**Graphs and Spells: a study on graph algorithms**

**and spell checking data structures**

Siarhei Miachkou, Andrii Venher

April 28, 2023

1. **Introduction**

In this paper we approach the problems of spell checking: that is building a dictionary and identifying whether the words from the given file belong there. The final goal of this research is to find the best data structure by comparing the most popular options available.

We will investigate four of the well-known data structures used for spell checking: Linked List (Niave approach), Balanced Binary Search Tree, Trie (Prefix Tree) and a Hash Map. Concerning the Hash Map we will also compare three ways to resolve collisions: Chaining, Linear Probing and Double Hashing. Finally, our study also involves practical examples of Breadth-First Search and Depth-First Search algorithms that operate on graphs.

* 1. **Linked List (Naive approach)**

When facing a new challenge, it is a good practice to try and solve it in the easiest way first, to get aquinted with all the nuances of the given task. In case of spell checking, the easiest way to store the dictionary is a linked list. It does not require any calculations to insert or retreive data, it is scalable and no maintainance is needed.

Unfortunately, its simplicity is its only advantage. It is extremly slow, since you have to go throuh each node to find an existing word or to add a new one, there is no way around it. This means that the insertion complexity is always O(n) and data retrieval is O(1) in the best case but O(n) in the worst.

* 1. **Balanced Binary Search Tree**

So, Linked list is terribly slow, let’s arrange our nodes in a better way. A Binary Search Tree is a classic way of organizing data, however it has a major disadvantage. A sorted stream of input turns your Binary Tree into a Linked List, since all the elements go down one branch. As we are building a dictionary, we actually expect our input data to be sorted, thus we really need to improve the concept of BST. We would like to control the way our tree grows to keep it balanced. Every time a new node is created, we check if the updated branch is too long now (has at least 2 more elements that other branches) and if it is case we rearrange our nodes to balance the tree out, by pulling the node at the roout of the problematic subtree to the shorter branch side.

This is a tough datat structure to implement, since we have to check its condition every time we insert a new element, but the payoff is a guranteed O(logn) complexity for both insertion and retrieval.

* 1. **Trie (Prefix tree)**
  2. Prefix tree is a very interesting data structure, here we do not store the words, we store only letters, building the paths that form our words. Every node here has not one, not two but N (length of the alphabet in use) pointers. And going through the list of our words we create a node anywhere we go following the ASCII codes inside of our tree. In the end we mark the node we finished at as the end of the word, to complete our journey. Now, when we want to check whether a given word is present in our dictionary, we just go down the same path and if we stop at the node that is flagged as the final one – the word belongs to our dictionary, if we stumble upon a null pointer or finish at the node that is not final – the word we were searching for is not present in the data structure.

The time complexity of this is O(m) for both insertion and reading which is wonderful, however the bad thing is the memory usage, the Trie requires O(N \* m \* n) which can get really big really fast.

* 1. **Hash Map**

Searching for words by checking all the nodes on your way is not very efficient. It would be so much faster if we could just go there immediately. And apparanetly we can do this. We allocate the array of >n elements and assing an index to every word of ours to save it there. To convert a string to the integer we use a Hash Function. Unfortunately, we are not guaranteed to have a new, unique index every time we hash a word, thus we need to manage the collisions. We will discuss three of the possible solutions later.

It is very difficult to determine the complexity of the hash map insertion or retrieval as it depends on many things such as its load factor, and collision resolivng method and hash funtion in use. In our case, both writing and reading words can be done in O(m) at best or O(n + m) time at worst.

* 1. **Triwizard Tournament (BFS algorithm)**

First we initialize a queue for BFS, a hashmap to store the shortest path to the exit for each wizard and a hashmap to track cells visited by each wizard. Then we enqueue the initial positions of the three wizards in the BFS queue, mark them as visited. While the BFS queue is not empty, dequeue a position and check if it is the exit. If it is, store the shortest path to the exit in the hashmap for the wizard who reached it. Otherwise, enqueue all adjacent positions that are not walls and have not been visited before. Mark enqueued cells as visited. After BFS is completed, calculate the time it would take each wizard to reach the exit based on their speed and the length of the shortest path (length / speed). Finally, we compare the calculated times to determine which wizard will reach the exit first.

1. **Aunt’s Namesday (DFS)**

First we initialize a hashmap which represents a graph to map every guest to the list of guests which he/she dislikes. Check if the graph is valid. Create a hashmap to store each guest's table number and a hashmap to store visited guests. Then, we assign all guests to the first table by default and mark them as not visited. Next, we use a recursive DFS algorithm to traverse the graph and assign guests to tables, making sure that no two guests who dislike each other end up at the same table. If a guest can be assigned to either table, arbitrarily choose one. Mark each visited guest and finally we continue previous step while there are not visited nodes.

1. **Full house (Hash Maps)**

As we mentioned before, we will investigate three ways of resolving hash values collisions: Chaining, Linear probing and Double hashing. It is important to understand the differences when choosing the one you are going to use.

* 1. **Chaining**

A separate chaining hash map is probably the easiest type of hash map to implement. Inside it has an array which items are singly linked lists. When we need to insert and element, we hash the key and get the index in the array where this element should be located. If it happens that at the index there are no other elements, then the element is just placed at the index. Otherwise, it is added to the end of a linked list which already exists at the index. If we need to retrieve the element, we hash its key and traverse the list comaring the keys until we find a match. Therefore, all operations are linearly dependant on the length of a underlying linked list. So it is a good idea to choose a well-defined hash function so it does not produce a lot of collisions, otherwise complexity could start reaching O(n).

* 1. **Linear Probing**

A linear probing is one of the many open addresing solutions. The idea is that you don’t have to store your word strictly at its hashed index, if needed you can make some shift and store it at a different place. The shift value used is the key difference between all of the open addressing solutions. Linear probing is the simplest possible one. If the spot has already been occupied we just take the next free one by linearly going through our array. We do the same when we retrieve the value by checking the contents of every node we are getting transferred to.

**3.3 Double Hashing**

Double Hashing works just the same way. The only difference is that we don’t search for the next free Node as it is to likely to get into a long line of occupied nodes inside of our array. Instead, we calculate our shift by using another hash function, hence the name Double hashing. This approach is capable making the search for the free spots faster.

1. **Methodology**

All data structures were implemented in C++. Spell checking algorithms were tested on text and dictionary of size 1000, 1100, 1200, ... 10000 characters. The text was generated by taking random words of the dictionary file, and adding an extra ‘z’ to 25% of them to simulate typos. Each dictionary size was tested 10 times. The result for each size is the average time it took to find all the wrong spelled in the text using a given data structure. For Hash maps the time of insertion at load factor from 0.001 to 1.0 was measured.

1. **Results**

As expected, the Linked List approach turned out to be devastatingly slow (Figure 1). The rest of the data structures have arranged themselves the following way: Balanced Binary Search Tree is the slowest solution, then goes Hash Map and Trie is the fastest data structure (Figure 2). Regarding the Hash Maps (Figure 3): Chaining is unpredictable and has random spikes every time the collision happens (Figure 4), open addressing algorithms don’t have this problem but start to struggle once Load Factor gets closer to 1.0. In addition, Double hashing method managed to solve the collisions a little bit faster than linear probing method (Figures 5 and 6).

1. **Conclusions**

To sum up this research, Linked list is too slow for large dictionaries and while BBST is much faster it still loses to Hash Map and Trie. Thus, the choice is between these two data structures. If you have a lot of memory and are not afraid to occupy it – go with the Trie, it is extremely fast. However, if you have any memory restrictions, Hash Map with Double Hashing collision resolving method is the best solution for you.

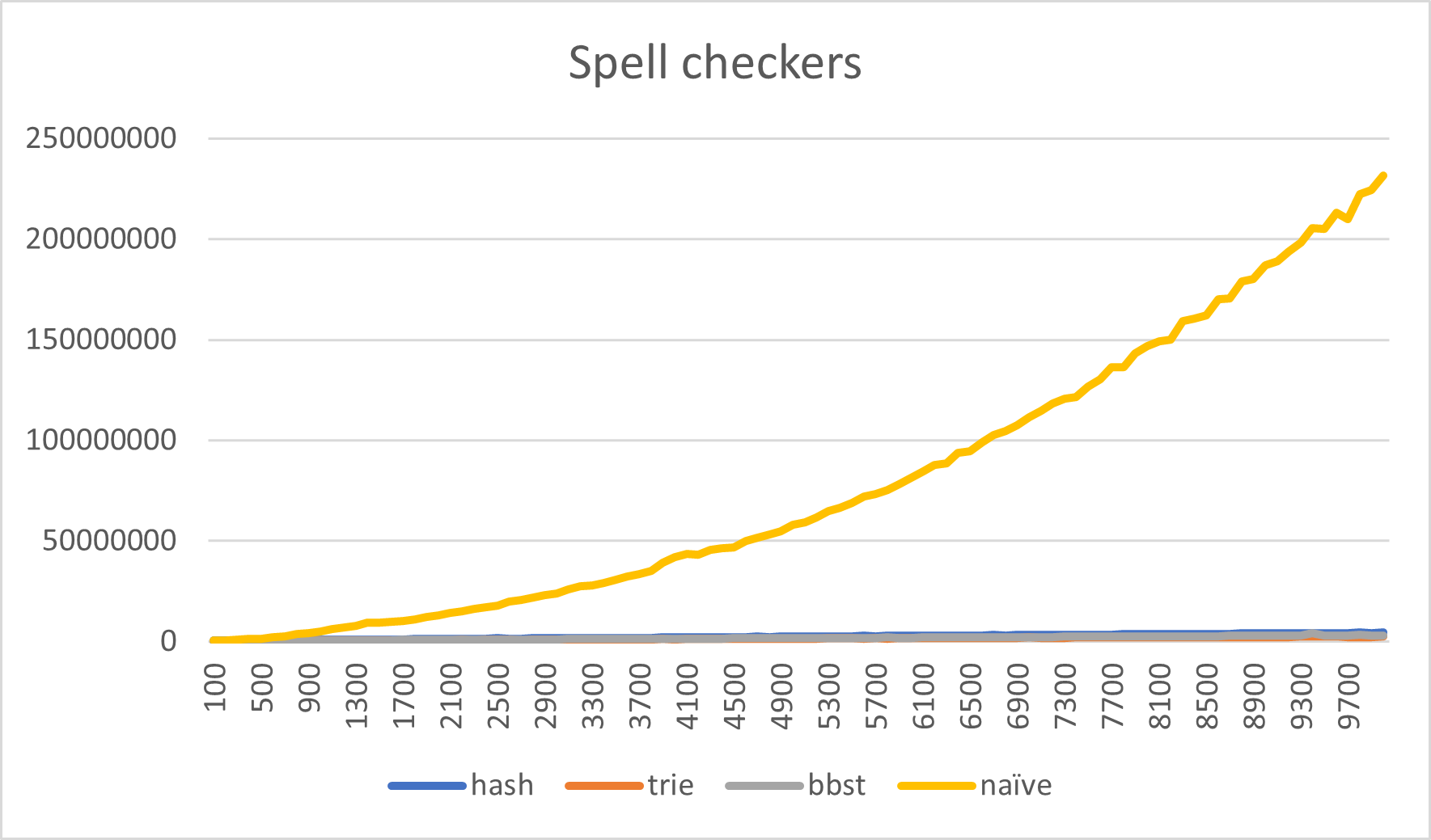


Figure 1: Spell checking data structures comparison

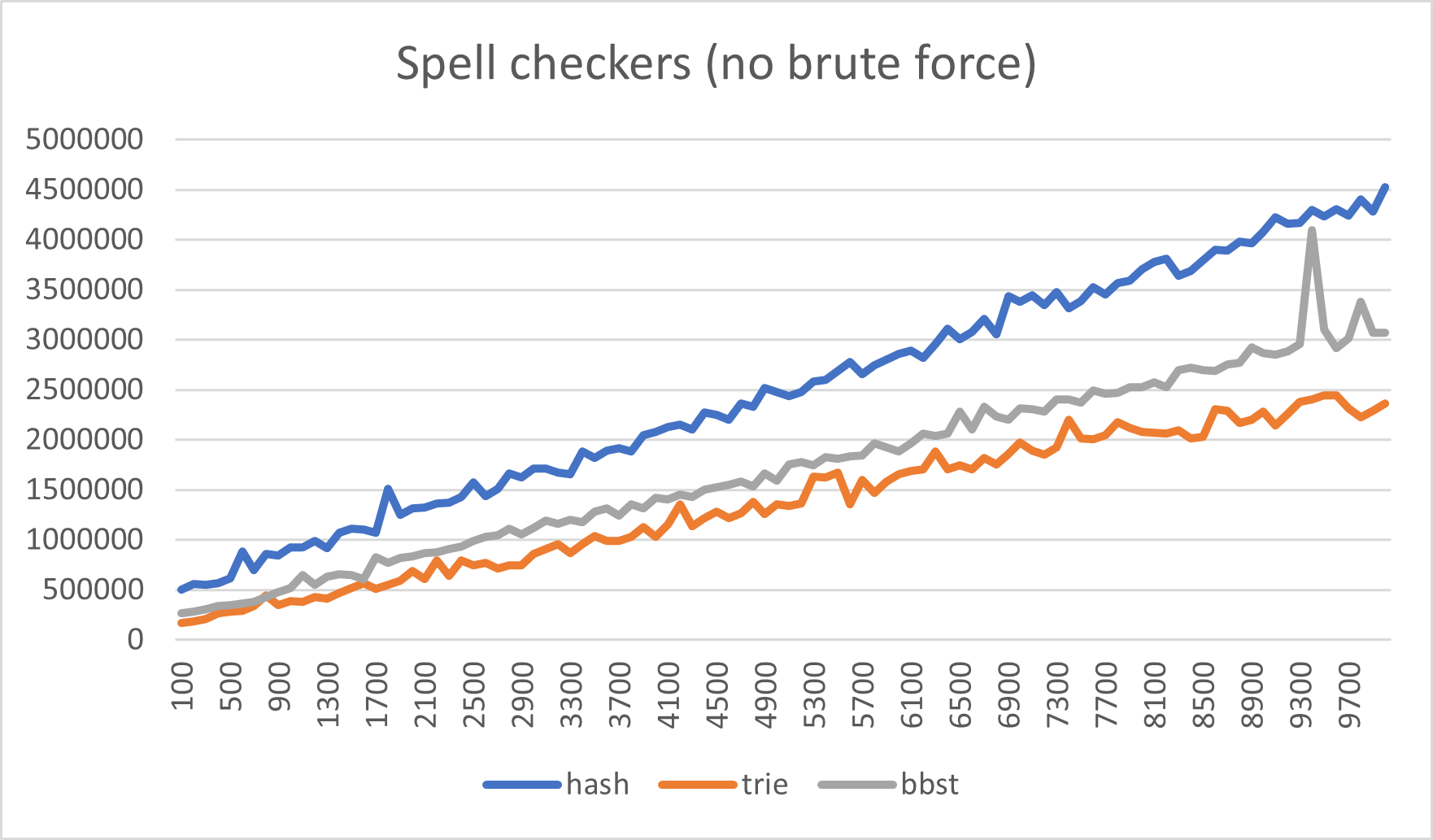


Figure 2: Spell checking data structures (no Linked List)

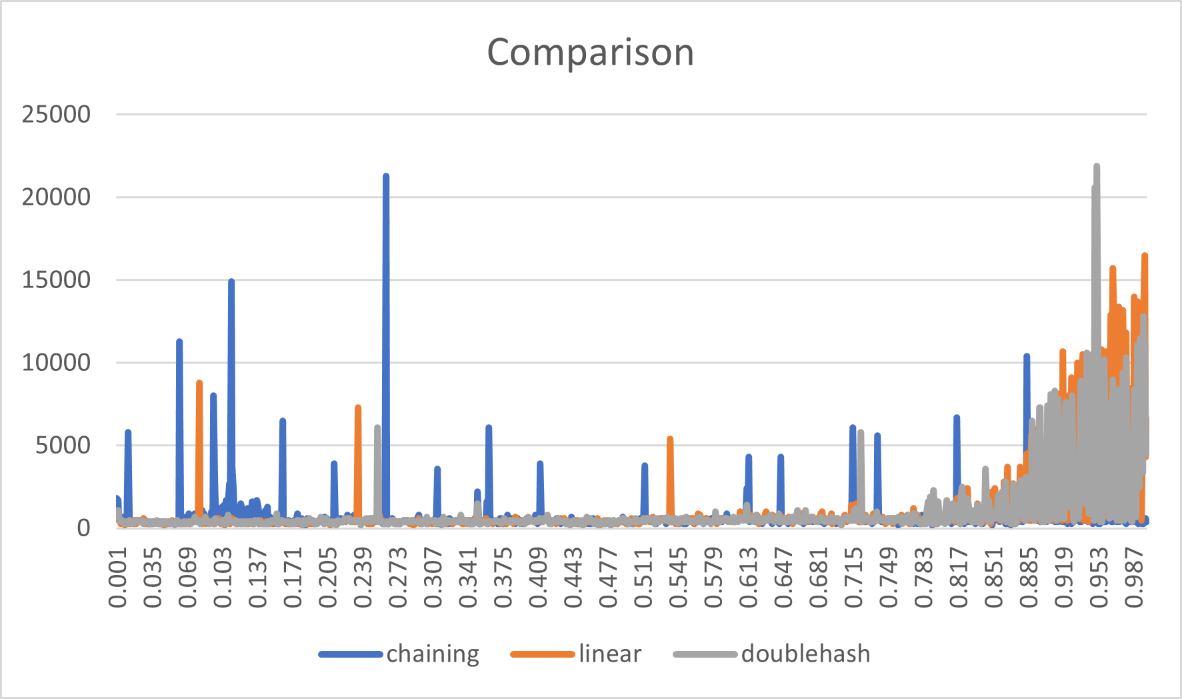


Figure 3: Hash Map collision resolving methods comparison

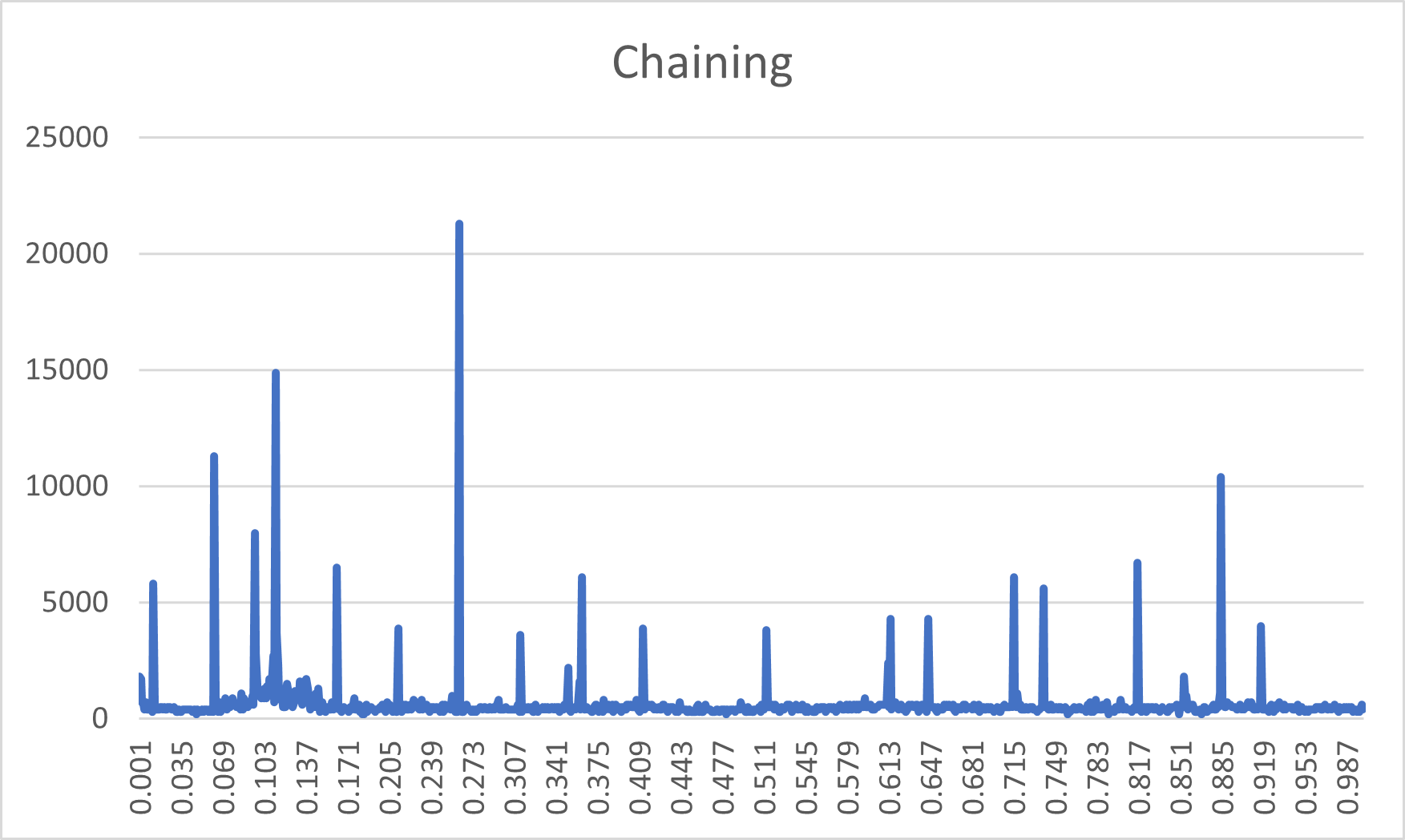


Figure 4: Chaining collision resolving method

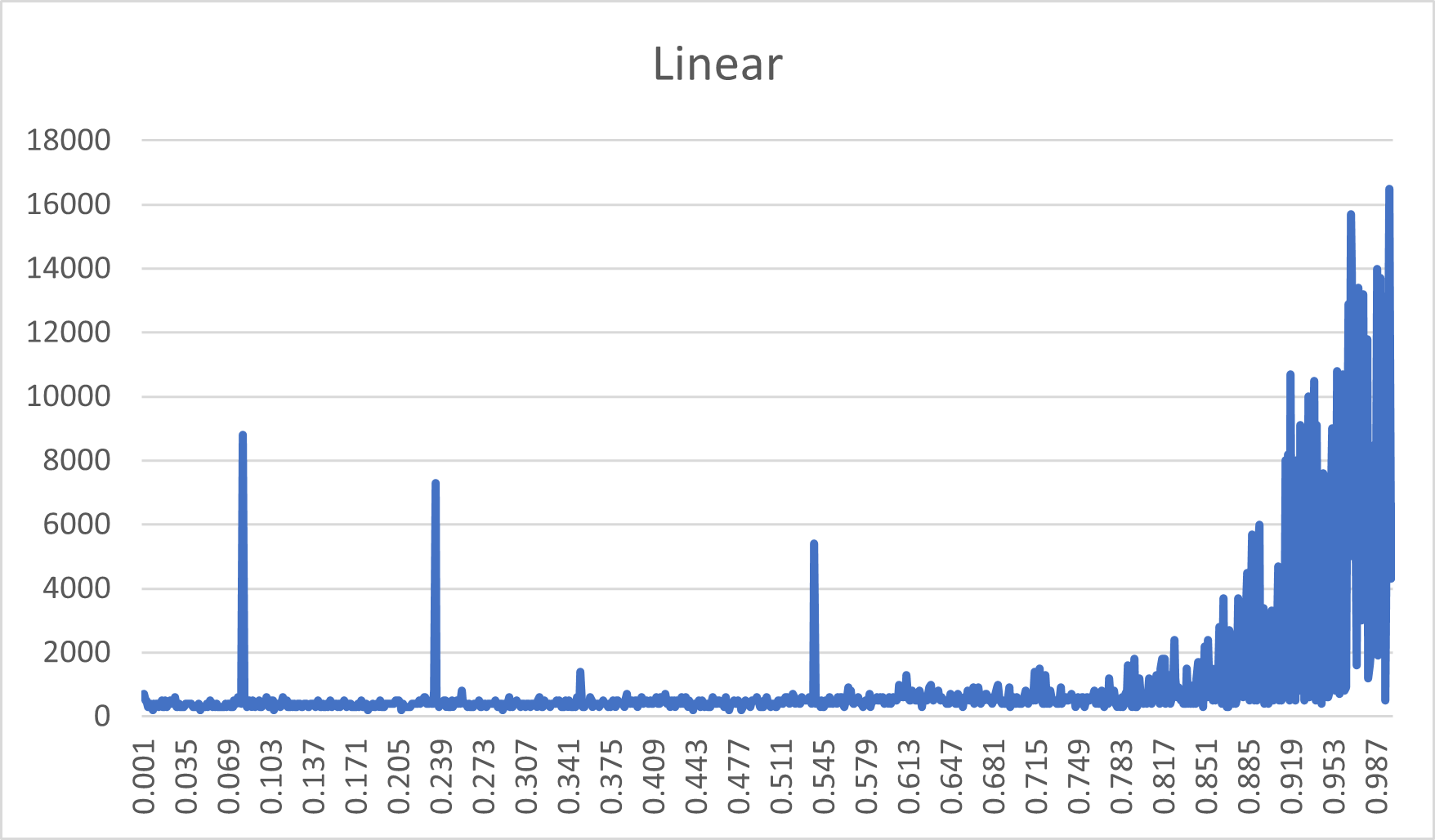


Figure 5: Linear Probing collision resolving method

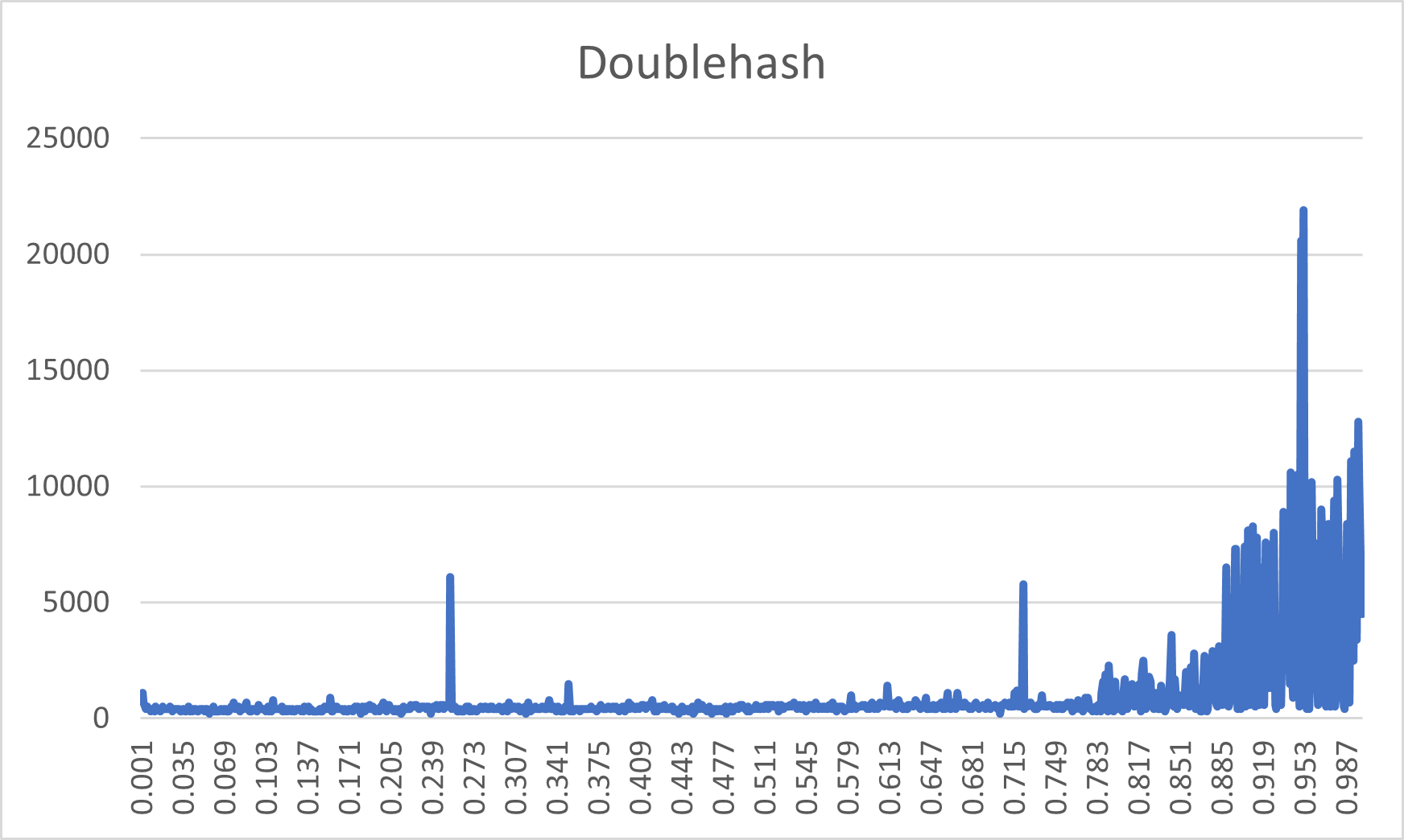


Figure 6: Double Hashing collision resolving method