**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

Laboratory work 5:

Study and Empirical Analysis of Algorithms: Prim and Kruskal

Elaborated:

st. gr. FAF-233Iamandii Ion

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2025

# ALGORITHM ANALYSIS

## Objective

## Analysis of Prim and Kruskal algorithms.

## Tasks:

## 1 Implement the algorithms listed above in a programming language

## 2 Establish the properties of the input data against which the analysis is performed

## 3 Choose metrics for comparing algorithms

## 4 Perform empirical analysis of these algorithms.

## 5 Increase the number of nodes in graphs and analyze how this influences the algorithms. Make a graphical presentation of the data obtained

## 6 Make a conclusion on the work done.

## Theoretical Notes:

An alternative to mathematical analysis of complexity is empirical analysis.

This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer.

In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm.
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).
4. The algorithm is implemented in a programming language.
5. Generating multiple sets of input data.
6. Run the program for each input data set.
7. The obtained data are analyzed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

## Introduction:

## Among all the algorithmic approaches, the simplest and straightforward approach is the Greedy method. In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.

## Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit. This approach never reconsiders the choices taken previously. This approach is mainly used to solve optimization problems. Greedy method is easy to implement and quite efficient in most of the cases. Hence, we can say that Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.

## In many problems, it does not produce an optimal solution though it gives an approximate (near optimal) solution in a reasonable time.

## Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n))

## Input Format:

The input for the minimum spanning tree (MST) algorithms consists of undirected, weighted graphs generated in both adjacency list format (for Prim's algorithm) and edge list format (for Kruskal's algorithm). Graph sizes vary from small to large (e.g., 10, 50, 100, up to 400 nodes), with edge weights uniformly distributed between 1 and 10. For each size, two graph variants are generated: sparse graphs (10% edge probability) and dense graphs (50% edge probability), simulating real-world scenarios of varying connectivity. All node identifiers are integers ranging from 0 to n−1n−1, and all edges are bidirectional with symmetric weights. These input specifications ensure compatibility with both MST algorithms and allow for consistent empirical benchmarking under controlled variations in graph size and density.

# IMPLEMENTATION

Both Prim’s algorithm (implemented using a min-heap priority queue with adjacency list representation) and Kruskal’s algorithm (using a sorted edge list and a disjoint-set union–find structure) are coded in Python and evaluated empirically. Prim’s algorithm operates from an arbitrary starting node and grows the MST incrementally, while Kruskal’s algorithm globally sorts all edges and adds them greedily without forming cycles. Each algorithm is tested on both sparse and dense graphs over a range of node counts. Execution time is recorded and plotted to visualize scalability and efficiency trends. Due to Kruskal's reliance on global edge sorting, it tends to perform better on sparse graphs, while Prim’s heap-based structure offers advantages in denser configurations where adjacency lists remain efficient. The resulting performance curves reveal practical trade-offs shaped by graph topology, data structure overhead, and the underlying time complexity of each approach.

The error margin determined will constitute 2.5 seconds as per experimental measurement.

Github repo: <https://github.com/ion190/aa-labs/tree/main/lab4>

## Prim Algorithm:

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 that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

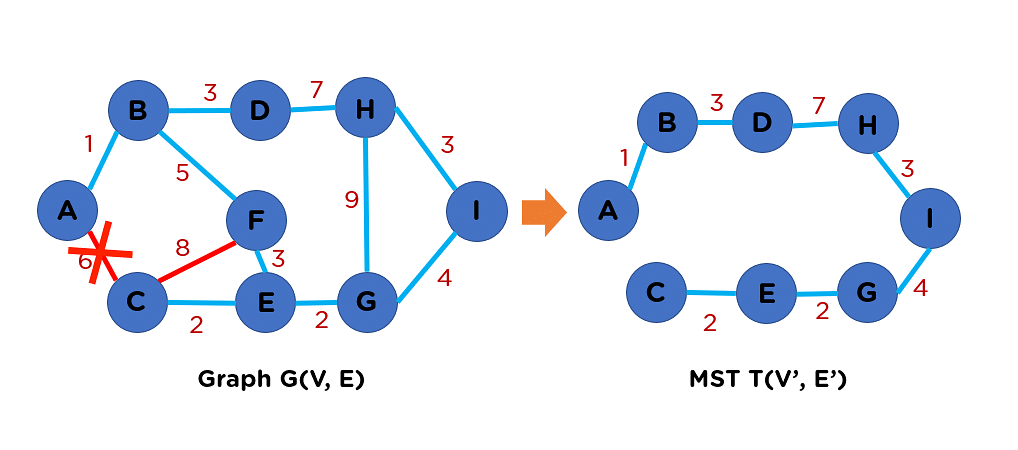


Figure 1. Prim example

*Algorithm Description:*

The Prim algorithm follows the algorithm as shown in the next pseudocode:

function Prim(vertices, edges) is

for each vertex in vertices do

cheapestCost[vertex] ← ∞

cheapestEdge[vertex] ← null

explored ← empty set

unexplored ← set containing all vertices

startVertex ← any element of vertices

cheapestCost[startVertex] ← 0

while unexplored is not empty do

// Select vertex in unexplored with minimum cost

currentVertex ← vertex in unexplored with minimum cheapestCost[vertex]

unexplored.remove(currentVertex)

explored.add(currentVertex)

for each edge (currentVertex, neighbor) in edges do

if neighbor in unexplored and weight(currentVertex, neighbor) < cheapestCost[neighbor] THEN

cheapestCost[neighbor] ← weight(currentVertex, neighbor)

cheapestEdge[neighbor] ← (currentVertex, neighbor)

resultEdges ← empty list

for each vertex in vertices do

if cheapestEdge[vertex] ≠ null THEN

resultEdges.append(cheapestEdge[vertex])

return resultEdges

*Implementation:*



*Figure 2 Prim algorithm in Python*

## Kruskal Algorithm:

Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. It is a greedy algorithm that in each step adds to the forest the lowest-weight edge that will not form a cycle. The key steps of the algorithm are sorting and the use of a disjoint-set data structure to detect cycles. Its running time is dominated by the time to sort all of the graph edges by their weight.

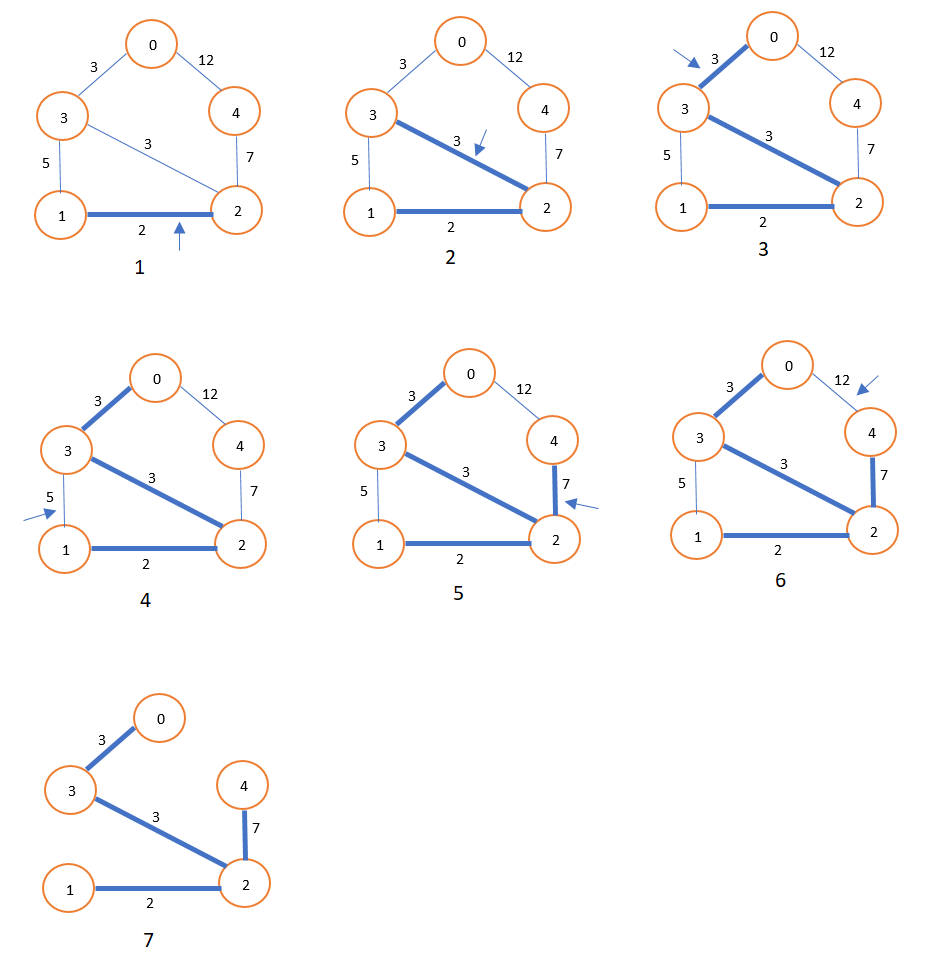


Figure 3. Kruskal Algorithm example

*Algorithm Description:*

The Kruskal Algorithm follows the algorithm as shown in the next pseudocode:

function Kruskal(Graph G) is

F:= ∅

for each v in G.Vertices do

MAKE-SET(v)

for each {u, v} in G.Edges ordered by increasing weight({u, v}) do

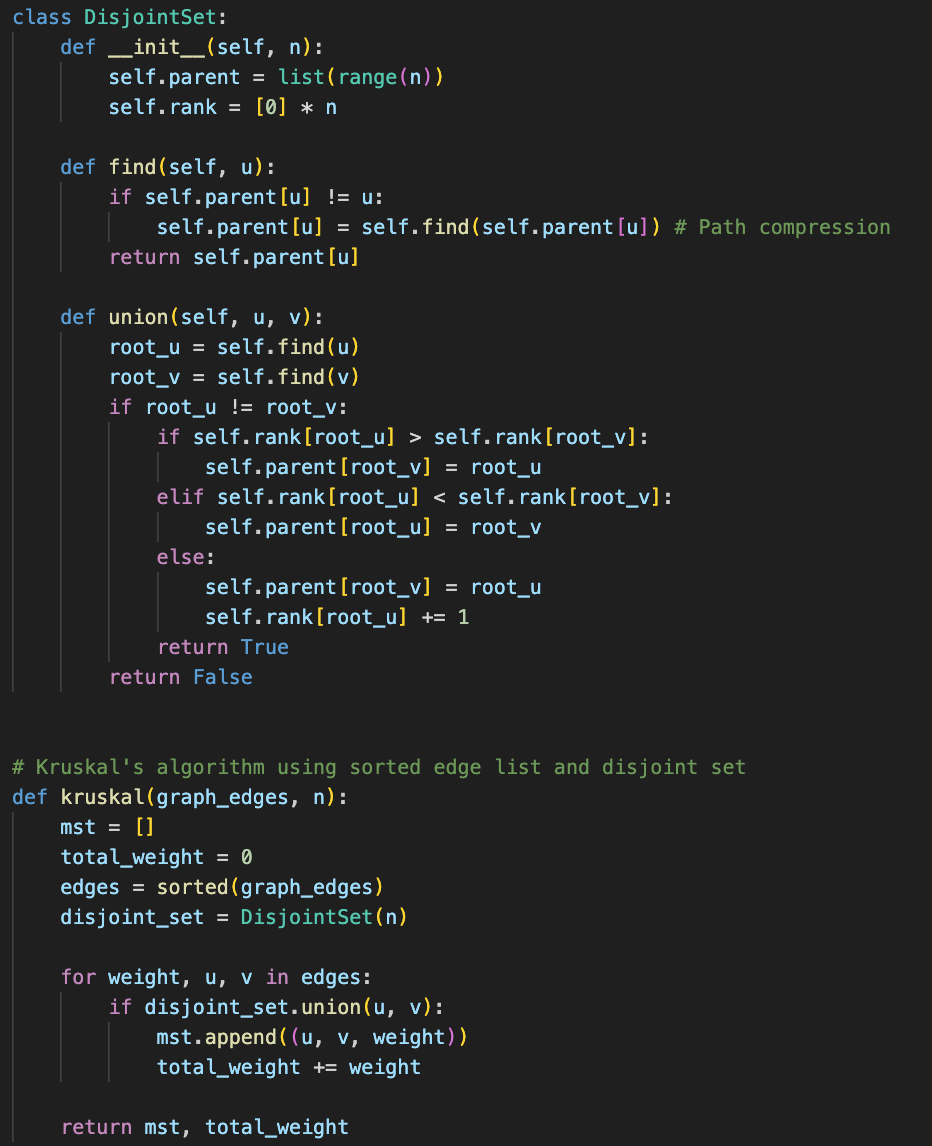
if FIND-SET(u) ≠ FIND-SET(v) then

F := F ∪ { {u, v} }

UNION(FIND-SET(u), FIND-SET(v))

return F

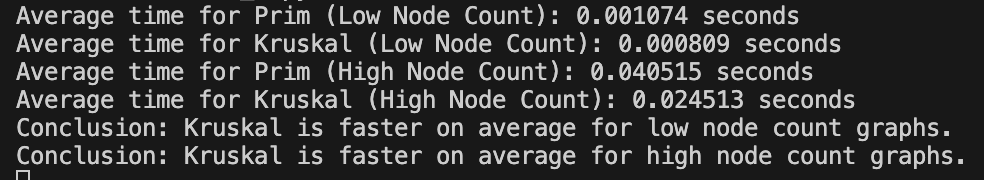
*Implementation:*



*Figure 4 Kruskal Algorithm in Python*

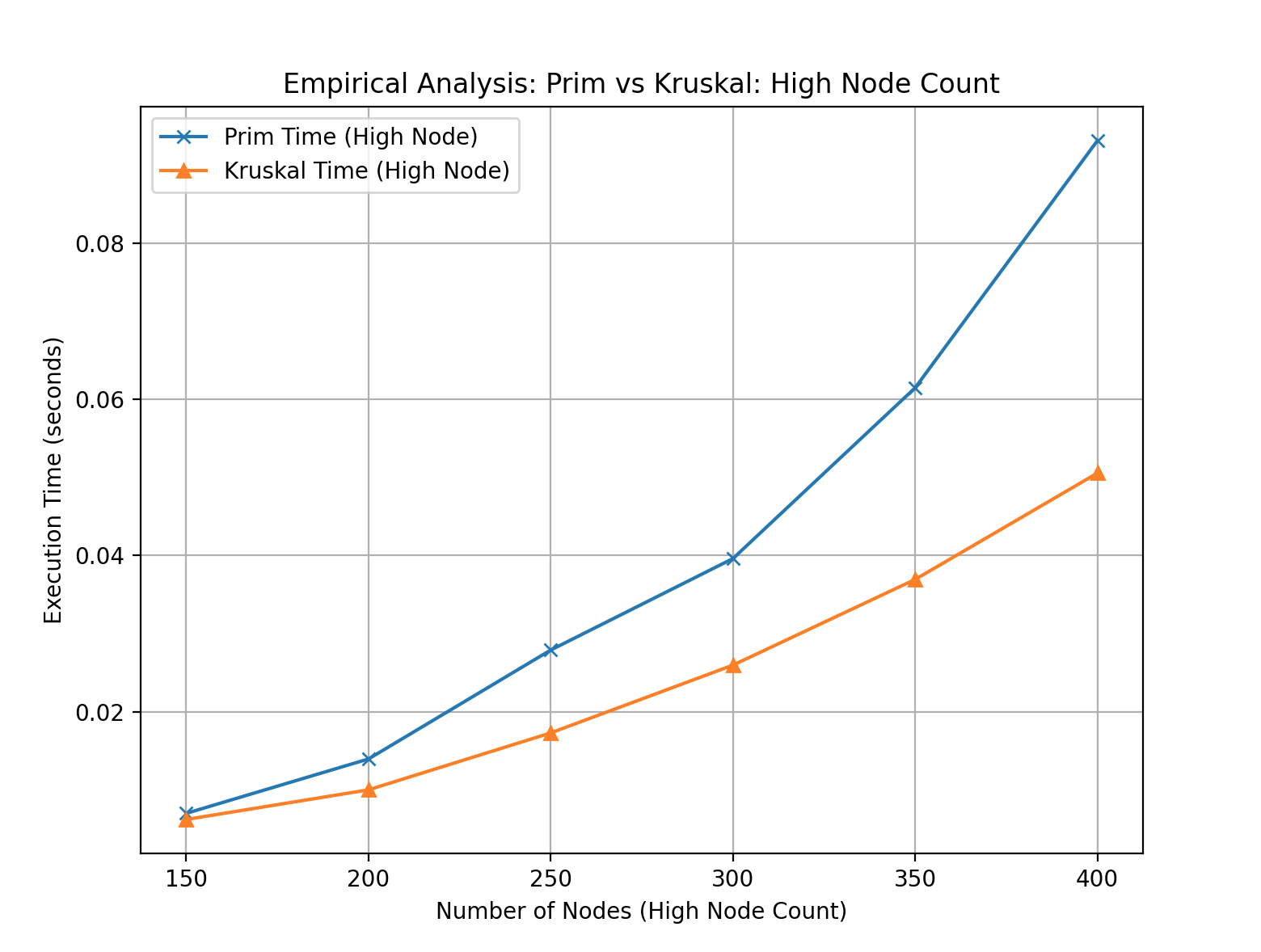
***Results:***

After running the function for different sizes of graphs and saving the time for each, we obtained the following results:



*Figure 5 Results for various set of inputs*

The graph showing the time needed to sort different sizes of graphs:



*Figure 6 Graph of Prims and Kruskal algorithms*

## Prim’s algorithm, when implemented with a min-heap priority queue and adjacency list, has a time complexity of O((V+E)logV)O((V+E)logV), where VV is the number of vertices and EE is the number of edges. This logarithmic factor arises from heap operations used to extract the minimum-weight edge efficiently. In sparse graphs—where E≈VE≈V—Prim’s algorithm performs near-linearly, offering fast execution even on larger graphs. In dense graphs, where E≈V2E≈V2, the number of heap insertions and updates increases substantially, leading to longer execution times. However, the algorithm remains competitive due to its incremental structure and localized decision-making. Practical performance is influenced by the efficiency of the heap implementation and the overhead of maintaining visited nodes and edge weights during traversal.

## Kruskal’s algorithm, on the other hand, has a time complexity of O(ElogE)O(ElogE), primarily due to the initial sorting of all edges, followed by near-constant-time union–find operations. In sparse graphs, where EE is much less than V2V2, the sorting step dominates but remains relatively fast, making Kruskal especially efficient in low-density scenarios. In dense graphs, the edge count increases quadratically, resulting in longer sorting times and more frequent union–find operations, although the use of path compression and union by rank keeps these efficient. Kruskal’s global edge-oriented approach performs well when edge weights are widely distributed and when the graph is not highly connected. Its simplicity and modularity make it a preferred choice for sparse graphs and batch-processing contexts where all edges are known upfront.

# CONCLUSION

Through empirical analysis, this study assesses the performance of two minimum spanning tree (MST) algorithms—Prim’s algorithm and Kruskal’s algorithm—focusing on execution time and scalability across sparse and dense graphs of varying sizes, with the aim of identifying their optimal application contexts.

Prim’s algorithm, implemented with a priority queue and adjacency list, constructs the MST incrementally with a time complexity of O((V+E)logV)O((V+E)logV). It performs exceptionally well on sparse graphs, where fewer edge evaluations lead to minimal heap operations and efficient growth of the spanning tree. In dense graphs, the increased edge count introduces more frequent heap updates, leading to slower execution, though the algorithm maintains reliable performance due to its localized and greedy nature. Its node-centric design makes it particularly effective when working with connected graphs and adjacency list representations.

Kruskal’s algorithm, by contrast, sorts all edges initially and uses a disjoint-set data structure to build the MST, yielding a time complexity of O(ElogE)O(ElogE). Its performance is particularly favorable on sparse graphs, where the edge set is small and sorting is quick. In dense graphs, sorting becomes more time-consuming, but union–find optimizations help maintain efficiency during edge selection. Kruskal’s edge-based approach is advantageous when the full edge list is available and global ordering is feasible.

Based on the empirical findings, Prim’s algorithm excels in sparse to moderately dense graphs where adjacency lists are preferred and incremental construction is advantageous. Kruskal’s algorithm is more efficient in scenarios with readily accessible edge lists and less connectivity, where its sorting-based strategy quickly identifies optimal connections. Both algorithms are effective MST solutions, and their suitability depends on graph representation, density, and the nature of the problem domain.