Evaluation of Behavior Tree and Finite State Machine based Artificial Intelligence Algorithms in Shooter Games

1st Özgür Çebi
Department of Computer Engineering
Yeditepe University
Istanbul, Turkey
ozgur.cebi@std.yeditepe.edu.tr

***Abstract*—Nowadays, artificial intelligence algorithms have a major role in our lives. One of these areas are computer games. Most of the games has AI-controlled components in order to control the game behaviour. It is a must to keep these algorithms up to date in order to provide the best experience for the users.**

This paper will focus on the importance of artificial intelligence algorithms in shooter games. The proposed system will consist of two teams with the same number of agents. Each team will use a different algorithm to control the agent behaviour which are Finite State Machine (FSM) and Behaviour Tree (BT) algorithms. FSM algorithm will contain the core mechanics of the game which will be simplified to eliminating the enemy team. BT algorithm on the other hand will evaluate the environmental factors and make tactical decisions which is expected to increase the win rate. Results will be used to determine the best fitting algorithm for shooter games. Two main mechanisms will be included in both algorithms which are movement decision system and enemy targeting system.

To test the algorithms, a shooter game will be created with Unity to gather the necessary data. The agents will be bound with the same rules such as morale system and weapon accuracy system. There will also be environmental factors such as covers which reduce the probability of being hit. The game will be over when a team has lost all of its agents. Algorithms will be tested on different maps. Win rate, win time, survived agent count and map design will be used to evaluate the results.

***Index Terms*—artificial intelligence, artificial intelligence in games, behavior trees**

I. Introduction

Artificial Intelligence (AI) has an important role in video games. A solid AI algorithm is a must for video games to offer a smooth user experience. Whether the game is a Massively Multiplayer Online (MMO) game for PC or a simple hyper-casual game for mobile devices, at some point an AI system might be required. On the other hand, the complexity level of the AI algorithm used may differ depending on the game type. While simple games can be implemented with simple AI algorithms, more advanced AI algorithms are required for complex games. It is a must to choose the right AI algorithm before implementing a game. Two of the most popular AI algorithms for games are Finite State Machine (FSM) and Behavior Tree (BT).

When programming the game AI, the right decision should be made about the algorithm. Game platform, game type and user profile should be considered. Without knowing the differences between algorithms, it is usually hard for a developer to choose the right one. FSM algorithms are known for their simplicity and ease of use whereas BT algorithms are known for scalability and capability of more complex decisions. If BT algorithm is chosen for a simple game, it may increase the development time and complexity level unnecessarily. Likewise, if FSM algorithm is chosen for a complex game, it may result in an unrealistic behavior among agents and it may reduce the scalability of the project. Purpose of this paper is to mark the strong and weak sides of these algorithms and to decide which fits best for shooter games.

Being two of the most popular algorithms in terms of game AI, FSM and BT will be compared in a shooter game environment. AI algorithms with FSM and BT will be used to simulate agent behavior on a game world designed for shooter games. Two teams will compete and results will be analyzed to compare the effect of algorithms in terms of win rate, win time and survived agent count. Each test will have different factors that affect the gameplay such as map type, agent number, and obstacles. The difference between algorithms may not be well understood in a single map. So the tests will be executed on different map types. Likewise, since the number of agents in the teams might affect the performance of the algorithm, tests will be executed with different agent numbers. Each map has its own unique obstacles. Tests will be conducted on maps with enabled obstacles and disabled obstacles in order to measure the effects of obstacles on agent performance.

II. Background

Digital games have become an important subject in daily life. COVID19 pandemic has forced people to find new ways to socialize and gaming is one of them. Both multiplayer and single player games are a great way to entertain people. Almost all types of games have a common subject when it comes to entertainment which is AI. In video games, there is usually a need for complex agent behaviors.

AI solutions used in video game industry differ from the ones used in other industries. In other industries, traditional techniques are usually implemented to solve problems by optimizing the solutions without time and resource constraints. In modern fast-paced computer games on the other hand, Martin Estgren and Erik S. V. Jansson [1] states that rendering of the graphics, physics calculations and AI updates are needed to complete in less than 16ms to give a decent experience to the users. It is also stated that clever techniques have been implemented to meet these constraints and reduce computational time.

This section will be focused on the use of existing AI algorithms in all types of games. The main focus will be FSM and BT algorithms since they are evaluated in this paper but other algorithms will be briefly mentioned as well. Following subsection will contain information about characters that are controlled by AI. Rest of the subsections will discuss different AI algorithms used in games.

A. Rule-Based Systems

Rule-based systems are considered to be the simplest version of AI. Yoones A. Sekhavat [2] mentions that control over the behavior is limited in these systems. A rule-based system usually has a knowledge-base and a predetermined if-then rules which are programmed to control an agent. Rules are evaluated to check which conditions are met. A corresponding action will be determined based on the conditions.

B. Finite State Machines

One of the most popular way of creating AI for an NPC is Finite State Machines. Programming different states for different behaviors simplifies the design process of behaviors. Gonzalo Florez-Puga et al. [3] states in their paper that the reason behind FSMs are still a common way of creating simple intelligent agents is their simplicity, efficiency and expressivity. A set of states exists in FSMs which are connected by transitions. FSM can be represented as a directed graph where each node represents a state. Only one state can be active at a time. Main problem of the FSM is considered as scalability in Y. A. Sekhavat's work [1]. Modern games require complex behaviors where a lot of conditions are checked at the same time. Increasing number of states makes it almost impossible to maintain a solid AI algorithm using FSMs. Another downside of FSMs is the lack of reusability. Same logic might be needed in a different context which requires the logic to be reimplemented.

C. Hierarchical Finite State Machines

Hierarchical Finite State Machines (HFSM) can be considered as an evolution of FSMs. Gonzalo Florez-Puga et al. [3] defines the purpose of HFSM as to group the states which have common transitions. These states are called super states. One of the most important features of Hierarchical State Machines is hierarchical state nesting which is also known as inheritance. Using inheritance while designing a system

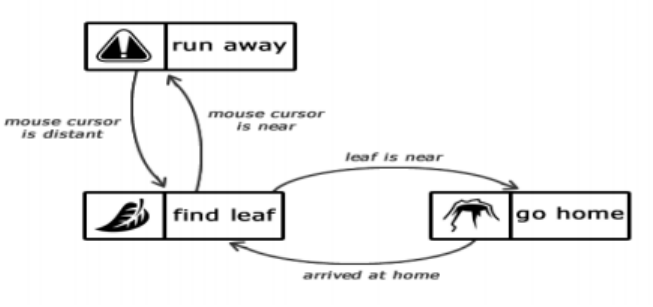

Fig. 1. Example of FSM that controls an ant [4]

considerably decreases the development time since it allows reusability of the classes and reduces redundancy. It also increases maintainability since the implementation of parent class may change which automatically affects the child classes. However, the work of Rahul Dey and Dr. Chris Child [5] reveals that high state count of HFSM requires a large number of transitions which may lead to maintainability problems.

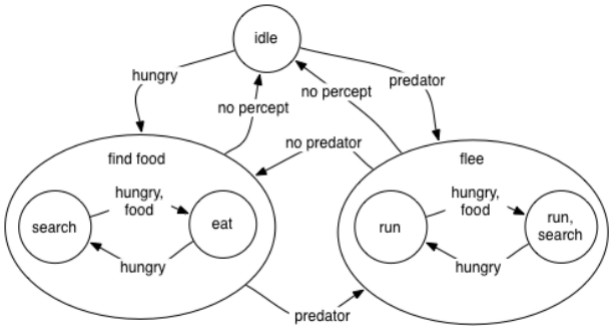

Fig. 2. Example of Hierarchical Finite State Machine [6]

D. Behavior Trees

Behavior Trees have become popular in the last decade in terms of AI algorithms in games. Ryan Marcotte and Howard J. Hamilton [7] states that Halo 2 (2004) was one of the first games that used BT to control intelligent agents. BTs are still widely used to implement AI algorithms in commercial games such as Starcraft. Shakir Belle et al. [8] mentions in their work that Unreal Engine which is a popular game engine comes with built in BT algorithms ready to use for digital games. BTs are usually used to control the decision making of non-player characters (NPC) used in the games. They are also becoming more popular in terms of an alternative for controlling robots as it is stated in the work of Ryan Marcotte and Howard J. Hamilton [7].

Y. A. Sekhavat [1] defines the BT as a directed tree which includes a set of nodes and edges. The first node of a BT is considered as the root node and does not have any parents. Nodes that do not have children are the leaves of the tree. BTs are associated with an AI agent in order to control each agents' AI logic. While executing, each component defined in a BT of the agent is traversed and handled based on the conditions. Rahul Dey and Dr. Chris Child [5] explains in their work that a depth-first traversal is performed on each update until a low level behavior is completed. Hence, behaviors are placed from left to right with respect to their significance. An important task should be placed at the most left part whereas a less significant task can placed at the right. Ryan Marcotte and Howard J. Hamilton [7] explains in their work that each component is executed with a maximum amount of processor time defined. Then, components return a status code to their parents. Those status codes are *SUCCESS*, *FAILURE* and *RUNNING*. There is also an *ERROR* status code but it will be ignored since it is only used while debugging the process. *SUCCESS* code is returned if the task associated with the component is succesfully completed. *FAILURE* code is returned if the task is completed without success. *RUNNING* code is returned if the task is not completed and still requires more steps to run. BTs consist of different nodes such as Selector, Sequence, Inverter, Succeeder, Condition and Action nodes.

- **Selector:** Selector node can be considered as if-else or switch-case structures in programming. Child nodes of the selector node are executed from left to right until one of the conditions return *RUNNING* or *SUCCESS*. Corresponding result is returned to the parent of the selector node. If all child nodes return *FAILURE*, selector also returns *FAILURE*.
- **Sequence:** Sequence node executes all of its children with respect to their placement. If a node returns *SUCCESS*, the node to its right begins executing. If any node returns *FAILURE* while executing, sequence stops the execution and returns *FAILURE* to its parent. If all children returns *SUCCESS*, sequence node also returns *SUCCESS*.
- **Inverter:** Inverter executes the child node and returns the inverted status to its parent. If child returns *FAILURE*, inverter returns *SUCCESS* and vice versa.
- **Succeeder:** Succeeder waits until the child node has finished running and returns *SUCCESS* regardless of the result of the child.
- **Failer:** Failer starts the execution of its child node and waits for its execution. When child node returns its value, failer always returns *FAILURE* to its parent.
- **Repeater:** Repeater starts the execution of its child node. It has an iteration variable. When child node finishes its execution, repeater moves to the next iteration and executes the child node again. When the iterations have finished, repeater returns *SUCCESS* to its parent.
- **Condition:** Ryan Marcotte and Howard J. Hamilton [7] states in their work that condition node evaluates a Boolean question which means the answer can either be true or false. Condition could be checking the distance between a position, checking player health is above a level or anything that could return a Boolean value. If the Boolean value is true, *SUCCESS* is returned to the parent node. *FAILURE* is returned otherwise. Condition node cannot return *RUNNING* to the parent since it is instant.
- **Action:** Action node can execute game code by calling a method in an entity which is considered as an action. This action could be anything such as moving the player to a position, setting a variable, playing an animation or shooting with a weapon. *SUCCESS* is returned to the parent if the action is completed. *FAILURE* is returned if the action is completed without success. If more time is required for the action to be completed, *RUNNING* is returned to the parent.

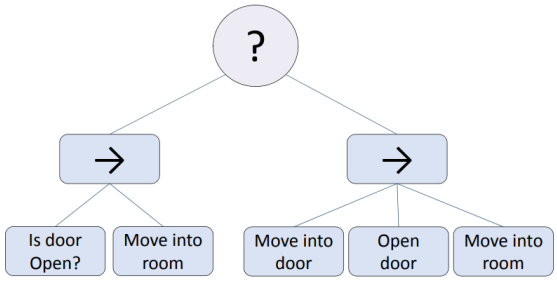

Fig. 3. Example of a behavior tree that includes selector, sequence, condition, and action nodes [2]

The work of Martin Estgren and Erik S. V. Jansson [1] states that action and condition nodes are considered as the leaf nodes and are concrete. All other nodes are either composite or decorator nodes which are abstract nodes. Hence, they need to be a parent of a node in order to become concrete, such as selector or inverter nodes.

E. Emotional Behavior Trees

Some BT algorithms may become easily predictable and unadaptive. To solve this problem, Anja Johansson and Pierangelo Dell'Acqua [9] proposes a new extension to the BT which is called Emotional BT. This method allows the algorithm to evaluate the emotions of the agent and take actions accordingly. A new priority selector is added in the work that is capable of dynamically evaluating priorities which allows the evaluation of emotions. AI agents that are capable of making decisions based on their emotions are a great way to increase the realism of games.

F. Q-Learning Integrated Behavior Trees

Rahul Dey and Dr. Chris Child [5] proposes a new system that integrates Q-Learning into Behavior Trees (QL-BT). This

way, Reinforcement Learning (RL) could be applied to BT. The system takes a BT as input. By applying RL, a new BT is generated that works more efficient. Results of the work shows that the output tree of QL-BT performs at least on a par with the original BT or outperforms it in all areas.

G. Supervised and Unsupervised Learning

The purpose of Supervised Learning (SL) is defined in the work of Fernando Fradique Duarte et al. [10] as to learn a predictive model and based on this model, make predictions about the unseen data in the future. On the contrary, Unsupervised Learning (UL) uses unlabeled data to extract relevant information and make predictions about the future.

Fernando Fradique Duarte et al. [10] also states that game AI domain has many applications related to SL and UL. One of the most important example was the board game Othello where the computer program defeated the world champion of Othello.

H. Neural Networks and Deep Learning

Kun Shao et al. [11] states that the origin of deep learning comes from the artificial neural networks. Purpose of deep learning is to learn data representation. The inspiration for deep learning comes from the theory of brain development.

Neural Networks (NN) and Deep Neural Networks (DNN) have been used in the game AI domain in many applications according to the work of Fernando Fradique Duarte et al. [10] They state that in the domain of classical board games, purpose of NNs were to lower the need of expert domain knowledge. NNs were also used to automate the discovering of good features and to improve the evaluation function used.

I. Reinforcement Learning

RL is considered as a subfield of Machine Learning (ML). Fernando Fradique Duarte et al. [10] describes the concern of RL with the design of agents which are capable of learning with trial and error. Agents interact with their environment in order to learn the desired actions. Unity recently provided ML-Agents Toolkit which allows developers to use RL in their games.

J. Evolutionary Algorithms

The work of Fernando Fradique Duarte et al. [10] describes the Evolutionary Algorithms (EA) as a family of metaheuristic optimization algorithms. It is stated that EAs are usually used to solve complex problems.

In terms of video games, EAs are used in multiple applications. Fernando Fradique Duarte et al. [10] describes the use cases as control, player characterization, procedural content generation, customization of games and support for game design.

III. Methodology

In video game industry, it is quite unclear for inexperienced developers to choose the right AI algorithm for a game. There are lots of AI algorithms on the table and each have unique upsides and downsides. Since implementing and comparing all of them is hardly possible, this research includes two of the most used algorithms which are FSM and BT.

For comparing the algorithms, a real-time shooter game is created with a game engine. The main mechanics of the game are implemented so that each agent uses the same rules. This allows agents to compete in a fair environment. FSM and BT algorithms are implemented to the agents on different teams. This way, quantitative data needed to compare the algorithms could be collected. The main factor in this data is the winner team. There are also factors such as map type, game time and number of agents left alive in the winner team. By analyzing the data, it should give an idea to determine which algorithm is the best fit for a shooter game.

IV. Evaluation

In this section, map types, test methods and test parameters will be explained and test results will be analyzed.

A. Map Types

3 different map types are chosen for testing the AI algorithms. These are small, medium and large maps.

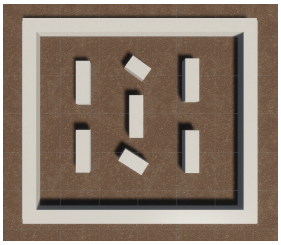

Fig. 4. Small Map

Small map (Figure 4) consists of narrow areas and measures the behavior of agents in close combat. Obstacles are an important factor in terms of combat. Utilizing the environment in combat is considered as an advantage to the AI algorithm.

Medium map (Figure 5) has more gaps between obstacles compared to small map. Medium map allows agents to execute flank attacks from sides or retreat to distant positions.

Large map (Figure 6) is the largest map in the game. It takes some time for agents to find each other and engage in

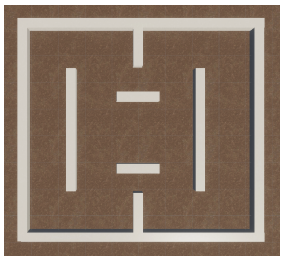

Fig. 5. Medium Map

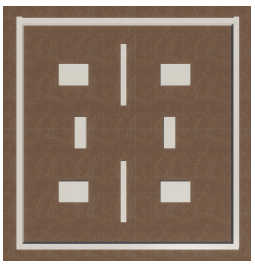

Fig. 6. Large Map

this map. When combat begins at a position, all agents hear the gun shots and move towards the conflict area. This map is useful to test the reaction times of agents.

B. Test Parameters

Purpose of the test is to compare the performance of AI algorithms in a shooter game. In order to maximize the difference of algorithms, 3 factors are chosen. These are map type, number of agents and include obstacles. Map type is related to the design of the environment that agents are competing. Number of agents is the agent number in each team. Varying number of agents in the scene has an effect on competition. It is possible for some algorithms to give a better performance when agent number is high whereas some algorithms might perform worse if the environment is too crowded. Each map has its own obstacles placed randomly around the map. Include obstacles means whether these obstacles are enabled and might block the vision of agents or disabled and agents have the maximum line of sight if they are facing the right direction. A unique combination of these 3 factors represents a test case. Test parameters are the parameters that are watched during a match. They are used to measure the difference between algorithms. There are 3 parameters in the test which are win rate, win time and agents survived. Win rate represents the win rate of each algorithm. This is the most important parameter that gives an idea about the algorithm performance. Win time is the time passed until a team eliminates all agents of another team. Besides winning the game, win time is also an important factor in terms of understanding the algorithm behavior. Lastly, agents survived means the number of agents that are alive in the winning team. This parameter gives an idea about the competition of teams. The more agents survive in the winner team, the more successful that algorithm is.

C. Test Methods

To gather the test data, each match result is saved to a file. Test cases are executed on all 3 maps. Iteration number is chosen as 50 so that each test on a map is executed 50 times. Agent number is chosen as 5 to 20 with an increment of 5. With 3 test parameters, 4 different agent numbers and 3 maps, a total of 24 tests are played. Since each test is executed 50 times, a total of 1200 matches are played which approximately took 17 hours. Either BT or FSM algorithm might win the game. There is also a third option which is tie. If teams cannot eliminate each other in 5 minutes, the match results in a tie. No tie is recorded in the tests.

D. Results

In this subsection, test results will be visualized in bar charts and analyzed. Since test results occupy too much space, tests are reduced from 24 to 18. Tests with 5,10 and 20 players will be analyzed. Each chart represents a test parameter that is watched. There are 3 charts that show Win Rate, Win Time and Survived Agent Count. Map type, number of agents and include obstacles parameters can be seen in Table I with respect to the test number shown in the chart. In the test results, BT algorithm dominates the FSM algorithm almost in all cases. Only exception is Test 1 where FSM win rate is greater by 2 points than BT win rate. Likewise, win time of BT algorithm is lower than FSM algorithm in all cases except Test 1.

Win times are usually close except the large map, where the difference goes up to 20 seconds. This is expected since travelling in the large map takes time. However, large win time difference also implies that BT algorithm is more efficient in terms of eliminating the enemy team whereas FSM algorithm wastes more time in the process.

Survived agent count in the winner teams is usually close with less than 1 point difference. Only exceptions are Test 12 and Test 18 where FSM agents survived considerably more than BT agents. Common parameters of Test 12 and 18 are agent number which is 20 and include obstacles which is false. Win rate of FSM is lower than BT in these tests. We can conclude from this result that despite the high win rate of BT algorithm, FSM algorithm can still win games with dominance.

When agent number is 20, it is seen from the tests that win rate of BT is decreasing as the map gets bigger. When obstacles are not included, this decrease is considerably large. There is still a decrease when obstacles are included. This result shows that BT algorithm performs better in small maps compared to FSM algorithm. It is also seen that obstacles have an effect on algorithm performance.

V. Conclusion

It is seen from the test results that BT algorithm dominates the FSM algorithm in terms of win rate in almost all of the cases. Only exception was Test 1 which was executed with only 5 agents in both teams. However, the randomness factor in the AI algorithms should be kept in mind since most of the tests indicate that win rate of BT algorithm is much more higher than FSM. By adding two simple tasks such as a more effective enemy targeting and a flee action, BT algorithm wins most of the games. Despite the high win rate of BT, agents controlled by FSM algorithm could still beat BT agents with a dominance. However, these cases were very rare. In terms of realistic shooter games, BT algorithm seems a better fit for AI agents rather than FSM.

Acknowledgment

I would like to thank my advisor Prof. Dr. Sezer Gören Uğurdağ for her guidance and contribution in this work.

References

[1] M. Estgren and E. S. V. Jansson, "Behaviour tree evolution by genetic programming," 2017.

[2] Y. A. Sekhavat, "Behavior trees for computer games," 2017.

[3] G. Florez-Puga, M. Gomez-Martin, B. Diaz-Agudo, and P. A. Gonzalez-Calero, "Dynamic expansion of behaviour trees," 2008.

[4] M. F. Syahputra, A. Arippa, R. F. Rahmat, and U. Andayani, "Historical theme game using finite state machine for actor behaviour," 2019.

[5] R. Dey and C. Child, "Ql-bt: Enhancing behaviour tree design and implementation with q-learning," 2013.

[6] F. W. P. Heckel, G. M. Youngblood, and N. S. Ketkar, “Representational complexity of reactive agents,” 2019.
[7] R. Marcotte and H. J. Hamilton, “Behavior trees for modelling artificial intelligence in games: A tutorial,” 2017.
[8] S. Belle, C. Gittens, and T. Graham, “Programming with affect: How behaviour trees and a lightweight cognitive architecture enable the development of non-player characters with emotions,” 2019
[9] A. Johansson and P. Dell’Acqua, “Emotional behavior trees,” 2012.
[10] F. F. Duarte, N. Lau, A. Pereira, and L. P. Reis, “A survey of planning and learning in games,” pp. 5–10, 2020.
[11] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, “A survey of deep reinforcement learning in video games,” 2019.

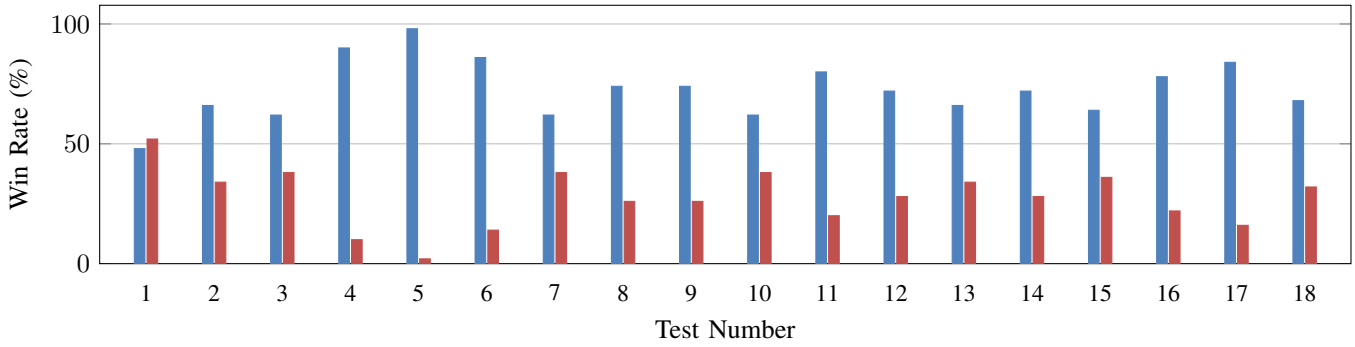

Fig. 7. Average win rate of BT (in blue) and FSM(in red) algorithms

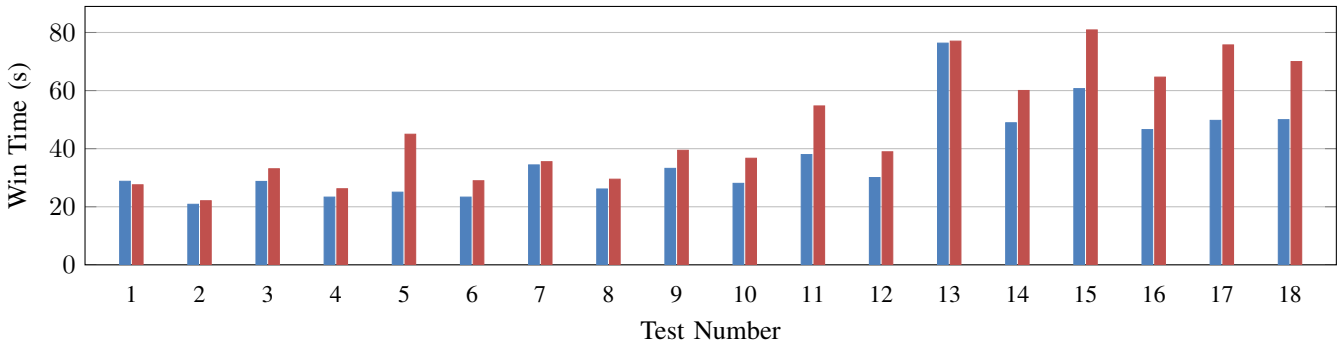

Fig. 8. Average win time of BT (in blue) and FSM (in red) algorithms

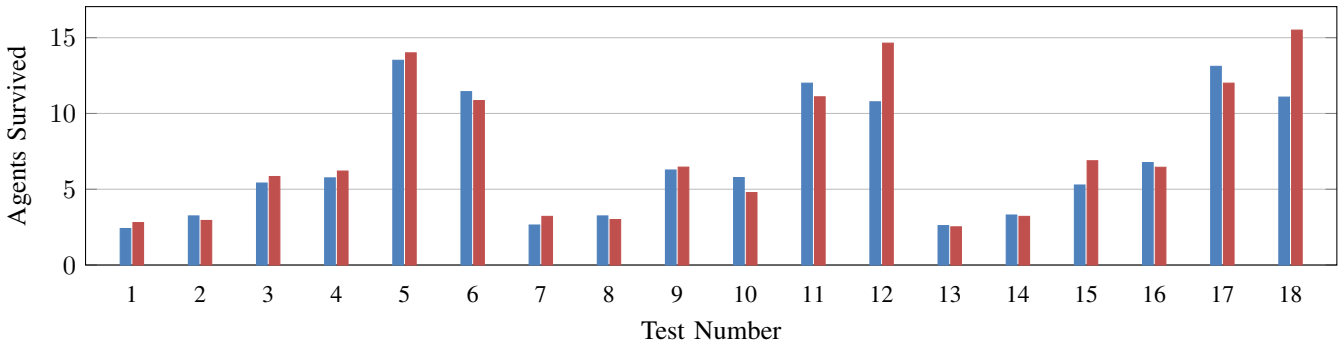

Fig. 9. Average survived agent count of BT (in blue) and FSM (in red) algorithms

TABLE I
TEST CASES

Test	Map	# of Agents	Include Obstacles
1	Small	5	TRUE
2	Small	5	FALSE
3	Small	10	TRUE
4	Small	10	FALSE
5	Small	20	TRUE
6	Small	20	FALSE
7	Medium	5	TRUE
8	Medium	5	FALSE
9	Medium	10	TRUE
10	Medium	10	FALSE
11	Medium	20	TRUE
12	Medium	20	FALSE
13	Large	5	TRUE
14	Large	5	FALSE
15	Large	10	TRUE
16	Large	10	FALSE
17	Large	20	TRUE
18	Large	20	FALSE