1 The Minimum Spanning Tree Problem (MST)

Given a connected graph \( G = (V, E) \), with weight \( w_e \) for all edge in \( E \), find a spanning tree \( G_T = (V_T, E_T) \) of minimum total weight. (Spanning means that \( V_T = V \).)

1.1 IP Formulation

The decision variables for the IP formulation of MST are:

\[
 x_e = \begin{cases} 
 1 & \text{if edge } e \in E_T \\
 0 & \text{otherwise} 
\end{cases}
\]

The constraints of the IP formulation need to enforce that the edges in \( E_T \) form a tree. Recall from Lecture 1 that a tree satisfies the following tree conditions: have \( n - 1 \) edges, be connected, be acyclic. Also remember that if any two of these conditions imply the third one. A possible IP formulation of MST is given in problem 1.

\[
\begin{align*}
\text{min} & \quad \sum_{e \in E} w_e x_e \\
\text{s.t.} & \quad \sum_{e \in E} x_e = n - 1 \\
& \quad \sum_{e \in (S,S)} x_e \leq |S| - 1 \quad \forall \ S \subseteq V \\
& \quad x_e \in \{0,1\} \quad \forall \ e \in E
\end{align*}
\]

where \( (S,S) \) denotes all edges that go from a node in the set \( S \) to another node in the set \( S \).

Equation (1c) enforce the constraint that the edges in \( E_T \) can’t form cycles.

We have the following observations.

1. The constraint matrix of problem (1) does not have a network flow structure, however Jack Edmonds proved that the LP relaxation has integral extreme points.

2. Even though the LP relaxation has integral extreme points, this does not imply that we can solve the MST problem in polynomial time. This is because the formulation has an exponential number of constraints. Nevertheless, we can use the following strategy to solve problem (1).

Relax the set of constraints given in (1c), and solve the remaining problem. Given the solution to such relaxed problem, find which of the relaxed constraints are violated (this process is called separation) and add them. Resolve the problem including these constraints, and repeat until no constraint is violated.

It is still not clear that the above algorithm solves in polynomial time MST. However, in light of the equivalence between separation and optimization (one of the central theorems in optimization theory), and since we can separate the inequalities (1c) in polynomial time, it follows that we can solve problem (1) in polynomial time.
3. An alternative IP formulation of MST is obtained by imposing the connectivity condition (instead of the acyclicity condition). This condition is enforce by the following set of inequalities.
\[
\sum_{e \in (S, \bar{S})} x_e \geq 1 \quad \forall \ S \subset V
\]

### 1.2 Algorithms of MST

The correctness of the two algorithms that we will give in this lecture follows from the following theorem.

**Theorem 1.** Let \( F = \{ (U_1, E_1), \ldots, (U_k, E_k) \} \) be a spanning forest (a collection of disjoint trees) in \( G \). Pick edge \([i, j]\) of minimum weight such that \( i \in U_1 \) and \( j \in U_1^c \). Then some MST containing \( E_1 \cup E_2 \cup \ldots E_k \) will contain edge \([i, j]\).

**Proof.** Suppose no MST containing \( F \) contains edge \([i, j]\). Suppose you are given \( T^* \) not containing \([i, j]\). Add \([i, j]\) to \( E_{T^*} \), this will create a cycle, remove the other edge from \( U_1 \) to \( U_1^c \) that belongs to the created cycle. Since the weight of this removed edge must be greater or equal than that of edge \([i, j]\), this is a contradiction. \( \Box \)

#### 1.2.1 Prim’s Algorithm

Begin with a spanning forest (in the first iteration let each node being one of the “trees” in this forest). Find the shortest edge \([i, j]\) such that \( i \in U_1 \) and \( j \in U_1^c \), add \([i, j]\) to \( T_1 \) and \( j \) to \( U_1 \).

Repeat this process of “growing” \( U_1 \) until \( U_1 = V \).

**begin**
\[ U = \{1\}; \ T = \emptyset. \]
\[ \text{while } U \neq V \text{ do} \]
\[ \quad \text{Pick } [i, j] \in E \text{ such that } i \in U, j \in U^c \text{ of minimum weight} \]
\[ \quad T \leftarrow T \cup [i, j]; P \leftarrow P \cup j \]
**end**

The while loop is executed \( n - 1 \) times, and finding an edge of minimum weight takes \( O(m) \) work, so this naive implementation is an \( O(nm) \) algorithm. However, note that similarity between this algorithm and Dijkstra’s algorithm; in particular note that we can apply the same tricks to improve its running time.

We can make this a better algorithm my maintaining a list of sorted edges leaving \( P \) by inserting and finding edges with binary search or by maintaining the edges leaving \( P \) as a binary heap. Finding a minimum edge takes \( O(1) \), which is \( O(n) \) total.

Each edge is added to the heap when one of its end points enters \( P \), while the other is outside \( P \). It is then removed from the heap when both its end points are in \( P \). Adding an element to a heap takes \( O(\log m) \) work, and deleting an element also takes \( O(\log m) \) work. Thus, total work done in maintaining the heap is \( O(m\log m) \), which is \( O(m\log n) \) since \( m \) is \( O(n^2) \). Thus, total work done in this heap implementation is \( O(m\log n) \).

With Fibonacci heaps, this can be done in \( O(m + n\log n) \). This is the best known complexity for the problem.
1.2.2 Kruskal’s Algorithm

Sort the edges in non-decreasing order of weight: \( w_{e_1} \leq c_{e_2} \leq \ldots c_{e_m} \).

\[
\begin{align*}
\text{begin} & \\
& k \leftarrow 1, i \leftarrow 1, E_T \leftarrow \emptyset \\
\text{while } i \leq n \text{ do} & \\
& \quad \text{let } e_k = [i, j] \\
& \quad \text{if } i \text{ and } j \text{ are not connected in } (V, E_T) \text{ do} \\
& \quad \quad E \leftarrow E \cup [i, j] \\
& \quad \quad T \leftarrow T \cup \{i, j\} \\
& \quad k \leftarrow k + 1 \\
\end{align*}
\]

Complexity: The while loop is executed at most \( m \) times, and checking if \( i \) and \( j \) are in the same component can be done with DFS on \((V, E_T)\), this takes \( O(n) \). Therefore this naive implementation has a running time of \( O(nm) \).

We will now show that Kruskal’s algorithm has a running time of \( O(m \log n) \) (plus the time to sort the edges) without special data structures.

Each component is maintained as a linked list with its size and last element stored, while each node of the component keeps track of the first element in the list. Merging the components is now \( O(1) \), and adds the smaller component to the larger by adding a link.

Checking if \([i, j] \) belong to the same component can be checked just by checking the first element of the lists that \( i \) and \( j \) belong to see if they are the same. This takes \( O(1) \).

The critical operation is updating the “first” element when two lists merge. The first element can be updated in the merged element just by running down the merged list and changing it. The main result is that updating the first element takes \( O(n \log n) \) throughout the algorithm. We note here that when two lists are merged, it is critical that the smaller list is merged to the larger list, i.e., the “first” value of all elements in the smaller list become the “first” value of the larger list after the merger.

Note that we only update the “first” element of a node when it belonged to the smaller of the to list merged. Also when we merge this lists, the nodes in the smaller list will belong to a list of at least twice the size of the smaller list. Since each time we have to update the “first” element of a node, we double the size of the list to which this node belongs, it follows that we can update the “first” element of a node at most \( O(\log n) \) times. We conclude that the total amount of work performed in the merge operation (overall the execution of the algorithm) is \( O(n \log n) \).

1.3 Maximum Spanning Tree

Note that we never assumed nonnegativity of the edge weights. Therefore we can solve the maximum spanning tree problem with the above algorithms after multiplying the edge weights by -1.