BLG336E - Analysis of Algorithms II

Enes Demirağ - 504201571

Spring 2021, Homework 1

1.a. Desing & Implementation Process

First of all, I designed a Node structure. While deciding the attributes and methods of this class I tried to make it lightweight in order to end up with a memory efficient program. Every node has 4 private attibutes. A char called key which holds the assigned letter to the node. An integer value which holds the numberic value of the node. A pointer to its parent node, and a list of pointers to its child nodes. With these attributes, I can go from any node to another node easily. Other than constructor and get/set methods, Node class has a method called addChildren(). This method gets a char and a list of integers and creates new nodes under itself. This medhod used while creating a tree. Also there are two functions closely related with this Node class. checkZeroCondition() and addNewLayer(). Former one used for the zero constrain of the first letters of the inputs and the latter was used to create tree. It is a recursive funstion which called itself again anad again until it creates every possible combination branch and reaches the end.

Various utility functions are designed and implemented in this project. findSolution() function takes a leaf node and goes back to the top root node while gathering every letter-number pairs and gives a solution. In order to check if this solution is corrects, there is another function called checkSolution() which task user inputs strings and maps them to integers and check if the sum operation holds. checkArguments(), getSystemTime(), and other utility function for printing/writing etc. is implemented and can be seen under project files.

For the Breadth-First Search and Depth-First Search implementations, I used queue and stack abstract data structures. My implementation of them can be seen under project files. Inside them, I explained my decisions and way of thinking in depth via inline documentation and comments. BFS function get a node, in our case its the root node of the precreated tree, and starts traversing the tree. If it comes a leaf node, it raises a possible solution and checks it with constrains. If the solution was false, it continues until is founds a valid letter-number pairs solution for the given inputs. After it finds a solution, it stops searching and returns the solution while setting required variables such as visited node number, maximum node in memory, and running time of the search process.

Inside main.cpp, firstly, we take inputs arguments and check them. Then a node object with no parent created as a root for the tree. Then with the help of resursive addNewLayer() function, tree will be created while checking constrains. Checking the constrains and creating a tree with relatively less nodes in it was not obligatory. But to make the program much more memory efficient, I implemented and look for those constrains too. Also it is possible to speed up the both tree creation and search processes as well. Actually in tree creation process I checked some constrains multiple times which was not a good thing when time is essential. But since optimization is not required, I wanted to keep them to show the different approaches which can be used. Finally, after gathering valid letter-number pairs, program will print required information about search process and creates a new file and writes the solution matrix.

1.b. Pseudo Code

Tree Creation: Recursive function pseudo-code. After creating a root node, this function will start. **(1.c.)** Time complexity of this algorithm is $O(10^n)$, because for every new unique letter there should be 10 times more nodes created.

```
Get the list of letters
Get the node
if there is no unique letter left:
    Finish
Get the first letter from the list.
Create a list of all numbers
if letter is a first letter of one of the input strings:
    Remove 0 from the possible numbers list.
Start from the given node and go back to the root node.
for every node in the way:
    Remove the number of node from the possible numbers list.
Create new nodes for all values in possible numbers list with same letter.
Connect these new created nodes to the given node.
Remove letter from unique letters list.
for every new created node:
    call this function again with new unique letters list.
```

Breadth-First Search: This function starts from the root of the tree and traverses every node while trying to find a valid solution.

```
Create an empty queue.
Put the starting node at the back of the queue.
while queue is not empty:
    Take a node from the front of the queue.
    if its a leaf node:
        Gather letter-number pairs and check if valid.
        if the letter-number pairs are valid:
            Stop searching and return the solution.
        else:
            Continue searching.
    Put every child node of the corresponding node to the back of the queue.
```

Depth-First Search: This function starts from the root of the tree and traverses every node while trying to find a valid solution.

```
Create an empty stack.
Put the starting node at the top of the stack.
while stack is not empty:
    Take a node from the top of the stack.
    if its a leaf node:
        Gather letter-number pairs and check if valid.
        if the letter-number pairs are valid:
            Stop searching and return the solution.
        else:
            Continue searching.
    Put every child node of the corresponding node to the top of the stack.
```

2. Analysis & Comparison Of the Algorithms

In the figure 1, you can see the comparison of two algorithms in terms of number of visited nodes, the maximum number of nodes kept in the memory, and running time. Because in order to achieve a fully finished solution search should reach at least one leaf node. **(2.a.)** Because of the BFS traversal approach, it will look every node in upper stages of the tree before coming to the bottom level. On the other hand, DFS easily reaches to the leaf nodes because of its traversal way.

```
enes@demirag:~/Projects/cryptarithmetic-puzzles$ g++ -std=c++11 -Iinclude -c src/*
enes@demirag:~/Projects/cryptarithmetic-puzzles$ g++ -std=c++11 -Iinclude -g *.o -o runme main.cpp
enes@demirag:~/Projects/cryptarithmetic-puzzles$ ./runme BFS SEND MORE MONEY out1.txt
Algorithm: BFS
Number of the visited nodes: 2027957
Maximum number of nodes kept in the memory: 1451520
Running time: 73.678 seconds
Solution: S: 9, E: 5, N: 6, D: 7, M: 1, O: 0, R: 8, Y: 2
enes@demirag:~/Projects/cryptarithmetic-puzzles$ ./runme DFS SEND MORE MONEY out2.txt
Algorithm: DFS
Number of the visited nodes: 83109
Maximum number of nodes kept in the memory: 43
Running time: 2.93 seconds
Solution: S: 9, E: 5, N: 6, D: 7, M: 1, O: 0, R: 8, Y: 2
enes@demirag:~/Projects/cryptarithmetic-puzzles$
```

Figure 1: Example of BFS and DFS

Also I want to mention that, while tree generation and both algorithms works perfectly with all examples, because of the memory and process timeout limitations of the ITU server, BFS algorithm was killed after 300seconds for some cases when unique letter number is big. Output of my program compiled and runned on the ITU server can be seen in figure 2.

```
[demirag16@ssh cryptarithmetic-puzzles]$ ./main BFS TWO TWO FOUR output.txt
Algorithm: BFS
Number of the visited nodes: 114812
Maximum number of nodes kept in the memory: 120960
Running time: 93.589 seconds
Solution: T: 7, W: 3, O: 4, F: 1, U: 6, R: 8
[demirag16@ssh cryptarithmetic-puzzles]$ ./main DFS TWO TWO FOUR output2.txt
Algorithm: DFS
Number of the visited nodes: 9302
Maximum number of nodes kept in the memory: 38
Running time: 0.175 seconds
Solution: T: 9, W: 3, O: 8, F: 1, U: 7, R: 6
[demirag16@ssh cryptarithmetic-puzzles]$ ./main BFS SEND MORE MONEY output3.txt
Killed
[demirag16@ssh cryptarithmetic-puzzles]$ ./main DFS SEND MORE MONEY output4.txt
Algorithm: DFS
Number of the visited nodes: 83109
Maximum number of nodes kept in the memory: 43
Running time: 1.562 seconds
Solution: S: 9, E: 5, N: 6, D: 7, M: 1, O: 0, R: 8, Y: 2
```

Figure 2: Results on ITU Server

(3.) We should maintain a list of every discovered node in order to prevent searching the same node again and again. In my application, while traversing with both BFS and DFS algorithms, I removed every node I discovered from the queue/stack. For the DFS implementation, instead of a stack, if I used a recursive structure, it was possible to search the same node multiple times, so we need to maintain a list of every discovered node.