Final Project report

Implementation and Comparison of Dijkstra's single source with Network Scaling algorithm

Team Xi

Matthew Pudlo
mrpudlo@ncsu.edu

Christian Morris
cgmorris@ncsu.edu

Aakriti Aakriti
aaakrit@ncsu.edu

Nagashree Mulukunte Nagaraju
nmuluku@ncsu.edu

Introduction:

Program three is an implementation and comparison of the performance of two algorithms namely Dijkastra's single source algorithm and the Network Scaling algorithm - derived from "Scaling Algorithms for Network Problems" on all types of graph instances.

- **Dijkstra's single source algorithm**: It is a greedy algorithm for finding the shortest paths between nodes in a graph from a single source. This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights. Our objective is to do experiments for runtime and number of edge weight comparisons. This algorithm can be run on each node of the graph and have results combine to produce

    - **Min Heap Data structure**: A min-heap is a binary tree such that - the data contained in each node is less than (or equal to) the data in that node's children.For edge weight comparisons, we implemented our own heap data structure which was required to perform decrease key and extract min operations for Dijkastra's algorithm. With these operations our Heap can act as the priority queue needed for the algorithm.

    - **D-ary Heap Data Structure:**.The *d*-ary heap is a priority queue data structure, a generalization of the binary heap in which the nodes have *d* children instead of 2.This data structure allows decrease priority operations to be performed more quickly than binary heaps, at the expense of slower delete minimum operations. This tradeoff leads to better running times for algorithms such as Dijkstra's algorithm in which decreased priority operations are more common than delete min operations. This is the second data structure we implemented for better comparison of edge weight in Dijkstra's shortest path algorithm

- **Network Scaling algorithm**: This algorithm serves as a scaling approach that is to improve upon a "near-optimum" solution to the shortest paths problem. A "near-optimum" solution would include a function such that the shortest path from the source to itself is zero, the minimum distance from source to destination (d_i) dominates any edges between two destinations, and that each vertex has a path from the source that is of weight between d_i and $d_i + m$, the number of edges in the graph. By iteratively or recursively breaking the shortest path problem into subproblems that can be solved efficiently, O(m)[3], the total runtime of the algorithm can be improved by transforming the sub optimal solution for the graph to an optimal one. A suboptimal solution can be found by solving a shortest path sub problem on a scaled down graph where the graph has identical vertices and edges, but with edge weights of half the original weight.

Implementation Description:

All graphs for this project were represented as an adjacency matrix. This matrix was implemented as a two dimensional array in java. This array is created by reading by reading the .gph file supplied to std in and is then passed as an argument to any functions where it is used.

Dijkstra's algorithm was implemented using the method provided in Introduction to Algorithms[2]. The priority queue was implemented in one of the two following ways.

D-ary heap implementation

D-ary Heap is implemented in consideration to be used for Dijkstra's as a priority queue. A heap class and HeapNode class are created for this implementation. Class DHeap implements a d-ary minimum heap data structure satisfying two properties: It must be a complete d - tree and the parent key must be minimum among all of its d children. Class HeapNode represents a data structure whose object when created stores two values vertex and distance. It is also used to create an array of HeapNode type objects in DHeap class. The major operation of the Heap class are as follows:

<u>percUp()</u> - percolate the newly added node up to its proper position in the binary minHeap
<u>percDown()</u> - Iterative method to restore the property of binary minHeap
<u>decreasekey()</u> - Decrease distance corresponding to a Vertex to a new value and restore min heap property by percolating up.
<u>extractMin()</u> - remove the top node(from index 1) from the binary minHeap and percolate down(to fix the violated minHeap's property)
<u>insert()</u> - insert a new node at the end of minHeap and percolate up(if the minHeap's property is violated)

As d ary heap is a heap having d children , consequently to calculate the child, it uses: d*(i-1) + k+1
where d is the no of children, i is the index of a parent for which child is to be calculated, and k={1,2...d} which child out of all d children to be found

Whenever a new node is inserted in the heap, it takes HeapNode type object and calls percUp() to maintain the property of the d-ary minheap and whenever the deletion of a node happens, it calls the percDown() which removes the root node and put the last node in the root node position and then swaps until node reaches to its right position in heap. Both percUp() and percDown() operations are implemented as an iterative approach.

DecreaseKey operation is a very common operation in Dijkstra's, it will change the distance corresponding to a vertex in HeapNode object to a new value and then call percUp() to restore the heap property.
A part of calculating a total number of comparisons is, to sum up the total number of Comparisons that happened during extractMin operation and DecreaseKey operation respectively. With each dijkstra call(with a source), creates a new priorityQueue object, then calculate total number of Comparisons for
- DecreaseKey operation: whenever upon each call to DecreaseKey operation happens in dijkstra logic, we calculate how many times in percUp operation's swaps have happened and in end returned a globally maintained variable totalNumHeapPercUp.
- extractMin operation : whenever, upon each call to extractMin operation happens in dijkstra logic, we calculate how many times in percDown operation's swaps have happened and in end returned a globally maintained variable totalNumHeapPercDown.
This data structure improves the runtime of Dijkstra's as compared to binary heap.

Min Heap Implementation:

Min heap is implemented in consideration to be used for Dijkstra's as a priority queue. It is a heap with 2 children and the parent key is minimum among both of its children . The major classes Dheap and HeapNode are kept the same along with major operations namely insert, extractmin, DecreaseKey, percUp, percDown.
Both percUp() and percDown() operations are implemented as an iterative approach.
To get the child of binary min heap, the implementation used 2*(i-1) + k+1
where 2 is the no of children, i is the index of a parent for which child is to be calculated, and k={1,2} which child out of both children to be found.
Min Heap is a primitive data structure to be used with dijkstra's which is less efficient as compared to D-ary heap for optimizing the runtime.
The most common DecreaseKey operation is not very fast in binary heap because performing a DecreaseKey we change the priority of a node in the tree and repeatedly swap it up with its parent until it either hits the root or its priority ends up becoming smaller. In worst case the number of swaps are given by the height of the heap which is O(lgn) in binary heap whereas for d-ary heap, the height of tree is given by $O(\log_d n)$ which is less for d > 2 and as a result with increase in number of children, the height of heap decreases and operations become fast, thus improving the runtime.

Scaling Implementation

The scaling technique to achieve a single source shortest path is done by iteratively breaking down the solution to scaled down sub problems that can be solved efficiently. The iterative approach for this project is adapted from an approach used by the MIT Computer and Artificial Intelligence Laboratory[4]. In our approach the reduction to sub optimal solutions is done by exposing one bit of the edge weight lengths at a time starting with the most significant bit. Unlike Kelly and schardl's approach our implementation does not perform an initial iteration of dijkstra's algorithm with no bits exposed. The purpose of the initial iteration is to test for connectivity in the graph between nodes and to initialize all connected nodes in the graph with a base distance of zero(since no bits are exposed). Since our implementation will be with undirected graphs we know all nodes will be connected allowing us to to begin iterating with the most significant bit exposed.

The amount of Iterations is dictated by the number of bits needed to represent the largest edge and is solved by taking the base two logarithm of the Largest edge weight plus one. This iterator is also used to right shift the edge weights to scale them down for the sub problem.

To perform shortest path comparisons a modified edge weight is used. This combines the newest exposed bit with the scaled up distance from the previous round. This allows scaling to work as this modified distance meets the criteria for a sub optimal solution allowing for the conversion to optimal solution.

Since the shortest path sub problems are solved by still using a version of dijkstra's algorithm, the efficiency of the overall algorithm is highly dependent on how the priority queue is implemented. If the priority queue is implemented properly, it can allow dijkstra's to be performed in O(m). In theory this is performed by using an array as a priority queue. The array must be of size k where 0<= k <= m number of edges[3]. The index of the array represents the distance from the start node to that vertex. Each vertex that is that distance away from the start is stored at that index. To handle multiple vertices with the same distance an instance of a doubly linked list is instantiated at each index.

In our implementation of this priority queue we utilized the java array list functionality. The outer array list is of constant size and is used over an array because the java compiler does not support making a fixed array of array lists. The inner array lists serve as the bucket because they support similar functionality to doubly linked lists as they allow for traversal in both directions and insertion to both the front and back of the list.

The extract min operation is performed traversing the outer array until a non empty index is found then the head of the array list at that index is removed and returned.

The DecreaseKey operation is performed on this bucket list priority queue by referencing the index where the desired vertex is located and removing it. That vertex is then reinserted into the array list at a lower index that corresponds to the decreased value. This is sufficiently faster than having to percolate nodes in a tree allowing for the tree.

Our implementation failed to integrate the bucket list priority queue, instead using a traditional heap, and thus has a substantially worse performance. The future work section further discusses optimizations of the algorithm, however our algorithm still provides proof that scaling works conceptually.

Experimental Design:

Hypothesis 1 - Performance of D-ary heap Dijkstara's algorithm is better than that of Dijkstra's with binary heap in dense graphs.

D-heap data structure allows decrease priority operations to be performed more quickly than binary heaps, at the expense of slower delete minimum operations.

This tradeoff leads to better running times for Dijkstra's algorithm in which decrease priority operations are more common than delete min operations.

Additionally, D-ary heaps have better memory cache behavior than binary heaps, allowing them to run more quickly in practice despite having a theoretically larger worst-case running time.

For undirected graphs, the Edge density (E.D.) is defined as

$$E.D. = m/n$$

Where, m - number of edges and n - number of nodes.

Edge Density is the number of edges in the graph over the maximum number of edges possible in the graph (Darlay). We considered a graph to be sparser if its E.D. is less than 0.4, denser if more than 0.8 and between 0.4 - 0.8 as an averagely denser graph.

We did following experiment:

- **Analysis of runtime with different Edge Density by varying node sizes**: We analysed the algorithm behavior for sparse and dense graphs by keeping Edge Densities 0.4 and 0.8, node size is varied from 200 to 1000 in increments of 100, keeping maximum edge weight of 25. We have considered three different graph types : random, geometric, geo-wrap. We have calculated the average number of DecreaseKey operations for three different graph types (Normalised).

From Figure 1 and 2, we can see that the number of DecreaseKey operations are more for Dijkstra's with binary heap than D-ary heap. The D-ary data structure reduces the cost of the DecreaseKey operations at the expense of added complexity and an increased constant factor for the extract-min operation.

Number of nodes	Dijkstra's with Binary heap	Dijkstra's with D-ary heap (3)	Dijkstra's with D-ary heap (5)	Dijkstra's with D-ary heap (7)
200	9423119	5777292	3132078	2122388
300	32151269	19566378	10018940	6960143

400	77607979	45352370	23478218	16068992
500	153445632	88943360	46007777	31870159
600	266312942	154671502	79369554	54701709
700	421954537	242015790	124778803	85838259
800	628857641	352161854	184927510	127499241
900	897874729	506085547	265110415	181726664
1000	1230999120	687867559	357575311	248447222

Table 1: Number of DecreaseKey operations for Dijkstra's Binary heap and D-ary heap with Edge density 0.4

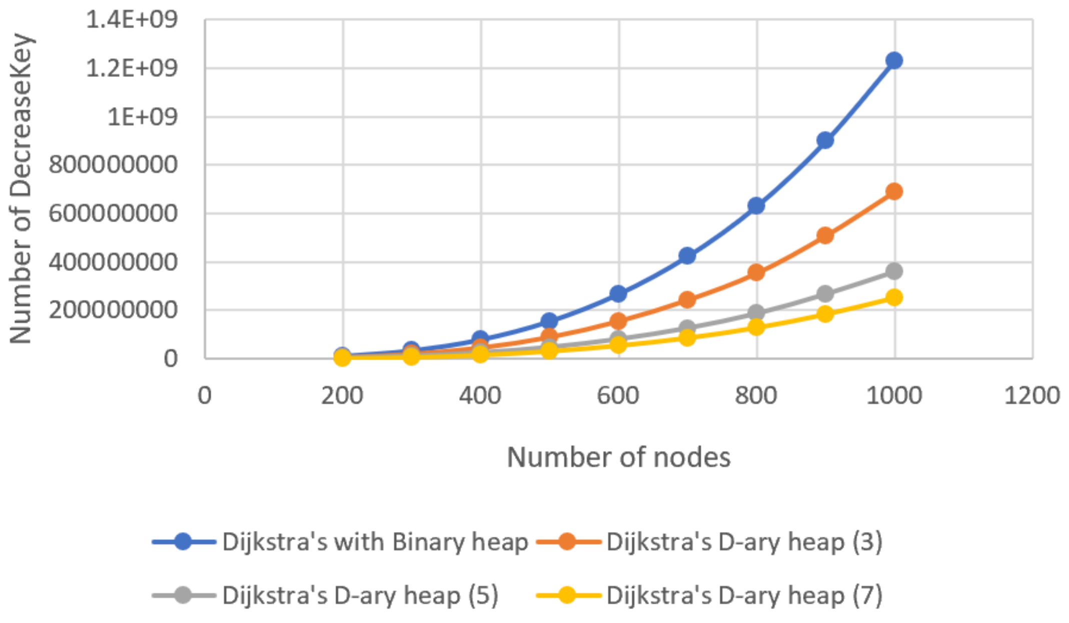


Figure 1: Number of DecreaseKey operations for varied graph sizes with Edge Density 0.4 (Sparse Graph)

Number of nodes	Dijkstra's with Binary heap	Dijkstra's with D-ary heap (3)	Dijkstra's with D-ary heap (5)	Dijkstra's with D-ary heap (7)
200	8935210	5182198	2812966	1910778
300	30076256	16853217	8923249	6184614
400	72019633	39656347	21019447	14360957
500	141222875	76349543	40914882	28414009

600	246836476	133665359	70872008	48870151
700	391554667	208846852	111180077	76731407
800	587167199	311451578	166368565	114988711
900	837887740	440653767	236786872	163039850
1000	1152716981	602789750	324402752	223902451

Table 2: Number of DecreaseKey operations for Dijkstra's Binary heap and D-ary heap with Edge density 0.8

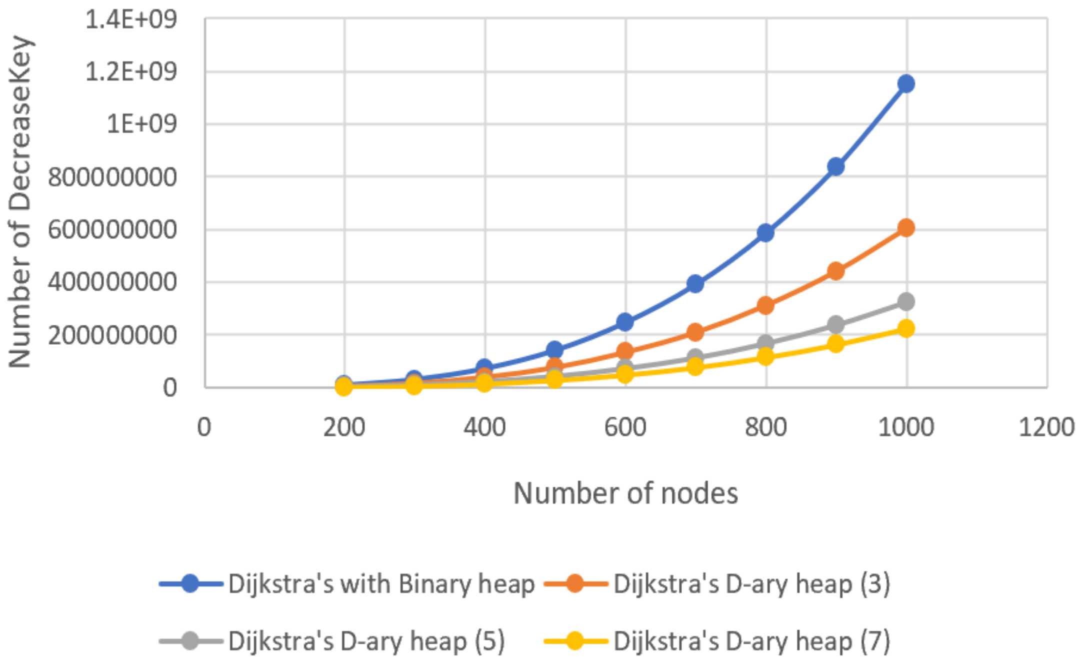


Figure 2: Number of DecreaseKey operations for varied graph sizes with Edge Density 0.8 (Dense Graph)

Number of nodes	Dijkstra's with Binary heap	Dijkstra's with D-ary heap (3)	Dijkstra's with D-ary heap (5)	Dijkstra's with D-ary heap (7)
200	0.1	0.1	0.1	0.1
300	0.2	0.233333333	0.233333333	0.233333333
400	0.433333333	0.5	0.6	0.566666667
500	0.733333333	0.833333333	1.1	0.966666667
600	1.4	1.333333333	1.9	1.666666667
700	2.1	2.233333333	2.8	2.866666667

800	3.066666667	3.066666667	4.2	4.066666667
900	4.166666667	4.366666667	5.966666667	6
1000	5.966666667	5.8	7.6	7.7
1500	20.13333333	19.2	28.16666667	25.56666667
2000	48.43333333	42.46666667	62.46666667	62.26666667
2500	95.66666667	94.1	122.4666667	129.3333333
3000	162.2333333	146.1333333	200.1333333	222.9666667
3500	258.9333333	229.0333333	359.4333333	375.7333333

Tabel 3: Normalised Runtime of Dijkstra's Binary heap and D-ary heap with Edge Density 0.4 (Sparse Graph)

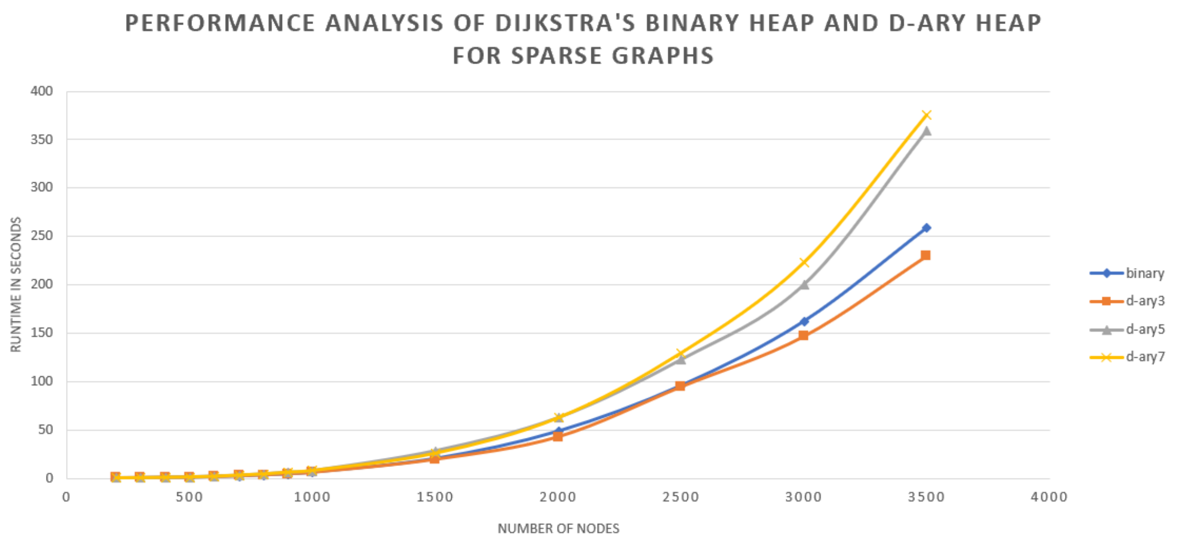


Figure 3: Normalised Runtime analysis of Dijkstra's Binary heap and D-ary heap with Edge Density 0.4 (Sparse Graph)

Number of nodes	**Dijkstra's with Binary heap**	**Dijkstra's with D-ary heap (3)**	**Dijkstra's with D-ary heap (5)**	**Dijkstra's with D-ary heap (7)**
200	0.1	0.1	0.1	0.1
300	0.233333333	0.266666667	0.233333333	0.233333333
400	0.5	0.466666667	0.533333333	0.533333333
500	0.833333333	0.866666667	0.933333333	0.933333333
600	1.4	1.466666667	1.533333333	1.433333333

700	2.166666667	2.233333333	2.333333333	2.266666667
800	3.333333333	3.233333333	3.433333333	3.4
900	4.733333333	4.633333333	4.966666667	4.8
1000	6.3	6	6.8	6.9
1500	21.53333333	21.73333333	24.06666667	24.3
2000	52.46666667	48.43333333	57.23333333	55.13333333
2500	98.36666667	94.63333333	107.4333333	110.5
3000	168.5666667	162.1333333	187.9	183.7666667
3500	287.6666667	247.8666667	331.2333333	305.8

Tabel 3: Normalised Runtime of Dijkstra's Binary heap and D-ary heap Edge Density 0.8 (Dense Graph)

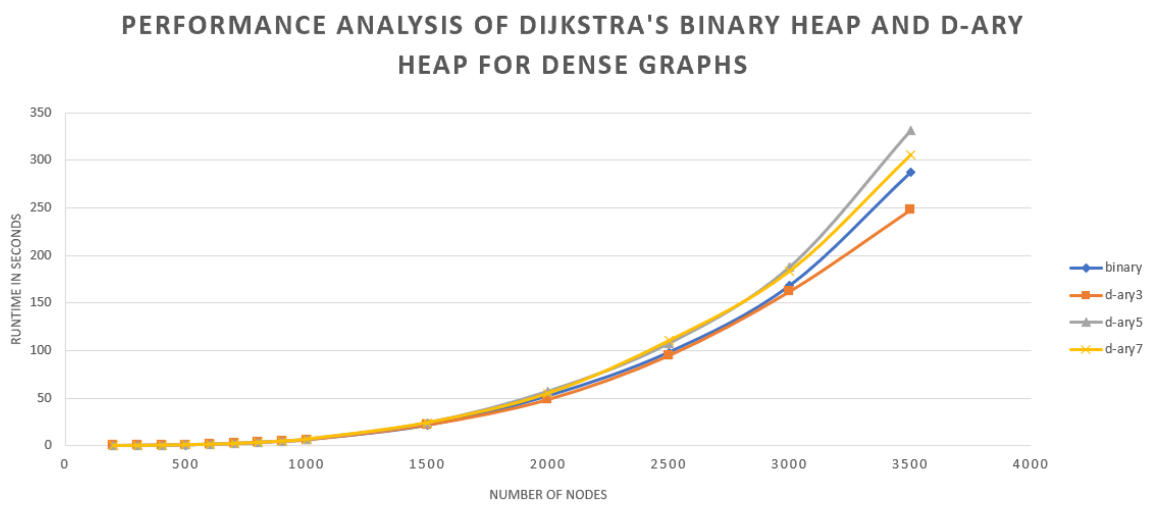


Figure 4: Normalised Runtime analysis of Dijkstra's Binary heap and D-ary heap Edge Density 0.8 (Dense Graph)

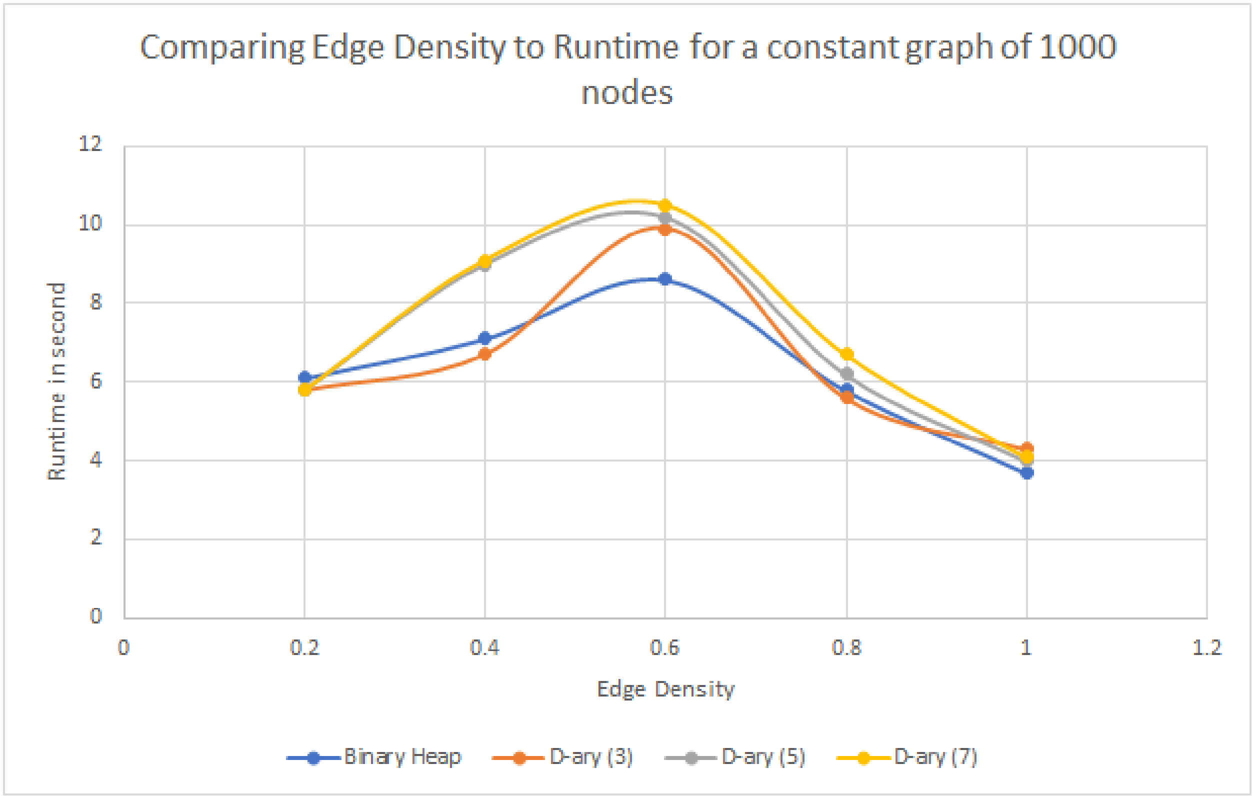
From Figure 3 and 4 we can see that D-ary implementation has a slight edge over binary data structure implementation. Slow performance of the binary heap could be attributed to binary heap supporting the optimal DecreaseKey operations.

	Insert, DecreaseKey	**deletemin**	**n * deletemin + (n+m) * insert**

binary heap	O(log n)	O(log n)	O((n+m) log n)
d-ary heap	O(logd n)	O(d logd n)	O((dn + m) logd n)

Where n - number of nodes, m - number of edges of the input graph.

From the above table we can observe Dijkstra's algorithm with D-ary performs better than binary heap for small values of d in practice.



When comparing edge density to runtime on a constant size graph it can be observed that d-ary heaps perform better with lower edge density than a min heap. The decrease in runtime for dense graphs can be attributed to the implementation using an adjacency matrix over a list. In sparse graphs there are more superfluous operations needed to iterate over edges that cause a delay in runtime.

Hypothesis 2 - Network scaling algorithm is more efficient than conventional Dijkstra's algorithm in low cost graphs (minimum weight graphs).

Since runtime of network scaling algorithms depends on the maximum edge weight of the input graph, We performed following experiments:

- **Runtime analysis of algorithms (Network scaling and Dijkstra's) with respect to varying edge weights and node sizes.**

We sweep node sizes from 600 to 3500 in increments of 100 to 100 and then 500 to 3500. and edge weights varied from N= 2 to N = 2000, keeping edge density constant (0.6).

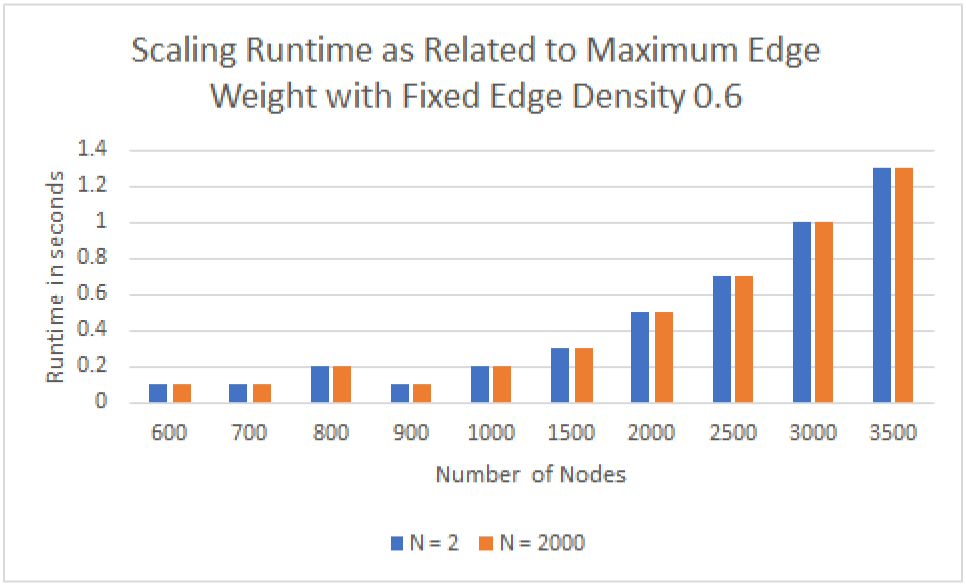


Figure 5: Runtime analysis of scaling algorithm having fixed edge density 0.6, and by varying max edge weight.

The scaling algorithm was compared against itself on identically sized graphs with the same edge density with the only change between the two graphs being the weight of the highest edge. We expected the scaling algorithm to run more efficiently for larger N values. Inspection of each experimental run showed that the overhead on our implementation vastly increased runtime and outweighed any benefit gained from the decreased number of runs. The N = 2 experiment only performed 2 iterations while the N = 2000 performed 11 iterations. The efficiency gained in the reduction of 9 runs was vastly offset by the overhead of the other priority queue.

- **Runtime analysis of algorithms (Network scaling and Dijkstra's) with respect to varying edge weights and edge densities.**

  We swept node sizes from 500 to 4000 in increments of 500 and maximum edge weight is kept constant at 25, Edge Density is varied from 0.2 to 1 in increments of 0.2, we have considered three different graph types: random, geometric and geo-wrap.

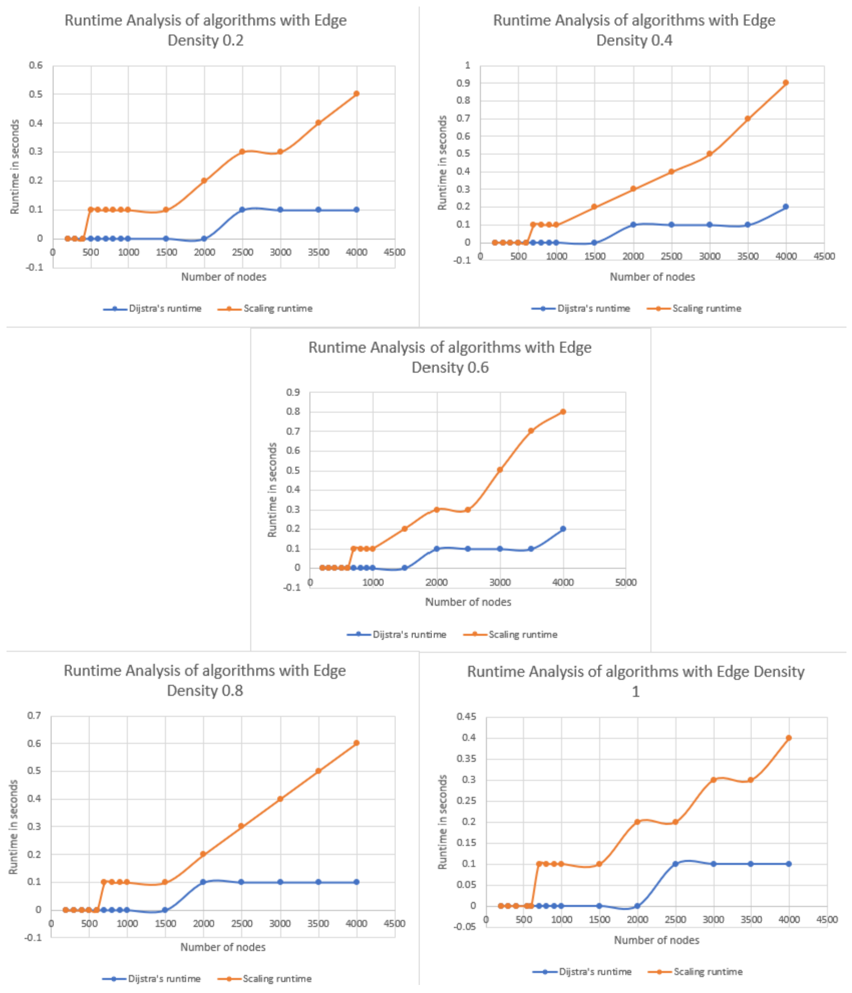


Figure 6: Runtime analysis of Dijkstra's and Scaling algorithm for graph size between 500 to 4000 for varying edge densities.

In both sparse and dense graphs we can observe that Dijkstra's runtime is better than scaling algorithm, this anomaly could be attributed to the fact that priority queue implementation for scaling was not optimal and also we have considered graph representation as adjacency matrix instead of adjacency list, this has a toll on extra operations to be performed on sparse graphs, which hurts the performance of both the algorithms.

Single Source Shortest Path Algorithm	**Runtime**
Network scaling	Theoretical Max: $O(m \log_{2+m/n} N)$[3] Implementation: $O((n \lg n) \log_{2+m/n} N)$
Dijkstra's	$O(m + n \lg n)$.

Where n - number of nodes, m - number of edges, N - max weight of the input graph.

From the above table, we expect that the Network scaling algorithm performs better in both sparse and dense graphs with large maximum edge weights. Since scaling is bound by the weight of the maximum edge in large dense graphs this algorithm should perform much better than dijkastras. Dijkastra's algorithm should perform closely with the scaling algorithm especially in sparse graphs with low edge weight.

Conclusion:

In conclusion, using a D heap as a priority queue improves the performance of Dijkastra's algorithm as long as the number of children per node stays relatively low. Our experiments showed that the using a d heap with 3 children per node led to an decrease in runtime. This was due to the reduction in operations needed to perform the decrease key operation. As the number of children grew the overhead for managing the data structure and the worsen time for extracting min caused a binary heap to outperform the d heap in our implementation. We noticed this occurred when the number of children grew to 5.

The principle of using scaling to solve a network problem was demonstrated in our implementation. The code successfully broke apart the graph and converted sub optimal solutions into the optimal one. The inefficient priority queue was shown to be the cause of the incorrect runtime of our implementation.

Future work:

Our scaling algorithm implementation has many inefficiencies that would drastically increase the runtime of the algorithm. The first and most important future upgrade is succestly integrating the priority queue. When attempting to integrate the priority queue the decrease key operation was not successfully moving the vertices in the bucket list. This causes vertices to be added twice and lead to a slow runtime of the algorithm. This fix should allow our scaling algorithm to be competitive with the implementation of dijkstra's algorithm. Another efficiency was that our algorithm should replace the adjacency matrix with an adjacency list. This replacement data structure for storing graphs would lead to a dramatic improvement in the scaling algorithm. Extra operations are introduced when needing to test for adjacency. These extra operations are compounded when each sub problem is solved.

References:

[1] Darlay, Julien. “Dense and sparse graph partition.” *Discrete Applied Mathematics*, vol. 160, no. 16-17, 2012, pp. 2389-2396. *Elsevier*, https://www.sciencedirect.com/science/article/pii/S0166218X12002399.

[2] C. H. Thomas, L. E. Charles, R. L. Ronald, and S. Clifford, Introduction to algorithms, 3rd ed. Cambridge, MA: MIT Press, 2009.

[3] H. N. Garbow, “Scaling algorithms for network problems,” *Journal of Computer and System Sciences*, vol. 31, no. 2, pp. 148–168, 1985.

[4] K. Kelley and T. B. Schardl, “Parallel Single-Source Shortest Paths,” http://courses.csail.mit.edu/, 2010. [Online]. Available: http://courses.csail.mit.edu/6.884/spring10/projects/kelleyk-neboat-paper.pdf. [Accessed: 07-May-2021].

[5] R. E. Tarjan, *Data structures and network algorithms*. Philadelphia, PA: Soc. for Industrial and Applied Mathematics, 1983.