

Relazione Algoritmi e Strutture Dati

Francesco Mauro, matricola: 949471

A.A. 2022/2023

Contents

1	Mege Binary Insertion Sort	2
1.1	Mege Binary Insertion Sort implementation	2
1.2	Introduction	2
1.3	Testing methodology	2
1.3.1	Some graphs	3
1.4	Conclusion based on the analysis	4
2	Skip List	5
2.1	Introduction	5
2.2	Testing methodology	5
2.3	Conclusion based on the analysis	7
2.4	Knwon Issues	7
3	Priority Queue	8
3.1	Introduction	8
3.2	Implementation	8
4	Prim's Algorithm	9
4.1	Intdoduction	9
4.2	Supporting classes	9
4.2.1	Graph	9
4.2.2	Key	9
4.3	Algorithm Implementation	10
4.4	Conclusion	10

Chapter 1

Mege Binary Insertion Sort

1.1 Mege Binary Insertion Sort implementation

1.2 Introduction

The purpose of this report is to provide the tests that have been done to find the best value of K for the Merge Binary Insertion Sort algorithm. Considering that Merge Binary Insertion Sort is a hybrid algorithm that has ability to handle large input and for its speed, but it became inefficient when having a small input, in the case of small input the library switch to Binary Insertion sort, that is more efficient on small input.

1.3 Testing methodolgy

To test the algorithm, I used a bash script¹ that run the program with different values of K and for each field to sort, saving the time and the value of the algorithm parameter K.

¹All the test are done on a Lenovo Thinkpad x390 yoga with an Intel Core i7-8665U and 16 gb of ram

1.3.1 Some graphs

Figure 1.1: Time used to sort string field with different K values

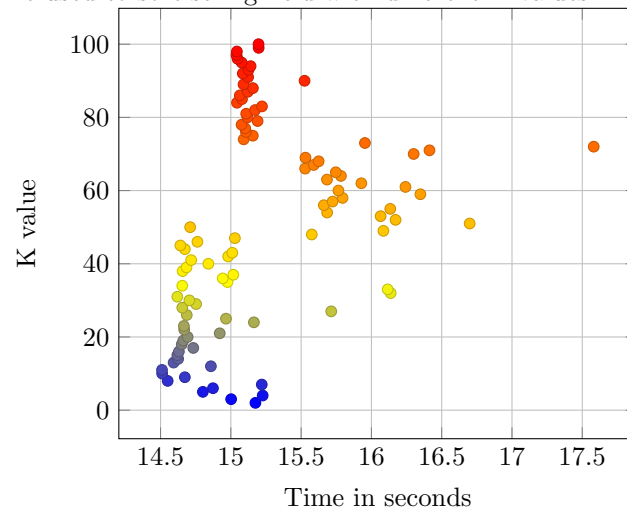


Figure 1.2: Time used to sort integer field with different K values

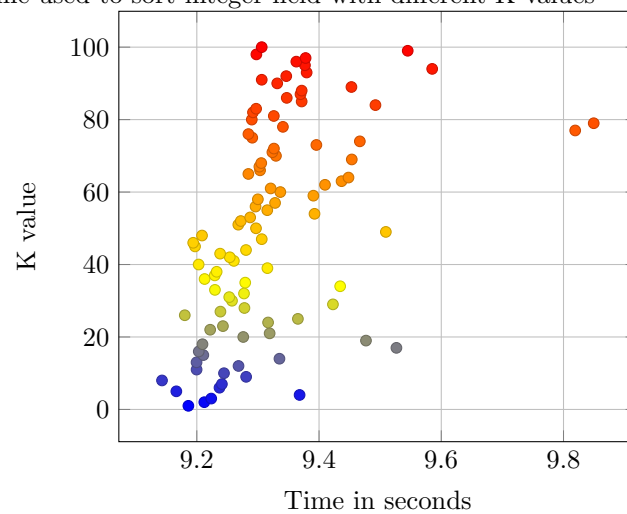
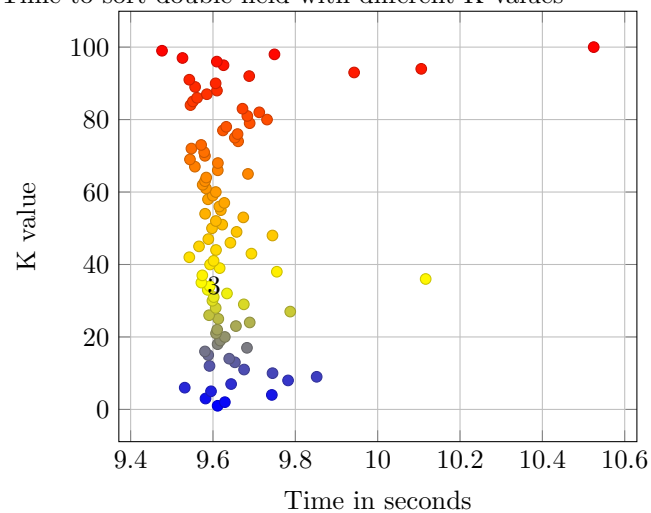


Figure 1.3: Time to sort double field with different K values



1.4 Conclusion based on the analysis

As we can see from the graphs, the optimal values for k is around 10 and 20, and the fact that based on the actual implementation is also important the type of field that the algorithm is sorting. In fact when sorting integer or double the algorithm is taking way less time than sorting string, probably caused by miss branch prediction. Also in the graph we can see some outliers, this can be caused by bottleneck of other processes or just the fact that the laptop in some point was charging.

Chapter 2

Skip List

2.1 Introduction

The purpose of this chapter is to report the tests that have been done to find the best value of height in a Skip List. Skip List is a probabilistic data structure that allows searching, insertion and deleting operation with time complexity of $O(\log n)$.

2.2 Testing methodology

I wrote a bash script¹ that runs the program with different values of height and saves the output in a file, the range of tested level is between 6 and 50

¹Included in the repo, the name is `time_taker_ex2.sh`

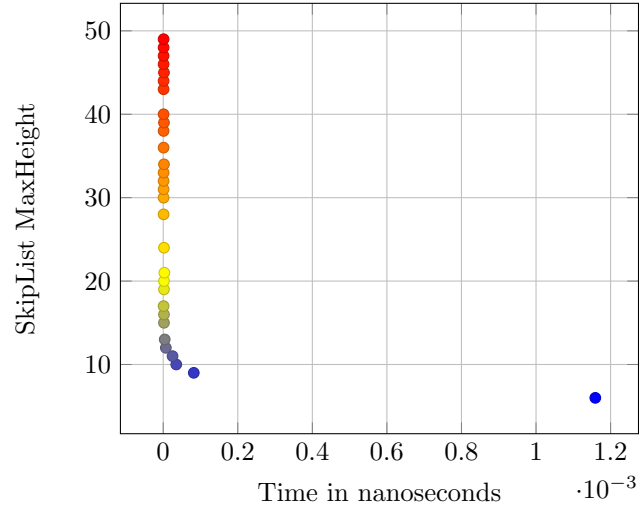


Figure 2.1: Graph 1: SkipList Max-Height vs. Time

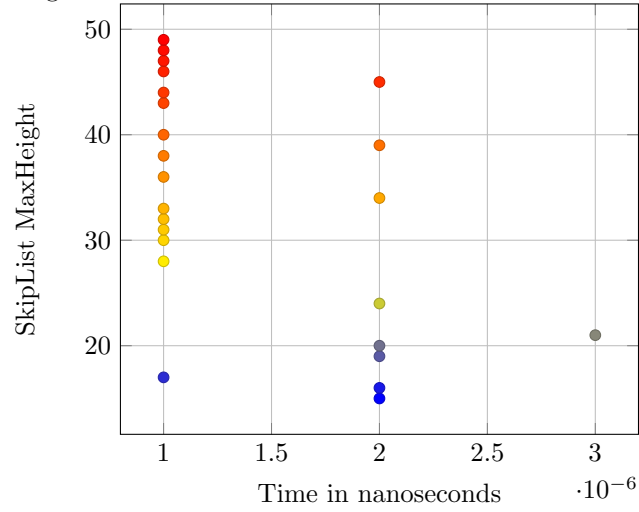


Figure 2.2: Graph 2: SkipList Max-Height vs. Time (Focused)

Range from 1 to 5 is omitted because the algorithm did not terminate for those values ²

²all test are done on a Lenovo Thinkpad x390 yoga with an Intel Core i7-8565U CPU and 16GB of RAM, with Arch Linux installed as only OS

2.3 Conclusion based on the analysis

Looking the graph the best values is between 15 and 20, because this range is the good balance between the time used to search the words in the skip list and the memory used.

2.4 Known Issues

Sometimes the main program gives a segmentation fault error caused by the **fgets()**, used to read the phrase to correct, on 50 run the program gave segmentation fault 7 times.

Chapter 3

Priority Queue

3.1 Introduction

The Priority Queue is a data structure that stores elements with associated priorities and allows retrieval of the highest priority element.

3.2 Implementation

The class *PriorityQueue* implements the generic interface *AbstractQueue*. From this interface the class inherits all the methods that allow to manage the queue.

Chapter 4

Prim's Algorithm

4.1 Introduction

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 starts from an arbitrary vertex and iteratively adds new vertices and edges to the tree. It terminates when all vertices are included in the tree.

4.2 Supporting classes

There are three classes used in this Implementation of Prim's algorithm:

- Graph
- Key
- PriorityQueue

In this report is possible to read about PriorityQueue in previous chapter, so we will focus on Graph and Key classes.

4.2.1 Graph

Graph class is utilized to represent a graph data structure. In the exercise is used to represent the input graph and the resulting MST. The implementation use an AdjacencyMap that stores vertices and edges.

4.2.2 Key

Key class is used to represent the information of vertex associated with the MST during the execution of Prim's algorithm. It contains the vertex identifier and,

the key value and parent vertex in the MST. The class key allows tracking of the construction process and determining the next node to be added to the MST. The Key object is stored in the PriorityQueue.

4.3 Algorithm Implementation

Prim's algorithm follows these main steps:

1. Read the input of the csv file passed as argument and create the graph from it.
2. Initialize the PriorityQueue with the Keys of the vertices of the graph.
3. Set the key of the starting vertex to 0 and populate the PriorityQueue.
4. Iteratively operate the following steps while the PriorityQueue is not empty:
 - (a) Extract the vertex with the minimum key from the PriorityQueue.
 - (b) Add this vertex to the MST.
 - (c) Update the key for all neighbors of the extracted vertex that are still in the PriorityQueue, so that are not already visited.
5. Calculate and print the total weight of the MST, number of vertices and edges created, and execution time.

4.4 Conclusion

All the implementation of the MST is around three main components that are the classes Graph, Key and PriorityQueue.