

DATA STRUCTURES

PROJECT - 2

Authors - Mamta Shah (63721511) and Shrishti Jain (42525381)

Problem Statement:

To implement Prim's algorithm using a k-ary heap and compare its performance for different values of k with the objective of finding an optimal 'k'.

Description:

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree including every vertex, where the total weight of all the edges in the tree is minimized. The algorithm starts building the tree from an arbitrary starting vertex, at each step it adds the cheapest possible connection from the tree to another vertex so as to include all the vertices in the graph, resulting in a single connected minimum spanning tree. Prim's algorithm uses a priority queue to find an edge with minimum cost connecting two unconnected components, hence the time taken to find the solution heavily depends upon the implementation of the priority queue.

We implement a generic priority queue as k-ary heap in which we can alter the value of k as required. We compare the algorithm's performance for different values of 'k' and further deduce a relation between number of nodes, number of edges and the optimal value of 'k' of the k-ary heap. We also explore how the relative performance(time) of the implementations depend on the density of the graph and on the number of vertices.

Input File:

The input file for Prim's algorithm is an undirected weighted graph. We generate such graphs by a python script namely *GenerateGraph.py*. It takes *number of nodes(n)* and *number of edges(e)* as input parameters to generate the output graph.

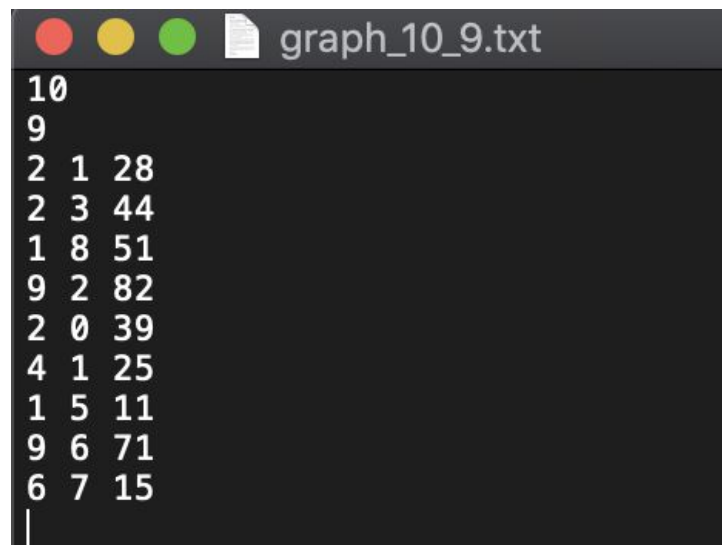
Example:

To generate a graph with 10 nodes and 9 edges we invoke the script as "**python GenerateGraph.py 10 9**". The script will automatically generate the required graph in an output file named graph_10_9.txt (graph_numberOfNodes_numberOfEdges.txt) which is as shown in figure.

In the generated input file, the first line represents the number of nodes in the graph, the second line represents the number of edges in the graph and the lines after that represent an edge generated for the graph as “node1 node2 weight(w)” which means that there is an undirected edge between node1 and node2 with a weight w.

Some specifics of the script are as follows:

1. The graph will have 0 to (n - 1) nodes when 'n' is the number of input nodes (n). (0 to 9 in case 'n' is 10)
2. The graph generates edges with a maximum weight of $n * (n - 1)$ since there can be at max nC_2 edges in a graph of 'n' nodes.
3. It ensures that the graph is connected and that all the edges are unique (as an edge from a to b is the same as an edge from b to a).



```
graph_10_9.txt
10
9
2 1 28
2 3 44
1 8 51
9 2 82
2 0 39
4 1 25
1 5 11
9 6 71
6 7 15
|
```

Output File:

For each input graph (graph_x_y.txt) two output files are generated.

1. 'MST_graph_x_y.txt' - Lists final edges of the minimum spanning tree obtained and their weights. The results obtained for each 'k' for that particular input file are listed in the output file. The value of 'k' ranges from 2 to the number of edges - 1.

```
MST_graph_10_9.txt
Minimum Spanning Tree in case of 2-ary heap:
0 <- 0 = 0
1 <- 2 = 28
2 <- 0 = 39
3 <- 2 = 44
4 <- 1 = 25
5 <- 1 = 11
6 <- 9 = 71
7 <- 6 = 15
8 <- 1 = 51
9 <- 2 = 82

Minimum Spanning Tree in case of 3-ary heap:
0 <- 0 = 0
1 <- 2 = 28
2 <- 0 = 39
3 <- 2 = 44
4 <- 1 = 25
5 <- 1 = 11
6 <- 9 = 71
7 <- 6 = 15
8 <- 1 = 51
9 <- 2 = 82
```

2. 'timeTaken_graph_x_y.txt' - It stores the execution time in milliseconds for every value of k (i.e. from 2 to the (number of edges - 1)) for that particular input file.

```
timeTaken_graph_10_9.txt
1. For 2-ary heap: 250 microseconds
2. For 3-ary heap: 220 microseconds
3. For 4-ary heap: 210 microseconds
4. For 5-ary heap: 245 microseconds
5. For 6-ary heap: 206 microseconds
6. For 7-ary heap: 202 microseconds
7. For 8-ary heap: 369 microseconds
8. For 9-ary heap: 488 microseconds
```

Later in the experimental analysis section, we plot various time vs k graphs using a python script named *plotGraphK-T.py*, it takes the above timeTaken file as input and generates plots to visualize this output file. The script is invoked as "python plotGraphK-T.py timetaken_graph_10_9.txt".

Design Choices:

1. For implementation of Prim's algorithm

The input graph file is read and number of edges and nodes obtained are used to initialize the matrix representation of the undirected input graph. It is stored in the form of vector of vectors of int called graph. Here $\text{graph}[i][j]$ stores the weight of the edge between node 'i' and node 'j' and has a value of infinity in case of absence of an edge between them. We use clock to record the system time for each iteration of k and then prim's algorithm is called to begin its execution.

We use three arrays for following functions-

- fWeight - Stores the minimum weight to reach each vertex. It is initially initialized to infinity for all vertices except source whose fWeight is zero.
- parent - Stores the parent of each vertex, so $\text{parent}[i] = x$ means we can reach vertex 'i' by visiting vertex 'x' and then following the x-i edge.
- visited - It is a boolean array which is initialized as false for all vertices to indicate that initially we have no path to reach any vertex. Whenever we remove an edge from the priority queue if this edge connects us to an unvisited node then we mark it as visited by setting $\text{visited}[i] = \text{true}$. This helps avoid redundant addition of its neighbours into the queue.

Now we begin by adding all the edges from node 0 to its neighbours into the heap. Till we have not obtained $(n - 1)$ edges, we repeatedly remove the edge with the minimum weight from the heap and check if it connects us to an unconnected vertex:

- If it does, we update the corresponding values in parent and fWeight arrays, mark the vertex as visited, add all the edges into the heap which connect the current vertex to an unconnected vertex and then finally update the count of edges we have obtained so far.
- However, if the answer is no then it implies that we haven't got any new edge and hence we iterate again without updating the count of edges.

The complexity of this algorithm is $O(E \log_k E + V \log_k E)$

E is the number of edges

V is the number of nodes in the input graph.

2. For implementation of k-ary heap

We have implemented k-ary heap as a dynamic array using vector where we store the k children of an item at index 'i', i.e. children of heap[i] at locations from heap[i*k+1] to heap[i*k + k]. Similarly, the parent of an item at index 'i' i.e. of heap[i] is stored at heap[floor(i-1)/k].

Using this we implemented the following functions which are then used by prim's algorithm:

- **insertIntoHeap**: This function takes two arguments, one is the value of 'k' and the other is an instance of struct node which is to be stored. The struct node instance has 3 fields in it namely, node1 and node 2 between which this edge is present and a weight field to store the weight of this edge. We maintain a min heap and hence during the insert operation we copy the input instance to the last spot in the heap and then call siftUp function which is private to the class heap. siftUp shifts the item up to its appropriate place based on the weight field of the node and using the value of 'k' passed to it by calculating the index of the parent node using the formula $\text{floor}((i-1)/k)$. If the value of the weight field of the newly inserted item is less than its parent then the items are swapped and the new index is set to be equal to the parent index until it is positive. Else we do nothing.
- **getMin**: It returns the root of the heap (i.e. an instance of the node with minimum value of weight field). It then copies the last item to the root and restores the heap property by internally calling siftDown. siftDown method calculates the indices of the children node by using formulas - $i*k+1$ to $i*k+k$. It then finds the minimum of these k nodes and checks whether the obtained value is less than the root. If yes, it swaps the two and recurs at the minimum child node index.
- **showHeap**: It is a method used to see the contents of the heap at any point in time. It was mainly helpful for debugging purposes.
- **size**: It gives the number of elements currently present in the heap.

3. For experimental analysis

We will be varying the following parameters:

- Number of nodes (n): To check for scalability and for having a varied level of connectivity in the graph we used input graphs with number of nodes(n) varying from 10 to 120. Values we taken are 10, 20, 30, 50, 80 and 120.
- Number of edges (m): To check for scalability and for having a varied level of connectivity in the graph such as sparse, medium and heavily connected (described in the next paragraph), we used input graphs with number of edges(e) as follows:
 - For sparsely connected graphs we used the following (n, e) pairs: (10, 9), (20, 28), (30, 152), (50, 301), (80, 899), (120, 2202).
 - For medium connected graphs we used the following (n, e) pairs: (10, 21), (20, 85), (30, 200), (50, 645), (80, 1545), (120, 4577).
 - For heavily connected graphs we used the following (n, e) pairs: (10, 33), (20, 150), (30, 368), (50, 1114), (80, 2677), (120, 6913).

Varied Connectivity in graphs:

We have categorized the graphs as sparse, medium and heavily connected based on the density ratio(d) i.e. number of edges / number of nodes. The higher the value of 'd' more densely connected will be the graph. Now to divide the value of d into 3 groups we use the fact that the maximum number of edges(e) possible for a graph with n nodes is nC_2 and so by dividing the number of edges in 3 groups we divide the density in the 3 groups as well. So we have $({}^nC_2 / 3) = ((n^2 - 3n - 2) / 6) = x$ as the size of each group of edges and hence for a graph with n nodes the type of graph we obtain depends on the choice of the number of edges.

1. Number of edges between $[0, x]$ -> sparse graph
 2. Number of edges between $[x+1, 2x+1]$ -> medium connected graph
 3. Number of edges between $[2x+2, n-1]$ -> heavily connected graph
- 'K' of the k-ary heap used (k): For each of the input graphs we run Prim's algorithm for k ranging from 2 to n-1. We stop at n-1, since with $k = n-1$ there is only one level in the tree which is the minimum number of levels possible. Increasing k further does not change the representation of graph in anyway and thus will not impact the performance of the algorithm.
 - So for each of the (n, e) pairs mentioned above we execute the Prim's algorithm for all above mentioned values of k and note the execution time for each (n, e, k) triplet.

Experimental Analysis and Observations:

We have deduced multiple relations (as mentioned in following sections) on the basis of the output graphs obtained. Relevant figures have been attached wherever necessary. Here 'n' and 'e' represent the number of nodes and number of edges respectively.

1. Relation of Time with K:

Figure1 is a graph of time taken (in milliseconds) vs value of k ranging from 2 to 19 plotted for graph with n = 20 and e = 28.

It can be observed that as the value of k increases the time taken decreases till a certain point i.e. k = 6 in this case where we get our minimum optimal k. After this the value either remains the same or an occasional spike is noticed in the variable - time taken. The probable reason for these spikes is that the getMin operation becomes costlier because of the siftDown function (which it depends on) takes relatively more time to find the minimum out of the 'k' children of a parent. However, again the time reduces after a spike in time at k = 9 because the cost of insertIntoHeap decreases as siftUp function becomes cheaper because of the decrease in depth of the tree.

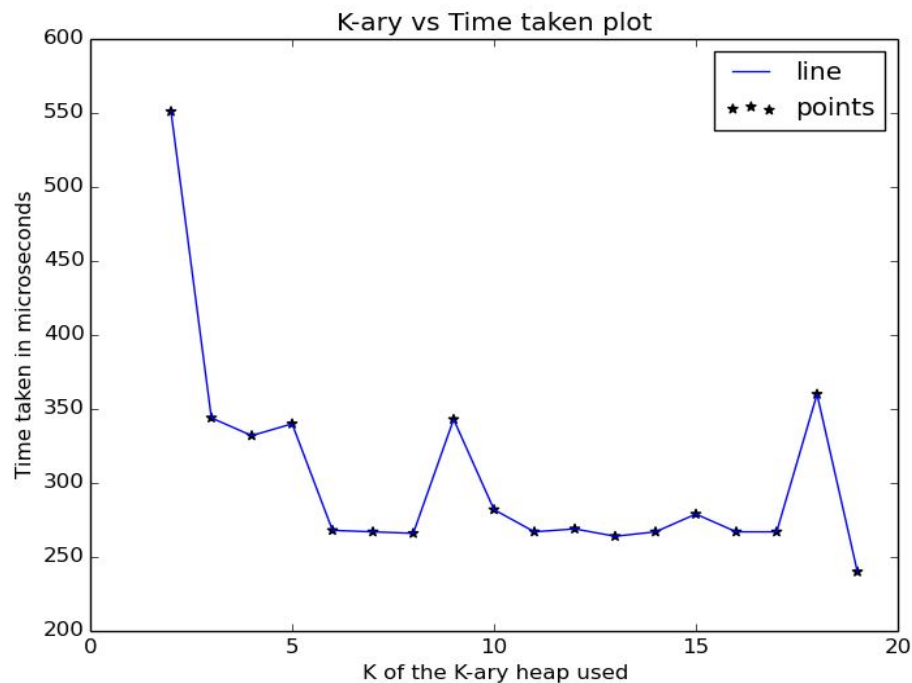


Figure 1: Plot of graph_20_28.txt

2. Relation between number of edges(e) and value of K:

The number of nodes(n) is kept constant for this observation. Analysis is done for two values of n which are 10 and 20. Figure 2, 3, and 4 correspond to n = 10 with 'e' being 9, 21 and 33 respectively. Figure 5, 6 and 7 correspond to n = 20 for which e is 28, 85 and 120 respectively.

- For plots in figure 2, 3 and 4, the minimum optimal value of k observed is 3, 4 and 5 respectively.
- For graphs in figure 5, 6 and 7, we observe the minimum optimal value of k to be 6, 8 and 8 respectively.
- Based on the above observations we can conclude that as the number of edges in a graph increases the minimal optimal value of k also increases.

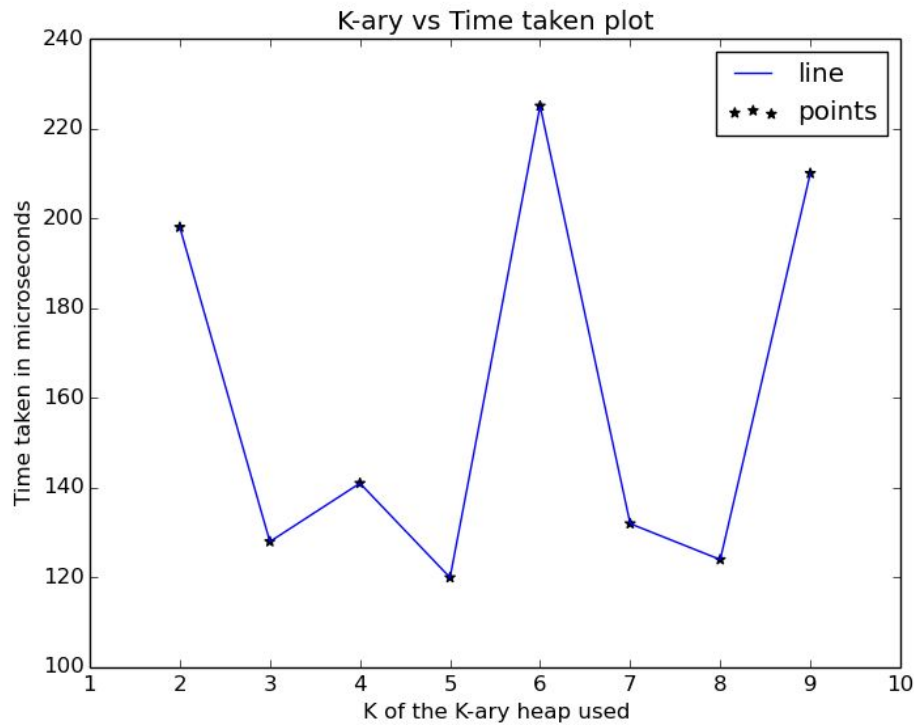


Figure 2: Plot of graph_10_9.txt

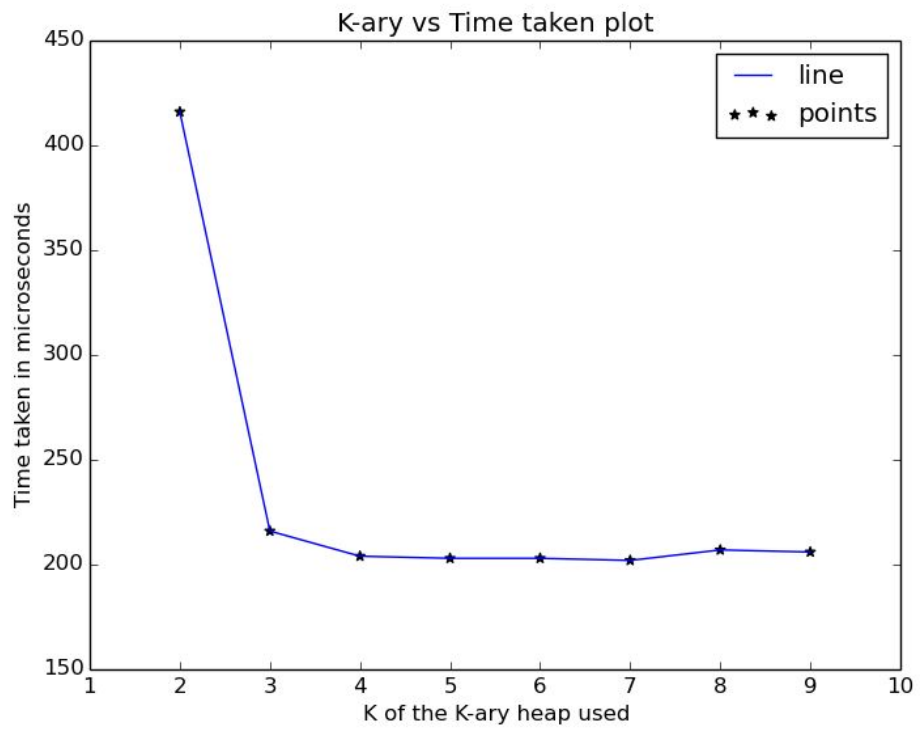


Figure 3: Plot of graph_10_21.txt

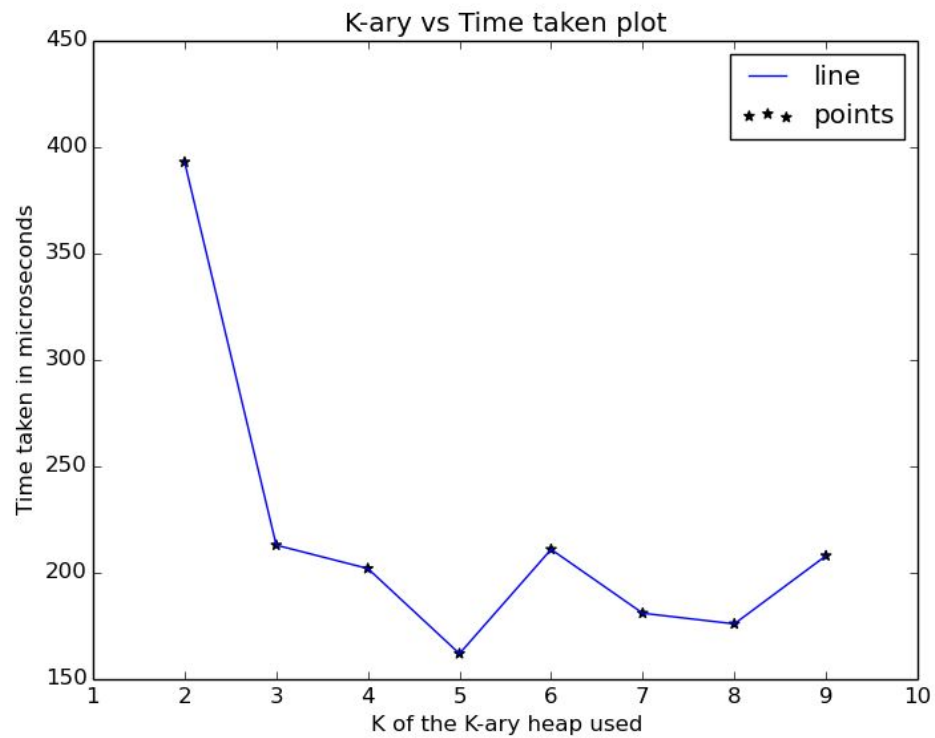


Figure 4: Plot of graph_10_33.txt

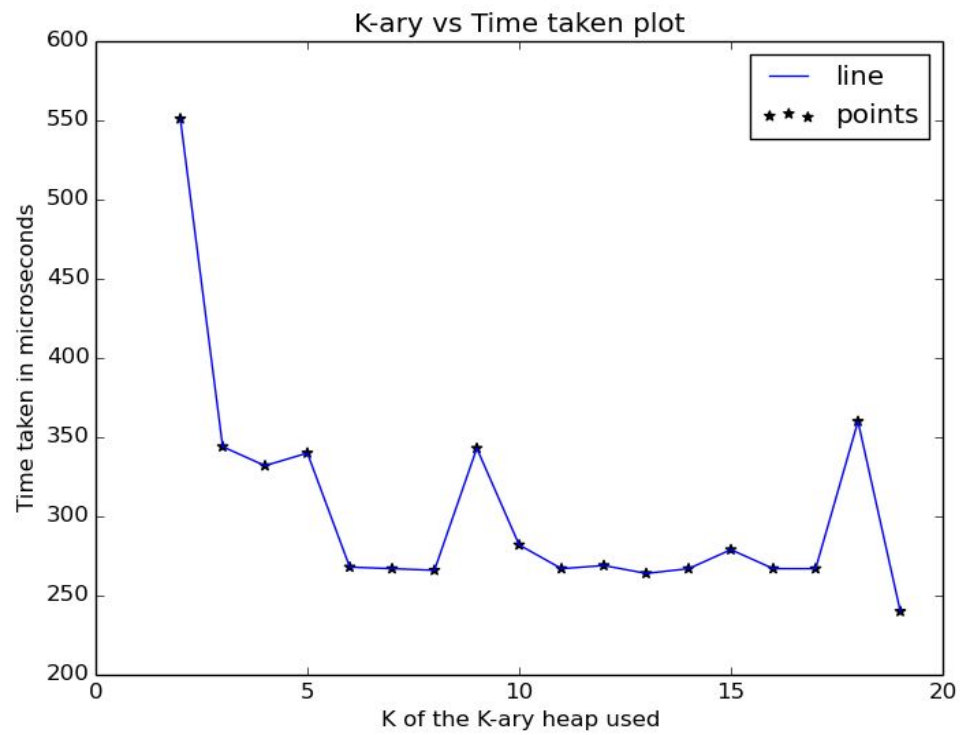


Figure 5: Plot of graph_20_28.txt

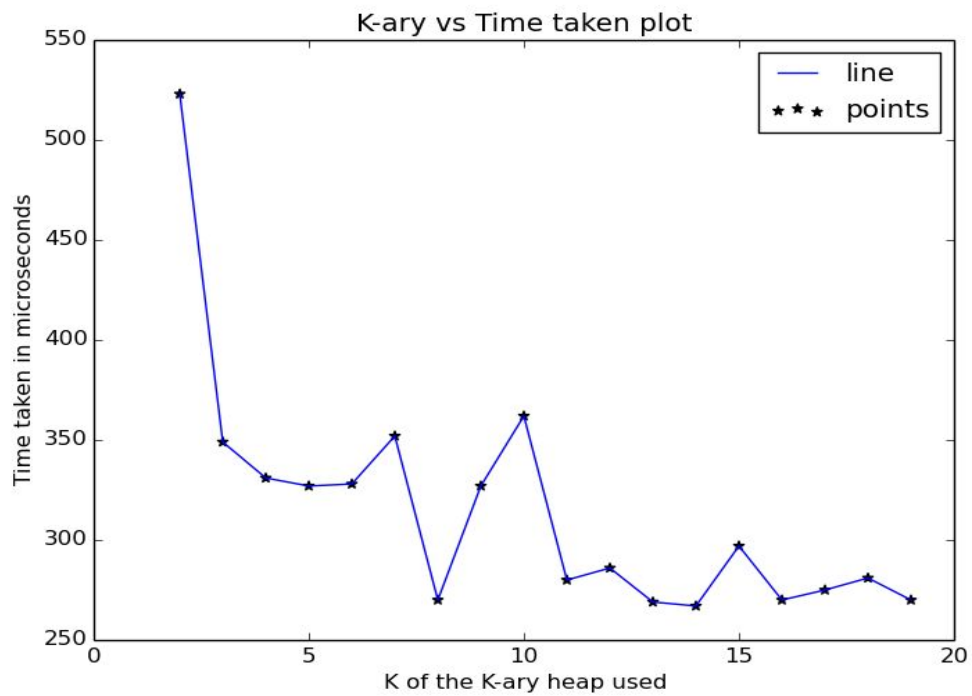


Figure 6: Plot of graph_20_85.txt

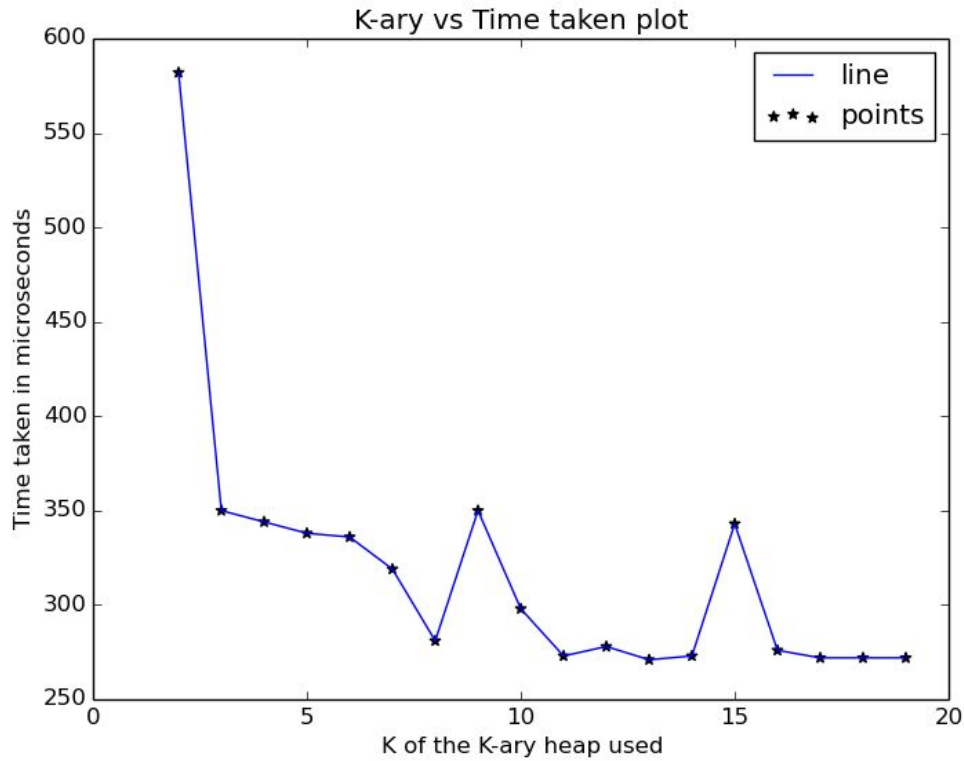


Figure 7: Plot of graph_20_150.txt

3. Relation of density vs optimal K:

For this part of analysis we keep the density (i.e. ratio of n/e) of the graph constant i.e. we choose three graphs which all have the same level of connectivity eg. are all densely connected. The plots chosen are: ($n = 10$, $e = 33$), ($n = 20$, $e = 120$), ($n = 30$, $e = 368$) i.e. figures 8, 9 and 10 respectively. We note that the minimal optimal value of 'k' for these graphs are 5, 8 and 16 respectively, which implies that it is better to choose a higher value of k for graphs which are more dense. This is mainly because of the fact that the number of edges also increases as the density of graph increases.

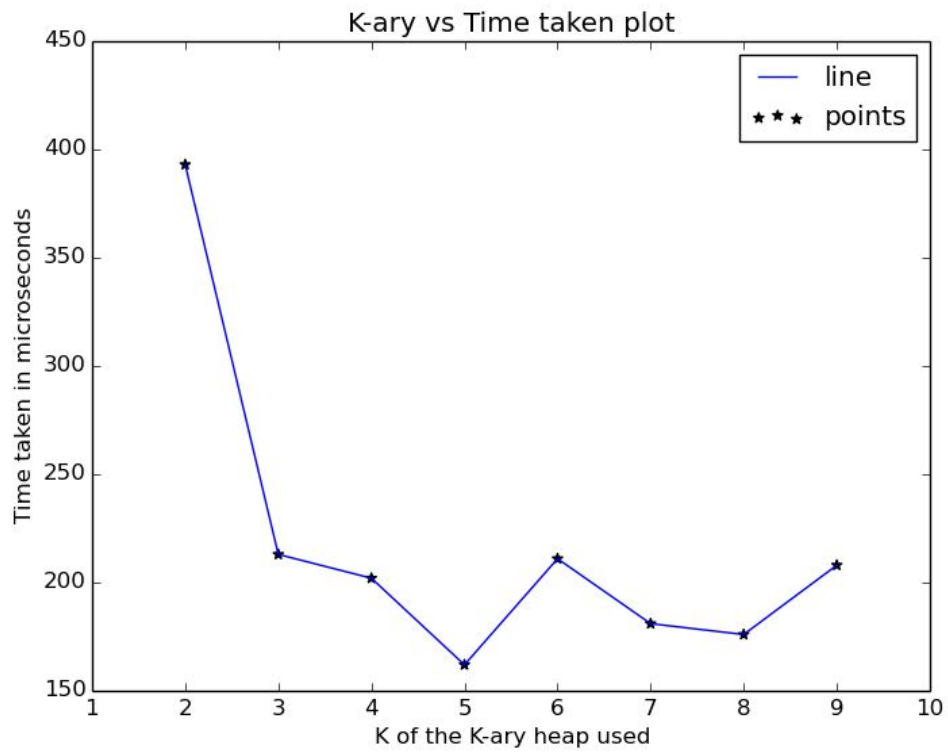


Figure 8: Plot of graph_10_33.txt

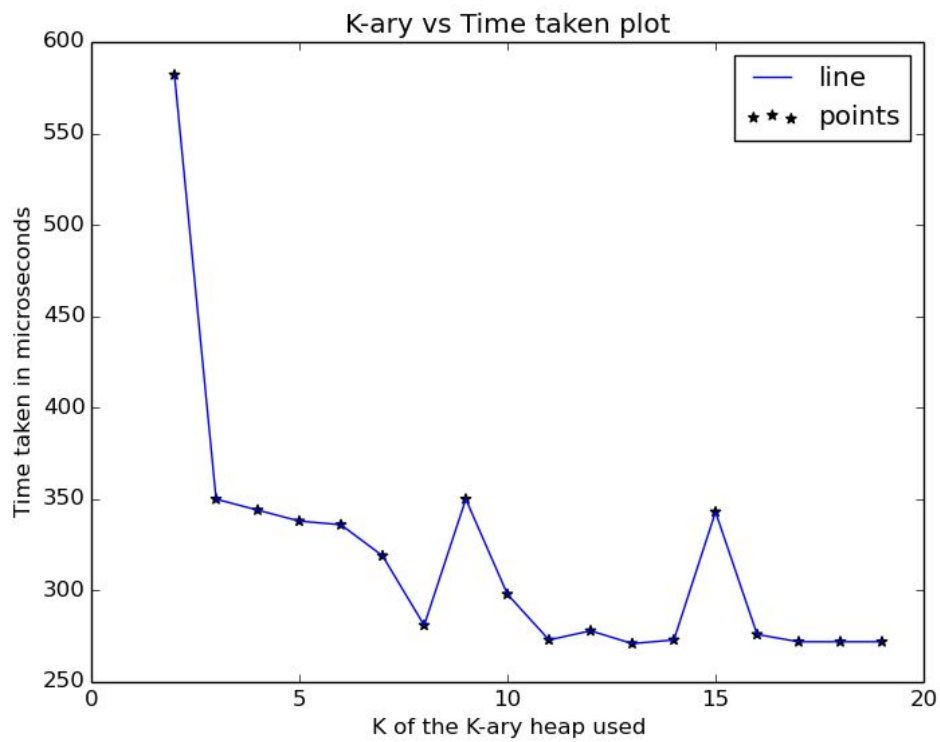


Figure 9: Plot of graph_20_120.txt

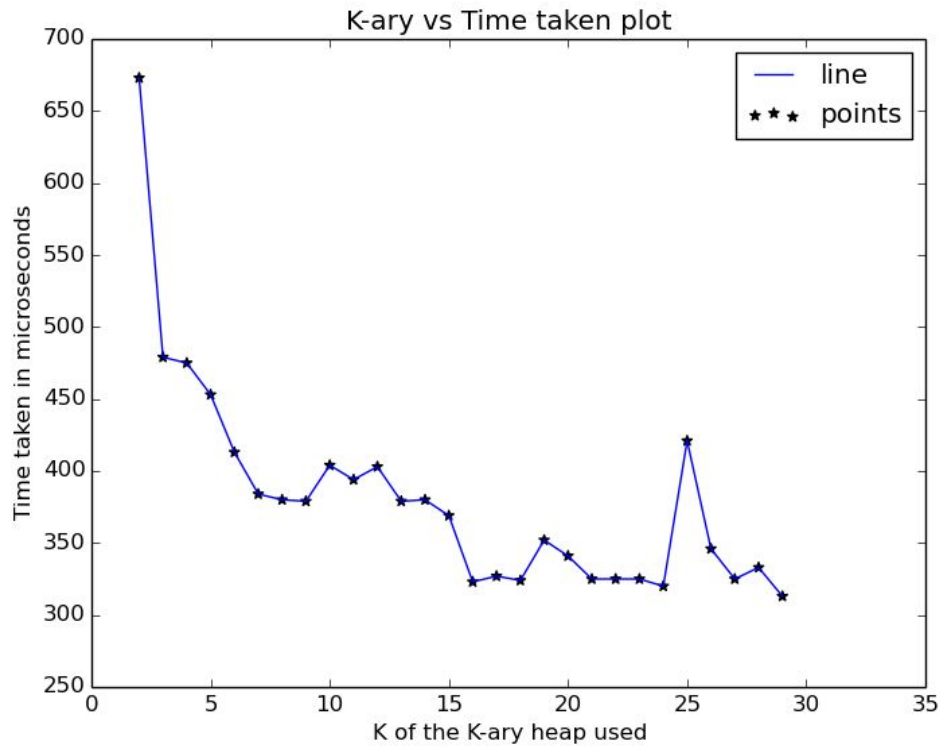


Figure 10: Plot of graph_30_368.txt

4. Optimal value of K when graphs are dense:

The run time complexity of Prim's algorithm highly depends upon the running time on insertIntoHeap operation(which is executed e times) and getMin operation(which is executed n times) of the k -ary heap. The operation insertIntoHeap becomes cheaper and getMin gets costlier as the value of k increases. So our aim is to find the optimal value of k such that both these operations compensate each others cost.

We noticed that for $k = \text{number of edges} / \text{number of vertices}$ these two operations balance out each other and hence the minimal optimal value of k will be obtained for which our algorithm's performance is at its best.

However, we also observed that while such relation is not true in cases where the graphs are sparse or medium connected, it does hold for densely connected graphs.

For eg: It can be seen in figures 11 and 12 (where $n = 80$, $e = 2677$ and $n = 120$, $e = 6913$) that the ratio of e/n is approximately 33 and 57 which is also the observed minimum optimal value of k in their respective plots.

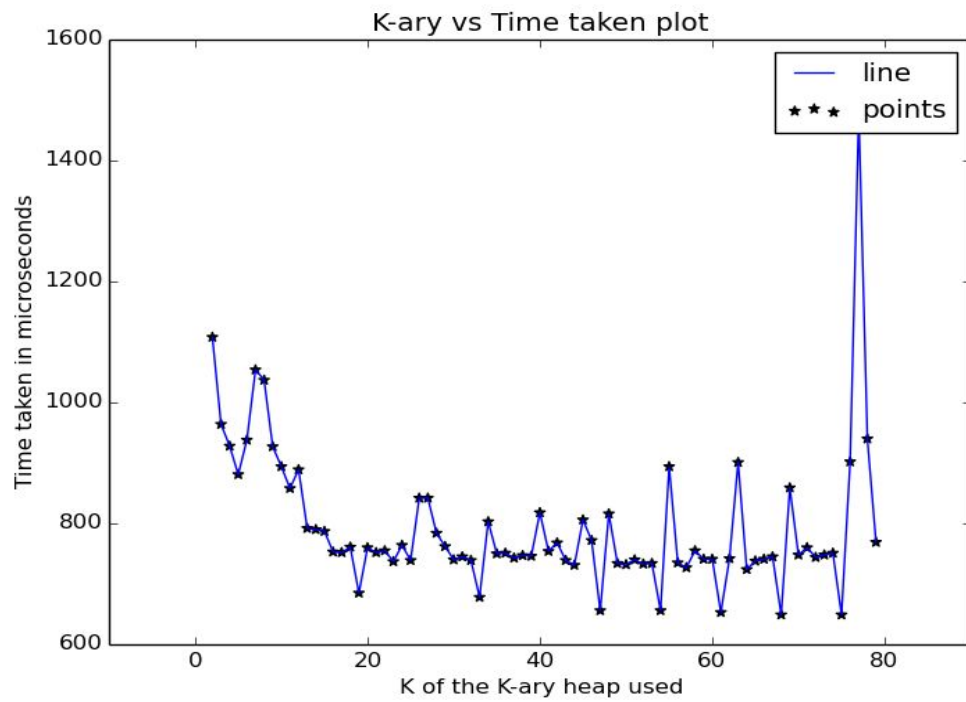


Figure 11: Plot of graph_80_2677.txt

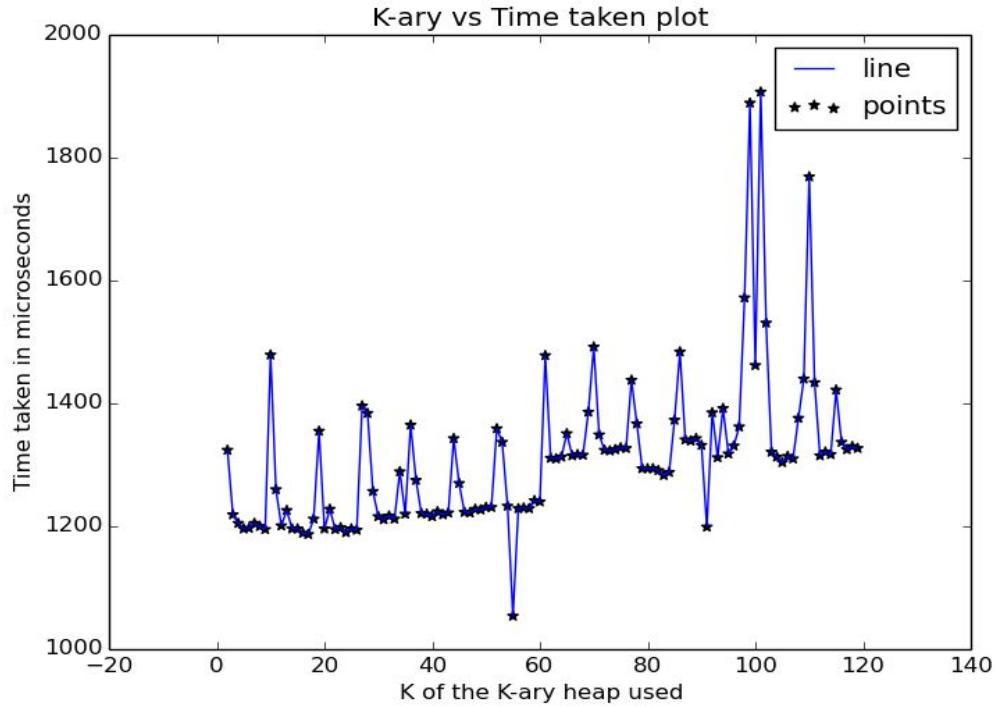


Figure 12: Plot of graph_120_6913.txt

Observations/Conclusion:

1. Relation between value of k and Time taken

As the value of k increases the time taken by the algorithm decreases until a certain value which is often the value of minimum optimal k. After this, time more or less hovers around the same value with occasional spikes which can be attributed to the varying complexities of getMin and insertIntoheap function of the implemented k-ary heap.

2. Relation between number of edges and value of k

It was observed that as the number of edges in a graph increases the minimum optimal value of k also increases.

3. Relation between density of graph and optimal k

As the density level of the input graph was increased the minimum optimal value of k obtained also increased. Thus it can be concluded that k is directly proportional to the density of the graph.

4. Relation between value of k and n & e

In case of dense graphs following relation was deduced:

$$K = \text{Number of edges} / \text{Number of vertices}$$

K - Minimum optimal value of k for the densely connected input graph

Openlab Run:

1. The system on which the code is tested is as follows:

You have logged into circinus-42.ics.uci.edu

Operating System: CentOS Linux release 7.6.1810 (Core)

Architecture: x86_64

RAM: 3.76 GiB/94.24 GiB

Uptime: 65 days

2. The code is compiled as follows:

```
mamtajs@circinus-42 01:18:44 ~/DS Project2
$ ls
.DS_Store graph_10_21.txt primsK-ARY.cpp
mamtajs@circinus-42 01:18:45 ~/DS Project2
$ g++ -std=c++11 primsK-ARY.cpp -o prims
mamtajs@circinus-42 01:19:02 ~/DS Project2
$ ls
.DS_Store graph_10_21.txt prims* primsK-ARY.cpp
```

3. The output of the code is tested on file graph_10_21.txt and output was generated as expected as shown below:

```
mamtajs@circinus-42 01:19:09 ~/DS Project2
$ ./prims graph_10_21.txt
mamtajs@circinus-42 01:19:18 ~/DS Project2
$ ls
.DS_Store  graph_10_21.txt  MST_graph_10_21.txt  prims*  primsK-ARY.cpp  timeTaken_graph_10_21.txt
mamtajs@circinus-42 01:19:21 ~/DS Project2
$ vim MST_graph_10_21.txt
```

Minimum Spanning Tree in case of 2-ary heap:

```
0 <- 0 = 0
1 <- 5 = 4
2 <- 0 = 57
3 <- 5 = 31
4 <- 1 = 36
5 <- 9 = 1
6 <- 2 = 1
7 <- 9 = 47
8 <- 9 = 31
9 <- 6 = 11
```

Minimum Spanning Tree in case of 3-ary heap:

```
0 <- 0 = 0
1 <- 5 = 4
2 <- 0 = 57
3 <- 5 = 31
4 <- 1 = 36
5 <- 9 = 1
6 <- 2 = 1
7 <- 9 = 47
8 <- 9 = 31
9 <- 6 = 11
```