Algorithms

Lecture : MST- Kruskal's Algorithm

Imdad Ullah Khan

Contents

1 Introduction **1**

2 Kruskal's Algorithm **2**
2.1 Example Run 2
2.2 Proof of Correctness 3
2.3 Implementation and Runtime 4

3 Union-Find Data Structure **5**
3.1 An Implementation of Disjoint Set Data Structure 5
3.1.1 Union by Rank 6
3.2 Complexity for Kruskal's Algorithm Using Union-Find Data Structures . . . 6

1 Introduction

Kruskal's algorithm is a greedy algorithm for finding minimum-spanning-tree of the given graph.

Input: An undirected weighted graph $G = (V, E), w$, where $w : E \to \mathbb{R}$. Note that unlike Dijkstra weights don't have to be non-negative.

Output: A spanning tree $T = (V, E')$, with $E' \subseteq E$ such that $w(T)$ is minimum among all spanning trees of G.

Note that G has unique spanning tree if the weights on the edges are unique. For the simplicity of the algorithm and understanding we assume that in G, edges have unique weights.

2 Kruskal's Algorithm

In each iteration, add the edge to E' with minimum weight which does not make any cycle. Keep adding the edges until E' has exactly $n-1$ edges.

Note that the graph spanned by chosen edges don't have to be connected in every iteration. Unlike Prim's algorithm, which grows a tree until it spans the whole vertex set, Kruskals algorithm keeps a collection of trees (a forest) which eventually gets connected into one spanning tree.

Algorithm 1 : Pseudocode of Kruskal's Algorithm

```
sort edges in increasing order of weights      ▷ let e_1, e_2, ..., e_m be the sorted order
F ← ∅                                          ▷ Begin with a forest with no edges
for i = 1 to m do
    if F ∪ e_i does not contain a cycle then
        F ← F ∪ {e_i}
return F
```

2.1 Example Run

First, we run this pseudocode on the following graph in Figure 1 as shown in 2.

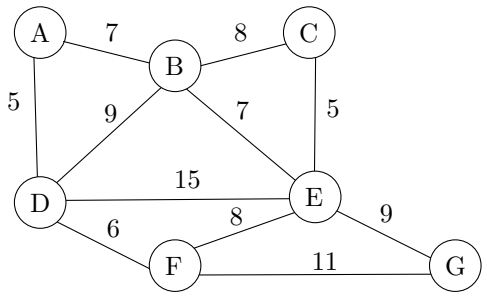
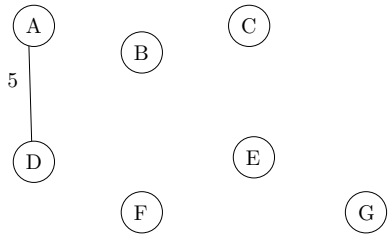
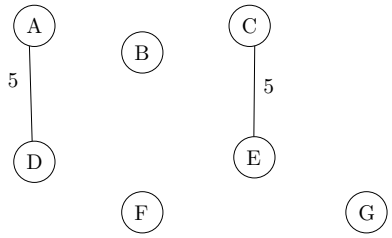


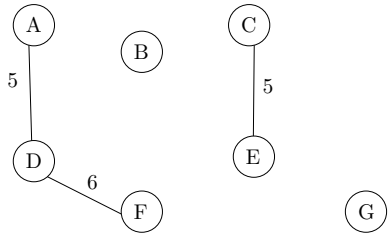
Figure 1: A weighted graph G on 7 vertices



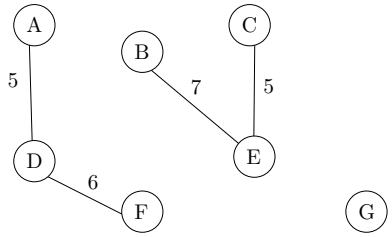
(a) AD and CE are the shortest edges, with length 5, and AD has been arbitrarily chosen.



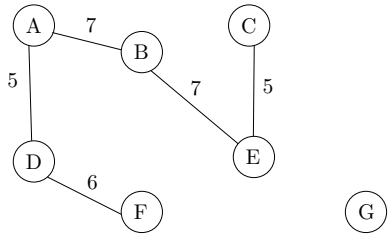
(b) CE is now the shortest edge that does not form a cycle, with length 5



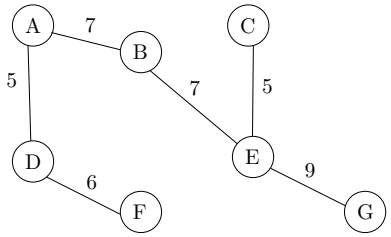
(c) The next edge, DF with length 6 is chosen.



(d) The next edge, BE with length 7 is selected



(e) The next edge, AB with length 7 is selected



(f) Finally, the process finishes with the edge EG of length 9, and the minimum spanning tree is found.

Figure 2: Example run of Kruskal Algorithm

2.2 Proof of Correctness

Proof of correctness follows from the cut property, which we proved earlier.

Fact 1 (The cut property (blue rule))**.** *For any cut $[S, \bar{S}]$ in a graph G, if e is a minimum weight edge crossing the cut $[S, \bar{S}]$, then e belongs to a MST of the graph.*

Fact 2 (Empty cut Property (red rule))**.** *A graph G is not connected iff there is a cut in G with no crossing edges, (empty cut).*

T is connected:

For a cut $[A, \bar{A}]$, now since G is connected $[A, \bar{A}]$ has at least one edge crossing $e_k = (u, v) \in$

$E, u \in A, v \notin A$.

Lets assume that the output graph T is not connected. Then T has atleast two components T_1 and T_2. In G there are edges between T_1 and T_2. Say $\{a_1, \ldots, a_l\} \subseteq E(T_1, T_2)$ and a_1 is the minimum weight edge among these edges. Certainly $a_1 = e_k$ for some k. In the kth iteration when e_k was considered, it was the first edge considered between T_1 and T_2 because its the minimum edge in between, so it couldn't create a cycle, and hence e_k must have been added to E' and is a part of T. It proves that T can not be a disconnected graph.

T is a tree

Obviously T has no cycle, since cycles are explicitly avoided to be in T.

T is a Minimum Spanning Tree

We prove that every edge in T is lightest of some cut which by Blue Rule will prove that T is minimum spanning tree. Kruskal maintains the invariant that there is no cycle. So consider connected components defined by T uptil kth iteration. The current edge added is $e_k = uv$ and e_k can't be within a connected component, as it will otherwise create a cycle.

So there is cut $[A, \bar{A}]$ separating $u \in A$ and $v \in \bar{A}$. Adding any edge from this cut will not create a cycle. Now by the sortedness of edges e_k is the first edge considered for this cut thus it is lightest edge accross this specific cut.

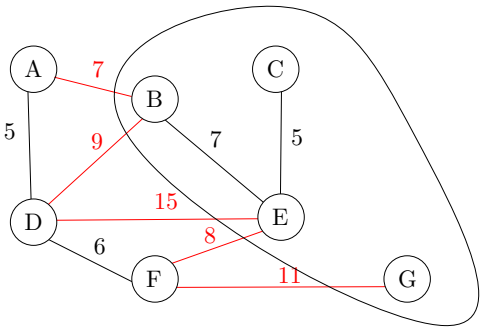


Figure 3: In the 6th iteration, edge AB was added and it is clear that it is the minimum edge across the cut $(\{A, D, F\}, \{B, C, E, G\})$.

2.3 Implementation and Runtime

Complexity of sorting is $O(m \log m) = O(m \log n)$ because $m = O(n^2)$. We have m iterations in the main loop. We need to check whether each edge creates a cycle. If (u, v) is the current edge considered, then (u, v) creates a cycle if and only if there is already a path between u and v. We can check if there is a path from u to v using BFS and DFS, it is going to take $O(n + |T|)$, so $O(n)$. So total time taken is $O(nm)$, the same as the brute-force implementation of Prim's algorithm.

It is quite obvious we need to do the cycle check quickly and by using union-find data structure, we can do cycle check in constant time. So taking this for granted, we get run time of the algorithm $O(m \log n) + O(m)$.

3 Union-Find Data Structure

Given a universal set U, recall that a partition of U is a collection of its subsets $S_1, S_2, \ldots, S_k$ such that

- $S_i \cap S_j = \emptyset$ for $1 \leq i < j \leq k$
- $\cup_{i=1}^{k} S_i = U$

We want a data structure to maintain a partition of U. This is usually called disjoint sets data structure or the union-find data structure.

We would like this data structure to maintain partition of U and support three operations, MakeSet, Union and Find.

- **Makeset** makes a set S_i in the partition. It also designate a representative of the set S_i. We identify a set S_i with its representative (we think of it as name of the set), we will identify a set by one of its element.
- **Find** For $x \in U$, $Find(x)$ should return the name of the subset containing x (Note that by partition property there is exactly one such subset). By comparing the result of two Find operations, one can determine whether the two elements are in the same subset.
- **Union** $Union(S_i, S_j)$ updates the partition such that all other subsets remain the same and S_i and S_j are union-ed together. The new set $S_i \cup S_j$ will get a new representative.

Suppose we have such a data structure. To see the connection, initially each vertex is a set by itself (a tree on one vertex each). When we add an edge (it must be between two different components (subtrees)), we make the union of two trees and make one bigger tree while all other trees remain as before. Here the universal set is $V(G)$, and the subtrees (components) made so far make a partition.

3.1 An Implementation of Disjoint Set Data Structure

A simple implementation of disjoint data set structure is to keep an array of linked lists. We store each set as a linked list and the first node of each list will be the representative of the set. Each element in the set will have a pointer to the representative of its set. We treat representative as name of the set. For each list we also keep the pointer to its last node.

- **MakeSet:** operation makes the trivial one-element linked list, with itself as representative. It runtime is $O(1)$.

- **Find:** $Find(x)$, returns the pointer in stored int the node of x. Runtime is $O(1)$, since
- **Union** joins two linked lists into a single subset. It is quite easy to implement this, all we have to do is to append one linked list to the other one. We make the next pointer of the last node point to the first node of list 2 (instead of NULL initially). Since we want representative to be the first of the list, for each node in list 2 we have to update its representative pointer to first of list 1. Which again needs one traversal of list 2. Total runtime is proportional to the length of list 2.

3.1.1 Union by Rank

Suppose S_u has to be union-ed with S_v where S_u and S_v are the sets of u and v respectively. We don't change the representatives of both sets, instead we keep the representative of one set, say S_u, same as before and make the representative of S_v as that of S_u. Specifically we change the representative of the set with fewer elements in it (the list with lower rank). For this purpose we keep the size of the list (in the first node of each list). When we union list 1 and list 2, it is easy to update rank of the resulting list.

Note that the only time consuming operation in union is the representative updates. Because we have to visit each node in a list to update its representative, we save a little by changing only reps of one list and that of a smaller one. In the worst case if lists are of size say $n/3$ and $n/4$, number of reps update is $O(n)$. Throughout execution of Kruskals algorithm we perform $n-1$ union operation and this sounds likes $O(n^2)$ work.

However, we can do a vertex centric runtime analysis by asking how many reps update a given vertex goes through. Rather than asking how many representative updates in a given iteration are done, what if we ask the question for a single element that is how many updates a given element goes through?

Suppose we keep the larger list's representative. So every time a fixed element x goes through an update rep, the size of its new set is at least double the size of its previous set. This is so because x is an element of the smaller list. Since a list can at maximum be of size n, a given vertex can go through at most $\log n$ rep updates.

3.2 Complexity for Kruskal's Algorithm Using Union-Find Data Structures

We see that $Find(x)$ operation takes $O(1)$ time and cycle check is just two $Find$ operations. Now we count the total pointer updates that are done in the algorithm. Since there are $n-1$ union operations, the total work done through all the union-ing is $O(n \log n)$. Sorting takes $O(m \log n)$ and $O(m)$ cycle checks each taking $O(1)$ time. Total run time of the algorithm is $O(m \log n)$.