Kruskal's MST Algorithm

Robin Dawes

February 27, 2021

DEMONSTRATES the truth of that rather horrific aphorism about skinning a cat. Kruskal's MST Algorithm is presented and evaluated.

.

Kruskal's MST Algorithm

WE WILL NOW look at the second of our two MST algorithms: Kruskal's Algorithm. Like Prim's Algorithm it is an instantiation of the Generic MST Algorithm: it builds an MST by repeatedly choosing edges that can be safely added to previously chosen edges. The efficient implementation of Kruskal's algorithm motivates the introduction of a new data structure.

Kruskal's MST Algorithm dates from 1956. It has a less convoluted provenance than Prim's Algorithm.

Where Prim builds an MST by choosing edges that always form a connected tree and expanding it until it is a spanning tree, Kruskal chooses edges from different parts of the graph and eventually joins them together to create an MST.

The fundamental idea of Kruskal's Algorithm is to choose edges that do not form cycles. Once we have chosen $n-1$ edges of G that do not form any cycles, we must have a spanning tree of G. The wonderful thing about Kruskal's algorithm is that it not only builds a spanning tree, but it actually builds a Minimum Spanning Tree. It is not difficult to prove that, but as with Prim's Algorithm we will defer the proof until later.

This may not be obvious, but it follows from a result about trees that is covered in CISC-203.

```
def Kruskal(G):

    S = {}              # S is the set of edges we choose for the MST

    while |S| < n-1:
        Let (x,y) be a minimum-weight edge in G  such that S + (x,y) contains no cycle
        S += (x,y)

    return S
```

There are two obvious methods for choosing the next edge. We could sort the entire list of edges in $O(m * \log n)$ time, then test the edges in order. Since we might need to examine every edge in the sorted list before we find an MST, the **while** loop may execute m times. On each iteration, the difficult task is testing the candidate edge to see if it can be added to the previously chosen edges without creating a cycle. If we use "total_test_time" to represent how long all of this testing takes, then the time for the **while** loop is simply $O(total\_test\_time)$. Thus the whole algorithm runs in $O(m * \log n \ + \ total\_test\_time)$.

As an alternative we could place all the edges in a min-heap (since we don't necessarily need to sort the whole list of edges - if we are lucky, the MST will be found quickly and most edges will never be considered). We can build the heap in $O(m)$ time, which is less than the $O(m * \log n)$ required to sort the whole set of edges. The heap would have height in $O(\log m)$ which is the same order class as $O(\log n)$. Since we might need to access and update the heap $O(m)$ times, this also works out to $O(m * \log n)$ to access the edges in the appropriate order and fix the heap after each access. Once again, we need to test each candidate edge to see if it creates a cycle, so again we get $O(total\_test\_time)$ for this, and the whole algorithm runs in $O(m * \log n + total\_test\_time)$ ... the same complexity as the version in which we sort the full set of edges. The actual running time of this heap-based version may be less than the "sort-all-the-edges" version, particularly if the MST is found without looking at too many edges.

Now we need to consider that "total_test_time". Suppose the edge being considered is $e = (x, y)$. We need a fast method to check to see if x and y are already connected by some of the edges already chosen (in which case we cannot use this edge). If x and y are not already connected then we choose the edge and add it to the S set. But then we need to update the "connected-to" information: we need a method to record that all the vertices that were already connected to x are now also connected to all the vertices that were already connected to y, and vice versa.

We will look at two methods Both are based on the idea of identifying groups of vertices that are currently joined by edges or paths in the set S of chosen edges.

There are others - you may want to research this.

Solution 1

We will identify each group of connected vertices by an unique integer, stored in an array $P[1..n]$: $P[x]$ holds the group_id for vertex x. When we want to check an edge (x, y) to see if it creates a cycle, we just need to compare the group_id of x to the group_id of y. This takes constant time. If they are the same, x and y are already connected and we cannot use the edge. If they are different, the edge is safe to use. However, adding the edge (x, y) to S means that all the vertices in the two groups are now connected. We need to find all the vertices u such that $P[x] = P[u]$, and change them all to $P[y]$.

In pseudo-code it looks something like this:

```
def Kruskal(G):

    S = {}
    for i = 1 to n:
        P[i] = i # each vertex is in its own group
    while |S| < n-1:
        Let (x,y) be a minimum-weight edge in G  such that P[x] != P[y]
        S += (x,y)
        temp = P[x]
        for i = 1 to n: # update the P values
            if P[i] == temp:
                P[i] = P[y]
```

The *for* loop that updates the P values is obviously in $O(n)$. It executes every time we choose an edge, which happens exactly $n - 1$ times. Thus the total time involved in testing and fixing the P values is in $O(n^2)$.

Using this method, Kruskal's MST Algorithm runs in $O(n^2)$.

Solution 2

INSTEAD OF USING an array that stores the group_id for each vertex, we will use special trees called DISJOINT-SET TREES to store this information. A disjoint-set tree consists of a number of vertices that represent set elements, with one designated as the root vertex. All edges in the disjoint-set tree point upwards (from the leaves to their parents, from those vertices to their parents, and so on up to the root).

Disjoint-set trees are sometimes called SET-UNION TREES because - as we shall see - they are particularly useful for implementing the operation of computing the union of two sets.

It is important to note that these disjoint-set trees are **not** subgraphs of G - they simply represent the groups of vertices that are connected by the edges that we choose during Kruskal's algorithm.

At the start of the algorithm, each vertex is the sole leaf (and also the root!) of a separate disjoint-set tree. As the algorithm progresses, the disjoint-set trees will be combined. To test an edge (x, y) to see if it can be used in the MST, we trace upwards from x in x's disjoint-set tree to the root of that disjoint-set tree, and then trace upwards from y in y's disjoint-set tree to the root of that disjoint-set tree. We will see that this can be done in $O(\log n)$ time or better. If the two roots are the same, then x and y are in the same disjoint-set tree - which means they are already connected by edges in S so the edge (x, y) cannot be added to the MST. If the two roots are not the same then the edge (x, y) is added to S and the two disjoint-set trees must be combined. We will see that this can be done in constant time.

We can define a very simple Object for vertices of disjoint-set trees:

```
class Disjoint_Set_Tree_Vertex():

    def init(int i):
        int id  = i
        Disjoint_Set_Tree_Vertex parent = nil
        int rank = 0      #rank indicates the height of the disjoint-set tree below this vertex

# end of class definition
```

and with this in hand, the algorithm looks something like this:

```
def Kruskal(G):
    S = {}
    for i = 1 to n:
        P[i] = new Disjoint_Set_Tree_Vertex(i)

    while |S| < n-1:
        let (x,y) be the next candidate edge
        # x and y are Disjoint_Set_Tree_Vertex objects
        # trace up from x until we find a disjoint_set_tree_vertex
        # with no parent.  This is the root of x's disjoint-set tree.
        temp = x
        while temp.parent != nil:
            temp = temp.parent
        Rx = temp      # Rx is the root of x's disjoint-set tree
        # do the same for y
        temp = y
        while temp.parent != nil:
            temp = temp.parent
        Ry = temp      # Ry is the root of y's disjoint-set tree

        if Rx != Ry:
            # add the edge to S and combine the disjoint set trees by attaching
            # the one with fewer levels to the one with more levels
            S += {(x,y)}
            if (Rx.rank >= Ry.rank) :
                Ry.parent = Rx
                Rx.rank = max(Rx.rank, Ry.rank+1)
            else:
                Rx.parent = Ry
```

As promised, combining the two sets takes constant time. However, we need to think carefully about which way we do the combination in order to minimize the number of levels in the combined disjoint-set tree. Clearly we could use either "**`Rx.parent = Ry`**" or "**`Ry.parent = Rx`**" to combine the two sets and the result would be valid. But if we can minimize the height of the combined tree, this will keep the "trace up" steps as efficient as possible. If the two disjoint-set trees being combined have the same rank, it doesn't matter which one becomes the parent of the other. But if (for example) **`Rx.rank > Ry.rank`** then making **`Rx`** the parent of **`Ry`** creates a combined disjoint-set tree in which the maximum trace-up time is less than if we make **`Ry`** the parent of **`Rx`**.

With some careful counting that I'm not going to go into here, we can show that if we always make the root of the taller disjoint-set tree the parent of the root of the shorter disjoint-set tree, then the disjoint-set trees will always have height in $O(\log n)$. This means the trace-up time is in $O(\log n)$.

Basically we show that the disjoint-set trees are balanced.

But we can do even better! Every time we search for the root of the disjoint-set tree for a vertex x, we can make x and all the vertices "above it" in the disjoint-set tree point directly to the root of the disjoint-set tree. This results in many vertices in the disjoint-set tree having the root of the tree as their parent ... which means finding the root of the disjoint-set tree for these vertices only takes 1 step. In other words finding the root takes ***effectively*** constant time. The algorithm for this tree-compression looks like this:

```
def find_root(x):
    if x.parent == nil:
        return x
    else:
        x.parent = find_root(x.parent)
        return x.parent
```

Since we might have to do the tracing up operation for every edge of the graph, total_test_time is ***effectively*** in $O(m)$.

A note about this ***effectively*** constant time for finding the root of a disjoint-set tree: it's not quite constant. Its actual growth rate is given by the inverse Ackermann function $\alpha(n)$, which grows **very very very slowly** ... so slowly that $\alpha(n) \leq 5 \ \forall\, n \leq 2^{2^{2^{2^{16}}}}$.

Thus for any plausible graph size, finding the roots of the disjoint set trees will take constant time.

Thus we see that thanks to the choice of a good data structure, Kruskal's MST algorithm can be ***effectively*** executed in $O(m * \log n + m)$ time, which reduces to ***effectively*** $O(m * \log n)$ time. (Here I am using the word ***effectively*** to mean "so close we can't tell the difference".)

Note that if m is in $O(n)$ (ie. G is sparse) then this algorithm ***effectively*** runs in $O(n * \log n)$ time. However if m is in $\Omega(n^2)$ (ie. G is dense) then this algorithm ***effectively*** runs in $O(n^2 * \log n)$ time.

Prim vs. Kruskal - the Showdown

Now that we have seen two MST algorithms it is natural to ask which is better. The answer of course is that it depends on the situation. Some authors suggest that for dense graphs Prim is faster (recall the $O(n^2)$ implementation) but that for "typical graphs" - whatever that means - Kruskal is faster. I have not seen any convincing evidence to support this claim but if "typical" means "sparse" then it is plausible.

Last Words on Data Structures for MST Algorithms

There are other, more sophisticated data structures that reduce the complexity of finding an MST (particularly when using Prim's Algorithm) even further. I encourage you to research them.

The current best MST algorithm that I know about is due to Chazelle - it uses disjoint-set trees and a data structure called a soft heap that Chazelle invented.