# BLG336E - Analysis of Algorithms II, Spring 2021 Homework 1

**Solving Cryptarithmetic Puzzles with Breadth-First Search & Depth-First Search**

**Student Name: Bahadır LÜLECİ**
**Student Number: 504201511**

**Date: 06/04/2021**

IMPORTANT NOTE: When executing the code,the -std=c++11 flag must be used.

## 1) Problem Fomulation:

### a. Describing node and assignment representations in detail

This section will explain in detail how the node structure is created.

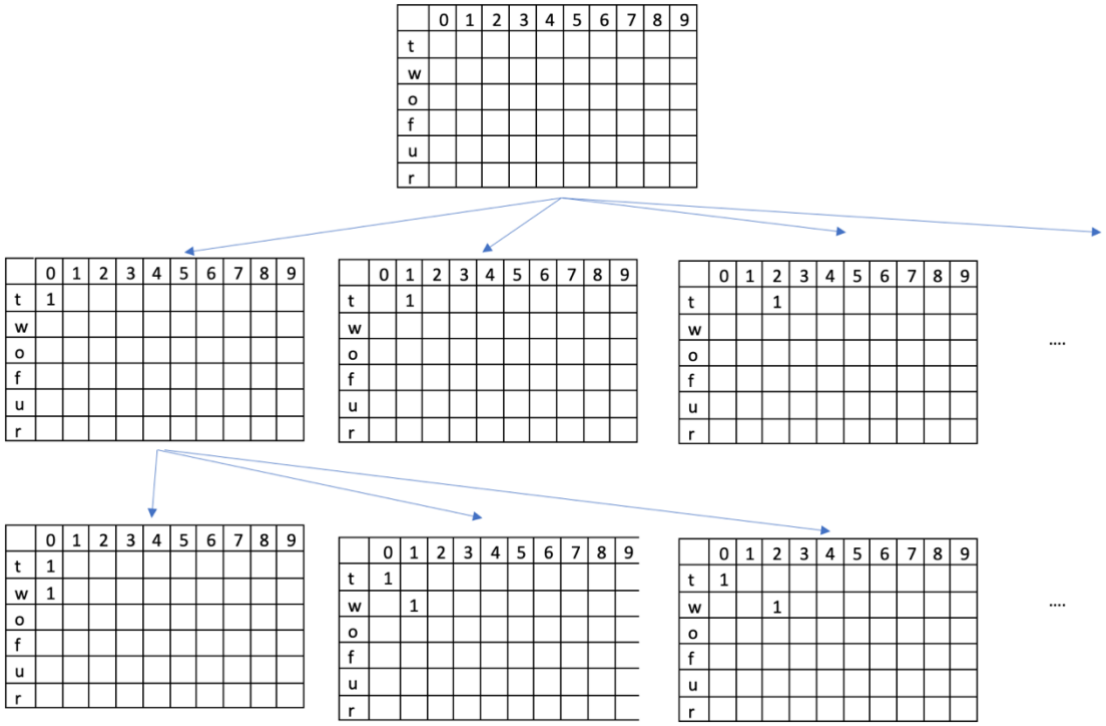


*Figure 1 Node Representation of Problem*

In Figure 1, each matrix will correspound to our node struct. It can be observed that a nod has one or more children. Therefore, the generic tree, namely the n-ary tree structure, has been deemed appropriate for this problem solution.

Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes. Every node stores address of its children and the very first node's address will be stored in a separate pointer called root. The Generic trees are the N-ary trees which have the following properties:

1. Many children at every node.
2. The number of nodes for each node is not known in advance.

```
struct Node
  {
     vector<pair<char, int > > key;
     bool visited;
     vector<Node*> child;
  };
```

**In our Struct Node:**

- vector <pair <char, int>> key corresponds to our matrix. Char and int values form a pair that connects with each other. Instead of Matrix complexity, this solution makes the code easy to read.
- The variable bool visited is placed inside the Node so that we have information about the nodes visited when using DFS and BFS searching algorithms.
- Since each node can have more than one child, the child variable of type vector <Node *> has been put in Node.

## b. Pseudo-Code

```
Class Tree
   struct Node
      key
      visited
      vector<Node > child
  endstruct

   struct root;
   public procedure newNode(key)
   public procedure availableNumbers(parentNodeData_key)
   public procedure createKeyVector(vector  letters, vector values)
   public procedure displayData(key)
   public procedure buildTree(struct root )
   public procedure DFS(struct root)
   public procedure BFS(struct root)
   public procedure solution_finder(parentNodeData)
   public procedure get_val(NodeData)
endclass

Function newNode
Pass In: Key(it contains letters and letter's values)

Temp = new Node
Temp's key = Key

Pass Out: Struct Node
Endfunction

Function availableNumbers
Pass In: parentNodeData(it contains letters and letter's values)

Vector int retvect

Flag = false
For(i =0 to i<10)
        For(j =0 to j<parentNodeData.SIZE )
                If(i == parentNodeData(j)'s letter's values(int values) )
                        Flag= true
                If(i ==0 AND  (count of parentNodeData(j)'s letter's (chars)> 0 ))
```

```
                                Flag= true
        If( flag == false)
                Retvect.push(i)

Pass Out: vector int retvect
Endfunction

Function createKeyVector
Pass In: letters(vector char) , values(vector int)

For(i=0 to letters.SIZE)
        Make pair of letters(i) and values(i)

Pass Out: Pair of letters and values
Endfunction

Function displayData
Pass In: key (vector pair of character and character's value)

For (iterate key)
        Print(key)
Endfunction

Function buildTree
Pass In: Node root
  nextChar = 0
  for (iterating root->key) {
    if (root->key.second == -1) {
      goto keepMoving;
    }
    nextChar increase by 1;

  }
  Return

  keepMoving:
  root->visited = false

    availNum = availableNumbers(root->key)

    for ( i = 0 to i < availNum.size()) {
        childData = root->key
        childData[nextChar].second = availNum[i]
        (root->child).pushback(newNode(childData))
    }

    for (int i = 0 to i < root->child.size()) {
      globalCounter increase by 1
      buildTree(root->child[i])
    }
  return
Endfunction

Function DFS
Pass In: Node root

  a= Pair(char, int)
```

```
    stack s
    s.push(root);
    while (s.empty == false ) {
        top = s.top();
        s.pop()
        globalDFSCounter increase by 1
        if(solution_finder(top->key)){
            a = top->key;
            return a
        }
        if(top->visited == false){
            top->visited = true
        }
        for (i = 0; to  i< top->child.size() ){
            if((top->child[i]->visited) == false)
                s.push(top->child[i])
        }
    }
    return a

Pass Out: vector pair key
Endfucntion


Function BFS
Pass In: Node root

A = Pair(char i int)
    Queue  q;
    root->visited = true;
    q.push(root);
    while(q.empty() == false)
    {
        currentNode = q.front()
        q.pop()
        globalBFSCounter increase by 1
        if(solution_finder(currentNode->key)){
            a = currentNode->key
            return a
        }

        currentChildren = currentNode->child;
        for (i = 0 to i < currentChildren.size() ) {

            if(currentChildren[i]->visited == false){
                currentChildren[i]->visited = true
                q.push(currentChildren[i])
            }
        }

    }
    return a
Pass Out: vector pair key
Endfuction


Function get_val
```

```
Pass In: NodeData (vector pair char,int)
   int val = -1
   for(int i=0 to  i<NodeData.size()){
      if(x==NodeData[i].first)
         val = NodeData[i].second;

   return val
Endfunction


Function solution_finder
Pass In: NodeData (vector pair char,int)
   input_one_local,  input_two_local, output_local = 0

   if(get_val(input_one[0], NodeData)== 0)
      return false
   if(get_val(input_two[0], NodeData)== 0)
      return false
   if(get_val(output[0], NodeData)== 0)
      return false

   for (i=0 to i<input_one.size(); i++){
      if(get_val(input_one[i], NodeData)== -1)
         return false;
      input_one_local += get_val(input_one[i], NodeData)*10^i(nput_one.size()-i-1))
   }
   for ( i=0 to i<input_two.size(); i++){
      if(get_val(input_two[i], NodeData)== -1)
         return false
      input_two_local += get_val(input_two[i], NodeData)*10^i(nput_two.size()-i-1))
   }
   for (i=0 to i<output.size(); i++){
      if(get_val(output[i], NodeData)== -1)
         return false
      output_local += get_val(output[i], NodeData)*10^(output.size()-i-1))
   }
   if(input_one_local + input_two_local == output_local)
      return true;
   else
      return false;

Pass Out: bool
Endfunction


Function main
  Pass In: argument1(selection of search algorithm) , argument2(input1of calculation), argument3(input2 of
calculation), argument4(output of calculation)

Create object tree of class Tree
tree root = newNode(initial values)
tree buildTree(root)

if argument1 DFS
            then DFS(root)
if argument1 BFS
            then BFS(root)
```

```
else
        quit program

with result of DFS or BFS , create Cryptarithmetic Puzzles solution file
print(algorithm)
print (number of visited nodes with DFS or BFS)
print(Maximum number of nodes kept in the memory)
print(Running time: seconds)
print(Solution of DFS or BFS)

 Pass Out: Cryptarithmetic Puzzles solution file
Endfunction
```

### c. Complexity of algorithm

First, let's examine all functions one by one.

n is distict letter number.

1) Function newNode : has only 3 basic operations O(1)
2) Function availableNumbers : has 2 nested for loops O(10*n)
3) Function displayData : it is not using in the driver code. It is writed for checking/printing nodes inside Tree
4) Function createKeyVector : has 1 for loop O(n)
5) Function buildTree : The time complexity of treebuild if the entire tree is traversed is O(V) . V is the number of vertexes.
6) Function DFS: The time complexity of DFS if the entire tree is traversed is O(V) where V is the number of nodes. In the case of a graph, the time complexity is O(V+E) where V is the number of vertexes and E is the number of edges. Inside the while loop we are calling solution_finder function so O($n^2(V+E)$)
7) Function BFS: O(V + E) where V is total number of vertices in graph and E(total number of edges in graph).
8) Function get_val: has 1 for loop O(n)
9) Function solution_finder : has 3 for loop and inside a loop it's calling get_val function. So its O($n^2$)
10) Function main: If Argument written as DFS then the time complexity is O($n^2(V+E)$)

    if Argument written as BFS then the time complexity is O($n^2(V+E)$)

## 2) Analyze and compare the algorithm results in terms of:

### a. The number of visited nodes

The number of visited nodes changes depending on the algorithm. In addition, as the number of distinct letters increases, the number of visited nodes increases. This situation can be observed by looking at Figure 2, Figure 3, Figure 4 and Figure 5. DFS is better when there are solutions away from source. DFS is better for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop.


*Figure 2 Algorithm: BFS args : SEND MORE MONEY*

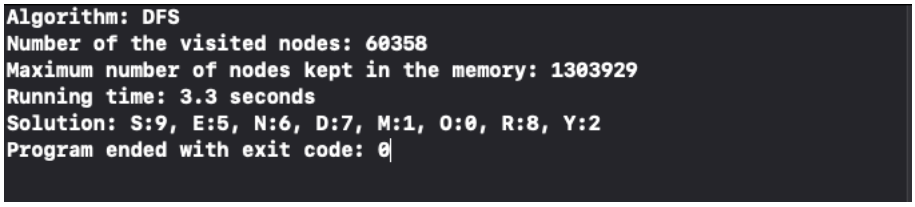


*Figure 3 Algorithm: DFS args : SEND MORE MONEY*


*Figure 4 Algorithm: BFS args : TWO TWO FOUR*

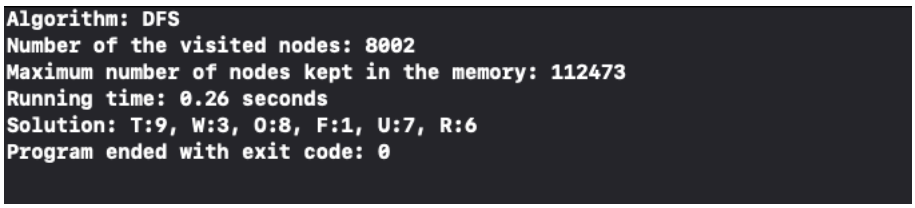


*Figure 5 Algorithm: DFS args : TWO TWO FOUR*

## b. The maximum number of nodes kept in the memory

int globalCounter = 0; as a global variable is defined. Purpose of the Defination of globalCounter is find the count of the maximum number of nodes kept in the memory. While calling the buildTree function, the globalCounter is incremented by 1 each time a node is created. As a result, enabling us to reduce the number of maximum number of nodes kept in the memory node depends on the content of the availableNumbers function that contains constraints and the number of distinct letters given as an argument. AvailableNumbers function handles 2 different constraints when creating a node. First, the pair of each distinct letter in the node must have different values. This is necessary for the puzzle to be solved. Second, the pair of the first letter of the words entered as argument cannot be 0. These two restrictions greatly reduce the number of nodes. The maximum number of nodes kept in memory is not dependent on the BFS or DFS algorithm in our code. Results can be observed by looking at Figure 2, Figure 3, Figure 4 and Figure 5.

### c. The running time

The changing of running time depends on many reasons. The main reasons are which algorithm was chosen for the solution, which constraints were taken into account when building the tree, and the word lengths entered, and the number of distinct letters in the entered words as arguments. The restrictions given when creating the tree are given in chapter 2.b.

## 3) Why we should maintain a list of discovered nodes?

In this implementation, visited information is kept in each node, not in a list.

**References:**


1) https://www.geeksforgeeks.org/generic-treesn-array-trees/
2) https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/
3) **BLG336E – Spring 2021 Course Materials**