Bomberman Reinforcement Learning Agent

Filipe Gonçalves	Miguel Beirão	João Borges	Daniela Dias	Gonçalo Machado
DETI	*DETI*	*DETI*	*DETI*	*DETI*
MRSI	*MEI*	*MEI*	*MEI*	*MEI*
Aveiro, Portugal	Aveiro, Portugal	Aveiro, Portugal	Aveiro, Portugal	Aveiro, Portugal
98083	98157	98155	98039	98359

***Abstract*—Throughout history, the gaming industry has continuously evolved. It began with simple games such as tic-tac-toe and chess and has now progressed to the cloud gaming and streaming era. This modern era allows players to stream games directly to their devices, eliminating the need for high-end hardware. However, many argue that the golden age of this evolution was during the emergence of the first PC and arcade games. During this period, iconic games like Super Mario Bros, Legend of Zelda, Donkey Kong, and Pac-Man captured the imagination of players and fueled the demand for new and improved ways to entertain them. In this context, the proposed project aims to develop a Reinforcement Learning agent capable of autonomously playing the game Bomberman.**

***Index Terms*—Bomberman, Reinforcement Learning, Grid, Game**

I. Introduction

Bomberman, a beloved video game series that originated in the 1980s, has established itself as a timeless classic in the gaming industry. The game objective is to strategically navigate the Bomberman character through a maze-like grid of blocks and walls while strategically placing bombs to eliminate obstacles, defeat enemies, and ultimately reach the exit of each level.

Whether played in single-player or multiplayer mode, Bomberman requires players to consider their positioning and timing when deploying bombs to avoid being caught in explosions. The bombs have a blast radius capable of destroying enemies and walls, but players must be cautious as they can also harm the Bomberman character if caught within the blast range.

Throughout the game, players can collect power-ups and items that augment Bomberman's abilities. These power-ups may increase the bomb's range, enable the simultaneous placement of multiple bombs, or provide additional capabilities such as enhanced speed or the ability to move through walls.

The popularity of Bomberman can be attributed to its straightforward yet captivating gameplay, the joy of engaging in multiplayer battles, and the nostalgic charm it evokes. This game has become a staple in the gaming industry and has garnered a dedicated fan base worldwide.

In light of this, our objective is to develop a Reinforcement Learning agent that learns through a reward/punishment-based system to successfully complete the levels and beat the game.

II. Method

When approaching this project, we made a deliberate decision to refrain from using external Machine or Deep Learning packages and, instead, devised a unique solution using only the core Python libraries.

The sequence of actions we implemented follows a straightforward approach:

Firstly, we establish a base grid that contains only the indestructible walls and, with each step, we receive information from the server regarding the presence of destructive walls, enemies, power-ups, the Bomberman's position, and the location of the exit. Using this information, we update the map accordingly.

Next, with the updated map, we employ a search algorithm that determines the optimal path and next position for the Bomberman based on the accumulated total reward. Taking into account various factors, such as potential risks (i.e. enemies and bombs), rewards, and strategic opportunities, we determined the best course of action to progress toward the desired position.

Finally, after determining the optimal position, we strategically place the bomb in the correct location, allowing the Bomberman to autonomously navigate the game.

Consequently, we created three distinct approaches, each employing different reward/punishment values and various methods and techniques to search for optimal moves. By exploring these different strategies, we aimed to maximize the Bomberman's performance and enhance its ability to conquer the game.

A. Approach 1

This approach distinguishes itself from the other ones by its combination of simple yet efficient ideas, with an implementation that is both complex and concise.

However, it is important to acknowledge the limitations of this approach. One notable drawback is that there are instances where Bomberman may get stuck and become unable to progress to the next level. Additionally, we are unable to utilize the Detonator power-up or leverage the Bombpass ability to our advantage.

Moving forward, we will outline our approach's key features and functionalities and explain how each component operates.

1) Reward/Punishment System: In this approach, we use the following rewards or punishments:

- Indestructive_Wall = -100
- Destructive_Wall = 6
- Destructive_Wall_Exit = 4.5
- Enemy_Area = -8
- Enemy_Position = -20
- Enemy_Bomb = 20
- Bomb_Drop_Position = -10
- Bomb_Damage = -10
- Step = -0.01
- Player = -0.02
- Powerup = 10
- Exit = 50

The **Indestructible Wall** refers to a wall we can neither break nor pass through. The **Destructible Wall** and **Destructible Wall Exit** represent walls that can be destroyed and walked through with the Bomberpass power-up when we don't know the exit and when we know the exit, respectively. **Enemy Area** is for the blocks close to an enemy position, **Enemy Position** is for the block where an enemy is located, and **Enemy Bomb** is the positive reward indicating the block to place the bomb in order to kill an enemy with a higher probability of success. **Bomb Drop Position** and **Bomb Damage** consider the position and radius of a bomb when it's placed in the game and, with this, we don't need to add additional code to the power-up Flames (which increases the radius of the bomb). **Step** is a normal move. **Player** is the current position of the player. **Powerup** is for the position of the power-up; finally, **Exit** is for the position of the Exit.

In this approach, we decided first to have different rewards for enemies and destructive walls. This takes into consideration that each enemy will give points, which matter to the highscore, while walls only give us access to the exit or to a powerup. Secondly, we separate the destructive wall into two different categories, because, if we have found the exit, we don't need to break more walls and should only do it if they are in the way of an enemy.

These rewards are the ones we will input into our grid, allowing our search algorithm to find the best possible next move.

2) Map Updater: Each time new information is received from the server, we need to update our map with our rewards, as seen previously.

As such we used a hierarchy based on importance:

- Indestructive walls, Step and Player
- Exit and Power-ups
- Bombs
- Walls
- Enemies

As such, we'll first update our map with all indestructible walls with reward -100, our player position with reward -0.02, and every other block with reward -0.01. Secondly, because the exit and power-up are always in different places, we add them to our map. After that, we add the placed bombs with their radius, and the destructive walls, which are -100 for their position and their respective reward for the positions beside them. Lastly, we place the enemies with -20 reward, update their area, and set the best positions to place a bomb, which will be discussed later.

3) Enemy Best Position to Kill Calculator: Whenever we want to kill an enemy, we need to know the best position to place a bomb. With this goal in mind, we created two different methods that complement each other: enemy direction detection and enemy radius system.

The enemy direction detection will predict the next position of an enemy based on the last position and last direction and it will place positive rewards, **Enemy Bomb**, on the three next positions, excluding the closest next position. The only two downsides are that, if an enemy is changing directions and we place a bomb where we thought the enemy would go next, then the enemy will evade the bomb. This is why we decided to also look into any walls next to the enemy - if the enemy reaches a wall, then we know he will need to change directions and we increase our radius. The second downside is that, if an enemy is in a loop, going back and forth, the Bomberman will become indecisive on where to plant a bomb.

The enemy radius system is very simple. We decided to create a radius of positive reward (7 blocks/positions) to lure the Bomberman into any enemy, with greater reward the closer the block is to the enemy. With this, we solved two problems:

- The search method no longer has problems finding good next positions for the Bomberman (avoiding the need for searching a lot of blocks/nodes).
- The Bomberman now chases after the enemies and, in case of a change of direction, he has plenty of time to go around them and place a bomb right in front of them in order to kill them.

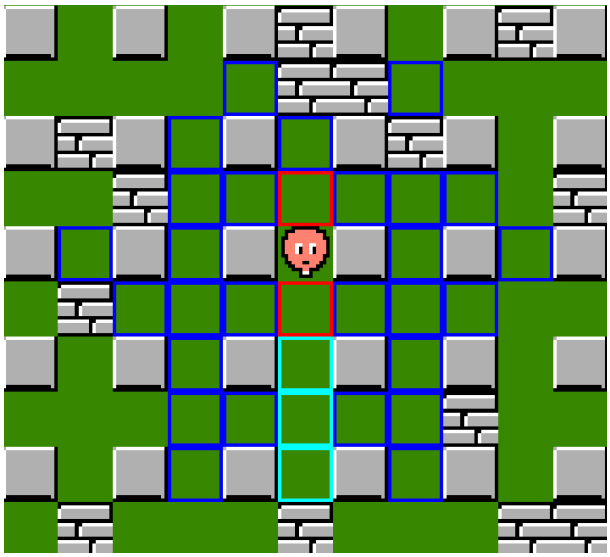

Fig. 1. Enemy Radius and Bomb placement

As we can see, in red we have a negative reward for the area near the enemy, in light blue a positive reward to place

a bomb, and in darker blue we have a positive radius reward.

4) Dynamic Exit: In this approach, we decided, instead of placing the **Exit** reward the moment we find the exit, putting into jeopardy killing all the enemies in the level, to only update the map with the exit the moment we know there are no more points available in the level.

We also implemented a radius system, similar to the enemy radius, however, it will decrease from 50, which is the **Exit** reward, until it reaches 0, which is not placed on the map. With this, we have almost the whole map filled with positive rewards, all nearing the exit. We noticed that, with this, the Bomberman will very often reach level 2, but also get stuck and not move up to the next level sometimes.

5) Greedy Search: The search method is very similar to the ones used in other projects and other classes, with the main features being:

- Search tries to find the best node until the current node opened has a depth equal to or higher than eighteen.
- If the reward of the current node is positive, then the search is concluded.
- If the position resulting from the action was already on any ancestor of the node, then the node is not opened.
- If the position resulting from the action was already in any nodes already opened but had less reward, then the node is not opened.
- If the position resulting from the action is not a valid position, then the node is not opened.
- If the total reward, based on the current opened node reward and the next action reward is very low, then the node is not opened.
- If after the loop there was no ideal node found, then the next Bomberman position will be the node with the best reward.

6) Bomb Placement: To place the bombs, we simply identify if the Bomberman is in the best position to place a bomb. If affirmative, we save a boolean value for the next step. In this step, we don't need to search for the best node, we simply place a bomb and negate the boolean saved before.

With this, we can use the power-up Bomb Counter, without having to do any additional code.

B. Approach 2

This approach distinguishes itself from the other ones by its greater adaptability to the most diverse sets of game conditions, simplified reward system, and use of countermeasures to navigate through the most difficult circumstances.

However, this approach also comes with its limitations. Similarly to Approach 1, there are instances, although fewer, where Bomberman may get stuck and become unable to progress to the next level due to its limited search. With this limited search, the Bomberman agent may also struggle to find enemies that are too far away.

Additionally, there's a lack of prediction and the Bomberman agent may place bombs in non-ideal positions, cornering itself between a bomb and an enemy in the next steps.

At the same time, the simplified reward system reduces complexity while still capturing the essential aspects of the game, but at the cost of potentially creating problems due to the lack of distinction between enemies and walls, which we'll address in later sections.

1) Reward/Punishment System: In this approach, we use the following rewards/punishments:

- Valid Move = -0.01
- Indestructible Wall = -100
- Damage = -55
- Reward = 30
- Powerup = 20
- Exit = 50
- Player = -10

In this approach, we simplified the reward system seen in Approach 1. We combine the concepts of Destructive Wall, Destructive Wall Exit, and Enemy Bomb into a single category called **Reward**. Meanwhile, Enemy Area, Enemy Position, Bomb Drop Position, and Bomb Game are covered by the negative reward **Damage**. This simplification allows us to streamline the reward system and make it more concise.

Additionally, the rewards for each category have been adjusted to reflect their significance in the game. For example, Valid Move and Player have negative rewards to discourage unnecessary movements and potential harm to the player. Indestructible Wall carries a high negative reward to discourage interaction with impassable walls. Damage incurs a significant penalty to discourage actions that could harm the player. On the other hand, **Reward**, **Powerup**, and **Exit** offer positive rewards to encourage beneficial actions such as collecting rewards and reaching the exit.

This simplified reward system reduces complexity while still capturing the essential aspects of the game and providing meaningful incentives for the agent's decision-making process. However, it comes with the trade-off of potentially creating problems due to the lack of distinction between enemies and walls. Here are a few issues that may arise:

- Without distinct rewards for enemies, the agent may not prioritize actions that eliminate or avoid enemies efficiently. The agent will give the same priority to destroying walls and destroying enemies, when destroying enemies should be a higher priority.
- By not assigning separate rewards for walls, the agent may not appropriately weigh the benefits of breaking destructible walls. Walls may block paths or prevent progress, and breaking them can potentially reveal power-ups, shortcuts, or the exit.

2) Map Updater: When receiving new information from the server, we need to update our map with the rewards. To accomplish this, we employ a hierarchy based on importance, depending if the level is already considered finished or not.

If the level is finished:

- Walls
- Exit

If the level isn't finished:

- Walls
- Bombs
- Enemies
- Powerups

3) Dynamic Rewards: This approach incorporates dynamic rewards in the form of varying and accumulating rewards that represent both the different game elements (or events) and the interactions between such elements.

For example, the rewards around destructible walls that are close to each other can overlap, creating positions where the reward is greater. These increased rewards describe positions where placing a bomb destroys multiple destructible walls at the same time. The same logic can be applied to enemies.

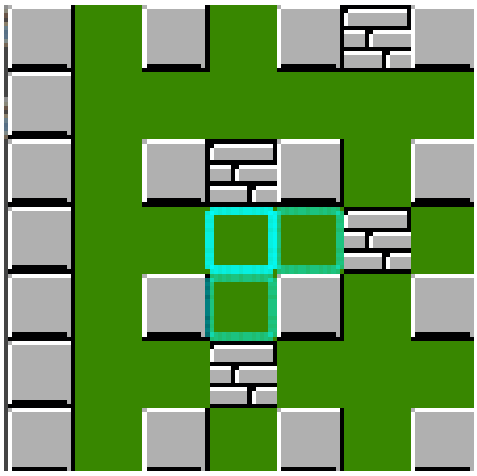

Fig. 2. Dynamic Rewards

These dynamic rewards are designed to provide feedback to the agent and influence its decision-making process. They also take into account the previous 7 positions to avoid most loops (a countermeasure to one of the mentioned limitations of Approach 2) - if a position in the map coincides with one of the 7 previous positions of the agent, the original reward is slightly decreased.

4) Wall Rewards: For each destructible wall in the game, we retrieve its position and update the game map by assigning a negative reward, specifically the value for Indestructible Wall. This indicates the presence of a wall that cannot be crossed.

However, if we have already found the exit and do not need to destroy any more walls (i.e., the number of walls is less than 8 and the exit has been found), we skip updating the area around the destructible wall with reward.

Next, we check whether the wall will be destroyed by a bomb placed by the Bomberman. To determine this, we consider the coordinates of the wall and the information stored in the list of all bombs. If it is determined that the wall will be destroyed by a bomb (i.e. the radius of a bomb explosion overlaps the wall), we continue to the next wall without updating the area around it, as it will soon be removed from the game.

If the wall is not targeted by a bomb, we proceed to update the reward values around the destructible wall. To accomplish this, we consider a propagation range, denoted as "Propagate Range", which is set to 20. Within this range, we calculate all possible positions based on the current range that are not occupied by an indestructible wall and update the corresponding positions with reward.

Furthermore, we introduce a decay factor. This factor begins at 1 and increases by 1 for each position that cannot be updated due to the presence of an indestructible wall. This mechanism ensures that the reward diminishes as more indestructible walls are found between the Bomberman and the destructible wall.

The reward value assigned to each position surrounding the destructible wall depends on the distance from the wall. It is inversely proportional to the square of the distance and is calculated as follows: **(Reward / currentRange^2) / decayFactor**.

By applying these calculations and considerations, we appropriately update the reward values in the area surrounding each destructible wall, taking into account the presence of indestructible walls and the diminishing impact of rewards with increasing distance.

5) Enemy Rewards: For each enemy in the game, we gather its position and name. Based on the enemy's name, we determine the values for its speed and whether it can pass through walls. First, we update the game map at the enemy's position with a negative reward value indicating the presence of the enemy (**Damage**).

If the enemy is currently moving, we calculate its movement direction. In cases where the enemy's direction is unknown or the enemy is not moving, we iterate over all possible positions around the enemy and update the game map with the **Damage** reward, considering positions that are not occupied by indestructible walls.

Movement Simulation: Using the enemy's movement direction, we simulate its movement by considering its speed. We calculate the next positions the enemy will occupy (number of positions equal to its speed) and update the game map with the **Damage** reward at those positions.

If any of the calculated positions are obstructed by an indestructible wall and the enemy does not have the ability to pass through walls, we exclude the following positions from the simulation.

Bomb Simulation: Next, we simulate the presence of a bomb in the direction of the enemy. We calculate the potential bomb positions based on the bomb's radius and timeout and update the game map with a reward value (if the position is not occupied by an indestructible wall). The reward value assigned to each position depends on the distance from the enemy and is inversely proportional to it **(Reward / distance)**.

If a calculated position is occupied by an indestructible wall and the enemy does not have the ability to pass through walls, we only add a reward value to the next position (if it is not an indestructible wall) and terminate the map update.

Propagation of Reward: Finally, we propagate the enemy's influence in the game map by assigning a slight reward. This is

motivated by our limited search algorithm, which may struggle to consider enemies located far away.

We propagate a small reward in the opposite direction of the enemy's movement. Additionally, we update the game map with slight reward values for positions adjacent to the enemy's movement direction. These rewards gradually diminish as the distance from the enemy increases, ensuring that the impact of distant enemies is accounted for in our map representation.

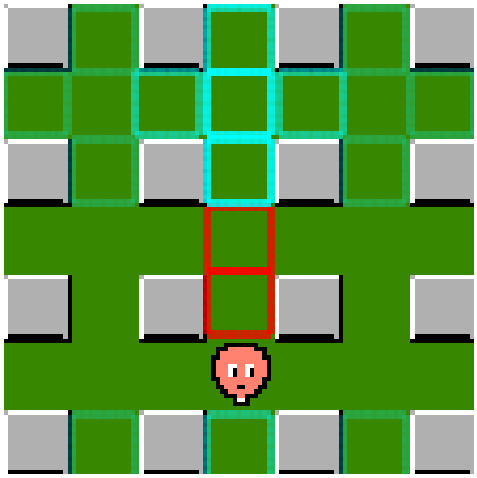

Fig. 3. Enemy Rewards

6) Bomb Rewards: For each bomb in the game, we retrieve its position, timeout, and radius. We update the game map at the bomb's position with a negative reward value indicating the presence of the bomb (**Damage**).

Bomb Explosion: Next, we iterate over all positions relative to the bomb within the range of its explosion radius. Starting from 1 up to the bomb's radius, we update the game map at each position with a reward value determined by the bomb's countdown and the distance from the bomb. A lower countdown and a position closer to the bomb result in worse rewards. The reward calculation for each position is **(4 / bombCountdown) * Damage / (currentRange + 1)**.

Safe Positions around the Bomb: In addition to the positions affected by the bomb's explosion, we also consider safe positions adjacent to those covered by the explosion. If a safe position is not occupied by an indestructible wall, we update the game map with a reward value equal to Reward / 2. This encourages the agent to prioritize choosing safe positions around the bombs.

By considering both the impact of the bomb's explosion and the availability of safe positions, we provide the agent with valuable information to make informed decisions and avoid areas of danger during gameplay.

7) Exit Rewards: In this approach, similarly to Approach 1, the exit reward is only updated on the map when we know that there are no more points available in the level (i.e. no more enemies).

By implementing a radius system (where the reward decreases with the distance to the exit, **Exit / (distance + 1)**), we ensure that the majority of the map is filled with positive rewards that gradually lead the Bomberman towards the exit.

However, by employing a breadth-first search instead of a greedy search, the Bomberman successfully navigates towards the exit and progresses to the next level without getting stuck in loops (overcoming the problem faced by Approach 1).

8) Breadth-first Search: Our search algorithm, with breadth-first search, aims to find the best node with the highest reward by considering different actions. The search continues until one of the following conditions is met:

The current node's depth exceeds 12. An ideal reward is found, either a close reward or a great reward.

A close reward is determined when the total reward is positive and the reward of the current node is equal to or greater than **Reward - 2**, and the depth of the next node is at most 2. A great reward is achieved when the total reward is equal to or greater than **Reward + 10**.

Certain conditions prevent a node from being opened. In this approach, we have included some conditions from Approach 1 and added new ones:

- If the position resulting from the action is already present in any ancestor of the current node.
- If the position resulting from the action is already in the opened nodes but had a lower reward.
- If the position resulting from the action corresponds to an indestructible wall.
- If the total reward is positive but the current reward is negative.

If the reward of the current node meets the criteria for an ideal reward, the search is concluded. However, if no ideal node is found after the loop, the next position for Bomberman will be determined based on the node with the best reward.

By applying these conditions, the search algorithm ensures that it explores the available nodes effectively, avoids redundant positions, and prioritizes nodes with higher rewards to guide the actions of the Bomberman agent.

C. Approach 3

1) Reward/Punishment System: In this approach, we use the following rewards/punishments:

- Indestructible_Wall = -100,
- Destructive_Wall = 4,
- Enemy_Area = -15,
- Enemy_Position = -20,
- Enemy_Bomb = 10,
- Bomb_Damage = -5,
- Powerup = 5,
- Step = -0.01,
- Bomb_Drop_Position = -10,
- Exit = 50,
- Player = -0.01

Indestructible Wall is for all the blocks that represent either a wall we cannot pass through or one we can break. **Destructive Wall** is for the blocks around the walls that can be broken. **Enemy Area** is for the blocks close to the enemy.

Enemy Position is for the block where an enemy is located. **Enemy Bomb** is for the blocks where if a bomb is placed an enemy is likely to die. **Bomb Damage** is for the blocks that the bomb explosion will reach. **Powerup** is for the block that contains a powerup. **Step** is for a normal move. **Bomb Drop Position** is for the block in which a bomb was placed. **Exit** is for both the block in which the exit is located and for the area around it. **Player** is for the current position of the player.

In this approach, similar to Approach 1, we decided to have different rewards for destructive walls and enemies due to the requirement of needing to kill all enemies before being able to advance to the next level.

The rewards are lowered due to a difference in how they are distributed around the map. In this approach, any overlapping rewards will be slightly increased.

2) Dynamic Rewards: Since the map is filled with randomly placed walls and moving enemies, we can affirm that some locations will be better than others due to the close proximity of rewards. For that reason, this approach tackles this problem by accumulating the rewards in a dynamic way.

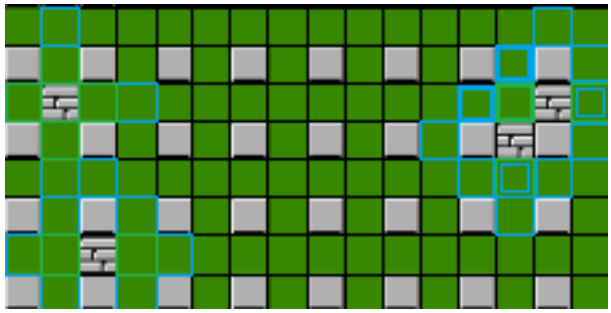

Fig. 4. Dynamic rewards visualization

Taking image 4 as an example, in which the radius of the wall was decreased to 2 for easier viewing. We can see that the radii of the walls on the left don't overlap, therefore, the locations around them are only affected by one wall and so have a simple reward.

On the other hand, the radii of the walls on the right do overlap, and where they overlap the reward should be higher. These rewards are based on all the rewards given to the location, with the greatest one used as the base and all the other rewards decreased by 90% and added to it.

This allows for the area surrounding the two close walls on the right side of the map to have better rewards than the ones on the left. This also allows the location where one bomb can hit both walls to have the best reward, without making the rewards too big.

Enemies' rewards are also affected by this, allowing locations where bombs that can hit multiple targets to have a slightly better reward than ones that can only hit one.

3) Map Updater: When new information from the server is received, the map needs to be updated. For this, we use a hierarchy based on importance:

- Player
- Bombs and enemies
- Walls
- Powerups
- Exit

4) Enemy Areas: When the enemies' data is received, their current location and names are stored. This data is then used to calculate the trajectory in which the enemy is moving. This is done by comparing each enemy's current position with the ones previously stored, and the difference will give us their current trajectory.

Afterward, the map is updated with the enemy's current location. Their location and all valid blocks within a 2-block radius are all given negative rewards, while blocks within $EnemyRadius$ [$EnemyRadius = 15blocks - enemiesleft$] are given 10% of **Enemy Bomb** reward, decreasing the further away they are from the enemy.

Finally, using the calculated trajectory and the bomb range, we give the positions that the enemy is headed to and that a bomb can reach **Enemy Bomb** reward.

5) Breadth-first Search: In this approach, we also used a Breadth-first search algorithm, with the aim of finding the location with the best local reward. If none was found then the location with the best total reward is considered the best. A local reward is a reward given to each position, while a total reward is all the rewards in a path.

The algorithm searches every node with a depth below 7. It searches for the location with the best local reward, updating the best node every time a better local reward is found or whenever a node with the same local reward but a lower depth is found. If no location is found with a better local reward than the location of the player, then the location with the best total reward is deemed the best.

We also used the conditions from Approach 1 to prevent nodes from being open, with some differences:

- If the resulting position from the action corresponds to an indestructible wall or enemy;
- If the total reward is below -30.

With these conditions, we can have a guarantee that only valid nodes are explored and nodes whose path is too risky are ignored.

III. Results

The first approach was the only one to reach level 3, finishing with a great total of 2700 points.

Fig. 5. Level 3 Reached in the 2700 run

To test the approaches, we ran them 25 times each and reach these results:

	Low Score	Average	High Score
Approach 1	100	700	1500
Approach 2	100	840	1200
Approach 3	100	450	900

TABLE I
RESULTS IN 25 RUNS

With the highest score while testing being:

	Highest Score
Approach 1	2700
Approach 2	1500
Approach 3	1100

TABLE II
HIGHEST SCORE PER APPROACH

While the first approach seems better when testing with a few runs, the second approach is much more consistent, reaching level 2 many more times.

IV. Improvements

To improve our approaches we should, firstly, end the possible loops with a simple loop detection and create other countermeasures for when that happens. We should also improve the rewards/punishments system as we might not have the most optimal and efficient values. Lastly, in the first approach, we should add the cumulative reward feature for walls and enemies, as seen in the other approaches.

V. Future Work

To adapt our approaches to Reinforcement Learning (RL), we would need to formulate the problem as a Markov Decision Process (MDP) and use RL algorithms to learn optimal policies [1].

For that purpose, we have two solutions:

First - We could simply run the game several times and each time the agent would reorganize its reward/punishment values based on the score it received. We would train for the two first levels (as it should be able to level up), test our new and improved values with a simple game, and watch the agent get a good score.

Second - We could change completely our agent and accomplish it following the next steps:

A. Define the state space

Determine the information that will be used as input to the RL agent. This includes the game map, the agent's position, enemy positions, power-up locations, bomb locations, and any other relevant information.

B. Define the action space

Determine the possible actions the RL agent can take. This includes moving in different directions, placing bombs, and using power-ups if available.

C. Define the reward/punishment function

Design a reward/punishment function that provides feedback to the RL agent based on the game state and actions taken. The reward/punishment function should align with the objectives of the game, such as maximizing the score, surviving, or reaching the exit. Here we can incorporate the rewards systems developed in our approaches.

D. Formulate the problem as an MDP

Specify the transition dynamics, which describe how the game state evolves in response to agent actions. This includes defining the probabilities of transitioning to different states based on the agent's chosen action.

E. Choose an RL algorithm

Select an RL algorithm that suits the problem and requirements of your game. Popular RL algorithms include Q-learning, SARSA, and Deep Q-Networks (DQN). These algorithms use the MDP formulation and interact with the environment to learn optimal policies.

F. Train the RL agent

Train the RL agent using the selected algorithm. This involves iteratively interacting with the game environment, observing the state, taking actions, receiving rewards and possible punishments, and updating the agent's policy or value functions based on the RL algorithm.

G. Evaluate and fine-tune

Evaluate the performance of the trained RL agent and iterate on the training process to improve its performance. This may involve adjusting hyper-parameters, modifying the reward/punishment function, or exploring different RL algorithms.

VI. Conclusion

Based on the overall results we got, we can say that our second approach is the best, because it is much more consistent in the long run, with many runs above 800, compared to the other approaches.

We also concluded that, even though our approach to this project is more creative, it shouldn't really be considered Reinforcement Learning, as the agent doesn't learn from its mistakes or good actions. With the improvements previously mentioned, we should be able to use our methods and have a RL agent.

References

[1] Shweta Bhatt. *Reinforcement Learning Essentials*. URL: https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292.