Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Algorithm Analysis

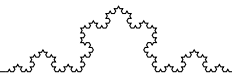
Laboratory work #4

Dynamic Programming

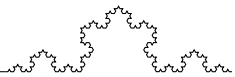
Author: Timur Cravtov
std. gr. FAF-231

Verified by: Cristofor Fistic
asist. univ

Chișinău
2025



Contents



1 Algorithm Analysis

1.1 Objective

Study and analyze different algorithms for finding the shortest path in a graph. The algorithms to be analyzed are:

1. Floyd-Warshall algorithm
2. Dijkstra's algorithm

1.2 Task

1. To study the dynamic programming method of designing algorithms.
2. To implement in a programming language algorithms Dijkstra and Floyd-Warshall using dynamic programming.
3. Do empirical analysis of these algorithms for a sparse graph and for a dense graph.
4. Increase the number of nodes in graphs and analyze how this influences the algorithms. Make a graphical presentation of the data obtained

1.3 Introduction

Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable when the subproblems are overlapping, meaning that the same subproblems are solved multiple times. Dynamic programming is often used in optimization problems, where the goal is to find the best solution among many possible solutions.

One of the application of dynamic programming is in finding the shortest path in a graph. The shortest path problem is a classic problem in computer science and has many applications, such as in transportation networks, communication networks, and social networks. The shortest path problem can be solved using various algorithms, including Dijkstra's algorithm and the Floyd-Warshall algorithm. Dijkstra's algorithm is a greedy algorithm that finds the shortest path from a source vertex to all other vertices in a weighted graph. The Floyd-Warshall algorithm is a dynamic programming algorithm that finds the shortest paths between all pairs of vertices in a weighted graph.

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.

```python
import networkx as nx
import heapq


def min_span_tree_prim(G: nx.Graph) -> nx.Graph:
    if not G.nodes:
        return nx.Graph()

    mst = nx.Graph()
    visited = set()
    min_heap = []

    # Start from an arbitrary node
    start_node = list(G.nodes)[0]
    visited.add(start_node)

    # Push all edges from the start node into the heap
    for neighbor in G.neighbors(start_node):
        weight = G[start_node][neighbor]['weight']
        heapq.heappush(min_heap, (weight, start_node, neighbor))

    while min_heap and len(visited) < G.number_of_nodes():
        weight, u, v = heapq.heappop(min_heap)

        if v not in visited:
            visited.add(v)
            mst.add_edge(u, v, weight=weight)

            for neighbor in G.neighbors(v):
```

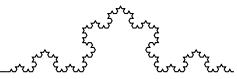
```
                if neighbor not in visited:
                    heapq.heappush(min_heap, (G[v][neighbor]['
                        weight'], v, neighbor))

    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.

```
import networkx as nx


def min_span_tree_kruskal(G: nx.Graph) -> nx.Graph:
    if not G.nodes:
        return nx.Graph()

    # Create a list of edges with weights
    edges = sorted(G.edges(data=True), key=lambda x: x[2]['
        weight'])

    parent = {node: node for node in G.nodes}
    rank = {node: 0 for node in G.nodes}

    def find(v):
        if parent[v] != v:
            parent[v] = find(parent[v])  # Path compression
        return parent[v]

    def union(u, v):
        root_u = find(u)
        root_v = find(v)
        if root_u != root_v:
            if rank[root_u] > rank[root_v]:
                parent[root_v] = root_u
```



```
25              elif rank[root_u] < rank[root_v]:
26                  parent[root_u] = root_v
27              else:
28                  parent[root_v] = root_u
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 lof 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.

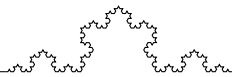
The algorithm complexity for best case and average case is $O(n \log n)$. However, the worst case is $O(n^2)$, and example of that is provided below in this section.

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 exection of dataset consisted with around 800 elements, the Python Compiler threw the exection, 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, devides 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 MERGE_SORT(ARRAY, LEFT, RIGHT)
2       IF LEFT < RIGHT THEN
3           SET MID TO (LEFT + RIGHT) / 2
4
```

```
5           CALL MERGE_SORT(ARRAY, LEFT, MID)
6           CALL MERGE_SORT(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
            TO RIGHT]
17  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 avaliable 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 dataasets, 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