Minimum Spanning Trees

Arnav Gupta

April 9, 2024

Contents

1 Spanning Trees

For a connected graph $G = (V, E)$, a spanning tree in G is a tree of the form (V, A) with A a subset of E (a tree with edges from E that covers all vertices).

The goal is to find a spanning tree with minimal weight.

2 Kruskal's Algorithm

```
GreedyMST(G):
    A = []
```

```
sort edges by increasing weight
for k = 1, ..., m:
    if e_k does not create a cycle in A:
        append e_k to A
```

2.1 Augmenting Sets without Cycles

Claim: Let G be a connected graph and let A be a subset of the edges of G. If (V, A) has no cycle and $|A| < n - 1$, then one can find an edge e not in A such that $A \cup \{e\}$ still has no cycle.

Proof:

- in any graph, # vertices - # connected components $\leq$ # edges
- for (V, A), this gives $n - c < n - 1$ so $c > 1$
- take any edge on a path that connects two components

2.2 Properties of the Output

Claim: If the output is $A = [e_1, \ldots, e_r]$ then (V, A) is a spanning tree $r = n - 1$.

Proof:

- (V, A) has no cycle
- suppose (V, A) is not a spanning tree, then there exists an edge e not in A such that $(V, A \cup \{e\})$ still has no cycle
  - for the case where $w(e) < w(e_1)$, this is impossible since e_1 has the smallest weight
  - for the case where $w(e_i) < w(e) < w(e_{i+1})$, this is impossible since at the moment we had inserted e_{i+1}, we decided not to include e which means that e created a loop with $e_1, \ldots, e_i$
  - for the case where $w(e_r) < w(e)$, this is impossible since if it was included in A since there is no loop in $A \cup \{e\}$

2.3 Exchanging Edges

Claim: Let (V, A) and (V, T) be 2 spanning trees and let e be an edge in t but not in A. Then there is some edge e' in A but not in T such that

$(V, T + e' - e)$ is still a spanning tree. Further, e' is on the cycle that e creates in A.

Proof:

- consider $e = \{v, w\}$
- $(V, A + e)$ contains a cycle $c = v, w, \ldots, v$
- removing e from T splits $(V, T - e)$ into two connected components T_1, T_2
- c starts in T_1, crosses over to T_2, so it contains another edge e' between T_2 and T_1
- e' is in A but not in T
- $(V, T + e' - e)$ is a spanning tree

2.4 Correctness: Exchange Argument

Let A be the output of the algorithm, (V, T) be any spanning tree. If $T \neq A$, let e be an edge in T but not in A. This means there is an edge e' in A but not in T such that $(V, T + e' - e)$ is a spanning tree and e' is on the cycle that e creates in A.

During the algorithm, we considered e but rejected it because it created a cycle in A. All other elements in this cycle have smaller (or equal) weight, so $w(e') \leq w(e)$ and so $T' = T + e' - e$ has weight $\leq w(T)$ and one more common element with A. This continues.

3 Data Structures for Kruskal's Algorithm

Operations possible on disjoint sets of vertices are:

- **find**: identify which set contains a given vertex
- **union**: replace 2 sets by their union

3.1 Implementation

```
GreedyMST_UnionFind(G):
    T = []
    U = {{v1}, ..., {vn}}
    sort edges by increasing weight
```

```
for k in range(1, m):
    if U.find(ek.1) != U.find(ek.2):
        U.union(U.find(ek.1), U.find(ek.2))
        append ek to T
```

3.2 Linked List

Uses an array of linked lists for U.

To do find, add an array of indices where $X[i]$ is the set that contains i.

In the worst case for this, find is $O(1)$ but union traverses one of the linked lists, updates the corresponding entries of X, and concatenates 2 linked lists, so union worst case is $\Theta(n)$.

This gives Kruskal's Algorithm to be $O(m\log(m))$ in sorting edges, $O(m)$ for find, $O(n)$ for union, and overall worst case $O(m\log(m)+n^2)$.

3.3 Simple Heuristics for Union

3.3.1 Modified Union

Each set in U keeps track of its size and only traverse the smaller list.

Also, add a pointer to the trail of the lists to concatenate in $O(1)$.

3.3.2 Key Observation

Worst case for 1 union is still $\Theta(n)$ but better total time:

- for any given vertex v, the size of the set containing v at least doubles when we update $X[v]$, so $X[v]$ updated at most $\log(n)$ times
- so the total cost of union per vertex is $O(\log(n))$