Lecture 22: Implementing Dijkstra’s

CSE 373: Data Structures and Algorithms

Administrivia

HW 5 Part 2 due Wednesday

Wednesday Review Session – How to Ace the Technical Interview

Dijkstra's Algorithm

Basic idea: ***Greedily*** pick the vertex with smallest distance, update other vertices distance based on choice, repeat until all vertices have been processed

(**Greedy** algorithms pick the locally optimal choice at each step and repeat to achieve a global solution)

Algorithm

1. Initialize all vertices initial distance from source. Set source's distance to 0 and all others to "∞"
2. For all unprocessed vertices
   A. Get the closest unvisited vertex, "current"
   B. Look at each of current's directly connected neighbors, "next"
      I. Calculate "newDistance" from current to next
      II. If newDistance is shorter than next's currently stored distance, update next's distance and predecessor
   C. Mark current as visited

Pseudocode

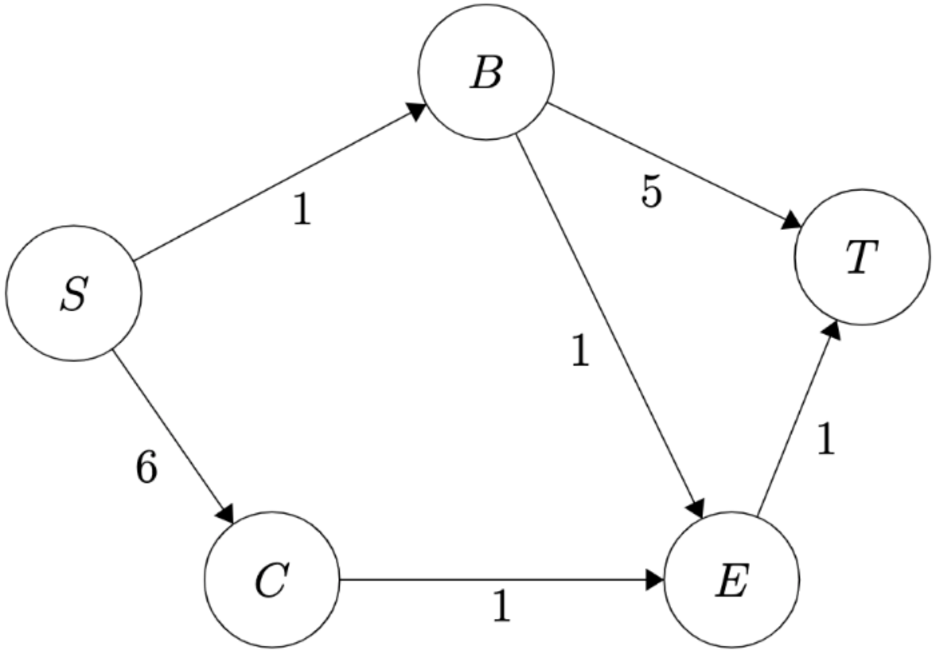
```
Dijkstra(Graph G, Vertex source)
    initialize distances to ∞
1.  mark all vertices unprocessed
    mark source as distance 0
2.  while(there are unprocessed vertices){
     A. let u be the closest unprocessed vertex
     B. for each(edge (u,v) leaving u){
        I. if(u.dist+weight(u,v) < v.dist){
                v.dist = u.dist+weight(u,v)
           II.
                v.predecessor = u
           }
        }
     C. mark u as processed
    }
```

Dijkstra's Run Through

Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark all vertices unprocessed
   mark source as distance 0
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      for each(edge (u,v) leaving u){
         if(u.dist+weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
S			
C			
B			
T			
E			

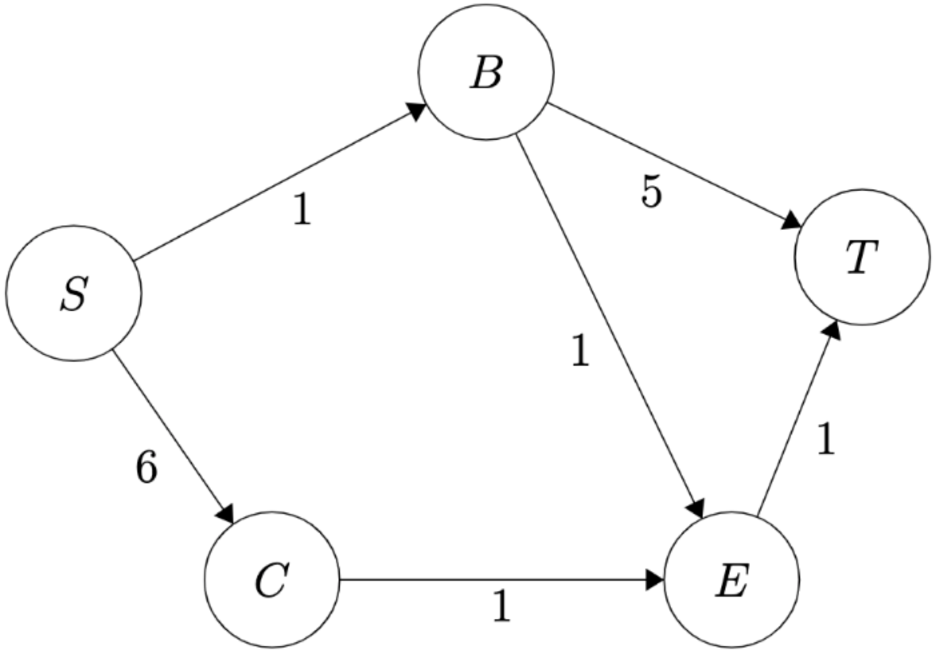


Dijkstra’s Run Through

Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark all vertices unprocessed
   mark source as distance 0
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      for each(edge (u,v) leaving u){
         if(u.dist+weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
S	0		No
C	∞		No
B	∞		No
T	∞		No
E	∞		No

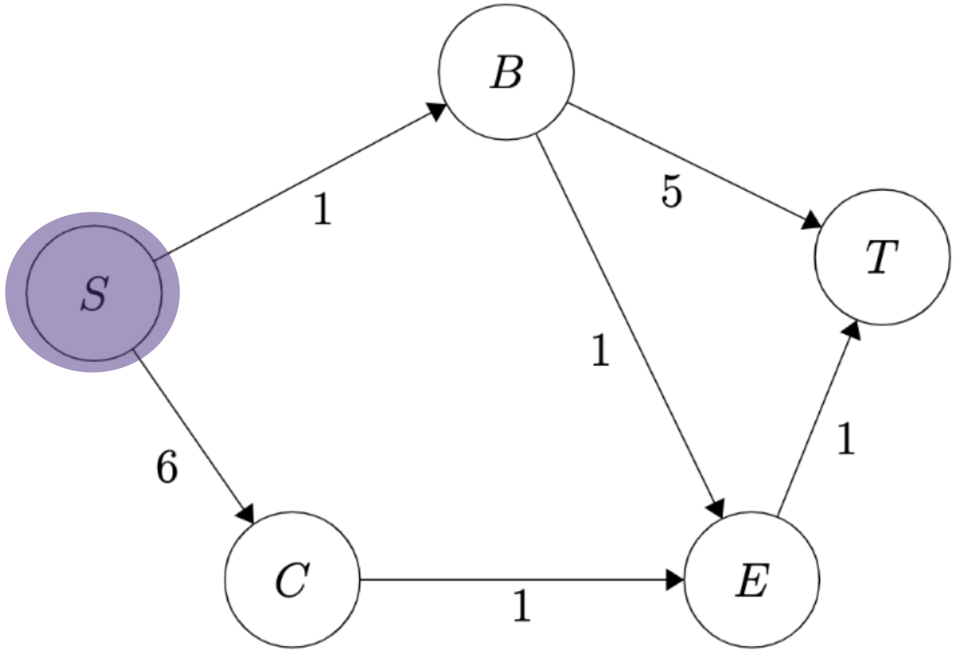


Dijkstra’s Run Through

Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark all vertices unprocessed
   mark source as distance 0
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      for each(edge (u,v) leaving u){
         if(u.dist+weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
S	0	--	No
C	6	S	No
B	1	S	No
T	∞		No
E	∞		No

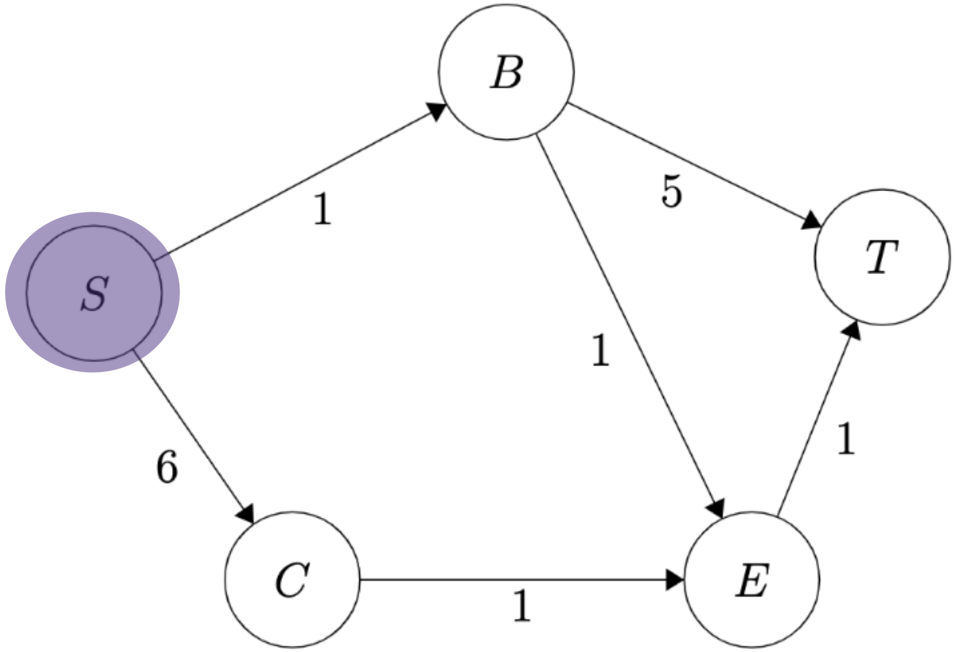


Dijkstra’s Run Through

Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark all vertices unprocessed
   mark source as distance 0
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      for each(edge (u,v) leaving u){
         if(u.dist+weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
S	0	--	Yes
C	6	S	No
B	1	S	No
T	∞		No
E	∞		No

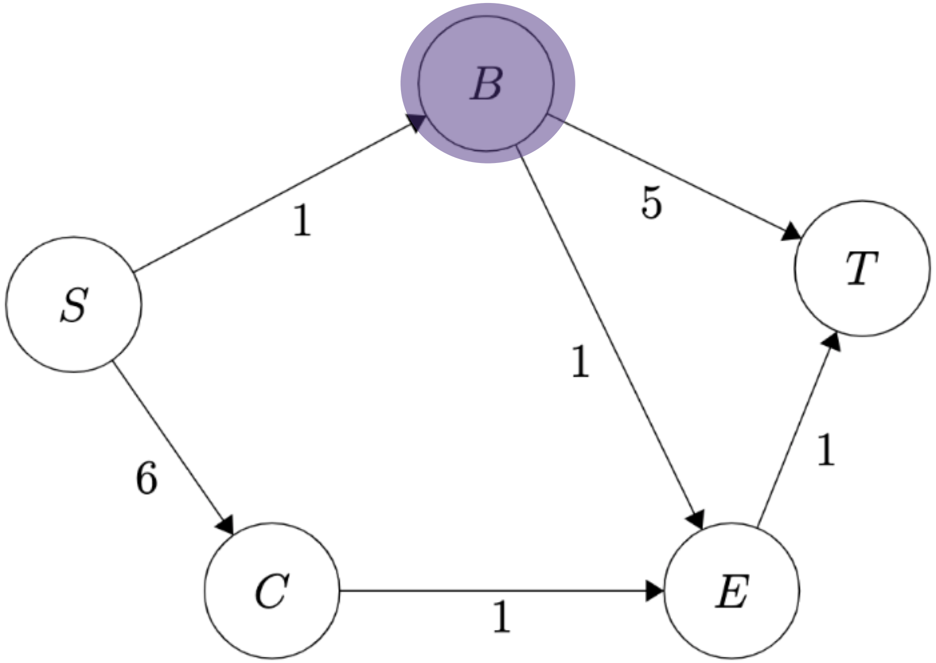


Dijkstra's Run Through

Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark all vertices unprocessed
   mark source as distance 0
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      for each(edge (u,v) leaving u){
         if(u.dist+weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
S	0	--	Yes
C	6	S	No
B	1	S	Yes
T	6	B	No
E	2	B	No

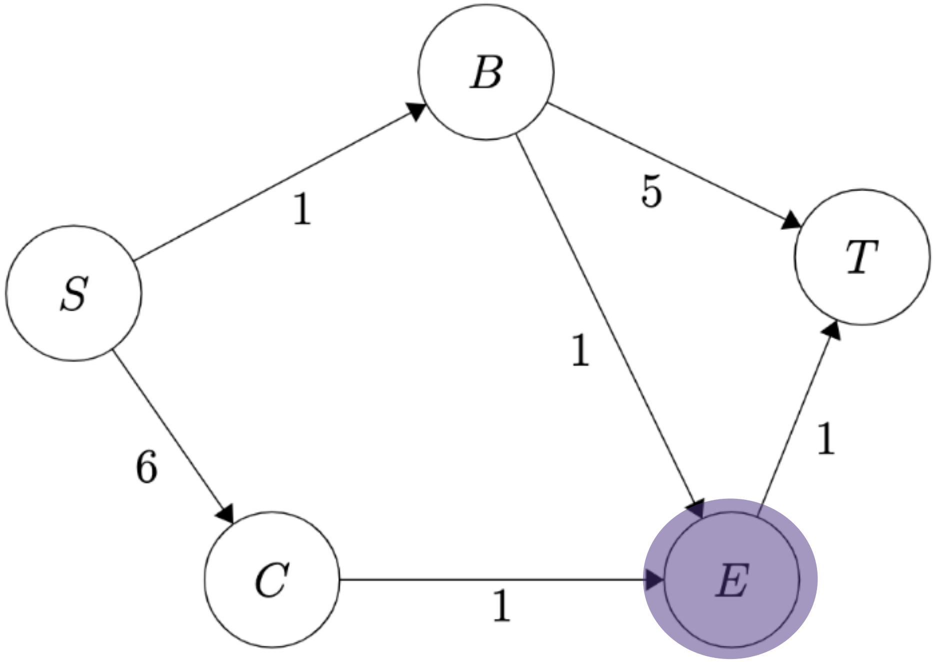


Dijkstra’s Run Through

Vertex	Distance	Predecessor	Processed
S	0	--	Yes
C	6	S	No
B	1	S	Yes
T	~~6~~ 3	E	No
E	2	B	Yes

Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark all vertices unprocessed
   mark source as distance 0
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      for each(edge (u,v) leaving u){
         if(u.dist+weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

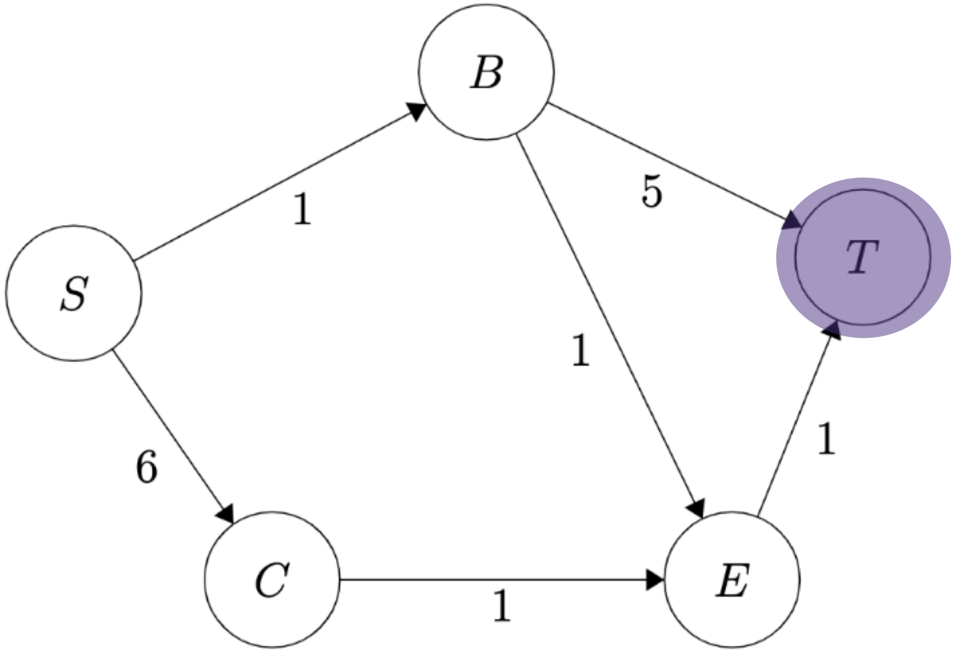


Dijkstra's Run Through

Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark all vertices unprocessed
   mark source as distance 0
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      for each(edge (u,v) leaving u){
         if(u.dist+weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
S	0	--	Yes
C	6	S	No
B	1	S	Yes
T	~~6~~ 3	E	Yes
E	2	B	Yes



Dijkstra’s Pseuodocode

```
Dijkstra(Graph G, Vertex source)
  initialize distances to ∞
  mark source as distance 0
  mark all vertices unprocessed
  while(there are unprocessed vertices){
    let u be the closest unprocessed vertex
    foreach(edge (u,v) leaving u){
      if(u.dist+weight(u,v) < v.dist){
        v.dist = u.dist+weight(u,v)
        v.predecessor = u
      }
    }
    mark u as processed
  }
```

Wut?

Min Priority Queue ADT

state

Set of comparable values - Ordered by “priority”

behavior

peek() – find the element with the smallest priority

insert(value) – add new element to collection

removeMin() – returns and removes element with the smallest priority

Dijkstra's Pseudocode

```
Dijkstra(Graph G, Vertex source)
  initialize distances to ∞
  mark source as distance 0
  mark all vertices unprocessed
  initialize MPQ as a Min Priority Queue, add source
  while(there are unprocessed vertices){
    u = MPQ.removeMin();
    foreach(edge (u,v) leaving u){
      if(u.dist+weight(u,v) < v.dist){
        v.dist = u.dist+weight(u,v)
        v.predecessor = u
      }
    }
    mark u as processed
  }
```

How?

Min Priority Queue ADT

state

Set of comparable values - Ordered by "priority"

behavior

peek() – find the element with the smallest priority

insert(value) – add new element to collection

removeMin() – returns and removes element with the smallest priority

Dijkstra's Pseudocode

```
Dijkstra(Graph G, Vertex source)
  initialize distances to ∞    ⎫
                               ⎬ How?
  mark source as distance 0    ⎭
  initialize MPQ as a Min Priority Queue, add source
  while(MPQ is not empty){
    u = MPQ.removeMin();
    foreach(edge (u,v) leaving u){  <------------ Wut?
      oldDist = v.dist; newDist = u.dist+weight(u,v)
      if(newDist < oldDist){
        v.dist = newDist          ⎫
                                  ⎬ Huh?
        v.predecessor = u         ⎭
        if(oldDist == INFINITY) { MPQ.insert(v) }
        else { MPQ.updatePriority(v, newDist) }
      }
    }
```

Min Priority Queue ADT

state

Set of comparable values - Ordered by "priority"

behavior

peek() – find the element with the smallest priority

insert(value) – add new element to collection

removeMin() – returns and removes element with the smallest priority

decreaseKey(e, p) – decreases priority of element e down to p

Dijkstra’s Pseuodocode

```
Dijkstra(Graph G, Vertex source)
    for (Vertex v : G.getVertices()) { v.dist = INFINITY; }
    G.getVertex(source).dist = 0;
    initialize MPQ as a Min Priority Queue, add source
    while(MPQ is not empty){
        u = MPQ.removeMin();
        for (Edge e : u.getEdges(u)){
            oldDist = v.dist; newDist = u.dist+weight(u,v)
            if(newDist < oldDist){
                v.dist = newDist
                v.predecessor = u
                if(oldDist == INFINITY) { MPQ.insert(v) }
                else { MPQ.updatePriority(v, newDist) }
            }
        }
```

Vertex<E>

state

data

dist

predecessor

behavior

…

Edge<E>

state

vertex1

vertex2

cost

behavior

…

AdjacencyListGraph<V, E>

state

Dictionary<V, Set<E>> graph

behavior

getEdges(v) – return set of outgoing edges from given vertex

getVertices() – return keyset of graph

getVertex(value) – return Vertex with given value stored

…

Dijkstra's Runtime

```
Dijkstra(Graph G, Vertex source)
  for (Vertex v : G.getVertices()) { v.dist = INFINITY; }
  G.getVertex(source).dist = 0;
  initialize MPQ as a Min Priority Queue, add source
  while(MPQ is not empty){
    u = MPQ.removeMin();
    for (Edge e : u.getEdges(u)){
      oldDist = v.dist; newDist = u.dist+weight(u,v)
      if(newDist < oldDist){
        v.dist = newDist
        v.predecessor = u
        if(oldDist == INFINITY) { MPQ.insert(v) }
        else { MPQ.updatePriority(v, newDist) }
      }
    }
  }
```

+V

$+C_1$

+logV

+V

+E of 1 V

$+C_2$

+logV

+?

(assume logV)

Code Model = $C_1 + V + V(\log V + E(C_2 + 2\log V))$

$= C_1 + V + V\log V + VEC_2 + VEC_3\log V$

O Bound = O(VElogV)

This actually doesn't run all E times – for every iteration of the outer loop. It actually will run E times in total; if every vertex is only removed from the priority queue (processed) once, then we will examine each edge once. So each line inside this foreach gets multiplied by a single E instead of E * V.

Tight O Bound = O(VlogV + ElogV)

More Dijkstra’s Implementation

How do we keep track of vertex costs?

- Create a vertex object with a cost field
- Store a dictionary that maps vertices to costs

How do we find vertex with smallest distance?

- Loop over dictionary of costs to find smallest
- Use a min heap with priority based on distance

How do we keep track of shortest paths?

- Create a vertex object with a predecessor field, update while running Dijkstra’s update fields
- While running Dijkstra’s build dictionary of vertix to edge backpointers

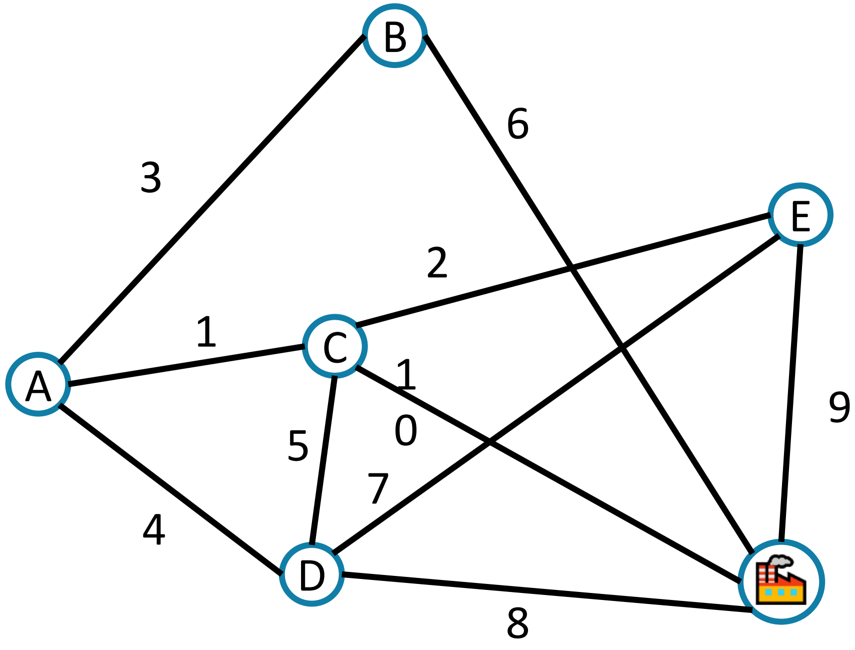
Find shortest path from A to B

- Run Dijkstra’s, navigate backpointers from B to A

Minimum Spanning Trees

Minimum Spanning Trees

It's the 1920's. Your friend at the electric company needs to choose where to build wires to connect all these cities to the plant.



She knows how much it would cost to lay electric wires between any pair of locations, and wants the cheapest way to make sure electricity from the plant to every city.

Minimum Spanning Trees

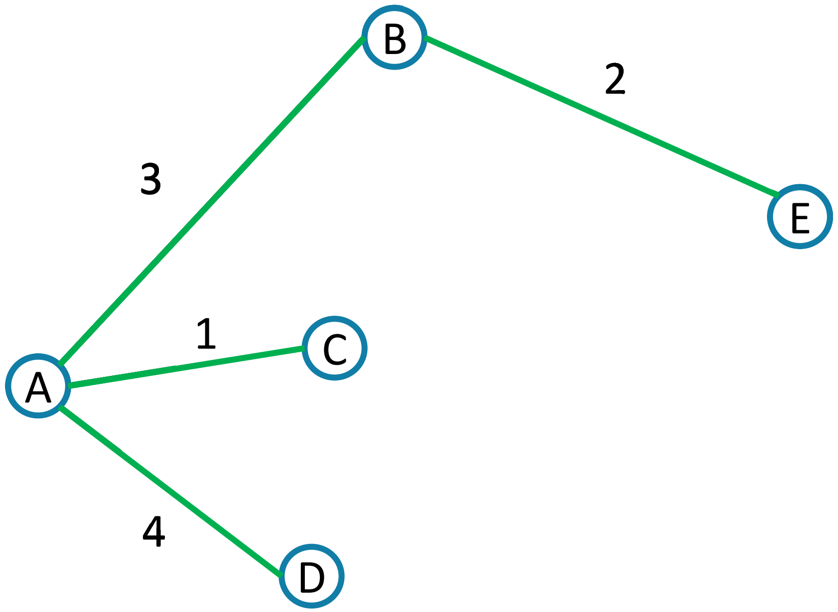
What do we need? A set of edges such that:

- Every vertex touches at least one of the edges. (the edges **span** the graph)
- The graph on just those edges is **connected**.
- The minimum weight set of edges that meet those conditions.

Notice we do not need a directed graph!

Assume all edge weights are positive.

Claim: The set of edges we pick never has a cycle. Why?



Aside: Trees

Our BSTs had:

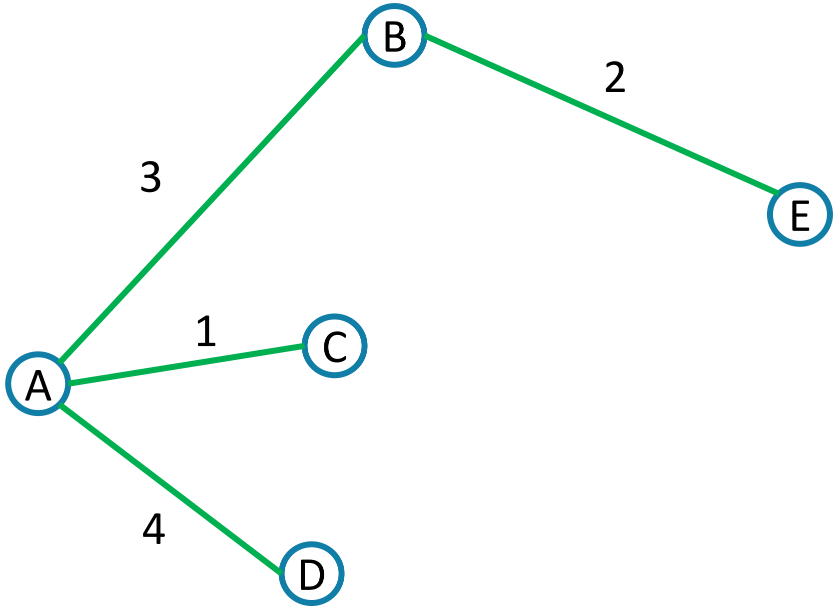
- A root
- Left and/or right children
- Connected and no cycles

Our heaps had:

- A root
- Varying numbers of children
- Connected and no cycles

On graphs our tees:

- Don’t need a root (the vertices aren’t ordered, and we can start BFS from anywhere)
- Varying numbers of children
- Connected and no cycles



Tree (when talking about graphs)

An undirected, connected acyclic graph.

MST Problem

What do we need? A set of edges such that:

- Every vertex touches at least one of the edges. (the edges **span** the graph)
- The graph on just those edges is **connected**.
- The minimum weight set of edges that meet those conditions.

Our goal is a tree!

Minimum Spanning Tree Problem

Given: an undirected, weighted graph G
Find: A minimum-weight set of edges such that you can get from any vertex of G to any other on only those edges.

We'll go through two different algorithms for this problem today.

Example

Try to find an MST of this graph:

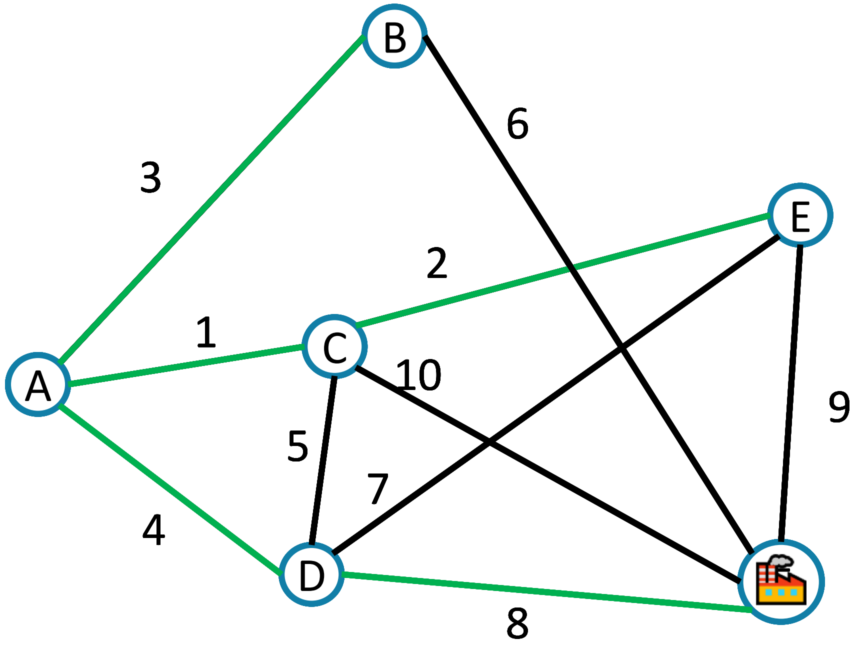
Graph Algorithm Toolbox

BFS/DFS

1. Pick an arbitrary starting point
2. Queue up unprocessed neighbors
3. Process next neighbor in queue
4. Repeat until all vertices in queue have been processed

Dijkstra's

1. Start at source
2. Update distance from current to unprocessed neighbors
3. Process optimal neighbor
4. Repeat until all vertices have been marked processed



Prim's Algorithm

Dijkstra's

1. Start at source
2. Update distance from current to unprocessed neighbors
3. Process optimal neighbor
4. Repeat until all vertices have been marked processed

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      foreach(edge (u,v) leaving u){
         if(u.dist+weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

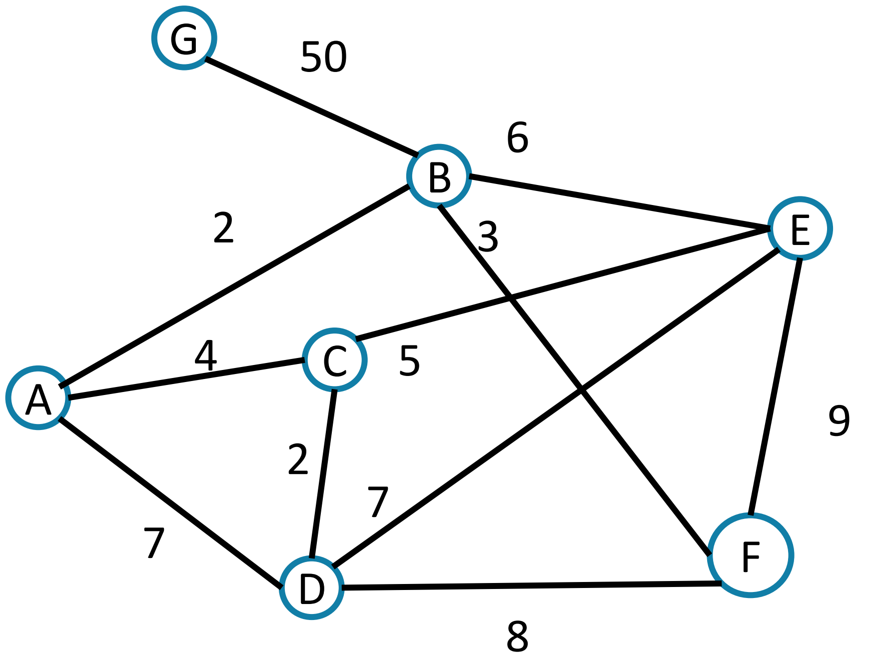
Algorithm idea:

1. choose an arbitrary starting point
2. Investigate edges that connect unprocessed vertices
3. Add the lightest edge to solution (be greedy)
4. Repeat until solution connects all vertices

```
PrimMST(Graph G)
  initialize distances to ∞
  mark source as distance 0
  mark all vertices unprocessed
  foreach(edge (source, v) ) {
    v.dist = weight(source,v)
    v.bestEdge = (source,v)
  }
  while(there are unprocessed vertices){
    let u be the closest unprocessed vertex
    add u.bestEdge to spanning tree
    foreach(edge (u,v) leaving u){
      if(weight(u,v) < v.dist && v unprocessed ){
        v.dist = weight(u,v)
        v.bestEdge = (u,v)
      }
    }
    mark u as processed
  }
```

Try it Out

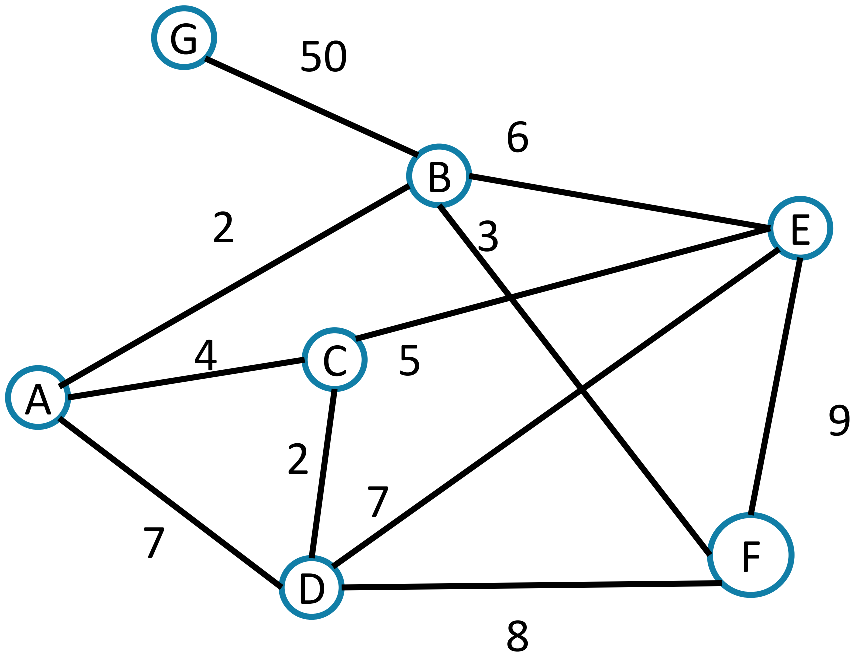
```
PrimMST(Graph G)
  initialize distances to ∞
  mark source as distance 0
  mark all vertices unprocessed
  foreach(edge (source, v) ) {
    v.dist = weight(source,v)
    v.bestEdge = (source,v)
  }
  while(there are unprocessed vertices){
    let u be the closest unprocessed vertex
    add u.bestEdge to spanning tree
    foreach(edge (u,v) leaving u){
      if(weight(u,v) < v.dist && v unprocessed ){
        v.dist = weight(u,v)
        v.bestEdge = (u,v)
      }
    }
    mark u as processed
  }
```



Vertex	Distance	Best Edge	Processed
A			
B			
C			
D			
E			
F			
G			

Try it Out

```
PrimMST(Graph G)
  initialize distances to ∞
  mark source as distance 0
  mark all vertices unprocessed
  foreach(edge (source, v) ) {
    v.dist = weight(source,v)
    v.bestEdge = (source,v)
  }
  while(there are unprocessed vertices){
    let u be the closest unprocessed vertex
    add u.bestEdge to spanning tree
    foreach(edge (u,v) leaving u){
      if(weight(u,v) < v.dist && v unprocessed ){
        v.dist = weight(u,v)
        v.bestEdge = (u,v)
      }
    }
    mark u as processed
  }
```



Vertex	Distance	Best Edge	Processed
A	-	X	✓
B	2	(A, B)	✓
C	4	(A, C)	✓
D	~~7~~ 2	~~(A, D)~~ (C, D)	✓
E	~~6~~ 5	~~(B, E)~~ (C, E)	✓
F	3	(B, F)	✓
G	50	(B, G)	✓

A different Approach

Prim’s Algorithm started from a single vertex and reached more and more other vertices.

Prim’s thinks vertex by vertex (add the closest vertex to the currently reachable set).

What if you think edge by edge instead?

Start from the lightest edge; add it if it connects new things to each other (don’t add it if it would create a cycle)

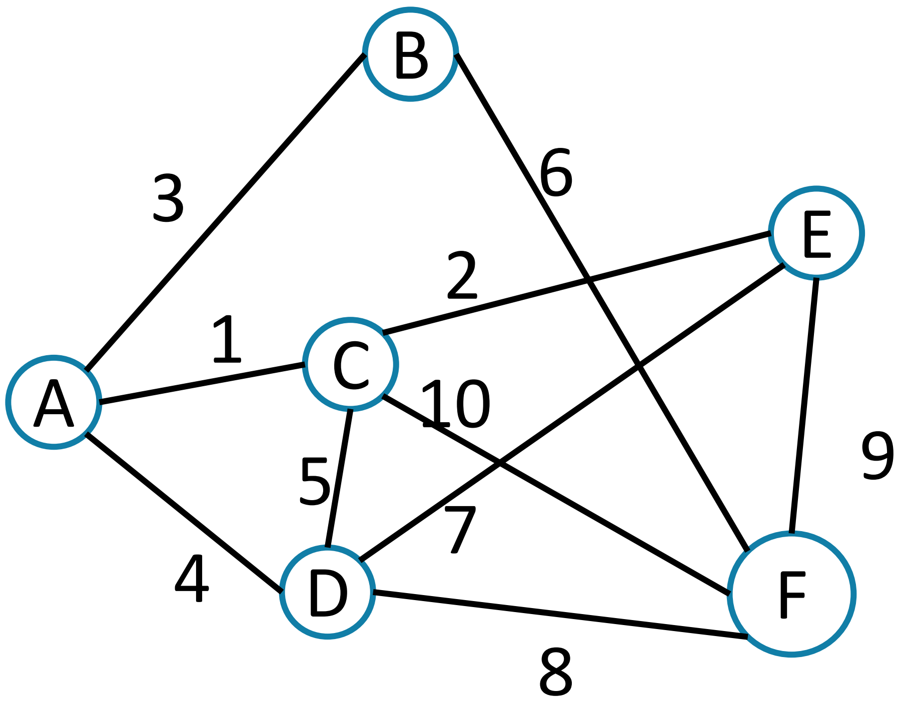
This is Kruskal’s Algorithm.

Kruskal’s Algorithm

```
KruskalMST(Graph G)
   initialize each vertex to be a connected
component
  sort the edges by weight
  foreach(edge (u, v) in sorted order){
    if(u and v are in different components){
      add (u,v) to the MST
      Update u and v to be in the same component
    }
  }
```

Try It Out

```
KruskalMST(Graph G)
   initialize each vertex to be a connected component
   sort the edges by weight
   foreach(edge (u, v) in sorted order){
      if(u and v are in different components){
         add (u,v) to the MST
         Update u and v to be in the same component
      }
   }
```

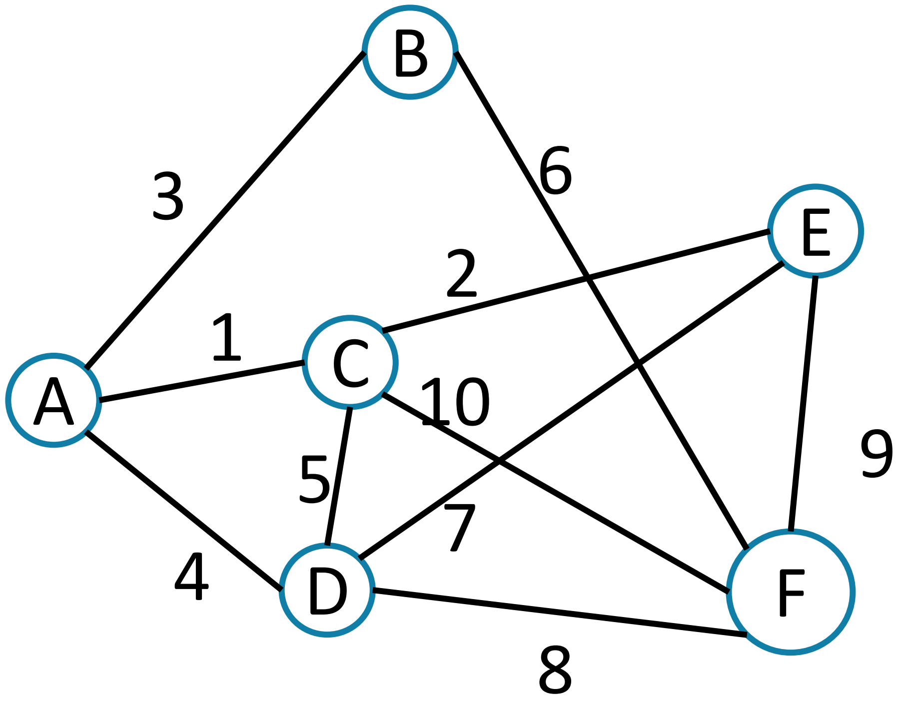


Edge	Include?	Reason
(A,C)		
(C,E)		
(A,B)		
(A,D)		
(C,D)		

Edge (cont.)	Inc?	Reason
(B,F)		
(D,E)		
(D,F)		
(E,F)		
(C,F)		

Try It Out

```
KruskalMST(Graph G)
   initialize each vertex to be a connected component
   sort the edges by weight
   foreach(edge (u, v) in sorted order){
      if(u and v are in different components){
         add (u,v) to the MST
         Update u and v to be in the same component
      }
   }
```



Edge	Include?	Reason
(A,C)	Yes	
(C,E)	Yes	
(A,B)	Yes	
(A,D)	Yes	
(C,D)	No	Cycle A,C,D,A

Edge (cont.)	Inc?	Reason
(B,F)	Yes	
(D,E)	No	Cycle A,C,E,D,A
(D,F)	No	Cycle A,D,F,B,A
(E,F)	No	Cycle A,C,E,F,D,A
(C,F)	No	Cycle C,A,B,F,C

Kruskal’s Algorithm: Running Time

```
KruskalMST(Graph G)
    initialize each vertex to be a connected component
   sort the edges by weight
   foreach(edge (u, v) in sorted order){
      if(u and v are in different components){
         add (u,v) to the MST
         Update u and v to be in the same component
      }
   }
```

Kruskal’s Algorithm: Running Time

Running a new BFS in the partial MST, at every step seems inefficient.

Do we have an ADT that will work here?

Not yet…