

MINISTRY OF EDUCATION OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
SOFTWARE ENGINEERING DEPARTMENT

ALGORITHM ANALYSIS

LABORATORY WORK #5

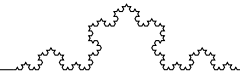
Greedy Algorithms

Author: Timur CRAVTOV
std. gr. FAF-231

Verified by: Cristofor FISTIC
asist. univ



Chişinău
2025



Contents

1	Algorithm Analysis	3
1.1	Objective	3
1.2	Task	3
1.3	Introduction	3
2	Implementation	3
2.1	Prim's algorithm	3
2.2	Kruskal's algorithm	5
2.3	Quick sort	6
2.4	Heap sort	7
2.5	Merge sort	7
3	Bonus: visualization	8
4	Conclusions	8



1 Algorithm Analysis

1.1 Objective

Study and analyze different algorithms for building minimum spanning trees. The analysis should be performed on the basis of the following algorithms:

1. Prim's algorithm
2. Kruskal's algorithm

1.2 Task

1. Study the greedy algorithm design technique
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 the proposed algorithms;
5. Make a graphical presentation of the data obtained;
6. Make a conclusion on the work done.

1.3 Introduction

Greedy algorithms are a class of algorithms that make a series of choices, each of which looks best at the moment. The idea is to make a locally optimal choice at each step with the hope that these choices will lead to a globally optimal solution. Greedy algorithms are often used for optimization problems, where the goal is to find the best solution among many possible solutions.

One of the application of Greedy algorithm is building a minimum spanning tree (MST) of a weighted undirected graph. A minimum spanning tree is a subset of the edges of the graph that connects all the vertices together without any cycles and with the minimum possible total edge weight. In this laboratory work, we will analyze two greedy algorithms for building a minimum spanning tree: Prim's algorithm and Kruskal's algorithm.

2 Implementation

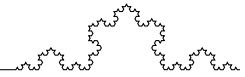
2.1 Prim's algorithm

Algorithm description



Prims algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. It works by starting with a single vertex and adding edges to the tree until all vertices are included. The algorithm maintains a priority queue of edges, and at each step, it adds the edge with the smallest weight that connects a vertex in the tree to a vertex outside the tree.

```
1 import networkx as nx
2 import heapq
3
4
5 def min_span_tree_prim(G: nx.Graph) -> nx.Graph:
6     if not G.nodes:
7         return nx.Graph()
8
9     mst = nx.Graph()
10    visited = set()
11    min_heap = []
12
13    # Start from an arbitrary node
14    start_node = list(G.nodes)[0]
15    visited.add(start_node)
16
17    # Push all edges from the start node into the heap
18    for neighbor in G.neighbors(start_node):
19        weight = G[start_node][neighbor]['weight']
20        heapq.heappush(min_heap, (weight, start_node, neighbor))
21
22    while min_heap and len(visited) < G.number_of_nodes():
23        weight, u, v = heapq.heappop(min_heap)
24
25        if v not in visited:
26            visited.add(v)
27            mst.add_edge(u, v, weight=weight)
28
29            for neighbor in G.neighbors(v):
30                if neighbor not in visited:
31                    heapq.heappush(min_heap, (G[v][neighbor]['weight'], v, neighbor))
32
```



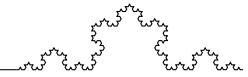
```
33 | return mst
```

2.2 Kruskal's algorithm

Algorithm description

Kruskal algorithm works by sorting all the edges in the graph by their weights and adding them to the minimum spanning tree one by one, as long as they do not form a cycle. The algorithm uses a disjoint-set data structure to keep track of which vertices are in which components.

```
1 import networkx as nx
2
3
4 def min_span_tree_kruskal(G: nx.Graph) -> nx.Graph:
5     if not G.nodes:
6         return nx.Graph()
7
8     # Create a list of edges with weights
9     edges = sorted(G.edges(data=True), key=lambda x: x[2]['
10         weight'])
11
12     parent = {node: node for node in G.nodes}
13     rank = {node: 0 for node in G.nodes}
14
15     def find(v):
16         if parent[v] != v:
17             parent[v] = find(parent[v]) # Path compression
18         return parent[v]
19
20     def union(u, v):
21         root_u = find(u)
22         root_v = find(v)
23         if root_u != root_v:
24             if rank[root_u] > rank[root_v]:
25                 parent[root_v] = root_u
26             elif rank[root_u] < rank[root_v]:
27                 parent[root_u] = root_v
28             else:
```



```

29         rank[root_u] += 1
30     return True
31     return False
32
33     mst = nx.Graph()
34
35     for u, v, data in edges:
36         if union(u, v):
37             mst.add_edge(u, v, weight=data['weight'])
38
39     return mst

```

Implementation

Results

The listing below shows the raw data of algorithm execution

2.3 Quick sort

Algorithm description

QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

The algorithm complexity for best case and average case is $O(n \log n)$. However, the worst case is $O(n^2)$, and example of that is provided below in this section.

There are different way to choose the pivot in Quick Sort:

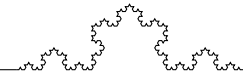
1. *The last element*: Simply take the pivot as the last element of the array
2. *Random*: Each pivot is selected randomly
3. *Mean of the data*: Before selecting the pivot, the mean of the subarray is calculated and set as pivot. In this approach, it is guaranteed that the left and right subarrays have the same size.

For this implementation, I choose the first option.

```

1 FUNCTION QUICK_SORT(ARRAY, LOW, HIGH)
2     IF LOW < HIGH THEN
3         SET PIVOT_INDEX TO PARTITION(ARRAY, LOW, HIGH)
4
5         CALL QUICK_SORT(ARRAY, LOW, PIVOT_INDEX - 1)
6         CALL QUICK_SORT(ARRAY, PIVOT_INDEX + 1, HIGH)

```



```

7   END IF
8 END FUNCTION

```

Implementation

Results

During the execution of dataset consisted with around 800 elements, the Python Compiler threw the execution, meaning the recursion is too deep. Since no improvement were made in this algorithm, the error was ignored, and sample data is adjusted.

It becomes clear from the data that the more sorted the data is, the more times it needs, being $O(n^2)$ in worse case scenario of sorted array. It can be explain in the following way: the pivot element, taken each time the last, divides the subarray in the element itself and the rest of the array. It doesn't performs any checks which doesn't stop the execution.

2.4 Heap sort

Heap sort is a comparison-based sorting technique based on Binary Heap Data Structure. It can be seen as an optimization over selection sort where we first find the max (or min) element and swap it with the last (or first). We repeat the same process for the remaining elements. In Heap Sort, we use Binary Heap so that we can quickly find and move the max element in $O(\log n)$ instead of $O(n)$ and hence achieve the $O(n \log n)$ time complexity.

Implementation

Results

2.5 Merge sort

Algorithm description

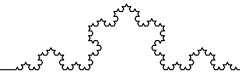
Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In listing below there is provided pseudocode of the algorithm.

```

1 FUNCTION MERGESORT(ARRAY, LEFT, RIGHT)
2   IF LEFT < RIGHT THEN
3     SET MID TO (LEFT + RIGHT) / 2
4
5     CALL MERGESORT(ARRAY, LEFT, MID)
6     CALL MERGESORT(ARRAY, MID + 1, RIGHT)
7
8     CALL MERGE(ARRAY, LEFT, MID, RIGHT)

```



```
9      END IF
10 END FUNCTION
11
12 FUNCTION MERGE(ARRAY, LEFT, MID, RIGHT)
13     CREATE LEFT.SUBARRAY FROM ARRAY[LEFT TO MID]
14     CREATE RIGHT.SUBARRAY FROM ARRAY[MID + 1 TO RIGHT]
15
16     MERGE LEFT.SUBARRAY AND RIGHT.SUBARRAY BACK INTO ARRAY[LEFT
17         TO RIGHT]
18 END FUNCTION
```

Implementation

Results

3 Bonus: visualization

The visualization service was made as generic as possible: it takes an array and the sorting function which must accept only one argument: the array.

To track each update of the array modification (such as swapping), the array was wrapped into a new class which stops the thread on each array element setting (swapping counts for two).

The full code is available on GitHub repository^[1] (lab2/code/visualizr.py). The image below is a frame of the visualization process of Heap Sort.

4 Conclusions

During this laboratory work, the analysis of four sorting algorithms were performed.

Each algorithm was tested on different datasets, such as random, sorted and partially sorted to determine their complexity in dependence of the data (to find out average, worst and best case complexity), as well as on different datasets to track the algorithm complexity growth with growth of the size of the array.

References

- [1] <https://github.com/TimurCravtov/AA-labs>