CS 600.226: Data Structures

Michael Schatz

Nov 28, 2018

Lecture 36: Dijkstra Algorithm

Assignment 9: StringOmics

Out on: November 16, 2018

Due by: November 30, 2018 before 10:00 pm

Collaboration: None

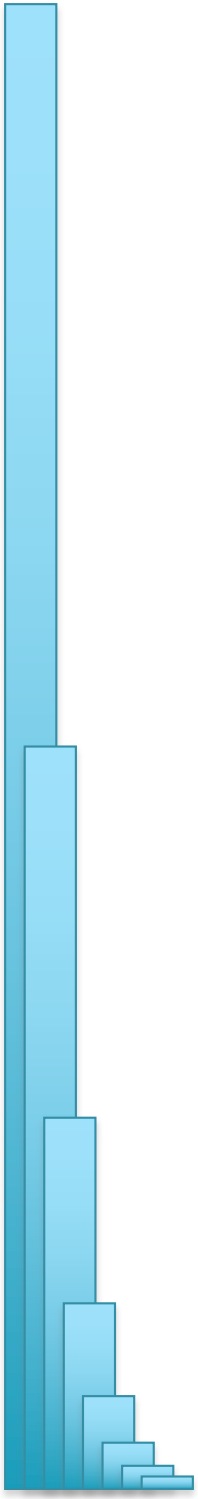
Grading:

Packaging 10%,
Style 10% (where applicable),
Testing 10% (where applicable),
Performance 10% (where applicable),
Functionality 60% (where applicable)

Overview

The ninth assignment focuses on data structures and operations on strings. In this assignment you will implement encoding and decoding using the Burrows Wheeler Transform as well as encoding and decoding in a simple form of run length encoding. In the final problem you will be asked to measure the space savings using run length encoding with and without applying the Burrows Wheeler Transform first.

Remember: javac –Xlint:all & checkstyle *.java & Junit (No JayBee)



Assignment 10: The Streets of Baltimore

Out on: November 30, 2018

Due by: December 7, 2018 before 10:00 pm

Collaboration: None

Grading:

Packaging 10%,
Style 10% (where applicable),
Testing 10% (where applicable),
Performance 10% (where applicable),
Functionality 60% (where applicable)

Overview

The tenth assignment returns to our study of graphs, although this time we are using a weighted graph rather than the unweighted movie/movie-star graph. Specifically, you will be touring the streets of Baltimore to find the shortest route from the JHU campus to other destinations around Baltimore.

Remember: javac –Xlint:all & checkstyle *.java & Junit
(No JayBee)

Assignment 10: The Streets of Baltimore

Out on: November 30, 2018

Due by: December 7, 2018 before 10:00 pm

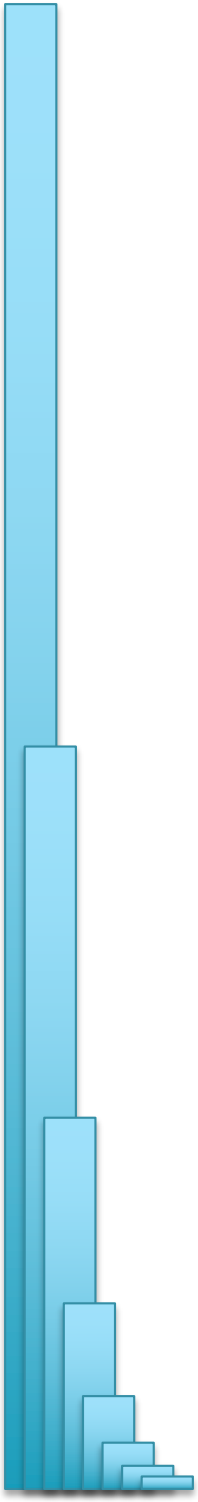
We have decided to make the tenth assignment ***optional.***

If you elect to do the assignment it can be used to ***replace*** the grade from one of your earlier assignments, so that we will only consider your 9 highest grades from all of the assignments. If you submit this assignment but score worse than your previous nine assignments, this assignment will be dropped.

If you elect to skip the assignment, there will be no penalty, although we strongly encourage you to do so only if you have very good grades for the other assignments. Also note that the concepts and implementation issues that arise for this assignment are all valid questions for the final exam.

star graph. Specifically, you will be touring the streets of Baltimore to find the shortest route from the JHU campus to other destinations around Baltimore.

Remember: javac –Xlint:all & checkstyle *.java & Junit
(No JayBee)

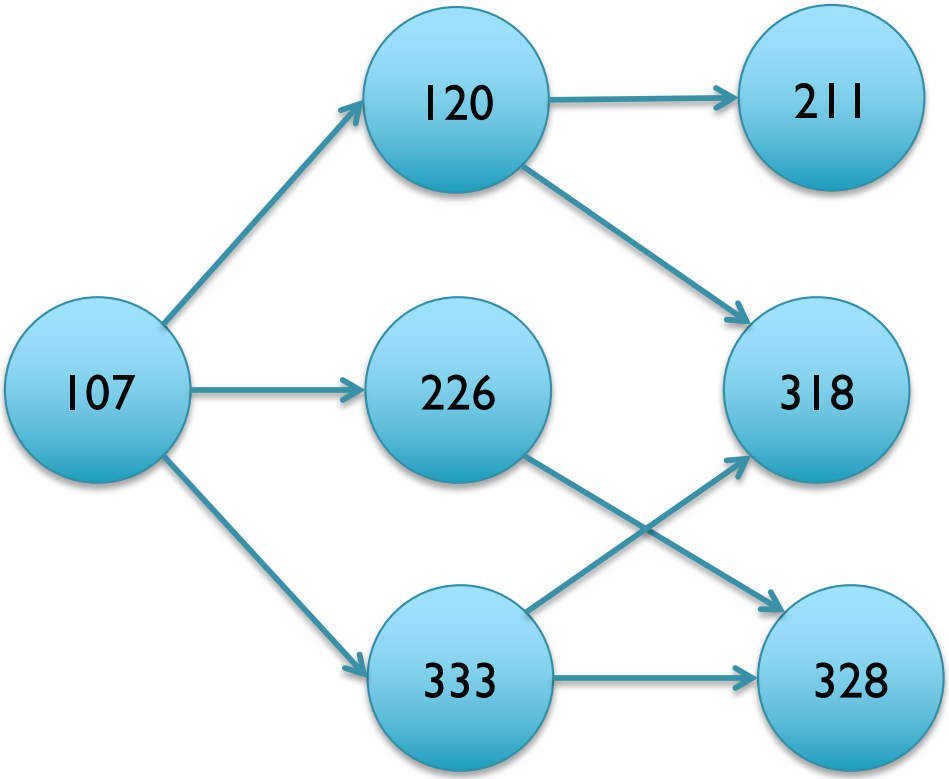


Part 1: Topological Sort aka Sorting graphs

Scheduling Challenges

- Consider the following class prerequisites:
  - 107 is required before taking 120, 226, or 333
  - 120 is required before taking 211 or 318
  - 226 is required before taking 328
  - 333 is required before taking 318 or 328

What courses should I take now so I can take all of these by senior year? ☺



Topological sorting

Analysis of directed, acyclic graphs (DAGs) with no weights on the edges

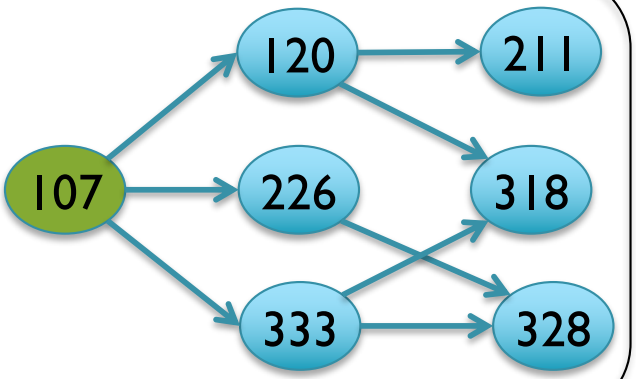
Vertices represent "***tasks***", edges represent some "***before***" relationship

Goal: Find a valid sequence of tasks that fulfill the ordering constraints

Topological sort

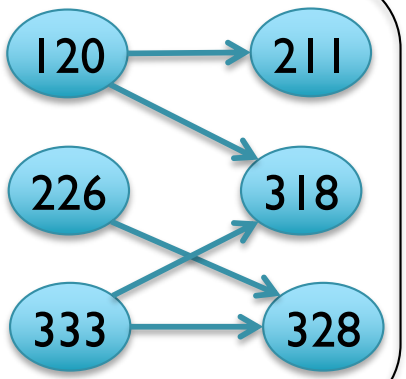
1. ***Pick a node that has no prior constraints***

Since we are working with DAGs there must be at least 1 such node with indegree 0



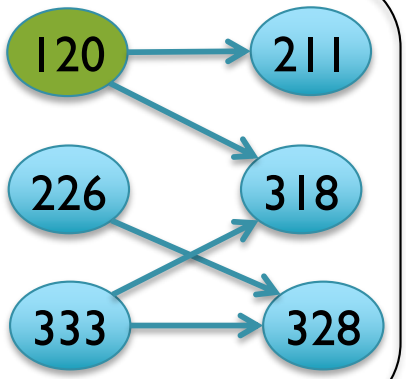
2. Remove that node from the DAG and all incident edges

The node you just removed starts a valid sequence of classes: 107



3. Arbitrarily pick another "free" node

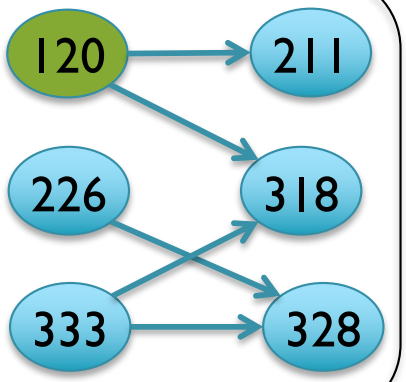
We could pick 120, 226, or 333. Picking 120 immediately opens up 211, but no new courses open up by picking 226 or 333



Topological sort

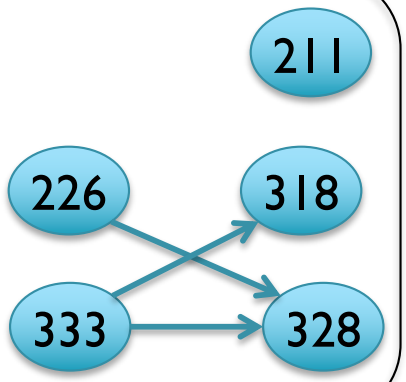
3. Arbitrarily pick another "free" node

We could pick 120, 226, or 333. Picking 120 immediately opens up 211, but no new courses open up by picking 226 or 333



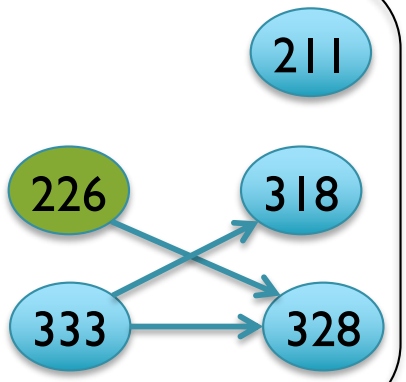
4. Remove that node

The valid sequence grows: 107, 120



5. Repeat!

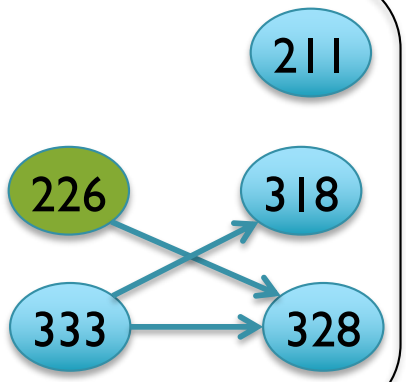
The valid sequence grows: 107, 120



Topological sort

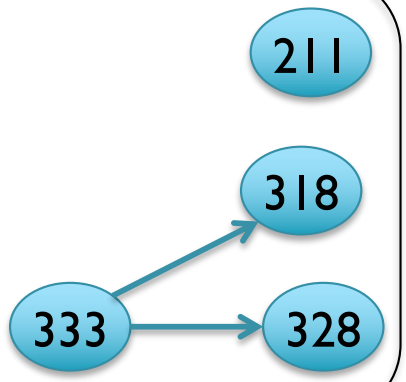
5. Repeat!

The valid sequence grows: 107, 120



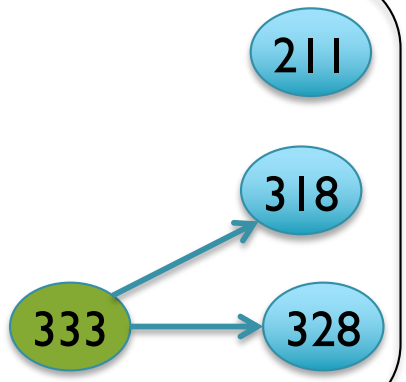
6. Repeat!

The valid sequence grows: 107, 120, 226



7. Repeat!

The valid sequence grows: 107, 120, 226



Topological sort

8. Repeat!

The valid sequence grows: 107, 120, 226, 333

211

318

328

9. Repeat!

The valid sequence grows: 107, 120, 226,333, 328

211

318

10. Repeat!

The valid sequence grows: 107, 120, 226, 333, 328, 211

318

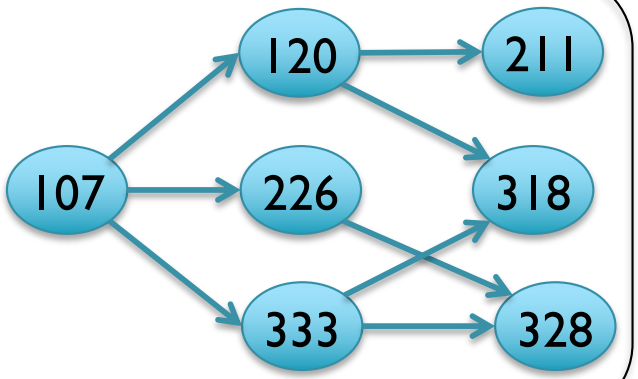
Topological sort

10. Repeat!

The valid sequence grows: 107, 120, 226, 333, 328, 211, 318

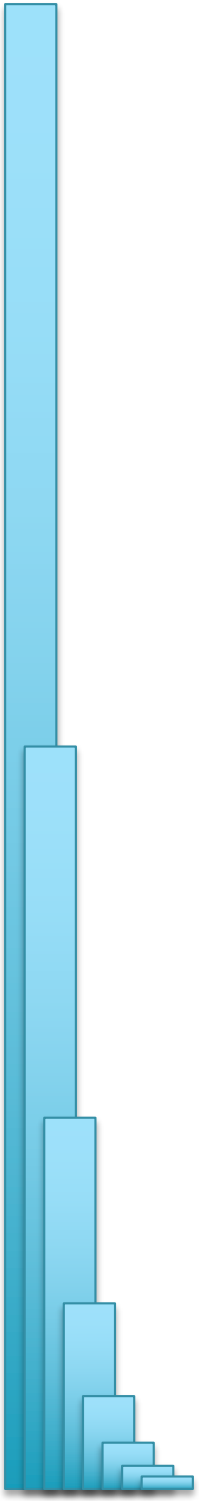
We just found one of potentially many valid sequences of courses

Alt1: 107, 226, 120, 211, 333, 328, 318
Alt2: 107, 333, 226, 328, 120, 318, 211



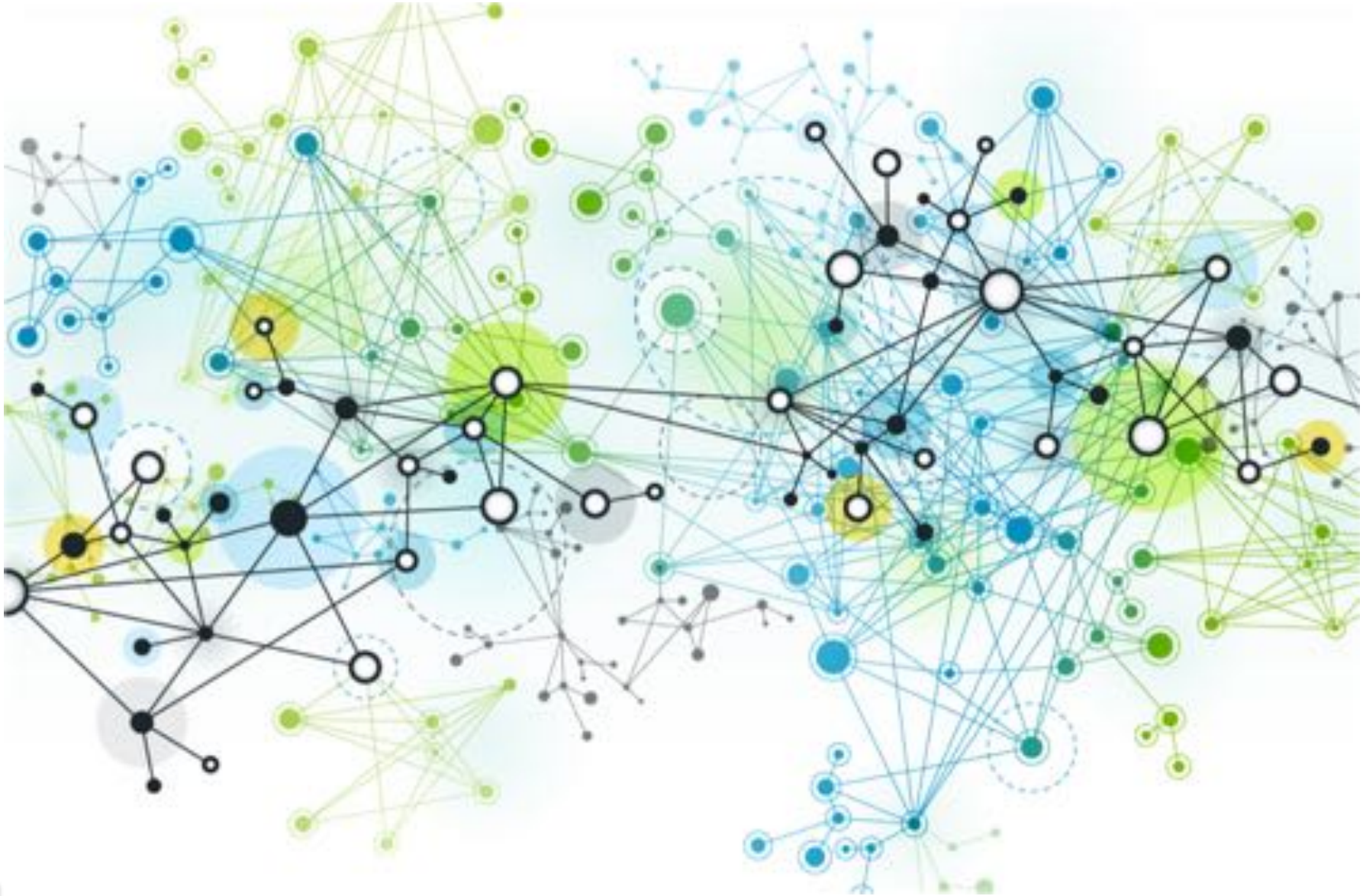
What is the running time to find a valid sort?

Linear Time: O(|V|) + O(|E|)



Part 2: Dijkstra's Algorithm aka Shortest Path Revisited

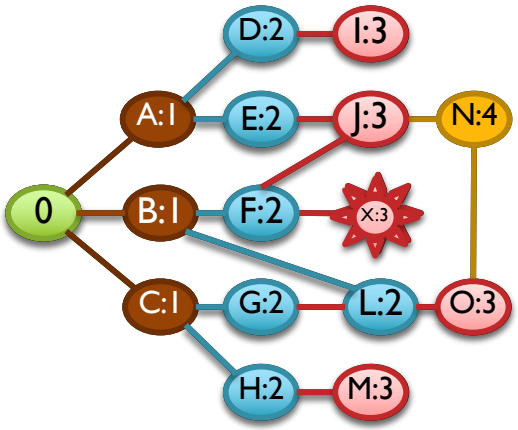
Graphs are Everywhere!



One of the most important queries
is finding the (shortest) path from A to B!

BFS

```
BFS(start, stop)
// initialize all nodes dist = -1
start.dist = 0
list.addEnd(start)
while (!list.empty())
   cur = list.getBegin()
   if (cur == stop)
      print cur.dist;
   else
      foreach child in cur.children
         if (child.dist == -1)
            child.dist = cur.dist+1
            list.addEnd(child)
```



0

A,B,C
B,C,D,E
C,D,E,F,L

D,E,F,L,G,H
E,F,L,G,H,I
F,L,G,H,I,J
L,G,H,I,J,X
G,H,I,J,X,O
H,I,J,X,O

I,J,X,O,M
J,X,O,M
X,O,M,N
O,M,N
M,N

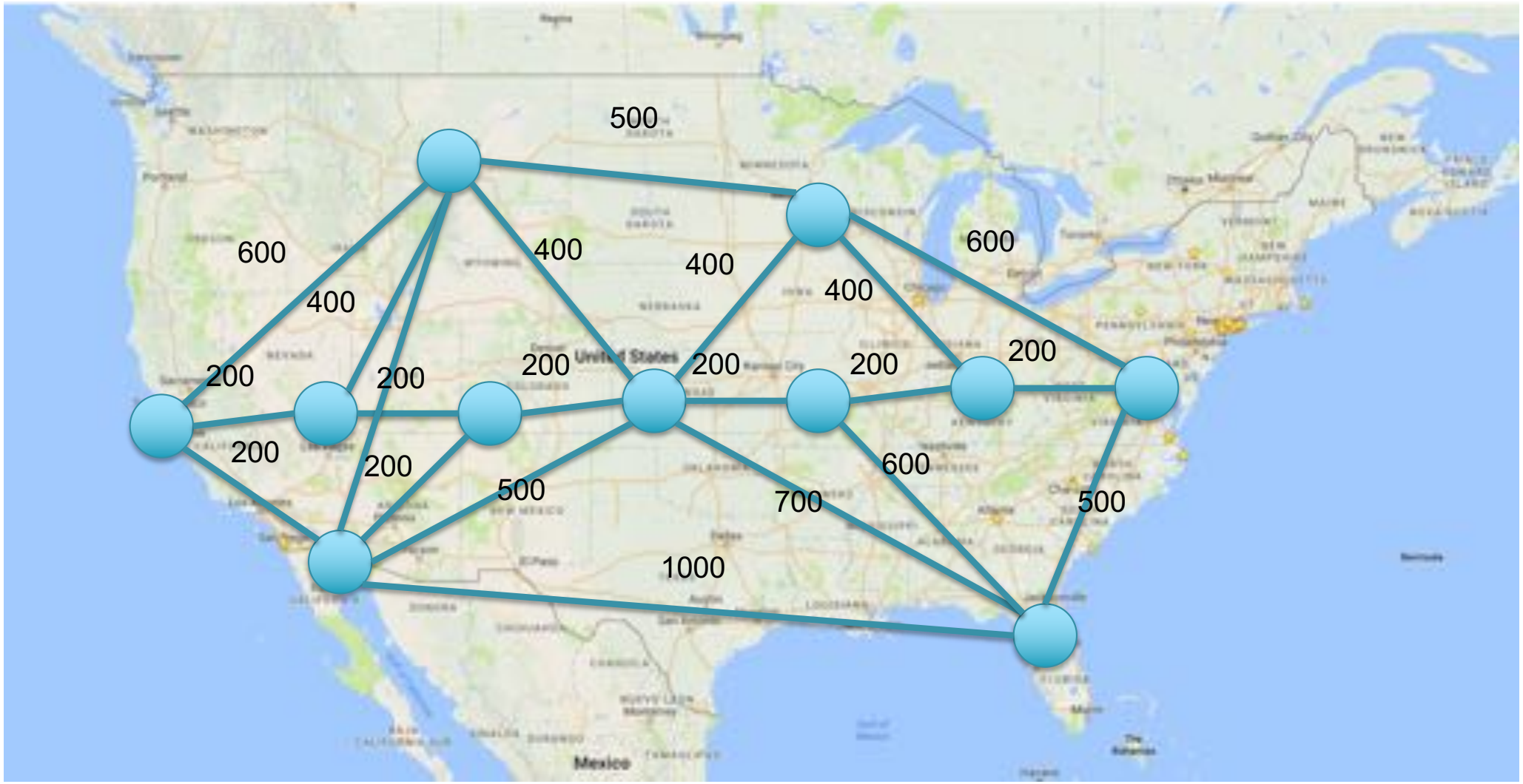
N

[How many nodes will it visit?]

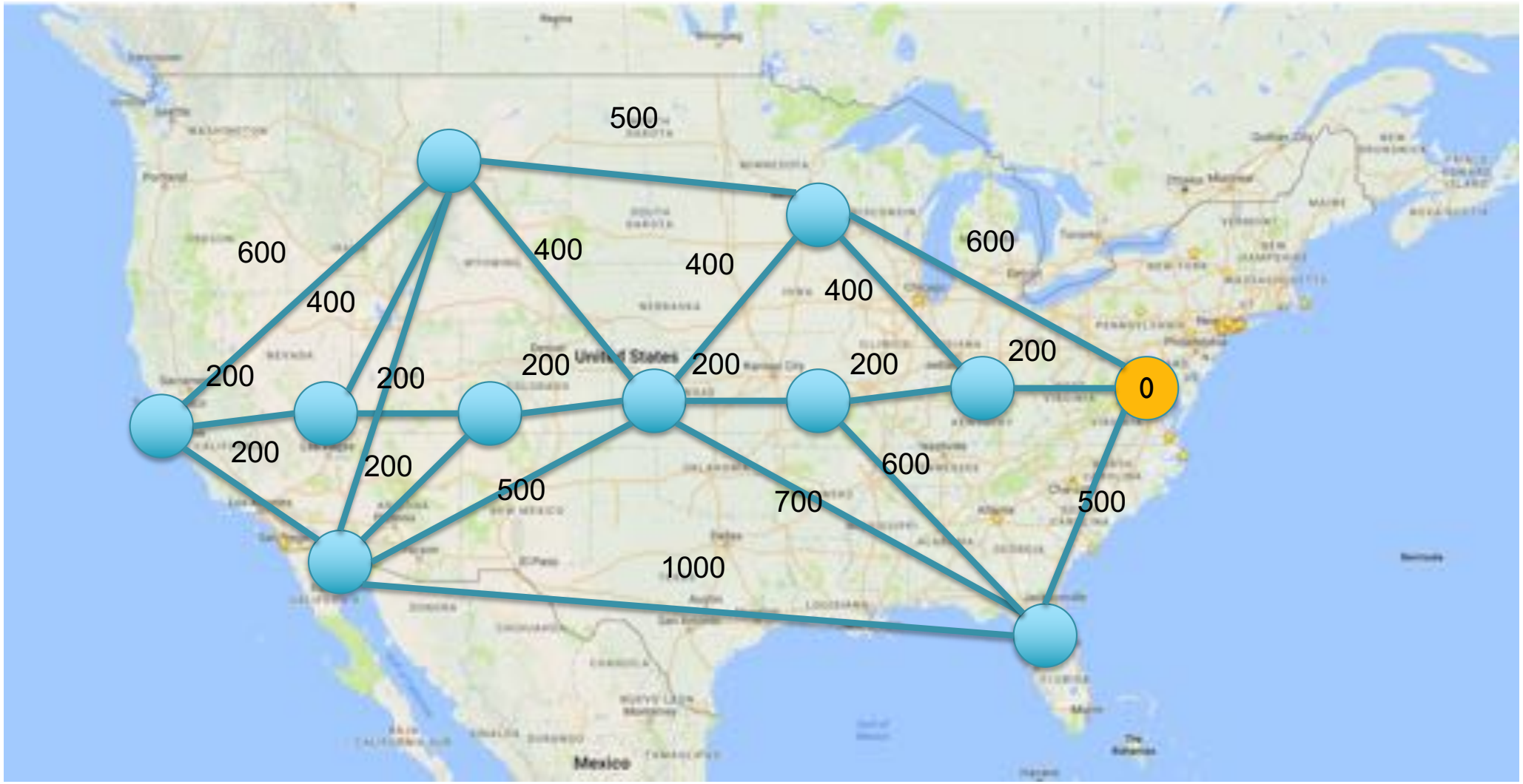
[What's the running time?]

[What happens for disconnected components?]

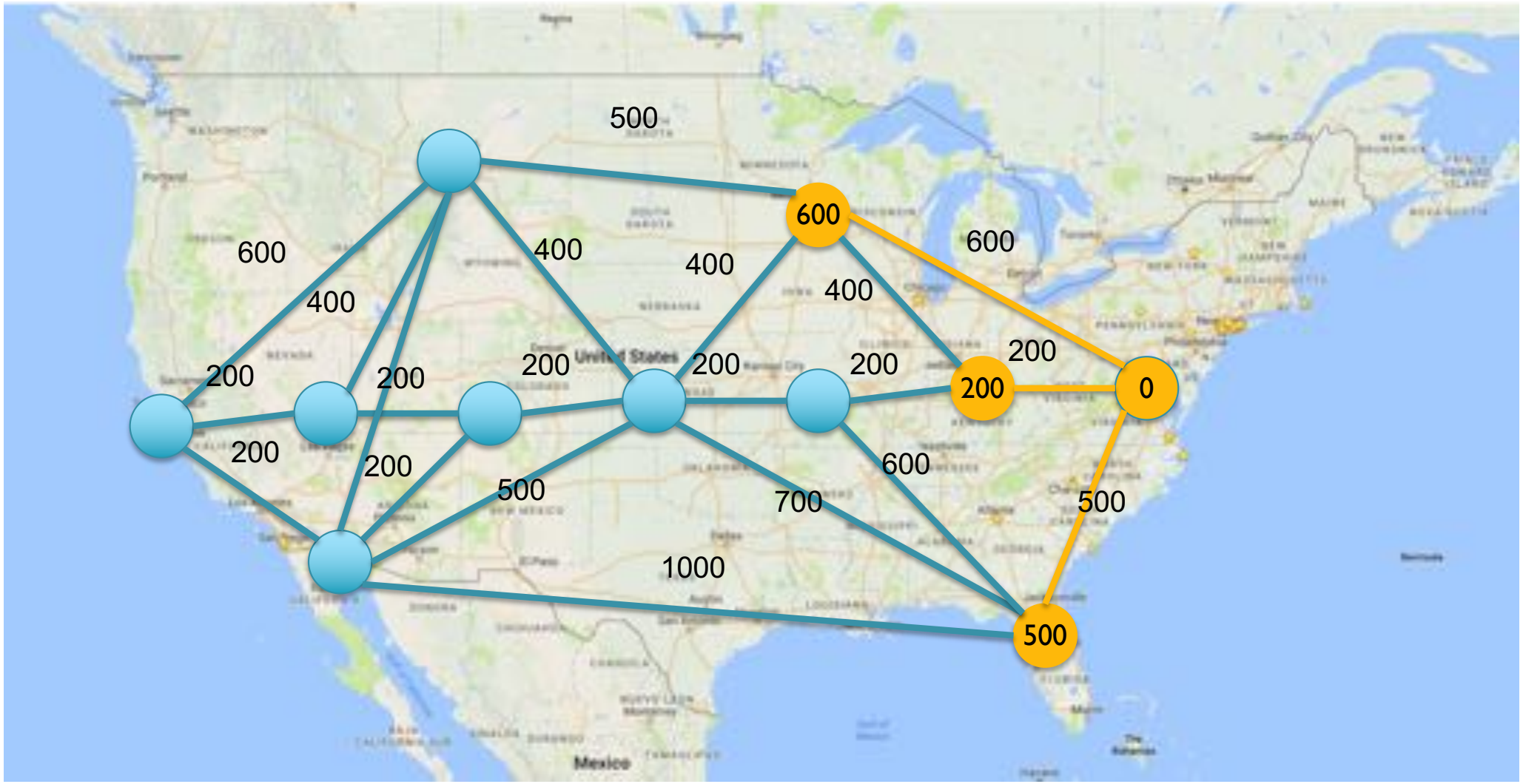
Why doesn't BFS work for maps?



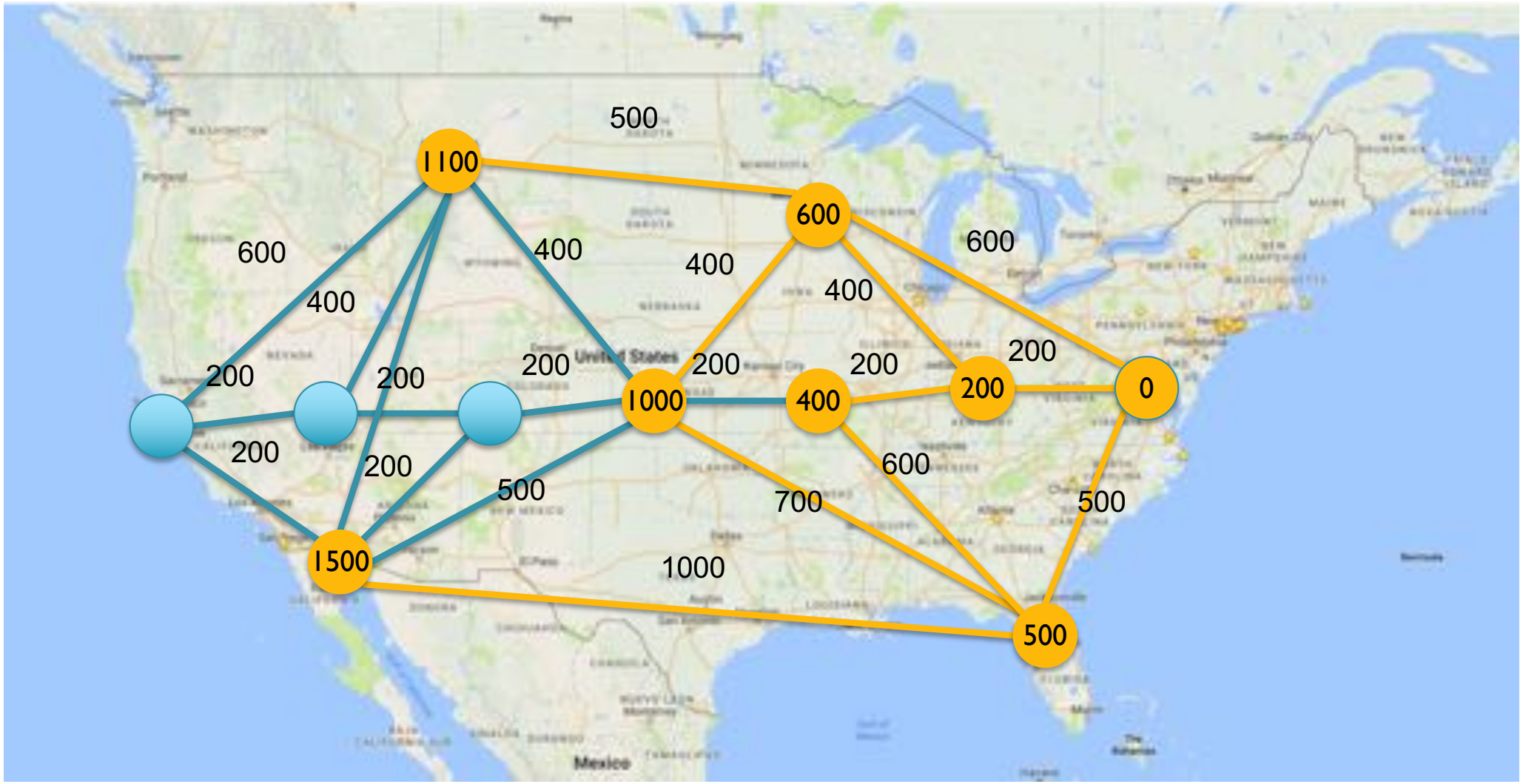
Why doesn't BFS work for maps?



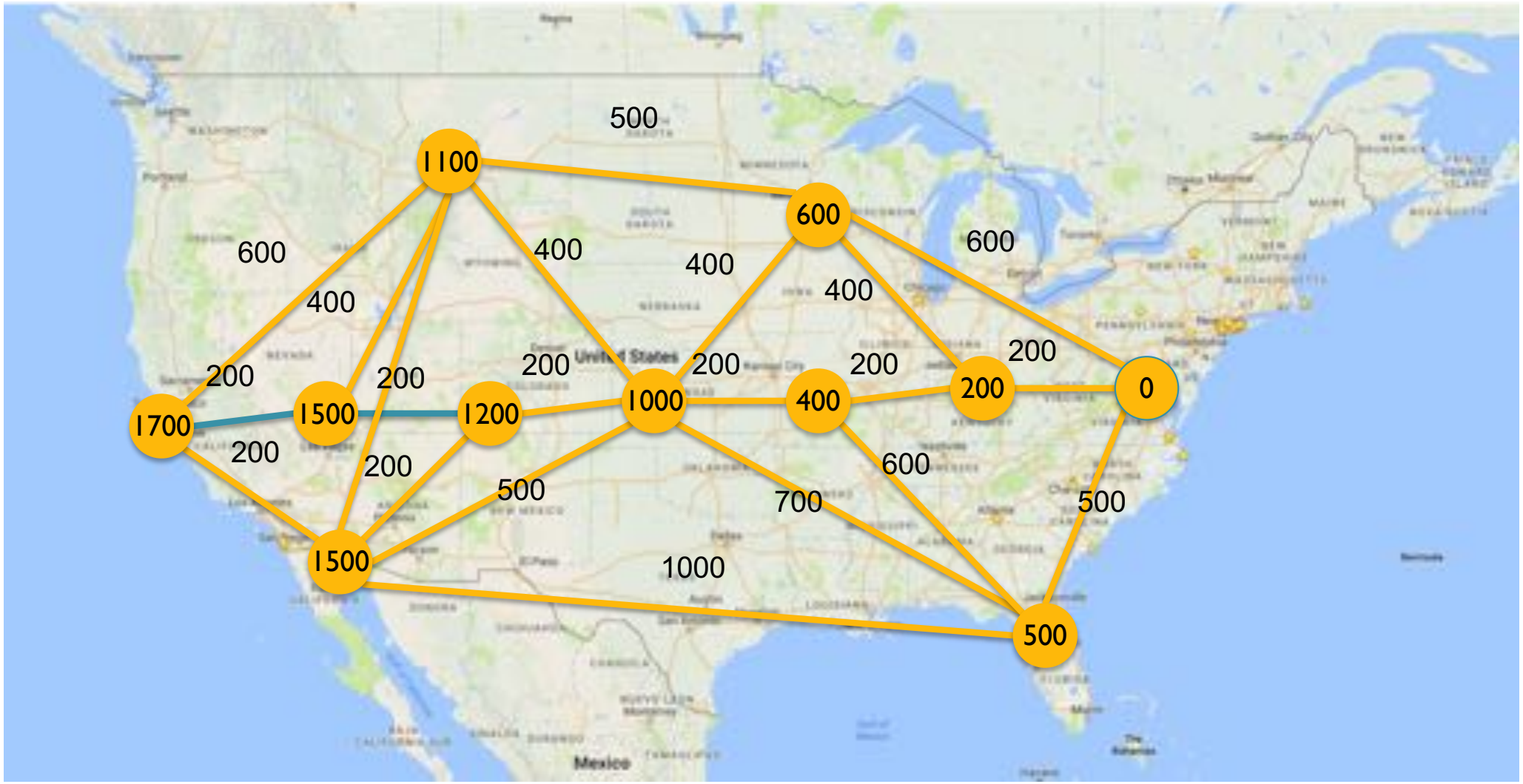
Why doesn't BFS work for maps?



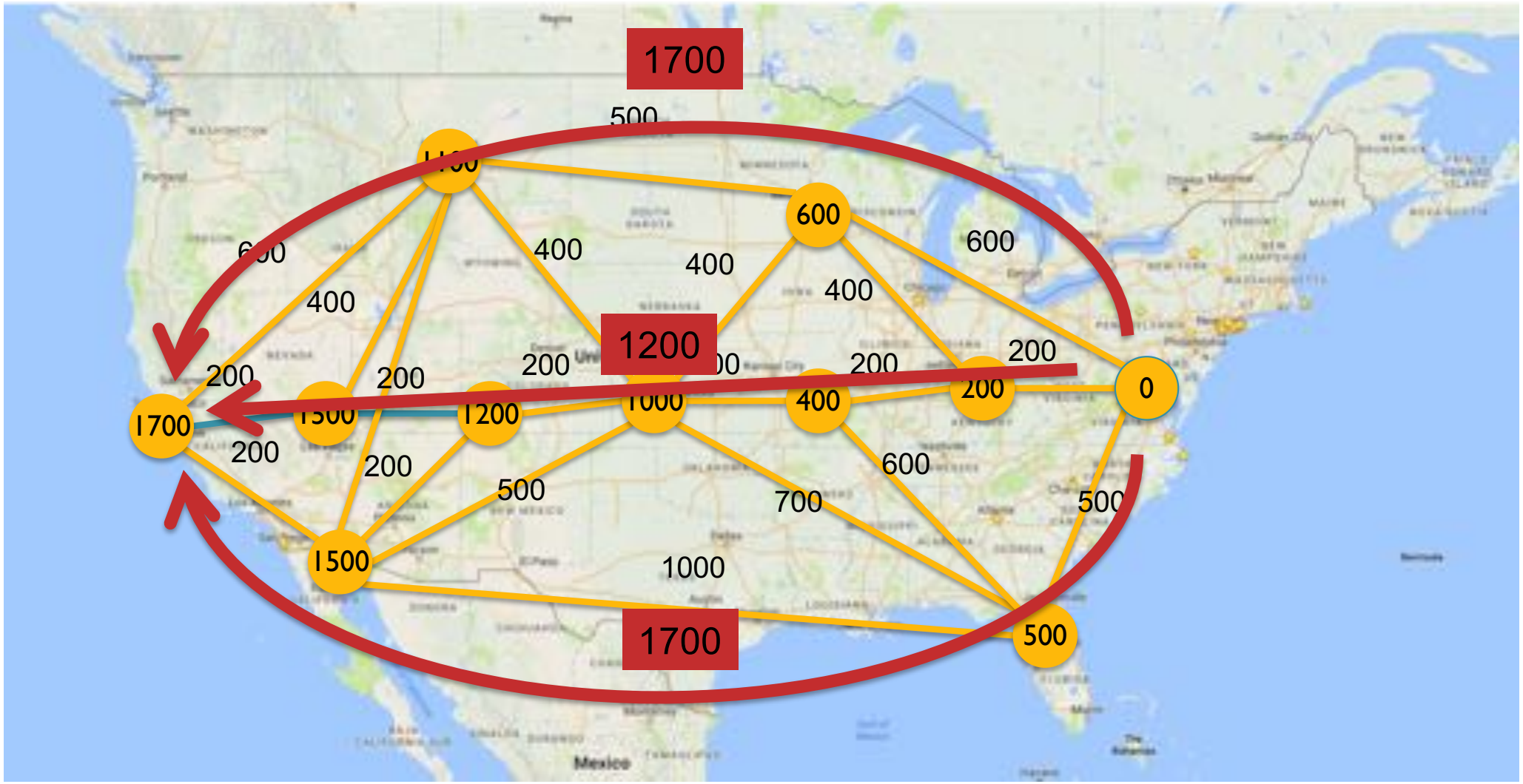
Why doesn't BFS work for maps?



Why doesn't BFS work for maps?

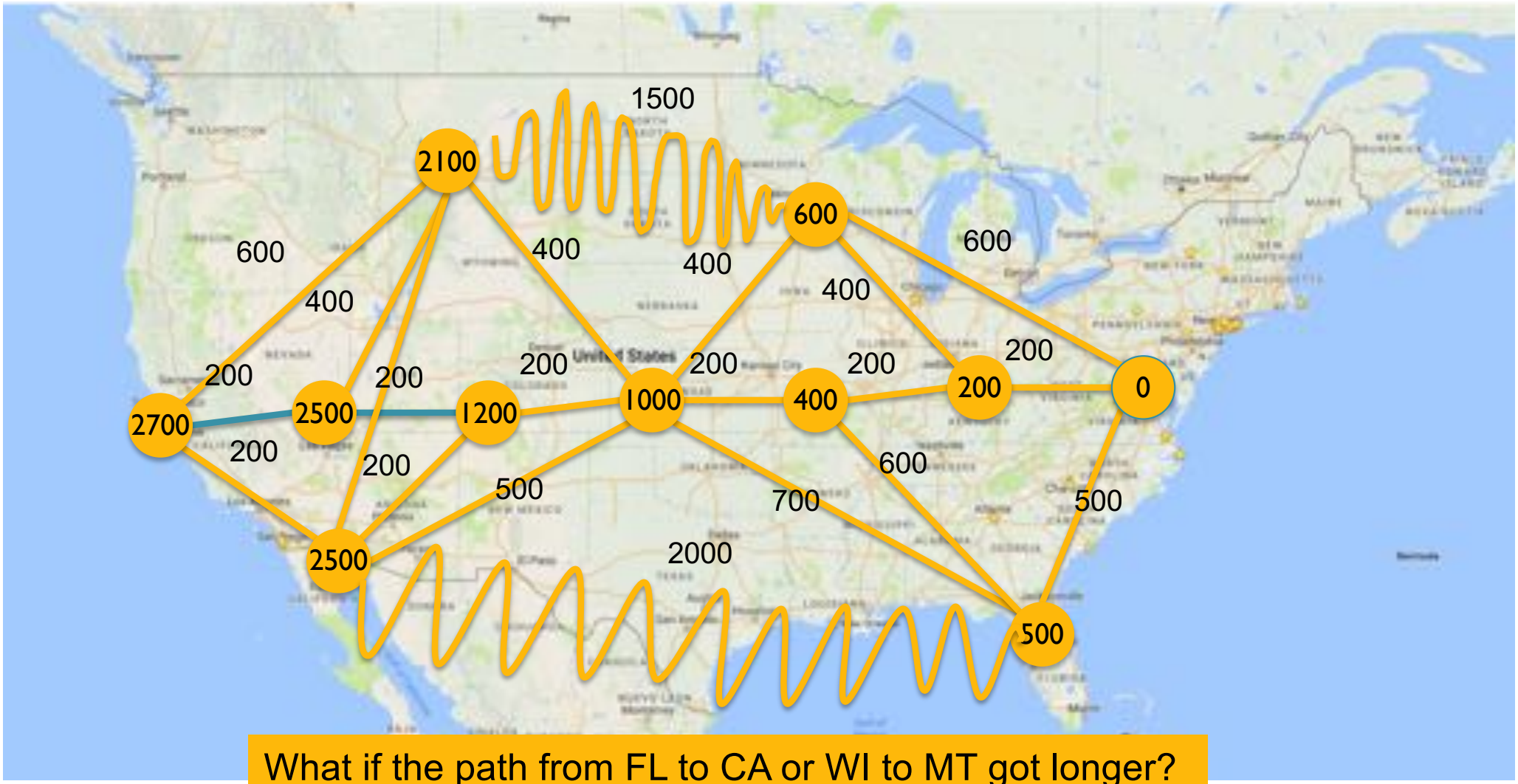


Why doesn't BFS work for maps?



Fewest hops != Shortest distance

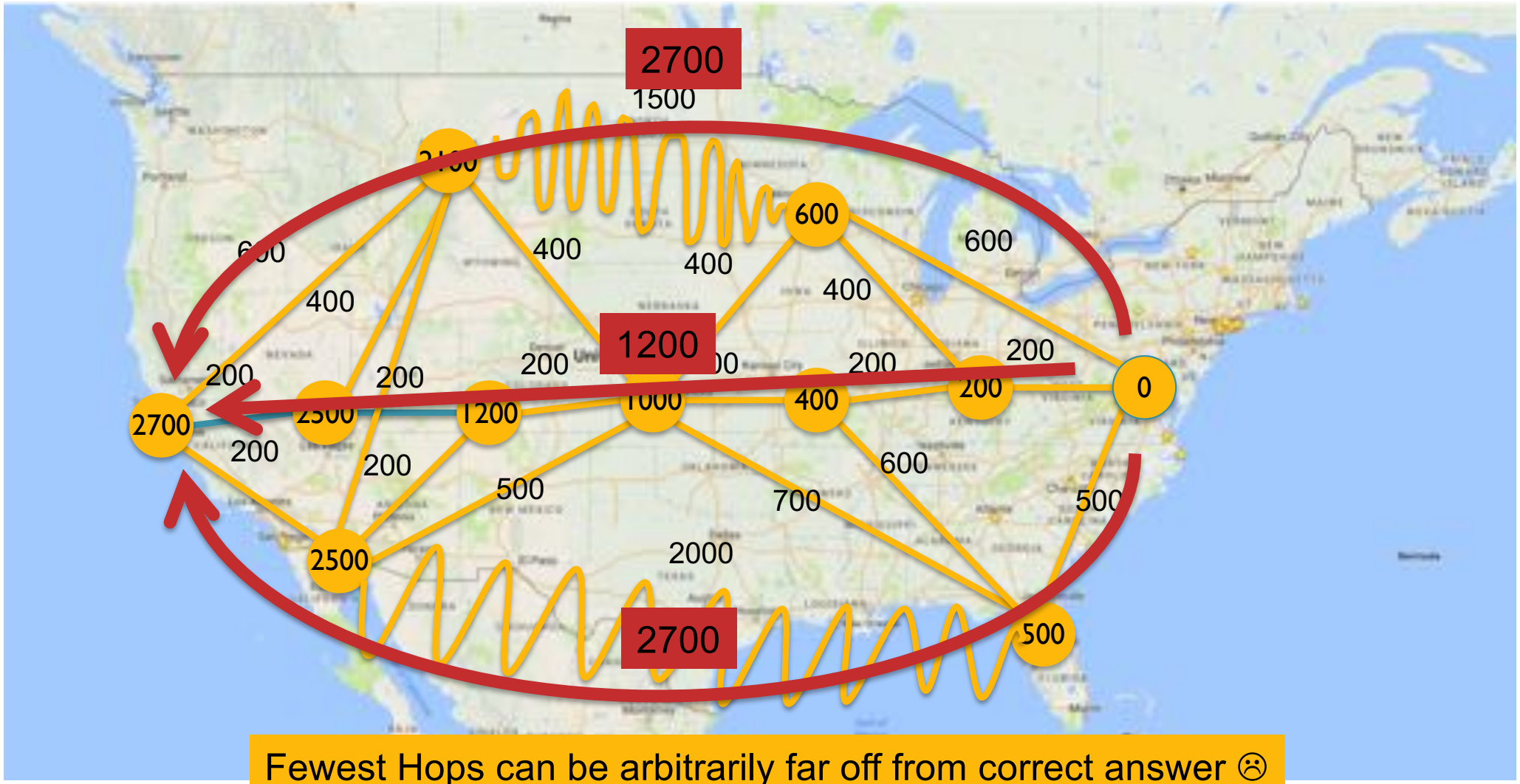
Why doesn't BFS work for maps?



What if the path from FL to CA or WI to MT got longer?

Fewest hops != Shortest distance

Why doesn't BFS work for maps?

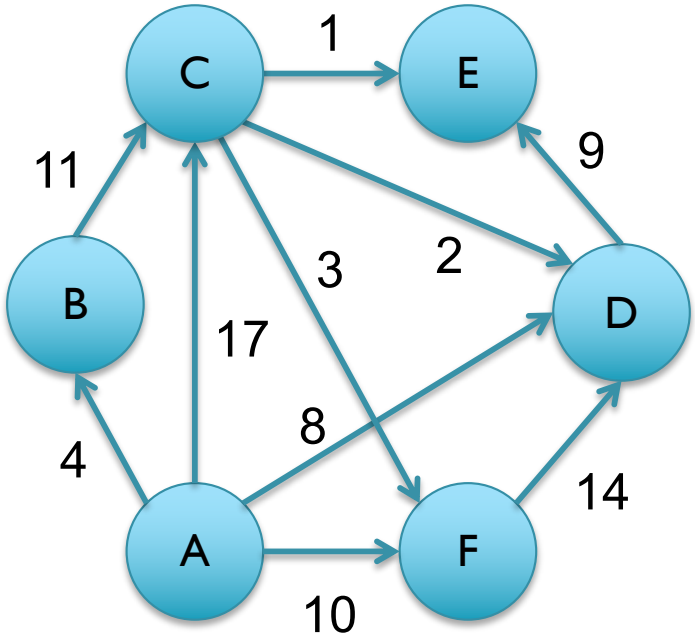


Fewest Hops can be arbitrarily far off from correct answer ☹

Fewest hops != Shortest distance

Simple Shortest Path Algorithm

Given a weighted directed graph, find the shortest (minimum weight) path from one start node to one final node.



Note, there may not be a path between those nodes:

- No roads from JHU to U. Hawaii ☹
- No path between nodes that only have outgoing edges

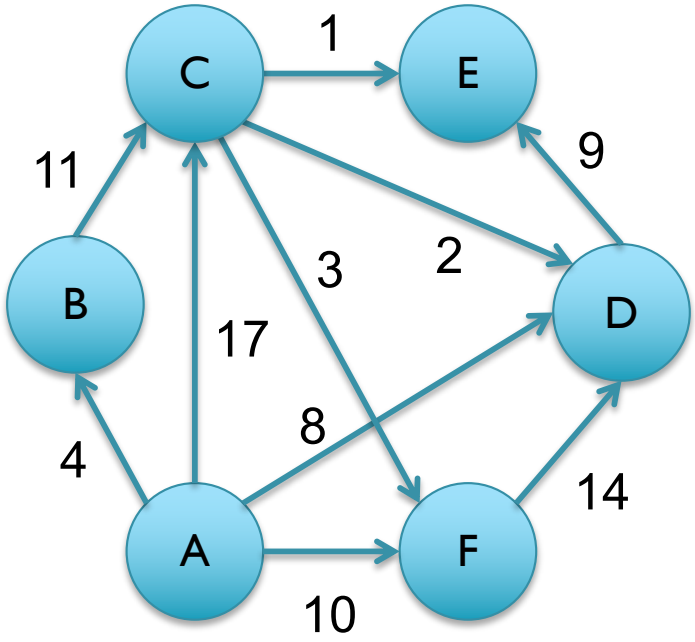
=> Report an infinite distance

Most commonly applied to graphs with non-negative edge weights

- What might happen with negative weights?

Simple Shortest Path Algorithm

Given a weighted directed graph, find the shortest (minimum weight) path from one start node to one final node.



Similar to BFS, maintain a search frontier of nodes that are further and further away

Similar to BFS, initialize the nodes as unvisited, and record the distance from start in the node

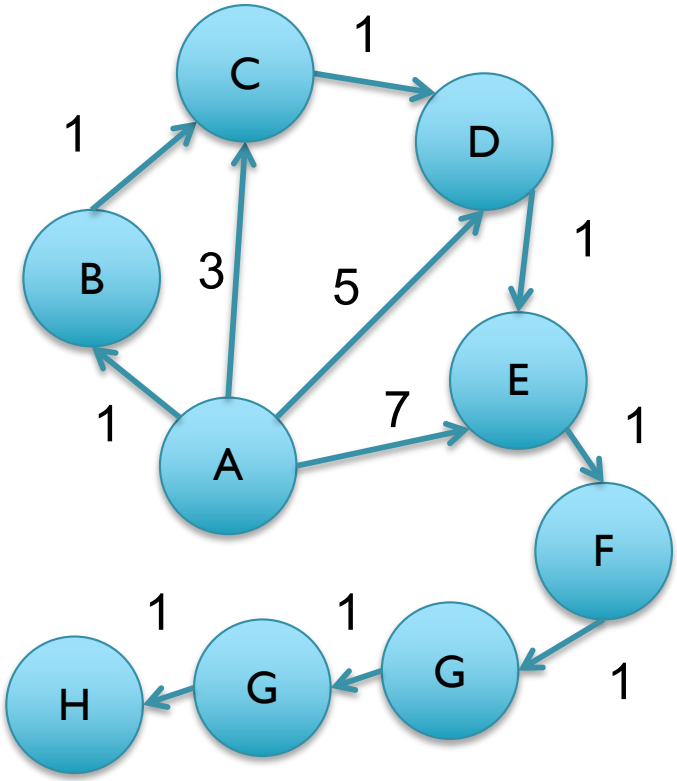
Similar to BFS, leave breadcrumbs along the way so we can retrace the path

Unlike BFS, we may have to revise the distance if we later find a cheaper path (A->B->C vs A->C)

Simple Shortest Path Algorithm

Given a weighted directed graph, find the shortest (minimum weight) path from one start node to one final node.

- Lets start with node A
- Initialize the distance to A as 0, and the distance everywhere else as infinity
- Lets explore all possible paths starting at A (all outgoing edges from A)
- Revise if the total distance we just traveled is less than the previously recorded distance (boring this round)
- Repeat! … repeat where?
- Repeat on all nodes that just had their distance updated!



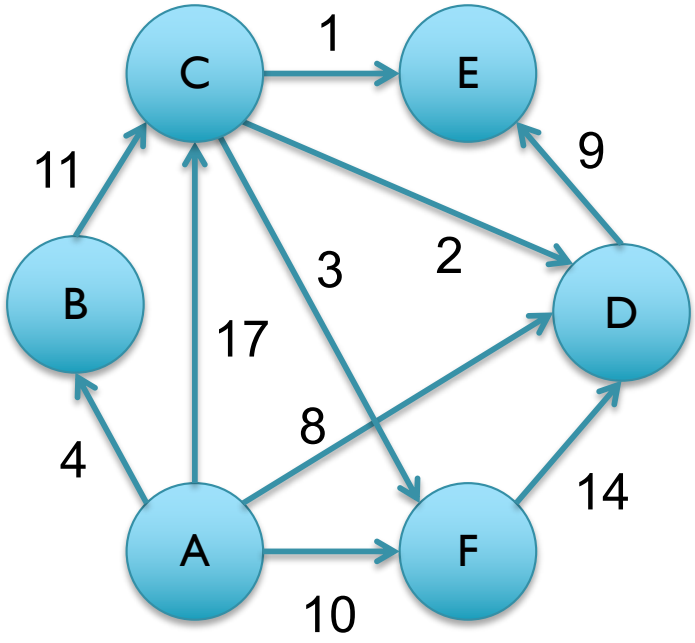
Whats wrong with this algorithm?

Correct but slow, same edges may be explored many times O(\|V\| * \|E\|)

Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

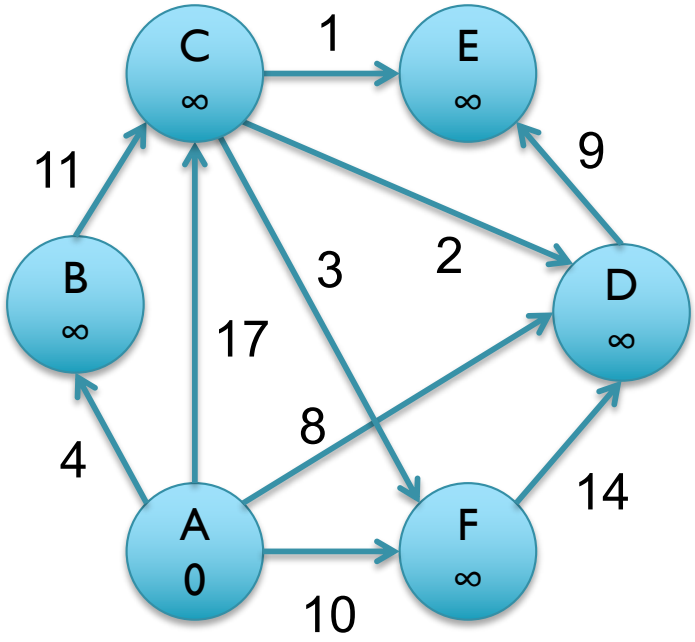


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

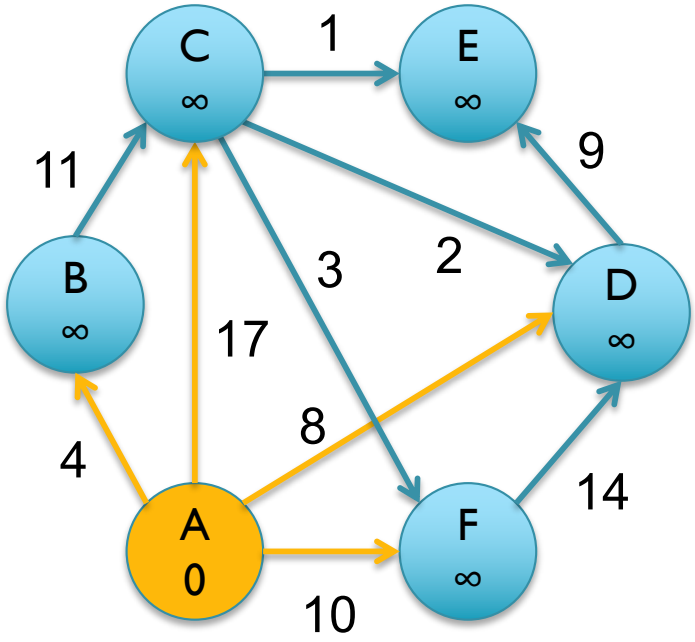


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

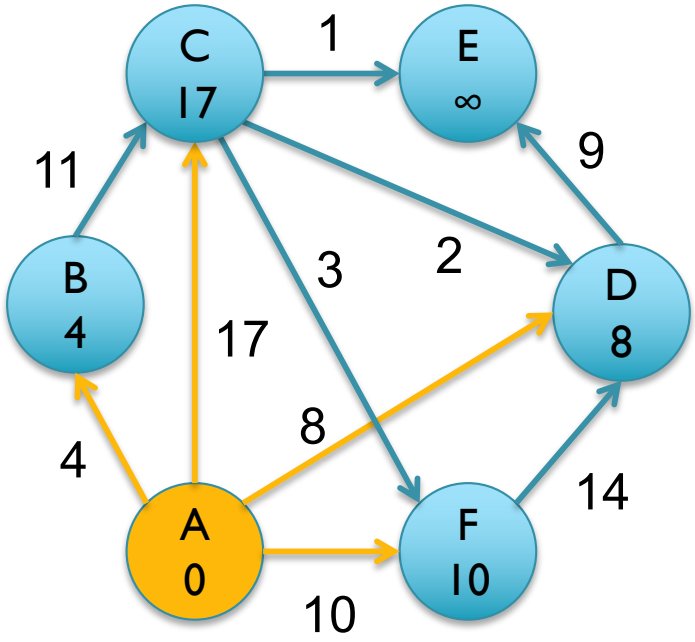


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

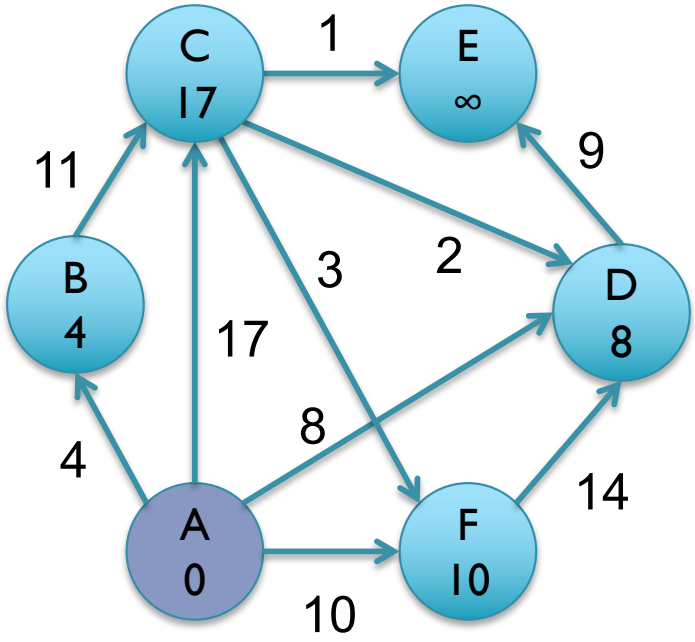


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

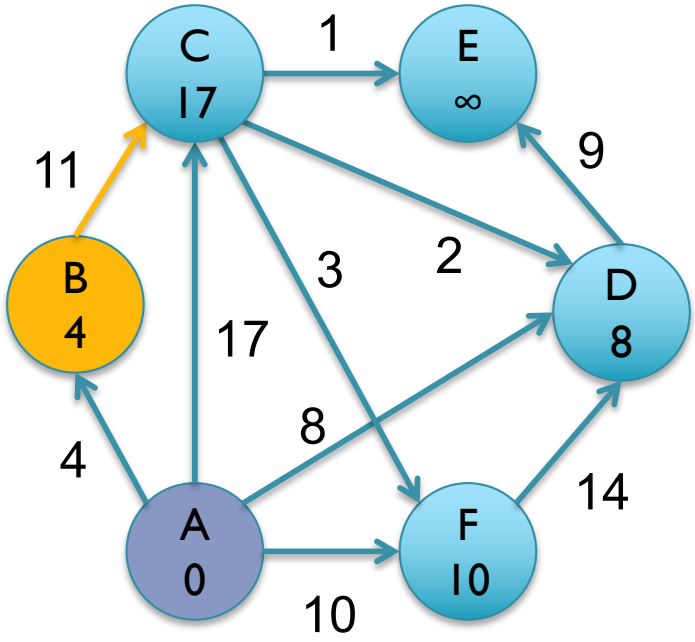


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

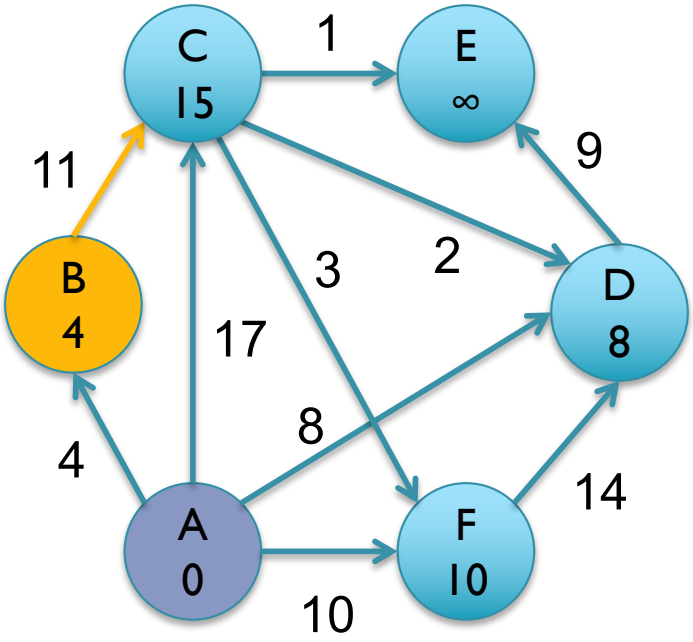


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

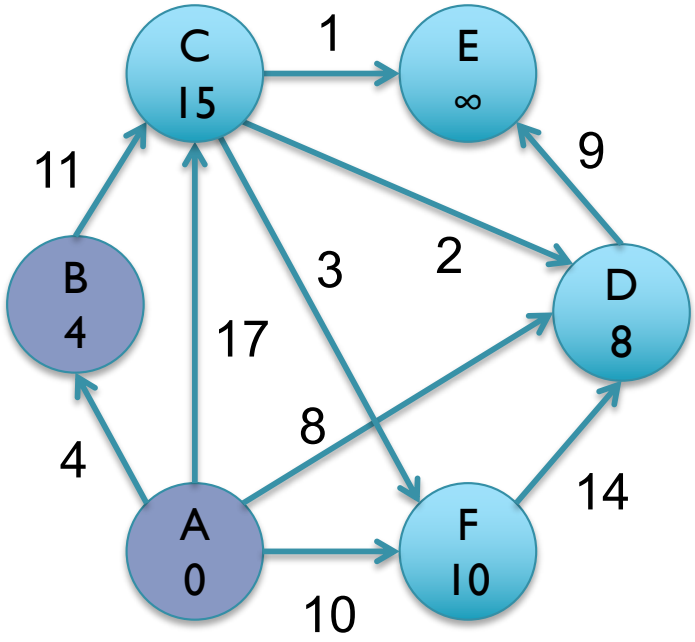


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

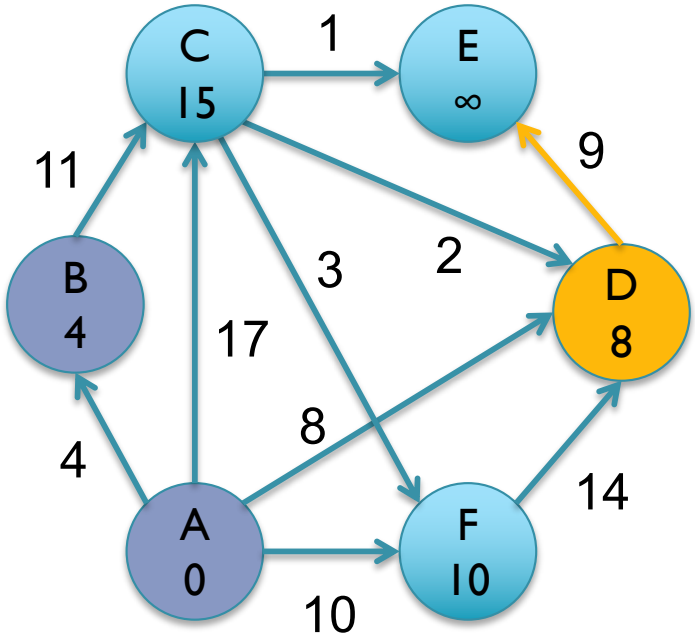


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

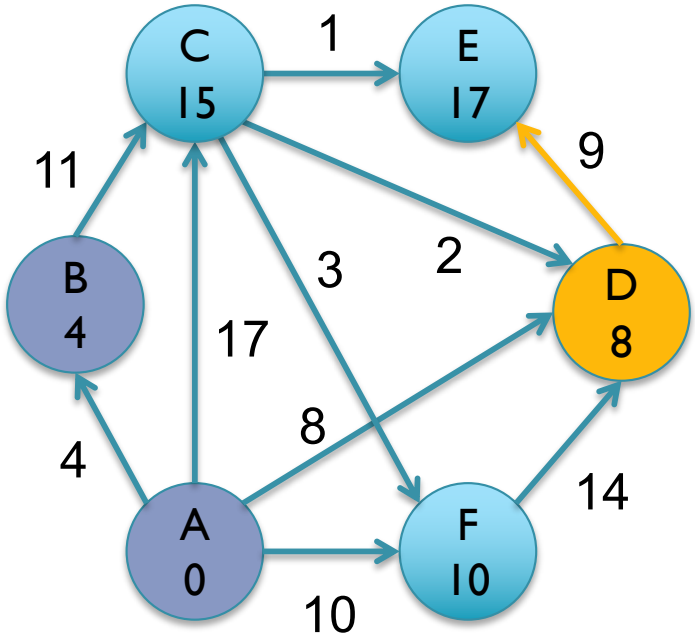


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

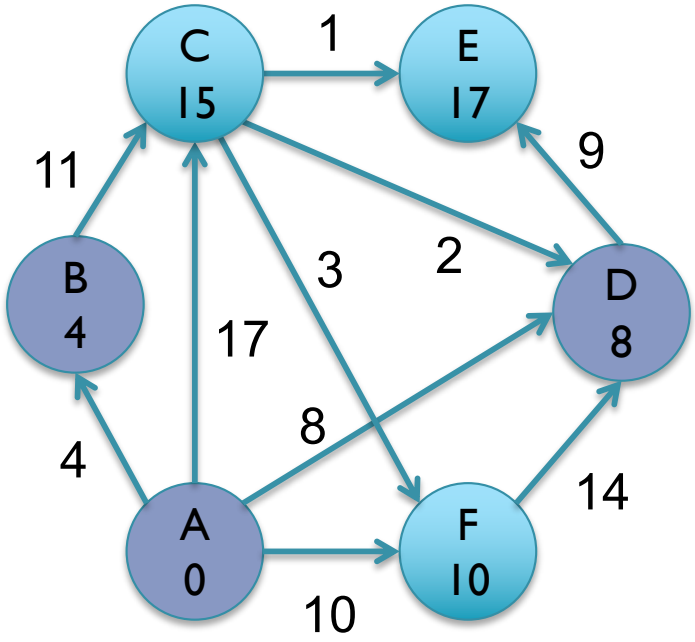


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

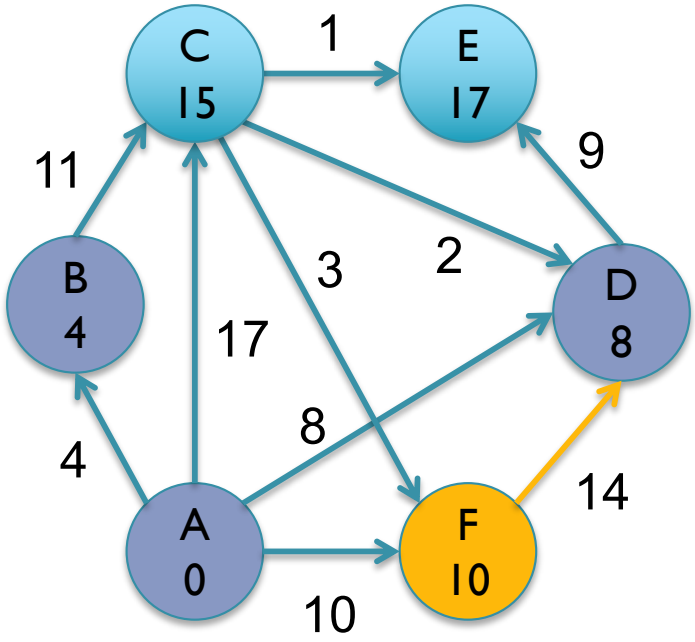


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

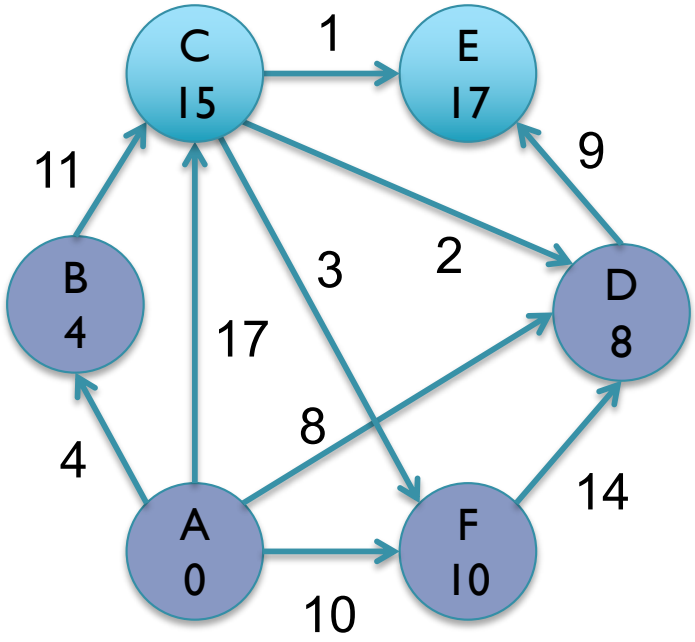


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

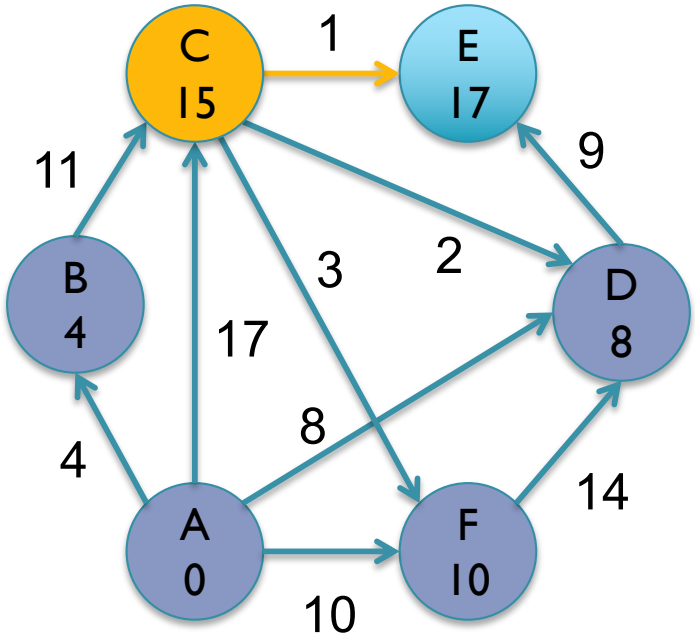


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

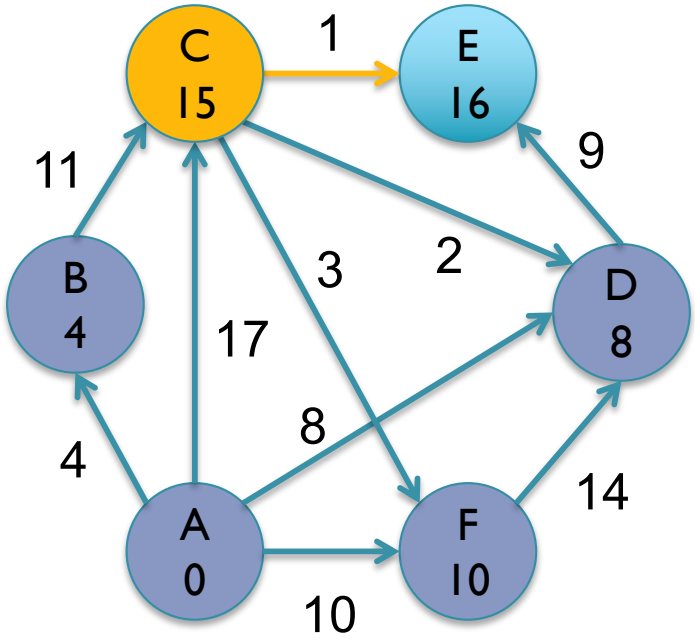


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

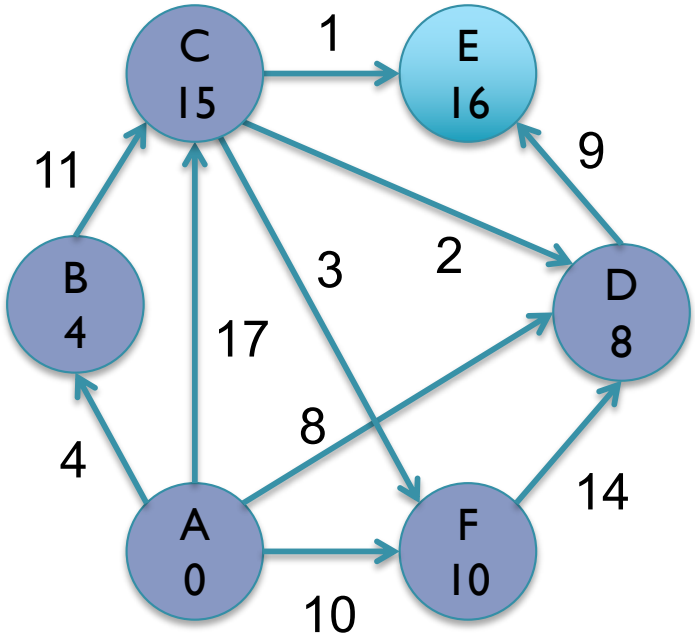


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

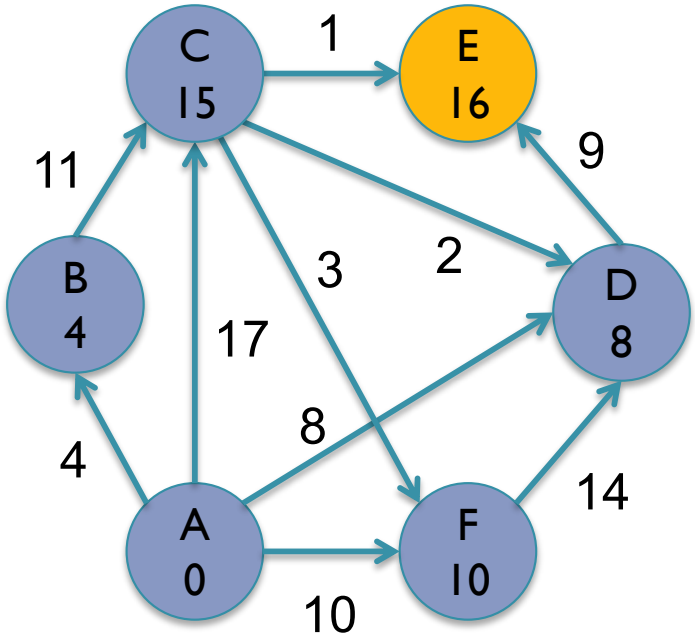


Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children



Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

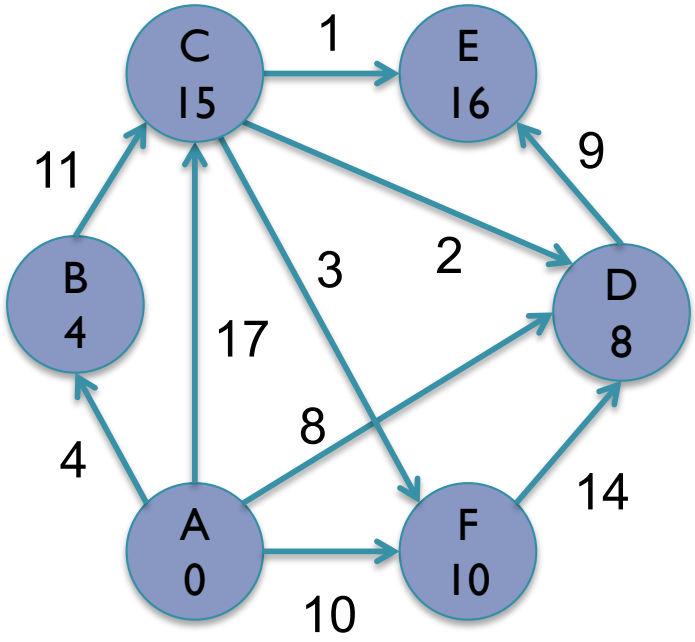
As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

No more unvisited nodes! Done

If we are looking for a path between a particular pair, could we terminate early?

Yes! As soon as we are done with target node, its distance will never change again



Dijkstra's Algorithm

Rather than uniformly pushing the whole search frontier like BFS, lets greedily push out the shortest paths so far

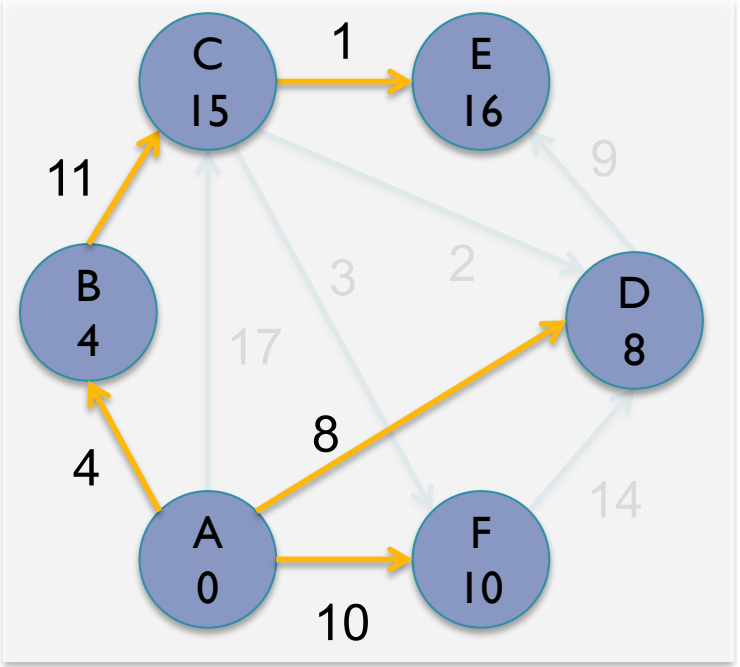
As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

Repeatedly pick the unvisited node with the smallest estimated distance, and revise the distances of its children

No more unvisited nodes! Done

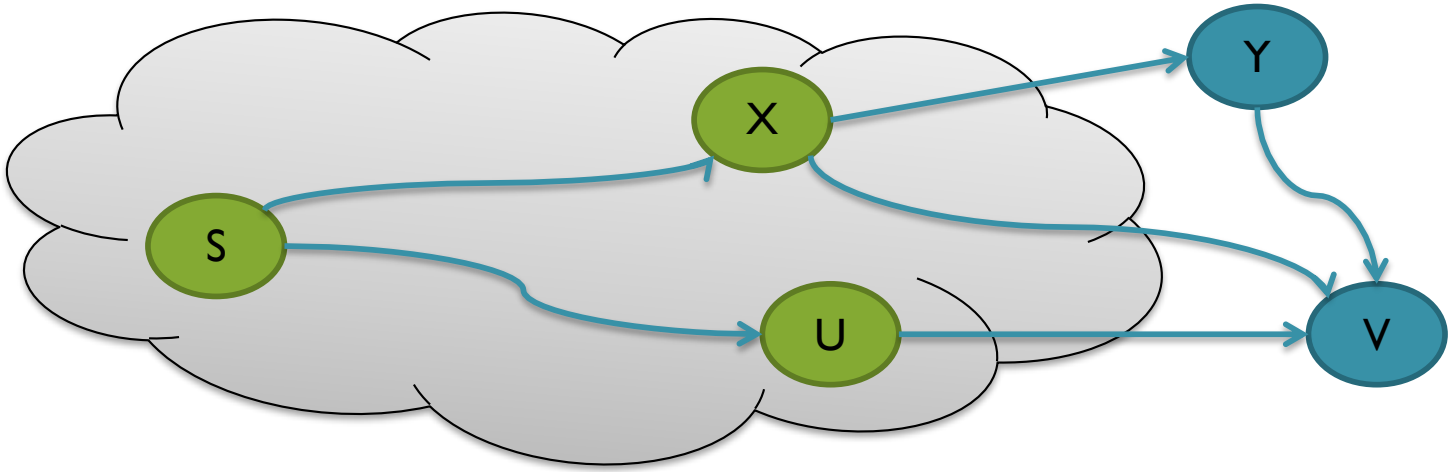
If we are looking for a path between a particular pair, could we terminate early?

Yes! As soon as we are done with target node, its distance will never change again



We are building a tree inside the graph of shortest paths from start ☺

Dijkstra's Correctness

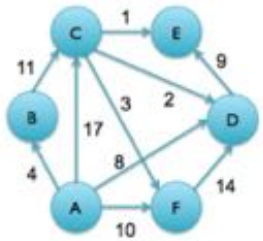


- Assume that Dikjstra's algorithm has correctly found the shortest path to the first N items, including from S to X and U (but not yet Y or V)
- It now decides the next closest node to visit is v, using the edge from u

Could there be some other shorter path to v (through y)?

No: cost(S->X->Y) must be greater than or equal to cost(S->U->V) (and therefore cost(S->X->Y->V) must be even greater) or it would be visiting node Y next

Dijkstra's Pseudocode




```
dijkstra(graph, start):
  distance = {}
  for v in graph.vertices:
      distance[v] = infinity
  distance[start] = 0
  unvisited = graph.vertices
  unvisited.remove(start)

  current = start
  while !unvisted.empty()
      for e in current.outgoing:
          v = e.toVertex
          if v in unvisited:
              d = distance[current] + e.weight
              if d < distance[v]:
                  distance[v] = d
      unvisited.remove(current)
      current = unvisited.findSmallestDistance()
      if distance[current] == infinity:
          break
```

Initialize distance to infinity except for start node

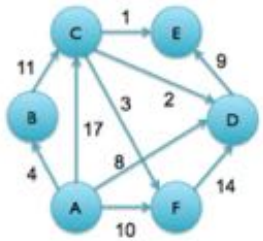
Initialize the set of unvisited nodes with everything except start

Update the distances from the current node

Advance smallest

In case >1 connected components

Dijkstra's Pseudocode




```
dijkstra(graph, start):
  distance = {}
  for v in graph.vertices:
      distance[v] = infinity
  distance[start] = 0
  unvisited = graph.vertices
  unvisited.remove(start)

  current = start
  while !unvisted.empty()
      for e in current.outgoing:
          v = e.toVertex
          if v in unvisited:
              d = distance[current] + e.weight
              if d < distance[v]:
                  distance[v] = d
      unvisited.remove(current)
      current = unvisited.findSmallestDistance()
      if distance[current] == infinity:
          break
```

Running time?

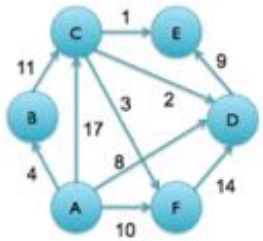
Explore every edge once,
Visit every node once

At every node, scan the unvisited nodes to find next smallest distance

$O(|E| + |V|^2)$

Can you do better?

Dijkstra's Pseudocode



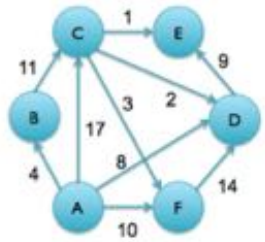

```
dijkstra(graph, start):
  distance = {}
  for v in graph.vertices:
      distance[v] = infinity
  distance[start] = 0
  unvisited = graph.vertices
  unvisited.remove(start)

  current = start
  while !unvisted.empty()
      for e in current.outgoing:
          v = e.toVertex
          if v in unvisited:
              d = distance[current] + e.weight
              if d < distance[v]:
                  distance[v] = d
      unvisited.remove(current)
      current = unvisited.findSmallestDistance()
      if distance[current] == infinity:
          break
```

If only there was a data structure that would let us find the minimum element very quickly ;-)

HEAP!

More specifically: MinPriorityQueue!

Dijkstra with Priority Queue




```
dijkstra(graph, start):
  distance = {}

  for v in graph.vertices:
      if v == start
          distance[v] = 0
      else
          distance[v] = infinity
      PQ.add_with_priority(v, distance[v])

  while !PQ.empty()
      cur = PQ.extract_min()
      if distance[current] == infinity:
          break
      for e in current.outgoing:
          v = e.toVertex
          if PQ.has(v):
              d = distance[current] + e.weight
              if d < distance[v]:
                  distance[v] = d
                  PQ.decrease_priority(v,d)
```

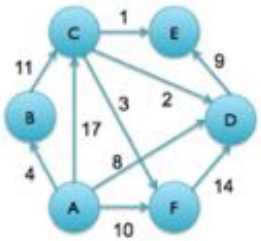
A min-priority queue that supports extract_min(), add_with_priority(), decrease_priority() and has()

Now Dikstra will run in O(|E| lg |V|)

Normally checking if a node is in the PQ would require O(n) but can store references to make it fast

Notice we have to do work on every edge

Min-priority queue

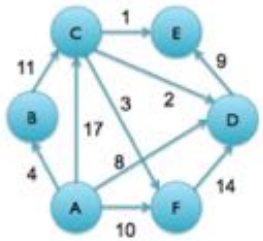



```
public interface PriorityQueue<T extends Comparable<T>> {
    void insert(T t);
    void remove() throws EmptyQueueException;
    T top() throwsEmptyQueueException;
    boolean empty();
}
```

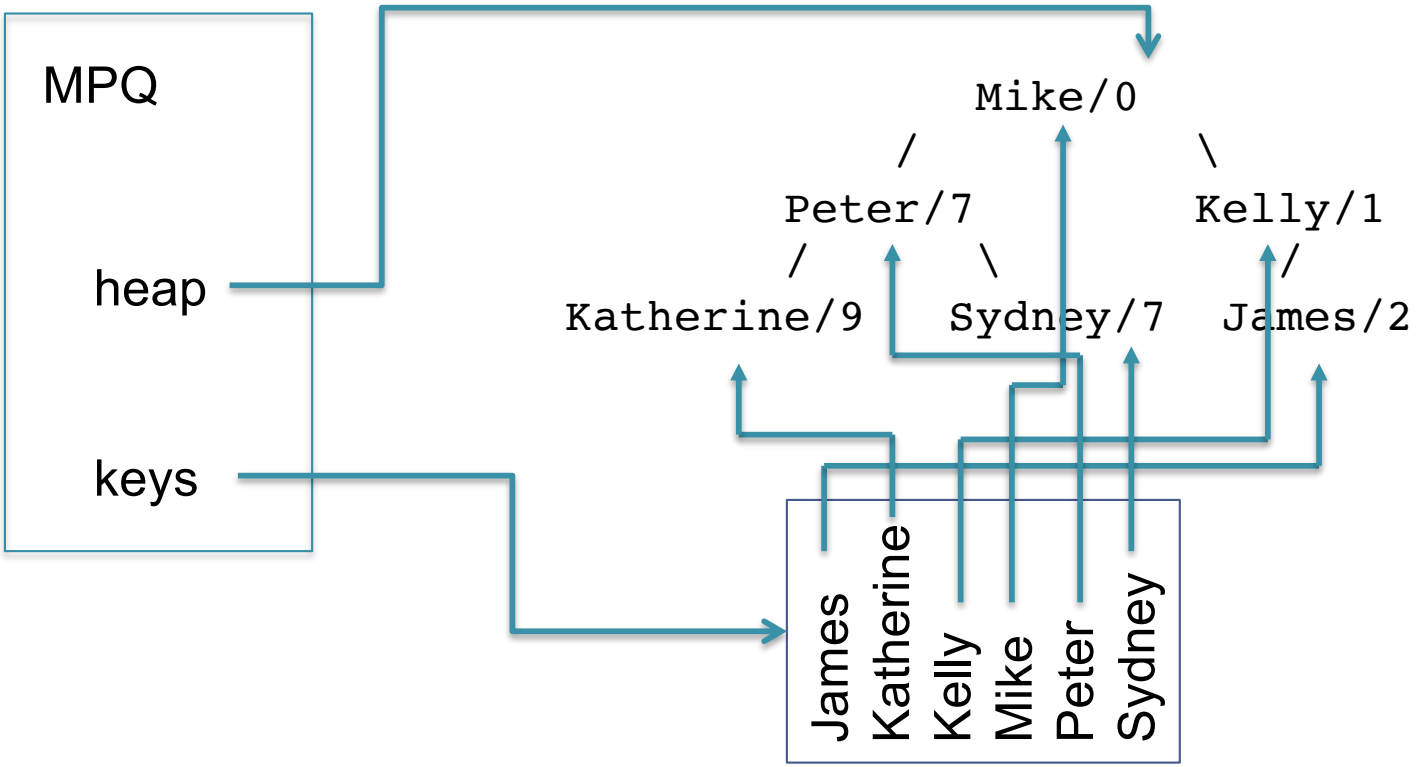
```
public interface MinPriorityQueue<K, P extends Comparable<T>> {
    void addWithPriority(K key, P priority);
    void decreasePriority(K key, P newp) throws InvalidItem;
    T extractMin() throws EmptyQueueException;
    boolean empty();
    boolean has();
}
```

Allows for priorities to be reset for keys inside the PQ

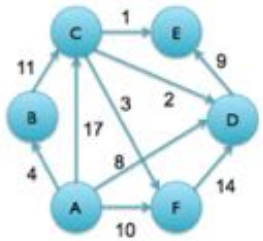
Min-priority queue



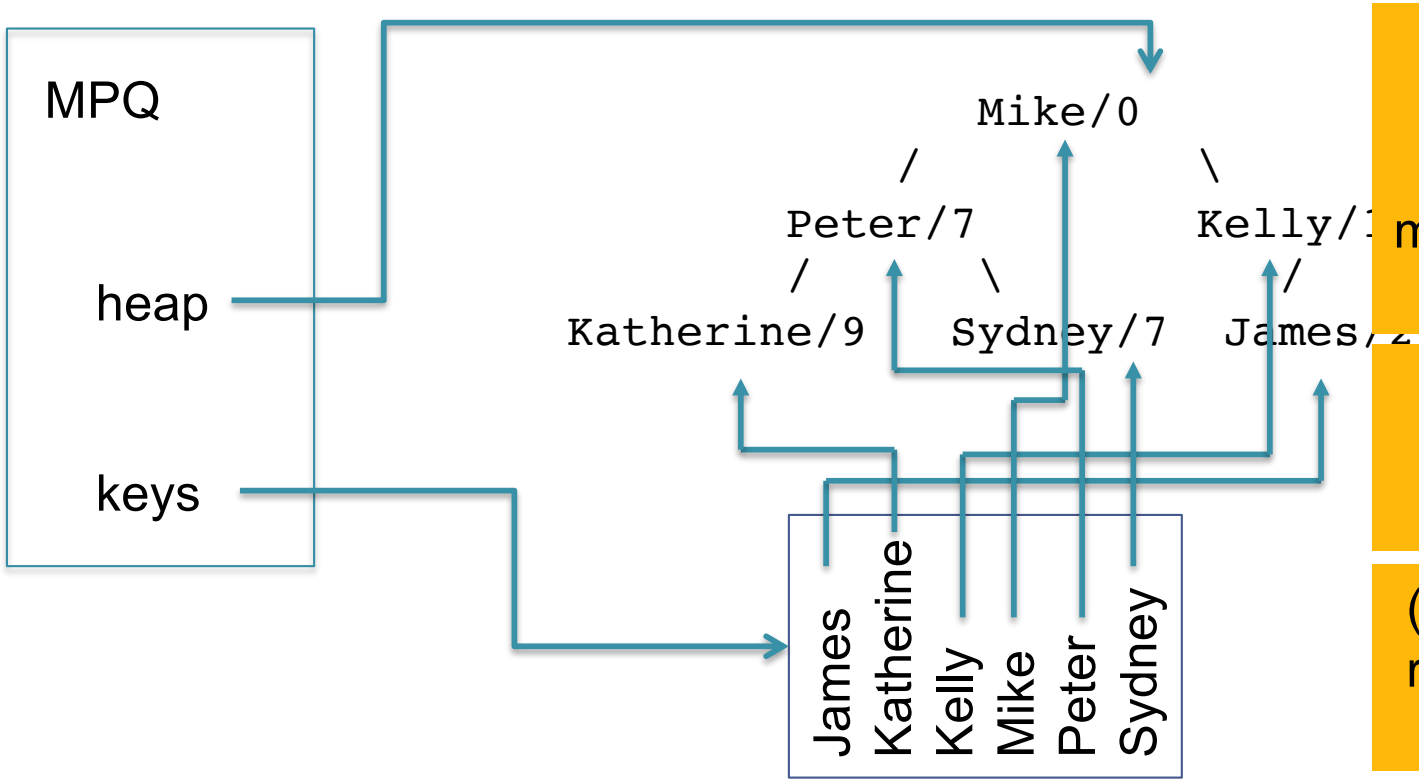

```
public interface MinPriorityQueue<K, P extends Comparable<T>> {
    void addWithPriority(K key, P priority);
    void decreasePriority(K key, P newp) throws InvalidItem;
    T extractMin() throws EmptyQueueException;
    boolean empty();
    boolean has();
}
```



Min-priority queue




```
public interface MinPriorityQueue<K, P extends Comparable<T>> {
    void addWithPriority(K key, P priority);
    void decreasePriority(K key, P newp) throws InvalidItem;
    T extractMin() throws EmptyQueueException;
    boolean empty();
    boolean has();
}
```



How to implement?

(1) If keys are integers in the range 0 to n, make an array of references

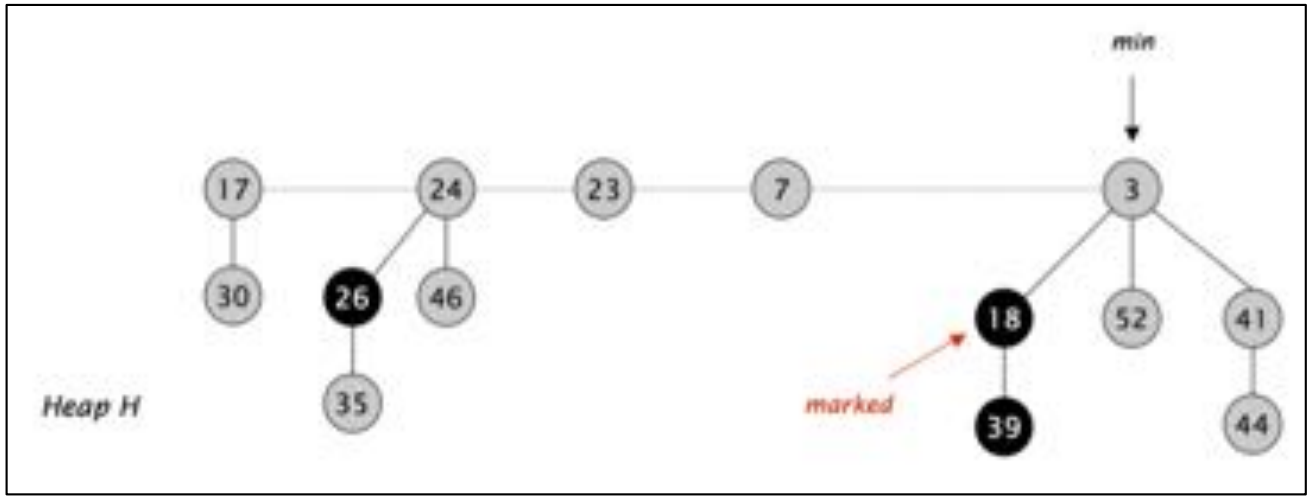
(2) Use a HashMap from keys to hash

(3) Give client a reference, store in graph node

Fibonnaci Heap

Special form of heaps specifically designed to allow fast updates to values

- ***Set of heap-ordered trees*** (value(n) < value(n.children))
- Maintain pointer to overall minimum element
- Set of marked nodes used to track heights of certain trees



	Find-min	Delete-min	Insert	Decrease-key	Merge
Binary Heap	O(1)	O(lg n)	O(lg n)	O(lg n)	O(n)
Fibonnaci Heap	O(1)	O(lg n)	O(1)	O(1)	O(1)

Reduces Dijkstra's run time to O(|E| + |V| lg |V|)

Not on final ☺

Next Steps

1. Reflect on the magic and power of Sorting!

2. Assignment 10 due on Friday December 7 @ 10pm