Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 163 (2019) 391–399

Procedia Computer Science

www.elsevier.com/locate/procedia

16th International Learning & Technology Conference 2019

Game of Bloxorz Solving Agent Using Informed and Uninformed Search Strategies

Tahani Q. Alhassan[a], Shefaa S. Omar[a], Lamiaa A. Elrefaei[a,b,*]

[a]*Computer Science Department, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, Saudi Arabia*
[b]*Electrical Engineering Department, Faculty of Engineering at Shoubra, Benha University, Cairo, Egypt*

Abstract

Bloxorz is a block sliding puzzle game that can be categorized as a pathfinding problem. Pathfinding problems are well known problems in Artificial Intelligence field. In this paper, we proposed a single agent implementation to solve level-1 of Bloxorz game using Informed and Uninformed searching algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), and A-star (A*) searching algorithms. The agent solves the problem using the three algorithms to compare their performance and conduct a conclusion that may help in improving the use of searching algorithms in this area. In this paper, A* and breadth-first search algorithms are founded to be more convenient to solve this problem.

Keywords: A*; agent, Bloxorz; breadth-first search; depth-first search; heuristic function; path-finding problem; puzzle.

1. Introduction

Search problems are one of the important areas that have long and distinguished history in artificial intelligence (AI). Search problems still attracting researchers' attention concerning the provision of high-performance solutions. Problems that can be solved by searching algorithms can be classified into three types: path-finding problems, constraint satisfaction problems, and two-player games [1]. Path-finding is a fundamental component of different

* Corresponding author.
E-mail address: tahaniqadi16@gmail.com, shefaasomar@gmail.com, Laelrefaei@kau.edu.sa, lamia.alrefaai@feng.bu.edu.eg

10.1016/j.procs.2019.12.121

applications, such as robotics, video games, and GPS applications [2]. The best-known path-finding problems are puzzles such as 8-puzzle game.

Agent is an umbrella term that covers many types of specific agents [3]. Software agent can be defined as a component that can act exactingly in order to accomplish tasks on behalf of its user [3]. Path-finding enables an agent to find a path between two points, initial and target points. An agent has to traverse from the initial point toward the target point following the optimal path. The environment that path-finding agent works on usually have obstacles or restricted by game rules. Agents that work to solve path-finding problems can be single or multi-agent.

Bloxorz game is a path-finding problem or can be described as a block sliding puzzle game. This problem can be solved by a single agent based on searching algorithms. In this paper, an implementation of Bloxorz level-1 solver agent is proposed by three searching algorithms: Breadth-first search (BFS), Depth-first search (DFS), and A-star (A*). The paper aims to compare the results of the three algorithms using the proposed agent. The paper is organized after the introduction section as follows: section 2 is an overview of some related works, section 3 is a description of Bloxorz problem, section 4 is the formulation of Bloxorz as a search problem, section 5 is the description and analysis of the algorithms, section 6 is the proposed design and implementation of the agent, section 7 is the results discussion, and last section is the work conclusion.

2. Related Work

The authors in [4] proposed an implementation of eight-puzzle problem with a Visual C++ interface using three different searching algorithms, BFS, DFS and A*. They solve it as a path-finding problem where goal state should be reached by minimum state movements. All searching algorithms used graph search strategy where visited states will not be repeatedly visited. They got good results for A* algorithm over the others in terms of expanded nodes and elapsed time. A* got less than a half of the time needed for BFS, the next better one after A*.

The authors in [5] is also worked on C++/OpenGL-based application for sliding tile puzzle to find the goal path with three algorithms: A*, Dijkstra and D*. The used heuristic evaluation function that helped A* and D* was Manhattan distance. They evaluated algorithms results on different environments of the same game. They found that A* and D* were better in finding shortest path more efficiently.

The work in [6] proposed a design and implementation of Bloxorz sliding puzzle, the same problem of this paper. They modified the breadth-first search algorithm to be heuristic algorithm and distinguish between nodes that should be relegated or explored. Their modification helps in saving time and effort. They also used Manhattan distance to be the heuristic evaluation function for BFS. Their works were expanded to include all the game levels and implementation of other game advance tools such as switch, bridge and block splitting.

On the other hand, the authors in [7] proposed different methodology to solve path-finding puzzles. They suggested to use model checker, the LTSmin toolset to know the puzzle is solvable and get the necessary steps toward the solution. They classify Bloxorz problem as block sliding problem and solve it using breath-first and depth-first searching algorithms. Moreover, they test the running and transitions count for concluding results. They found a great difference between DFS and BFS time and depth explored where the second was better.

3. The Description of Bloxorz Problem

Bloxorz is a 3-D block sliding puzzle game consists of a terrain that is built by 1×1 tile with a special shape and size, and a 1×1×2 size block. This game is a single agent path-finding problem that involves moving the block from its initial position using four directions (right, left, up, and down) and ensuring that its ends are always within the terrain boundary, until it falls into a 1×1 square hole in the terrain that represents our goal state. The block can be in three states, standing, lying horizontally, and lying vertically. When the block reaches the hole, it must be in standing state to fall in it.

In this paper, we've implemented the level-1 of the game where the size of the terrain is 6×10 rows and columns, starting position is at row 2 and column 2 or as user determines it, and goal position is at row 4 and column 7. The shape of the terrain in the first level is shown in Fig.1.

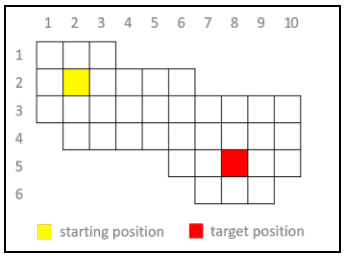


Fig. 1. Terrain shape, starting and target positions of Bloxorz first level.

4. Bloxorz Formulation as A Search Problem

Solving problems is an intelligent capability. To have an agent that solves Bloxorz automatically, the problem needs to be represented by a common representation [8]. In search problems, first, the problem needs to be formulated by defining a starting point for the agent (initial state), description of possible actions used to solve the problem (actions), description of what each action does (transition model), the goal state, and path cost function to evaluate the cost of each path. Then the agent searches to find the sequence of actions that solves the problem [8].

Here is the formulation of Bloxorz problem as a search problem:

- States Representation: position of the block in the terrain (x_1, y_1, x_2, y_2).
- Initial State: is (2,2,2,2) or as user determines.
- Actions: block can move right, left, up and down to go to a new state. Only legal moves, which are those moves that generate a new state within the dimensions of the terrain can be applied.
- Transition Model: Return the new legal position of the block after applying any legal move.

The initial state along with the actions and the transition model constitute the state space of the game problem. Example of the state space is shown in Fig.2.

- Goal State: Position (5,8,5,8) which means the block fall inside the hole as shown in Fig.1.
- Path Cost function *g(n)*: Step cost is 1 for every move. The path cost of any state/node *n* is the summation of all steps cost from the initial state to that state/node *n*.

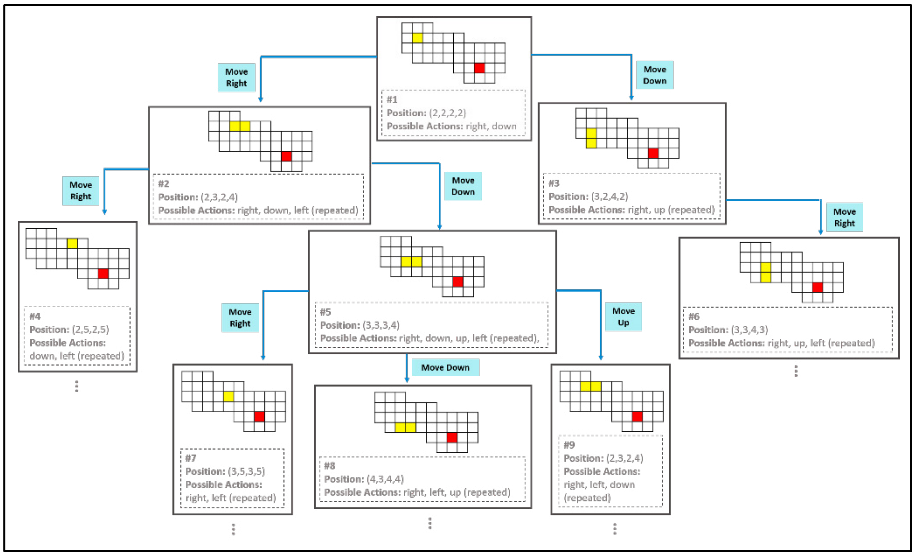

#1
Position: (2,2,2,2)
Possible Actions: right, down
Move Right
Move Down
#2
Position: (2,3,2,4)
Possible Actions: right, down, left (repeated)
#3
Position: (3,2,4,2)
Possible Actions: right, up (repeated)
Move Right
Move Down
Move Right
#4
Position: (2,5,2,5)
Possible Actions: down, left (repeated)
#5
Position: (3,3,3,4)
Possible Actions: right, down, up, left (repeated),
#6
Position: (3,3,4,3)
Possible Actions: right, up, left (repeated)
Move Right
Move Down
Move Up
#7
Position: (3,5,3,5)
Possible Actions: right, left (repeated)
#8
Position: (4,3,4,4)
Possible Actions: right, left, up (repeated)
#9
Position: (2,3,2,4)
Possible Actions: right, left, down (repeated)

Fig. 2. Part of the state space of Bloxorz first level.

5. Analysis and Design of The Algorithms

Bloxorz problem can be solved using both uninformed search and informed search strategies. Uninformed search which is Blind search includes BFS and DFS algorithms [8]. Informed search uses heuristic function to guide the search process, so it is a heuristic search algorithm [8] and we are using A* as an example.

For those algorithms, we need two data structures, one to store the nodes to be expanded called "Expand" and another called "Visited" to keep track of visited (already expanded) nodes. Nodes/States here represent the current position of the block in the terrain. The "Expand" data structure depends on the search algorithm itself but the "Visited" is a list contains nodes already explored which are positions the block already been in.

A. Breadth-First Search

In this search strategy, the "Expand" data structure is a simple FIFO (first in first out) queue. When a current node gets expanded, its children get inserted at the end of the queue and when polling out, always poll out the node at the first. This strategy searches in a wide development direction.

Pseudocode of the algorithm is as follows:

```
1) Predefine the initial node (initial position of the block) and the goal node (position of the hole).
2) Initialize the MoveSequence string of the initial node to empty.
3) Add the initial node into the Expand queue and the Visited list.
4) Check if the queue is empty → there is no solution and exit, otherwise continue.
5) Poll out the first node of the Expand queue (called parent node).
6) Check if it is equal to the goal node (block position defined in this node equals hole position defined in the
   goal node) → solution is found and return the MoveSequence string of that node, otherwise continue.
7) Expand the node and generate its child nodes which represent the new positions of the block after moving
   in all 4 directions from the current position defined in the parent node.
8) For each of the generated nodes, if it is valid (block position is valid in the terrain), its MoveSequence
   equals to its parent's plus the move it generated from, add the node to the Expand queue and the Visited list.
   Finally go back to step 4.
```

B. Depth-First Search

In this search strategy, the "Expand" is a LIFO (last in first out) data structure which is simply a stack. When a current nod gets expanded, its children get inserted at the top of the stack and when popping out, always pop out the node at the top. This strategy searches in a deep development direction.

Pseudocode of the algorithm is as follows:

```
1) Predefine the initial node (initial position of the block) and the goal node (position of the hole).
2) Initialize the MoveSequence string of the initial node to empty.
3) Add the initial node into the Expand stack and the Visited list.
4) Check if the stack is empty → there is no solution and exit, otherwise continue.
5) Pop out the top node of the Expand stack (called parent node).
6) Check if it is equal to the goal node (block position defined in this node equals hole position defined in the
   goal node) → solution is found and return the MoveSequence string of that node, otherwise continue.
7) Expand the node and generate its child nodes which represent the new positions of the block after moving
   in all 4 directions from the current position defined in the parent node.
8) For each of the generated nodes, if it is valid (block position is valid in the terrain), its MoveSequence
   equals to its parent's plus the move it generated from, add the node to the Expand stack and the Visited list.
   Finally go back to step 4.
```

C. A Search*

In this search strategy, the "Expand" data structure is a priority queue. Its priority is based on an Evaluation function *f(n)* and the highest priority is for node with the minimum *f(n)* value. When polling out a node, always poll out the node with the highest priority and the children of an expanded node can be inserted anywhere. This strategy expands only the nodes that's more likely will reach the goal. Finally, in this search algorithm we'll follow Tree Search strategy which doesn't exclude the visited nodes from expanding again since the heuristic evaluation function *h(n)* used is not consistent.

The main idea of any heuristic search algorithm is to use an estimated evaluation function to guide the search, assess each node and find out the node that more likely leads to the goal. The evaluation function is defined in (1):

$$f(n) = g(n) + h(n) \tag{1}$$

where *g(n)* is the actual path cost from the initial state *S* to the state defined in the current node. The path cost is represented by the number of moves required to reach that state. *h(n)* is used to estimate path cost from the state defined in the current node to reach the goal state *G*. Hence, *f(n)* represents the overall path cost from the initial state *S* passing by state in node *n* to the goal state *G*.

Since the heuristic value is an estimated cost calculated from specific node to the goal node, many calculations could be used as a heuristic function, choosing which one is appropriate depends on the problem itself. In the case of Bloxorz, Chebyshev distance gives good estimated distance between current and goal state, plus the cheaper calculation it has when comparing with Euclidean distance as an example. As presented in [9], Chebyshev distance is named after Panfnuty Chebyshev. It is also known by chessboard distance and maximum metric. The formula of the function is in (2):

$$h(n)=\max(|x-x_g|,|y - y_g|) \tag{2}$$

where x is the row number of current positions of block, x_g is the row number of goal position, y and y_g are the same but in terms of columns instead of rows. Since the block has 2 bricks and represented by two x,y coordinates, the heuristic value for the both bricks $h(n)_1$ and $h(n)_2$ are calculated, then the maximum one is considered as the overall heuristic. For example, the heuristic of the state represented in Fig.3 is $h(n) = 5$ which is the maximum of $h(n)_1 = max(|2-5|,|3-8|) = 5$ and $h(n)_2 = max(|2-5|,|4-8|) = 4$.

Pseudocode of the algorithm is as follows:

1) Predefine the initial node (initial position of the block) and the goal node (position of the hole).
2) Initialize the MoveSequence string of the initial node to empty.
3) Add the initial node into the Expand queue.
4) Check if the queue is empty → there is no solution and exit, otherwise continue.
5) Poll out the highest priority node of the Expand queue (called parent node).
6) Check if it is equal to the goal node (block position defined in this node equals hole position defined in the goal node) → solution is found and return the MoveSequence string of that node, otherwise continue.
7) Expand the node and generate its child nodes which represent the new positions of the block after moving in all 4 directions from the current position defined in the parent node.
8) For each of the generated nodes, if it is valid (block position is valid in the terrain), its MoveSequence equals to its parent's plus the move it generated from, compute its *h(n)* value and *g(n)* which is equal to its parent's plus 1, then set its *f(n) = h(n) + g(n)*, add the node to the Expand queue. Finally go back to step 4.

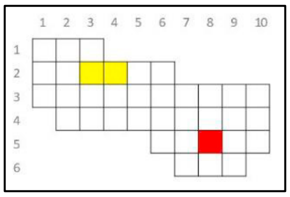


Fig. 3. A state where block position (x_1, y_1, x_2, y_2) is (2, 3, 2, 4) and the goal position (x_g, y_g) is (5, 8).

6. Implementation of The Algorithms

In this paper, we implement an agent to solve level-1 of Bloxorz game. Since the chosen algorithms to solve by are searching algorithms, the problem needs to be formulated and implemented as a search problem. Searching algorithms need work through a state space where initial state and target state are determined. Fig.2 shows part of the state space of Bloxorz level-1.

A. Agent Implementation

In the implementation, every position the block moves to is considered as a state/node. Every node holding state information like two *x* and *y* coordinates and path cost. The algorithm expands the node (if it can be expanded) and generate four children nodes where every node takes the block to one of the four directions, right, left, up, or down. Note that in this paper, RLUD is an abbreviation refer to the move directions order.

The terrain itself is considered as a grid implemented as 2-dimensional array. Every position in the grid has a value to indicates if it's a legal position which is within the terrain boundaries.

B. Moves Direction Order

Children generated nodes can be arranged according to some priority to help the algorithms finding the goal faster. There are four moves direction generate 24 different permutations. Moves direction order are arranged at the same order for all the three algorithms. Nodes arrangement is taking randomly to see if it affects algorithms performance.

C. Interface Design

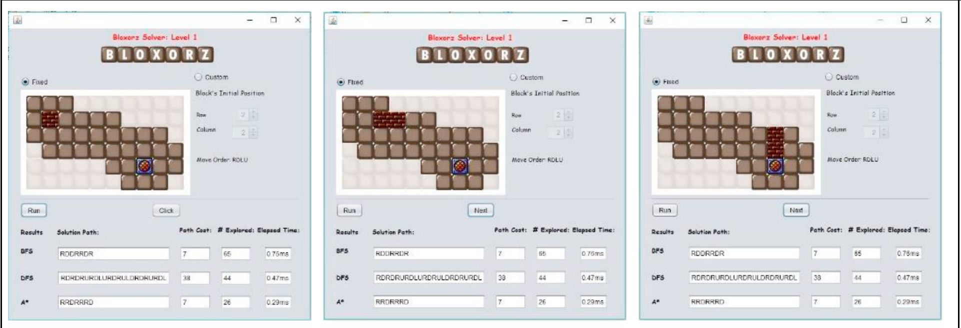
Graphical user interface is used to display the three algorithms results with a detailed example of one of the cases displaying the path steps. JFrame in Java platform is used to design the GUI. The interface displays two options to run the three algorithms, default and interactive. The results of all the three algorithms are displayed in the GUI, path solution, path cost, and number of expanded nodes by the algorithm, and the elapsed time in milliseconds.

1) Default option

In the default (fixed) option, user run the default first level of Bloxorz where the initial state fixed to row 2 and column 2. The moves directions arrangement is fixed to RDLU where this arrangement shows a typical case of the three algorithms. This allows the GUI to display all steps the algorithms take to solve the problem graphically Fig.4 (a).

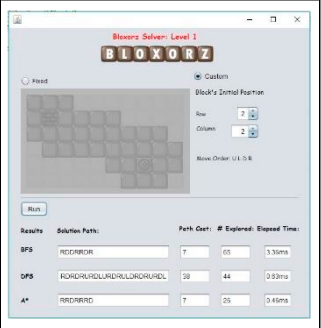
2) Interactive option

In the interactive (custom) option, user determine the position of the initial state in first level of Bloxorz by choosing a legal row and column. The moves directions arrangement is chosen randomly in each run case. This allow the GUI to display different cases of algorithms performance, as shown in Fig.4 (b).



Results	Solution Path:	Path Cost:	# Explored:	Elapsed Time:
BFS	RDDRRDR	7	65	0.75ms
DFS	RDRDRURDLURDRULDRDRURDL	38	44	0.47ms
A*	RRDRRRD	7	26	0.29ms

(a)



Results	Solution Path:	Path Cost:	# Explored:	Elapsed Time:
BFS	RDDRRDR	7	65	3.35ms
DFS	RDRDRURDLURDRULDRDRURDL	38	44	0.83ms
A*	RRDRRRD	7	26	0.45ms

(b)

Fig. 4. GUI options. a) the default option selected, and the solution graphically shown from left to right: initial, intermediate, and final state, b) the interactive option where the initial position is manually selected.

7. Results and Discussion

Searching algorithms founded to be a good choice for implementing a single agent path-finding solver. The differences between the algorithms strategies should affects the results. Bloxorz first stage is the implemented stage which is enough for testing and comparison purposes. Testing made on a machine with windows 10 operating system, 4G RAM, Intel core i5 processor and using the NetBeans environment.

Two test cases were selected for testing and comparing. Each of them has different initial state as shown in Fig.5. For each test case, the agent was run using the three algorithms and repeated 3 times with different moves orders, and the elapsed time, path cost, and number of expanded nodes were displayed each time.

A. Test Case-1

In the first test case (Fig.5 (a)), the initial state is as same as the level-1 of Bloxorz game where the initial position is at (2,2,2,2). This level is solvable by 7 moves as the minimum possible number of moves to solve it. The block in the initial state is placed at the left of goal position. Therefore, different moves orders chosen. One of the moves orders should takes the block toward the goal, firstly goes Right, then if it's not available goes Down, Left, and finally Up. Other move order should give higher priority to the directions that takes the block away from the goal position, i.e. Left, Up, Down, Left. Results of test case-1 are shown in Table 1. The optimal solution known for the game is in 7 moves which is represented by the path cost factor. Therefore, the results of the path costs in Table 1 shows that BFS and A* both always found the optimal solution (path cost = 7) unlike DFS (path cost = 7, 38, and 13). A*, however, outran the BFS with fewer number of explored nodes while searching, hence less elapsed time. We note that nevertheless one of the solutions found by the DFS got the fewest number of explored nodes (14 nodes) and the least elapsed time (0.06s). This is due to the searching strategy of the DFS algorithm that searches in a deep development direction. Therefore, it always finds the first correct solution it comes across, regardless of the optimality of the solution.

B. Test Case-2

As mentioned before, the level-1 of Bloxorz is the level implemented in this paper and to support the results, another initial state has been selected for as the second test case. This test case is a modified version of Bloxorz first level. The Block is placed at (6,8,6,8) under the goal position where 6 is the row number and 8 is the column number in the terrain. Test case-2 can be solve by 6 moves as the minimum possible moves. The same moves orders that used in test case-1 is also used here. The results are shown in Table 2, where the minimum path cost of a solution is 6 moves; hence, it is most likely to be the optimal solution for this custom case. Again, BFS and A* found the optimal solution every time and DFS found it once (path cost = 9, 39, and 6). However, in this test case, the BFS was better than A* in terms of the number of explored nodes (39, 47, and 33 vs. 165, 137, and 232 and elapsed time (0.08s, 0.09s, and 0.07s vs. 0.09s, 0.11s, and 0.19s). It means that the performance of the heuristic function used by the A* depends on the initial state which wasn't in its favor in this case. The DFS in this case got the fewest number of the explored nodes and elapsed time in every column, and whereas it found the optimal solution, it was the best result regarding all the factors. However, the fact that A* uses tree search instead of graph search should be considered. In graph search, the number of nodes to be explored is more than the tree search because the visited states don't get discarded. Hence, A* has higher number of nodes for searching to begin with comparing to the BFS and DFS.

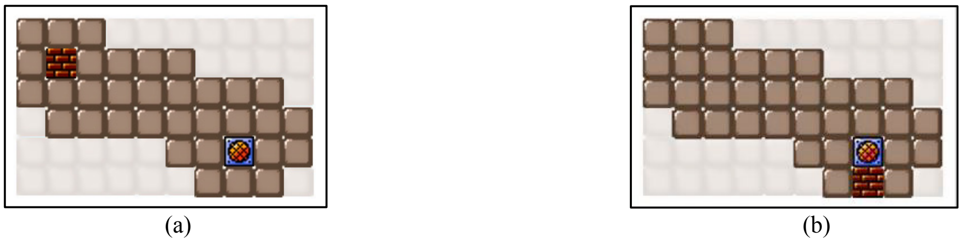

(a) (b)

Fig. 5. (a) Test case-1 where the brown block is the initial position and blue hole is the goal position. (b) Test case-2 where the brown block is the initial position and blue hole is the goal position.

Table 1. Test case-1 results for breadth-first search, depth-first search A* using the proposed agent.

Algorithm	Metrics	Move Order		
		RDUL	LUDR	DURL
BFS	Path cost	7	7	7
	# of explored nodes	56	65	65
	Elapsed time (ms)	0.43	0.28	0.22
DFS	Path cost	7	38	13
	# of explored nodes	83	43	14
	Elapsed time (ms)	1.86	0.25	0.06
A*	Path cost	7	7	7
	# of explored nodes	18	46	27
	Elapsed time (ms)	0.16	0.28	0.12

Table 2. Test case-2 results for breadth-first search, depth-first search A* using the proposed agent.

Algorithm	Metrics	Move Order		
		RDUL	LUDR	DURL
BFS	Path cost	6	6	6
	# of explored nodes	39	47	33
	Elapsed time (ms)	0.08	0.09	0.07
DFS	Path cost	9	39	6
	# of explored nodes	11	43	7
	Elapsed time (ms)	0.02	0.12	0.02
A*	Path cost	6	6	6
	# of explored nodes	165	137	232
	Elapsed time (ms)	0.09	0.11	0.19

8. Conclusion

Path-finding problems like Bloxorz founded to be easily solvable by searching algorithms and their performances differs depending on the search strategy being used (graph or tree) and the strategy of the algorithm itself. In this paper, a single agent was implemented to solve level-1 of Bloxorz game using Breadth-First Search (BFS), Depth-First Search (DFS), and A-star (A*) searching algorithms.

In pathfinding problems, the optimal path is mostly required like in Bloxorz problem. BFS and A* algorithms can be both good choices to it with considering the appropriate heuristic function for A*. Other important aspects in pathfinding problem is whether space and time matter. A* needs to work with a consistent heuristic evaluation function to save space and time by switching to graph search strategy. On the other hand, DFS is not guaranteed to find the optimal path but it sometimes saves time and give better performance than BFS and A*.

The results of both test cases show that BFS algorithm can always find the optimal path (if exists) in reasonable time and number of expanded nodes. Therefore, BFS is simple and straightforward algorithm for this sake. A* algorithm also reached the goal using the optimal path in both test cases. A* performance is strongly depending on the heuristic evaluation function that the algorithm uses. The heuristic function guides the search of the algorithm by giving good estimated values for each state. Moreover, since A* algorithm use tree search and the other two algorithms use graph search, the number of explored nodes of A* is higher than BFS and DFS in many cases but it didn't much affect the elapsed time. DFS algorithm cannot always find the optimal path. Its strategy depends on finding the first correct path it come across. Therefore, in many cases DFS solved the problem in less time than BFS and A*. DFS can reach the goal by the optimal path only if the optimal path was the first explored path the algorithm came across, and this case can be achieved if the moves order is taking the block toward the goal correctly. The case seen in test case-1 by moves order RDUL and in test case-2 by moves order DURL.

9. Acknowledgement

We would like to extend our sincere thanks and appreciation to our colleague Fatema Binafeef for her help.

References

[1] G. Weiss, *Multiagent systems: a modern approach to distributed artificial intelligence*, 3rd ed. USA: MIT Press, 2006.
[2] Z. Abd Algfoor, M. S. Sunar, and H. Kolivand, "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games," *International Journal of Computer Games Technology,* vol. 2015, p. 11, 2015, Art. no. 736138.

[3] H. S. Nwana and D. T. Ndumu, "A Brief Introduction to Software Agent Technology," in *Agent Technology: Foundations, Applications, and Markets*, N. R. Jennings and M. J. Wooldridge, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 29-47.

[4] H. Shi, "Searching Algorithms Implementation and Comparison of Eight-puzzle Problem," presented at the International Conference on Computer Science and Network Technology (ICCSNT), 2011.

[5] N. Krishnaswamy, "Comparison of Efficiency in Pathfinding Algorithms in Game Development," The College of Computing and Digital Media, DePaul University, 2009.

[6] N. I. G. Nwulu E. A., "Single Agent Shortest Path Determination in The Game of Bloxorz," *African Journal of Natural Sciences,* pp. 9-14, 2014.

[7] B. Kamies, "Solving Logic Puzzles using Model Checking in LTSmin," presented at the 24th Twente Student Conference on IT, Enschede, The Netherlands, 2016.

[8] S. J. R. Peter Norvig, *ARTIFICIAL INTELLIGENCE: a modern approach*. PEARSON, 2018.

[9] V. Kumar, J. K. Chhabra, and D. Kumar, "Performance Evaluation of Distance Metrics in the Clustering Algorithms," *INFOCOMP,* no. 1, pp. 38-52%V 13, 2014-09-01 2014.