FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
ITMO UNIVERSITY

Report
on the practical task No. 8
"***Practical analysis of advanced algorithms***"

Performed by
Abizer Safdari
J4134c
Accepted by
Dr Petr Chunaev

St. Petersburg
2020

Goal

Practical analysis of advanced algorithms

Formulation of the problem

I Foundations

4 Divide-and-Conquer

VI Graph Algorithms

23 Minimum Spanning Trees

IV Advanced Design and Analysis Techniques

16 Greedy Algorithms

I. *Choose* ***two*** *algorithms (interesting to you and not considered in the course) from the above-mentioned book sections.*

II. *Analyse the chosen algorithms in terms of time and space complexity, design technique used, etc. Implement the algorithms and produce several experiments. Analyse the results*

Brief theoretical part

- **Divide and conquer** is a way to break complex problems into smaller problems that are easier to solve, and then combine the answers to solve the original problem. Divide and conquer is a powerful algorithm design technique used to solve many important problems such as mergesort, quicksort, calculating Fibonacci numbers, and performing matrix multiplication.
    - **Divide:**
      The divide step breaks the original problem into subproblems that are smaller instances of the original problem.
    - **Conquer:**
      The conquer step solves the subproblems recursively.
    - **Combine:**
      The combine step puts the solved subproblems together to solve the original problem.
- Merge Sort is a kind of Divide and Conquer algorithm in computer programming. It is one of the most popular sorting algorithms and a great way to develop confidence in building recursive algorithms.
    - The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. `p == r`.
    - After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.
- Spanning Tree:-Given an undirected and connected graph G=(V,E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)
- The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.
- A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.
- Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems.
- **Kruskal's Minimum Spanning Tree (MST):** In Kruskal's algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the

smallest weight edge that doesn't cause a cycle in the MST constructed so far.

Results

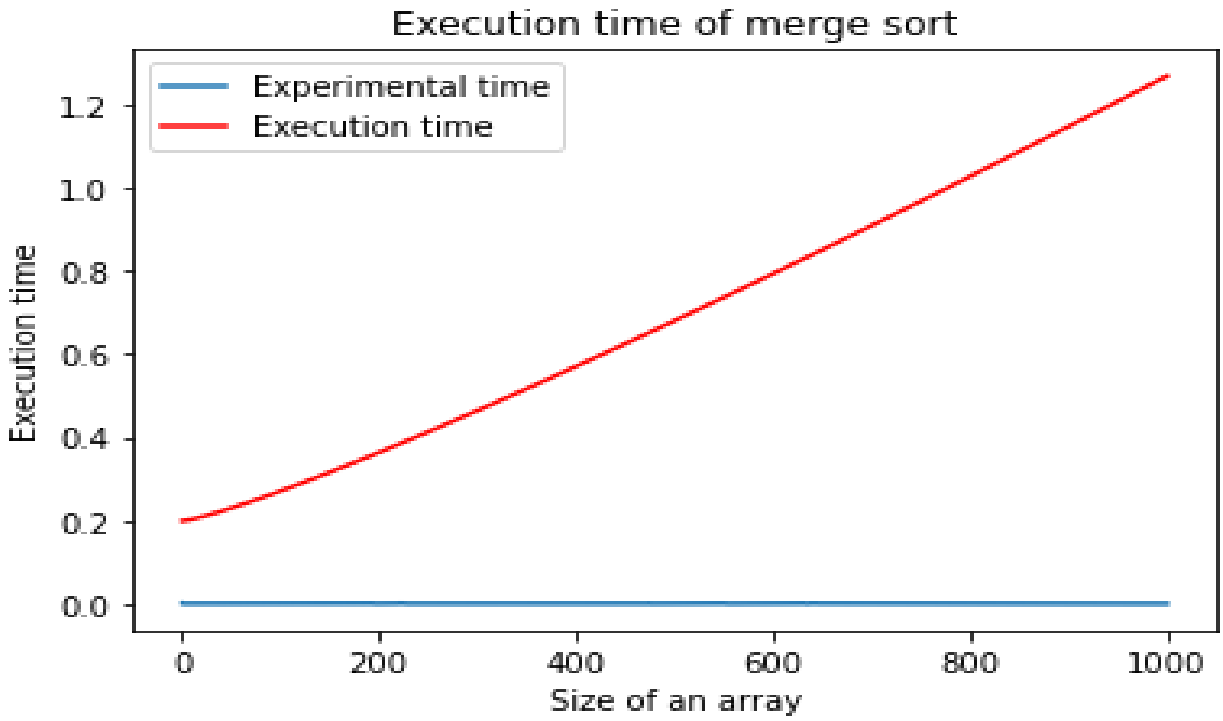

Figure1: *graph represent the time complexity of merge sort.*

Following are the edges in the constructed MST

…….
……
…..
….
1 - 22: 2
1 - 55: 2
1 - 75: 2
1 - 76: 2
1 - 81: 2
1 - 87: 2
2 - 41: 2
3 - 26: 2
3 - 47: 2
3 - 58: 2
4 - 29: 2
4 - 50: 2
4 - 56: 2
4 - 84: 2

4 - 91: 2
5 - 39: 2
6 - 11: 2
7 - 33: 2
7 - 67: 2
7 - 74: 2
8 - 38: 2
8 - 53: 2
9 - 82: 2
1 - 31: 3
1 - 59: 3
2 - 20: 3
3 - 88: 3
5 - 63: 3
6 - 32: 3
6 - 90: 3
1 - 100: 4
2 - 61: 5
Result of kruskal's algo.

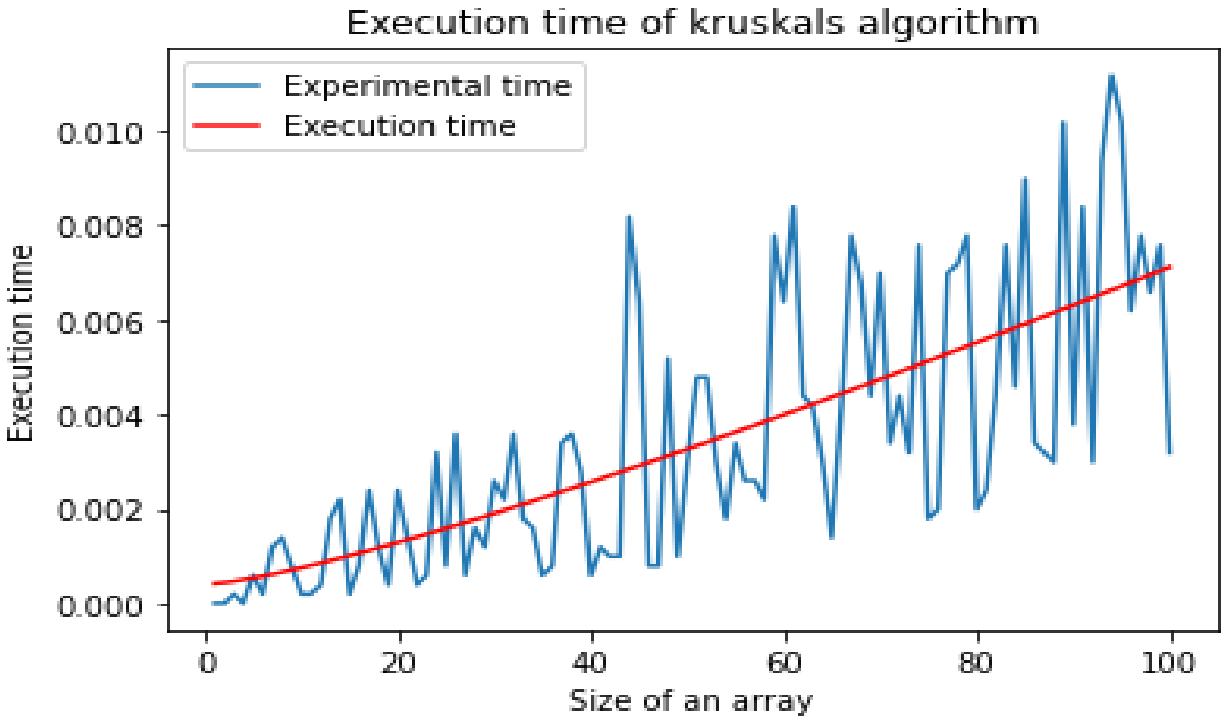

Figure 2: graph represent the time complexity of Kruskal's Algorithm

Conclusions

Time complexity of merge sort is : O(n*log n)

Space complexity of merge sort is : O(n)

Time complexity of Kruskal's Algorithm is: O(E log E).

Appendix

```
"""
Created on Mon Oct 10 03:01:22 2020

@author: abizer
"""
(For Merge Algorithm)
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import time
import numpy as np
def time1(v):
    a=[]
    for i in range(5):
        start_time=time.time()
        mergeSort(array)
        a.append(time.time()-start_time)
    a=sum(a)/5
    return a

#merge sort using divide and conqure
def mergeSort(array):
    if len(array) > 1:

        #  r is the point where the array is divided into two subarrays
        r = len(array)//2
        L = array[:r]
        M = array[r:]

        # Sort the two halves
        mergeSort(L)
        mergeSort(M)

        i = j = k = 0

        # Until we reach either end of either L or M, pick larger among
        # elements L and M and place them in the correct position at A[p..r]
        while i < len(L) and j < len(M):
            if L[i] < M[j]:
                array[k] = L[i]
                i += 1
            else:
                array[k] = M[j]
                j += 1
```

```
            k += 1

        # When we run out of elements in either L or M,
        # pick up the remaining elements and put in A[p..r]
        while i < len(L):
            array[k] = L[i]
            i += 1
            k += 1

        while j < len(M):
            array[k] = M[j]
            j += 1
            k += 1


# Print the array
def printList(array):
    for i in range(len(array)):
        print(array[i], end=" ")
    print()


# Driver program
n=np.arange(1,1001)
t1=[]
for i in range(1,1001):
    array=[]
    for i in range(100):
        array.append(np.random.randint(100))
    t1.append(time1(array))


#Curve Fitting function
def func(x, a, b):
 return a*(x**(b))
#Curve fitting function for linearthamtic time
def func2(x, a, b):
 return a *x* np.log(x ) + b
# Initial guess for the parameters
initialGuess = [0.1,0.1]
#x values for the fitted function
xFit=np.arange(1,1001)
#Plot experimental data points of execution time of quick sortig
#Perform the curve-fit
popt, pcov = curve_fit(func, n, t1)
```

```
plt.figure()
plt.plot(n, t1, label="Experimental time")
#Plot the fitted function
plt.plot(n, func2(n, *popt), '-r', label="Execution time")
plt.legend()
plt.xlabel('Size of an array')
plt.ylabel('Execution time')
plt.title('Execution time of merge sort')
plt.show()
```

(For Kruskal's Algorithm)

```
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import time
import numpy as np
def a_matrix(x):
    ''' generates a adjacancy matrix of size x'''
    g=np.zeros([x+1, x+1])
    g[:, 0]= np.arange(x+1)
    g[0, :]=np.arange(x+1)
    for i in range(1,x+1):
        for j in range(1,x+1):
            if i !=j:
                g[i, j]=g[j, i]=np.random.randint(1,10)
            if sum(sum(g[1:, 1:]))>=9000:
                break
        if sum(sum(g[1:,1:]))>=9000:
            break
    return g
def time1(v):
    a=[]
    g= Graph(len(v))
    for i in range(len(v)):
        for j in range (i,len(v)):
            c=v[i][j]
            if c!=0:
              g.add_edge(i, j, c)
    for i in range(5):
        start_time=time.time()
        g.kruskal_algo()
        a.append(time.time()-start_time)
    a=sum(a)/5
```

```
    return a

# Kruskal's algorithm in Python


class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    # Search function

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def apply_union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    #  Applying Kruskal algorithm
    def kruskal_algo(self):
        result = []
        i, e = 0, 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
```

```
            y = self.find(parent, v)
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.apply_union(parent, rank, x, y)
        for u, v, weight in result:
            print("%d - %d: %d" % (u, v, weight))


# Driver program
n=np.arange(1,101)
t1=[]
for i in range(1,101):
    f=a_matrix(i)
    t1.append(time1(f))


#Curve Fitting function
def func(x, a, b):
 return a*(x**(b))
#Curve fitting function for linearthamtic time
def func2(x, a, b):
 return a *x* np.log(x ) + b
# Initial guess for the parameters
initialGuess = [0.1,0.1]
#x values for the fitted function
xFit=np.arange(1,101)
#Plot experimental data points of execution time of quick sortig
#Perform the curve-fit
popt, pcov = curve_fit(func2, n, t1)
plt.figure()
plt.plot(n, t1, label="Experimental time")
#Plot the fitted function
plt.plot(n, func2(n, *popt), '-r', label="Execution time")
plt.legend()
plt.xlabel('Size of an array')
plt.ylabel('Execution time')
plt.title('Execution time of merge sort')
plt.show()
```