CS 600.226: Data Structures

Michael Schatz

Dec 3 2018

Lecture 37: Minimum Spanning Trees

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.

ne

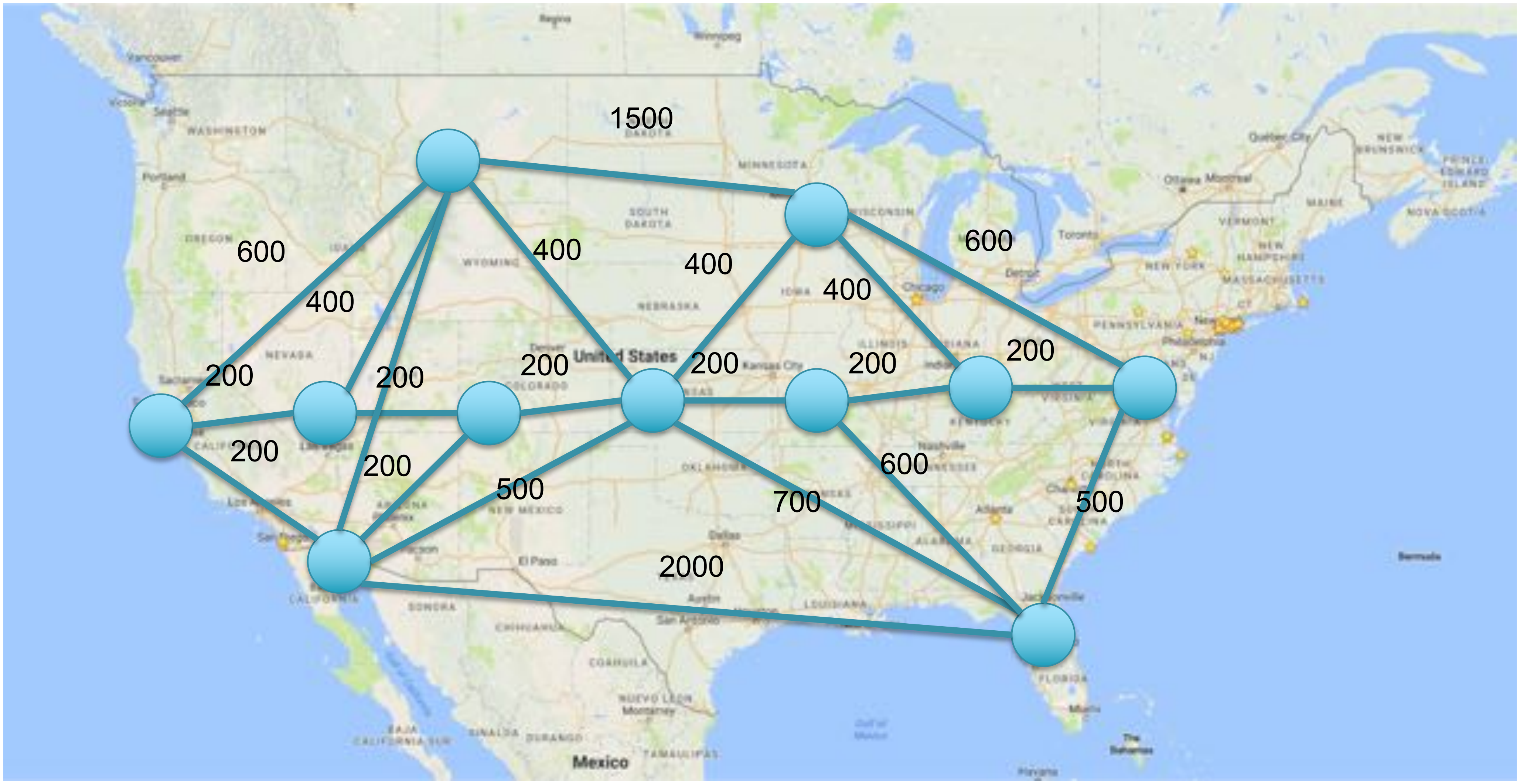
vie-

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.

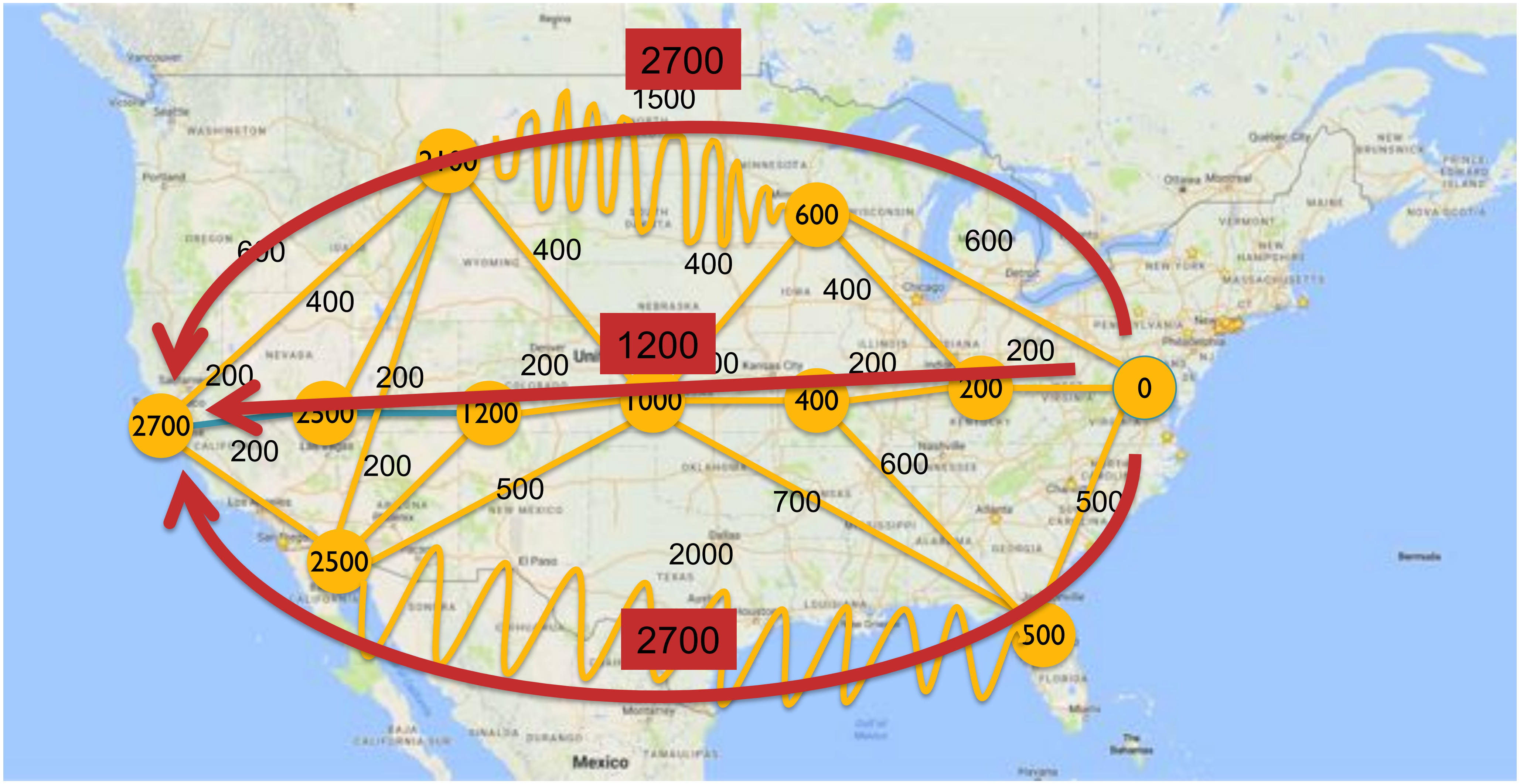
Remember: javac –Xlint:all & checkstyle *.java & Junit
(No JayBee)

Part 1: Dijkstra’s Algorithm aka Shortest Path Revisited

Why doesn't BFS work for maps?



Why doesn't BFS work for maps?

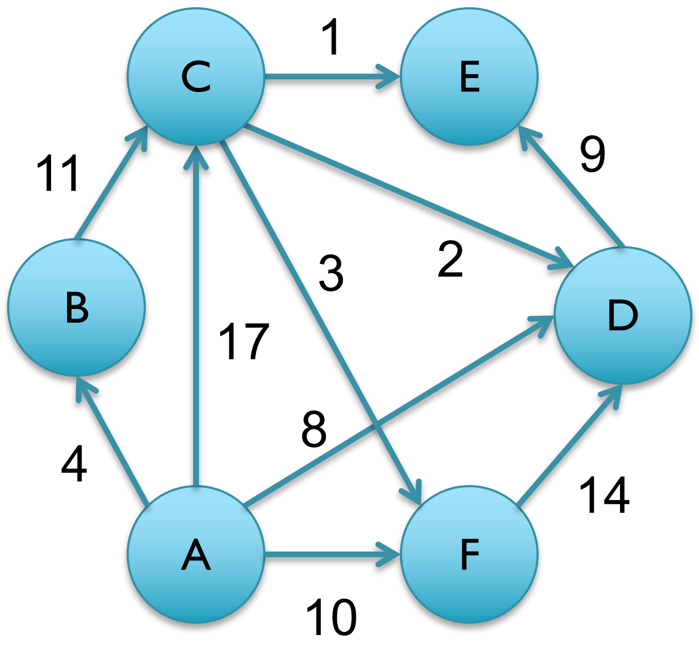


Fewest hops != Shortest distance

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

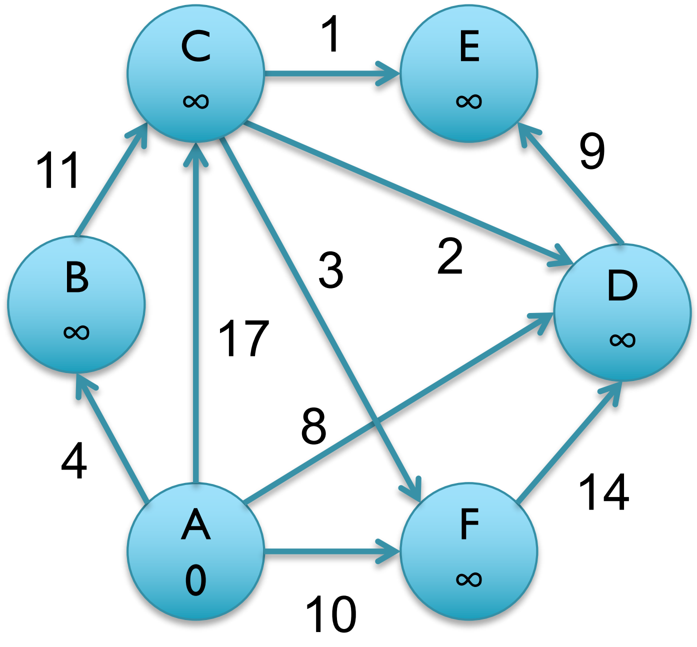


Dijkstra's Algorithm

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

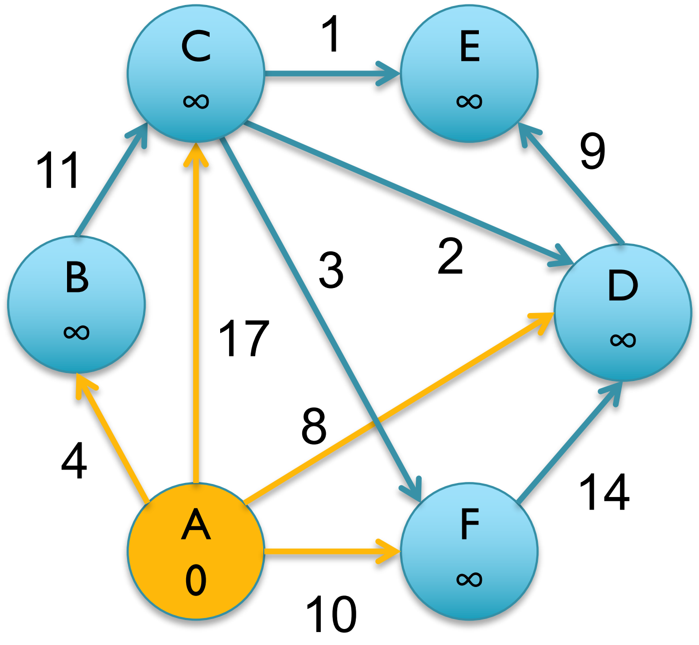


Dijkstra's Algorithm

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

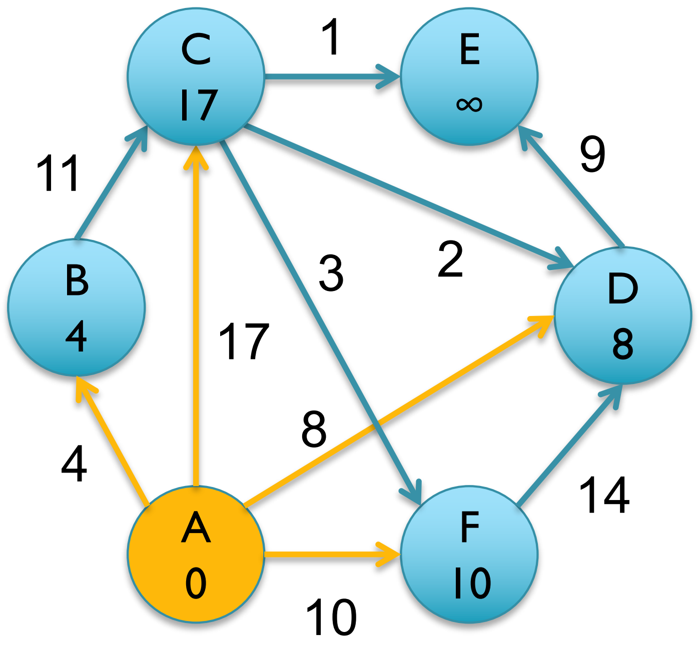


Dijkstra’s Algorithm

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

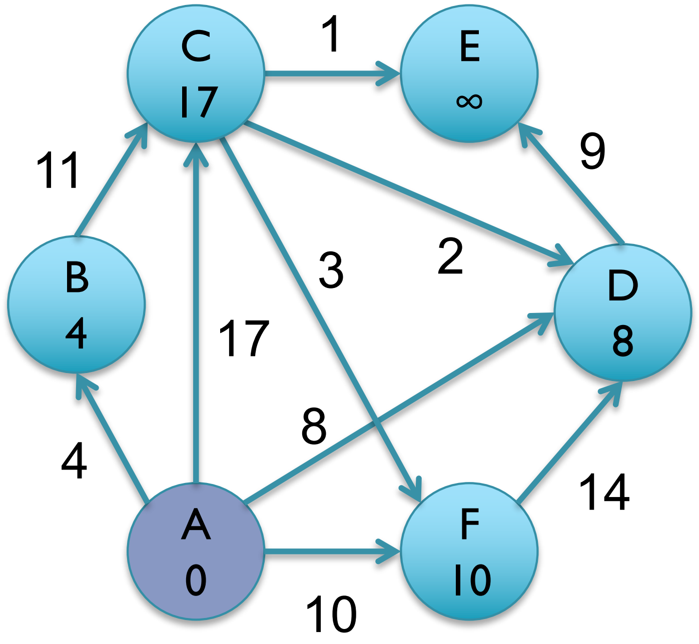


Dijkstra's Algorithm

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

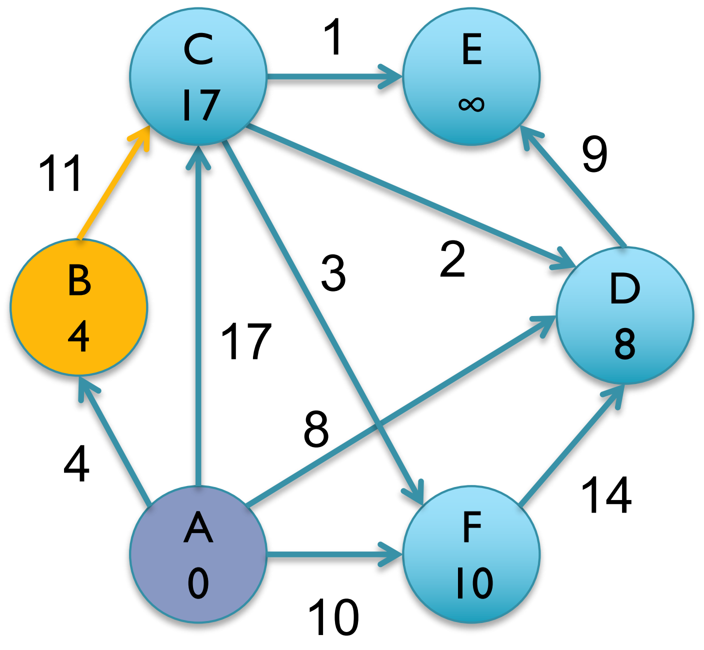


Dijkstra's Algorithm

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

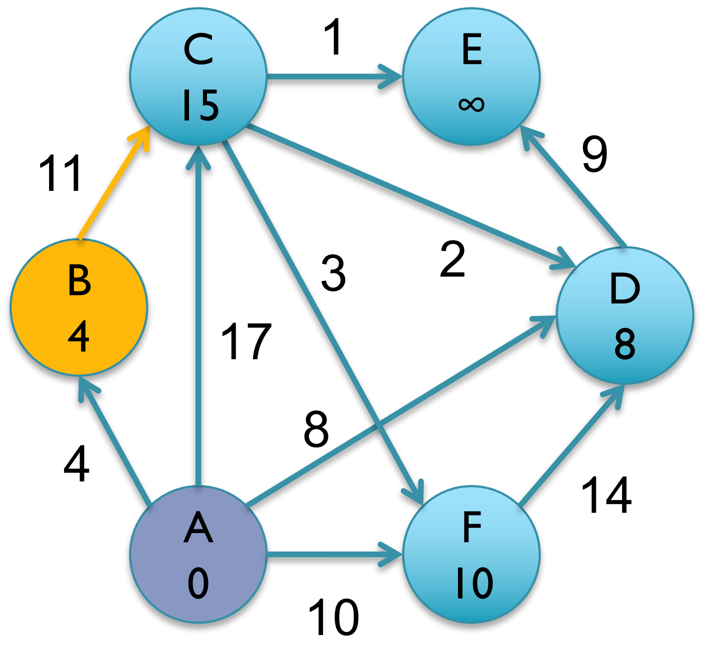


Dijkstra's Algorithm

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

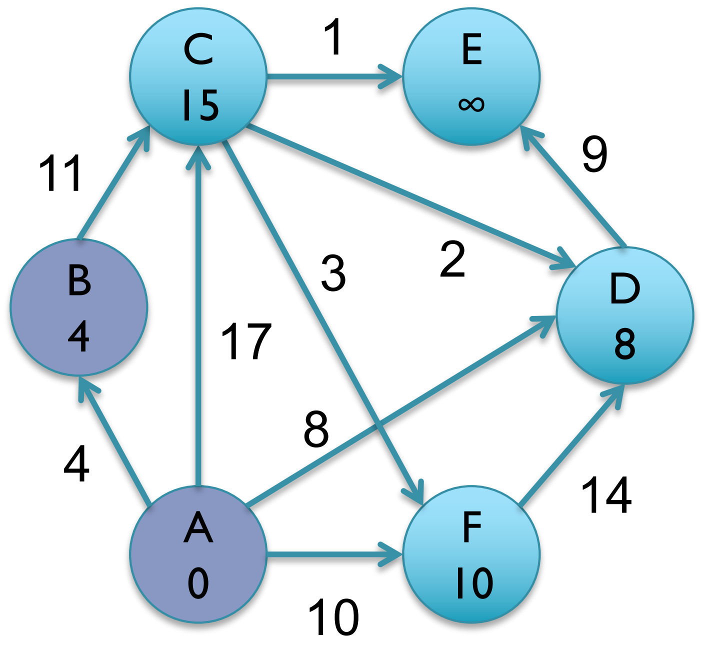


Dijkstra's Algorithm

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

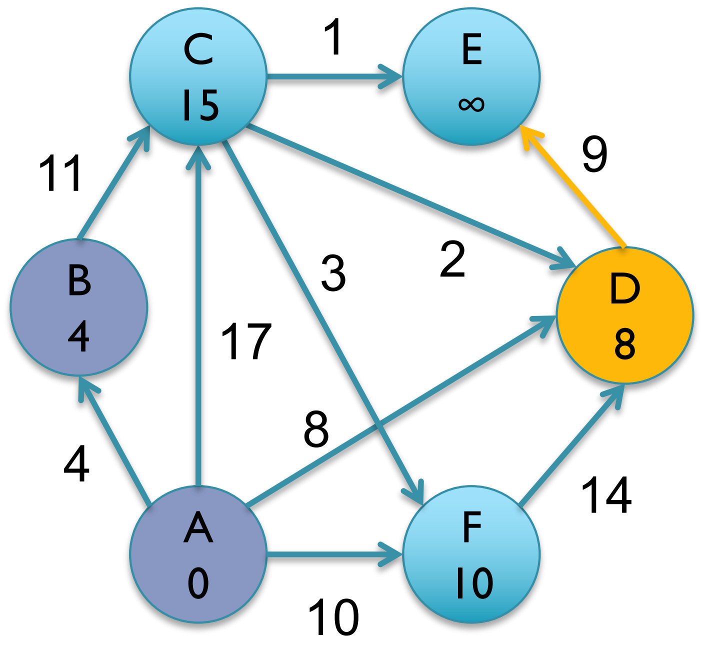


Dijkstra's Algorithm

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

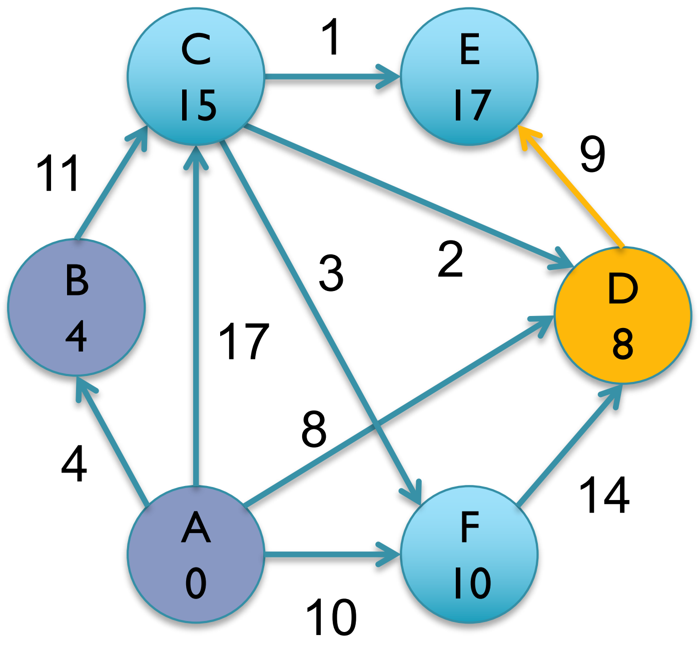


Dijkstra's Algorithm

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

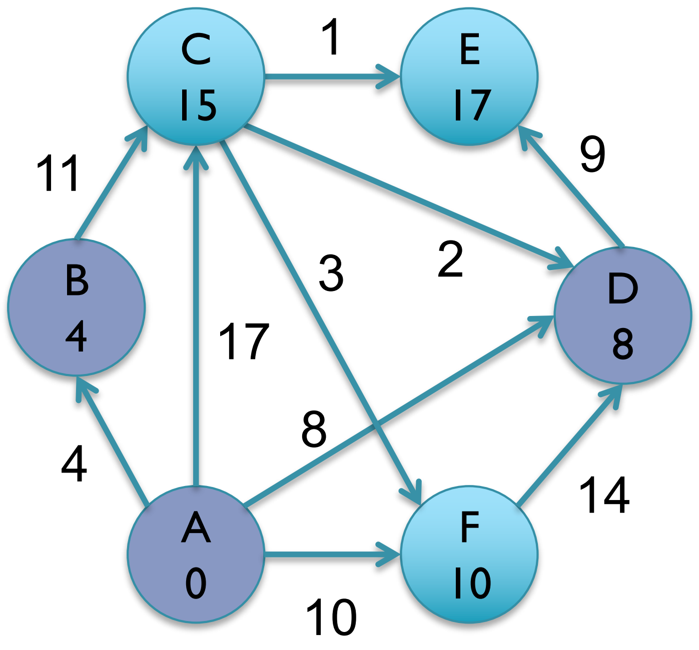


Dijkstra's Algorithm

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

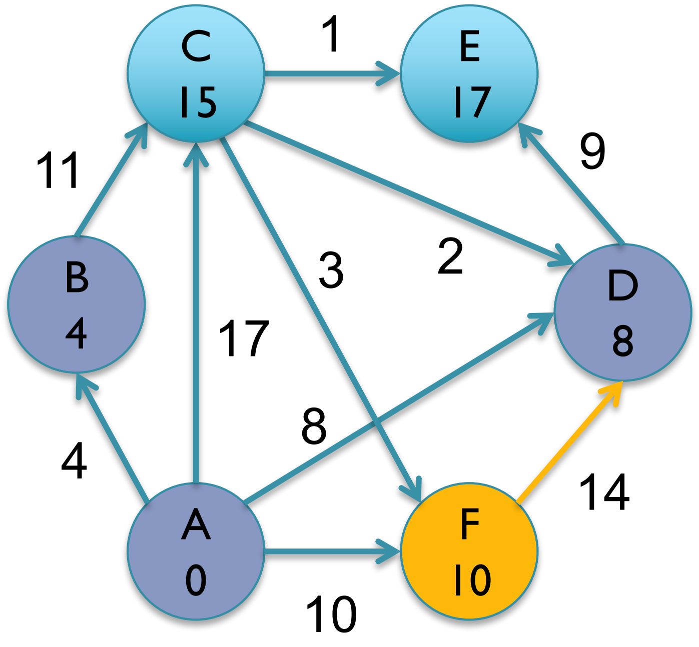


Dijkstra's Algorithm

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

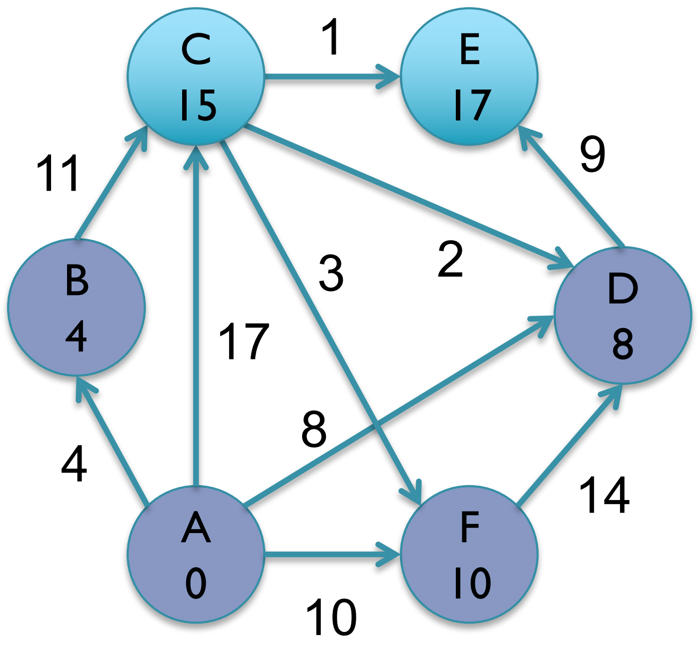


Dijkstra's Algorithm

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

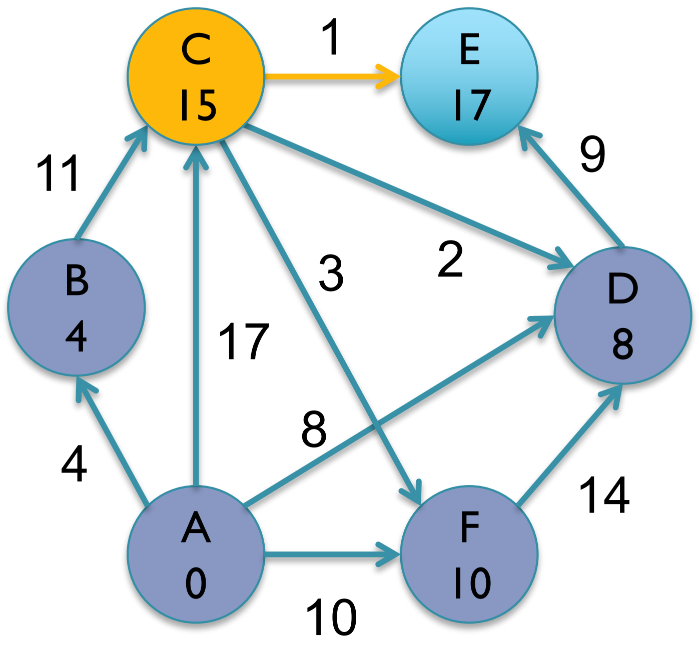


Dijkstra's Algorithm

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

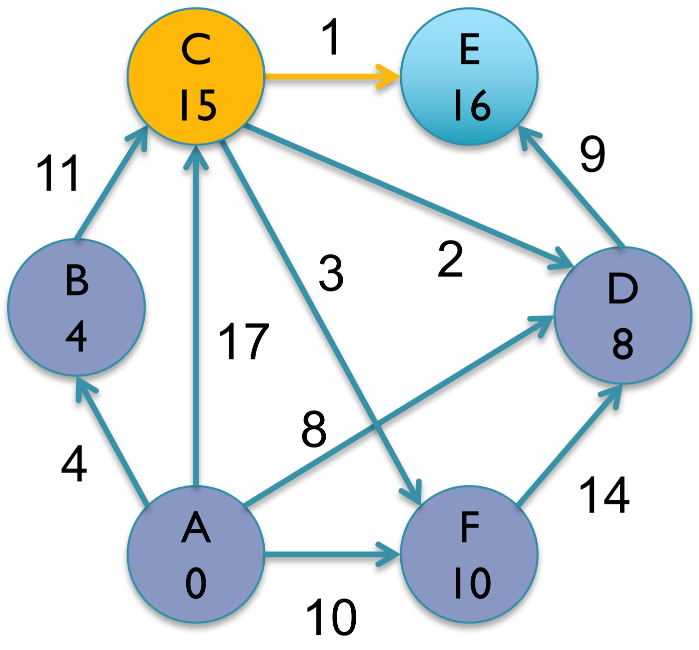


Dijkstra's Algorithm

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

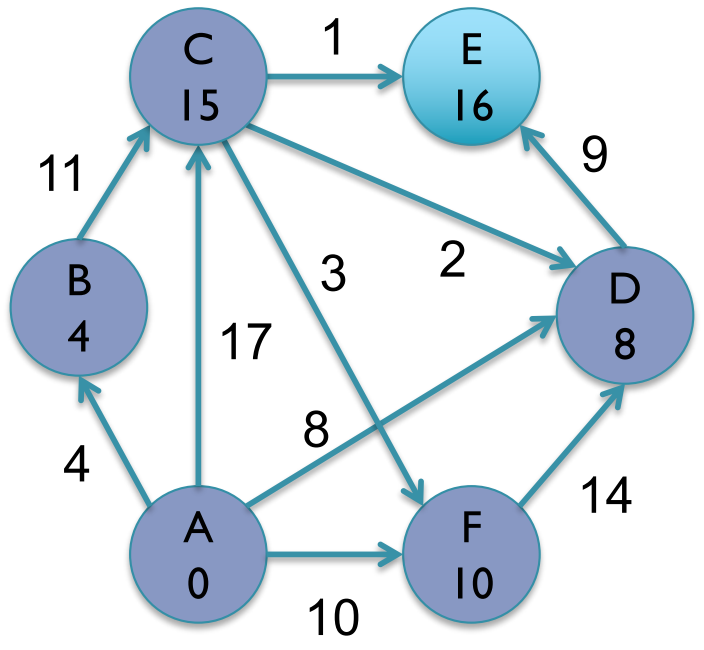


Dijkstra's Algorithm

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

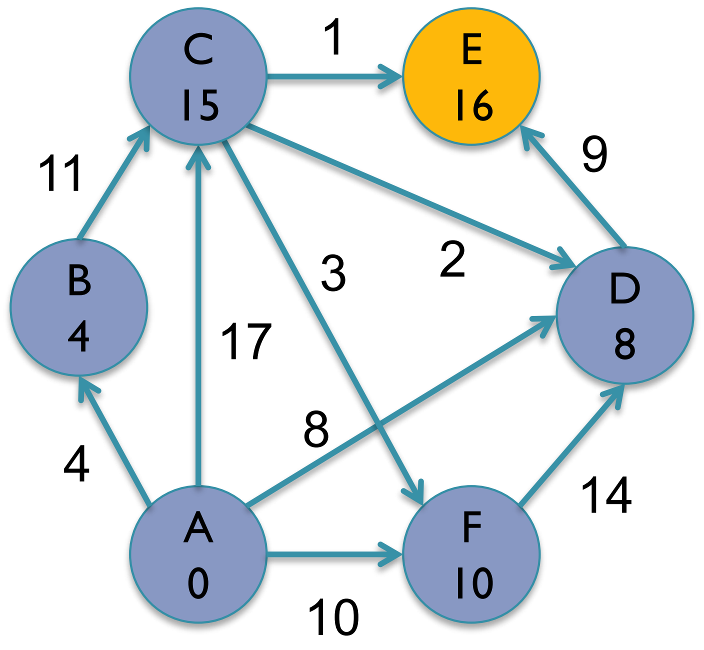


Dijkstra's Algorithm

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

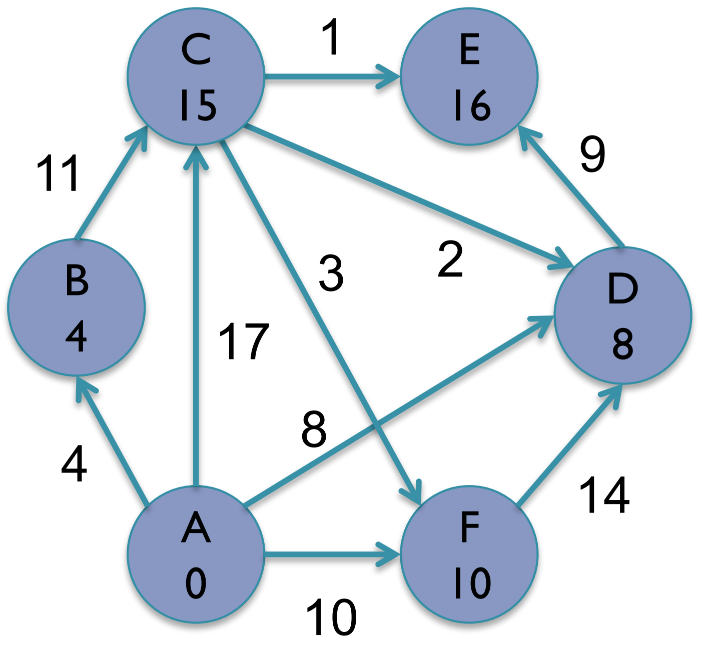
As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

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

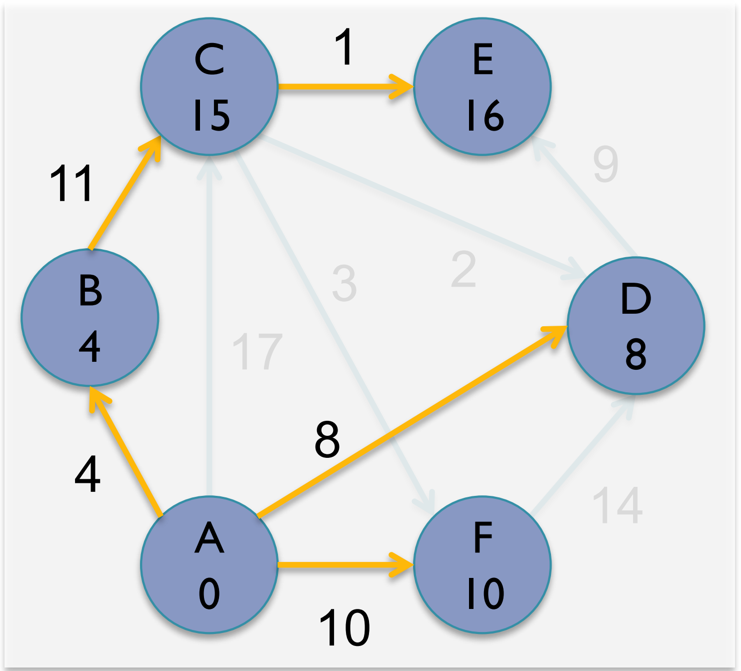
As before, initialize distance to start (A) as 0, and "estimated distance" to every other node as infinity

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

No more unvisited nodes! Done

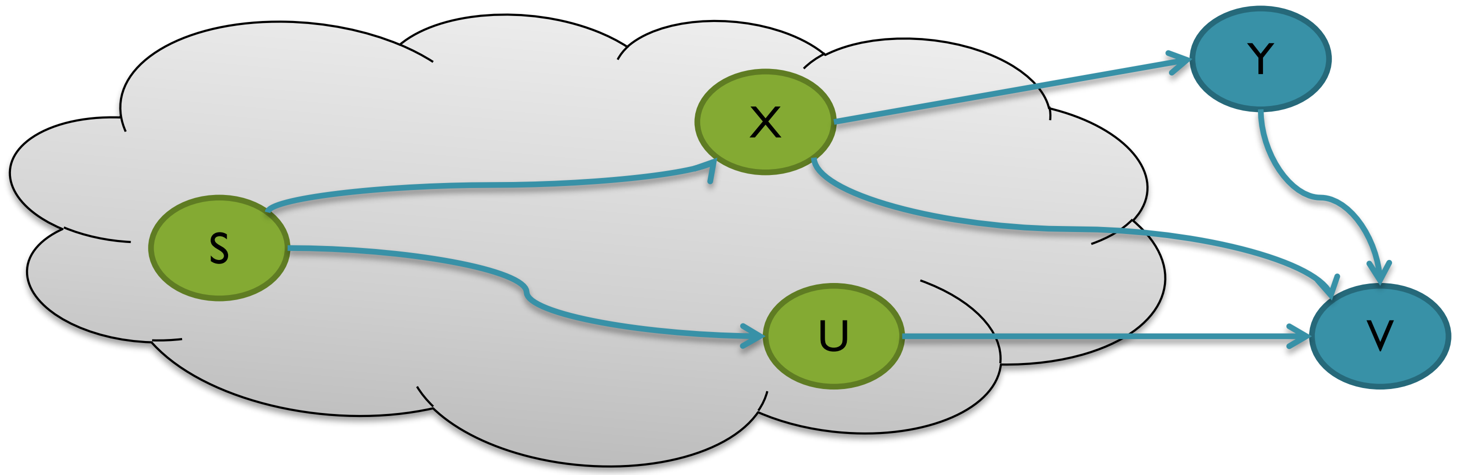
If we are looking for a path between a particular pair, could we terminate early?

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



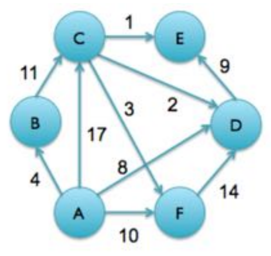
- Assume that Dikjstra’s algorithm has correctly found the shortest path to the first N items, including from S to X and U (but not yet Y or V)
- It now decides the next closest node to visit is v, using the edge from u

Could there be some other shorter path to v?

No: cost(S->X->V) must be greater than or equal to cost(S->U->V) or it would have already revised the cost of v to go through X

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 with Priority Queue



BFS

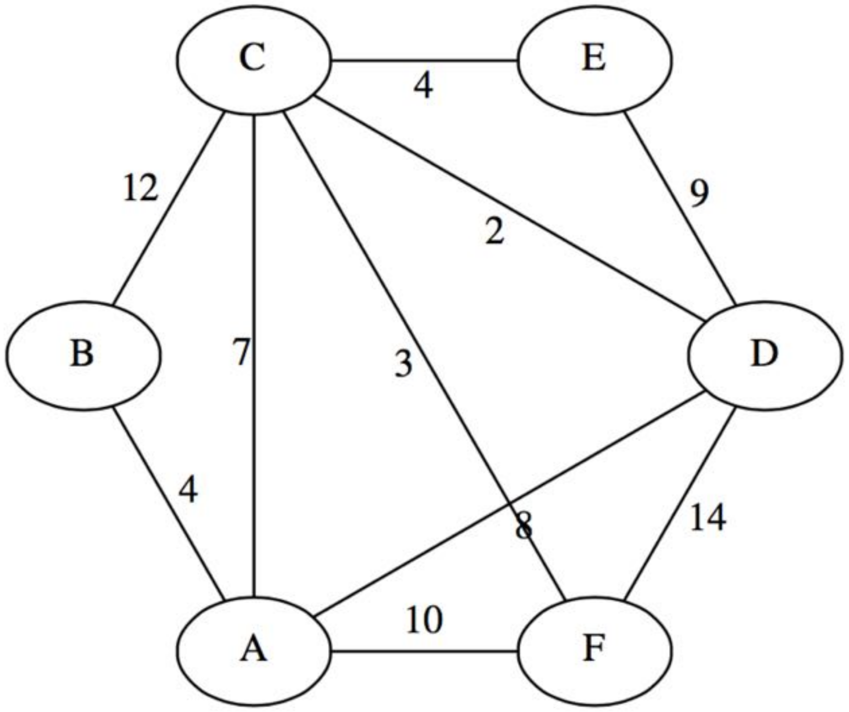
```
BFS(start, stop)
// initialize all nodes dist = -1
start.dist = 0
list.addEnd(start)
while (!list.empty())
   cur = list.getBegin()
   if (cur == stop)
      print cur.dist;
   else
      foreach child in cur.children
         if (child.dist == -1)
             child.dist = cur.dist+1
          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

Dijstra

```
Dijstra(start, stop)
// initialize all nodes dist = -1
start.dist = 0
PQ.add (start)
while (!Q.empty())
   cur = PQ.getMin()
   if (cur == stop)
      print cur.dist;
   else if cur.visited
        // already visited, nothing to do
   else
      cur.visted = true
      foreach e in cur.children
        d = cur.dist + e.weight
        if (d < e.child.dist)
             e.child.dist = d
             PQ.add (e.child, e.child.dist)
```

Part 2: Minimum Spanning Trees

Long Distance Calling



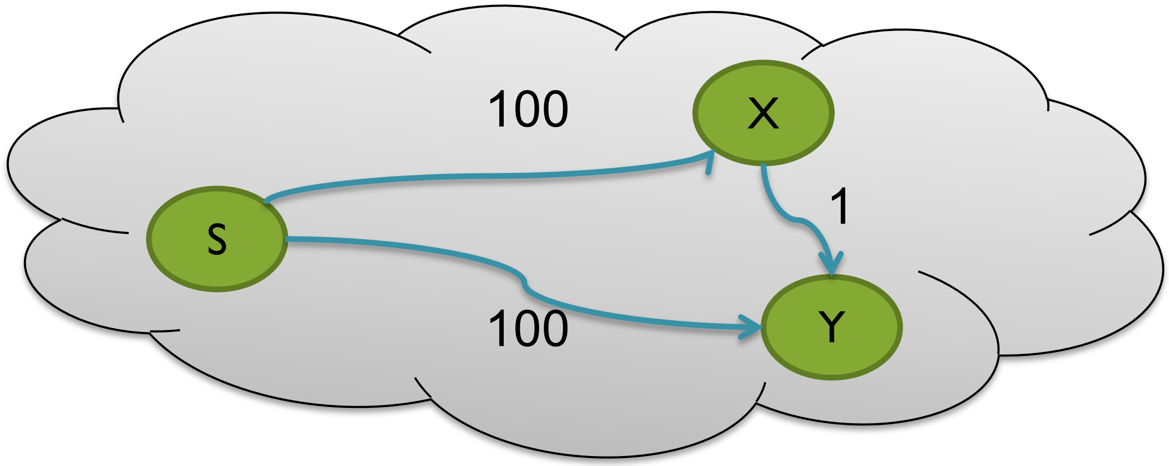
Supposes it costs different amounts of money to send data between cities A through F. Find the least expensive set of connections so that anyone can send data to anyone else.

Given an undirected graph with weights on the edges, find the subset of edges that (a) connects all of the vertices of the graph and (b) has minimum total costs

This subset of edges is called the minimum spanning tree (MST)

Removing an edge from MST disconnects the graph, adding one forms a cycle

Dijkstra's != MST



Dijkstra's will build the tree S->X, S->Y
(tree visits every node with shortest paths from S to every other node)

but the MST is S->X, X->Y
(tree visits every node and minimizes the sum of the edges)

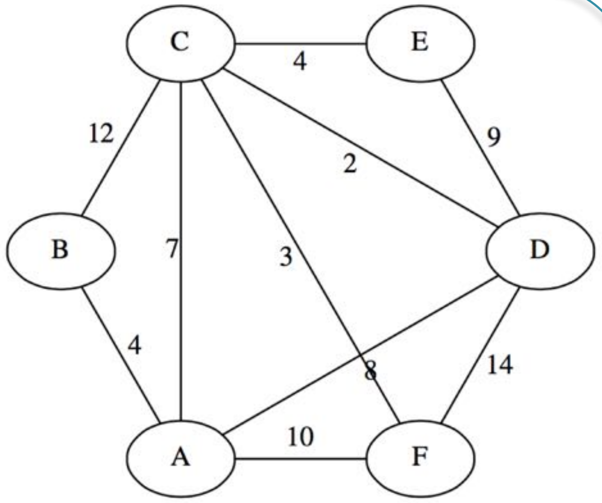
Any ideas?

Prim's Algorithm

1. ***Pick an arbitrary starting vertex and add it to set F. Add all of the other vertices to set Q***

Every vertex will be included eventually, so doesn't matter where you start

F={A}   Q={B,C,D,E,F}

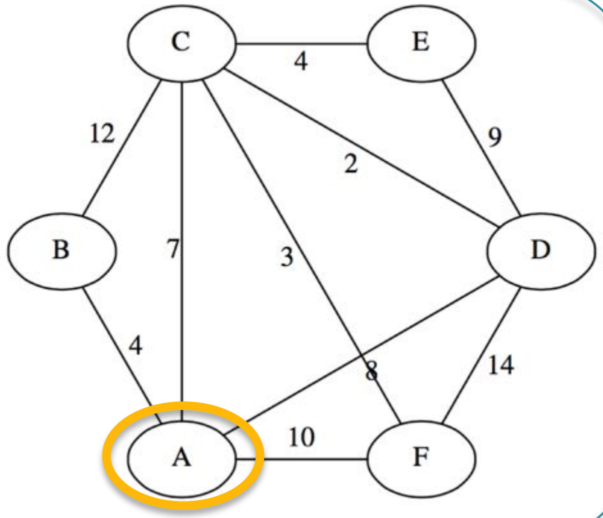


Prim's Algorithm

1. Pick an arbitrary starting vertex and add it to set F. Add all of the other vertices to set Q

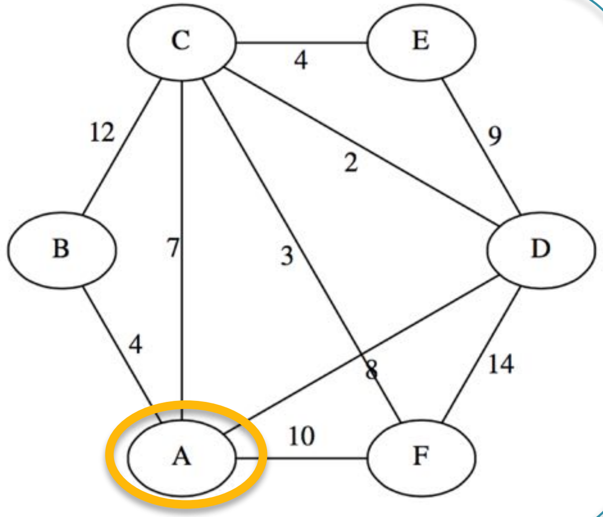
Every vertex will be included eventually, so doesn't matter where you start

F={A}   Q={B,C,D,E,F}



2. While Q is not empty, pick an edge from F to Q with minimum cost

A-B: 4   <- pick me
A-C: 7
A-D: 8
A-F: 10

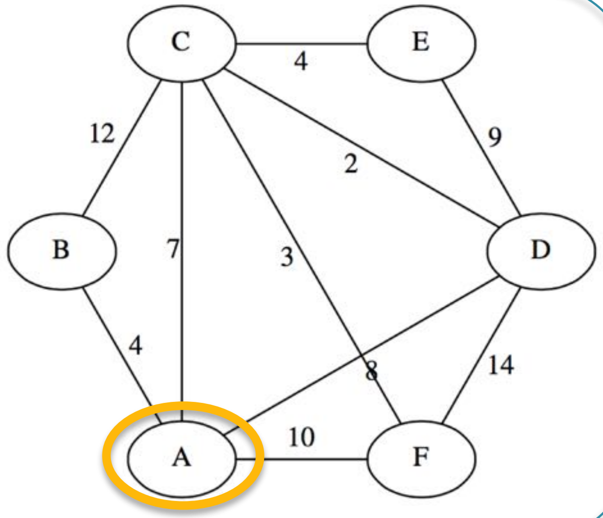


Prim's Algorithm

1. ***Pick an arbitrary starting vertex and add it to set F. Add all of the other vertices to set Q***

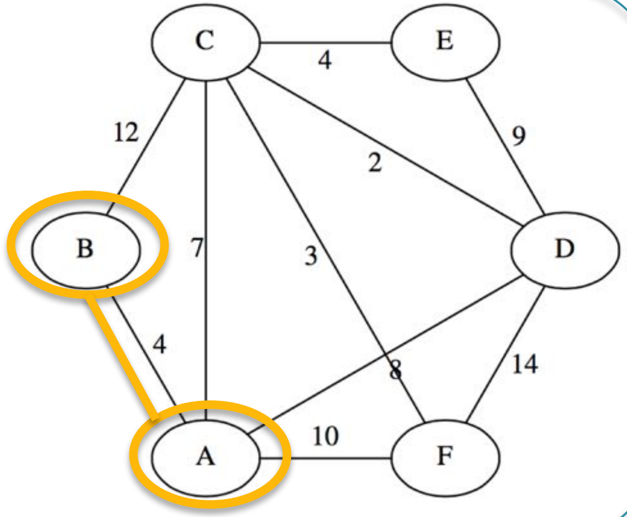
Every vertex will be included eventually, so doesn't matter where you start

F={A}   Q={B,C,D,E,F}



2. While Q is not empty, pick an edge from F to Q with minimum cost

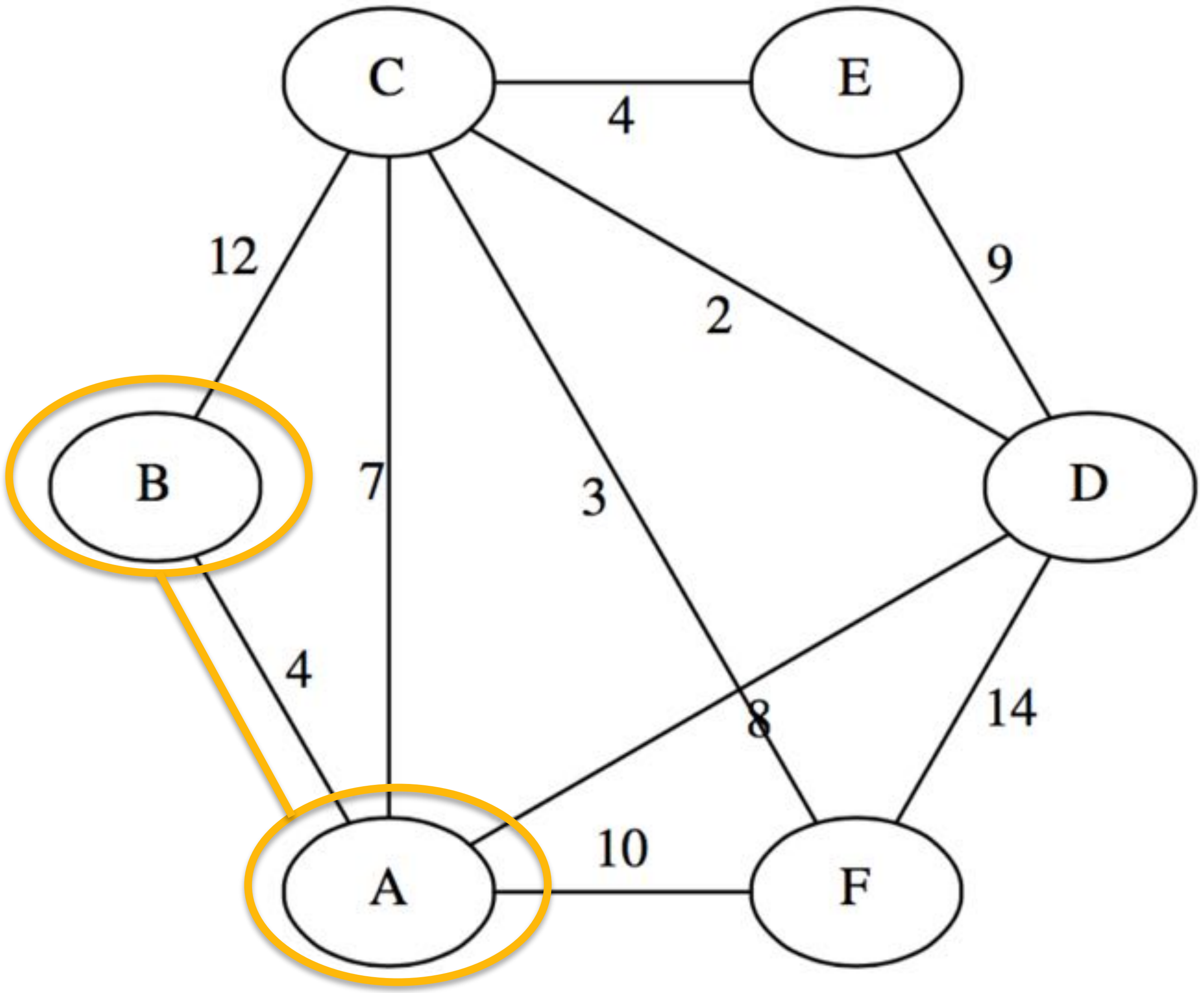
A-B: 4   <- pick me
A-C: 7
A-D: 8
A-F: 10



Prim's Algorithm

3. Repeat!

A-B

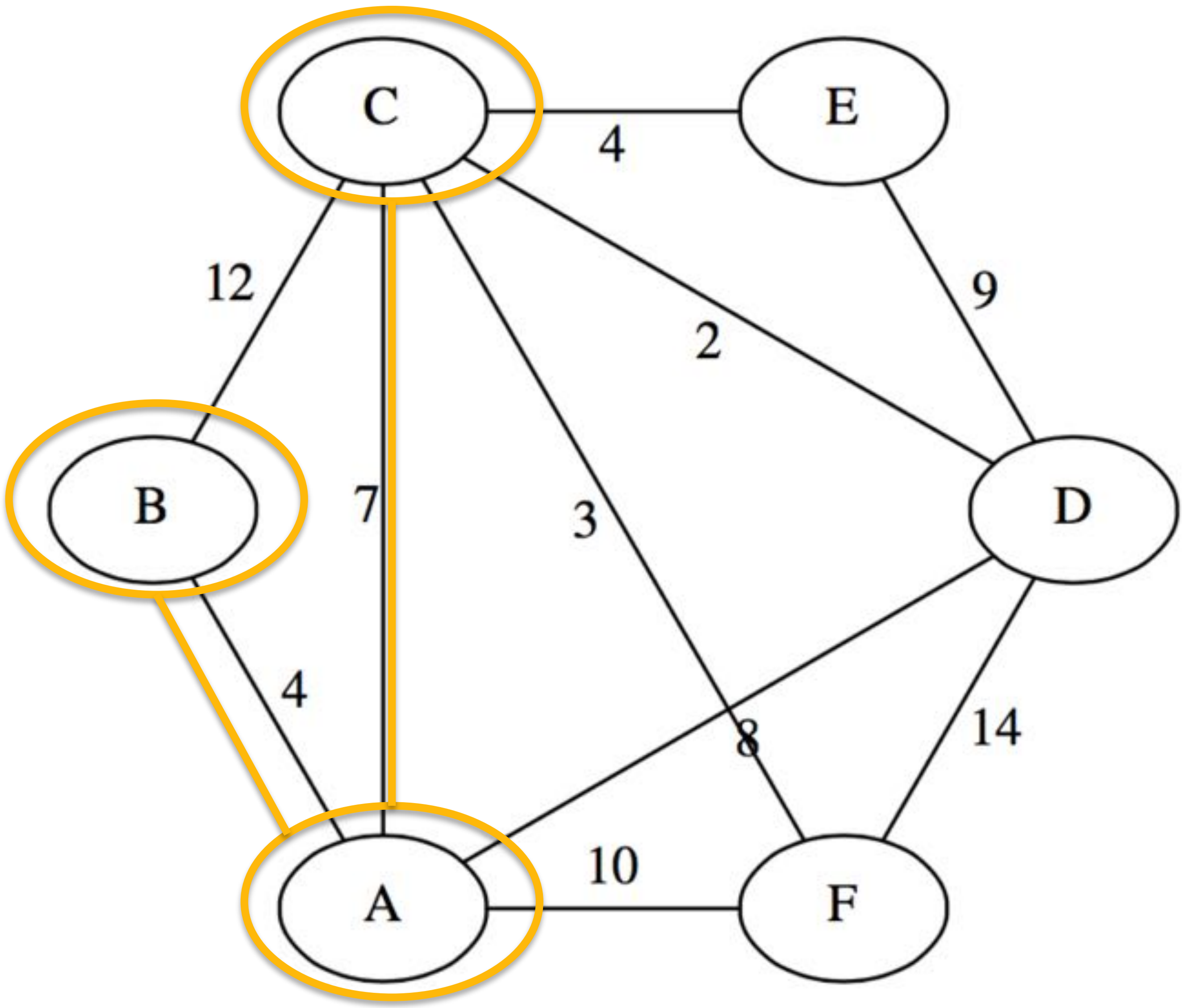


Prim's Algorithm

3. Repeat!

A-B

A-C



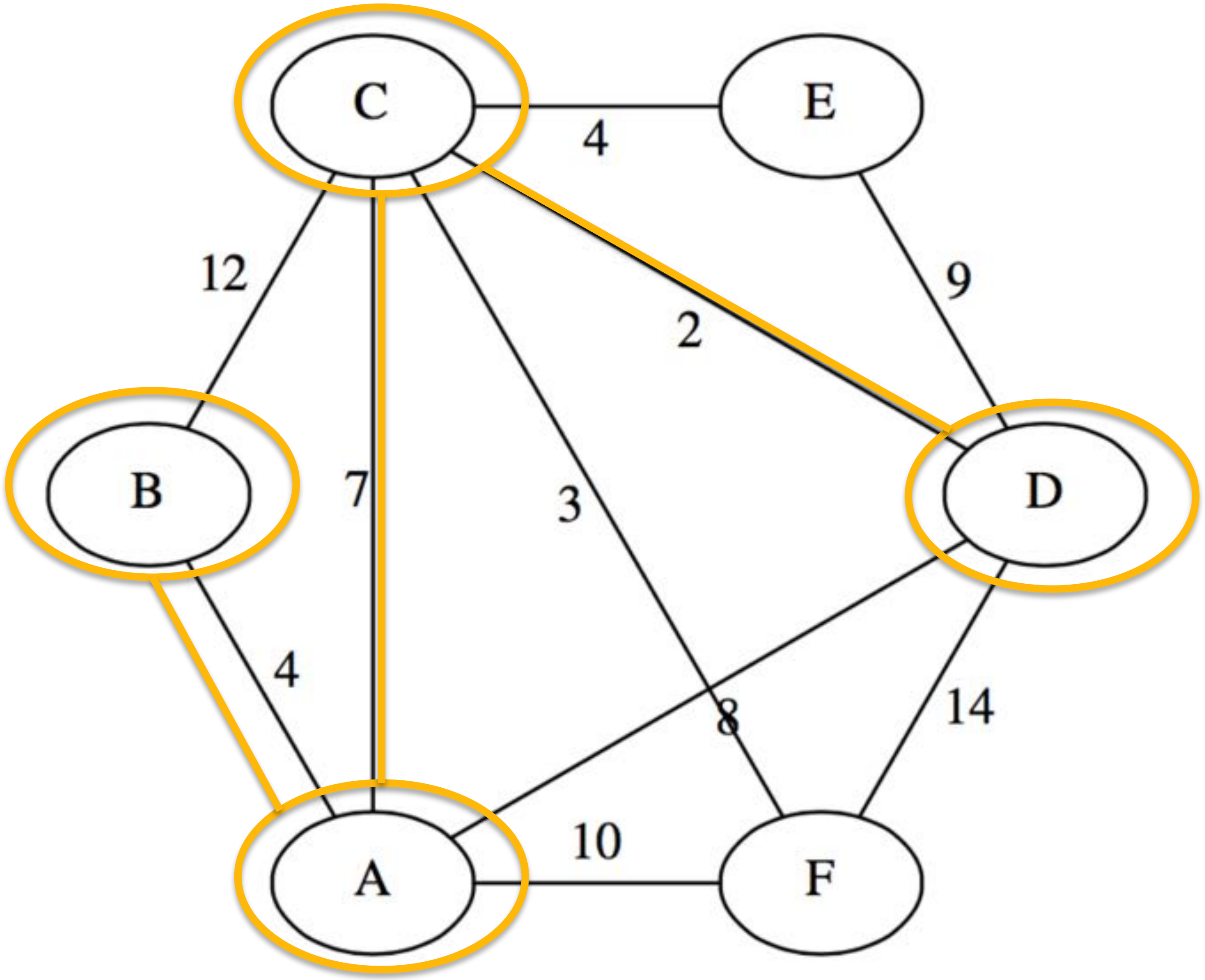
Prim's Algorithm

3. Repeat!

A-B

A-C

C-D



Prim's Algorithm

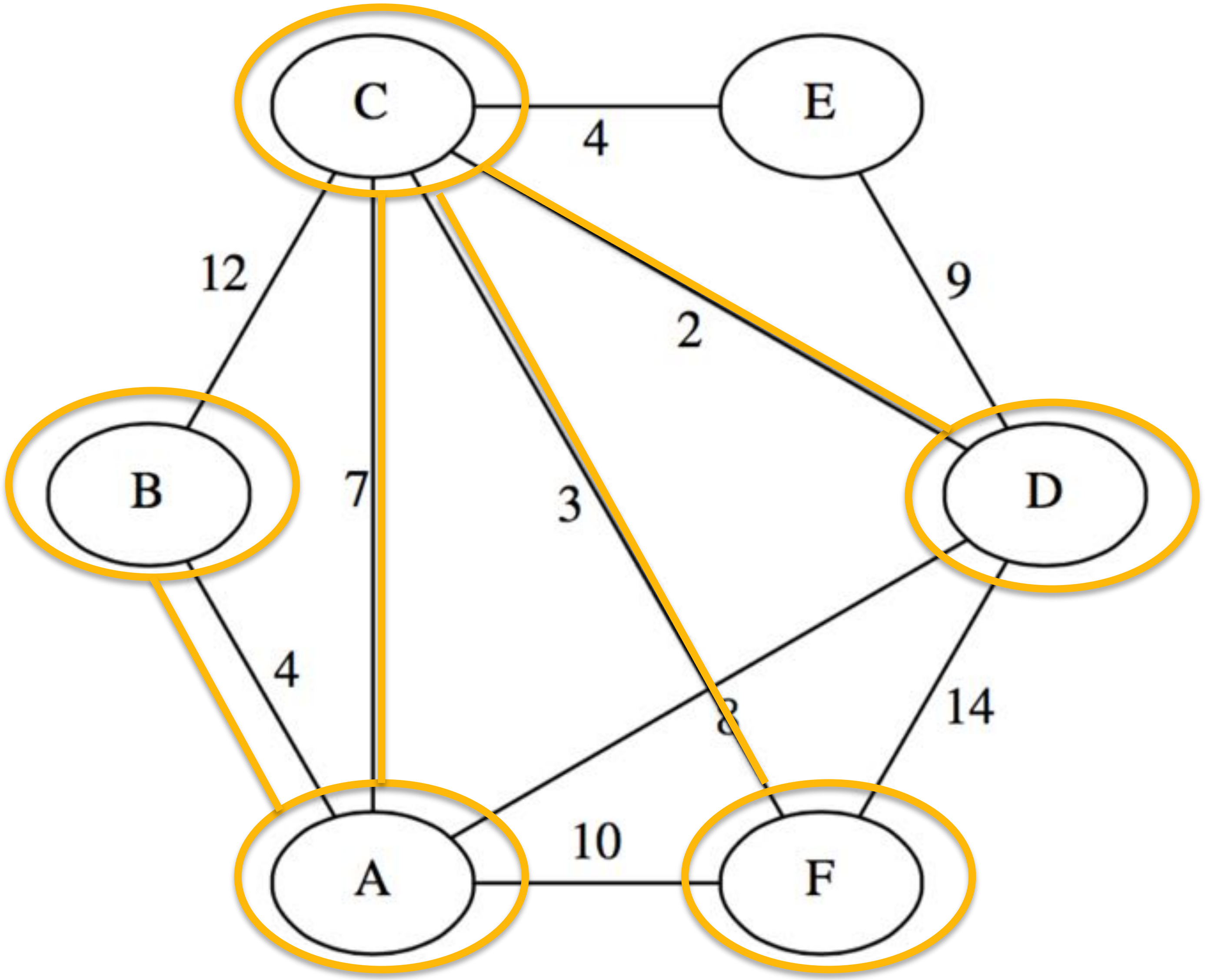
3. Repeat!

A-B

A-C

C-D

C-F



Prim's Algorithm

3. Repeat!

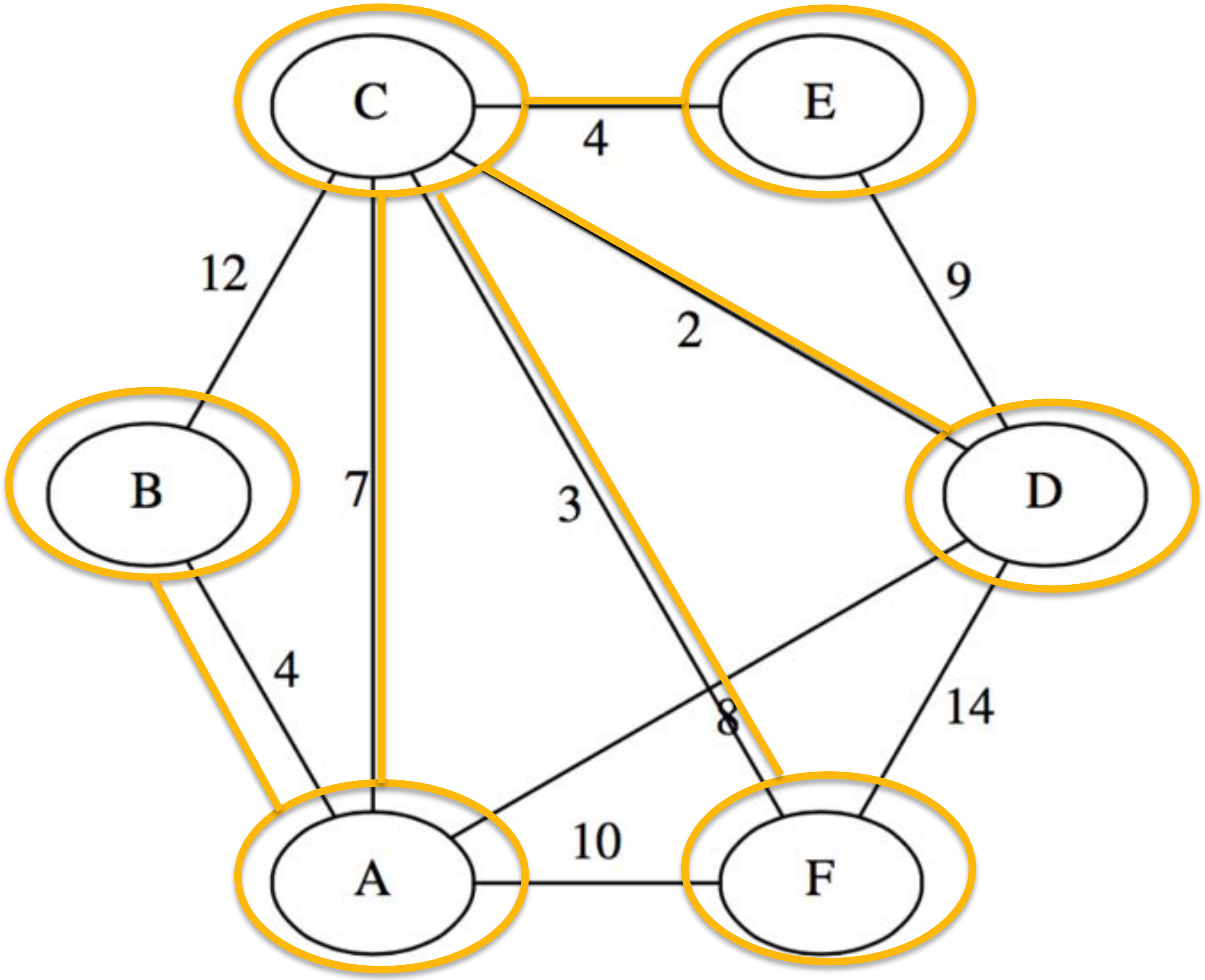
A-B

A-C

C-D

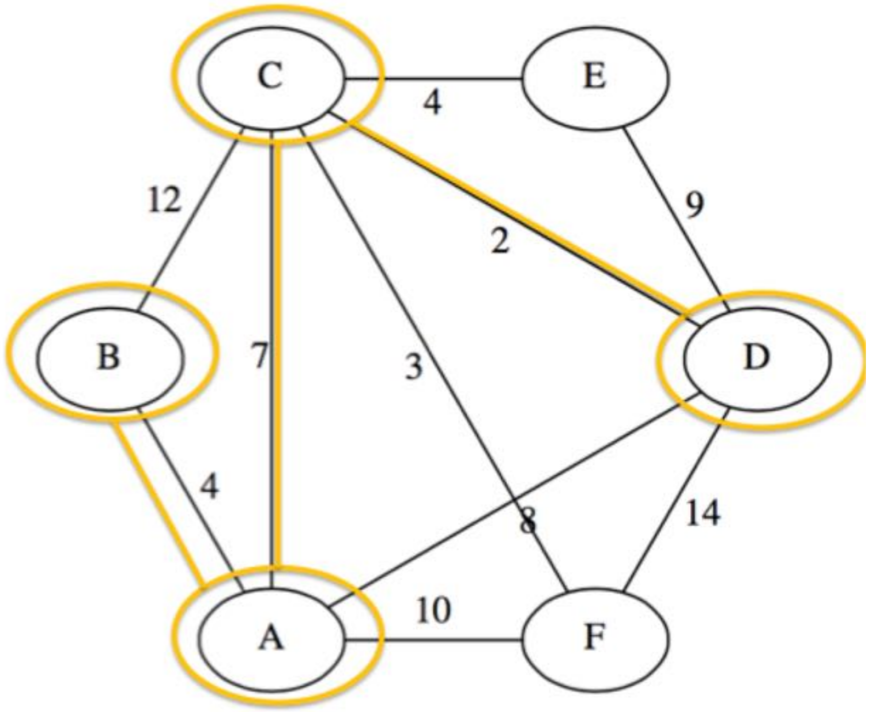
C-F

C-E



By making a set of simple local choices, it finds the overall best solution
The greedy algorithm is optimal ☺

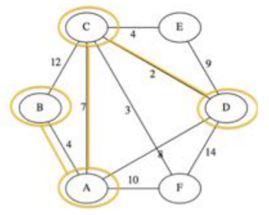
Prim’s Algorithm



Prim’s Algorithm Sketch

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge:
    - Of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and add it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

Prim’s Algorithm



Prim’s Pseudo-code

```
1. For each vertex v,
     • Compute C[v] (the cheapest cost of a connection to v from the MST)
       and an edge E[v] (the edge providing that cheapest connection).
     • Initialize C[v] to ∞ and E[v] to null indicating that there is no edge
       connecting v to the MST
2. Initialize an empty forest F (set of trees) and a set Q of vertices that have
   not yet been included in F (initially, all vertices).
3. Repeat the following steps until Q is empty:
     1. Find and remove a vertex v from Q with the minimum value of C[v]
     2. Add v to F and, if E[v] is not null, also add E[v] to F
     3. Loop over the edges vw connecting v to other vertices w. For each
        such edge, if w still belongs to Q and vw has smaller weight than
        C[w], perform the following steps:
          1. Set C[w] to the cost of edge vw
          2. Set E[w] to point to edge vw.
4. Return F
```

How fast is the naïve version? $O(|V|^2)$

What data structures do we need to make it fast? HEAP

Prim's Algorithm

Faster Prim's Pseudo-code

```
1. Add the cost of all of the edges to a heap
2. Repeatedly pick the next smallest edge (u,v)
     • If u or v is not already in the MST, add the edge uv to the MST
```

How fast is this version

$O(|E| \lg |E|)$

Fastest Prim's Pseudo-code

```
1. Add all of the vertices to a min-priority-queue prioritized by the min
   edge cost to be added to the MST. Initialize (key=v, dist=∞, edge=<>)
2. Repeatedly pick the next closest vertex v from the MPQ
     1. Add v to the MST using the recorded edge
     2. For each edge (v, u)
          • If cost(v,u) < MPQ(u)
               • MPQ.decreasePriority(u, cost(v,u), (v,u))
```

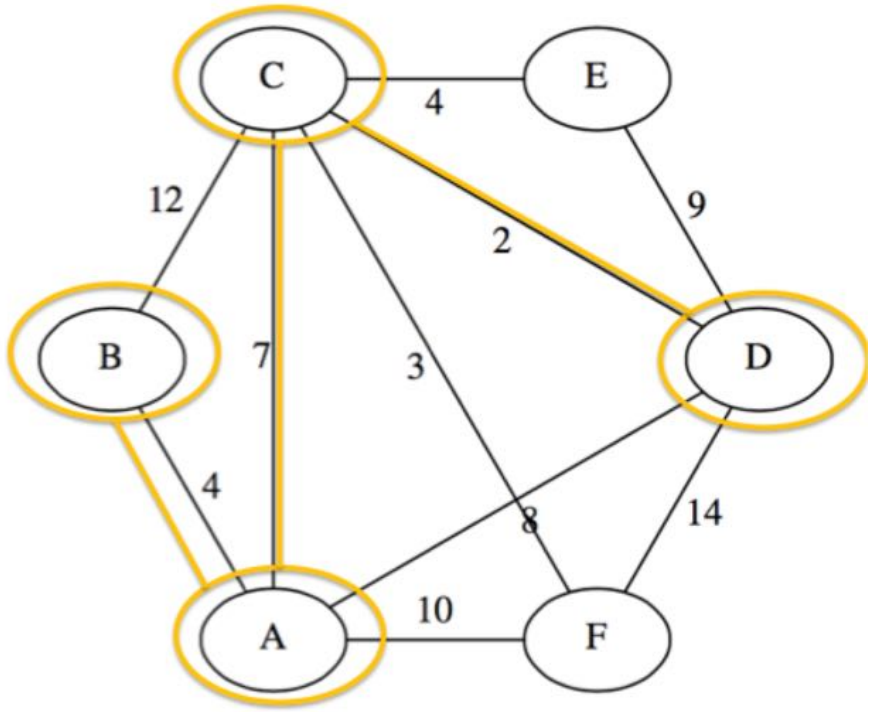
How fast is this version

$O(|E| \lg |V|)$

Using Fibonacci Heap

$O(|E| + |V| \lg |V|)$

Teaser: Kruskal's Algorithm



Kruskal's Algorithm Sketch

1. Create a forest F (a set of trees), where each vertex in the graph is a separate tree
2. Create a set S containing all the edges in the graph
3. while S is nonempty and F is not yet spanning
    1. remove an edge with minimum weight from S
    2. if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree

Union-Find

Part 3: Graph Complexity

Traveling Salesman Problem

- What if we wanted to compute the minimum weight path visiting every node once?

ABDCA: 4+2+5+3 = 14

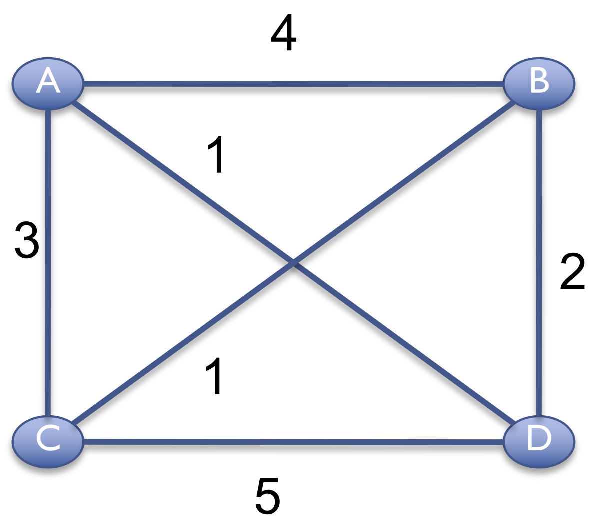
ACDBA: 3+5+2+4 = 14*

ABCDA: 4+1+5+1 = 11

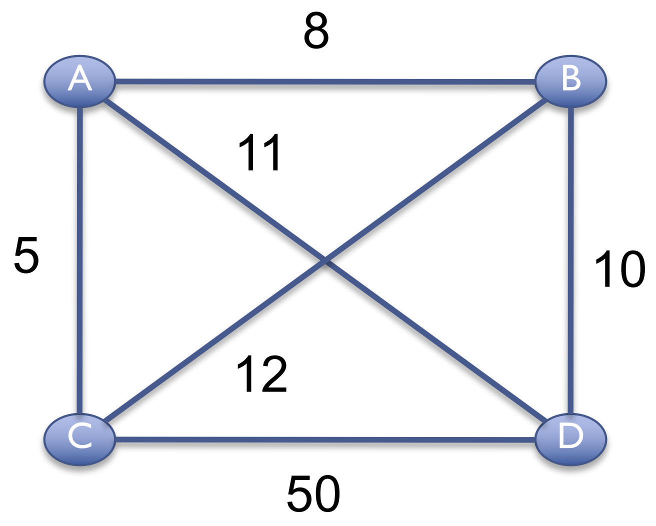
ADCBA: 1+5+1+4 = 11*

ACBDA: 3+1+2+1 = 7

ADBCA: 1+2+1+3= 7 *



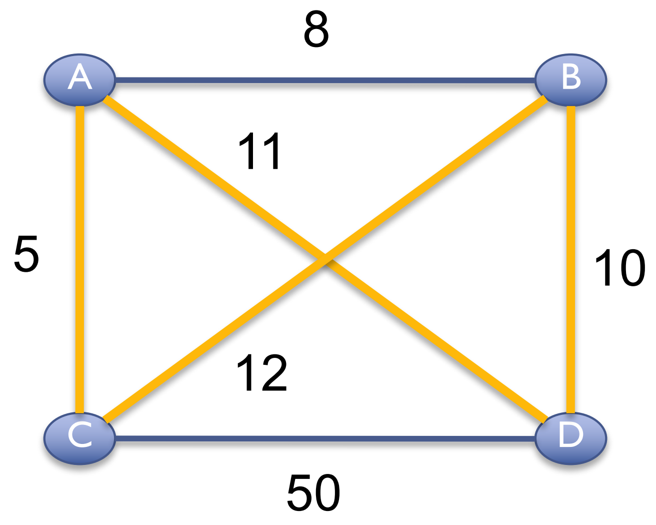
Greedy Search



Greedy Search

Greedy Search

```
cur=graph.randNode()
while (!done)
    next=cur.getNextClosest()
```



Greedy: ABDCA = 5+8+10+50= 73

Optimal: ACBDA = 5+11+10+12 = 38

Greedy finds the global optimum only when

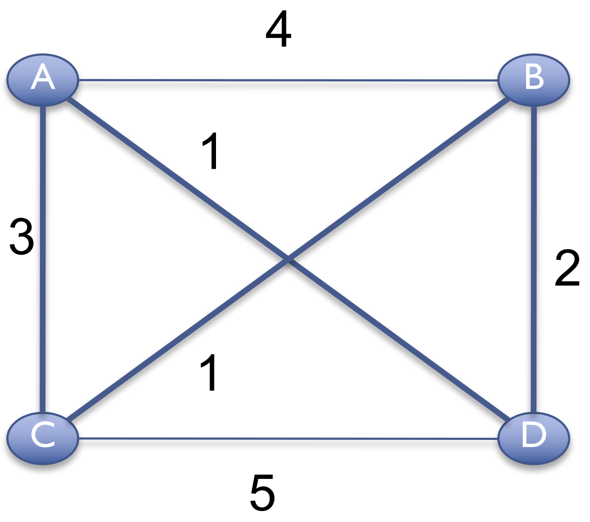
1. Greedy Choice: Local is correct without reconsideration
2. Optimal Substructure: Problem can be split into subproblems

Optimal Greedy: Dijkstra's, Prim's, Cookie Monster

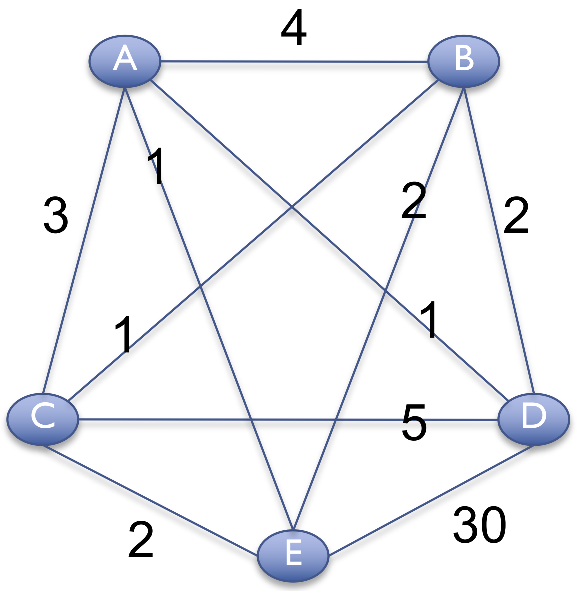
TSP Complexity

- No fast solution
  - Knowing optimal tour through n cities doesn't seem to help much for n+1 cities

[How many possible tours for n cities?]

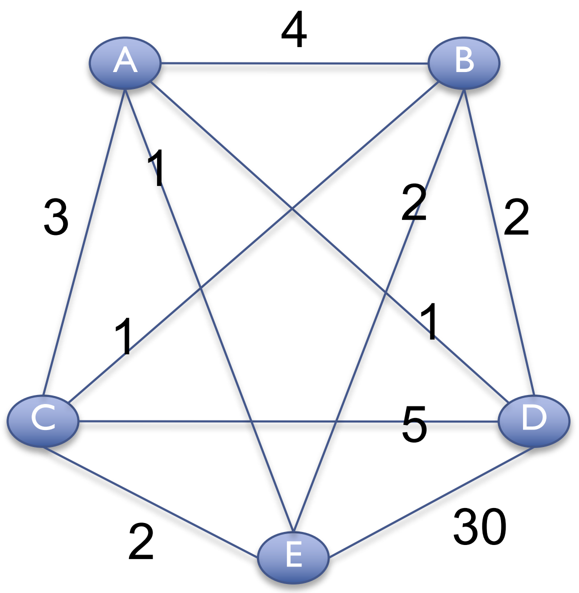


- Extensive searching is the only provably correct algorithm
  - Brute Force: O(n!)
    - ~20 cities max
    - 20! = 2.4 x 10^{18}



Branch-and-Bound

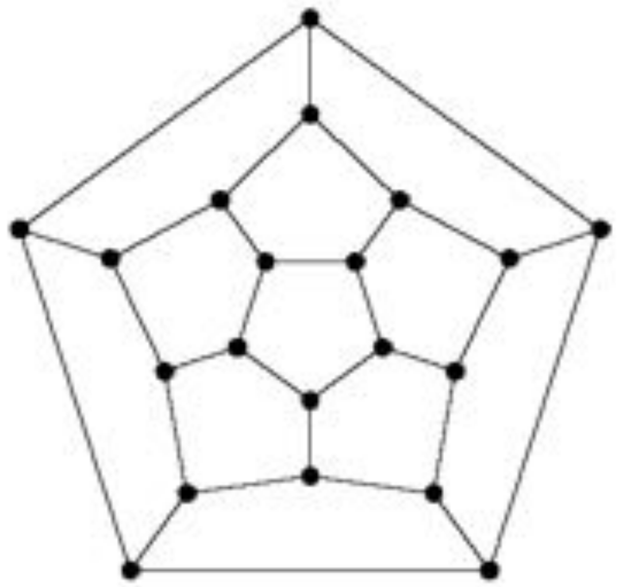
- Abort on suboptimal solutions as soon as possible
  - ADBECA = 1+2+2+2+3 = 10
  - ABDE = 4+2+30 > 10
  - ADE = 1+30 > 10
  - AED = 1+30 > 10
  - …



- Performance Heuristic
  - Always gives the optimal answer
  - Doesn't always help performance, but often does
  - Current TSP record holder:
    - 85,900 cities
    - 85900! = 10^{386526}

[When not?]

TSP and NP-complete



- TSP is one of many extremely hard problems of the class NP-complete
  - Extensive searching is the only way to find an exact solution
  - Often have to settle for approx. solution

- WARNING: Many "natural" problems are in this class
  - Find a tour the visits every node once (TSP)
  - Find the smallest set of vertices covering the edges (vertex cover)
  - Find the largest clique in the graph (max clique)
  - Deciding if two graphs have the same structure (graph homomorphism)
  - Find the best set of moves in tetris or battleship
  - …
  - http://en.wikipedia.org/wiki/List_of_NP-complete_problems

Next Steps

1. Reflect on the magic and power of Dijkstra's & Prim's algorithm!

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