BLG 336E – Analysis of Algorithms II

Homework 1

Breadth-First Search (BFS) and Depth-First Search (DFS)

150170092 : Barış İncesu

4 APRIL 2020

Part 2. Report

2.1.a

We were expected to solve the homework with Cryptarithmetic Puzzles. First of all, we got the puzzle from the command line. And we have created a word from distinct letters in order to use it meaningfully below.

4 kinds of values are stored in the Node. These are the map * pointer that reaches the map structure that stores possible solutions, variables that indicate the letter and number that distinguish the node, and pointers that provide access to child nodes.

```
class Node
{
private:
public:
    map<char, int> *key_value;
    int a;
    char c;
    Node *C[DIGIT];
    Node()
    {
        for (int i = 0; i < DIGIT; i++)
        {
            C[i] = NULL;
        }
    };
    ~Node(){};
};
```

The structure of the Tree is as follows. The **root** that enables us to access the tree is initially assigned as **NULL**. **DepthSearchFirst** and **BreathSearchFirst** methods execute the search with algorithms specified on the tree. In these methods, they call another function called Solution and test each value, the node to be tested, the arguments for Cryptarithmetic puzzles and the file name to write the outputs are sent as parameters to this function.

```
class Tree
{
private:
public:
    Node *root;
    Tree() { root = NULL; };
    ~Tree(){};

    Node *getRoot() { return root; };
    void DepthFirstSearch(Node *, string, string, string, string, string);
    void BreathFirstSearch(Node *, string, string, string, string, string);
};
```

Since the number of layers and nodes of the tree is variable, I thought it was laborious to create it using recursive or loops. Instead, I took power in the Stack structure we saw in the data structures design. For this construction, which I will explain in detail in the pseudo-code section later, it is necessary to define an empty node that will be at the top of our tree. The map inside this node is empty, I assigned the value **-** as a letter, and **-1** as a number. Later, I created this node and gave the order to build a tree by pushing it to the stack. Afterwards, each node will be created by inheriting the next node and passing certain conditions, such as same numbers 989, and will be connected to its parent's with pointers. The tree will be completed when the nodes whose necessary connections have been completed pop up on the Stack.

```
Tree t1;
    stack<Node *> s;

    Node *emptyNode = new Node;
    emptyNode->key_value = new map<char, int>;
    emptyNode->a = DIGIT + 1;
    emptyNode->c = '-';

    if (!t1.getRoot())
    {
        t1.root = emptyNode;
    }
    int counterMaxMemory = 1;
    s.push(emptyNode);

    // Creating tree structure
    while (!s.empty())
    {
        Node *current = s.top();
        s.pop();

        int index = puzzle.find(current->c) + 1;
        if (index < puzzle.length())
        {
            char password = puzzle[index];

            for (int i = 0; i < DIGIT; i++)
            {
                bool matching = false;
                for (auto it = current->key_value->begin(); it != current-
>key_value->end(); ++it)
                    if (it->second == i)
                    {
                        matching = true;
                    }

                if (!matching)
                {

                    current->C[i] = new Node;
                    current->C[i]->key_value = new map<char, int>;

                    map<char, int>::iterator itr; // copy map root to chield
```

```
                    for (itr = current->key_value->begin(); itr != current-
>key_value->end(); ++itr)
                    {
                        current->C[i]->key_value->insert(pair<char,
int>(itr->first, itr->second));
                    }

                    current->C[i]->a = i;
                    current->C[i]->c = password;
                    current->C[i]->key_value->insert(pair<char,
int>(password, i));
                    counterMaxMemory++;

                    if (password != puzzle[puzzle.length() - 1])
                        s.push(current->C[i]);
                }
            }
        }
    }
```

2.1.b.

```
// Tree Construction Pseudo-code on TWO TWO FOUR example and 10 DIGIT numeral.

We know all layers represent by 1 letter. Such as root "-".

Create root.                                                    // "-"
Create root's 1.generation leafs. T0,T1,T2,T3…,T9               // "F"
Create root's 2.generation leafs. W1,W2,W3…,W9                  // "O"
Create root's 3.generation leafs. R2,R3…,R9                     // "R"
Create root's 4.generation leafs.                               // "T"
Create root's 5.generation leafs.                               // "U"
Create root's 6.generation leafs.                               // "W"

We can see our layer number is on puzzle distinct letters. // puzzle="FORTUW"

When we create node we should consider its children's too. So we need memory
unite for remembering which nodes not completed.     //stack s

When all nodes and its children completed pop it from memory. // pop()
It means no uncompleted node remain.                 // s.empty()


So we should do again and again until uncompleted nodes finish.
!NOTE: We should different numbers of children. T0 node will not have W0 R0
F0 child.

While(s.empty)
        Current = the node to create children
        Pop(Current)
        Control children number is same parents?
                Create children with genetics
                Push children to memory until last letter of puzzle.
```

```
//DFS Pseudo-code
Like as a tree construction same rule will be valid.
When you search tree, we should look all nodes. So we should keeps nodes
in memory unit.                                          //memoryStack
We should put(push) bottom our scans here, and we have to remove on top
(pop) what we scanned from here.
When there are no more nodes to be scanned, that is, when we look at each
node, the algorithm will be complete.

While(memoryStack.empty)
        Current = the node to check solution
        Pop(Current)
        Solution(Current)
        If there is no solution, send your children to the stack, if any.
```

```
//BFS Pseudo-code
Like as a DFS same rule will be valid. But this time we should roots
before childrens. So I selected queue structure.
When you search tree, we should look all nodes. So we should keeps nodes
in memory unit.                                          //memoryQueue
We should put(push) to back our scans here, and we have to remove on
front (pop) what we scanned from here.
When there are no more nodes to be scanned, that is, when we look at each
node, the algorithm will be complete.

While(memoryQueue.empty)
        Current = the node to check solution
        Pop(Current)
        Solution(Current)
        If there is no solution, send your children to the queue, if any.
```

2.1.c.

The time complexity of **DFS** if the entire tree is traversed is ***Big O Notation Θ(n)*** where n is the number of nodes.

The time complexity of **BFS** if the entire tree is traversed is ***Big O Notation Θ(n)*** where n is number of nodes.

2.2

Table 1: BFS – DFS Compare on Cryptarithmetic Puzzles

Search Algorithm	the number of visited nodes	the maximum number of nodes kept in the memory	The running time	Puzzle
DFS	26751	187301	0.18 s	TWO+TWO=FOUR
	1975115	2606501	16.60 s	SEND+MORE=MONEY
	171514	187301	1.20 s	DOWN+WWW=ERROR
BFS	57694	187301	0.38 s	TWO+TWO=FOUR
	2166987	2606501	18.83 s	SEND+MORE=MONEY
	174554	187301	1.25 s	DOWN+WWW=ERROR

Since the trees on which we run the algorithms are the same, the number of nodes in memory will be the same. The reason BFS takes longer to find, with DFS, is priority aware. DFS first examines the leaf nodes, that is, it goes down to the bottom in a short time. BFS, on the other hand, takes time to reach the lowest tier where the solutions are found and the right node because it progresses layer by layer. **BFS has to visit more nodes** to reach the result.

2.3

The reason why we see the nodes we pass over in a structure is that we do not want to make extra effort to browse the children of this node again. It also offers a more understandable and systematic solution for the search function to work systematically.