Faculty of Information Technology,
Monash University

FIT2004: Algorithms and Data Structures

Week 5: Greedy (Graph) Algorithms

Overview

Divide and conquer (W 1-3)	Greedy algorithms (W 4-5)	Dynamic programming (W 6-7)	Network flow (W 8-9)	Data structures (W 10-11)

- Last Lecture
  - Introduction to Graphs
  - Graph Traversal Algorithms

- Today's Lecture
  - Dijkstra's Algorithm
  - Prim's Algorithm
  - Kruskal's Algorithm

Greedy Algorithm

- Greedy paradigm for algorithms: make a **locally optimal** choice at each stage and commit to it.

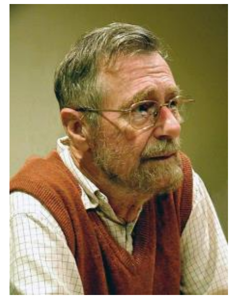
- Often it is easy to come up with ideas to solve a problem using a greedy strategy, but in many cases the proposed greedy algorithm does not find the overall optimal solution for the problem.

- **Correctness proofs are very important**.

- Today we will take a look at three very important greedy algorithms:
  - Dijkstra's algorithm solves the shortest path problem in graphs with non-negative weights.
  - Prim's and Kruskal's algorithms present two different ways to find a minimum spanning tree of a graph (an important problem for which there are actually multiple distinct greedy strategies that work fine).

Outline

1. Dijkstra's Algorithm
2. Prim's Algorithm
3. Kruskal's Algorithm



Edsger W. Dijkstra
Turing Award 1972

Dijkstra's Algorithm

- Solves the single source, all targets shortest path problem on graphs with non-negative weights. Closely related to BFS.
- Keeps track of a set *S* of nodes whose distance to the source has already been determined.
- Initially *S* only contains the source node, and it grows by one element at a time until finding all nodes that are reachable from the source node.
- At each iteration, from all nodes that are one edge away from *S*, add the node with the smallest distance to the source to S.
- By doing so, all the nodes will be added to *S* in the increasing order of distance to the source.

Dijkstra's Algorithm

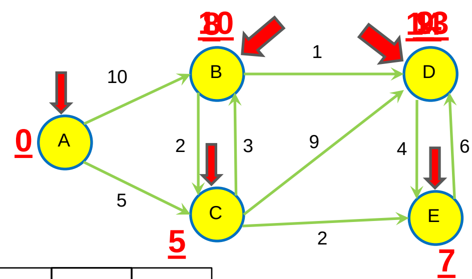
Initialize a list called Discovered and insert the source node A in it with distance 0

While Discovered is not empty

- Get the vertex v from the Discovered List **with smallest distance**
- For each outgoing edge (v, u, w) of v
  - If u is not in Discovered/Finalized
    - Insert u in Discovered with distance **v.distance + w**
  - Else If u.distance > v.distance + w
    - update the distance of u in Discovered to v.distance + w
- Move v from Discovered to Finalized

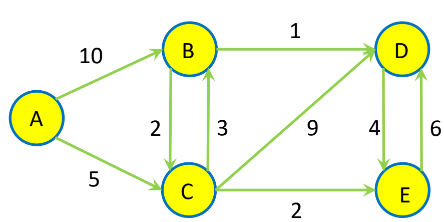
Discovered:

Finalized:

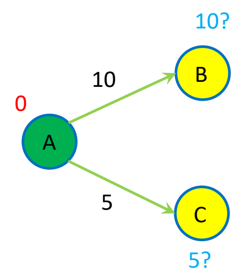


Discovered:	A, 0	B,10 / B,8	C,5	D,19 / D,9	E, 7
Finalized:	A,0	C,5	E, 7	B,8	D, 9

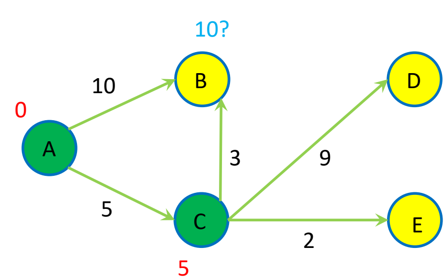
Dijkstra's Algorithm



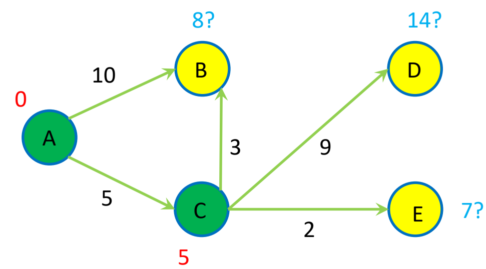
Dijkstra's Algorithm



Dijkstra's Algorithm

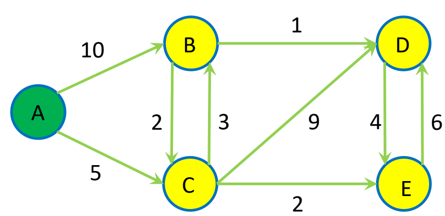


Dijkstra's Algorithm



Dijkstra's Algorithm

u:



Q:

A	B	C	D	E
0	Inf	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	-	-	-

Dist:

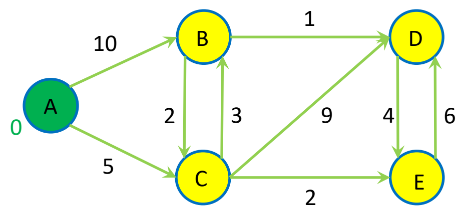
A	B	C	D	E
0	Inf	Inf	Inf	Inf

Q is a priority queue, where priority is based on distance

Pred and Dist are the usual ID-indexed arrays

Dijkstra's Algorithm

u: A



Q:

B	C	D	E
Inf	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	-	-	-

Dist:

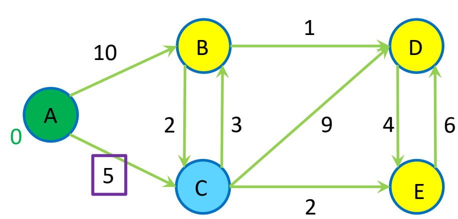
A	B	C	D	E
0	Inf	Inf	Inf	Inf

Q is a priority queue, where priority is based on distance

Pred and Dist are the usual ID-indexed arrays

Dijkstra's Algorithm

u: A



Q:

B	C	D	E
Inf	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	-	-	-

Dist:

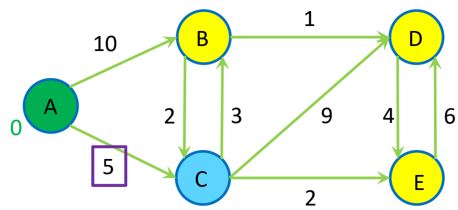
A	B	C	D	E
0	Inf	Inf	Inf	Inf

For each neighbour v of u, relax along that edge

- If dist[u] + w(u,v) < dist[v]
- Update dist[v]
- Set pred[v] = u

Dijkstra's Algorithm

u: A



Q:

B	C	D	E
Inf	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	-	-	-

Dist:

A	B	C	D	E
0	Inf	Inf	Inf	Inf

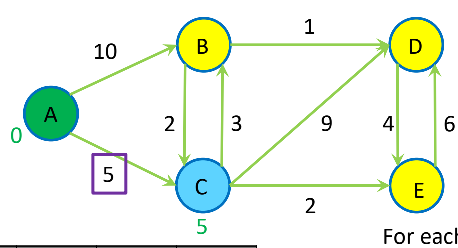
For each neighbour v of u, relax along that edge

- If dist[u] + w(u,v) < dist[v]
- Update dist[v]
- Set pred[v] = u

- 0 + 5 < Inf

Dijkstra's Algorithm

u: A



Q:

C	B	D	E
5	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	A	-	-

Dist:

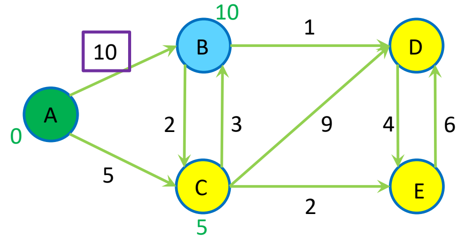
A	B	C	D	E
0	Inf	5	Inf	Inf

For each neighbour v of u, relax along that edge

- If dist[u] + w(u,v) < dist[v]
- Update dist[v]
- Set pred[v] = u

- 0 + 5 < Inf
- Set dist[C] = 5
- Set pred[C] = A
- Note that this changes the order of Q

Dijkstra’s Algorithm



u: A

Doing the same for B
- 0 + 10 < Inf
- Dist[B] = 10

Q:

C	B	D	E
5	10	Inf	Inf

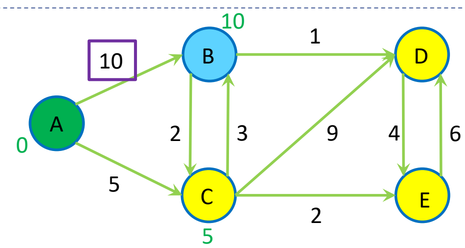
Pred:

A	B	C	D	E
-	-	A	-	-

Dist:

A	B	C	D	E
0	10	5	Inf	Inf

Dijkstra’s Algorithm



u: A

Doing the same for B
- 0 + 10 < Inf
- Dist[B] = 10
- Pred[B] = A

Q:

C	B	D	E
5	10	Inf	Inf

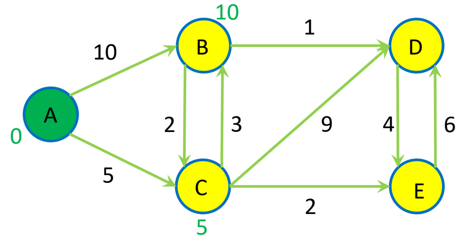
Pred:

A	B	C	D	E
-	A	A	-	-

Dist:

A	B	C	D	E
0	10	5	Inf	Inf

Dijkstra's Algorithm



u: A

Q:

C	B	D	E
5	10	Inf	Inf

Pred:

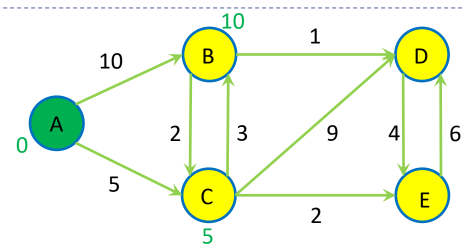
A	B	C	D	E
-	A	A	-	-

Dist:

A	B	C	D	E
0	10	5	Inf	Inf

- Finished with A, so pop from Q
- Notice that this will always be the vertex with the smallest distance estimate.

Dijkstra's Algorithm



u: C

Q:

B	D	E
10	Inf	Inf

Pred:

A	B	C	D	E
-	A	A	-	-

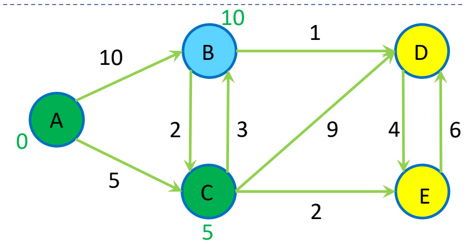
Dist:

A	B	C	D	E
0	10	5	Inf	Inf

- Finished with A, so pop from Q
- Notice that this will always be the vertex with the smallest distance estimate.
- The distance of this vertex is now finalised

Dijkstra's Algorithm

u: C



Q:

B	D	E
10	Inf	Inf

Pred:

A	B	C	D	E
-	A	A	-	-

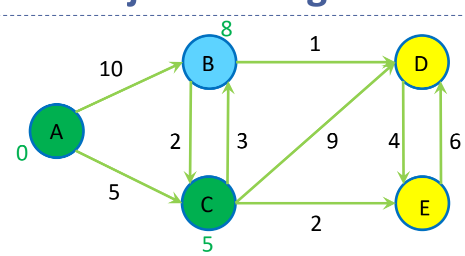
Dist:

A	B	C	D	E
0	10	5	Inf	Inf

- Relax B from C
- Dist[C] + w(C, B) = 5 + 3 < 10

Dijkstra's Algorithm

u: C



Q:

B	D	E
8	Inf	Inf

Pred:

A	B	C	D	E
-	C	A	-	-

Dist:

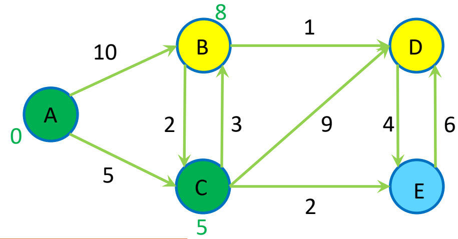
A	B	C	D	E
0	8	5	Inf	Inf

- Relax B from C
- Dist[C] + w(C, B) = 5 + 3 < 10

- Dist[B] = 8
- Pred[B] = C

Dijkstra's Algorithm

u: C



Q:

B	D	E
8	Inf	Inf

- Relax E from C
- Dist[C] + w(C, E) = 5 + 2 < Inf

Pred:

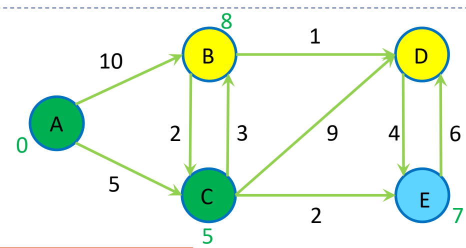
A	B	C	D	E
-	C	A	-	-

Dist:

A	B	C	D	E
0	8	5	Inf	Inf

Dijkstra's Algorithm

u: C



Q:

E	B	D
7	8	Inf

- Relax E from C
- Dist[C] + w(C, E) = 5 + 2 < Inf

- Dist[E] = 7
- Pred[E] = C

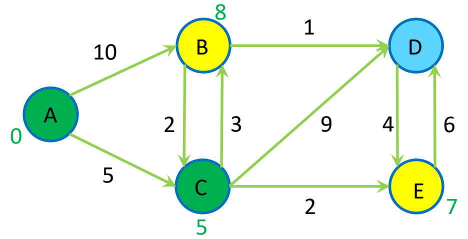
Pred:

A	B	C	D	E
-	C	A	-	C

Dist:

A	B	C	D	E
0	8	5	Inf	7

Dijkstra’s Algorithm



u: C

Q:

E	B	D
7	8	Inf

- Relax D from C
- Dist[D] + w(C, D) = 5 + 9 < Inf

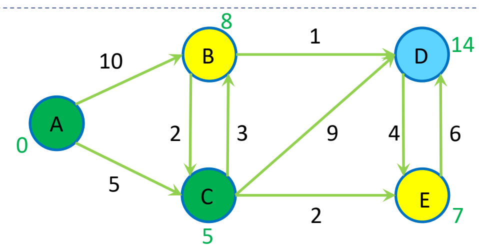
Pred:

A	B	C	D	E
-	C	A	-	C

Dist:

A	B	C	D	E
0	8	5	Inf	7

Dijkstra’s Algorithm



u: C

Q:

E	B	D
7	8	14

- Relax D from C
- Dist[D] + w(C, D) = 5 + 9 < Inf

- Dist[D] = 14
- Pred[D] = C

Pred:

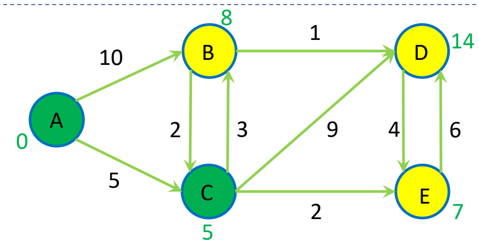
A	B	C	D	E
-	C	A	C	C

Dist:

A	B	C	D	E
0	8	5	14	7

Dijkstra’s Algorithm

u: C



Q:

E	B	D
7	8	14

Pred:

A	B	C	D	E
-	C	A	C	C

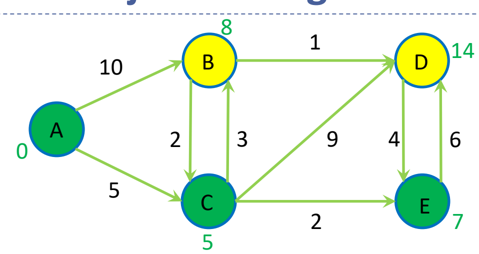
Dist:

A	B	C	D	E
0	8	5	14	7

- Done with C
- Pop another vertex from Q and finalise it

Dijkstra’s Algorithm

u: E



Q:

B	D
8	14

Pred:

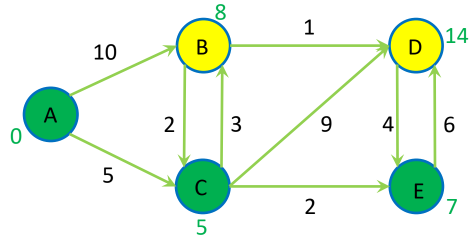
A	B	C	D	E
-	C	A	C	C

Dist:

A	B	C	D	E
0	8	5	14	7

Dijkstra's Algorithm

u: E



Q:

B	D
8	14

Pred:

A	B	C	D	E
-	C	A	C	C

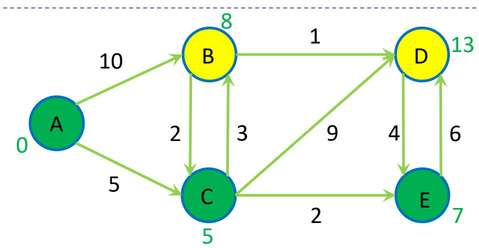
Dist:

A	B	C	D	E
0	8	5	14	7

- Relax D from E
- Dist[E] + w(E, D) = 7 + 6 < 14 = Dist[D]

Dijkstra's Algorithm

u: E



Q:

B	D
8	13

Pred:

A	B	C	D	E
-	C	A	E	C

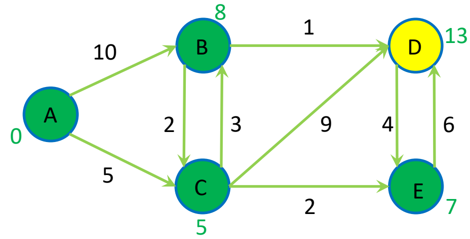
Dist:

A	B	C	D	E
0	8	5	13	7

- Relax D from E
- Dist[E] + w(E, D) = 7 + 6 < 14 = Dist[D]
- Dist[D] = 13
- Pred[D] = E
- Done with E
- Pop B

Dijkstra’s Algorithm

u: B



Q:

D
13

- Relax D from B
- Dist[B] + w(B, D) = 8 + 1 = 9 < 13 = Dist[D]

Pred:

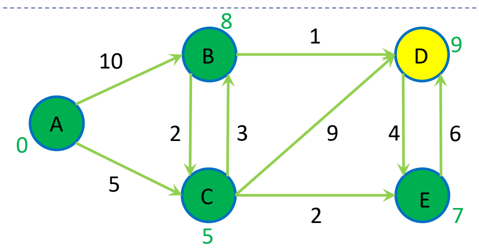
A	B	C	D	E
-	C	A	E	C

Dist:

A	B	C	D	E
0	8	5	13	7

Dijkstra’s Algorithm

u: B



Q:

D
9

- Relax D from B
- Dist[B] + w(B, D) = 8 + 1 = 9 < 13 = Dist[D]

Pred:

A	B	C	D	E
-	C	A	B	C

- Dist[D] = 9
- Pred[D] = B
- Done with B

Dist:

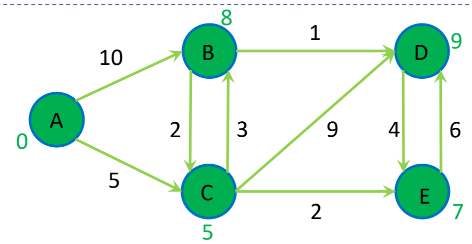
A	B	C	D	E
0	8	5	9	7

- Pop D

FIT2004: Seminar 5 - Greedy (Graph) Algorithms

Dijkstra's Algorithm

u: D



Q:

- Relax E from D
- Dist[D] + w(D, E) = 9 + 4 = 13 > 7
    - No update
- Done!

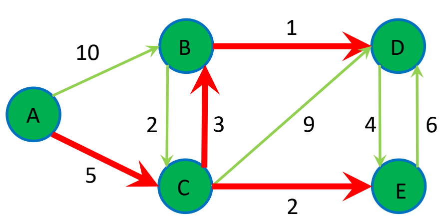
Pred:

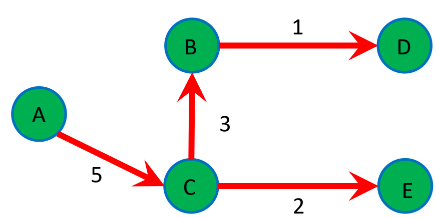
A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	8	5	9	7

Dijkstra's Algorithm





Q:

Pred:

A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	8	5	9	7

Dijkstra's Algorithm

Algorithm 38 Dijkstra's algorithm

```
function DIJKSTRA(G = (V, E), s)
    dist[1..n] = ∞
    pred[1..n] = 0
    dist[s] = 0
    Q = priority_queue(V[1..n], key(v) = dist[v])
    while Q is not empty do
        u = Q.pop_min()
        for each edge e that is adjacent to u do
            // Priority queue keys must be updated if relax improves a distance estimate!
            RELAX(e)
    return dist[1..n], pred[1..n]
```

Algorithm 37 Edge relaxation

```
function RELAX(e = (u, v))
    if dist[v] > dist[u] + w(u, v) then
        dist[v] = dist[u] + w(u, v)
        pred[v] = u
```


Dijkstra's Algorithm

Algorithm 38 Dijkstra's algorithm

```
function DIJKSTRA(G = (V, E), s)
    dist[1..n] = ∞
    pred[1..n] = 0
    dist[s] = 0
    Q = priority_queue(V[1..n], key(v) = dist[v])
    while Q is not empty do
        u = Q.pop_min()
        for each edge e that is adjacent to u do
            // Priority queue keys must be updated if relax improves a distance estimate!
            RELAX(e)
    return dist[1..n], pred[1..n]
```

While loop: $\Theta(V)$

For loop: $\Theta(E)$

Time Complexity:

- While loop executes $\theta(V)$ times
  - Find the vertex with smallest distance and delete it: depends on priority queue implementation (Q.extract_min)
- Each edge visited once → $\theta(E)$
  - Updating the priority queue: depends on implementation (Q.update)
  - Relaxation is $\theta(1)$ since we can find distances and compare them in $\theta(1)$
- Total cost: θ(E*Q.update+ V*Q.extract_min)

Dijkstra's Algorithm using min-heap

Total cost: θ(E*Q.update+ V*Q.extract_min)

Required additional data structure:

- Create an array called Vertices.
- Vertices[i] will record the location of i-th vertex in the min-heap

Updating the distance of a vertex v in min-heap in θ(*log V*)

- Find the location in the queue (heap) in θ(1) using Vertices
- Now do the normal heap-up operation in θ(log V)
  - For each swap performed between two vertices x and y during the upHeap
    - Update Vertices[x] and Vertices[y] to record their updated locations in the min-heap

Time Complexity of Dijkstra's Algorithm

- While loop executes θ(V) times
  - Extract the vertex with smallest distance using min_heap: θ(*log V)*
- Each edge visited once → θ(E)
  - Updating the priority queue using min_heap: θ(*log V)*
  - Relaxation is θ(1) since we can find distances and compare them in θ(1)
- Total cost: θ(E*Q.update+ V*Q.extract_min)
- Total cost: θ(*E log V + V log V)*
  - Which term dominates?
- If the graph is connected, then **E ≥ V-1**
  - E dominates V
- Total cost: θ(*E log V)*

Algorithm 38 Dijkstra's algorithm

```
function DIJKSTRA(G = (V, E), s)
    dist[1..n] = ∞
    pred[1..n] = 0
    dist[s] = 0
    Q = priority_queue(V[1..n], key(v) = dist[v])
    while Q is not empty do
        u = Q.pop_min()
        for each edge e that is adjacent to u do
            // Priority queue keys must be updated
            RELAX(e)
    return dist[1..n], pred[1..n]
```

Proof of Correctness

Claim: For every vertex v which has been removed from the queue, dist[v] is correct.

- Notation:
  - V is the set of vertices
  - Q is the set of vertices in the queue
  - S = V / Q = the set of vertices that have been removed from the queue

Base Case

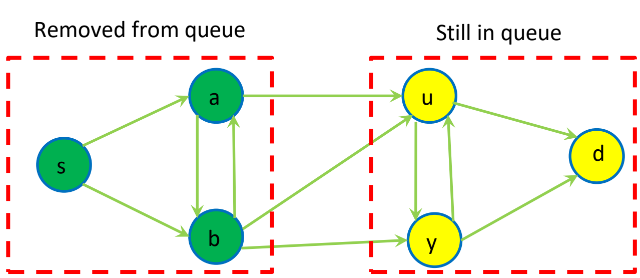
- dist[s] is initialised to 0, which is the shortest distance from s to s (since there are no negative weights).

Proof of Correctness

Claim: For every vertex v which has been removed from the queue, dist[v] is correct.

Inductive Step:

- Assume that the claim holds for all vertices which have been removed from the queue, in other words it holds for all vertices in the set S.
- Let u be the next vertex which is removed from the queue.
- We will show that dist[u] is correct.



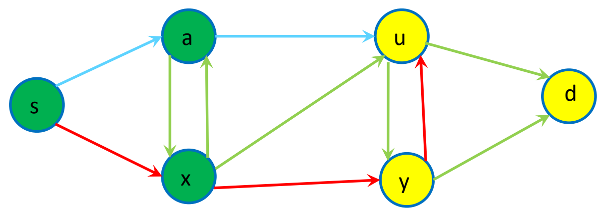
Proof of Correctness

Claim: For every vertex v which has been removed from the queue, dist[v] is correct.

Inductive Step:

- Suppose (for contradiction) there is a shortest path P, s⇝u with len(P) < dist[u].
- Let x be the furthest vertex on P which is in S (i.e. has been finalised).
- By the inductive hypothesis, dist[x] is correct (since it is in S).

Current path
Assumed shorter path (P)

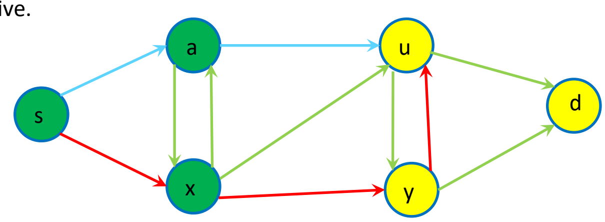


Proof of Correctness

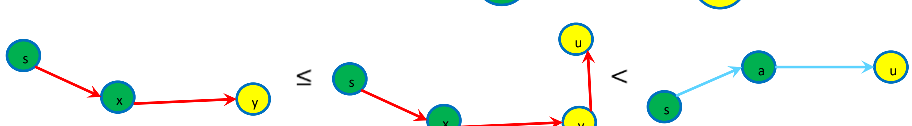
Claim: For every vertex v which has been removed from the queue, dist[v] is correct.

Inductive Step:

- By the inductive hypothesis, dist[x] is correct (since it is in S).
- Let y be the next vertex on P after x.
- By assumption len(P) < dist[u].
- Edge weights are non-negative.
- len(s⇝y) ≤ len(P) < dist[u].



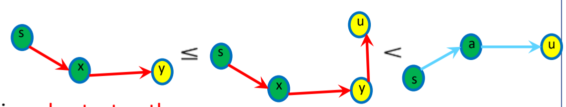
Current path
Assumed shorter path (P)



Proof of Correctness

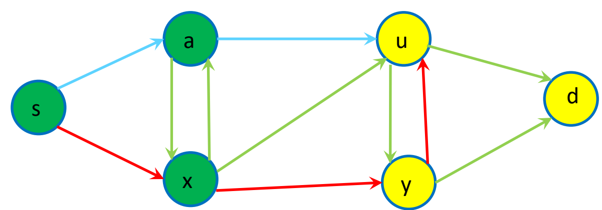
Claim: For every vertex v which has been removed from the queue, dist[v] is correct.

Inductive Step:



- len(s⇝y) ≤ len(P) < dist[u]
- Since we said that P (via x and y) is a shortest path…
- dist[y] = len(s⇝y) < dist[u].
- So dist[y] < dist[u]…
- If y ≠ u, why didn't y get removed before u???
- If y = u, how can dist[y] < dist[u]???

Current path
Assumed shorter path (P)



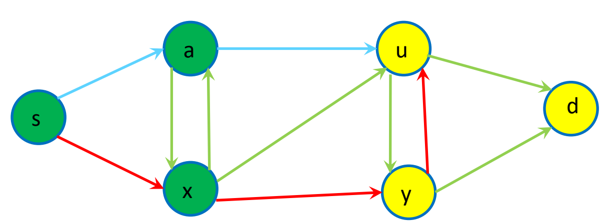
Proof of Correctness

Claim: For every vertex v which has been removed from the queue, dist[v] is correct.

Inductive Step:

- Having obtained a contradiction, we can **negate our assumption**, namely:
- "Suppose (for contradiction) there is a shorter path P, s⇝u with len(P) < dist[u]."
- **So there is no such path and dist[u] is correct**.
- So by induction, **the distance of every vertex is correct when Dijkstra's algorithm terminates.**

Current path
Assumed shorter path (P)



Outline

1. Dijkstra's Algorithm
2. Prim's Algorithm
3. Kruskal's Algorithm

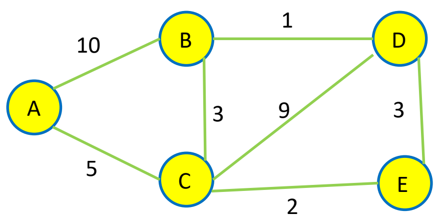
What is a Spanning Tree

Tree:

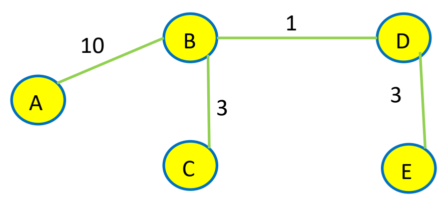
A tree is a connected undirected graph with no cycles in it.

Spanning Tree:

- A spanning tree of a general undirected weighted graph G is a tree that spans G (i.e., a tree that includes every vertex of G) and is a subgraph of G (i.e., every edge in the spanning tree belongs to G).

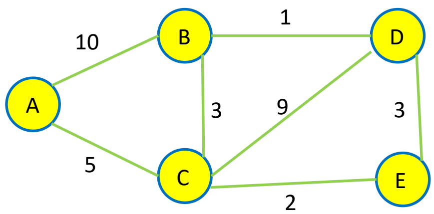


An undirected graph

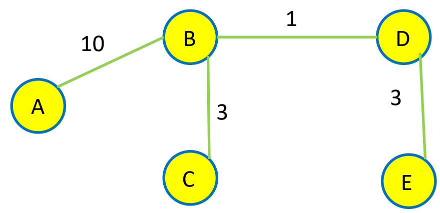


Spanning Tree

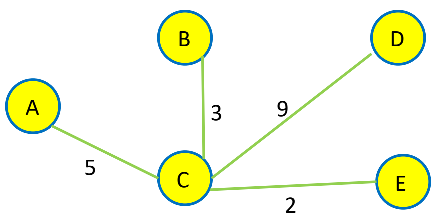
Spanning Tree Examples



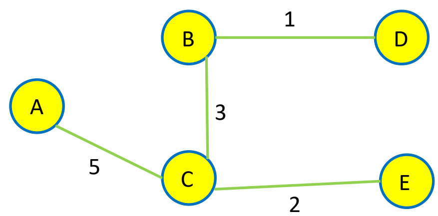
Graph



Spanning Tree 1



Spanning Tree 2



Spanning Tree 3

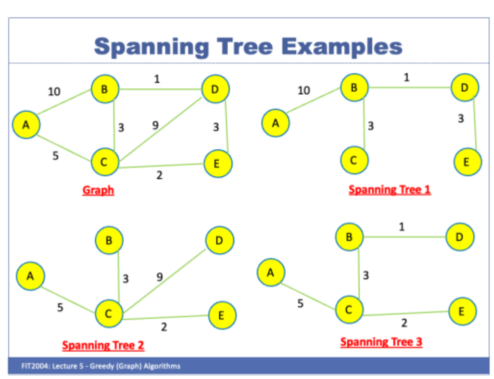
What is a Spanning Tree

Is it true that a spanning tree of a connected graph G is a maximal set of edges of G that contains no cycles?

- Yes

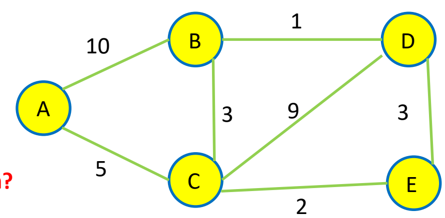
Is it true that a spanning tree of a connected graph G is a minimal set of edges that connect all vertices?

- Yes



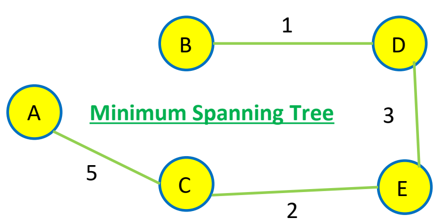
Minimum Spanning Tree (MST)

- **Weight** of a spanning tree is the sum of the weights of the edges in the tree.
- A minimum spanning tree of a weighted general graph G is a tree that spans G, whose **weight is minimum** over all possible spanning trees for this graph.
- There may be **more than one minimum spanning tree** for a graph G (e.g., two or more spanning trees with the same minimum weight).

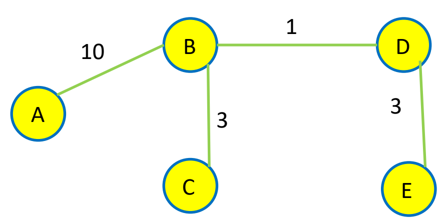


What is the weight of the MST in this graph?
How many MSTs are in this graph?

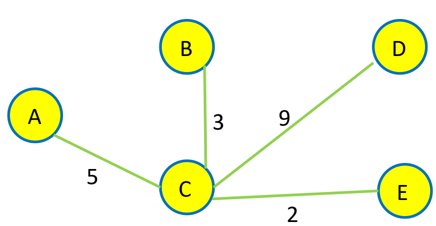
Spanning Trees and MSTs



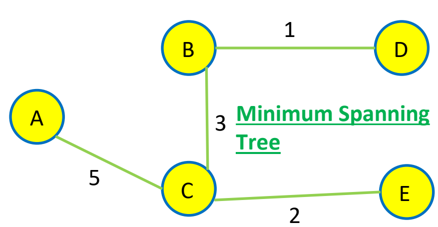
Spanning Tree 1: Weight 11



Spanning Tree 2: Weight 17



Spanning Tree 3: Weight 19



Spanning Tree 4: Weight 11

MST Algorithms

- Let T denote the MST we are constructing, initialised to be empty.
- An edge e is said to be safe if {T U e} is a subset of some MST

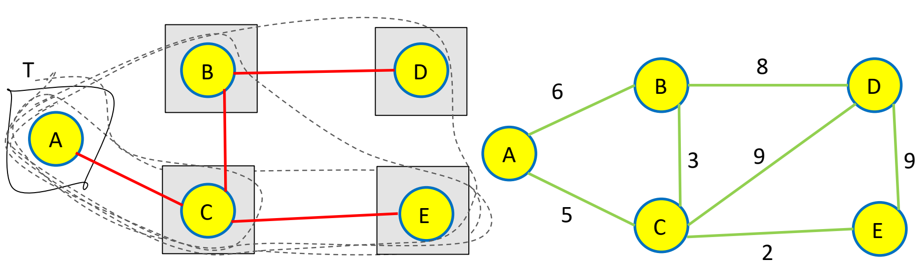
General Strategy:

- T = null
- **while** T can be grown safely**:**
-     find an edge e=<x,y> along which T **is** safe to grow
-     T = {T} union {<x,y>}
- **return** T

- We will study two **greedy** algorithms that follow this strategy.

Prim's Algorithm: Overview

- Start by picking any vertex v to be the source of T.
- While T does not contain **all** vertices in the graph:
  - Find shortest edge e that goes from T to a different connected component.
  - Add e to the tree T.



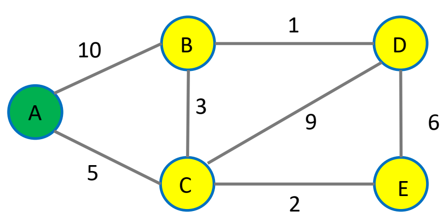
Connected components

MST Algorithms: Prim's Algorithm

Prim's Algorithm (small modification of Dijkstra's Algorithm)

- We start with no edges in T, so there are V connected components in T (each with a single node) and select one node as the source.
- At each iteration, we add to T the shortest edge that links the connected component containing the source to a different connected component.
- In each iteration, the number of connected components decreases by 1 and the number of nodes in the connected component containing the source increases by 1.
- Cycles are avoided as no edges linking two nodes that are already in the same connected component are ever added to T.
- Time complexity: $\theta(E \log V)$

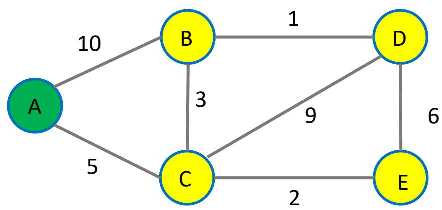
Prim's Algorithm



Robert C. Prim

Prim’s Algorithm

u:



Q:

A	B	C	D	E
0	Inf	Inf	Inf	Inf

parent

A	B	C	D	E
-	-	-	-	-

Dist:

A	B	C	D	E
0	Inf	Inf	Inf	inf

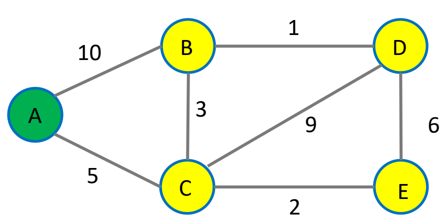
Prim's deals with distance estimates to reach T instead of distance estimates to the source

Q is a priority queue, where priority is based on distance

parent and Dist are the usual ID-indexed arrays

Prim’s Algorithm

u: A



Q:

B	C	D	E
Inf	Inf	Inf	Inf

parent

A	B	C	D	E
-	-	-	-	-

Dist:

A	B	C	D	E
0	Inf	Inf	Inf	inf

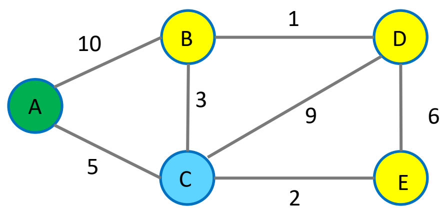
Prim's deals with distance estimates to reach T instead of distance estimates to the source

Q is a priority queue, where priority is based on distance

parent and Dist are the usual ID-indexed arrays

Prim’s Algorithm

u: A



Q:

B	C	D	E
Inf	Inf	Inf	Inf

parent

A	B	C	D	E
-	-	-	-	-

Dist:

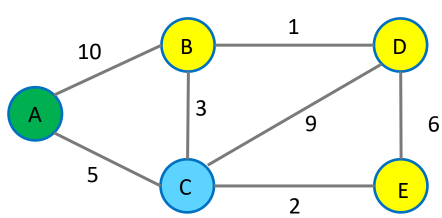
A	B	C	D	E
0	Inf	Inf	Inf	inf

For each neighbour v of u, try to update distance estimate

5 < inf
Dist[C] = 5
parent[C] = A

Prim’s Algorithm

u: A



Q:

B	C	D	E
Inf	Inf	Inf	Inf

parent

A	B	C	D	E
-	-	-	-	-

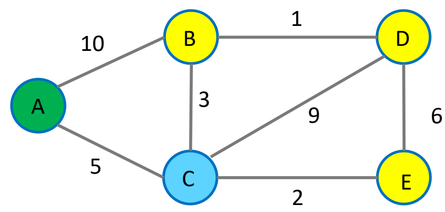
Dist:

A	B	C	D	E
0	Inf	Inf	Inf	inf

For each neighbour v of u, try to update distance estimate

This time we just care about the edge, not the distance to u + the edge

Prim's Algorithm



u: A

Q:

B	C	D	E
Inf	Inf	Inf	Inf

parent

A	B	C	D	E
-	-	-	-	-

Dist:

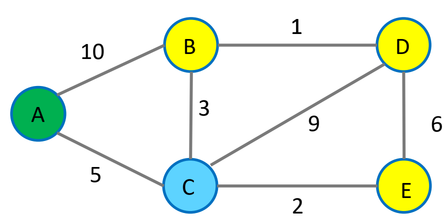
A	B	C	D	E
0	Inf	Inf	Inf	inf

For each neighbour v of u, try to update distance

W(A,C) = 5
Dist[C] = inf

5 < inf, so update
Dist[C] = 5
parent[C] = A

Prim's Algorithm



u: A

Q:

C	B	D	E
5	Inf	Inf	Inf

parent

A	B	C	D	E
-	-	A	-	-

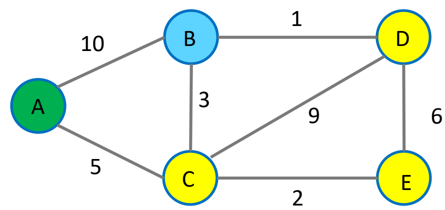
Dist:

A	B	C	D	E
0	Inf	5	Inf	inf

Note that this changes the order of Q

Prim's Algorithm

u: A



Q:

C	B	D	E
5	10	Inf	Inf

parent

A	B	C	D	E
-	-	A	-	-

Dist:

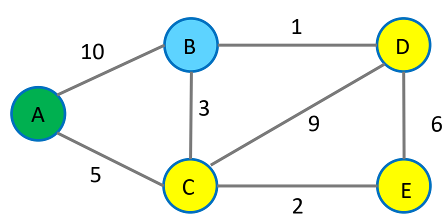
A	B	C	D	E
0	Inf	5	Inf	inf

Doing the same for B
W(A,B) = 10
Dist[B] = inf

10 < inf

Prim's Algorithm

u: A



Q:

C	B	D	E
5	10	Inf	Inf

parent

A	B	C	D	E
-	A	A	-	-

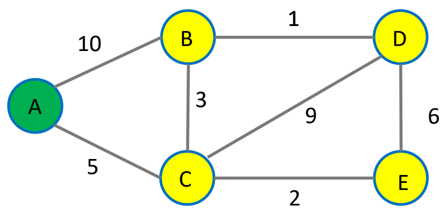
Dist:

A	B	C	D	E
0	10	5	Inf	inf

Doing the same for B
W(A,B) = 10
Dist[B] = inf

10 < inf, so update
- Dist[B] = 10
- parent[B] = A

Prim’s Algorithm



u: A

Q:

C	B	D	E
5	10	Inf	Inf

parent

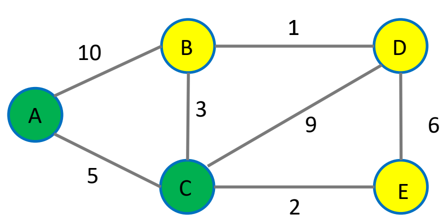
A	B	C	D	E
-	A	A	-	-

Dist:

A	B	C	D	E
0	10	5	Inf	inf

- Finished with A, so pop from Q
- Notice that this will always be the vertex with the smallest distance estimate on Q

Prim’s Algorithm



u: C

Q:

B	D	E
10	Inf	Inf

parent

A	B	C	D	E
-	A	A	-	-

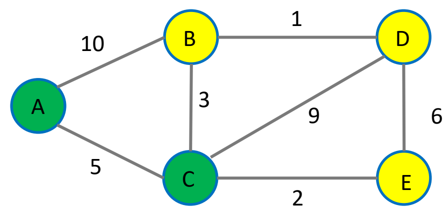
Dist:

A	B	C	D	E
0	10	5	Inf	inf

- Finished with A, so pop from Q
- Notice that this will always be the vertex with the smallest distance estimate on Q
- This vertex is now in the MST

Prim's Algorithm

u: C



Q:

B	D	E
10	Inf	Inf

parent

A	B	C	D	E
-	A	A	-	-

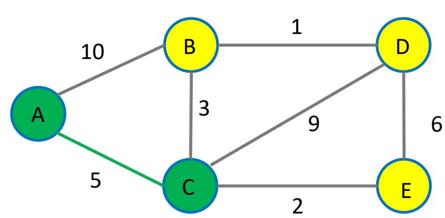
Dist:

A	B	C	D	E
0	10	5	Inf	inf

- The edge we add is the edge between u and parent[u]
- So in this case, A->C

Prim's Algorithm

u: C



Q:

B	D	E
10	Inf	Inf

parent

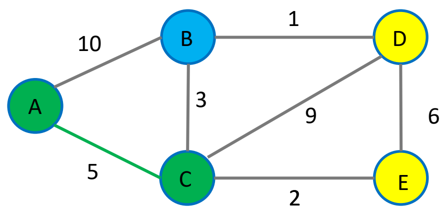
A	B	C	D	E
-	A	A	-	-

Dist:

A	B	C	D	E
0	10	5	Inf	inf

- The edge we add is the edge between u and parent[u]
- So in this case, A->C

Prim's Algorithm



u: C

Q:

B	D	E
10	Inf	Inf

parent

A	B	C	D	E
-	A	A	-	-

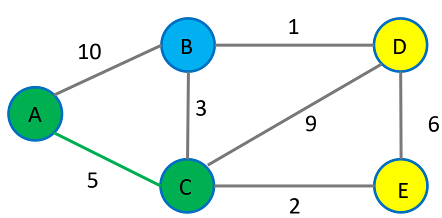
Dist:

A	B	C	D	E
0	10	5	Inf	inf

- Update B from C
- w(C, B) = 3
- Dist[B] = 10

- 3 < 10 so update dist[B] and parent[B]

Prim's Algorithm



u: C

Q:

B	D	E
3	Inf	Inf

parent

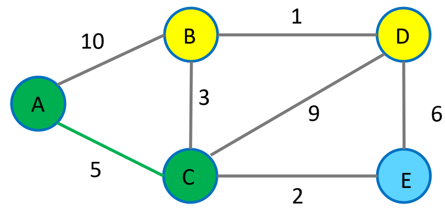
A	B	C	D	E
-	C	A	-	-

Dist:

A	B	C	D	E
0	3	5	Inf	inf

Prim's Algorithm

u: C



Q:

B	D	E
3	Inf	Inf

- Update E from C

parent

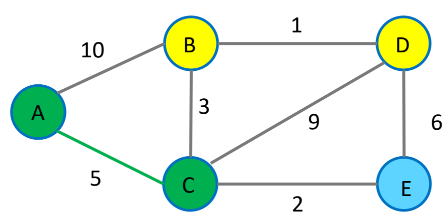
A	B	C	D	E
-	C	A	-	-

Dist:

A	B	C	D	E
0	3	5	Inf	inf

Prim's Algorithm

u: C



Q:

E	B	D
2	3	inf

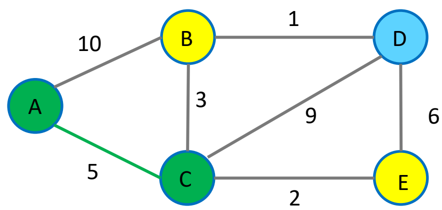
parent

A	B	C	D	E
-	C	A	-	C

Dist:

A	B	C	D	E
0	3	5	Inf	2

Prim's Algorithm



u: C

Q:

E	B	D
2	3	inf

parent

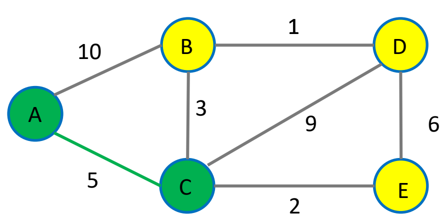
A	B	C	D	E
-	C	A	-	C

Dist:

A	B	C	D	E
0	3	5	Inf	2

- Update D from C

Prim's Algorithm



u: C

Q:

E	B	D
2	3	9

parent

A	B	C	D	E
-	C	A	C	C

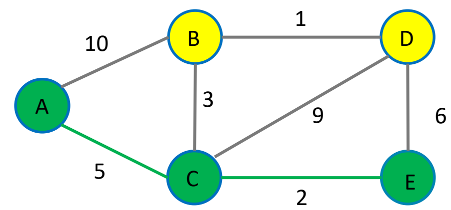
Dist:

A	B	C	D	E
0	3	5	9	2

- Done with C
- Pop another vertex from Q and add it to the MST

Prim's Algorithm

u: E



Q:

B	D
3	9

parent

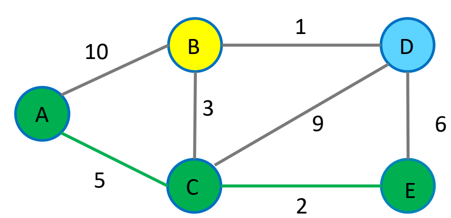
A	B	C	D	E
-	C	A	C	C

Dist:

A	B	C	D	E
0	3	5	9	2

Prim's Algorithm

u: E



Q:

B	D
3	6

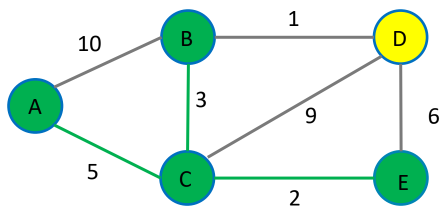
parent

A	B	C	D	E
-	C	A	E	C

Dist:

A	B	C	D	E
0	3	5	6	2

Prim’s Algorithm



u: B

Q:

D
6

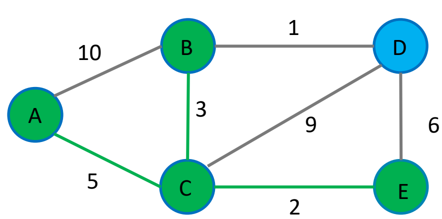
parent

A	B	C	D	E
-	C	A	E	C

Dist:

A	B	C	D	E
0	3	5	6	2

Prim’s Algorithm



u: B

Q:

D
6

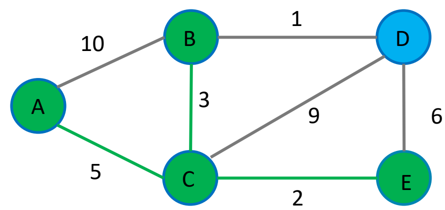
parent

A	B	C	D	E
-	C	A	E	C

Dist:

A	B	C	D	E
0	3	5	6	2

Prim’s Algorithm



u: B

Q:

D
1

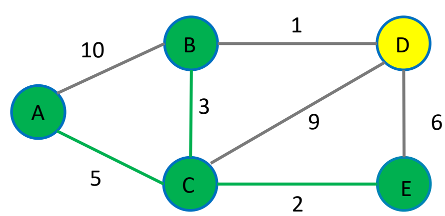
parent

A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	3	5	1	2

Prim’s Algorithm



u: B

Q:

D
1

parent

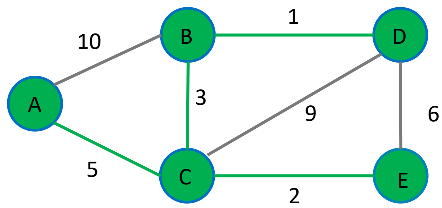
A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	3	5	1	2

Prim’s Algorithm

u: D



Q:

parent

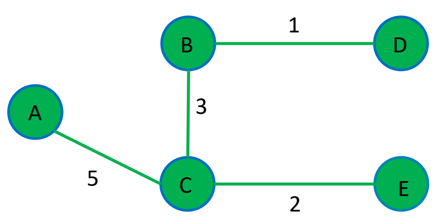
A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	3	5	1	2

Prim’s Algorithm

u: D



Q:

parent

A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	3	5	1	2

Prim's Algorithm

Algorithm 69 Prim's algorithm

```
function PRIM(G = (V, E), r)
    dist[1..n] = ∞
    parent[1..n] = null
    T = ({r}, ∅)
    dist[r] = 0
    Q = priority_queue(V[1..n], key(v) = dist[v])
    while Q is not empty do
        u = Q.pop_min()
        T.add_vertex(u)
        T.add_edge(parent[u], u)
        for each edge e = (u, v) adjacent to u do
            if not v ∈ T and dist[v] > w(u, v) then
                // Remember to update the key of v in the priority queue!
                dist[v] = w(u, v)
                parent[v] = u
    return T
```


Prim's Algorithm: Complexity

It is very similar to Dijkstra's Algorithm and its complexity is the same as Dijkstra's Algorithm

- θ(V log V + E log V) if min-heap is used

Since the input graph G is connected, E ≥ V-1. Hence, the complexity can be simplified to *θ(E log V).*

Prim's Algorithm: Correctness

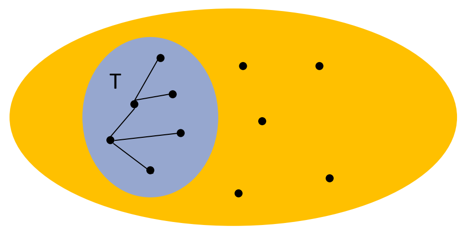
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**



Prim's Algorithm: Correctness

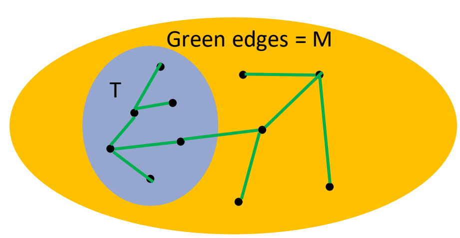
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**



Prim's Algorithm: Correctness

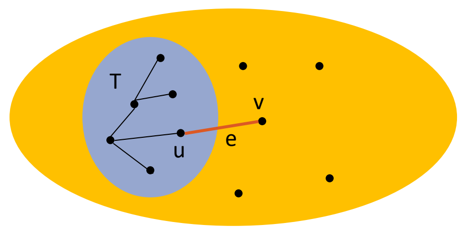
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**
- Let **e = (u,v)** be the lightest edge that connects some **u** in **T** to some **v** not in **T** (i.e. this is the edge Prim's algorithm will choose in this iteration)
- If **e** is in **M**, then **T ∪ {e}** is a subset of **M**, which is an MST, so the invariant holds



Prim's Algorithm: Correctness

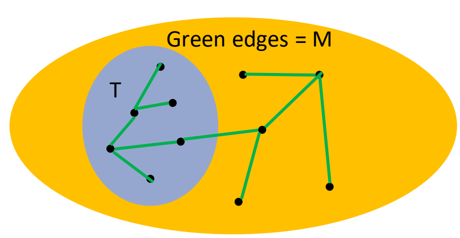
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**
- Let **e = (u,v)** be the lightest edge that connects some **v** in **T** to some **u** not in **T** (i.e. this is the edge Prim's algorithm will choose in this iteration)
- If **e** is in **M**, then **T ∪ {e}** is a subset of **M**, which is an MST, so the invariant holds



Prim's Algorithm: Correctness

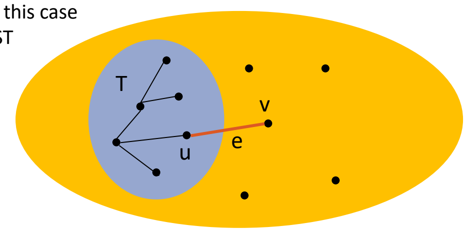
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**
- Let **e = (u,v)** be the lightest edge that connects some **v** in **T** to some **u** not in **T** (i.e. this is the edge Prim's algorithm will choose in this iteration)
- If **e** is in **M**, then **T ∪ {e}** is a subset of **M**, which is an MST, so the invariant holds
- The interesting case is where **e** is not in **M**. In this case we have to show that there is some other MST which contains **T ∪ {e}**



Prim's Algorithm: Correctness

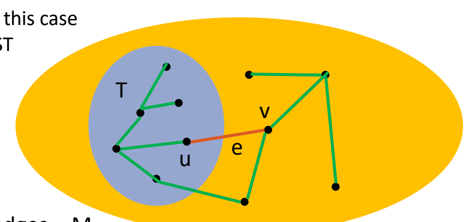
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**
- Let **e = (u,v)** be the lightest edge that connects some **v** in **T** to some **u** not in **T** (i.e. this is the edge Prim's algorithm will choose in this iteration)
- If **e** is in **M**, then **T ∪ {e}** is a subset of **M**, which is an MST, so the invariant holds
- The interesting case is where **e** is not in **M**. In this case we have to show that there is some other MST which contains **T ∪ {e}**

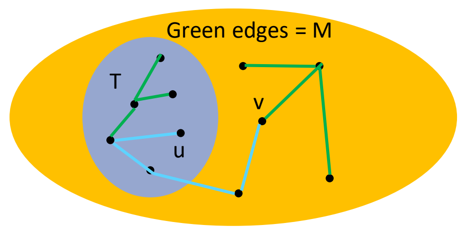


Green edges = M

Prim's Algorithm: Correctness

Inductive step:

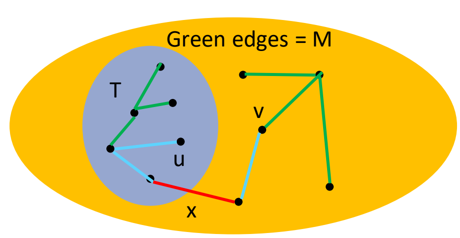
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- **u** and **v** are not connected in **T** (since **v** is not in **T**). Consider the first edge on the blue path which is **not** contained in **T** (call this edge **x**).



Prim's Algorithm: Correctness

Inductive step:

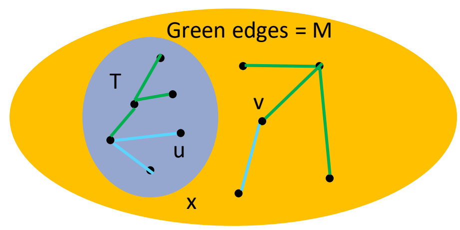
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- **u** and **v** are not connected in **T** (since **v** is not in **T**). Consider the first edge on the blue path which is **not** contained in **T** (call this edge **x**).
- One vertex of this edge is in **T**, the other is not.
- Removing this edge would disconnect **M**



Prim’s Algorithm: Correctness

Inductive step:

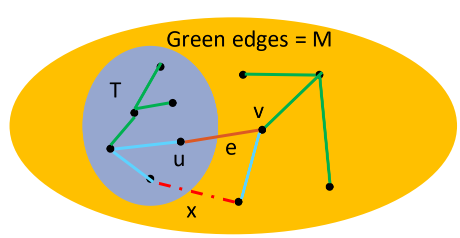
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- **u** and **v** are not connected in **T** (since **v** is not in **T**). Consider the first edge on the blue path which is **not** contained in **T** (call this edge **x**).
- One vertex of this edge is in **T**, the other is not.
- Removing this edge would disconnect **M**



Prim’s Algorithm: Correctness

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- **u** and v are not connected in **T** (since **v** is not in **T**). Consider the first edge on the blue path which is **not** contained in **T** (call this edge **x**).
- One vertex of this edge is in **T**, the other is not.
- Removing this edge would disconnect **M**
- Adding the edge **e=(u,v)** would form a new spanning tree, **M’**
- Since Prim's algorithm always selects the lightest edge incident to **T**, we know that **w(e) ≤ w(x)**
- So the weight of **M’** is no greater than the weight of **M**, therefore choosing **e** is correct



Outline

1. Dijkstra's Algorithm
2. Prim's Algorithm
3. Kruskal's Algorithm

MST Algorithms: Kruskal's Algorithm

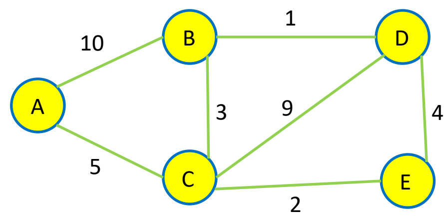
Kruskal's Algorithm

- We start with no edges in T, so there are V connected components in T (each with a single node).
- We process edges in ascending order of edge weights, and add an edge e to T if this addition does not create a cycle.
- Each time that one edge e is added to T, e is the smallest edge that links two distinct connected components of T.
- Cycles are avoided as no edges linking two nodes that are already in the same connected component are ever added to T. At any point of the execution T is a forest (i.e., a set of trees)
- Time complexity: $\Theta(E \log V)$.

Kruskal's Algorithm

Kruskals(G(V, E))
- Sort the edges in ascending order of weights
- Let T be a graph with V as its vertices, and no edges
- For each edge (u, v) in ascending order
  - If adding (u, v) does not create a cycle in T
    - Add (u, v) to T
- Return T


Joseph Kruskal



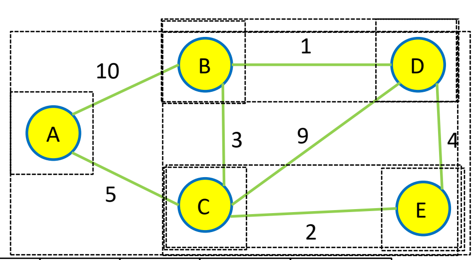
FIT2004: Seminar 5 - Greedy (Graph) Algorithms

Kruskal's Algorithm

Kruskals(G(V, E))
- Sort the edges in ascending order of weights
- Let T be a graph with V as its vertices, and no edges
- For each edge (u, v) in ascending order
  - If adding (u, v) does not create a cycle in T
    - Add (u, v) to T
- Return T

Connected Components

How to determine if the edge will create a cycle???



Sorted Edges:

B↔D, 1	C↔E, 2	C↔B, 3	E↔D, 4	A↔C, 5	C↔D, 9	A↔B, 10

Finalized (in MST):

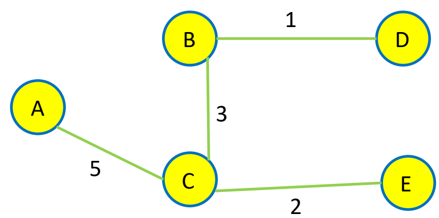
B↔D	C↔E	C↔B	A↔C

FIT2004: Seminar 5 - Greedy (Graph) Algorithms

Kruskal’s Algorithm

Kruskals(G(V, E))
- Sort the edges in ascending order of weights
- Let T be a graph with V as its vertices, and no edges
- For each edge (u, v) in ascending order
  - If adding (u, v) does not create a cycle in T
    - Add (u, v) to T
- Return T

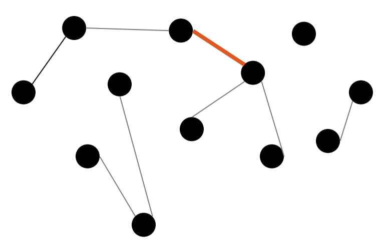
Connected Components



Finalized (in MST):	B↔D	C↔E	C↔B	A↔C

Kruskal’s Algorithm

At some point in the algorithm, we want to check if an edge can be added.
Can we add the red edge?

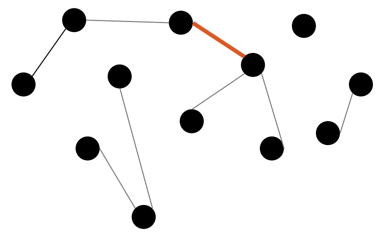


Kruskal’s Algorithm

At some point in the algorithm, we want to check if an edge can be added.
Can we add the red edge?

An edge can be added if it does not create a cycle.
OR
An edge can be added if it is not between two nodes which belong to the same connected component.

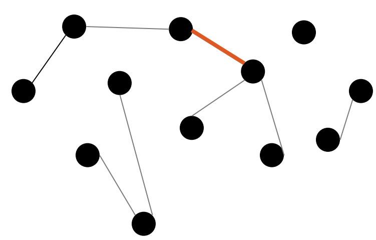
Importantly, adding an edge between two distinct connected components “merges” those connected components into one.



Kruskal’s Algorithm

An edge can be added if it is NOT between two nodes which belong to the same connected component.

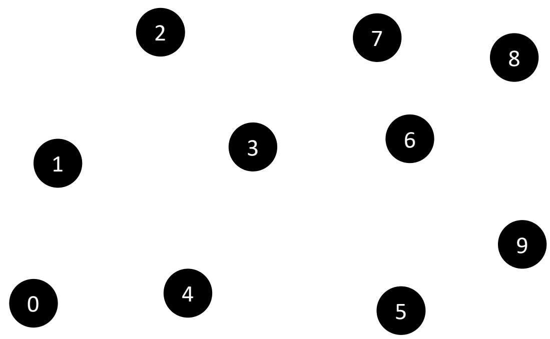
Importantly, adding an edge between two distinct connected components “merges” those connected components into one.



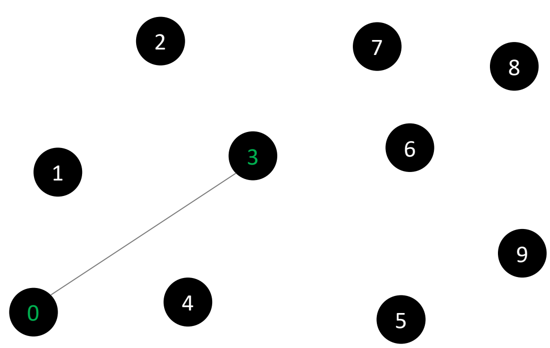
Remember that in Kruskal’s algorithm, each time that one edge e is added to T, e is the smallest edge that links two distinct connected components of T.

Intuitively, Kruskal’s algorithm runs for V-1 iterations, and in each iteration it merges the two distinct connected components that have the smallest distance between themselves.

Kruskal’s Algorithm

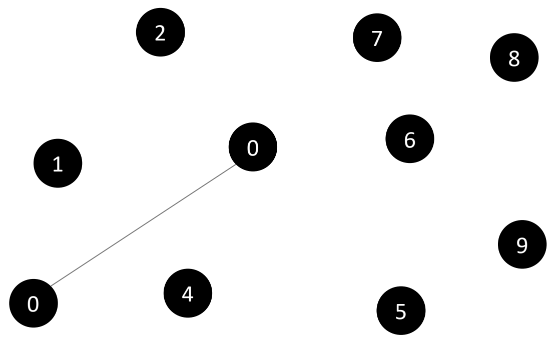


Kruskal’s Algorithm

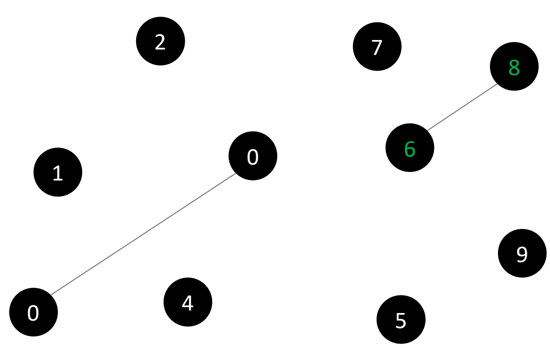


FIT2004: Seminar 5 - Greedy (Graph) Algorithms

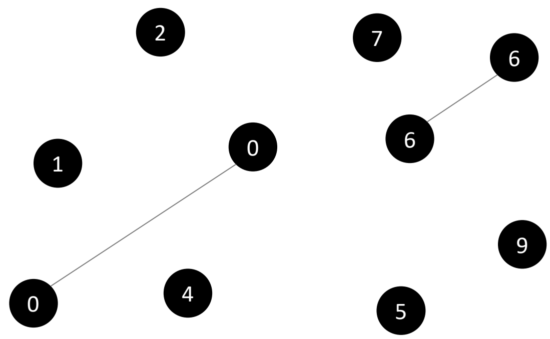
Kruskal's Algorithm



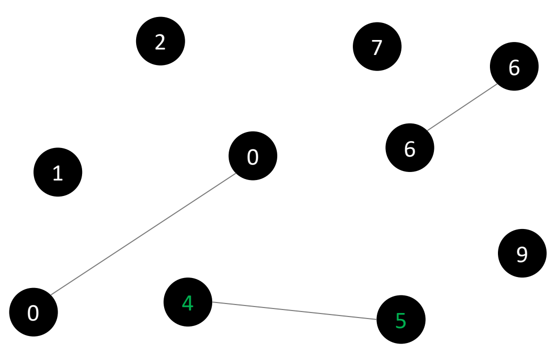
Kruskal's Algorithm



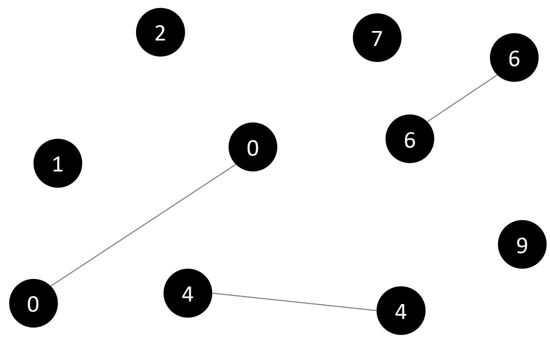
Kruskal’s Algorithm



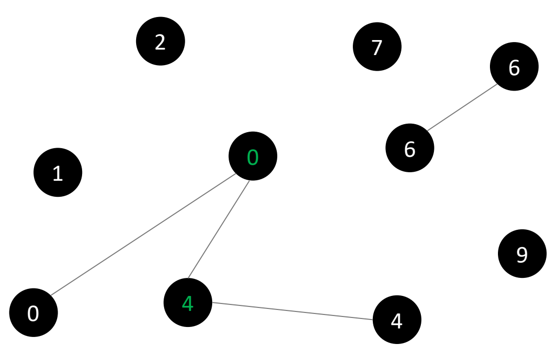
Kruskal’s Algorithm



Kruskal’s Algorithm

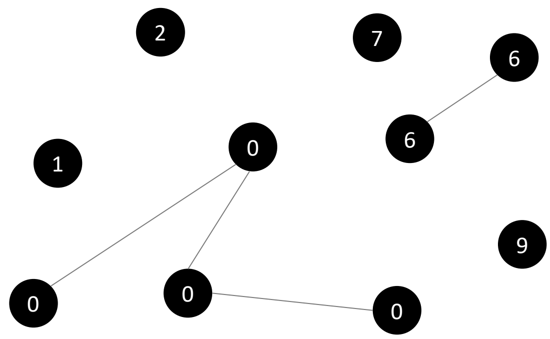


Kruskal’s Algorithm

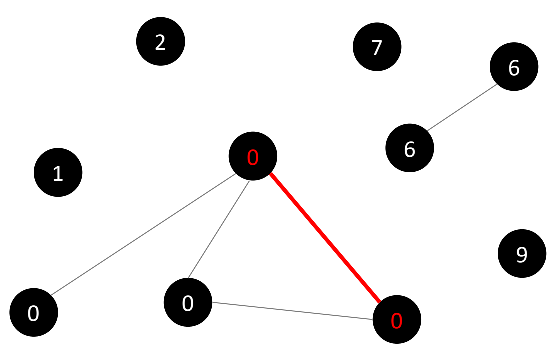


FIT2004: Seminar 5 - Greedy (Graph) Algorithms

Kruskal's Algorithm



Kruskal's Algorithm



Kruskal's Algorithm

Algorithm 41 Kruskal's algorithm

```
function KRUSKAL(G = (V, E))
    sort(E, key((u, v)) = w(u, v))          // Sort edges in ascending order of weight
    forest = UnionFind.initialise(n)
    T = (V, ∅)
    for each edge (u, v) in E do
        if forest.FIND(u) ≠ forest.FIND(v) then          // Ignore edges that would create a cycle
            forest.UNION(u, v)
            T.add_edge(u, v)
    return T
```

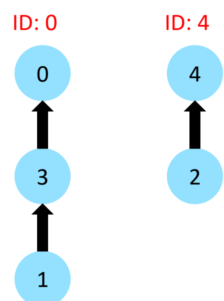
How can we quickly find the set a vertex belongs to, and also union two sets?

Union-Find Data Structure

- Define two operations: **find(u)** and **union(u,v)**
- **Find(u):** Given a vertex u, return its set ID
- **Union(u,v):** Given two vertices u and v, if they have different set IDs, union the two sets they belong to (and update all the set IDs of the vertices in one of the sets)
- We need both **find(u)** and **union(u,v)** to be fast.
  - What would be the time complexity of find(u) and union(u,v) if we store the set ID of each vertex in an array?
    - **Find** would be θ(1), but **union** would be θ(V) in the worst case, thus not good enough…

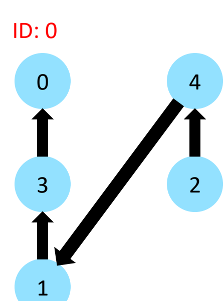
Union-Find Data Structure

- We want to union faster
- Linked lists are fast to union (i.e. append one linked list to another)



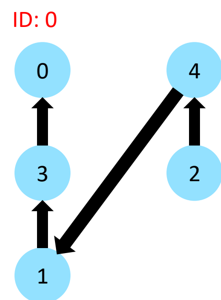
Union-Find Data Structure

- We want to union faster
- Linked lists are fast to union (i.e. append one linked list to another)



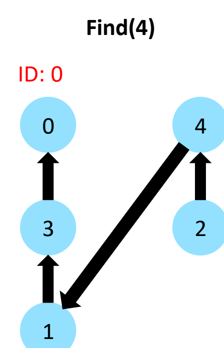
Union-Find Data Structure

- We want to union faster
- Linked lists are fast to union (i.e. append one linked list to another)
- How do we find, with linked lists?



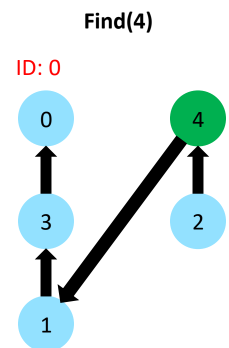
Union-Find Data Structure

- Linked lists are fast to union (i.e. append one linked list to another)
- How do we find, with linked lists?
- Heads know their set ID
- Traverse to the head to find the ID of an element in the list
- This is Θ(size of the linked list)



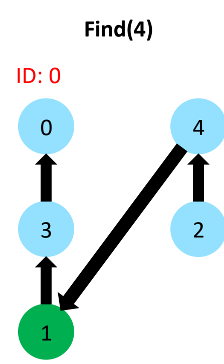
Union-Find Data Structure

- Linked lists are fast to union (i.e. append one linked list to another)
- How do we find, with linked lists?
- Heads know their set ID
- Traverse to the head to find the ID of an element in the list
- This is θ(size of the linked list)



Union-Find Data Structure

- Linked lists are fast to union (i.e. append one linked list to another)
- How do we find, with linked lists?
- Heads know their set ID
- Traverse to the head to find the ID of an element in the list
- This is θ(size of the linked list)

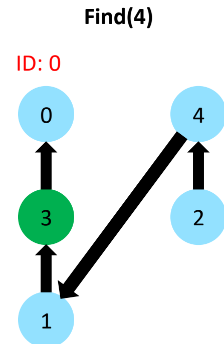


FIT2004: Seminar 5 - Greedy (Graph) Algorithms

Union-Find Data Structure

- Linked lists are fast to union (i.e. append one linked list to another)
- How do we find, with linked lists?

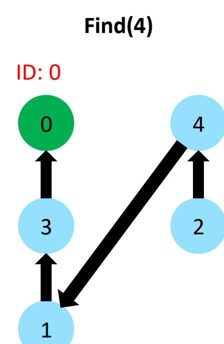
- Heads know their set ID
- Traverse to the head to find the ID of an element in the list
- This is θ(size of the linked list)



Union-Find Data Structure

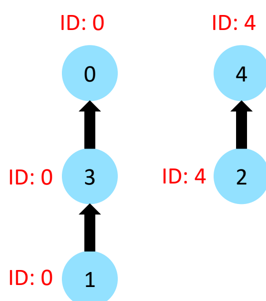
- Linked lists are fast to union (i.e. append one linked list to another)
- How do we find, with linked lists?

- Heads know their set ID
- Traverse to the head to find the ID of an element in the list
- This is θ(size of the linked list)



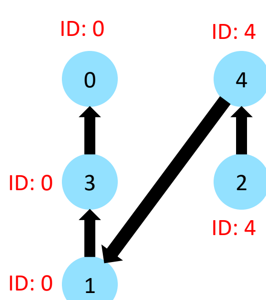
Union-Find Data Structure

- Alternatively, every node could know its ID
- Now find is θ(1)
- But Union is now slower, we have to update all the IDs



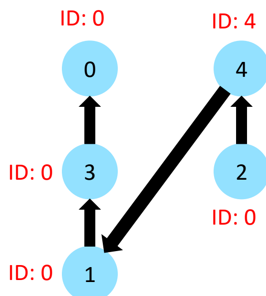
Union-Find Data Structure

- Alternatively, every node could know its ID
- Now find is θ(1)
- But Union is now slower, we have to update all the IDs



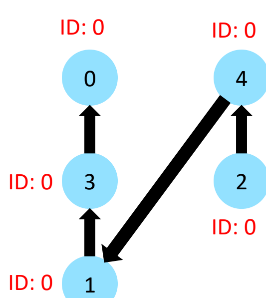
Union-Find Data Structure

- Alternatively, every node could know its ID
- Now find is θ(1)
- But Union is now slower, we have to update all the IDs



Union-Find Data Structure

- Alternatively, every node could know its ID
- Now find is θ(1)
- But Union is now slower, we have to update all the IDs

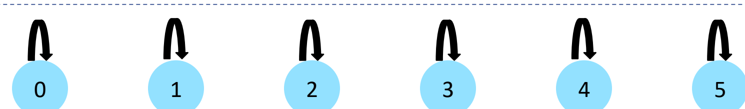


Union-Find Data Structure

- Where are we?
- We want find and union operations to be fast
- Linked lists are an improvement over arrays, since they stop us looking at items which are not relevant to the union we are currently doing
- Linked lists allows θ(1) union
- We can't make find in θ(1) since we have to store the ID at every node which makes union slow (we have to change all the IDs)
- Solution: Change from linked list to linked tree

Union-Find Data Structure

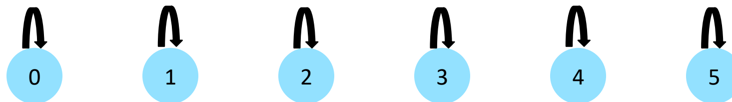
Operations:



Vertex ID	0	1	2	3	4	5
Parent	0	1	2	3	4	5

Union-Find Data Structure

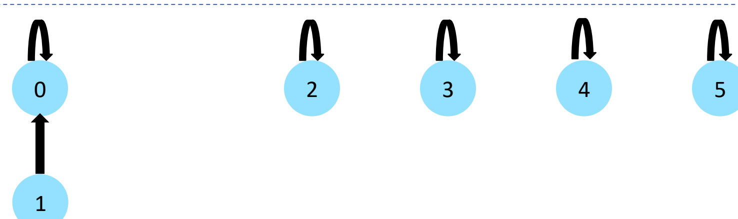
Operations:
Union(0,1)
Find(0) = 0
Find(1) = 1



Vertex ID	0	1	2	3	4	5
Parent	0	1	2	3	4	5

Union-Find Data Structure

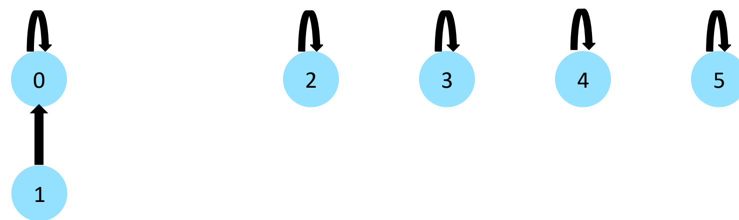
Operations:
Union(0,1)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	3	4	5

Union-Find Data Structure

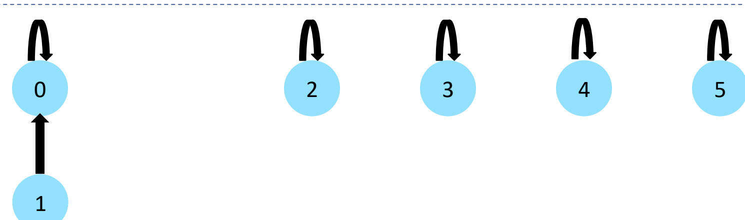
Operations:
Union(0,1)
Union(2,3)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	3	4	5

Union-Find Data Structure

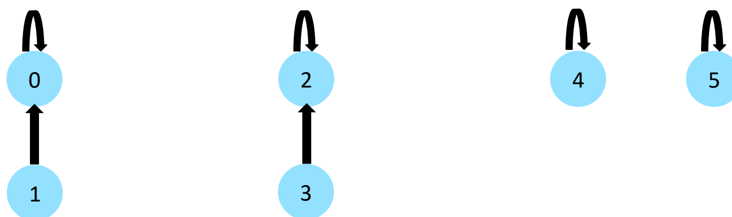
Operations:
Union(0,1)
Union(2,3)
Find(2) = 2
Find(3) = 3



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	3	4	5

Union-Find Data Structure

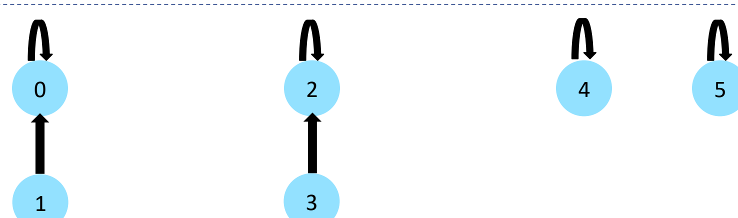
Operations:
Union(0,1)
Union(2,3)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

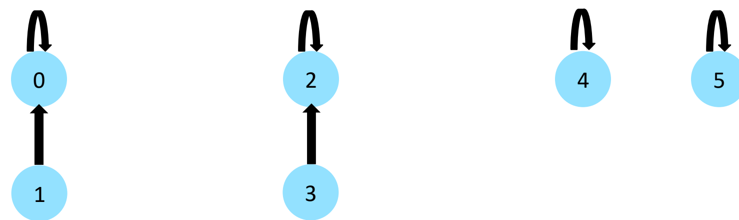
Operations:
Union(0,1)
Union(2,3)
Find(3)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

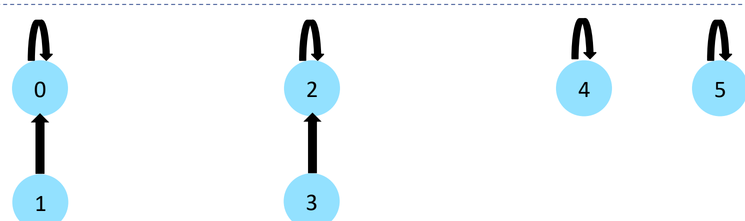
Operations:
Union(0,1)
Union(2,3)
Find(3)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

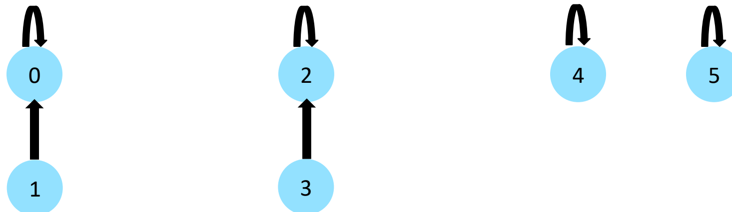
Operations:
Union(0,1)
Union(2,3)
Find(3)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

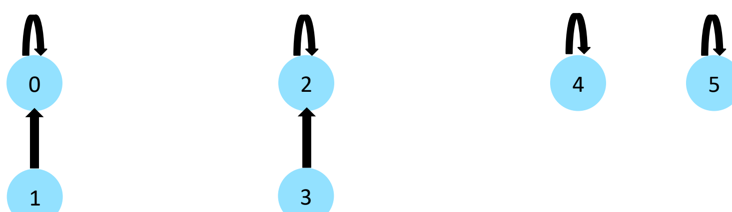
Operations:
Union(0,1)
Union(2,3)
Find(3)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

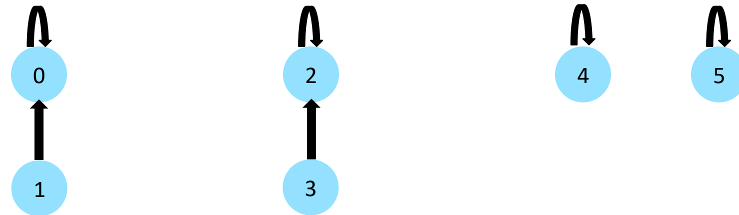
Operations:
Union(0,1)
Union(2,3)
Find(3)=2



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

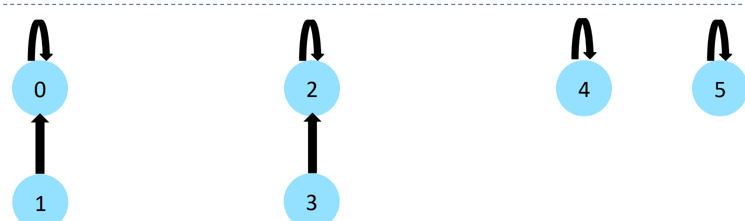
Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

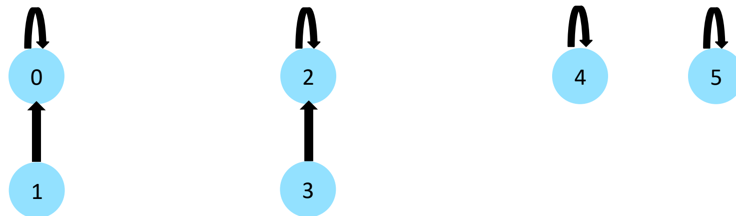
Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
Find(0) = 0
Find(1) = 0



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

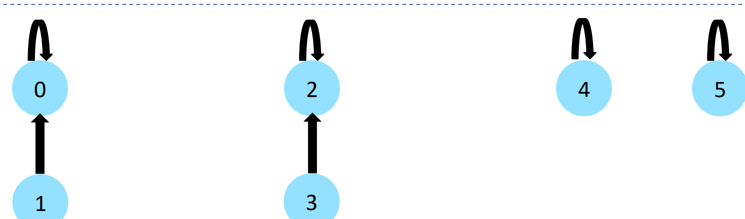
Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
Union(1,3)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
Union(1,3)
Find(1) = 0
Find(3) = 2



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

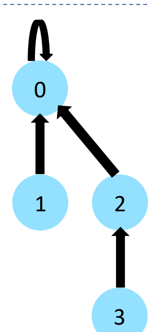
Union-Find Data Structure

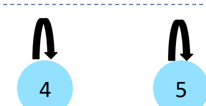
Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
Union(1,3)
  Find(1) = 0
  Find(3) = 2

Vertex ID	0	1	2	3	4	5
Parent	0	0	0	2	4	5

Union-Find Data Structure

Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
Union(1,3)
  Find(1) = 0
  Find(3) = 2





Find: traverse parent pointers until you find a vertex who is its own parent (i.e. a root). That vertex ID is the set ID

Union(u,v): If find(u) != find(v), set parent[find(u)] = find(v) (or vice versa)

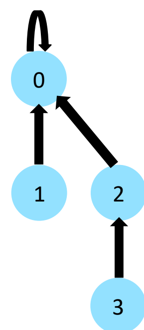
What is the complexity of union and find?
Union is θ(find). Find could in theory be θ(V), but if we can keep the heights of the trees low, then it will be at most θ(max height)

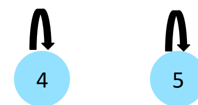
Vertex ID	0	1	2	3	4	5
Parent	0	0	0	2	4	5

FIT2004: Seminar 5 - Greedy (Graph) Algorithms

Union-Find Data Structure

Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
Union(1,3)
  Find(1) = 0
  Find(3) = 2





Optimisation: When we union, we have a choice of the new root.

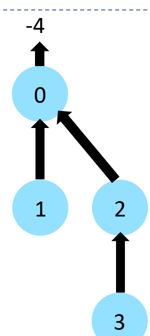
What should we choose?

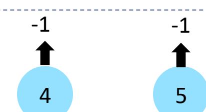
The set with more nodes!

Vertex ID	0	1	2	3	4	5
Parent	0	0	0	2	4	5

Union-Find Data Structure

Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
Union(1,3)
  Find(1) = 0
  Find(3) = 2





Optimisation: When we union, we have a choice of the new root.

What should we choose?

The set with more nodes!

Union by size: When doing a union, add the sizes and update the parent value of the root appropriately

Vertex ID	0	1	2	3	4	5
Parent	-4	0	0	2	-1	-1

Parent array values are now either parents OR negative sizes

Kruskal's Algorithm: Complexity

Algorithm 41 Kruskal's algorithm

```
function KRUSKAL(G = (V, E))
    sort(E, key((u, v)) = w(u, v))               // Sort edges in ascending order of weight
    forest = UnionFind.initialise(n)
    T = (V, ∅)
    for each edge (u, v) in E do
        if forest.FIND(u) ≠ forest.FIND(v) then   // Ignore edges that would create a cycle
            forest.UNION(u, v)
            T.add_edge(u, v)
    return T
```

But this is not tight. A closer look reveals that the total cost of all UNION_SETS is θ(V log V).

Time Complexity:

- Sorting edges: θ(E log E)
  - E log E <= E log V^2 = 2 E log V so θ(E log V)
- Initialization of union find: θ(V)
- For loop executes θ(E) times
  - SET_ID() takes θ(x) where x is height of the tree
  - UNION_SET() takes θ(1) + 2 finds, so it takes θ(x) where x is the height of the **deeper** of the two trees to be unioned (which could be at most V)
- Total cost: θ(EV)

Complexity of UNION_SETS

- We can show that, when using the **union-by-size** rule, the number of elements, N, in any tree is at least 2^h, where h is the height of the tree.
- In other words, h ≤ $\log_2$ N.

- Union needs 2 find operations + Θ(1) effort.
- Find needs an effort equal to the height of the tree, which is ≤ $\log_2$ N.
- So Union is Θ($\log_2$ N) where N is the number of nodes in the taller tree being unioned.

- We need to do V-1 unions in total.
- Each one has a worst case cost of Θ(log V) (this is a significant over-estimation, but its makes the maths easy).
- So the **total cost of all unions** is bounded by **Θ(V log V)**.

Kruskal's Algorithm: Complexity

Algorithm 41 Kruskal's algorithm

```
function KRUSKAL(G = (V, E))
    sort(E, key((u, v)) = w(u, v))            // Sort edges in ascending order of weight
    forest = UnionFind.initialise(n)
    T = (V, ∅)
    for each edge (u, v) in E do
        if forest.FIND(u) ≠ forest.FIND(v) then    // Ignore edges that would create a cycle
            forest.UNION(u, v)
            T.add_edge(u, v)
    return T
```

Time Complexity:

- Sorting edges: θ(E log E)
  - E log E = E log V^2 = 2 E log V so θ(E log V)
- Initialization of union find: θ(V)
- For loop executes θ(E) times
  - The two finds take the same effort as the union, log(v)
- UNION takes θ(V log V) in total
- Total cost: θ(E log V + V log V) = θ(E log V)

Complexity of UNION_SETS

- We can improve the disjoint sets data structure significantly with 2 other optimisations:
  - Union by rank
  - Path compression
- These are discussed in the notes, but are not examinable.
- The complexity can be improved from V log V to Vα(V), where α denotes the inverse Ackermann function, an **extremely** slow growing function.
- Note: α(any number which can be represented using the matter in the universe) < 5, so Vα(V) is effectively θ(V).

Kruskal’s Algorithm: Correctness

- The proof of correctness for Kruskal's algorithm can be done following the same approach as Prim's one. Please try to write it down by yourself and check the details on the course notes.

Comparison- Greedy Algorithms

	Dijkstra	Prim’s	Kruskal
Purpose	Finds shortest path from a single source to all vertices	Finds Minimum Spanning Tree (MST)	Finds Minimum Spanning Tree (MST)
Graph Type	Weighted (both directed & undirected)	Weighted, connected, undirected	Weighted, connected, undirected
Approach	Greedy algorithm using priority queue	Greedy algorithm (like Dijkstra)	Greedy algorithm using sorting
Process	Expands the closest vertex and updates distances	Expands the closest vertex and adds it to MST	Sorts edges & adds the smallest **non-cyclic** edge to MST
Data Structures Used	Priority Queue (Min-Heap), Adjacency List/Matrix	Priority Queue (Min-Heap), Adjacency List/Matrix	Disjoint Set (Union-Find), Edge List
Time Complexity	$O((V+E)\log V)$ with a priority queue (simplified to *O(E log V))*	$O((V+E)\log V)$ with a priority queue (simplified to *O(E log V))*	*O(V log V)* or *O(VαV)* where α is inverse Ackermann function

Reading

- Course Notes: Chapter 6

- You can also check algorithms' textbooks for contents related to this lecture, e.g.:
  - CLRS: Chapter 23, Section 24.3
  - KT: Chapter 4

Concluding Remarks

Take home message

- Dijkstra's algorithm is a greedy algorithm for determining shortest paths on graphs with non-negative weights.
- Prim's and Kruskal's algorithms are both greedy algorithms that correctly determine minimum spanning trees. While Prim's algorithm keeps growing the same connected component at each iteration; Kruskal's algorithm merges the two closest connected components.

Things to do (this list is not exhaustive)

- Make sure you understand:
  - The algorithms, and how to implement Union-Find data structure for Kruskal's algorithm.
  - How proofs of correctness for greedy algorithms work.

Coming Up Next

- Dynamic Programming