Solving the MST Problem with the Skewed Filter-Kruskal Algorithm

Ermanno Righini

June 1, 2021

1 Introduction

The minimum spanning tree problem consists of finding for a graph $G = (V, E)$, the minimum weight spanning tree of G. The most used and successful algorithms to solve this problem are the Prim algorithm [1] and the Kruskal algorithm [2].

The Kruskal algorithm has been improved over the years, first in [3] where the authors mixed the kruskal algorithm with the idea of incremental sort. Then it was improved again in [4] where a filtering step was introduced during the sorting.

2 Prim Algorithm

The Prim algorithm starts with a tree consisting of a single node, then it iteratively adds to the tree the smallest edge that connects to a node not part of the partial spanning tree. When there are no edges meeting this criteria, the procedure has produced a minimum spanning tree. The only case where this doesn't work is when the input graph is not connected and a spanning tree doesn't exist. In this case the problem becomes of finding the minimum spanning forest, and the Prim algorithm can be adapted to work on this case by simply running the Prim algorithm on all nodes of the graph that are not yet part of a spanning tree.

To efficiently find the next smallest edge it is necessary to use a priority queue, like for example the Pairing Heap data structure [5] that I used in this project. I chose to use the Pairing Heap because it is simple to implement and has a good performance in practice. The time complexity of the Prim algorithm is $\mathcal{O}(E + V \log V)$ if implemented with min-heap that performs the decrease-key in constant time, it is instead $\mathcal{O}(E \log V)$ in the case of the pairing heap.

3 Kruskal Algorithm

Kruskal algorithm instead follows a different approach: the list of edges is sorted in increasing weight and a forest is initialized where each node represents a tree on its own. Then starting from the smallest edge, we add all the edges that connects nodes in different trees of the forest. By the end of the procedure the forest gets collapsed to a minimum spanning tree if the graph is connected or to a minimum spanning forest otherwise.

```
function KRUSKAL($V$, $E$, $T$)                    ▷ With $T$ list of MST edges
    for $(x, y) \in$ SORTED($E$) do              ▷ Sort by increasing edge weight
        if FIND($x$) = FIND($y$) then
            UNION($x$, $y$)
            $T := T \cup \{(x, y)\}$
```

For an efficient implementation of the algorithm a Disjoint Set data structure can be used which accelerate the operations of $find(x)$, which returns the root of the tree containing x and $union(x, y)$ which merges the trees of x and y into a single tree.

The time complexity of the algorithm is dominated by the sorting of the edge list and therefore is $\mathcal{O}(E \log E)$.

3.1 Disjoint Set

The main component of the Kruskal algorithm is the Disjoint Set data structure, which implements an efficient way of representing a collection of disjoint sets of elements and perform operation on them such as:

- FIND(x): returns the identifier set containing x.
- UNION(x, y): merges the two sets containing x and y.

An efficient implementation of FIND can use the path compression technique, which (in combination with union by rank) reduces the time complexity from $\mathcal{O}(n)$ to an amortized $\mathcal{O}(\alpha(n))$ as analized in [6]. Path compression consists in caching the set identifier of an element x every time the FIND procedure is called upon it.

```
function FIND($x$)                                 ▷ With Path Compression
    if $x.parent \neq x$ then
        $x.parent =$ FIND($x.parent$)
    return $x.parent$
```

The UNION procedure can instead be efficiently implemented by using the union by rank variation, where the basic idea is to always add the smaller set to the bigger one when performing a union operation. (this on average should lead to updating fewer elements)

```
function UNION($x$, $y$)                                  ▷ Union by Rank
    $px, py :=$ FIND($x$), FIND($y$)
    if $px \neq py$ then
        if $px.rank < py.rank$ then
            SWAP($px$, $py$)
        $x.parent =$ FIND($x.parent$)
```

3.2 Quick-Kruskal Algorithm

The Quick-Kruskal algorithm variation instead of sorting the edge list before building the MST, does the two things simultaneously. It performs sorting in a similar way to QuickSort and when the partitions become too small it falls back to the classing Kruskal algorithm.

```
function QUICKKRUSKAL($V$, $E$, $T$)
    if $|E| <$THRESHOLD($V$, $E$) then
        return KRUSKAL($V$, $E$, $T$)
    $p :=$ PICKPIVOT($E$)
    $E_1, E_2 :=$ PARTITION($E$, $p$)
    QUICKKRUSKAL($V$, $E_1$, $T$)
    if $|T| < |V| - 1$ then
        QUICKKRUSKAL($V$, $E_2$, $T$)
```

Where THRESHOLD should be $\mathcal{O}(n)$ to produce asymptotically correct results. In all of my implementations I chose THRESHOLD$(V, E) = 1000$ because empirically it gave me the best results.

With this strategy the algorithm becomes much faster for graph with randomly generated weights, since the edge list needs to be sorted only up to the last edge contained in the MST and for randomly generated graphs the MST is contained in the first $\mathcal{O}(|V| \log |V|)$ smaller edges as analyzed in [7]. On the other hand this algorithm is not very robust, because if the MST has at least one heavy edge, then the list needs to be sorted up until the heaviest edge in the MST and the complexity falls back to the standard Kruskal algorithm.

The average time complexity of the algorithm for random graphs with random edges is $\mathcal{O}(E + V \log^2 V)$ as seen in [3]

3.3 Filter-Kruskal Algorithm

The Filter-Kruskal variation is an improvement upon the Quick-Kruskal algorithm, where after the partition operation, the right side of the partition is filtered of any edge adjacent to two vertices in the same tree of the forest ($\textsc{Find}(x) = \textsc{Find}(y)$).

```
function FilterKruskal($V$, $E$, $T$)
    if $|E| < \textsc{Threshold}(V, E)$ then
        return $\textsc{Kruskal}(V, E, T)$
    $p := \textsc{PickPivot}(E)$
    $E_1, E_2 := \textsc{Partition}(E, p)$
    $\textsc{FilterKruskal}(V, E_1, T)$
    if $|T| < |V| - 1$ then
        $E_f := \{(x, y) \in E_2 : \textsc{Find}(x) \neq \textsc{Find}(y)\}$        ▷ Filter pass
        $\textsc{FilterKruskal}(V, E_f, T)$
```

This simple modification of the algorithm leads to much better performance since if the graph is not very sparse, most of the edges are not part of the MST and get filtered away during the incremental sorting of the edge list, making the sorting of the remaining portion of the list easier as more edges are added to the MST.

The time complexity of the algorithm for random graphs with random edge weights is $\mathcal{O}(|E| + |V| \log|V| \log \frac{|E|}{|V|})$ as shown in [4].

3.4 Skewed Filter-Kruskal Variation

To further improve the Filter-Kruskal algorithm I made the assumption that in random graphs with random edges the MST is usually contained in the first $\mathcal{O}(|V| \log|V|)$ smallest edges and more generally in most graphs a good portion of the MST is contained in the smallest edges of the graph if the graph is dense enough.

To leverage this, in the first recursion level of $\textsc{FilterKruskal}$ I perform a skewed partitioning, where the first partition has size $k|V| / \log|V|$ where k is a constant that can be tuned, and it should usually be slightly bigger than 1 since its purpose is to have some headroom in the case that the MST is heavier that expected. In every other recursion level instead I split the edge list in half. From my tests I actually found out that using a skewed partitioning method even in inner recursion levels doesn't degrade performance. I think this happens because by using a skewed partitioning I improve the branch miss-prediction rate and the edges get filtered away earlier, decreasing the cost of the sorting. On the other hand the recursion depth increases, so the two effects roughly balance each other.

3.5 Picking a Pivot

I tested different strategies for picking a pivot, such as:

- **1 Sample Random Pivot**: Select one element randomly from the edge list and use its weight as a pivot.
- **3 Samples Random Pivot**: Select three elements randomly and use the median as the pivot.
- $\sqrt{n}$ **Samples Random Pivot**: Select $\sqrt{n}$ elements randomly and use the median as pivot.

For the FilterKruskal implementation I found that picking just one sample turned out to be the fastest, though this strategy introduces more variability in the execution time of the procedure, since in particular in the first recursion levels a bad pivot can worsen the performance significantly.

For the SkewedFilterKruskal implementation, since I need to be more precise with the pivot selection I decided to use instead the "root of n" approach. This produces much better pivots but at the same time it's much slower (for very dense graphs the number of pivot is close to the size of the MST).

One way to mitigate this performance penalty is to pick a sample of size $\sqrt{n}$ one time at the start of the procedure and reuse the pivot from the sample at every recursion level until all the pivots in a given range have been exhausted, and then go back to a single random sample per pivot.

4 Using Multiple Pivots

Another approach I tried to improve the Filter-Kruskal algorithm is to use more than one pivot in the partitioning phase.

The rationale behind this idea is that if the MST edges are contained in the smallest edges of the graph then I only need to explore the first partition (which should be smaller than the one generated by the Filter-Kruskal algorithm) and I only read once the rest of the unneeded edges. In the case that instead I need to read the entire edge list the sorting is facilitated since I already partitioned the input list in many small lists. So if the constant costs introduced by adding more pivots are small I expect to see an improvement in execution times.

4.1 Sample Sort

The Samplesort algorithm can be thought of as a generalization of quicksort to k pivots. The basic idea is that we extract a sample of elements of size $\mathcal{O}(k\sqrt{n})$ from the input list so that we can calculate good pivots to perform the partitioning. Implementations of samplesort vary the pivot count from a few dozen to a few thousands.

In particular I decided to adopt the implementation of Super Scalar Sample Sort [8] where the author showed that this implementation can beat the standard C++ sorting algorithm by a sizeable margin. The algorithm is recursive and each recursion is organized in a few steps:

- **Pivot Sampling** The input list is sampled to extract k pivots, which are used to subdivide the elements in $k+1$ buckets.
- **Build Decision Tree** A binary decision tree is built on the extracted pivots, so that for every element the corresponding bucket can be found efficiently. The bucket can be calculated without any branches.
- **Assign Buckets** For every element we calculate the corresponding bucket and store all the bucket ids in a list.
- **Calculate Bucket Sizes and Prefix Sum** During the bucket assignment phase we also calculate the bucket sizes and calculate the prefix sum of the bucket sizes in order to easily find where each bucket starts in the output continuous array.
- **Distribute Elements** Distribute all the elements in the corresponding bucket disposed in a contiguous array.
- **Recursion** Repeat the procedure on all the buckets. If the bucket size is under a certain threshold, instead fallback to a simpler sorting procedure.

The algorithm was adapted to the MST problem by checking before each recursion step if the MST has been found, and in that case interrupting the procedure. Also a filter step has been added before each recursion on all buckets that are not the first one. Finally, when the bucket size is below a predefined threshold we run the basic kruskal algorithm instead of a sorting algorithm.

4.2 Dual Pivot QuickSort

I also tried the partitioning scheme proposed by [9] as an improvement to quicksort. The basic idea is that we use two pivots, instead of only one, so at every recursion level we partition the list in three partitions. This compared to samplesort should introduce lower constant cost for the partitioning phase of Filter-Kruskal.

5 Graph Generation

Another important part of the project was the generation of graphs to use as a benchmark suite for the different algorithms. I ended up focusing on three categories of graphs:

- **Random Graphs with Random Edges** These are graphs where the edges are added to the graph with a certain uniform probability p and where the weights are chosen randomly between a minimum and maximum weight.

- **Random Graphs with 1 Long Edge** These graphs are generated in the same way as the previous category with the exception that in the graph is always present at least one edge that must be contained in the MST and that is the longest among all the other edges. This property is useful to test what happens in the case where algorithms based on Kruskal are forced to scan the entire edge list.

- **Random 2D Geometric Graphs** In these category of graphs, vertices are points on a 2D plane, and edges between them are the euclidean distances. For each vertex only the edges to its k nearest neighbors are added to the graph. This class of graphs, compared to the other two has the property of having more clusters of vertices.

5.1 Random Sparse Graphs Generation

One of the problems I faced during this project was to efficiently generate sparse graphs. That is why I developed an algorithm specifically for generating sparse graphs which in my tests is faster than all other methods.

The basic idea is that instead of checking for every edge if we want to include it or not into the graph, I randomly skip to the next edge that will be included in the graph by following the correct probability distribution.

First I imagine to have an enumeration of all the possible edges for a graph with V vertices, where the first edge has index 0 and the last $V-1$.

So given p as the probability of any possible edge to be included in the graph, I define the probability of skipping at least n edges as:

$$p_n = (1-p)^n$$

Because I need to *not* choose an element for at least n times. I now solve in function of n and I obtain that:

$$n = \left\lfloor \frac{\log p_n}{\log(1-p)} \right\rfloor$$

This means that by plugging a random probability value in p_n I can randomly calculate the skip distance to the next edge. In order to prove that every edge e has probability p of being chosen we need to show that all the probabilities of all the paths that lead to choosing e sum to p.

It is obvious that the first edge has probability p of being chosen since the only way we can choose it is if we skip 0 edges, so:

$$C_0 = p(1-p)^0 = p$$

If we instead by any chance choose an edge k it means that we either skipped all the edges before it (this has probability $p(1-p)^k$) or before picking k we picked another edge j where $0 \leq j < k$.

In this case the probability is the probability of picking j times the probability of skipping from j to k, that is: $C_j p(1-p)^{k-j-1}$

Knowing that:

$$\begin{aligned}
\sum_{j=0}^{n-1}(1-a)a^j &= (1-a)a^0 + (1-a)a^1 + \ldots + (1-a)a^{n-1} \\
&= 1 + (-a+a) + (-a^2+a^2) + \ldots + (-a^{n-1}+a^{n-1}) - a^n \\
&= 1 - a^n
\end{aligned}$$

If we consider all paths leading to choosing k, we get that the probability of picking k is:

$$\begin{aligned}
C_k &= p(1-p)^k + \sum_{j=0}^{k-1} C_j p(1-p)^{k-j-1} \\
&= p(1-p)^k + \sum_{j=0}^{k-1} p^2(1-p)^{k-j-1} && \text{(Assume by induction that } C_j = p\text{)} \\
&= p(1-p)^k + p\sum_{j=0}^{k-1} p(1-p)^j \\
&= p(1-p)^k + p\sum_{j=0}^{k-1} (1-q)q^j && \text{(With } q = (1-p)\text{)} \\
&= p(1-p)^k + p(1-q^k) \\
&= p(1-p)^k + p(1-(1-p)^k) \\
&= p(1-p)^k + p - p(1-p)^k \\
&= p
\end{aligned}$$

By using this method for the generation of graphs the complexity becomes linear in the number of generated edges, instead of linear in the number of total possible edges.

In my implementation I decided to enumerate the edges of the upper triangular matrix from the adjacency matrix of the graph, so that $0 \leq a < b < N$ for any valid edge (a, b).

5.2 Random Geometric Graphs Generation

For the generation of geometric graphs I first randomly place all the graph vertices on a 2D area, and then for each vertex I select add the edges relative to

```
function GenGraph($N$, $p$, $E$)                    ▷ $N$ number of vertices
    $a := b := 0$
    while True do
        $k :=$ Random$(0, 1)$
        $skip := 1 + \left\lfloor \frac{\log(k)}{\log(1-p)} \right\rfloor$
        $b := b + skip$
        while $b \geq N$ do
            $a := a + 1$
            $b := a - n + 1$
        if $a \geq N - 1$ then return
        $E := E \cup \{(a, b)\}$
```

its k closest neighbors. Finding the k closest neighbors has cost $\mathcal{O}(n \log k)$, for a total cost of $\mathcal{O}(n^2 \log k)$ for generating the graph.

We can bring the time complexity down to $\mathcal{O}(kn \log k \log n)$ by organizing the vertices in a k-d tree, which is an n-dimensional generalization of a binary search tree. In a k-d tree at a given depth d, the points are compared along the dimension d mod k.

The construction of a balanced tree can be achieved recursively, where at each step the elements belonging to the current node are divided into two partitions of equal or similar size by finding the median element in linear time, with a total cost of $\mathcal{O}(n \log n)$.

The k-closest neighbors of a given vertex can be found in time $\mathcal{O}(k \log k \log n)$ by keeping track of the k closest elements in a priority queue, and searching the tree while pruning the nodes that are too far.

6 Experimental Results

All test were performed on a Intel Core i5-6600 CPU @ 3.9GHz and all the source code is available at `https://github.com/righier/filter-kruskal`

All experiments were performed on graphs with 60000 vertices, and varying the number of edges between 10^4 and 10^9.

As can be seen in Figure 1, the best performing algorithm in all graph types is Skewed Filter-Kruskal, even in the case of graphs with long edges, where the procedure is forced to read and filter all edges. And for the case geometric graphs, where the higher clustering coefficient introduces heavier edges into the MST.

The Filter-Kruskal variants with multiple pivots, like Samplesort Filter-Kruskal are performing worse than the standard Filter-Kruskal implementation, probably because of the too much higher constant cost of the partitioning method.

Lastly the Prim algorithm implemented with Pairing Heap performs better than the standard Filter-Kruskal algorithm for denser graphs.

7 Conclusions

The results of this project didn't turn up as I expected, in particular I thought that the Filter-Kruskal variants based on multiple pivots would perform much better than they did in reality.

One way in which these kinds of algorithms (in particular samplesort) could bring an advantage is in an eventual parallel implementation of the algorithm, since individual steps of the samplesort algorithm are good candidates for parallelization.

References

[1] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[2] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[3] Rodrigo Paredes and Gonzalo Navarro. Optimal incremental sorting. In *2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 171–182. SIAM, 2006.

[4] Vitaly Osipov, Peter Sanders, and Johannes Singler. The filter-kruskal minimum spanning tree algorithm. In *2009 Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 52–61. SIAM, 2009.

[5] Michael L Fredman, Robert Sedgewick, Daniel D Sleator, and Robert E Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1-4):111–129, 1986.

[6] Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, 1979.

[7] Svante Janson, Donald E Knuth, Tomasz Łuczak, and Boris Pittel. The birth of the giant component. *Random Structures & Algorithms*, 4(3):233–358, 1993.

[8] Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *European Symposium on Algorithms*, pages 784–796. Springer, 2004.

[9] Vladimir Yaroslavskiy. Dual-pivot quicksort. *Research Disclosure*, 2009.

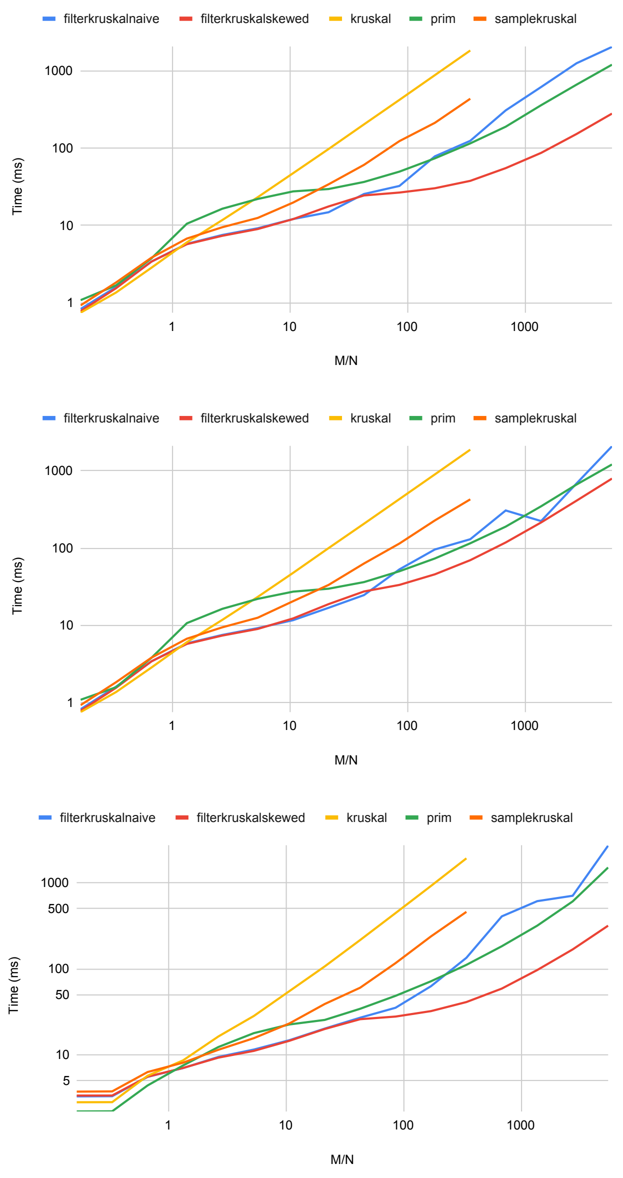

Figure 1: Comparison of different algorithms on different graph types. From top to bottom: 1) graphs with random weights, 2) graphs with one long edge, 3) random geometric graphs