# MARMARA UNIVERSITY – FACULTY OF ENGINEERING

# CSE 4082 ASSIGNMENT #1 REPORT

**Name, Surname:** Hasan Şenyurt – Melisa Durmuş

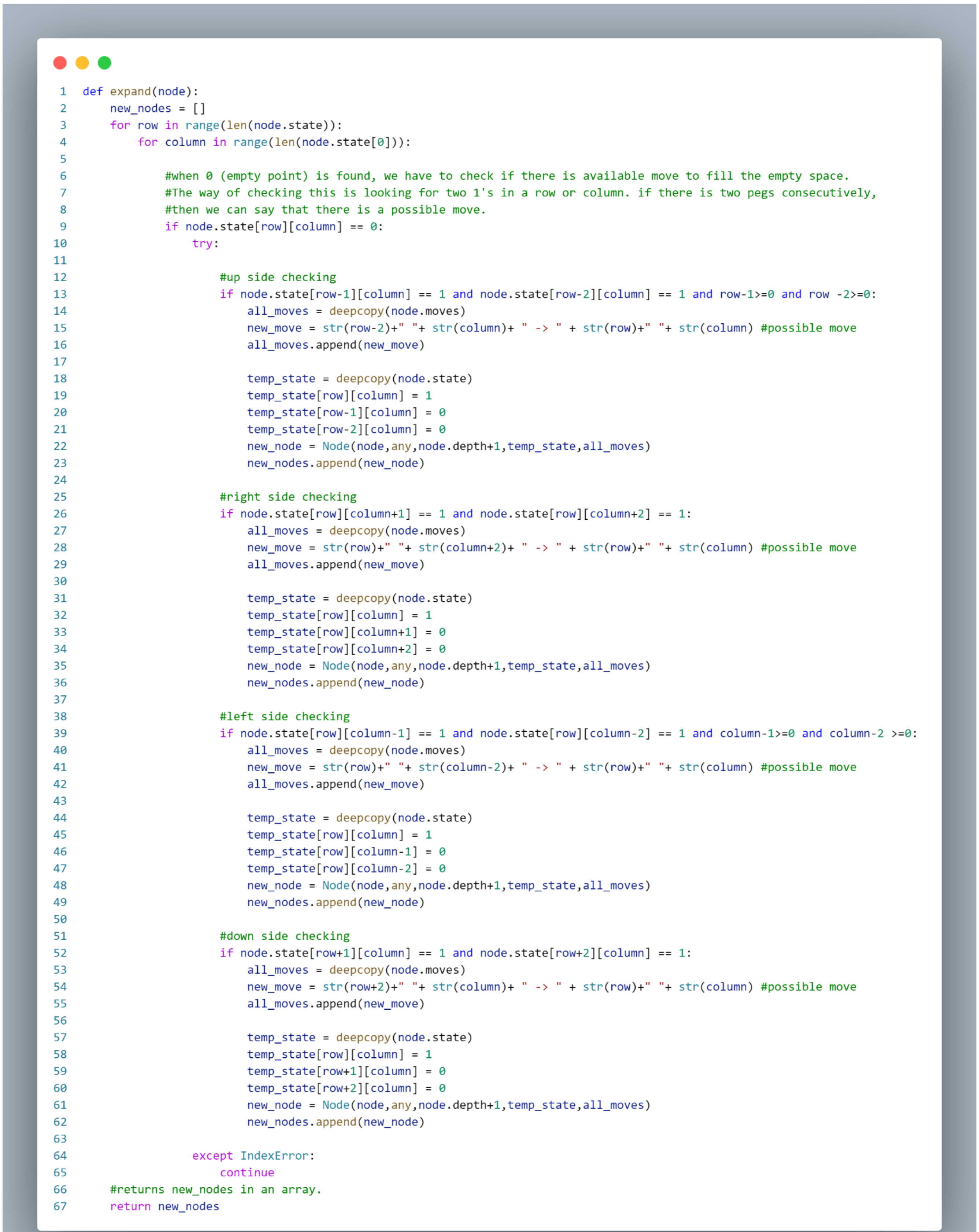
**Student ID:** 150120531 – 150119727

**Department:** Computer Engineering

## TABLE OF CONTENTS

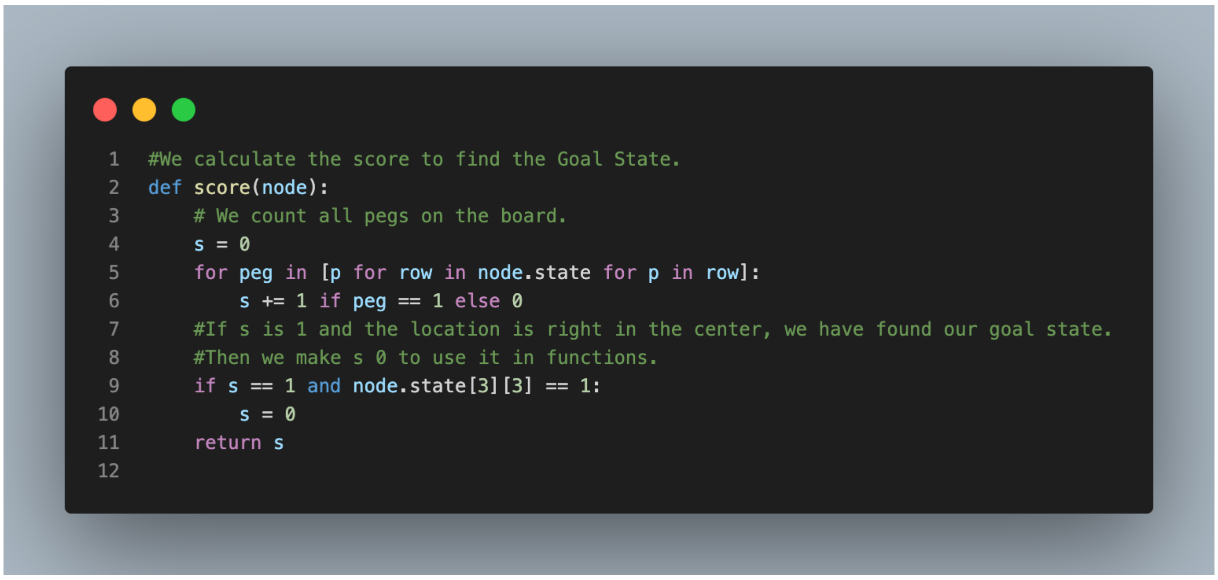
## Expand Function

Main idea of ‘expand’ function is finding zeros (empty spaces) in the peg solitaire board. Rows and columns of the state of the node that is sent as parameter to this function are searched. After searching, if there is an empty space in board, function checks if there are two ones (two pegs) consecutively. If the answer is yes, then new node is created from the node that is parent of new child. After nodes are created, function gathers them as a list and also returns that list.



```python
def expand(node):
    new_nodes = []
    for row in range(len(node.state)):
        for column in range(len(node.state[0])):

            #when 0 (empty point) is found, we have to check if there is available move to fill the empty space.
            #The way of checking this is looking for two 1's in a row or column. if there is two pegs consecutively,
            #then we can say that there is a possible move.
            if node.state[row][column] == 0:
                try:

                    #up side checking
                    if node.state[row-1][column] == 1 and node.state[row-2][column] == 1 and row-1>=0 and row -2>=0:
                        all_moves = deepcopy(node.moves)
                        new_move = str(row-2)+" "+ str(column)+ " -> " + str(row)+" "+ str(column) #possible move
                        all_moves.append(new_move)

                        temp_state = deepcopy(node.state)
                        temp_state[row][column] = 1
                        temp_state[row-1][column] = 0
                        temp_state[row-2][column] = 0
                        new_node = Node(node,any,node.depth+1,temp_state,all_moves)
                        new_nodes.append(new_node)

                    #right side checking
                    if node.state[row][column+1] == 1 and node.state[row][column+2] == 1:
                        all_moves = deepcopy(node.moves)
                        new_move = str(row)+" "+ str(column+2)+ " -> " + str(row)+" "+ str(column) #possible move
                        all_moves.append(new_move)

                        temp_state = deepcopy(node.state)
                        temp_state[row][column] = 1
                        temp_state[row][column+1] = 0
                        temp_state[row][column+2] = 0
                        new_node = Node(node,any,node.depth+1,temp_state,all_moves)
                        new_nodes.append(new_node)

                    #left side checking
                    if node.state[row][column-1] == 1 and node.state[row][column-2] == 1 and column-1>=0 and column-2 >=0:
                        all_moves = deepcopy(node.moves)
                        new_move = str(row)+" "+ str(column-2)+ " -> " + str(row)+" "+ str(column) #possible move
                        all_moves.append(new_move)

                        temp_state = deepcopy(node.state)
                        temp_state[row][column] = 1
                        temp_state[row][column-1] = 0
                        temp_state[row][column-2] = 0
                        new_node = Node(node,any,node.depth+1,temp_state,all_moves)
                        new_nodes.append(new_node)

                    #down side checking
                    if node.state[row+1][column] == 1 and node.state[row+2][column] == 1:
                        all_moves = deepcopy(node.moves)
                        new_move = str(row+2)+" "+ str(column)+ " -> " + str(row)+" "+ str(column) #possible move
                        all_moves.append(new_move)

                        temp_state = deepcopy(node.state)
                        temp_state[row][column] = 1
                        temp_state[row+1][column] = 0
                        temp_state[row+2][column] = 0
                        new_node = Node(node,any,node.depth+1,temp_state,all_moves)
                        new_nodes.append(new_node)

                except IndexError:
                    continue
    #returns new_nodes in an array.
    return new_nodes
```

## Score Function



```python
#We calculate the score to find the Goal State.
def score(node):
    # We count all pegs on the board.
    s = 0
    for peg in [p for row in node.state for p in row]:
        s += 1 if peg == 1 else 0
    #If s is 1 and the location is right in the center, we have found our goal state.
    #Then we make s 0 to use it in functions.
    if s == 1 and node.state[3][3] == 1:
        s = 0
    return s

```

The purpose of the score function is to find out if we have found a goal state. this function counts all pegs above the incoming state and returns an integer. Our goal is to have 1 peg left, and the remaining peg to be in the center. If there is no 1 peg left, the score function only returns the number of pegs remaining. But if there is only 1 peg left, it also checks the location of the remaining peg. if the remaining peg is in the center, it says goal state found. S is set to 0, and other functions also recognize that when s is 0, it is the goal state.

## 1- Breadth First Search

The purpose of the breadth first search algorithm is to visit all nodes in depth. After visiting all nodes of the same depth, it goes to one depth below. therefore, it cannot go down without visiting all nodes at the same depth. If the solution we are looking for is in the region of the tree with less depth, this algorithm will be the best algorithm for us.

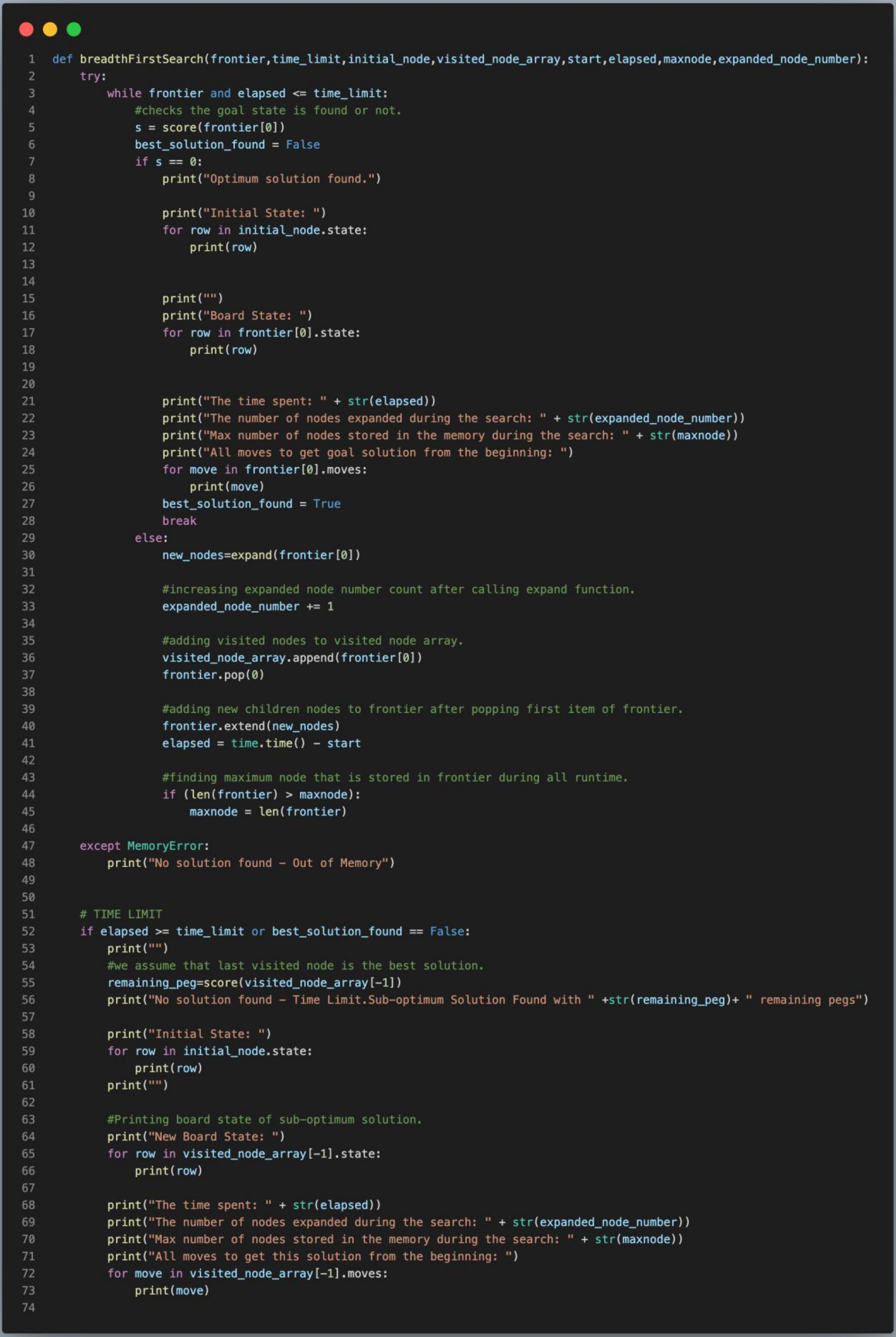
The depth of some search trees can be very long depending on the problem. In such cases, breadth first search can cause to fail to find the result, or take too long if the goal state is too deep in the tree.

The solution we are looking for in solotest is to have only one peg left on the board and the remaining peg to be in the very center of the board. There are 33 pegs on the board. The case of 1 peg remaining indicates that the solution is at the 32nd depth of the tree. And because there is a case of branches increasing as you go deeper, the tree becomes very large and very deep. and since the time limit is 1 hour, this algorithm will not be optimal for solotest.

In the breadth first search algorithm, we first checked whether the incoming node is the goal state. If s returned from the score function is equal to 0, we have reached the goal state. When we reach the goal state, we print the solution we found, the moves made until we find the solution, and the data such as the number of extended nodes. We write this in a try except block. Because our tree will be very large and many nodes will be kept in memory, we may get a memory error.

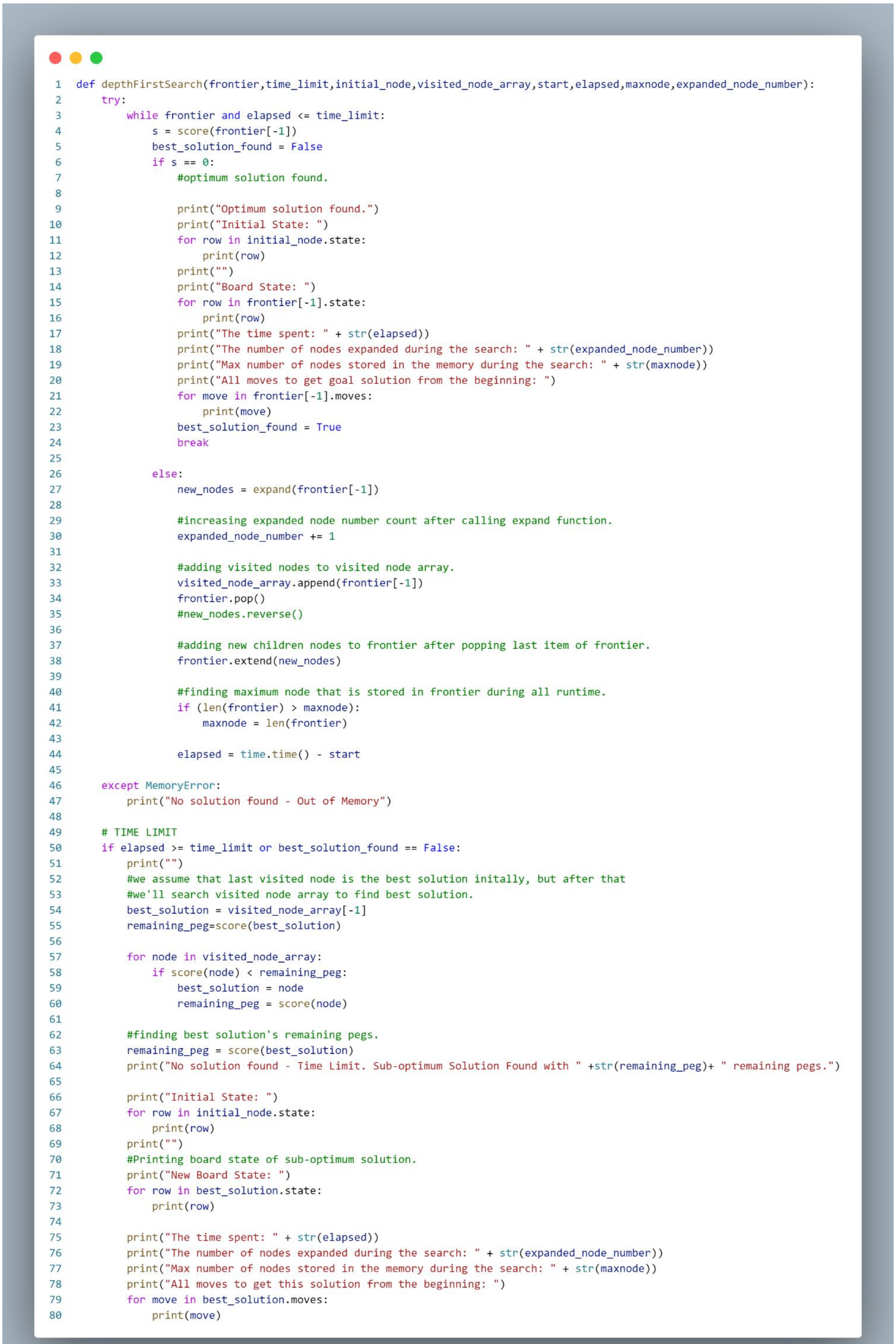
If the incoming node is not the goal state we are looking for, it will go inside the else block. Our original breadth first search algorithm starts here. New children from the incoming node, namely frontier[0], are extended and thrown into the frontier array. Then we increase the expanded node number because to keep track of how many times we have expanded the node. After creating the children of the later node, we remove that node from the frontier by throwing it into the visited_array_list. We threw the extended node array to the frontier and continued the algorithm.

Specific to the breadth first search algorithm, we consider the last visited node to be the best result. so if the time runs out, we return the last element of the visited_node_array as suboptimal result and output the result.



```python
def breadthFirstSearch(frontier,time_limit,initial_node,visited_node_array,start,elapsed,maxnode,expanded_node_number):
    try:
        while frontier and elapsed <= time_limit:
            #checks the goal state is found or not.
            s = score(frontier[0])
            best_solution_found = False
            if s == 0:
                print("Optimum solution found.")

                print("Initial State: ")
                for row in initial_node.state:
                    print(row)


                print("")
                print("Board State: ")
                for row in frontier[0].state:
                    print(row)


                print("The time spent: " + str(elapsed))
                print("The number of nodes expanded during the search: " + str(expanded_node_number))
                print("Max number of nodes stored in the memory during the search: " + str(maxnode))
                print("All moves to get goal solution from the beginning: ")
                for move in frontier[0].moves:
                    print(move)
                best_solution_found = True
                break
            else:
                new_nodes=expand(frontier[0])

                #increasing expanded node number count after calling expand function.
                expanded_node_number += 1

                #adding visited nodes to visited node array.
                visited_node_array.append(frontier[0])
                frontier.pop(0)

                #adding new children nodes to frontier after popping first item of frontier.
                frontier.extend(new_nodes)
                elapsed = time.time() - start

                #finding maximum node that is stored in frontier during all runtime.
                if (len(frontier) > maxnode):
                    maxnode = len(frontier)

    except MemoryError:
        print("No solution found - Out of Memory")


    # TIME LIMIT
    if elapsed >= time_limit or best_solution_found == False:
        print("")
        #we assume that last visited node is the best solution.
        remaining_peg=score(visited_node_array[-1])
        print("No solution found - Time Limit.Sub-optimum Solution Found with " +str(remaining_peg)+ " remaining pegs")

        print("Initial State: ")
        for row in initial_node.state:
            print(row)
        print("")

        #Printing board state of sub-optimum solution.
        print("New Board State: ")
        for row in visited_node_array[-1].state:
            print(row)

        print("The time spent: " + str(elapsed))
        print("The number of nodes expanded during the search: " + str(expanded_node_number))
        print("Max number of nodes stored in the memory during the search: " + str(maxnode))
        print("All moves to get this solution from the beginning: ")
        for move in visited_node_array[-1].moves:
            print(move)

```

## 2- Depth First Search

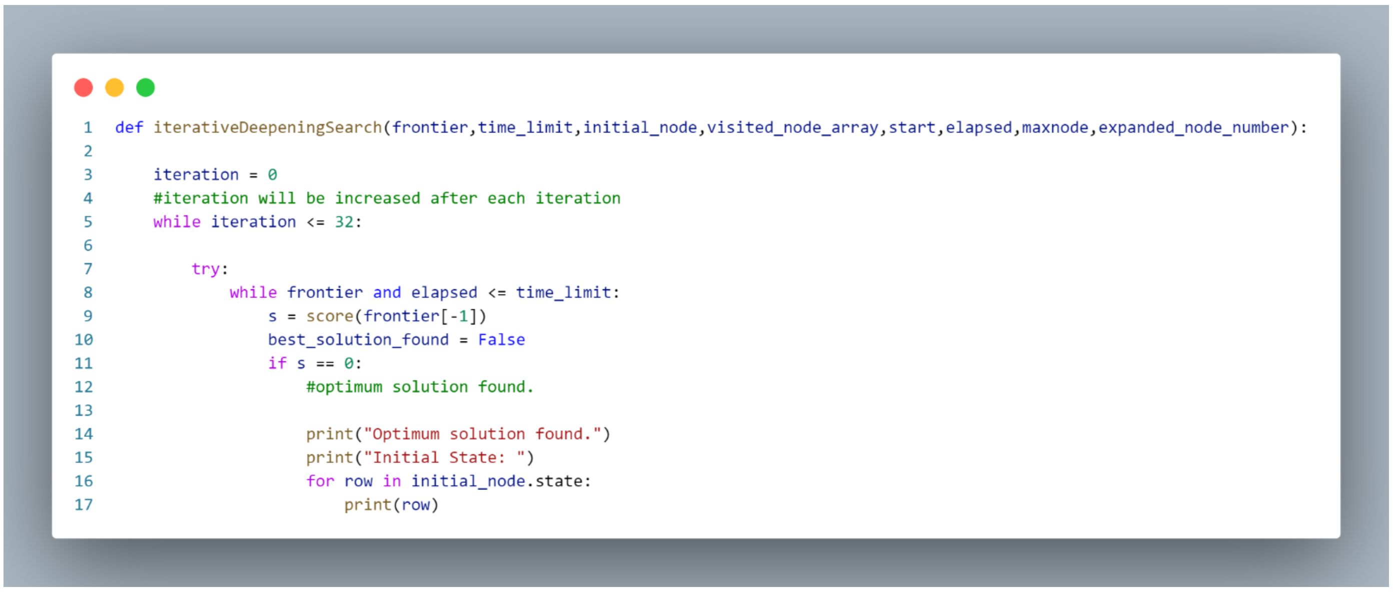


```python
def depthFirstSearch(frontier,time_limit,initial_node,visited_node_array,start,elapsed,maxnode,expanded_node_number):
    try:
        while frontier and elapsed <= time_limit:
            s = score(frontier[-1])
            best_solution_found = False
            if s == 0:
                #optimum solution found.

                print("Optimum solution found.")
                print("Initial State: ")
                for row in initial_node.state:
                    print(row)
                print("")
                print("Board State: ")
                for row in frontier[-1].state:
                    print(row)
                print("The time spent: " + str(elapsed))
                print("The number of nodes expanded during the search: " + str(expanded_node_number))
                print("Max number of nodes stored in the memory during the search: " + str(maxnode))
                print("All moves to get goal solution from the beginning: ")
                for move in frontier[-1].moves:
                    print(move)
                best_solution_found = True
                break

            else:
                new_nodes = expand(frontier[-1])

                #increasing expanded node number count after calling expand function.
                expanded_node_number += 1

                #adding visited nodes to visited node array.
                visited_node_array.append(frontier[-1])
                frontier.pop()
                #new_nodes.reverse()

                #adding new children nodes to frontier after popping last item of frontier.
                frontier.extend(new_nodes)

                #finding maximum node that is stored in frontier during all runtime.
                if (len(frontier) > maxnode):
                    maxnode = len(frontier)

                elapsed = time.time() - start

    except MemoryError:
        print("No solution found - Out of Memory")

    # TIME LIMIT
    if elapsed >= time_limit or best_solution_found == False:
        print("")
        #we assume that last visited node is the best solution initally, but after that
        #we'll search visited node array to find best solution.
        best_solution = visited_node_array[-1]
        remaining_peg=score(best_solution)

        for node in visited_node_array:
            if score(node) < remaining_peg:
                best_solution = node
                remaining_peg = score(node)

        #finding best solution's remaining pegs.
        remaining_peg = score(best_solution)
        print("No solution found - Time Limit. Sub-optimum Solution Found with " +str(remaining_peg)+ " remaining pegs.")

        print("Initial State: ")
        for row in initial_node.state:
            print(row)
        print("")
        #Printing board state of sub-optimum solution.
        print("New Board State: ")
        for row in best_solution.state:
            print(row)

        print("The time spent: " + str(elapsed))
        print("The number of nodes expanded during the search: " + str(expanded_node_number))
        print("Max number of nodes stored in the memory during the search: " + str(maxnode))
        print("All moves to get this solution from the beginning: ")
        for move in best_solution.moves:
            print(move)
```

In depth first search, firstly we add initial state to frontier in main function. Secondly, we checked whether the last element of frontier is goal state or not. If it is not new nodes are created from that node, and the node that is not goal state is popped from frontier. New nodes added to frontier. Also, all visited nodes are added to visited node array. The critical thing is in depth first search is that we always look to end of the frontier, and we always popped from end of the frontier. We managed to look depth by depth in this way. If the goal state cannot be found in given time, code will go to time limit section. In that section, we search the best solution that is found in given time by looking at the visited node array. After finding best solution, we print the results.

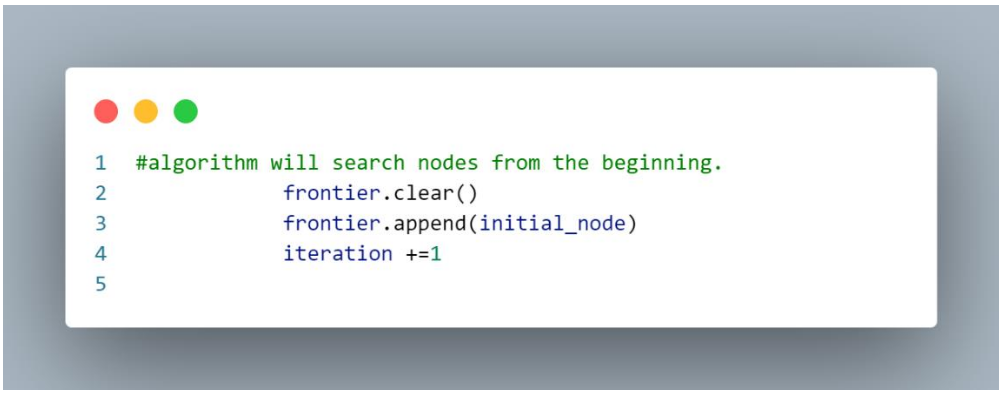
## 3- Iterative Deepening Search

Iterative deepening search is very similar to depth first search. Only difference between them is iterative deepening search has limit in terms of depth. In each iteration the limit of depth is increased, and algorithm starts repeatedly. We added one loop to depth first search code, and we clear frontier after each iteration. These little features make depth first search iterative deepening search.



```python
def iterativeDeepeningSearch(frontier,time_limit,initial_node,visited_node_array,start,elapsed,maxnode,expanded_node_number):

    iteration = 0
    #iteration will be increased after each iteration
    while iteration <= 32:

        try:
            while frontier and elapsed <= time_limit:
                s = score(frontier[-1])
                best_solution_found = False
                if s == 0:
                    #optimum solution found.

                    print("Optimum solution found.")
                    print("Initial State: ")
                    for row in initial_node.state:
                        print(row)
```

Iteration is added.



```
#algorithm will search nodes from the beginning.
            frontier.clear()
            frontier.append(initial_node)
            iteration +=1

```

Frontier is cleared.

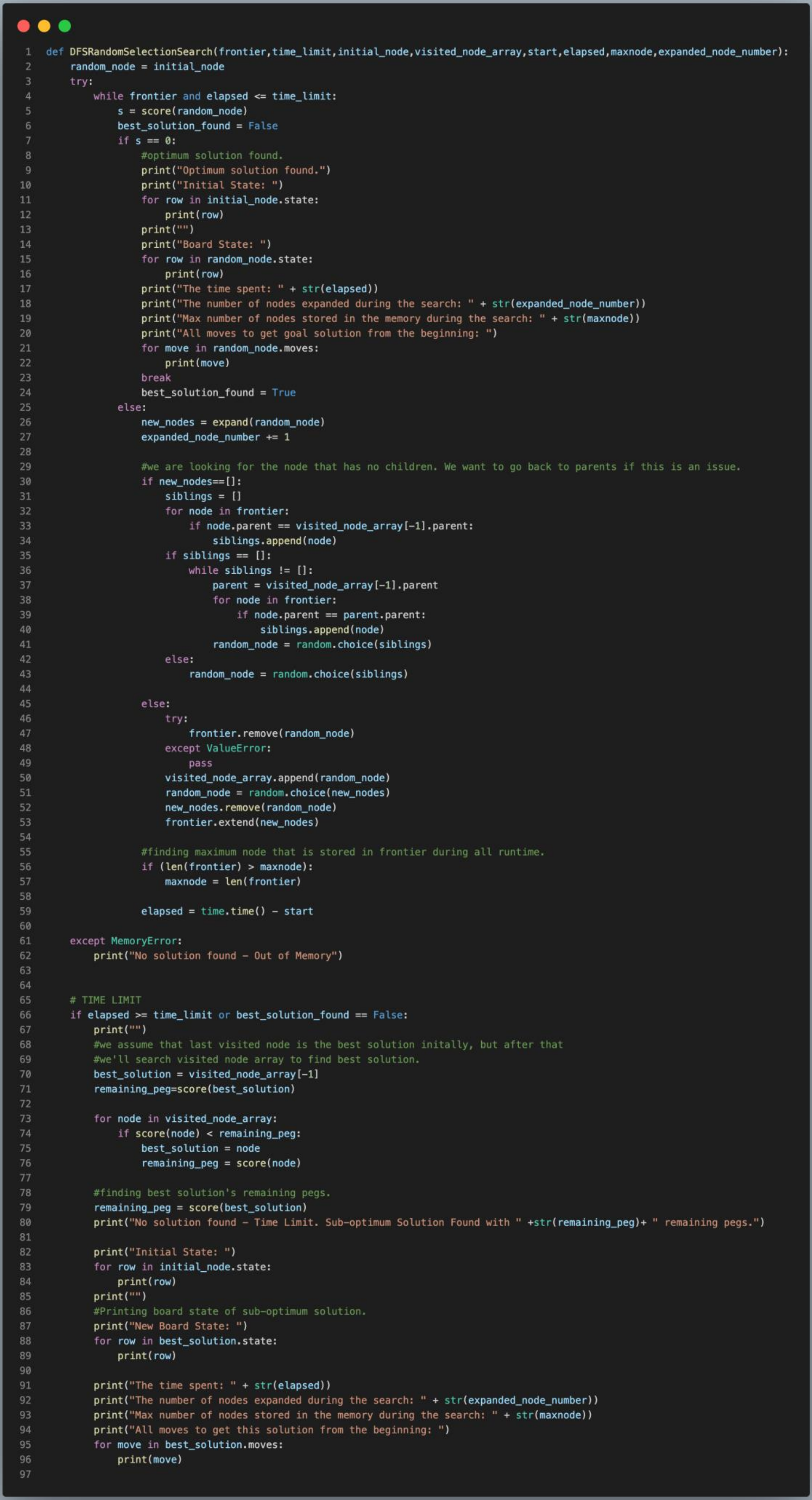
## 4- Depth First Search Algorithm with Random Selection

The depth first search with random selection algorithm works with the same logic as depth first search. The only difference between them is that when descending to the next depth, it randomly chooses the child to go to among the children of the parent node. It proceeds by choosing the children randomly in this way until it reaches the depth at the bottom. When it goes down to the bottom depth, it continues to progress by randomly choosing from the other children of the parent, namely its siblings, since it will not have any more children. This way it tries to find the goal state until the time reaches its limit.

We equate the first node, that is, the initial node, as a random node. In this algorithm, as in the others, we look at whether the preceding node is the goal state. if random node is not goal state, we extend new children from random node. If the extended node array is empty, then the node is at the lowest depth, so it has no children. If it has no children, we look after its siblings. if they don't have siblings, we look at their parent's siblings. We continue in this way until we find a new node.

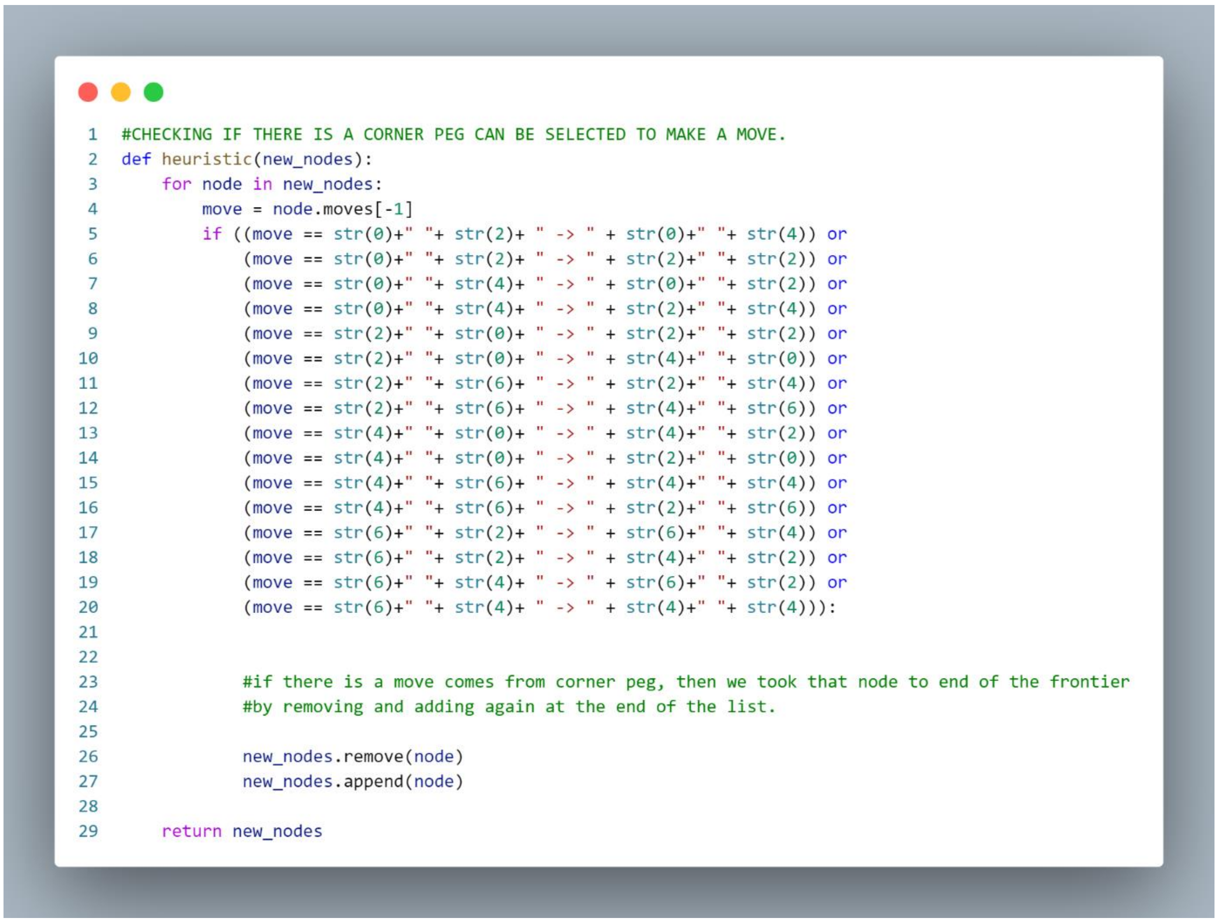
If the originally extended nodes do not return an empty array, we remove our random node from the frontier and add it to the visited_array_list. Then we add one of the newly arrived nodes to the frontier randomly and continue the algorithm.

If we can't find the goal state before the timer runs out, we print the best result from the nodes we've looked at so far, as a suboptimal result.


```python
def DFSRandomSelectionSearch(frontier,time_limit,initial_node,visited_node_array,start,elapsed,maxnode,expanded_node_number):
    random_node = initial_node
    try:
        while frontier and elapsed <= time_limit:
            s = score(random_node)
            best_solution_found = False
            if s == 0:
                #optimum solution found.
                print("Optimum solution found.")
                print("Initial State: ")
                for row in initial_node.state:
                    print(row)
                print("")
                print("Board State: ")
                for row in random_node.state:
                    print(row)
                print("The time spent: " + str(elapsed))
                print("The number of nodes expanded during the search: " + str(expanded_node_number))
                print("Max number of nodes stored in the memory during the search: " + str(maxnode))
                print("All moves to get goal solution from the beginning: ")
                for move in random_node.moves:
                    print(move)
                break
                best_solution_found = True
            else:
                new_nodes = expand(random_node)
                expanded_node_number += 1

                #we are looking for the node that has no children. We want to go back to parents if this is an issue.
                if new_nodes==[]:
                    siblings = []
                    for node in frontier:
                        if node.parent == visited_node_array[-1].parent:
                            siblings.append(node)
                    if siblings == []:
                        while siblings != []:
                            parent = visited_node_array[-1].parent
                            for node in frontier:
                                if node.parent == parent.parent:
                                    siblings.append(node)
                            random_node = random.choice(siblings)
                    else:
                        random_node = random.choice(siblings)

                else:
                    try:
                        frontier.remove(random_node)
                    except ValueError:
                        pass
                    visited_node_array.append(random_node)
                    random_node = random.choice(new_nodes)
                    new_nodes.remove(random_node)
                    frontier.extend(new_nodes)

                #finding maximum node that is stored in frontier during all runtime.
                if (len(frontier) > maxnode):
                    maxnode = len(frontier)

                elapsed = time.time() - start

    except MemoryError:
        print("No solution found - Out of Memory")


    # TIME LIMIT
    if elapsed >= time_limit or best_solution_found == False:
        print("")
        #we assume that last visited node is the best solution initally, but after that
        #we'll search visited node array to find best solution.
        best_solution = visited_node_array[-1]
        remaining_peg=score(best_solution)

        for node in visited_node_array:
            if score(node) < remaining_peg:
                best_solution = node
                remaining_peg = score(node)

        #finding best solution's remaining pegs.
        remaining_peg = score(best_solution)
        print("No solution found - Time Limit. Sub-optimum Solution Found with " +str(remaining_peg)+ " remaining pegs.")

        print("Initial State: ")
        for row in initial_node.state:
            print(row)
        print("")
        #Printing board state of sub-optimum solution.
        print("New Board State: ")
        for row in best_solution.state:
            print(row)

        print("The time spent: " + str(elapsed))
        print("The number of nodes expanded during the search: " + str(expanded_node_number))
        print("Max number of nodes stored in the memory during the search: " + str(maxnode))
        print("All moves to get this solution from the beginning: ")
        for move in best_solution.moves:
            print(move)
```

## 5- Depth-First Search with a Node Selection Heuristic

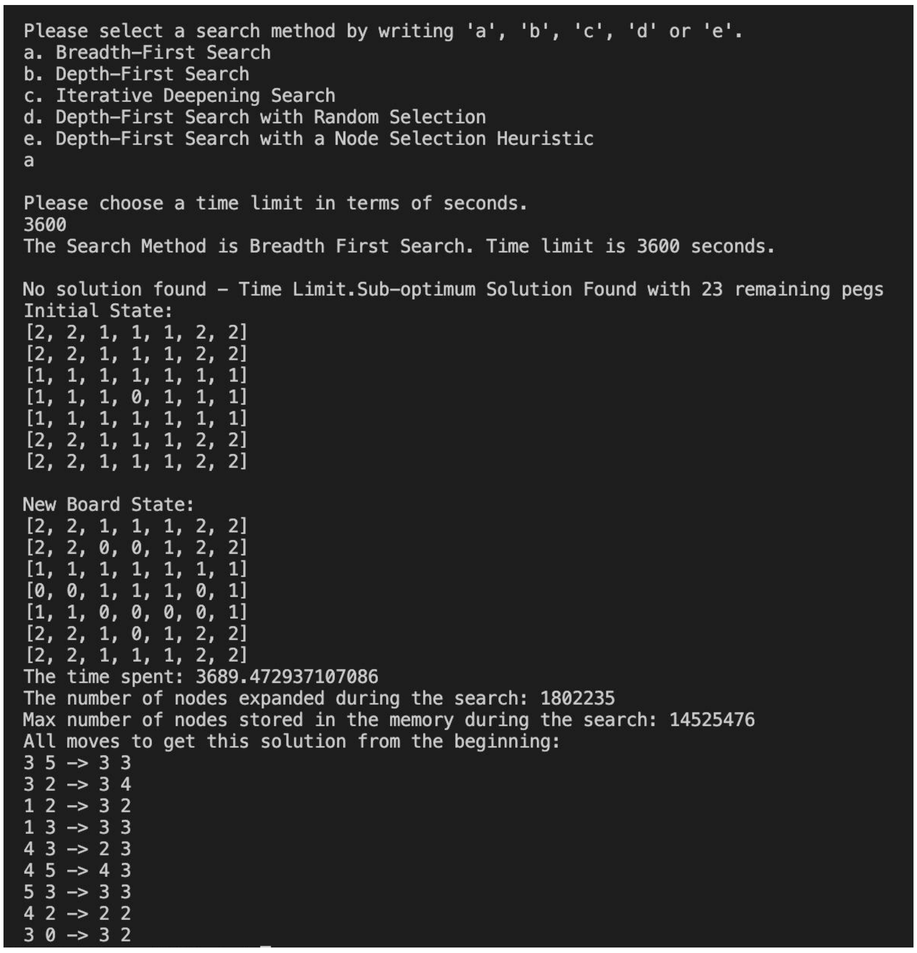
The idea behind our heuristic function is playing the pegs where at corners first if there is a move like that because if the peg remains in the corners towards the end of the game, then there will not be any move to remove that peg in the board. So, the function will look if there is any move from the corners, then that node will be selected first.



```python
#CHECKING IF THERE IS A CORNER PEG CAN BE SELECTED TO MAKE A MOVE.
def heuristic(new_nodes):
    for node in new_nodes:
        move = node.moves[-1]
        if ((move == str(0)+" "+ str(2)+ " -> " + str(0)+" "+ str(4)) or
            (move == str(0)+" "+ str(2)+ " -> " + str(2)+" "+ str(2)) or
            (move == str(0)+" "+ str(4)+ " -> " + str(0)+" "+ str(2)) or
            (move == str(0)+" "+ str(4)+ " -> " + str(2)+" "+ str(4)) or
            (move == str(2)+" "+ str(0)+ " -> " + str(2)+" "+ str(2)) or
            (move == str(2)+" "+ str(0)+ " -> " + str(4)+" "+ str(0)) or
            (move == str(2)+" "+ str(6)+ " -> " + str(2)+" "+ str(4)) or
            (move == str(2)+" "+ str(6)+ " -> " + str(4)+" "+ str(6)) or
            (move == str(4)+" "+ str(0)+ " -> " + str(4)+" "+ str(2)) or
            (move == str(4)+" "+ str(0)+ " -> " + str(2)+" "+ str(0)) or
            (move == str(4)+" "+ str(6)+ " -> " + str(4)+" "+ str(4)) or
            (move == str(4)+" "+ str(6)+ " -> " + str(2)+" "+ str(6)) or
            (move == str(6)+" "+ str(2)+ " -> " + str(6)+" "+ str(4)) or
            (move == str(6)+" "+ str(2)+ " -> " + str(4)+" "+ str(2)) or
            (move == str(6)+" "+ str(4)+ " -> " + str(6)+" "+ str(2)) or
            (move == str(6)+" "+ str(4)+ " -> " + str(4)+" "+ str(4))):


            #if there is a move comes from corner peg, then we took that node to end of the frontier
            #by removing and adding again at the end of the list.

            new_nodes.remove(node)
            new_nodes.append(node)

    return new_nodes
```

## Outputs

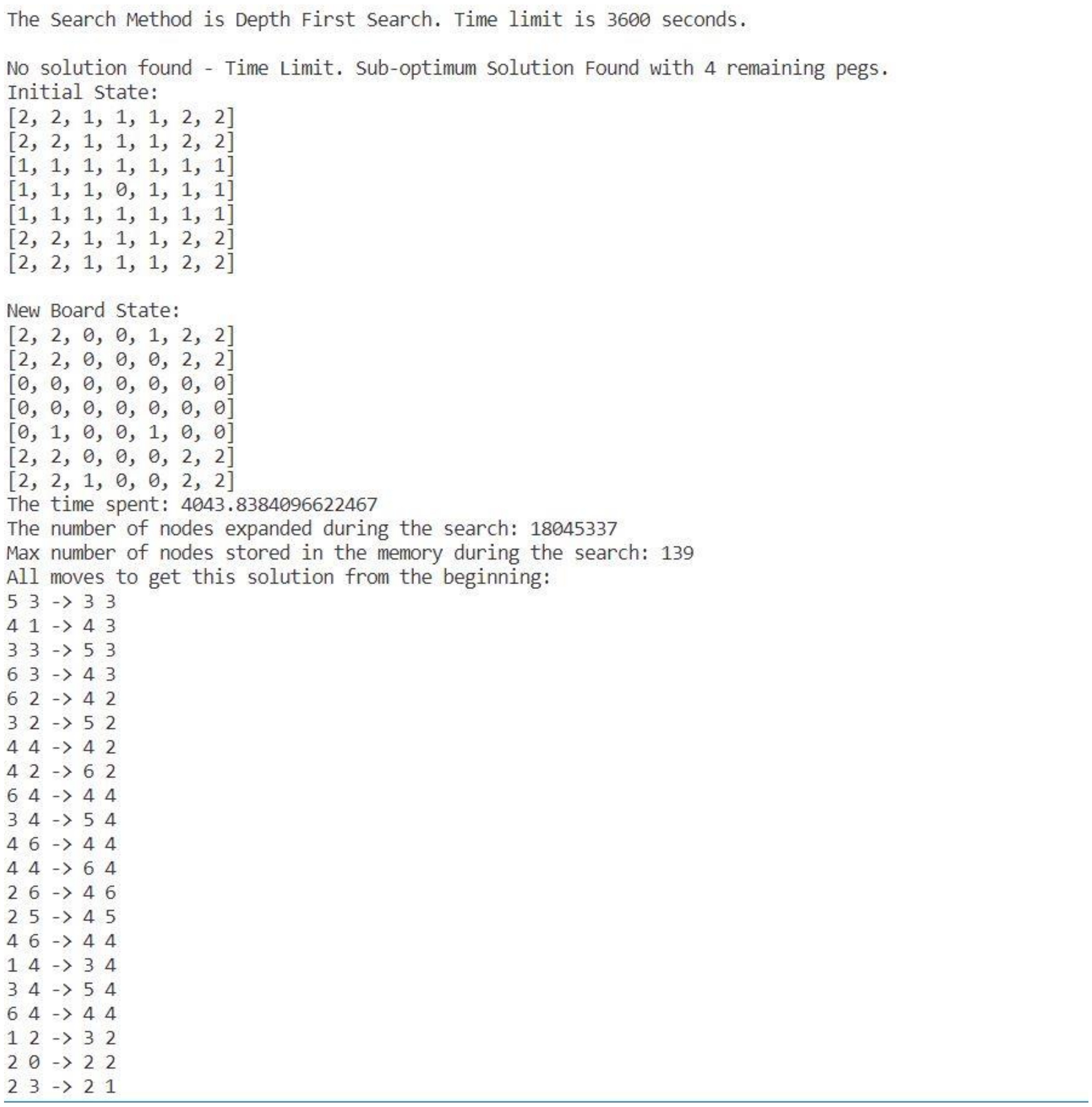
### 1- Breadth First Search



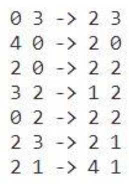
```
Please select a search method by writing 'a', 'b', 'c', 'd' or 'e'.
a. Breadth-First Search
b. Depth-First Search
c. Iterative Deepening Search
d. Depth-First Search with Random Selection
e. Depth-First Search with a Node Selection Heuristic
a

Please choose a time limit in terms of seconds.
3600
The Search Method is Breadth First Search. Time limit is 3600 seconds.

No solution found - Time Limit.Sub-optimum Solution Found with 23 remaining pegs
Initial State:
[2, 2, 1, 1, 1, 2, 2]
[2, 2, 1, 1, 1, 2, 2]
[1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 0, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1]
[2, 2, 1, 1, 1, 2, 2]
[2, 2, 1, 1, 1, 2, 2]

New Board State:
[2, 2, 1, 1, 1, 2, 2]
[2, 2, 0, 0, 1, 2, 2]
[1, 1, 1, 1, 1, 1, 1]
[0, 0, 1, 1, 1, 0, 1]
[1, 1, 0, 0, 0, 0, 1]
[2, 2, 1, 0, 1, 2, 2]
[2, 2, 1, 1, 1, 2, 2]
The time spent: 3689.472937107086
The number of nodes expanded during the search: 1802235
Max number of nodes stored in the memory during the search: 14525476
All moves to get this solution from the beginning:
3 5 -> 3 3
3 2 -> 3 4
1 2 -> 3 2
1 3 -> 3 3
4 3 -> 2 3
4 5 -> 4 3
5 3 -> 3 3
4 2 -> 2 2
3 0 -> 3 2
```

The photo above is the output we get when we try to solve the peg solitaire with the breadth first search algorithm. For the reasons we mentioned earlier, the algorithm could not find the optimum solution. It could only go down to the 10th depth. We kept track of how many nodes were expanded and how many nodes were kept in memory during the search. They all appear in the photo above.

## 2- Depth First Search



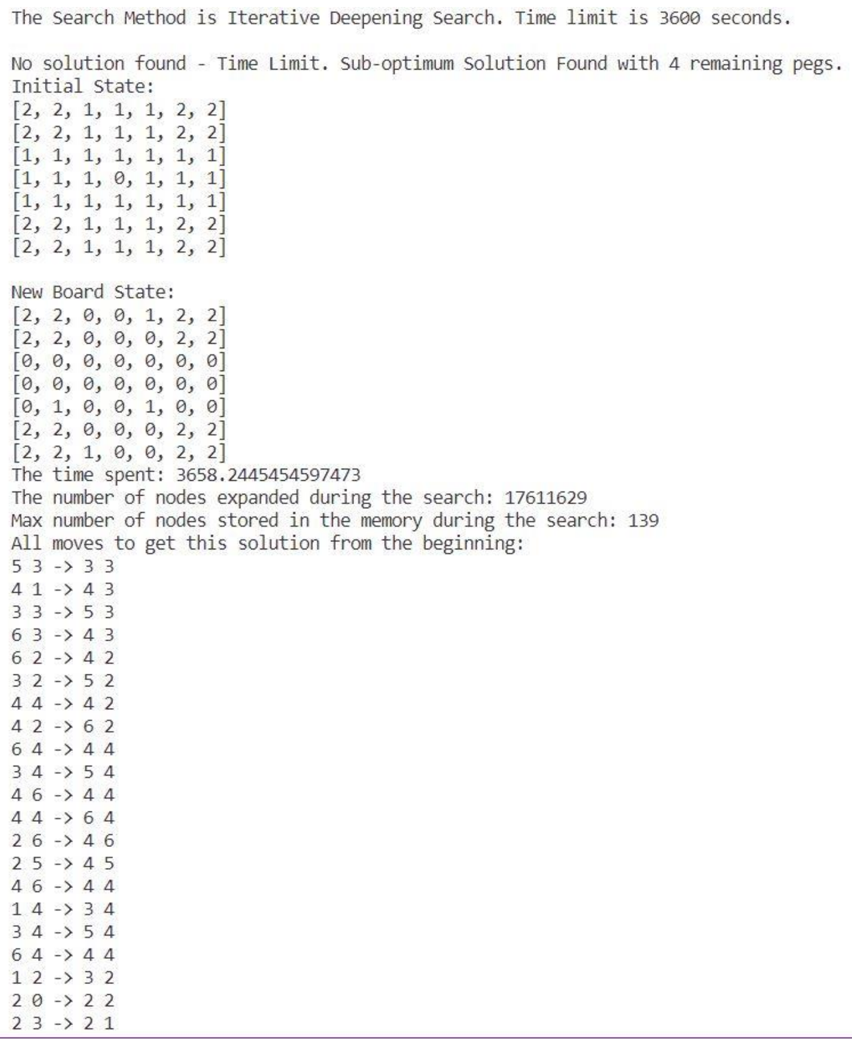
```
The Search Method is Depth First Search. Time limit is 3600 seconds.

No solution found - Time Limit. Sub-optimum Solution Found with 4 remaining pegs.
Initial State:
[2, 2, 1, 1, 1, 2, 2]
[2, 2, 1, 1, 1, 2, 2]
[1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 0, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1]
[2, 2, 1, 1, 1, 2, 2]
[2, 2, 1, 1, 1, 2, 2]

New Board State:
[2, 2, 0, 0, 1, 2, 2]
[2, 2, 0, 0, 0, 2, 2]
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 1, 0, 0]
[2, 2, 0, 0, 0, 2, 2]
[2, 2, 1, 0, 0, 2, 2]
The time spent: 4043.8384096622467
The number of nodes expanded during the search: 18045337
Max number of nodes stored in the memory during the search: 139
All moves to get this solution from the beginning:
5 3 -> 3 3
4 1 -> 4 3
3 3 -> 5 3
6 3 -> 4 3
6 2 -> 4 2
3 2 -> 5 2
4 4 -> 4 2
4 2 -> 6 2
6 4 -> 4 4
3 4 -> 5 4
4 6 -> 4 4
4 4 -> 6 4
2 6 -> 4 6
2 5 -> 4 5
4 6 -> 4 4
1 4 -> 3 4
3 4 -> 5 4
6 4 -> 4 4
1 2 -> 3 2
2 0 -> 2 2
2 3 -> 2 1
```



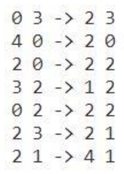
```
0 3 -> 2 3
4 0 -> 2 0
2 0 -> 2 2
3 2 -> 1 2
0 2 -> 2 2
2 3 -> 2 1
2 1 -> 4 1
```

After one hour searching, the best solution that depth first search found is that 4 peg remains. The moves to get new board state are listed above also. Time spent value is larger than 3600 seconds because it lost some time when it looks for best results in visited node array. So, 3600 seconds for searching and the remaining time is selecting the best solution so far if goal state cannot be found.

## 3- Iterative Deepening Search



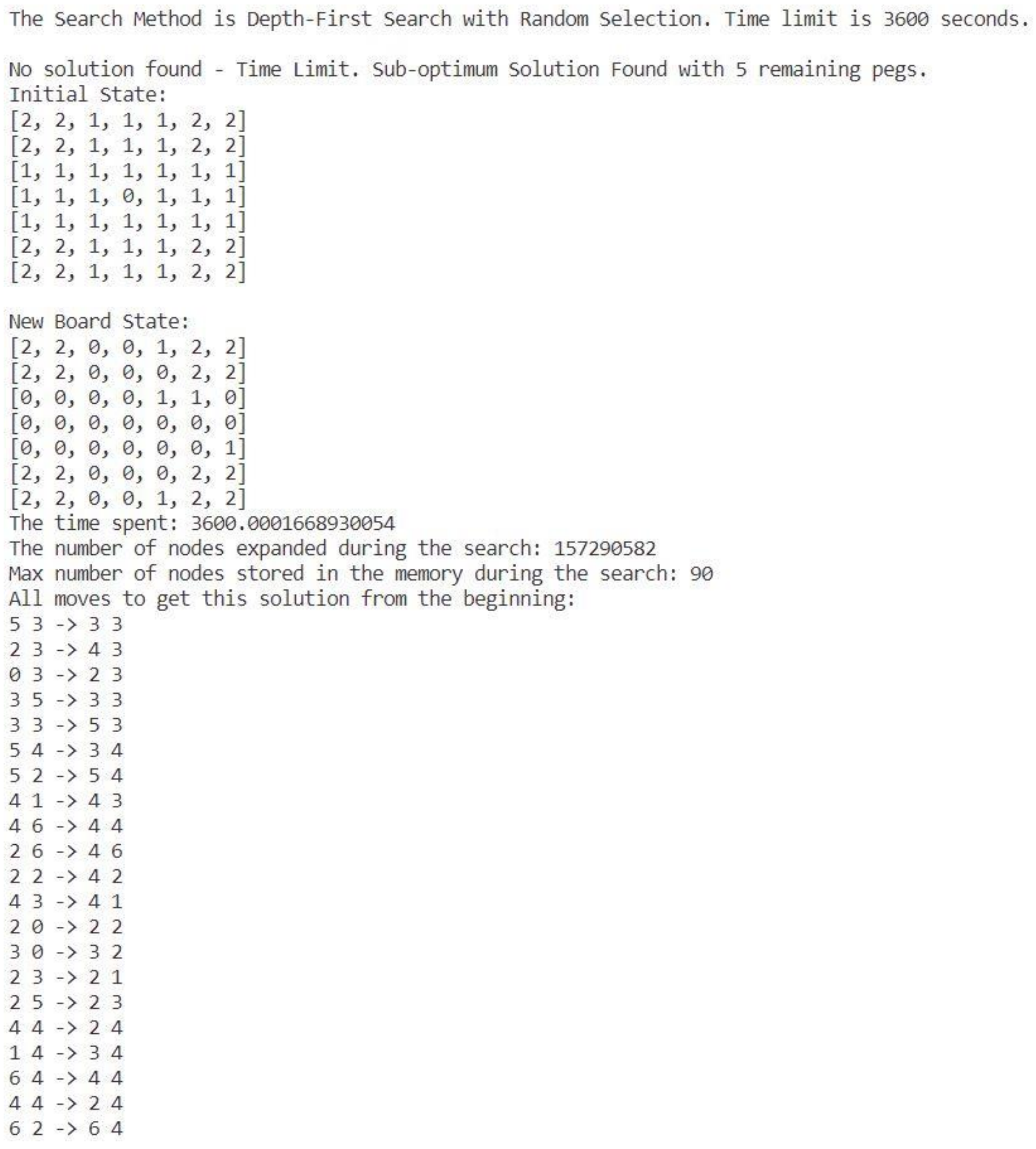
```
The Search Method is Iterative Deepening Search. Time limit is 3600 seconds.

No solution found - Time Limit. Sub-optimum Solution Found with 4 remaining pegs.
Initial State:
[2, 2, 1, 1, 1, 2, 2]
[2, 2, 1, 1, 1, 2, 2]
[1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 0, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1]
[2, 2, 1, 1, 1, 2, 2]
[2, 2, 1, 1, 1, 2, 2]

New Board State:
[2, 2, 0, 0, 1, 2, 2]
[2, 2, 0, 0, 0, 2, 2]
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 1, 0, 0]
[2, 2, 0, 0, 0, 2, 2]
[2, 2, 1, 0, 0, 2, 2]
The time spent: 3658.2445454597473
The number of nodes expanded during the search: 17611629
Max number of nodes stored in the memory during the search: 139
All moves to get this solution from the beginning:
5 3 -> 3 3
4 1 -> 4 3
3 3 -> 5 3
6 3 -> 4 3
6 2 -> 4 2
3 2 -> 5 2
4 4 -> 4 2
4 2 -> 6 2
6 4 -> 4 4
3 4 -> 5 4
4 6 -> 4 4
4 4 -> 6 4
2 6 -> 4 6
2 5 -> 4 5
4 6 -> 4 4
1 4 -> 3 4
3 4 -> 5 4
6 4 -> 4 4
1 2 -> 3 2
2 0 -> 2 2
2 3 -> 2 1
```



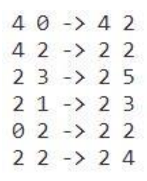
```
0 3 -> 2 3
4 0 -> 2 0
2 0 -> 2 2
3 2 -> 1 2
0 2 -> 2 2
2 3 -> 2 1
2 1 -> 4 1
```

Iterative deepening search found exactly same solution with depth first search because both algorithms are looking exactly same way except iteration feature of iterative deepening search. So, this algorithm can be a good way to find solution of peg solitaire.

## 4- Depth First Search with Random Selection



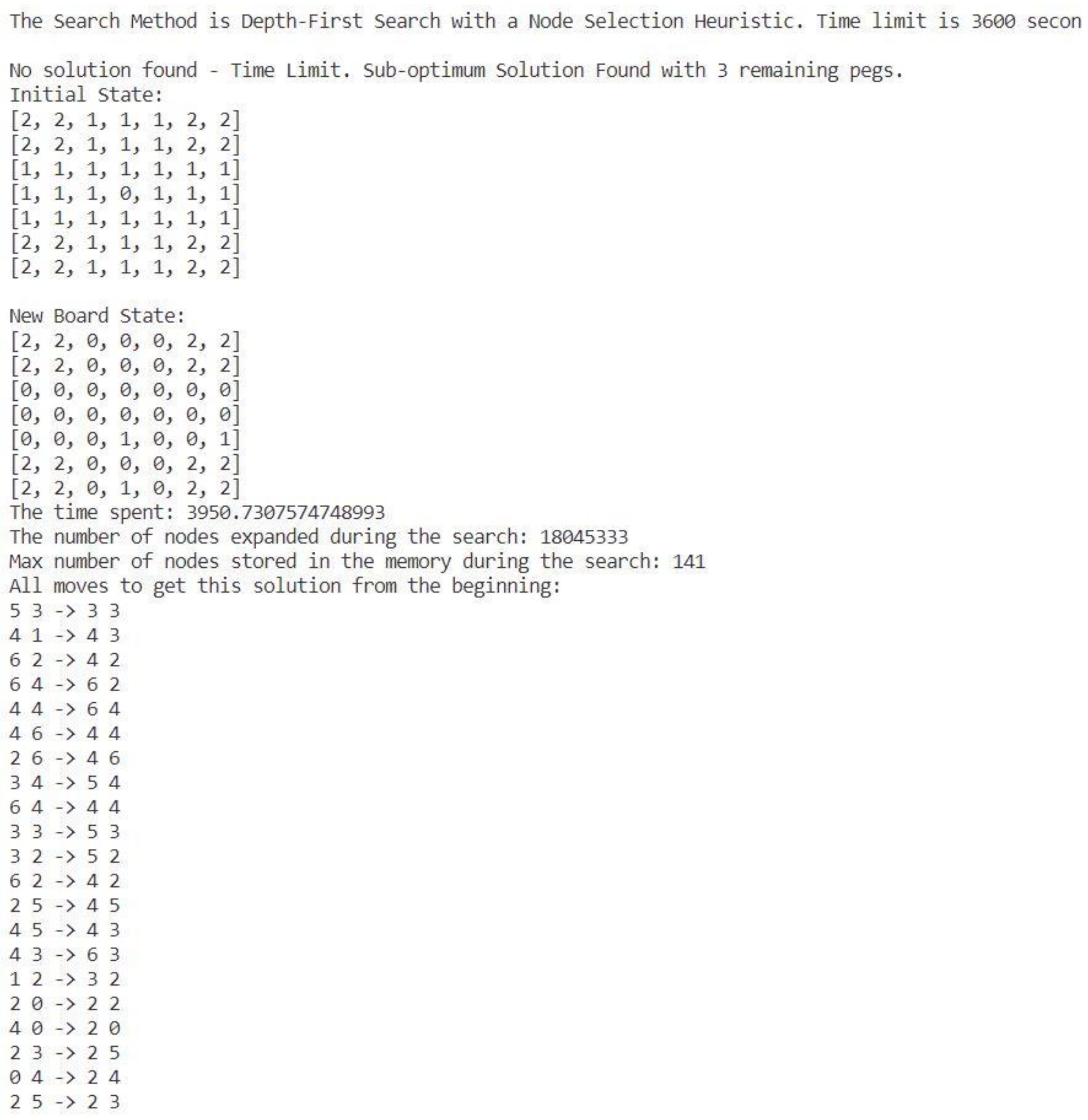
```
The Search Method is Depth-First Search with Random Selection. Time limit is 3600 seconds.

No solution found - Time Limit. Sub-optimum Solution Found with 5 remaining pegs.
Initial State:
[2, 2, 1, 1, 1, 2, 2]
[2, 2, 1, 1, 1, 2, 2]
[1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 0, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1]
[2, 2, 1, 1, 1, 2, 2]
[2, 2, 1, 1, 1, 2, 2]

New Board State:
[2, 2, 0, 0, 1, 2, 2]
[2, 2, 0, 0, 0, 2, 2]
[0, 0, 0, 0, 1, 1, 0]
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1]
[2, 2, 0, 0, 0, 2, 2]
[2, 2, 0, 0, 1, 2, 2]
The time spent: 3600.0001668930054
The number of nodes expanded during the search: 157290582
Max number of nodes stored in the memory during the search: 90
All moves to get this solution from the beginning:
5 3 -> 3 3
2 3 -> 4 3
0 3 -> 2 3
3 5 -> 3 3
3 3 -> 5 3
5 4 -> 3 4
5 2 -> 5 4
4 1 -> 4 3
4 6 -> 4 4
2 6 -> 4 6
2 2 -> 4 2
4 3 -> 4 1
2 0 -> 2 2
3 0 -> 3 2
2 3 -> 2 1
2 5 -> 2 3
4 4 -> 2 4
1 4 -> 3 4
6 4 -> 4 4
4 4 -> 2 4
6 2 -> 6 4
```



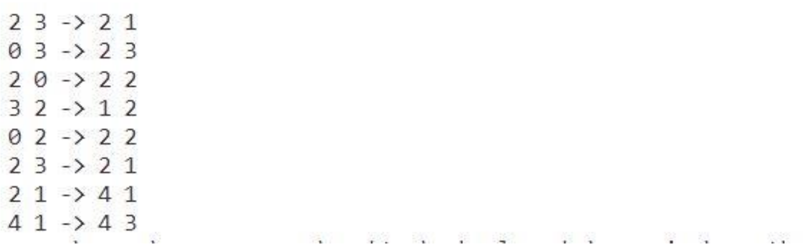
```
4 0 -> 4 2
4 2 -> 2 2
2 3 -> 2 5
2 1 -> 2 3
0 2 -> 2 2
2 2 -> 2 4
```

Since the random depth first search algorithm will randomly choose the node to which it will go each time, it is possible to get a different result each time. It gave a worse result than depth first search as we run it now, but it may give a better result the next time we run it because it is completely random.

## 5- Depth First Search with a Node Selection Heuristic



```
The Search Method is Depth-First Search with a Node Selection Heuristic. Time limit is 3600 secon

No solution found - Time Limit. Sub-optimum Solution Found with 3 remaining pegs.
Initial State:
[2, 2, 1, 1, 1, 2, 2]
[2, 2, 1, 1, 1, 2, 2]
[1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 0, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1]
[2, 2, 1, 1, 1, 2, 2]
[2, 2, 1, 1, 1, 2, 2]

New Board State:
[2, 2, 0, 0, 0, 2, 2]
[2, 2, 0, 0, 0, 2, 2]
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 1]
[2, 2, 0, 0, 0, 2, 2]
[2, 2, 0, 1, 0, 2, 2]
The time spent: 3950.7307574748993
The number of nodes expanded during the search: 18045333
Max number of nodes stored in the memory during the search: 141
All moves to get this solution from the beginning:
5 3 -> 3 3
4 1 -> 4 3
6 2 -> 4 2
6 4 -> 6 2
4 4 -> 6 4
4 6 -> 4 4
2 6 -> 4 6
3 4 -> 5 4
6 4 -> 4 4
3 3 -> 5 3
3 2 -> 5 2
6 2 -> 4 2
2 5 -> 4 5
4 5 -> 4 3
4 3 -> 6 3
1 2 -> 3 2
2 0 -> 2 2
4 0 -> 2 0
2 3 -> 2 5
0 4 -> 2 4
2 5 -> 2 3
```



```
2 3 -> 2 1
0 3 -> 2 3
2 0 -> 2 2
3 2 -> 1 2
0 2 -> 2 2
2 3 -> 2 1
2 1 -> 4 1
4 1 -> 4 3
```

With heuristic function, our algorithm has improved, but still cannot find goal state in one hour. 3 pegs remained after one hour search and this is the best solution among other four algorithms. Probably, depth first search with a node selection heuristic algorithm can find goal state in more than one hour. So, we think looking for corner moves is not a bad idea.