CSC505 - Project 2 Report

Andy Sauerbrei (amsauerb@ncsu.edu)
AJ Bulthuis (ajbulthu@ncsu.edu)
Tyrone Wu (tkwu@ncsu.edu)

April 12, 2023

1 Experimental Design

In our experiment, we plan on experimenting in how the density of graphs affect the `runtime` and total number of `compares` on the following MST algorithms:

- Prim-Jarnik's (edge-based) with lazy deletion on Binary Heap
- Prim-Jarnik's (vertex-based) with `decreaseKey` on D-ary Heap.
- Kruskal's algorithm with Disjoint Sets and Union by Rank on Binary Heap.

1.1 Hypothesis

Hypothesis 1: When selecting the optimal value of k in D-ary Heap, we expect optimal performance around $k = 3$ or $k = 4$ since the overall efficiency of `decreaseKey` outweighs the tradeoff of `removeMin`. As the value of k increases, we expect to see the efficiency of `decreaseKey` fall off.

Hypothesis 2: Even though Prim-Jarnik's algorithm is known to perform better in dense graphs, we expect that the efficiency of a Disjoint Set (with union by rank) in Kruskal's algorithm will outperform Prim-Jarnik's lazy deletion variant. However, when comparing performance of Prim-Jarnik's `decreaseKey` variant with Kruskal's algorithm, we also expect that Prim-Jarnik's `decreaseKey` variant will out perform Kruskal's in dense graphs.

*Note: The data structures and functions mentioned above will later be discussed in more detail.

1.2 Methodology

For the trials, we have generated the graphs using the following parameters:

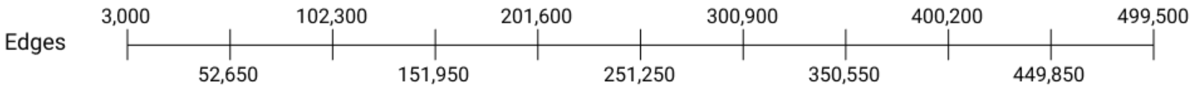
- `Vertex Size`: The number of vertices in the graph.
- `Edge Size`: The number of edges in the graph. This value determines how dense the graph is.
- `Seed`: The seed used to generated the graph.
- `Max Edge Weight`: The maximum possible weight of an edge.
- `Graph Type`: The type of graph, which is set to `rn` (random).

1.2.1 Vertex Size & Edge Size

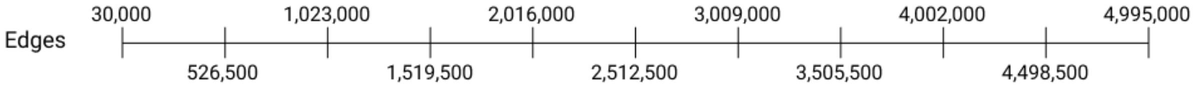
On the top level, our generated graphs start with vertex sizes (V) on 1,000 and 10,000. Of those graphs, we set the number of edges to increment linearly from 3 times the number of vertices ($3 * V$), up to the max possible edges $\left(\frac{V * (V-1)}{2}\right)$. The number of edges increment 10 times, which is calculated from:

$$\left(\frac{\frac{V * (V-1)}{2} - 3 * V}{10}\right)$$

Number of Vertices: 1,000
Number of Edges:



Number of Vertices: 10,000
Number of Edges:



1.2.2 Seed

For each graph with the specified number of vertices and edges, we have arbitrarily selected seeds 1, 2, 3, 4, and 5 to generate 5 graphs.

1.2.3 Max Edge Weight

The maximum possible weight of an edge is set to be equal to the number of vertices in the graph. This ensures that the total weight of the minimum spanning tree does not overflow the `unsigned long long` data type, and still preserve a wide range of possible weights in the graph.

2 Implementation Description

2.1 Graph Data Structures

Our `graph` data structure contains the following attributes:

- `vertices`: Array of `vertex` objects.
- `edges`: Array of `edge` objects.

2.1.1 Vertex Data Structure

The `vertex` data structure keeps track of the following attributes:

- `id`: The unique ID of the vertex $(v_1, \cdots, v_n)$.
- `x`: The x coordinate of the vertex (optional).
- `y`: The y coordinate of the vertex (optional).
- `incidentEdges`: Array of `edge`s incident to the vertex.
- `idHeap`: The location of vertex in the D-ary Heap.
- `edge`: The `edge` associated with the vertex in the D-ary Heap.
- `distance`: The distance of the vertex in the D-ary Heap. This is initially set to $+\infty$.
- `marked`: Whether the vertex has been visited or not. This is initially set to `false`.

2.1.2 Edge Data Structure

The `edge` data structure keeps track of the following attributes:

- `source`: The index position of the source vertex in the array.
- `destination`: The index position of the destination vertex in the array.
- `weight`: The weight of the edge.

2.2 D-ary Min-Heap

All the minimum spanning tree algorithms utilize the D-ary Heap data structure. Instead of a traditional Binary Heap which only contains 2 children, the D-ary Heap allows for D number of children. The value of D is calculated from a given k, in which $D = 2^k$. The underlying structure of the D-ary Heap is an an array. The notable operations of the D-ary Heap are:

- `heapify`: Performs downwards swapping with the D children until the properties of min-heap is satisfied.
- `removeMin`: Swaps the first and last element in the array, and removes/returns the last element of the array. Then call `heapify` on the root vertex (which is first element of the array).

- `addEdge`: Appends the `edge` to the end of the array, and performs upwards swapping with the parent until the properties of min-heap is satisfied (using edge weight as the metric).
- `decreaseKey`: Updates the vertex's distance and performs upward swapping with the parent until the properties of min-heap is satisfied (using vertex distance as the metric).

2.3 MST Algorithms

2.3.1 Prim-Jarnik's (edge-based) with lazy deletion on Binary Heap

This MST algorithm utilizes the binary heap as a priority queue of edges and views the graph as a list of vertexes, with incident edges, that may or may not have been added to the MST. The algorithm starts by taking the first vertex out of the list, marking it as added, and then adding its incident edges to the priority queue. The algorithm then enters a loop of removing the minimum edge from the priority queue, checking to see if both its endpoints have been added to the MST, and if one of them has not been added, its incident edges are added to the priority queue if they end in a node that has not been added and the node is marked as added. The aforementioned loop continues until the priority queue is empty.

2.3.2 Prim-Jarnik's (vertex-based) with `decreaseKey` on D-ary Heap

Unlike the prior variation of Prim-Jarnik's, this one makes use of a D-ary Heap based priority queue of vertexes where each vertex has an associated distance, parent edge, and queue index—each of these values begins as ∞, NULL, and the given starting location in the priority queue respectively. Additionally, the priority queue treats the distance as the key by which to sort values and the vertex as the element. The algorithm begins by adding all of the vertexes to the heap before entering its main loop. While the priority q is not empty, the minimum vertex is removed and added to the MST, and then its incident edges are observed. For each of the incident edges we look at the endpoint, if the weight of the edge is less than the distance associated with that endpoint, that vertex has associated its distance as well as edge to correspond with the given edge, and the `decreaseKey` method is applied to that vertex to update its position in the priority queue.

2.3.3 Kruskal's with Disjoint Sets and Union by Rank on Binary Heap

Our implementation of this MST algorithm uses the binary heap to mimic a priority queue of edges that it takes from the given graph. The graph's vertices are then each given a disjoint set of themselves in order to guard against back edges being added to the MST during calculation. The queue treats the weight of each edge as the way to tell which edge to output each iteration. The queue pops the minimum weight edge each time we loop through algorithm. It then gets the parent of each vertex of the edge, the parent being the root node for the disjoint set that vertex is a part of. If the two vertices don't have the same parent, then the algorithm knows that it can add the edge to the MST, and then merge the two sets together into one. This loop continues until the queue is empty, and then the MST is output.

2.4 Comparison Counter

The global `compares` counter is incremented whenever two `edge` weights are compared. This occurs whenever `heapify` (`removeMin` calls `heapify` internally) and `addEdge` is called.

In Prim-Jarnik's vertex-based algorithm, `compares` is also incremented whenever two `vertex` distances are compared in `decreaseKey`, and whenever a `vertex`'s distance is compared to an `edge`'s weight.

In Kruskal's algorithm, `compares` is also incremented every time a `findParent` operation is performed in the `disjoint set` data structure.

2.5 Experimental Environment

2.5.1 Compilation Specifications

Our entire implementation of the program was done in `C++` with the following compilation specification:

- `C++` Standard: `C++ 20`
- Compiler: `clang++`
  - Version: `Debian clang version 11.0.1-2`
  - Target: `x86_64-pc-linux-gnu`

2.5.2 Hardware:

The experiment was conducted inside a Linux (Debian) Docker container on a Windows machine with the image:
`mcr.microsoft.com/devcontainers/cpp:latest`:

Hardware specification (including the container):

- System Type: 64-bit OS, x64-based processor
- Operating System:
  - Linux (Debian) Container: Linux version 5.4.72-microsoft-standard-WSL2 (gcc version 8.2.0 (GCC))
  - PC: Windows 10 Education Version 21H2 (OS Build 19044.2604)
- CPU:
  - Model Name: AMD Ryzen 5 2600X Six-Core Processor
  - Base Speed (MHz): 3.60 MHz
  - Cores: 6 cores
  - Logical Processors: 12 processors
    - Processor Speed (MHz): 3.60 MHz
    - Cache Size (KB): 512 KB
- Memory:
  - Total Physical Memory (GB): 16 GB
  - Total Virtual Memory (GB): 26 GB
  - RAM Speed (MHz): 2667 MHz
  - RAM Type: DDR4

3 Experimental Results

3.1 Hypothesis 1 Revisited

Prim Runtimes on D-ary Heapat 1,000 Vertexes						
Edges	K=1	K=2	K=3	K=4	K=5	K=6
3,000	0.00	0.00	0.00	0.00	0.00	0.00
52,650	0.01	0.01	0.01	0.01	0.01	0.01
102,300	0.01	0.01	0.01	0.02	0.02	0.02
151,950	0.02	0.03	0.02	0.03	0.02	0.03
201,600	0.04	0.04	0.04	0.04	0.04	0.04
251,250	0.05	0.05	0.05	0.05	0.05	0.05
300,900	0.06	0.07	0.07	0.06	0.07	0.07
350,550	0.08	0.08	0.08	0.08	0.08	0.08
400,200	0.09	0.10	0.10	0.10	0.09	0.09
449,850	0.11	0.11	0.11	0.11	0.11	0.11
499,500	0.13	0.13	0.13	0.12	0.12	0.12

Table 1: Prim-Jarnik's `decreaseKey` variant runtimes on $V = 1,000$ & varying $|E|$

Prim Runtimes on D-ary Heap at 10,000 Vertexes						
Edges	K=1	K=2	K=3	K=4	K=5	K=6
30,000	0.02	0.02	0.02	0.02	0.03	0.04
5,026,500	1.90	1.90	1.96	1.68	1.77	1.76
10,023,000	4.10	4.14	4.14	3.42	3.58	3.65
15,019,500	6.49	6.50	6.51	5.58	5.62	5.69
20,016,000	8.88	9.04	9.07	7.94	8.00	8.08
25,012,500	11.59	11.55	11.65	10.54	10.54	10.60
30,009,000	14.39	14.40	14.42	13.23	13.33	13.31
35,005,500	17.25	17.20	17.33	15.95	16.42	16.32
40,002,000	20.14	19.99	20.23	19.24	18.72	19.30
44,998,500	23.11	23.19	23.15	22.01	22.16	22.00
49,995,000	26.07	26.15	26.14	25.12	25.10	-

Table 2: Prim-Jarnik's `decreaseKey` variant runtimes on $V = 10,000$ & varying $|E|$

Looking at the difference between runtimes amongst the three different k values we didn't really get great values for our 1,000 vertex graph above (Table 1), runtimes often only varied by a fraction of a second and there wasn't really a clear winner amongst them. The values were a bit better for the 10,000 vertex graph above (Table 2), but values at most only varied by about a second at the highest numbers of edges. For this bigger graph the higher k values, specifically $k = 4$ and $k = 5$, seem to have a slight edge once we get past the first row of values.

Prim Compares on D-ary Heap at 1,000 Vertexes						
Edges	K=1	K=2	K=3	K=4	K=5	K=6
3,000	28,832	27,318	32,529	45,718	67,142	116,646
52,650	133,349	130,303	135,062	148,084	168,953	218,692
102,300	233,656	230,286	234,912	247,831	267,995	317,647
151,950	333,489	329,926	334,105	347,266	366,664	415,946
201,600	433,040	429,446	433,616	446,776	465,612	514,896
251,250	532,449	528,691	532,688	545,645	564,349	613,579
300,900	631,724	627,757	631,754	644,982	662,705	710,761
350,550	730,564	726,780	730,880	743,784	760,586	809,022
400,200	829,801	826,197	829,910	843,443	858,715	906,509
449,850	928,619	925,154	928,872	942,347	956,751	1,003,182
499,500	1,027,181	1,023,908	1,027,911	1,040,875	1,054,383	1,100,260

Table 3: Prim-Jarnik's `decreaseKey` variant comparisons on $V = 1,000$ & varying $|E|$

Prim Compares on D-ary Heap at 10,000 Vertexes						
Edges	K=1	K=2	K=3	K=4	K=5	K=6
30,000	376,991	349,907	423,335	589,700	933,046	1,465,539
5,026,500	10,462,110	10,408,864	10,473,413	10,634,456	10,972,177	11,509,893
10,023,000	20,467,340	20,408,038	20,468,195	20,624,843	20,958,173	21,490,438
15,019,500	30,465,816	30,402,951	30,460,232	30,613,717	30,938,463	31,473,912
20,016,000	40,460,898	40,393,693	40,447,965	40,598,646	40,917,241	41,451,299
25,012,500	50,455,226	50,384,633	50,440,214	50,592,124	50,898,508	51,441,461
30,009,000	60,439,998	60,373,786	60,423,920	60,569,797	60,888,866	61,409,822
35,005,500	70,429,510	70,364,030	70,415,262	70,559,795	70,860,378	71,400,030
40,002,000	80,414,111	80,352,370	80,403,768	80,552,703	80,828,484	81,406,284
44,998,500	90,398,886	90,336,343	90,390,685	90,540,114	90,791,317	91,399,250
49,995,000	100,382,651	100,321,616	100,378,692	100,524,897	100,751,069	-

Table 4: Prim-Jarnik's `decreaseKey` variant comparisons on $V = 10,000$ & varying $|E|$

The results for runtimes were odd when compared to our results for the number of comparisons. For both the 1,000 and 10,000 vertex graphs (Tables 3 and 4) above we had K=2 having an overall smaller number of comparisons across the board with the differences being the greatest at 30,000 edges in the 10,000 vertex graph (Table 3) with the closest two Ks being $k = 1$ and $k = 3$ at 1.08 and 1.20 times as many comparisons respectively.

This difference in the comparisons is reflected in the 10,000 vertex runtimes table (Table 2) for the first row, but while we would expect a bias towards $k = 2$ as it routinely has a mildly smaller number of comparisons the bias seems to be toward the higher k values. We think that this may indicate that there is a discrepancy between the `decreaseKey` and remove min methods where there is a comparison for `removeMin` is less costly than a comparison for `decreaseKey`. For `decreaseKey`, each comparison would equate to the decision to swap with a node or not as the node is propagating up the tree (if it is now of lower value than its parent there should be a swap), while for `removeMin` a value needs to make k number of comparisons as for the decision to swap as it is propagating down the tree (it needs to swap with the minimum of its $D = 2^k$ children if it is lower than it).

3.2 Comparing The Three Algorithms

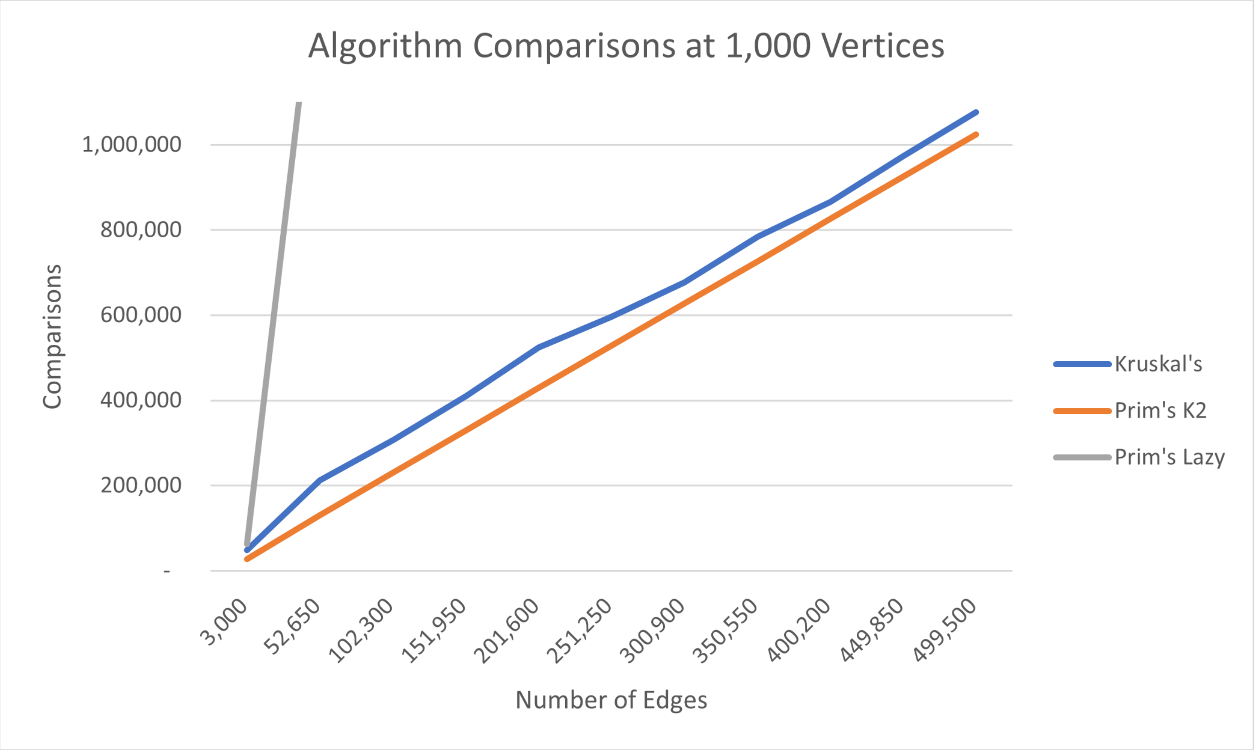


Figure 1: Graph Comparing the number of Comparisons for each Algorithm at 1,000 Vertices

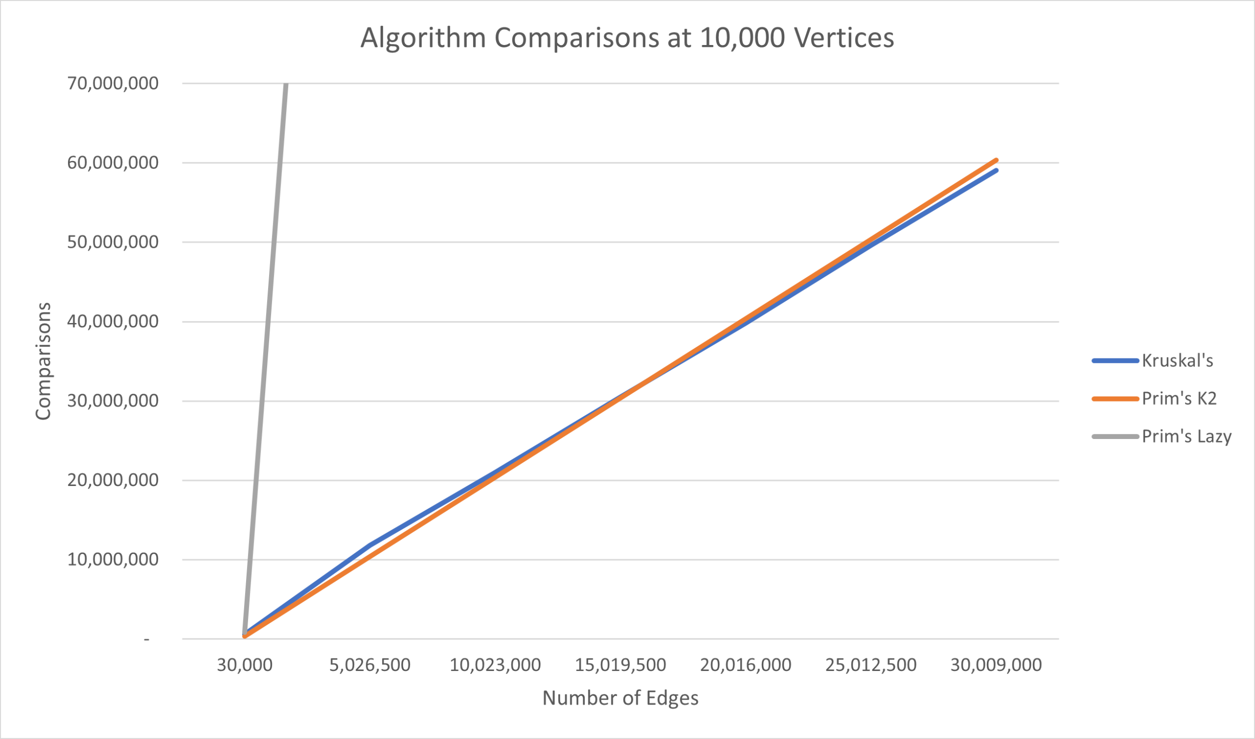


Figure 2: Graph Comparing the number of Comparisons for each Algorithm at 10,000 Vertices

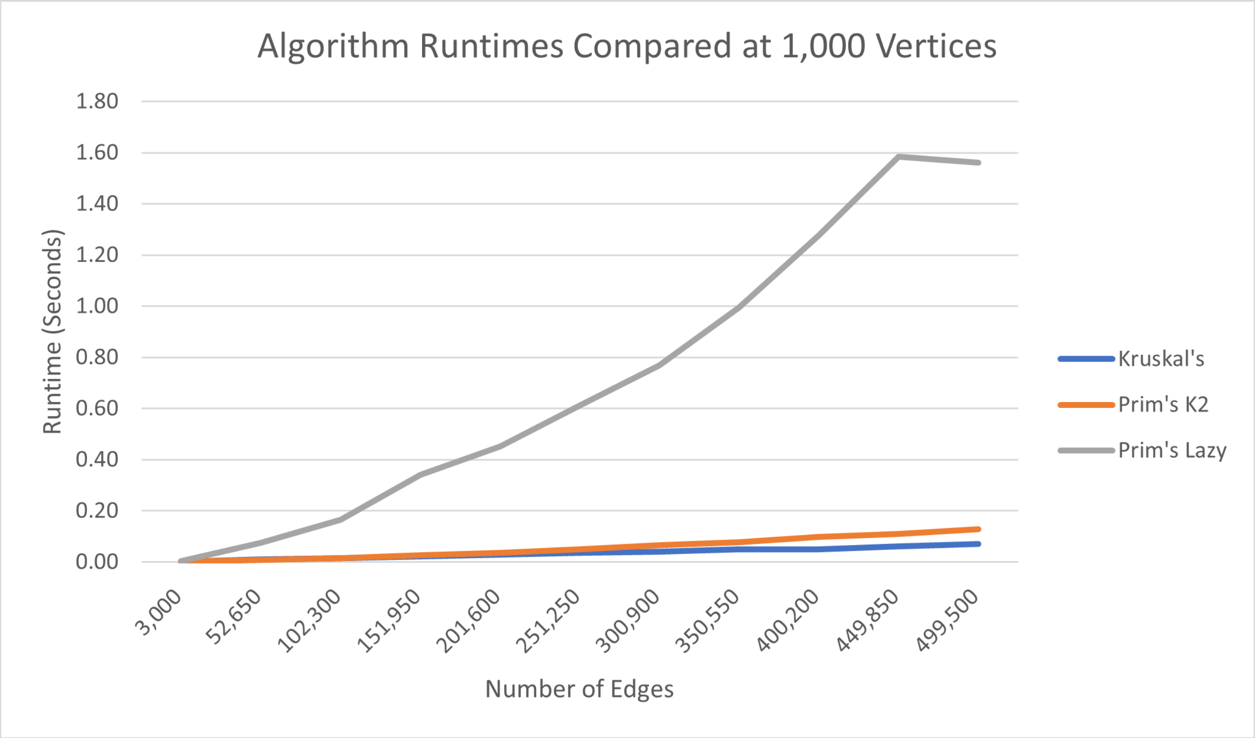


Figure 3: Graph Comparing the Runtime for each Algorithm at 1,000 Vertices

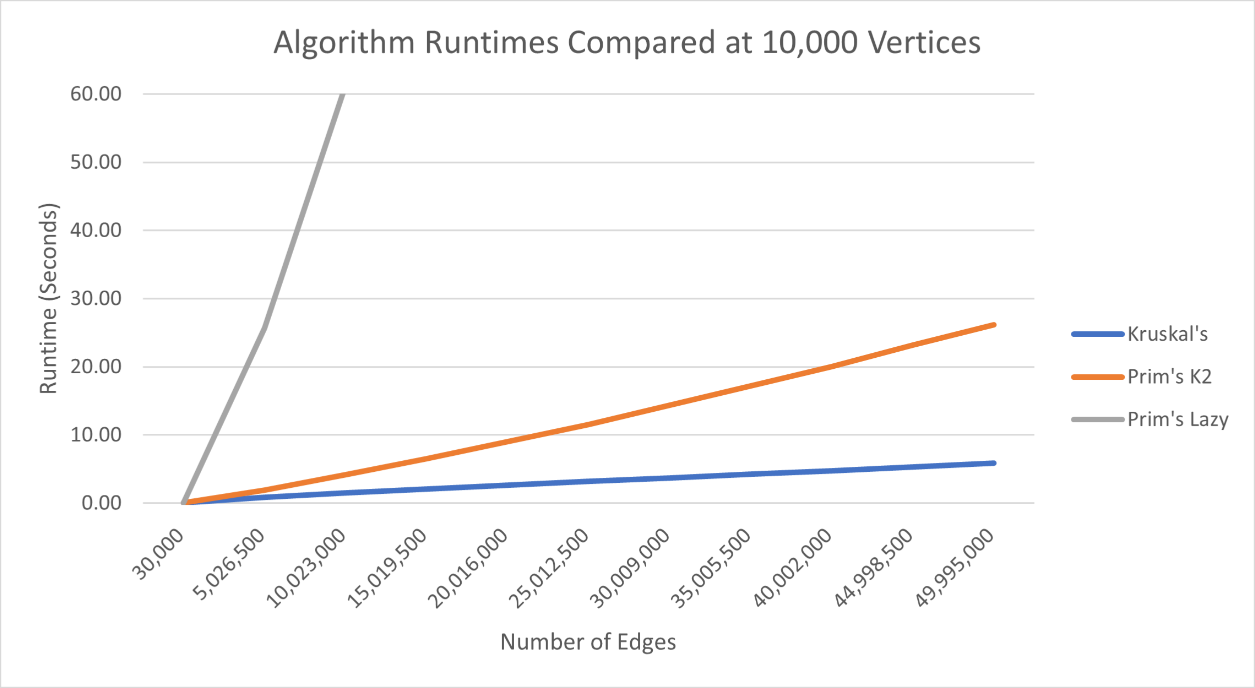


Figure 4: Graph Comparing the Runtime for each Algorithm at 10,000 Vertices

3.3 Hypothesis 2 Revisited

Our second hypothesis revolved around us determining which algorithms we thought would outperform each other. We thought that Prim-Jarnik's would do the worst with the lazy delete. This method of comparing would result in the runtime and number of comparisons to increase much quicker than the other algorithms we were going to test, so we theorized that it would do the worst. As we can see in our comparison graphs above, Figures 1 and 2, the lazy delete version of Prim's, which is the grey line, is very clearly the worst in terms of comparisons. We see the same trend in the runtime graphs, Figures 3 and 4. So this part of the hypothesis we were spot on about.

The second half this hypothesis was that Kruskal's would do better than the Lazy Delete Prim but not better than the `decreaseKey` Prim. Through our analysis of the two algorithms and how they would function, we thought that the decrease key technique would result in better runtimes for Prim's compared to the Disjoint Sets of Kruskal's. Our data showed, however, that the `decreaseKey` variant of Prim's performed as good or slightly worse the Kruskal's as the graphs got denser. In Figures 3 and 4, you can see that the blue and yellow lines remain very close to each other, but the blue (Kruskal's) begins to pull away as the density of the graph peaks. This flew in the face of what we had expected to occur, as we thought that the DK Prim would continue to do better as the graph density increased. This same discrepancy can be seen in Figures 1 and 2, where Kruskal's has more comparisons on smaller, less dense graphs, but as we increased the number of vertices and the number of edges, Kruskal's outperformed `decreaseKey`. In the end, we theorize that we might have underestimated the efficiency of the Disjoint Sets in Kruskal's given that it uses Rank to help be more efficient.

4 Conclusions and Future Work

4.1 Conclusion

Overall, the results that we collected were not what we expected. We found that Kruskal's algorithm (using the Disjoint Set data structure) turned out to perform the best in terms of runtimes on all densities of the graphs. Despite our initial hypothesis stating that Kruskal's algorithm should perform worse on highly dense graphs, the algorithm performs about the same, if not better, than Prim-Jarnik's `decreaseKey` variant.

When we initially implemented Prim-Jarnik's algorithm, we started with the lazy deletion variant using a Binary Heap. The results that we collected were far worse when compared to the performance of Kruskal's algorithm, so we decided to also implement the `decreaseKey` variant as well as the D-ary Heap for variable branching. In the end, the our results were still not what we expected to see. However, it was interesting to see how the efficient Kruskal's algorithm was using the Disjoint Sets data structure that was mentioned in class.

While generating the graphs for the experiment, we capped the number of edges at 50,000,000 so we did not spend too much time generating the input (about 4 hours to generate the input, which totals out to 21.5 GB for the dataset). In hindsight, one explanation for our results could be that we did not experiment on large enough graphs to truly observe the performance.

4.2 Future Work

In terms of future work, we would like to delve deeper into the following items of interest:

- Extend the experiment to run on 500,000,000 edges, or the maximum possible edges until the program crashes.
- The smaller quantity of comparisons in Prim's with $k = 2$ in sparse or smaller graphs when compared to Kruskal's (Figures 1 and 2) and how that doesn't necessarily equate to better runtimes.
- See how Kruskal's algorithm compares to Prim's algorithms when it is not enhanced by disjoint sets.
- Run the experiments again with a static number of edges and a scaling quantity of vertexes.