Minimum Spanning Trees

Tree: A Graph or a subset of a graph that does not have any cicles (ie is acyclic).

Minimum Spanning Trees: A minimum spanning tree is a susbset of a weighted undirected graph such that the total weight of the graph is minimized.

Let $G = V, E$ be a graph, and for each edge $(u, v) \epsilon E$ a weight $w(u, v)$ is specified. The goal of the minimum spanning tree problem is to find an acyclic subset of the weighted graph $T \subset G$ that connects all the vertices and whose total weight

$$w(T) = \sum_{(u,v)\epsilon T} w(u, v)$$

is minimized.

Here, we discuss two possible solutions for the minimum spanning tree problem, namely the Kruskal's Algorithm and the Prim's Algorithm. Both the algorithms are **greedy algorithms**. Each step of a greedy algorithm must make one of several possible choices. The greedy strategy advocates making the choice that is the best at the moment. Such a strategy does not generally guarantee that it always ûnds globally optimal solutions to prob- lems. For the minimum-spanning-tree problem, however, we can prove that certain greedy strategies do yield a spanning tree with minimum weight.

Generic MST Algorithm

The input to the minimum-spanning-tree problem is a connected, undirected graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$. The goal is to find a minimum spanning tree for G.

This greedy strategy is captured by the procedure GENERIC-MST, which grows the minimum spanning tree one edge at a time.

```
GENERIC-MST(G,w)
A = empty array
while A does not form a spanning tree:
    find an edge (u,v) that is safe for A
    A = A U {(u ,v)}
return A
```

The generic method manages a set A of edges, maintaining the following loop invariant:

`Prior to each iteration, $A$ is a subset of some minimum spanning tree.`

Initialization: The loop invariant is satisfied trivally after line 1.

Maintainence: The loop in lines 2-4 maintains the invariant by only adding the safe edges.

Termination All edges added to A belong to a minimum spanning tree, and the loop must terminate once all the edges are considered. Therefore the set A returned in line 5 must be a minimum spanning tree.

The challenge comes from how the safe edges are found in line 3. For this, we have a theorem that can help us find safe edges.

First we need some definitions.

- A **cut** $S; V \setminus S$ of an undirected graph $G = (V, E)$ is a partition of V.
- We say that an edge $(u, v) \in E$ **crosses the cut** $S; V \setminus S$ if one of its endpoints belongs to S and the other belongs to $V \setminus S$.
- A cut **respects a set** A of edges if no edge in A crosses the cut.
- An edge is a **light edge crossing a cut** if its weight is the minimum of any edge crossing the cut. There can be more than one light edge crossing a cut in the case of ties.

- More generally, we say that an edge is a light edge satisfying a given property if its weight is the minimum of any edge satisfying the property.

> ***Theorem***: Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V \setminus S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V \setminus S)$. Then, edge (u, v) is safe for A.

The Kruskal's And Prim's algorithms elaborate on the generic method outlined above, and the correctness argument carries over. Both algorithms assume the input graph is connected and is represented by an adjacency list.

Kruskal's Algorithm

In brief, Kruskal's algorithm works by iteratively selecting the edges of the least weight from the sorted list and adding them to the tree if they do not create a cycle. It begins by sorting all the edges in ascending order of weight. Then, for each edge in this sorted list, it checks whether adding the edge to the current set of edges forms a cycle or not. If it does not create a cycle, the edge is included in the minimum spanning tree. This process continues until all vertices are included in the tree or until the number of edges reaches $|V| - 1$, where $|V|$ is the number of vertices in the graph.

Pseudocode

The pseudocode of the algorithm is given below.

```
MST-KRUSKAL(G, w)
1. A ← ∅
2. for each vertex v in G.V
3.     MAKE-SET(v)
4.create a single list of the edges in G: E
5.sort the list of edges into monotonically increasing order
6.for each edge (u, v) taken from the sorted list in order
7.     if FIND-SET(u) ≠ FIND-SET(v)
8.         A ← A ∪ {(u, v)}
9.         UNION(u, v)
10. return A
```

Explanation of the Algorithm

The algorithm begins by cosnidering a set of all the edges and all the vertices taken from the graph. It also initializes the tree in the variable A which is an empty set.

1. The algorithm sorts the edges by the increasing order of weights.
2. Takes the edge of least weight, and checks if it is a safe edge. If it is, it adds it to the Tree
3. Step 2 is iterated upon for all the edges in the increasing order of edge weights.

In step 2, it checks if the edge is safe or not by checking if the Find-Set of both vertices of the edge are the same. Given the Tree A, The Find-Set operation essentially returns the set of all the vertices that are reachable from the input vertex v in the tree A. For an edge (u, v), if the Find-Set operation returns the same set for both u and v, then they both are already reachable from each other, without including this edge. So the inclusion of this edge would mean that a cycle is formed, and the edge is not a safe edge to add to the tree.

Prim's Algorithm

In Brief, Prim's algorithm starts with an arbitrary vertex and gradually grows the minimal spanning tree by selecting edges of minimum weight. By iteratively updating vertex properties and exploring neighboring vertices, it constructs the MST rooted at the chosen starting vertex.

Pseudocode

The pseudocode of the algorithm is given below

```
MST-PRIM(G, w, r)
1. // Initialize each vertex's parent and key values
2. for each vertex v in G:
3.     v.pi = NIL  // Initialize the parent of each vertex t
4.     v.key = infinity  // Initialize the key (priority) of
5.
6. r.key = 0  // Set the key of the root vertex r to 0.
7. Q = Vertices of G  // Initialize a priority queue Q with
8.
9. // Main loop for constructing the MST
10. while Q is non-empty:
11.     u = extract min(Q)  // Extract the vertex with minir
12.
13.     // Iterate over all adjacent vertices of u
14.     for each vertex v in G.Adj(u):
15.         // Check if v is in Q and if the weight of edge
16.         if v is in Q and w(u, v) < v.key:
17.             v.pi = u  // Update the parent of v to u.
18.             v.key = w(u, v)  // Update the key of v to t
```

Explanantion of the Algorithm

In lines 1 -7, the code initializes the priori and the key of each of the vertices. By default, the vertices have no priori and have a weight of infinity. The root of the queue chosen (typically taken randomly, as the MST has all the vertices anyways), is chosen to have a key of zero. The variable Q has all the vertices of the graph stored in them.

In the main loop of the algorithm (line 8), first the vertex with the minimum key is extracted (removed from Q) and assigned to u. One by one, the adjacent vertices of u are checked.

If an adjacent vertex v is in the set Q and the weight of u, v is less than the `v.key`, then it proceeds to update the value priori and the key of v. The priori is now set to u and the key is set to the weight of u, v. The inutition is, if this above condition is not satisfied, then the vertex is connected to a different vertex with which it has a smaller weighted edge. If the condition is satisfied, then u is a better choice to be parent of v.

The process is repeated till all the vertices are checked, and removed from the Q.

Properties of Minimal Spanning Trees

For the following discussion, let the number of vertices in G be n and number of edges be m.

1. A MST has $n-1$ edges.

   Adding any more edges would create a cycle, and thus violate the acyclic nature of the tree. Regardless of the specific algorithm used to find the MST, the resulting MST will always contain $n-1$ edges.

   This also implies that there are $m-n+1$ edges not a part of the MST

2. If we add an edge to a spanning tree, the resultant graph becomes a cycle.

   Since a spanning tree is a tree that includes all vertices of GG, it must also have nn vertices. If we add an edge to a spanning tree, it will create a cycle.

3. Removal of an edge from a spanning tree will disconnect the graph

   A spanning tree ensures that the graph remains a single connected component, which means that there is a path between every pair of vertices in the graph. When an edge is removed from the spanning tree, the graph will split into two disconnected components.

4. Given a graph, there may be multiple minimal spanning trees

   Given two edges have the same weight, then it is possible that the algorithm may encounter a tie. Depending on how the tie-breaker is resolved, the resulting MST will be different, even though the end result is an MST of the same weight.

5. If each edge has a distinct weight, then the resulting MST will be unique for a given weighted graph.

   There would not be any tie-breaker situtations here, and will not have any deviations that lead to distinct MSTs.

6. Cycle property of MST

   For any cycle C in the graph, if the edge weight of e is larger than all the other edges of the graph, then the MST will not contain the edge e.

7. Cut Property of MSTs:

> For any cut C in the graph, if the weight of an edge in the cut set is strictly smaller than all the other weights of edges in the cut-set, then this edge must be a part of the MST.
>
> Basically this property is identical to the theorem mentioned above.

Heaps or Priority Queues

A **Binary Max-Heap** is a complete binary tree where every parent node's value is greater than the child node's value.

A **Complete Binary Tree** is a Binary Tree, that is fully filled except for the last level, and is filled from left to right in the last level.

A complete binary tree is represented as an array (index starting from 1), where the left child of the ith entry is at 2i and the right child is at 2i + 1. The parent node will be given by floor(i).

Max-Heapify Algorithm

The max-heapify algorithm is a procedure used in the context of binary heap data structures, specifically in the case of max-heaps.

In a max-heap, every parent node has a value greater than or equal to the values of its children nodes. Max-heapify ensures that a given subtree rooted at a specified node follows the max-heap property.

Here's a brief explanation of the max-heapify algorithm:

1. **Input**:
    - The input to max-heapify consists of an array representing the binary tree (often implemented as an array) and the index of the node at which to begin max-heapifying.
2. **Procedure**:
    - Max-heapify works recursively, assuming that the subtrees rooted at the left and right children of the current node already satisfy the max-heap property.
    - It compares the value of the current node with the values of its left and right children.
    - If the value of the current node is less than the value of its largest child, it swaps the current node's value with that of its

largest child.
    - After the swap, it recursively applies max-heapify to the affected child node to ensure the max-heap property is maintained downwards.
3. **Termination**:
    - Max-heapify continues recursively until the subtree rooted at the current node satisfies the max-heap property, or until it reaches leaf nodes (which trivially satisfy the max-heap property).
4. **Time Complexity**:
    - The time complexity of max-heapify is (O(\log n)), where (n) is the number of nodes in the subtree being heapified.

In summary, max-heapify is a recursive algorithm that adjusts the structure of a binary tree to ensure it adheres to the max-heap property, which is essential for efficient operations on heaps, such as extracting the maximum element or heapifying an entire array into a max-heap.

```
Max-Heapify(A, i):
1. // Calculate the index of the left child of node i
2. left = 2*i
3. // Calculate the index of the right child of node i
4. right = 2*i + 1
5. // Assume the current node i has the largest value initia
6. largest = i
7. // Check if the left child exists and is larger than the
8. if left <= A.heap-size and A[left] > A[largest]:
9.     largest = left
10. // Check if the right child exists and is larger than th
11. if right <= A.heap-size and A[right] > A[largest]:
12.     largest = right
13. // If the current node is not the largest, swap it with
14. if largest != i:
15.     swap A[i] with A[largest]
16.     // Recursively call Max-Heapify on the affected chil
17.     Max-Heapify(A, largest)
```