SOLVING MINIMUM SPANNING TREES THROUGH IMPLEMENTATIONS AND COMBINATIONS OF KRUSKAL'S ALGORITHM AMONG OTHER APPROACHES

Nadula Kadawedduwa

1. Introduction

The resolving of minimum spanning trees, while being a simple problem with a long established approach, enjoys great popularity and continuous research due to its inherent and peripheral applications in a multitude of combinatorial as well as practical problems [1]. The MST problem is often abstracted in such a way:

> *Given a weighted graph G of N vertices where all its weights are positive, it is possible to continuously remove redundant edges that currently form cycles such that the final state of the graph G' is one where there are no cycles (a spanning tree has exactly (V-1) edges), the graph is complete (assuming the unmodified graph was complete), and the sum of all the weights in the modified graph is the minimum. G' ⊆ G [2].*

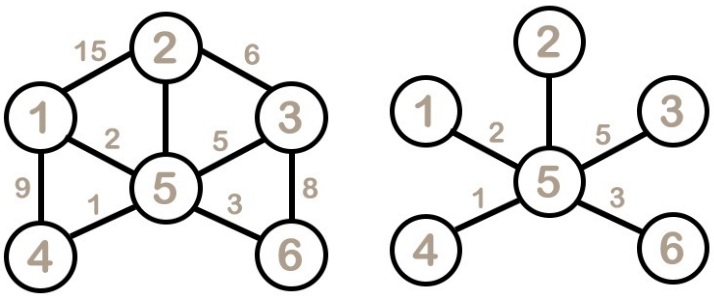
To make this problem more approachable, it can be rephrased to liken each node of the connected graph to a city, and the edges between these vertices to roads or power lines, each of which has a cost associated with placing it down. Thus, a company laying down telephone wires, building pipelines, or wiring networks would be interested in minimizing cost and maximizing efficiency. On the periphery, MSTs show up in testing network reliability, categorical problems, and machine learning [1].

These and many other instances contain MSTs as a sub-problem which simplifies the process or allows for approximations, such as in the case of the Traveling Salesman Problem. While the Traveling Salesman (TS) problem is incredibly well known, it will be covered here for thoroughness. The TS problem involves a similar set up to our MST problem above. However, rather than reducing our initial connected graph G, the aim is to traverse all N vertices of *G* using the lowest possible cost path. It won't be expanded upon here, but the TS problem is known to be NP-Hard, and, more importantly, it is NP-Complete (it is possible to convert any problem in NP to the TS problem in polynomial time). This is significant due to the possible applications of an efficient algorithm for solving TS problem which would in turn provide an efficient methodology to solve any problem in the NP classification. Using an algorithm to develop an MST from a connected graph will provide an upper bound on the TS problem as a whole (knowing that an MST can be created in P time).

As outlined above, assuming it is possible, we may reduce any connected, weighted graph *G* into a minimum spanning tree, G', which maintains connectivity to N vertices with lowest possible total cost. Using said spanning tree *G'* an upper bound can be achieved by simply doubling the total cost of the MST (accounting for the worst case where every vertex $v_1 \ldots v_N$ is connected to one vertex v_i) *Fig. 1*.

Figure 1

A graph who's MST contains vertex 5, adjacent to every other vertex. This is the worst case scenario in reduction of a TS problem.



In addition to providing an upper bound on the TS problem, MSTs can provide an approximation of the complete solution. The simplest approach to the development of a path using an MST would be to traverse every vertex and then take short cuts as needed to reduce repeated traversals. The above can be accomplished through an implementation of a depth first search algorithm. Using the MST of a connected graph G, conduct a depth first traversal and record the order of vertices, then, short cut in between any repeated nodes. The short-cut part of this algorithm can be done in linear time O(N-1) by simply deleting any repeated vertices in our traversal. Of course, this can only be done assuming our graph is complete (any given vertex is connected to every other vertex). Alternatively, one could check if such an edge exists between two vertices before taking the short-cut, also in linear time assuming the MST is stored as a matrix or an equivalent structure. Ultimately, this form of DFS traversal to create a path has a worst case of runtime *2*(Optimal Solution),* hence, it is referred to as a 2-Approximation.

This same approach was improved in 1976 by Nicos Christofides who provided a three-halves approximation ($\frac{3}{2}$**Optimal Solution*) [4]. Given any connected graph *G* (preferably a complete one), find its MST, G'. Using said *G'* graph, consider only the vertices of odd-degree; find a minimum-cost perfect matching of these vertices (algorithms such as the Ford Fulkerson Algorithm can be used for this purpose) [5]. If the above perfect matching's edges are of set S, then consider S ∪ G'as G''. This newly created graph can be further reduced to remove any redundancy. Using the methodology of the simple DFS algorithm, reduce our repeated edges by conducting a depth first traversal while recording all visited vertices then deleting any repeated vertices (after checking for the existence of an edge to shortcut over). The resulting path will be an approximated "tour" according to the earlier bounds [4].

Getting back on track, the above and many more useful algorithms employ MSTs as a method of simplifying and reduction. The development of an efficient algorithm for MST production contributes to much more than the simple direct applications. While there are many notable modern algorithms for this purpose, it is only right start with the two main figures whose notoriety is tied to this very problem itself. But, before that, a short exploration of an MST's intrinsic properties from which those two algorithms were deduced.

2. Properties of MSTs

A minimum spanning tree with distinct or indistinct weights has two core properties that are central to the up-coming proofs, the cut and cycle properties, both of which imply that an MST has only one edge between any pair of vertices; the MST is "unique" [6]. *Going forwards, all graphs will be assumed to be undirected*.

2a. Cut Property

A *cut* (specifically a cut set) in this section is defined as a set of edges which, when removed, will split the graph *G* into two distinct, unconnected graphs G_1 and G_2. In other words, the set *C* (the cut set) is the set of all edges that connect two vertices, one in G_1 and the other in G_2 [6].

1. Suppose a connected graph *G* has vertices in set $V = \{v_1, v_2, v_3, \dots, v_n\}$ and edges in set $E = \{e_1, e_2, e_3, \dots, e_m\}$ *Fig.2.1*
2. Construct two subgraph S_1 and S_2, each containing an arbitrary number of vertices, such that $V = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$ *Fig.2.2*
3. Find cut set *C* by finding all edges who have one vertex in S_1 and the other in S_2 *Fig.2.3*
4. The lowest cost edge in *C* will be included in the final minimum spanning tree

Figure 2

A graph divided into two arbitrary sets of vertices. The cut set contains and edge with the weight of 2 which will be in the MST.

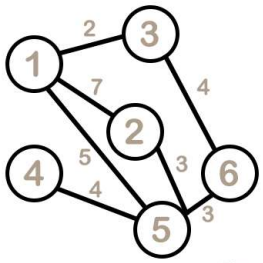


Fig. 2.1

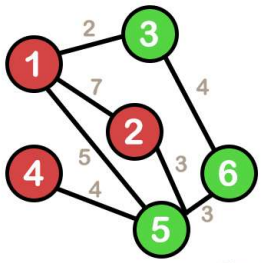


Fig. 2.2

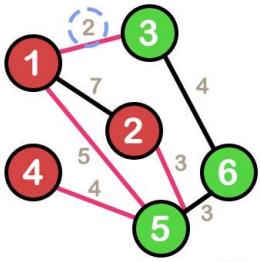


Fig. 2.3

Cut Proof by Contradiction

As repeated above:

Suppose a connected graph *G* has vertices in set $V = \{v_1, v_2, v_3, \dots, v_n\}$ and edges in set $E = \{e_1, e_2, e_3, \dots, e_m\}$. Construct two subgraph S_1 and S_2, each containing an arbitrary number of vertices, such that $V = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$.

Furthermore, there is set *C* comprised of all edges that connect one vertex in S_1 to another in S_2. There is edge $\{e_i | e_i \in C\}$ such that the weight of e_i is less than the weight of every other element of set *C* (assuming all weights in graph *G* are distinct).

For this proof, the completed MST will be enumerated as *G'* with set of *N* vertices: *V'*; and set of *N-1* edges: *E'*.

- ⇒ Assume, by way of contradiction, that the edge e_i is not a member of *E'* and is not in our MST *G'* $\{e_i | e_i \notin E'\}$
- ⇒ By the property of MSTs, *G'* must be connected; there is some edge e_j that is an element of the set *C* and connects some element in S_1 with some element in S_2 $\{e_j | e_j \in E', C\}$
- ⇒ Both e_i and e_j are members of set *C*, but the weight of e_i is the smallest in set *C*. Therefore, the weight of e_j must be greater than that of e_i
- ⇒ It is impossible for both e_i and e_j to be members of *E'* due to the limited size of *E'*: $|E'| = N - 1$
  - ⇒ If $|E'| > N - 1$ then *G'* is not acyclic and is therefore not an MST
  - ⇒ If $|E'| = N - 1$ and $\{e_i, e_j | e_i, e_j \in E'\}$ then *G'* is not complete and is therefore not an MST
- ⇐ Thus, *G'* is not an MST of graph *G* as it does not have the lowest possible total cost (it is, however, a spanning tree) [7]

2b. Cycle Property

A cycle is the same here as it is in standard graph theory; One cycle in graph *G* is an ordering of distinct edges *K* which, when traversed, will return you to the starting vertex. *K* may be represented as an ordered list of form $[e_a, e_b, e_c, \dots, e_n]$ or as an unordered set of form $\{e_a, e_b, e_c, \dots, e_n\}$.

1. Suppose a connected graph *G* has vertices in set $V = \{v_1, v_2, v_3, \dots, v_n\}$ and edges in set $E = \{e_1, e_2, e_3, \dots, e_m\}$ *Fig.3.1*
2. Identify any cycle within G (This can be accomplished through the implementation of a simple Depth First Traversal which is mentioned in the following section) *Fig.3.2*
3. Find set of distinct edges *K* that form the cycle using our previous traversal
4. The highest cost edge in *K* will NOT be included in the final minimum spanning tree *Fig.3.3*

Figure 3

A graph with one possible cycle identified, and the edge with the highest weight in that cycle, the edge with weight 6 in this case, will not be included in the final MST.

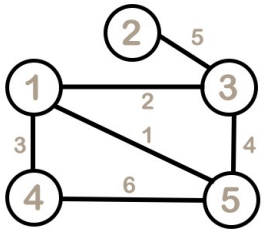


Fig. 3.1

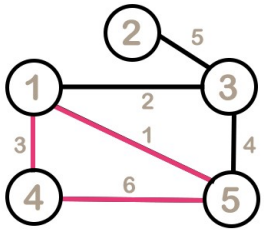


Fig. 3.2

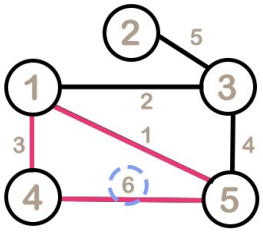


Fig. 3.3

Simple Algorithm to Find Cycles

While it is not very useful or complex, it is worth considering the simple DFT approach to finding cycles to understand how Kruskal's algorithm and others improve on it. Most algorithms approach the problem of finding cycles through the implementation of a "Disjoint-Set Data Structure" which will be covered in a future section.

The existence of a cycle in graph *G* (stored in adjacency list format since we do not need the weights of each edge in this step) with edge list *E* and vertex list *V* can be found by running a Depth First Traversal wherein you keep track of any visited vertices in an array. Check if the current vertex is in our visited list, if it is not: record the current vertex then recursively do the same for all vertices adjacent to this current one. This implementation takes up $O(|V| + |E|)$ time on its own, as well as $O(|V|)$ space to store our list of visited vertices. This algorithm, as well as Tarjan's strongly connected components algorithm and the topological sort [Both $O(|V| + |E|)$] are rarely used and are periphery to the main point, among some other earlier mentioned algorithms, its pseudo code will still be included for completeness *Fig. 4*.

```
1. Cycle(G, v; P)
2.    All vertices are unvisited
3.    Let V := {}
4.    Let R be a recursion stack initialized as empty
5.    Let P be a list that records the current path
6.    Let C := Call Cycle_h from vertex v with V, P, and R
7.    /*C contains a cycle where the last vertex in C is also the start of the loop*/
8.    C := Sublist of C from first occurrence of C[length of C - 1] to the end
9.    Exit.

10.   Cycle_h(G, V, v)
11.   If v is not in V (v ∉ V) then
12.      Add v to V
13.      Insert v on the end of P
14.      For all vertices i in G[v]
15.         Push i on to the recursion stack R
16.      If the recursion stack is not empty then
17.         Return the recursive call Cycle_h(G, V, k) where k is popped off the stack R
18.      Else
19.         Fail with error "No Cycle"
20.   Else (v is in V and has been traversed before) (v ∈ V)
21.      Insert v on the end of P
22.      Return P
```

Fig. 4 Procedure for detecting and recording a cycle in graph G starting at vertex v in $O(|V|+|E|)$ time

Cycle Proof by Contradiction

Suppose a connected graph G has vertices in set $V = \{v_1, v_2, v_3, \dots, v_n\}$ and edges in set $E = \{e_1, e_2, e_3, \dots, e_m\}$.

Furthermore, there is cycle C comprised of edges that form a cycle starting and ending at vertex v_i. There is edge $\{e_h | e_h \in C\}$ such that the weight of e_h is greater than the weight of every other edge comprising cycle C.

For this proof, the completed MST will be enumerated as G' with set of vertices V' and edges E'.

- $\Rightarrow$ Assume, by way of contradiction, that the edge e_h is a member of E' and is in our MST G' $\{e_h | e_h \in E'\}$
- $\Rightarrow$ By the property of MSTs, G' must be connected by $(|V'| - 1)$ edges; Therefore, deleting any single edge would split G' into two separate subgraphs
  - $\Rightarrow$ Suppose edge e_h is deleted and the two remaining subgraphs are referred to as S_1 and S_2 $\{e_h | e_h \in E', C\}$
- $\Rightarrow$ Since e_h was part of cycle C, there must be one other edge e_g (one edge previously excluded in that was also in the cycle) that has one vertex in S_1 and the other in S_2
  - $\Rightarrow$ There is another possible spanning tree G'', with set of vertices V'' and edges E'', that has e_g and not e_h $\{e_h, e_g | e_h, e_g \in C; e_h \in E''; e_g \notin E''\}$
- $\Rightarrow$ The weight of e_g must be less than the weight of e_h since both e_g and e_h were members of cycle C and the greatest cost edge in C was e_h.
  - $\Rightarrow$ It follows that the cost of G'' must be less than G'. $Cost\ (G'') < Cost\ (G')$
- $\Leftarrow$ Therefore, G' is not an MST of graph G as it does not have the lowest possible total cost (it is, however, a spanning tree) [7]

3. Brief History on MSTP

Before moving on to the solutions developed so far for MSTP, it is apt to cover the history and background behind its rise. While Kruskal and Prim are certainly the two largest figures in this scene, and justifiably so, they are far from being the first to describe the MST problem and propose a solution. Earlier accounts date back to the beginning of the 1900s and appear in a multitude of places such as Poland, France, and Czechoslovakia [1].

The first proposal of the MST problem was by Otakar Borůvka, a Czech mathematician and scientist most well known now for his work on graph theory, in 1926. Moravia, a region in the eastern part of the now Czech Republic, was engineering their power lines to span their residences; during this time, Borůvka noticed and abstracted the problem into a form similar to today's MSTP, albeit without the mention or use of graph theory [8]. A translation of Borůvka's original proposal and translation to English can be found here: J. Nešeťril, E. Milková [9].

In the following years, Borůvka's problem and approach would be adapted in terms of known graph theory and inspire others to improve on his foundation. Come the 1950s, the big names in MSTP, Kruskal and Prim, among many others, had published their findings and differing approaches to the standardized problem.

Prim's approach in particular was discovered and rediscovered multiple times, compounding layers of naming confusion; while it was first published by Vojtěch Jarník in 1930, it was later rediscovered by Robert Prim in 1957, and then once again republished in 1959 by Edsger Dijkstra [8].

Kruskal's algorithm preceded Prim's in 1956. Their implementations differ, but all of these algorithms can be implemented in $O(|V|log_2|E|)$ where *V* and *E* are the vertices and edges of graph *G* respectively (both were greedy algorithms). There are other algorithms developed by proceeding mathematicians in the future, such as Tarjan and Karger, who developed better and better algorithms and will be discussed later on.

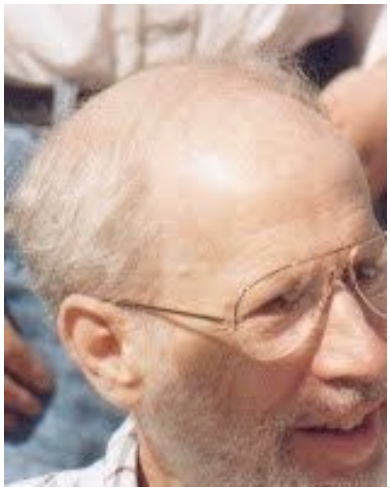


Fig.7

Joseph Kruskal
Source: cyclowiki.org

3. Kruskal's Algorithm

Consistent with previous declarations, suppose a connected graph *G* has vertices in set $V = \{v_1, v_2, v_3, \dots, v_n\}$ and edges in set $E = \{e_1, e_2, e_3, \dots, e_m\}$.

Using the two properties of MST that we established earlier, Kruskal's algorithm procedurally builds up a tree from graph *G* by considering each edge one at a time. For the purposes of building up a MST, we can declare *F* as a forest such that $(F \subseteq G')$. A given edge e_i can be determined as a member of our growing forest *F*, and eventually our MST *G'*, by applying the cut and cycle properties.

Suppose the algorithm is determining if edge e_i is a member of *F*. This edge that connects vertex *v* and vertex *u* would either be the first edge in some cut set *C* that splits the *F* into two sets S_1 and S_2, or the last edge in some cycle *Y* if there was already a path from vertex *v* to vertex *u* [6]. By first ordering all of the edges, it is ensured that the edge is either:

- The lowest weighted edge in some cut set *C'* that splits the graph *G* into set S_1 and S_2 (any lower cost edge has already been added to *F*) and so should be included in *F* by the Cut Property
- The last added edge in some cycle of *F*, and also therefore the heaviest (edges ordered by increasing weight), so it should not be included in *F* by the Cycle Property

In this way, Kruskal's Algorithm can come to a negative decision using information about the current state of the forest *F* ($e_i \notin F$), but a positive decision on whether $e_i \in F$ is done through inductive properties, assuming the forest $F \subseteq G'$. Note that the positive decision is a slightly weak induction as the algorithm adds the edge e_i if it would be the first edge added that satisfies a cut rather than the first edge considered; due to ordering the edges and adding them after filtering, it is the case the that the first edge added is also the first edge considered, however, this relies on the previous steps in the induction being correct [6].

Kruskal's Algorithm first sorts edges by weight in increasing order, an operation that takes up the greater portion of its runtime. Using a comparison sort, which is the only universally consistent way since we don't know anything about the weights a graph will have, always has a $O(|E|log_2|E|)$, reflecting the $O(nlog_2 n)$ worst case of most comparison sorts. The second, lesser part of the runtime belongs to the Union-Find Algorithm that is responsible for finding any pre-existing paths in *F* from *u* to *v* through the use of Disjoint Set Data Structures (which will be covered below as promised) in $O(|E|\alpha(|V|))$ where

$\alpha(|V|)$ is the inverse Ackermann function, a slow growing function that ensures that Kruskal's Algorithm doesn't falter for larger datasets [6].

The two possible steps taken in Kruskal's Algorithm will be visualized using the figure below. For our partially complete forest *F*, we look at our sorted edge list which contains our unconsidered edges in order of increasing weight [3,4,5,7] Fig. 5.1. For simplicity's sake the edges will only be represented by their unique weights in this example but in a working implementation each edge would contain more data. Popping off the first edge will be the lowest cost one possible which we then check the *u* and *v* of Fig. 5.2. In this case, the 3 cost edge connects vertices 3 and 2; there is currently no path between these two vertices so we accept this edge into *F* Fig. 5.3. The new sorted edge list is [4,5,7], and so we consider the edge with weight 4 Fig. 5.4. In this case, there is already a path from vertex 2 to 1 which the 4-cost edge also spanned, therefore we do not have this edge in *F* by the earlier stated proofs Fig. 5.5.

Figure 5

The series of graphs depicts a few steps of Kruskal's Algorithm on a partially transformed forest *F*. Solid lines are edges that have been established to be in *F* while the dotted lines to be considered that are in *G*. Green vertices are ones that have already been connected by some path.

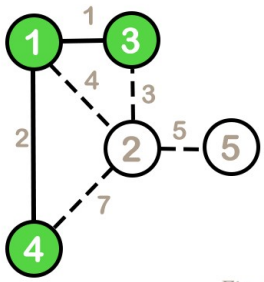


Fig. 5.1

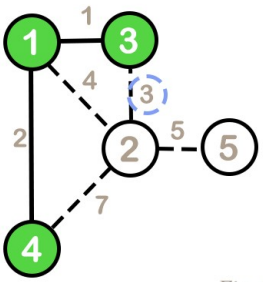


Fig. 5.2

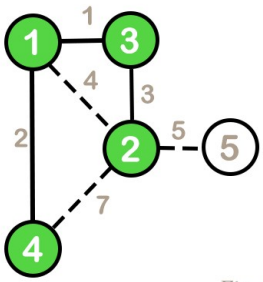


Fig. 5.3

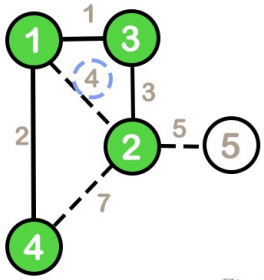


Fig. 5.4

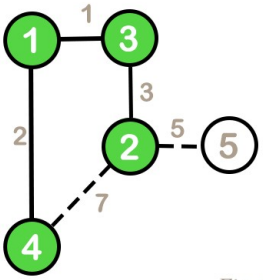


Fig. 5.5

3b. Disjoint Datasets and Ackermann Function

Below is the common, unmodified implementation of Kruskal's Algorithm which uses a merge sort for consistent O($nlog_2n$) operations regardless of the distribution of edge weights [Fig. 8]. A disjoint set data structure allows checking for connectivity more efficiently than with any traversing algorithm.

The particular instance of disjoint data structure used in Kruskal's Algorithm is known as a disjoint set forest which can perform union operations and find operations in almost constant time; specifically, in O($|E|\alpha(|V|)$) where a is the inverse Ackermann Function, typically represented by A. The DSF itself consists of $|V|$ sets, initialized with each set containing only its respective vertex, that is unioned repeatedly until all edges have been unioned into one set (the below implementation is slightly inefficient in that it checks all edges regardless of having completed the MST or not).

The Ackermann Function can be defined recursively as below (in the context of graph theory), the inverse of which represents the scaling of operations in a disjoint set forest which progressively take less time as the sets are modified in each step through repeated unions, decreasing the number of total sets.

$$A(|E|,|V|) = \begin{cases} |V|+1 & if\ |E| = 0 \\ A(|E|-1,1) & if\ |E| > 0\ and\ |V| = 0 \\ A\big(|E|-1, A(|E|,|V|-1)\big) & if\ |E| > 0\ and\ |V| > 0 \end{cases}$$

Fig. 6 The below chart from [source] depicts the scaling of values in the Ackermann Function

Values of $A(m, n)$

$m\backslash n$	0	1	2	3	4	
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2=$
2	3	5	7	9	11	$2n+3$
3	5	13	29	61	125	$2^{(n+3)}$
4	13	65533	2^{65536}	$2^{2^{65536}}$	$2^{2^{2^{65536}}}$	

Note:

Java, which was used for testing and data collection in this paper, uses a dual-pivot Quicksort for sorting arrays (`Array.sort()`) by default but can be manually specified or implicitly changed. Using an array of Objects will push the JVM to instead use a stable, iterative implementation of Mergesort which was the method chosen here.

```
1.  Kruskal(G)
2.  Let F := the empty set ∅
3.  For each vertex v in G.vertices
4.      Make a set containing only vertex v
5.      /*Suppose all of these sets are contained in one
        data structure for grouping*/
6.  Let E := the list of all edges of G
7.  Sort E by increasing order of weights (the first element is the lowest cost)
8.  For each edge in E (starting from the first element)
9.      /* u and v are the endpoints of current edge*/
10.     If there is no single set containing both vertices u and v
11.         Combine the two disjoint sets (one contains u and the other v)
12.         Add the current edge to F
13.         Discard the edge from list E
14.     Else
15.         Discard the edge from list E
16. Return F
```

Fig. 8 Kruskal's Algorithm using disjoint sets to determine connectivity in overall $O(|E|log_2|V|)$ time

In relation to the above demonstration of Kruskal's Algorithm [Fig. 5], the disjoint set forest for each step respectively, including previous and future unseen steps, looks like:

$$\langle\{1\}\{2\}\{3\}\{4\}\{5\}\rangle \quad E = 1 \to 2 \to 3 \to 4 \to 5 \to 7 \to \emptyset$$

$$\xrightarrow{\{1\}\cup\{3\}} \langle\{1,3\}\{2\}\{4\}\{5\}\rangle \quad E = 2 \to 3 \to 4 \to 5 \to 7 \to \emptyset$$

$$\xrightarrow{\{1,3\}\cup\{4\}} \langle\{1,3,4\}\{2\}\{5\}\rangle \quad E = 3 \to 4 \to 5 \to 7 \to \emptyset$$

$$\xrightarrow{\{1,3,4\}\cup\{2\}} \langle\{1,3,4,2\}\{5\}\rangle \quad E = 4 \to 5 \to 7 \to \emptyset$$

$$\xrightarrow{\{1,3,4,2\}\cup\emptyset} \langle\{1,3,4,2\}\{5\}\rangle \quad E = 5 \to 7 \to \emptyset$$

$$\xrightarrow{\{1,3,4,2\}\cup\{5\}} \langle\{1,3,4,2,5\}\rangle \quad E = 7 \to \emptyset$$

4. Kruskal's Algorithm Data

Multiple types of graphs were used to test Kruskal's Algorithm over repeated iterations. Sparse, dense, and completely randomized graphs depict how Kruskal's algorithm is appropriate for relatively limited graphs that are preferably sparse.

The below simulations were run under consistent conditions: *Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz* while limited to one core to reduce variability. For each point of data, 500 random graphs were generated and solved from which the average time in milliseconds was taken as its efficiency for V vertices. Each subtype of graph was tested for vertices ranging from 100 to 4000 until a noticeable trend surfaced. The code for the generic Kruskal's Algorithm as stated above was provided by Aakash Hasija while the code the graph generator and test iteration was written by Nadula Kadawedduwa. The data collected can be viewed in full at the following link along with the code used in developing it:

https://github.com/NadulaKadawedduwa/mst

4b. Creating Data

The above tool can be used by simply calling the method `exp1(int v, int s, int i, Type t)` where the first parameter is the number of vertices to which the user wishes to generate data, the second is the step (how many vertices the method increases by each iteration), and the third is the type of graph to be tested defined by an enum of `SPRASE`, `DENSE`, or `RANDOM`. A sparse graph, in this implementation, is defined as having between $(|V| - 1)$ and $((|V|)(|V| - 1)/4)$ while a dense graph has between $((|V|)(|V| - 1)/4)$ and $((|V|)(|V| - 1)/2)$ edges.

> *Note:*
>
> *The range of possible weights is set to be from 0 to 2*|V| by default for sufficient variation. A few iterations of the data will be shown below for clarity; this is the 500 graphs created and solved for each number of vertices, in this case 3100 and 3200:*

3100																																						
226	602	217	587	98	169	732	119	651	40	493	109	731	148	361	808	14	655	333	309	110	524	214	639	241	5	242	675	106	313	175	155	596	310	276	333	307	500	6
566	579	329	48	728	228	16	551	114	466	401	78	719	510	403	396	580	383	563	601	34	193	307	273	143	17	30	729	49	598	402	42	263	13	224	656	398	213	3
459	7	295	49	143	493	510	108	386	299	448	217	525	307	338	3	373	388	63	146	605	587	343	602	36	58	567	13	170	454	783	672	193	577	253	177	9	205	1
185	256	443	614	459	378	63	111	168	15	174	595	320	295	212	461	28	505	643	294	535	660	75	462	388	62	343	452	548	89	575	113	387	430	230	20	410	130	4
220	68	480	608	539	104	203	691	587	568	140	117	540	543	421	713	686	278	568	377	298	433	465	199	132	517	528	531	186	541	39	418	475	165	160	40	496	23	
508	550	252	682	309	355	375	335	135	231	239	539	78	330	577	14	797	5	562	262	11	356	321	13	107	59	92	319	213	227	159	373	513	106	206	626	214	935	
741	500	293	91	505	263	294	432	355	84	469	217	422	286	564	649	115	77	539	503	564	642	614	482	680	352	191	360	449	366	218	536	96	405	126	111	673	401	5
468	610	26	62	629	629	494	563	128	354	341	66	41	512	437	358	513	181	59	343	212	559	1	259	29	172	535	300	12	175	198	95	308	461	230	137	36	218	2
378	486	392	136	142	75	447	538	434	451	218	269	707	299	140	142	504	549	283	346	598	195	400	21	465	525	415	230	231	7	7	25	608	222	153	436	90	44	1
643	589	194	278	4	41	184	629	13	126	657	307	556	370	69	553	504	237	610	152	303	334	402	194	137	245	458	491	208	415	380	215	596	623	19	175	440	720	1
3200																																						
300	646	85	185	306	259	593	171	226	266	271	624	326	303	572	632	209	532	5	45	101	476	410	261	408	558	103	373	411	316	567	633	399	189	470	358	40	565	3
590	705	102	454	510	117	378	603	544	593	215	506	600	334	405	678	383	703	14	224	447	490	546	438	553	87	413	274	402	152	390	441	25	327	578	326	185	224	4
460	213	72	27	356	490	371	682	610	562	326	102	160	332	417	436	327	323	442	204	542	623	614	121	208	233	458	187	801	25	113	135	300	596	419	486	352	171	1
255	13	515	282	124	144	554	588	663	557	193	427	107	688	311	420	189	136	832	486	341	100	28	392	189	658	409	45	509	259	541	420	242	364	5	363	479	487	
641	434	477	587	151	13	487	467	474	193	399	419	612	15	471	599	33	447	263	329	404	35	569	285	472	402	420	34	224	553	678	10	64	407	133	412	64	530	

Fig. 9 (Above) Two datasets generated for the two corresponding points on the graph below (Sparse)

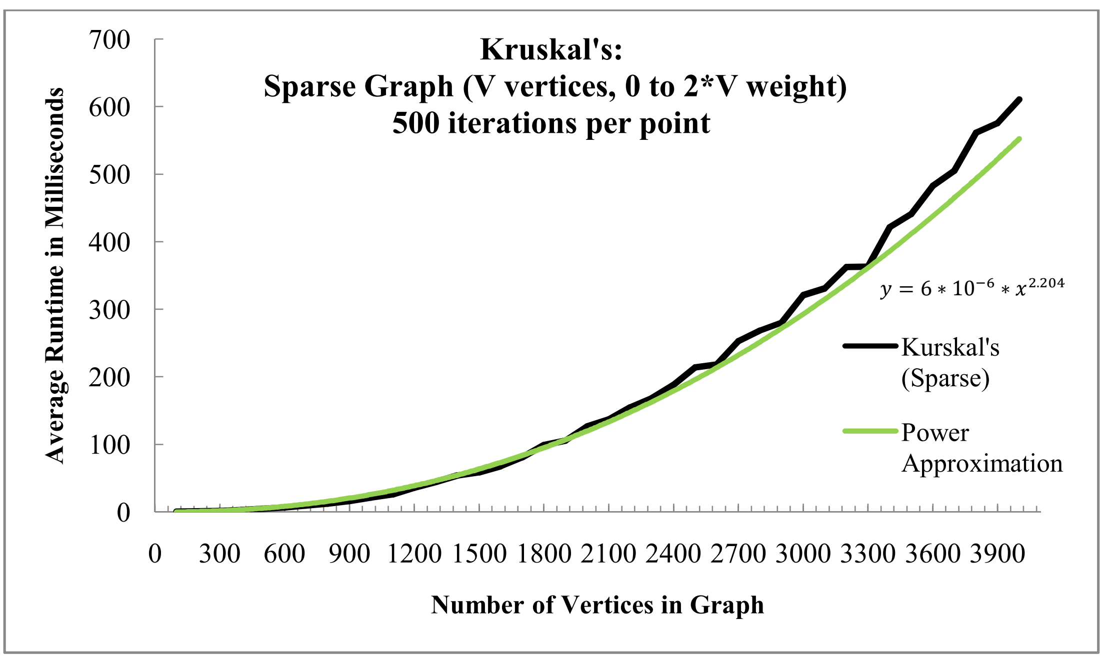


Fig. 10 Above is the runtime trend of Kruskal's Algorithm on sparse graphs for number of vertices V along with an approximation using a power function. Full data set can be found at the above linked repository. (Data points were collected at 100 vertex intervals)

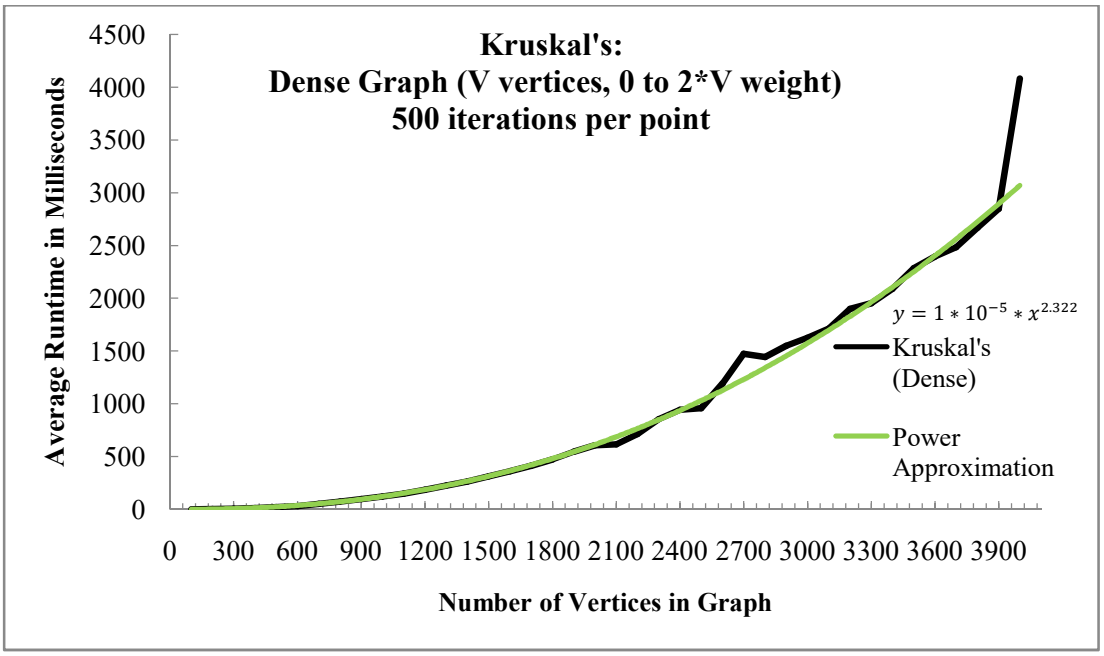


Fig. 11 Above is the runtime trend of Kruskal's Algorithm on dense graphs for number of vertices V along with an approximation using a power function. Full data set can be found at the above linked repository. (Data points were collected at 100 vertex intervals)

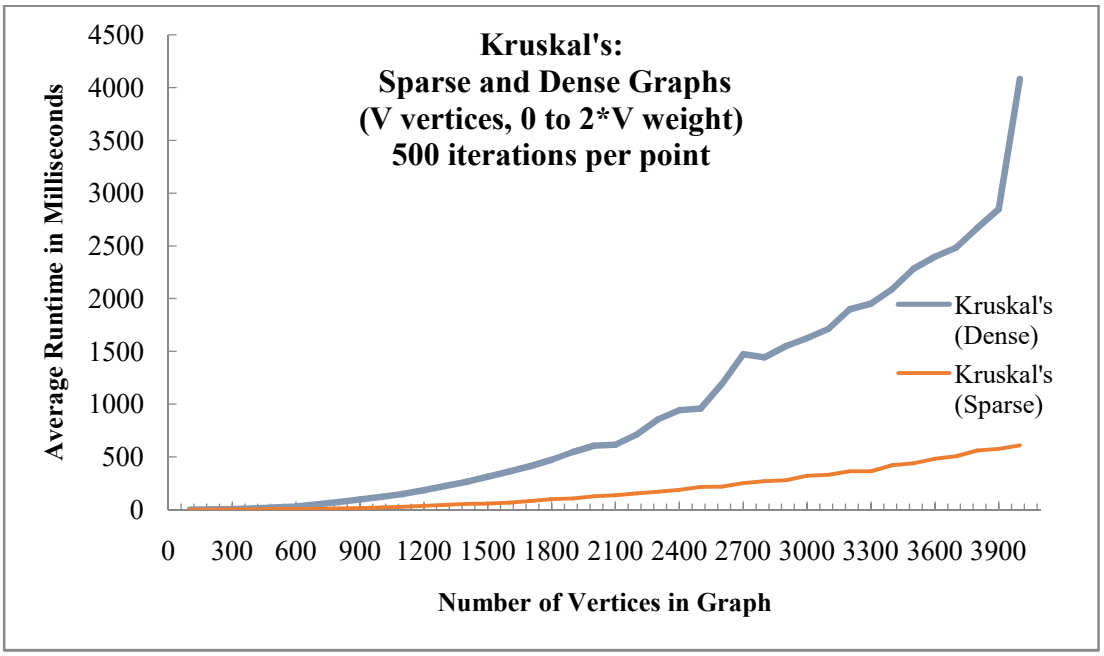


Fig. 13 Above is the runtime trend comparison of Kruskal's Algorithm on both dense and sparse graphs for number of vertices V. Full data set can be found at the above linked repository. (Data points were collected at 100 vertex intervals)

5. Analysis

Kruskal's algorithm suffers greatly in sorting dense graphs as it is limited by the efficiency of its sorting algorithm. This problem is reflected clearly in the data above in a consistently greater average runtime by an order of magnitudes in dense graphs, as well as a steeper rate of increase of average runtime in relation to the linear increase in vertex number. The efficiency issue is especially apparent in the comparison graph. The bottleneck caused by the sorting algorithm precludes it from direct use in real-world applications that deal with significantly larger data sets, begetting even larger edge sets upwards of the millions.

The worst case scenario for Kruskal's algorithm occurs in a complete graph where the edges are also of the worst case in regards to that particular sort. Since an implementation of Kruskal's typically uses a Mergesort to sort the edge list, the above implementation included, a completely unsorted set of edges along with a graph that forces inclusions of certain edges will drag out this algorithm.

In relation to the sorting portion, Kruskal's is less than optimal for dense graphs because of this step alone which requires the sorting of all edges, even those that will not be in the graph. This trend is reflected in both the dense and sparse data sets as runtime increased non-linearly corresponding to the linear increase in the number of vertices.

The second instance that should be considered is in the arrangement of edges. Graphs which contain vertices that have only one connecting edge inherently force Kruskal's algorithm to include said edge, regardless of its weight: $\{e_i | E[v_i] = \{e_i\}, G' \nsubseteq \{G - e_i\}\}$. An extreme illustration of these can be seen in "Lollipop Graphs" which care, loosely speaking, a union between a largely complete graph and a series of vertices connected by one edge between them *Fig. 11*. More than a few instances of lollipop graphs occurred in the data set used above and resulted in said data points spanning three standard deviations away from the displayed mean.

That being said, even in the best case graph G wherein the first $(|V| - 1)$ edges of the sorted list E of size $|E|$ are all included in the MST G', the entire list had still been sorted as mergesort does not allow for interruptions or partial sorting of data. In addition to this, using an implementation of mergesort precludes the possible best case scaling as mergesort will make $n * log_2 n$ comparisons regardless.

With these shortcomings in mind, a few possible approaches stand out as possible remedies:

1. Implement a sorting algorithm which allows for interruption or syncing with Kruskal's during its sorting process
    a. If such a condition is possible then it may also substantiate using a lower worst-case efficiency and higher best-case efficiency algorithm in place of mergesort
2. Filter out un-needed edges during the sorting process so they will not be considered later on
3. Parallelizing portions of Kruskal's to cut down on wait times between comparisons

6. Filter Kruskal's MST Algorithm

In a very recent (2009) and substantial study, Vitaly Osipov and co. of the Karlsruhe Institute of Technology in Germany developed an adaptive filtering approach to MST creation using Kruskal's Algorithm as a subroutine while drawing heavily from Prim's MST Algorithm. This algorithm was able to achieve an $O(|E| + |V| * log_2 log_2 \frac{|E|}{|V|})$ [11] where E and V are the edge and vertex sets of a graph G respectively. This is, in fact, the same time complexity as Jarník–Prim's Algorithm using binary heaps

and parallelization. However, this adaptation of Kruskal's lends itself more completely to parallelization as can be observed in its pseudo code below [10].

Note:

$O(|E| + |V| * log_2 log_2 \frac{|E|}{|V|})$ *time is actually linear when* $\frac{|E|}{|V|} \leq 2$ *resulting in* $O(|E|)$.

$|E| = \Omega(|V| * log_2(|V|) * log_2 log_2(|V|))$. *Also, while the original Kruskal's Algorithm is inherently sequential, verifying one edge at a time after sorting, it is possible to use parallelization in the mergesort to scalably quicken comparisons.*

```
1.  ModifiedKruskal(E, F, S)
2.      /*E is the edge set; F is Forest being built;*/
3.      /*S is disjoint set data structure*/
4.      Sort edges of E using any in-place sort: δ
5.      For each edge e_i in E
6.          /*Same as normal Kruskal's check if vertices are in same subset*/
7.          If e_i has endpoints {u_i, v_i} where {u_i, v_i | {u_i, v_i} ⊈ S_k, k ∈ N, 0 ≤ k ≤ |S|}
8.              Add edge e_i to F
9.              Union the set containing u_i and the set containing v_i in S

10. FilterKruskal(E, F, S)
11.     If |E| ≤ Threshold(|V|, |E|, |F|)
12.         ModifiedKruskal(E, F, S)
13.     Else
14.         Let e_j := a pivot edge in E {e_j ∈ E}
15.         Let LesserE := edges in E with weight less than e_j ⟨e_l ∈ E : w(e_l) ≤ w(e_j)⟩
16.         Let GreaterE := edges with weight greater than e_j ⟨e_l ∈ E : w(e_l) > w(e_j)⟩
17.         FilterKruskal(LesserE, F, S)
18.         GreaterE := Filter(GreaterE, S)
19.         FilterKruskal(GreaterE, F, S)

20. Filter(E, S)
21.     Return only edges of E that cover new vertices
22.     ⇒ ⟨{u_i, v_i} ∈ E : u_i and v_i are not both of the same subset⟩
23.     /*{u_i, v_i | {u_i, v_i} ⊈ S_k, k ∈ N, 0 ≤ k ≤ |S|}*/
```

Fig. 14 The above pseudocode describes the Filter Kruskal's MST Algorithm which contains a part of Kruskal's as a subroutine. This code was adapted from [Osipov] [11].

Note:

The sort used in the above code(δ) *is interchangeable with any in-place sorting algorithm. Additionally, the* **Threshold** *method will determine when Kruskal's is run on the partition of graph size given by its return value.*

6b. Small Improvements

Now that Kruskal's algorithm has essentially been broken down into more compartmentalized sections, it is possible to minutely improve efficiency for extremely large datasets that Kruskal's typically suffers in.

One such improvement in the above pseudo code that can be seen in the actual implementation Java is the inclusion of a biased pivot. For larger sets of data, it would be incredibly beneficial to have a pivot that splits the data in exactly half; this, however, would significantly hinder runtime as one would have to sort through the entire list, the very bottleneck that the algorithm sought to avoid. Rather than taking a fully accurate median, the algorithm takes a small random sample ($\sqrt{|E|}$ in this implementation) [11] of the given edge list in that iteration and finds the median of that using `Quickselect` (a variation of `Quicksort` that only partially sorts the given list in order to find the k'th smallest element) or a similar efficiency algorithm. This precaution prevents obscenely long run times for larger inputs if a poor pivot was picked by chance.

Another corner left to be cut can be seen in the number of `Find` operations done; it is currently the same number as a standard implementation of Kruskal's. For each edge in a dense, or simply a moderately-sparse graph, it is increasingly likely that any processed edge cannot be used in the final MST F. As the MST gets built and subsets get unioned, it is possible that the two vertices of any new edge already have the same parent, let alone the same root (the root is the representative of any subset). Rather than comparing roots using the `Find` operation, the algorithm first compares the direct parents which have a high likelihood of being the same in denser graph. Again, this improvement mainly helps in large datasets and may even, like the previous improvement, hinder the algorithm in excessively small datasets. Ideally, these small datasets are covered by our `Threshold` value and sent directly to standard Kruskal's.

7. Filter Kruskal's Operation

The above pseudocode was heavily abstracted and so did not explain the functionality of the algorithm in full. The overall goal of this algorithm is to split up Kruskal's algorithm into smaller steps and improve the efficiency of those compartmentally.

Firstly, Filter-Kruskal's seeks to improve efficiency in the sorting step by reducing the number of edges given to Kruskal's at any time. This is accomplished by only running Kruskal's after the length of the input is below a specified minimum, if it doesn't meet this condition, the edge list is split into two partitions based on a randomly chosen pivot. Any edge with weight less than the pivot is allocated to a "Lesser" list and any edge greater is allocated to a "Greater" list

$$w(e_{rand}) = p \begin{cases} w(e_i) \leq p & e_i \in ELesser \\ w(e_i) > p & e_i \in EGreater \end{cases}$$

After this partition, Filter-Kruskal's is recursively called on $ESmaller$ thereby not only limiting the number of edges given but also preferring edges of smaller weight which are more likely to be in the final MST F (assuming random input, which it is in the experimentation). Once the edge-list is of adequate length, Kruskal's is called on just those edges (sorting of those few edges are done within Kruskal's) and the appropriate ones are added to F, the disjoint sets are also modified here *Fig. 12*. In the case where the $ELesser$ list is not small enough, it is split again with a random pivot in a recursive call to Filter-Kruskal (line 17).

After recursing all the way down to a $ELesser$ list of operable size, the Filter operation is applied to the $EGreater$ list. At this step, each edge in $EGreater$ is compared against the disjoint set forest to determine if any of the edges cover two already connected vertices, in which case that vertex will not be

in the final list so it can be removed from $EGreater$, precluding the need to sort that edge later on. This step is the main difference between Filter-Kruskal's and Kruskal's algorithm and is mostly effective in larger data sets.

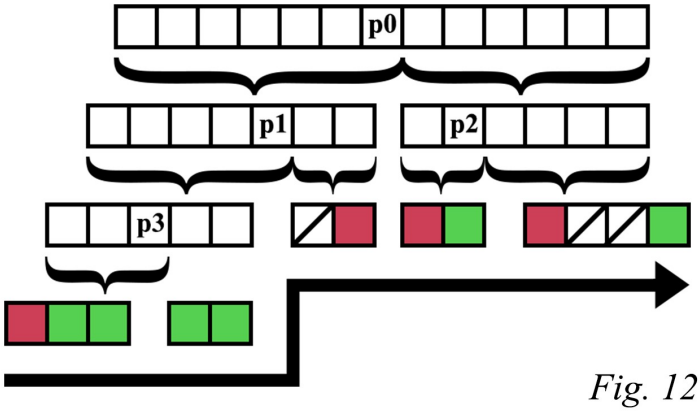


Fig. 12 This tree depicts a possible path that Filter-Kruskal could take through a graph.

While the edge lists are shown as arrays and partitioned accordingly above, each partition is not known to be sorted; just that every edge in the left partition has weight less than all those in the right partition. The *threshold* for this graph is set to 4 such that Kruskal's will only operate on edge lists of length 4 or less. After reaching the end of a recursion an edge is colored green if it's included, red if it isn't, and crossed out if it is removed by the *filter* step of the algorithm.

For the sake of brevity, the proof of correctness for Filter-Kruskal's Algorithm developed by Vitaly Osipov will not be included in this paper but can be found from the bellow source [11] or at the following link in the "Omitted Proofs" section of the appendix: http://algo2.iti.kit.edu/documents/fkruskal.pdf

The first of two lemmas used by Osipov and linked above will be mentioned here as it describes the distribution of runtime for the Filter-Kruskal's implementation and will be relevant to a later section on improvements.

Lemma 1: N is the number of comparisons done by FK (Filter-Kruskal) in relation to the weight of edges. Furthermore, FK will perform $\leq |V| - 1$ Union operations; each operation combines two unique vertices or non-inclusive sets of vertices so to combine $|V|$ vertices into one set in a disjoint set data structure always takes at least $|V| - 1$ combinations. In line with this, FK will perform $\leq 2 * |E| + C$ find operations and O($|E| + C$) work outside these operations [11].

Proof: The number of union operations is trivial as explained above (happens once every time FK encounters a new edge belonging in the MST). Notice that FK performs $2 * |E|$ `Find` operations, specifically in the part where it calls Kruskal's as a subroutine on smaller edge counts (This can be observed in the above pseudo code and more clearly in the code linked in the above repository in the `FilterVersion` class under the `ModifiedKruskal` static method). It follows that FK will perform the same number of `Find` operations as a standard implementation of Kruskal's in its calls to Kruskal's as a subroutine only in the worst case. Ideally, some calls to `Find` are done in the calls to `Filter`. After a pivot is picked (in line 14-16 in the above pseudocode), the following call to `Filter` will perform $|EGreater|$ comparisons and $(2 * (|E| - (rank\ of\ pivot)))$ `Find` operations. In a truly random set of

data, the rank of our chosen pivot can be expected to be $\frac{|EGreater|}{2} \Rightarrow \left(2 * \left(|E| - \left(\frac{|EGreater|}{2}\right)\right)\right) = |E|$ which the same as the number of comparisons mentioned above. All other operations are either comparisons or calls to `Find`, thereby proving the claim on the complexity [11].

7b. Observations

> *Note:*
>
> *For the sake of testing correctness and functionality of the written implementation, along with limitations in computing power available at this time, the* `Threshold` *method was set to return a flat output of 1000; meaning there will be a partitioning if the given list has more than 1000 edges. This is incredibly inefficient as* `Threshold` *should, in-fact, represent the point at which the implementation of* `Quicksort` *becomes inefficient and led to the long runtimes seen in the below data. When testing using the same methodology seen here, change the output of* `Threshold` *as desired.*

From the above illustration *Fig. 12* it can be seen that Filter Kruskal's takes the same path and does the same comparisons as a standard implementation of Kruskal's for any graph G, albeit with some small improvements and shortcuts. It follows that for small graphs that are relatively sparse, Filter-Kruskal's could display poorer performance than standard Kruskal's due to the extra steps in partitioning and recursion which innately cost runtime, even though they may not directly contribute to the complexity.

Indeed, this limitation becomes glaringly obvious in the following limited experiment that depicts consistently greater runtimes and a steeper increase in average runtime compared to standard Kruskal's, even for denser graphs which Filter-Kruskal's claims to be more suited for. This outcome was caused by the limited range of the vertices tested. The thousands of vertices in the previous experiments are not enough to highlight the problems with Kruskal's that emerges for larger sets of data, namely inefficiencies with `Quicksort` and repeated `Find` operations.

The inefficiencies in `Quicksort` show up in much larger data sets, typically in the millions, with repeated values that could cause problems in partitioning by introducing trivial steps that only sort one value but iterate through a great number of elements to do so. The overhead caused by partitioning along with the innate cost of memory manipulation in calling methods in Java result in poor performance seen in the limited testing provided here. However, in practice, Filter-Kruskal's will perform significantly better than Kruskal's for sets where sorting the entire array is inefficient using `Quicksort`.

To summarize:

- The behavior seen in these tests would ideally never happen as `Threshold` would define the point at which standard Kruskal's becomes inefficient and Filter-Kruskal's becomes applicable
- A correct `Threshold` will reduce the only overhead in Filter-Kruskal's to the check for length of the edge list and then comparison to the `Threshold`
- The actual functionality of `Threshold` is entirely system and implementation dependant where the runtime/complexity of the sorting algorithm chosen and the overhead caused by partitioning are the primary influences

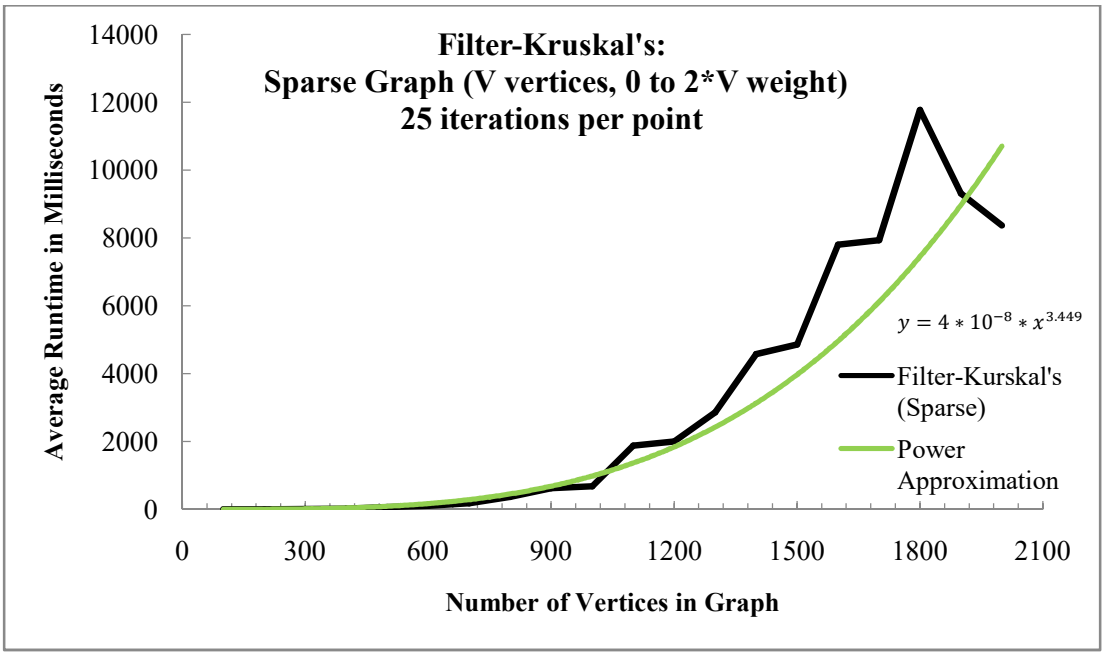


Fig. 15 Above is the runtime trend of Filter-Kruskal's Algorithm on sparse graphs for number of vertices V. Full data set can be found at the above linked repository.
(Data points were collected at 100 vertex intervals)

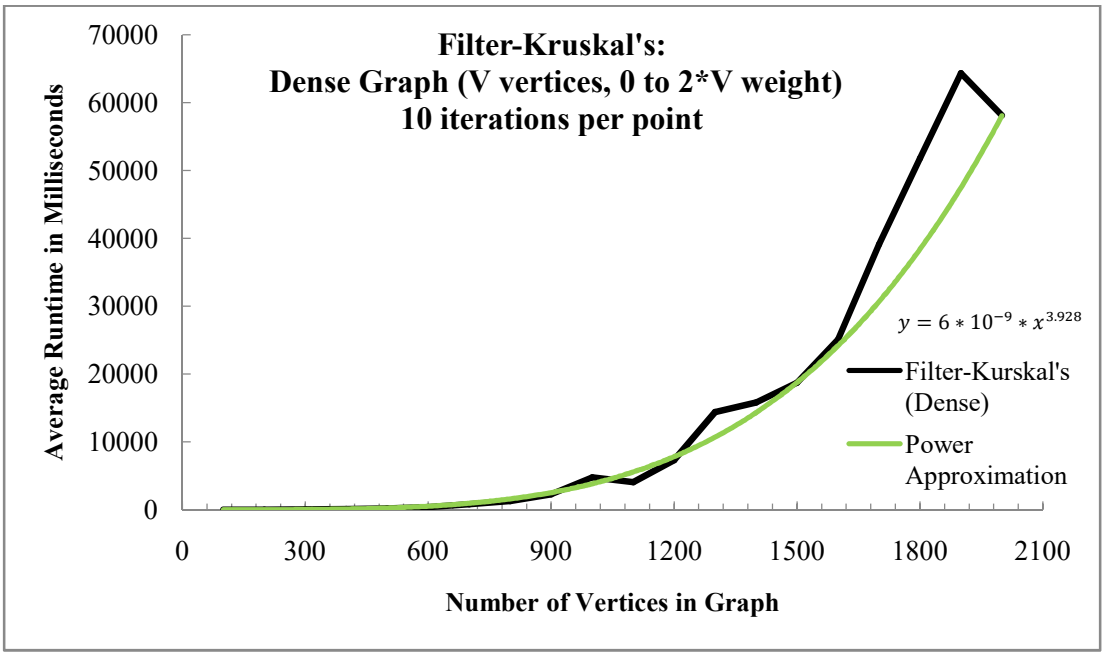


Fig. 16 Above is the runtime trend of Filter-Kruskal's Algorithm on dense graphs for number of vertices V. Full data set can be found at the above linked repository.
(Data points were collected at 100 vertex intervals)

There won't be much elaboration on this type of situation as it would never happen in correct application. The data shown is mainly here as proof of concept and in correlation to the steps outlines above.

Note:

The testing for the `FilterVersion` *(Filter-Kruskal's) algorithm was done using an explicitly defined stack in the JVM environment to prevent stack overflow. The default stack size in Java is too small to run tests on graphs with upwards of thousands of vertices. You may also increase stack size by passing a JVM argument when running tests:* `-Xss2048k` *will set the stack size to 2048kb.*

8. Improvements through Parallelization

From the diagram provided of a typical iteration through the Filter-Kruskal's Algorithm [Fig.12], it could be seen, when observed in a breadth first manner, that many steps spanning down the tree are locally independent of each other and in no significant order; ideally, the operation on one partition will not affect the work done on the other partition. However, the seemingly obvious implementation of depth based parallelism is not correct. The Union operations done in one instance of Kruskal's may be vital to those done in a separate partition.

While the framework of Filter-Kruskal's lends itself particularly well to parallelization, there are certain serial operations that should be preserved when doing so:

- Kruskal's should always be run on the set of partitioned edges containing the lowest weights first, only then progressing on to higher ones
- The $EGreater$ edge list must always be filtered before Filter-Kruskal is recursively called upon it
- Writes to any disjoint sets, lists, or arrays must be done in order but reads can be concurrent

When viewed with these requirements, the options for parallelization are not as far-reaching as they first seem. These few rules have effectively limited the traversal of the above shown tree [Fig.12] to one branching path at a time, by order of weights. This does not mean that the algorithm is wholly resistant to parallelization; each partition may still perform certain tasks in local concurrency:

- The sorting done for one instance of Kruskal's can be done in parallel
- Partitioning an edge list can be done in parallel such that the list of edges smaller than the pivot could be completed before those that are larger
    - This improvement is very situational in that it only shows promise in large data sets where a significant number of additions to a list cause noticeable overhead
    - No improvement in complexity can be seen, the list is iterated over twice cause O($2n$), but there will be an improvement in runtime
- The removal of edges in `Filter` can be done in parallel as there is no addition throughout the whole process; Keep in mind that there could possibly be overlapping removals

While keeping in mind the requirements and freedoms allotted, certain parts of the Filter-Kruskal method were parallelized through the implementation of multithreading. The sorting algorithm used in the standard Kruskal subroutine was improved through the use of a parallel `Quicksort` as defined by Java's `Arrays.parallelSort` method. This implementation of parallel sorting uses a fork/join structure to split an array into smaller partitions, sort those, then merge at the end. It should be noted that this implementation of parallel sort only shows greater efficiency when inputs are of length greater than 10,000 items. Any smaller data sets can be sorted faster using the standard `Quicksort` algorithm; this pushes the `Threshold` value up to at least 10,000.

The partitioning and removal cases are relatively straightforward in their parallelization. In the implementation of Filter-Kruskal linked above in the repository, there was no parallelization of partitioning done as it would be trivial and even harmful for smaller sets of data that were tested.

> *Note:*
>
> *Multithreading in Java done by the built in Runnable class is subject to some restrictions. While there is a maximum of 256 threads per Java application, parallelism only occurs when there are fewer threads running than physical cores on the CPU. Any threads created above this limit will be run concurrently but not truly in parallel, thereby reducing efficiency. 6 cores were used in this paper. The exact bounds for the efficiency of a parallel sort relative to the number of CPUs can be found in [13].*

Filtering was done in a parallel manner by splitting up the $EGreater$ edge list into even partitions based on the number of cores available in the system. These sublists were processed independently then merged back into one to be used in later steps.

At the time of writing this paper (12/01/2022), the implementation used did not prove to be thread-safe on LTS versions of Java as it used experimental features, this will be resolved and refactored soon.

9. Conclusion

Throughout this paper and these experiments, the applications of solving MSTs and their use has been justified, multiple approaches to the problem have been considered, and the latest solutions to the MST problem were discussed and implemented. The data produced confirms the proven bounds on complexity while also going more in-depth to discuss the practical runtimes of these algorithms.

A point of emphasis that should have been more accentuated was that of parallelization. The process of developing algorithms that lend themselves well to parallel processing is a counterintuitive one that requires the algorithm to be expanded and sometimes convoluted in order to provide more access to its inner workings. In Kruskal's algorithm, the once compact and efficient algorithm had to be broken down into its core steps using the proofs it was based off of in order to find locations to implement parallel structures.

The scalability of parallel processing is also not to be understated; parallel merge sort has a time of $O(log\ |E|)$ for $|E|$ input keys using $|E|$ processors. As technology improves and more computational resources are made available, certainly more than what were available to me throughout this project, the direction of algorithms will likely shift away from the compact scene shown today. One final remark on things that could have been done differently and would certainly be improved on in future iterations of this project:

- Larger datasets for parallel algorithms
- More computational resources to run longer tests with more dense graphs
- Test other variants of graphs such as ones with many duplicate weights
    - Kruskal's is known to suffer with these; particularly due to the nature of Quicksort
- Further exploration into alternative MST algorithms such as Prim's which is known to do better with denser graphs than Kruskal's
- Implementation in a more memory efficient language with less overhead than Java

10. Bibliography

1. R. L. Graham, Pavol Hell. *On the History of the Minimum Spanning Tree p. 43 – 54,* January 1985.

2. Ellis Horowitz, Sartaj Sahni. *Fundamentals of Computer Algorithms p. 175 – 187*, 1978.

3. Lance Fortnow. *The Status of the P versus NP Problem,* Northwestern University. September 2009.

4. N. Christofides. *Worst case analysis of a new heuristic for the traveling salesman problem.* Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, 1976.

5. Myint Than Kyi, Lin Lin Naing. *Application of Ford-Fulkerson Algorithm to Maximum Flow in Water Distribution Pipeline Network*. Technological University (Mandalay), December 2018.

6. Jason Eisner. *State-of-the-Art Algorithms for Minimum Spanning Trees*. University of Pennsylvania, 1997.

7. Seth Pettie, Vijaya Ramachandran. *An Optimal Minimum Spanning Tree Algorithm*. The University of Texas at Austin, Austin, Texas, January 2002.

8. Pallavi Jayawant, Kerry Glavin. *Minimum spanning trees*. Department of Mathematics, Bates College, March 2009.

9. J. Nešetřil, E. Milková, and H. Nešetřilová, "*Otakar Borůvka on minimum spanning tree problem: translation of both the 1926 papers, comments, history*", Discrete Math, 2001.

10. B. A. Wichmann. *Ackermann's Function: A Study In The Efficiency Of Calling Procedures*, (https://history.dcs.ed.ac.uk/archive/docs/Imp_Benchmarks/ack.pdf), National Physical Laboratory, Teddington, Middlesex, January 1976.

11. Vitaly Osipov, Peter Sanders, and Johannes Singler, *The Filter-Kruskal Minimum Spanning Tree Algorithm*, Karlsruhe Institute of Technology, Germany, January 2009.

12. K. Noshita. *A theorem on the expected complexity of Dijkstra's shortest path algorithm. Journal of Algorithms, 6:400–408, 1985.*

13. Minsoo Jeon, Dongseung Kim *Parallel Merge Sort with Load Balancing*, Department of Electrical Engineering, Korea University, Seoul, 2022.