CITS2200 project analysis

By Tony Guo(21960332) And

Hei Wong(21862296)

**Constructor:**

For the constructor, we have used the bufferedreader from the test class to read the example graph text file and stored this in a HashSet of String called mySet, this will automatically delete the repeated String of Lines (vertices) and only have all distinct vertices stored in the set. After reading the entire example graph text file, we then change the mySet into an array of Strings called vertex, which has the same size as the mySet.

Next We created an adjacency matrix called matrix by using the list implementation from java.util, and creating a Linked List in each index of the array, with the length of the array equivalent to the length of the vertex array and the length of each list equivalent to the number of edges every index for every list have. We have adapted the use of adjacency matrix in a 2d array at first, however we found that the worst-case scenario for the time complexity to go through the 2d arrays would be O(n). So, by using the adjacency list, it can access the next edge in a time of O(1), which is a lot more efficient comparing the worst case

scenarios in both implementations.

**Add edge:**

For the method addEdge, we are given two values in the parameter String urlFrom and String urlTo. We are to add in the value of the corresponding integer of urlto in the list of the corresponding index of urlfrom in the array to our adjacency list, for every urlFrom (vertex) to urlTo (vertex) read from 2 consecutive lines at a time in our graph text file by using bufferreader in our CITS2200ProjectTester class. To do this We introduced two integer variables f and t and initiate them both to be 0, since we already have all the vertices in our vertices array, we create two while loops, one for urlFrom and one for urlTo.

The first while loop is: while urlFrom doesn’t equal to the index f in vertices array we increment f by 1, the second while loop is: while urlTo does not equal to the index t in vertices array we increment t by 1. So, by the end of the while loops we have found the exact index which represents the vertices urlFrom and urlTo and stored them in f and t. We can then find the exact index for the array in the adjacency lists which stores our desired vertex using the integer f, and add the value of integer t in the exact list found through the index of the array which shows that there’s an edge from urlFrom to urlTo.

**getShortestPath:**

For getShortestPath there is also two values in the parameter String urlFrom and String urlTo. We first created a Priority Queue p to that loops through and enqueue every vertices that have not been visited. Then we created an array called visited which marks every vertices as visited at the index it is being stored in the adjacency list, once it has been through the Priority Queue. Next we created an array called distance which stores the shortest distance from the vertex urlFrom to every other vertices and store it in their individual index in the array.

First we loop through the entire distance array and set every value to inifinity which represents unreachable for every vertex that can’t be reached from the starting vertex(urlFrom). Next we created a search method in which it has a parameter String s which returns the position of the index for the String s in the vertices array. By using this method we can find out the the index for the String urlFrom and stored the value in int s.

After that we enqueue value s in the Priority Queue p and set its priority to 0 which is the highest priority, and set the index value s for the distance array to 0 to initialise the loops. We then loop while the Priority Queue p is not empty, and set int v1 equals to the value of p.dequeue then set the index v1 for array visited to 1 to show that we have visited this index already. After that we create an iterator for the Linked list at index of v1 in the adjacency matrix array and use another while loop for if there is a next element in the iterator, this test which vertices v1 is adjacent to. We then set integer j to be the next element adjacent to v1 in the iterator and use an if loop for if the index value j havn’t been visited, we’ll create a new int value called newDist which equals to the index v1 in the distance array plus one, this value represents the new distance from the starting vertex, if the value in the distance array with index j is bigger than this newDist value, then the distance with vertex j will equal to this newDist value. This will get the shortest path value from the starting vertex to the current j value vertex. We then enqueue in the priority queue p the value of distance array with index of j with priority of integer j to loop through the whole thing until the priority queue is empty.

After that we find the index value for urlTo using the search method we created and set a new integer value sp to equal to the index of the index value for urlTo in the distance array . So integer sp would give the distance of the shortest path from urlFrom to the urlTo. We then return sp to get this shortest path. If no shortestpath was found which is when the distance array has infinity for every value, we return -1 instead of sp. we have adapted to use Dijkstra’s algorithm for this method.

**Pseudo code for getShortestPath:**

**function** getShortestPath(urlFrom, urlTo):

**set** f = index of urlFrom

**for each** vertex v in Graph:

distance[v] = infinity

parent[v] = 0

visited[v] = 0

distance[f] = 0

visited[f] = 1

p := the priorityQueue of all nodes in Graph

add(f, 0) into the priorityQueue

**while** P **is not** empty:

v1 is node in p with smallest distance[]

remove v1 from P

**for each** neighbor j of v1:

newDist = dist[v1] + 1

**if** newDist < distance[j] **then**

distance[j] = newDist

parent[j] = v1

add(j, distance[j]) into the priorityQueue

**end if**

**end for**

**end while**

**set** t = index of urlTo

**set** sp = distance[t]

**If** sp = infinite **then return** -1

**return** sp

**Time complexity Analysis for getShortestPath:**

**Dijkatrs algorithm:**

Each vertex can be connected to (V – 1) vertices, where V is the number of vertices and E is the total number of edges

Finding the adjacent vertex’s weight which is 1 in this case, in the lower priority queue is O(log(V)) + O(1)

Since we are using Adjacency list instead of Adjacency matrix, the complexity would O(VE), instead of O(V^2).

Plus we used a search method to search the index of the string in the parameter,

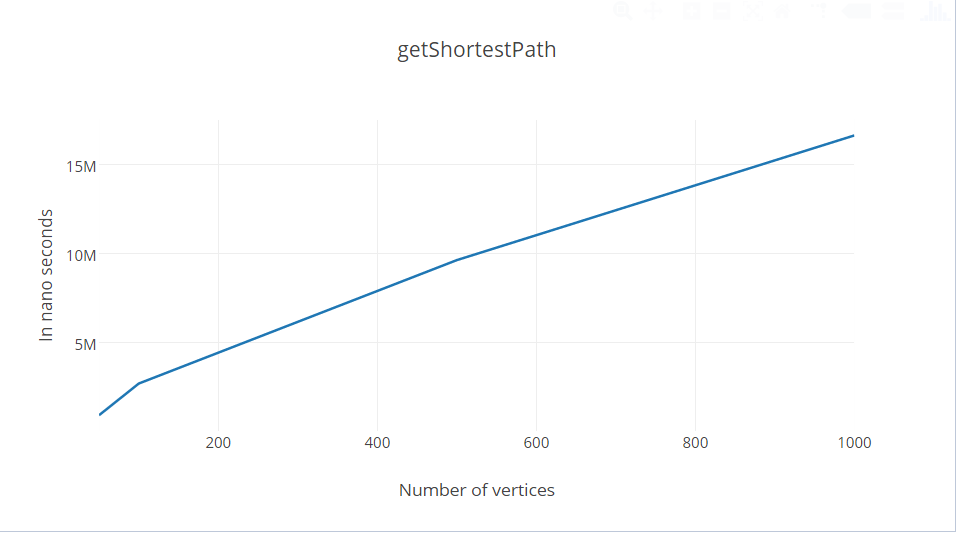
In the worst case, it might go through the whole loop and reaches the last index in terms of V and we need to do that for urlFrom and urlTo.

Therefore the Complexity is O(VElog(V) + 2V)

As number of vertices reaches infinity, complexity becomes O(VElog(V))

(Barbehenn & Hutchinson, 1995)

**Time complexity analysis Graph for getShortestPath method in our code:**

****

**getCenters:**

For getCenters the first half of our code is very similar to the shortest path method however we have to put the whole code for shortest path inside another for loop in which changes the starting vertex every time we loop through this for loop. We need to use this for loop because we have to change the starting vertex since we need to find the shortest distance path using every vertices as the starting vertex to every other vertices.

After that, we compare every shortest path with each starting vertex to see which starting vertex/vertices has the shortest path to every other vertices. To do this we need 2 Priority queues one priority queue to do the same job as in shortest path method and the other priority to store the shortest maximum distances(will elaborate further) and 2 different arrays, the visited and distance arrays being the same as in the getShortestpath. So first we need to create one for loop to loop through the starting vertices, then we create another for loop inside the first for loop to change every index for distance array to infinity, and all the index for the visited array to 0. After that we do the exact same thing as in the getShortestpath to generate a distance array that has all shortest distance values to every vertices for the current starting vertex as we’re generating the distance array, an integer “max” will be compared, so that everytime a new value is added into the distance array, it will compare with the max value. This process will automatically ignore the infinite index and the integer max will have the max distance value from the current distance array in the end of the while loop. We then enqueue the index value for the current starting vertex with priority of the current max value into the second priority queue called p1.

After this we finally end the first for loop which looped through every starting vertices. Now our p1 priority queue will have all the maximum distances as the priority with the index of different starting vertices as the values and the first element with the lowest priority which has the smallest value of the maximum distances, this is the center of the graph that we want. then we create and assign integer lowp to the first element by using the examine method and go through the priority queue p1 from the starting element and compare it with the lowp because there might be more than one centers with the same "shortest max distance". After that we put the center vertex/vertices into an arraylist without worrying about the size of the arraylist, then we can assign an array with a correct size and store the vertices from the arraylist to the array and then finally return the array containing the center/centres of the graph.

We decided to ignore the (potential) center which is at the edge in the graph where no vertices can reach the vertex, because our sense of centers are vertex/vertices that are surrounded by all the other vertices.

**Pseudo code for getCenters():**

**function** getCenters():

**p** := the priorityQueue of all nodes in Graph

**p1** := the priorityQueue of all nodes longest distance in Graph

**for each** vertex i in Graph:

**for each** vertex v in Graph:

parent[v] = 0

distance[v] = infinite

visited[v] = infinite

**End for**

**set** max = 0

**add**(i, 0) into the priorityQueue

distance[i] = 0

**while** p **is not** empty:

v1 is node in p with smallest distance[]

**remove** v1 from p

**for each** neighbor j of v1

newDist = distance[v1] + 1

If newDist < distance[j] then

distance[j] = newDist

parent[j] = v1

**add**(j, distance[j]) into the **priorityQueue p**

**If** distance[j] > max **then** max = distance[j]

**end if**

**end for**

**If** max is not 0

**add**(i, max) into **priorityQueue p1**

**end if**

**End while**

**set** lowp = first element in p1, which is the lowest maximum distance

**Initialize** ArrayList obj

**While** p1 **is not** empty **and** first element is lowp:

**add** first item into the arraylist

**end while**

**Initialize** String array s

**for each** index o in the arraylist:

put the index of the string into the array s

**end for**

**return** s[]

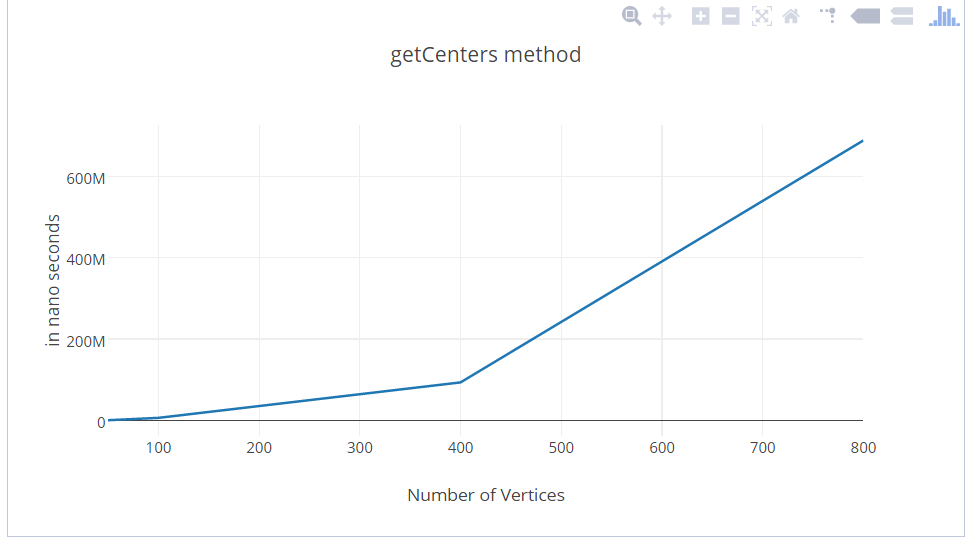
**Time complexity Analysis for getCenters:**

O(V^2\*Elog(V^2)), it is one more V unit time more than get shortest path, because we using every vertex as a starting point.

In the worst case if it is a complete graph the time complexity for getcenters will be O(V^2\*Elog(V^2) +2V) the 2V comes from storing objects into the arraylist and putting it back into the output array, which is V + V in the worst case if its a completed graph where every vertex is linked to one and other.

As V increases to infinite, the complexity is O(V^2 \* E log(V^2))

**Time complexity analysis Graph for getCenters method in our code:**



**getStronglyConnectedComponents:**

For strongly connected component we first created a Stack called stack and an array called visited, aswell as a Queuelist called q, and 2 integers one called rows and one called columns both starting as 0, then we create a method depth first search, this DFS method will go through the whole graph and put every index of vertex array into the stack if the vertex has not been visited by recursively calling itself.

After this we create a reverse graph by reversing the array in our adjacency list, which generate a new graph called rgraph that is the complete opposite of our original graph. Next while the stack is not empty we will use another depth first search for the reverse graph, to loop through the children of the poped vertex from the stack that have not been visited and enqueuing everything into a new Queue until there’s no children left for this specific vertex in which we then enqueue -1 into the queue to separate each strongly connected components made in the queue and everytime -1 is enqueued rows integer is incremented by 1.

Then we turn the queue q into an array called s. We’ll create another integer called index and initiated as 0. Then we create a new 2d array of Strings called ss with the number of rows being integer rows and the number of columns the same size as the adjacency list. After that, we loop through the s array and add if the value equals -1 in the array we increment index by 1 and make the integer column equals to 0 again then move on to the next value in the array s, else we will put the string of the current vertex in array s into the row with the current interger index as the index and the column with the current integer columns as the index. So in the end we’ll have all the strongly component seperated into different rows in the 2d array ss and finally we return the 2d array ss.

**Pseudo Code for getStronglyConnectedComponents:**

**procedure** getStronglyConnectedComponent():

**Initialize** a stack

**Initialize** an array called visited with length of number of vertices

**for each** vertex i in Graph:

**if** visited[i] = 1 **then**

continue the loop

**end if**

DFS(i, visited, stack)

**end for**

Create an array of list called Adjreversed

Adjreversed = Reverse the adjacency list

refresh the visited array with 0

**Initialize** q as a Queue of Integers

**while** stack **is not** empty:

Set v1 = pop off stack

**if** visited[v1] = 1 **then**

continue the while loop

**end if**

DFSR(v1, visited q)

add(-1) into q //to indicate next sets of SCC

**end while**

turns q into an array

array s = q

**for each** index i in s

whenever it senses a -1

**row** = row + 1 //so that we know how many rows do we need for the final output 2d array / //for how many sets of SCC

**end for**

**set** index = 0

**Initialize** ss[row][matrix.length]

**for each** index j in s[]

**If** (s[j] = - 1) **then**

index++

columns = 0

**continue** the loop

**end if**

ss[index][columns] = the String index of s[j]

columns+1

**end for**

**return** ss

**procedure** DFS(int v, int[] visited, Stack<Integer> stack):

**set** visited[v] = 1

**set** i = an edge of v

**if** visited[i] = 0 **then**

DFS(i, visited, stack)

**end if**

**push** v into the stack

**procedure** DFSR(int v, int[] visited, Queue<Integer> q):

Set visited[v] = 1

add v into the q

i = edge of the v in the reverseGraph

**if not** visited **then**

DFSR(i, visited, q)

**end if**

**Time complexity Analysis for getStronglyConnectedComponents:**

**Kosaraju's algorithm:**

Since we are using an adjacency list, our adaption of the Kosaraju's algorithm will perform multiple traversals, first it will call the DFS(depth first search) method and then a reverse of the graph will be generated and then DFSR will be called(depth first search reversed) . our DFS and DFSR will both take O(V+E) time. Reversing the graph also takes O(V+E) time as we traverse the adjacency list as well. Plus for the output 2d array, it takes V time to fill the 2D arrays for different sets of strongly connected components.

The complexity of getStronglyConnected method is **O(4V+3E)**, if we used adjacency matrix, the complexity is O(3V^2 + V)

But as number of vertices is approaching infinity, the complexity will just become **O(V+E)**.

(Griffith,2013)

**Time complexity analysis Graph for getStronglyConnectedComponents in our code:**

**(All the edges are randomly generated)**

|  |  |
| --- | --- |
|  |  |
|  | C:\Users\Tony\AppData\Local\Microsoft\Windows\INetCache\Content.Word\getstronglyconnect.png |
|  |  |

**getHamiltonianPath:**

For this method we have adapted to use a backtracking method by introducing depth first search in our code. We have first Created a List of arraylist of strings called l with each arraylist in list l storing a possible Hamiltonian path and l stores every possible Hamiltonian path from the graph. We also have two arrays a path array holding the path of the hamiltonian path and a visited array which contains all the vertices that has been visited so far.

We then create a for Loop to loop through every starting vertices, after that we have an integer already created in the field called pathcount with a value of 1, and also an integer already created in the field called count and we make pathcount equals to count. then we put the first starting vertex we have looped into the path array at index 0 and put the starting vertex as visited in the visited array at index 0 aswell, then we look at our depth first search method which is called DFSH. DFSH will search through the starting vertex for whether it has an adjacent vertex that is directly linked from the starting vertex (the children of the starting vertex) or not by using a while loop for the adjacency list we created, if a children is found, that children will be put into the path array at the index pathcount and mark it as visited, pathcount will also be incremented. DFSH will then search through all of it’s children and when it reaches the last children and when no more children can be found, we check the path count does it equal to matrix .length then the while loop would break, if it doesn’t, the visited array of that children will be marked as unvisited(0) again and go back to the parent of that children to look for another children from that parent. If that parent have no more children to be explored, last index of the path will return to 0 and path count will be decremented.

After that looking back at the getHamiltonianPath method the starting vertex will be changed to the next one, the path array and visited array will both be refreshed and pathcount and count both equals to 0 again, then we do the whole thing again with the starting vertex back to the DFSH method. This happens until, pathcount equals to the Length of the adjacency matrix of the graph, the DFSH loop will halt and cease to continue anything furthur, and it goes back to the Hamiltonian path method again.

From that, an arraylist of string called al will be created, this arraylist will store the String of the correlated index in the path array, and then in the list of arraylist l, the arraylist al will be added into the list of arraylist l. This process will still be continued on for every possible starting vertex there is to find every possible hamiltonian path in the graph. At last, a new arraylist call alf will be created and alf will be equal to the first arraylist in the list of arraylist l, then an array called s will store everything in the arraylist alf and finally, we return s as the hamiltonian path. If the l is empty which means there’s no hamiltonian path, we’ll return the s aswell which is an empty array in this case.

This implementation is one of the easier methods for hamiltonian path however it is one of the slowest with a time complexity of around O(!n), we have found another method that could be implemented at a much faster time complexity of around O(2^n \* n^2), it uses the Bellman-Held-Karp algorithm, which implements the dynamic programming algorithm. The basic idea is that we need a method which generates all sets of combinations depending on the number of vertices.

Firstly, it will generate an empty set where it determines the starting vertex to every other vertices, after that it generates sets with single vertex, where the sets are for calculating the distance from all the vertices to that single set and then it generates sets with 2 vertices and so on. After that, we calculate the distance from a vertex to the vertices in the sets, we then store the distance inside a data structure such as a map. Therefore we can continuously use the value we stored in the map to determine the distance for different sets.

For example, there are 4 vertices 0 1 2 3, and the matrix is given:

0 2 9 10

1 0 6 4

15 7 0 8

6 3 12 0

Firstly, empty sets are generated, we take 0 as the starting vertex

Set {}:

{2, empty} = 1

{3, empty} = 15

{4, empty} = 6

Then we store all these these values inside a map,

Secondly, we generate more sets with one vertex

Set {2}, set {3} and set{4}:

(3, {2}) = From 2 to 3 + {2, empty} = 7 + 1 = 8

{4, {2}) = from 2 to 4 + {2, empty} = 3 + 1 = 4

Then, we continuously store these values into a map and the parent vertex will be stored in the map as well, so we can use the value for the next sets.

After that, we find out all the possible sets and calculate the values for every single one of them, then we know all the possible Hamiltonian path.

In this case, if the value of the set is equal to the number of vertices, then a hamiltonian path is found. To output the path, we simply continuously find the parent of the the vertex found in the map and output it in an array.

We ran into trouble with generating sets of all combinations, storing values and parent in a data structure where we can also get the value for calculating different sets. Therefore we couldn’t implement this algorithm.

**Pseudo Code for getHamiltonianPath:**

**procedure** getHamiltonianPath():

**Initialize** path, visted and s array

**Initialize** a List of ArrayList of String called l

**for each** vertex i in Graph:

**set** pathcount, count = 0

fill array visited and path with zeros

**set** pathcount = 1

**set** path[0] = i

**set** visited[i] = 1

DFSH(i, visited, path)

**set** pathcount = count

**if** pathcount = matrix.length then

**Initialize** an ArrayList of String called al

store every String of the correlated index in the path array to ArrayList al

then add al into l

**end if**

**end for**

**if** size of the l is 0

**return** s //which is an empty array

**Initialize** an ArrayList of String called alf

**set** alf = first arraylist in the List

**for each** Strings value in alf

put it into Array s

**end for**

**return** s

**procedure** DFSH(int v, int[] visited, int[] path):

**while** has edge:

**set** i to the edge of v

**if not** visited **then**

**set** path[pathcount] = i

pathcount + 1

**set** visited[i] = 1

**if** pathcount = graph length **then**

count = pathcount

**break** the loop

**end if**

DFSH(i, visited, path):

**if** count is not graph length **then**

**set** visited[path[pathcount]] = 0

**end if**

**end while**

**if** count **is not** graph length **then**

**set** path[pathcount] = 0

pathcount - 1

**end if**

**Time complexity Analysis for getHamiltonianPath:**

**Backtrack Algroithm**

In Hamiltonian path that we implemented, for each recursive call (DFSH method), one of the remaining vertices is selected in the worst case. Recursion in this case can be thought of as n nested loops of each vertices, where in each loop the number of iterations decreases by one. For example, in this case no more than 20 vertices will be accessed, In each recursive call the branch factor decreases by 1. Hence the time complexity is given by:

O(20 \* (20 - 1) \* (20 - 1) … (N - 1))

O(20!)

Plus we need to put a path with every vertex inside the arraylist into an array which will also take V amount of unit time.

Therefore complexity is O((20!) + 20)

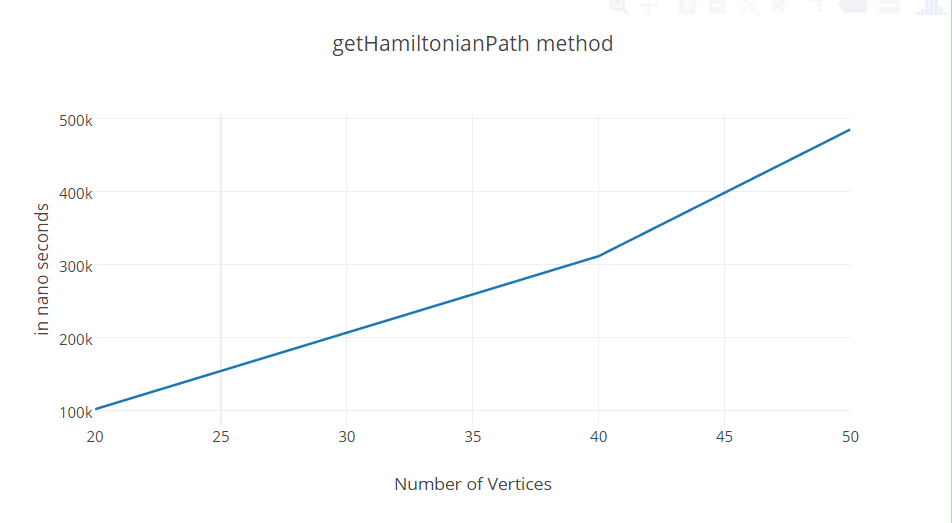
O(V ! + V)

As V approaches infinite, complexity becomes O(V!).

(Introduction To Backtracking Programming | Algorithms, 2015)

**Time complexity analysis Graph for getHamiltonianPath in our code:**

**(All the edges are randomly generated)**

****

Referncing:

Barbehenn, M., & Hutchinson, S.(1995). "Efficient Search and Hierarchical Motion Planning by Dynamically Maintaining Single-Source Shortest Paths Trees", *IEEE Trans. Robotics and Automation*, vol. 11, no. 2, pp. 198-214.

Griffith, R. (2013). *Algorithms in Dart: Kosaraju's Algorithm - { Dart Code Here }*

Retrieved from https://www.greatandlittle.com/studios/index.php?post/2013/02/23/Algorithms-in-Dart-Kosarajus-Algorithm

*Introduction To Backtracking Programming | Algorithms*. (2015). *Algorithms*. Retrieved from http://algorithms.tutorialhorizon.com/introduction-to-backtracking-programming/