# Beating Bomberman with Artificial Intelligence

**Juarez Monteiro [1], Roger Granada [1], Rafael C. Pinto [2], Rodrigo C. Barros [1]**

[1]Escola Politécnica - Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre – RS – Brazil

{juarez.santos, roger.granada}@acad.pucrs.br, rodrigo.barros@pucrs.br

[2]Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul
Campus Canoas, Canoas – Brazil

rafael.pinto@canoas.ifrs.edu.br

***Abstract.*** *Artificial Intelligence (AI) seeks to bring intelligent behavior for machines by using specific techniques. These techniques can be employed in order to solve tasks, such as planning paths or controlling intelligent agents. Some tasks that use AI techniques are not trivially testable, since it can handle a high number of variables depending on their complexity. As digital games can provide a wide range of variables, they become an efficient and economical means for testing artificial intelligence techniques. In this paper, we propose a combination of a behavior tree and a Pathfinding algorithm to solve a maze-based problem using the digital game Bomberman of the Nintendo Entertainment System (NES) platform. We perform an analysis of the AI techniques in order to verify the feasibility of future experiments in similar complex environments. Our experiments show that our intelligent agent can be successfully implemented using the proposed approach.*

## 1. Introduction

Advances in Artificial Intelligence (AI) have been promoting a lot of changes in our society while trying to solve many types of tasks. We now can apply AI in business systems [Wang et al. 2015], movies [Idrees et al. 2017], and games [Miranda et al. 2016, Mnih et al. 2015, Silver et al. 2016]. For instance, games can be improved using different kinds of AI algorithms, such as problem solving techniques, planning, autonomous agent control, machine learning *etc.*. AI is also applied in business when a human being is unable to process the amount of data. For example, when making an investment decision the system has to perform the analysis of several variables using complex data.

A common use of AI relies on the implementation of an intelligent agent in a digital environment to simulate techniques that may be applied to real world problems. An intelligent agent can be defined as an entity that observes the environment through sensors, processes acquired knowledge and acts upon an environment using actuators [Russell and Norvig 1995]. Although the development of an intelligent agent is important, sometimes, building the environment in order to test the agent is a difficult task. Hence, testing AI techniques using digital game scenarios is the first step in order to validate the approach.

In this paper we propose the combination of a behavior tree and a *Pathfinding* algorithm in a decision-making process to solve a maze-based video game problem. In

order to test the AI techniques, we use the digital game *Bomberman* of the Nintendo Entertainment System (NES) platform as the environment, since the number of variables and states is large enough to provide interesting challenges, similar to real world environments. In our experiments, we vary the *Pathfinding* algorithm using *Breadth-First Search* (BFS) and *A-Star* (A*) to verify the feasibility of these AI techniques in digital games.

This paper is organized as follows: In Section 2, we provide an introduction to AI in games and the complexity of its application. Section 3 presents an overview of the digital game *Bomberman* and the emulator used in this work. Section 4 details our approach, with the techniques we apply in the game, whereas Section 5 presents the application of the approaches and its results, as well as the discussion about the results achieved in the experiments. Section 6 describe the work related to the application of AI in games, and we finish this work with our conclusions and future work directions in Section 7.

## 2. Artificial Intelligence in Games

Digital games are software for recreational, educational and other purposes, covering various kinds of public, from children to elderly and even people with disabilities. [Rogers 2014] describes 11 genres of digital games, including *action*, *shooter*, *adventure* and *strategy*. An example of strategy game is *Bomberman*, where the player is faced with various situations and often have to make fast decisions to achieve his goal. This game's environment occurs in real time and is partially observable, dynamic, and uncertain (stochastic). Taking these characteristics into account, *Bomberman* can be considered as a game having a complex environment.

Games have been seen as ideal environments for AI testing because they have a lot of characteristics that can define a complex environment, such as a large space of states. Examples of games with large space of states include Chess, Go game and the NES platform, containing respectively $10^{41}$, $10^{170}$, $10^{602}$ number of maximum states. We find this maximum number of states for the NES since the console has 2kB of RAM, which is equivalent to $2^{2000}$ or approximately $10^{602}$ states. Thus, areas such as artificial intelligence have made use of these complex environments in order to apply several techniques, with the goal of testing and substantiating their applications.

Many works already use digital games as platforms in order to apply AI techniques, such as artificial neural networks and genetic algorithms [Holmgard et al. 2014, Liapis et al. 2013]. Recent researches have presented the use of deep learning in games, combining different types of AI algorithms [Mnih et al. 2015, Schuurmans and Zinkevich 2016]. One important contribution to science using games is the recently presented *AlphaGo* algorithm [Silver et al. 2016], which has defeated a professional Go player in a competition organized by Google. This shows us how much advance AI is doing in the area.

## 3. The Game

Published in 1985, *Bomberman* is a strategic, maze-based video game developed by Hudson Soft for Nintendo Entertainment System (NES), an 8-bit home video game console that was developed and manufactured by Nintendo. The single-player game contains a bird's eye (top down) view and takes you on a series of stages where the goal is killing every enemy and escaping through the exit door before the time limit is up. Controls are

limited to four movements (up, down, left and right) and the ability to drop bombs, which then explode in a cross-shaped blast (creating a vertical and horizontal stream of fire) after a set time. Bombs can destroy enemies as well as blow up parts of the scenery to open up new paths and uncover power-ups. Power-ups add certain abilities to *Bomberman*, such as increasing the size of the blast, increasing the number of bombs *Bomberman* can drop at once, increasing the speed of the *Bomberman* or walking through walls. Each stage has an exit door, which is always hidden underneath some bricks of the stage. The exit door is exposed when blowing the brick up. However, hitting the exit with a bomb releases more enemies which the player must also kill. After killing all the enemies, the exit is activated to go to the next stage.

In this work, we use the FCEUX[1] emulator in order to emulate the NES console as well as the *Bomberman*[2] game. The emulator allows us to code AI algorithms inside the game allowing the computer to play the game autonomously. Figure 1 illustrates the "Start" screen and the first stage of the game using the emulator, where the *Bomberman* character is on the top left corner, enemies in orange are spread over the map, solid green blocks are indestructible blocks and bricks are destructive blocks.

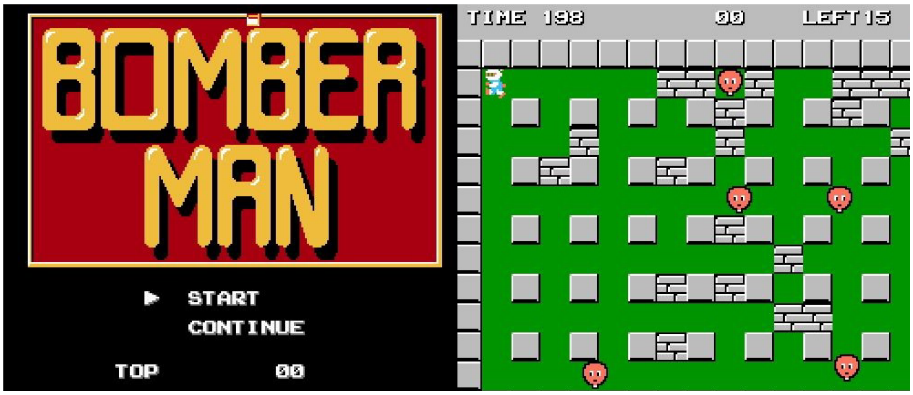



**Figure 1. Start screen and first stage of *Bomberman* to the *NES* platform.**

## 4. Inserting Agents in the Game

In this section, we describe the implementation of the agents we use to test our algorithms. The developed agents play the role of *Bomberman* (the main character of the game) and are divided into a random agent and an intelligent agent. The random agent is used to validate our implementation of the intelligent agent. The intelligent agent implements AI algorithms to achieve the goal of each stage.

### 4.1. Random Agent

A random agent is developed in order to validate our approach, *i.e.*, if a random agent can clear a stage, then the development of an intelligent agent is not necessary. The random agent executes random controls such as drop bombs and direction movements, such as up, down, left and right in each stage. As expected the random agent usually is killed by an

[1] `http://www.fceux.com/web/home.html`

[2] For the sake of fair use, we should mention that a purchased copy of *Bomberman* along with Nintendo console are owned by the authors. The emulator software is used purely for academic reasons.

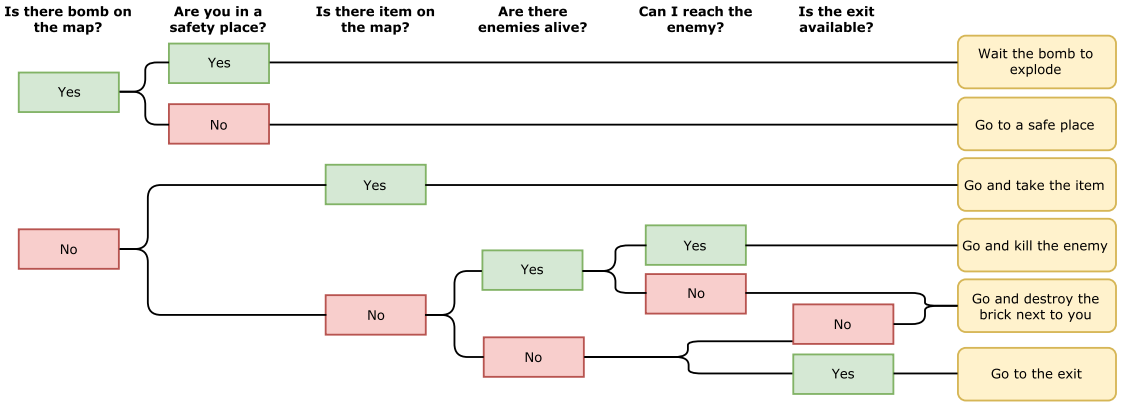



**Figure 2. Behavior tree for decision-making algorithm.**

enemy or by a bomb when exploding a brick. As a random agent is not enough to clear a stage, we decide to insert some intelligence to take actions according to the goal of the game.

### 4.2. Intelligent Agent

According to [Russell and Norvig 1995] an agent becomes intelligent when it is able to perceive its environment and acting upon that environment. In order to play the game, our intelligent agent has to perceive the environment before taking any action. *Bomberman* is a game containing an environment with arbitrary states, *i.e.*, the states are random when taking into account the enemies movements, the position of the bricks, power-ups and the exit door. Thus, before the agent perform each action, it has to collect data from the environment to the decision making algorithm.

Our decision-making algorithm can be represented as a behavior tree, as illustrated in Figure 2. As we can see in the behavior tree, the algorithm first searches for bombs in the map in order to rescue the agent in a safe place. In case there is a bomb in the map, the agent goes for a safe place and waits for the bomb to explode. Considering that there is not any bomb in the map, the agent searches for power-ups. If there is a power-up in the map, the agent collects the item, otherwise, the agent checks if there are reachable enemies in the environment. If so, the agent tries to kill the enemy, otherwise, the agent looks for the exit. In case the exit is not available, the agent searches for the closest brick to destroy in order to find the exit door. It is important to note that all actions that start with "*Go*" in Figure 2 (*i.e.*, "*Go to a safe place*", "*Go and take the item*", "*Go and kill the enemy*", "*Go and destroy the brick next to you*" and "*Go to exit*") use a *pathfinding* algorithm (Section 4.3) to perform the search for the best path to achieve the goal. The decision-making algorithm is performed for each frame using 60 frames per second (FPS).

### 4.3. Pathfinding Algorithms

Our intelligent agent uses *pathfinding* algorithms in actions that require the agent to walk through the map, such as the actions that start with "*Go*" in Figure 2. *Pathfinding* has been investigated for many years and generally refers to find the shortest route between two end points, being probably the most popular but frustrating AI problem in game industry [Cui and Shi 2011]. As such algorithms are developed to work on graphs, the first step is

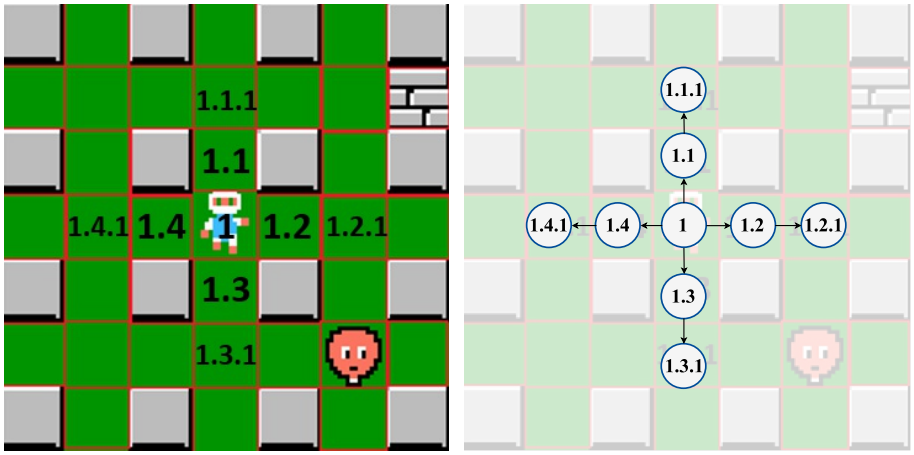



**Figure 3. Transforming the map of the game into an undirected graph.**

to transform the map of the stage into a graph. A graph is made up of nodes represented as the tiles the agent can walk through and edges represented by the connection of two adjacent tiles of the map. The generated graph must be directed since the agent should not return to the same position multiple times. An example of the transformation of the map into a graph is illustrated in Figure 3, where the tile where the agent stands is the root node (represented as "1"). Its four adjacent tiles ("1.1", "1.2", "1.3" and "1.4") create four nodes in the graph with directed edges connecting them with the root node. Nodes for tiles that are connected to adjacent tiles ("1.1.1", "1.2.1", "1.3.1" and "1.4.1") are created and directed edges related them to their adjacent nodes, and so on.

Having generated the directed graph, the next step is to apply *pathfinding* algorithms on the graph to search the best path to perform an action. In this work, we test two *pathfinding* algorithms: *Breadth-First Search* (BFS) and *A-Star Search* (A*).

- *Breadth-First Search (BFS)* is an important building block of many graph algorithms and commonly used to compute the shortest paths of unweighted graphs. Its strategy uses a FIFO (First In First Out) queue in the frontier, in which the shallowest unexpanded node is chosen for expansion. Thus, the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded [Russell and Norvig 1995]. Algorithm 1 describes the BFS process of selecting the best path to the agent perform each action.

- *A-Star Search (A*)* is classified as an informed search strategy, *i.e.*, instead of exploring the search tree blindly, the strategy tries to reduce the search space by making intelligent choices during the expansion. In order to evaluate the likelihood that a given node is on the solution path, the strategy uses an evaluation function $f(n)$ that determines which node is probably the "best" to expand. This evaluation function is constructed as a cost estimate, so the node with the lowest evaluation is expanded first, and is composed by a heuristic function $h(n)$ and the lowest path cost $g(n)$ as presented in Equation 1.

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

**Algorithm 1** Breadth-First Search Algorithm [Russell and Norvig 1995].

```
1: function BREADTH-FIRSTSEARCH(problem) return a solution, or failure
2:     node ← a node with STATE = problem.INITIAL-STATE
3:     PATH-COST = 0
4:     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
5:     end if
6:     frontier ← a FIFO queue with node as the only element
7:     explored ← an empty set
8:     loop
9:         if EMPTY?(frontier) then
10:            return failure
11:        end if
12:        node ← POP(frontier)                    ▷ chooses the shallowest node in frontier
13:        add node.STATE to explored
14:        for each action in problem.ACTIONS(node.STATE) do
15:            child ← CHILD-NODE(problem, node, action)
16:            if child.STATE is not in explored or frontier then
17:                if problem.GOAL-TEST(child.STATE) then
18:                    return SOLUTION(child)
19:                end if
20:                frontier ← INSERT(child, frontier)
21:            end if
22:        end for
23:    end loop
24: end function
```

In this work we consider the heuristic function $h(n)$ as the Manhattan distance, *i.e.*, the distance is based on a strictly horizontal and/or vertical path between the agent and the goal (enemy or brick). We decide to use Manhattan instead of any other distance since the map of each stage is a grid-like map leading to this type of measure. In order to calculate the distance between the two points, we sum the absolute difference between the coordinates, *e.g.*, taking into account the coordinate of the agent as $x_1$ and $y_1$ and the coordinate of the goal as $x_2$ and $y_2$, the resulting heuristic function is represented as $h(n) = |x_1 - x_2| + |y_1 - y_2|$.

For the lowest path cost $g(n)$ we consider the depth of the node in the graph. Thus, we consider the root node as $cost = 0$, its adjacent nodes have $cost = 1$, the nodes adjacent to adjacent nodes have $cost = 2$ and so on. An example of the application of the A* search is illustrated in Figure 4, where the agent has to walk up to the enemy in order to kill it. The algorithm to decide which path to walk calculates the heuristic function as $h(n) = |3-6|+|6-4| = 5$, where the first terms represent the positions in the $x$ axis and the last terms represent the position in the $y$ axis. Next step is to calculate the lowest path cost using the depth in the graph to reach the enemy. Thus, the algorithm must decide between going up (path "A") or going down (path "B"). Using the Manhattan distance, the algorithm calculates $g(n) = 9$ to the path "A" and $g(n) = 5$ to the path "B". Finally, the evaluation function for the path "A" results $f(n) = 5 + 9 = 14$ and for the path "B" results $f(n) = 5 + 5 = 10$. As the evaluation function searches for the lowest value, the path "B" is selected.

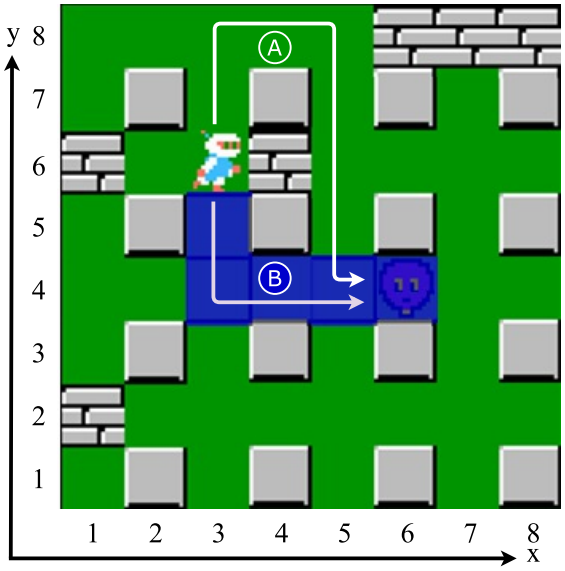



**Figure 4. Application of A* search to walk up to the enemy, where (A) and (B) are two possible paths the algorithm can take.**

## 5. Results and Discussion

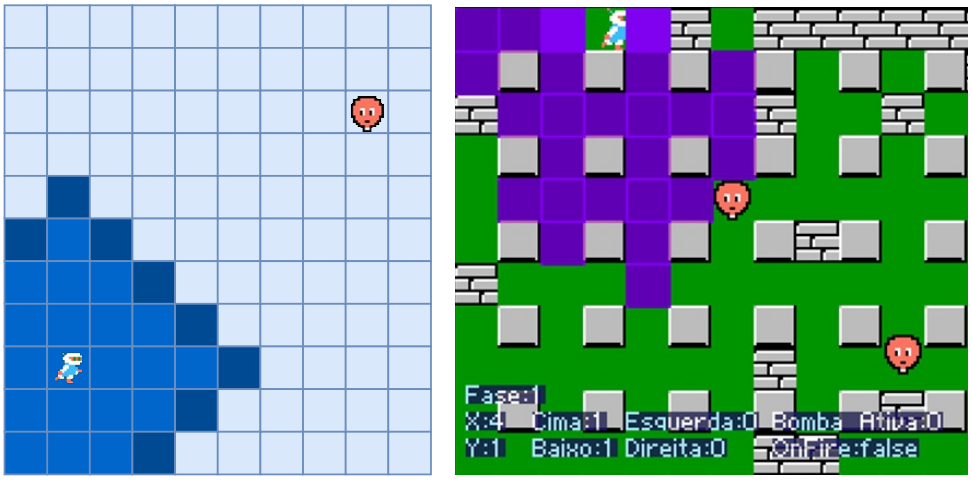
In our tests, both *pathfinding* algorithms implemented in the intelligent agent achieve success when playing *Bomberman*, *i.e.*, in both algorithms all enemies are killed and the exit door is found in every stage. In order to visualize each approach when the AI is playing *Bomberman*, we decide to print actions on the screen as well as to paint all blocks the *pathfinding* algorithm is exploring. Such visualization allows us to analyze how each approach expands the graph of the scenery and all possible paths the algorithm can take. Based on this visualization, we explain the impact of each algorithm when playing the game.

### 5.1. Breadth-First Search

BFS algorithm has a circular aspect since it explores nodes equally in all directions, *i.e.*, it explores all nodes of the same level, before exploring the next level, as illustrated in Figure 5 (a). Thus, when visualizing the approach in *Bomberman*, we expect that many tiles should be on the frontier until the algorithm reaches the enemy. As (b) in Figure 5 demonstrates, the algorithm explores most tiles around the agent and the frontier is expanded in all directions. Although this algorithm generates the optimal path to the enemy, *i.e.*, the best path the agent can take to reach the enemy, it requires a lot of memory since it keeps all nodes of the graph in memory as the frontier is expanded.

### 5.2. A-Star Search

Unlike BFS, A* search algorithm explores nodes using an evaluation function that contains a heuristic and the weights of each path, thus, giving to the agent a new approach to explore the way to the goal. The evaluation function reduces considerably the number of nodes to be expanded each step when compared with BFS. As A* is an informed search, it tends to explore the path directly to the goal, as illustrated in Figure 6 (a). Figure 6 (b) shows the application of the A* algorithm in the game, where we can observe that the path is divided into two alternative ways (*i.e.*, going up or down the fixed tile).




(a) (b)

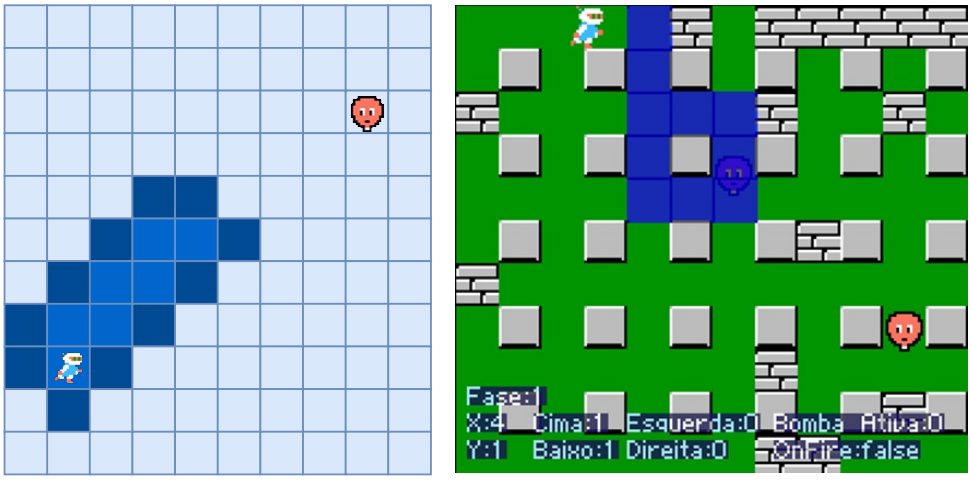
**Figure 5. Search space to find a path between the agent and the enemy using BFS algorithm. (a) illustrates how the search space has to expand, and (b) the search space in the game.**

Comparing Figure 5 (b) and Figure 6 (b) we can see that A* achieves better performance than BFS, since this method does not result in an exhaustive search. This difference is also observed when analyzing the frames per second (FPS) generated when each algorithm is running. When running A* algorithm, we captured a maximum of 800 FPS and a minimum of 120 FPS, while running BFS algorithm we captured a maximum of 600 FPS and minimum of 20 FPS. The idea of analyzing the FPS is that it correlates with the processing load, thus the more processing the computer needs, the lower the FPS of the game. Although the difference in FPS, it is important to mention that the only difference between the approaches relies on the performance aspect since the result for both algorithms must be optimum.

In order to compare the performance of our algorithms, we use the score and time the agent takes to clear the stage. Although the emulator allows us to start each stage with the same configuration (position and number of enemies), the game is stochastic in the sense that the enemies may change their moves at random. Thus, although the agents of both algorithms start at the same point, they will hardly perform the same path to the exit door. When trying few runs of the game we observed that both algorithms take about the same time to clear the stage. A difference of about 3 seconds in favor of A* was observed, but we suspect that this difference occurred due to the non-determinism characteristic of the enemies' movements.

### 5.3. Intelligent Agent vs. Human Player

It is important to say that not all artificial intelligence in games plays in a human-like manner. As pointed out by [Ortega et al. 2013], usually controllers that are hand-coded to play a particular game and controllers that are trained to play a game using some sort of machine learning mechanism, frequently display behavior that strikes observers as "unnatural" or "mechanical". On the other hand, some AI agents can play impressively in a human-like manner that usually may confuse other users (*e.g.*, Garry Kasparov complained when he lost the chess match to the IBM software/hardware Deep Blue that the




(a) (b)

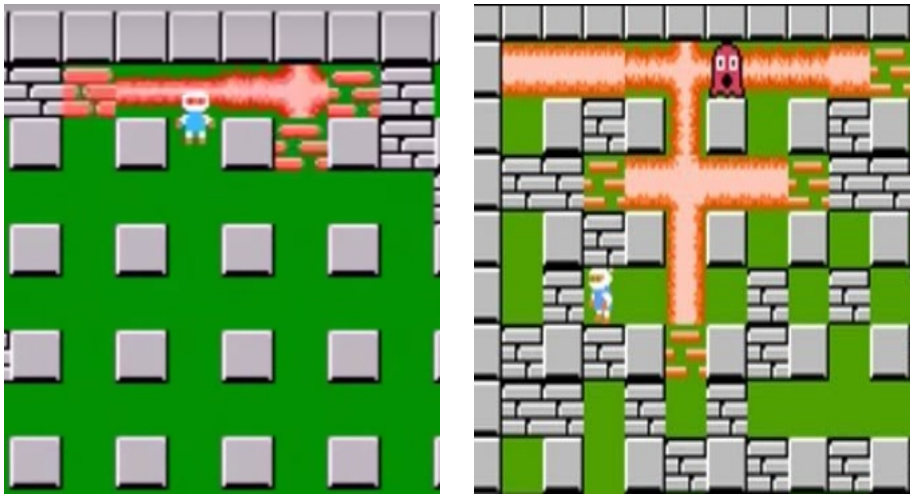
**Figure 6. Search space to find a path between the agent and the enemy using A* algorithm. (a) illustrates how the search space has to expand, and (b) the search space in the game.**

computer played too much similar to the human, given that the previous computers played in a machine-like manner [Newborn and Newborn 1997]).

Unlike imitation learning techniques where the agent learns the movements from the expert demonstration and tries to mimic his movements, our agent uses a behavior tree as decision-making algorithm. As our algorithm works similarly to the human vision in the sense that our agent can see the same board as a human, *i.e.*, we do not allow the agent to discover what items are underneath each tile before breaking the tile, the agent should perform similarly to a human being. On the other hand, as our agent is not influenced by the human behavior, it could not try to mimic the playing style of a human player.

Observing the intelligent agent in action, it seems very similar to a human player, since it drops bombs close to bricks, and when there is not an enemy close to the agent, it tends to explode the nearest brick. The unique fact that we can identify the player as an intelligent agent is the fact that the AI knows which is the first tile in which the agent is safe from explosion. Thus, when running away from an explosion, the agent usually stays very close to the border of the tile, being very close to the explosion, while a human player tends to stay in the middle of the tile. Figure 7 illustrates the difference between the intelligent agent position when getting away of an explosion (a) and the position of *Bomberman* when a human being is playing the game (b).

When comparing how long the agent takes to clear the stage against the human player, a subject was invited to play two stages (stage 1 and stage 31). We choose these states because in the first stage *Bomberman* does not have any power-up, while in stage 31 *Bomberman* has power-ups, such as the clock-bomb, invulnerability of fire, *etc.*. It is important to note that the human player has previous knowledge on how *Bomberman* is played and how to clear the stage. Both the intelligent agent and the subject start in the same conditions and position. Time was recorded in four rounds in both stages and the average time and standard deviation are presented in Table 1.



(a) (b)

**Figure 7. Difference between the Artificial Intelligent and a human playing *Bomberman*.**

**Table 1. Human and AI average time (in seconds $s$) and standard deviation ($\sigma$) when playing stage 1 and stage 31.**

| Player | | $s$ | $\sigma$ |
|---|---|---|---|
| Human | Stage 1 | 129,50 | 21,44 |
| | Stage 31 | 40,00 | 47,71 |
| Intelligent Agent | Stage 1 | 139,50 | 35,35 |
| | Stage 31 | 46,00 | 8,57 |

Observing Table 1 we can see that the human player clear the first stage in less time than the intelligent agent. Although the human achieved a better result when compared to the agent, the human could not clear the stage in 4 opportunities, while the intelligent agent was not killed by the enemy in any trial. The standard deviation of both, AI and human, shows that the difference between the averages is not significant, concluding that AI and human need approximately the same amount of time to conclude the stage 1. In stage 31 the human complete the phase in a shorter time than the intelligent agent. However, the results generated from this analysis demonstrate that the human player and the AI have closer completion time for stage 31 when we observe the average time and the standard deviation.

The major advantage of the intelligent agent when compared to humans is undoubtedly the ability to process information quickly. In the tests discussed above, the AI was executed at the same speed as we would play, however, the agent can be executed at very high speeds, which only depends on the hardware that is executing the game, being able to complete the 50 stages of the game in minutes. However, as we are limited when processing information, we are unable to make fast decisions when playing at 800 frames per second, since it is inconceivable for our brain to process so much information at high speed. Finally, it is important to note that the intelligent agent was not developed to re-

place the human being, but to assist it, serving as the basis for receiving AI techniques that need to be tested in the digital game.

## 6. Related Work

[Lucas 2008] mentions the benefits of using artificial intelligence in games for testing and solving real world problems. He shows how parameter optimization techniques can be applied in games and what are the benefits obtained. He concluded that for most analyzed games the obtained results are excellent. Thus, for the optimization techniques applied, games as test environments are a successful choice.

[Silver et al. 2016] and [Mnih et al. 2015] use games to apply Deep Q-Learning (DQN) in order to perform tasks like or better than humans. In both work, the application returned excellent scores. In the first case, the algorithm is applied in order to play various Atari games, achieving good scores in most of them. In the second case, the algorithm is applied to play the board game Go, obtaining an excellent performance even against a Go world champion. Although reinforcement learning approaches achieve good results, they demand a lot of time to train and a reward function. It is also important to mention is that this approach will not work properly in states that have not seen before.

## 7. Conclusions and Future Work

In this work, we analyzed the feasibility of applying artificial intelligence in the digital game *Bomberman*. We developed a behavior tree in order to decide the actions of the intelligent agent. In order to take some decisions, the behavior tree contains two *pathfinding* algorithms[3]: BFS and A*. The A* technique demonstrated itself efficiently when compared with BFS, once it does not need to compute all the nodes of the tree path to find a solution. Comparing the AI with the human being playing the game, we can observe several similarities taking into account the behavior of the agent as well as the time it takes to clear a stage. From the obtained results we observed that digital games are capable of receiving AI techniques.

As future work, we intend to explore the application of ontologies and planning strategies in *Bomberman*, in order to help the process of decision-making. We also intend to apply multi-agent approaches or even transpose these techniques for a physical environment using robotics.

## Acknowledgement

The authors would like to thank the Brazilian funding agency CAPES and Motorola Mobility for the financial support for the development of this work.

## References

Cui, X. and Shi, H. (2011). A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130.

Holmgard, C., Liapis, A., Togelius, J., and Yannakakis, G. N. (2014). Personas versus clones for player decision modeling. In Pisan, Y., Sgouros, N., and Marsh, T., editors,

[3]A demo containing both algorithms can be found in the authors' YouTube channel: `https://www.youtube.com/channel/UCUM-vwwuUYbPwSUfL7rsC5w`

*Entertainment Computing – ICEC 2014*, volume 8770 of *Lecture Notes in Computer Science*, pages 159–166. Springer Berlin Heidelberg.

Idrees, H., Zamir, A. R., Jiang, Y.-G., Gorban, A., Laptev, I., Sukthankar, R., and Shah, M. (2017). The {THUMOS} challenge on action recognition for videos "in the wild". *Computer Vision and Image Understanding*, 155:1–23.

Liapis, A., Yannakakis, G., and Togelius, J. (2013). Generating map sketches for strategy games. In *Proceedings of 16th European Conference on Applications of Evolutionary Computation*, pages 264–273, Berlin, Heidelberg. Springer Berlin Heidelberg.

Lucas, S. M. (2008). Computational intelligence and games: challenges and opportunities. *International Journal of Automation and Computing*, 5(1):45–57.

Miranda, M., Sánchez-Ruiz, A. A., and Peinado, F. (2016). A neuroevolution approach to imitating human-like play in ms. pac-man video game. In *Proceedings of the 3rd Congreso de la Sociedad Española para las Ciencias del Videojuego*, CoSeCiVi'16, pages 113–124. CEUR-WS.org.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Newborn, M. and Newborn, M. (1997). *Kasparov versus Deep Blue: Computer chess comes of age*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Ortega, J., Shaker, N., Togelius, J., and Yannakakis, G. N. (2013). Imitating human playing styles in super mario bros. *Entertainment Computing*, 4(2):93–104.

Rogers, S. (2014). *Level Up! The guide to great video game design*. John Wiley & Sons.

Russell, S. J. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Ed. Prentice Hall, New Jersey, 2nd edition.

Schuurmans, D. and Zinkevich, M. A. (2016). Deep learning games. In *Advances in Neural Information Processing Systems 29*, pages 1678–1686. Curran Associates, Inc.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Wang, H., Wang, N., and Yeung, D.-Y. (2015). Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1235–1244, New York, NY, USA. ACM.