Graphs 2

Joonho Kim

Instructions

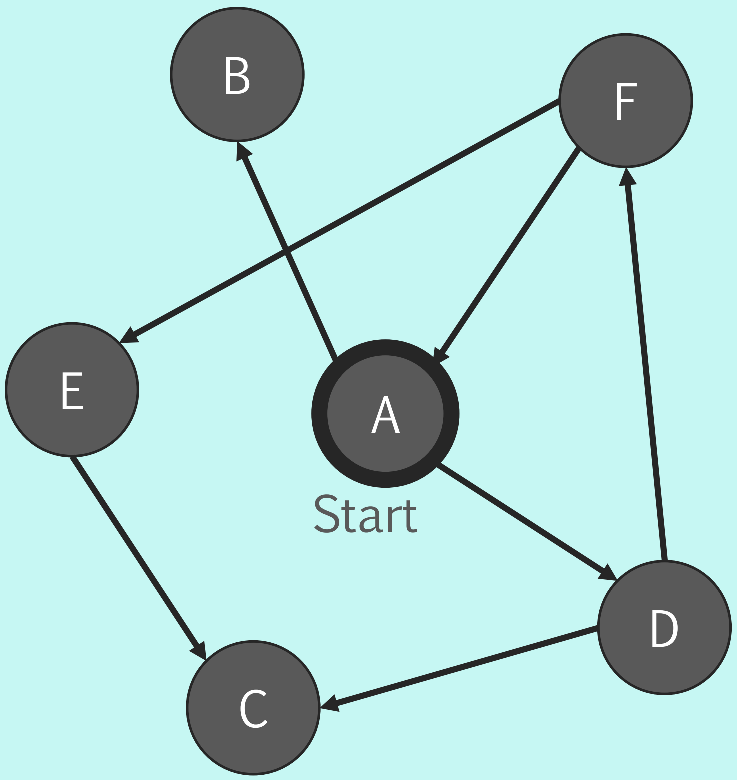
- There will be questions on these slides. Please have a clean piece of paper to write your answers. Write your name on the top right corner for our record. At the end of lecture, we will collect these pieces of paper for your participation grade. Scribes should get ready to scribe.

Announcements

- 5 scribes please
  - Write names on white board to remember
- Homework 8 Graphs is Due next Thu

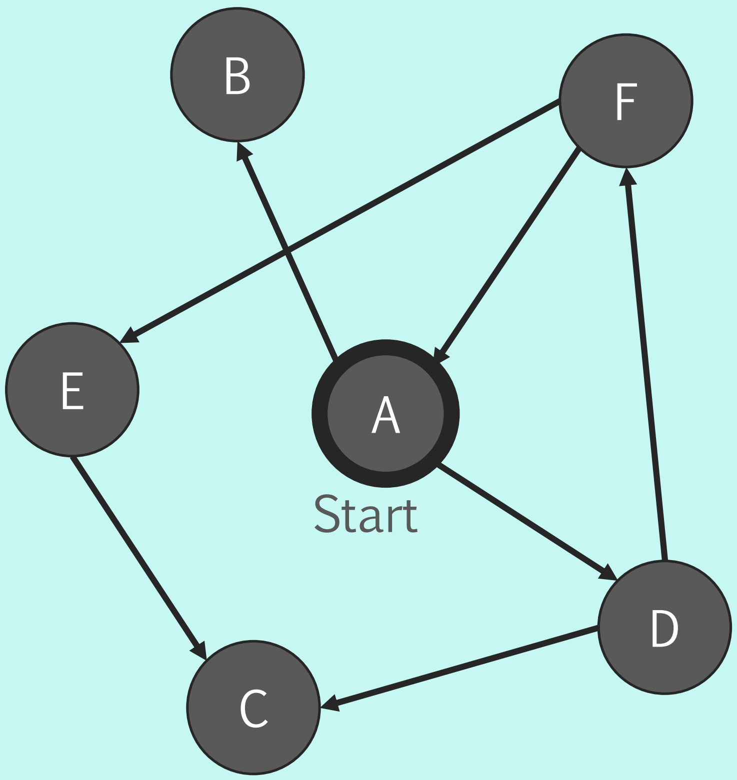
Last Time…

- Take 5 min to write down a the following:
  - For the Graph:
    - Perform Depth First Search
    - Perform Breadth First Search
  - If a Vertex has multiple neighbors, add the Neighbors to your Queue/Stack in alphabetical order.



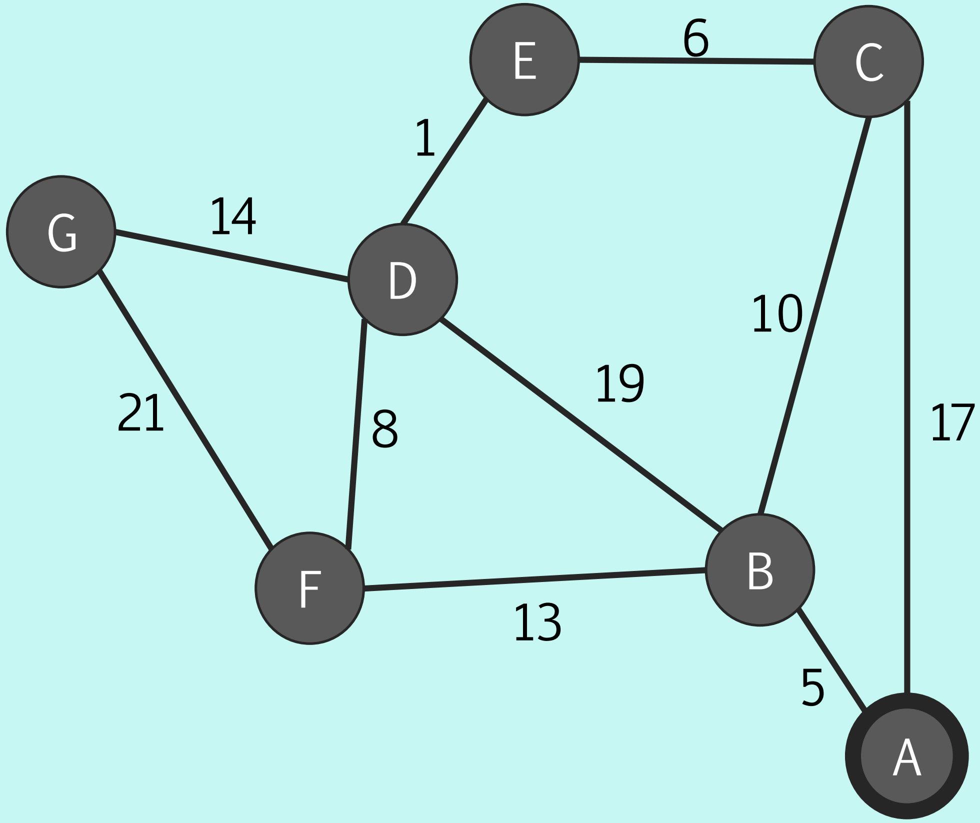
Last Time…

- Take 5 min to write down a the following:
  - For the Graph:
    - Perform Depth First Search
    - Perform Breadth First Search
  - If a Vertex has multiple neighbors, add the Neighbors to your Queue/Stack in alphabetical order.
  - DFS: A, D, F, E, C, B
  - BFS: A, B, D, C, F, E



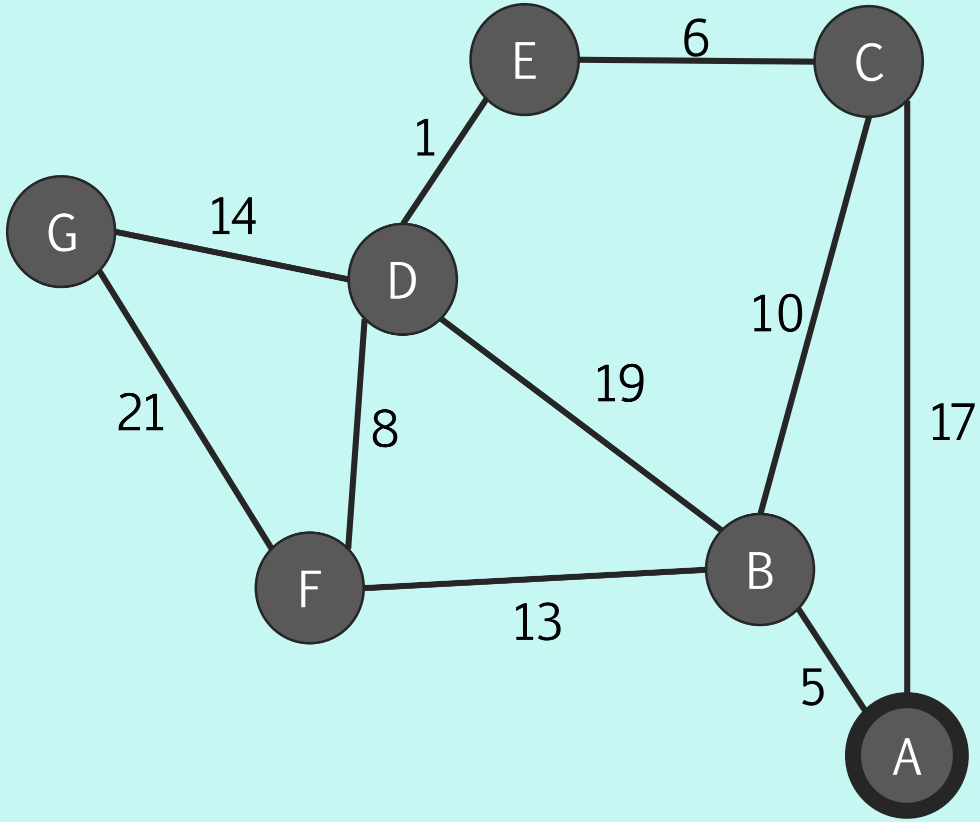
Shortest Path

- Let's say I start at A and I want to go to G. The edge weights represent walking distance on that edge.
- What is the fastest way to get from A to G? What is the shortest path from A to G?



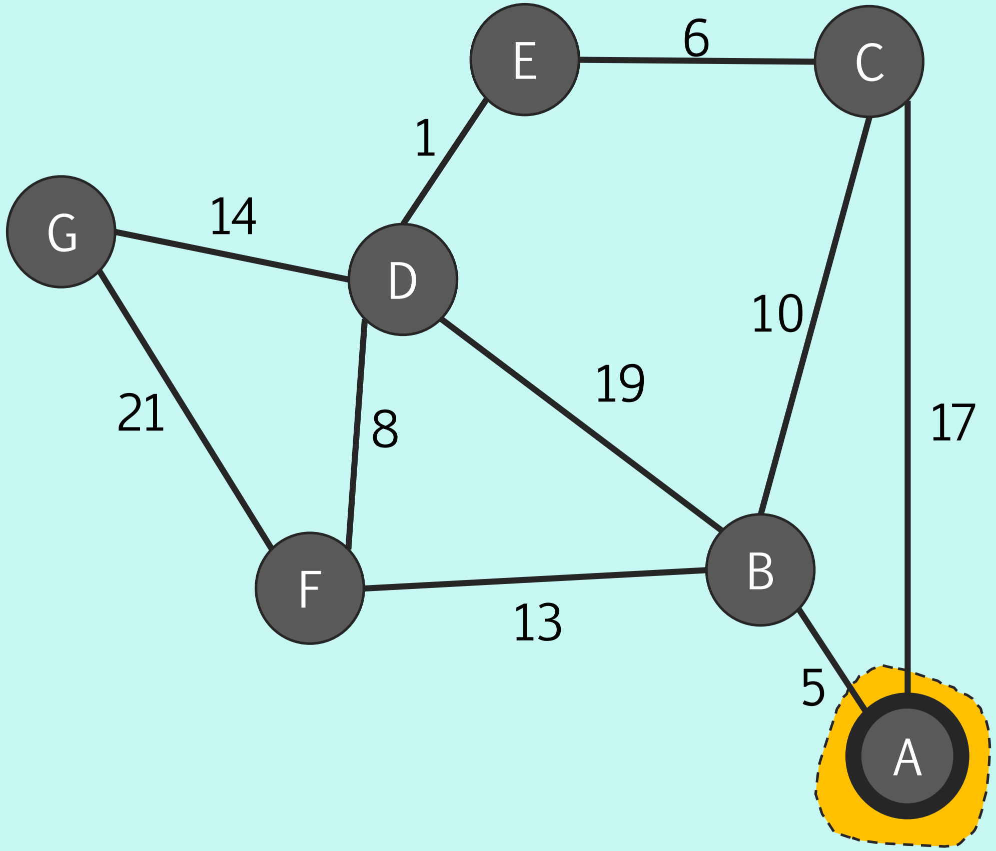
Shortest Path

- Let's say I start at A and I want to go to G.   The edge weights represent walking distance on that edge.
- What is the fastest way to get from A to G?   What is the shortest path from A to G?
- Performing DFS and BFS will **not** give me the shortest path.
  - These don't account for edge weights.

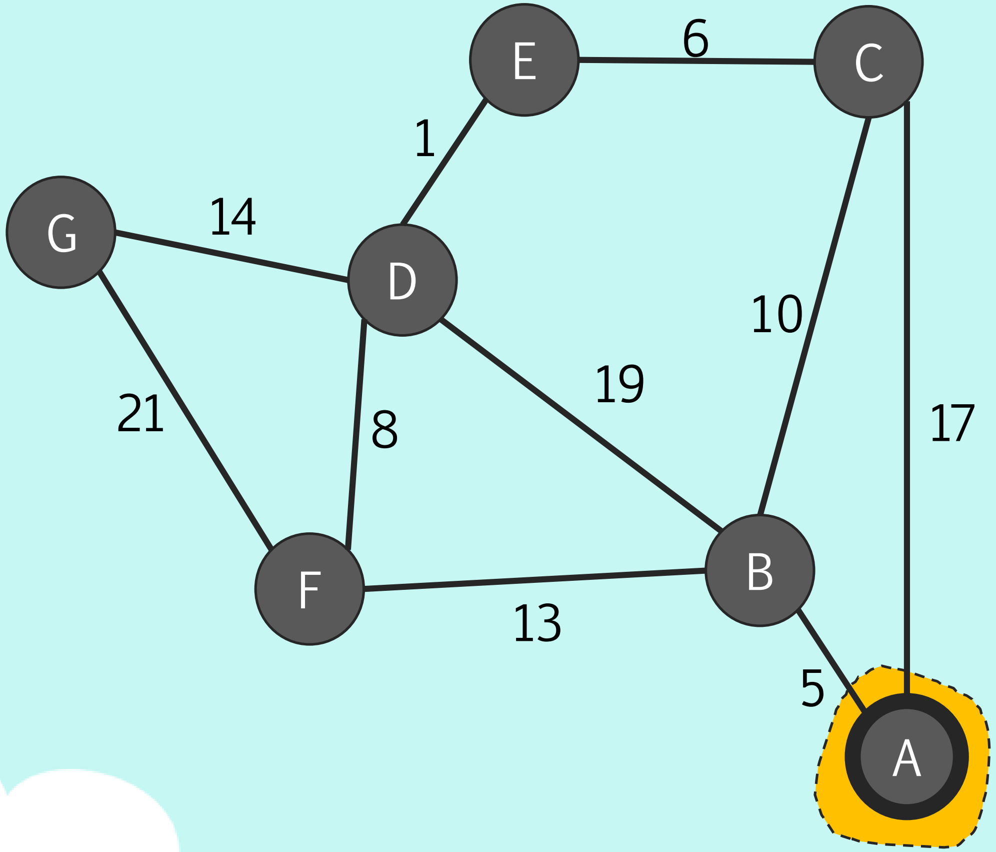


Shortest Path Strategy

- To get from A to G, we'll have to travel through other vertices.
  - Let's try to find the shortest path from A to the other vertices as well. This will help us get to G.
- The orange cloud represents the shortest path from A to any vertex.
- From our orange cloud, we will find all vertices we can reach.



Shortest Path Strategy



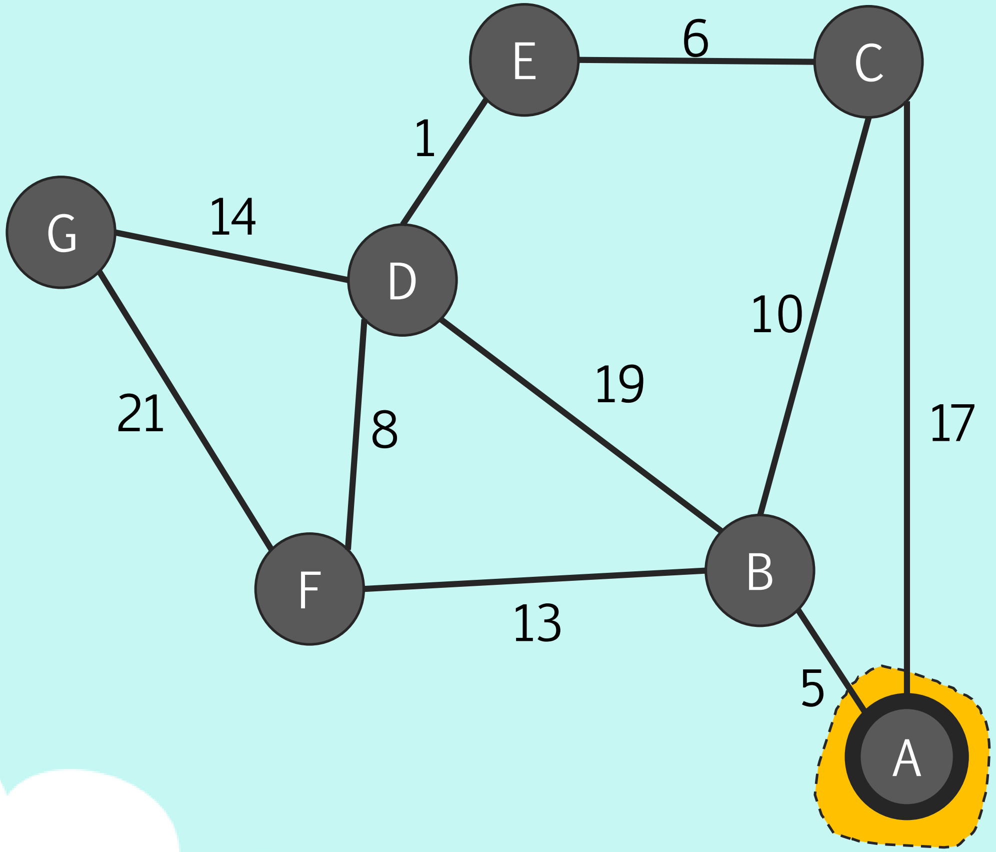
Vertex	Path	Dist
A	A	0
B		INF
C		INF
D		INF
E		INF
F		INF
G		INF

Shortest Path Strategy

From A, I can go to:

- A in a distance of 0

I will solidify that as the shortest path from A to A.

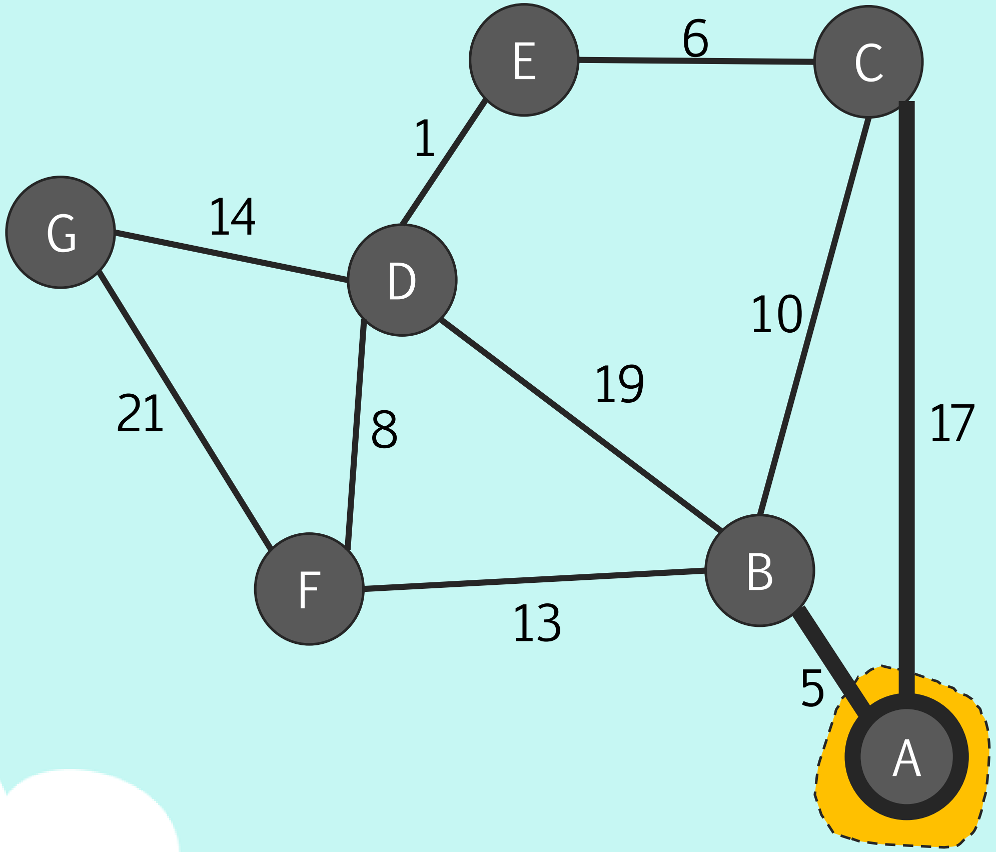


Vertex	Path	Dist
A	A	0
B		INF
C		INF
D		INF
E		INF
F		INF
G		INF

Shortest Path Strategy

From A, I can go to:

- B in a distance of 5
- C in a distance of 17



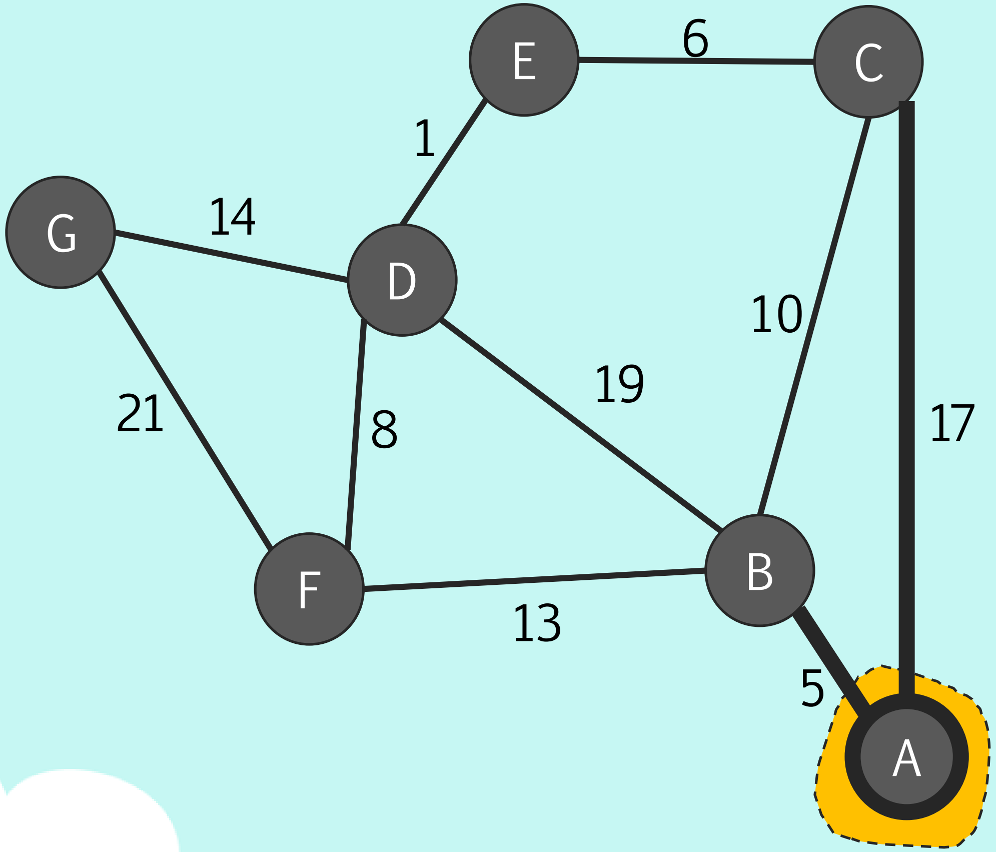
Vertex	Path	Dist
A	A	0
B		INF
C		INF
D		INF
E		INF
F		INF
G		INF

Shortest Path Strategy

From A, I can go to:

- B in a distance of 5
- C in a distance of 17

Let's update our distances to B and C in our table.



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, C	17
D		INF
E		INF
F		INF
G		INF

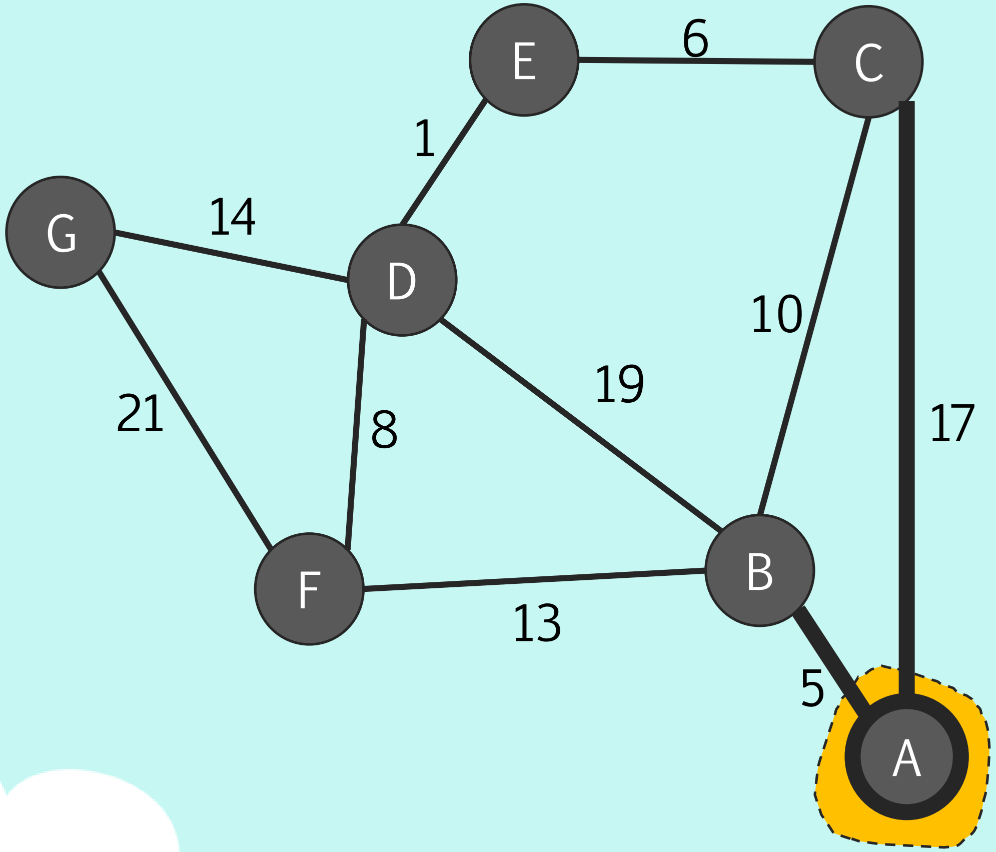
Shortest Path Strategy

From A, I can go to:

- B in a distance of 5
- C in a distance of 17

Let's update our distances to B and C in our table.

Out of the vertices we can reach, expand to the vertex with shortest distance. **B**



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, C	17
D		INF
E		INF
F		INF
G		INF

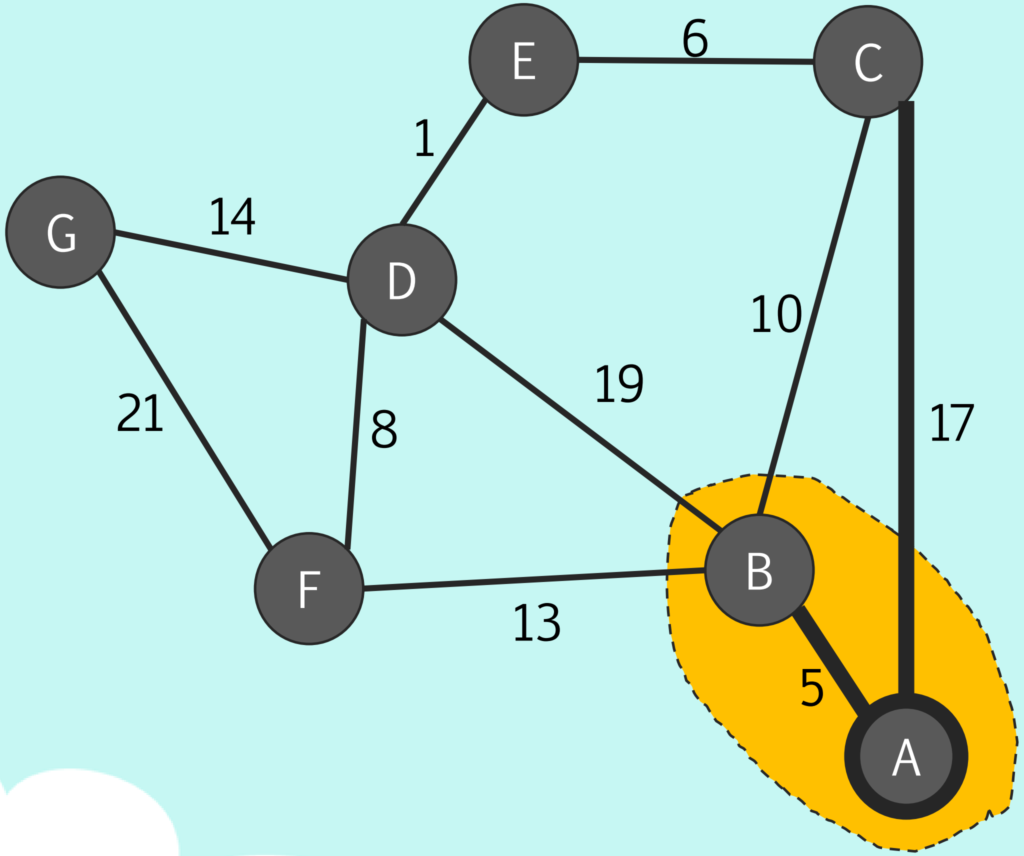
Shortest Path Strategy

From A, I can go to:

- B in a distance of 5
- C in a distance of 17

Let's update our distances to B and C in our table.

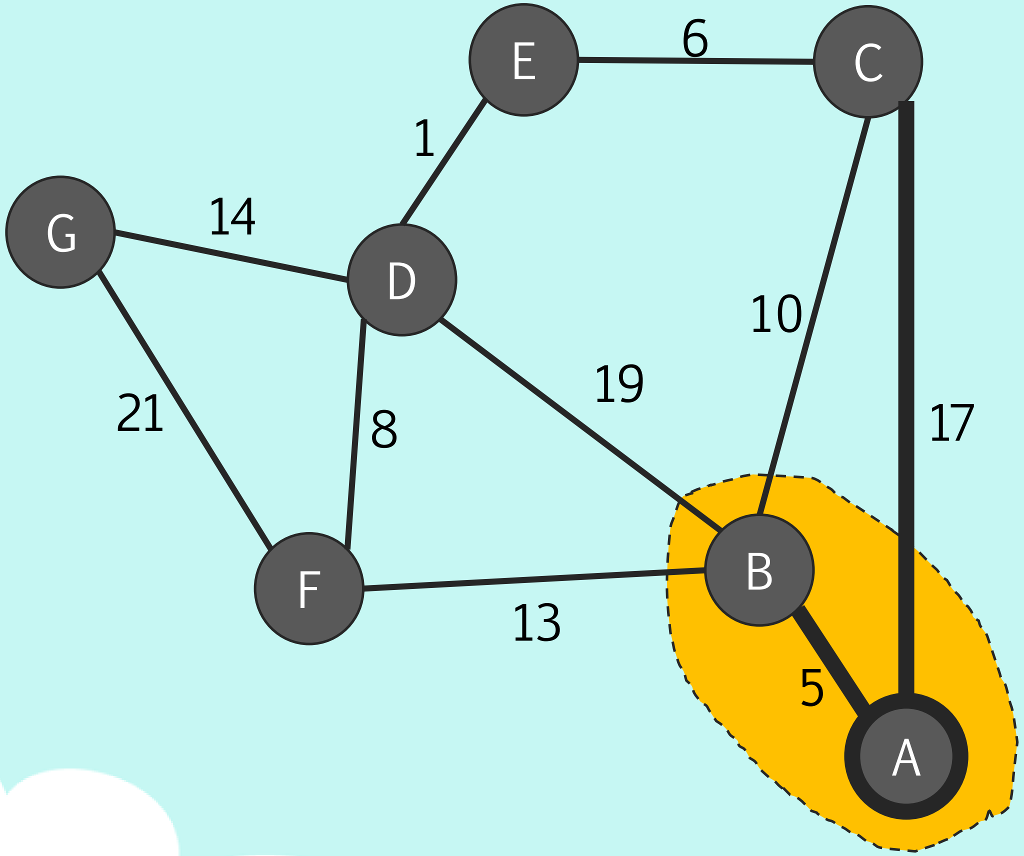
Out of the vertices we can reach, expand to the vertex with shortest distance. **B**



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, C	17
D		INF
E		INF
F		INF
G		INF

Shortest Path Strategy

We have now solidified the shortest path from A to B is [A, B] with a distance of 5.

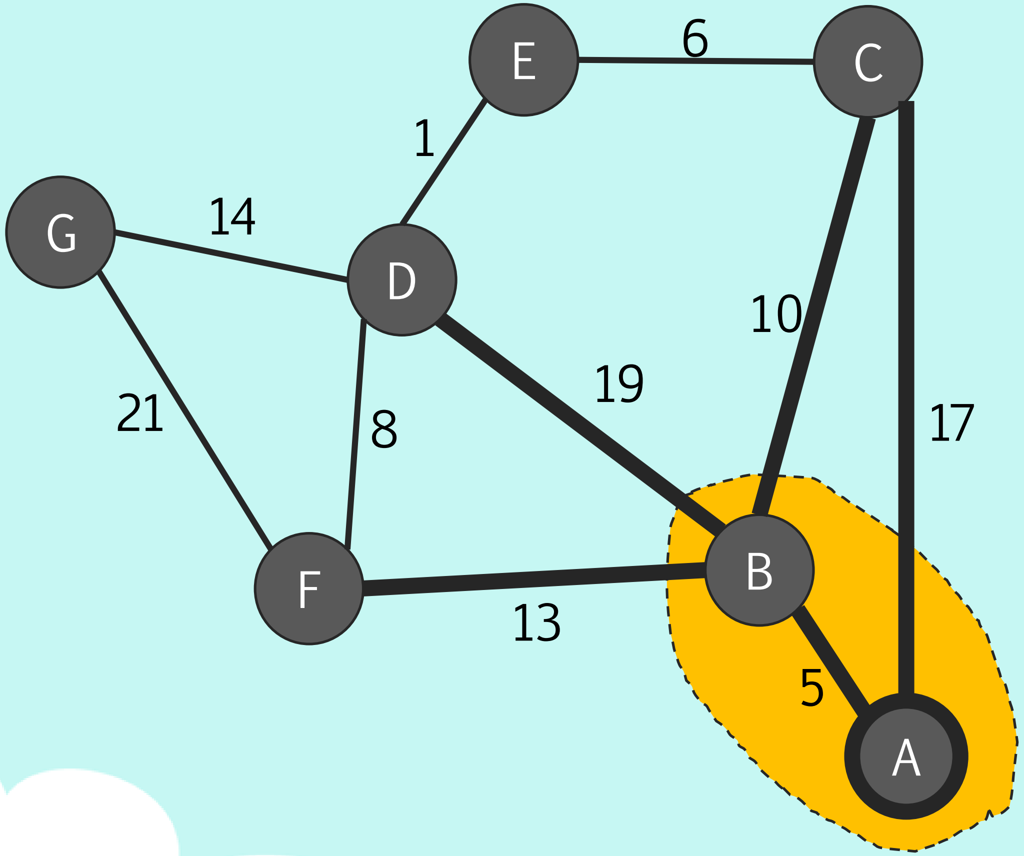


Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, C	17
D		INF
E		INF
F		INF
G		INF

Shortest Path Strategy

We have now solidified the shortest path from A to B is [A, B] with a distance of 5.

Now that we've expanded to B, we can visit B's neighbors {C, D, F}



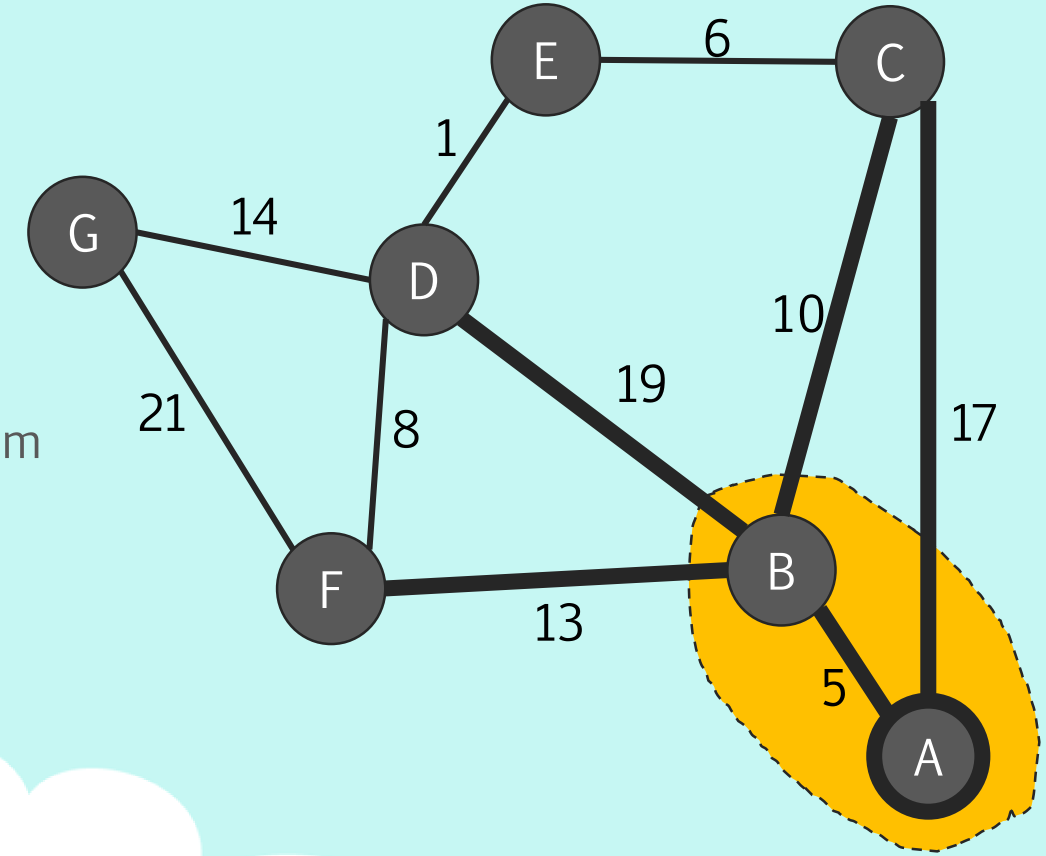
Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, C	17
D		INF
E		INF
F		INF
G		INF

Shortest Path Strategy

From A, I can go to:

- C in a distance of 17
- C in a distance of 5 + 10
- D in a distance of 5 + 19
- F in a distance of 5 + 13

The last 3 distances are from path [A, B].



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, C	17
D		INF
E		INF
F		INF
G		INF

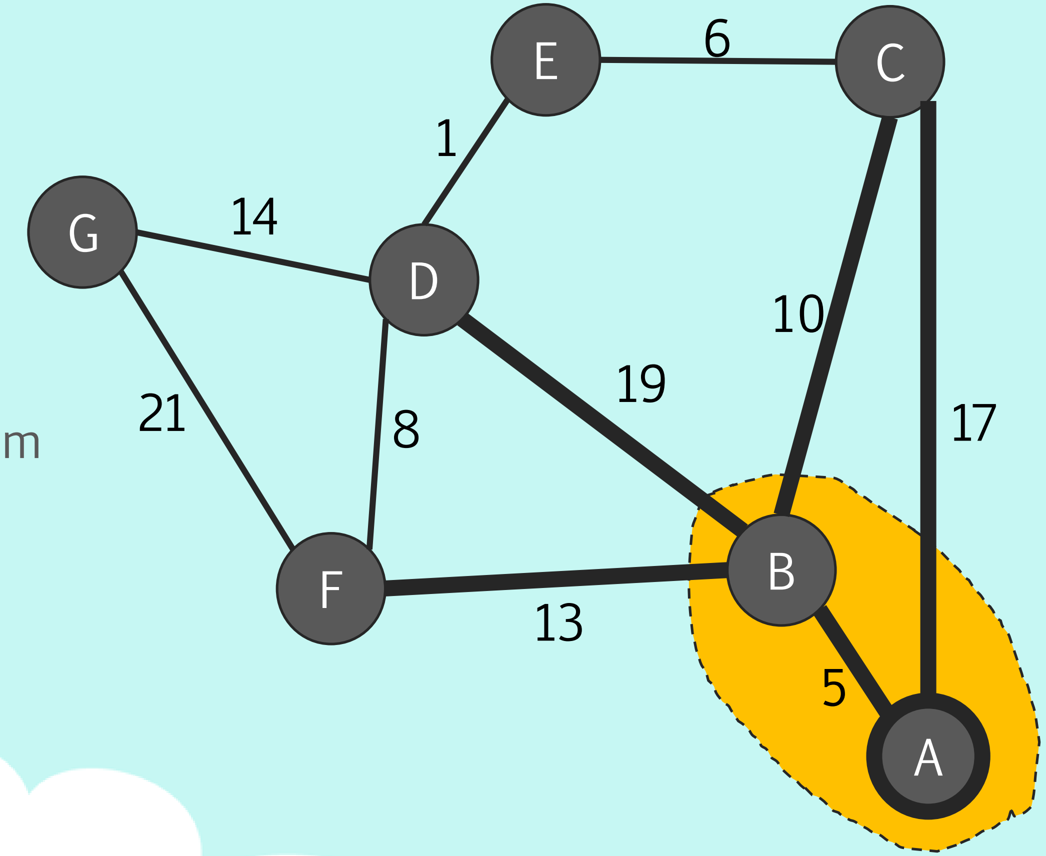
Shortest Path Strategy

From A, I can go to:

- C in a distance of 17
- C in a distance of 5 + 10
- D in a distance of 5 + 19
- F in a distance of 5 + 13

The last 3 distances are from path [A, B].

Let's update our table with these new distances and paths.



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, C	17
D		INF
E		INF
F		INF
G		INF

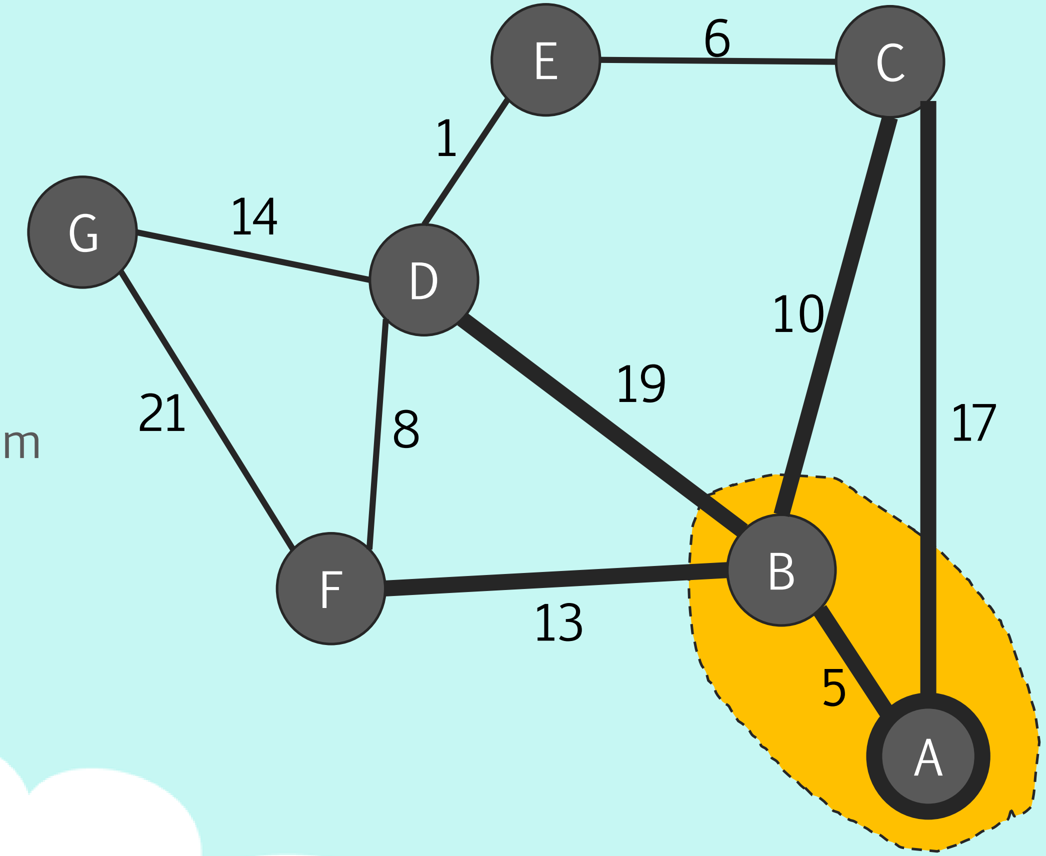
Shortest Path Strategy

From A, I can go to:

- C in a distance of 17
- C in a distance of 5 + 10
- D in a distance of 5 + 19
- F in a distance of 5 + 13

The last 3 distances are from path [A, B].

Let's update our table with these new distances and paths.



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, C	17
D		INF
E		INF
F		INF
G		INF

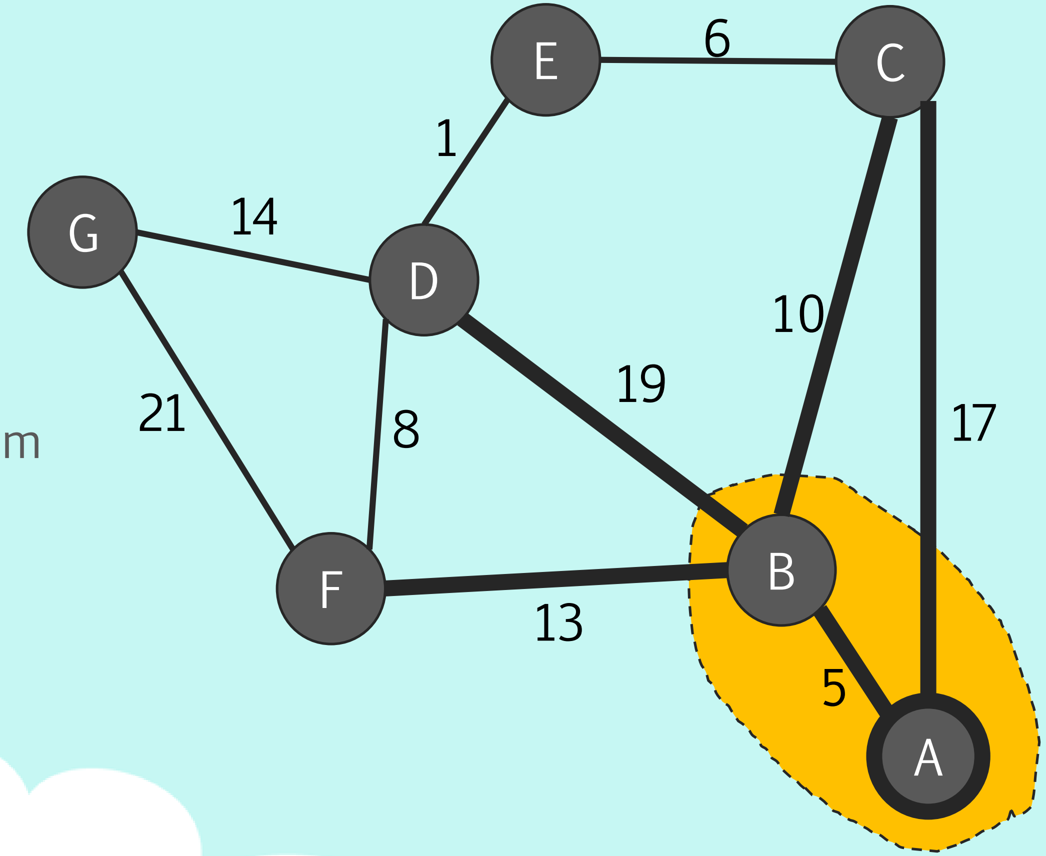
Shortest Path Strategy

From A, I can go to:

- C in a distance of 17
- C in a distance of 5 + 10
- D in a distance of 5 + 19
- F in a distance of 5 + 13

The last 3 distances are from path [A, B].

Let's update our table with these new distances and paths.



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E		INF
F	A, B, F	18
G		INF

Shortest Path Strategy

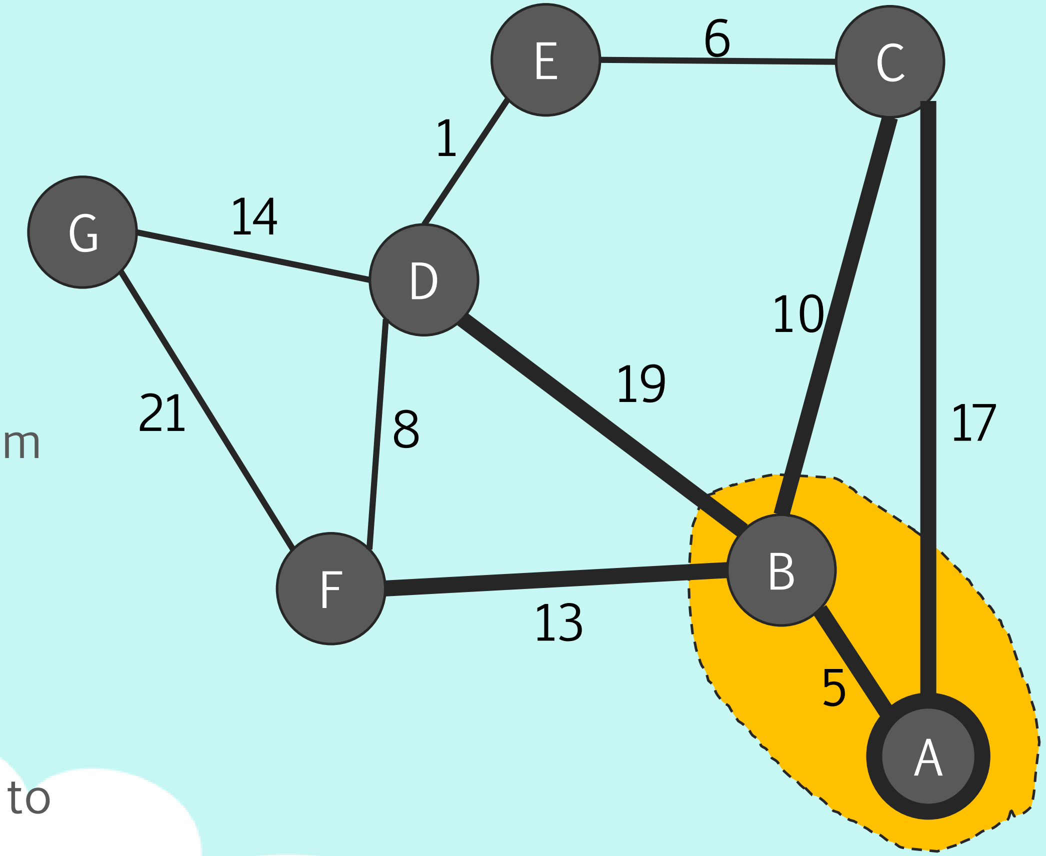
From A, I can go to:

- C in a distance of 17
- C in a distance of 5 + 10
- D in a distance of 5 + 19
- F in a distance of 5 + 13

The last 3 distances are from path [A, B].

Let's update our table with these new distances and paths.

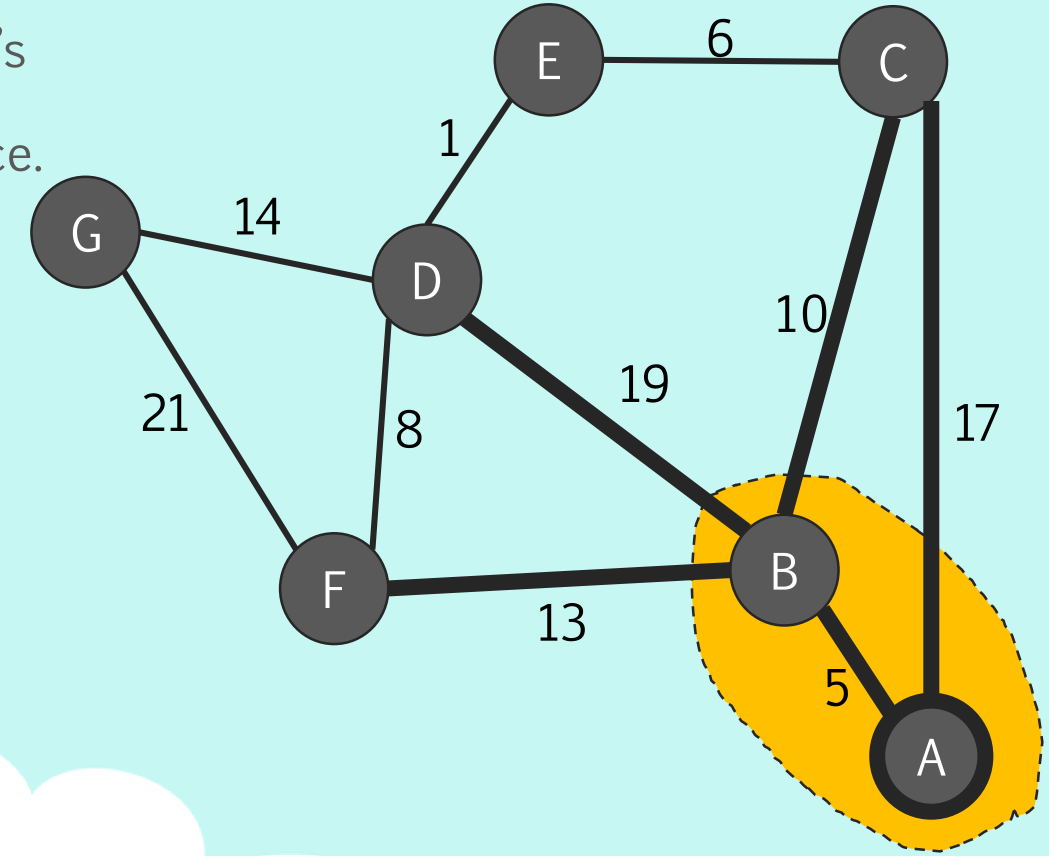
We've found a shorter path to C through the path to B!



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E		INF
F	A, B, F	18
G		INF

Shortest Path Strategy

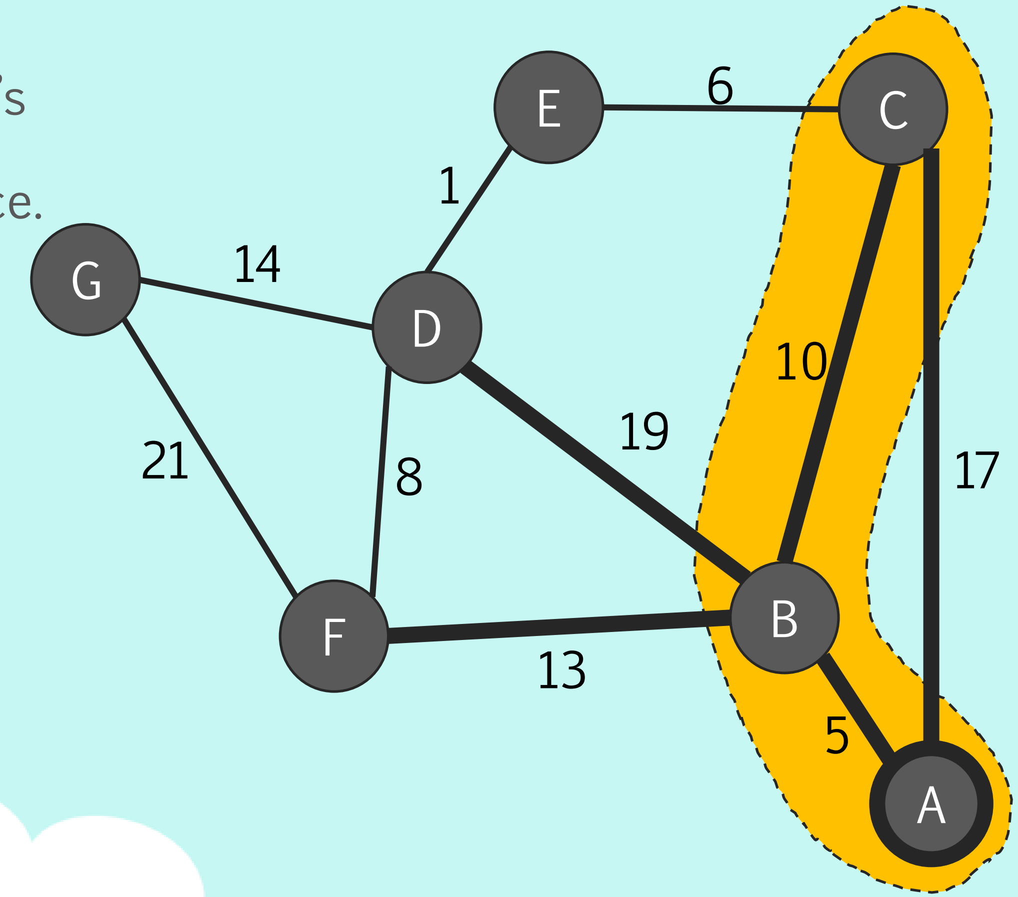
Now out of these paths, let's expand our cloud to the vertex with shortest distance.
C



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E		INF
F	A, B, F	18
G		INF

Shortest Path Strategy

Now out of these paths, let's expand our cloud to the vertex with shortest distance.
C



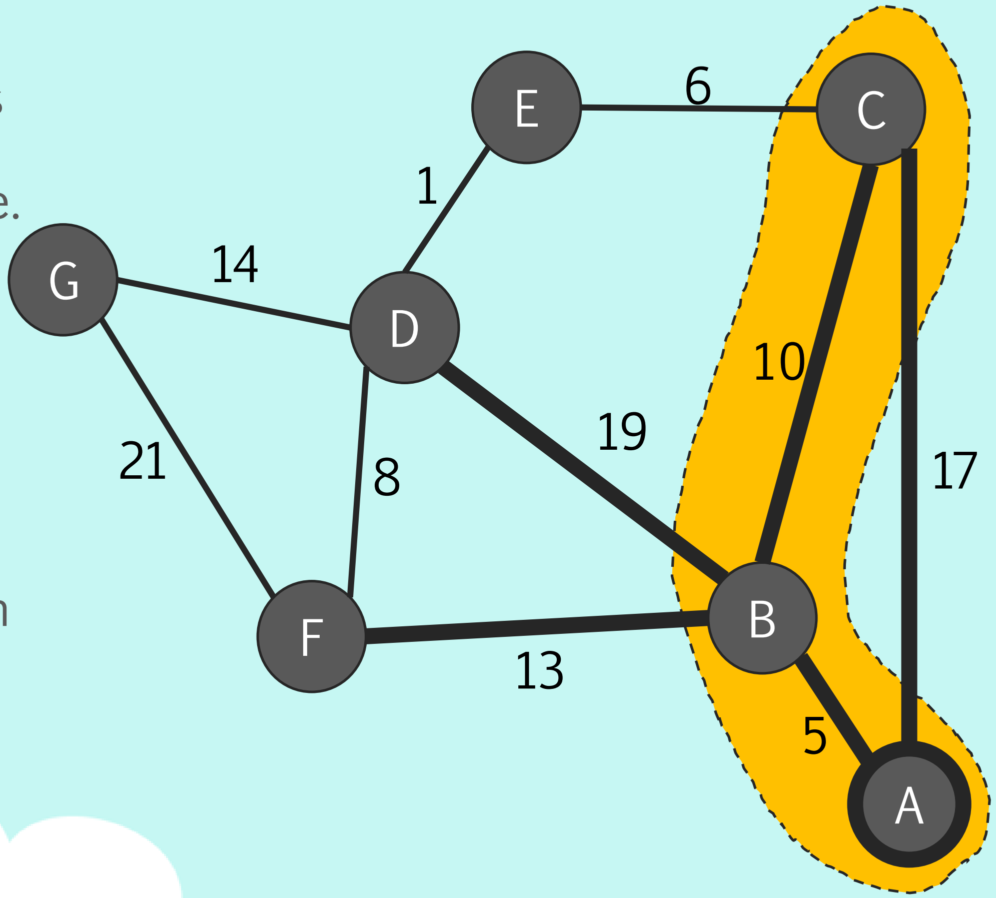
Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E		INF
F	A, B, F	18
G		INF

Shortest Path Strategy

Now out of these paths, let's expand our cloud to the vertex with shortest distance. C

Now we've solidified the shortest path from A to C as [A, B, C] with a distance of 15.

We use that path rather than path [A, C] which is a distance of 17.



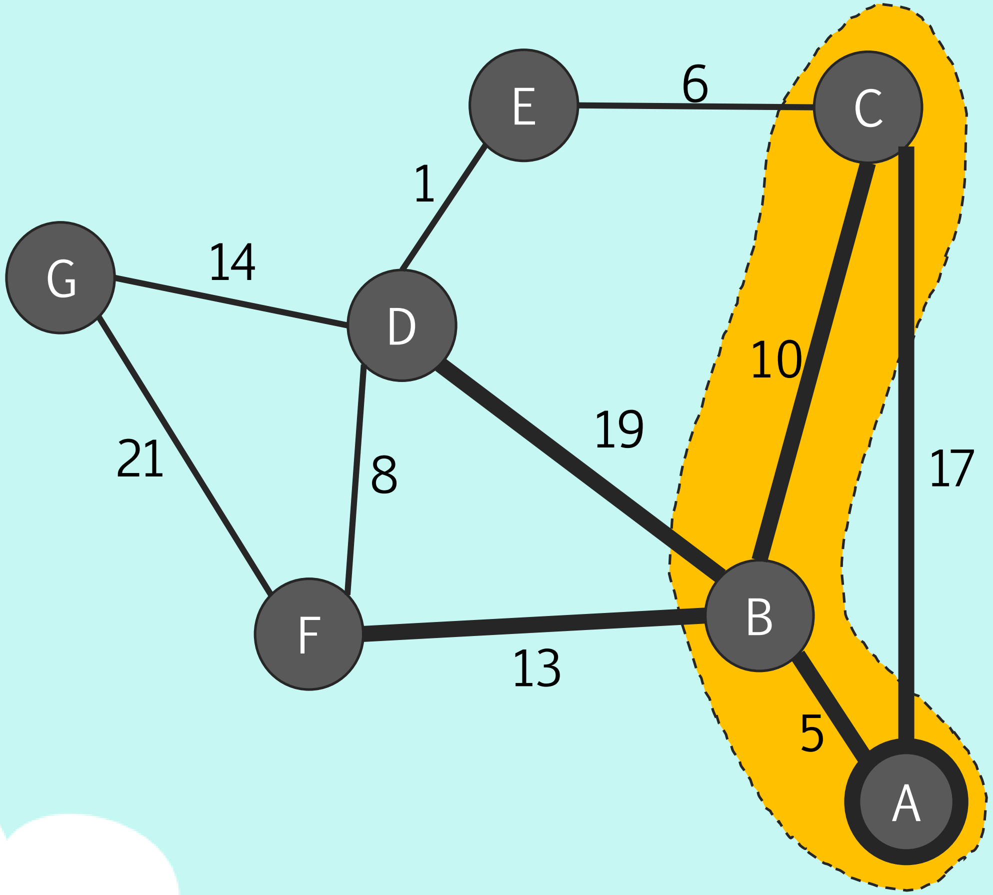
Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E		INF
F	A, B, F	18
G		INF

Shortest Path Strategy

So right now our strategy is:

1. Calculate distances to vertices reachable from our cloud.
2. Update these potential shortest paths in our table.
3. Expand our cloud to the vertex not in our cloud with shortest distance.

Anything within the cloud is the shortest path from A to that vertex.



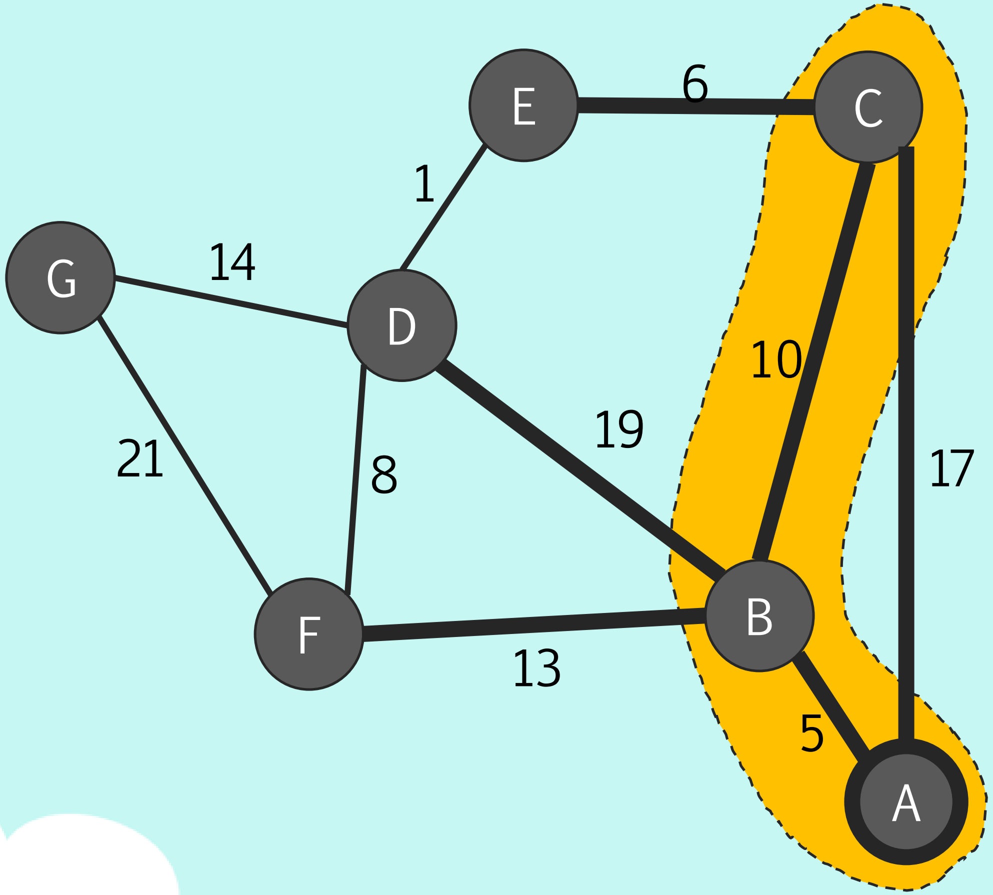
Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E		INF
F	A, B, F	18
G		INF

Shortest Path Strategy

From A, I can go to:

- D in a distance of 5 + 19
- F in a distance of 5 + 13
- E in a distance of 15 + 6

Let's update our table with these new distances and paths.



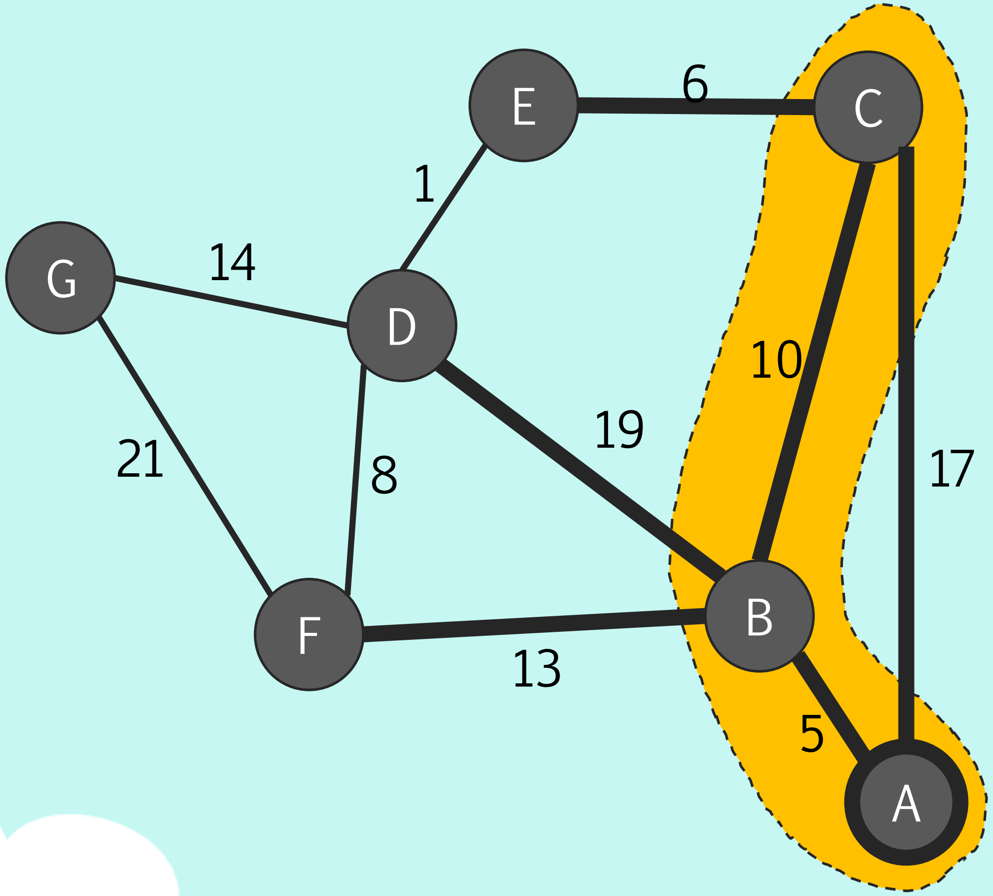
Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E		INF
F	A, B, F	18
G		INF

Shortest Path Strategy

From A, I can go to:

- D in a distance of 5 + 19
- F in a distance of 5 + 13
- E in a distance of 15 + 6

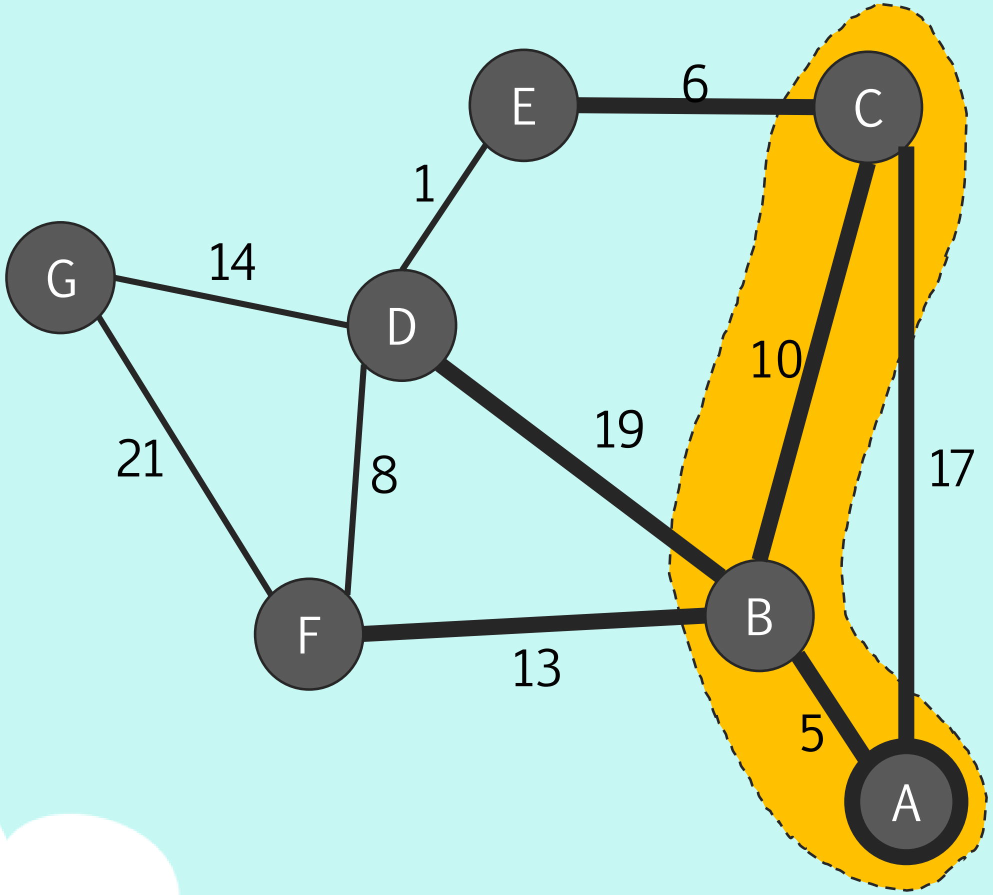
Let's update our table with these new distances and paths.



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E	**A, B, C, E**	**21**
F	A, B, F	18
G		INF

Shortest Path Strategy

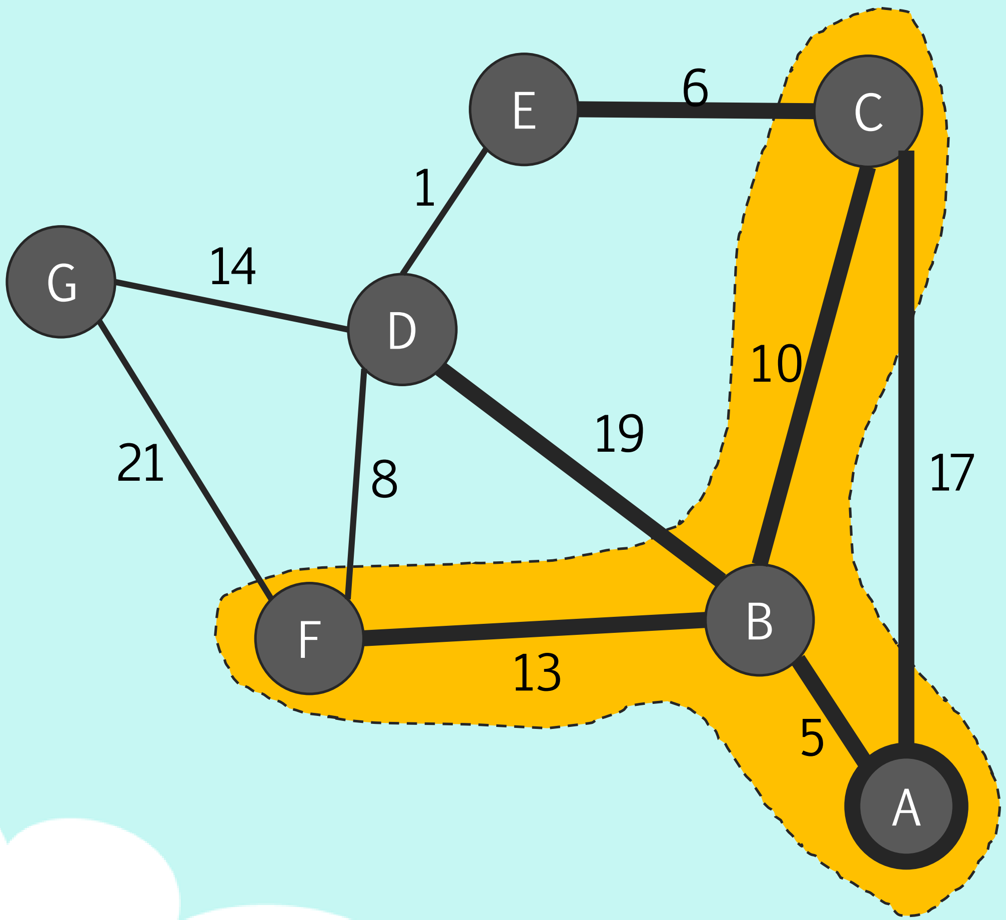
Now out of these paths, let's expand our cloud to the vertex with shortest distance.
F



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E	A, B, C, E	21
F	A, B, F	18
G		INF

Shortest Path Strategy

Now out of these paths, let's expand our cloud to the vertex with shortest distance.
F

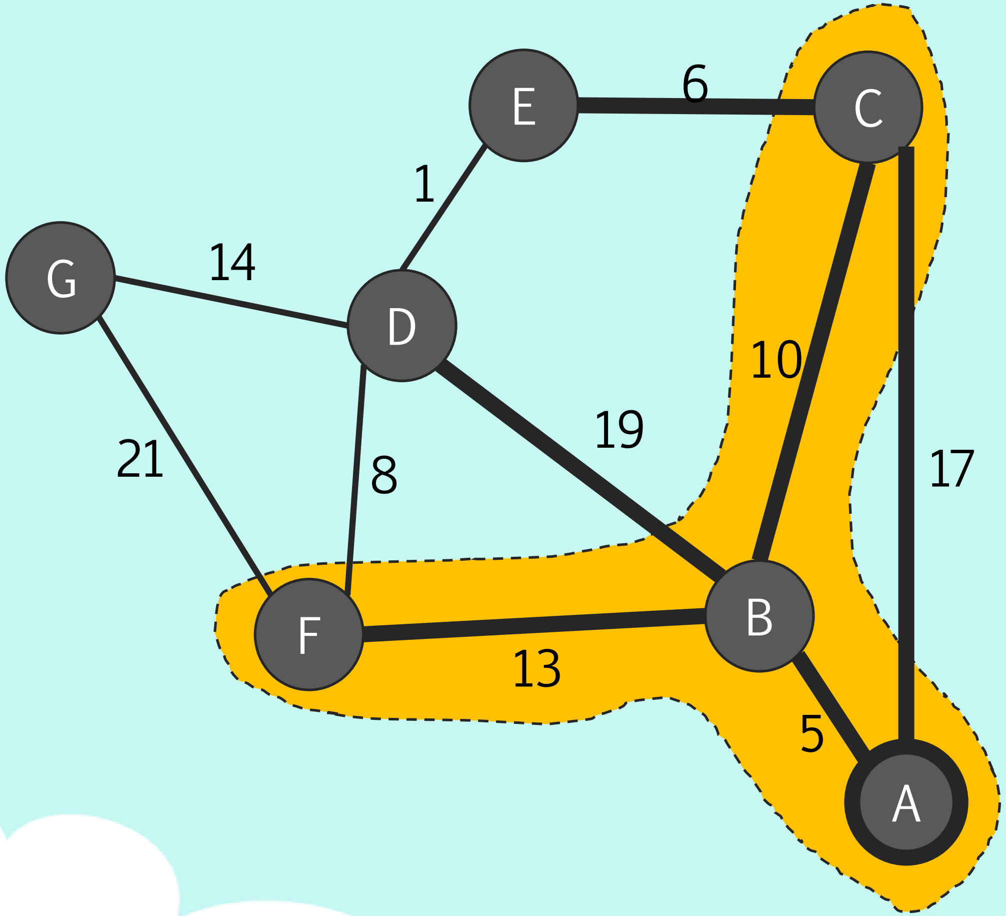


Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E	A, B, C, E	21
F	A, B, F	18
G		INF

Shortest Path Strategy

Now out of these paths, let's expand our cloud to the vertex with shortest distance.
F

Now we've solidified the shortest path from A to F as [A, B, F] with a distance of 18.



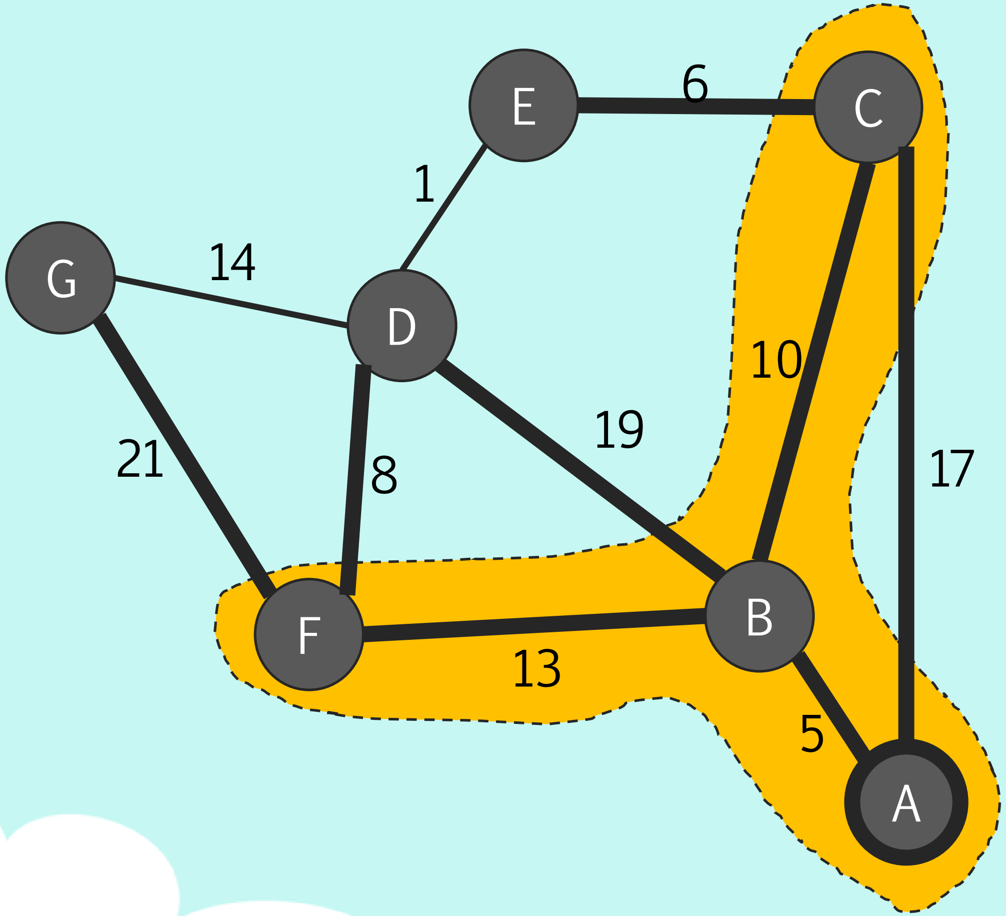
Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E	A, B, C, E	21
F	A, B, F	18
G		INF

Shortest Path Strategy

From A, I can go to:

- D in a distance of 5 + 19
- E in a distance of 15 + 6
- D in a distance of 18 + 8
- G in a distance of 18 + 21

The last two paths are using vertex F.



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E	A, B, C, E	21
F	A, B, F	18
G		INF

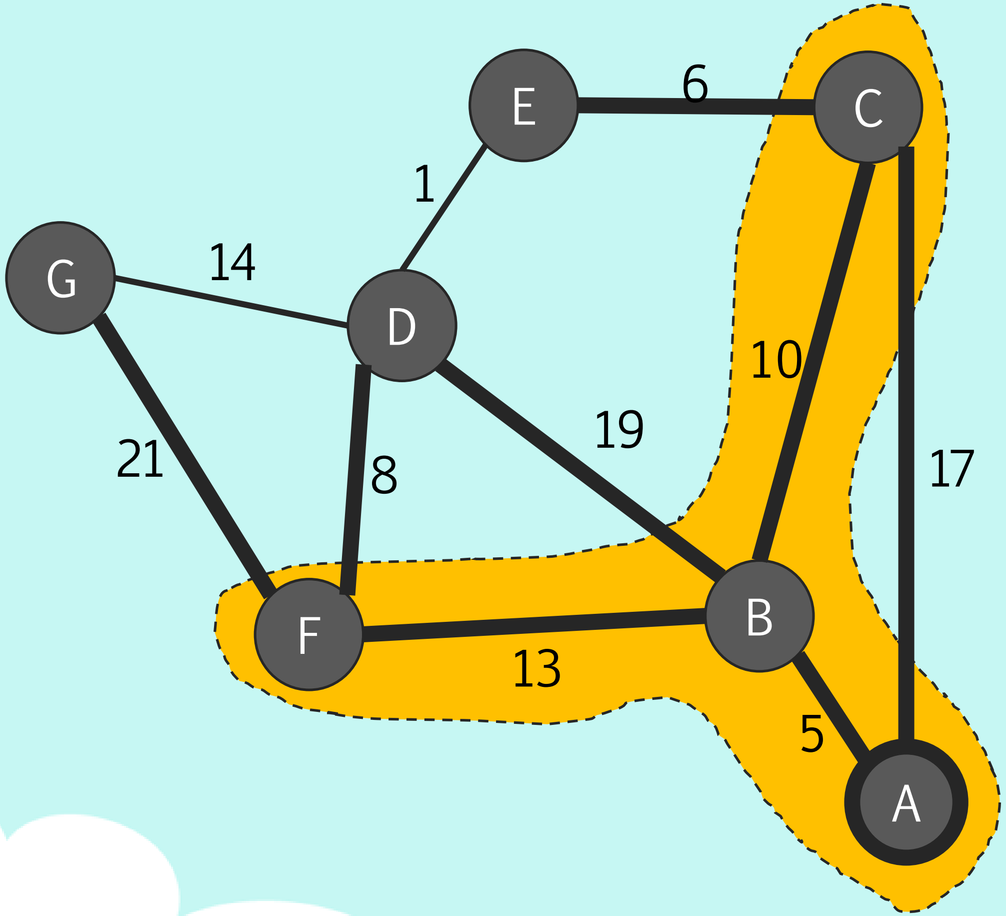
Shortest Path Strategy

From A, I can go to:

- D in a distance of 5 + 19
- E in a distance of 15 + 6
- D in a distance of 18 + 8
- G in a distance of 18 + 21

The last two paths are using vertex F.

Let's update our table with these new distances and paths.



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E	A, B, C, E	21
F	A, B, F	18
G		INF

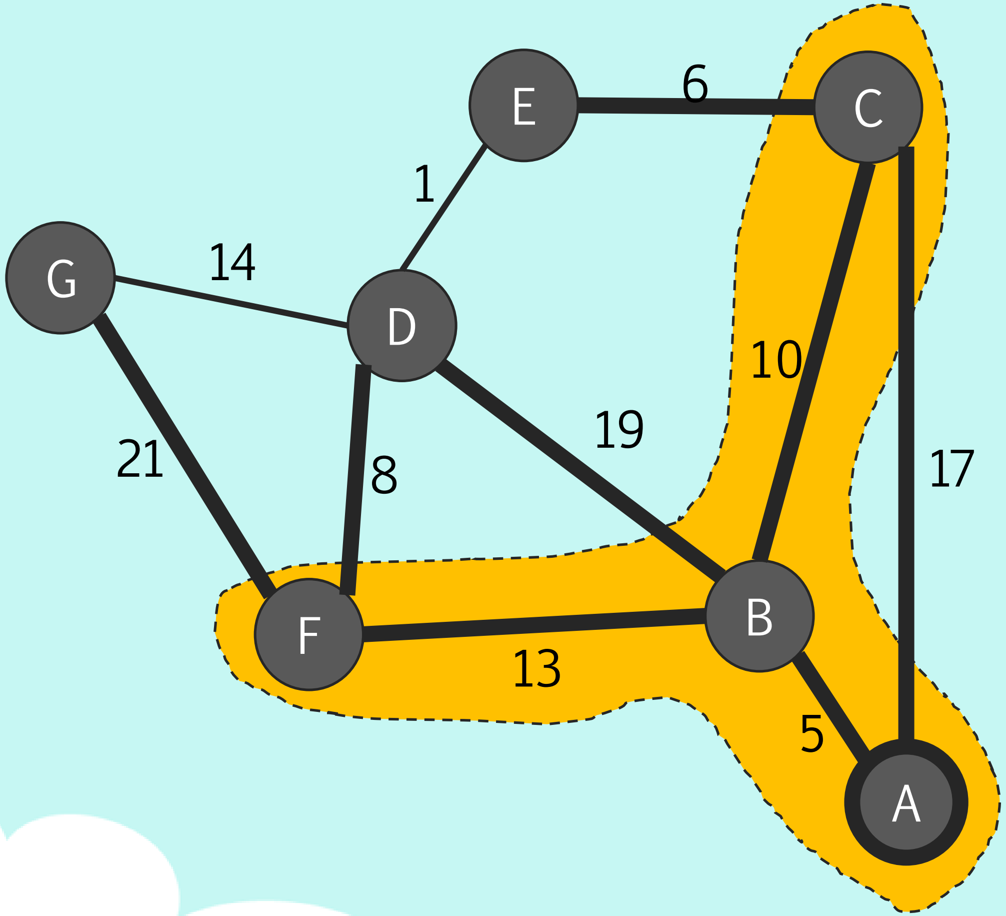
Shortest Path Strategy

From A, I can go to:

- D in a distance of 5 + 19
- E in a distance of 15 + 6
- D in a distance of 18 + 8
- G in a distance of 18 + 21

The last two paths are using vertex F.

Let's update our table with these new distances and paths.



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E	A, B, C, E	21
F	A, B, F	18
G	A, B, F, G	39

Shortest Path Strategy

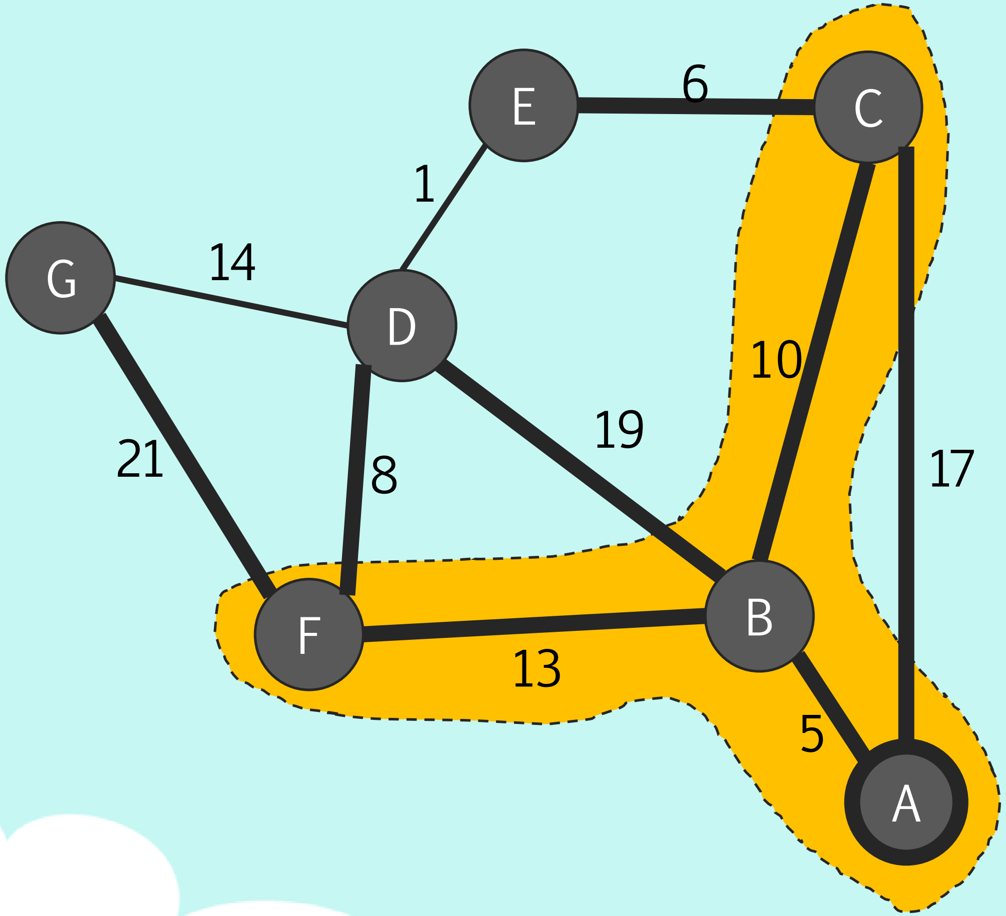
From A, I can go to:

- D in a distance of 5 + 19
- E in a distance of 15 + 6
- D in a distance of 18 + 8
- G in a distance of 18 + 21

The last two paths are using vertex F.

Let's update our table with these new distances and paths.

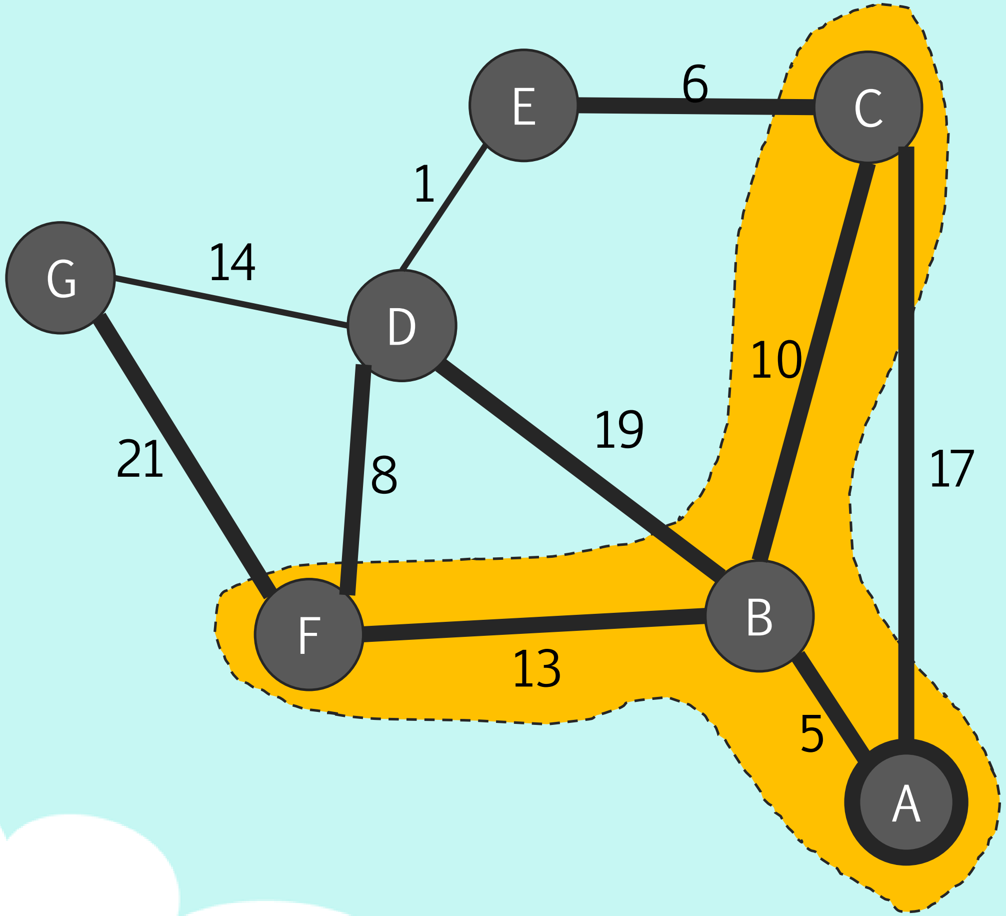
G is now reachable, but it's not guaranteed this current path [A, B, F, G] is the shortest.



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E	A, B, C, E	21
F	A, B, F	18
G	A, B, F, G	39

Shortest Path Strategy

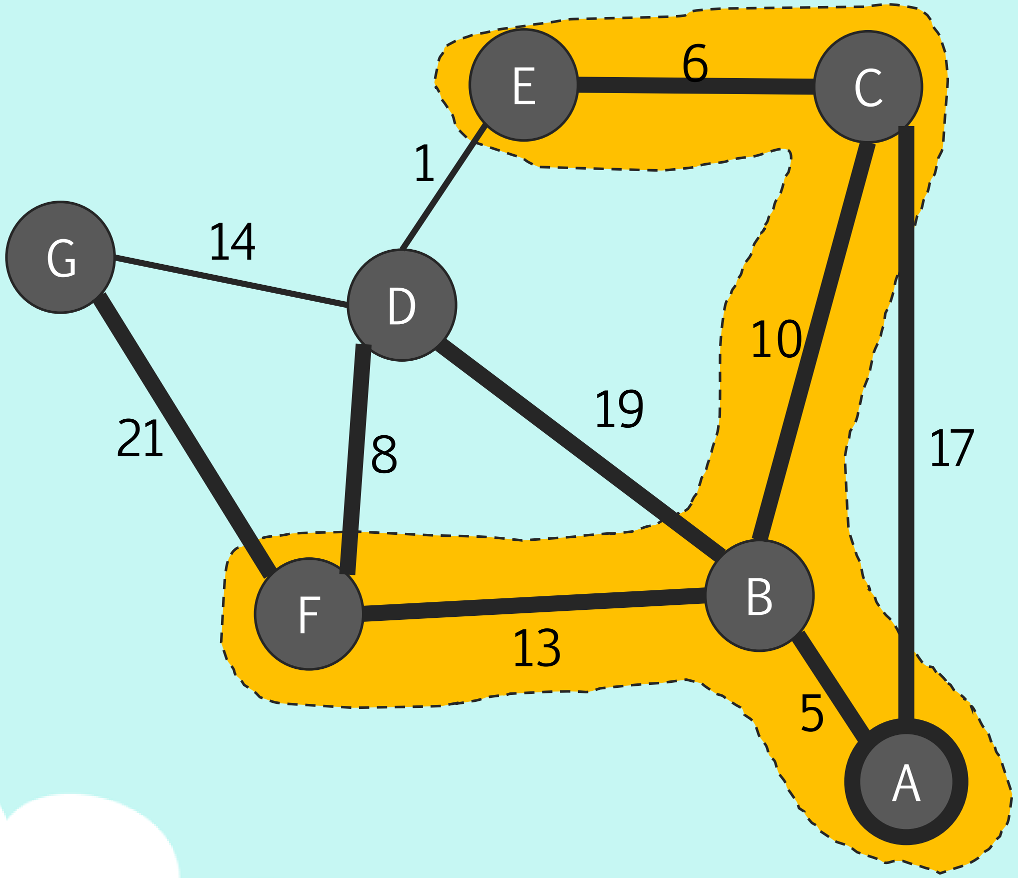
Now out of these paths, let's expand our cloud to the vertex with shortest distance.
E



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E	A, B, C, E	21
F	A, B, F	18
G	A, B, F, G	39

Shortest Path Strategy

Now out of these paths, let's expand our cloud to the vertex with shortest distance.
E

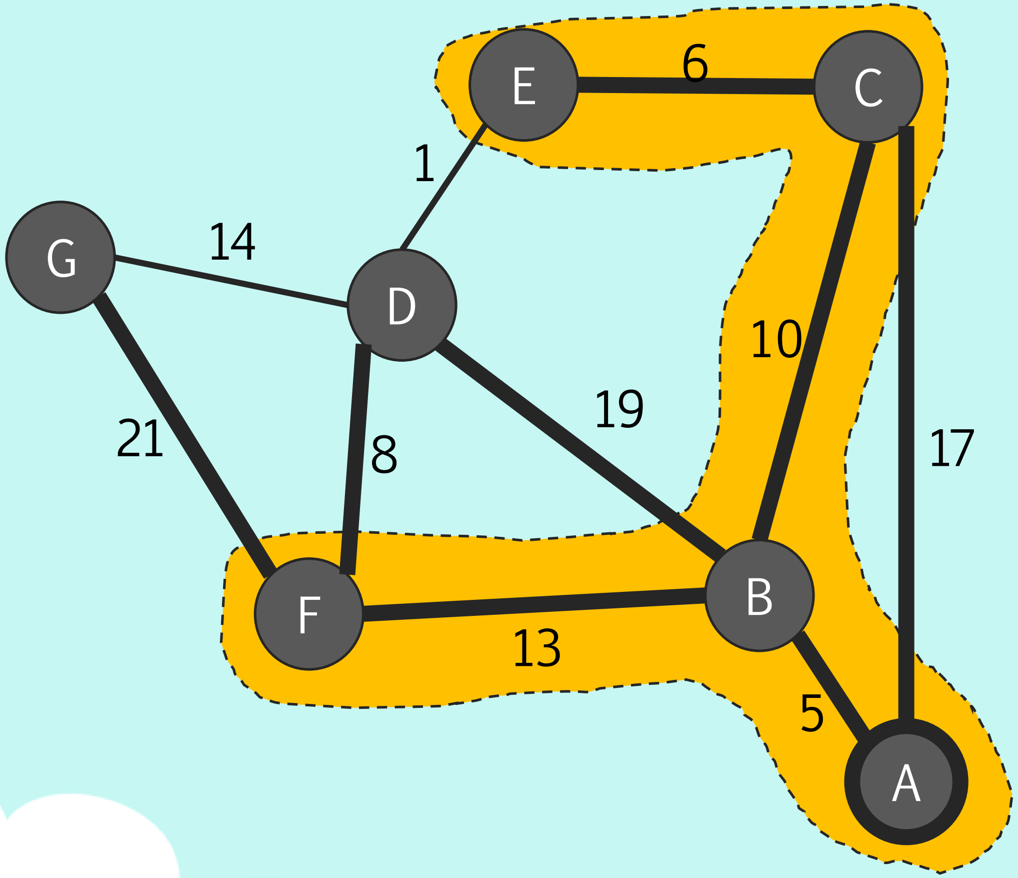


Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E	A, B, C, E	21
F	A, B, F	18
G	A, B, F, G	39

Shortest Path Strategy

Now out of these paths, let's expand our cloud to the vertex with shortest distance.
E

Now we've solidified the shortest path from A to E as [A, B, C, E] with a distance of 21.

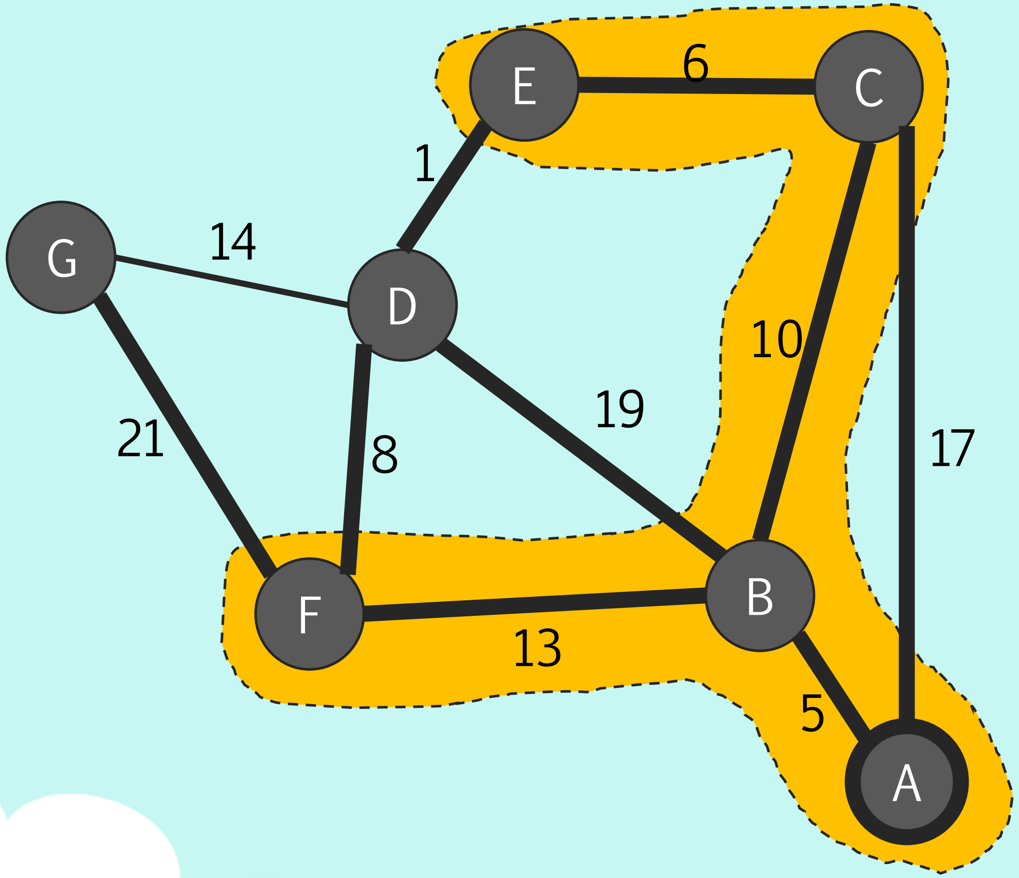


Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E	A, B, C, E	21
F	A, B, F	18
G	A, B, F, G	39

Shortest Path Strategy

From A, I can go to:

- D in a distance of 5 + 19
- D in a distance of 18 + 8
- D in a distance of 21 + 1
- G in a distance of 18 + 21



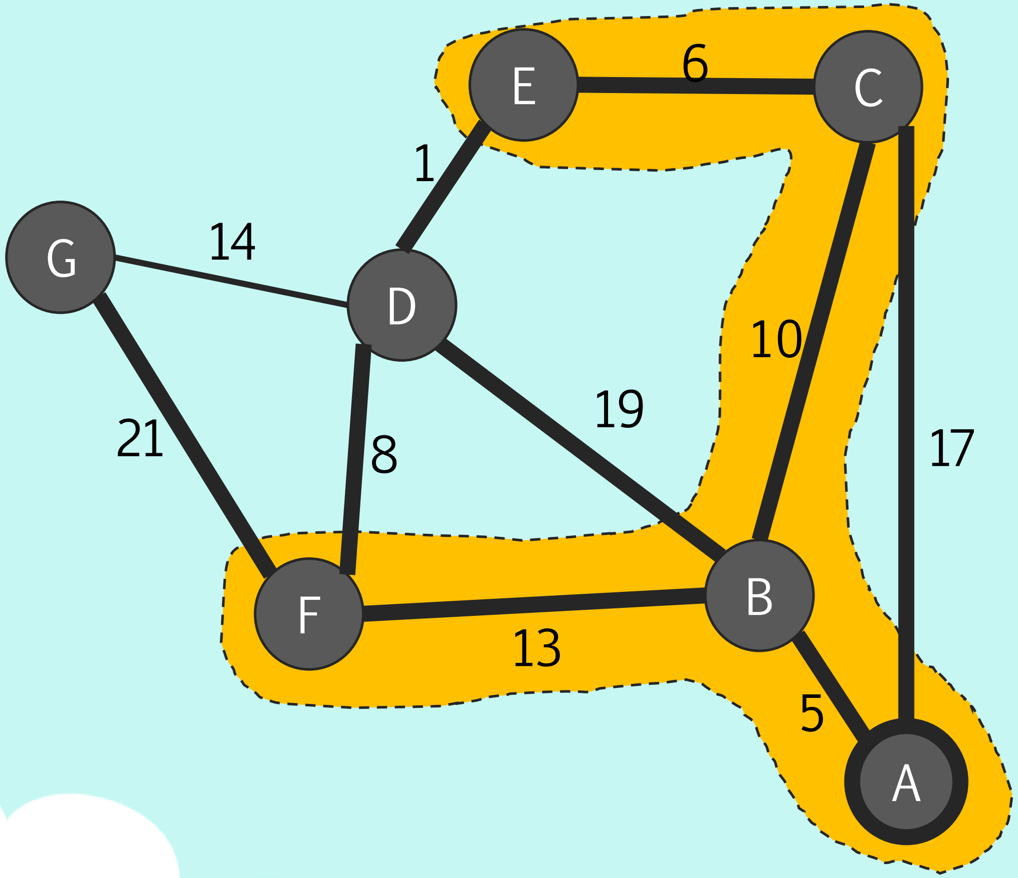
Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E	A, B, C, E	21
F	A, B, F	18
G	A, B, F, G	39

Shortest Path Strategy

From A, I can go to:

- D in a distance of 5 + 19
- D in a distance of 18 + 8
- D in a distance of 21 + 1
- G in a distance of 18 + 21

We now have 3 possible paths to D: [A, B, D], [A, B, F, D], and [A, B, C, E, D]



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E	A, B, C, E	21
F	A, B, F	18
G	A, B, F, G	39

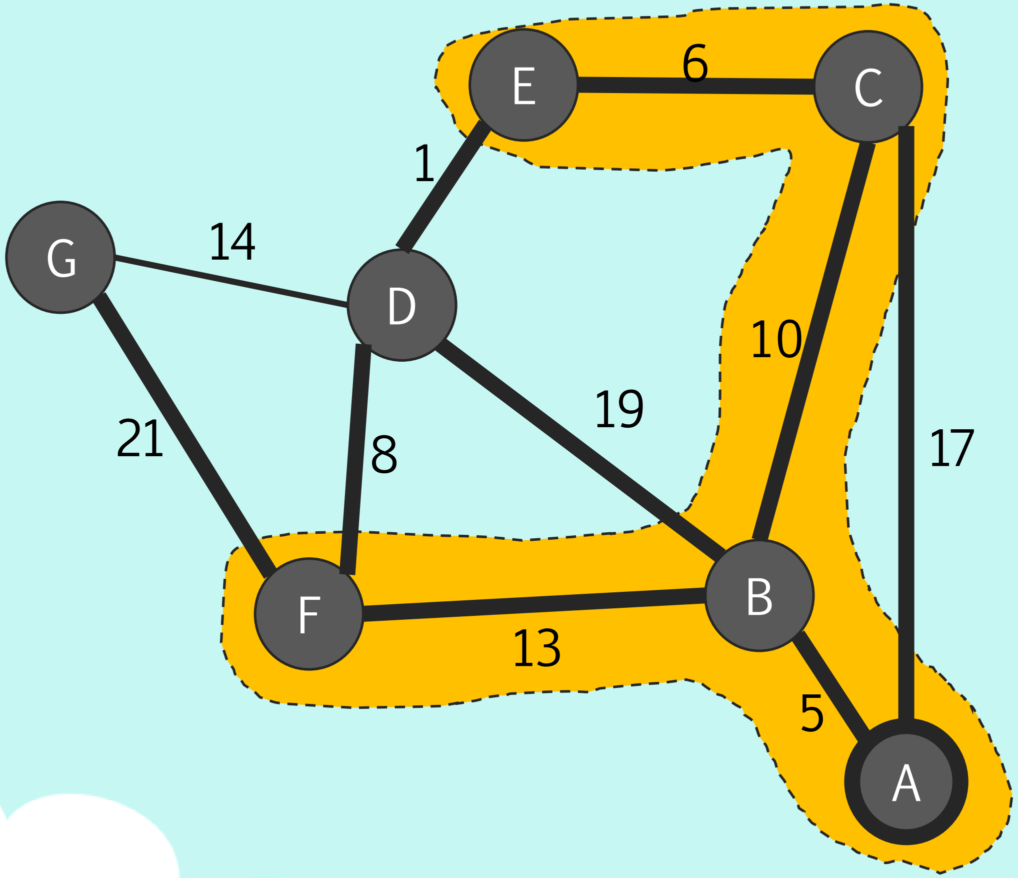
Shortest Path Strategy

From A, I can go to:

- D in a distance of 5 + 19
- D in a distance of 18 + 8
- D in a distance of 21 + 1
- G in a distance of 18 + 21

We now have 3 possible paths to D: [A, B, D], [A, B, F, D], and [A, B, C, E, D]

Let's update our table with these new distances and paths.



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, D	24
E	A, B, C, E	21
F	A, B, F	18
G	A, B, F, G	39

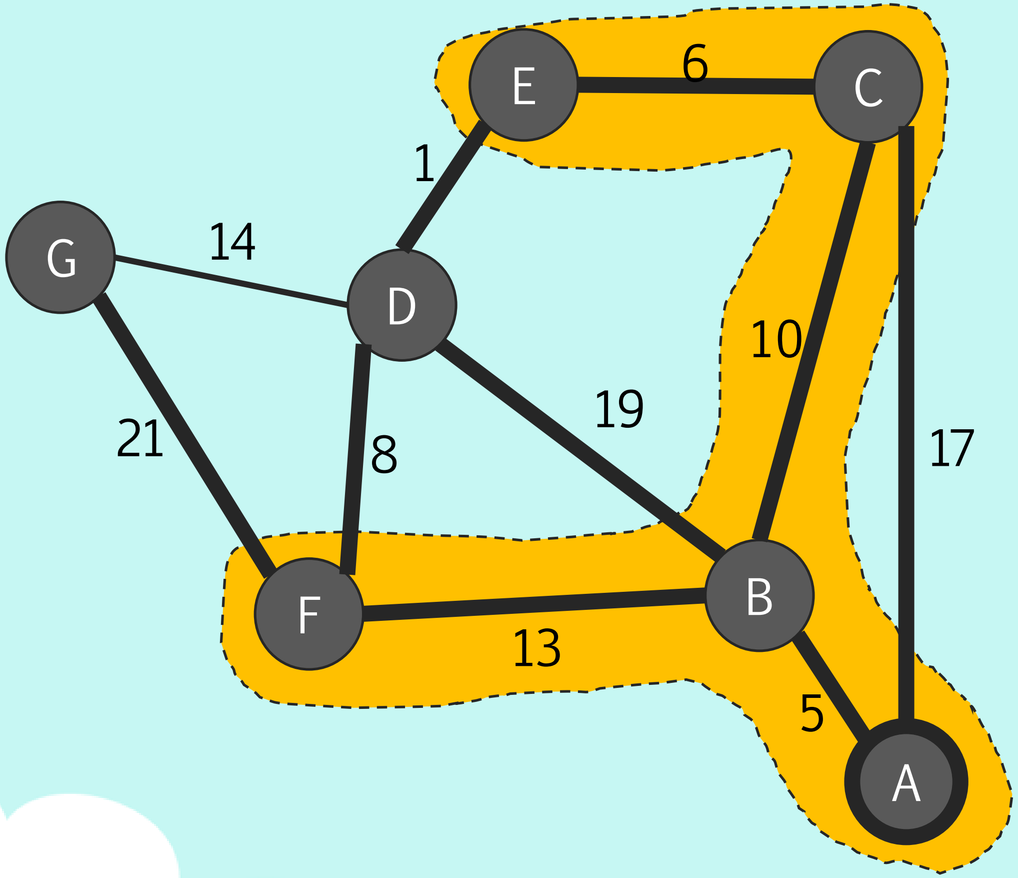
Shortest Path Strategy

From A, I can go to:

- D in a distance of 5 + 19
- D in a distance of 18 + 8
- D in a distance of 21 + 1
- G in a distance of 18 + 21

We now have 3 possible paths to D: [A, B, D], [A, B, F, D], and [A, B, C, E, D]

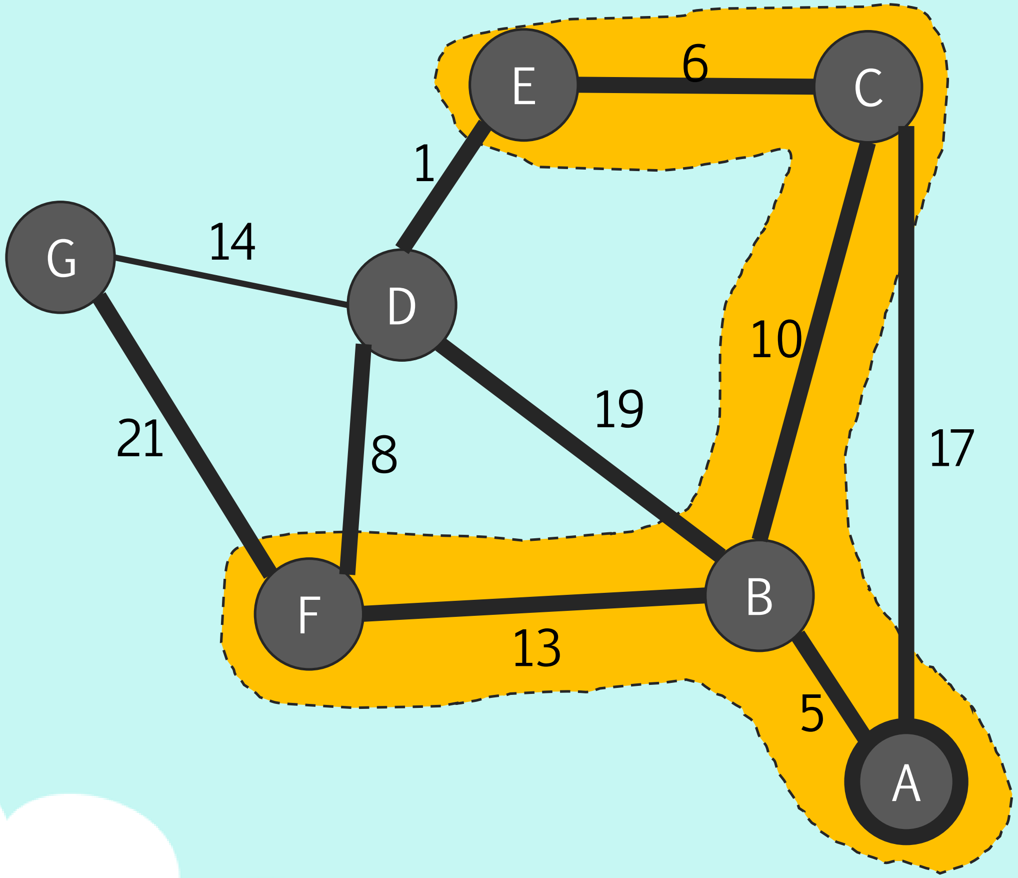
Let's update our table with these new distances and paths.



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, C, E, D	22
E	A, B, C, E	21
F	A, B, F	18
G	A, B, F, G	39

Shortest Path Strategy

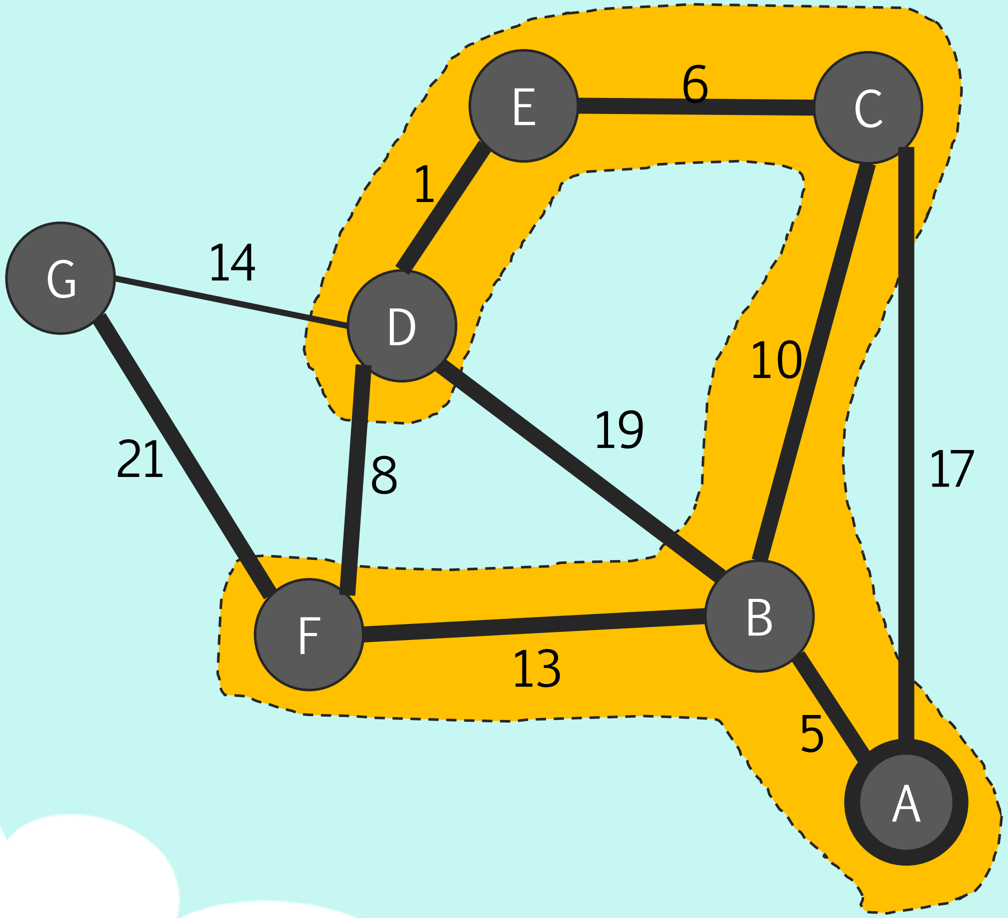
Now out of these paths, let's expand our cloud to the vertex with shortest distance.
D



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, C, E, D	22
E	A, B, C, E	21
F	A, B, F	18
G	A, B, F, G	39

Shortest Path Strategy

Now out of these paths, let's expand our cloud to the vertex with shortest distance.
D

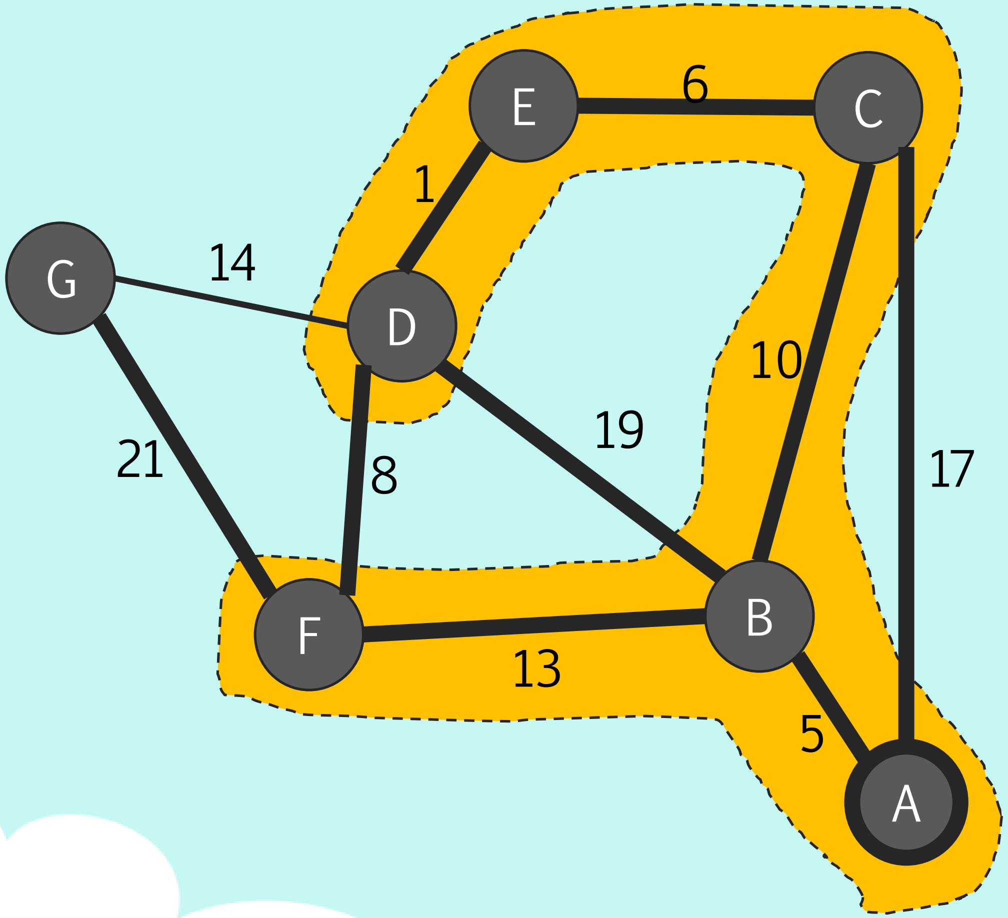


Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, C, E, D	22
E	A, B, C, E	21
F	A, B, F	18
G	A, B, F, G	39

Shortest Path Strategy

Now out of these paths, let's expand our cloud to the vertex with shortest distance.
D

Now we've solidified the shortest path from A to D as [A, B, C, E, D] with a distance of 22.

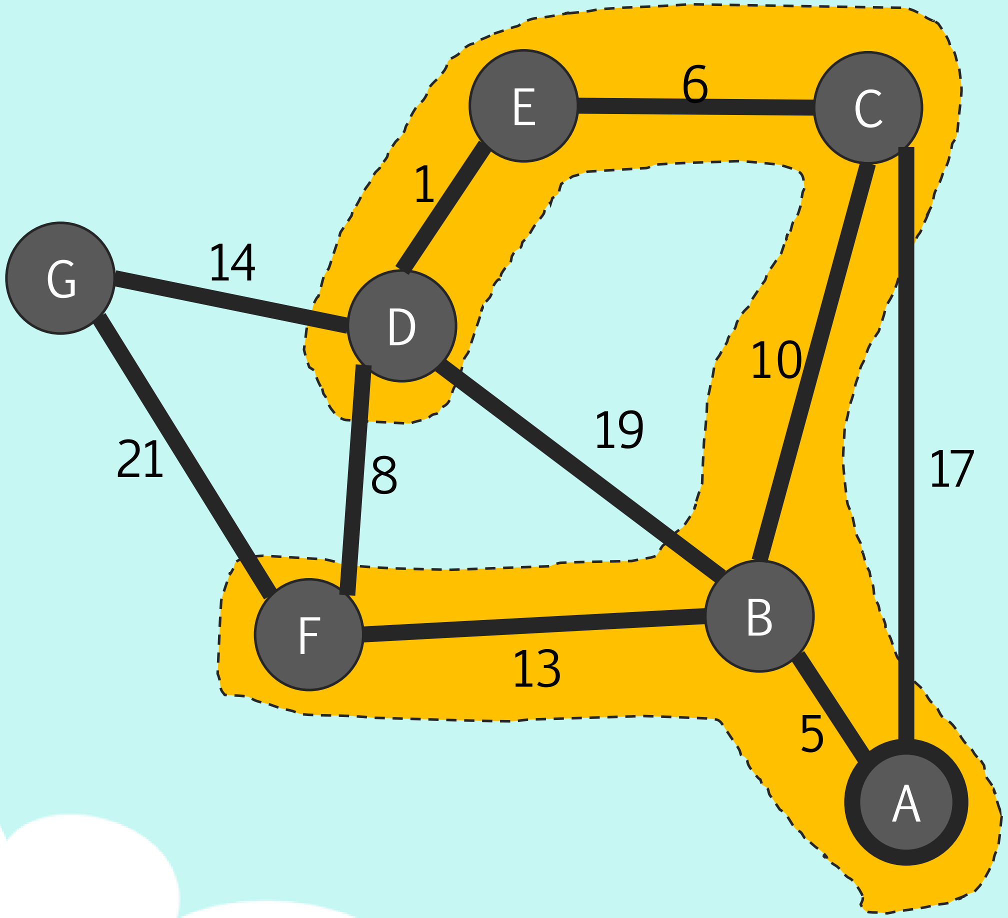


Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, C, E, D	22
E	A, B, C, E	21
F	A, B, F	18
G	A, B, F, G	39

Shortest Path Strategy

From A, I can go to:

- G in a distance of 18 + 21
- G in a distance of 22 + 14



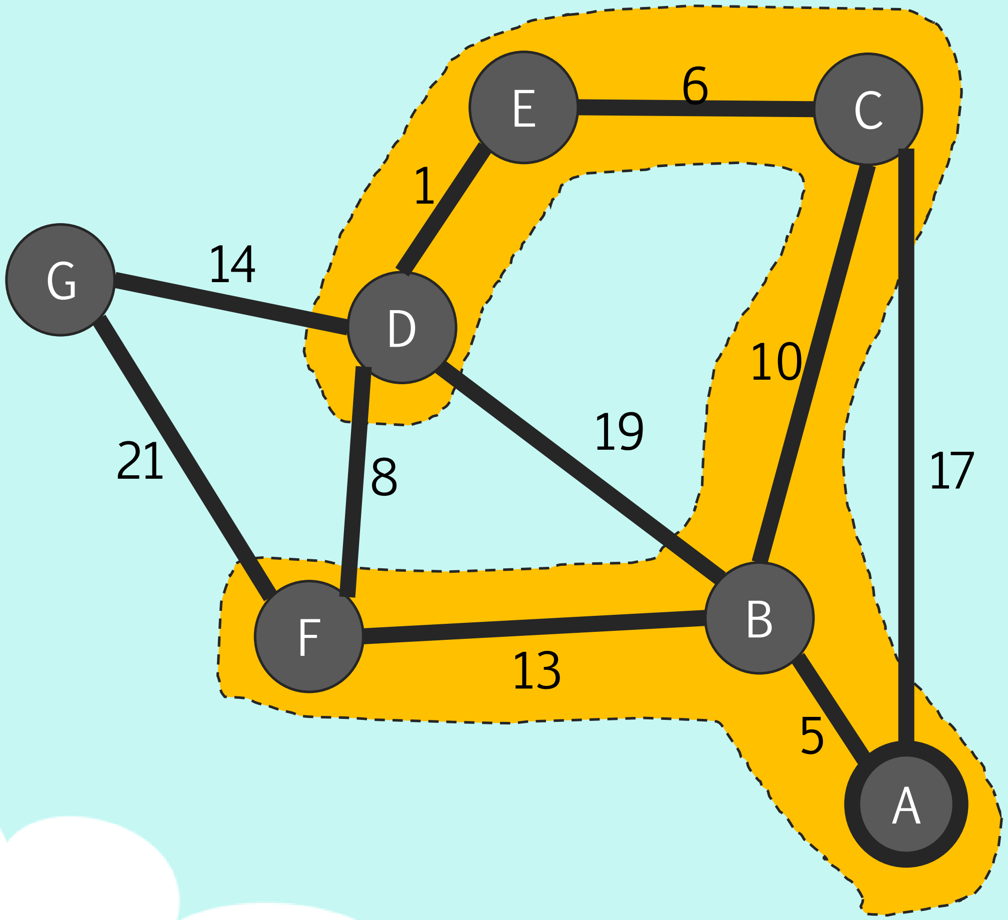
Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, C, E, D	22
E	A, B, C, E	21
F	A, B, F	18
G	A, B, F, G	39

Shortest Path Strategy

From A, I can go to:

- G in a distance of 18 + 21
- G in a distance of 22 + 14

Let's update our table with these new distances and paths.



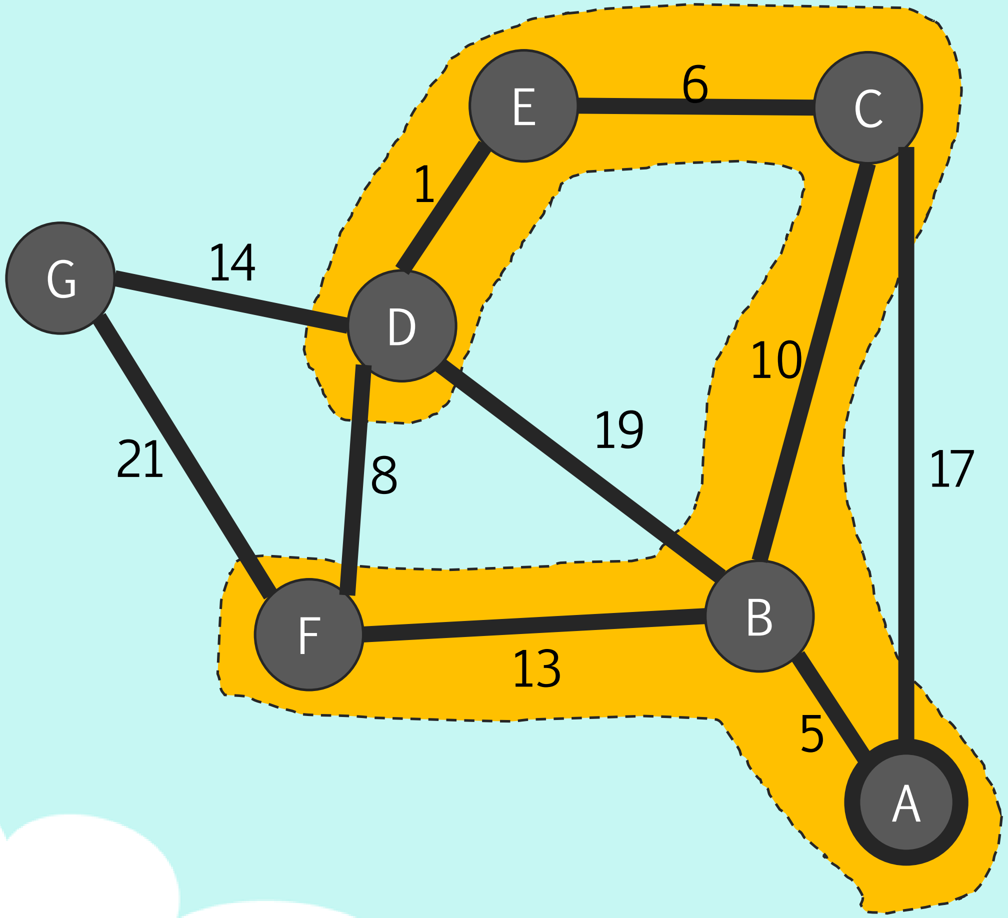
Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, C, E, D	22
E	A, B, C, E	21
F	A, B, F	18
G	A, B, F, G	39

Shortest Path Strategy

From A, I can go to:

- G in a distance of 18 + 21
- G in a distance of 22 + 14

Let's update our table with these new distances and paths.



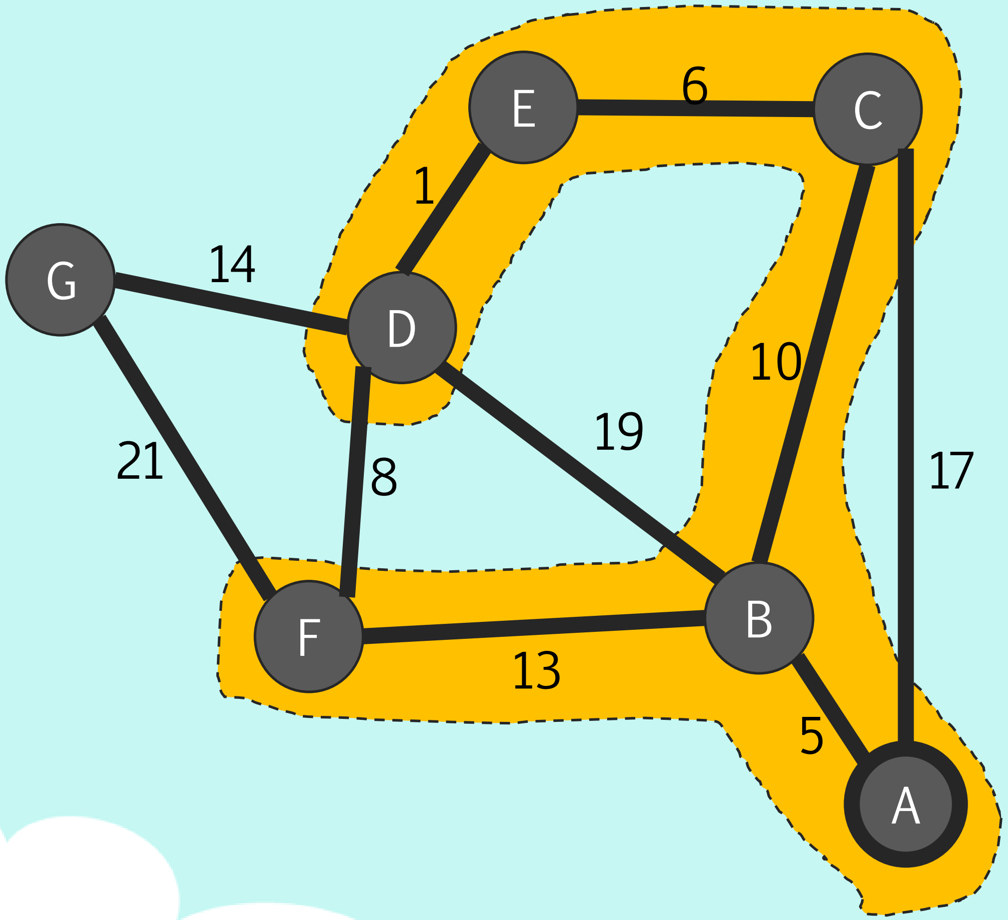
Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, C, E, D	22
E	A, B, C, E	21
F	A, B, F	18
G	A, B, C, E, D, G	36

Shortest Path Strategy

From A, I can go to:

- G in a distance of 18 + 21
- G in a distance of 22 + 14

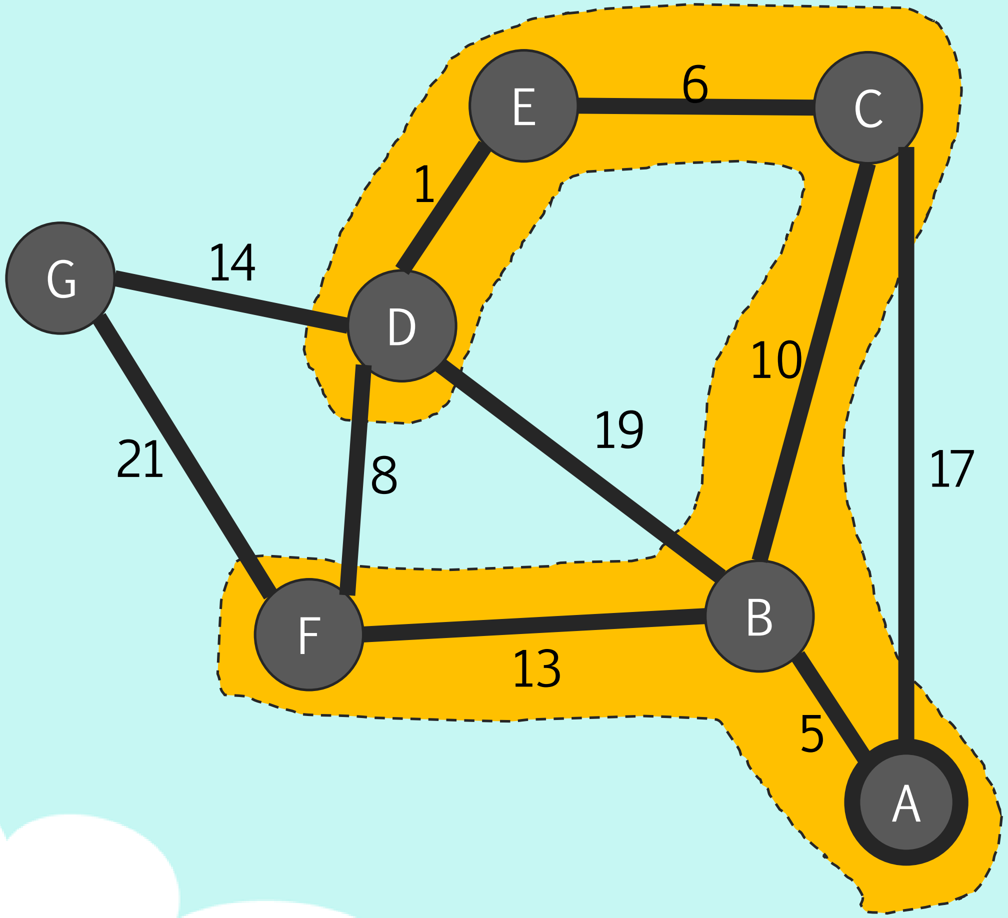
Let's update our table with these new distances and paths.



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, C, E, D	22
E	A, B, C, E	21
F	A, B, F	18
G	A, B, C, E, D, G	36

Shortest Path Strategy

Now out of these paths, let's expand our cloud to the vertex with shortest distance.
G

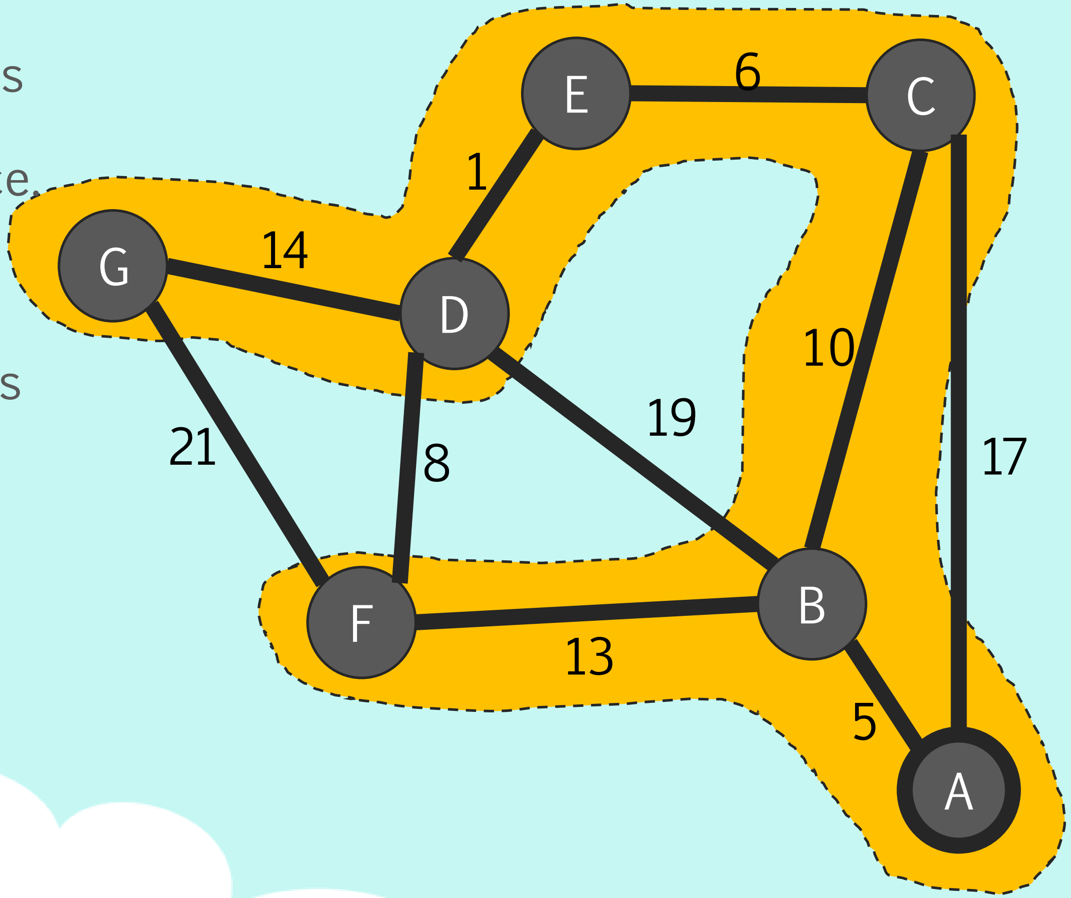


Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, C, E, D	22
E	A, B, C, E	21
F	A, B, F	18
G	A, B, C, E, D, G	36

Shortest Path Strategy

Now out of these paths, let's expand our cloud to the vertex with shortest distance, **G**

Now we've solidified the shortest path from A to G as [A, B, C, E, D, G] with a distance of 36.

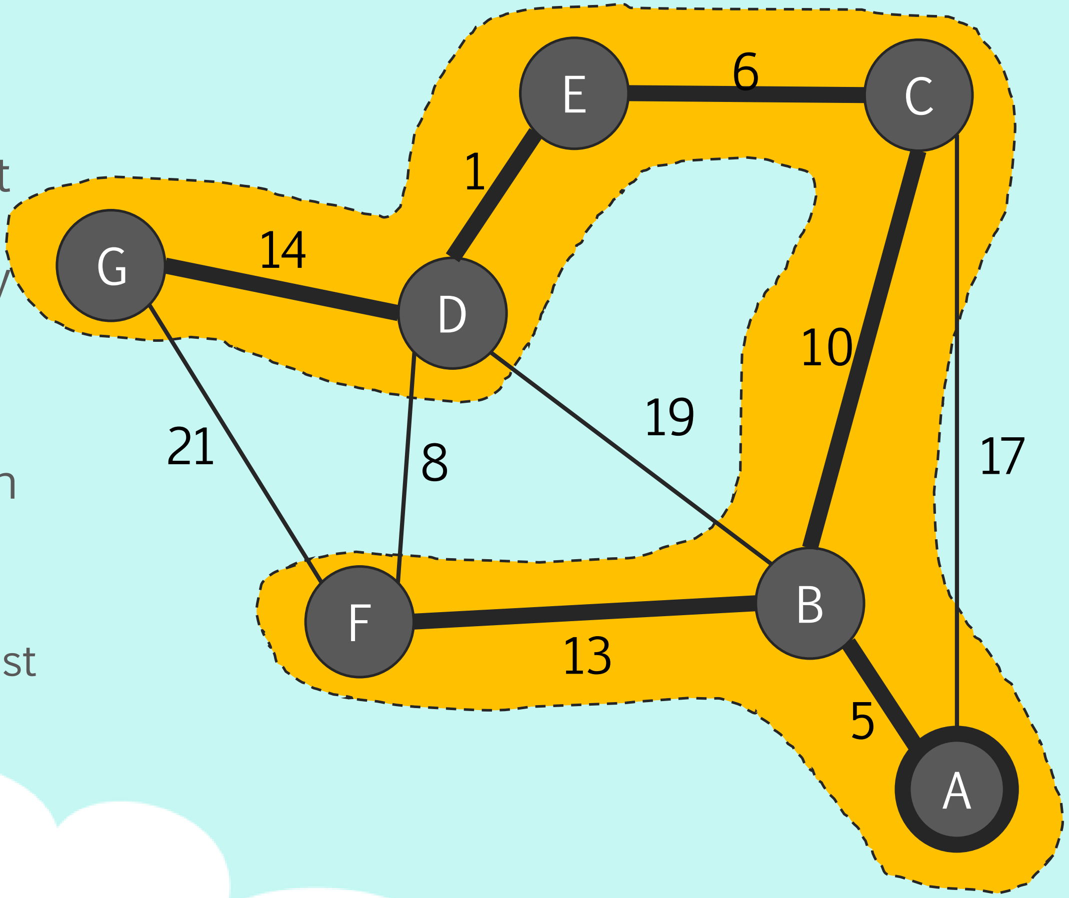


Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, C, E, D	22
E	A, B, C, E	21
F	A, B, F	18
G	**A, B, C, E, D, G**	**36**

Shortest Path Strategy

Observations:

- Starting at A, the shortest paths we solidified were from vertices immediately available from the cloud.
- The shortest path A to G involved the shortest path from A to vertices in between A and G.
  - We calculated the shortest path to these in-between vertices first.



Vertex	Path	Dist
A	A	0
B	A, B	5
C	A, B, C	15
D	A, B, C, E, D	22
E	A, B, C, E	21
F	A, B, F	18
G	**A, B, C, E, D, G**	**36**

Dijkstra's Shortest Path Algorithm

- What we did is called Dijkstra's Shortest Path Algorithm
- Dijkstra's algorithm will calculate the shortest path distnace from a start vertex to every other vertex in a graph.
  - In our case, we had a specific goal vertex: G.
- Dijkstra's performs as a **greedy algorithm**.
  - Given calculated distances to vertices, we expanded our cloud to the vertex with shortest distance. We then used these shortest distances to get to our goal.
- Graph Assumptions:
  - Graph is connected.
  - Edge Weights are non-negative.

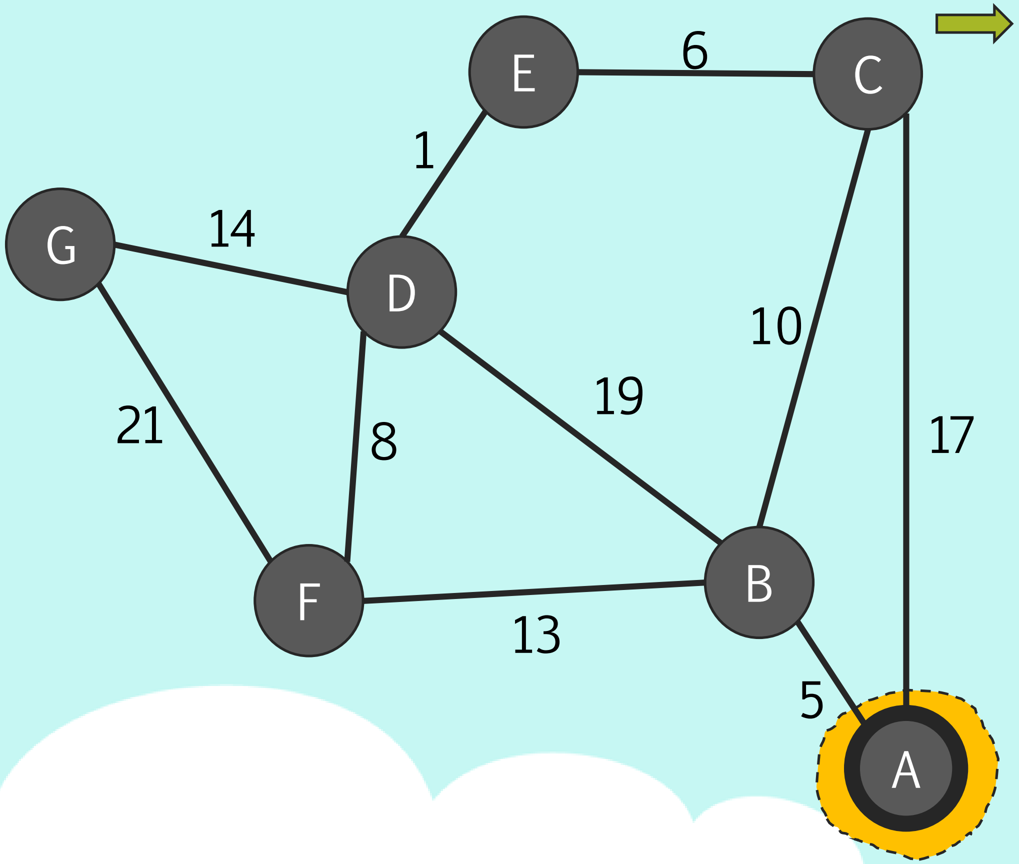
Dijkstra's w/ General Graph Search

```
GraphSearch(start, goal)
  Set visited
  Structure s
  s.add(start)
  while (s not empty)
    curr = s.remove()
    if (curr is visited)
      continue
    visited.add(curr)
    evaluate(curr) // do something if curr is the goal
    for Vertex u in neighbors(curr)
      s.add(u)
```

Dijkstra's w/ General Graph Search

```
Dijkstra(start, goal)
  Map<Vertex, Integer> paths  // Map of Vertex and Distance
  initialize(paths)  // All V have a distance of INF except start
  PriorityQueue s              // Stores tuples (Vertex, Distance)
                               // Removes tuples by smallest distance
  s.add( (start, 0) ) // (Vertex, Distance)
  while (s not empty)
    curr = s.remove_min()
    if (paths[curr.vertex] is not INF)
      continue
    paths[curr.vertex] = curr.distance
    evaluate(curr)             // do something if curr is the goal
    for Vertex u in neighbors(curr.vertex)
      if (paths[u] is INF)  // Checks to see if vertex is visited
        s.add((u,
          paths[curr.vertex] + edge(curr.vertex, u)))
```

Shortest Path Strategy w/ Priority Queue



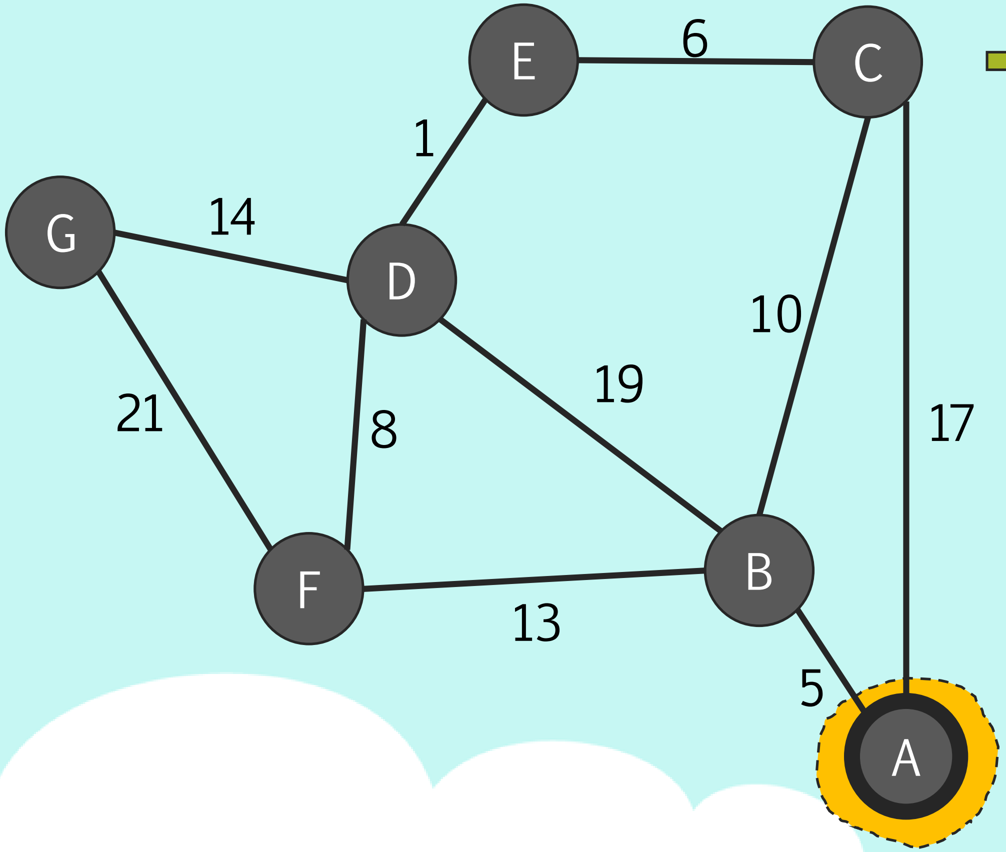

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (A, 0)

Curr:

Vertex	Dist
A	INF
B	INF
C	INF
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



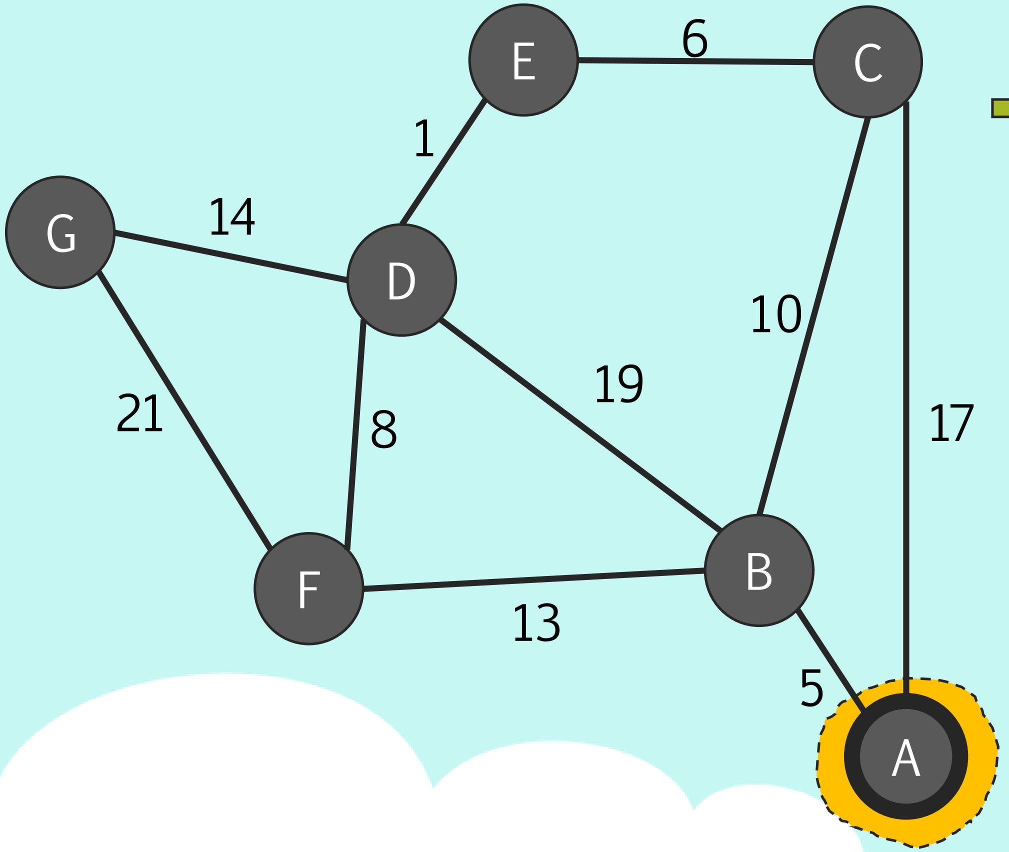

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ:

Curr: (A, 0)

Vertex	Dist
A	INF
B	INF
C	INF
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



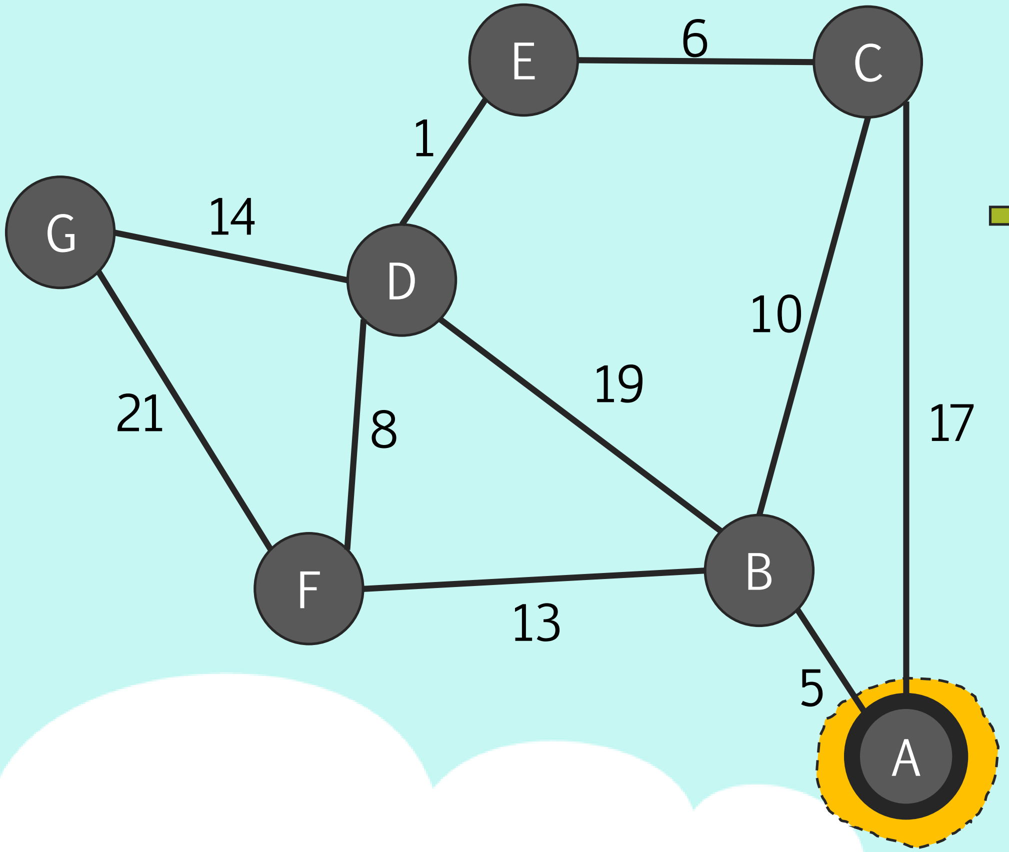

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ:

Curr: (A, 0)

Vertex	Dist
A	INF
B	INF
C	INF
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



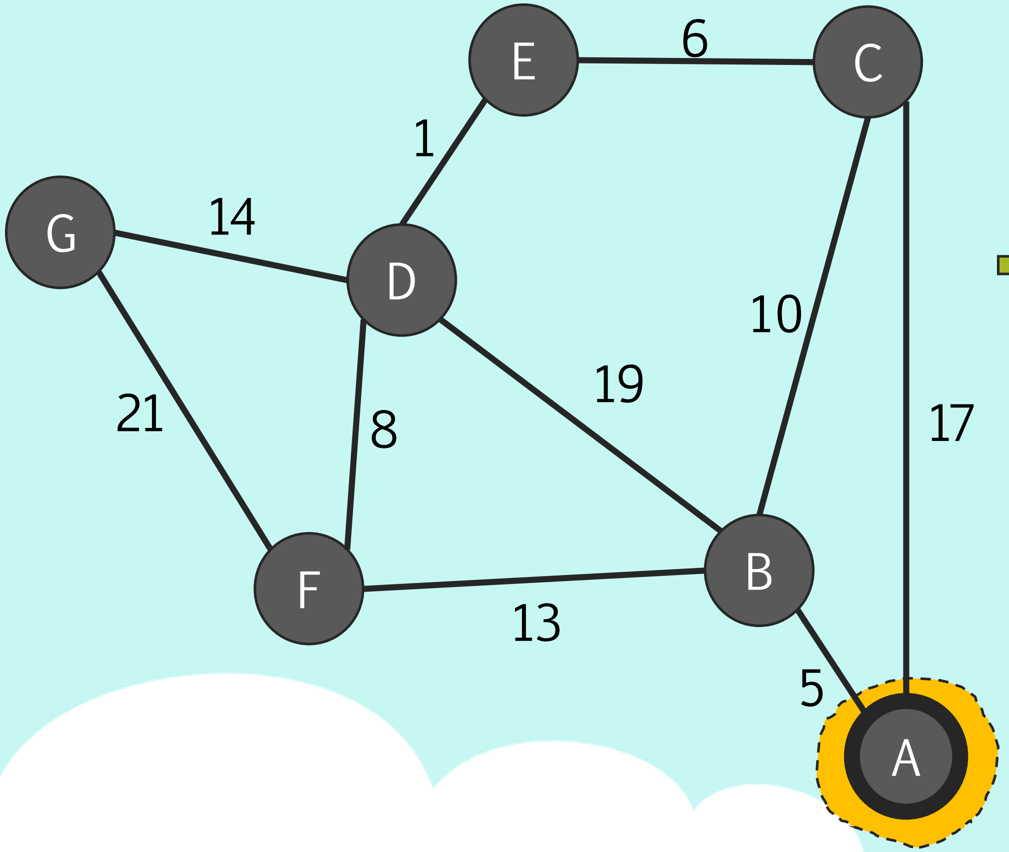

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ:

Curr: (A, 0)

Vertex	Dist
A	0
B	INF
C	INF
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



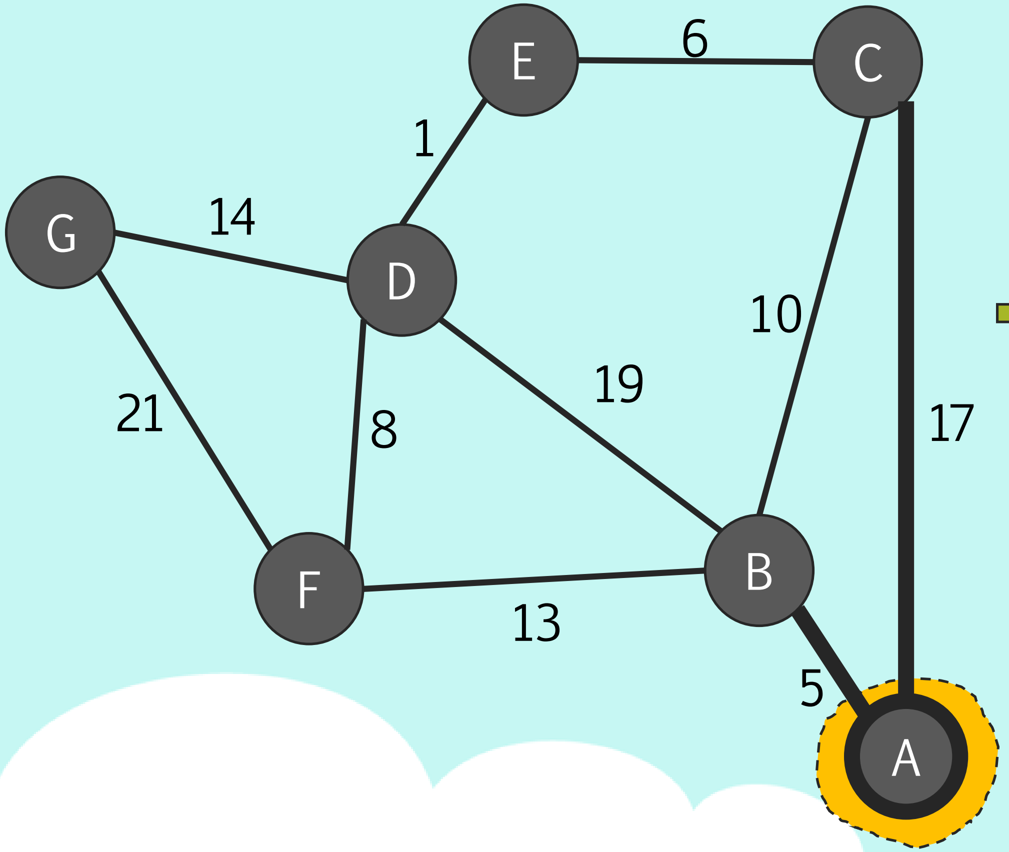

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ:

Curr: (A, 0)

Vertex	Dist
A	0
B	INF
C	INF
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



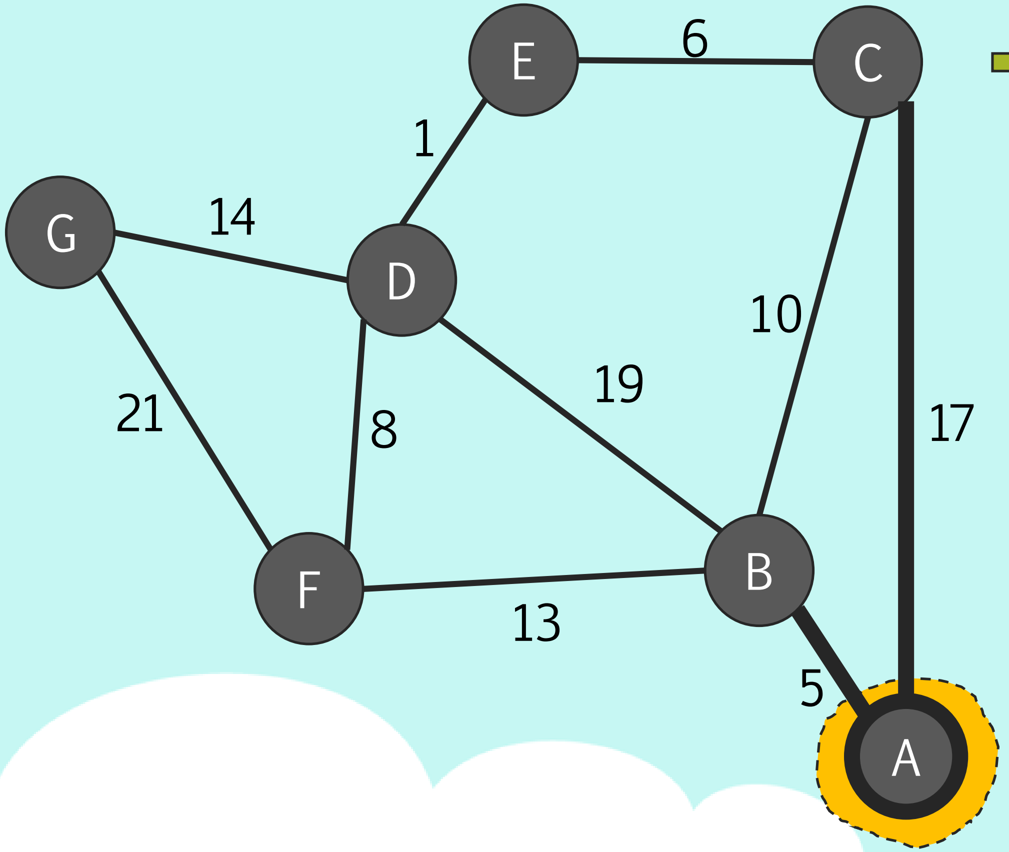

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)  // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (B, 5), (C, 17)

Curr: (A, 0)

Vertex	Dist
A	0
B	INF
C	INF
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



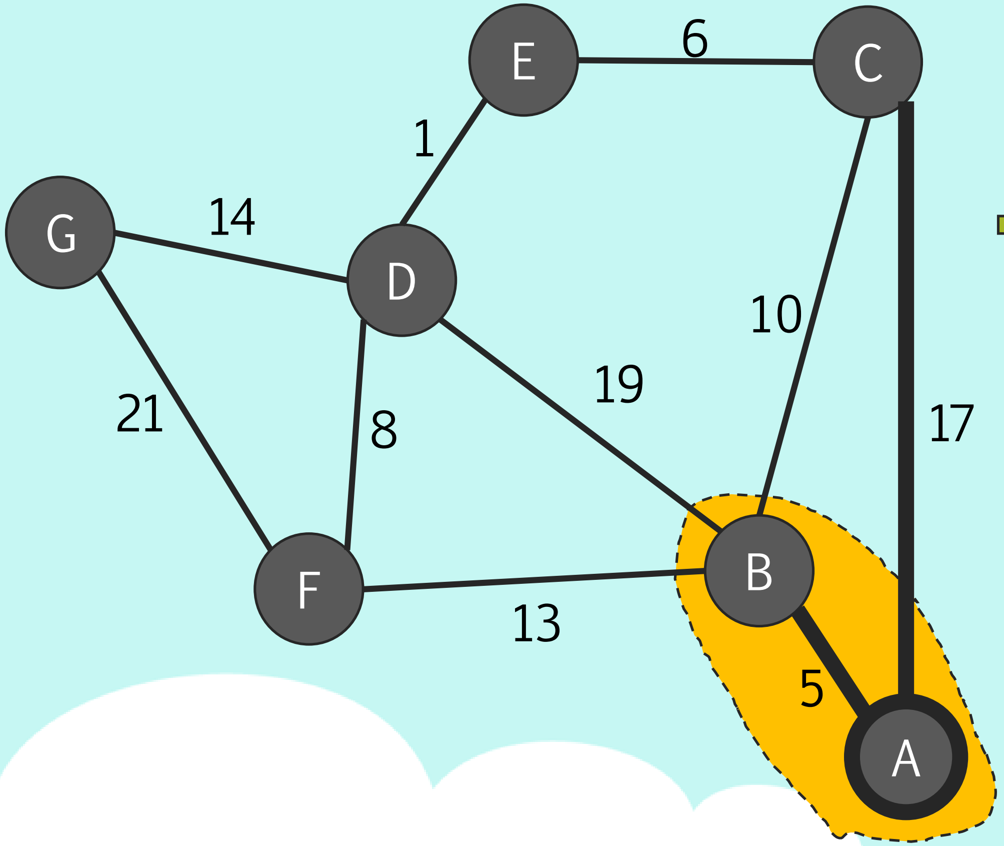

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (C, 17)

Curr: (B, 5)

Vertex	Dist
A	0
B	INF
C	INF
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



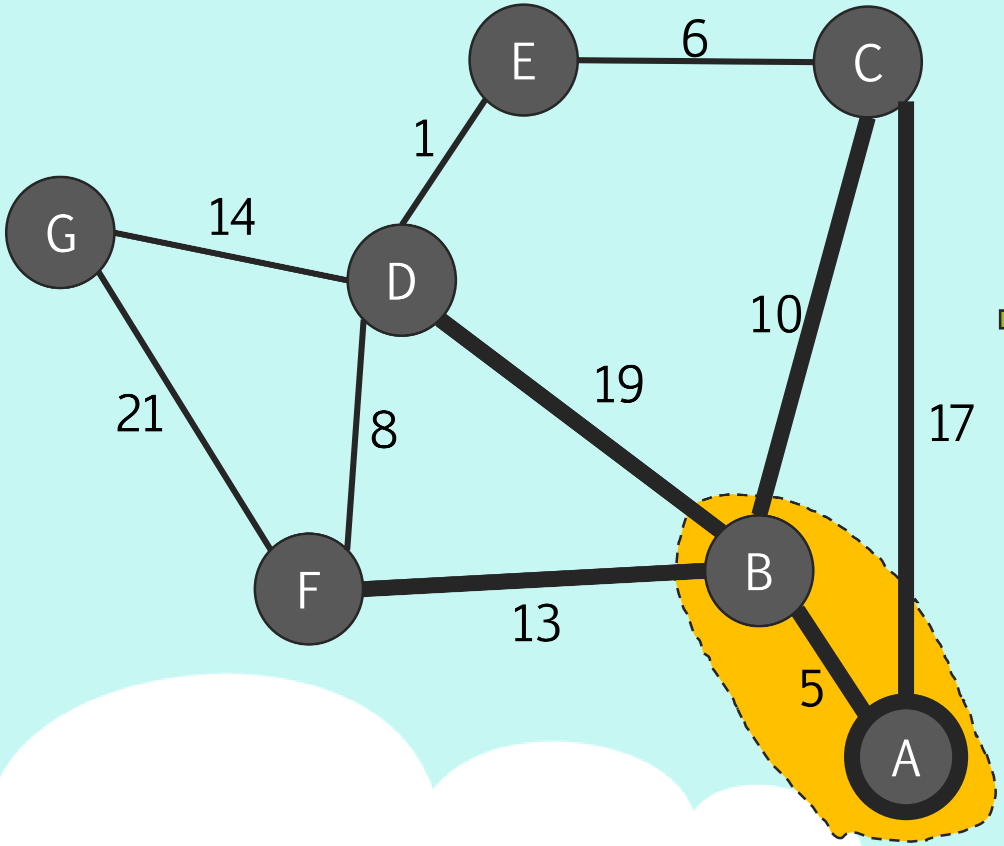

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (C, 17)

Curr: (B, 5)

Vertex	Dist
A	0
B	5
C	INF
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



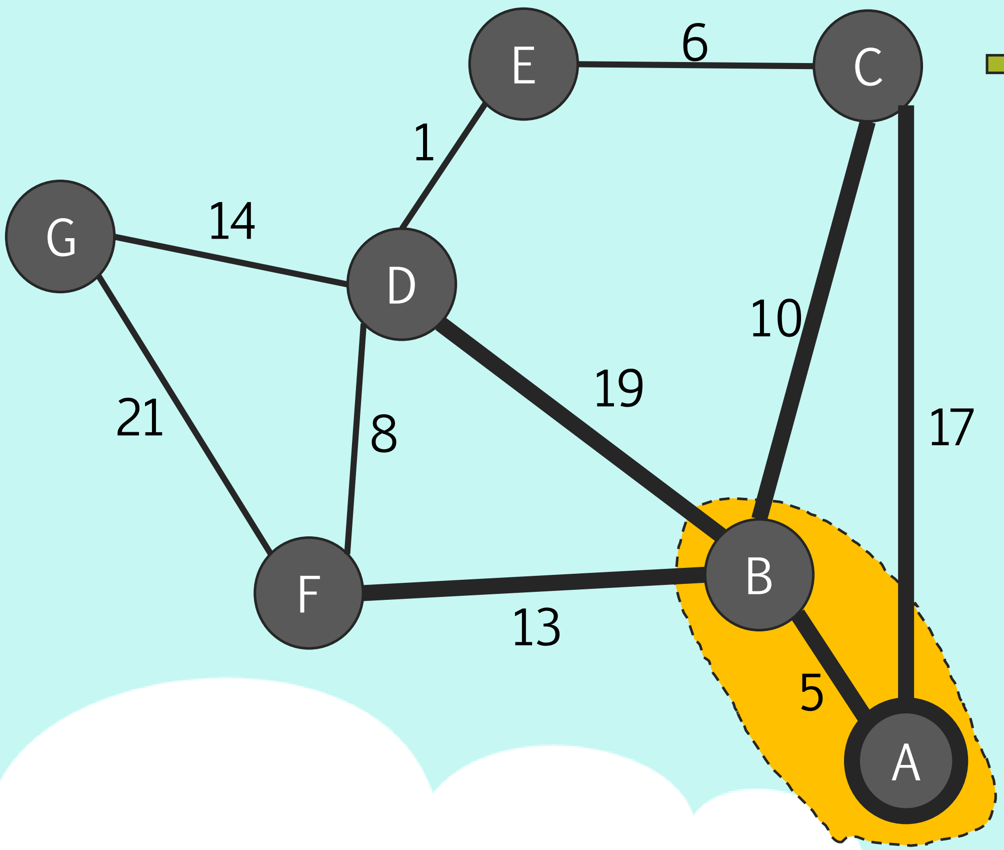

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)  // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (C, 15) (C, 17), (F, 18), (D, 24)

Curr: (B, 5)

Vertex	Dist
A	0
B	5
C	INF
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



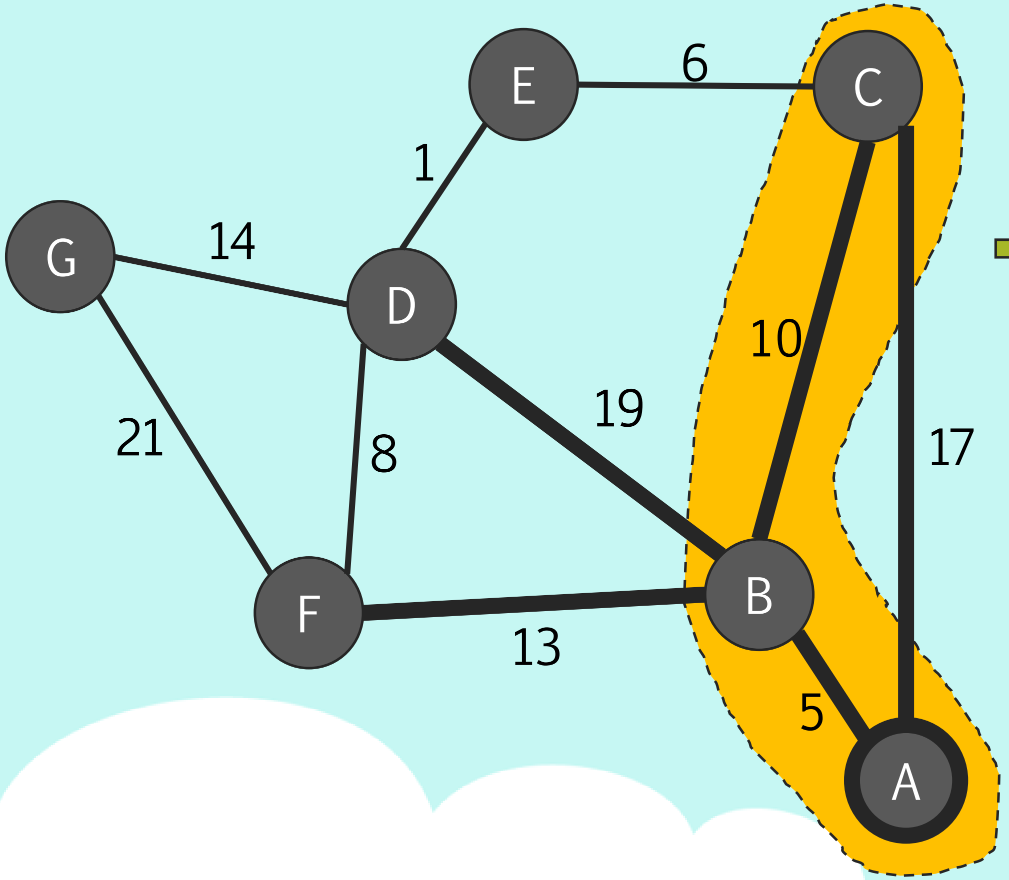

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)  // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (C, 17), (F, 18), (D, 24)

Curr: (C, 15)

Vertex	Dist
A	0
B	5
C	INF
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



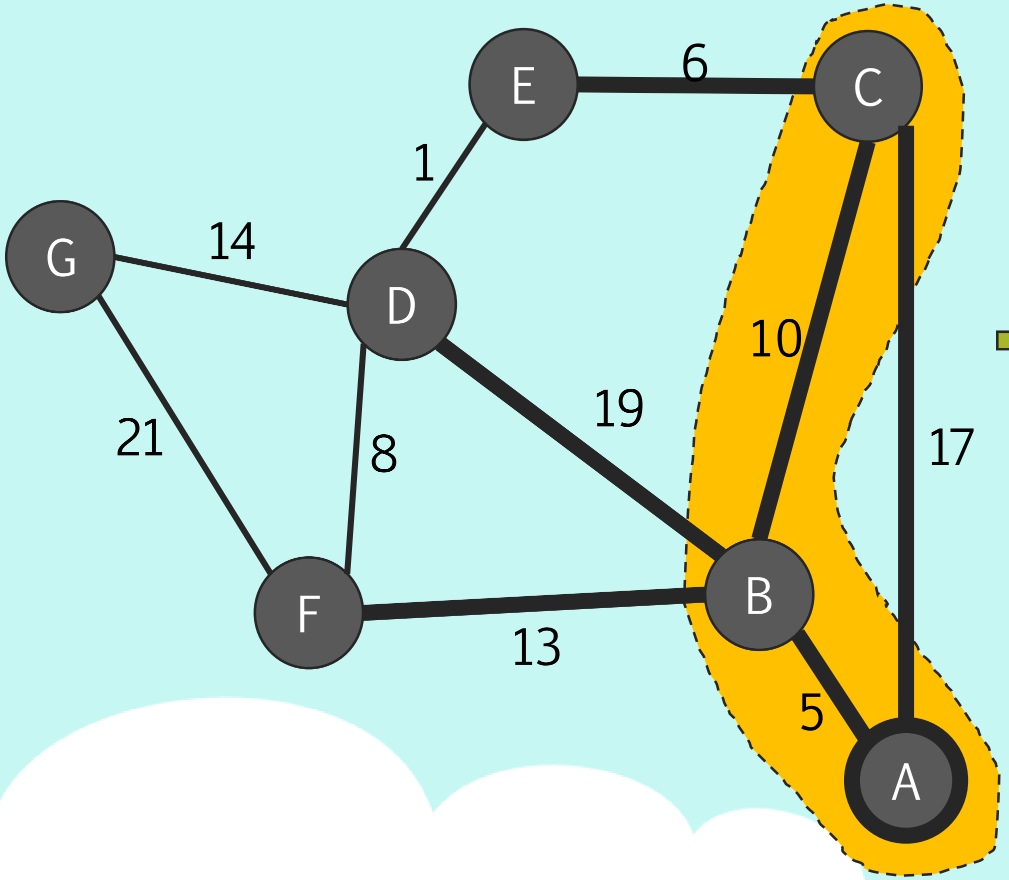

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)  // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (C, 17), (F, 18), (D, 24)

Curr: (C, 15)

Vertex	Dist
A	0
B	5
C	15
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



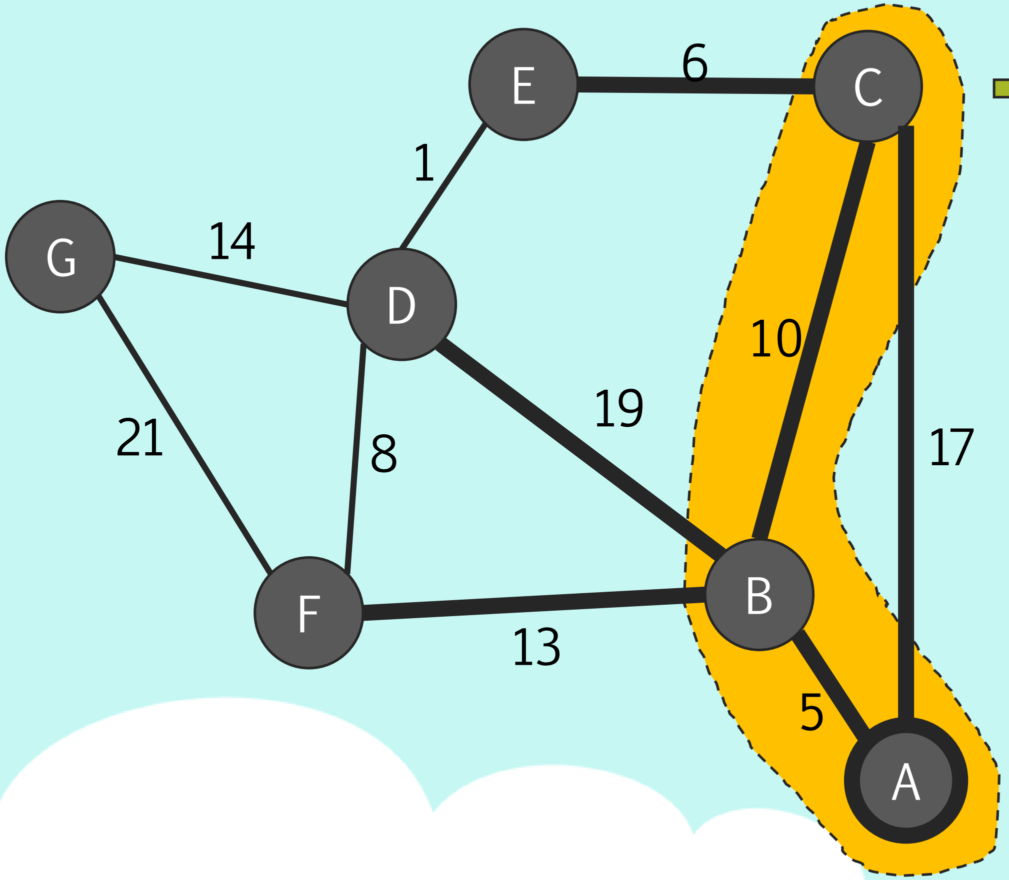

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (C, 17), (F, 18), (E, 21), (D, 24)

Curr: (C, 15)

Vertex	Dist
A	0
B	5
C	15
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



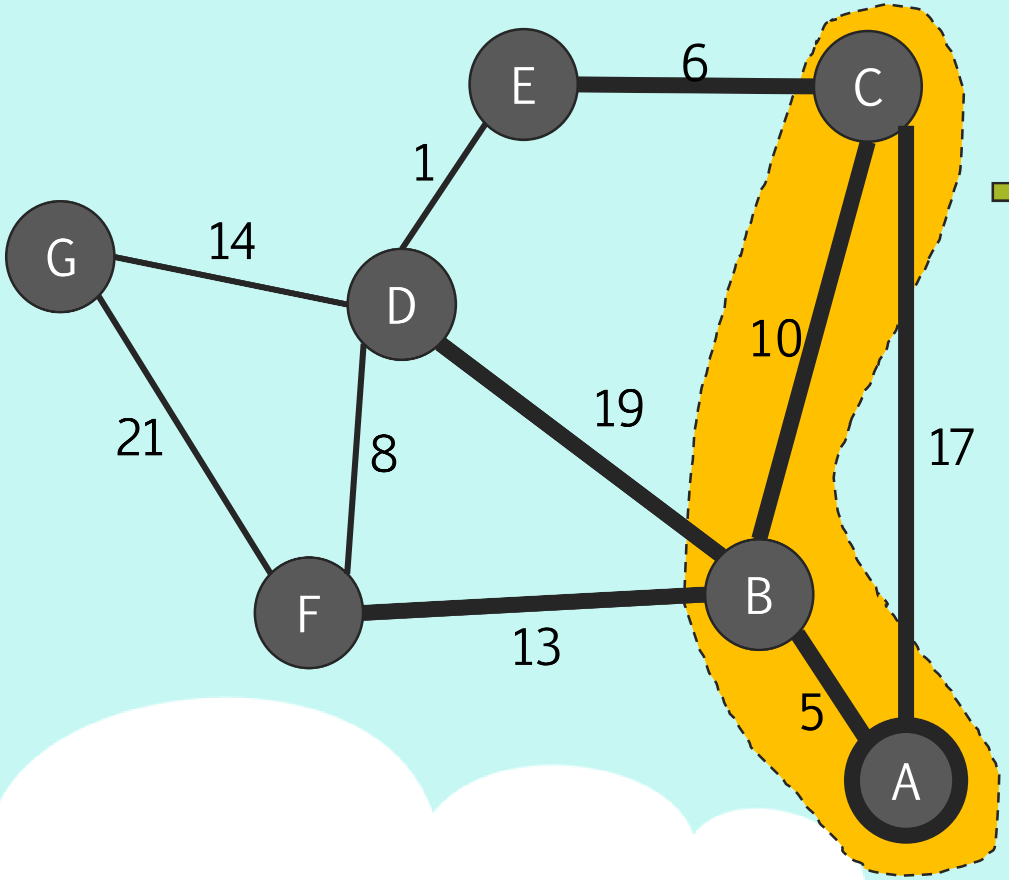

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (F, 18), (E, 21), (D, 24)

Curr: (C, 17)

Vertex	Dist
A	0
B	5
C	15
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



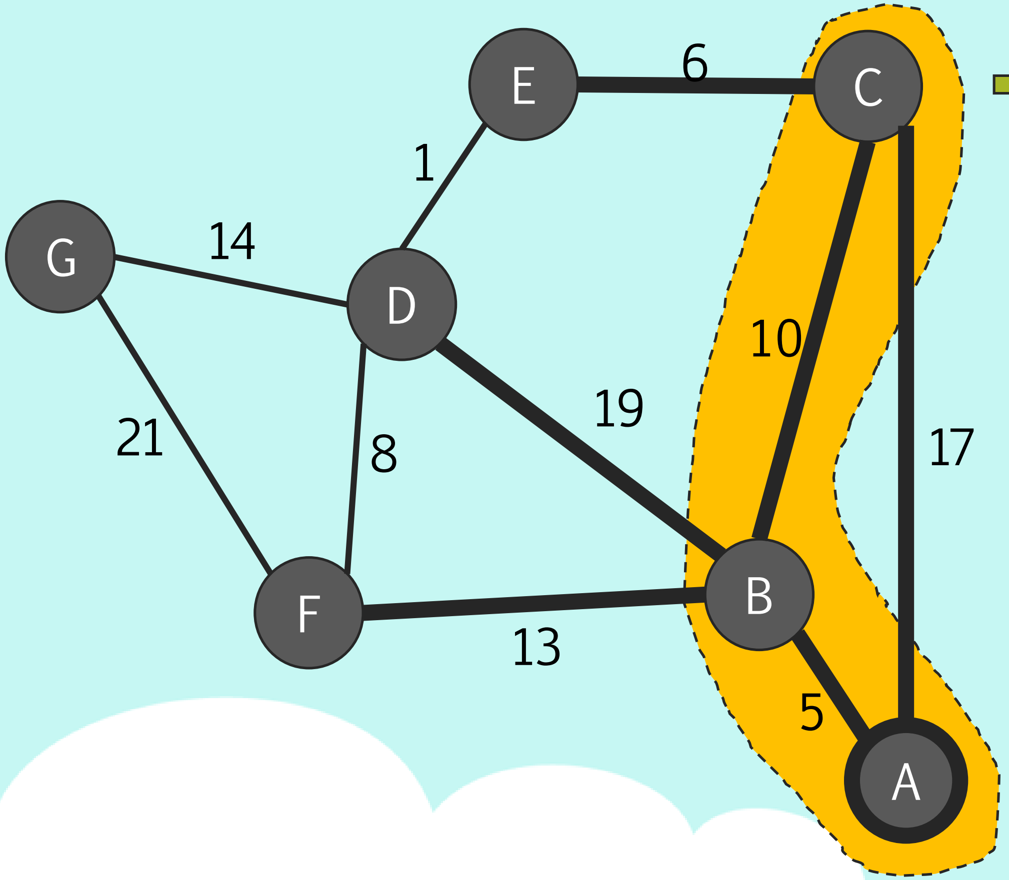

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)  // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (F, 18), (E, 21), (D, 24)

Curr: (C, 17)

Vertex	Dist
A	0
B	5
C	15
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



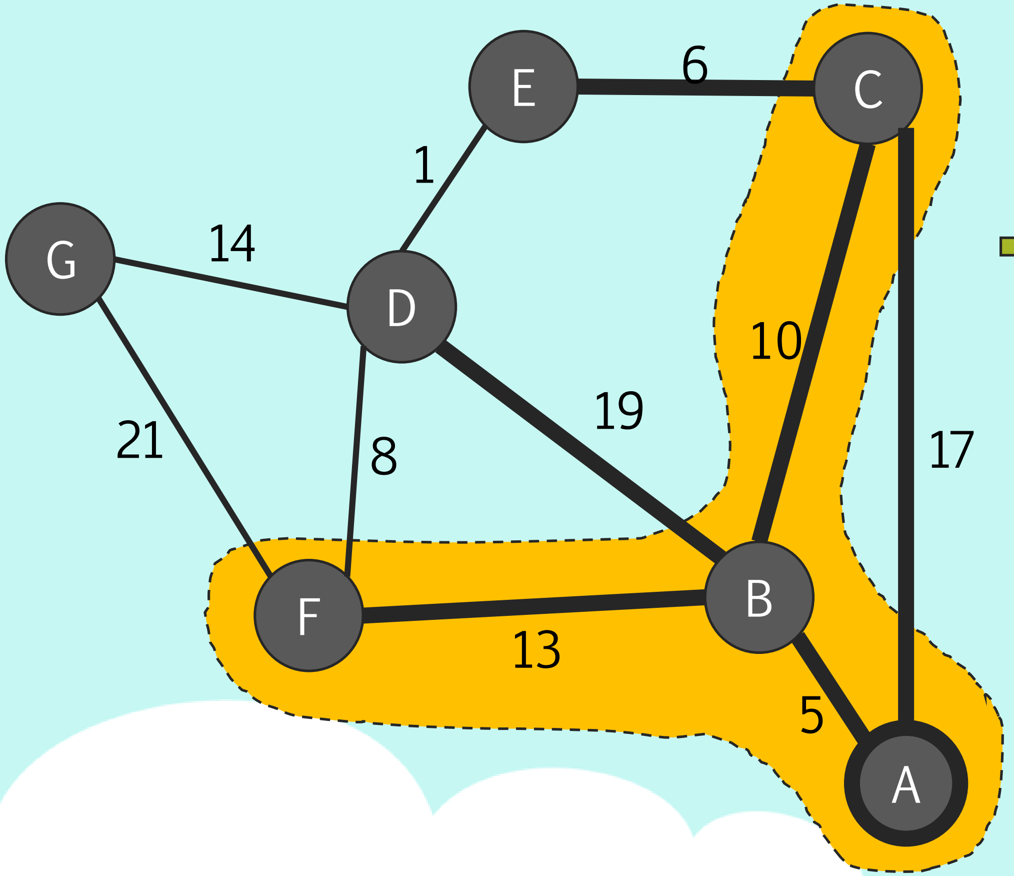

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (E, 21), (D, 24)

Curr: (F, 18)

Vertex	Dist
A	0
B	5
C	15
D	INF
E	INF
F	INF
G	INF

Shortest Path Strategy w/ Priority Queue



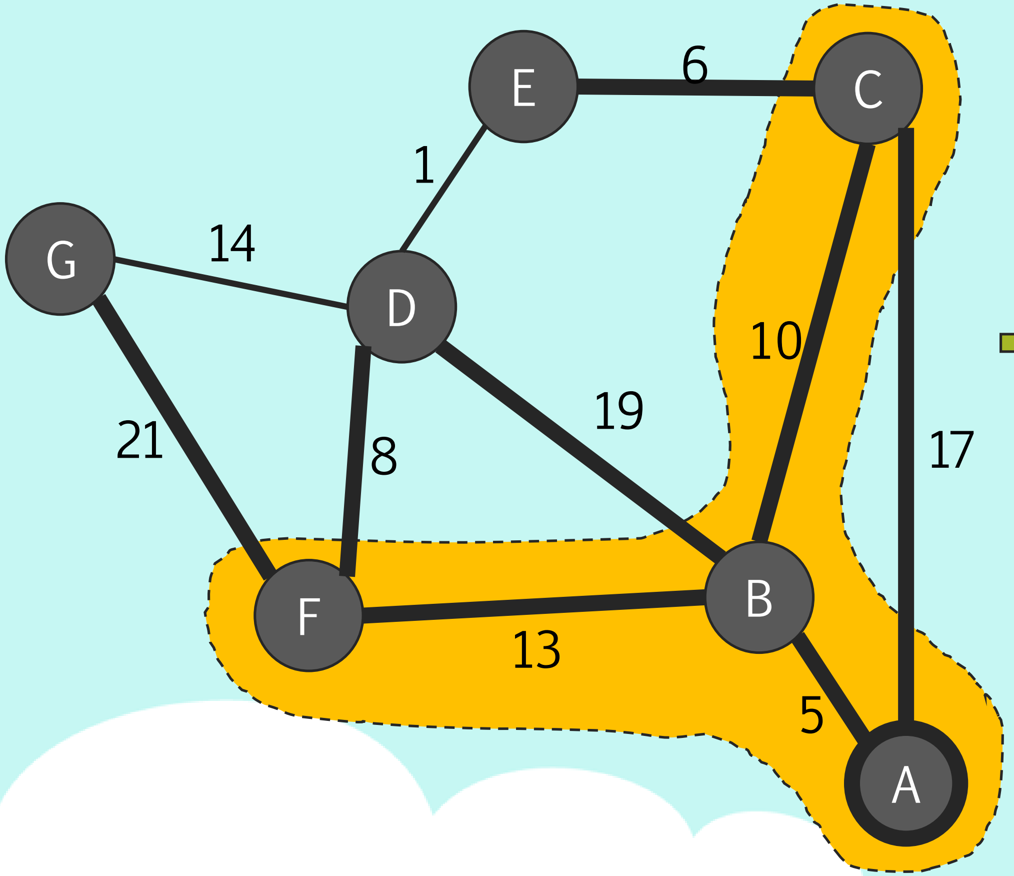

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)  // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (E, 21), (D, 24)

Curr: (F, 18)

Vertex	Dist
A	0
B	5
C	15
D	INF
E	INF
F	18
G	INF

Shortest Path Strategy w/ Priority Queue



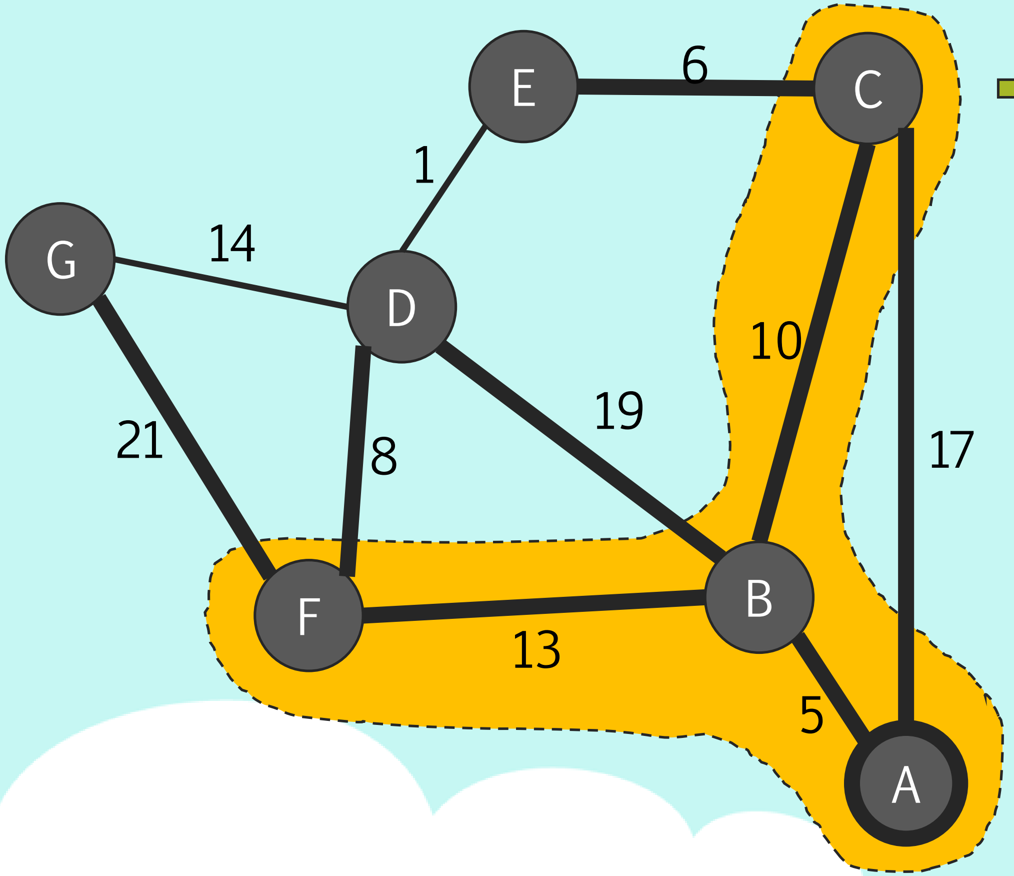

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (E, 21), (D, 24), (D, 26), (G, 39)

Curr: (F, 18)

Vertex	Dist
A	0
B	5
C	15
D	INF
E	INF
F	18
G	INF

Shortest Path Strategy w/ Priority Queue



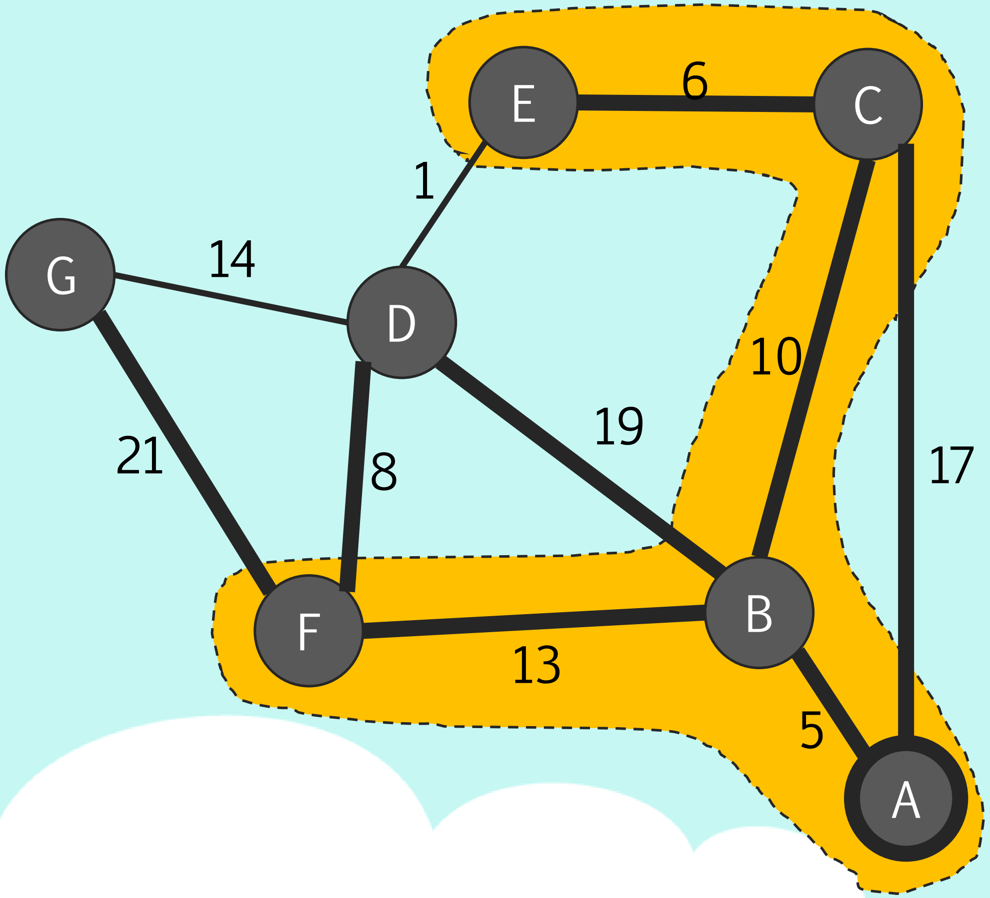

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)  // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (D, 24), (D, 26), (G, 39)

Curr: (E, 21)

Vertex	Dist
A	0
B	5
C	15
D	INF
E	INF
F	18
G	INF

Shortest Path Strategy w/ Priority Queue



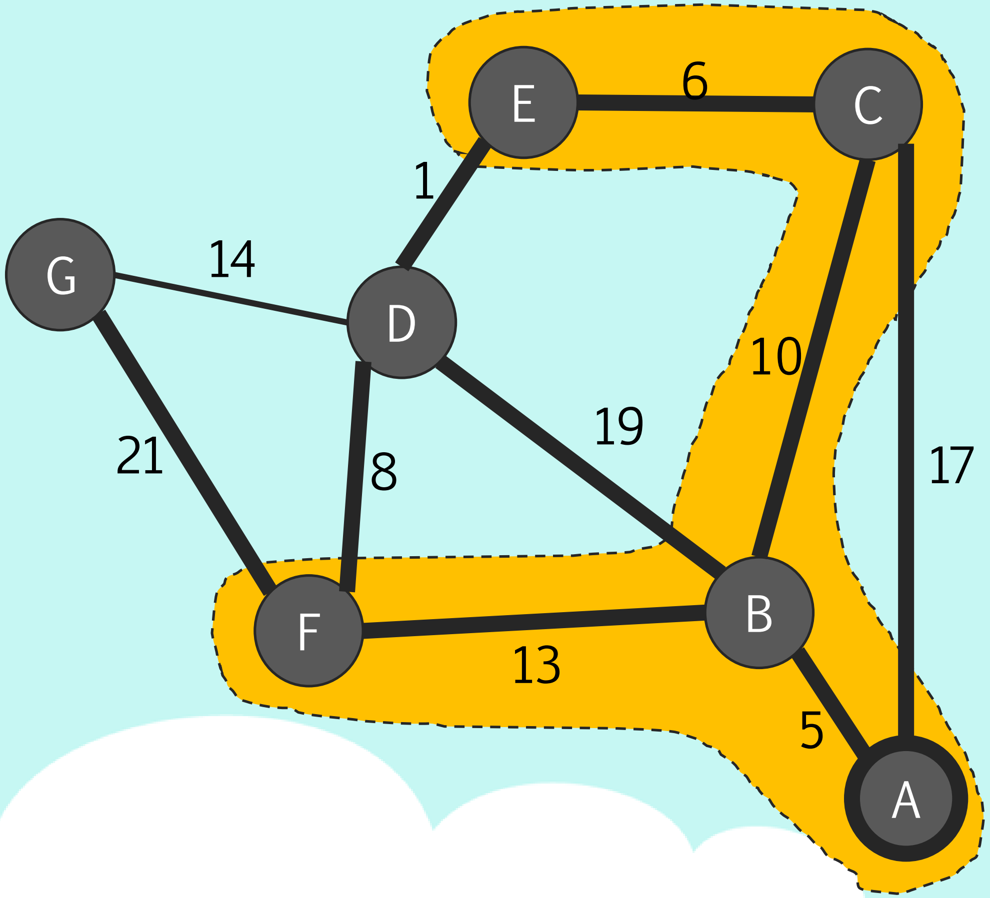

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (D, 24), (D, 26), (G, 39)

Curr: (E, 21)

Vertex	Dist
A	0
B	5
C	15
D	INF
E	21
F	18
G	INF

Shortest Path Strategy w/ Priority Queue



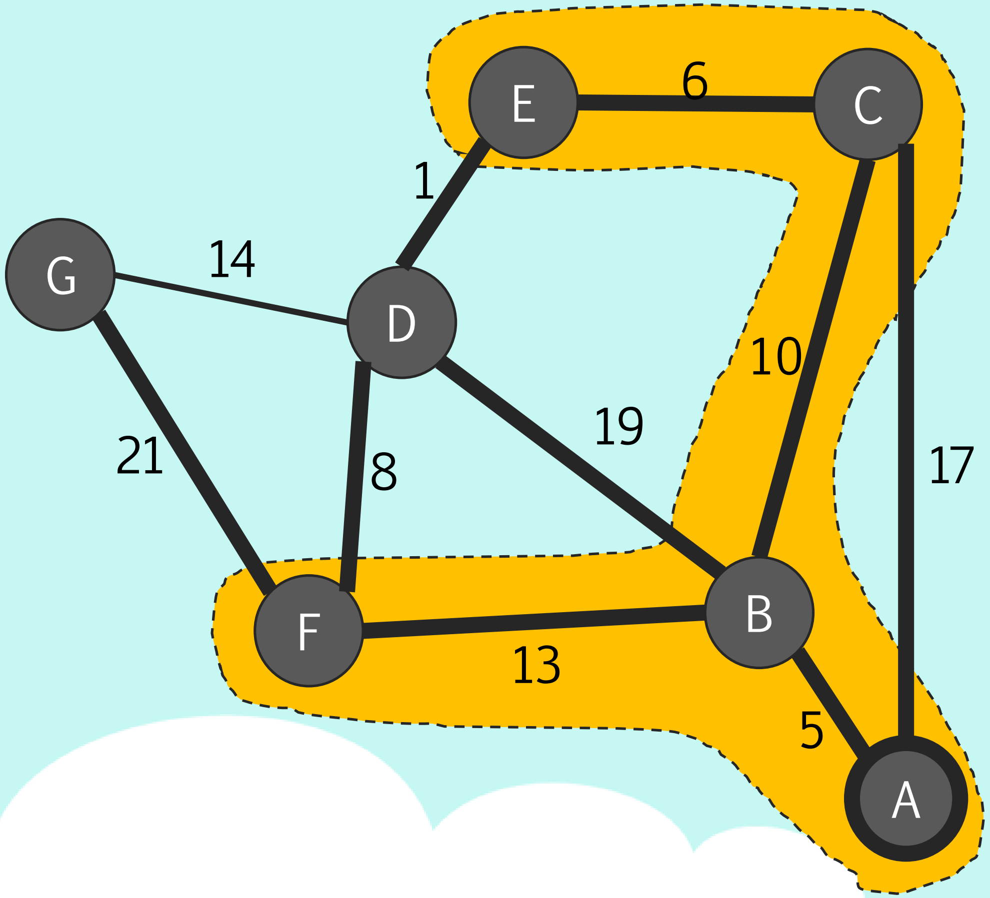

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)  // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (D, 22), (D, 24), (D, 26), (G, 39)

Curr: (E, 21)

Vertex	Dist
A	0
B	5
C	15
D	INF
E	21
F	18
G	INF

Shortest Path Strategy w/ Priority Queue



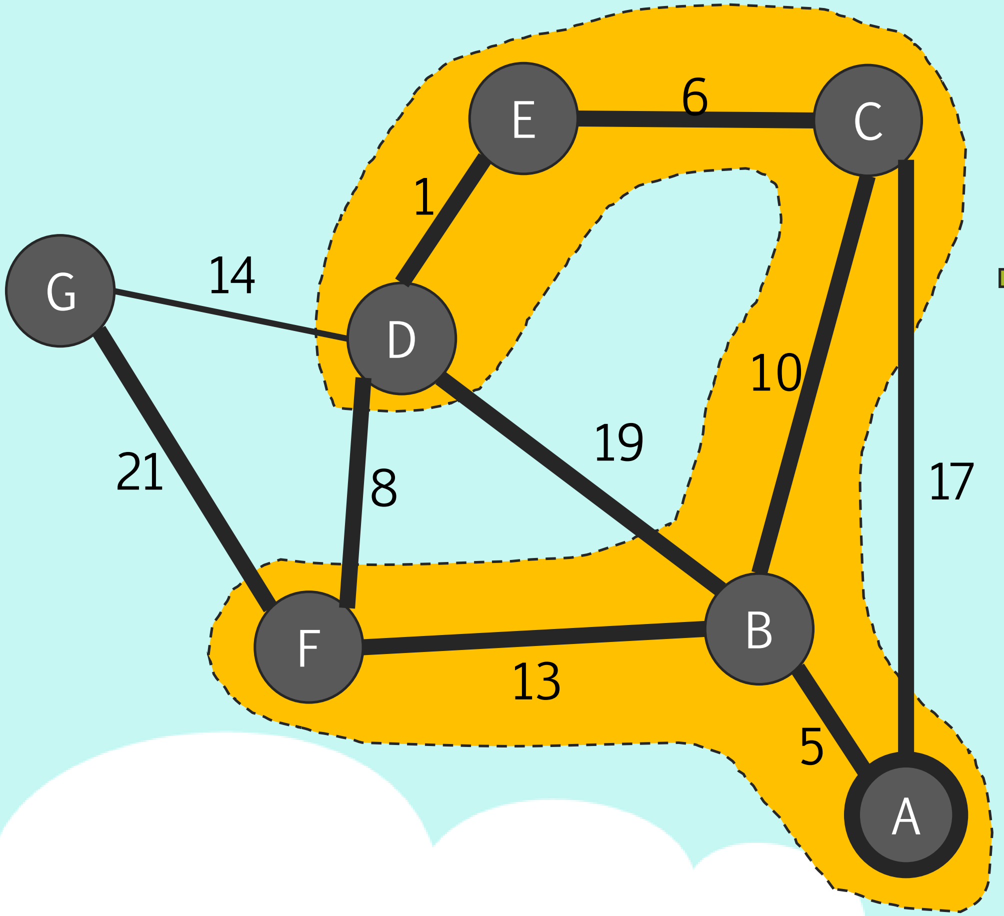

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (D, 24), (D, 26), (G, 39)

Curr: (D, 22)

Vertex	Dist
A	0
B	5
C	15
D	INF
E	21
F	18
G	INF

Shortest Path Strategy w/ Priority Queue



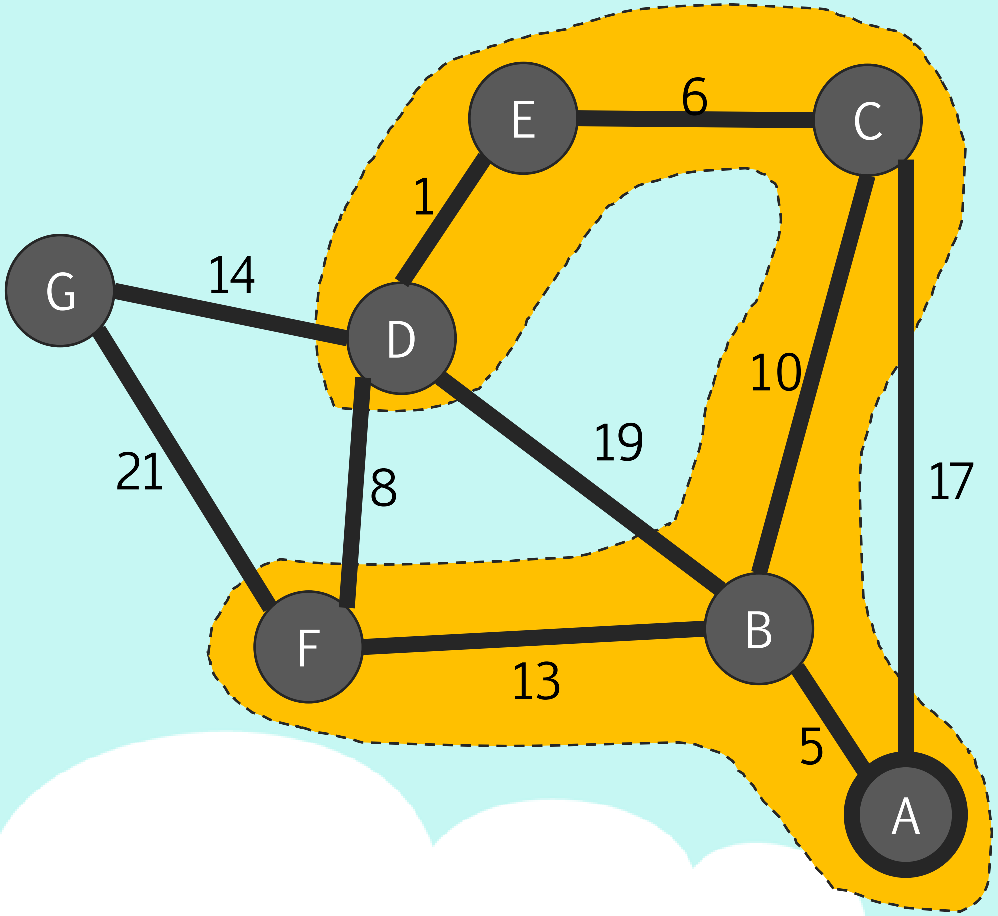

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (D, 24), (D, 26), (G, 39)

Curr: (D, 22)

Vertex	Dist
A	0
B	5
C	15
D	22
E	21
F	18
G	INF

Shortest Path Strategy w/ Priority Queue



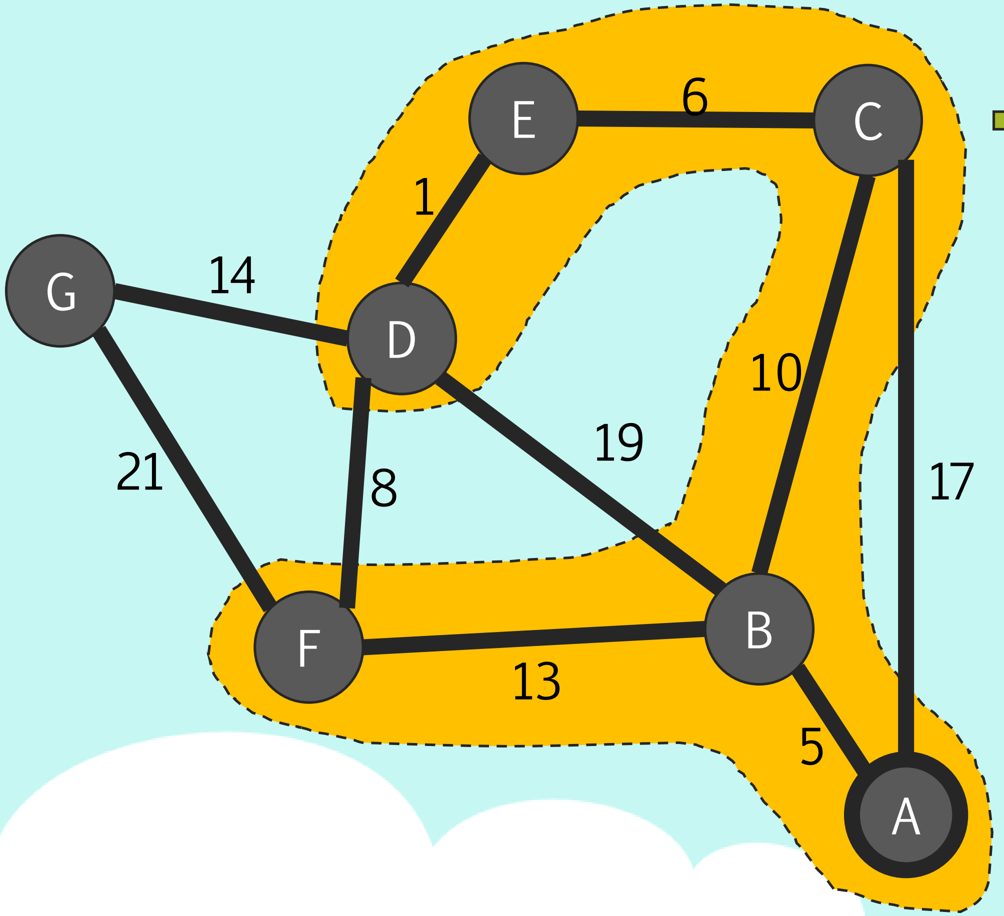

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (D, 24), (D, 26), (G, 36), (G, 39)

Curr: (D, 22)

Vertex	Dist
A	0
B	5
C	15
D	22
E	21
F	18
G	INF

Shortest Path Strategy w/ Priority Queue



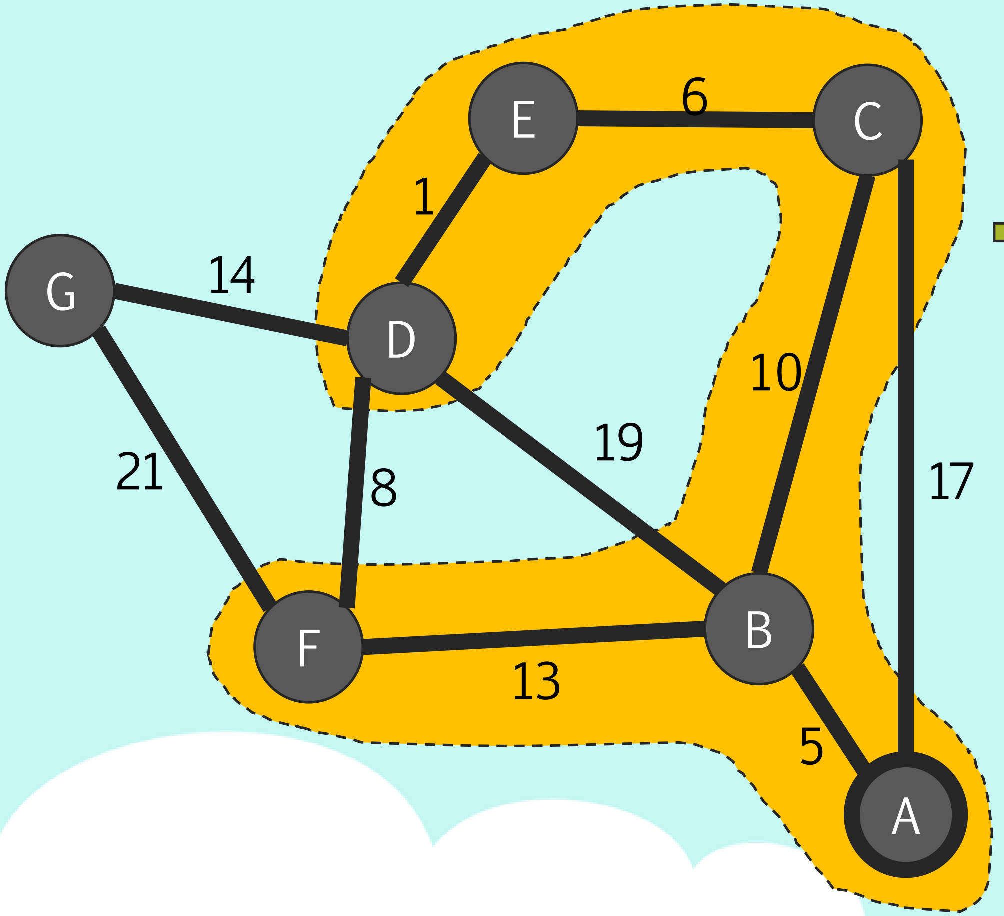

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (D, 26), (G, 36), (G, 39)

Curr: (D, 24)

Vertex	Dist
A	0
B	5
C	15
D	22
E	21
F	18
G	INF

Shortest Path Strategy w/ Priority Queue



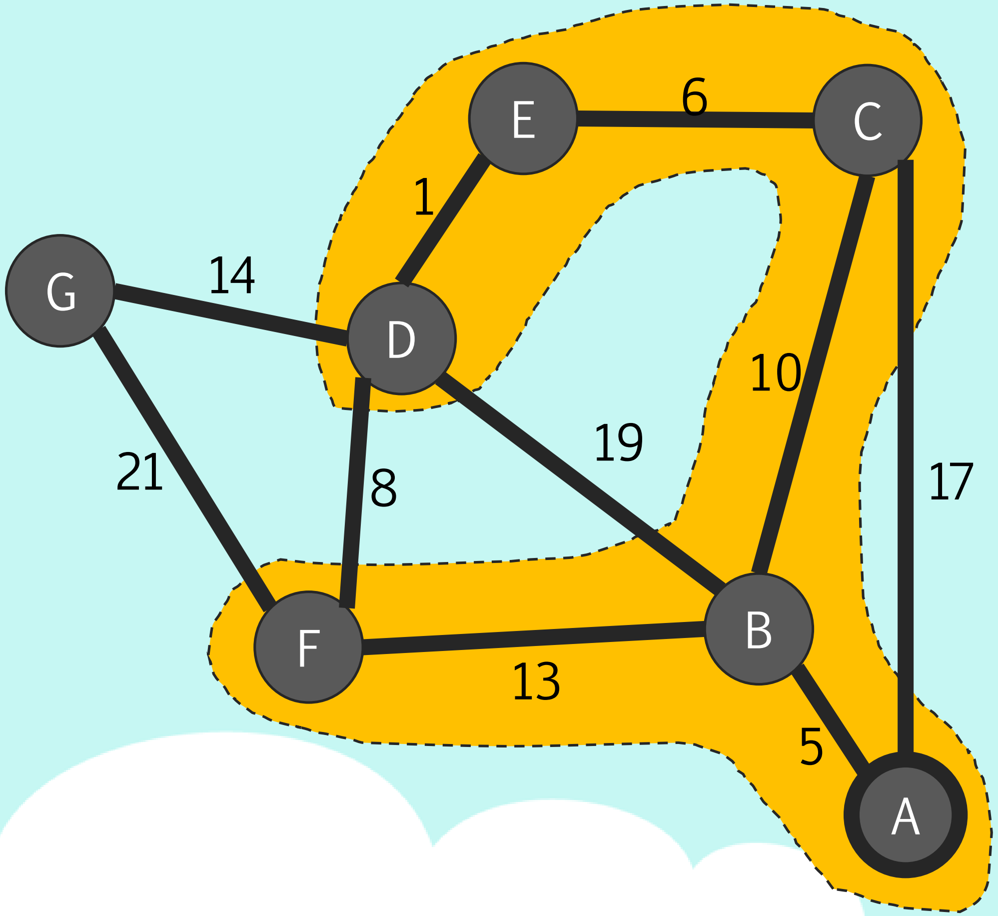

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (D, 26), (G, 36), (G, 39)

Curr: (D, 24)

Vertex	Dist
A	0
B	5
C	15
D	22
E	21
F	18
G	INF

Shortest Path Strategy w/ Priority Queue



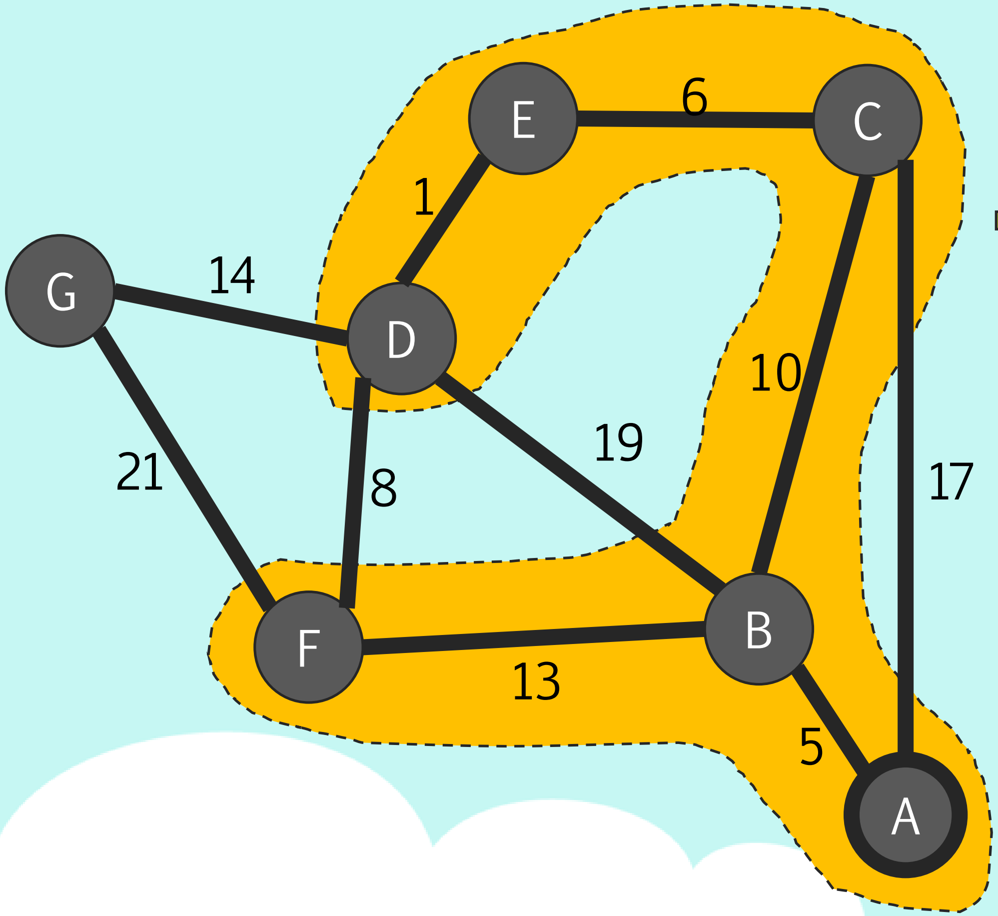

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (G, 36), (G, 39)

Curr: (D, 26)

Vertex	Dist
A	0
B	5
C	15
D	22
E	21
F	18
G	INF

Shortest Path Strategy w/ Priority Queue



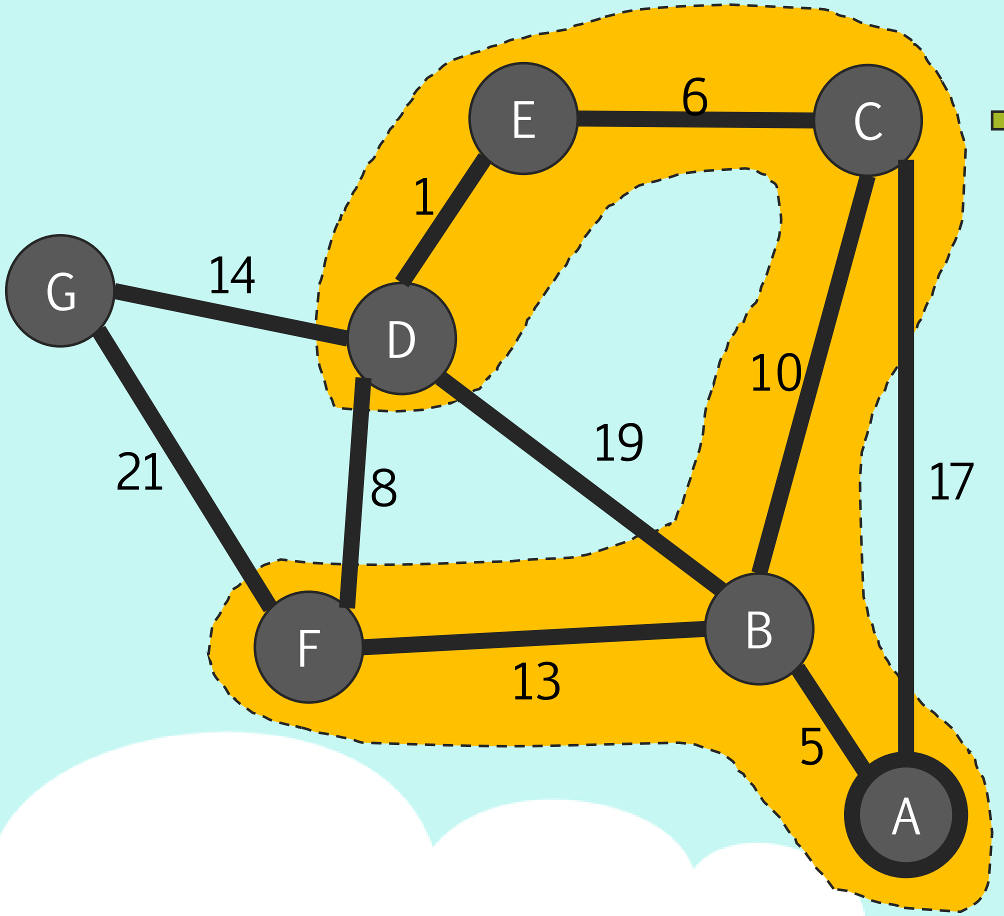

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (G, 36), (G, 39)

Curr: (D, 26)

Vertex	Dist
A	0
B	5
C	15
D	22
E	21
F	18
G	INF

Shortest Path Strategy w/ Priority Queue



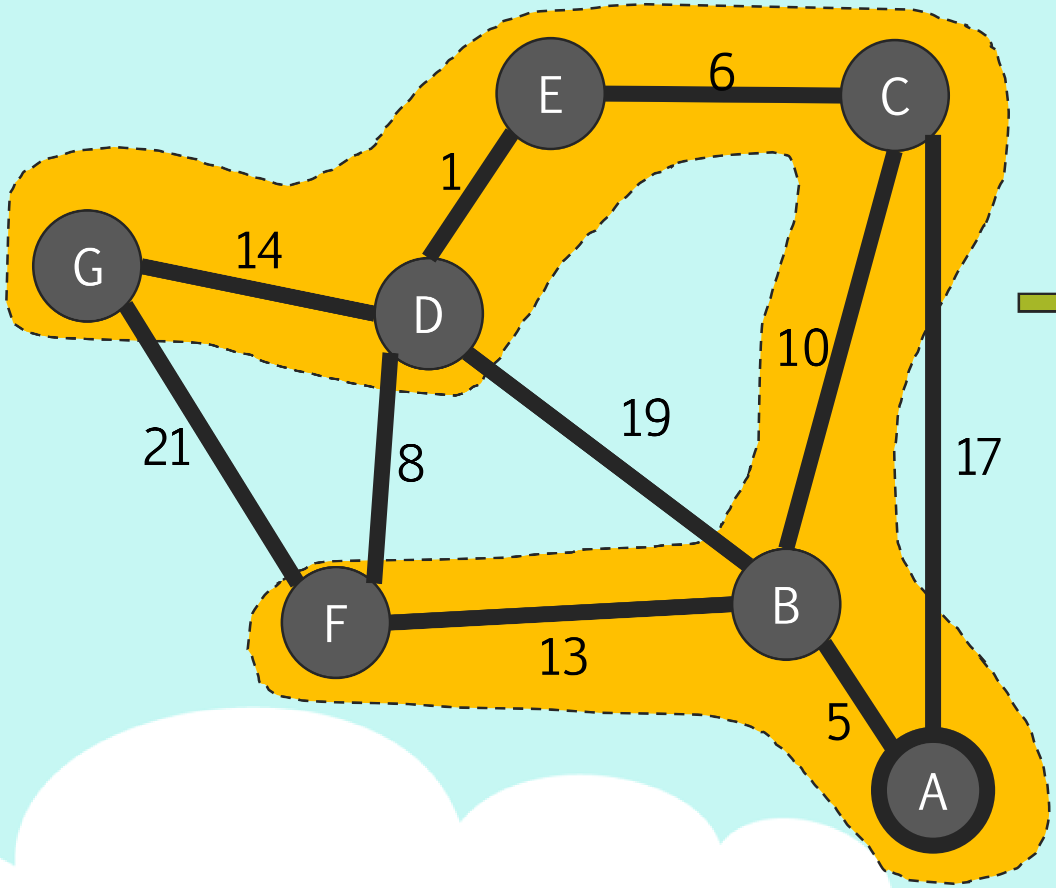

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (G, 39)

Curr: (G, 36)

Vertex	Dist
A	0
B	5
C	15
D	22
E	21
F	18
G	INF

Shortest Path Strategy w/ Priority Queue



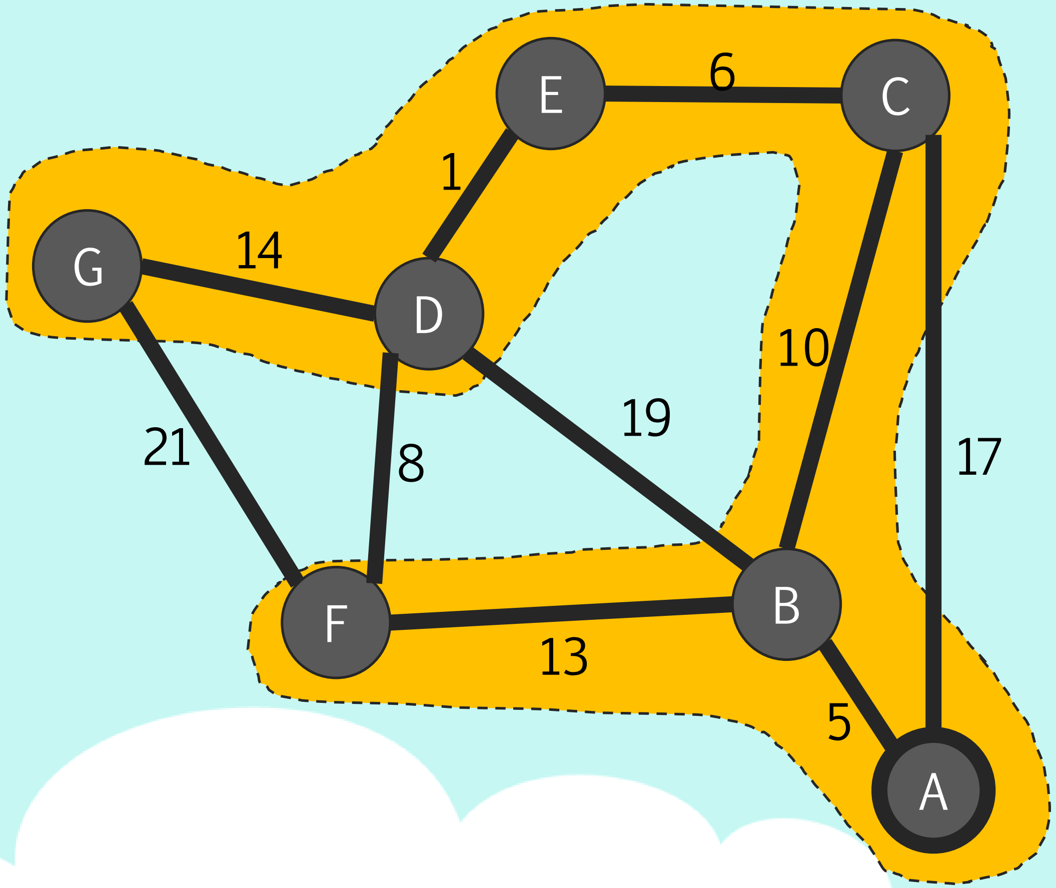

```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)   // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

PQ: (G, 39)

Curr: (G, 36)

Vertex	Dist
A	0
B	5
C	15
D	22
E	21
F	18
G	36

Shortest Path Strategy w/ Priority Queue




```
while (s not empty)
  curr = s.remove_min()
  if (paths[curr.vertex] is not INF)
    continue
  paths[curr.vertex] = curr.distance
  evaluate(curr)  // End if goal is reached
  for Vertex u in neighbors(curr.vertex)
    if (paths[u] is INF)
      s.add((u,
        paths[curr.vertex] + edge(curr.vertex, u)))
```

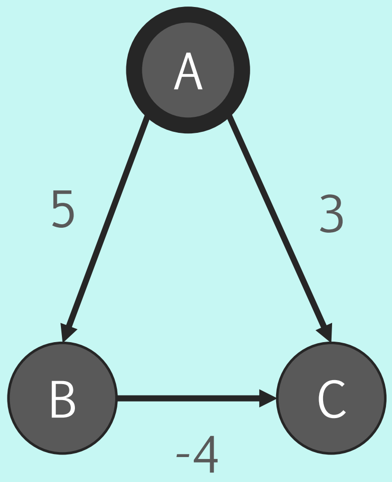
Vertex	Dist
A	0
B	5
C	15
D	22
E	21
F	18
G	36

Dijkstra's and Negative Edge Weights

- Dijkstra's is a greedy algorithm. When it calculates a new distance to a vertex, Dijkstra's assumes that distance is the shortest distance to that vertex.

- When we introduce negative edge weights, this greedy heuristic does not hold. An encounter with a negative edge weight can provide us a shorter distance to a vertex than previously calculated. However, Dijkstra does not revisit these calculated distances.

- In this graph, Dijkstra would calculate the shortest distance to C as 3. Running more iterations will reveal that the shortest distance is actually 1, but Dijkstra will keep C : 3.



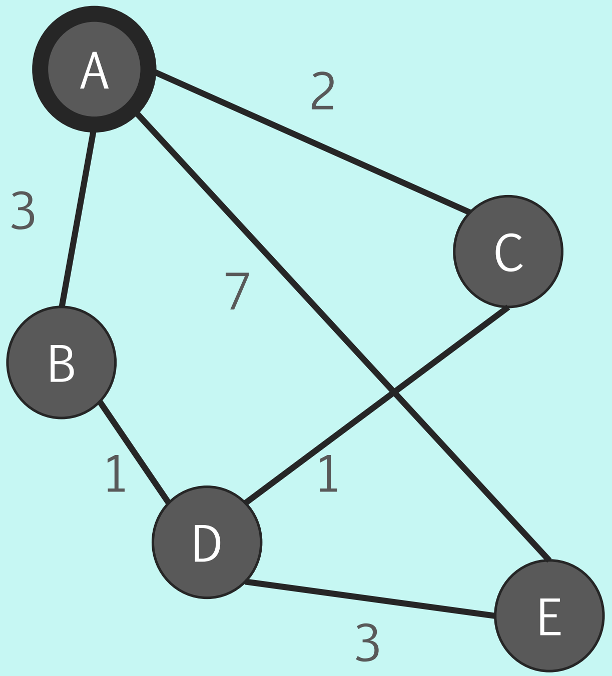
Dijkstra Analysis

- Dijkstra runs in $O((|V| + |E|) \log(|V|))$. If we use a min-heap for our priority queue, calling PQ.remove_min() will yield $O(\log(|V|))$.
  - If we visit each vertex and edge at most once, we will call PQ.remove_min() $O(|V| + |E|)$ times

Practice

- For the graph:
  - Find the shortest path from A to all vertices

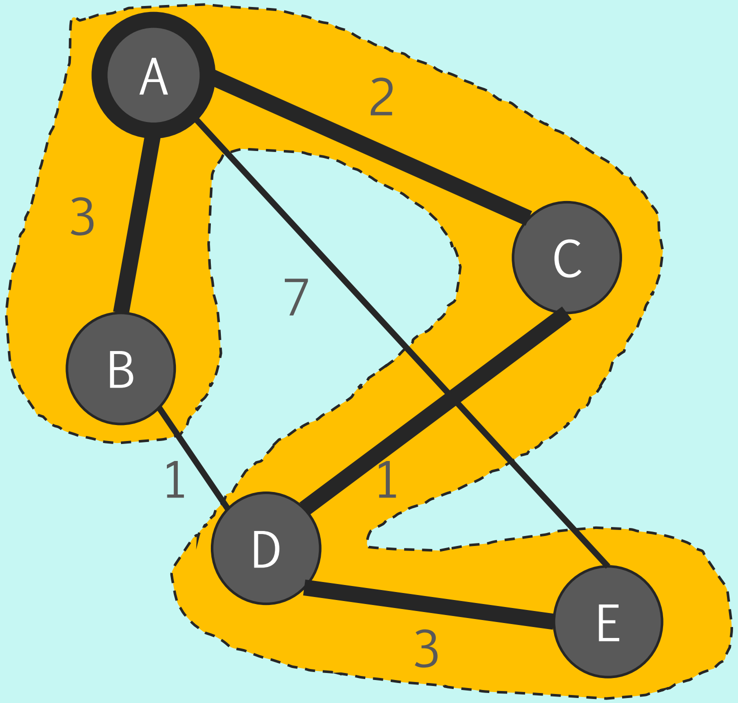
Vertex	Dist
A	INF
B	INF
C	INF
D	INF
E	INF



Practice

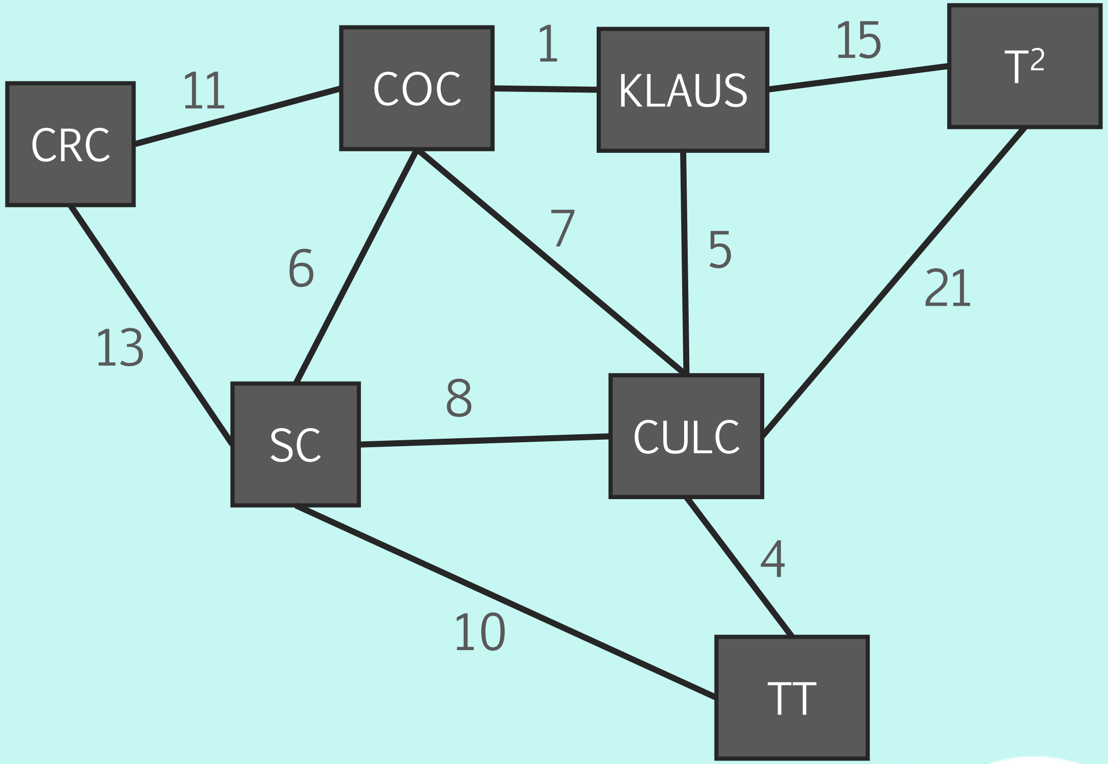
- For the graph:
  - Find the shortest path from A to all vertices

Vertex	Dist
A	0
B	3
C	2
D	3
E	6



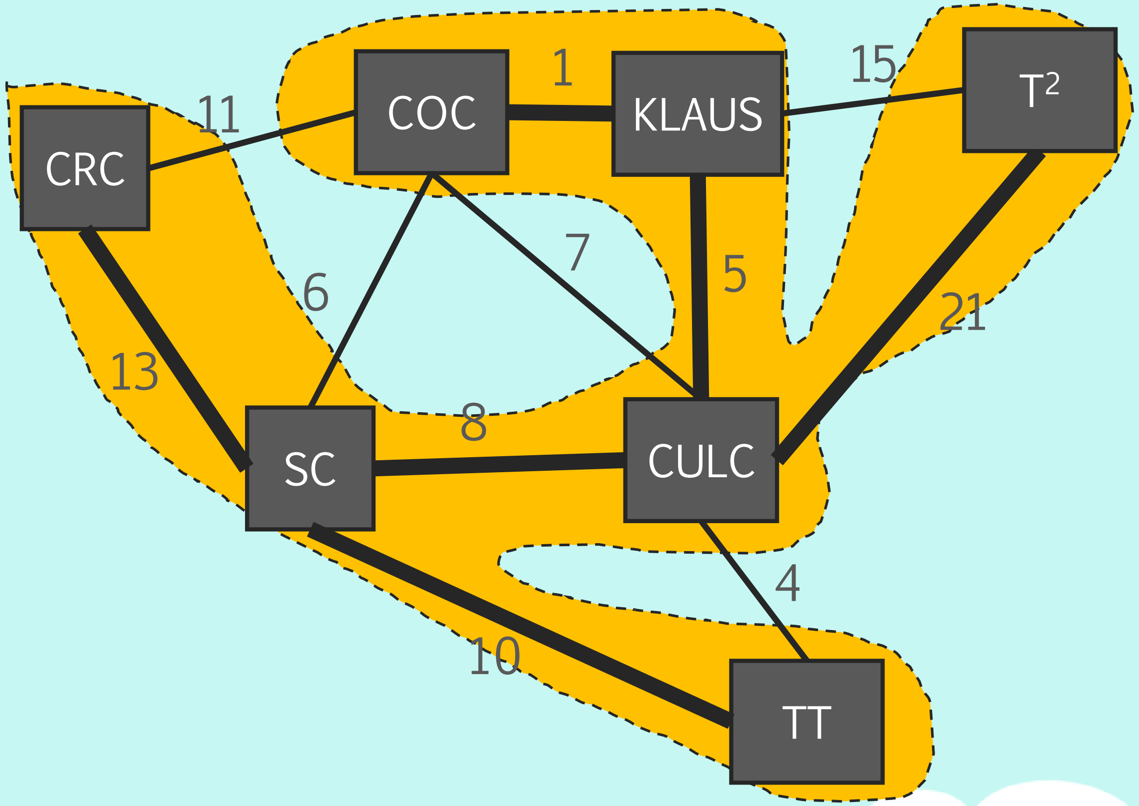
Connecting the Campus

- Our campus to the right has the following buildings and sidewalks.
- Let's say Bud Peterson implemented budget cuts to side walks, so we need to pick sidewalks to keep.
- We want the least set of sidewalks that will still connect the campus.
  - Every building has a path to every other building.



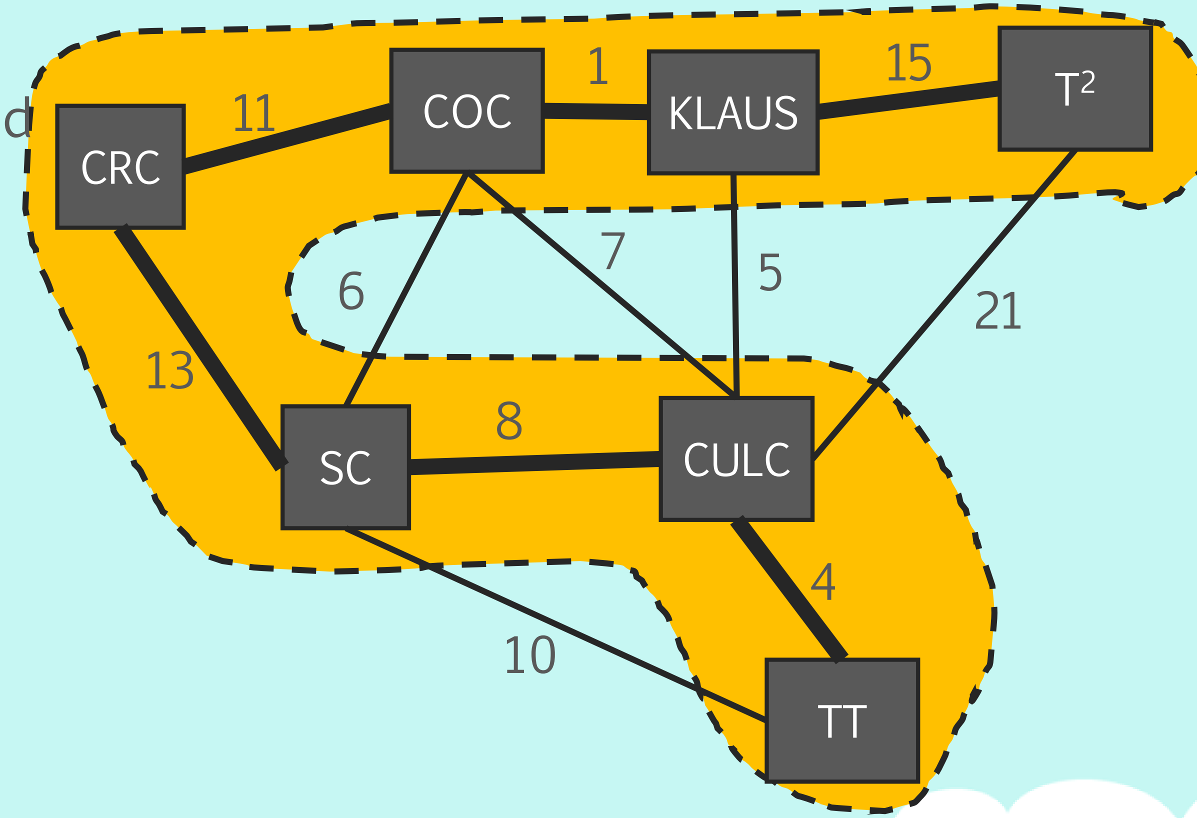
Connecting the Campus

- Our campus to the right has the following buildings and sidewalks.
- Let's say Bud Peterson implemented budget cuts to side walks, so we need to pick sidewalks to keep.
- We want the least set of sidewalks that will still connect the campus.
  - Every building has a path to every other building.



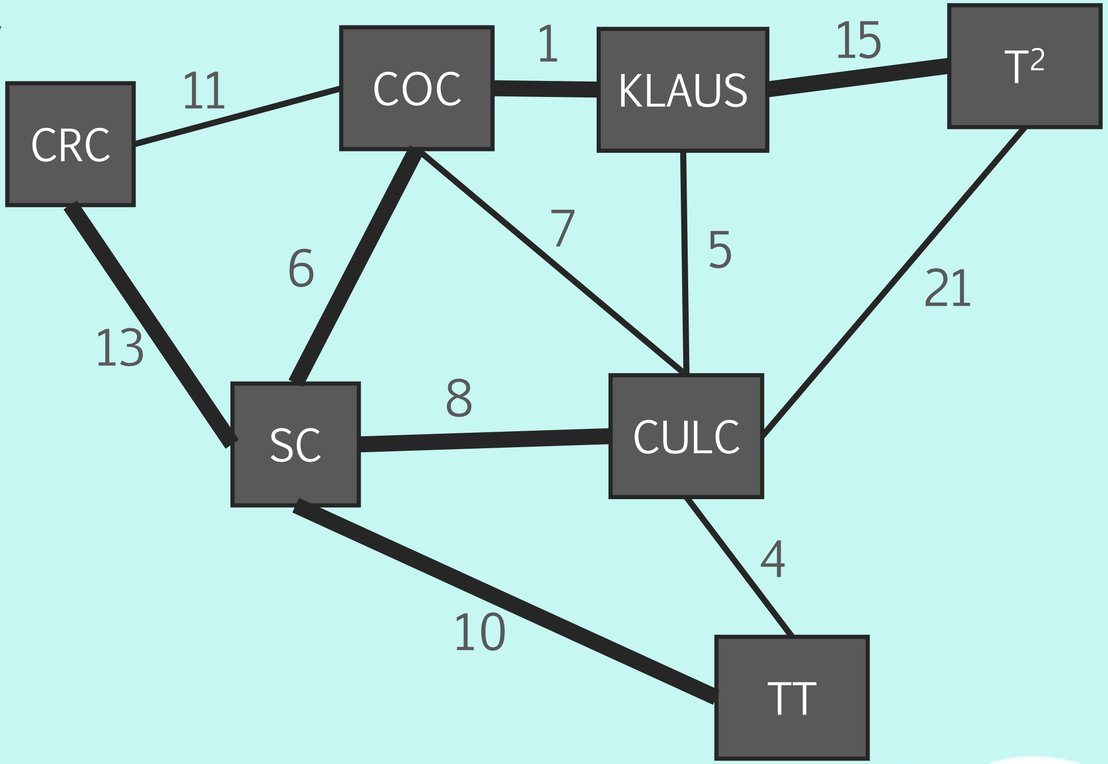
Connecting the Campus

- Our campus to the right has the following buildings and sidewalks.
- Let's say Bud Peterson implemented budget cuts to side walks, so we need to pick sidewalks to keep.
- We want the least set of sidewalks that will still connect the campus.
  - Every building has a path to every other building.



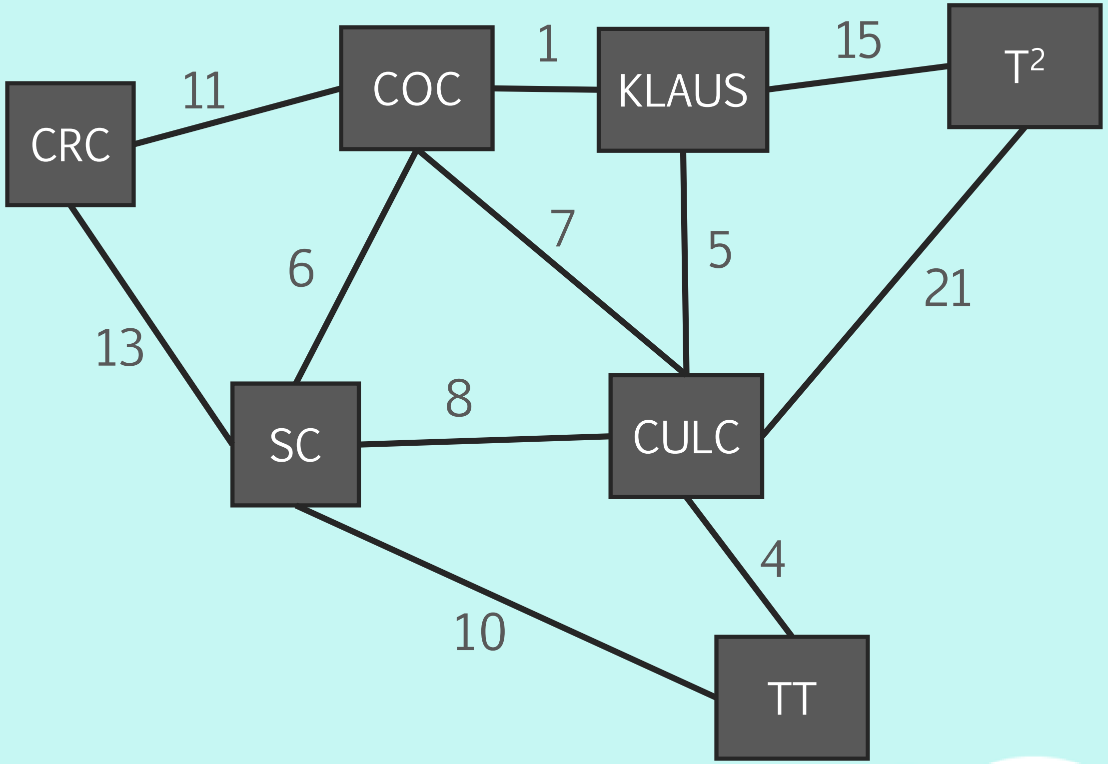
Spanning Tree

- In an undirected graph, a spanning tree is the set of edges that connect every vertex with the least number of edges.
  - There for, the number of edges in a spanning tree is equal to |V|-1.
- Spanning tree's cannot have cycles.
  - If there exists a cycle in a spanning tree, then we can remove one edge in the cycle and still maintain connectivity.



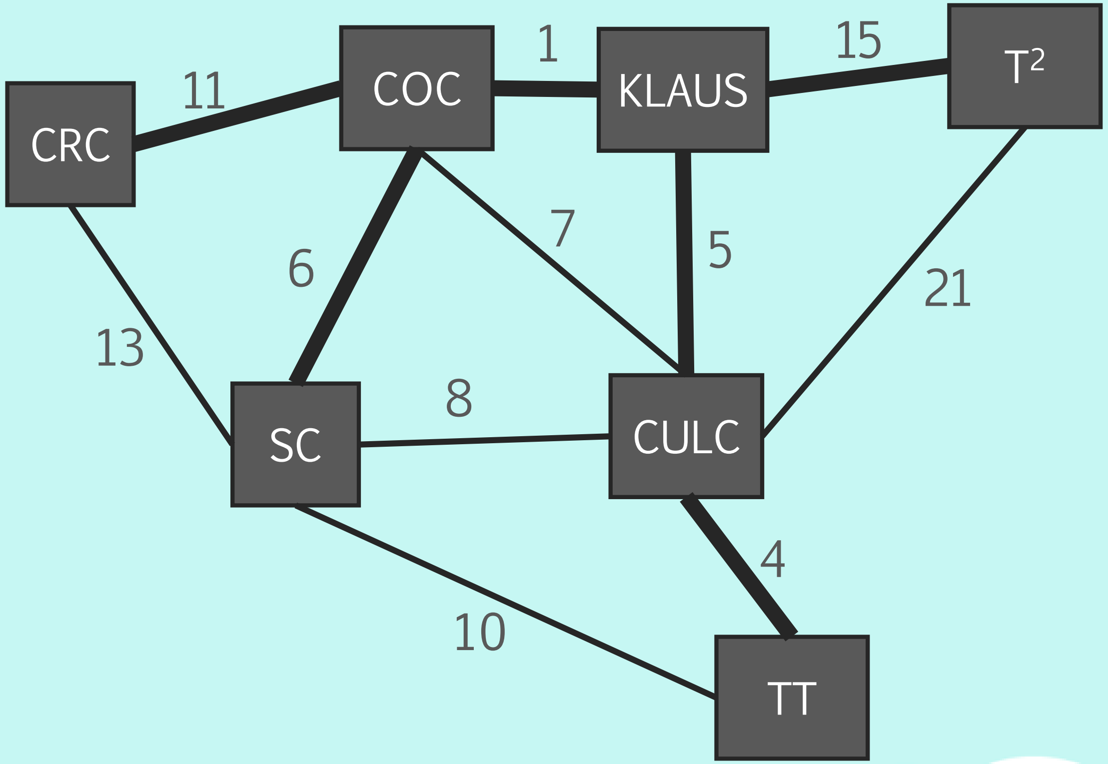
Connecting the Campus (minimum)

- In this campus there are multiple spanning trees, but with budget cuts, we want to keep sidewalks with the least amount of distance.
  - (distance = $$)
- In this graph, what is the spanning tree of sidewalks with the least cost?



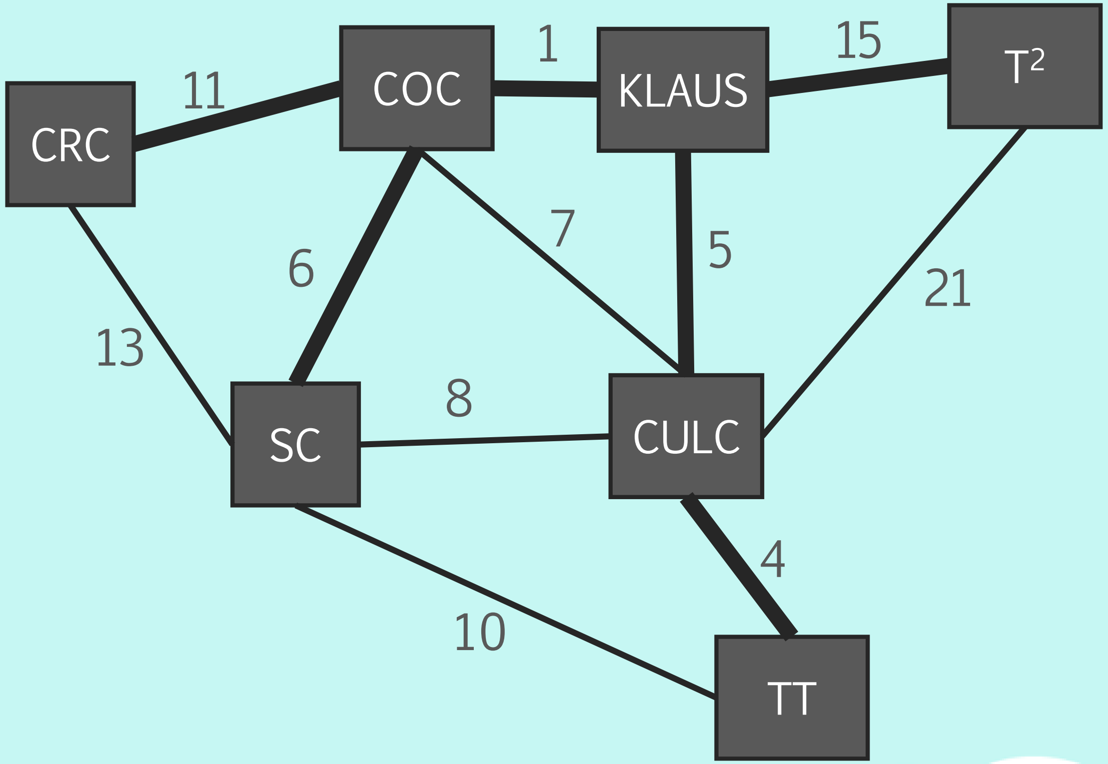
Connecting the Campus (minimum)

- In this campus there are multiple spanning trees, but with budget cuts, we want to keep sidewalks with the least amount of distance.
  - (distance = $$)
- In this graph, what is the spanning tree of sidewalks with the least cost?



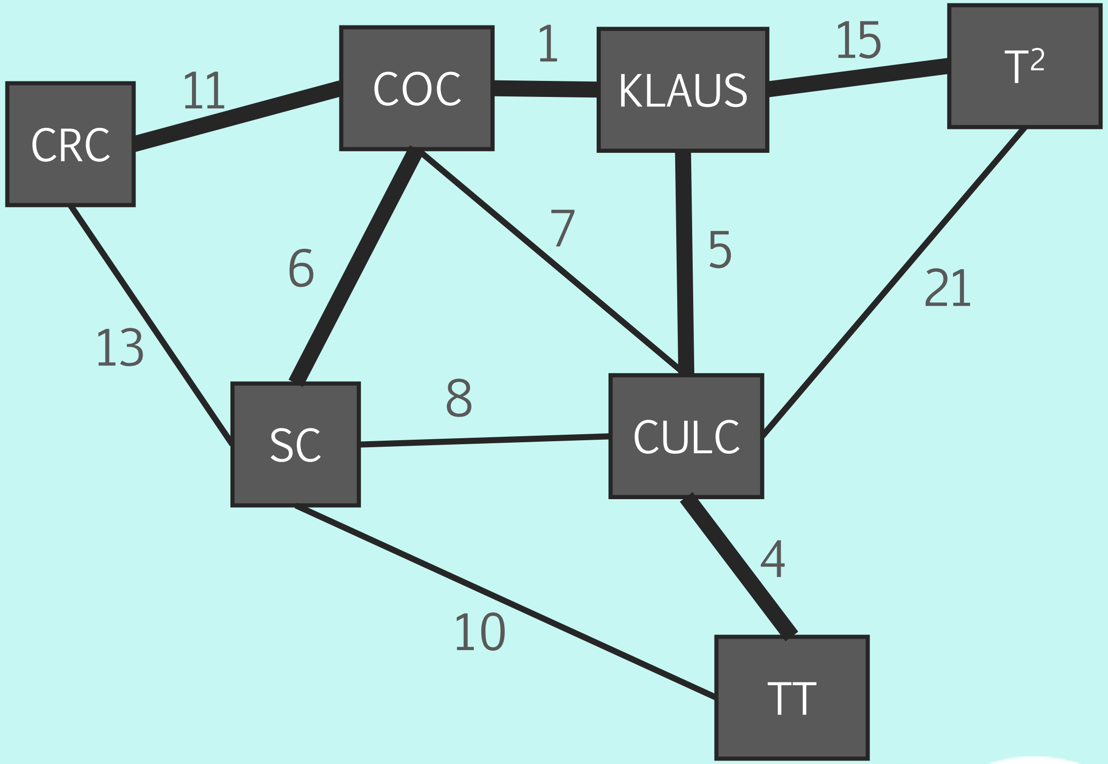
Minimum Spanning Tree

- The Minimum spanning tree of a graph is a spanning tree of a weighted graph with minimum total edge weight.
  - This MST has a edge weight of 42.
- MST's are useful for:
  - Transportation networks (subways)
  - Network Cabling



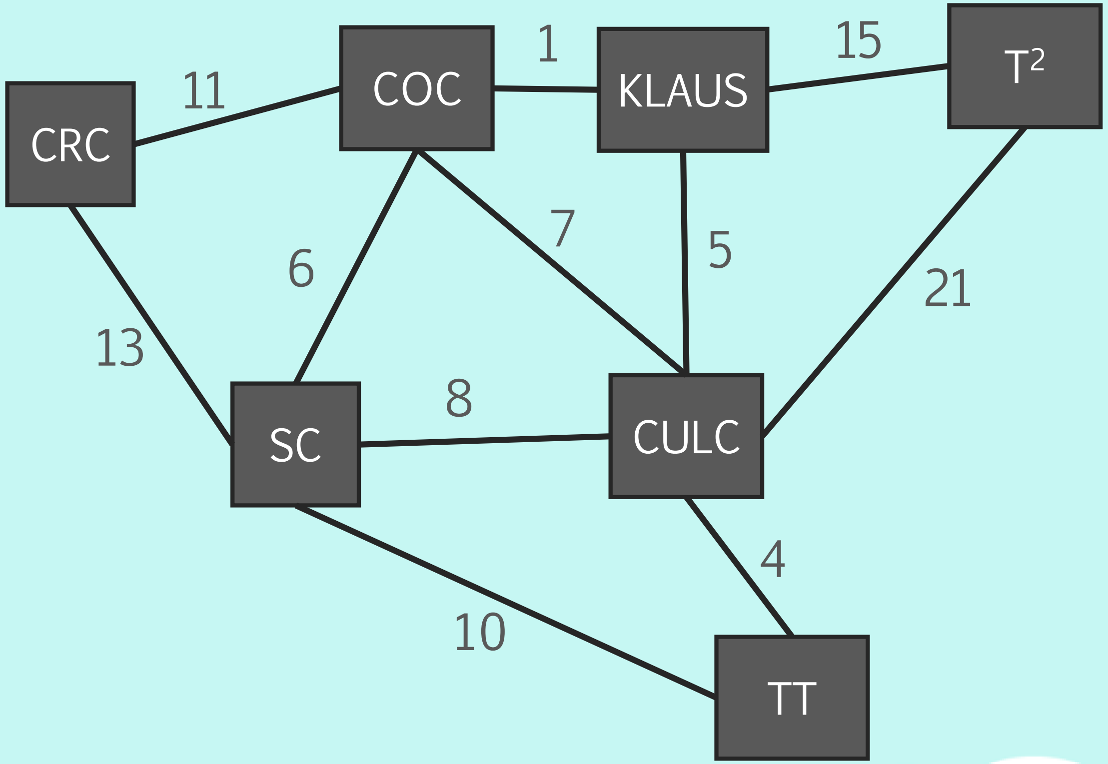
Minimum Spanning Tree

- The Minimum spanning tree of a graph is a spanning tree of a weighted graph with minimum total edge weight.
  - This MST has a edge weight of 42.
- MST's are useful for:
  - Transportation networks (subways)
  - Network Cabling
- How did you find the MST of this graph?

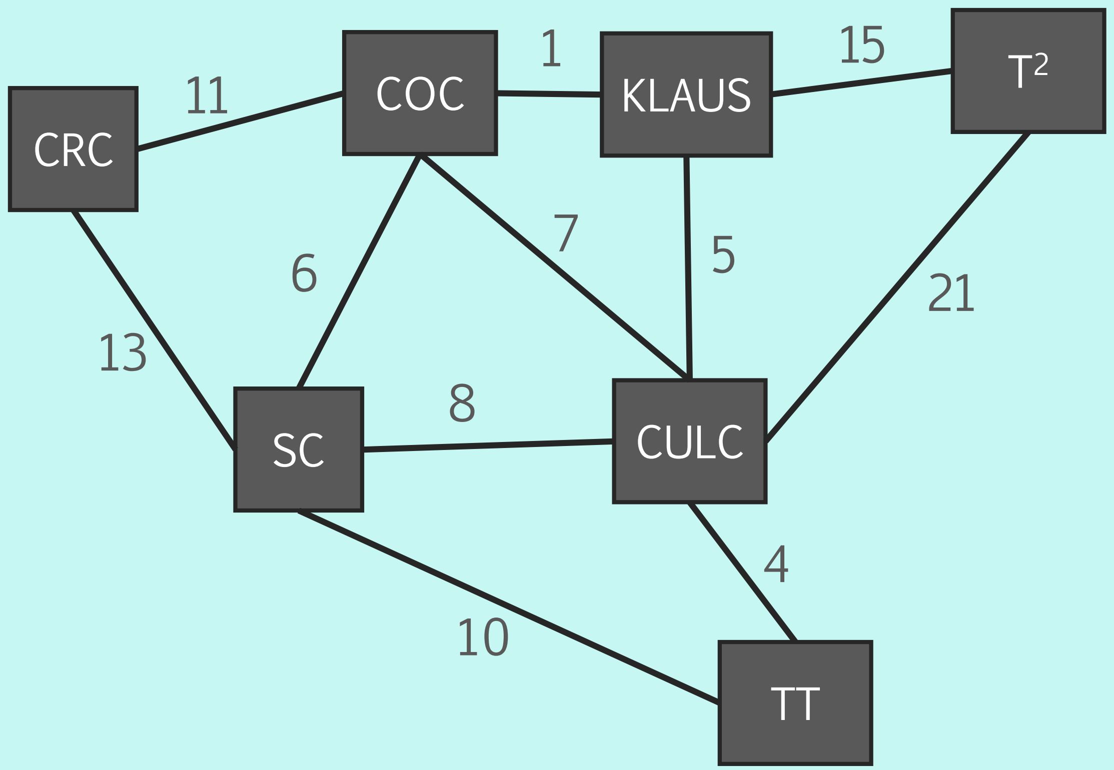


Kruskal’s Algorithm

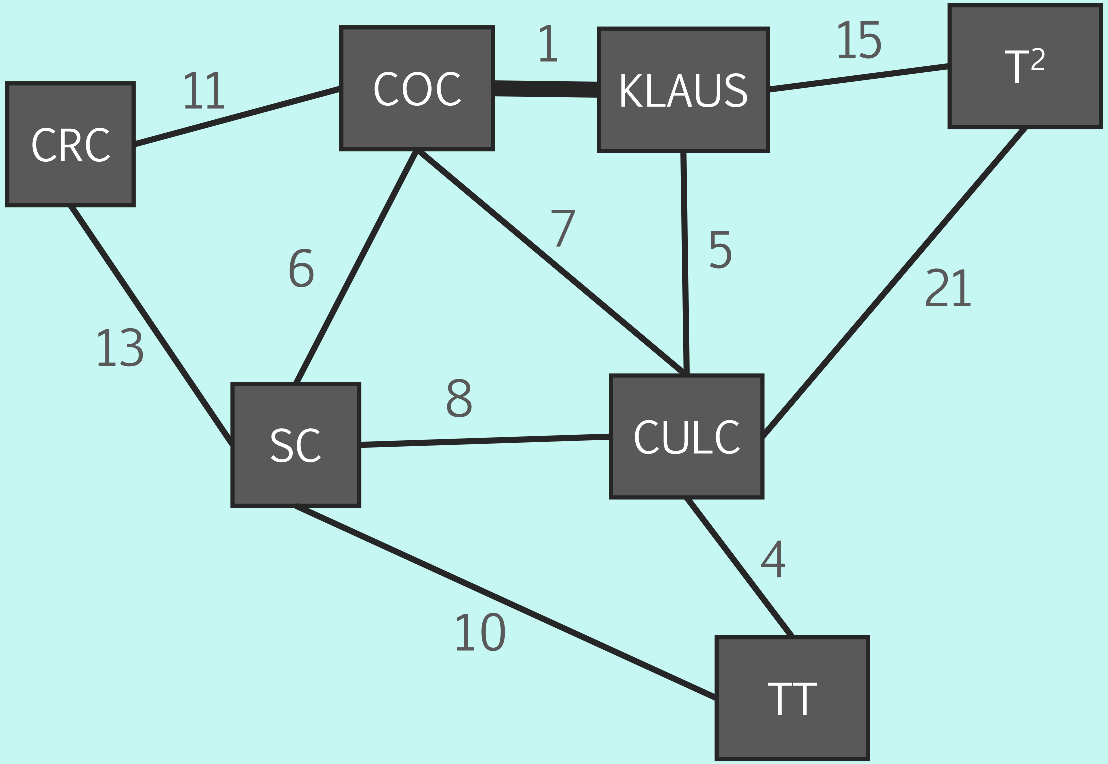
- Finds the MST of a weighted graph.
- To Perform by hand and diagram
  1. Start with the smallest edge and add it to your spanning tree.
  2. If the edge creates a cycle within your spanning tree, skip it.
  3. Repeat this until all of your vertices are connected.



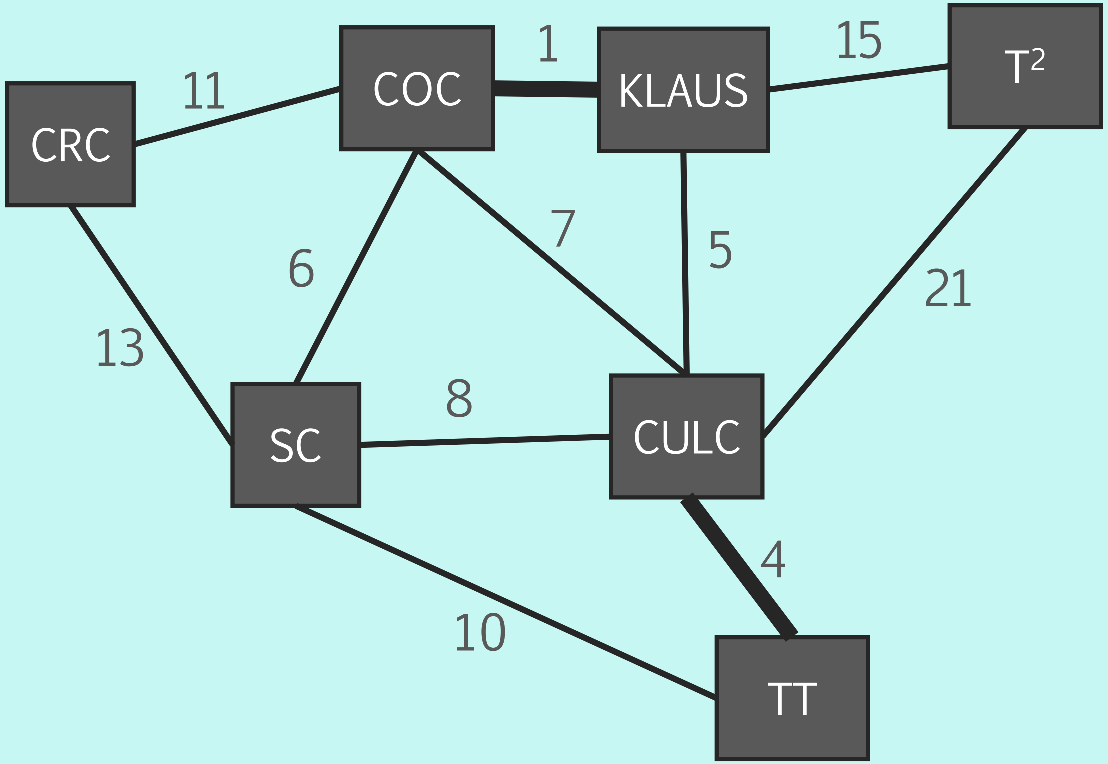
Kruskal's Algorithm Example



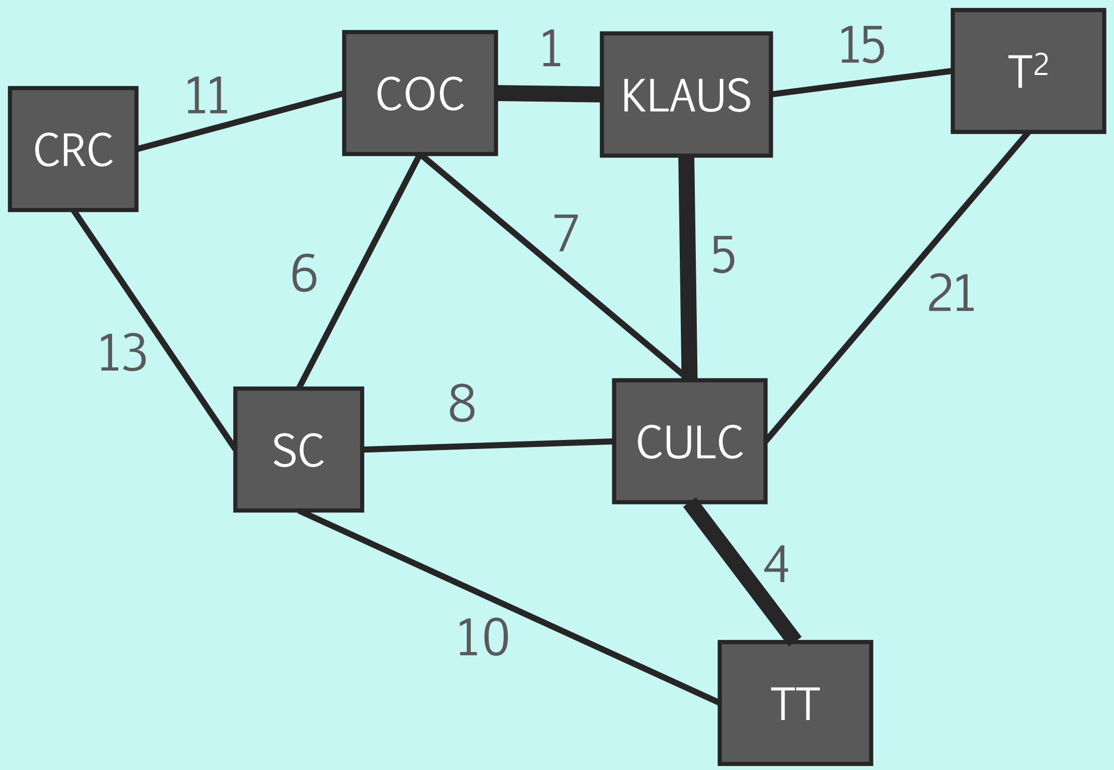
Kruskal's Algorithm Example



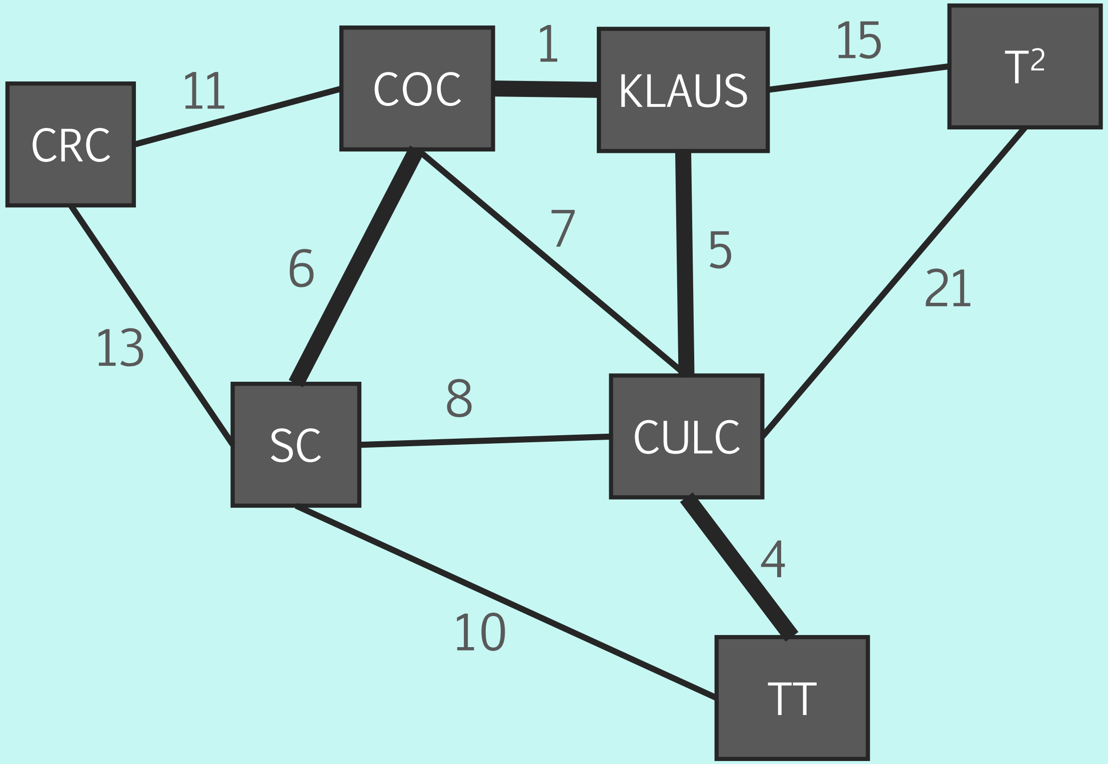
Kruskal's Algorithm Example



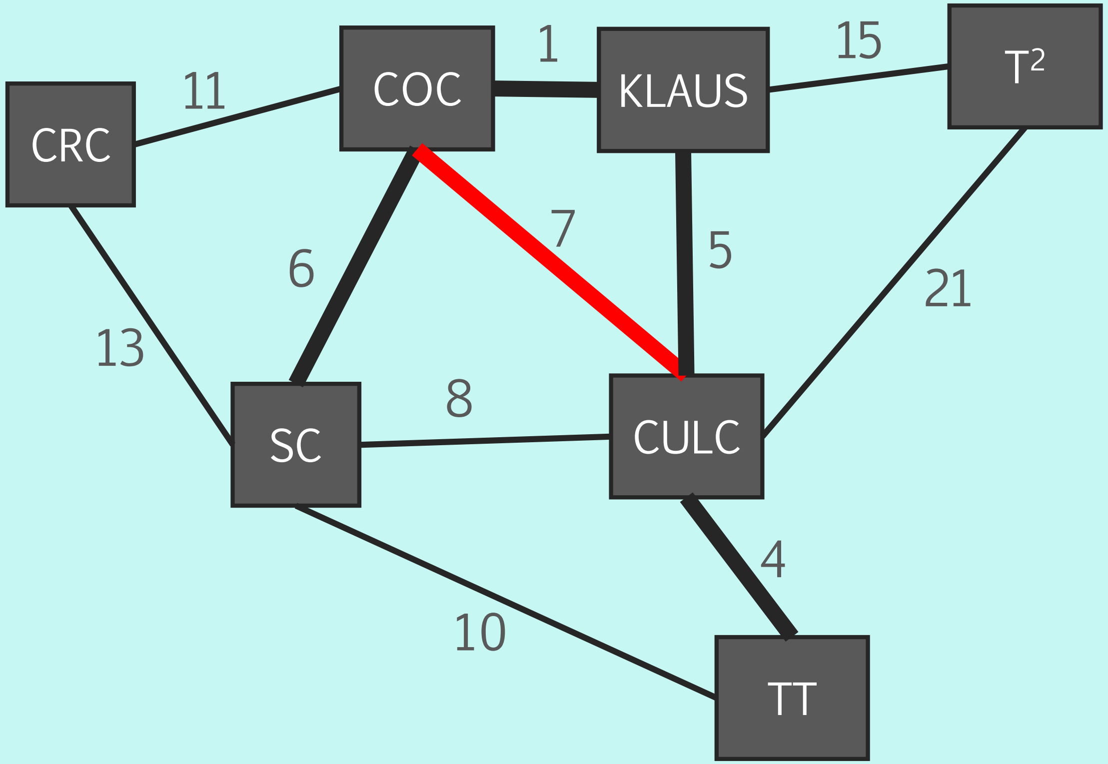
Kruskal's Algorithm Example



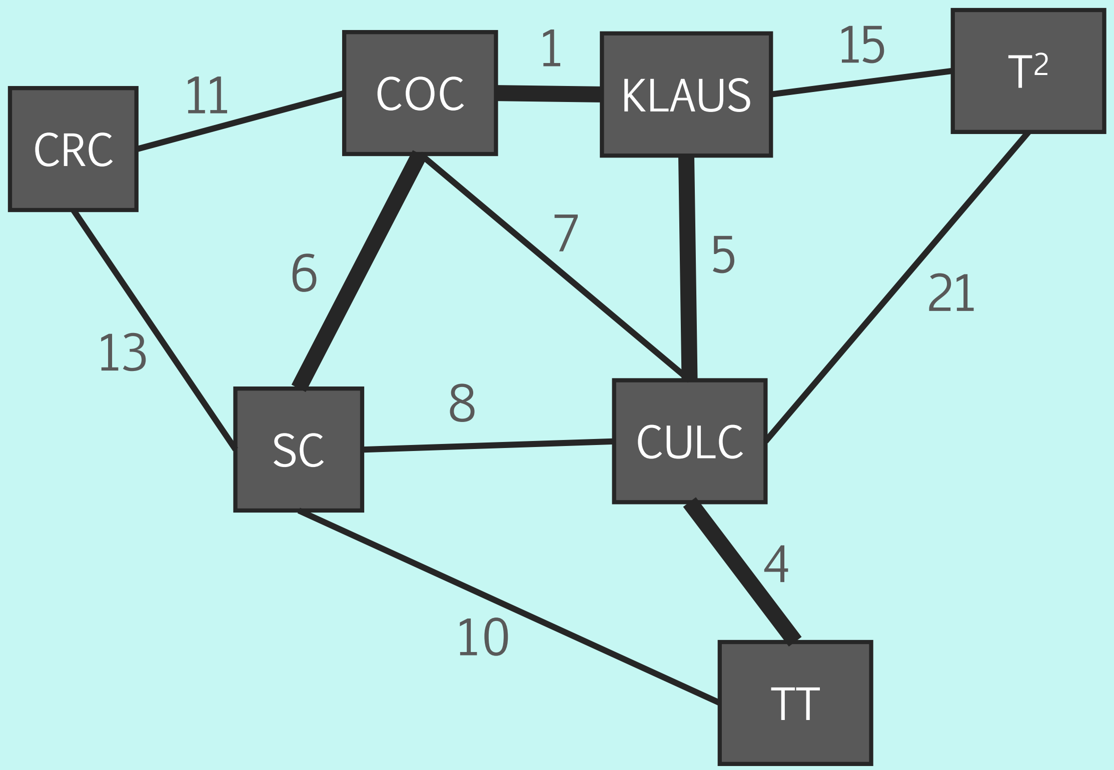
Kruskal's Algorithm Example



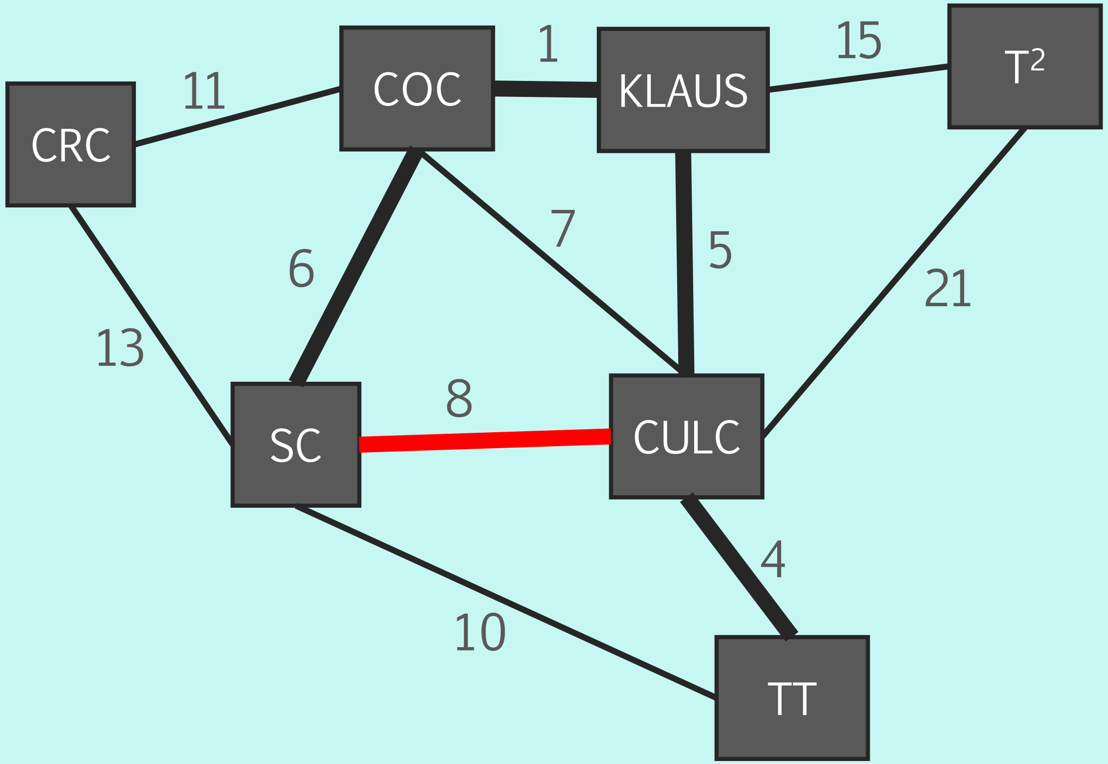
Kruskal's Algorithm Example



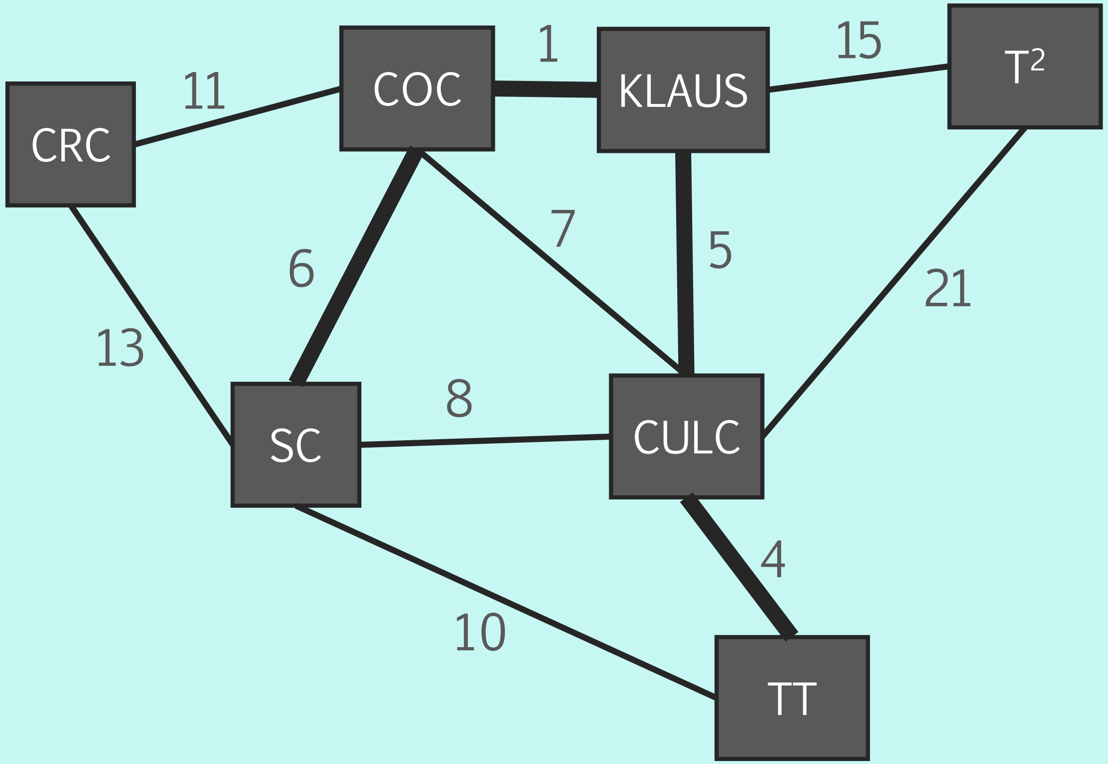
Kruskal’s Algorithm Example



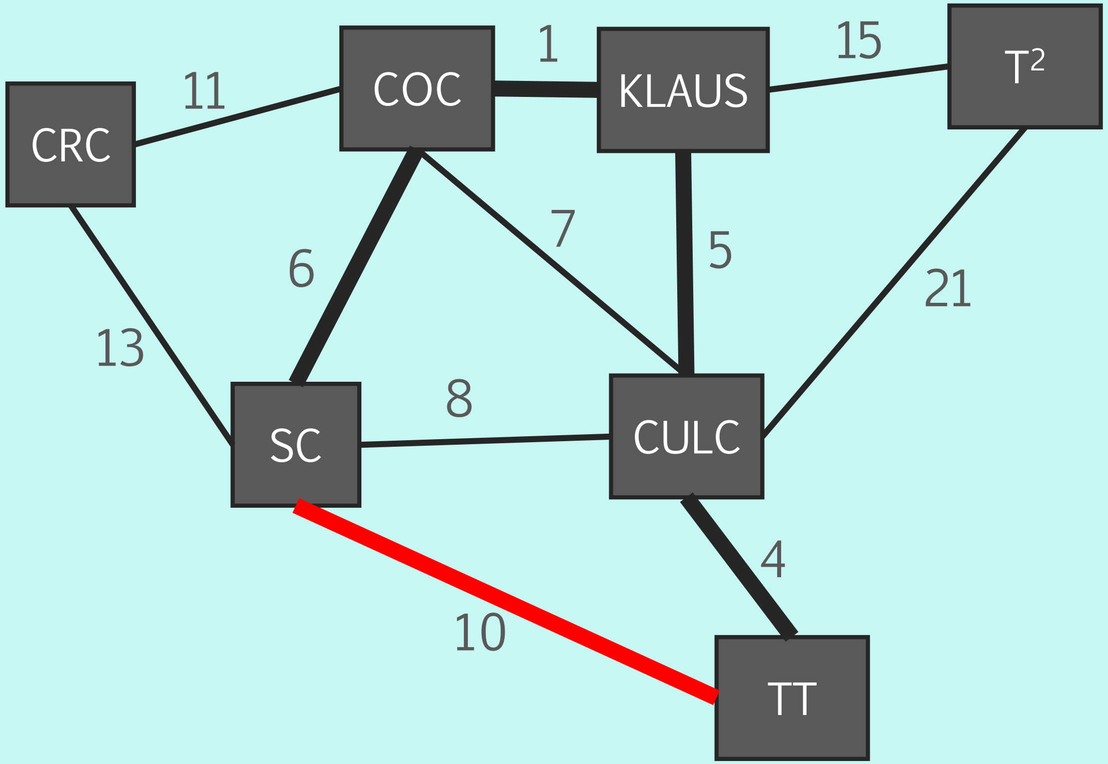
Kruskal's Algorithm Example



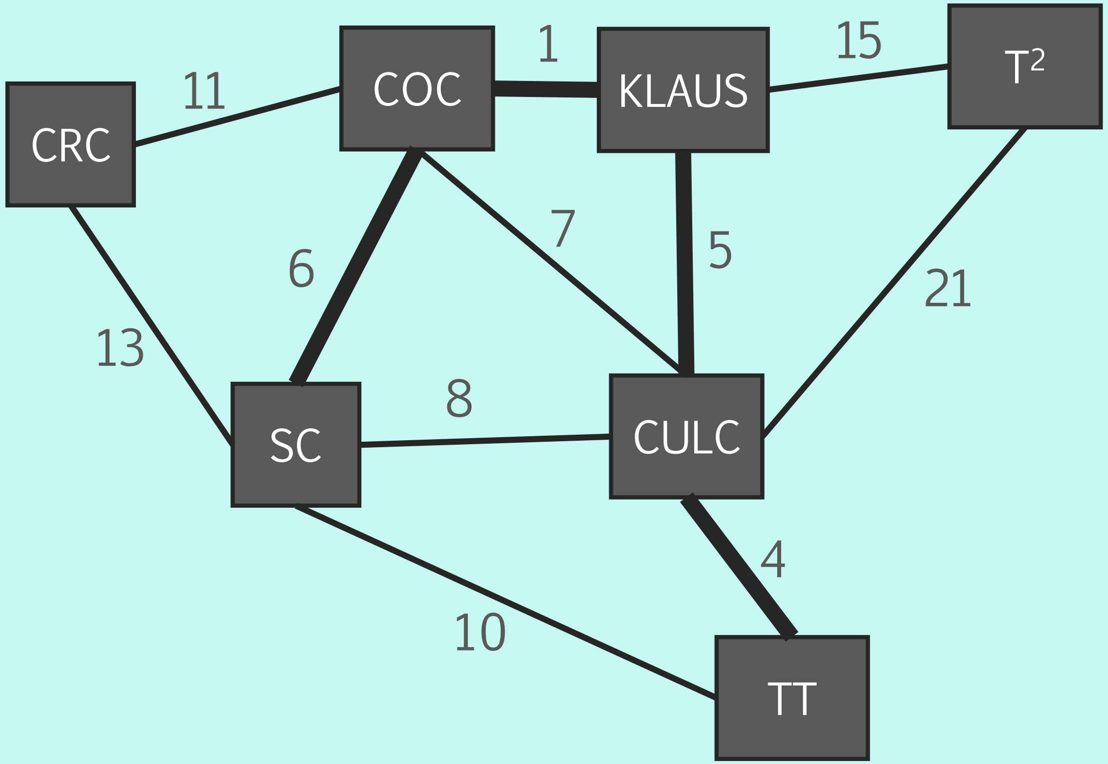
Kruskal's Algorithm Example



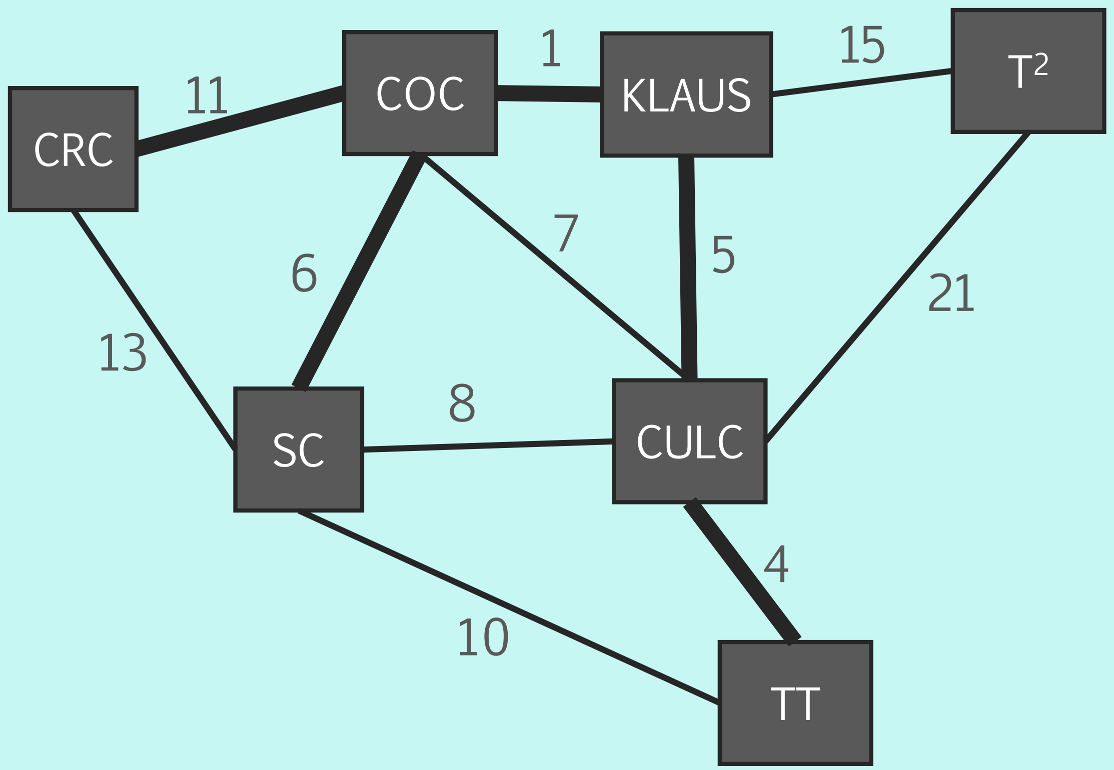
Kruskal's Algorithm Example



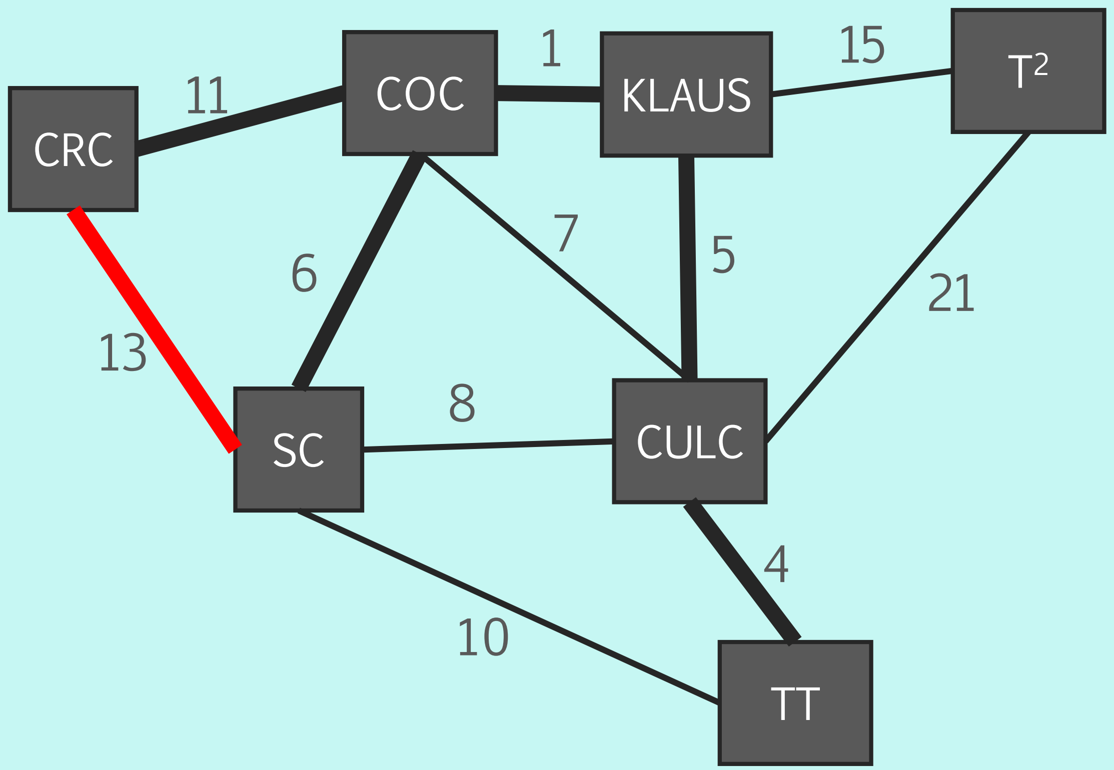
Kruskal's Algorithm Example



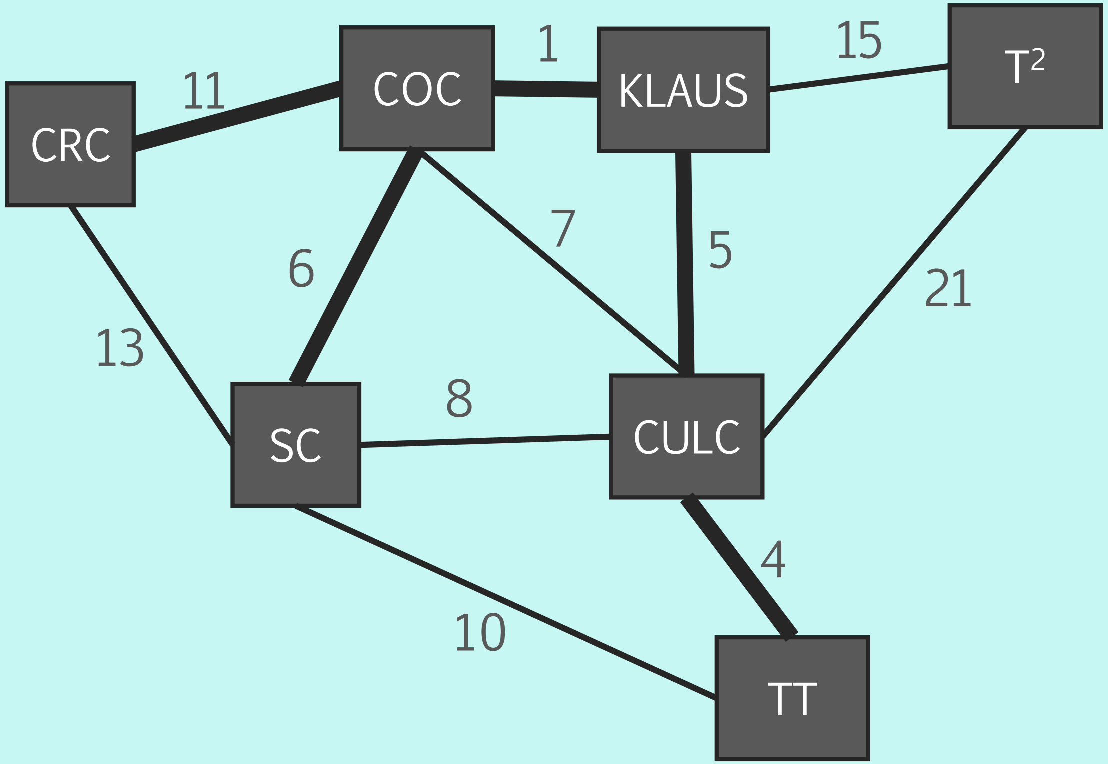
Kruskal's Algorithm Example



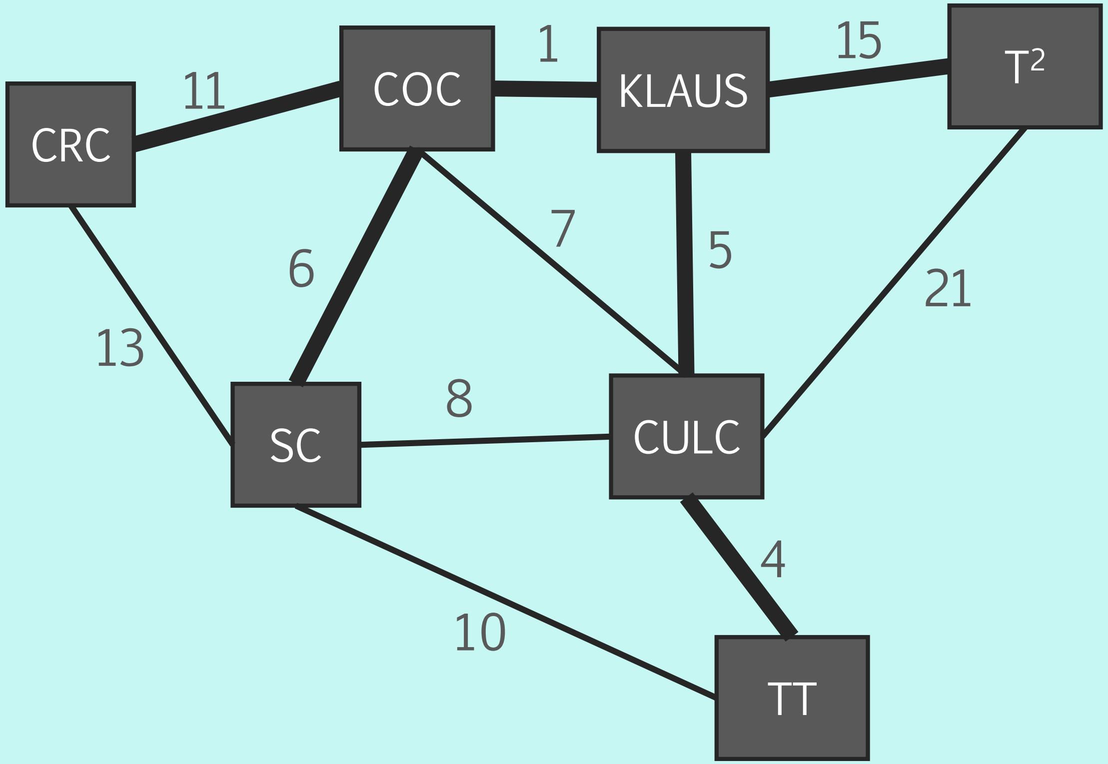
Kruskal's Algorithm Example



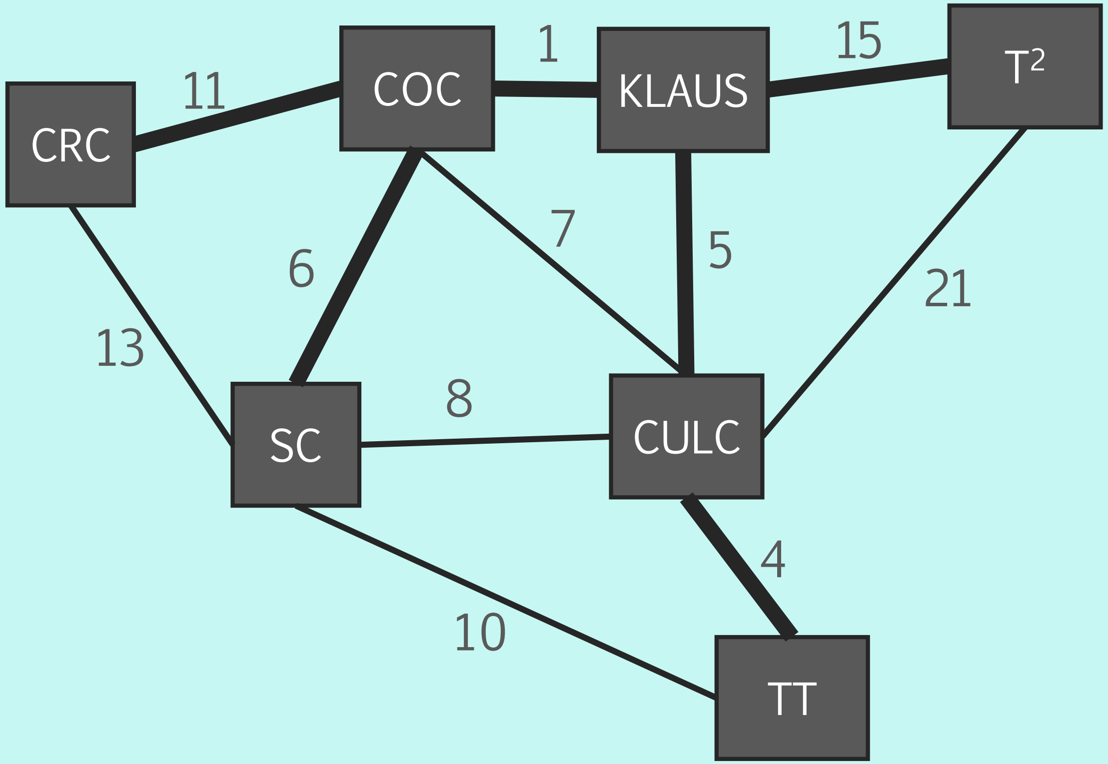
Kruskal's Algorithm Example



Kruskal's Algorithm Example

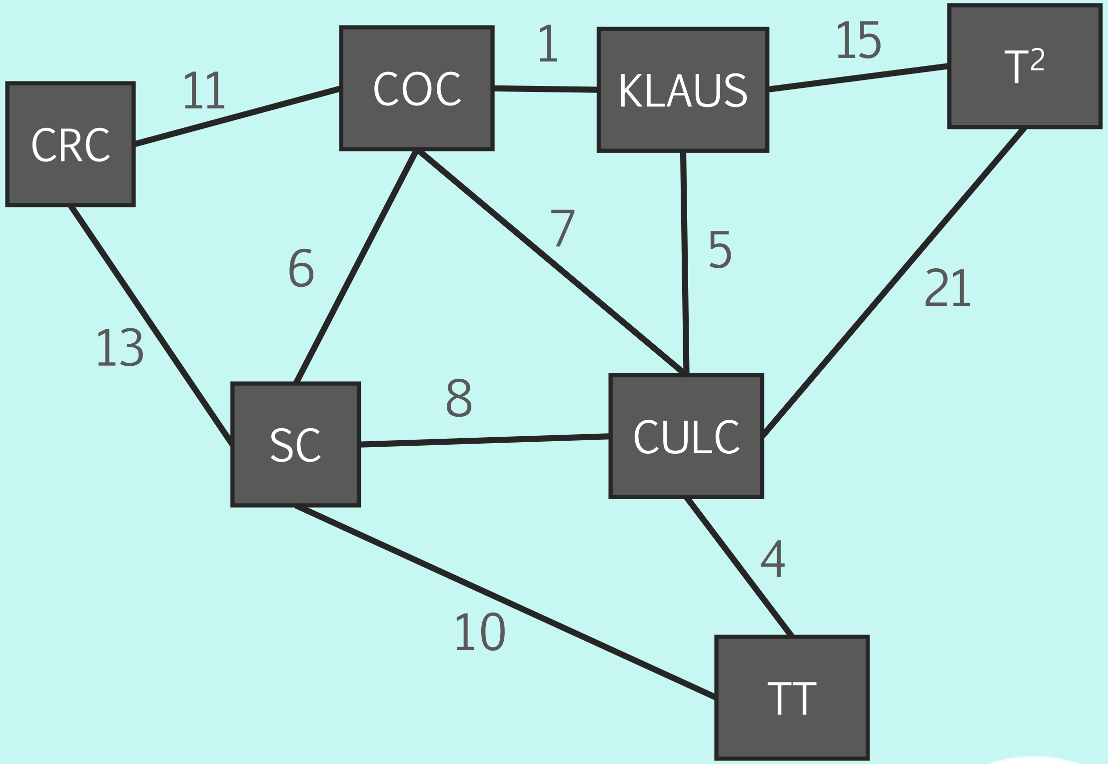


Kruskal's Algorithm Example



Kruskal's Algorithm Implementation

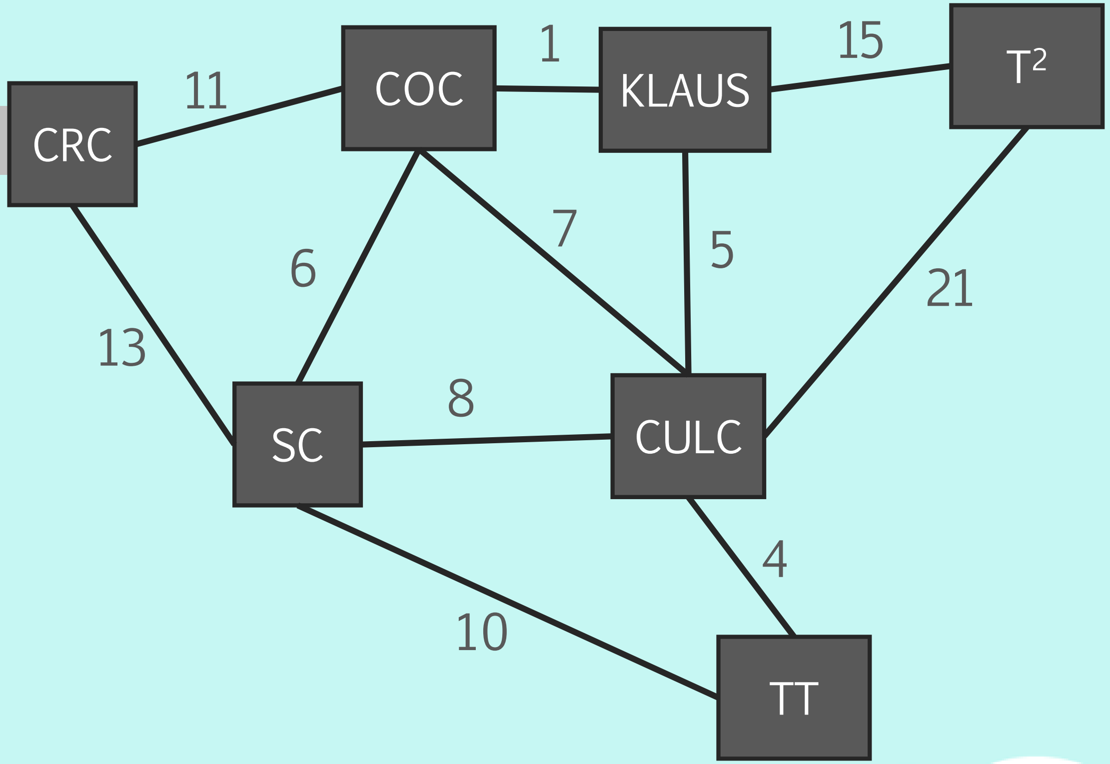
1. To Perform Algorithmically:
    1. Add all your edges into a Priority Queue.
    2. Pull our edges 1 by 1 and add them to your spanning tree.
    3. Stop when all vertices are included in your spanning tree.



Kruskal’s Algorithm Implementation

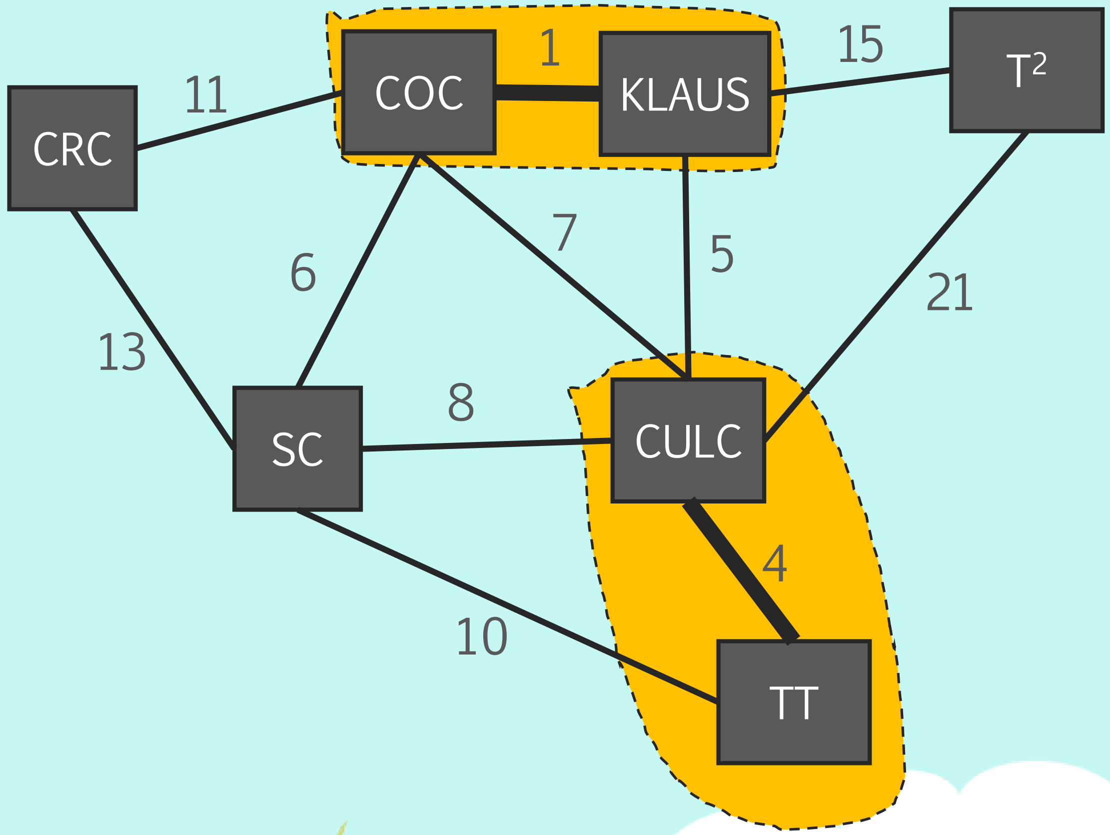
1. To Perform Algorithmically:
   1. Add all your edges into a Priority Queue.
   2. Pull our edges 1 by 1 and add them to your spanning tree.
   3. Stop when all vertices are included in your spanning tree.

- When should you add an edge to your spanning tree?



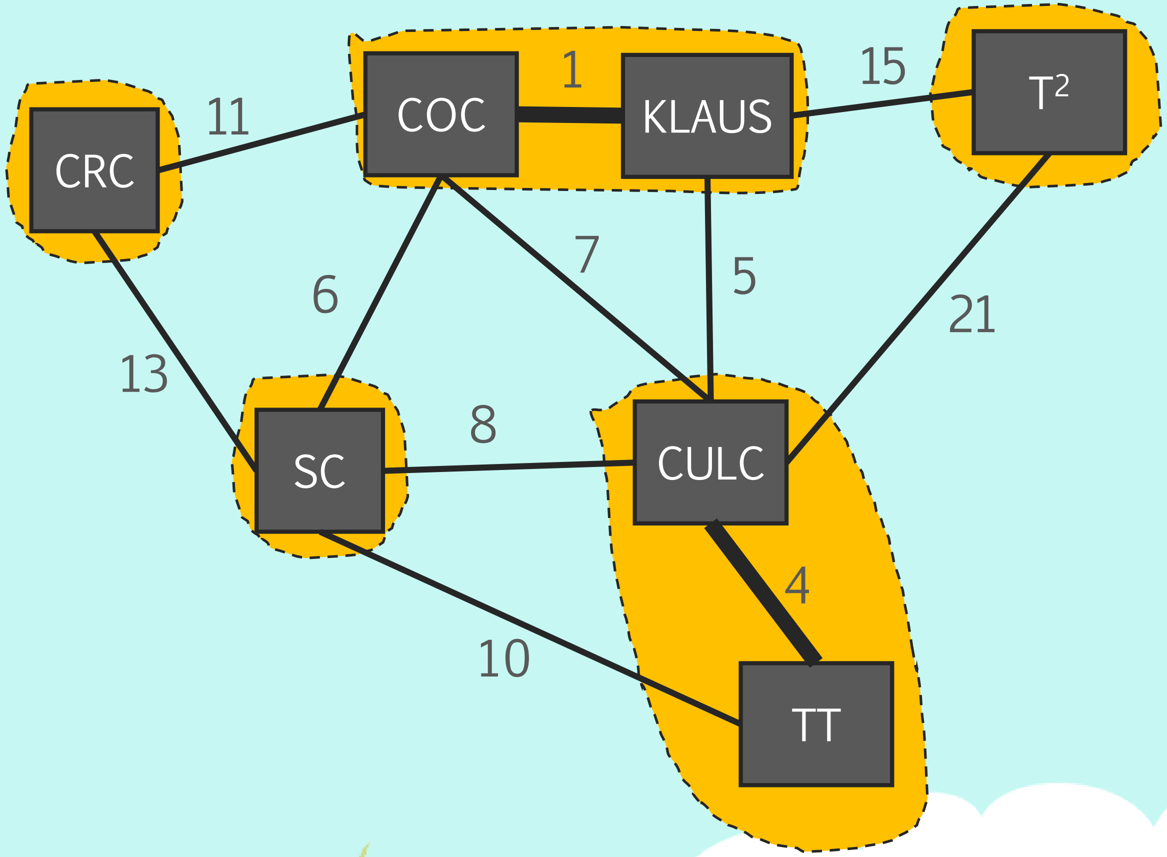
Kruskal's Algorithm Vertex Sets

- When we add edge 1 and 4, we end up with two separate set of vertices.



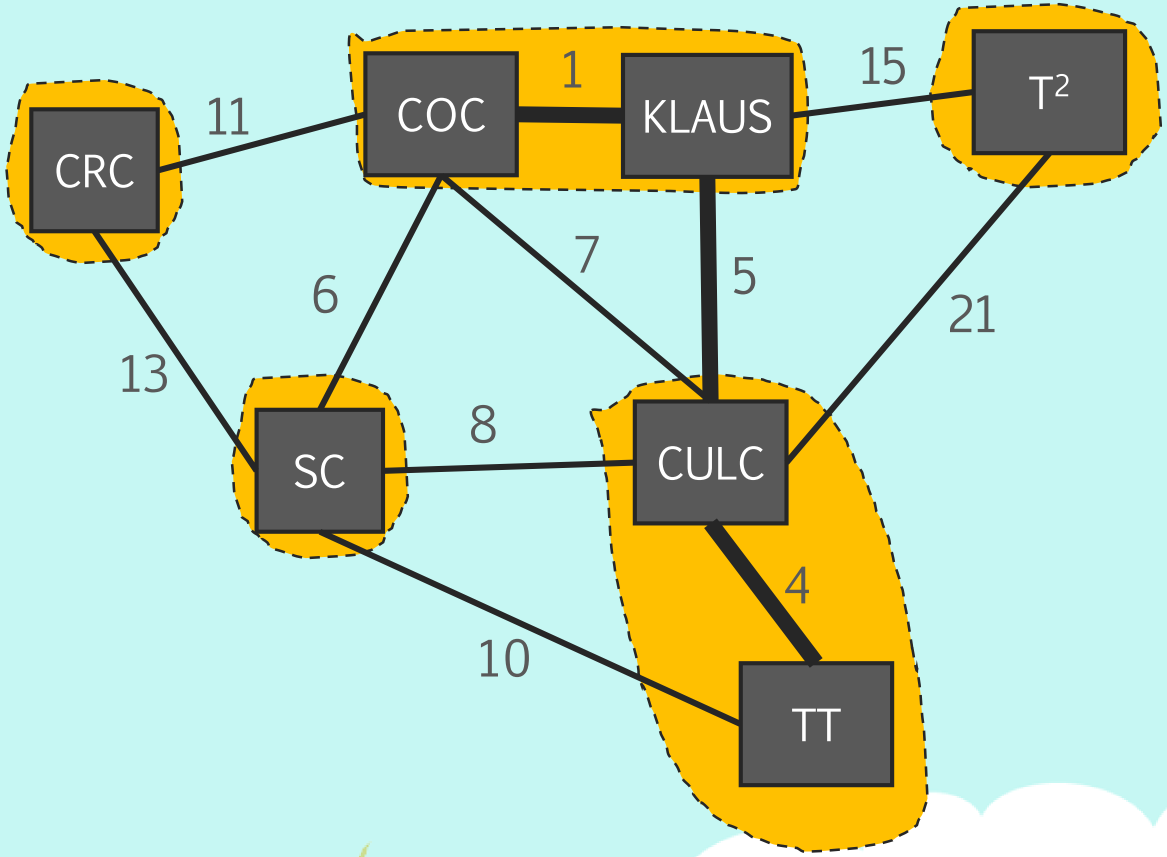
Kruskal's Algorithm Vertex Sets

- When we add edge 1 and 4, we end up with two separate set of vertices.
- In fact, every vertex is in it's own set of vertices.



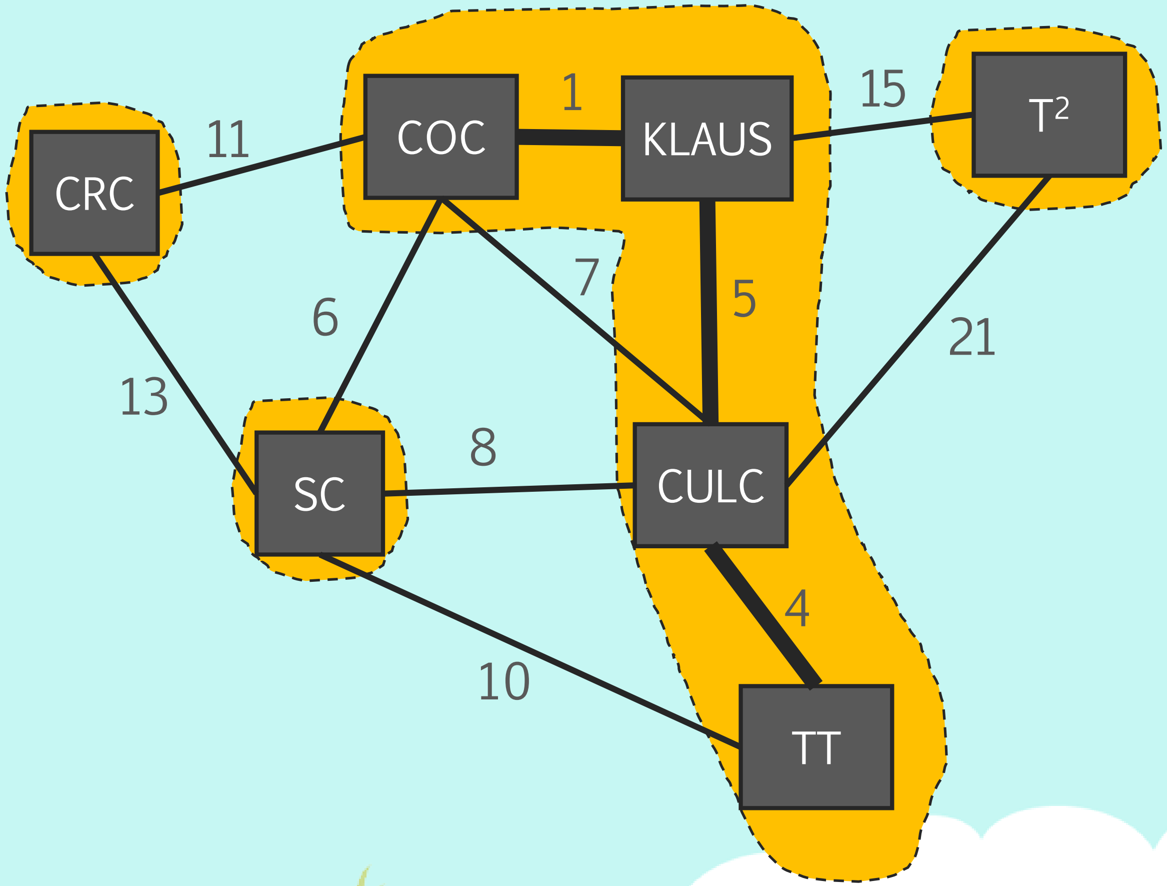
Kruskal's Algorithm Vertex Sets

- When we add edge 1 and 4, we end up with two separate set of vertices.
- In fact, every vertex is in it's own set of vertices.
- If we add edge 5, we connect two separate sets of vertices.



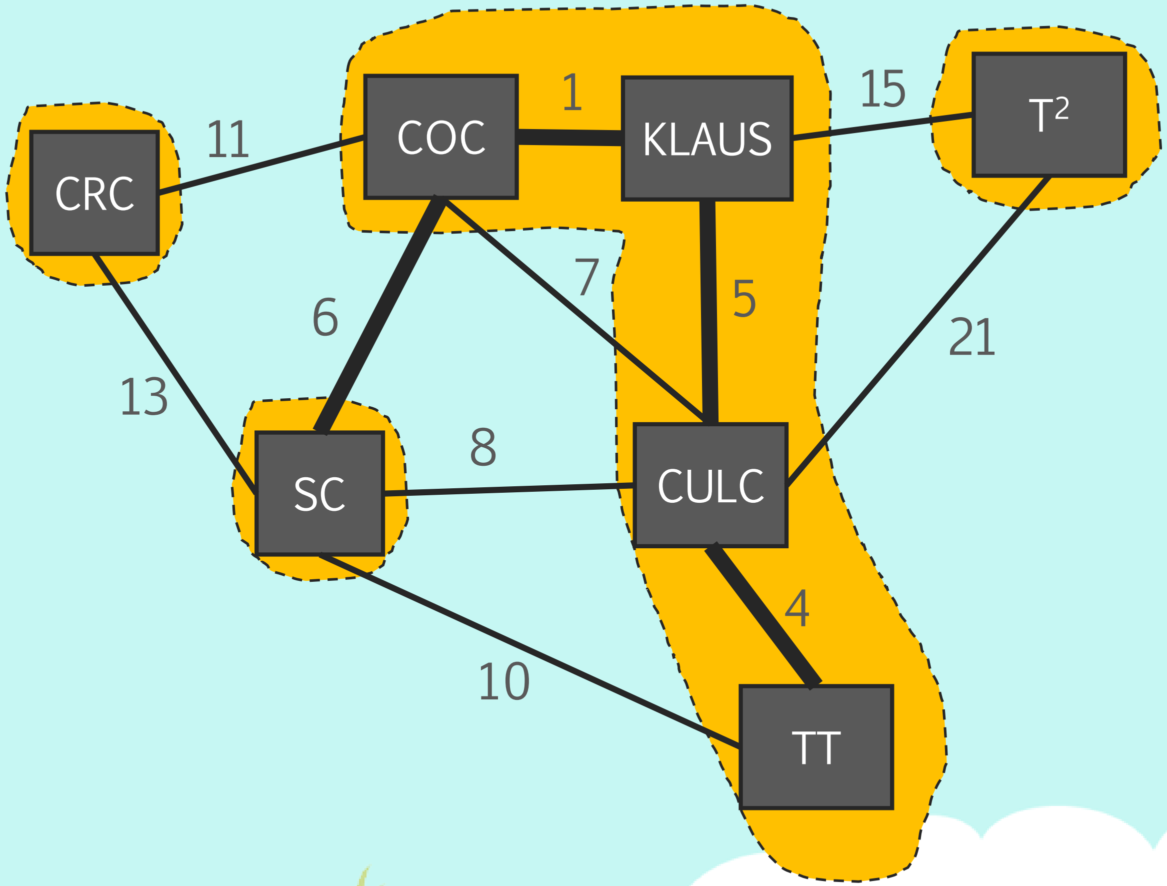
Kruskal's Algorithm Vertex Sets

- When we add edge 1 and 4, we end up with two separate set of vertices.
- In fact, every vertex is in it's own set of vertices.
- If we add edge 5, we connect two separate sets of vertices.



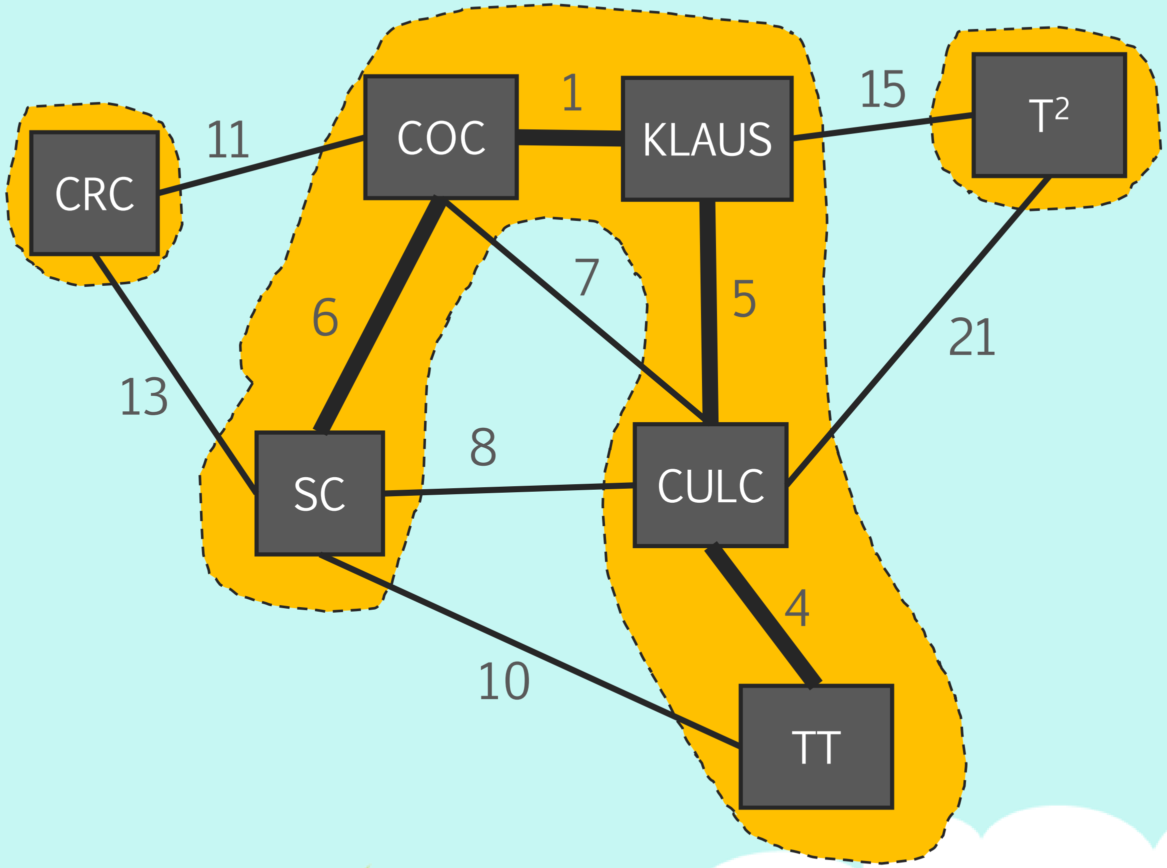
Kruskal's Algorithm Vertex Sets

- When we add edge 1 and 4, we end up with two separate set of vertices.
- In fact, every vertex is in it's own set of vertices.
- If we add edge 5, we connect two separate sets of vertices.
- Adding edge 6 also connects two separate sets of vertices.



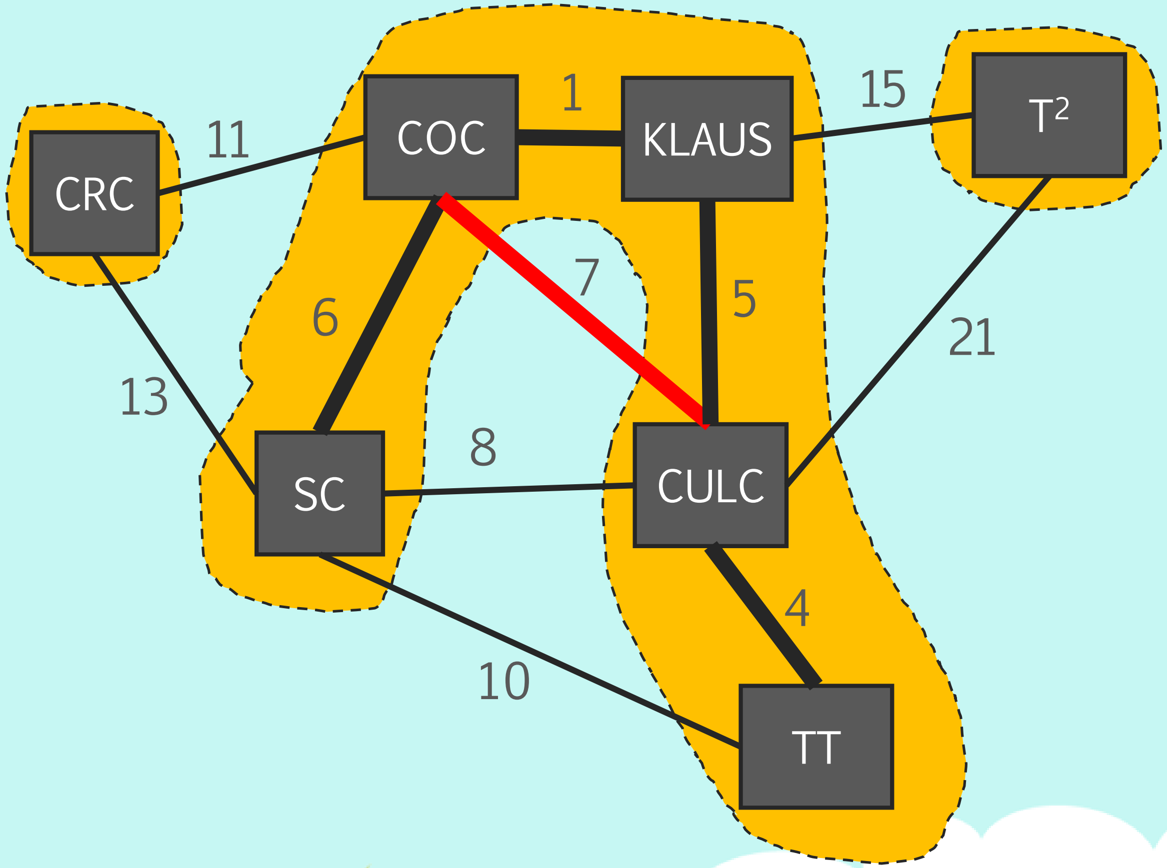
Kruskal's Algorithm Vertex Sets

- When we add edge 1 and 4, we end up with two separate set of vertices.
- In fact, every vertex is in it's own set of vertices.
- If we add edge 5, we connect two separate sets of vertices.
- Adding edge 6 also connects two separate sets of vertices.



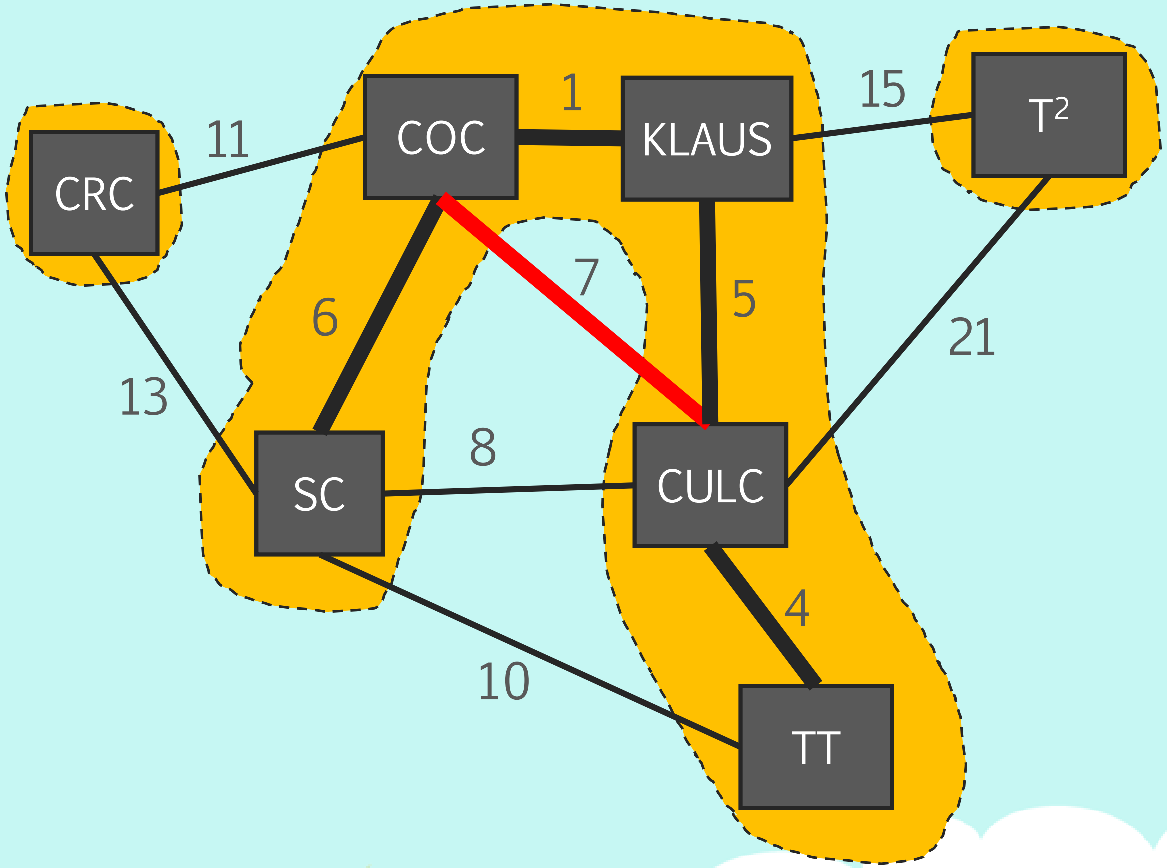
Kruskal's Algorithm Vertex Sets

- When we try to add edge 7, we attempt to connect two vertices from the same set together.
  - we end up with a cycle.
  - This prevents us from adding edge 7.



Kruskal’s Algorithm Vertex Sets

- When we try to add edge 7, we attempt to connect two vertices from the same set together.
    - we end up with a cycle.
    - This prevents us from adding edge 7.
- We’ll use the rule:
    - For a candidate edge to add to our spanning tree, if the vertices u, v from (u, v) are part of the same set of vertices, do not add the edge.
- To organize these sets of vertices, we’ll use a new data structure.
    - Disjoint Set Data Structure

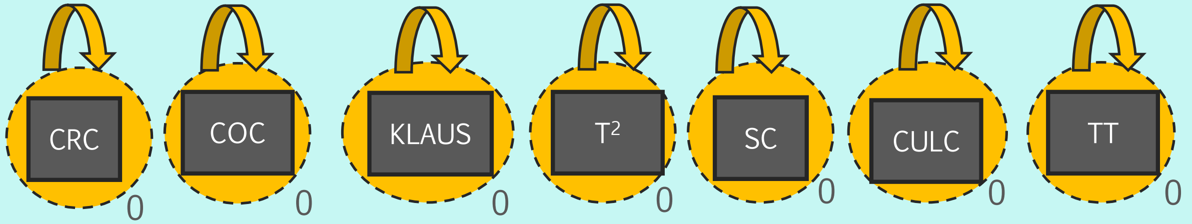


Disjoint Set Data Structure (Union-Find)

- Disjoint-Set maintains a set of subsets. The main purpose is to merge subsets together (Union) and see if elements are in the same subset (Find).
  - Union(A, B) – Find the two subsets elements A and B are in and merge together.
  - Find(A) – Finds the subset A is in.
- In our case with Kruskal's, we'll maintain a set vertices. Initially each vertex will be in its own subset.
  - When Kruskal's attempts to add edge (u, v) to the spanning tree, we see if Find(u) and Find(v) are the same subset.
    - If the subsets are not the same, then we add edge (u, v) to our spanning tree and Union(u, v).
    - Else, we ignore edge (u, v).

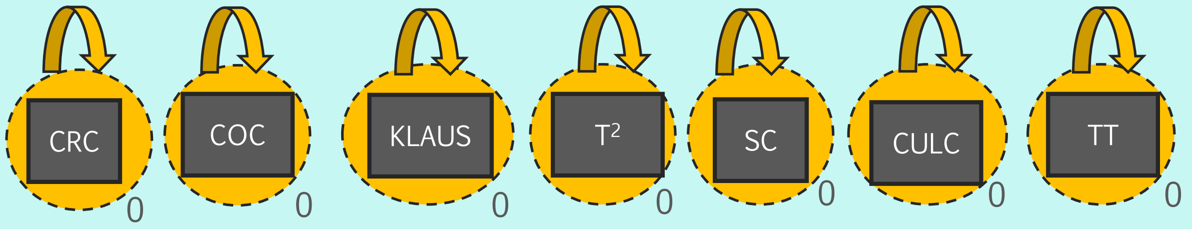
Disjoint Set Subset Representation

- Subsets are represented as Trees.
- ```
  Node {
     Data data
     Node parent
     int  rank = 0
  }
  ```
  - Data is data in the subset
  - Parent pointer points to a parent node. Root nodes point to themselves.
  - Rank is similar to height. A root node of higher rank has more nodes in the tree. The rank of a node can change.


  ```

Disjoint Set Subset Representation

- Find(A): finds the root of the tree A is in recursively and returns the root.
  - *Path Compression* - All nodes from A to root have their parent pointers point to the root.
    - This optimizes Find() operations for later uses.

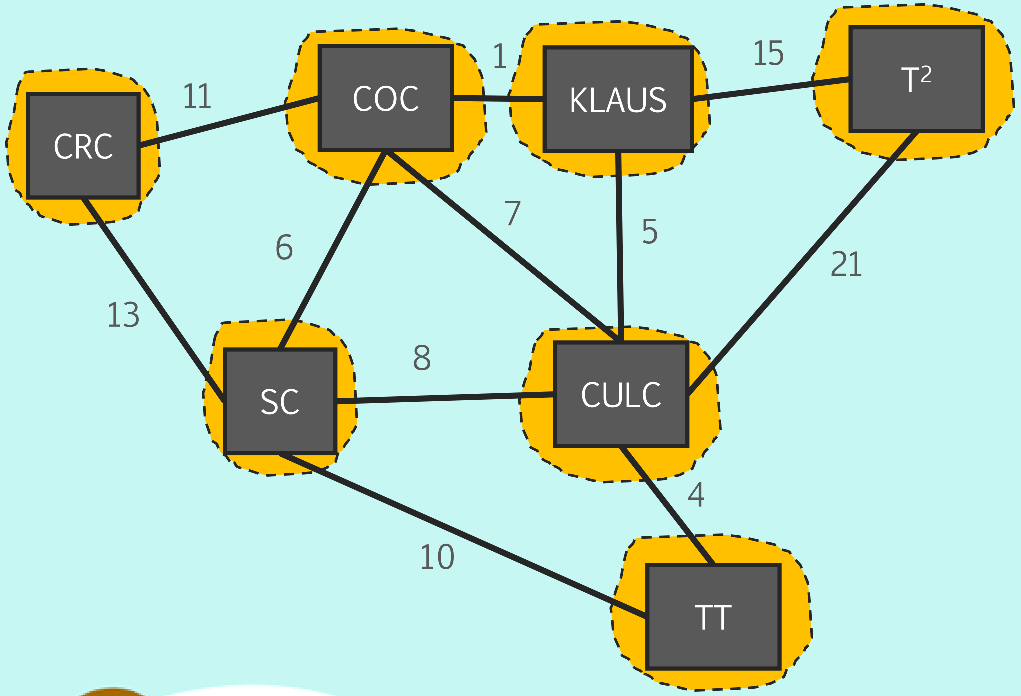


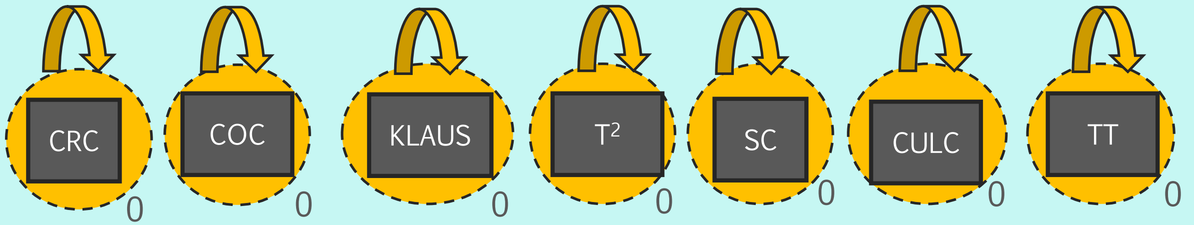
- Union(A, B): find the root of trees A and B are part of and have one root point to the other.
  - *Union by rank* - the root of lower rank points to the root of higher rank. If both are the same, arbitrarily point one to the other and increase the rank of the new root.

Disjoint Set Analysis

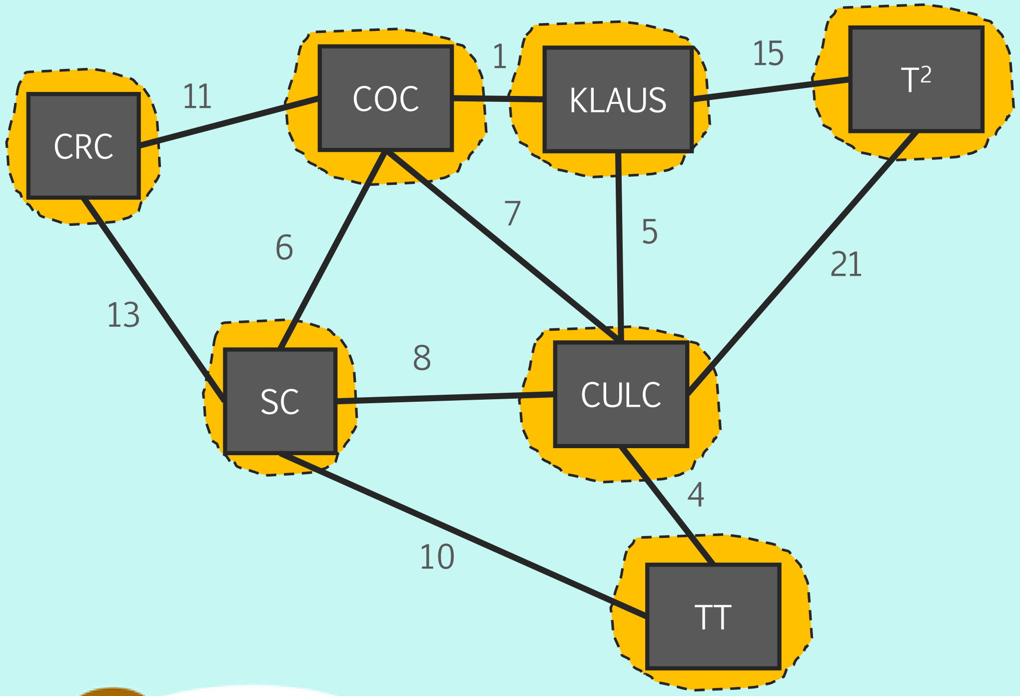
- With both path compression and union by rank, each operation has an amortized running time of $O(\alpha(n))$.
  - $\alpha(n)$ is the inverse Ackermann Function. This is an extremely slowly growing function. Practically $\alpha(n) <= 4$. You can treat this as $O(1)$.
  - Formal Proof: CLRS 21.4

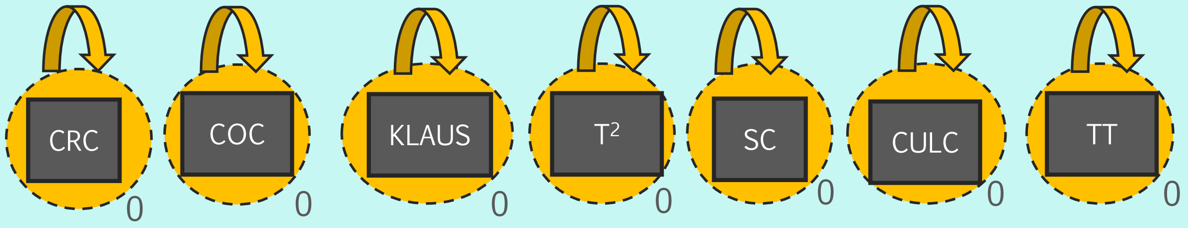
Kruskal’s + Disjoint Set





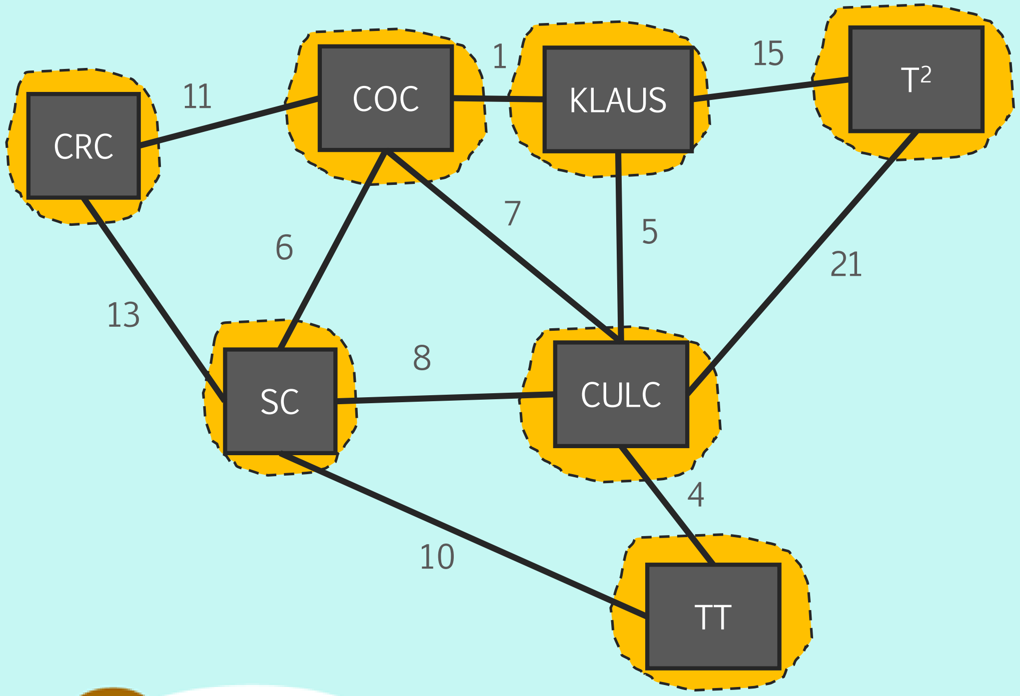
Kruskal's + Disjoint Set

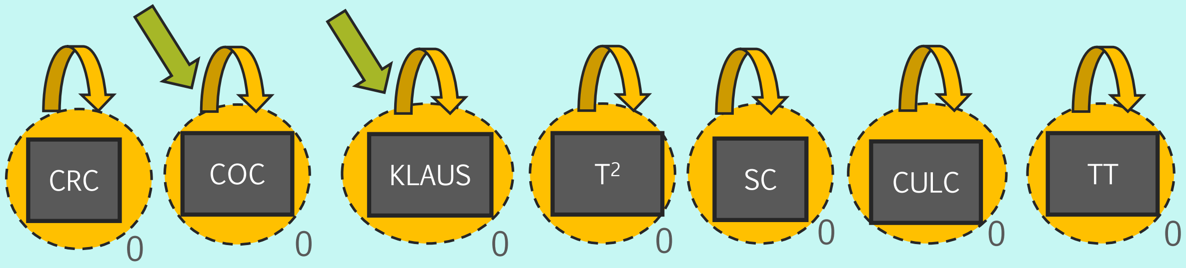




Try adding (COC, KLAUS) to our spanning tree.

Kruskal's + Disjoint Set

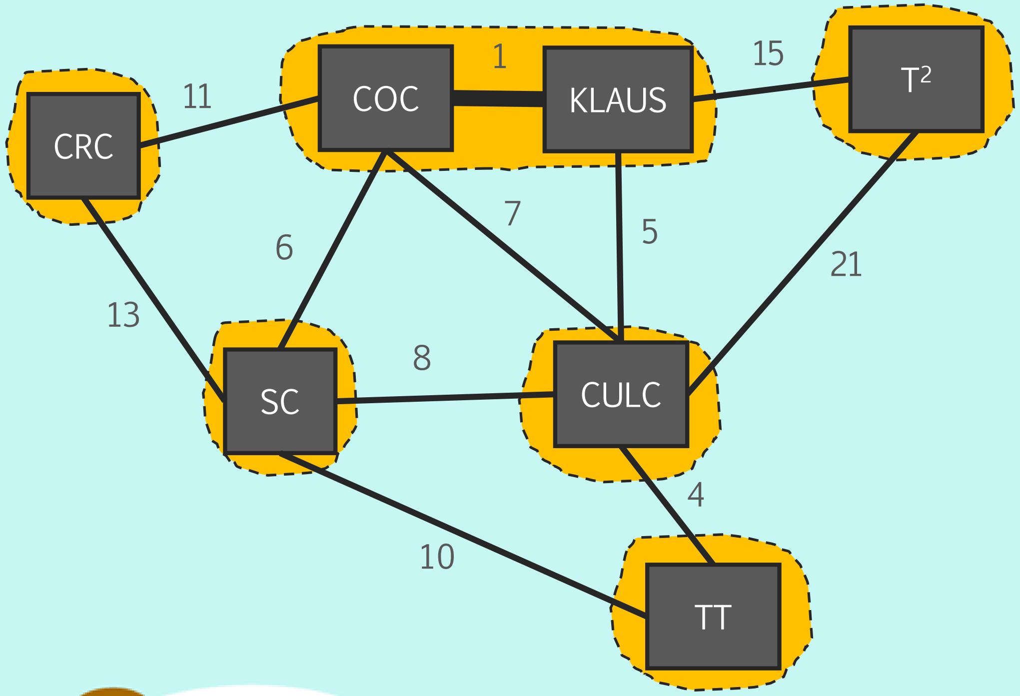


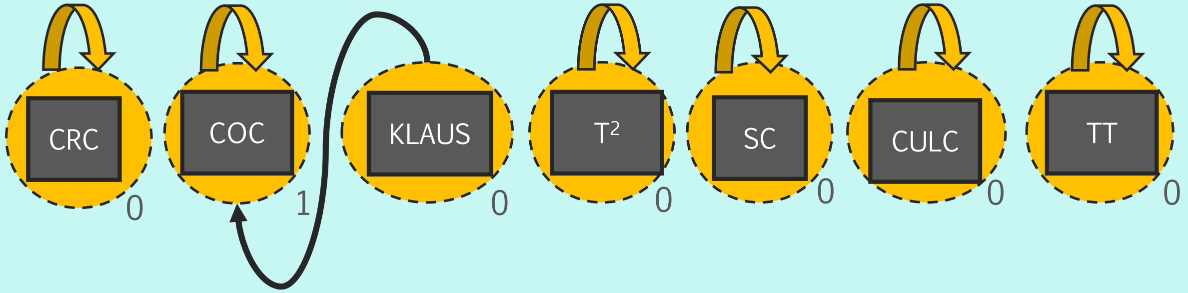


Try adding (COC, KLAUS) to our spanning tree.

Find(COC) != Find(KLAUS).   COC and KLAUS are in separate subsets.

Kruskal's + Disjoint Set



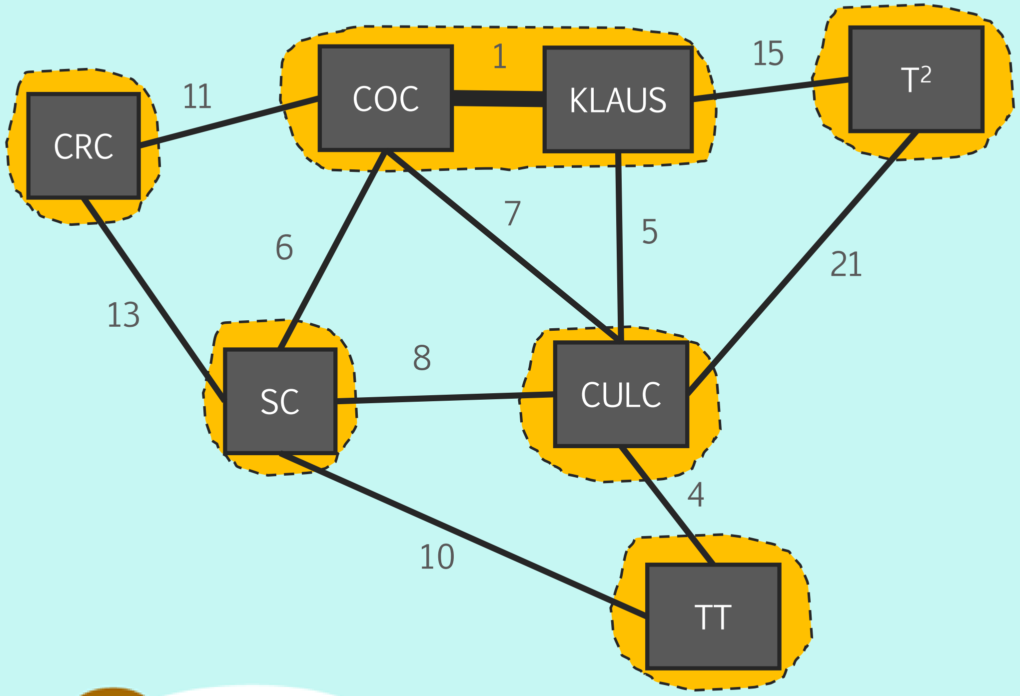


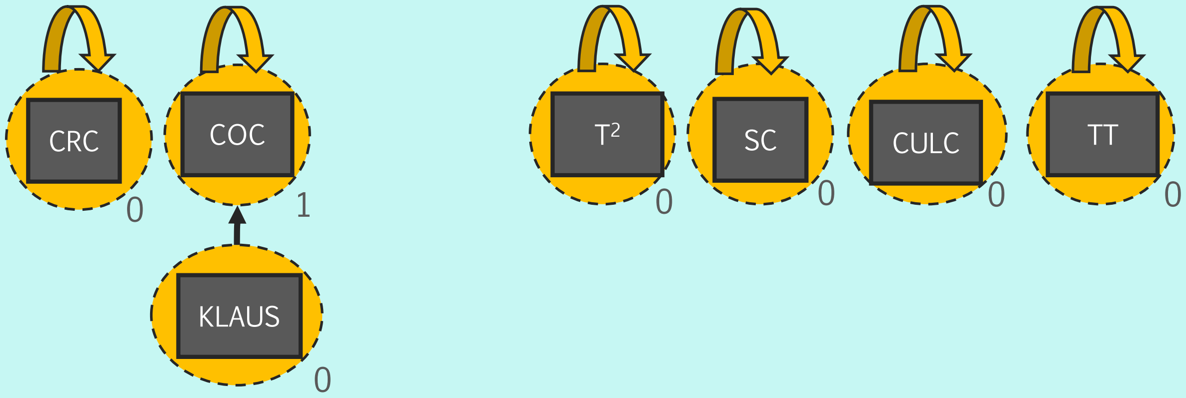
Try adding (COC, KLAUS) to our spanning tree.

Find(COC) != Find(KLAUS). COC and KLAUS are in separate subsets.

We can add (COC, KLAUS) to our spanning tree. We also call Union(COC, KLAUS).

Kruskal's + Disjoint Set



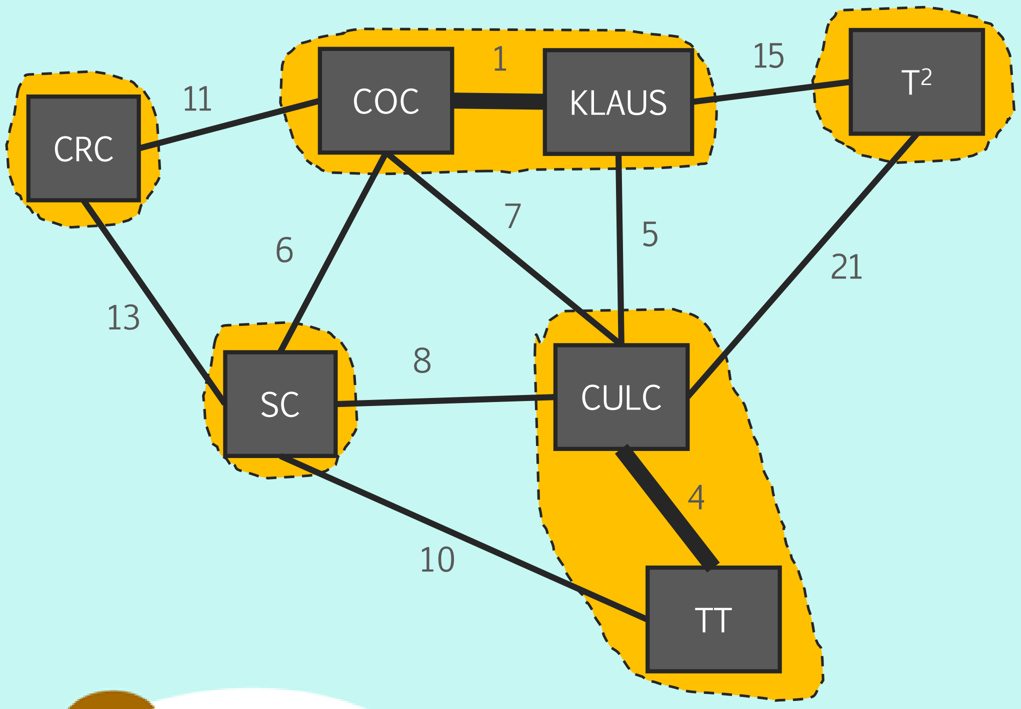


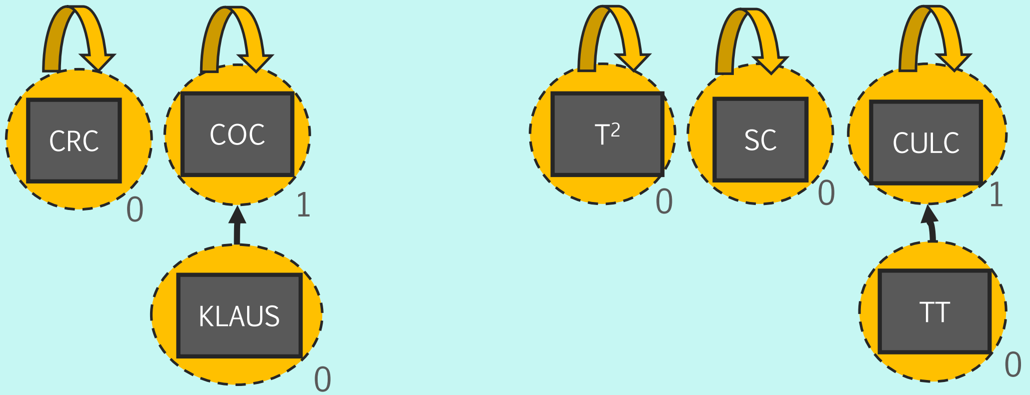
Try adding (COC, KLAUS) to our spanning tree.

Find(COC) != Find(KLAUS). COC and KLAUS are in separate subsets.

We can add (COC, KLAUS) to our spanning tree. We also call Union(COC, KLAUS).

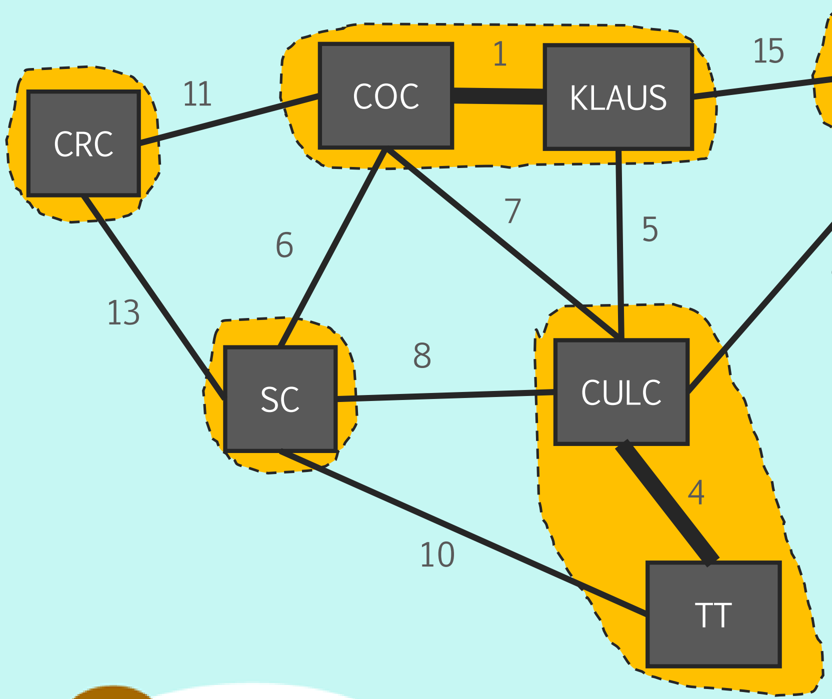
Kruskal's + Disjoint Set

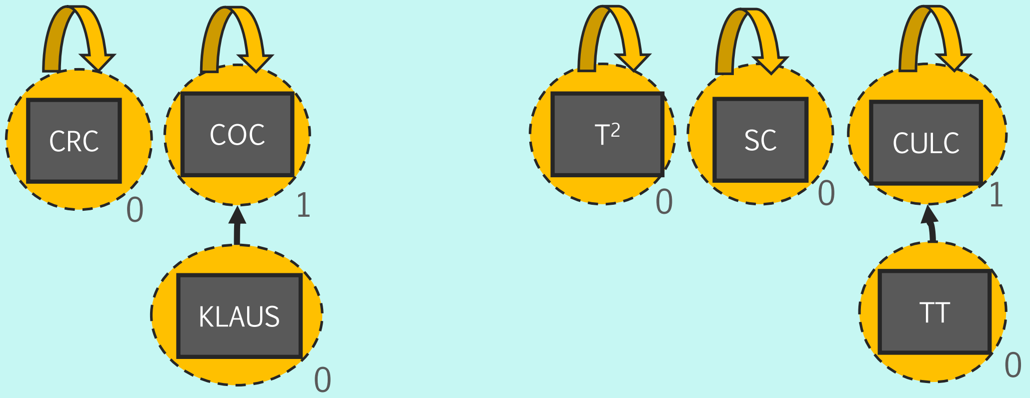




Same for (CULC, TT).

Kruskal's + Disjoint Set

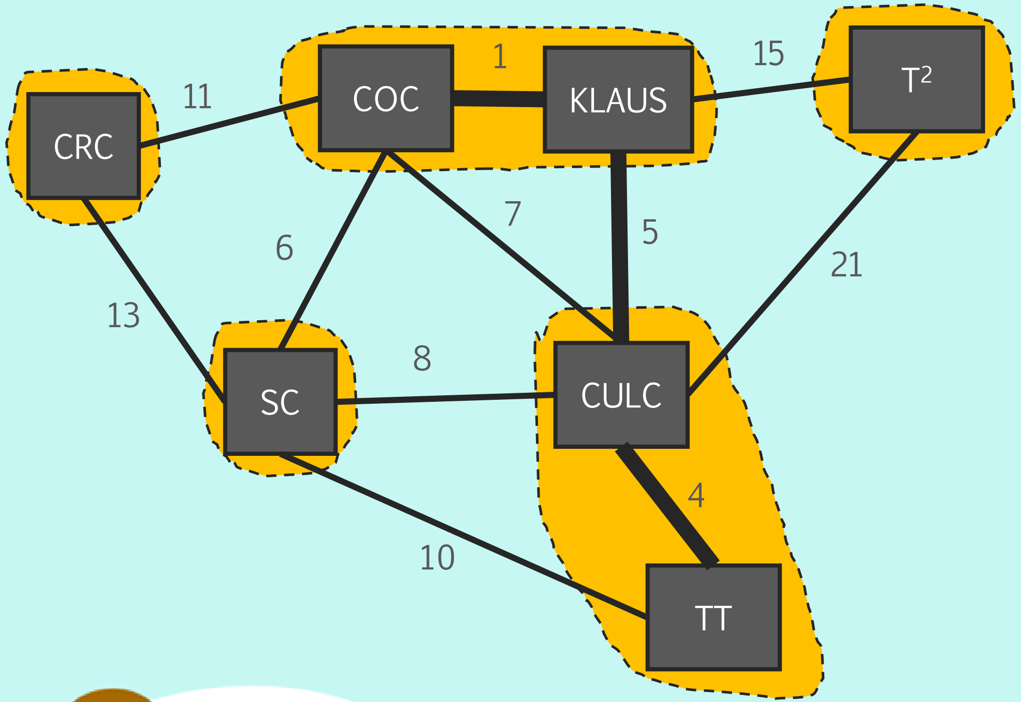


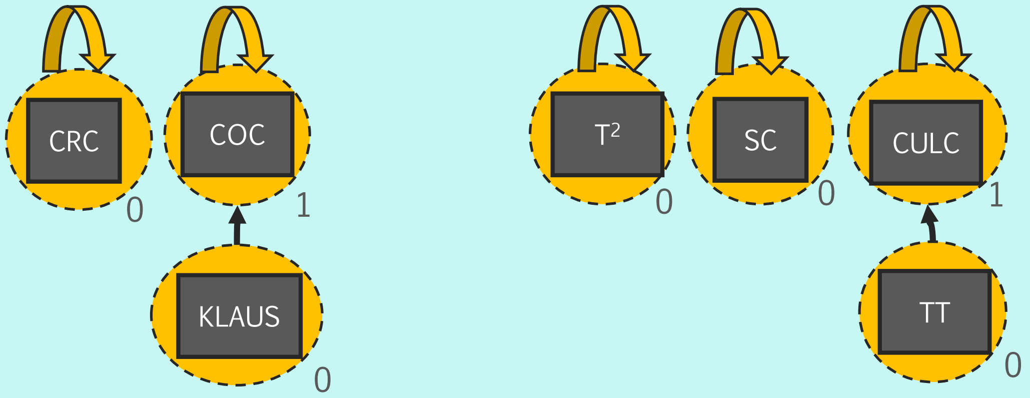


Now let's check (KLAUS, CULC).

Find(KLAUS) = COC.
Find(CULC) = CULC.
Not the same root, so this edge is okay.

Kruskal's + Disjoint Set



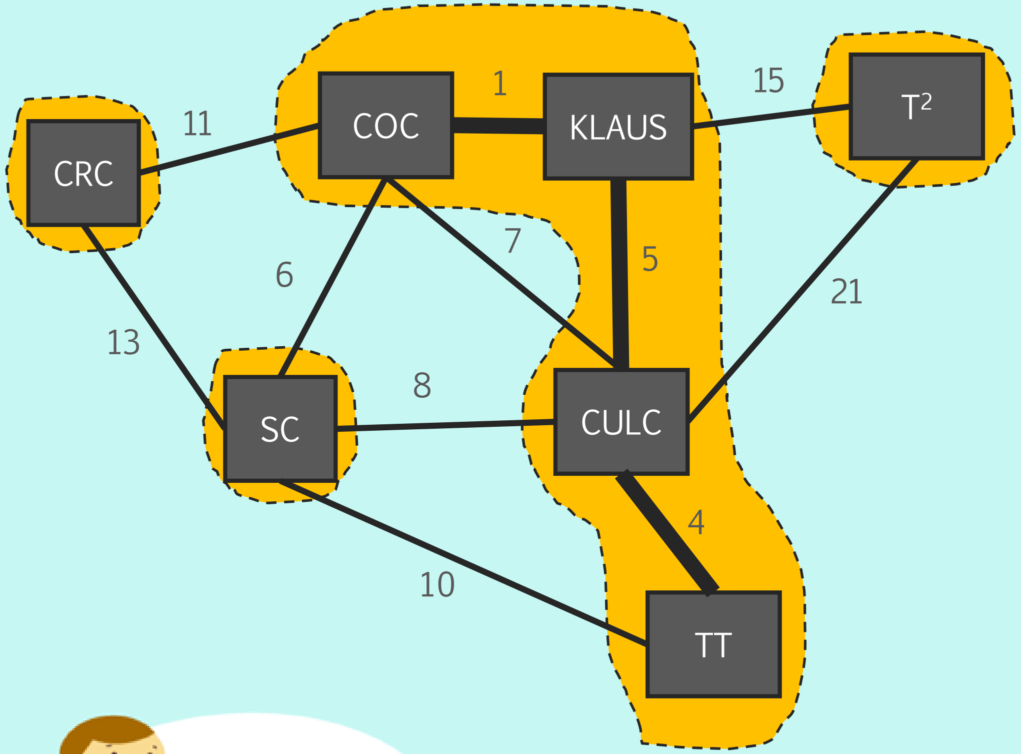


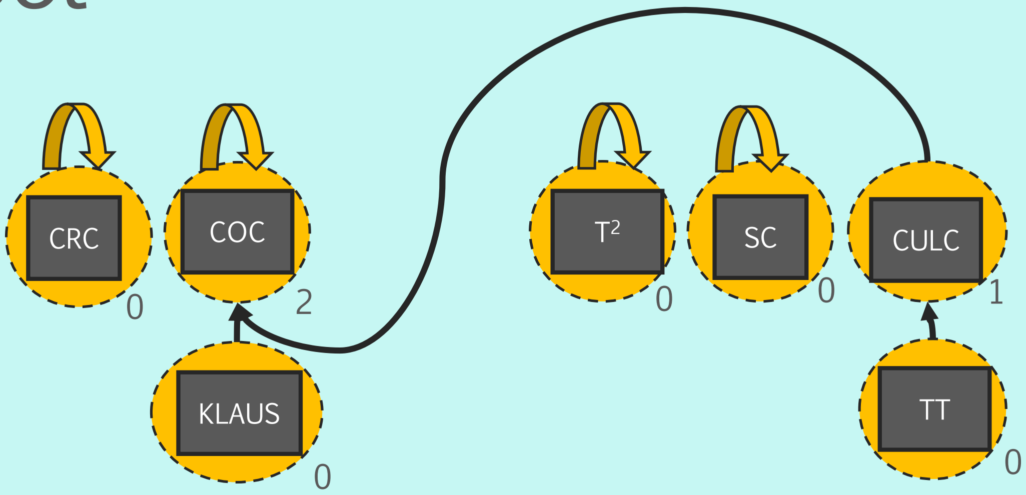
Now let's check (KLAUS, CULC).

Find(KLAUS) = COC.
Find(CULC) = CULC.
Not the same root, so this edge is okay.

Add (KLAUS, CULC) to our spanning tree.

Kruskal's + Disjoint Set





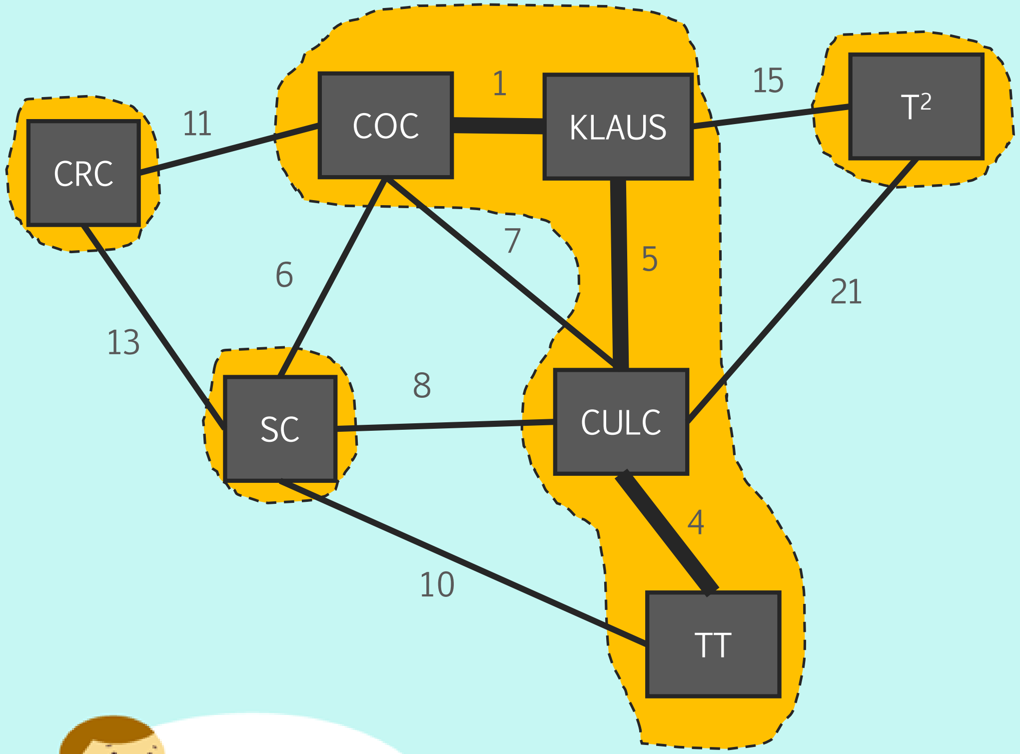
Now let's check (KLAUS, CULC).

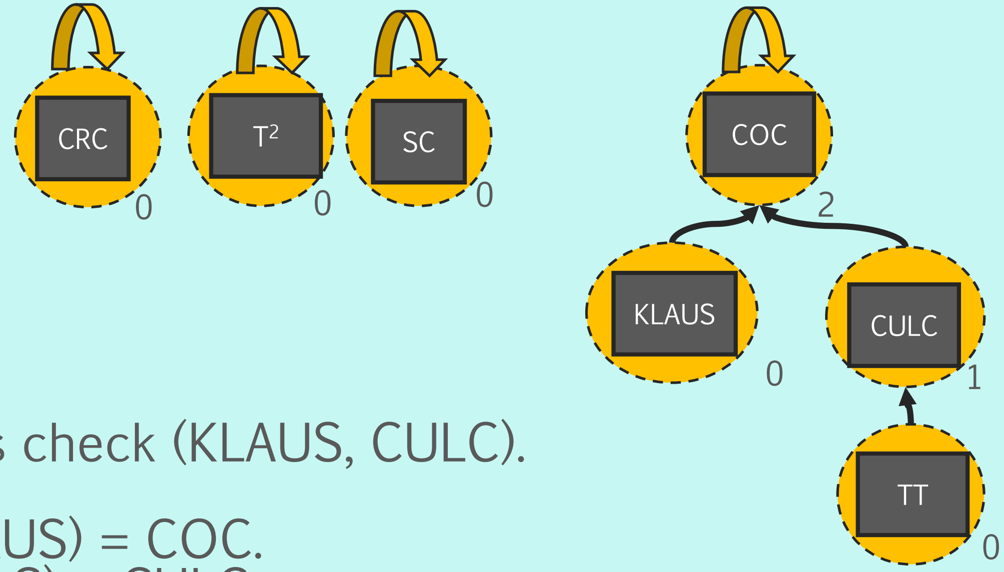
Find(KLAUS) = COC.
Find(CULC) = CULC.
Not the same root, so this edge is okay.

Add (KLAUS, CULC) to our spanning tree.

Union(KLAUS, CULC) will have one root point to the other.

Kruskal's + Disjoint Set





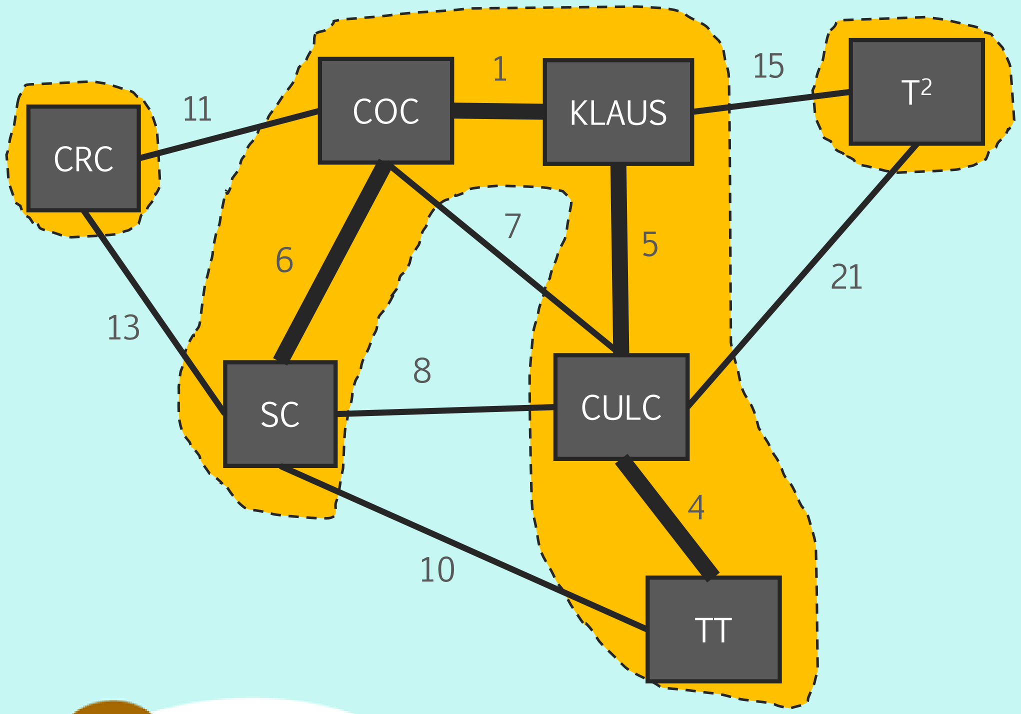
Now let's check (KLAUS, CULC).

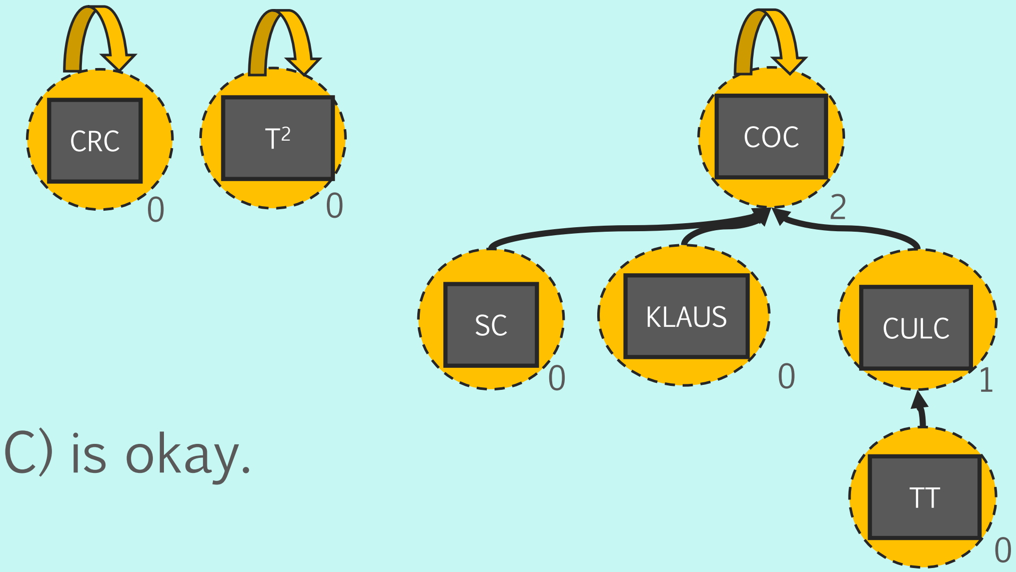
Find(KLAUS) = COC.
Find(CULC) = CULC.
Not the same root, so this edge is okay.

Add (KLAUS, CULC) to our spanning tree.

Union(KLAUS, CULC) will have one root point to the other.

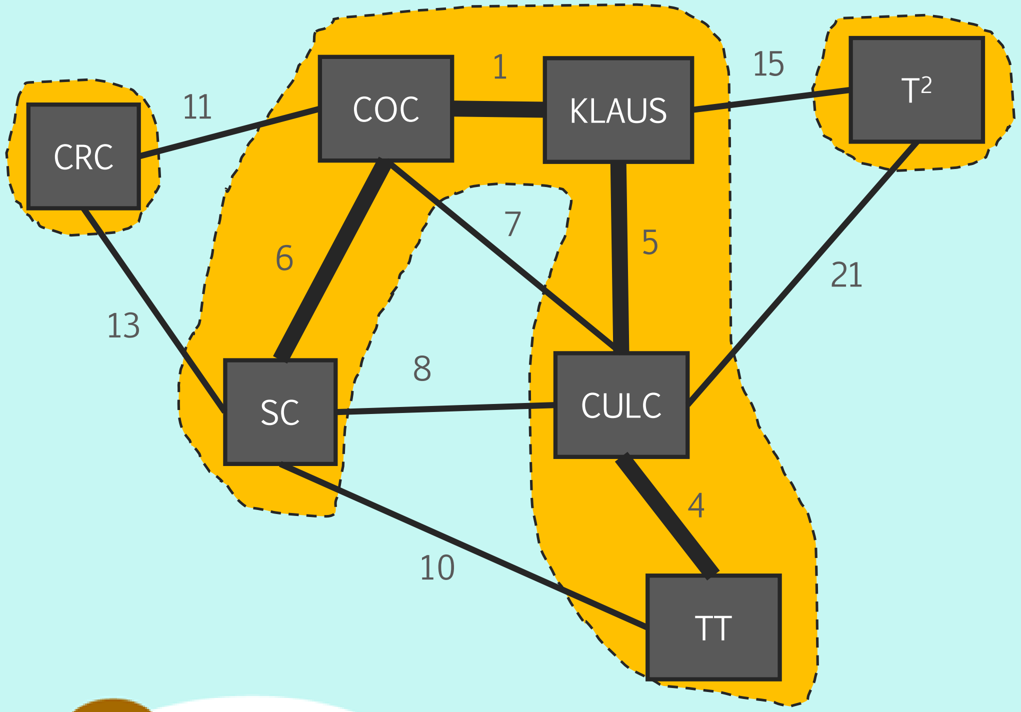
Kruskal's + Disjoint Set

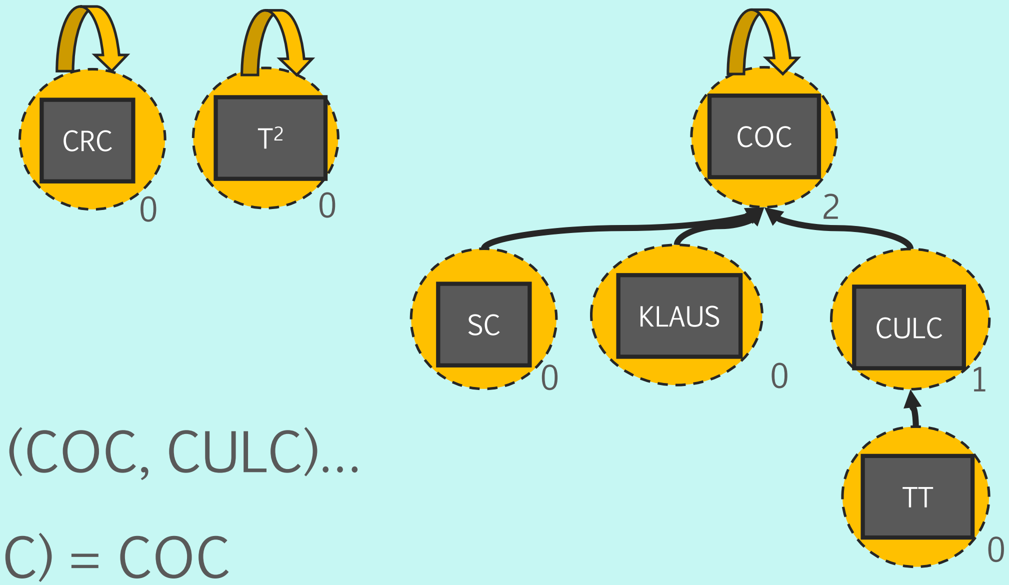




(COC, SC) is okay.

Kruskal's + Disjoint Set

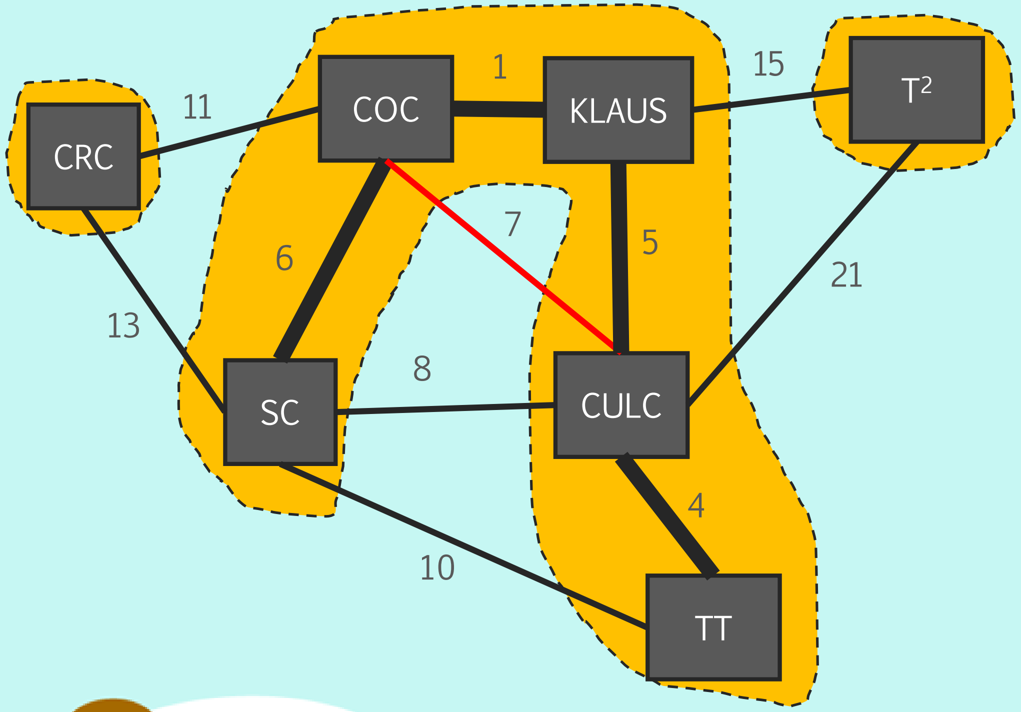


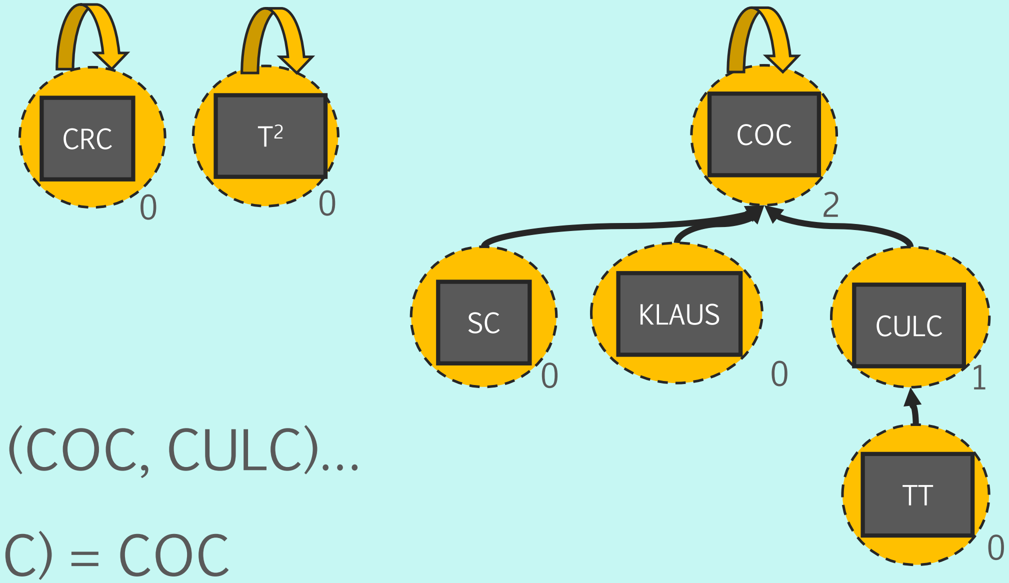


Now for (COC, CULC)...

Find(COC) = COC
Find(CULC) = COC

Kruskal's + Disjoint Set



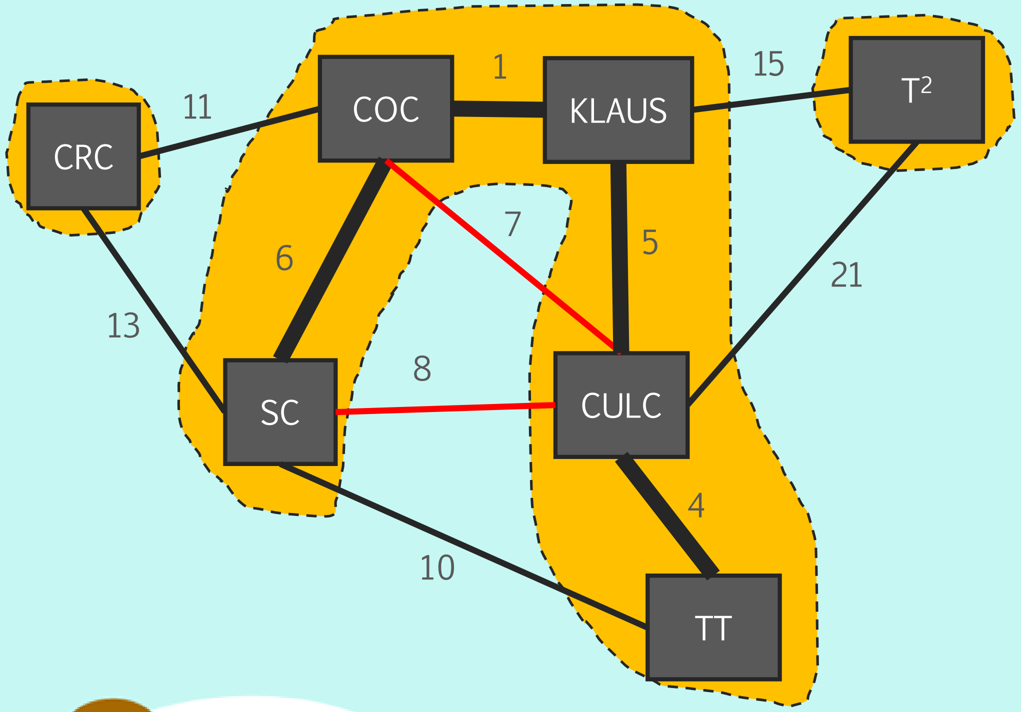


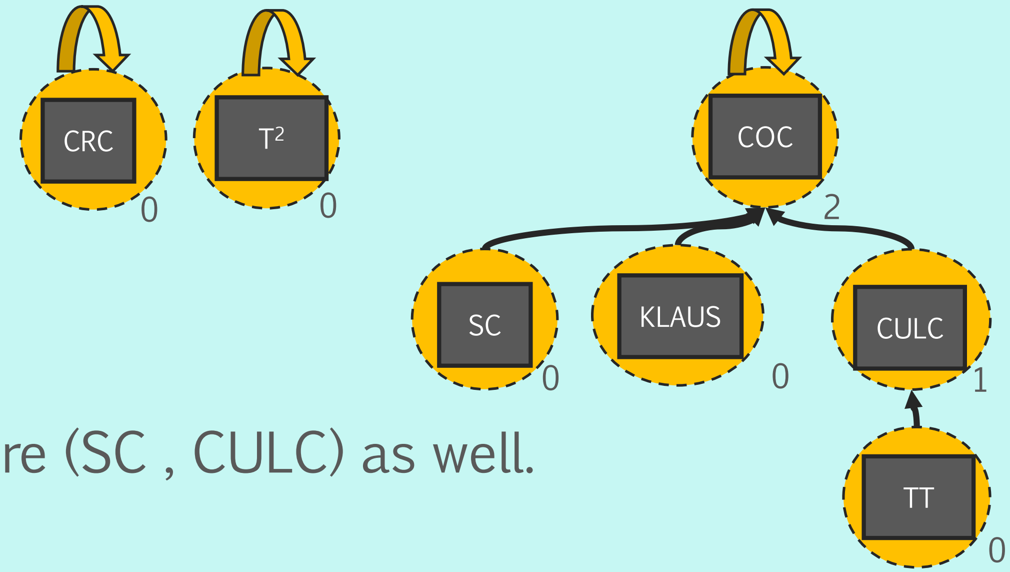
Now for (COC, CULC)...

Find(COC) = COC
Find(CULC) = COC

Since the roots are the same (COC), we ignore this edge.

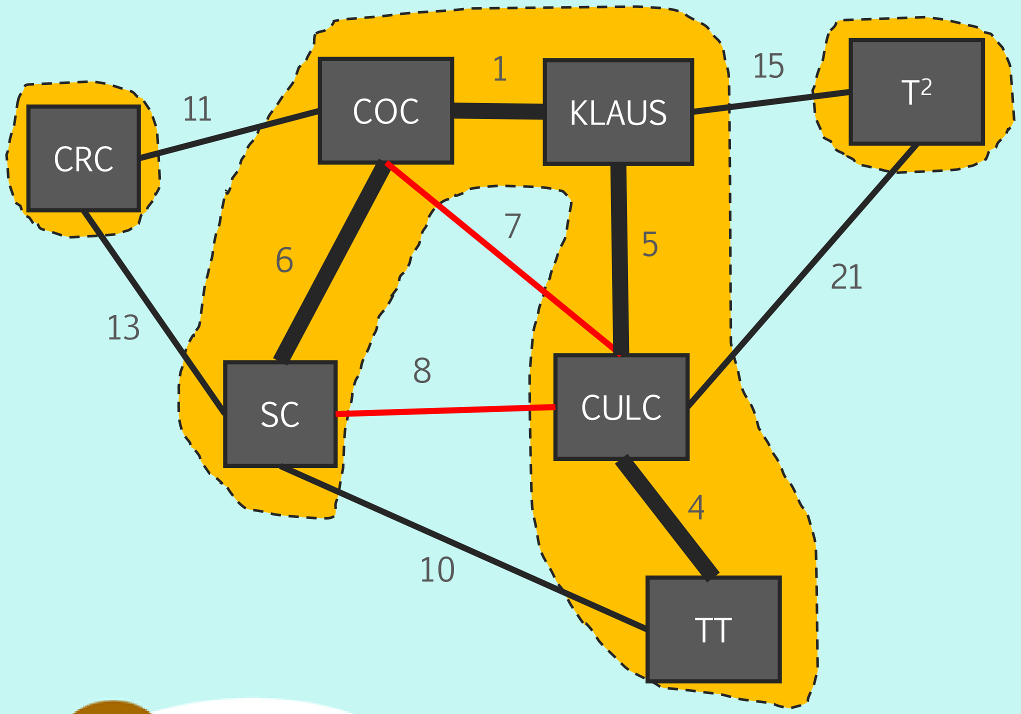
Kruskal’s + Disjoint Set

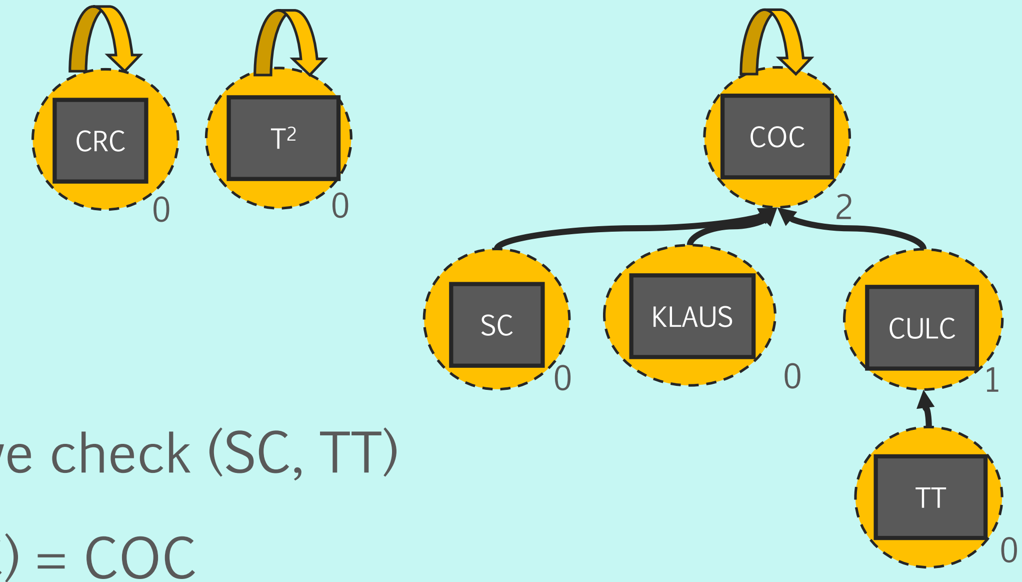




We ignore (SC , CULC) as well.

Kruskal's + Disjoint Set

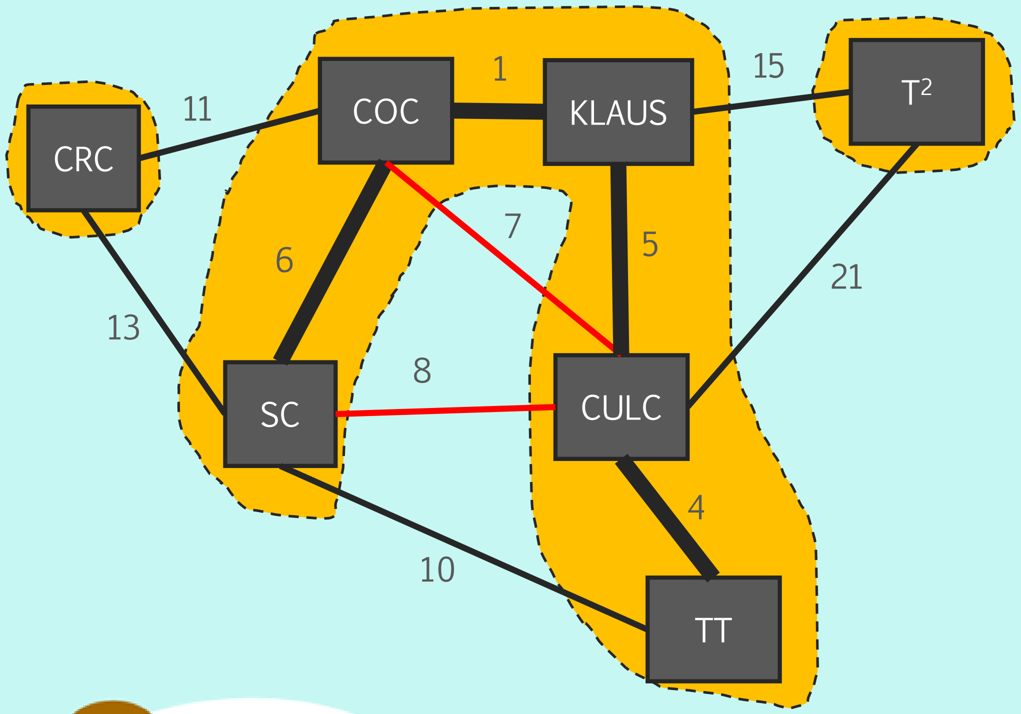


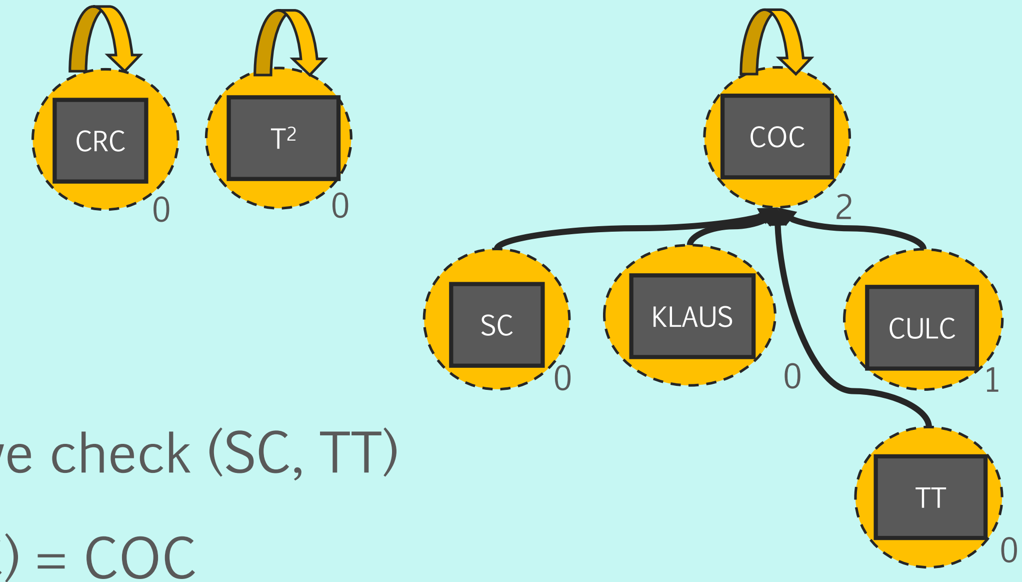


When we check (SC, TT)

Find(SC) = COC

Kruskal's + Disjoint Set



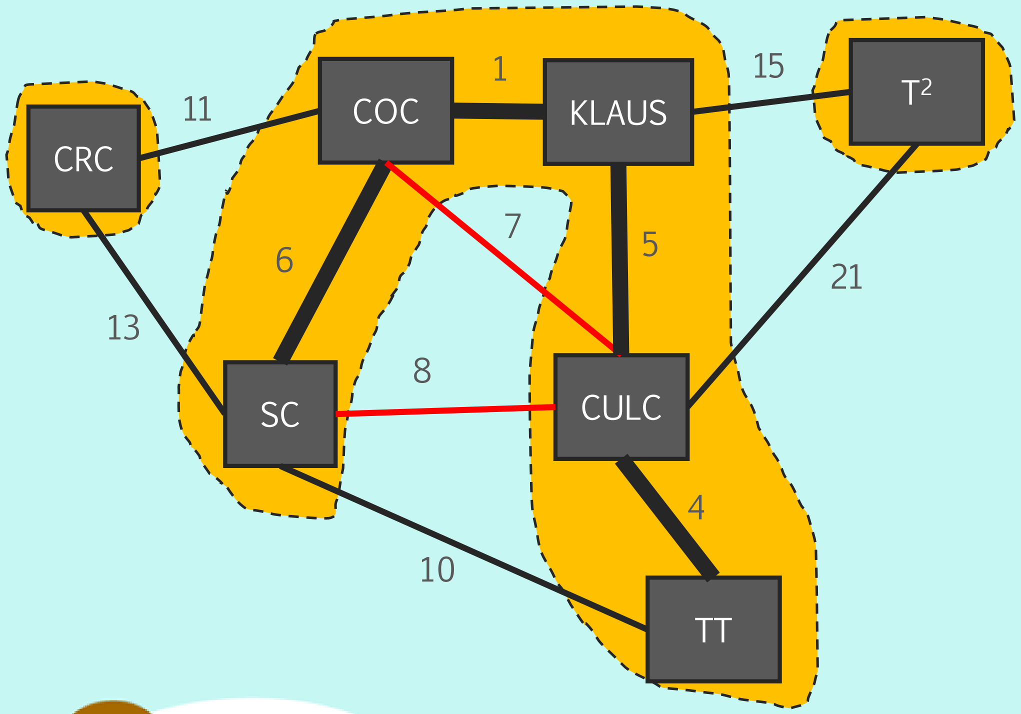


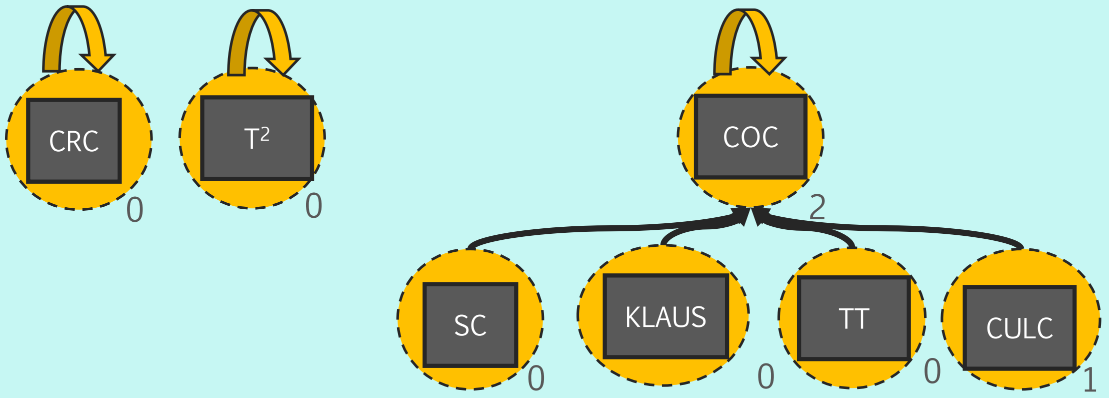
When we check (SC, TT)

Find(SC) = COC
Find(TT) = COC

TT's parent will now be COC. This is due to **Path Compression.**

Kruskal's + Disjoint Set



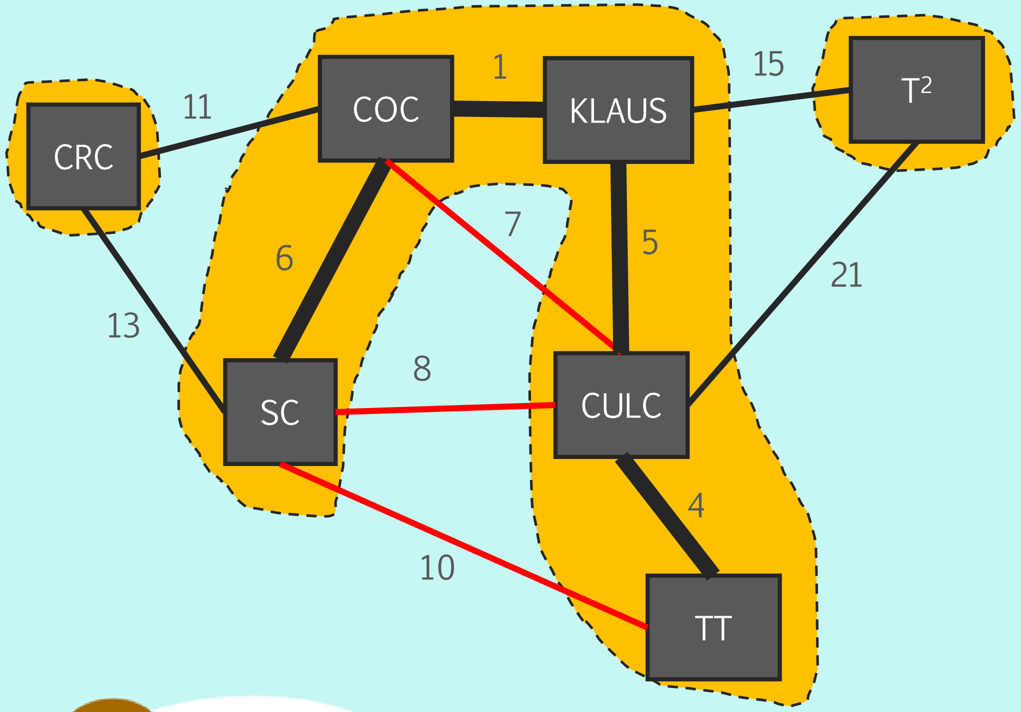


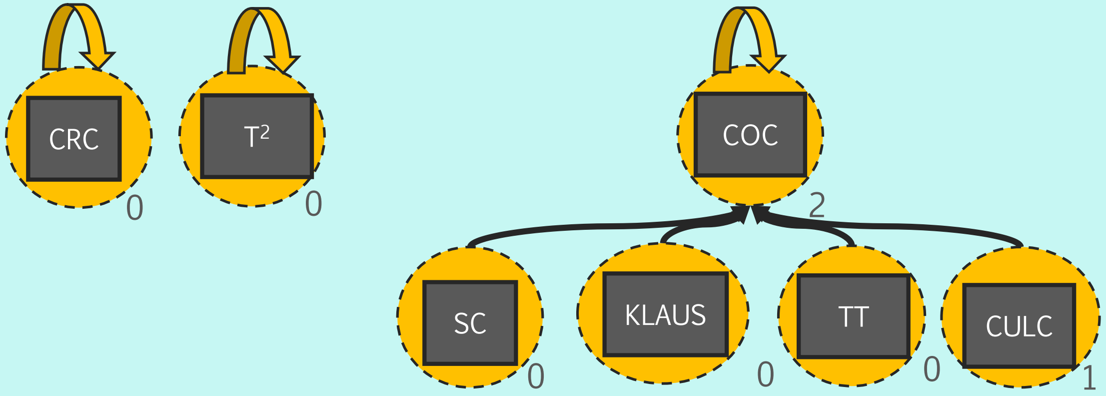
When we check (SC, TT)

Find(SC) = COC
Find(TT) = COC

TT's parent will now be COC. This is due to **Path Compression.**

Kruskal's + Disjoint Set





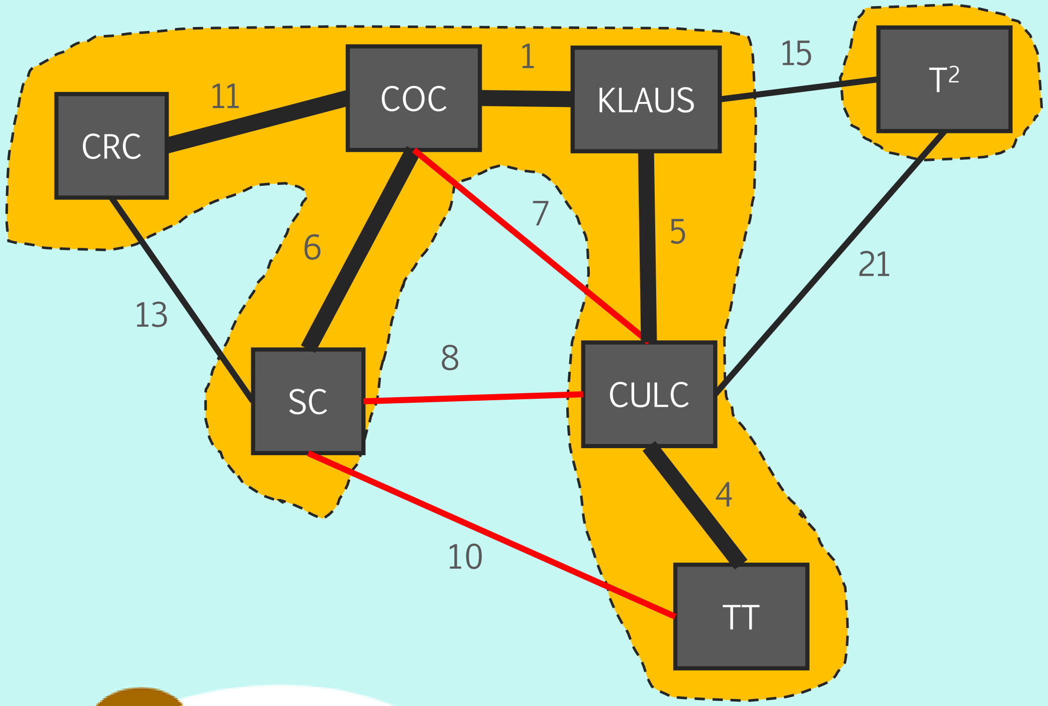
When we check (SC, TT)

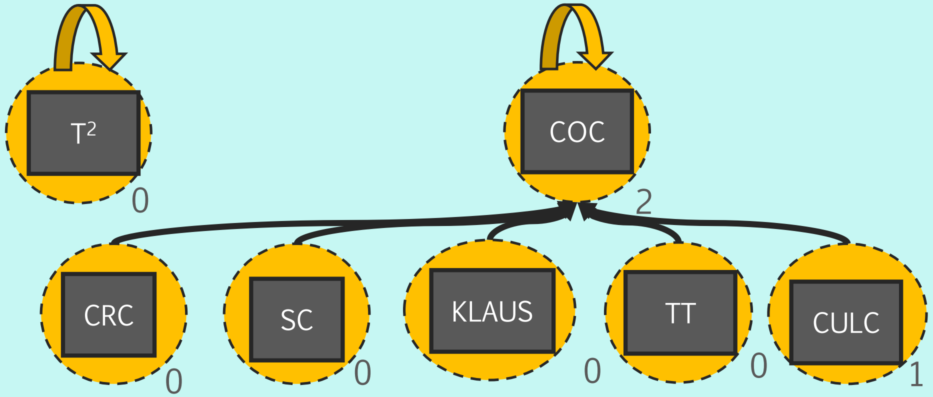
Find(SC) = COC
Find(TT) = COC

TT's parent will now be COC. This is due to **Path Compression.**

We ignore (SC, TT).

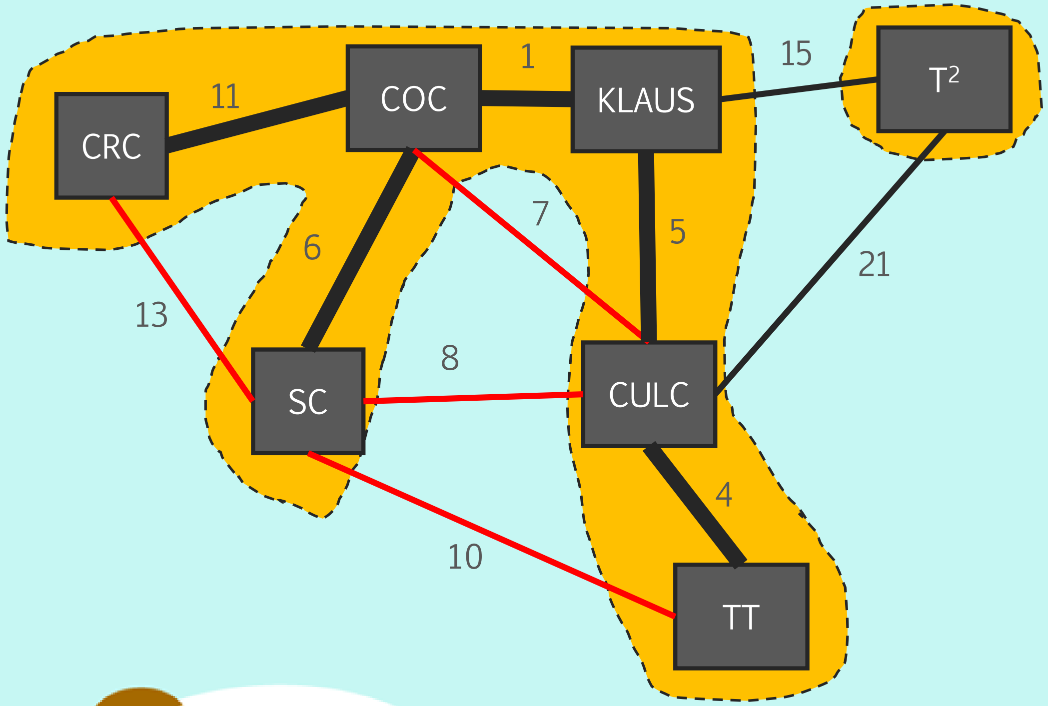
Kruskal's + Disjoint Set

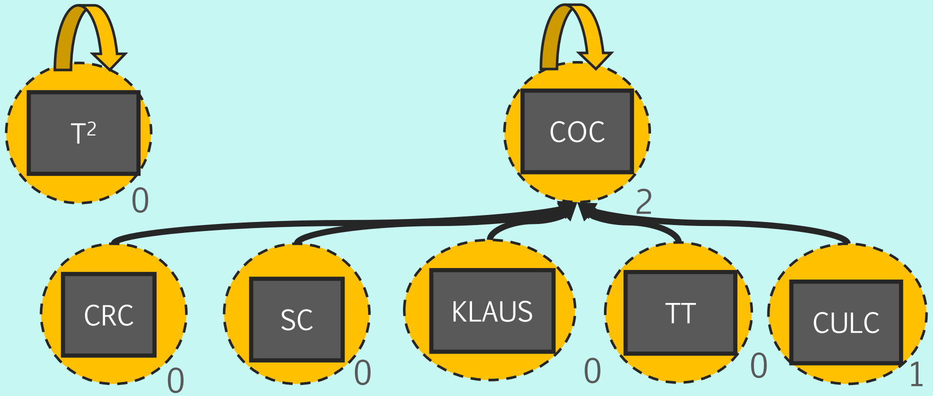




(CRC, COC) will be added to our spanning tree.

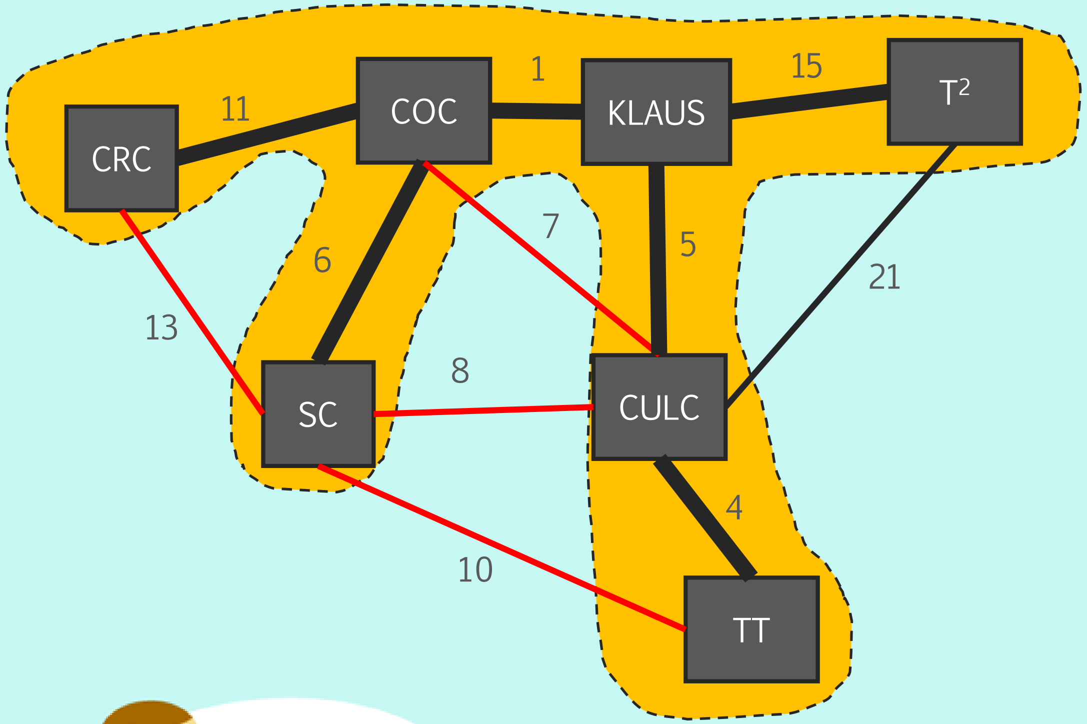
Kruskal's + Disjoint Set

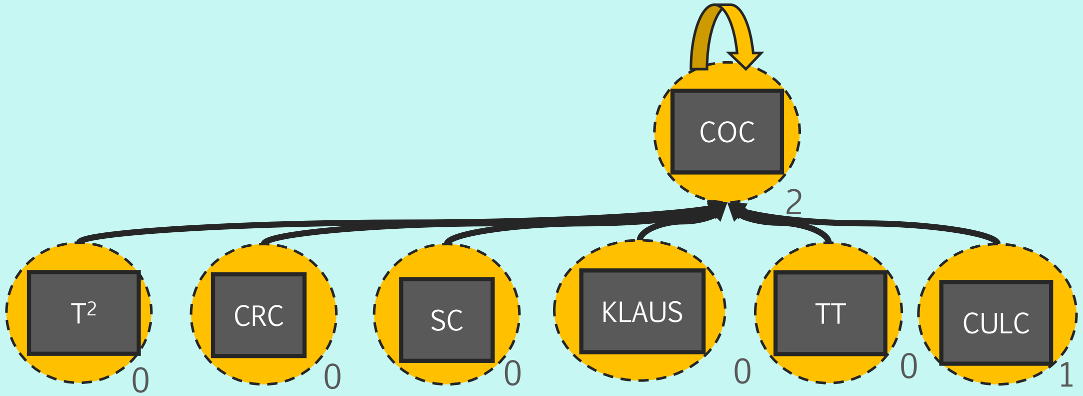




(CRC, SC) is ignored.

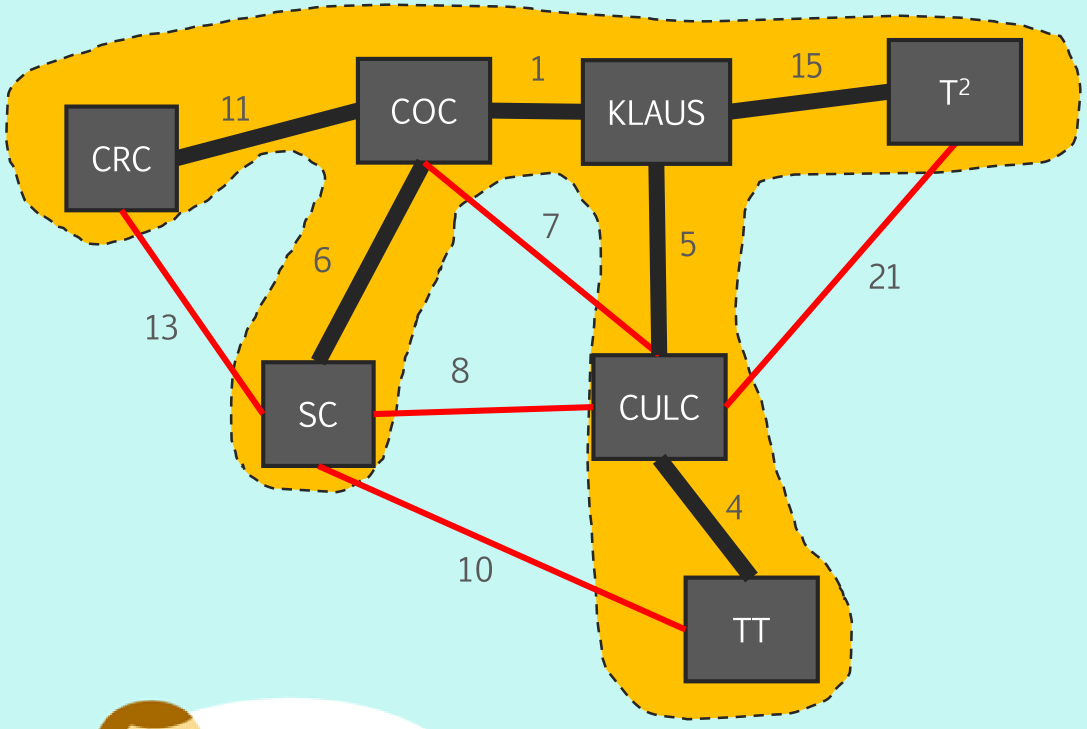
Kruskal's + Disjoint Set

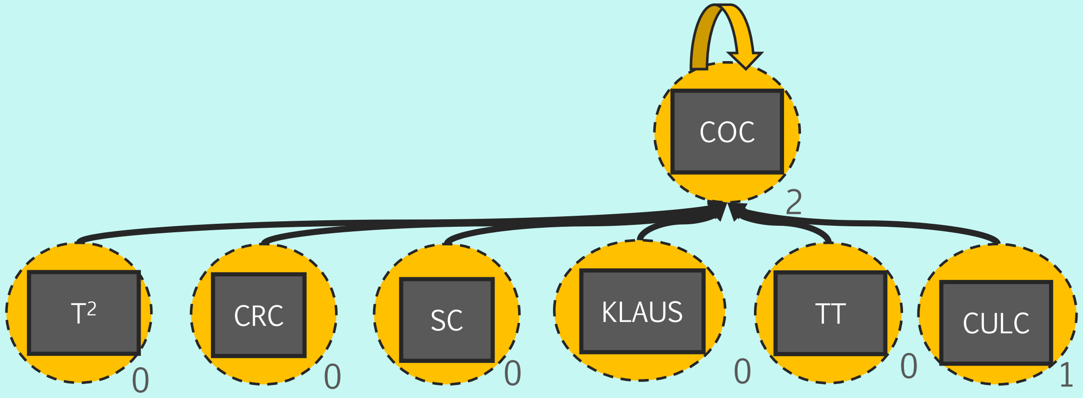




(KLAUS, T^2) is okay.

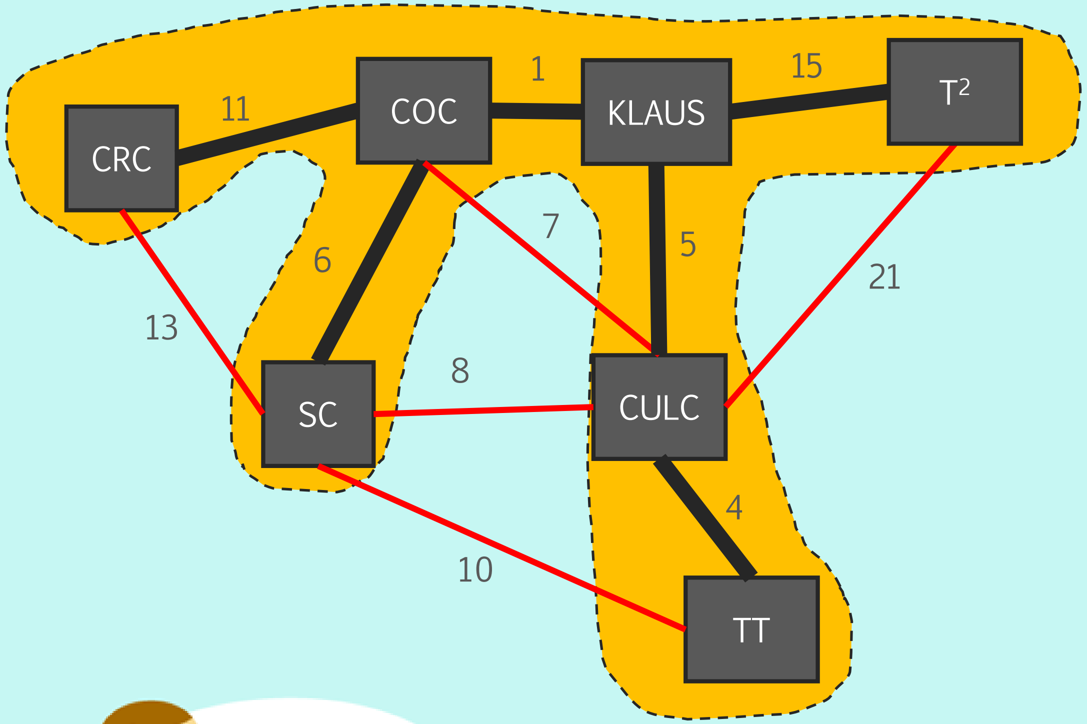
Kruskal's + Disjoint Set

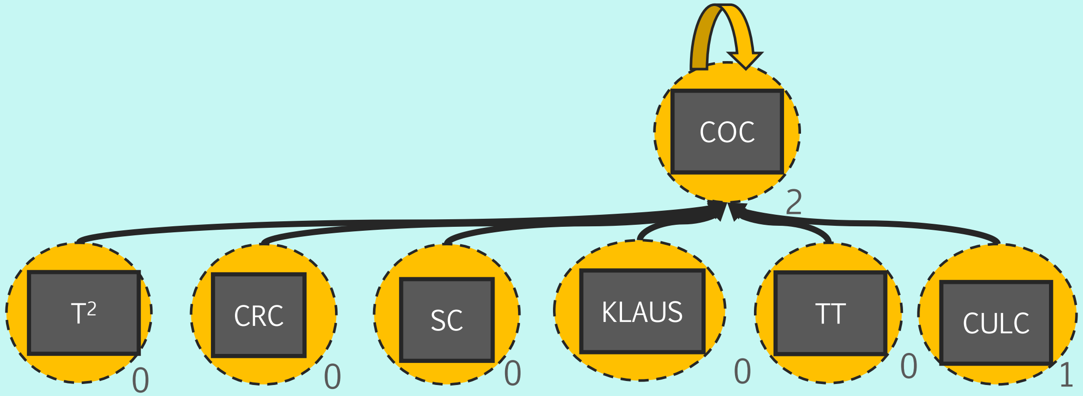




(CULC, T^2) is ignored.

Kruskal's + Disjoint Set





(CULC, T^2) is ignored.

We'll continue this until our spanning tree is |V| - 1 edges or until we exhaust all our edges.

Kruskal's Analysis

- Kruskal's Algorithm is $O(|E| \log |V|)$.
  - Having a Priority Queue of $|E|$ edges is $O(E \log E)$
    - You could also presort the list of edges in $O(E \log E)$.
  - For every edge, we perform Union() and Find(), and since these operations are $O(\alpha(V))$, we have $O(E\alpha(V))$.
    - $\alpha(n) = O(\log V) = O(\log E)$, so we end up with $O(E \log E)$.
  - So adding the priority queue operations and Disjoint Set operations, we have $O(2E\log E) = O(E\log E)$.
- Because $|E| < |V^2|$, we can change $O(E \log E)$ to $O(E \log V)$.

Prim's Algorithm

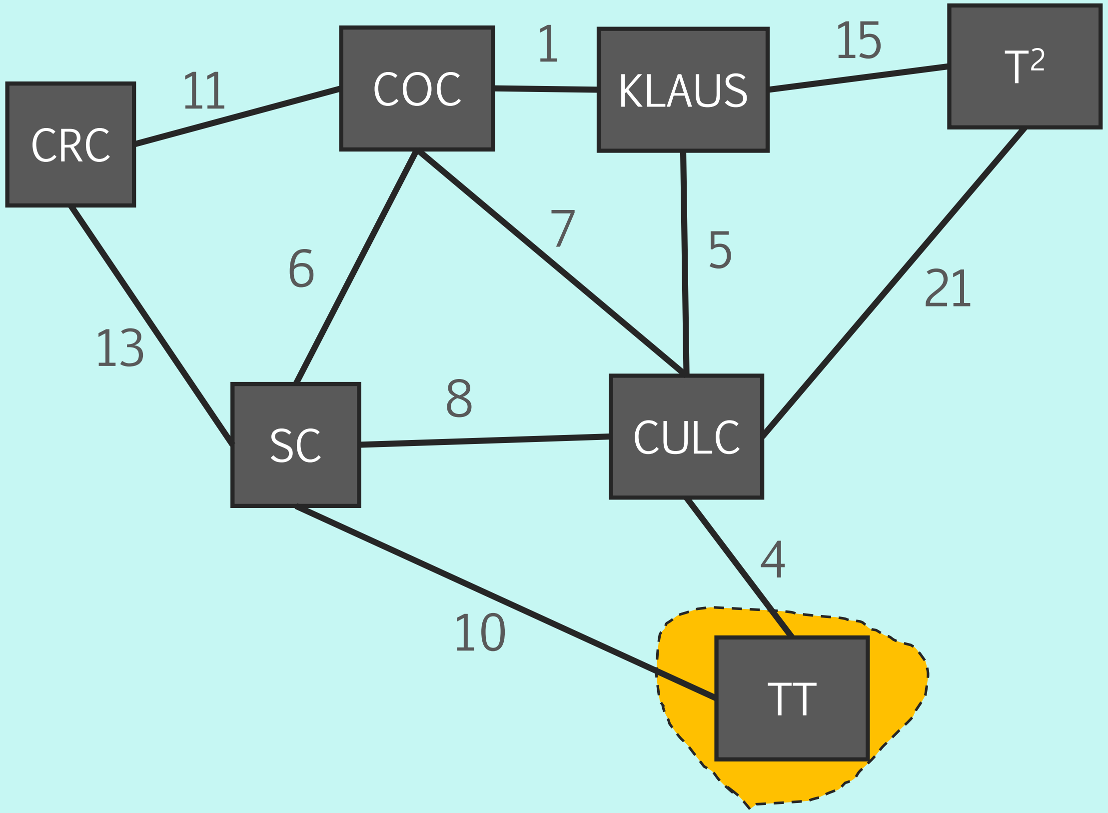
- Prim's is another MST finding algorithm.
- The behavior is similar to Dijkstra's Algorithm except the priority queue will hold edges instead of (Vertex, distance) tuples.
- Prim's begins with a starting vertex, and we branch to neighboring vertices over smallest edge weight. The edges we traverse over are part of our spanning tree.

Prim's Algorithm

```
Prims(G, start):
  visited = {start}
  spanning_tree = {}
  PriorityQueue PQ = {start.edges}
  loop while PQ isn't empty:
    currEdge = PQ.extract_min()    // (u, v) = (currVertex, destination)
    if visited contains edge.u and edge.v:
      continue
    spanning_tree.add(currEdge)
    for edge e of currEdge.v.edges:          // for all of v's edges,
      if visited doesn't contain e.v:
        PQ.add(e)
  return spanning_tree
```

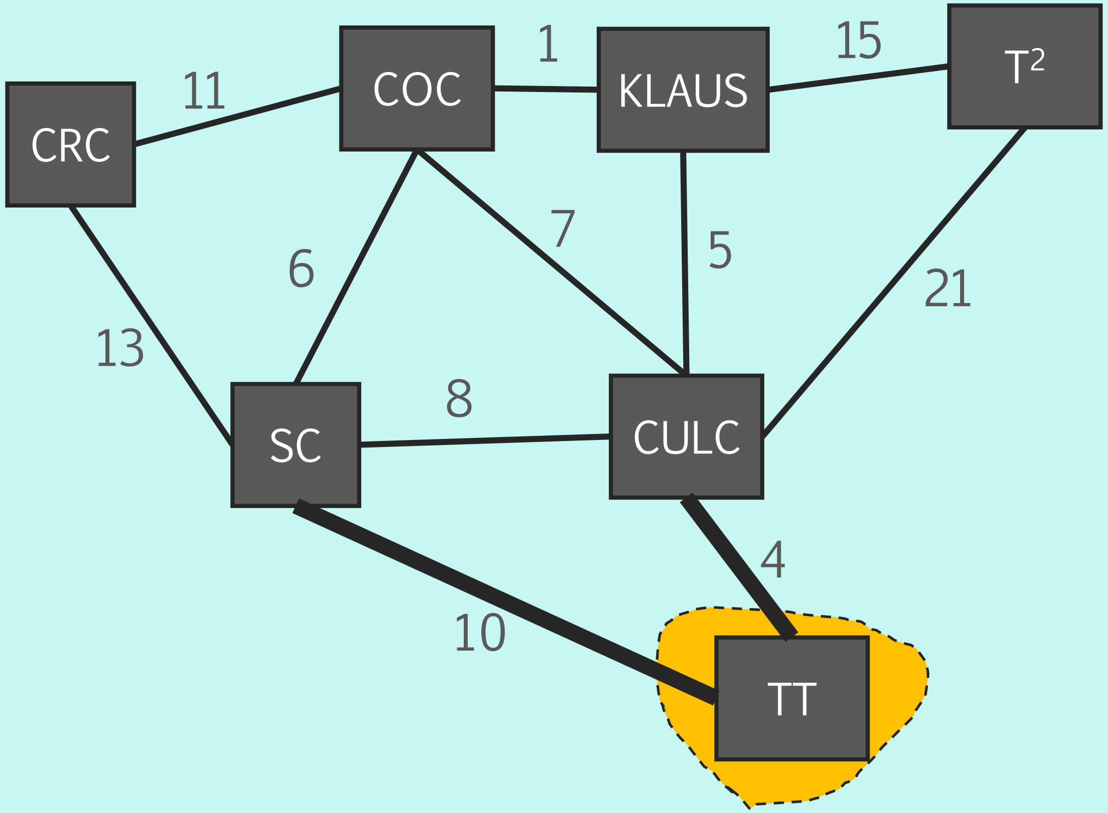
Prim's Examplme

- We call Prims(Graph, TT), so we start at TT.   Visited vertices are in the orange cloud.



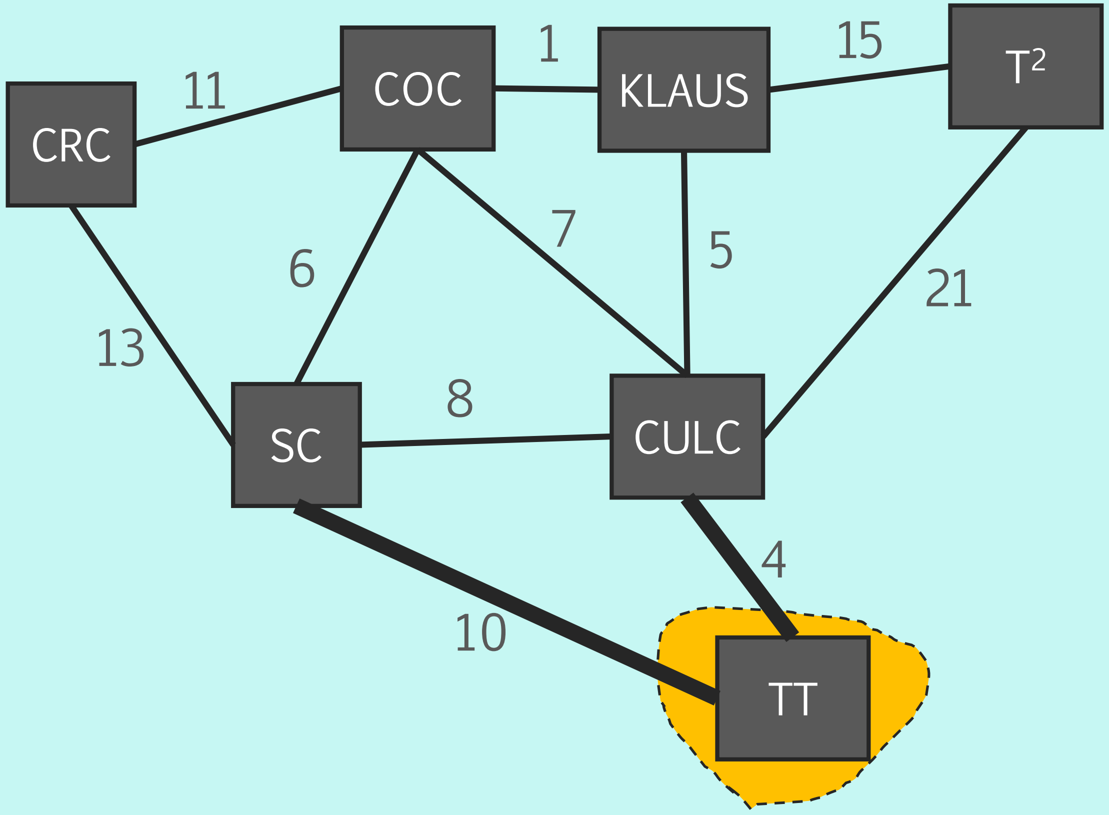
Prim's Examplme

- We call Prims(Graph, TT), so we start at TT. Visited vertices are in the orange cloud.
- We then look at all edges connected to TT and attempt to traverse the smallest edge.
  - In this case we have edge 10 and 4. We will attempt to traverse edge 4.



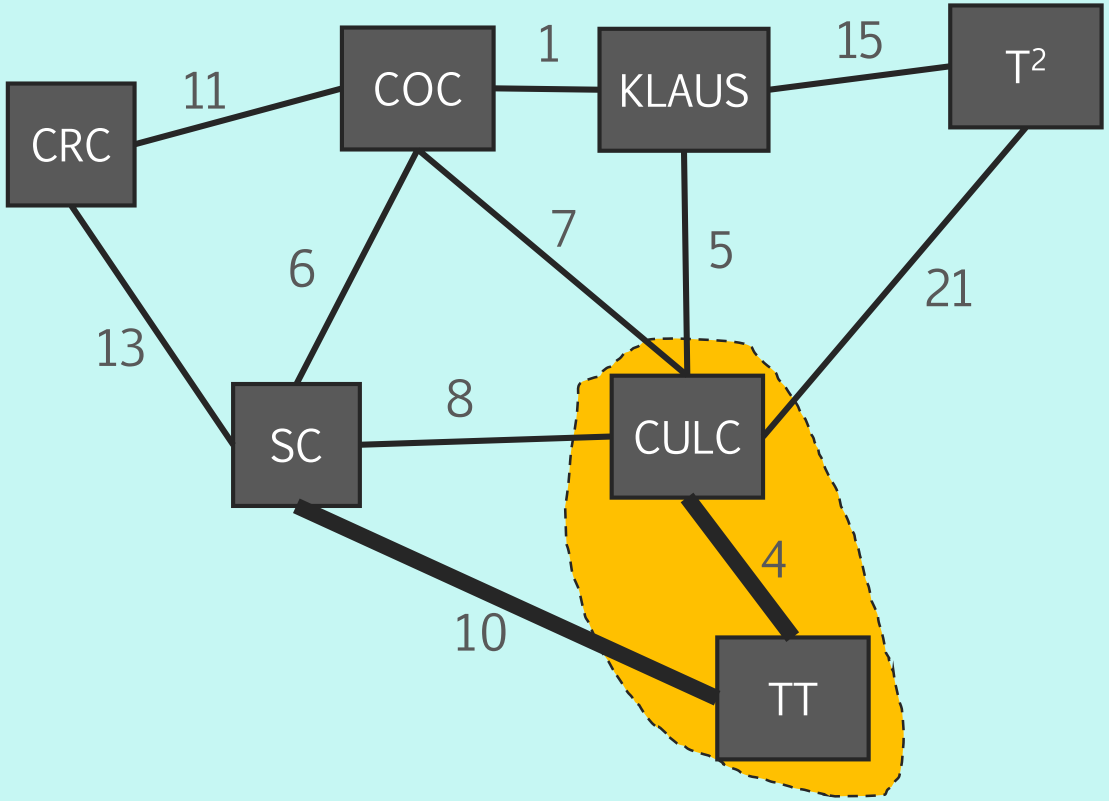
Prim’s Examplme

- We call Prims(Graph, TT), so we start at TT.   Visited vertices are in the orange cloud.
- We then look at all edges connected to TT and attempt to traverse the smallest edge.
  - In this case we have edge 10 and 4. We will attempt to traverse edge 4.
- We have not visited CULC, so we include edge 4 into our spanning tree.



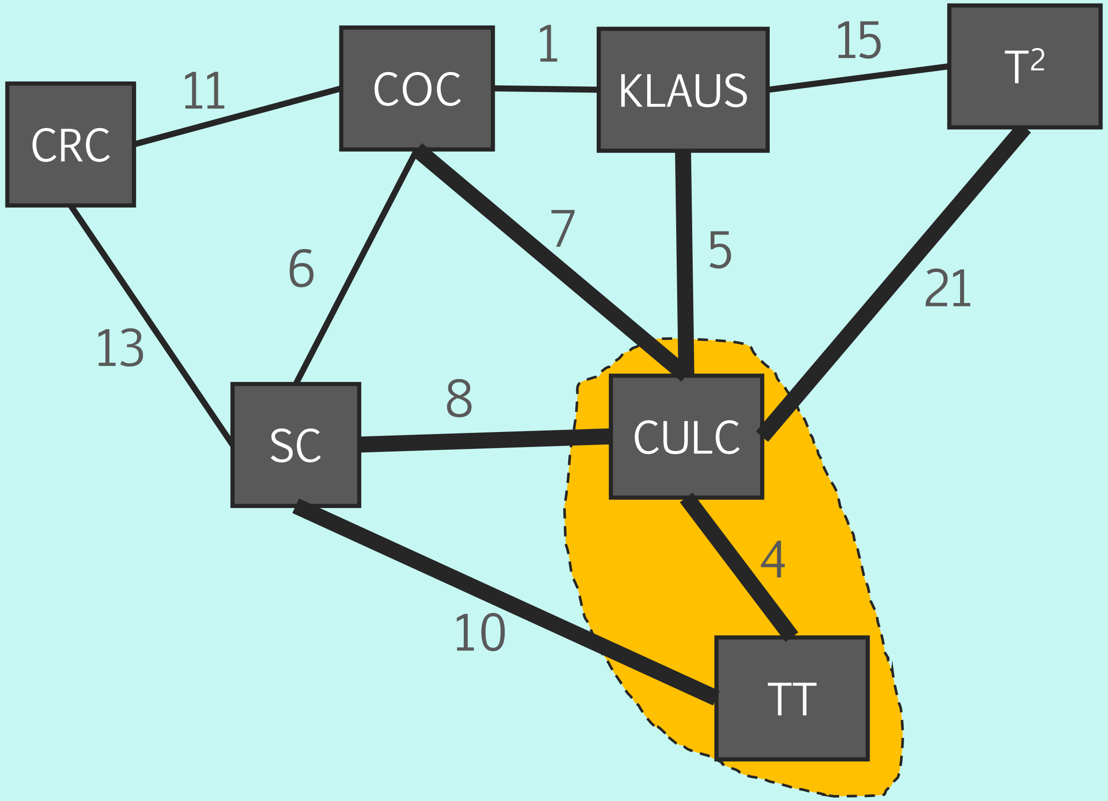
Prim’s Examplme

- We call Prims(Graph, TT), so we start at TT. Visited vertices are in the orange cloud.
- We then look at all edges connected to TT and attempt to traverse the smallest edge.
  - In this case we have edge 10 and 4. We will attempt to traverse edge 4.
- We have not visited CULC, so we include edge 4 into our spanning tree.



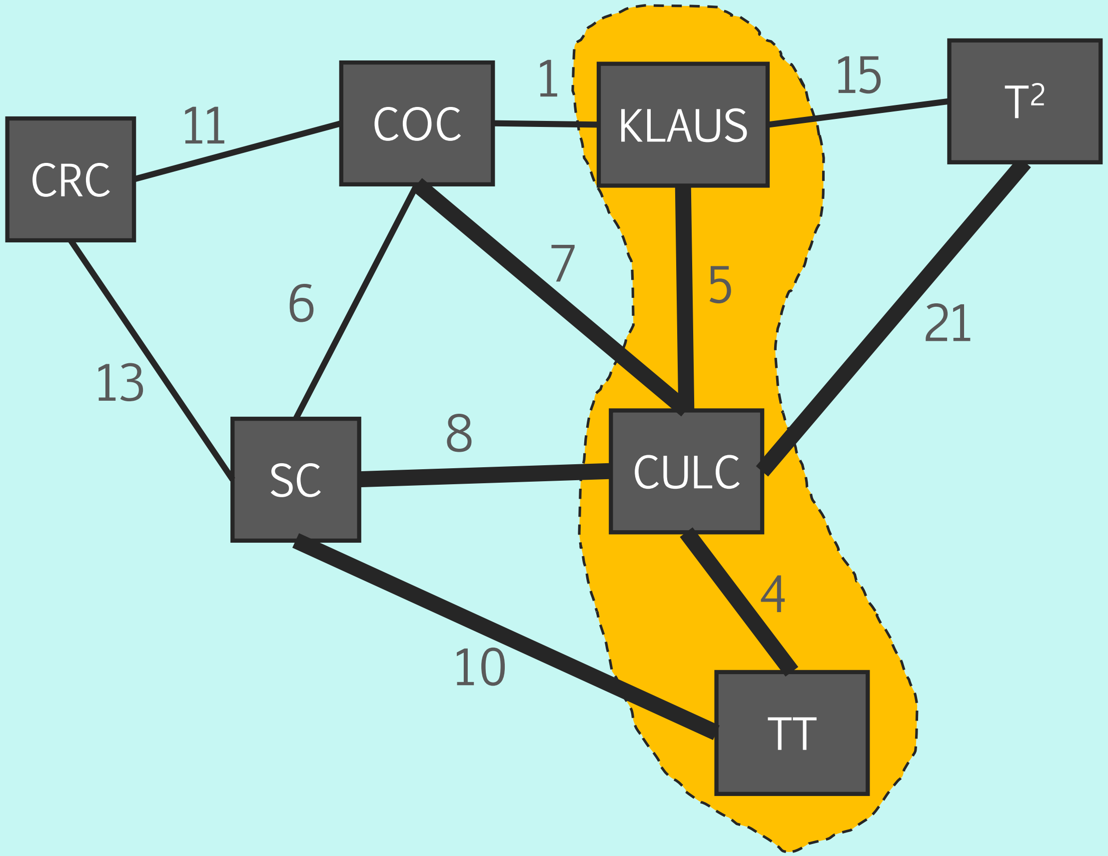
Prim’s Examplme

- Now we look at all edges connected to our cloud and attempt to traverse the smallest edge.
  - This is edge 5.



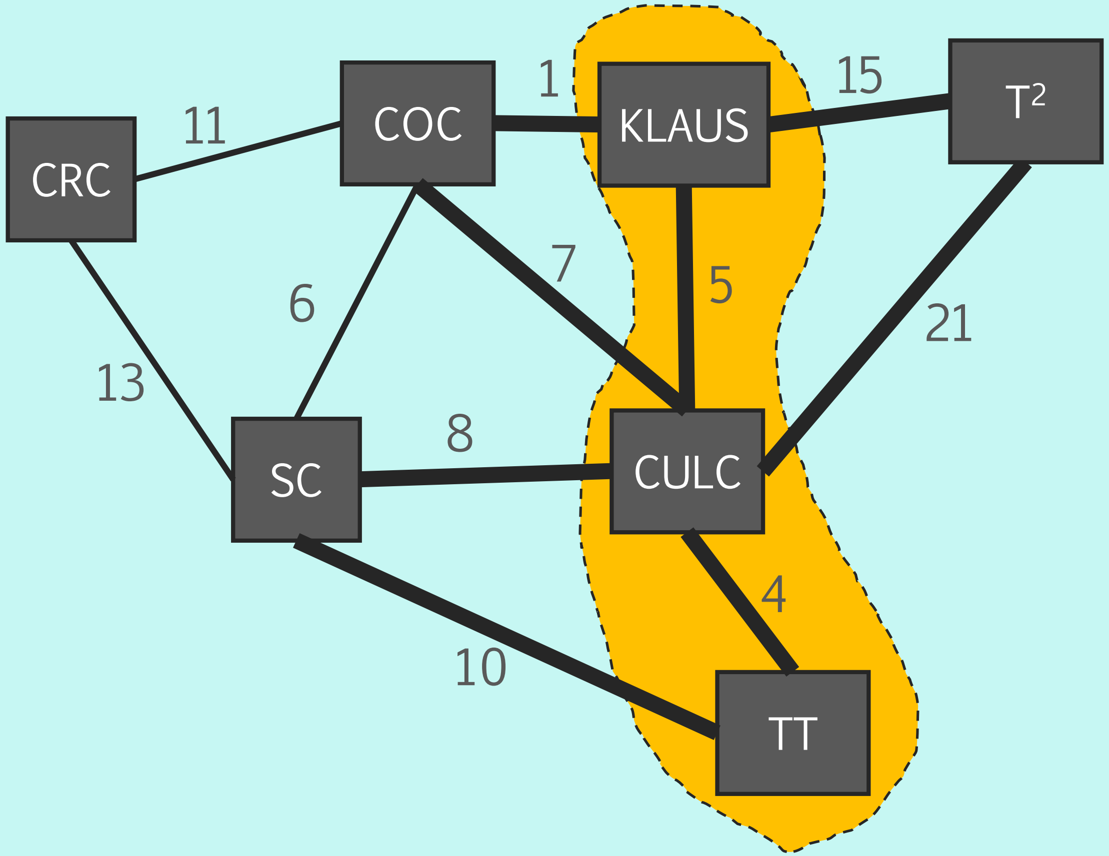
Prim's Examplme

- Now we look at all edges connected to our cloud and attempt to traverse the smallest edge.
  - This is edge 5.
- KLAUS has not been visited yet, so we can include edge 5 in our spanning tree.



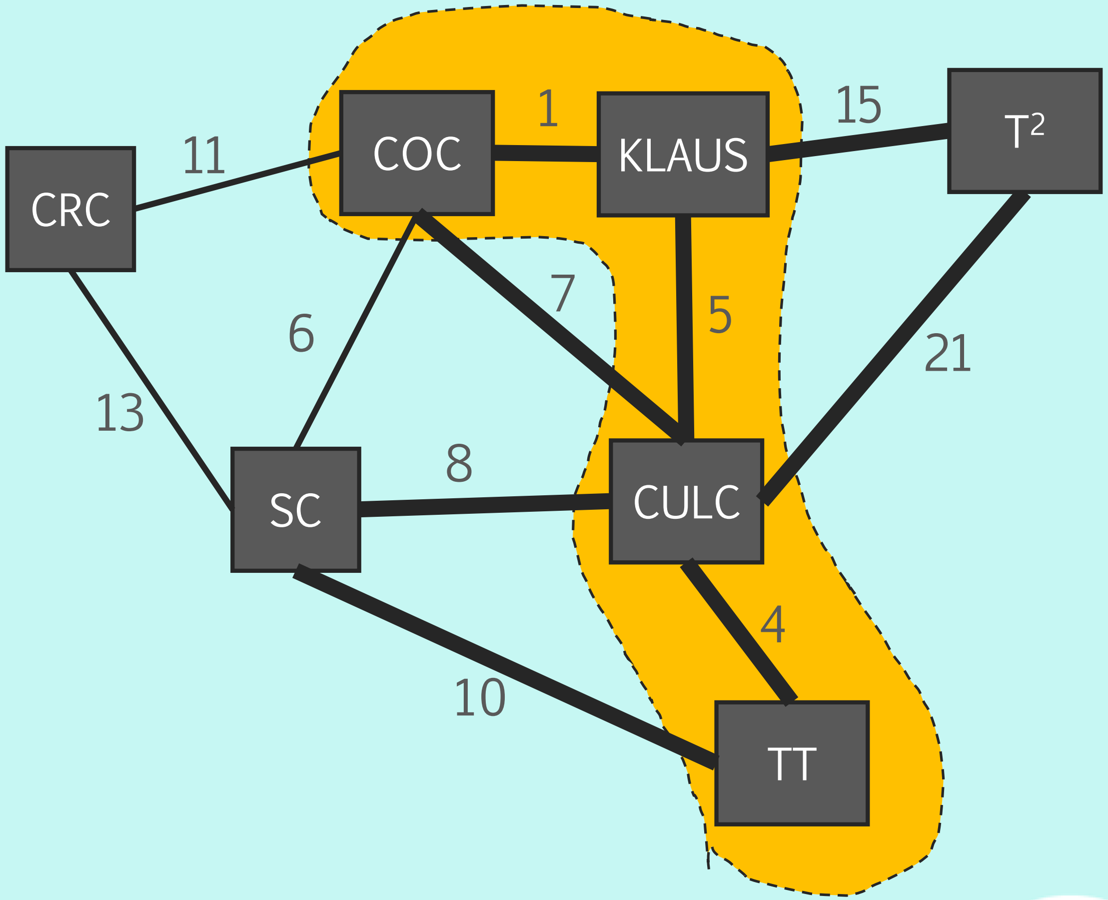
Prim’s Examplme

- Now we look at all edges connected to our cloud and attempt to traverse the smallest edge.
  - This is edge 1.



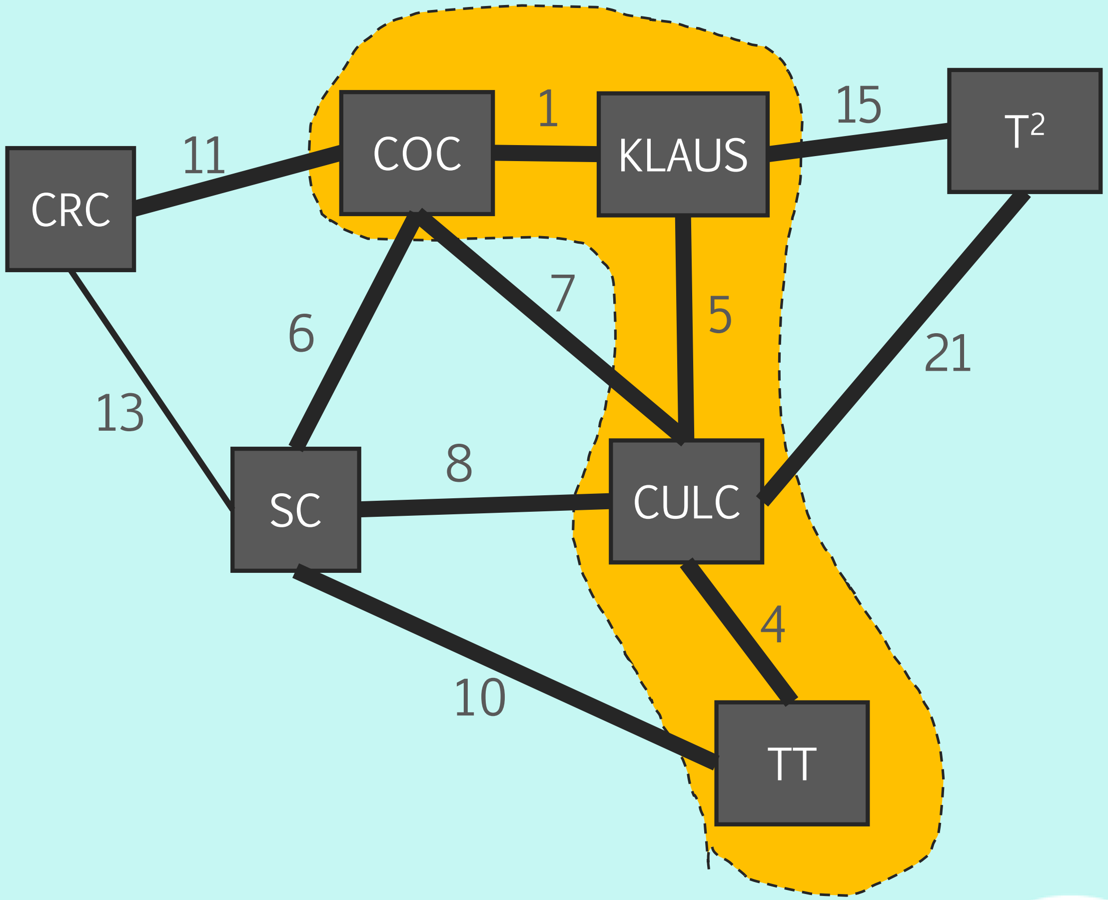
Prim's Examplme

- Now we look at all edges connected to our cloud and attempt to traverse the smallest edge.
  - This is edge 1.
- COC has not been visited yet, so we include edge 1 in our spanning tree.



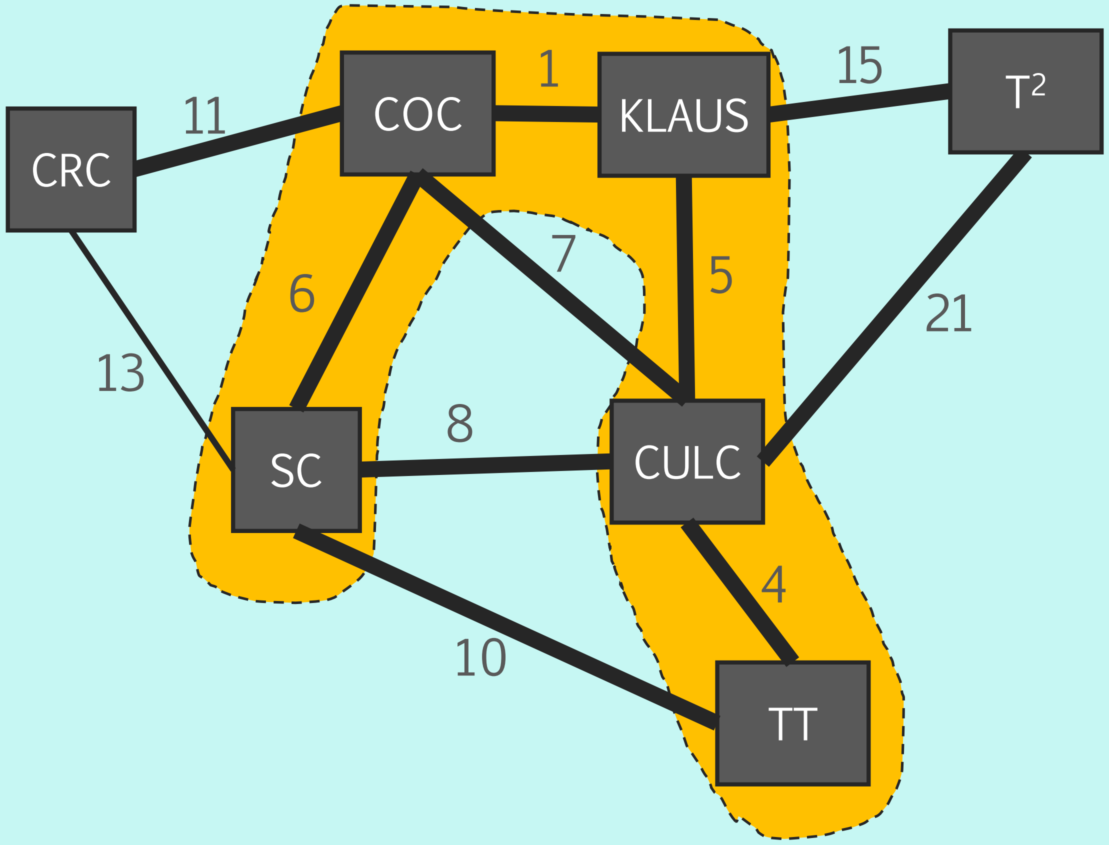
Prim's Examplme

- Now we look at all edges connected to our cloud and attempt to traverse the smallest edge.
  - This is edge 6.



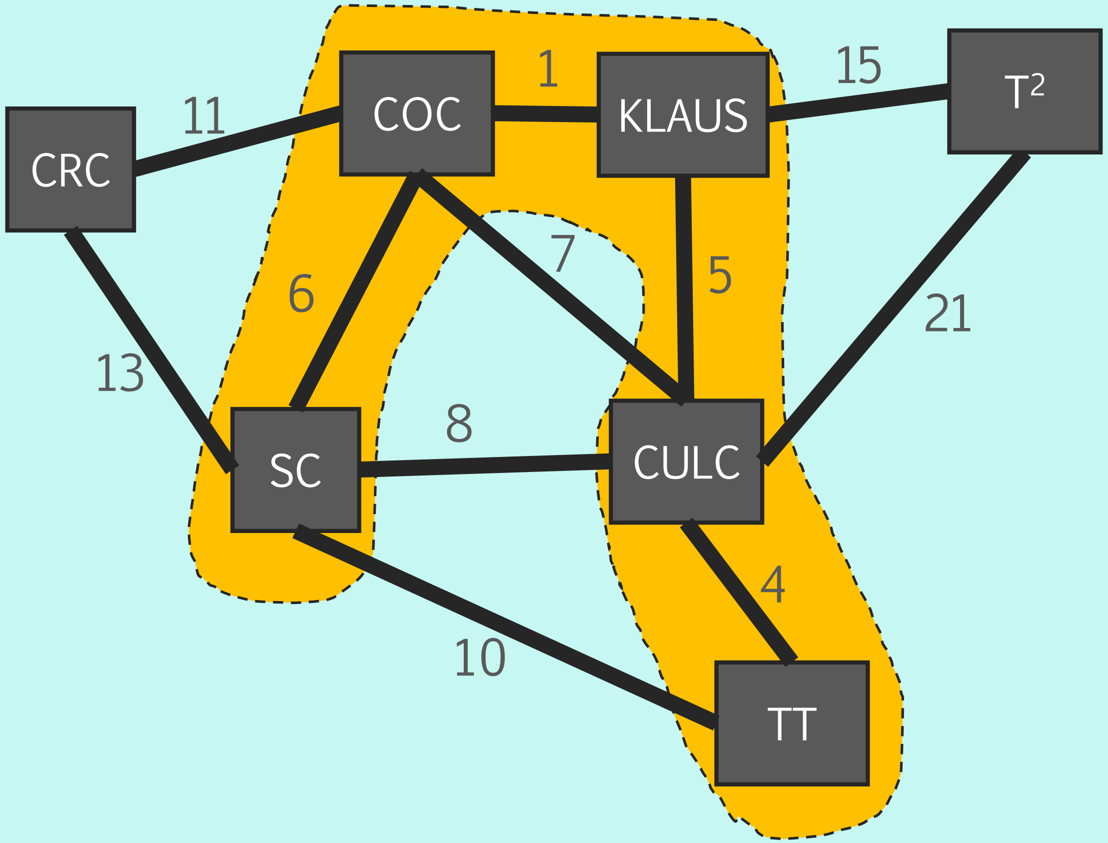
Prim’s Examplme

- Now we look at all edges connected to our cloud and attempt to traverse the smallest edge.
  - This is edge 6.
- SC has not been visited yet, so we can include edge 6 in our spanning tree.



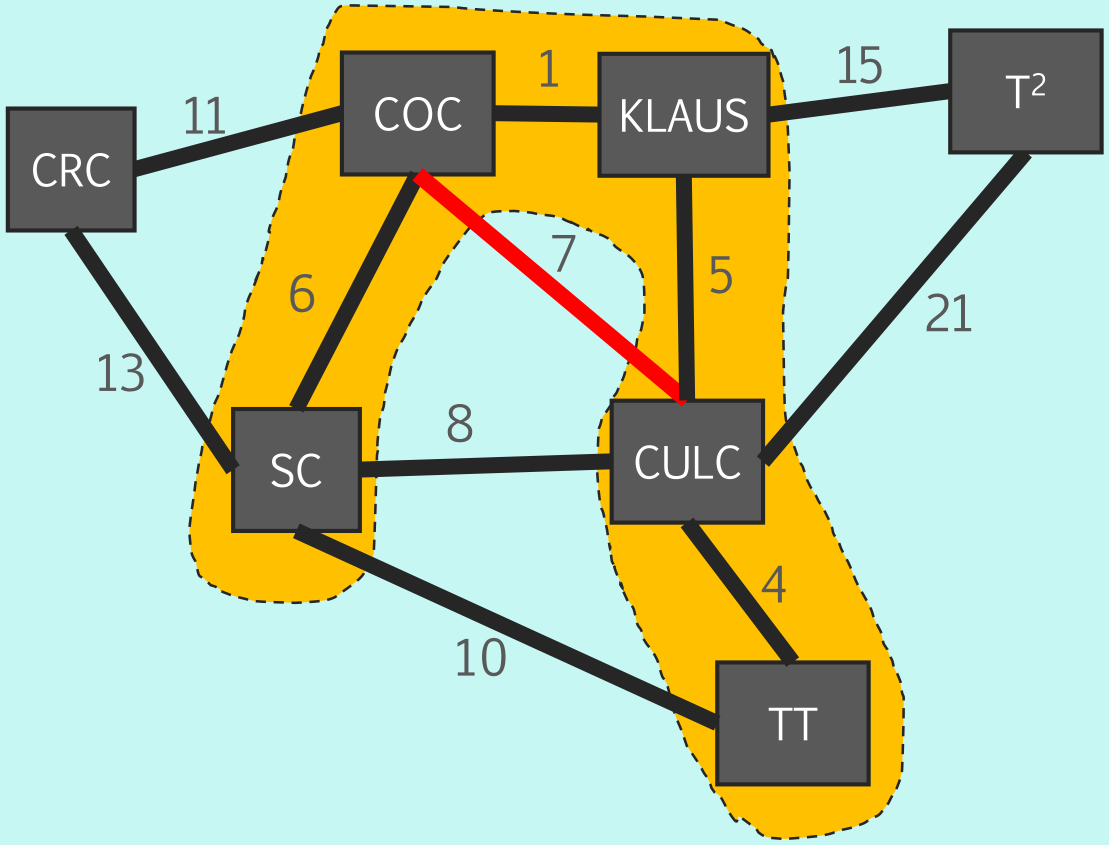
Prim’s Examplme

- Now we look at all edges connected to our cloud and attempt to traverse the smallest edge.
  - This is edge 7.



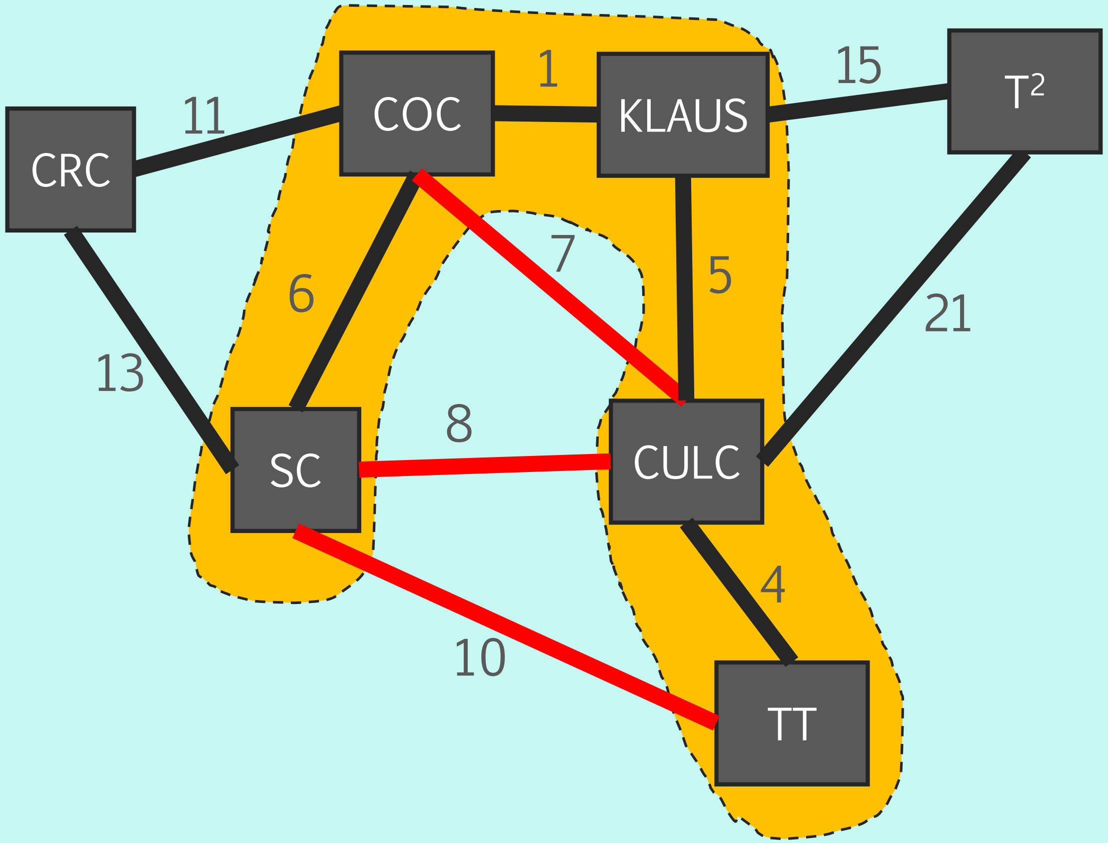
Prim’s Examplme

- Now we look at all edges connected to our cloud and attempt to traverse the smallest edge.
  - This is edge 7.
  - However, both vertices in edge 7 have already been visited, so we ignore this edge.



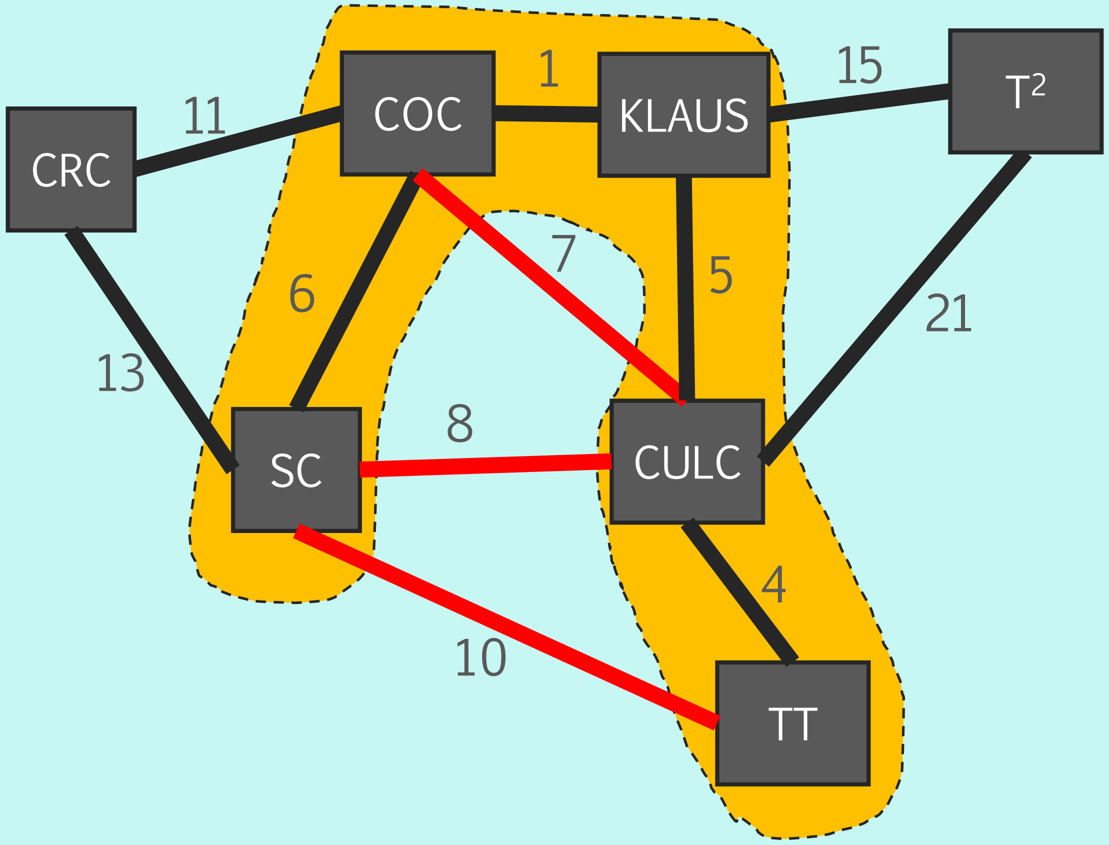
Prim's Examplme

- Now we look at all edges connected to our cloud and attempt to traverse the smallest edge.
  - This is edge 7.
  - However, both vertices in edge 7 have already been visited, so we ignore this edge.
- The same goes for edge 8, and edge 10.



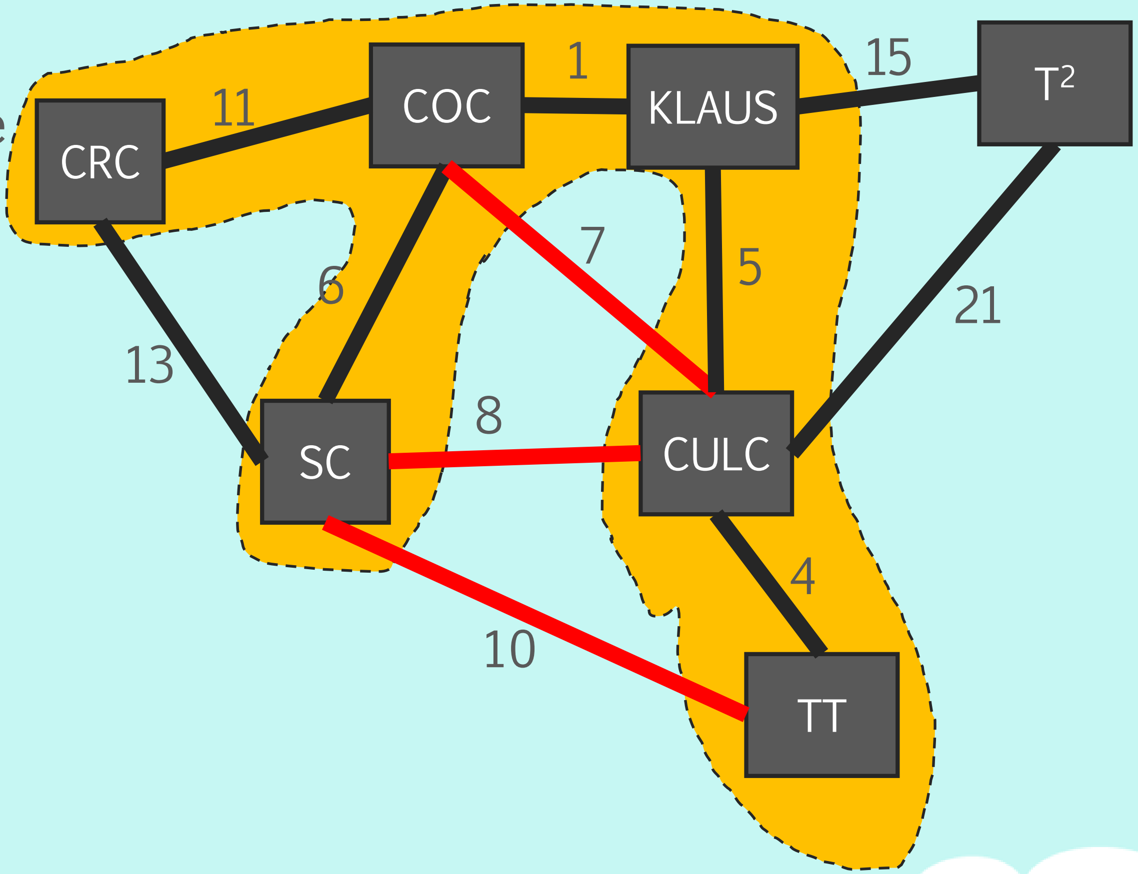
Prim's Examplme

- Now we look at all edges connected to our cloud and attempt to traverse the smallest edge.
  - This is edge 11.



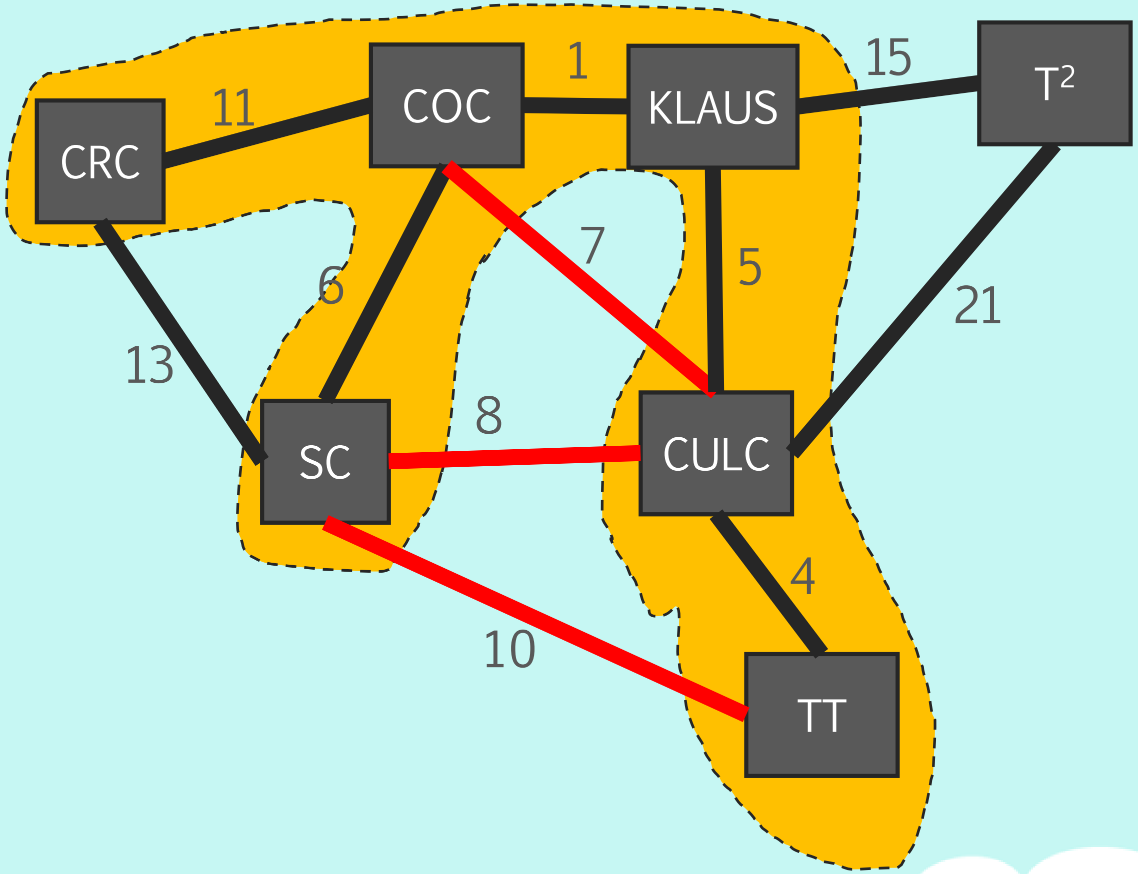
Prim’s Examplme

- Now we look at all edges connected to our cloud and attempt to traverse the smallest edge.
  - This is edge 11.
- CRC has not been visited yet, so we can include edge 11 in our spanning tree.



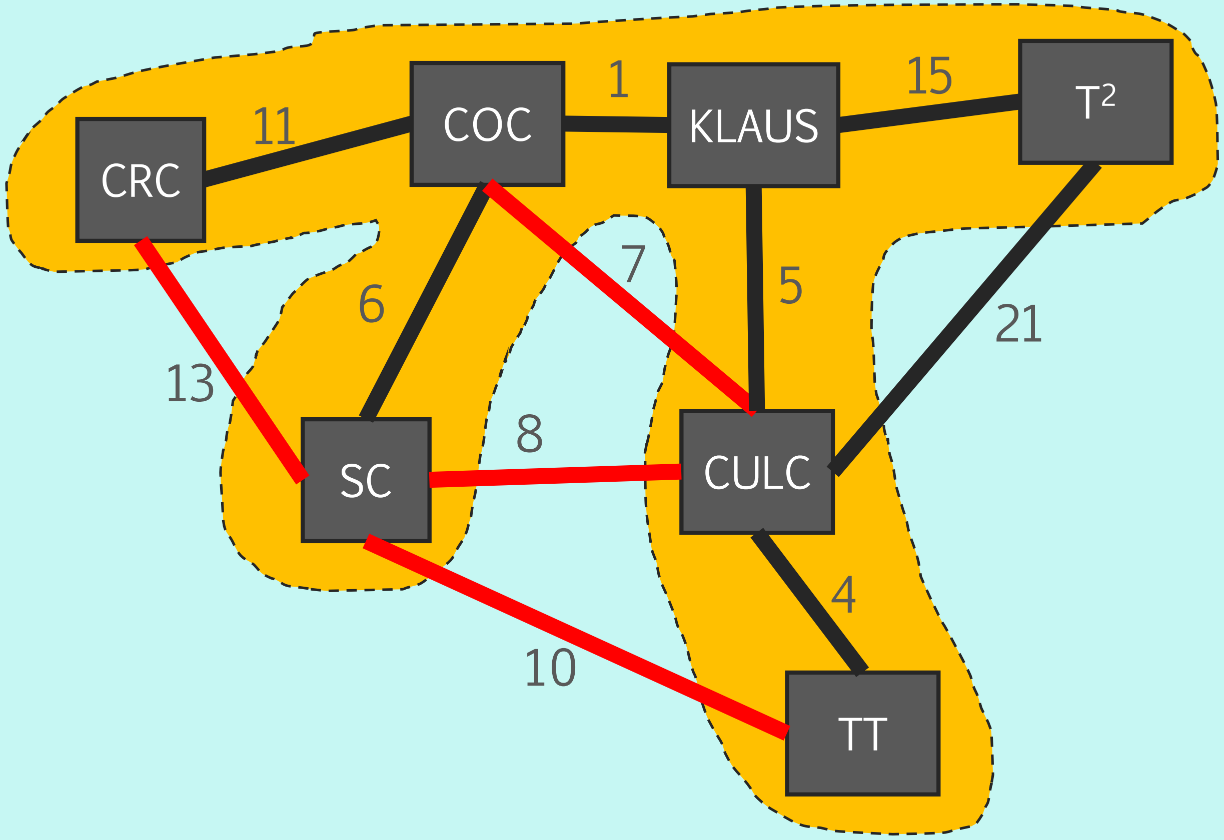
Prim's Examplme

- We skip edge 13.



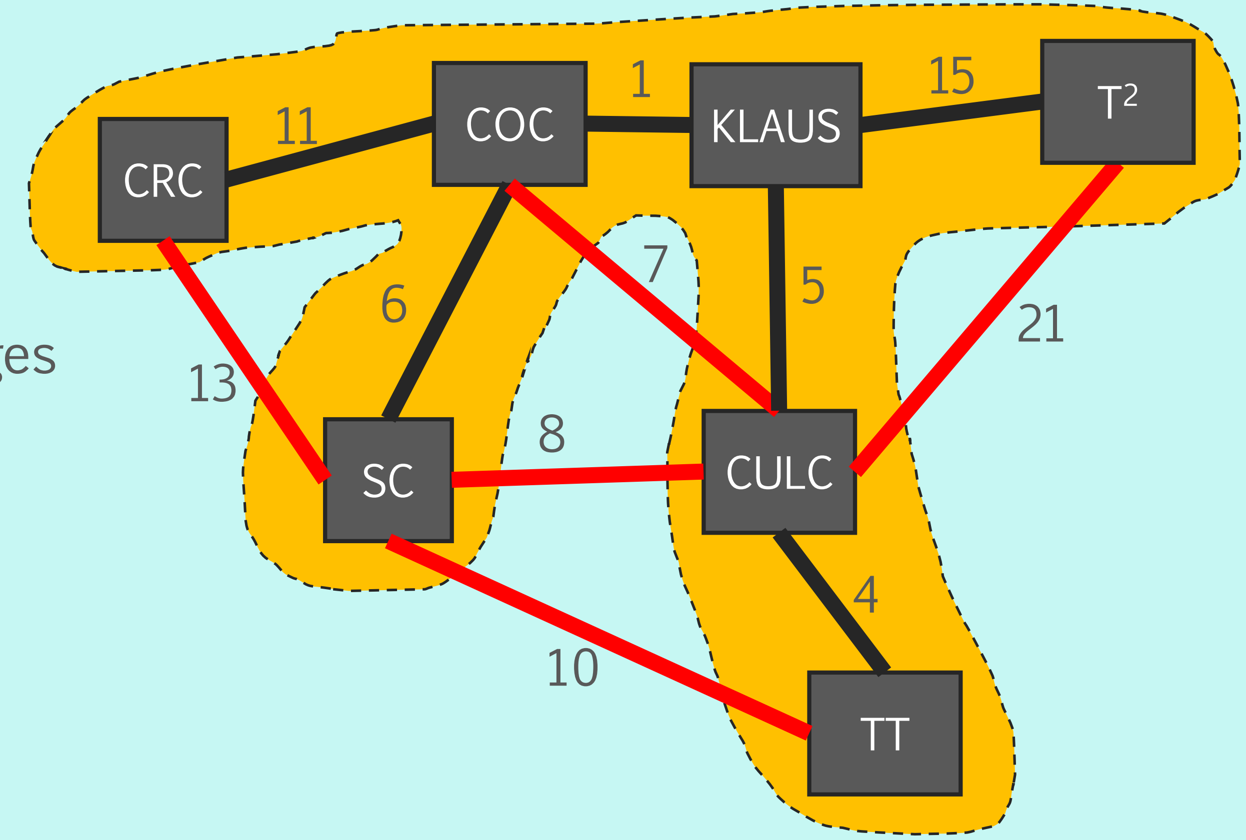
Prim's Examplme

- We skip edge 13.
- And we add edge 15.



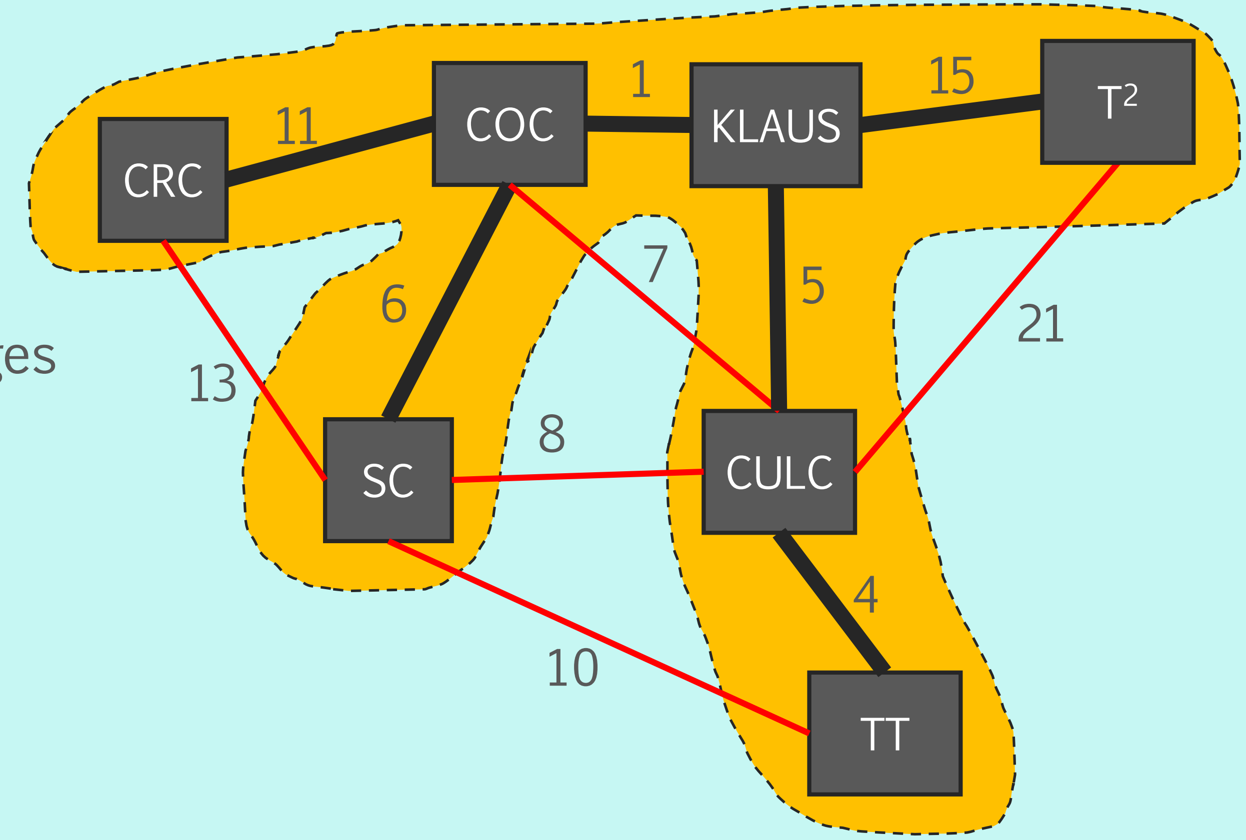
Prim's Examplme

- We skip edge 13.
- And we add edge 15.
- We can end once all our vertices have been visited, or when all edges have been looked over.



Prim’s Examplme

- We skip edge 13.
- And we add edge 15.
- We can end once all our vertices have been visited, or when all edges have been looked over.



Prim's Analysis

- Prim's runs in O(E log V).
  - The main loop runs in O(E) because our priority queue will include all edges.
  - extract_min() runs in O(log E), so we run O(E log E) extract_mins()'s.
  - The inner loop over all neighbors of a vertex runs in total 2|E| times. Our adjacency list will have each edge twice (u, v) and (v, u).
    - The inner loop will add a total of |E| edges into our priority queue, so this is O(E log E).
  - Assuming the graph is connected, $E < V^2$, so $|E|\log|E| = O(E \log V)$

TODO

- On your paper to turn in
  - What was something important that you learned
  - What do you have a question about?
- Also feedback form
  - Don’t write your name on it.