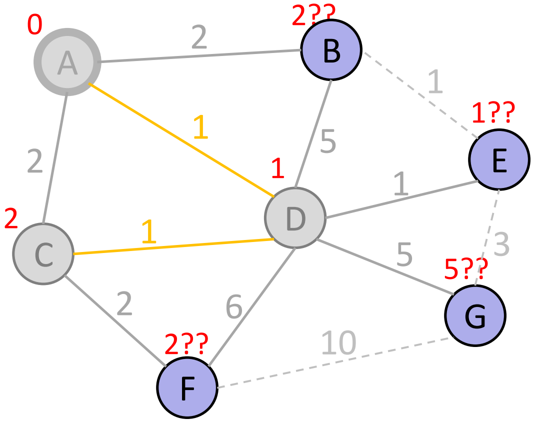
Prim's Algorithm: Example



Vertex	Known?	Distance	Previous
A	Y	0	\
B		2	A
C		1	D
D	Y	1	A
E		1	D
F		6	D
G		5	D

Order Added to Known Set:
A, D

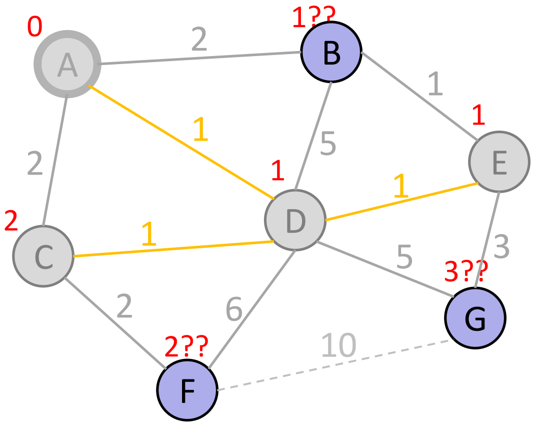
Prim's Algorithm: Example



Order Added to Known Set:
A, D, C

Vertex	Known?	Distance	Previous
A	Y	0	\
B		2	A
C	Y	1	D
D	Y	1	A
E		1	D
F		2	C
G		5	D

Prim's Algorithm: Example

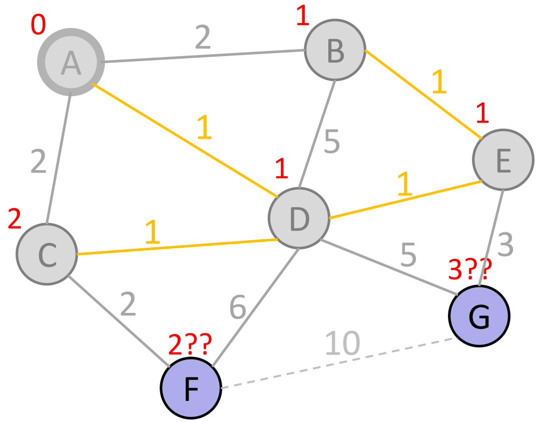


Order Added to Known Set:

A, D, C, E

Vertex	Known?	Distance	Previous
A	Y	0	\
B		**1**	**E**
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		**3**	**E**

Prim’s Algorithm: Example

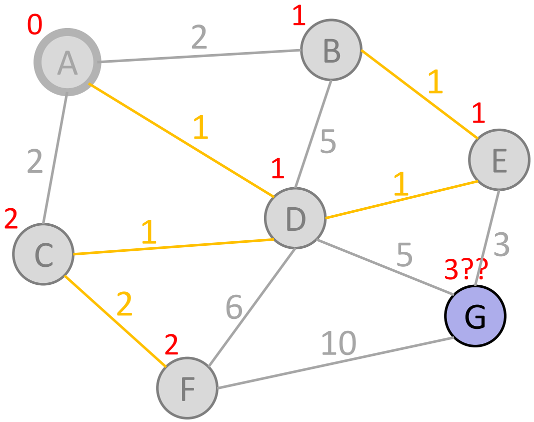


Order Added to Known Set:

A, D, C, E, B

Vertex	Known?	Distance	Previous
A	Y	0	\
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

Prim's Algorithm: Example

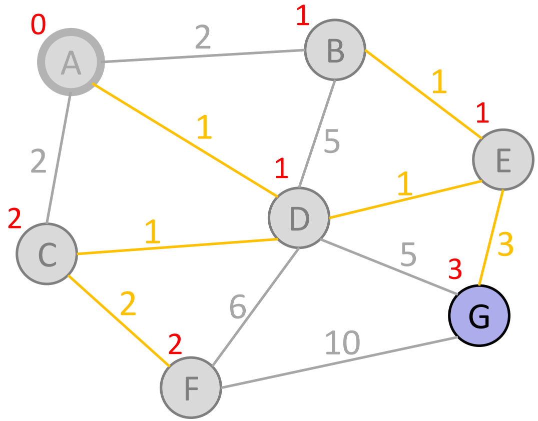


Order Added to Known Set:

A, D, C, E, B, F

Vertex	Known?	Distance	Previous
A	Y	0	\
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G		3	E

Prim's Algorithm: Example



All Done!!!

Total Cost: 9

Order Added to Known Set:

A, D, C, E, B, F

Vertex	Known?	Distance	Previous
A	Y	0	\
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G	Y	3	E

Prim's Algorithm: Demos and Visualizations

- Dijkstra's Visualization
  - https://www.youtube.com/watch?v=1oiQ0hrVwJk
  - Dijkstra's proceeds radially from its source, because it chooses edges by *path length from source*. Within the fringe, it "jumps around"
- Prim's Visualization
  - https://www.youtube.com/watch?v=6uq0cQZOyoY
  - Prim's proceeds radially from the MST-under-construction, because it chooses edges by *edge weight* (there's no source). Within the fringe, it "jumps around"

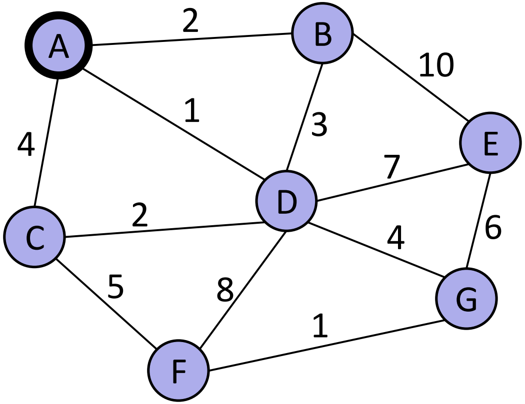
- Demo:
  - https://docs.google.com/presentation/d/1GPizbySYMsUhnXSXKvbqV4UhPCvrt750MiqPPgU-eCY/present?ueb=true&slide=id.g9a60b2f52_0_205

Prim's Algorithm: Analysis

- Correctness:
  - A bit tricky to prove, but intuitively similar to Dijkstra
  - Might return to this time permitting (unlikely)

- Run-time:
  - Same as Dijkstra's! $O(|E|\log|V| + |V|\log|V|)$ using a priority queue
  - But since $E \in O(|V|^2)$, can also state as $O(|E|\log|V|)$

Prim's Algorithm: Student Activity



Order Added to Known Set:

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

Lecture Outline

- Dijkstra’s Algorithm
    - Review
    - For Reading: Correctness and Runtime

- Minimum Spanning Tree
    - Introduction
    - Prim’s Algorithm
    - **Kruskal’s Algorithm, sorta**

Kruskal's Algorithm: A Different Approach

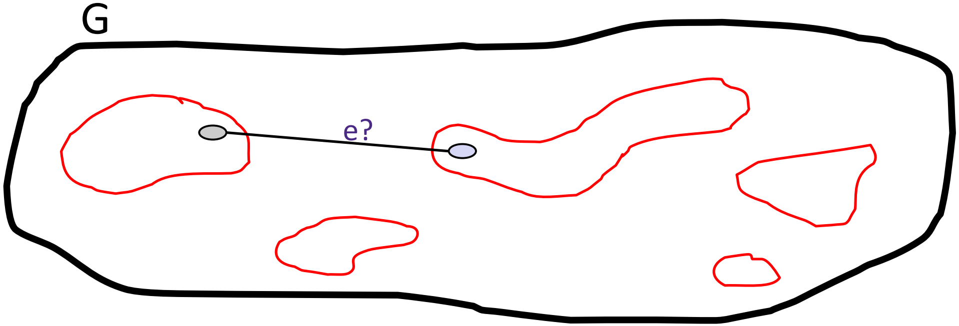
- Prim's thinks vertex by vertex
    - Eg, add the closest vertex to the currently reachable set

- What if you think edge by edge instead?
    - Eg, start from the lightest edge; add it if it connects new things to each other (don't add it if it would create a cycle)

Kruskal’s Algorithm

- *Intuition*: an edge-based greedy algorithm
  - Builds MST by greedily adding edges

- *Summary*:
  - Start with a ***forest*** of |V| MSTs
  - Successively connect them ((ie, eliminate a tree) by adding edges
  - Do not add an edge if it creates a cycle



Kruskal's Algorithm: Pseudo-pseudocode

```
kruskals(Graph g) {
  mst = {}
  forests = buildForests(g.vertices)
  edges = buildHeap(g.edges)

  while (forests.numForests() != 1):
    e = edges.deleteMin()
    u_id = forests.getForestId(e.u)
    v_id = forests.getForestId(e.v)
    if (u_id != v_id):
      mst.addEdge(e)
      forests.connect(e.u, e.v)
}
```

Does this fit our 5-step pattern for a graph traversal?

What data structure is this?!?!