MSc

2.º
CICLO

FCUP
2016

U.PORTO

Bomberman as an Artificial Intelligence Platform

Manuel António da Cruz Lopes

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Bomberman as an Artificial Intelligence Platform

Manuel António da Cruz Lopes

Dissertação de Mestrado apresentada à
Faculdade de Ciências da Universidade do Porto em
Área Científica

2016

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

All corrections determined by the jury,
and only those, were incorporated.

The President of the Jury,

Porto, ______/______/_________

MSc

Manuel António da Cruz Lopes

Bomberman as an Artificial Intelligence Platform

U. PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2016

Manuel António da Cruz Lopes

Bomberman as an Artificial Intelligence Platform

U. PORTO

FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Tese submetida Faculdade de Ciências da Universidade do Porto para obteno do grau de Mestre em Ciência de Computadores

Advisors: Prof. Pedro Ribeiro

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Junho de 2016

Para os meus pais e para o meu irmão

Acknowledgments

I would like to thank my supervisor, Prof. Pedro Ribeiro, for the opportunity to work on this research topic, giving me enough freedom to experiment and learn on my own, and yet he has always been there to guide me. Also to my friends and family that gave me the moral support needed to endure this journey.

Thank you.

Abstract

Computer games have a long tradition in the *Artificial Intelligence* (AI) field. They serve the purpose of establishing challenging goals with long term positive repercussions, with researchers trying to develop AIs capable of beating the world human experts. They also provide a good amount of exposition in mainstream media, whilst also giving an engaging environment for teaching computer science concepts in general and AI methods in particular.

This thesis aims precisely to contribute to the educational gaming landscape, by providing a novel Bomberman themed platform and several associated intelligent agents. Our first major contribution is precisely the platform itself, *Bomberman as an Artificial Intelligence Platform* (BAIP). We provide a fully functional open-source, graphical, language agnostic platform, aimed towards facilitating the study and development of AI methods.

We showcase the potential of BAIP and the richness of the selected game environment by developing a plethora of AI agents using different methodologies. At an initial level we introduce a set of *basic heuristic* methods capable of providing agents with the most essential behavior primitives, equipping them for basic survival in the game environment. At a more advanced level we introduce *search-based* methods capable of some form of planning.

From a machine learning perspective we show how BAIP can be integrated and used with *Reinforcement Learning* (RL). We use a simple RL problem within the platform and implement both Q-learning and Sarsa algorithms in order to tackle it.

This thesis also provides detailed and thorough experimental results about all the developed agents and methods, using the developed platform capabilities.

To sum up, this work introduces a new AI platform, gives a strong baseline agent and demonstrates the feasibility of using it for machine learning tasks.

Resumo

Desde sempre os videojogos estiveram presentes no campo da *Inteligência Artificial* (IA). Estes têm por objetivo estabelecer desafios pertinentes que trarão repercussões positivas, enquanto que os investigadores procurarão desenvolver IAs capazes de vencer os melhores jogadores do mundo. Os videojogos são também uma boa forma deste meio se expor às pessoas fora da área e de proporcionar um ambiente de ensino para os conceitos da Ciência dos Computadores ligado às IAs.

Esta tese tem por objetivo dar uma contribuição ao meio que usa as ferramentas dos jogos como meio de ensino, a qual consiste numa plataforma, criada de raiz, baseada no jogo do Bomberman e vários agentes inteligentes. A primeira contribuição foi a plataforma, *Bomberman as an Artificial Intelligence Platform* (BAIP), uma plataforma gráfica, open-source, agnóstica à linguagem e está operacional e capaz de ajudar no estudo e criação dos métodos de IA.

Desenvolveu-se um vasto leque de agentes de IA de maneira a demonstrar as potencialidades da BAIP e do ambiente de jogo escolhido. Começamos por introduzir um conjunto de métodos que seguiam *heurísticas básicas*, estas heurísticas deram a capacidade de sobrevivência aos agentes. Posteriormente foram introduzidos os métodos *baseados em procura.*

Do ponto de vista de *machine learning* demonstrou-se como é que a BAIP consegue ser integrada e usada com *Reinforcement Learning* (RL). Para tal partiu-se de um problema de RL bastante simples e implementamos os algoritmos de Q-learning e Sarsa, de maneira a demonstrar a viabilidade da plataforma na integração do RL.

Nesta dissertação também se apresentam detalhadamente os resultados experimentais sobre todos os agentes e métodos desenvolvidos, usando para tal as capacidades da plataforma.

Resumidamente, este trabalho introduz uma nova plataforma de IA, e dá-nos também um agente que pode servir como caso base para futuros estudos e demonstra a viabilidade da plataforma para resolver tarefas ligadas ao *machine learning*

Contents

List of Tables

List of Figures

List of Algorithms

Introduction 1

Computer games play a very important role on promoting general interest in Computer Science. They have been extensively used as engaging pedagogical tools. In this thesis we introduce a game-based platform capable of providing an interesting background for a multitude of tasks. In particular we use the strategic maze-based game franchise Bomberman and we make the case for why it is a rich and very flexible game environment for learning artificial intelligence methodologies.

1.1 Motivation

Artificial Intelligence (AI) can improve the daily quality of human lives. One example of those improvements, on the verge of becoming a daily reality, are self-driving cars [Gui11]. Real-world problems such as this one are however a very difficult AI testbed. It is therefore very common that before trying to develop intelligent agents for real-world environments, we use synthetic and more controlled abstract environments. Game platforms, such as the one we created in this thesis, are therefore almost ideal to develop methods and gain knowledge, providing the basis that can afterwards be applied on more dynamic and complex problems [Lai01].

When looking at history, it is easy to see that AI has a very deep connection with games [A$^+$94]. One of its greatest accomplishments took place in 1997, when a chess-playing computer, Deep Blue, defeated the human world champion, Garry Kasparov. To achieve this feat it was necessary to combine advances in several fields, ranging from the hardware itself, to massively parallel systems and advanced refinements of tree based search techniques [CHH02]. Deep Blue's contributions are acknowledged as very significant, not only on the AI side, but also on promoting interest of the society in the area [New00]. This connection with games continues to present days and another major milestone was achieved precisely in 2016, during the work on this dissertation, when the computer program AlphaGo defeated the 18-time world Champion Lee Sedol on Go, a strategic and very complex board game deemed as very difficult challenge for

AI [SHM+16]. Again, this had a considerable impact on main stream media and some of its advances are already being considered for different applications such as clinical medicine [Zha16].

Besides this capability of introducing measurable goals (winning against the best humans), games also provide an excellent educational platform. An interested and motivated student is more likely to be a successful learner and games really excel in actively engaging students in the material being taught. Several universities have therefore introduced games as important components in the teaching process, from the very introductory programming courses teaching basic data structures [Law04] to upper level machine learning courses [Tay11]. Games have also been used for teaching very different programming paradigms, from more imperative object-oriented languages such as Java [MTR11], to more declarative ones such as the logic programming language Prolog [RFS+09]).

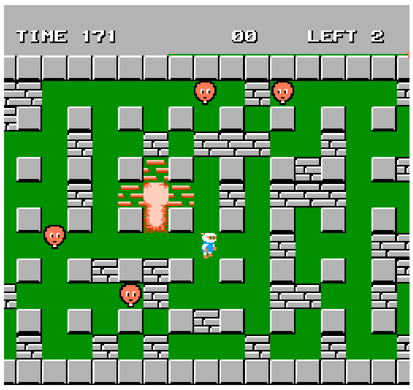
From the point of view of teaching AI, games provide highly graphical environments that can really illustrate the outcome of the methods being taught, increasing the appeal, and providing a very rich background. Examples of well known games that have been used for teaching AI are Pac-Man [DK10], Spacewars [MF07], Super Mario [Tay11] or Angry Birds [YK15]. Some more general game platforms have also been introduced, such as Botzone [ZGL+12], SEPIA [SECR13] or GAIGE [Rie15], but they are generally limited in scope to certain languages or game types, making it hard for us to use them on the desired context with all the necessary requirements, such as detailed logs of the games begin run for a more thorough analysis of the implemented AI methods.

While one must take care of the pros and cons of using games in a learning environment [Bay09], they have been proven to be a very useful teaching tool. The main purpose of this dissertation is precisely to contribute to the educational gaming landscape, by providing a novel platform centered around the Bomberman game, while also showcasing its potential. We are aware of only two other works based around *Bomberman* [WC07, BT02]. Our work differs from these two because we implemented a multilingual platform geared towards generic AI problem, as opposed to only supporting a single language and focusing on introductory C programing [WC07], or only focusing on the development of the AI agents [BT02].

1.2 Bomberman Game

Bomberman, also known as *Dynablaster*, is a strategic maze-based video game franchise originally developed by Hudson Soft with that name in 1985[1]. Some screenshots of this original version can be seen in Figure 1.1. Since then more than 80 different versions of the game have been published for different gaming platforms [2], including a 2016 official mobile version celebrating 30 years of the Bomberman franchise [3]. Since its creation the game has kept a large amount of followers, including a dedicated community of users that created a Bomberman Wiki with more than 1,400 articles [4].

(a) Title screen



(b) Example of game action

Figure 1.1: Screenshots from the original 1985 Bomberman game. Taken from MobyGames website.

The general goal of Bomberman is to complete the levels by strategically placing bombs in order to kill enemies and destroy blocks. Most versions of the games also feature a multi-player mode, where other players act as opponents and the last one standing is the winner. Independently of the game mode, the game logic basics are identical. Each room is a rectangle surrounded with a grid of indestructible blocks. The pathway can be blocked with different types of blocks in order to create routes around the screen in a maze-like fashion. A player can destroy some of the blocks in the pathway by placing bombs near it. As the bombs detonate, they will create a burst of vertical and horizontal lines of flames. The contact with the blast will destroy anything on the screen, except for the indestructible blocks. There is also a plethora of different power-ups (such as more bombs or bigger detonation range) capable of changing and enriching the gameplay experience. A more complete and detailed description of the game logic implemented by our platform can be seen in Section 3.1.

[1]`http://www.mobygames.com/game-group/bomberman-games`
[2]`http://www.uvlist.net/groups/info/bomberman`
[3]`http://www.pocketgamer.co.uk/r/iPhone/Taisen!+Bomberman/news.asp?c=68842`
[4]`http://bomberman.wikia.com/wiki/Main_Page`

The simple 2D graphics, the multiplayer aspect, the dynamic environment and the strategic reasoning all contribute to make Bomberman an attractive platform for developing and testing AI algorithms. Simple greedy behaviors must be implemented even for providing basic surviving capabilities in such a hostile gaming environment. More careful planning strategies, involving path finding and strategic bomb placement must be devised in order to construct more intelligent actions. Furthermore, the game provides plenty of situations for learning tasks, with new maps, power-ups and adversaries providing challenging opportunities for testing general game playing that does not depend on predefined rules.

1.3 Goals and Contributions

The main goal of this thesis is to develop a bomberman-based AI platform, and to create AI agents using that platform.

The first contribution is the platform itself, which we call BAIP. We are particularly interested in making the platform easy to use, agnostic in terms of programming language and rich enough to be used in many possible AI problem formulations. We have achieved a fully functional state and our platform works with any language, communicating with using standard input and output. It also boosts an appealing graphical view of the world state, detailed logs of every simulation, the possibility of customizing the maps and power-ups, and also out-of-box low level capabilities towards the development of learning agents.

The second contribution is an implementation of a set of simple heuristics capable of providing basic functionality for an agent, namely the capability to avoid explosions, to travel through the maze (by destroying blocks) and to kill other agents. We provide a detailed analysis of these behaviors, shedding light on which basic strategy is the most essential, which has more impact in the number of moves, and how one should deal with the dilemma between exploring and actively pursuing enemy agents.

The third contribution is a search-based agent capable of producing an action tree and to efficiently explore the state-space. We show a detailed analysis of its competitiveness against both the simple basic agents and to different variations of the planning agent, with a focus on discovering the optimal plan size.

The fourth contribution is a simple formulation of a Reinforcement Learning (RL) problem within the platform, coupled with an implementation of both Sarsa and Q-learning algorithms that are able to tackle the problem. The obtained results conform to what was expected, showing both the validity and feasibility of our approach and showcasing its ease of use.

1.4 Thesis Outline

This thesis is structured in five major chapters. A brief description for each one is provided below.

Chapter 1 - Introduction . An overall view of the problem being studied and the motivation behind it, as well as a summary of our goals and contributions.

Chapter 2 - Background . An explanation of how Reinforcement Learning (RL) works and how we can formulate problems using it, along with a more detailed explanations on some specific RL methods such as Q-Learning and Sarsa. We also include a brief description of some related work that inspired this research, namely the Arcade Learning Environment (ALE). This chapter is a review to help readers that are not experienced in these subfields of the AI area.

Chapter 3 - BAIP Platform. A description of the developed AI platform, including an explanation of its key features and possible applications.

Chapter 4 - Agents. A detailed view on all the agents developed using the platform. First we show basic primitive behaviors that lead to a baseline agent. Next we present the search-based agents capable of planning. We also introduce an RL problem and how we use our platform to test RL methods on it. We conclude the chapter with a discussion of the obtained experimental results.

Chapter 5 - Conclusion and Future Work. Concludes the thesis with the progresses achieved and gives directions for future work.

Background 2

In this chapter we give an overview of a number of fundamental concepts that are required for the rest of this thesis. In particular, we will cover reinforcement learning and its associated methods. We also look at other state-of-the-art training platforms.

2.1 Search Strategies

Maze-themed games such as *Bomberman* can be seen as path-finding challenges. So given a graph $G = (V, E)$ and a distinguished source vertex s we need a strategy to systematically explore the edges of G to find the path reachable from s that best suits our challenge. So to incrementally explore paths from the start nodes we need to maintain a *frontier* of paths from the start node that have been explored. The *frontier* contains all of the paths that could form initial segments of paths from a start node to a goal node [LK10].

Different strategies are obtained by modifying how the selection of paths is implemented. We can group these strategies in two groups:

- *Uninformed Search Strategies*, do not take into account the location of the goal. Intuitively, they ignore where they are going until they find a goal and report success.

- *Informed Search Strategies*, search for information about which nodes seem the most promising, they achieve that by using a heuristic function $h(n)$ which takes a node n and returns a non-negative real number that is an estimate of the path cost from node n to a goal node.

Next we give an overview of some the *Search Strategies* that can be implemented using our platform. From the *Uninformed* we study the *Breadth-First Search* and *Depth-First Search* methods. Finally from the *Informed* we study the *A^* Search*.

2.1.1 Breadth-First Search (BFS)

Breadth-first search produces a tree with root s that contains all reachable vertices. For any vertex ν reachable from s , the simple path in the breadth-first tree from s to ν corresponds to a shortest path from s to ν in G, that is, a path containing the smallest number of edges.

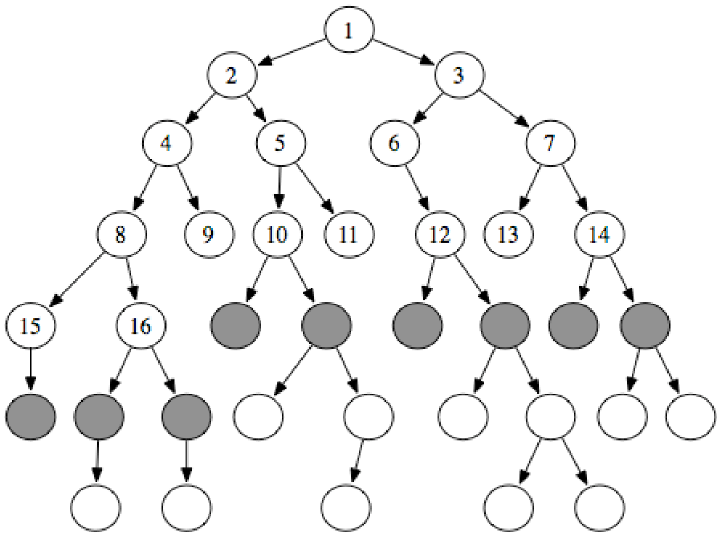


Figure 2.1: The order in which nodes are expanded in breadth-first search. Image taken from [LK10]

The *frontier* can be implemented as a FIFO (first-in, first-out) queue, following this implementation the path that is selected from the frontier is the one that was added earliest.

Although this method is guaranteed to find a solution, if one exists and will find a solution with the fewest arcs, its time and space complexities are exponential in the number of arcs of the path to a goal with the fewest arcs. So it is appropriate to use BFS when we have enough space, or the solution contains few arcs, or few solutions may exist or infinite paths may exists [LK10].

2.1.2 Depth-First Search (DFS)

Depth-first search explores edges out of the most recently discovered vertex ν that still has unexplored edges leaving it. Once all of νs edges have been explored, the search backtracks to explore edges leaving the vertex from which ν was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex [LK10]. Some paths may be infinite when the graph has cycles or infinitely many nodes, in which case a depth-first search may never stop.

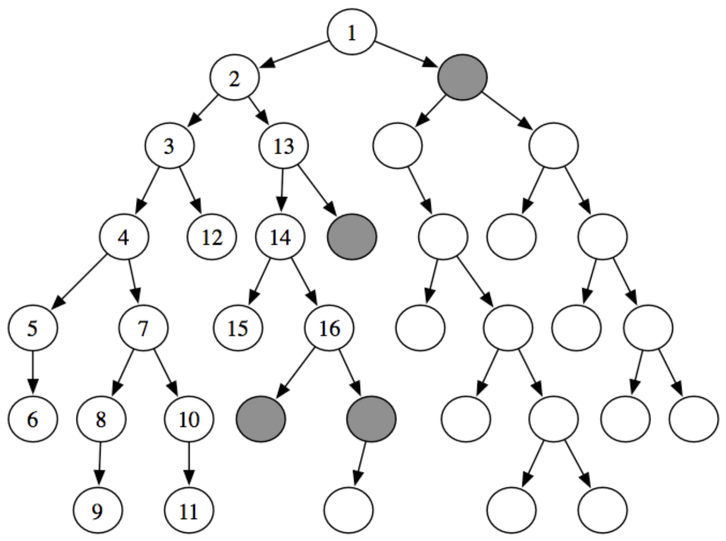


Figure 2.2: The order in which nodes are expanded in depth-first search. Image taken from [LK10]

The *frontier* acts like a LIFO (last-in first-out stack). The elements are added to the stack one at a time. The one selected and taken off the frontier at any time is the last element that was added.

This method is not guaranteed to find a solution and is sensitive to the order in which the neighbors are added to the *frontier*. The efficiency of the algorithm is sensitive to this ordering. So it is appropriate to use DFS when space is restricted, or many solutions exist, or the order of the neighbors of a node are added to the stack can be tuned so that solutions are found on the first try [LK10].

2.1.3 A^* Search

A^* is the most popular choice for path-finding, because it can be used in a wide range of contexts. It considers both path cost and heuristic information in its selection of which path to expand so it avoids expanding paths that are already expensive, but expands most promising paths first.

In the standard terminology used when talking about A^*, $g(n)$ represents the exact cost of the path from the starting point to any vertex n, and $h(n)$ represents the heuristic estimated cost from vertex n to the goal. A^* balances the two as it moves from the starting point to the goal. Each time it examines the vertex n that has the lowest $f(n) = g(n) + h(n)$.

It can be implemented by treating the frontier as a priority queue ordered by $f(n)$.

This method may be exponential in both space and time, but they are guaranteed to find a solution if one exists. And it is guaranteed to find the least-cost solution as the first solution found [LK10].

2.2 Planning

Planning the two major areas of Al: search and logic. A planner can be seen either as a program that searches for a solution or as one that (constructively) proves the existence of a solution.*Planning* can be described as the process of finding a sequence of actions that can get our agent from the start state to the goal state.

Ordinary problem-solving agent using standard search algorithms use atomic representations of states, and that might not be enough to represent a complex object. In order to assess this problem, researchers settled on a *factored representation*, in which a state of the world is represented by a collection of variables [RN03].

Everything from the environment that is available to the agent is known as the *state*, defined by $s \in S$ where the S is the set of possible steps, and the agent must be able to take a set of actions, defined by $a \in A$, that affect the state. These representations make possible the derivation of effective heuristics and the development of powerful and flexible algorithms for solving problems [RN03].

Planning systems are problem-solving algorithms that operate on explicit propositional representations of states and actions. The main difference between *Planning agents* and *Problem-solving agents* is how they represent the states. Which can bring some difficulties to the latter, such as becoming overwhelmed by irrelevant actions, the difficulty of finding a good *heuristic function* and not taking advantage of *problem decomposition*. The design of many planning systems is based on the assumption that most real-world problems are *nearly decomposable*, that is the planner can work on subgoals independently [RN03].

There are different approaches to *Planning* and each one has its own advantages. We can see some of them next:

- *State-space search*, that can operate in the forward direction (progression) or the backward direction (regression);
- *Partial-order planning (POP) algorithms*, that explore the space of plans without committing to a totally ordered sequence of actions. They work back from the goal, adding actions to the plan to achieve each subgoal;

- *Planning graph algorithms*, they process the planning graph in order to extract a plan. A *planning graph* can be constructed incrementally, starting from the initial state. Each layer contains a superset of all the actions that could occur at that time step.

But while these *planning* algorithms assume complete and correct information and deterministic, fully observable environments. Many domains violate this assumption, thus while planning and acting in the real world we need to take into account that many actions can generate and consume *resources* and that time is one of the most important *resources.* To address this we need to approach it differently, such as:

- *Contingent planning* allow the agent to sense the world during execution to decide what branch of the plan to follow;
- *Online planning* agent that monitors its execution and repairs as needed to recover from unexpected situations, which can be due to nondeterministic actions, exogenous events, or incorrect models of the environment.

Using these methods we can plan even when the environment is *nondeterministic.* Following *Online planning*, in the next section 2.3 we can see how an agent can learn to behave from past successes and failures [RN03].

2.3 Reinforcement Learning (RL)

Reinforcement Learning is a learning process in which interactions with the *environment* produce stimulus, and agents learn from their past actions, as a *trial-and-error* approach. In this case the *environment* S is the emulator created, and the actions will belong to the set of legal game actions $A = \{1, ..., K\}$.

Positive stimulus might be seen as rewards, so the agent must try to maximize them and must learn which actions produce them, without any help. Sometimes, actions might affect all subsequent rewards. These two characteristics, *trial-and-error search* and *delayed reward*, are the two most important distinguishing features of reinforcement learning [SB98].

Trial and error learning leads to the *exploration vs exploitation* dilemma, where an agent must choose its actions based either on past experiences or to explore new options. To obtain better results, an agent must *exploit* the actions that he has already tried and that he knows that will produce better rewards. But to find the actions that produce better results the agent needs to *explore*, so it needs to occasionally select a random move from among all the set of legal actions, this randomly selection might lead us to experience new states. Neither of these approaches should be chosen exclusively, else it may lead to poor results.

So we can say that RL agents have specific goals and interact with their environment, through actions that influence it, to achieve those goals. But this goal-seeking problem has its own sub-problems that derive from the environment, we can divide them in problems that involve *planning* and problems that involve *supervised learning*. In problems that reinforcement learning involves planning, it has to address the interplay between planning and real-time action selection, as well as the question of how environmental models are acquired and improved. When it involves supervised learning, it does so for specific reasons that determine which capabilities are critical and which are not [SB98].

To understand the *reinforcement learning* system, we start by learning what constitutes this system, and after we observe what happens at each time step.

In our problem we consider the reinforcement learning system to have: *i*) an *Agent* that is the learner and decision-maker *ii*) an *Environment* that is everything outside the agent that interacts with it *iii*) a *Policy* that defines the learning agent's way of behaving at a given time. *iv*) a *Reward Function* that defines the goal in a reinforcement learning problem, the reward and indicates the intrinsic desirability of a state. *v*) a *Value function* that specifies what is good in the long run. *vi*) a *Model of the environment* that mimics the behavior of the environment, used to predict the resultant next state and next reward, this is usually used for planning).

As we seen before let S be the set of possible steps, and the agent must be able to take actions that affect the state. From each action a_{t-1} taken the agent receives a corresponding *reward* signal r_t. The agent's goal, broadly, is guided by those *reward* signals, and its aim is to find a *policy* r^* that maximizes the sum of all rewards.

At each time step t the agent has access to the *state* s_t and chooses an action a_t based on that state, $a_t \in A(s_t)$ where $A(s_t)$ is the set of actions available in s_t. In the next time step s_{t+1}, the agent receives, usually, a numeric *reward* signal, defined by $R_{t+1} \in \mathbb{R}$. We can observe in Figure 2.3 a diagram that represents the interaction between the agent and the environment.

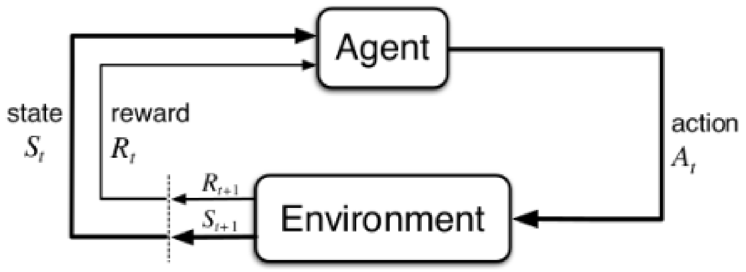


Figure 2.3: The agent-environment interaction in reinforcement learning. Image taken from [SB98]

In our *reinforcement learning* task the agent-environment interaction breaks down into a sequence of separate episodes. An episode starts when the player is able to act and finishes when the game ends or if we reach a predefined maximum number of frames per episode. Each episode has a finite sequence of time steps and the state representation at time t is s_t. Usually it is assumed that the states have the*Markov property*, where each state only affects it's next state, s_{t+1}, or that s_{t+1} only depends on the representation of the current state and action.

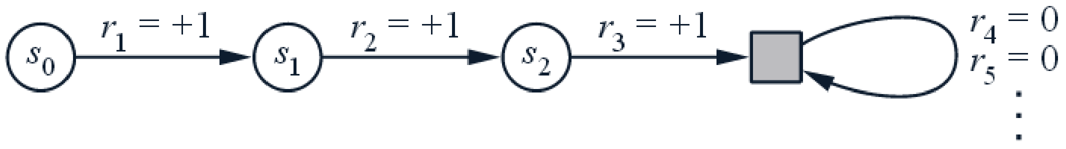


Figure 2.4: The agent state transition diagram. Image taken from [SB98]

In figure 2.4 we can observe a general representation of the state transition over an episode, where the solid square is an absorbing state that corresponds to the end of an episode. This representation assumes that in each time step, t , the agent observes the reward R_t that is originated in the previous time step s_{t-1},and using that information the agent takes an action a_t, while following a *policy*, π, moving to the next state, s_{t+1}.

As what was said before, the agent's goal is to find a *policy* that maximizes the sum of all rewards, so our *expected return* for an episode is the sum of all the states' rewards r observed in each time step t. Let's define that sum as R_t, and let T be the final time step of an episode, so

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T \tag{2.1}$$

Where, $t = 0, 1, 2, 3, \ldots$ is a sequence of discrete time steps, because this only makes sense in an environment where there is a final step and where each iteration breaks naturally into subsequences, also know as *episodes*.

So the agent tries to select actions so that the sum of rewards over the future is maximized, this leads to an agent that selects the maximum immediate reward. This kind of behavior, most of the times, will not generate the maximum overall sum of rewards, in order to solve this problem it was introduced the concept of *discounting*. After introducing this concept in our *expected return*, it becomes:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Also known as *discounted return*, where γ is the *discount rate*, $0 \leq \gamma \leq 1$, this parameter determines how much is worth a reward that will be received k time steps in the future. It is an important concept that allows the agent to be capable of determining the present value of future rewards, and through this parameter we can manipulate the agent's far sight, managing the importance that the agent gives to future rewards.

When an environment satisfies the *Markov property* it is called *Markov decision process* (MDP), in our case because there is a final episode the MDP is finite, and a finite MDP is constituted by its state and action sets and by the one-step dynamics of the environment. Where given any state s and action a, the probability of each possible next state, s' is defined by:

$$p(s'|s, a) = \mathbf{Pr}\left\{S_{t+1} = s' | S_t = s, A_t = a\right\} \tag{2.2}$$

Where P determines how the environment state, s, changes with regard to the agent actions, a. Based on the state, action and any next state we can calculate the expected value of the next reward, $r(s, a, s')$, defined by:

$$r(s, a, s') = \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s']. \tag{2.3}$$

With these *transition probabilities* we can derive *value functions*, used to evaluate the benefit of taking a certain action at a given state, following particular policies. A *policy*, π, is a mapping from each state, $s \in S$, and action, $a \in A$, to the probability $\pi(a|s)$ of selecting a possible action a when in state s. [SB98]

Value functions assign to each state, or state-action pair, the expected return from that state, or state-action pair, given that the agent uses the policy. So we have *state-value functions* and *action-value functions.*

- *state-value function*, $v_\pi(s)$, is the expected return of starting in s and following π, this function can be estimated by averaging the rewards encountered in every state when following a policy π. More formally:

$$v_\pi(s) = \mathbb{E}[R_t|S_t = s] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s]. \tag{2.4}$$

- *action-value function*, $q_\pi(s, a)$, is the expected return of starting in s and taking action a, and subsequently following π, this function can be estimated by keeping an average for each action taken in a state. More formally:

$$q_\pi(s, a) = \mathbb{E}[R_t|S_t = s, A_t = a] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a]. \tag{2.5}$$

Because of the numerous states it is unpractical to keep separate averages for each state individually. Instead, the agent would have to maintain $v_\pi(s)$ and $q_\pi(s, a)$ as parameterized functions and adjust the parameters to better match the observed returns. Both types of *value functions* follow a *policy* π, so to find an *optimal value function* our agent needs to follow an *optimal policy*, there may be more than one *optimal policy* and they will be denoted as π^*. For a *policy*, π, to be better than or equal to another *policy*, π', its expected return needs to be greater or equal to the expected return of, π', for every state. So an *optimal policy* is better than or equal to all other policies, although there may be more than one that they share the same *optimal state-value functions*, denoted as v^* and defined as

$$v^*(s) = \max_x v_\pi(s), \quad \forall s, s \in S \tag{2.6}$$

They also share the same *optimal action-value functions*, denoted as q^* and defined as

$$q^*(s,a) = \max_{x} q_\pi(s,a), \quad \forall s \forall a, s \in S, a \in A \tag{2.7}$$

For the state-action pair (s,a), this function gives the expected return for taking action a in state s and thereafter following an optimal policy, thus, we can write q^* in terms of v^* as follows:

$$q^*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma\, v^*(S_{t+1}) | S_t = s, A_t = a\right]. \tag{2.8}$$

All *value functions* obey the fundamental property that expresses the relationship between the value of a state and the values of its successor states, this property is expressed as the *Bellman equation.* This equation averages over all the possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way. [SB98]

$$\begin{aligned} q^*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q^*(S_{t+1}, a'), S_t = s, A_t = a\right] \\ &= \textstyle\sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma\, q^*(s',a')\right]. \end{aligned} \tag{2.9}$$

The *optimal value functions* also satisfy the *Bellman equation* for state values, but also follows the *Bellman optimal equation*, defined in (2.9), that says that the value of a state under an *optimal policy* must be equal to the expected return for the best action. For finite MDPs this equation has a unique solution independent of the *policy.* Although it is called *Bellman optimal equation* it is actually a system of equations, one for each state. If we know the dynamics of the environment, $p(s'|s,a)$ and $r(s,a,s')$, then the system of equations for v^* can be solved, but the overhead of looking ahead at all possibilities, computing their probabilities of occurrence and their desirabilities in terms of expected reward is impractical due to the large state-space.

In reinforcement learning one has to settle for approximate solutions. There are methods that estimate the optimal value function by interacting with the environment and looking at samples of the state transitions and the received rewards. [Nad10] So the basic idea behind many reinforcement learning algorithms is to estimate the *action-value function*, by using the *Bellman equation* as an iterative update.

There are three different classes of methods for solving the *Reinforcement Learning* problem, each has its strengths and weaknesses:

- *Dynamic Programming (DP)*, if given a perfect model of the environment as a MDP these methods can be used to compute optimal policies, but the assumption of a perfect model and the great computational expense make this class of methods limited and not practical.
- *Monte Carlo methods*, although it does not assume complete knowledge of the environment's dynamics it can still attain optimal behavior, it requires only sample sequences of states, actions, and rewards from on-line or simulated interaction with an environment.
- *Temporal-difference learning*, is a combination of the previous ideas, as they can learn directly from raw experience without a model of the environment's dynamics.

It is desirable to combine these ideas and methods in many ways and apply them simultaneously.

2.3.1 Temporal-difference (TD) methods

TD learning is an unsupervised technique in which the learning agent learns to predict the expected value of a variable occurring at the end of a sequence of states. RL extends this technique by allowing the learned state-values to guide actions which subsequently change the environment state. These methods can learn from raw experience without a model of the environment's dynamics and update estimates based in part on other learned estimates, without waiting for a final outcome.

Given some experience following a policy π, both TD and Monte Carlo methods update their estimate V of V^π. If a nonterminal state s_t is visited at time t, then both methods update their estimate $V(s_t)$ based on what happens after that visit. But whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(s_t)$, TD methods need to wait only until the next time step. At time $t+1$ they immediately form a target and make a useful update using the observed reward r_{t+1} and the estimate $V(s_{t+1})$. The simplest TD method, known as TD(0), is

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right] \tag{2.10}$$

So TD methods are based on the n next reward, using the value of the state n steps later as a proxy for the remaining rewards. One kind of intermediate method, then, would perform a backup based on an intermediate number of rewards: more than one, but less than all of them until termination [SB98].

2.3.1.1 Q-Learning

The *Q-learning* algorithm is an off-policy method and follows a model-free approach and a *temporal difference.* It is a frequently used when the environment model is unknown. So the objective of a *Q learner* is to determine π^* without initially knowing the reward and transition dynamics of the environment, $p(s'|s,a)$ and $r(s,a,s')$.

There are different types of *mode of control of behavior*, each are based on the internal representation of knowledge, they can be divided on whether or not the agent can predict *state-transitions* and *rewards* - that is, *model-based* if the agent can predict, *state-transitions* and *rewards*, and *model-free* if the agent only considers the present state. *Learning is a process of improving a policy and value function*[WD]. There are two approaches to ensure the improvement of a policy, resulting in what we call on-policy methods and off-policy methods. *On-policy* methods attempt to evaluate or improve the policy that is used to make decisions, but in *Off-policy* methods these two functions are separated. The policy used to generate behavior, called the *behavior policy*, may in fact be unrelated to the policy that is evaluated and improved, called the *estimation policy.*

Let Q be the learned *action-value function*, as a direct approximation of the *optimal action-value functions*, q^*. In order for the agent to improve Q through its experience it uses the *one-step Q-learning* method. In this method the agent's experience consists of a sequence of distinct episodes. In n^{th} episode the agent adjusts its Q_{n-1} values using a learning rate α_n, according to:

$$Q_n(S_t,A_t) = Q_{n-1}(S_t,A_t) + \alpha_n \left[R_n + \gamma \underset{a}{max} Q_{n-1}(S_{t+1},a) - Q_{n-1}(S_t,A_t)\right]. \quad (2.11)$$

where the learning rate α, $0 \leq \alpha \leq 1$, is usually a small constant, this value is fundamental as it is responsible for the rate at which new information is combined with the existing knowledge about the value of the state. The learning speed, thus the learning time, varies accordingly to α. Where if α is close to 0 the learning times are long, else if α is close to 1 it causes oscillations in the *value function* [Tok10] [GP06]. The starting Q values, $Q_0(S_t,a)$, for all states and actions are given.

While following the function (2.11) the agent still follows a *policy*, however, for Q to converge to q^*, $Q_n \to q^*$ as $i \to \infty$, all the *state-action* pairs need to continue to be updated. As was presented in [WD], *Q-learning* converges with probability 1 under reasonable conditions on the learning rates and the *Markovian environment.*

The key issue with this approach is *generalization*, in our task most states encountered will never have been experienced exactly before, and for each sequence we need to estimate separately the *action-value function*, $q_\pi(s)$. In order to learn in these conditions we need to generalize from previously experienced states to ones that have never been seen. Thus we need a *function approximation* to take examples from the *action-value function*, and attempt to generalize from it to construct an approximation of the entire function. This approximation technique allows the policies to be represented in a low-dimensional form, decreasing the number of parameters that need to be learned or stored.

There are two types of function approximations, *i) linear function approximations* that are commonly used, usually states are not represented as a table but are mapped to feature vectors with fewer components than the number of states. Until recently this method was the focus of the majority of the work in reinforcement learning, as it was believed that it gave better convergence guarantees. *ii) non-linear function approximations* were not commonly used, although the success behind TD-gammon [Tes95]. It was shown that the combination of model-free reinforcement learning algorithms such as Q-learning with non-linear function approximators, such as a neural network, caused the *Q-network* to diverge, but these issues were resolved through *gradient temporal-difference* methods. Recently the success story behind *non-linear function approximations* is due to the *deep Q-network (DQN) agent*, that combines deep neural networks with reinforcement learning. [MKS+15].

2.3.1.2 Sarsa

We now introduce the *Sarsa* algorithm, an on-policy method that stochastically approximates the optimal value function based on the state transition and the reward samples received from online interaction with the environment.

The first step is to learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $Q^\pi(s,a)$ for the current behavior policy π and for all states s and actions a. This can be done using essentially the same TD method described above for learning v^*.

$$Q_n(S_t, A_t) = Q_{n-1}(S_t, A_t) + \alpha_n \left[R_n + \gamma Q_{n-1}(S_{t+1}, a) - Q_{n-1}(S_t, A_t)\right]. \tag{2.12}$$

This update is done after every transition from a nonterminal state s_t. If s_{t+1} is terminal, then $Q(s_{t+1}, a_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events, $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$, that make up a transition from one state-action pair to the next. This quintuple gives rise to the name *Sarsa* for the algorithm [SB98].

2.3.1.3 Eligibility Traces

In order to obtain a more general temporal-difference (TD) method, that may learn more efficiently, we can combine it with *Eligibility Traces.* When applied to TD methods they generate methods that range between MC methods and one-step TD methods, all intermediate methods are often better than either extreme method. We can see an *Eligibility Traces* as a temporary record of the occurrence of an event. That is, if a TD error occurs on a given step it is assigned to the previous steps as determined by the eligibility trace. λ determines the extent to which the prediction values for previous observations are updated by errors occurring on the current step.

Let *Sarsa(λ)* be the eligibility trace version of *Sarsa.* We need a trace for each state-action pair, let $e_n(S_t, A_t)$ denote the trace for state-action pair (S_t, A_t).

$$Q_n(S_t, A_t) = Q_{n-1}(S_t, A_t) + \alpha_n e_n \left[R_n + \gamma Q_{n-1}(S_{t+1}, a) - Q_{n-1}(S_t, A_t)\right]. \tag{2.13}$$

where

$$e_n(S_t, A_t) = \begin{cases} \gamma\lambda e_{n-1}(S_t, A_t) + 1 & if \quad S_n = S_t \;\; and \;\; A_n = A_t; \\ \gamma\lambda e_{n-1}(S_t, A_t) & \qquad otherwise. \end{cases} \tag{2.14}$$

The states that have recently been visited can be maintained through traces, where temporarily is defined by $\gamma\lambda$, this can indicate the degree to which each state is eligible for undergoing learning changes, should a reinforcing event occur.

One method that combines eligibility traces and *Q-learning* is the *Watkins's Q(λ).* The main difference from other methods is that, its lookahead stops at the first exploratory action, or at episode's end if there are no exploratory actions before that. *Eligibility traces* are used just as in *Sarsa(λ)*, except that they are set to zero whenever an exploratory (nongreedy) action is taken. The trace update is best thought of as occurring in two steps. First, the traces for all state-action pairs are either decayed by $\gamma\lambda$ or, if an exploratory action was taken, set to 0. Second, the trace corresponding to the current state and action is incremented by 1 [SB98].

2.4 Related Work and State of the Art

In this section we present some related work that influenced and inspired our developed plaftorm, namely the Arcade Learning Environment (ALE) [BNVB13].

ALE is a software framework for interfacing with emulated Atari 2600 game environments, spanning a diverse range of genres such as shooters, beat'em ups, puzzle, sports, and action-adventure games. The user receives a a single game frame (in a 2D array of 7-bit pixels, 160 pixels wide by 210 pixels high) and/or RAM from the Atari 2600 and sends joystick motions (the action space consists of the 18 discrete actions). It also provides a game-handling layer which transforms each game into a standard reinforcement learning problem by identifying the accumulated score and whether the game has ended. The reward at each time-step is defined on a game by game basis, typically by taking the difference in score or points between frames. An episode begins on the first frame after a reset command is issued, and terminates when the game ends. The game-handling layer also offers the ability to end the episode after a predefined number of frames.

Tools such as ALE aid in the study of the previously described fields such as RL, and they also help in the growth of the AI field. One of the most significant recent developments was that of *Deep Reinforcement Learning*, which we now describe in more detail.

Deep Reinforcement Learning can be viewed as the application of *Deep Learning* to *Reinforcement Learning.* The objective of *Deep Supervised Learning* methods is to find a powerful synaptic modification rule that will construct an arbitrarily connected neural network to learn an internal structure, with multiple levels of abstraction, and allow a system to learn complex functions mapping the input to the output directly from data, without depending on human-crafted features [Ben09].

Deep architectures are composed of multiple levels of non-linear operations, such as neural nets with many hidden layers. The learning methods must decide when and what hidden units should be active. Therefore, a deep representation can be viewed as a function composed by several individual functions. There are three common architectures used in Deep Learning: *feedforward neural networks* or *multilayer perceptrons* (MLPs), *recurrent neural networks* and *convolutional neural networks.* One of the *DQN agents* core components is a deep convolutional network architecture. For a more detail explanation on the types of architecture it's advisable to consult [IYA16]. So to combine it with RL in order to create an agent that is capable of learning, we need a learning algorithm, and DQN agent uses the *stochastic gradient descent (SGD)* [Bot10, LBH15, RHW88].

There are three ways to apply Deep Learning to RL: we can use a deep network to represent the value function, the policy or the transition model. In each case we need to try to combine them with stochastic gradient descent and optimizing it by defining an appropriate loss function. The current state of the art is the DQN agent and *Deep Q-Learning* [MKS+15, MKS+13].

2.5 Summary

In section **2.1 Search Strategies**, we gave an overview on *Search Strategies*, in order to understand how an agent is can explore in order to select the best possible actions. It is important to retain from this section the notion of the *Uninformed Search Strategies*, the *Informed Search Strategies* and their difference. We also detailed some algorithmic methods, namely Breadth-First Search, Depth-First Search and A^* Search.

In section **2.2 Planning**, we gave an overview on *Planning Strategies*, in order to understand the problem of *planning* in *deterministic*, fully observable, static environments. This chapter also extends classic planning to cover *nondeterministic* environments.

In section **2.3 Reinforcement Learning (RL)**, we gave an overview on *Reinforcement Learning*, in order to understand how everything is setup and how an agent is able to learn in order to select the best possible actions, maximizing the sum of rewards over the future. It is important to retain from this section the notion of the *Markov decision process*, the *value functions* and how these functions satisfy *Bellman equation.* We also detailed some algorithmic methods, namely Q-Learning and SARSA.

In section **2.4 Related Work and State of the Art**, we gave an overview of a similar platform that we used as inspiration in the creation of our work, and also presented the state of the art *Deep Reinforcement Learning* method.

BAIP Platform 3

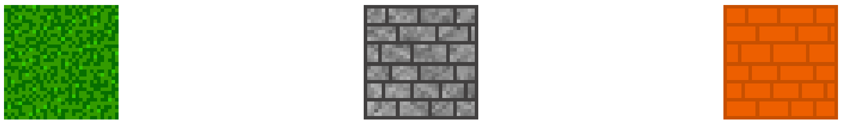
In this chapter we present the platform that was developed during the thesis: *Bomberman as an Artificial Intelligence Platform* (BAIP)[1].

BAIP suports two main modes: (1) *multi-player mode*, where the user can test his novel agents against other agents and the last one standing will be the winner; (2) *learning mode* directed to Reinforcement Learning (RL) agents where a user can define a score position and test the behavior and learning habitabilities of his agent. BAIP also offers the possibility to test human players against the developed agents. This is achieved by creating a agent that receives its actions from the keyboard. This is one of the features that the platform has to offer.

3.1 Game Logic

BAIP is an adaptation of the *Bomberman* games, and therefore it follows the same concept of strategy and maze-based theme of the original series. The general goal is to place bombs in order to kill enemies, with the winner being the last standing agent. An agent can perform the following actions: *up, down, left, right, place bomb* and *wait*.

The maps are customizable, as explained in Section 3.3.3, and they all follow the same principles. Each map is a grid surrounded with indestructible orange blocks (or walls) and destructible stone gray blocks that block the pathways. As in the original series the agent can destroy the blocks in the pathway by placing bombs near it. The game tiles that constitute the basic building blocks of every map are:



(a) Grass (pathway) **(b)** Destructible Wall **(c)** Indestructible Wall

Figure 3.1: Game map tiles

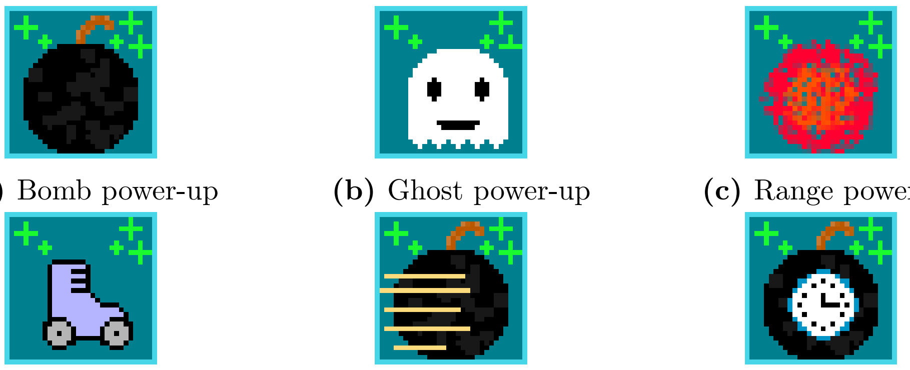
[1]`https://github.com/LopesManuel/BombermanAI-Platform`

At the game's start each bomb takes, by default, 3 game updates to explode. The number of updates it takes to explode can be altered before the game start. Each explosion resulting from the bombs will remain in the respective tiles, by default, during 5 turns, this value can be altered before the simulation. After the destruction of a block a power-up might spawn at that block's location. Power-ups are objects that instantly grant extra abilities to the character. Using these blocks we implemented the following game logic:

- When a bomb explodes near a wall, that wall will be transformed to a grass tile.
- Power-ups are not destroyed by bombs' explosions.
- When a explosion encounters any type of wall, stops at that position.
- The agents can only move to positions where there is a tile of grass
- If there is an agent in the same position of an explosion tile, this agent dies
- Agents place bombs at their current position
- There can only be one planted bomb per cell
- More than one agent can be at the same position

BAIP supports a multitude of power-ups. As seen in Figure 3.2, the power-ups implemented were:

- *Bomb power-up*, increases the max number of bombs an agent can plant at the same time.
- *Range power-up*, increases the max number of cells that the explosion gets to.
- *Ghost power-up*, gives the permanent ability of walking through stone walls.
- *Speed power-up*, diminusih the number of updates a bomb takes to explode an agent.
- *Slide power-up*, gives the ability to push the bombs to the nearest obstacle.
- *Switch power-up*, gives the ability of blowing up the agent's bomb at any time.



(a) Bomb power-up **(b)** Ghost power-up **(c)** Range power-up **(d)** Speed power-up **(e)** Slide power-up **(f)** Switch power-up

Figure 3.2: Game icons of the power-ups

In order for an agent to catch a power-up it just needs to move to the cell where the power-up is. Power-ups are not affected by explosions, making them indestructible, but they also do not stop explosions meaning that an agent cannot hide from explosions behind them. In Chapter 4 we study how the addition of power-ups changes the performance of an agent. From the power-ups presented we only study the addition of the *Bomb power-up* and the *Range power-up*.

Every simulation done in the platform can be seen as a standard RL problem. To achieve this, the user just needs to keep an accumulated scoring function. Each state consists of a game frame, represented in a 2D array of characters, with the dimensions of this array varying according to the map size. The action space consists of 6 discrete movements [*up, down, left, right, place bomb, wait*], with the agents moving one grid cell per turn.

3.2 Architecture

The system was originally developed for Unix systems using the Simple DirectMedia Layer (SDL). It allows the user to interface with BAIP using the keyboard, sending information to the screen, and simulating the game. When in *display mode*, in order for the simulation to be perceptible for the human eye, the agents are slowed down to 5 actions per second, making it possible for a user to play using the manual controls against his own AI agent. There is also the option of not showing the simulation on screen, making the agent as fast as they can possibly be.

BAIP can be used with any type of artificial intelligence, since it passes all the information to the agent. If a user wants to create an RL agent, he needs to give a reward at each time-step based on the information received from the platform, and then store the score or points between frames. An episode begins on the first frame after the start command, and terminates when the game ends. A game ends when only one agent is alive or if the simulation has run more than the predefined number of maximum states allowed per simulation. This limit on the maximum number of frames was set to prevent simulations to run forever.

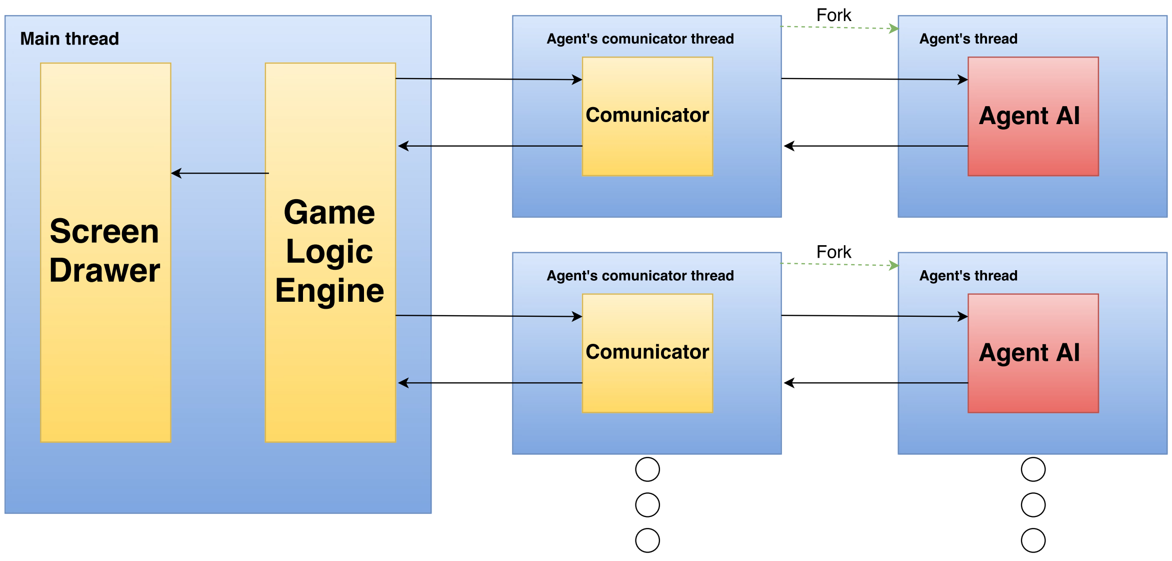


Figure 3.3: Platform architecture design

The platform is based in the overall structure presented in Figure 3.3, where we see that its structure is divided in three major components: the *game logic*, the *screen drawer* and the *agents communicator*. In the rest of this section we will be explaining how these three components come together.

From a new user perspective, the only thing one needs to worry about is the programing of it's agent's AI. Let's start by understanding the game cycle. After all the initializations are done, BAIP follows the flow described in Algorithm 3.1.

It is important to notice that the server does not wait for the agents to respond in order to update the *game logic*, but the changes to the game map are done in a synchronous fashion, following a first come first served policy. These updates are done at regular intervals, the time for these intervals is adjustable, by default it is 1 second. Hence, the agents need to respond in that time frame else their action will not be considered in that update.

Algorithm 3.1 Pseudo-code representing an overview of the high-level logic of a game cycle (details omitted included features such as logging games and RL mode)

```
procedure MAIN
    Parse commands
    Start up SDL and create window
    Load map and all images
    Create a thread for each AI agent
    while game is not over do
        Look for an event
        Update map
        Update agents
        Update screen
        if all agents are dead or there is only one alive then
            game is over
        Send updates to all the agents
        Draw everything to screen
```

For each playing agent, BAIP will create a separate thread to run the *agents communicator* and handle all the communications between the agent and the main thread. The game only starts after all the connections between the platform and the agents are established.

All the communications are in the format of ascii readable strings, we can see an example in Figure 3.4, and follow the communication protocol described at . The information the server sends to the clients includes the updated game map and the complete information about all agents, including active power-ups and life state. In response the agents only need to send a character with the number of the pretended movement. We can observe these communications in Figure 3.5.

```
C 20 15 2 0
M *********************x0++++-+++++-++000**0*+*+*+*++*+*+*-*0**++-
+++++++++++0++-**0*+*+*+*-+*+*+*+*0**+++++00++++++0+-++**+*+*+*+*+
+*+*+*+*0**+++-+++++++++++++-**+*+*+*+*+0*+*+*+*0**+++++++++0++-+0
0-+**-*0*+*+*++*+*+*+*0**+++-+++++++-+++++0**0*+*+*+*++*+*+*+*0**0
0++++-++++-+++00x*********************
P 1 1 2 1 5 1 18 13 2 1 5 2 2 13 2 1 5 3 18 1 2 1 5 4
```

Figure 3.4: Server to client communication example

Every communication described here follows the communication template presented in Algorithm 3.2, we can see the algorithm applied in C++ at communicationcode.

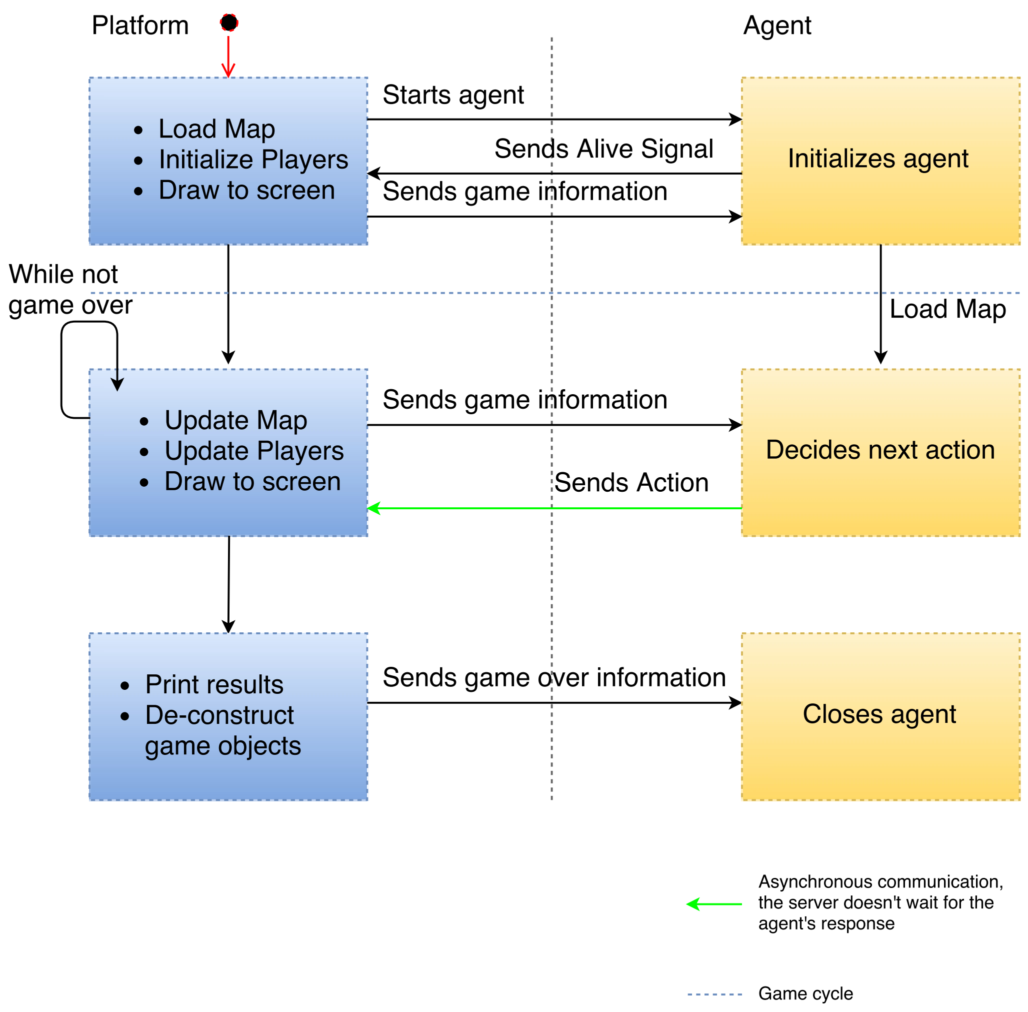


Figure 3.5: Platform communication architecture design

3.3 Key Features

In the following subsections we will present the most relevant BAIP features and we explain how to use each one. Each feature was conceived in order to facilitate the development of new AI agents.

3.3.1 Multi-language Agent

We consider that the ability to communicate with different programs written in different programming languages is an important feature, because our aim is not to teach a certain programming language, but to teach and test AI algorithms and heuristics.

BAIP communicates with the agents through standard output and input, and follows a communication protocol based on the grammar seen in Figure Grammar. Given this, independently of its programming language, an agent can understand and communicate with the platform, making it even possible to have agents written in different programming languages competing with each other. Any new agent just needs to follow the communication template described in Algorithm 3.2.

Algorithm 3.2 Communication Template

```
1: procedure Main
2:     initialization
3:     while game is not over do
4:         read map from input
5:         read players' position from input
6:         if agent is dead then
7:             game is over
8:         else
9:             take next action
```

The user just needs to program the ”next action” procedure, because it will be in this procedure that the agent decides which will be the next action to take, based on the available data. At each time step the agent receives the map and all the players' positions, the current turn,each players' power-ups and ranges. All the information passed is relevant and the ability to adapt to the information can change who wins the game.

3.3.2 Record and Replay

When a user is developing a new Agent it is useful to review its actions in order to understand what, and where, anything went wrong. BAIP allows the user to record the simulations. They will be recorded as log files, in the log folder, and whenever the user wants to review the game he just needs to execute the program with the *replay flag* and pass it the log path file.

When activating the *log flag*, the user activates the recording function. The record will be saved in the Log folder, the name of the log will be generated with the seed chosen by the agent, a random number and the time at the moment.

Each log file has a header describing the presets of the simulation, as seen in Figure 3.6, describing the number of players, the number of players that were AI agents, if there was a manual player and which was his id and finally in which map the simulation took place

```
----------------------------- LOG 2769 ------------------------------
Number of players: 4
Number of agents : 4
Manual player id : -1
Screen height    : 480                          Screen width    : 640
Number of columns: 20                           Number of rows  : 15
Map              : Lvls/lvl0.txt
----------------------------------------------------------------------
```

Figure 3.6: Example of a log file header, where the simulation had 4 agents and where tested in the map constructed from "lvl0.txt"

The controls in the *Replay mode* are the arrow keys: the left and right keys will do a one frame backward pass and forward pass, respectively, and the up and down keys will pass a larger number of frames. We can see a replay example in Figure 3.7.

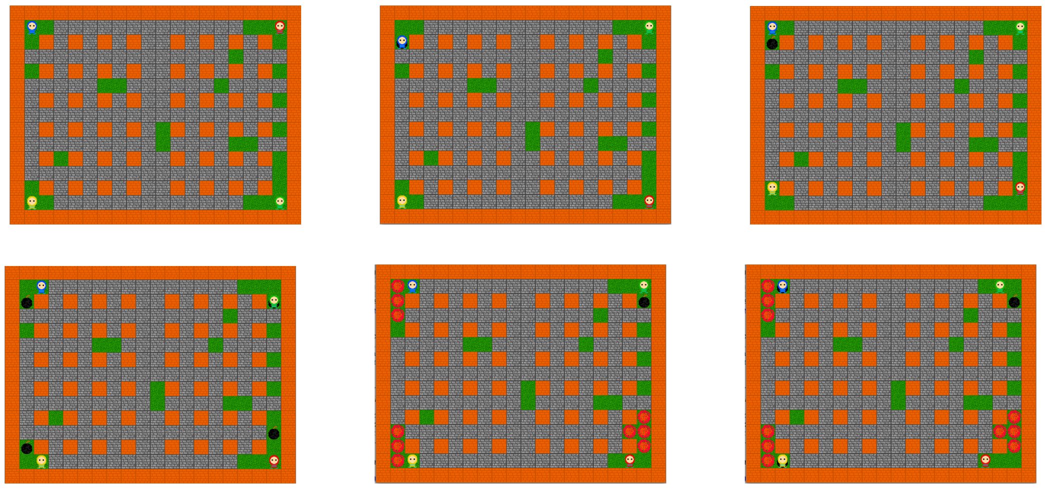


Figure 3.7: Replay mode example, the images show consecutive frames of a simulation in replay mode

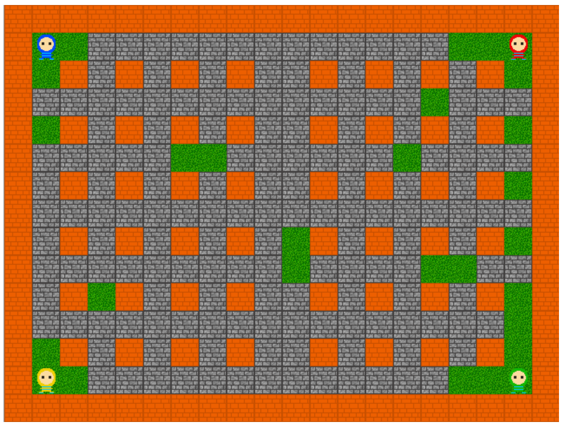
3.3.3 Level Creation

Different users have different necessities: some might want to test how an agent respond against others in an unblocked pathway, others might simply want to test path finding abilities. In order to respond to these necessities BAIP allows its users to create their own levels. Levels are stored as a text file, and they use the following nomenclature. The pathway is seen in the text file as '0' and can be seen in the screen as in Figure 3.1a. The pathway is blocked by gray blocks represented in the text file as '+'. The user can also include blocks with power-ups that are spawned randomly: they are represented in the text file as '-'. Both gray blocks and power-ups can be seen in the screen as in Figure 3.1b. Finally we have the indestructible blocks, represented by '*' in the text file and seen in the screen as in Figure 3.1c. The creator of the map must define the initial position for each player, they are represented in the text file by '1','2','3','4'.

In the current implementation the maps are rectangular following a grid with 20 width and 15 height, but in future versions it will be possible to create maps with different sizes. In the following images we can see an example of a map, and how the text file corresponds to the image drawn in the screen.

```
********************
*10++++-+++++-++003*
*0*+*+*+*++*+*+*-*0*
*++-+++++++++++0++-*
*0*+*+*+*-+*+*+*+*0*
*+++++00++++++0+-++*
*+*+*+*+*++*+*+*+*0*
*+++-+++++++++++++-*
*+*+*+*+*+0*+*+*+*0*
*+++++++++0++-+00-+*
*-*0*+*+*++*+*+*+*0*
*+++-+++++++-+++++0*
*0*+*+*+*++*+*+*+*0*
*20++++-++++-+++004*
********************
```

(a) Text file of the map



(b) Result map from the text file

Figure 3.8: Drawing a map from a text file

3.4 Possible Applications

As AI problems can be formulated in different ways, we wanted to build a platform suitable to investigate these different formulations. From the beginning, BAIP was designed to support the creation and testing of different AI approaches.

Another application is in education. The platform can be seen as a friendly environment to teach AI algorithms to new students. Learning new AI algorithms is not always easy and some concepts might be more comprehensible in practice. One way that the platform helps is in visualization of the actions that the agent takes, making it easier for the student to understand the behavior of certain algorithms. It also makes learning more fun because it allows each student to develop his own AI and then test it against each other.

Its competitive side allows to compare different approaches and analyze which are more effective in the type of game that the platform simulates. It also helps us to understand which algorithm improvements work better.

3.5 Summary

In section **3.1 Game Logic**, we gave an overview of the game logic of the *Bomberman* game and how they were implemented in our platform.

In section **3.2 Architecture**, we gave an overview of BAIP architecture. Some communication examples and how to setup a new agent.

In section **3.3 Key Features**, we give a more detailed explanation on each feature that BAIP supports, and some examples of how to use them.

Finally, in section **3.4 Possible Applications**, we discuss some of the possible applications.

Agents 4

In this chapter we start by analyzing the problem that arises from the *Bomberman game*. We present some simple heuristics that we had in consideration when we were developing the agents. We then follow some novel search-based agents developed in the platform, and some agents that follow the conventional *Reinforcement Learning* (RL) methods. We also present a brief discussion, overviewing all the developed agents.

4.1 Basic Agents

The game as described in the previous chapter has a mazed-base theme, where the map is a grid. It is only natural that most of the essential problems will originate from the exploration of pathways. Our main goal is to be the last agent alive, to achieve this we need to avoid dying, open pathways and target other players. Using these goals we created different approaches in order to achieve them.

Before we developed more complex agents, there was a need to provide a baseline performance for traditional techniques, establishing a point of comparison with future improvements. Using the notions present in section 4.1.1 we created agents corresponding to each heuristic, and in the end we aggregated them all in one novel simple agent. Afterwards we compared the performance of the more complex algorithms against the simpler one, trying to understand if there was really an improvement, and how much did it improve.

Our main testing methodology consists on running a set of n simulations, and observing the results, including who wins, in what order the agents are losing, how many bombs were placed and how many movements were made. We run several simulations, and in different maps, trying to understand what affects the behavior of the agents, and in which circumstances they had the upper hand.

4.1.1 Simple Heuristics

For now we will focus our attention in the heuristics we devised. They are pretty straightforward but overall they improve a lot the performance of our agents, and without them they would probably not even be able to complete the simulations. These approaches are used mainly to give the basic concepts of the game to our agents, in order to improve the agents' results.

4.1.1.1 Avoiding Death

The only way an agent dies in this game is if he is in a position of a grid cell that is currently filled with fire. A cell is filled with fire when a bomb explodes. An explosion will fill all the cells within the bomb range, both vertically and horizontally. The range might vary from player to player depending on the power-ups a player catches.

In order to avoid death one must avoid being in a cell that is in range of a bomb. This is exactly what we will do, by searching the grid around our agent. The first thing we need to do is discover the possible range that the bomb has. There are several ways to know this, and we opted to get the max range that the bomb can have at that moment. After knowing the max possible range a bomb can have, we search the grid and see if we are inside the range of a bomb. This is illustrated in Figure 4.1.

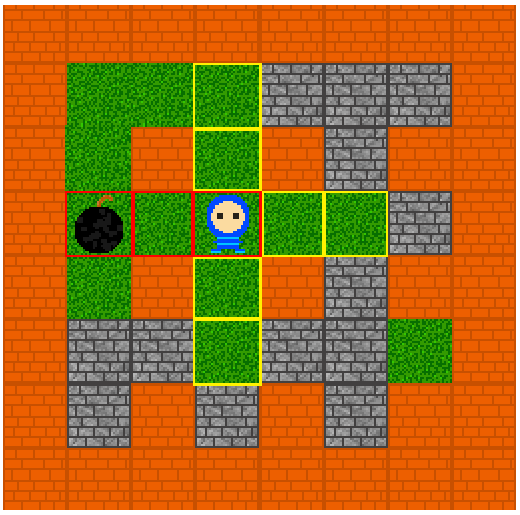


Figure 4.1: Example of an agent checking if it is in the range of a bomb.

As shown, if an agent finds itself in the range of a bomb it needs to search for some place in the grid where he is safe, else if there is no bomb in range the agent can follow a different policy. In the picture above, an yellow tile in the search grid represents a safe place, while the red color indicates danger zones. Every time an agent is in the latter situation, it needs to go to one of the yellow cells.

Using this concept we developed an agent that prioritizes this behavior and is always checking for bombs, running away from them. Its performance is not the best because it can easily be cornered, thus in our search-based agents we had this in mind.

4.1.1.2 Explorer

The pathway is blocked by gray destructible walls. Therefore, in order for the agents to explore the map, they need to unblock the pathway. To achieve this goal, we need to place bombs near the walls. Figure 4.2 illustrates the possible places where an agent could place its bombs in order to unblock the pathway.

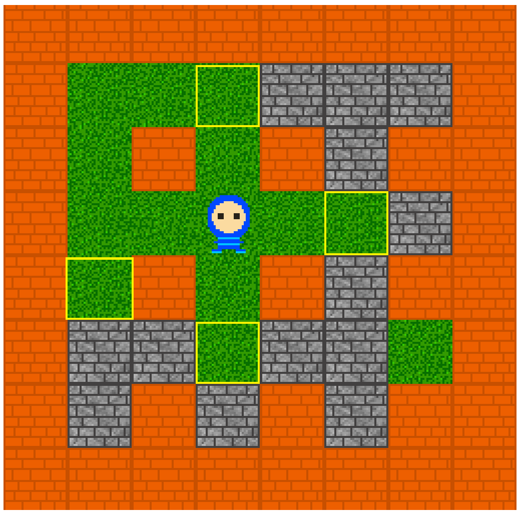


Figure 4.2: Example of an agent checking for walls near

How an agent searches for those places can determine its effectiveness. In our basic agent this search is limited to the surrounding blocks. After placing a bomb an agent uses the *Avoiding Death* heuristic in order not to be killed by its own bomb.

4.1.1.3 Targeting Players

The main objective of our game is to be the last one standing. To achieve this we need to survive the bombs of our adversaries, and we also must try to kill them, or wait for them to kill themselves. The latter might take a long time and the simulation might end before that. So, as established before, to kill an agent we need to strategically place a bomb near the enemy in a position that would not allow him to dodge.

To target a player we first need to understand the pathway of an agent, although an agent has all the information about the map and the positions of all the players, even though they are not in its field of vision, it might not have a pathway unblocked to that player. So, as we can observe in Figure 4.3, an agent can only target what is in it's own pathway.

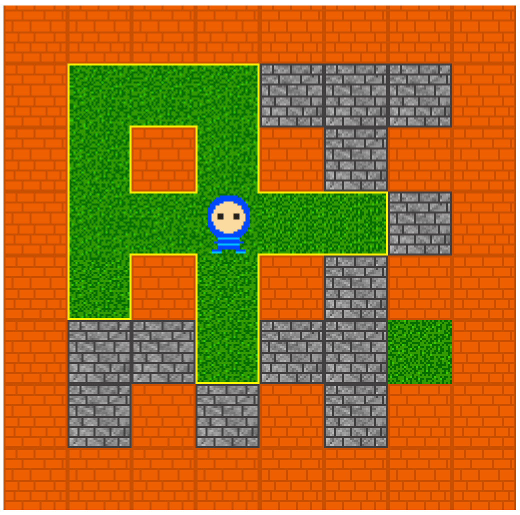


Figure 4.3: Example of an agent field of vision

Checking what an agent can or cannot reach is an important task, and how we explore is crucial to performance of our agent. In the basic agent this search is limited to a certain range surrounding its position. After finding an agent it greedily moves in the direction of the enemy and when it is finally in the enemy's range it places a bomb. This is obviously a really basic way of targeting agents.

4.1.2 Different Basic Agents

Combining all the heuristics described before and somehow following in the footsteps of the work developed previously by [BT02], we created an agent that is able to complete an episode of the Bomberman game. To better understand which decisions an agent takes we can look at the diagram of Figure 4.4.

The basic agents that were constructed all follow the depicted state machine. What differentiates them is how they prioritize their actions. We also did some experiments to see how the performance of agents was affected when they played without some of the heuristics. Given this, we ran our simulation with agents that follow the following four different playing strategies:

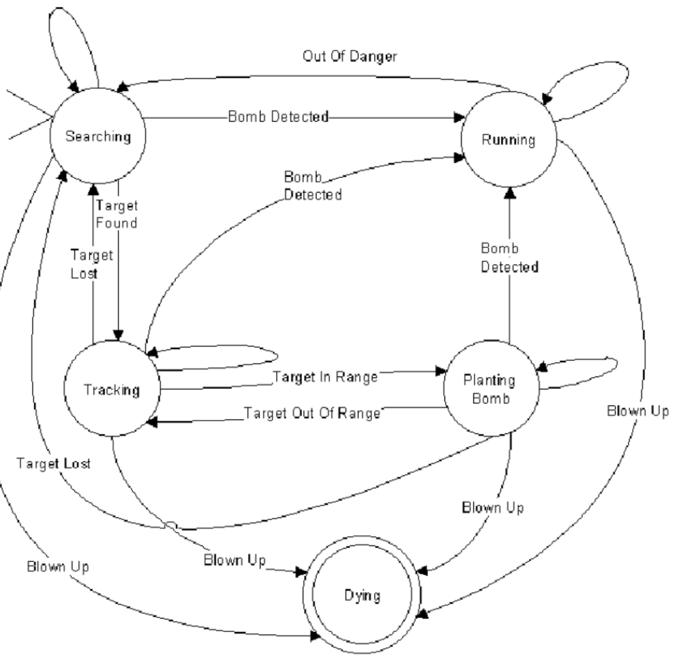


Figure 4.4: State Machine definition for Bomberman agent. Image taken from [BT02]

- *Aggressive strategy* - the agent prioritizes the targeting of the enemies over avoiding death heuristic.
- *Safety strategy* - the agent prioritizes the avoiding death over any other heuristic.
- *Exploratory strategy* - the agent prioritizes exploring the map and blowing up walls over any other heuristic.
- *Random strategy* - when the agent has to decide which heuristic to apply he choses it randomly over the possible heuristics.

In following section we discuss the obtained experimental results using these strategies.

4.1.3 Experimental Results

4.1.3.1 Baseline Agent

In this section we are going to analyze how priorities in the heuristics influence the results of agents created using the same basic set of primitive behaviors, by putting them in competition against each other. These experimental results will help us decide which heuristic we will be using as a baseline agent for further experiments. To do this we created four agents, each one based on one of the basic strategies previously described.

We started by testing all the agents against each other, using the default map without any power-ups and with the players starting randomly in one of the four possible initial positions. First we tested the agents in a one versus one (1vs1) match, where each agent fought every other agent and finally we tested them all together. Table 4.1 summarizes the results for the 1vs1 experiments, with 500 games for every pair of competing agents.

Safety Vs Aggressive	Wins	%
Safety	221	44.20%
Aggressive	213	42.60%
Draw	41	8.20%
Out of time	25	5.00%
Total	500	100.00%

Safety Vs Random	Wins	%
Safety	347	69.40%
Random	56	11.20%
Draw	66	13.20%
Out of time	31	6.20%
Total	500	100.00%

Safety Vs Explorer	Wins	%
Safety	276	55.20%
Explorer	170	34.00%
Draw	10	2.00%
Out of time	44	8.80%
Total	500	100.00%

Random Vs Explorer	Wins	%
Random	85	17.00%
Explorer	330	66.00%
Draw	26	5.20%
Out of time	59	11.80%
Total	500	100.00%

Aggressive Vs Explorer	Wins	%
Explorer	145	29.00%
Aggressive	185	37.00%
Draw	40	8.00%
Out of time	130	26.00%
Total	500	100.00%

Random Vs Aggressive	Wins	%
Random	93	18.60%
Aggressive	354	70.80%
Draw	33	6.60%
Out of time	20	4.00%
Total	500	100.00%

Table 4.1: Simulation results of the one versus one matches between every agent

From these simulations we collected the information of how many times an agent had won, draw and the simulation had run out of time. In order to identify which agent performed better we considered not only the number of wins as a deciding factor but also the number of times it run out of time, as the latter means that the agent is either getting stuck or not efficient enough. Having this in mind we observed that the most successful agents in the 1vs1 simulations are the ones that follow either an *Aggressive strategy* or a *Safety strategy.*

Although the winning rate of the agent that follows the *Aggressive strategy* is superior to the others we did not choose him as the baseline agent, because we also had in consideration the number of draws and especially the number of times the simulation ran out of time. By analyzing the table of the simulations of *Aggressive vs Explorer* and *Safety vs Explorer* agents we can say that the number of times that the *Aggressive*

All vs All	Wins	%
Safety	314	31,40%
Explorer	180	18,00%
Random	55	5,50%
Aggressive	368	36,80%
Draw	49	4,90%
Out of time	34	3,40%
Total	1000	100,00%

Table 4.2: Simulation results of the matches with all the agents at the same time

agent runs out of time when playing against the *Explorer* agent is greater than the number of times that this happens in the *Safety* agent simulations.

We also tested how the agents behaved when they played altogether in the same simulation, against each other. The results are detailed in Table 4.2.

By analyzing the previous table we can say that the performances of both*Aggressive agent* and *Safety agent* are similar. In the end the only thing that really distinguishes these agents was the simulation against the *Explorer agent*, and therefore so we decided to use the *Safety agent* as the baseline agent.

4.1.3.2 The Effects of Power-ups

In this section we show how adding power-ups to the game affects the simulations and its results. From the power-ups presented in section 3 we only tested with the *Bombs power-up*, that increases the number of bombs a player can place at the same time, and the *Range power-up*, that increases the range of the explosions of a player's bomb.

These tests consisted of simulations with the 4 basic agents, starting in random initial positions. The only factor that changes between tests was the map. We used as a baseline the default map without any power-ups, and then we tested with maps that had one of those power-ups and finally we tested with maps that had both of the power-ups. The power-ups used in these simulations were randomly placed in blocks through the map at the beginning of the simulation.

In order to assess how these changes affected the outcome of the simulations we used the following metrics: number of moves per episode and number of bombs placed per episode. We use these statistics to measure the lifetime and the efficiency of the agents, and how the power-ups affect them. Table 4.3 shows the obtained experimental results. It is important to notice that when there was a tie as the result of a simulation, both users were classified as 2nd place, or if it was a three agent tie they were all classified as 3rd place.

TURNS					
	1°	2 °	3 °	4 °	Avg
BOMB&RANGE	14824.4	17982.2	12120.8	10235.3	13790.7
BOMB	16399.6	18458.5	13313.0	10827.0	14749.5
RANGE	12283.3	16875.7	11407.4	9311.9	12469.6
NO POWER-UP	10440.0	19882.0	13390.2	10836.7	13637.2
Average	13486.8	18299.6	12557.8	10302.7	

BOMBS					
	1 °	2 °	3 °	4 °	Avg
BOMB&RANGE	25.1	15.1	25.3	29.8	23.8
BOMB	24.7	15.4	24.8	29.1	23.5
RANGE	24.3	14.8	24.4	26.0	23.2
NO POWER-UP	25.3	15.6	26.7	27.1	23.7
Average	24.8	15.2	25.3	28.0	

TURNS PER BOMB					
	1 °	2 °	3 °	4 °	Avg
BOMB&RANGE	691.7	1188.3	478.9	343.7	675.7
BOMB	664.8	1197.2	637.7	372.0	717.9
RANGE	604.6	1143.0	468.1	367.6	645.8
NO POWER-UP	412.3	1278.8	501.9	399.5	648.1
Average	543.4	1201.8	496.7	368.2	

Table 4.3: Statistics showing number of turns, number of bombs and number of turns per bomb as we vary the type of power-ups available, per ranking placement.

We start our analysis by focusing on the metric that measures the lifetime of the agents. Using the no power-ups case test as baseline we can say that introducing the *range power-up* shortens the average lifetime of the simulations and introducing the *bomb power-up* extends it. The number of bombs also decreases with the introduction of the *range power-up*, but although the number of bombs per episode decreases the average frequency as they place the bombs increases. It is important to keep in mind that the number of turns is largely due to the time it takes for the bomb to explode.

Now that we have seen the overall picture let's understand how the power-ups modified the individual performance of each agent, as detailed in Figure 4.5 and Table 4.4.

By analyzing the individual agent data we can clearly see that when we introduce the

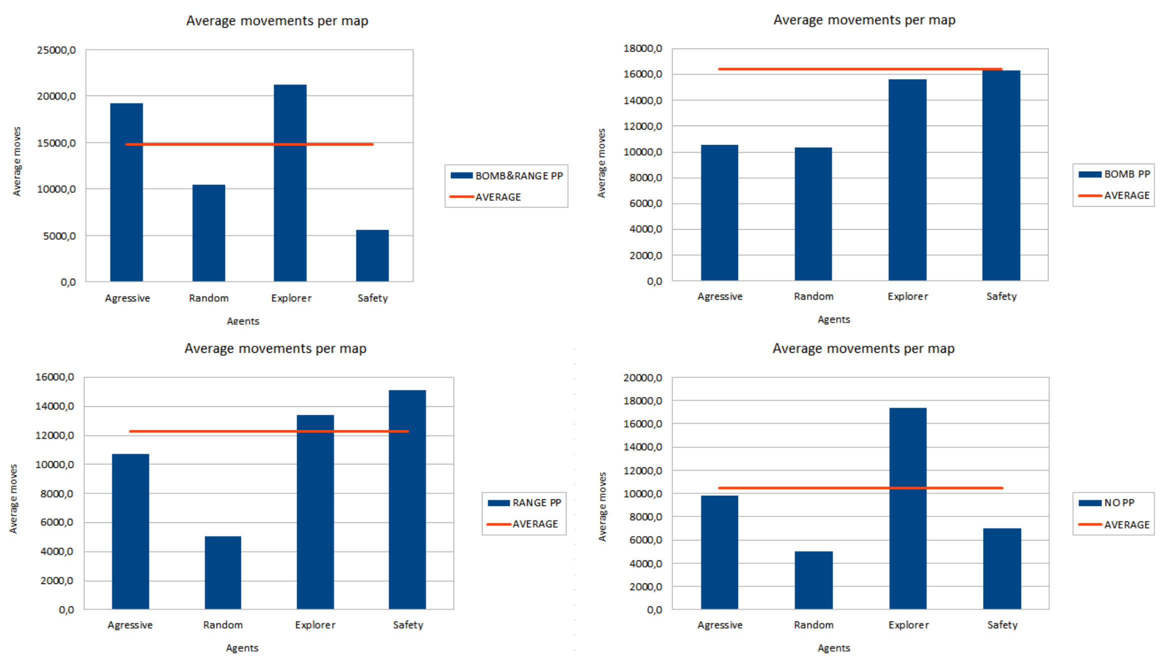


Figure 4.5: Charts comparing the average move per episode of the different ranks.

BOMB&RANGE PP				
	1 °	2 °	3 °	4 °
Aggressive	27	97	166	210
Random	15	455	25	5
Exploratory	72	205	135	88
Safety	22	114	166	198

BOMB PP				
	1 °	2 °	3 °	4 °
Aggressive	20	118	146	216
Random	18	434	40	8
Exploratory	89	208	107	96
Safety	21	91	187	201

RANGE PP				
	1 °	2 °	3 °	4 °
Aggressive	25	127	159	189
Random	11	439	38	12
Exploratory	61	210	142	87
Safety	25	115	151	209

NO PP				
	1 °	2 °	3 °	4 °
Aggressive	74	74	165	187
Random	23	400	49	28
Exploratory	55	151	180	114
Safety	61	84	188	167

Table 4.4: Average number of wins per agent in each different map.

power-ups there is a change in the winning trend, we can also observe that maybe the *Aggressive strategy* does not work as well as it did in the maps without bonus, and finally conclude that the best strategy when there is a map with power-ups is the *Exploratory strategy*. Finally, from the average number of movements per episode of the agents and the number of times they were classified in 1st or 2nd we can verify that there is a correlation between the two.

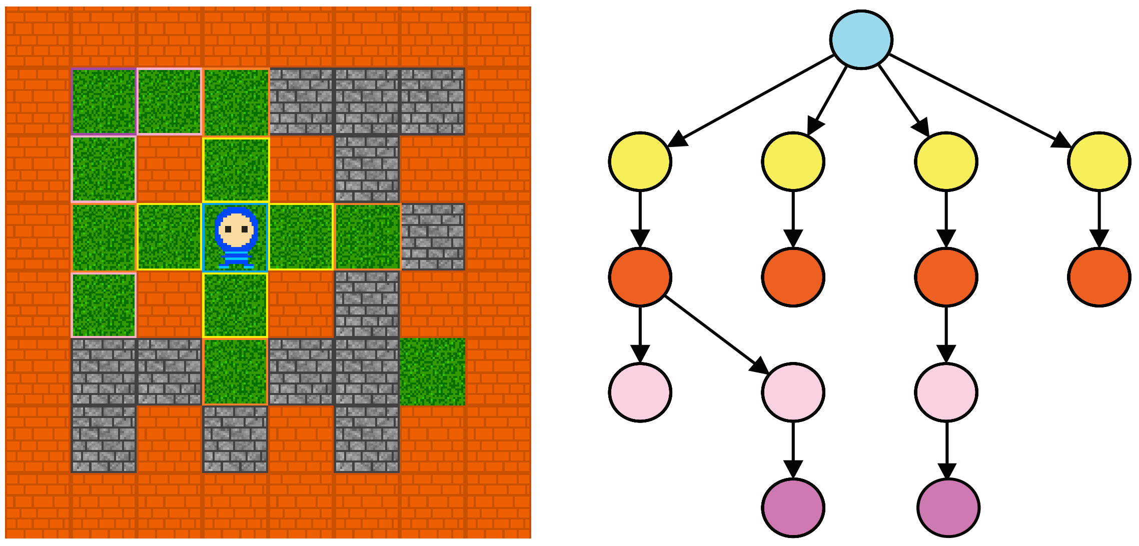
4.2 Search-based Agents

In the previous section we presented some basic approaches to the *bomberman* game. We will now present more complex agents that try to solve the associated exploration problems *search-based algorithms* . These agents are more time efficient as they do not need to search all of the grid in order to make a decision, since they only search what is in reach of an agents inside its own pathway. We will present them in the following subsections.

4.2.1 Generating an Action Tree

To solve the exploration problem that arises from the exploration of the pathway we decided to use a graph traversing algorithm, and went with the *Breadth-first search (BFS)* algorithm because this search method is described as being complete, as it is guaranteed to find a goal state if one exists.

Our grid map can be considered as a directed graph where each cell is a node and they are connected to their horizontal and vertical neighboring cells by actions that are directed edges that represent the actions. Using BFS we start exploring at cell where the player was at. The neighbor nodes are the neighbor cells that were explorable (a cell is explorable when it is either a cell with grass or a power-up block). We can observe in Figure 4.6 a visual representation of an example search tree.



(a) Grid representation of the search tree **(b)** Search tree

Figure 4.6: From a grid representation to a search tree

Using the search tree, calculated at the begin of each iteration, we can now prioritize our heuristics from section 4.1.1. Our first priority was to ensure the survival of the agent, making it search for any bomb in the proximities; the second priority was to check if there was any reachable player; the third was to check if there was any power-up in reach as well and finally if there was nothing left to do we would search for the nearest wall to blow up.

Now that we have explored our pathway, generated a search-tree and also identify where our goals are at, we can finally create an *action tree*. The process of creating an action tree is really simple: after finding the node where our goal is at, we just need to traverse our tree from the goal node to the root and save the edges that lead to the goal node. The root of the created tree is the last edge, the one that comes from the root of the search-tree.

The result is an *action tree* because there can be more than one leaf node at the end. Given that the BFS algorithm searches by levels, there can be more than one goal in the same level. In the next section we present our implementation that uses the *action tree* presented here.

4.2.2 Search-based Agent

Having as a basis the baseline agent presented in section 4.1 we improved on it, adding the generation of the *action trees*. This addition globally and significantly improves the performance of our agent, as can be seen in detail in Section 4.2.5.

To give a better insight on what we did, we now give some intuitions about the improvements. Starting by the agent's survival approach, we keep how it searches for bombs because it is a simpler way and gets the job done. Moreover, by using a search-tree, created when we use the BFS algorithm, we can now search for a safe place to hide from bombs. This addition improves the escape routes for our agent, as it always finds the closest safe cell to its position and the shortest path to it, making our agent harder to eliminate.

The next thing that needed to be improved was our search for goals. In the cases where there is more than one goal it decides which one to pursue by the heuristics priority. In the case of a draw we decided randomly. The agent will always take the shortest path to the desired goal.

We prioritize the targeting of enemy agents because if there is an agent in the pathway there will be a possibility that it will target us, so "the best defense is to attack". We place the bombs when we are in range. If we are safe and there is no agent in our pathway then we will search for the nearest goal, a wall or a power-up, and follow the path to that goal. The agent chooses always the nearest goal, but there is room to improve this selection with some heuristics.

At every time step this agent will search for goals. This can lead to undesired effects, such as the agent becoming overwhelmed by irrelevant actions, or actions undoing the previous planned action. To overcome this problem we present a planning strategy detailed in the next section.

4.2.3 Action Planning Agent

Assuming that the game problem is *nearly decomposable*, meaning that the planner can work on subgoals independently, we decided to use the *action tree* created at each time step as an action plan. When an agent follows a plan, it sends actions from the action tree to the server, starting from the root until it reaches the end. The plan's size is the number of actions an agent will take while following it, and it varies with the *action tree* height.

While using this action plan its size affects the performance of an agent because of the changes in the environment. For instance, if our plan was to target a player and if moves around our final plan state would be outdated. Given this, the plan size is an important factor. We did a comparative analysis of the effectiveness of different plan sizes, in order to decide the most appropriate number. With this goal in mind, we ran several simulations with agents that had different plan sizes.

The results of these simulations are presented in Section 4.2.5. We can conclude that the best plan size for the tested maps is 2.

4.2.4 Improving Search-based Agents with some Heuristics

One major problem that arouse from the previous agents' plans, was that their plan routes did not take in consideration danger zones (cells that are in range of a bomb). Meaning that although they were not in immediate danger they could eventually pass in a node that was considered a danger zone, and this would lead to is death. As we can see ilustrated in figure 4.7.

In order to fix this problem we implemented a heuristic similar to the *Avoiding death* heuristic, but applied in the search-tree. So we are going to prune branches from the

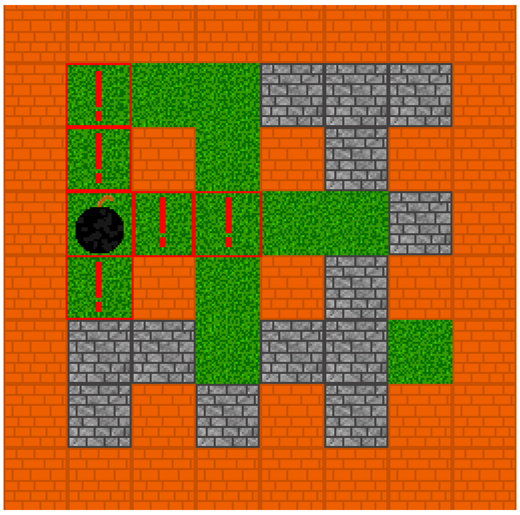


Figure 4.7: Ilustrastes how danger zones are perceived while creating the action tree

search-tree that pass in nodes that are in range of a bomb. Completely avoiding the danger of passing there when the explosion occurs.

We tested agents with this improvement against others without it the results can be seen in Section 4.2.5.

4.2.5 Experimental Results

In this section we will show how we found the best k plan size to our simulations. In order to understand if the changes in the plan size affected the performance of the *search-based agent* we ran simulations against the baseline agent, in the default map in a 1vs1 simulation. The only thing that changed between simulations was the k plan size.

The metric that we used to evaluate the performance of our agent was the winning percentage. We consider that an agent performs better than the opponent when it's winning rate is greater than 50 %.

By observing the chart in Figure 4.8 we can say that the three best results were from the agents that had a plan size where k was either 1, 2 or 3. These agents had a winning rate greater or equal to 75%. To analyze in more detail the behavior of these plan sizes against more intelligent competitors, we tested them against each other using the agents at the same time, in the default map with and without power-ups. From this experiment we obtained Table 4.5.

From this experiment we arrived at the conclusion that the best value for k when

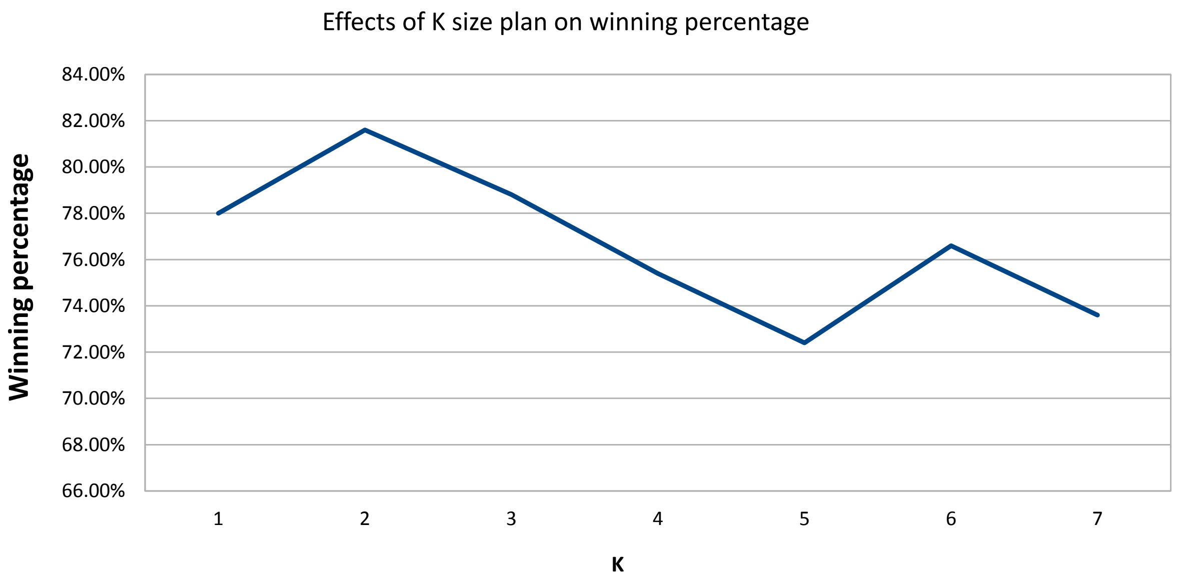


Figure 4.8: Effects of plan's size on winning percentage

K	Normal Map	Random Map
1	159	168
2	165	176
3	153	148

Table 4.5: Table with number of wins of each agent from the simulation of the different plan size agents. We performed 500 episodes in this experiment

playing in the default map is 2. After knowing the best value for k we tested if adding a new survival heuristic to our search algorithm would improve its performance. To test this, we tested our agent with and without the improvement, both against each other and against our baseline agent, in default and random maps.

From this experiment we assess that the newly added heuristic, despite slightly improving the survival rate of our agent on random maps, does not improve its winning rates, as the other agent wins more times. One possible interpretation is that maybe the risk of passing by paths with cells that are in danger zones gives a higher reward than avoiding them and as future work it would be interesting to develop a heuristic to decide if the risk to reward rate ratio is good enough to take a risk.

In conclusion the search based agents can perform around 80% times better than the baseline agent. The best agent that was developed was the agent that uses the search-based planning with a value k of 2 without the heuristic improvement described.

	Default	Random
Safety	366	397
Improved	222	188
Draw	12	15

	Default	Random
Improved	301	276
Baseline	250	148
Draw	49	176

	Default	Random
Safety	467	403
Baseline	97	64
Draw	36	133

Table 4.6: Tables of the experiments where we tested if adding a new survival heuristic would improve the performance. The numbers indicate the number of times each agent wins.

4.3 Reinforcement Learning Agents

The main difference between learning and planning algorithms is that a planning algorithm has access to a model of the world, whereas the latter involves not knowing how the world works and learning how to behave from direct experience with the world. This chapter demonstrates how we can formulate our game problem as a *Reinforcement Learning (RL)* problem, and how we can solve it using *Q-learning* and *Sarsa*. As reviewed in Section 2.3, agents try to learn the optimal policy from its history of interaction with the environment.

The history of an agent is a sequence of state-action-reward, so the first thing we need to do is to establish how they are represented in our game environment. Let the states be represented by the game matrix and the players positions, the actions by the possible actions an agent can take and the rewards defined by the user's criteria. Recall that these methods use temporal differences to estimate the value of $Q^*(s, a)$, and keep a table of $Q(S, A)$ where S is the set of states and A is the set of actions.

Section 4.3.1 introduces an RL-based framework in the Bomberman game environment, shows how we can develop agents in this mode, and hints about its potential. Section 4.3.3 present the parameters chosen and how they affect our agent. In Section 4.3.2 we compare our initial implementation with an implementation that uses *eligibility* *traces*. Finally we show every experiment we did using these RL-agents and the respective comparisons.

From all the AI formulations, we chose the RL representation even though it is difficult because of *delayed rewards*, the *explore-exploitation dilemma* or even by the changing dynamics of the world. Regardless of these problems, RL representations have demonstrated to have the most potential, and recent developments in the AI field demonstrated their almost unlimited possibilities [LBH15, MKS+15, MKS+13]. Even though we were not able to replicate all of the most recent results (given the time constraints for producing this thesis), we tried to create a platform that is able to accommodate them in the future.

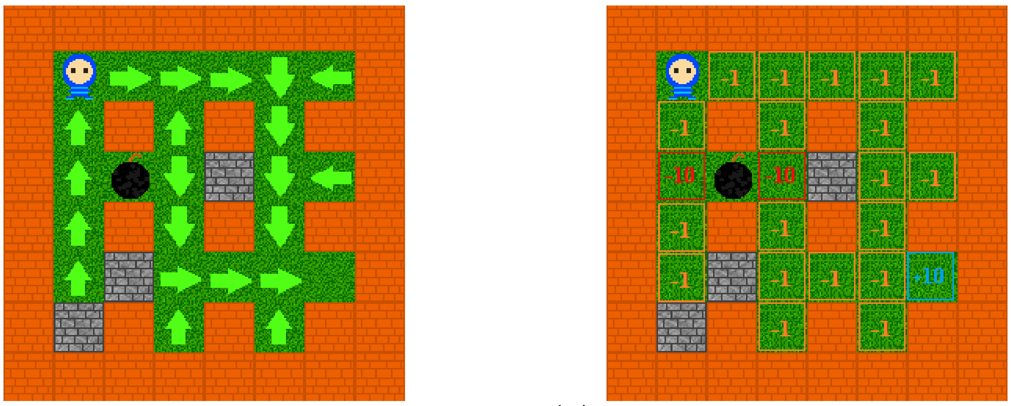
4.3.1 BAIP Learning Mode

Using BAIP RL-mode we started by solving the canonical example of the *Grid World* [SB98]. Because our agents move in a grid and walls block the agent's path, this problem is an ideal test for any RL-agent as we can easily check if our agent finds the optimal policy. We tested this with *Sarsa(λ)* and *Watkins's Q(λ)* implementations.

As we could see in section 2.3, Q-learning is an off-policy learner and learns the value of the optimal policy independently of the agent's actions, while *Sarsa* is an on-policy learner that learns the value of the policy being carried out by the agent, including the exploration steps [LK10]. In a *Grid World* this difference could be experienced in the agent's exploration steps, because sometimes, when exploring, some large penalties may occur, and *Sarsa* takes this into account.

Using the game board presented previously we now want to transform it into the *Grid World* problem. This was achieved by letting the user define an objective point, and every time an agents gets to that point of the map it will be repositioned to its' original point. In order to leave the platform the more generic possible, the scoring function will be handled by the agents. This will leave the scoring decisions to the agents' creator. In Figure 4.9 we can observe the scoring for each grid cell and the respective optimal actions to take in the grid positions. This is an illustration of what an agent is supposed to arrive at, in the end of a learning simulation.

In the previous figures we can observe that eventually the agent will learn that passing near a bomb is bad and will find the shortest and safest pass to the position of the blue ball. Considering all of this, we implemented on-policy (Sarsa(λ)) and off-policy (Watkins's Q(λ)) control methods using function approximation, where both methods use linear function approximation with binary features and use an ε-greedy policy



(a) The green arrows represent the optimal actions in those positions.

(b) An example of rewards that an agent receives in those positions.

Figure 4.9: Example of a *Grid World* problem

for action selection. Different function approximations, such as a neural network or a decision tree with a linear function at the leaves, could have been used [LBH15, MKS$^+$15, MKS$^+$13].

4.3.2 Improving with Eligibility traces

Until now only the previous state-action pair was updated when a reward was received. This leads to the delayed reward problem, because when an agent takes a number of steps that lead to a reward, all of the steps along the way could be held responsible and thus receive some of the credit or the blame for a reward [LK10].

Considering this we implemented in our agents an *eligibility trace*, that specifies how much a state-action pair should be updated at each time step. When a state-action pair is first visited, its eligibility is set to 1, and at each subsequent time step is reduced by a factor of $\lambda\gamma$. On any subsequent visit its eligibility is incremented by 1. The eligibility trace is implemented by an array e[S,A], where S is the set of all states and A is the set of all actions. After every action is carried out, the *Q-value* for every state-action pair is updated [LK10].

We tested agents that use eligibility traces against agents that do not and concluded that the agents that use are able to learn faster the optimal policy. These tests can be observed in section 4.3.4.

4.3.3 Choosing Parameters

There are a set of parameters that are used in Sarsa and Q-learning algorithms. For choosing these, we did an empirical parameter search, looking for values that would improve the algorithm overall performance.

In order to test our RL methods we used the *Grid World* problem testbed, and tested how well an agent scores at the end of an episode. An episode starts with the agent at position $(0, 0)$ and ends at position $(18, 14)$. The agent needs to find the shortest path in order to have the best possible score.

We started by searching what was the most suitable α (learning rate) parameter and how its variance would affect the outcome in the scores of our agent. Figure 4.10 shows the obtained variance, while using the *Sarsa* algorithm.

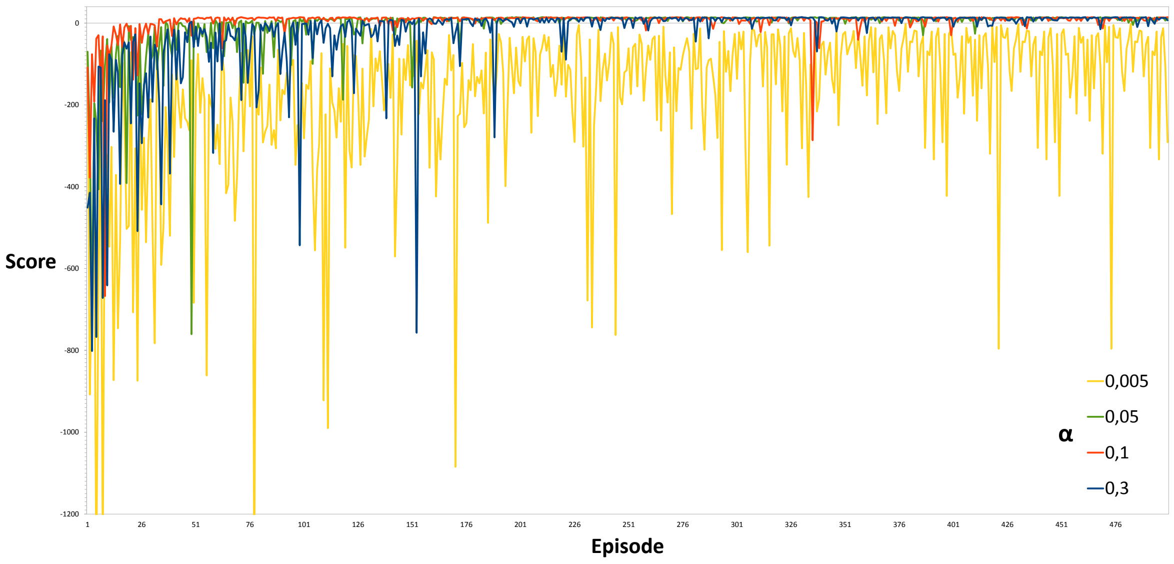


Figure 4.10: Chart showing how the score changes when we change the α value of the agents.

When running our tests we set our range between $[0.005, 0.3]$, because every value outside this range will cause our agent to behave poorly. By analyzing the previous graphic we can conclude that the bigger the value of α the faster the agent finds a good policy, but that policy might not be the best possible policy. On the other hand the smaller the value of α the agent takes longer to find a good policy that produces positive scores, but might find a better policy, and even find the optimal policy. After searching for a balanced value we decided to use a learning rate of 0.1.

In the previous tests we used a static γ (discount factor) value, afterwards we tested how the changes in the discount factor affected the performance. The range of this value will be different of the previous set range. We tested the values 0.7 and 0.99, since every simulation we tried with a value under 0.7 performed too poorly.

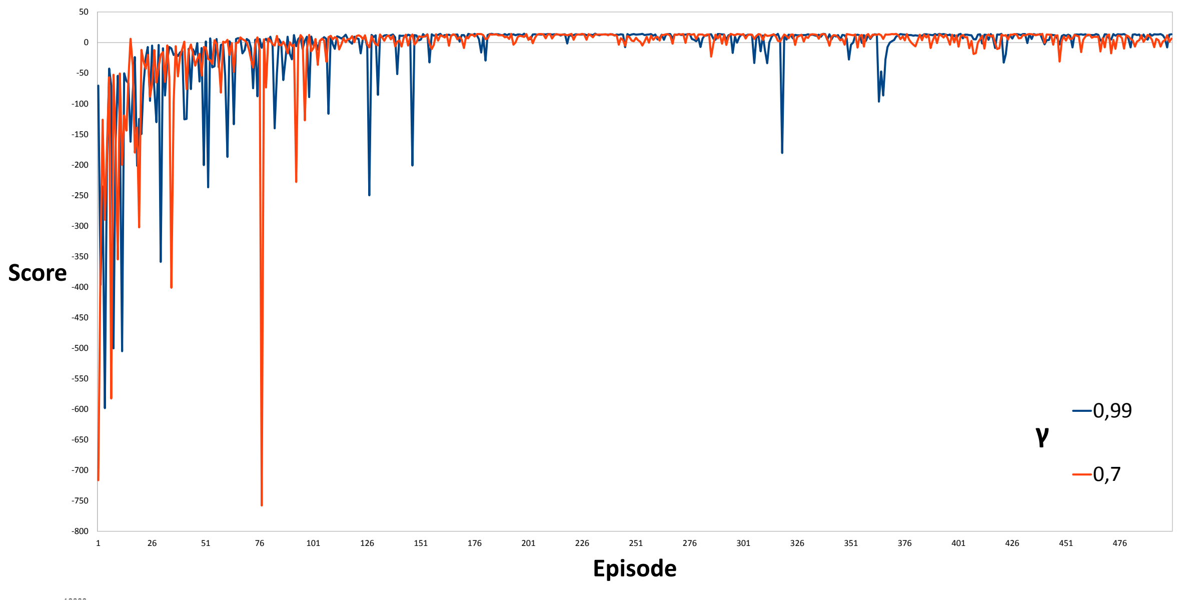


Figure 4.11: Chart showing how the score changes when we change the γ value of the agents.

γ determines the importance of future rewards, a value near 0 will make the agent only consider current rewards, while a value approaching 1 will make it search for a long-term high reward. By analyzing Figure 4.11 we can conclude that values near 1 yield better results.

We then searched the best value for ϵ (Exploration / Exploitation Rate) trying to address the exploration/exploitation dilemma. We know that after a finite number of plays, each cell has been visited once [AMS09], thus we need to find a value that balances the gathering of information with the ability to make the best decision given current information. Our search range was set between $[0.1, 0.9]$. We see how this factor affects the score in Figure 4.12. We found that the best value of ϵ for our problem was 0.2.

Finally, using the Q-learning method with eligibility traces, we tested the λ value. This factor affects learning speed of our agent, particularly when rewards are delayed by many steps as it is the case. The change in learning speed can be observed in Figure 4.13. We can see that when λ is closer to 1 the agent learns faster.

In summary, considering mainly the number of episodes it took the agent to find a good policy, we decided to use the following parameters:

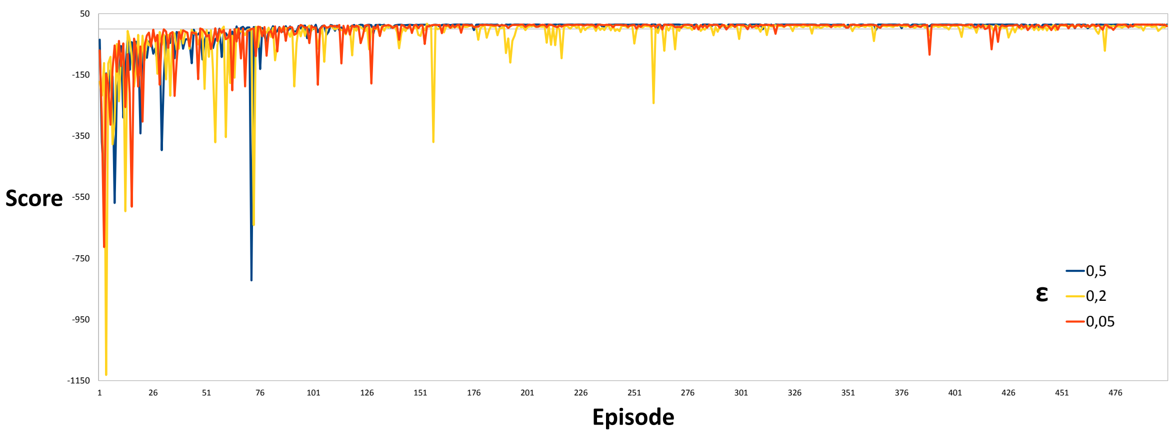


Figure 4.12: Chart showing how the score changes when we change the ϵ value of the agents.

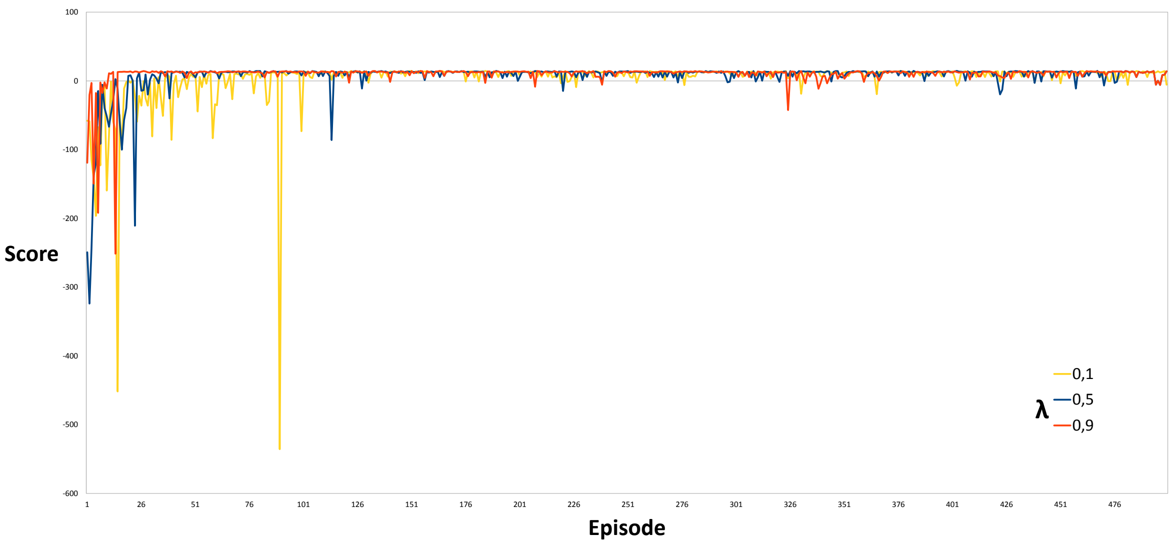


Figure 4.13: Chart showing how the score changes when we change the λ value of the agents.

- α (learning rate) - 0.1
- λ (lambda) - 0.9
- γ (discount factor) - 0.99
- ϵ (Exploration / Exploitation Rate) - 0.2

4.3.4 Experimental Results

Using the parameters described in the previous section and the same *Grid World* problem, we tested how an *eligibility trace* would affect our Q-learning agent. As we can see in Figure 4.14, an agent with an *eligibility trace* is faster to learn the optimal policy.

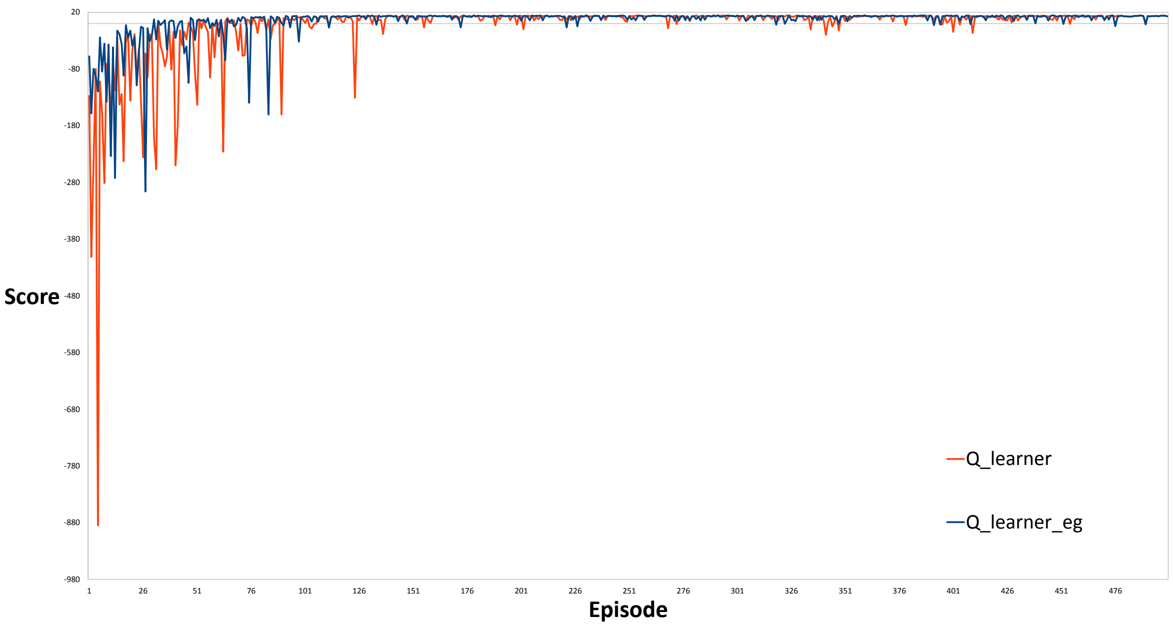


Figure 4.14: Chart comparing the score in each episode of two agents, one that uses eligibility trace and one that does not.

With this we confirm that it often makes sense to use eligibility traces when data is scarce and cannot be repeatedly processed, as is often the case in on-line applications, as the Bomberman game.

Finally, we tested how the *Sarsa* and *Q-learning* algorithms compare with each other, in order to conclude which one has a better performance in this problem. We can see the results in Figure 4.15.

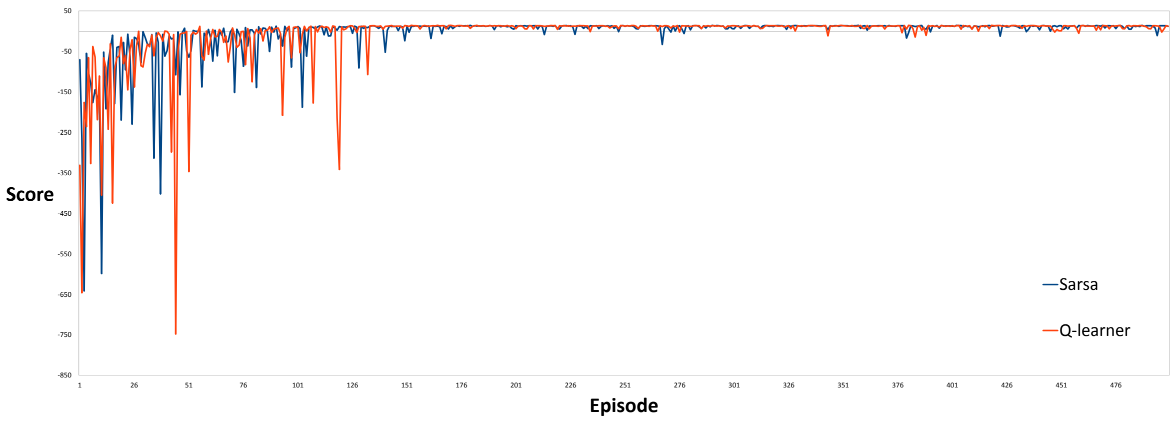


Figure 4.15: Chart showing the score in each episode of the agents.

Since in both cases the agents learned a behavior by experience and then proceed to act based on what they had learned, and in the end both agents converged to the same optimal policy, in this scenario we cannot say which algorithm is definitely better.

4.3.5 Further Improvements

In this section we will detail some possible improvements that were not implemented due to the limited amount of time available for this dissertation.

Going from a predefined position to another as in the *Grid World* problem is not really of practical usefulness in a complete game environment, because the other agents might behave in completely different ways and our main objective is to survive. For this we would need to teach our agent more than only the path finding function, and we this could be achieved for instance running our RL agents in a normal game environment, where each time they die they would receive a negative score, being re-spawned.

Furthermore, a method of improving our RL agent would be to use *Feature Selection*. There are too many states and thus, in order to create a more skillful agent, we would need to give him some information about the important features of the game.

Using features of both the state and the action to provide features for the linear function, where $F_1(s,a),\ldots,F_n(s,a)$ are numerical features for the state s and the action a, these would be used to represent the *Q-function.* [LK10]

$$Q_w(s,a) = w_0 + w_1F_1(s,a) + \cdots + w_nF_n(s,a) \tag{4.1}$$

where,

$$w_i \leftarrow w_i + \eta\delta F_i(s,a) \tag{4.2}$$

The selection of features needs to be made very carefully. Some possible features that we would select in a future implementation would be the following:

- $F_1(s,a)$ - has value 1 if action a would most likely take the agent from state s into a location where the agent could die and has value 0 otherwise.
- $F_2(s,a)$ - has value 1 if action a would most likely take the agent from state s into a location where there is a wall and has value 0 otherwise.
- $F_3(s,a)$ - has value 1 if action a would most likely take the agent from state s into a location where there is a power-up and has value 0 otherwise.
- $F_4(s,a), F_5(s,a), F_6(s,a)$ - has a value equal to the distance between our agent's position and one of the other agent's.

4.4 Discussion

We started by presenting some heuristics that we need to consider. Different approaches lead to different results, and we considered the behaviors presented as the fundamental ones to assure the minimal survival of any agent. Using them we created a baseline performance for traditional agents, and tried to improve it with other AI methods.

Afterwards we gave the structure needed to created search-based methods and presented only one search method here, using the *Breadth-first search (BFS)* algorithm. Some other methods can be used, such as *Depth-first search (DFS)* or *A* search algorithm*, with the latter being the most popular choice for path-finding problems on large environments. A* can be seen as a *Greedy Best-First-Search* in that it can use a heuristic to guide itself.

Using the search methods we created very simple planning agents, tested the plan's size, and concluded that its main flaw is when there are changes in the environment that the agent did not foresee. Given this, it is important to find a good plan size.

We also explored the challenges of implementing RL agents, such as the sensitivity of the learning rate (α), the state dynamics and choosing the right reward signals, the delayed reward problem and delayed consequences (as the bombs took some time to explode). In order to tackle these problems we based our solutions in some examples from [Nad10].

In conclusion, this chapter served to explore different types of AI problems formulations, and although these different approaches might arrive to different policies, they could be combined to either learn faster or to perform better. There are already examples of successful combinations of search methods and learning methods that worked [SSM08].

BAIP can now be used as a fully functional platform for learning, developing and testing AI methods.

4.5 Summary

In section **4.1 Basic Agents**, we introduced the environment and some heuristics to solve the game problem. We created a baseline agent that is used as a comparison to check if the other agents have improvements.

In section **4.2 Search-based Agents**, we introduced the search-based agents. We started by explaining how we created an action tree, and how we use it for planning agents. We show some improvements to the initial implementations and afterwards we try to find an optimal plan size.

In section **4.3 Reinforcement Learning Agents**, we introduced RL in the context of our platform. We started by showing the BAIP learning mode, that facilitates the development of RL-agents and we studied how different algorithms and parameters behave better.

Finally, in section **4.4 Discussion**, we give an overview of all experimental results.

Conclusion and Future Work 5

In this section we give a summary of what was achieved and we explore some possible future directions.

5.1 Contributions

The main goal was to conceive a Bomberman-based AI platform and to use it to create AI agents, proving its feasibility and ease of use. This goal was achieved and whilst pursuing it we made the following major contributions:

- **A multi-language AI platform using Bomberman**, a language agnostic platform that is capable of communicating, via input and output, with an agent written in any programing language (it only needs to follow the communication protocol presented). The platform achieved a fully functional state, with an appealing graphical view of the world state, detailed logs of every simulation and the possibility of customizing the maps and power-ups.

- **A set of simple heuristics** capable of providing basic functionality for an agent, namely the capability to avoid explosions, to explore the maze (by destroying blocks) and to kill other agents. From the experimental results and the analysis of these behaviors, we have proven that the capability to avoid explosions is the one that has proven to be the most essential. We also analyzed which power-up has more impact in the number of moves. The results lead us to believe that the range power-up decreases the number of moves and the bomb power-up increases them. Finally, we have also shown how one should deal with the dilemma between exploring and actively pursuing enemy agents. The response for this dilemma varies, with results from maps with power-ups showing that exploring is the best approach, but in maps with no power-ups being aggressive presented us with a better performance.

- **A search based agent** capable of producing an action tree and to efficiently explore the state-space. From the experimental results this agent has proven to be better than any basic agent. After a detailed competitive analysis of the different variations of the planning agent, with a focus on discovering the optimal plan size, we arrived at the conclusion that, for the maps tested in our work, the best plan size was two.

- **A simple formulation of a Reinforcement Learning (RL) problem** within the platform, coupled with an implementation of both Sarsa and Q-learning algorithms that are able to tackle the problem. The obtained results conform to what was expected, showing both the validity and feasibility of our approach and showcasing its ease of use. We have also conducted an empirical search trough algorithm parameters and we have explored the *eligibility traces* improvement.

5.2 Future Work

This work only briefly touched some of the subjects in the AI area, and a number of possibilities for future directions arose whilst developing our platform and our agents. Some possible topics for future research are the following:

- We only research the effects of the bombs the range power-ups. In the future it should be investigated how the other power-ups affect the game.

- While the platform tries to follow every aspect that constitutes a good teaching platform, as a future work it should be tested in a real teaching environment. A good example for testing our platform in a pedagogical context would be to use it either in an introductory programming course or in a introductory AI course.

- We would like to develop more heuristics, such as opening pathways having in consideration the location of the nearest opponent or trying to trap another agent using the map and our bombs. There are a lot of possibilities and heuristics that can be developed.

- We would like to implement agents that use *Monte Carlo methods*, and test its performance against our previous developed agents.

- We would like to further improve our RL agents using for instance more complex problems and feature selection, as described in Section 4.3.5.

- Finally, it would be interesting to use BAIP to implement agents that use *Deep Reinforcement Learning* methods, such as the DQN method, and analyze its performance against our previous developed agents.

5.3 Closing Remarks

During this work I have greatly extended my knowledge in the *Artificial Intelligence* area and contacted with *General Game Playing*. It involved knowledge representation, planning, logic and search-based challenges. I have also deepened my knowledge in *Reinforcement Learning* and had a chance to get familiarized with the state of the art in the area. I tried to implement some of the most recent advances, following new papers being published during the lifetime of this thesis, and although somehow unsuccessful in incorporating the most recent developments, I have learned how some of these very recent algorithms work and how to cope with the adversities of this area. Finally, building the platform gave me a chance to improve my software engineering and design skills.

Appendix A

A.1 Server-Client Communication Code in C++

Listing A.1: Server-Client Communication Code

```
//Holds world map matrix
char **wordl_map;
// World map width and height
int NUM_COLS; //width
int NUM_ROWS; //height
// Number of active players
int NUM_PLAYERS;
// Id of the current agent
int PLAYER_ID;
int bomb_range = 4;
//Reinforcment learning objectives
int OBJECTIVE_X;
int OBJECTIVE_Y;
//To save RL q values
std::fstream log_data;
// Players' position
int *x;
int *y;
// Players' ranges
int *r;
// Who is alive  1 - alive || 0 - dead
int *alive;
int *speed;
int *teams;
```

Listing A.2: Server-Client Establish Communication Code

```
/* Establishes a connection with the server */
bool connect()
{
    std::string mesg;
    //Get line from pipe input
    std::getline (std::cin ,mesg);
    //Break line into protocol NUM_COLS NUM_ROWS
    std::istringstream iss(mesg);
    char pmesg;
    iss >> pmesg;
    iss >> NUM_COLS;
    iss >> NUM_ROWS;
    iss >> NUM_PLAYERS;
    iss >> PLAYER_ID;

    if (pmesg == CONNECT)
        std::cout << "CONNECTED";
    else
        std::cout << "ERROR_CONNECTING" << std::endl;
        return false;

    return true;
}
```

Listing A.3: Server-Client Establish Communication Code When Using Reinforcement Learning

```
/* Establishes a connection with the server */
bool connect_RL()
{
    char line[20];
    //Get line from pipe input
    std::cin.getline (line,20);
    std::string mesg(line);
    //Break line into protocol NUM_COLS NUM_ROWS
    std::istringstream iss(mesg);
    char pmesg;
    iss >> pmesg;
    iss >> OBJECTIVE_X;
    iss >> OBJECTIVE_Y;
    iss >> NUM_COLS;
    iss >> NUM_ROWS;
    iss >> NUM_PLAYERS;
    iss >> PLAYER_ID;
   // std::cout << NUM_COLS <<NUM_PLAYERS << std::endl;
    if (pmesg == CONNECT)
        std::cout << "CONNECTED";
    else
        std::cout << "ERROR_CONNECTING" << std::endl;
        return false;

    return true;
}
```

Listing A.4: Server-Client Update Map Code

```
/* Receives and updates world map */
bool update_Map()
{
    std::string mesg;
    //Get line from pipe input
    std::getline (std::cin ,mesg);
    //Break line into protocol  wordl_map
    std::istringstream iss(mesg);
    char pmesg;
    std::string world;
    iss >> pmesg;
    iss >> world;
    if (pmesg == MAP)
        for(int i = 0; i < NUM_ROWS; i++)
            for(int j = 0; j < NUM_COLS; j++)
                wordl_map[i][j] = world[mIndex(i,j)];
    else
        std::cout << "ERROR_UPDATING_MAP" << std::endl;
        return true; // gameover ?

    //Inform the server that the map was successfully received
    std::cout << "RECEIVED";
    return false; // gameover ?
}
```

Listing A.5: Server-Client Update Players' Status Code

```
/* Receives and updates all players positions and ranges*/
bool update_Players()
{
    std::string mesg;
    //Get line from pipe input
    std::getline (std::cin ,mesg);
    //Break line into protocol NUM_COLS NUM_ROWS
    std::istringstream iss(mesg);
    char pmesg;
    iss >> pmesg;

    if (pmesg == POSITIONS)
        for( int i = 0; i < NUM_PLAYERS; i++)
            iss >> y[i]; // x position
            iss >> x[i]; // y position
            iss >> r[i]; // ranges
            iss >> alive[i]; // is player i alive?
            iss >> speed[i]; // player speed
            iss >> teams[i]; // player's team

    else
        std::cout << "ERROR_CONNECTING" << std::endl;
        return true;// gameover ?

    return false;// gameover ?
}
```

References

[A$^+$94] Louis Victor Allis et al. *Searching for solutions in games and artificial intelligence.* Ponsen & Looijen, 1994.

[AMS09] Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvári. Exploration-exploitation tradeoff using variance estimates in multi-armed bandits. *Theor. Comput. Sci.*, 410(19):1876–1902, April 2009.

[Bay09] Jessica D Bayliss. Using games in introductory courses: tips from the trenches. In *ACM SIGCSE Bulletin*, volume 41, pages 337–341. ACM, 2009.

[Ben09] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[BNVB13] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 06 2013.

[Bot10] Léon Bottou. *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*, chapter Large-Scale Machine Learning with Stochastic Gradient Descent, pages 177–186. Physica-Verlag HD, Heidelberg, 2010.

[BT02] Arran Bartish and Charles Thevathayan. Bdi agents for game development. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2*, AAMAS '02, pages 668–669, New York, NY, USA, 2002. ACM.

[CHH02] Murray Campbell, A Joseph Hoane, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1):57–83, 2002.

[DK10] John DeNero and Dan Klein. Teaching introductory artificial intelligence with pac-man. In *Proceedings of the Symposium on Educational Advances in Artificial Intelligence*, 2010.

[GP06] Abraham P. George and Warren B. Powell. Adaptive stepsizes for recursive estimation with applications in approximate dynamic programming. *Machine Learning*, 65(1):167–198, 2006.

[Gui11] Erico Guizzo. How googles self-driving car works. *IEEE Spectrum Online, October*, 18, 2011.

[IYA16] Goodfellow Ian, Bengio Yoshua, and Courville Aaron. Deep learning. Book in preparation for MIT Press, 2016.

[Lai01] John E Laird. Using a computer game to develop advanced ai. *Computer*, 34(7):70–75, 2001.

[Law04] Ramon Lawrence. Teaching data structures using competitive games. *IEEE Transactions on Education*, 47(4):459–466, 2004.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[LK10] Poole David L. and Mackworth Alan K. *Artificial Intelligence: Foundations of Computational Agents.* Cambridge University Press, New York, NY, USA, 2010.

[MF07] Amy McGovern and Jason Fager. Creating significant learning experiences in introductory artificial intelligence. *ACM SIGCSE Bulletin*, 39(1):39–43, 2007.

[MKS$^+$13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[MKS$^+$15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[MTR11] Amy McGovern, Zachery Tidwell, and Derek Rushing. Teaching introductory artificial intelligence through java-based games. In *AAAI Symposium on Educational Advances in Artificial Intelligence, North America*, 2011.

[Nad10] Yavar Nadaf. Game-independent ai agents for playing atari 2600 console games, 2010.

[New00] Monty Newborn. Deep blue's contribution to ai. *Annals of Mathematics and Artificial Intelligence*, 28(1-4):27–30, 2000.

[RFS+09] Pedro Ribeiro, Michel Ferreira, Hugo Simões, et al. Teaching artificial intelligence and logic programming in a competitive environment. *Informatics in Education-An International Journal*, (Vol 8_1):85–100, 2009.

[RHW88] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[Rie15] Mark O Riedl. A python engine for teaching artificial intelligence in games. *arXiv preprint arXiv:1511.07714*, 2015.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning:An Introduction*. MIT Press, 1998.

[SECR13] Scott Sosnowski, Tim Ernsberger, Feng Cao, and Soumya Ray. Sepia: A scalable game environment for artificial intelligence teaching and research. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence*, 2013.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[SSM08] David Silver, Richard S. Sutton, and Martin Müller. Sample-based learning and search with permanent and transient memories. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 968–975, New York, NY, USA, 2008. ACM.

[Tay11] Matthew E Taylor. Teaching reinforcement learning with mario: An argument and case study. In *Proceedings of the Second Symposium on Educational Advances in Artifical Intelligence*, pages 1737–1742, 2011.

[Tes95] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[Tok10] Michel Tokic. *KI 2010: Advances in Artificial Intelligence: 33rd Annual German Conference on AI, Karlsruhe, Germany, September 21-24, 2010. Proceedings*, chapter Adaptive ϵ-Greedy Exploration in Reinforcement Learning Based on Value Differences, pages 203–210. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[WC07] Wai-Tak Wong and Yu-Min Chou. *An Interactive Bomberman Game-Based Teaching/ Learning Tool for Introductory C Programming*, pages 433–444. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[WD] Christopher J.C.H. Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8(3):279–292.

[YK15] Du-Mim Yoon and Kyung-Joong Kim. Challenges and opportunities in game artificial intelligence education using angry birds. *IEEE Access*, 3:793–804, 2015.

[ZGL$^+$12] Haifeng Zhang, Ge Gao, Wenxin Li, Cheng Zhong, Wenyuan Yu, and Cheng Wang. Botzone: A game playing system for artificial intelligence education. In *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.

[Zha16] Zhongheng Zhang. When doctors meet with alphago: potential application of machine learning to clinical medicine. *Annals of translational medicine*, 4(6), 2016.