PERFORMANCE COMPARISON OF PRIM'S ALGORITHM USING DIFFERENT PRIORITY QUEUES

A MINI-PROJECT REPORT

of

BACHELOR OF TECHNOLOGY

in

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

at

DAYANANDA SAGAR UNIVERSITY

SCHOOL OF ENGINEERING, BENGALURU - 560068

4TH SEMESTER

(Course Code: 16CS274)

ANUSHA J - ENG18CS0041

ANUSHA R - ENG18CS0043

ARUN KUMAR - ENG18CS0047

BHAVANA G - ENG18CS0059

DAYANANDA SAGAR UNIVERSITY

CERTIFICATE

This is to certify that the Algorithms Design and Analysis mini-project report entitled **"Performance Comparison of Prim's Algorithm using Different Priority Queues"** is being submitted by Anusha J, Anusha R, Arun Kumar, and Bhavana G to Department of Computer Science and Engineering, School of Engineering, Dayananda Sagar University, Bangalore, for the 4th semester B.Tech C.S.E of this university during the academic year 2019-2020.

Date:_____ _____

Signature of the Faculty in Charge

_____

Signature of the Chairman

ACKNOWLEDGEMENT

We are pleased to acknowledge **Bindu Madavi KP, Assistant Professor,** Department of Computer Science & Engineering for her invaluable guidance, support, motivation and patience during the course of this mini-project work.

We extend our sincere thanks to **Dr. MK Banga**, **Chairman,** Department of Computer Science & Engineering who continuously helped us throughout the project and without his guidance, this project would have been an uphill task.

We have received a great deal of guidance and co-operation from our friends and we wish to thank one and all that have directly or indirectly helped us in the successful completion of this mini-project work.

Team Members

ANUSHA J - ENG18CS0041

ANUSHA R - ENG18CS0043

ARUN KUMAR - ENG18CS0047

BHAVANA G - ENG18CS0059

ABSTRACT

Prim's algorithm is one of a few algorithms to find the minimum spanning tree of an undirected graph. The algorithm uses a priority queue, and much of its time complexity depends on the priority queue used. This project compares the performances of Prim's algorithm using different priority queues. These are (i) a list which is traversed to find the minimum whenever the ExtractMin operation is performed, (ii) a binary heap, and (iii) a Fibonacci heap.

All of these priority queues' operations have differing time complexities. Theoretically, the Fibonacci heap is superior to other less sophisticated priority queues because its DecreaseKey runs in $\Theta(1)$ amortized time. But due to the complexity in its structure, it is known not to run fast practically for small input sizes. When we compared our implementations, the Fibonacci heap and the binary heap took more or less the same running time for input sizes of above 2000 vertices.

TABLE OF CONTENTS

INTRODUCTION

PROBLEM STATEMENT

To implement Prim's minimum spanning tree algorithm using

1. Lazy and Eager Naive Approaches
2. Binary Heap as Priority Queue
3. Fibonacci Heap as Priority Queue

and compare their performances on large inputs.

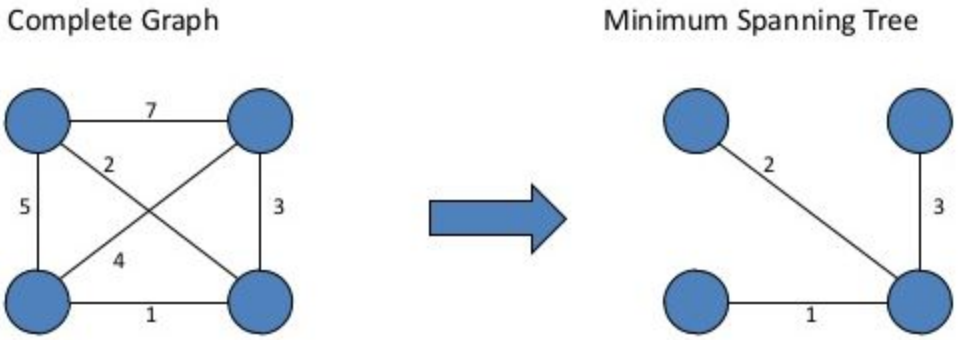
The lazy and eager naive approaches will use the adjacency matrix representation for the graph whereas the other approaches will use the adjacency lists representation.

Our project will only be focusing on connected graphs.

OBJECTIVES

- We will analyse the performances of the different implementations on many undirected graphs of differing edge densities.
- We know that theoretically the Fibonacci heap implementation has the best time complexity. We will test this empirically.
- We will create an interface for a user to run our implementations on a given graph and to visualize the minimum spanning tree produced by Prim's algorithm.

Minimum Spanning Tree: A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.



Prim's Algorithm: Prim's algorithm is an algorithm to construct an MST from the given graph. It was developed by Jarnik in 1930, and was later rediscovered and republished by Prim in 1957 and Dijkstra in 1959.

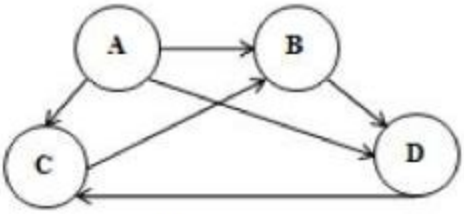
1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.

2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.

3. Repeat step 2 until all vertices are in the tree.

Representations of Graph: The two most common ways of representing graphs are adjacency matrix and adjacency list. Suppose the n vertices of a graph are numbered from 0 to n - 1. We take every edge to be of the form (i, j); the edge originates at vertex i and terminates at vertex j. An adjacency matrix is an n*n matrix in which the value of [i, j]th cell is the weight of the edge if there is an edge

or 0 if there isn't. An adjacency list is an array of separate lists. The array has length n and the ith element of the array is a list of all those vertices which are directly connected to the ith vertex.



Graph

	A	B	C	D
A	0	1	1	1
B	0	0	0	1
C	0	1	0	0
D	0	0	1	0

Adjacency Matrix

Node	Adjacent Node(s)
A	B C D
B	D
C	B
D	C

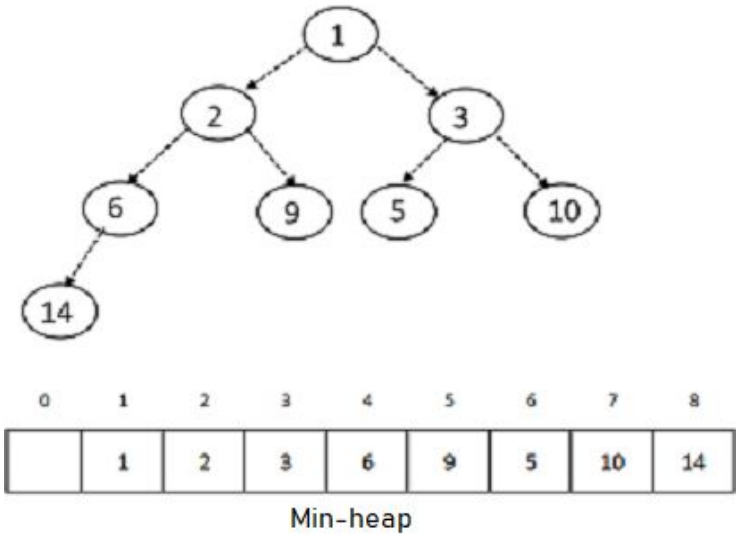
Adjacency Lists

Binary Heap: A binary heap is a complete binary tree where the key stored in each node is either greater than or equal to (for a max-heap) or less than or equal to (for a min-heap) the keys in the node's children. It was introduced by John Williams in 1964 as a data structure for heapsort. We require a min-heap for Prim's algorithm.

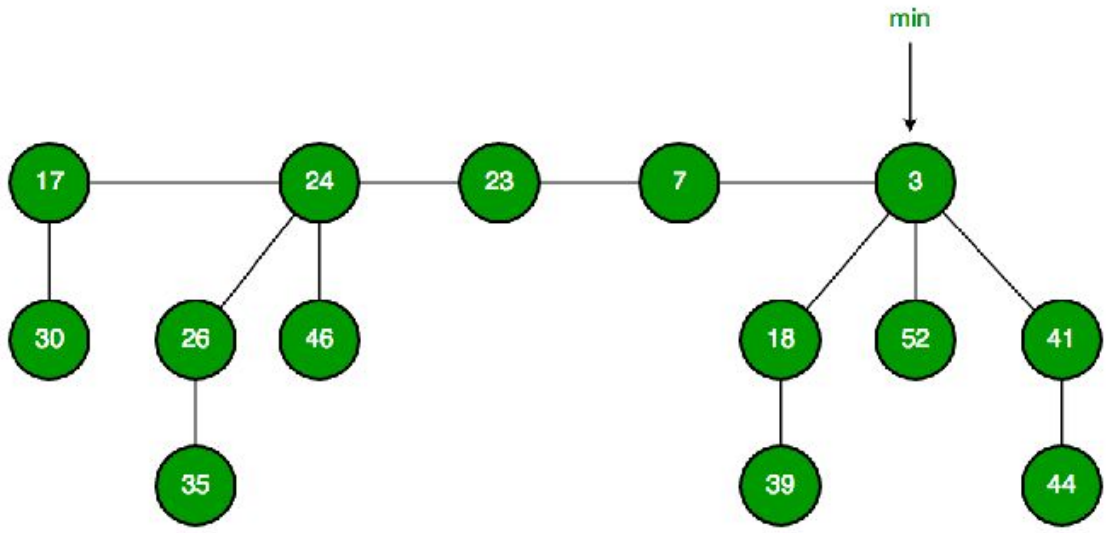
The heap will be implemented using an array rather than self-referential pointers.

The common operations performed on a min-heap are insert, find_min, extract_min and decrease_key. Both insert and extract_min modify the heap to conform to the shape property (heap being a complete binary tree) first, by adding or removing from the end of the heap. Then the heap property is restored by traversing up or down the heap. Both operations take O(log n) time, where n is the number of nodes. We will only use the extract_min and decrease_key operations in Prim's algorithm.

The extract_min operation removes the smallest element i.e. the root from the heap. This is done by replacing the root with the last element on the last level and then repeated swapping of keys until the heap property is satisfied. The decrease_key operation is used to change the key value of a given node to a new smaller value. Since the new value is smaller, key exchanges are upward. This operation also takes O(log n), but that is a less strict upper bound.



Fibonacci Heap: Invented by Fredman and Tarjan in 1984, a Fibonacci heap is a heap data structure similar to the binomial heap, only with a few modifications and a looser structure. The Fibonacci heap was designed in order to improve Dijkstra's shortest path algorithm. Its name derives from the fact that the Fibonacci sequence is used in the complexity analysis of its operations. It is a collection of heap-ordered trees, each tree having an order that is based on the number of children (i.e. a tree having a single node has degree 0, a tree having one root and two children has degree 2). The trees can have any shape (trees can even be single nodes).



Fibonacci Heap

All tree roots are connected using a circular doubly linked list. A pointer to the minimum root (which is the minimum value) is maintained. The Fib heap executes operations the lazy way. “Do work only when you must, and then use it to simplify the structure as much as possible so that your future work is easy.” Insert operation simply adds a new tree with a single node and merge/meld operation simply links two heaps. ExtractMin does the most work -- removing the minimum from the list of roots, making its children roots, and then consolidates all trees with the same order. For DecreaseKey, if the node that has been decreased causes a heap violation, then it is cut from its parent and added to the root list. If any node has lost one child, it is marked, and if it loses another child, it is cut from its parent and added to the root list.

Operation	find_min	extract_min	insert	decrease_key
Binary heap	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Fibonacci heap	$\Theta(1)$	$O(\log n)$*	$\Theta(1)$	$\Theta(1)$*

*amortized time

Minimum edge weight data structure	**Time complexity (total)**
adjacency matrix, searching	$O(\|V\|^2)$
binary heap and adjacency list	$O((\|V\| + \|E\|) \log \|V\|) = O(\|E\| \log \|V\|)$
Fibonacci heap and adjacency list	$O(\|E\| + \|V\| \log \|V\|)$

SYSTEM REQUIREMENTS

FUNCTIONAL REQUIREMENTS

In this project, we have programmed data structures of binary heap and Fibonacci heap priority queues and four implementations of Prim's algorithm.

For demonstration: We have a user interface, where the user enters the file name of the test case and the names of the implementations that they want to run it through. The program runs it through these implementations and displays the time taken for each. The program also displays the MST produced by Prim's algorithm. This graph visualization is done by one of the modules using the networkX library.

For analysis: We have a module that generates test cases and then puts them through the implementations. The durations are gathered into CSV files and are used to plot performance comparison plots.

HARDWARE REQUIREMENTS

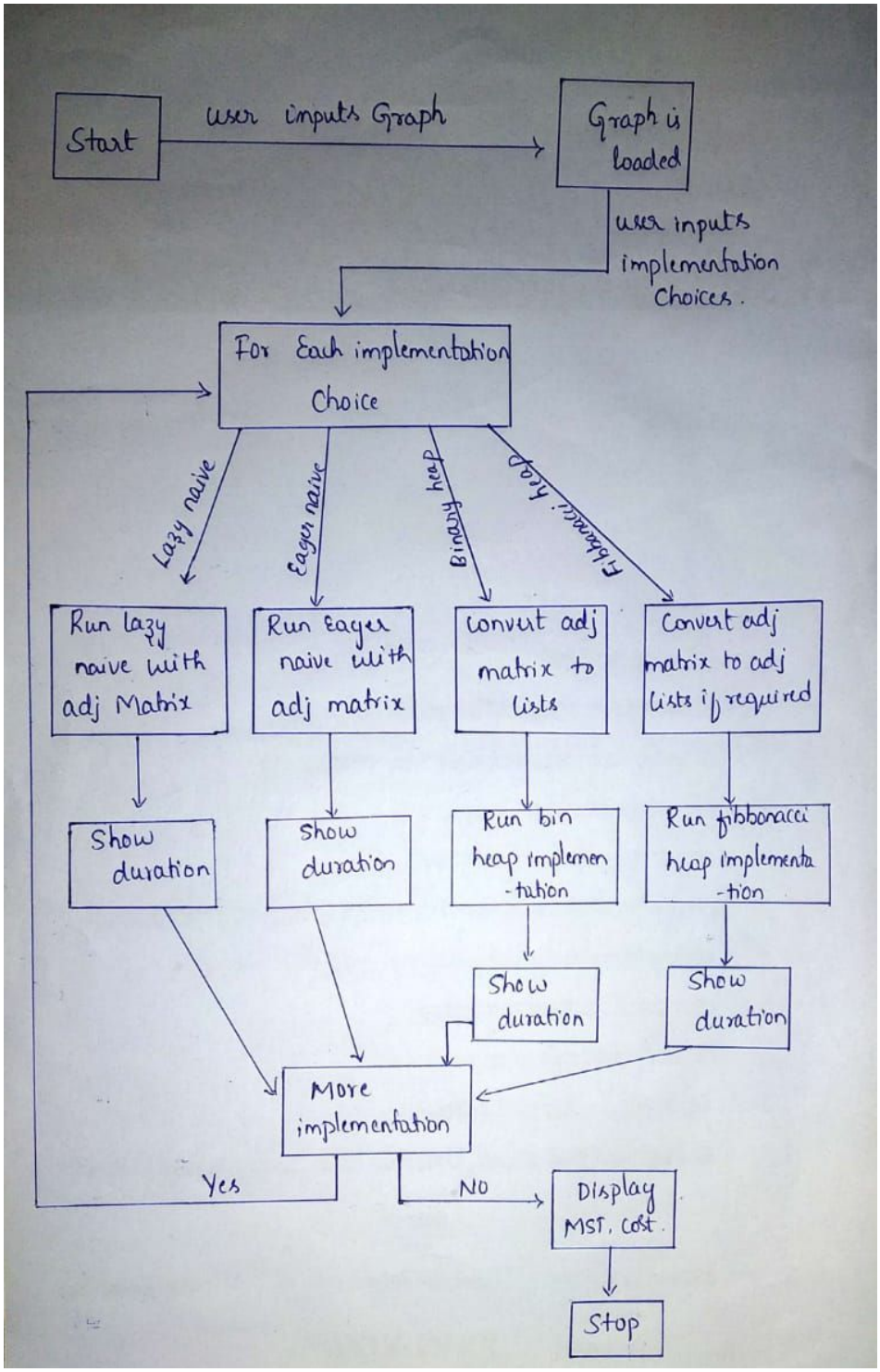
- RAM of 4 GB or above.
- Internal storage of 256 GB or above.
- Processor Intel i5 or above.

SOFTWARE REQUIREMENTS

- Programming Language: Python, version 3.6 or above.
- External Libraries Used: Numpy, Pandas, Matplotlib, NetworkX
- Version Control (For Development): Git and GitHub
- Operating System: Windows 10 or Ubuntu Linux.

SYSTEM DESIGN

ARCHITECTURE / DATA FLOW DIAGRAM



MODULES

User Interface: Driver.py

Data Structure for Graph: Graph.py

Binary Heap Data Structure: BinaryHeap.py

Fibonacci Heap Data Structure: FibHeap.py

Lazy Naive Implementation: LazyNaivePrims.py

Eager Naive Implementation: EagerNaivePrims.py

Implementation using Binary Heap: BinaryHeapPrims.py

Implementation using Fibonacci Heap: FibHeapPrims.py

Graph Visualization: VisualizeMst.py

Creation and Analysis of Test Cases: GraphGeneration.py, AutomatingTests.py, "Performance Comparison Plotter.ipynb"

SYSTEM IMPLEMENTATION

MODULE DESCRIPTION

Driver.py

Renders the user interface. Takes the name of the test file as a command line argument, but if not provided, prompts for it. Also reads implementation choices and display choice. Runs the algorithm's desired implementation and times it. Calls functions in other modules to display the MST.

Graph.py

Contains the class Graph. This class has functionalities like reading and printing the graph and changing the graph data from adjacency matrix to adjacency lists representation. This module also contains a function to find the adjacency matrix of the MST and its cost given a parent node - child node map of the MST.

LazyNaivePrims.py

This module contains the lazy naive implementation of Prim's algorithm, where edges are extracted and inserted, as opposed to vertices. This module contains the class EdgeList, inherited from built-in type list. EdgeList is the priority queue of edges.

EagerNaivePrims.py

This module contains the eager naive implementation of Prim's algorithm and the class KeyList, inherited from built-in type list. KeyList is the priority queue of vertices.

BinaryHeap.py

Contains the classes MinHeap and MinHeapForPrims. MinHeap is a binary min-heap which uses an array to store key values. MinHeapForPrims is inherited from MinHeap and is a priority queue to store vertex-key pairs. It maintains a map connecting vertices to nodes in the heap (since DecreaseKey takes the desired vertex as one of its inputs). Each heap node also contains the label of the vertex it is associated with (for ExtractMin since it returns a vertex).

BinaryHeapPrims.py

The implementation of Prim's algorithm which uses MinHeapForPrims as the priority queue.

TestsForHeap.py

Unit tests for binary heap operations.

FibHeap.py

Contains the classes FibHeap and FibHeapForPrims. FibHeap is a Fibonacci heap which stores key values. The data structure is built on multiple doubly circular linked lists. FibHeapForPrims serves a similar purpose as MinHeapForPrims.

FibHeapPrims.py

The implementation of Prim's algorithm which uses FibHeapForPrims as the priority queue.

TestsForFibHeap.py

Unit tests for Fibonacci heap operations.

VisualizeMst.py

Uses the networkX module to display the graph with the MST highlighted.

GraphGeneration.py

To generate test cases -- undirected connected graphs.

AutomatingTests.py

To create many test files for performance comparison purpose, put them through all the implementations, time the durations and store the durations in CSV files.

"Performance Comparison Plotter.ipynb"

Uses the data from CSV files to plot input size vs duration graphs to observe the differences in the performances of the implementations. The plotting is done with the help of the Matplotlib library.

PSEUDOCODE

Prim's Algorithm - Lazy Approach:

1) visited := {0}

2) pq := PriorityQueue()

3) Add all edges from vertex 0 to pq.

4) Repeat |V| - 1 times.

4.1) from_vx, cur_vx, weight = pq.ExtractMin()

4.2) If cur_vx is in visited, go to 4.1.

4.3) visited += {cur_vx}

4.4) Add all edges from vertex cur_vx to pq.

Prim's Algorithm - Eager Approach:

This is also how the other implementations work.

1) visited := {0}

2) pq := PriorityQueue()

3) Add all vertices to pq with key infinity.

4) pq[0] := 0

5) Repeat |V| times.

5.1) u = pq.ExtractMin()

5.3) visited += {u}

5.4) For all unvisited vertices adjacent to u (say v)

5.4.1) If pq[v] > weight of edge u-v

5.4.1.1) pq.DecreaseKey(v, weight of edge u-v)

Binary Heap Operations:

SiftUp(idx):

1) Repeat until heap property is restored:

1.1) parent_idx := (idx - 1) // 2 (where // is integer division)

1.2) if arr[idx] < arr[parent_idx]

1.2.1) Swap(arr[idx], arr[parent_idx]).

1.2.2) idx := parent_idx

SiftDown(idx):

1) Repeat until heap property is restored:

1.1) left_child_idx := idx * 2

1.2) right_child_idx := left_child_idx + 1

1.3) desired_child_idx := index of minimum child

1.4) If arr[idx] > arr[desired_child_idx]:

1.4.1) swap(arr[idx], arr[desired_child_idx])

1.4.2) idx = desired_child_idx

Insert(key):

1) arr.append(key)

2) SiftUp(size(arr) - 1)

DecreaseKey(idx, newkey):

1) arr[idx] := newkey

2) SiftUp(idx)

ExtractMin():

1) Swap(arr[0], arr[size(arr) - 1])

2) Discard and return arr[size(arr) - 1]

Fibonacci Heap Operations

Merge (fibheapA, fibheapB):

1) fibheapC.root_list = concatenate(fibheapA.root_list, fibheapB.root_list)

2) fibheapC.min_node = min(fibheapA.min_node, fibheapB.min_node)

ExtractMin():

1) Remove min_node from root_list.

2) Add the min_node's children to root_list.

3) Consolidate root_list i.e. merge trees of the same rank.

4) min_node = min(root_list).

Insert(key):

1) Create a new fibheap consisting of one node containing key.

2) Merge(fibheap, new fibheap)

DecreaseKey(node, newkey):

1) node.key := newkey

2) If node.parent is None:

2.1) Return

3) If node.parent.key <= node.key:

3.1) Return

4) Cut(node)

5) CascadingCut(node.parent)

Cut(node):

1) Remove node from node.parent.child list.

2) Decrement node.parent.rank.

3) Add node to root list.

CascadingCut(node):

1) If node is marked:

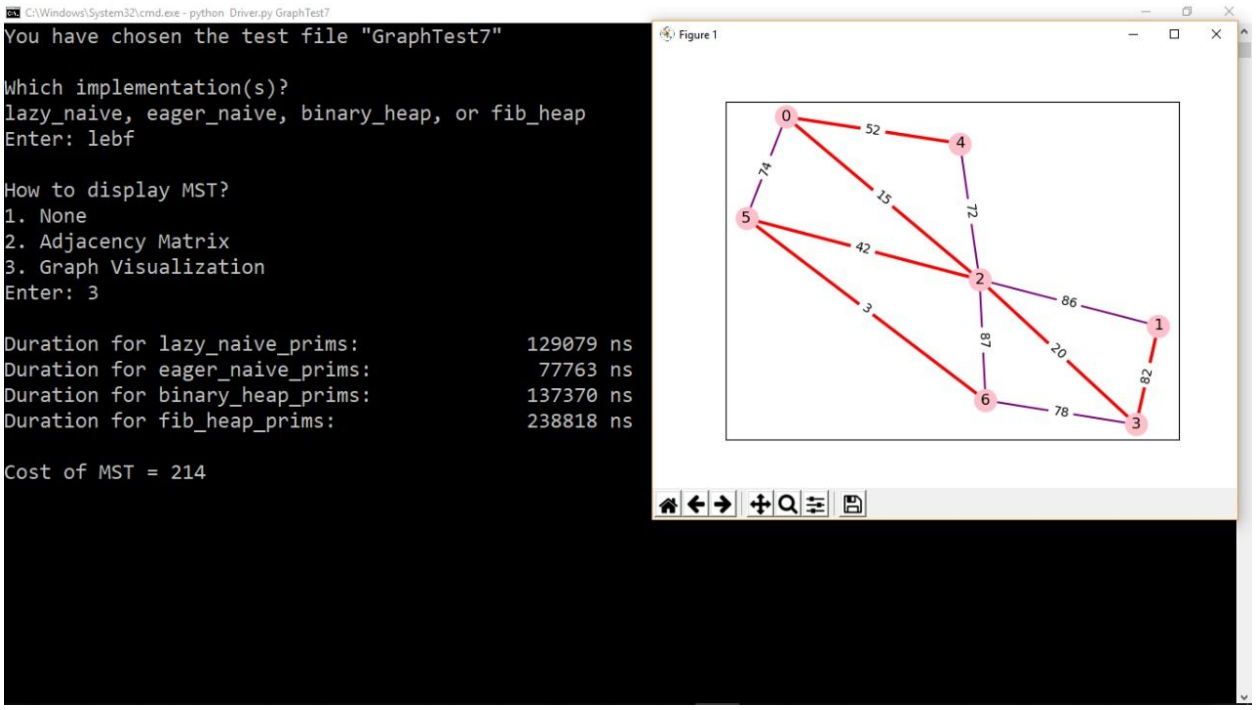
1.1) Cut(node)

1.2) CascadingCut(node.parent)

2) Else:

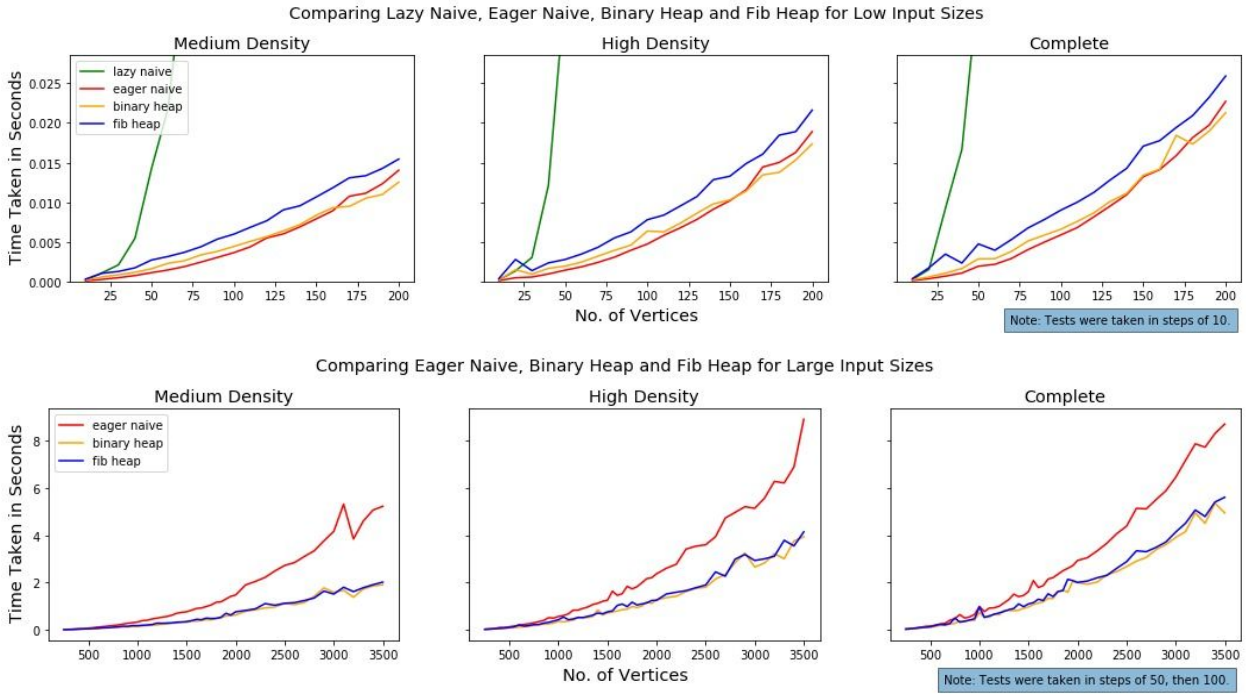
2.1) Mark node.

OUTPUT SCREENSHOTS



OBSERVATIONS

We generated three tests each with input sizes from 10 to 200 in steps of 10, then 250 to 2000 in steps of 50, and then 2100 to 3500 in steps of 100. Note that input size refers to the number of vertices. For each input size, we generated one medium density graph, one high density graph and one complete graph. There is no low density category because we are considering a low density graph to be a graph solely consisting of a spanning tree.



On running the implementations on these tests, we observed that:

1) Lazy Naive Approach takes the most time for every test case and therefore is an inefficient way to find the MST for a graph.

2) For low input sizes (10 to 200), the Fibonacci heap implementation took much more time than the other two. The eager naive approach took slightly lesser time than the binary heap implementation until the input size of approximately 160 vertices.

3) For large input sizes (250 to 3500), the two heap implementations have more or less the same duration. The eager naive approach proves to be very inefficient when the input size exceeds 1000 vertices.

Since the Fibonacci heap implementation has a better time complexity than the others, we expected that it would take the least amount of time for very large graphs. But, it looks like we will need larger test cases to reach that threshold. We tried with an input size of 10,000 but the program ran out of memory.

CONCLUSION

The lazy naive approach is not an efficient implementation of the Prim's algorithm for any input size.

The eager naive approach is easily implementable and proves efficient for graphs with less than one thousand vertices.

The Fibonacci heap implementation takes a lot of time for graphs with low input sizes but its performance is almost the same as the binary heap implementation for graphs with more than 500 vertices.

REFERENCES

1. Prim, R. C. (November 1957), "Shortest connection networks and some generalizations", Bell System Technical Journal
2. Fredman, Michael Lawrence; Tarjan, Robert E. (July 1987). "Fibonacci heaps and their uses in improved network optimization algorithms". Journal of the Association for Computing Machinery.
3. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990]. "Chapter 20: Fibonacci Heaps". Introduction to Algorithms