Edge-Weighted Graphs

1. Definition:
   - Each edge has a weight associated to it. We denote the weight of an edge by $w(e)$. Usually, $w(e) \geq 0$, but it can also be less than 0.
   - An edge-weighted graph consists of:
     1. A set of vertices, V.
     2. A set of edges, E.
     3. Weights: A map from edges to numbers. $w: E \to R$

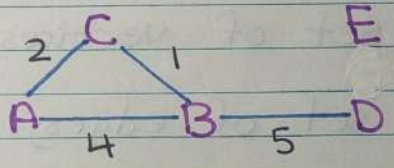
        Note: For undirected graphs, $\{v,u\} = \{u,v\}$, so they have the same weight.

        Note: For directed graphs, (u,v) and (v,u) may have different weights.

     4. Notations: In addition to $w(e)$, we may also use $w(u,v)$ and $weight(u,v)$ to denote the weight of an edge.

2. Storing a weighted Graph:

- We may use an adjacency list or an adjacency matrix to store a weighted graph.
- Example: Consider the weighted graph below.



Adj Matrix:

	A	B	C	D	E
A	0	4	2	∞	∞
B	4	0	1	5	∞
C	2	1	0	∞	∞
D	∞	5	∞	0	∞
E	∞	∞	∞	∞	0

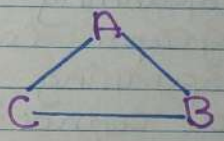
Adj List:

	Adj List
A	(B, 4), (C, 2)
B	(A, 4), (C, 1), (D, 5)
C	(A, 2), (B, 1)
D	(B, 5)
E	

3. Min Cost Spanning Trees

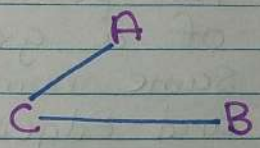
- A spanning tree is a subset of a graph that is a tree which includes all the vertices of the graph.

E.g. Consider the graph below.

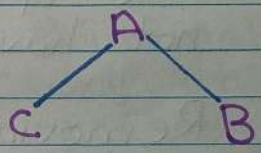


The spanning trees of the graph are:

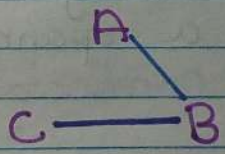
1.



2.



3.



- Properties of Spanning Trees:

1. Spanning trees does not have any cycles and connected be disconnected.

2. Every connected and undirected graph has at least one spanning tree. A disconnected graph can't have any spanning trees. Furthermore, a connected, undirected graph can have more than one spanning trees.

3. All possible spanning trees of a graph have the same number of vertices and edges.

4. The spanning tree does not have any cycles.

5. Removing one edge from a spanning tree will disconnect it. This means that a spanning tree is **minimally connected**.

6. Adding one edge to a spanning tree will create a cycle. This means that a spanning tree is **maximally acyclic**.

7. A spanning tree has n-1 edges where n is the number of vertices.

8. From a complete graph, by removing max e-n+1 edges, we can construct a spanning tree.

9. Both BFS and DFS creates spanning trees.

- A min cost spanning tree (MST) is a spanning tree such that the sum of the weights is the smallest possible.

kruskal's Algorithim

1. Greedy Algorithim

   - The algorithim makes the optimal choice at each step as it attempts to find the overall optimal way to solve the problem.

   - Never reconsiders past choices.

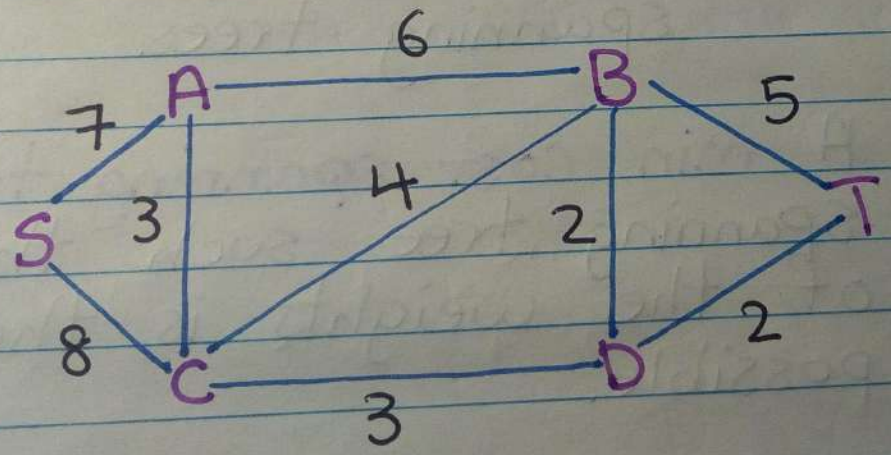
   - In many problems, a greedy strategy does not produce an optimal solution.

Hilroy

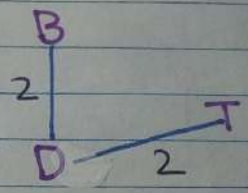
2. Definition and Example

- Kruskal's algorithim grabs the edges with the smallest weights as long as it doesn't create a cycle.

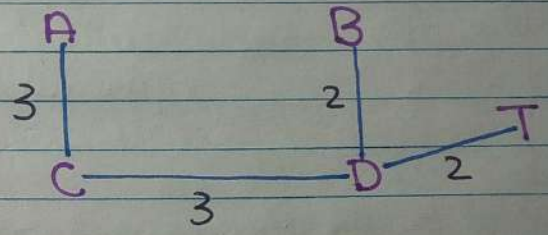
- Example: Consider the graph below.



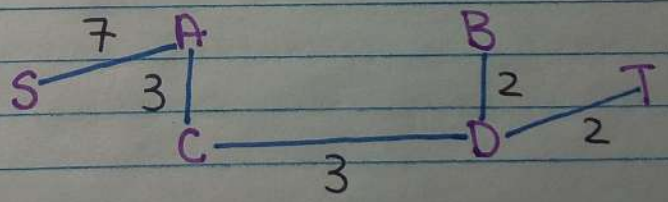
1. Grab the edges with weight 2.



2. Grab the edges with weight 3.



3. Because grabbing the edges of weight 4, 5 and 6 would create a cycle, we skip those. We grab the edge of weight 7. We are done.



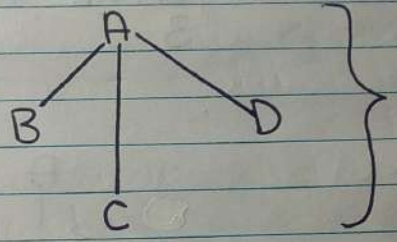
Hilroy

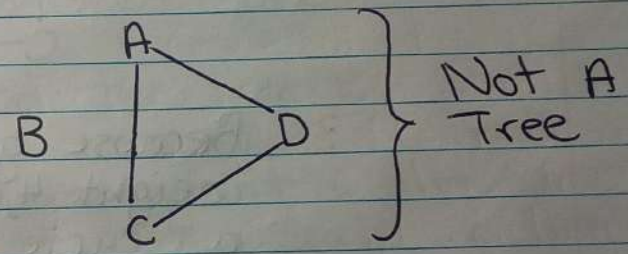
- We know we are done when:

1. We have visited each node.
2. We have n-1 edges.

- Note: If we have a tree, then we must have n-1 edges. However, if we have n-1 edges, it does NOT mean we have a tree.

E.g.





3. Proof of Correctness:

We can prove Kruskal's algorithm is correct by a proof by contradiction.

Proof:

- We order the edges in non-decreasing order of weight.
  I.e. $w_1 \leq w_2 \leq w_3 \dots \leq w_n$
- Let K be the spanning tree returned by Kruskal's algorithm.
- Let O be an optimal MST, s.t. O's weight is less than K's weight. This means that K is not optimal.
- Let $e_i = (u, v)$ be the first edge in our ordering that is not in both K and O.
  Note: $e_i \in K$ and $e_i \notin O$ because K only omits edges if they create a cycle. Since O is a MST, it can't have any cycles. Therefore, it is not possible for $e_i \notin K$ and $e_i \in O$.

Hilroy

- Since O is connected, there must exist a unique path p from u to v and an edge e' on p that is not in K.

- Since K didn't choose e', but had the option to, this means $w(e') \geq w_i$.

  Case 1: $w(e') = w_i$. Then, we can switch e_i and e' and O still has the same weight as before but is more similar to K. Repeat this argument until either **Case 2** or the 2 trees are the same and K is optimal.

  Case 2: $w(e') > w_i$. Now, consider a tree O' constructed by removing e' from O and adding e_i. Now, O' has less weight than O contradicting that K and O differ. Therefore, K must be optimal.

4. Implementation and Complexity

- We can store the edges sorted in non-decreasing weight in a min priority-queue.

- To add edges and to make sure that no cycle is induced, we can use linked lists. Think of it as joining together clusters (subtrees) of connected vertices. Each cluster is a linked list.

- To make sure that adding an edge does not induce a cycle, check to see if the vertex the new edge points to is in the linked list or not. If it is, then adding the edge would create a cycle. If it isn't, then adding the edge wouldn't create a cycle.

- Merging 2 clusters is merging 2 linked lists. However, a lot of vertices need their cluster pointers updated. If we move the smaller linked list to the bigger one, updating each vertex takes $O(\lg n)$ time, at most. In total, it takes $O(n \lg n)$.

- Building PQ and removing edges takes $\Theta(m \lg m)$
- Updating all the vertices takes $O(n \lg n)$.
- The rest is $\Theta(1)$ per edge or vertex.
- In total, it takes $O(m \lg m + n \lg n)$, but because $m \leq n^2$, $\lg m \in O(\lg n)$.

  $\therefore$ The time complexity is $O((m+n) \lg n)$

Prim's Algorithm:

1. Algorithm and Example:
   - Finds an MST by doing something similar to BFS.
   - Uses a min PQ
   - The algorithm grows a tree T one edge at a time.
   - The priority of vertex v is the smallest edge weight between v and T, so far. We use ∞ if there is no such edge.

2. Complexity

- Every vertex enters and leaves the min-heap once, so it takes $\Theta(\lg n)$ per vertex. In total, it takes $\Theta(n \lg n)$.
- Decreasing all the vertice's priorities takes $O(m \lg n)$ time.
- In total, it takes $O((m+n) \lg n)$ time.

3. Proof of Correctness:

1. Cut Property

- Let S be a nontrivial subset of V in G.
  I.e. $S \neq \emptyset$ and $S \neq V$
  If (u, v) is the lowest-cost edge crossing $(S, V-S)$, then (u, v) is in every MST of G.
- Proof:
  - Suppose there exists an MST T that does not contain (u, v).
  - Consider the sets S and $V-S$. There must be a path from u to v.
  - On this path, there must exist an edge e that crosses between S and $V-S$.

- Since (u, v) is the least weighted edge crossing between S and $V-S$, swapping e with (u, v) will reduce the weight of T.
- $\therefore$ T is not an MST.

2. Proving Prim's Algorithm
   - Proof:
     - Consider an optimal MST O and Prim's Algorithm's tree T.
     - Order the edges of T and O in the order they were selected.
     - Consider the first edge $e = (u, v)$ in that ordering that is in T and not in O.
     - At the stage of Prim's Algorithm when e was added, there was a set S of vertices s.t. $u \in S$ and $v \in V-S$.
     - If the edge weights are unique, then by the Cut Property, e must belong to O.

- If the edge weights are not unique and if $e \notin O$, that means there exists a path p from v to v s.t. an edge $e' = (x, y)$ exists on p and $x \in S$ and $y \in V - S$.

- If $w(e) = w(e')$, then we can swap e with e' and the spanning tree will still be optimal.

- It is impossible for $w(e')$ to be less than $w(e)$ or Prim's Algorithm would have chosen it.

- If $w(e) < w(e')$, then swapping e' with e reduces the weight of O, which is a contradiction.

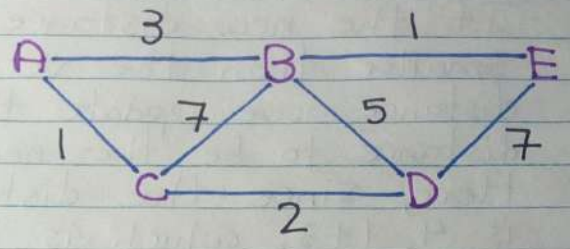
Dijkstra's Algorithm

1. Definition and Example

- Dijkstra's algorithm finds the shortest path between 2 nodes.

- It is very similar to Prim's algorithm.

- E.g. Consider the graph below.



1. Start at any vertex. For this example, I'll start at C.

Vertex	C	B	A	D	E
Priority	0	∞	∞	∞	∞
Pred					

2. We find the distance of all the neighbours of C. To get the distance, you add the priority of the current node to the weight of the edge that takes you to the neighbour. After, you remove the current node from the min-heap and go to the neighbour with the smallest priority.

Vertex	~~C~~	A	D	B	E
Priority	~~0~~	1	2	7	∞
Pred		C	C	C	

We move to A and remove C from the stack.

Hilroy

A

3. At , we find the distances of all its unvisited neighbours. If the new distance is smaller than the vertex's old distance, we update that vertex's distance to be the new distance. Here, since the distance to B is 4, 1+3, which is lower than 7, we replace B's distance with 4. After we are done, we remove A from the min-heap and go to the vertex with the lowest priority. In our case, it's D.

Vertex	~~C~~	~~A~~	D	B	E
Priority	0	1	2	4	∞
Pred		~~C~~	C	A	

4. At D, we find the distance of all of its unvisited neighbours. If the new distance is smaller than the old distance, we update the node's distance to the new distance. After we are done, we remove D from the min-heap and go to the vertex with the lowest priority.

Vertex	~~C~~	~~A~~	~~D~~	B	E
Priority	0	1	2	4	9
Pred		~~C~~	~~C~~	A	D

5. At B, we find the distance of all of its unvisited neighbours and update their distance, accordingly. Then we remove B from the min-heap and go to the node with the smallest priority.

Vertex	~~C~~	~~A~~	~~D~~	~~B~~	E
Priority	~~0~~	~~1~~	~~2~~	~~4~~	5
Pred		~~C~~	~~C~~	~~A~~	B

6. Since E does not have any unvisited neighbours, we are done.

2. Algorithm:

1. We add our start vertex s to the set of reached vertices S and give it distance d[s] = 0. This creates a distance tree rooted at s.

2. At each stage we consider the next closest vertex to s from vertices not in S, or alternatively, the vertex with the next shortest path to s.

3. We can use a PQ to determine the shortest path by when a new vertex v is added to S, we consider each neighbour u of v s.t. $u \notin S$. If we get a smaller priority for u, we update u's priority to the new priority.

3. Proof of Dijkstra's Algorithm

- Let Ts be the distance tree constructed by Dijkstra's algorithm, starting at s.
- Let Os be an optimal distance tree rooted at s.
- Order the edges $\langle e_1, e_2, \ldots e_m \rangle$ according to how they are added to Ts.
- Consider the first edge $e_i = (u, v)$ s.t. $e_i \in Ts$ and $e_i \notin Os$.
- Then, $e_1, \ldots e_{i-1} \in Ts$. Let S be the set of vertices added so far. I.e. $\langle e_1, \ldots e_{i-1} \rangle$
- Each node in S has a min path distance to s.

- Since $(u,v) \notin O_S$, there must be a shorter path p from s to v.
- Consider the edge $e_j = (x,y)$, $j > i$ on p that has one endpoint in S and one in $V-S$.

Case 1: $y \neq v$. $d_O[y] < d_T[v]$
This is a contradiction because Dijkstra's alg would have chosen it.

Case 2:

A: If $y = V$ and $d_O[y] < d_T[v]$, then this is a contradiction because Dijkstra's alg would've chosen it.

B: If $y = V$ and $d_O[y] = d_T[v]$, we can swap (x,y) with (u,v) and O_S is closer to T_S.