HO CHI MINH CITY NATIONAL UNIVERSITY

POLYTECHNIC UNIVERSITY

FACULTY OF COMPUTER SCIENCE - COMPUTER ENGINEERING

ARTIFICIAL INTELLIGENCE SUBJECT

Big exercise 1

Solve Bloxorz game using Searching algorithms

GVHD: Vuong Ba Thinh

Student: Nguyen Thanh Hung - 1511358

Pham Qui Luan - 1511899

Tran Manh Hoang - 1511150

Nguyen Duy Dao - 1510654

Phan Viet Duc - 1510809

City. HO CHI MINH CITY, MAY 2018

Table of contents

List of drawings

1 Summary

The game bloxorz is an entertaining game that involves moving a block of size 2x1x1 from its location into a target hole, guaranteed to always find a path from the initial position to the origin. In this big exercise, our team will implement an artificial intelligence tool to find solutions for the game using 2 classic algorithms, Breadth First Search and Depth First Search, and 1 Heuristic algorithm, Best-First- Search, with the price function implemented by calculating distance. In this report, our team will describe the approach to the problem, necessary definitions, and implemented algorithms. The code in the report is implemented in python, or is pseudocode. Finally, there is the results section, which shows the level of time and memory consumption, along with the good level of the solution.

2 Identify the problem

2.1 Introduction to the game Bloxorz Bloxorz is

a quiz game developed by Damien Clarke and released on June 21, 2007.
The goal of the game is to move 1 block (rectangular block) to the destination, on the map containing the bricks. There are a total of 33 stages to complete the game. The player moves the block using the arrow keys, and does not let it fall off the map.

There are bridges and switches placed in many levels. Switches are activated when blocked. There are 2 types of switches, Heavy X-shaped and Soft O-shaped. The soft switch is activated when any part of the block is above it. The heavy switch is activated when the entire block is above it. When activated, the switches will have different actions. Some will only turn on the bridge, some will only close the bridge, others will do both.

Orange bricks will be more fragile than the other bricks. If the block stands above
That brick, the brick will fall out and the block will fall off the map.

Finally, the third type of switch is teleport: ( ), when the block standing on it will be split into two smaller blocks and moved to two different locations. Both can be controlled by pressing the space button to move to another block. They will be reattached if one block is next to another. Small blocks can still trigger the O-shaped switch, but they are not 'heavy' enough to trigger the X-shaped switch. Small blocks cannot pass through the target cell, only complete blocks can do this.

2.2 Definition of the problem

2.2.1 State space

Is the set of possible states of a block on the map, no part of the block falls off the map. A state is defined including the following attributes: coordinates of block, board, rotation, parent.

In which the coordinates of the block are stored in tupple (x,y) which is the coordinates of the top - left point of the block in any state. Board is the variable that stores the state of the map. This state saving step has a huge weakness: it consumes memory, but it solves many problems that will be mentioned below. Rotation is the existence state of the block, there are 4 states: ST ANDING, LAY ING_X, LAY ING_Y , SP LIT. ST ANDING state is assigned when two parts of the block have the same coordinates, LAY ING_X state is assigned when two parts of the block are on the same row, LAY ING_Y state is assigned when two parts of the block are on the same column, state SP LIT is determined when the block is split into two smaller blocks, at this time tuple (x, y) will store the position of one small block, and tuple (x1, y1) will store the position of the other block, (x1, y1) is initialized when the state is initialized, but is only considered when rotation is SP LIT, luckily the team chose this storage method to handle the teleport switch situation, even though the team had previously Once decided to create a new state for the child block, this will cause trouble when managing. The Parent element is a reference to the parent state so we know which state led to the current state, which helps solve the problem of finding a path to victory.

The above idea of defining a state is implemented in the code as follows:

```
class Block:


    def __init__(self, x, y, rot, parent, board, x1=None,y1=None):
        self.x          = x

        self.y
        self.rot        =y=rot

        self.parent = parent
        self.board = copy.deepcopy(board)
        self.x1         = x1

        self.y1         = y1
```

2.2.2 Introducing map files

Map file is a text file that provides information about the starting state, ending state, coordinates of bricks on the map, location of switches and bridges as well as information about the bridges that the switch manages. Depending on each map, the initial coordinates are different, the initial rotation is ST ANDING, parent is None, board is a python 2-dimension list representing the state of the map. There are a total of 33 map files from map01.txt to map33.txt located in the map/ directory. For example, map05.txt will have the following content.

```
10 15 13 1
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
0 1 1 1 1 1 1 1 5 1 1 1 1 1 1
0 1 1 1 1 0 0 0 0 0 0 0 1 1 1
0 1 1 6 1 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 4 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 1 1 1 1 5
1 1 1 0 0 0 0 0 0 0 1 1 1 1 1
1 9 1 1 1 1 1 1 1 1 1 1 1 0 0
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
8 1 2 1 6 1 5
3 3 2 8 5 8 6
6 5 2 8 5 8 6
14 6 2 8 5 8 6
```

In which 10 15 13 1 indicates that the map has 15 columns, 10 rows, the starting position is at coordinates (x, y) = (13, 1). The x axis is assigned to the positive horizontal row to the right, the y axis is assigned to the positive vertical column facing down, the coordinates of the top left point are (0,0).

The next ten rows show the map of the initial state. It is specified as follows:

- 0: Is the position without any bricks. If any part of the block falls here, the game will end in a losing state.
- 1: Is the position of normal bricks.
- 2: Is the position of the orange bricks, only allowing part of the block to lie on it,
  Do not allow blocks to stand on it.
- 3: Is the position of the X-shaped switch
- 4: Is the position of the O-shaped switch (only allows closing the bridge)
- 5: Is the position of the O-shaped switch
- 6: Is the position of the O-shaped switch (only allows opening the bridge)
- 7: Is the location of the teleport switch
- 8: Is the position of the X-shaped switch (only allows opening bridges)
- 9: Is the winning position

The remaining rows will show ManaBoa (a list of switch coordinates, and the coordinates managed by that switch), in the following format:

(x, y) whether <iX, iY>

For example:

- 8 1 2 1 6 1 5 indicates that the switch has coordinates (1,8) managing 2 locations (1,6) and (1,5).

- 3 3 2 8 5 8 6 indicates that the switch has coordinates (3,3) managing 2 locations (8,5) and (8,6).
- 6 5 2 8 5 8 6 indicates that the switch has coordinates (5,6) managing 2 locations (8,5) and (8,6).

At first, because of the approach to solving each stage, the team only thought there were less than 10 cases that needed to be encoded into numbers, so the O-shaped was divided into 3 cases 4,5,6 as above, but at different levels. After that, the number of cases that need to be resolved is much higher than that. If you encode the case with a 2-digit number, it will result in a map file that is difficult to read. Therefore, the group combines some cases as follows:

Instead of the initial format being (x, y) num <iX, iY>, when the program reads the position O-shaped (number 5), and X-shaped (number 3) will be read in the following format:

(x, y) numToggle <xT, yT> numClose <xC, yC> numOpen <xO, yO>

In which, numToggle is the number of bridges whose state is flipped each time the switch is activated, numClose is the number of bridges that are only closed, and numOpen is the number of bridges that are only opened. For example:

- 13 0 4 7 2 7 3 1 4 1 5 0 0 indicates that the switch with coordinates (0,13) has a bridge toggle count of 4, which is (7,2) (7,3) (1,4) (1,5), and numClose = 0, numOpen = 0, so there is no list of these coordinates.

2.2.3 Initial state The initial state is

the first state of the block when the stage starts, it is determined from the first line of the map file passed into the program. Initial rotation is ST ANDING, parent is None.

2.2.4 Target state

The target state is the state where the block stands above the target point. That means coordinates (x, y) = (xGoal, yGoal), rotation is ST ANDING.

The goal state is defined in the code as follows:

```
def isGoal(block):

    # local definition x =
    block.x y =
    block.y rot =
    block.rot
    board = block.board

    # statements
    if rot == "STANDING" and board[y][x] == 9:
        return True
    else:
        return False
```

2.2.5 Valid transitions From a state we

can generate transitions by moving the block up, down, left or right. In the rotations ST ANDING, LAY ING_X, LAY ING_Y, 4 transitions will be generated. But in rotation SP LIT, 8 moves are generated because each child block will generate 4 moves.

Move up and down for rotation ST ANDING, LAY ING_X, LAY ING_Y is defined as follows:

```
def move_up(self):

    newBlock = Block(self.x, self.y, self.rot, self, self.board)

    if self.rot == "STANDING":
        newBlock.y -= 2
        newBlock.rot = "LAYING_Y"

    elif newBlock.rot == "LAYING_X": newBlock.y
        -= 1

    elif newBlock.rot == "LAYING_Y": newBlock.y
        -= 1 newBlock.rot =
        "STANDING"

    return newBlock
```

```
def move_down(self):

    newBlock = Block(self.x, self.y, self.rot, self, self.board)

    if newBlock.rot == "STANDING":
        newBlock.y += 1
        newBlock.rot = "LAYING_Y"

    elif newBlock.rot == "LAYING_X": newBlock.y
        += 1

    elif newBlock.rot == "LAYING_Y": newBlock.y
        += 2 newBlock.rot =
        "STANDING"

    return newBlock
```

The upward move for rotation SP LIT is defined as follows:

```
def split_move_up(self):
        newBlock = Block(self.x, self.y, self.rot, self, self.board, self.x1, self.y1)

        newBlock.y -= 1
        return newBlock
```

```
def split1_move_up(self):
        newBlock = Block(self.x, self.y, self.rot, self, self.board, self.x1, self.y1)

        newBlock.y1 -= 1
        return newBlock
```

3 Search strategies to solve the problem

There are three strategies used to solve the problem, although there are significant differences in time and memory consumption. But it still basically follows the same development path. Initially, the team implemented the Depth-First-Search algorithm to solve each stage one by one. This process took quite a long time because after each stage there were new events, and sometimes the data structure had to be changed. to save the state. After completing 33 stages using the Depth-First-Search algorithm, switching to the Breadth-First-Search algorithm only takes a very short time. Finally, there is the Heuristic algorithm. Initially, the group chose the Hill Climbing algorithm to do but because they found it more difficult than Best-First-Search in that they did not know how to choose the appropriate evaluation function. So the team switched to implementing the Best-First-Search algorithm with an evaluation function that calculates the distance from the current state to the target state. After implementing and viewing the results, the team found that there were some significant advantages and disadvantages, which will be detailed in the implementation section.

3.1 Depth-First-Search algorithm Because the team

used the Depth-First-Search algorithm to approach the problem from the beginning, in this section, the team will outline the approach to solve each stage, with attachments. Include code in file ai.py and explanation.

Consider the following pseudocode implementing the Depth-First-Search algorithm:

```
# solve DFS python pseudo code def
DFS(block):

    Stack.put(block)

    while Stack:
        current = Stack.pop() if
        isGoal(current):
            return SUCCESS
        else:
            Stack.put(current.all_possible_move())
    return False
```

The team used a while loop to implement the algorithm instead of using the recursive method. The advantage of implementing the code using a loop is that it is easier to manage code and risks, and fixing programming errors also becomes much easier. . The algorithm starts from adding the start state to the Stack and running the loop. In the loop body, we take the first word from the Stack and see if it is the target state. The program will stop if this condition is true. If this condition is false, the program will browse through the loops in turn. possible logical transition of current, and put this entire transition on the Stack and iterate the next time.

Below is a code snippet that implements the algorithm in python code, with some changes added. Saving the passed states into the passState variable helps us avoid the program repeating the passed states over and over again. Although it consumes memory and time to compare each time, it is more optimal in terms of the number of reasonable moves, avoiding the trap of moving too many unreasonable times. The virtualStep variable is a variable to store the total number of states that the program actually has to go through to get to the destination. The goal is to calculate the memory consumed by the algorithm, and the complexity of the algorithm when finding the solution. , the number of virtualSteps is recorded in the Time column in Figure 1 Table of data on time and memory consumption of each algorithm for each stage.

```
# solve DFS
def DFS(block):

    # local definitions
    board = block.board
    Stack = []
    Stack.append(block)
    passState.append(block)
    virtualStep = 0

    # statements
    while Stack:
        current = Stack.pop()

        if isGoal(current):
            printSuccessRoad(current)
            print("COMSUME", virtualStep, "VIRTUAL STEP")
            print("SUCCESS")
            return True
        else:
            if current.rot != "SPLIT":
                virtualStep += 4
                move(Stack,current.move_up(), "up")
                move(Stack,current.move_right(), "right")
                move(Stack,current.move_down(), "down")
                move(Stack,current.move_left(), "left")
            else:
                virtualStep += 8
                move(Stack,current.split_move_left(), "left0")
                move(Stack,current.split_move_right(), "right0")
                move(Stack,current.split_move_up(), "up0")
                move(Stack,current.split_move_down(), "down0")

                move(Stack,current.split1_move_left(), "left1")
                move(Stack,current.split1_move_right(), "right1")
                move(Stack,current.split1_move_up(), "up1")
                move(Stack,current.split1_move_down(), "down1") return False
```

In the body of the move function, it will consider whether the move is reasonable (through the isValidBlock function) and see if it has been reviewed before (through the isVisited function). If not, we will proceed to review this status. The move function is defined as follows:

```
def move(Stack, block, flag): if
        isValidBlock(block):
                if isVisited(block):
                        return None
                Stack.append(block)
                passState.append(block) return
                True
        return False
```

The processing steps for switches are implemented in the isValidBlock function body. Pseudocode
Its function is as follows (because its function body is too long, it is not included in the report).

```
def isValidBlock(block): if
        isFloor(block): if
                        isSwitch(block):
                                #handle
                return True
        return False
```

Another notable point is the group's isVisited function. Initially, a state only stores coordinates and rotation, to see if a state has been traversed, just compare the coordinates and rotation. But in the following stages (stage 8), it is necessary to go back to the state that has already been reviewed. The problem cannot be solved at this time. The team found many workarounds such as allowing the program to cycle through already visited nodes, but it leads to failure when the program gets trapped in an endless loop, to solve the endless loop it is possible to bind depth of the Depth-First-Search tree, but then the algorithm no longer makes sense. Another approach is to save the map in each stage, the advantage of this approach is that it solves the problem above, and each of its states is now unique and must not be repeated. The biggest and most significant disadvantage of this solution is that it consumes a large amount of memory. For each state, we must save a map matrix. And it wastes a lot of time when we check to see if this status has been approved or not. However, this drawback has been accepted to complete the algorithm.

3.2 Breadth-First-Search algorithm After successfully

implementing the Depth-First-Search algorithm, switching to the Breadth-First-Search algorithm is very easy by instead of traversing the tree using Stack, we will Use Queue. Let's consider the following pseudocode:

```
# solve BFS python pseudo code def
BFS(block):

    Queue.enqueue(block)

    while Queue:
        current = Queue.dequeue() if
        isGoal(current):
            return SUCCESS
        else:
            Queue.enqueue(current.all_possible_move())
    return False
```

3.3 Best-First-Search algorithm

Let's consider the following pseudocode:

```
# solve Best-first-search python pseudo code def BEST(block):

    BestQueue = PriorityQueue() startEval
    = evalFunction(block)
    BestQueue.enqueue((startEval, block))

    # until priority queue is empty while
    BestQueue.not_empty: item =
        BestQueue.dequeue() # item = (block, distance) iDista = item[0] iBlock = item[1]

        if isGoal(iBlock):
            return SUCCESS

        BestQueue.enqueue(iBlock.all_possible_state())
```

In which the evalFunction function is the price function of the algorithm, any state is evaluated by measuring the distance from it to the target state. The team uses formula 2 to calculate the distance in the price function, Pythagorean: distance = (x1 ÿ x2) + (y1 ÿ y2)^2 to avoid because wasting time calling the root function, the given group returns distance2 pseudo code like or:

```
def evalFunction(block): # we
    read board to get goal goal =
    block.getGoal() curr =
    block.getCurr() return
    distance(goal,curr)
```

4 Execution time and memory consumption metrics

The source code is run to test the execution time taken on the running VMWARE virtual machine
Linux operating system installed on Windows 7 64-bit computer, Processor Intel(R) Core(TM)
i3-2328M CPU 2.20GHz. RAM have 5.89GB usable.

4.1 Data table for algorithms

The data table on time and memory consumption for each algorithm is presented as follows:

Stage	Depth-First-Search Time			Breadth-First-Search			Best-First-Search		
	Step Memory Time Step Memory Time Step Memory								
01 0.016s 156 0.049s 30296		18				296 0.012s		10	
02 0.017s 108			848 0.020s		18	1060 0.009s		32	1436
03 0.012s 04		55	340 0.017s 360		17	428 0.027s 276		26	452
0.012s		35	0.020s		29	0.011s		29	304
05 0.009s 06		92	444 0.005s 176		34	1264 0.009s 448		42	1228
0.005s 07		36	0.005s 472		36	0.004s 716		40	380
0.025s		55	0.016s		45	0.017s		51	708
08 0.017s 39 09 0.017s			460 0.037s 2644		11	4228 0.020s 5596		14	9300
269 10 0.024s 954			0.042s 8848		25	0.017s 53 45712 0.044s			4700
			0.165s		58	199			29352
11 0.012s 48 12 0.017s			668 0.008s 1144		48	744 0.025s 1296		48	632
130 54			0.028s		66	0.041s		68	1220
13 0.008s			596 0.008s		47	532 0.016s		51	472
14 0.008s 114 15 0.005s			1276 0.024s 1600		68	1840 0.028s		72	1252
176 57			0.240s		58	33224 2.941s 684		71	119048
16 0.009s			3032 0.012s		25	0.053s		25	3240
17 0.021s 338 18 0.009s			2892 0.040s 107 1528			3932 0.185s 158 2392			3960
161			0.036s 86			0.070s 109			2356
19 0.009s		72	596 0.016s		68	892 0.089s		68	880
20 0.028s 260 21 0.025s			5868 0.171s 612		57	32280 6.392s 936		111	158344
73			0.024s		72	0.119s		73	1232
22 0.017s 150 23 0.028s			1432 0.022s		66	1304 0.021s 81 29560			1376
413 24 0.008s 59			12632 0.097s 636		76	1.822s 100 2184 0.042s 58			28616
			0.036s		58				1712
25 0.012s 95 26 0.084s			708 0.009s 56 34908			1896 0.024s 56 75188			1556
1159 27 0.016s 74			0.317s 105 356 0.024s 72			0.482s 126 1256 0.020s 74			53124
									936
28 0.084s 781 29 0.008s			31812 0.345s		101	52520 0.076s 131 7248			11280
132			5960 0.025s 105 1112			0.040s 121 2224 0.012s			4448
30 0.032s 115			0.024s 115			115			1448
31 0.008s 221 32 0.016s			2256 0.024s 92 1220 0.028s			3504 0.045s 133 2520			4372
204			130 2784 0.057s			0.025s 130			2108
33 0.012s		82			66	2952 0.020s		77	2768

Figure 1: Data table on time and memory consumption

Some explanations:

- The Time column is the CPU time spent in the kernel for the current process, extracted from the sys row in the time command. For example: "time python ai.py 01 DFS" the time will be about 0.016s.

- Step is the number of steps needed by the algorithm to get from the starting position to the target state.

- Memory is the total number of nodes that the algorithm must go through before finding the target state. The memory needed for the algorithm is calculated according to the formula: AmountOfMem = Memory ÿ200bytes. In which 200 bytes is the estimated capacity to store a state including coordinates, rotation, map, and pointer to the parent element.

4.2 Visual graphs comparing algorithms Below are some visual graphs

showing the differences of algorithms in the first 10 stages. In Figure 2 is a graph showing the difference in time consumption. Figure 3 compares the number of moves to the target state found by the algorithms. Figure 4 compares the memory consumption of each algorithm.

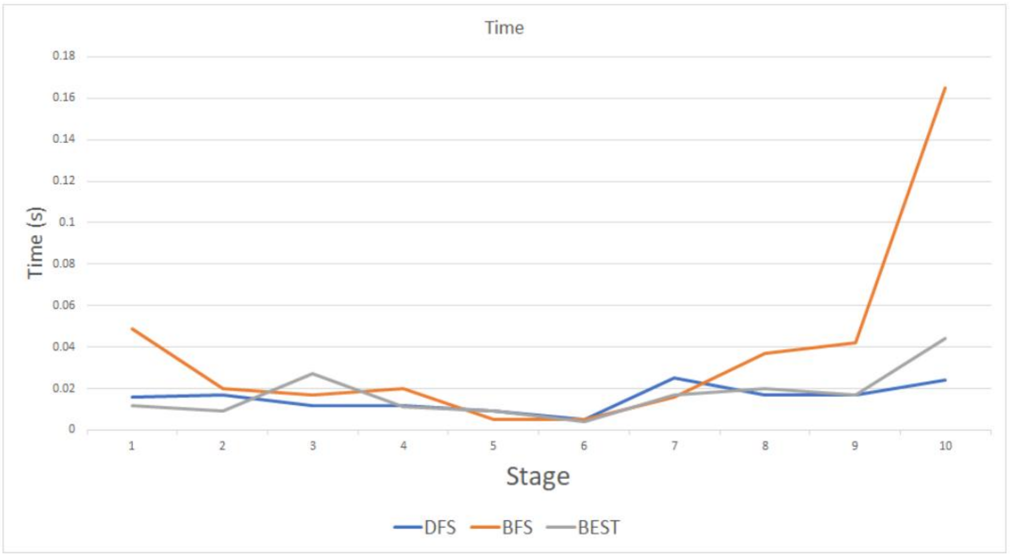

Figure 2: Comparison of running time of 3 algorithms in the first 10 stages

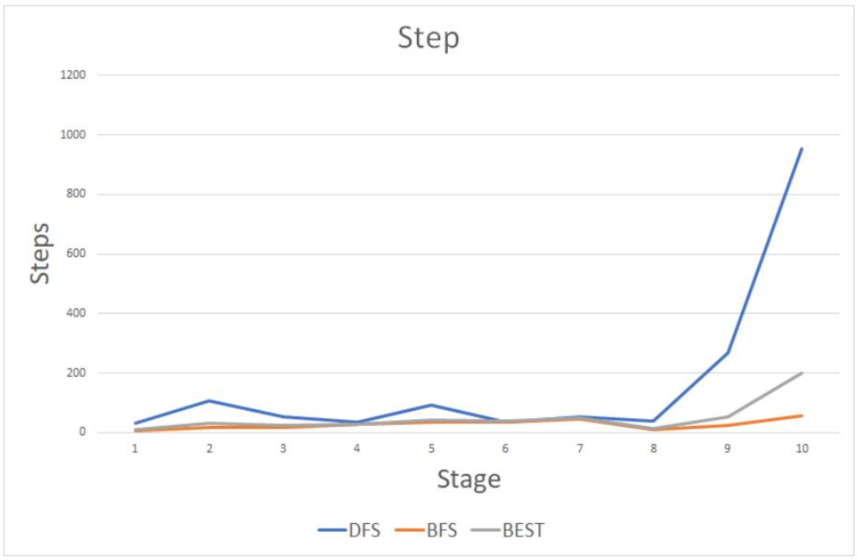

Figure 3: Comparison of the number of steps (level of intelligence) of the 3 algorithms in the first 10 stages

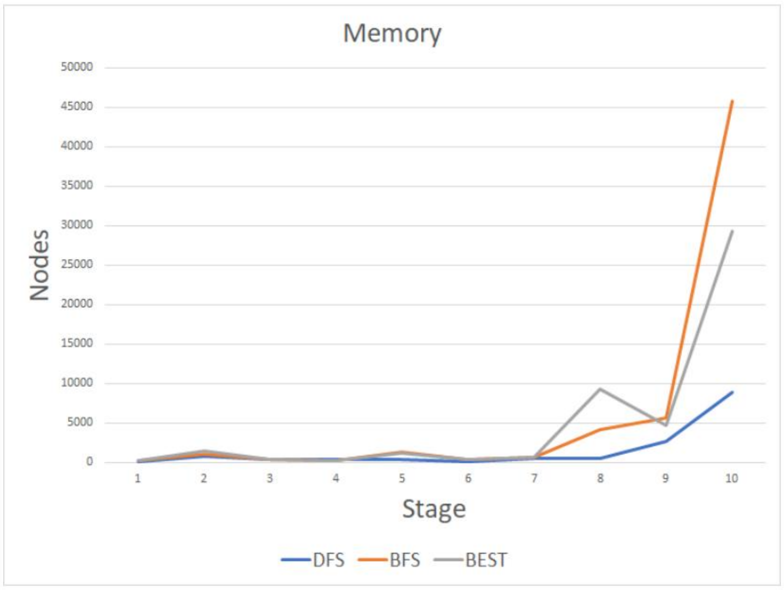

Figure 4: Comparison of memory usage of 3 algorithms in the first 10 stages

4.3 Evaluation

Because of the differences between stages, this assessment is not absolute compared to stages.

	Time spent Results		Number of states that must be approved
Breadth-First-Search	the most	the best	the most
Best-First-Search	average at least	average	average
Depth-First-Search		worst at least	

Figure 5: Comparison table for evaluating algorithms

In theory, we can see that BFS searches for results similar to humans most, because the results it finds by breadthwise traversal always have a distance to shortest initial state. In contrast, the DFS algorithm finds (almost) bad results best. The Heuristic algorithm is somewhere in between these two algorithms, but there are some stage where the Heuristic is not effective, for example stage 20, the Heuristic has to go through nearly 160 thousand new states found results, while BFS only needs to browse about 32 thousand states and DFS only needs to browse 5868 states, but the number of steps to its success is 111 steps, much better than DFS of 260 steps.

5. Conclusion

Through this assignment, we will better understand how to define state space For a problem, understand the implementation of each search algorithm such as BFS, DFS, Best First Search and implement them through the Python language. In the process of realization, they I can practice problem solving, improve algorithmic thinking, and use many structures Different algorithms make implementation easier.

Document

1. Solving Logic Puzzles using Model Checking in LTSmin Kamies, Bram University of Twente PO Box 217, 7500AE Enschede The Netherlands b.kamies@student.utwente.nl
2. SINGLE AGENT SHORTEST PATH DETERMINATION IN THE GAME OF BLOX-ORZ Nwulu E. A., Ndukwe I. G. Department of Computer Science, University of Jos Jos, Nigeria