UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

# Playing video-games via planning with simulators

**TRABAJO FIN DE MASTER**

Master Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

*Author:* Cristóbal Cervantes Rodríguez
*Tutor:* Eva Onaindia de la Rivaherrera
*Tutor:* Sergio Jimenez Celorrio

Course 2018-2019

# Resum

Una de las tasques importants de la intel·ligència artificial consisteix en crear intel·ligència capaç de dominar múltiples tasques en lloc d'especialitzar-se en una sola. Els videojocs són un entorn ideal per provar i comparar tot tipus de tècniques d'intel·ligència artificial. Una de les vessants d'aquesta àrea consisteix a aconseguir que un mateix programa sigui capaç de enfrontar-se a diversos tipus de jocs (*General Video Game*), de manera que un mateix agent pugui dominar diferents tipus de joc.

Aquest projecte es centra en la creació d'un agent amb les bases de la competició General Video Game - Artificial Inteligence el que vol dir que l'agent serà capaç de jugar en diferents videojocs inspirats en l'estil arcade (similars al Atari 2600) A més de enfrontarse a diferents tipus de jocs, l'agent no tindrà informació prèvia sobre el funcionament i els objectius del joc. De fet l'agent es pot trobar jocs contra els que no ha jugat prèviament.

Un dels objectius del nostre projecte consistirà en avaluar una tècnica de recerca anomenada **Iterative Width** (IW) com a nucli principal del nostre agent. Es buscarà entendre les bases d'aquesta tècnica alhora que desenvolupem l'agent. Estudiarem l'ús d'aquesta tècnica per *General Video Game*, enfrontant el nostre agent basat en IW contra 30 jocs de diferents tipus.

Compararem el nostre agent amb un altre agent basat en Monte Carlo Tree Search que és una tècnica àmpliament utilitzada en *General Video Game*. Finalment es dissenyarà i implementaran millores sobre l'algoritme bàsic de iterative Width, que explotin les fortaleses d'aquest algoritme i supleixin les debilitats amb determinats jocs.

**Paraules clau:** Agents, Videojocs, General video game, Intel·ligència artificial, Iterative Width

# Resumen

Una de las tareas importantes de la inteligencia artificial consiste en crear inteligencia capaz de dominar múltiples tareas en lugar de especializarse en una sola. Los videojuegos son un entorno ideal para probar y comparar todo tipo de técnicas de inteligencia artificial. Una de las vertientes de este área consiste en conseguir que un mismo programa sea capaz de enfrentarse a varios tipos de juegos (*General Video Game*), de manera que un mismo agente pueda dominar distintos modos de juego.

Este proyecto se centra en la creación de un agente con las bases de la competición General Video Game - Artificial Inteligence lo que quiere decir que el agente será capaz de jugar en diferentes videojuegos inspirados en el estilo arcade (similares al Atari 2600) El agente no solo debe ser capaz de enfrentarse a distintos tipos de juegos, sino que carecerá de información previa sobre el funcionamiento y los objetivos del juego. De hecho el agente se puede encontrar juegos contra los que no ha jugado previamente.

Uno de los objetivos de nuestro proyecto consistirá en evaluar una técnica de búsqueda denominada **Iterative Width** (IW) como núcleo principal de nuestro agente. Se buscará entender las bases de esta técnica a la vez que desarrollamos el agente. Estudiaremos el uso de esta técnica para *General Video Game*, enfrentando nuestro agente basado en IW contra 30 juegos de diferentes tipos.

Compararemos nuestro agente con otro agente basado en Monte Carlo Tree Search que es una técnica ampliamente utilizada en *General Video Game*. Por último se diseñará

e implementarán mejoras sobre el algoritmo básico de iterative Width, que exploten las fortalezas de dicho algoritmo y suplan las debilidades con determinados juegos.

**Palabras clave:** Agentes, Video Juego, General video game, Inteligencia artificial, Iterative Width

# Abstract

One important task of artificial intelligence is to create intelligence capable of mastering multiple tasks rather than specializing in a single task. Video games are an ideal environment to test and compare all kinds of artificial intelligence techniques. One field of this area is to achieve that the same program is able to face several types of games (*General Video Game*), allowing the same agent to dominate different game styles.

This project focuses on the creation of an agent with the bases of the General Video Game - Artificial Inteligence competition which means that the agent will be able to play in different arcade-inspired video games (e.g. similar to Atari 2600). The agent must not only be able to deal with different types of games, but will also lack prior information on the functioning and objectives of the game. In fact the agent can find games against which he has not previously played.

One of the goals of our project will be to evaluate a search technique called textbfIterative Width. (IW) as the nucleus of our agent. We will try to understand the bases of this technique at the same time that we develop the agent. We will study the use of this technique for *General Video Game*, confronting our agent based on IW against 30 games of different types.

We will compare our agent with another agent based on Monte Carlo Tree Search which is a technique widely used in textGeneral Video Game. Finally we will design and implement improvements on the basic algorithm of iterative Width, which exploit the strengths of that algorithm and compensate the weaknesses with certain games.

**Key words:** Agents, Video Game, General video game, Artificial intelligence, Iterative Width

# Contents

Appendices

# List of Figures



# List of Tables

# CHAPTER 1
# Objectives/Motivation

One of the great challenges of our time is to get algorithms to perform tasks that until a few years ago we thought were reserved only for humans. A subset of this great challenge consists of elaborating programs capable of playing games like a human or even better. One of the great milestones in this field was made by AlphaGo in 2015 when it became the first machine to beat a professional Go player.

There are many examples of AI focused on this field, although we could differentiate two large groups. Those who like AlphaGo are focused on a single game, and those in which the AI is capable of adapting and playing several games. In single games, programmers has the possibility of provide specific knowledge about the game. That is, using advisable or unadvisable movements or even by means of the use of databases, such as for example openings in the game of chess.

On the other hand, there is another branch in which the aim is to develop AI techniques capable of playing several games, being able to adapt to new environments and to learn the rules and functioning of the game on the fly, thus simulating the learning period that a human being would have. In this case we have platforms such as the Arcade Learning Environment (ALE) or the General Video Game AI competition (GVGAI) that facilitate the development of these agents.

Our goal in this project is to develop an agent capable of performing in several games, according to the bases of the GVGAI competition. This agent must be able to be faced with games that it has not seen previously and of which he does not know their rules. It will analyze the games functionality applying state-of-the-art search algorithms to achieve the goals of each game at many different scenarios as possible.

# CHAPTER 2
# Related Work

Games have long been popular benchmarks for Artificial Intelligence (AI). Many researchers have studied algorithms and techniques that try to approximate optimal play in computer games as different as *Chess, Go, Car Racing games, Ms. PacMan, Real-Time Strategy (RTS)* games and *Super Mario Bros*. Research on these and other games has enabled some interesting advances in algorithmic AI such as the use of parallelized Alpha-Beta pruning (in *Chess*), or the progress seen in one of the most popular algorithms in Game AI, Monte Carlo Tree Search (MCTS), in the game of *Go* [24] [28].

Early research put the focus on board games like *Chess, Scrabble* or *Checkers*. Thanks to the great advances in the development of tree-search methods, games like *Scrabble* or *Checkers* are now resolved [25].

Nowadays researchers are more interested in video games where the complexity is harder and there is room for improvements. The main games addressed by researchers are those in which an agent can easily learn how to play but it is difficult to master. In order to create and evaluate agents developed by researchers, platforms such as The Arcade Learning Environment [7] and the General Video Game - Artificial Intelligence (GVG-AI) [2] have emerged.

In this project we will focus on a specific type of video games developed for the GVG-AI competition framework with some particular features that we will detail below. Most of them are inspired by old arcade games like *Pacman, Alliens* or *Boulder Dash*, among others.

## 2.1 Game definition

Following we will define some relevant concepts of game theory that will allow us to describe and to contextualize the kind of games addressed in this research.

- According to the number of players, games can be divided into:
  - **One-player games.** There is only one character in the game, who must solve the problem or obtain the best score possible. This kind of games are sometimes named *puzzle games* [15].
  - **Two-player games.** These games are characterized by trying to beat an opponent. There are often two fronts with opposing interests trying to achieve the game objectives. Board games like *Chess* or *Go* are examples of two-player games with a large tradition in AI for games researches [10].
  - **Multi-player games.** Games that feature more than two players are comprised in this group. The complexity of these games increases because players can

form alliances or competitions among themselves. In these games, there are common and opposed interests with the rest of players, and in some cases it is necessary to carry out some type of negotiation in the best interest of a subset of players.

Games are characterized by different features that determine the type of game. Following we enumerate a list of features that are used to characterize games. The concepts described below are not incompatible, so a game may present several of the following features.

- **Zero-Sum games.** In some games, the payoff for player A equals exactly the negative of the payoff for player B. This means that whatever A wins, B must pay, and vice versa. In this type of games, there does not exist a win-win solution. *Poker* [8] and *Gambling* are popular examples of zero-sum games since the sum of the amounts won by some players equals the combined losses of the others. Games like *Chess* and *Tennis*, where there is one winner and one loser, are also zero-sum games.[1]

- **Stochastic vs deterministic games.** Stochastic games are dynamic games with probabilistic transitions played by one or more players. In each step, a new random state is created whose distribution depends on the previous state and the actions chosen by the players. In contrast, a game is deterministic if the result of the actions taken by the players leads to completely predictable outcomes [31].

- **Simulatenous games vs sequential games.** In game theory, a simultaneous game is a game where each player chooses his action without knowledge of the actions chosen by other players. In other words, more than one player can issue actions at the same time. Games like *Rock-paper-scissors* or *RTS games* are some examples of simultaneous games [22]. In contrast, sequential games are those in which one player chooses his action before the others choose theirs. In this case, the latter players have information available of the first players' choice, which allows them to create more informed strategics. *Chess* or *Go* are some examples of sequential games.

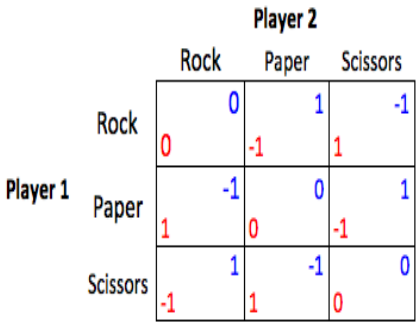
  In game theory, the representation of the games varies according to the type of game. Simultaneous games are often represented by a matrix that encompasses all possible combinations of players' moves. Each cell specifies the reward and penalty for each player if that combination of moves is feasible. The game *Rock Paper Scissor* is represented in Figure 2.1a. Sequential games are commonly represented by decision trees in which the time axis is represented by the depth of a node in the tree. The first steps of the game *Tic-Tac-Toe* are represented in Figure 2.1b.

- **Perfect, imperfect and incomplete information games.** Perfect information refers to the fact that each player has the same information that would be available at the end of the game. This is, each player knows or can see other player's moves. A good example would be *Chess*, where each player sees the other player's pieces on the board.

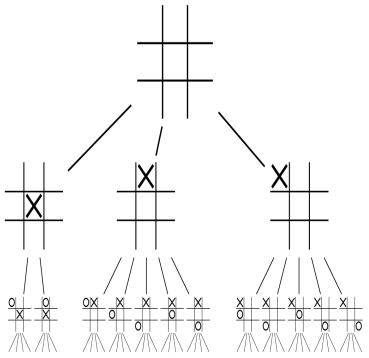
  Imperfect information games are those where players know perfectly the types of other players and their possible strategies, but are unaware of the actions taken by them.

  In incomplete information games, other details of the game are unknown to one or more players. This may be the player's type, their strategies, their payoffs, their

[1] https://www.investopedia.com/terms/z/zero-sumgame.asp




**(a)** *Rock Paper Scissor* normal matrix.



**(b)** Decission tree in *Tic-Tac-Toe*

**Figure 2.1:** Game theory representation of simultaneos and sequential games.

preferences or a combination of these. Imperfect information games are therefore incomplete information games but not vice versa.

Games with simultaneous moves are generally not considered games of perfect information. This is because players hold information that is hidden to the others and so every player must make a move without knowing the opponent's hidden information [16].

- **Combinatorial games.** In this type of games, the difficulty of finding an optimal strategy stems from the combinatorial explosion of possible moves. Combinatorial game theory typically studies sequential games with perfect information. Thanks to the advances in mathematical techniques over the last years, the complexity of combinatorial games has been largely reduced. This improvement has led to the creation of different variants of games with a high combinatorial component. One example is the *Infinite Chess*, where the game is played on an unbounded chessboard [18].

## 2.2 Control strategies

In this section we present some of the most relevant control strategies for developing game agents.

- **Hand-coded strategies**

  This type of strategy consists of the design of manually coded heuristic to guide the search process.

  Rule-based controllers use hand-coded strategies that return the action that complies with a set of relatively simple conditions. An example of rule-based controllers can be found in the *Super Mario Bros* game, which defines a particular behaviour where the character Mario constantly runs right and jumps whenever possible. In this case, the game agent would contain a single rule that determines when to jump.

  Creating a rule-based agent entails having a detailed knowledge on the dynamics of the game so as to be able to extract all possible situations and actions. In addition, an unforeseen situation could lead to an unexpected behaviour.

  Another limitation of rule-based agents is that the rules are derived from the mechanics of the game. As a result, we will get an agent specialized in that game but

unable to adapt its actions to new environments. For example a world-champion *StarCraft* agent will not be able to play *Pac-Man* at all [9].

- **Combinatorial Search**

  This strategy consists in turning the goal of finding the best move into a search problem. The game is simulated at each player's turn producing all possible move combinations. The result of the simulations is then used to build a search tree that helps find the best possible move to realize [27].

  In most cases the search space is too large and so we cannot guarantee the optimal solution, thus making necessary to implement heuristics to guide the search. In dynamic environments it is also recommended to start the search at every step of the game. This is because the new information enables to narrow down the search space and hence discard the exploration of useless parts of the search tree.

  The sequential two-player games typically use this type of control strategy. The complete information of sequential games makes the simulation unique without possible variations as it happens in stochastic games. In addition, the fact that players move by turns provides players with a natural computational time to think about the next move, a time that can be used to execute AI algorithms.

  Algorithms like *minimax* or *alpha-beta* [17] are designed to exploring sequential games where a player tries to find the movement that maximizes an evaluation function while minimum values of this function favor the opponent. Alpha-beta incorporates a pruning condition that improves the search speed with the same result of a minimax algorithm.

  Unlike sequential games, in stochastic games there is no guarantee that the game evolution will follow the search exploration carried out by the player. One possible solution to overcome the difficulties of stochastic games is to apply stastical approximations along several iterations of the game. One example of combinatorial search algorithms is *Monte carlo tree search* (MTCS) [11]. The MTCS analyzes the most promising moves, expanding the search tree based on random sampling of the search space from a selected leaf node of the tree. In the selection process, better nodes are more likely to be chosen.

- **Learning-based controllers**

  This type of control strategy consists in improving the behaviour of a game agent by using previously recorded examples of the game. The agent is trained by playing multiple games or observing professional matches, and iteratively penalizing or rewarding the agent actions in order to obtain better outcomes [29].

  The state of the art in learning-based controllers is plagued with numerous variations of neural networks (NN), specially deep convolutional neural networks. In games like *Chess* or *Go*, NN are used to evaluate a concrete intermediate state of the game and predict the final outcome. In addition, NN are used to predict the best action to apply in the current state of the game. The combination of search strategies like MTCS with NN is very helpful to conduct a search simulation.

  One more advantage of learning-based controllers over combinatorial search controllers is that is not necessary to run a simulation process to find the best action. Instead, as a result of the training process we get a model embedded in the controller that acts as a function, returning the action to execute in every state.

  A different approach that generally outputs good results is using Genetic algorithms. They are stochastic, parallel search algorithms based on the mechanics of natural selection and evolution. Genetic algorithms were designed to efficiently

search large, non-linear, poorly-understood search spaces where expert knowledge is scarce or difficult to encode and where traditional optimization techniques fail [12]. The development of a game agent based on genetic algorithms requires first to decide the game parameters to be optimized and then encode these parameters into a chromosome layout. Subsequently, an initial population with sufficiently different chromosomes to guarantee genetic diversity is created. Iteratively, we simulate the behaviour of each individual of the population in the game and create a new generation crossing some individuals, which are chosen according to the results obtained. The idea is to select the fittest individuals and let them pass their genes to the next generation. In order to avoid the tendency towards local optimization, a mutation component is introduced, where genes not coming from any of the parents are included with certain probability.

- **Hybrid Approaches:**

  Hybrid approaches emerge as a combination of some of the aforementioned described techniques. The *Dynamic scripting* [23] game is an example of the utilization of an hybrid approach, where hand-coded strategies are combined with learning-based algorithms. Under this hybrid approach, the agent is guided by a set of rules but the decision of which rule to choose is made by means of a reinforcement learning process. *Dynamic scripting* executes the agent multiple times, adding a reward or penalty proportional to how well the agent performed in a particular level of the game.

## 2.3 Multi-game algorithms

Research in videogames aims at developing two types of game agents:

- Agents oriented to play a single game; e.g., AlphaGo [1] which was the first AI program that won versus a professional championship of *Go*; or Stockfish [5], which won the $14^{th}$ Top Chess Engine Championship of *Chess*; or competitions like the Mario AI championship [4] shown in Figure 2.2.

- Agents able to play several games and to adapt themselves to new environments. The implementation of agents for playing different games is encouraged by platforms such as the Arcade Learning Environment or the General Video Game AI competition (GVG-AI).

The goal of agents that play a single game is to find a behaviour that allows them to master in such game, trying to maximize the score or minimize the time it takes to solve the game. To this end, researchers implement techniques like the ones explained in section 2.2 and build an specialized agent for a specific game. The result is an agent that excels in the game in question. The limitation of one-game agents is that they are unable to adapt to the conditions of a new game and they would most likely get poor results when playing a different game [32].

On the other hand, some researchers are interested in developing general game agents capable of playing different games, even games for which the agent has never been trained for. In this case, the final objective is to test agents in general game competitions and to develop AI techniques that can possibly be later applied to some real-life tasks. Ultimately, researchers aim to develop general AI techniques applicable to different games instead of a specialized tool for only one game.

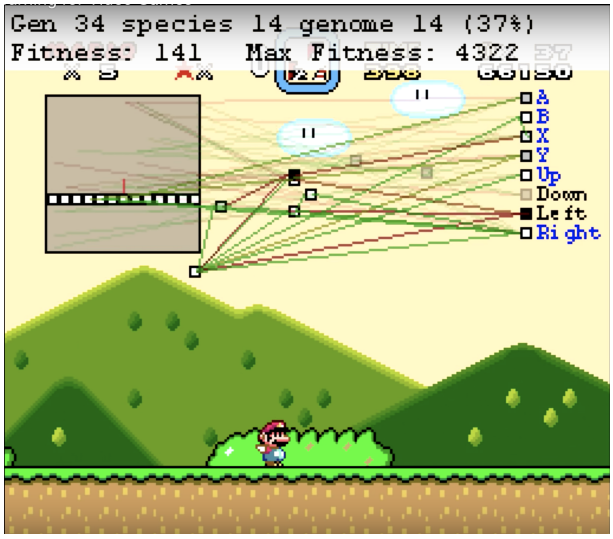



**Figure 2.2:** Genetic-based controller in Mario AI competition

Our aim in this Master Thesis is to develop a general game agent, specifically adapted to the framework provided by the GVG-AI competition, whose details will be explained in section 3.1. Since we are interested in agents that are capable of playing various different games, we discard the use of hand-coded strategies as they would excel in one game but would most likely perform badly in other games.

Among the remaining control strategies presented in section 2.2, we believe the use of combinatorial search strategies are more appropriate for general game agents. Compared to learning-based controllers, combinatorial search algorithms do not require massive training data and they are more easily extendible to different games, allowing us to draw conclusions about their strengths and weaknesses.

There exist a large number of combinatorial search algorithms applicable to general video games. In the following section, we present the Monte Carlo Tree Search algorithm, one of the most widely spread technique for game playing.

### 2.3.1. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a tree-search algorithm that became very popular in the last decade due to its performance in the game *Go*. It is a search algorithm that builds an asymmetric tree in memory using a forward model of the game. MCTS builds on the idea of estimating the true value of an action through random simulations and using these values for an efficient best-first policy. The algorithm works iteratively in an anytime fashion, that is, it progressively computes a more accurate estimate as long as more time is given to the algorithm and it can be terminated at any point, returning the current best estimation [14].

The MCTS algorithm is used in General Video Games context to decide the next action to be played by the agent at the game. For this purpose at each movement the agent runs a search process that allows to make more informed decision. To guide the search process, MCTS is based on four stages that are summarized in the figure 2.3. These four stages are repeated iteratively until a stop condition is reached, which can be a time limit or any other condition. In General Video Game that limit should be a maxim time allowed to return the next move. The first action to the best path obtained at the moment of the stop will be returned for be played by the agent. The stages are:

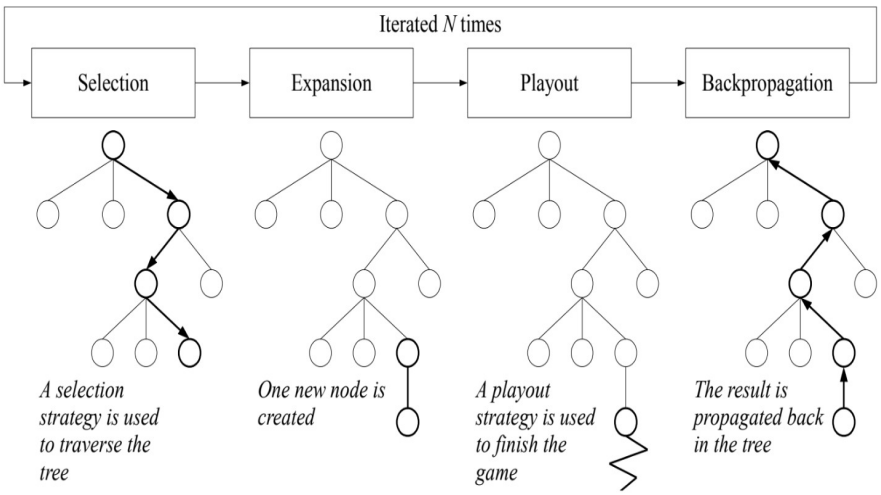



**Figure 2.3:** Monte Carlo Tree Search loop

1. **Selection:** In the selection step, the algorithm choose the tree node that will be expanded in the current iteration. For this purpose, MCTS uses a utility function that measures how desirable is the node to be expanded. One of the most widely used functions is Upper Confidence Bound for Trees (UCT) as it combines the experience provided by previous iterations (*explotation*) along with the desire to explore paths not yet explored (*exploration*). MCTS utility function must have an equilibrium between exploration and explotation. Otherwise the behaviour would not be as desired. If we mitigate the exploration component, the algorithm will behave voraciously based on the experience gained in the Playout phase. That is to say, the search will be marked by the first rewards obtained being always the same nodes the ones selected for the expansion. On the other hand, if we eliminate the experience component, the algorithm will always choose the unexplored nodes, obtaining a behavior similar to a BFS. In the example of the figure 2.3 we can appreciate how the algorithm starts at the root node, chooses the child located to his right because it is the one with the highest score according to the evaluation function. It continue descending by the tree and ends in the lower level choosing the right by the same criterion. The selection process ends when it reaches a node that either represents a terminal state (where the game has been ended or the branch has been pruned), or a node that is not completely extended (i.e. a node in which there are some applicable movements or actions have not been considered yet).

2. **Expansion:** An unexplored successor of the node that was not fully extended is randomly chosen and a new node is added to the tree to be evaluated.

3. **Simulation:** A simulation (also called a playout) is performed from the expanded node in order to determine its value. That is, from the recently added state a complete game is simulated, either randomly, with a weighted heuristic, or using computationally expensive evaluations with more sophisticated heuristics. The result of simulation (Whether the game has been won or lost, or the score obtained in the simulation) is used for to obtain a value (prize, reward, etc.) that determines the usefulness of that branch for the player.

4. **Backpropagation**: The value of the new node is back-propagated all the way up to the root node through the selected nodes. This value, is extracted from the simulation result. It can be the game score of final simulation node, or an evaluation function applied to last node. In any case, the value propagated to the explored

nodes of the search tree will be part of the exploitation component of the utility function of step 1 (*selection*).

MCTS includes a family of algorithms that perform the above steps but they vary in the strategies involved along the process. The three strategies are:

- A strategy for selecting tree nodes according to their usefulness (in the selection phase). Among the possible choices for selecting a node are: (1) selecting the successor with the greatest profit; (2) selecting the node which has been visited in the greatest number of winning games; (3) selecting a node that verifies the two previous conditions; (4) selecting a node that maximizes a function like UCT (Upper Confidence Bound for Trees).
- A strategy for the creation of new nodes (in the expansion phase). A uniform random creation strategy is usually chosen, but if some additional information on the problem is available, it can be used to make a decision on which action to take (and, consequently, which node is created).
- A strategy of generating a complete game from the newly created node (in the simulation phase). Here it is relatively common to introduce, if necessary, some specific knowledge of the domain of the problem (of the game).

The main difficulty in selecting a child node is to maintain an equilibrium between the exploitation of the average win rate obtained in previous simulations and the exploration of moves with few new simulations. The main formula for balancing exploitation and exploration in games is called UCT (Upper Confidence Bound for Trees)[14].

$$UCT_j = \bar{X}_j + C\sqrt{\frac{\ln n}{n_j}}$$

This equation is solved at every selection step. In this equation, $\bar{X}_j$ is the average value of node $j$ determined by the simulations and is normalized to be in [0, 1]; $n$ is the total number of selections performed from the parent node, and $n_j$ is the number of times the child node $j$ is selected. In the equation, the left term ($\bar{X}_j$) stands for exploitation while the right term ($\sqrt{\frac{\ln n}{n_j}}$) represents the exploration. The parameter C controls this relationship allowing us to direct the algorithm, favoring one of the two factors.

Also the function resolves the selection of unvisited nodes. When the successor has not yet been visited, the UCT value of the node will be $\infty$ and, therefore, the successor will be chosen by the strategy.

# CHAPTER 3
# Background

In this chapter we will detail the necessary concepts to follow the contributions of this project. First, we introduce The General Video Game AI Competition (GVG-AI) [2], where all the rules that govern a game agent are defined. Next, we will explain the most relevant concepts about automated planning as well as the selected algorithm to implement our game agent.

## 3.1 General Video Game - Artificial Intelligence competition

General Video Game-AI (GVG-AI) competition is a platform whose objective is to explore the problem of creating controllers for general video game playing. In other words, agents oriented to play in multi-game environments.

The competition has been running since 2014 and is organized annually by the University of Essex. In the competition, participants can upload their agents to be tested with different games. Figure 3.1 shows the main web page of the GVG-AI competition that took place in 2018 [2].

The aim of the GVG-AI competition is for participants to create a single agent capable of playing any given game, that is, an agent that is able to play a wide variety of games but that when playing a game it does not actually know which game is playing. The competition is meant to put complex algorithms to the test, challenging their adaptability to new situations.

GVG-AI provides a framework that facilitates the task of developing game controllers. The framework presents a batch of different games in which the developed game agents will be executed and it also provides mechanisms to allow the agent interact with states of the game. Hence, the task of the participants focuses on creating and developing the algorithms that will allow the avatar (game agent) to play. That is, developers can put their efforts in creating the AI strategies without requiring any knowledge or particular skills on game programming. This allows experts in the field to quickly and effectively develop and test new techniques.

According to the rules of the GVG-AI competition, the game agent is called at every step of the game, and the agent must return a discrete action to apply in a particular game state state in no more than 40ms. In this time period, the agent needs to explore the different paths or combinations of actions in order to obtain the best rewarding action and eventually win the game. The deadline of 40ms together with the complexity of the games makes it difficult to explore all possible combinations. For this reason, an intelligent strategy that enables to obtain a good action and avoid the "game over" is needed. In addition, the agent has one second time at the start of the game to perform

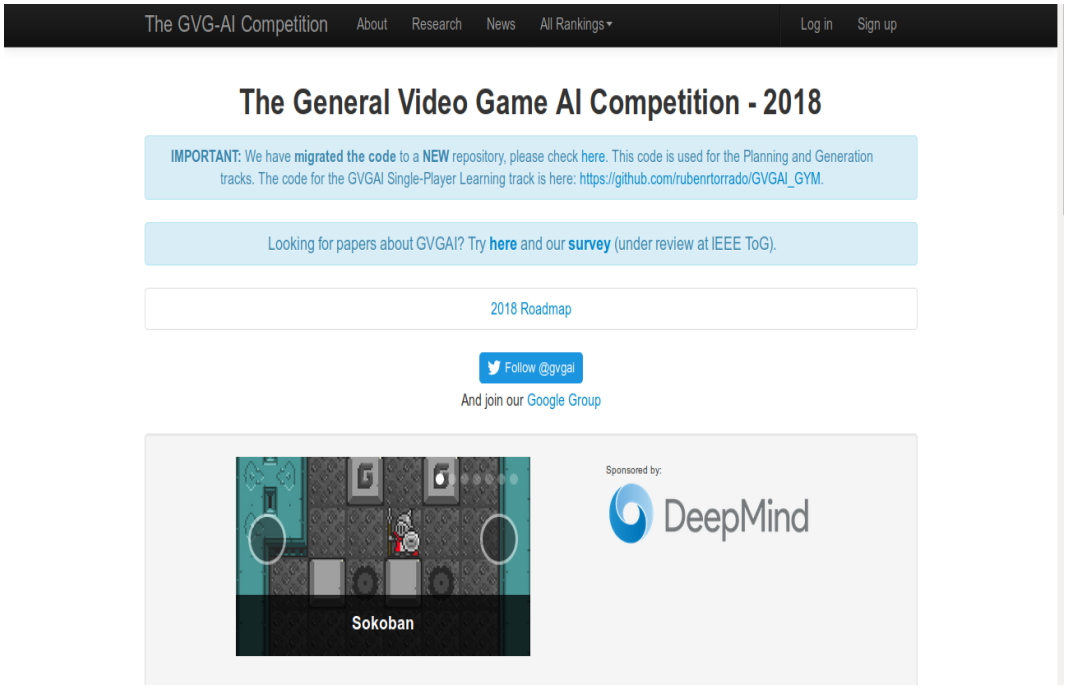



**Figure 3.1:** Snapshot of the GVG-AI competition main page

any loading or pre-processing task that it requires. In case of exceeding any of the two time limits (40ms to return an action and 1s. to perform pre-processing tasks), the agent would automatically lose the game.

Information about the games is given following an object-oriented representation. The framework uses the Video Game Description Language (VGDL) (see section 3.1.3) to define games in a more general way. Although the game description is available at the time of developing the game agent, the agent has not such information at execution time. This forms part of the GVG-AI rules, which determine that the agent has to face a completely unknown environment when playing a game. It is during the game playing that the agent discovers the game rules and the goals to achieve.

During game playing, the agent can obtain information of a game state like the position of the elements of the game (also called 'sprites'), the available resources, the score of the moves, etc. Also the framework provides a simulator to explore the different paths that come up after each possible move. With this information, the agent must discover the game mechanics and do its best to win the games.

### 3.1.1. Games and guidelines

In this project, we will work with a subset of games among the ones provided in the framework of the GVG-AI. We will deal with games that have a limited number of applicable actions, particularly a maximum of four directional actions plus one action for interacting with the game. The behaviour of this extra action corresponds to pushing a "fire" button and depends on the game; in some games this action is not permitted, in other games the agent is allowed to shoot, attack or create objects among other possibilities.

The type of games treated in this project are neither very complex nor very large-size problems. Hence, the application of search techniques enable to explore the search space of the game with an affordable computational cost. Nevertheless, although the search space may not have a very large branching factor, these games cannot be solved with

algorithms like BFS that explore the entire space to find the optimal solution but quickly exhaust the available memory.

More specifically, we handle games that deploy in a virtual world composed of a two-dimension grid where the position of the sprites (characters of the game) are represented via $x$ and $y$ coordinates, and there can be characters of different types in one same position. The main characteristics that define these games are:

- **Zero-sum games:** All of the games define an agent that attempts to win or to lose and so they are zero-sum games. In some cases a defeat will be given by the death of our avatar. This happens in games like *Aliens* where our battleship can be destroyed by an enemy laser shot. In other cases the game over is caused when the agent collides with a static element like in the game *Frogs*, where the agent dies if he falls into the water. In the rest of cases, the loss of game will be given by a time limit.
- **Two-player video games:** Most of games fall into this category in which the game agent represents the avatar controlled by the human. The game agent is confronted by a virtual opponent that we will refer to as the "computer", which controls the rest of the characters in the game and will attempt to make our avatar lose the game.
- **Stochastic games:** Every game in the two-player subset is a stochastic game where it is not possible to determine the direction in which the characters controlled by the computer will move. This is an important factor to take into account because the simulation made from the current state is only an approximation of the next steps of the game. Consequently, the actual future steps will not necessarily correspond with the result of the simulation even if when the executed action is the same as the simulated action.
- **Simultaneous games:** The set of two-player games is also characterized for being simultaneous games; i.e., the computer plays at the same time as our game agent.
- **Imperfect information:** The GVG-AI framework provides all past information of a game. In other words, there is no hidden information regarding the past actions of both players (the game agent and the computer). However, we do not have information about the rules of the game or the possible actions to be carried out by the computer. This is so because the aim of the competition is that the agent discovers the rules of the game and adapts to new environments.
- **Puzzle games:** Among the selected games, some of them are characterized as one-player games (puzzle games). Puzzle games are deterministic since the only actions that affect the state of the game are those performed by our avatar.

To summarize, the game agent to develop in this work will play two different types of games:

(a) Two-player simultaneous games, with stochastic transitions between movements.

(b) One-player deterministic games that we will refer to as "Puzzle games".

Once the characteristics of the collection of games used in this project have been described, we now explain the guidelines we followed to design and implement the game agent. In order to control our agent, we must first decide what the agent's objective is. There are two main possibilities:

(a) To seek a victory independently on how this is obtained.

(b) To maximize the game score by trying to achieve all the intermediate subgoals.

In this project we will evaluate the games following the first approximation and our objective will be to maximize the number of wins across the different games. For this purpose, the agent must decide what action should use at each game-step and our algorithm have to determine the best action to be applied. In order to do so, the framework allow us to make simulations at each step of the game and determine which action is the most profitable. We can make those simulations iteratively to explore large paths using searching algorithms with different heuristics. Additionally, we should take into account the non-deterministic behaviour of some of the games and obtain a safe route that minimize the possibilities of losing the game.

As mentioned before, we choose to use **combinatorial search** for developing the game agent. In order to explore the different combinations of movements we will use a search tree that represents the evolution of the game. A node of the search tree represents a state of the game, being the root node the representation of the current state at the each step at each step of the game. The applicable actions in each state generate the successor nodes of such state.

There are two possible ways of representing a game state:

(a) By reading the pixels of the screen, that is, using directly the game representation where each pixel has a RGB discrete value, or

(b) By using the game state representation provided by the GVG-AI framework.

Using the pixel representation would entail a very large number of variables, as many as necessary to account for the resolution of the game ( number of width pixel $*$ number of heigth pixels), and each variable would take on a value between $255 * 255 * 255$ Colors. Instead the GVG-AI framework provides a high level representation of the state. As a result a lower number of variables and values are needed, which increases the number of nodes that can be explored within a time slot. The representational scheme of GVG-AI is detailed in the following section.

In order to build the search tree we use the simulator supplied in the GVG-AI framework to execute a series of actions in a node and obtain the resulting states. We can use different approximations to measure the quality of a node:

(a) We can use the score of the game in the particular game state.

(b) We can infer the score from the specific characteristics of the game for example in *Super Mario*, as in many platform games the goal is to maximize the further point in the map minimizing the time to arrive to this point, for this reason the quality of a node in that kind of games could be measured with a function that maximize the furthest position the player can reach and minimize the time to reach this position.

In this project we decide to use game score to guide our search algorithm because it is a common property accessible in all GVG-AI games that provides us information about the objective of the game. It will be 0 or $> 0$ in games without reward where the score $> 0$ belongs to a state of the game where agent reach the victory, in rest of cases the score will be kept at 0. In games with rewards the score will be increased by obtain these rewards and it will be a clue that the agent is behaving correctly within the game.

With these guidelines we developed our algorithm capable of playing different games of the GVG-AI. But before explaining the method used we must understand how to interact with the game, as well as the mechanisms used to perform the search. In the following

section we will explain in more detail the simulator provided by the framework and the tools provided for it.

### 3.1.2. Simulator

The framework of the GVG-AI competition provides participants with an API that the controller can use to query the state of the game at every game cycle. This API allows the programmers to focus in logic implemented in agents. Programmers can use the API as a gateway between the logic and the mechanics of the game. The API provide some functions that allow the controller to extract information about the current state of the game. Besides we can make **simulations** about futures movements in order to find the best combination of actions to obtain our best score. One of the elements provided by the API is the StateObservation Object. This element provides information about the game (score, winner, game tick, available actions). This attributes of the current state of the game can be asked by means of some of the next functions[3]:

**Queries to the State**

- Advance: The framework includes a Forward Model that allows the agent to advance the game state (by simulating that an action is taken). This function updates all entities in the game.
- Copy: This function allow to make a copy of the actual so we can keep intermediate copies of the state and expand them without the need to perform the complete simulation from the initial state.

**Queries to the Game**

- getGameScore: This methods is used to obtain the score of the game in a concrete observation. Thanks to this it is possible determine if an action is increasing, decreasing or not affect to the game reward.
- getGameWinner: Indicates if there is a game winner in the current observation.
- isGameOver: Indicates if the game is over or if it hasn't finished yet.

**Queries to the Avatar**

- getAvailableActions: Allow the agent to know which actions are available in the provided game.

  That set could vary throughout the games too. For example in *Plants vs zombies*, which we can see a screenshot in the image 3.2, the agent has the four actions of movement plus the action button with which it installs plants to defend itself, meanwhile in *Space invaders* agent only can displace in two directions and can also use the action_use action to shot and kill its enemies.

  The following are the set of possible actions in each game:

  - ACTION_NIL: It's a reserved action used by the system when the agent do not choose an action in the time provided previously to disqualify.
  - ACTION_UP: Mainly permit to the agent to move or to orientate upwards.
  - ACTION_LEFT: Mainly permit to the agent to move or to orientate to the left.
  - ACTION_DOWN: Mainly permit to the agent to move or to orientate downwards.

**Figure 3.2:** Screenshot of games *Plants vs zombies*

  - ACTION_RIGHT: Mainly permit to the agent to move or to orientate to the right.
  - ACTION_USE: It is the most versatile action. Its result depends on each game although in many of the games it is not operative.

- getAvatarResources: Returns the resources in the avatar's possession. As there can be resources of different nature.

**Queries to the Observations**

- getObservationGrid: With this query we can obtain the map representation of the game. This method provide information about the sprites positioned at each cell in the game.

  The sprites can be the different kinds:

  - Non-player-characters (NPC): It's composed by that characters that interfere in the game excepting the principal character that is controlled by us. They could be enemies that try to hurt us or sprites that we should defend.
  - Immovable: Formed by all those sprites that had a fixed position in our grid. Mainly contain sprites that form the ground, the walls, obstacles and other elements that remain static throughout the course of the game.
  - Movable: Included in this group are those elements that can be moved and are not included in the NPC set. They can vary greatly depending on the game, but include bullets, boxes that can be moved by our avatar, and many others.
  - Resources: That list contain the resources that can be used by our avatar.
  - Portals: In this list are included the positions of that elements in the game able to transport us between distant positions of the game, as well as to create or to make disappear other sprites.

  Also there are queries for each kind of sprite. Agent can ask about one determined type of sprite.

Sprites have different behaviors depending of the game and could vary a lot depending of game. It made almost impossible to extrapolate rules for play in all games. Being

necessary to use strategies that allow discover the mechanics of the game and find the best actions in each step.

### 3.1.3. Video Game Definition Language

Thanks to the simulator, we can visualize and play all the available games. That is possible thanks a common language to be created, represented and interpreted for our simulator. The Video Game Definition Language (VGDL) is the language chosen by this task. It's a language design by Tom Schaul, originally implemented by him in Python using py-game.

It's a simple, high-level description language for 2D video games that permit a library to parse and play those games. The streamlined design of the language is based on defining locations and dynamics for simple building blocks, and the interaction effects when such objects collide, all of which are provided in a rich ontology [26].

To define a game we need crate two text files, on the one hand the Game Description file define all the elements and the interaction between them. On the other hand, the level description, fix the initial state. This file defines the characters that will appear in the game and their positions in the grid.

```
BasicGame
  LevelMapping
    G > goal
    + > key
    A > nokey
    1 > monster
  SpriteSet
    goal   > Immovable color=GREEN
    key    > Immovable color=ORANGE
    sword > Flicker limit=5 singleton=True
    movable      >
      avatar     > ShootAvatar stype=sword
        nokey    >
        withkey > color=ORANGE
      monster    > RandomNPC cooldown=4
  InteractionSet
    movable wall   > stepBack
    nokey goal     > stepBack
    goal withkey   > killSprite
    monster sword > killSprite
    avatar monster> killSprite
    key  avatar    > killSprite
    nokey key   > transformTo stype=withkey
  TerminationSet
    SpriteCounter stype=goal    win=True
    SpriteCounter stype=avatar win=False
```

**Figure 3.3:** Game description of zelda in lenguaje VGDL.

The Game description is simply a text string file with four differenced blocks:

- The LevelMapping permit read the level description to generate the initial state, transforming each character in the corresponding sprites.

- The SpriteSet section defines all the ontology of sprites. Using nested indentations we define the hierarchy where nested elements share the objects properties of their parents. We can augmented the definition of a sprite class adding keywords options in its definition.

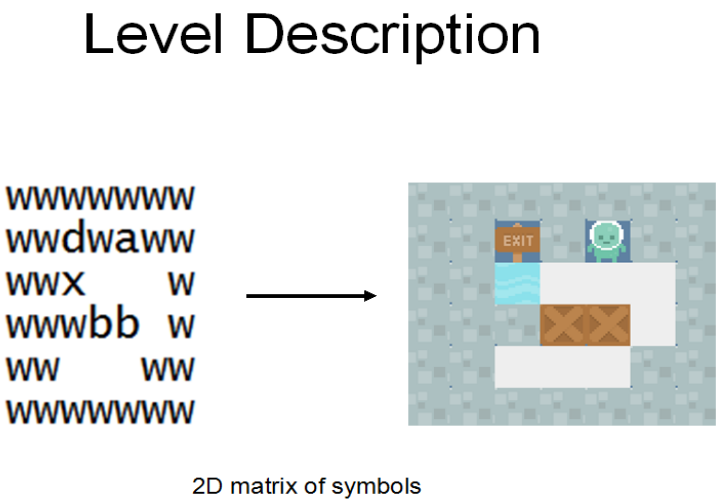



**Figure 3.4:** Level description file of VGDL game and its transformation on the simulator representation.

- The InteractionSet governs the transitions between states, especially in relation to collisions between sprites. Here we define which sprites are destroyed or created or when a sprite can't move to a position grid for be occupied by another sprite.
- The TerminationSet defines different ways by which the game can end. Each termination instance defines if game end with our victory or in a game over.

And example of game description of game *Zelda* is seen in the figure 3.3

The Level description file its a text a lines for each row of the grid Map similar to the left image of figure 3.4. Each line is composed for a number of chars equal to the number of columns in map. Each char represents one element of the LevelMapping and decide which sprite is located in that cell of the grid. An example of this representation is seen in the right image of figure 3.4. One game can have different levels each one represented by one level description File. The level executed will be one parameter for the framework.

## 3.2 Automated planning

Automated planning in Artificial Intelligence (AI) can be defined as the art of building control algorithms for dynamic systems. More precisely, a planning task is a search problem whose purpose is finding a set of actions that leads the system to an objective state from a given initial situation. The vast majority of approaches model planning as a single-agent procedure, in which a single entity or agent carries out the entire search process, developing the complete course of action to solve the task at hand.

This section analyses the principal elements of a planning task. First we define the most important planing concepts than make possible to understand the main paradigms in planning. Then we define the most relevant planning paradigms, paying special attention to the state-based planning approach, which is the paradigm our game agent draws upon.

### 3.2.1. Planning concepts

Single-agent planning is a search process in which starting from an initial situation, the agent has to find a plan or course of actions that allows it to reach a final state that includes the goals to achieve. In a classical planning model we adopt a series of assumptions to reduce the complexity of the problem and to be able to define its components more easily.

- The world is represented through a finite set of situations or states.
- The world is fully observable. In other words, the single agent has complete knowledge of the environment.
- The world is deterministic; that is, the application of an action can only generate a single other state.
- The world is static, the state of the world does not evolve until an action is applied.
- The planner handles explicit and immutable goal states.
- The planning process is carried out offline, so a planner does not consider external changes that occur in the world.

We will now detail the most relevant components of a planning task. A state is represented by a set of instantiated state variables named literals. The literals reflect those characteristics of the world that are interesting for the task at hand. The states of the world change through the application of the planning actions. Actions define the conditions that must hold in the world for an action to be applicable and the effects that result from the application of the action. Conditions are statements quering the value of a variables and effects are statements assigning a value to a variable.

**Definition 3.2.1.** *Action.* An action is a tuple $\alpha = PRE(\alpha) \rightarrow \{ADD(\alpha), DEL(\alpha)\}$, where $PRE(\alpha)$ is a set of literals describing the preconditions necessary for $\alpha$ to be applied, $ADD(\alpha)$ is the set of literals added to the state once the action has been applied and $DEL(\alpha)$ is the set of literals deleted.

Given a world state $s$, the set of all actions whose preconditions satisfy in $s$ form the set of applicable actions in $s$. The application of an action $a$ in state $s$ will generate a new state $s'$ as the result of adding the "ADD" effects of $a$ in $s'$ and deleting the "DEL" effects from $s$.

**Definition 3.2.2.** *Single-agent planning task.* A single-agent planning task is a tuple $T = \langle I, A, G \rangle$. I is a state that represents the initial situation of the world. A is a set of actions that can be applied by the planning agent to solve T and G is the goal state we desire to reach.

**Definition 3.2.3.** *Solution plan.* A solution plan for a task T is a sequence of actions $\{\alpha_0, \cdots, \alpha_n\}$ whose application over I leads to a state S, where $G \subseteq S$

An important aspect in planning is how to represent all the components of a task with a compact and expressive language. One of the first planning languages is STRIPS (STanford Research Institute Problem Solver), which has influenced most of the existing planners. STRIPS is a compact and simple language that allows the specification of planning domains and problems.

Despite its advantages, STRIPS has some expressive limitations that make it difficult to describe some real problems. As a result, many extensions have been developed over the past years, enriching its expressiveness and simplifying the definition of planning domains. One of these extensions is Planning Domain Definition Language (PDDL), the standard language used in the International Planning Competitions (IPC) within the planning community. When designing a PDDL problem, we need to define two separate blocks. On the one hand we must define the **domain** that includes the rules that govern the world of the problem. On the other hand we must define the **problem**, that entails to expose a particular situation within the domain previously described. Into the problem we must specify the initial state, as well as the objectives to be solved. The domain describes the general features of a particular domain, such as the types of objects, the predicates that describe situations of the world and the operators that can be applied by the planning entity to solve the task. The problem block models the specific details of the task, such as the actual objects in the world, the initial situation of the task and the goals that must be achieved in order to solve the planning task.

An example of PDDL code can be seen in the figure 3.5 where is defined an action in domain block of game *Sokoban*.

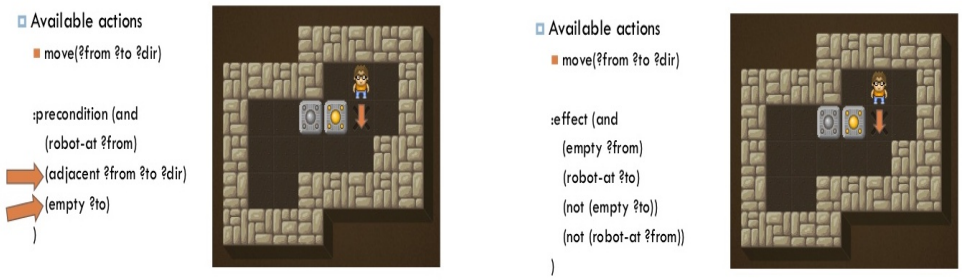



**Figure 3.5:** Action move in game sokoban

### 3.2.2. Planning paradigms

Single-agent planning systems are usually classified according to the search space they explore and the direction of the search. The next subsections describe in detail the most important planning paradigms.

#### State-based planning

In State-based planning, world is described through a finite number of states and define a plan as a sequence of actions whose application over the initial situation of the world leads to a certain state.

Most state-based planners are forward search algorithms, they start the construction of the plan in the initial state and move forward using available actions to the final state. An heuristic is usually used to guide the search. A function classifies states according to their desirability and the next state is selected according to this ranking.

Some of the most important state-based planners are:

- The **Heuristic Search Planner (HSP)** is one of the first state-based systems which uses domain-independent heuristic search. The additive heuristic of HSP is defined

as the sum of costs of the individual goals in G, where the cost of a single atom is estimated by considering a relaxed planning task in which all delete lists of the actions are ignored.

- The **Fast Forward (FF)**planning system is one of the most influential approaches to state-based planning. It uses the relaxed plan heuristic $h_{FF}$ , which is defined as the number of actions of a plan that solves the relaxed planning task. FF works with a Enforced Hill Climbing search, that is searching exhaustively nodes with a better heuristic value.

- **Fast Downward (FD)** is a heuristic-based planner that uses a multi-valued representation for the planning tasks. FD use SAS+ [13] to model the facts that conform states. Each variable has associated a Domain Transition Graph (DTG). In this structure is reflexed the evolution of that variable according to the actions of the task. DTGs are used to compile the Causal Graph in which are reflected the dependencies between different state variables. FD sue a best-first multi-heuristic search alternating $h_{FF}$ and $h_{CG}$ a heuristic inferred of the Causal Graph

- **LAMA** satisficing planner apply landmarks to improve the accuracy of the heuristic search. A landmark is a fact that holds at some point in every solution of a planning task. LAMA is based in FD pllanning but reuses the multi-heuristic search strategy of FD to alternate a landmark-based estimator and a variant of the $h_{FF}$ heuristic.

### Partial-Order Planning

Partial-Order planners (POP) works over all the task goals simultaneously, maintaining partial-order relations between actions without compromising a precise order among them.

In a pop system, the search is build as a search tree in which each node represents a partial plan. It begin the planning task with the goals, and build the solution plan backwards. The concrete order of actions is only established when it is necessary to ensure the objectives.

A partial order plan is formed by:

- A set of nodes where each node represents a partial state where some literals are defined.

- A set of order relations $S_i < S_j$ where $S_i$ should occurs in some moment previously to $S_j$

- A set of variable's restrictions, e.g. X=A.

- A set of casual links $S_i \xrightarrow{c} S_j$ that means that $S_i$ obtain c for $S_j$. They also represent order relations between nodes. Furthermore, casual links protect preconditions. They are not allowed if the action remove the preconditions of next action.

A solution plan should be complete and consistent.

**Definition 3.2.4.** Complete plan. It is a plan in which each operator's precondition must be satisfied by another operator.

**Definition 3.2.5.** Consistent plan. It is a plan where there is not contradictions in order restrictions or restrictions on variables.

When an action destroys the effect of a bond, it constitutes a threat that must be solved by an ordering constraint. That action should be executed after or before the actions involved in the link. This take the name of promotion or demotion respectively.

Once all preconditions have been propagated from the final state to the initial state and all threats have been solved, we have the partial plan solution.

A partially ordered plan corresponds to a set of fully ordered plans. Therefore, we only need to obtain a total order plan from the solution by adding the order restrictions necessaries.

Figure 3.6 shows an example of resolution of Sussman Anomaly [30]. Sussman Anomaly is an classic example of planing task into the world blocks in which two goals collided and it is impossible achieve both objectives without first eliminate some preconditions of one objective.

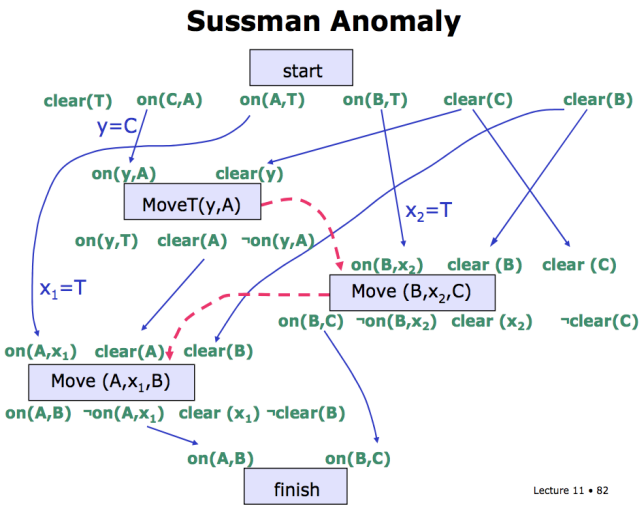



**Figure 3.6:** Pop solution of Sussman Anomaly

### Planning graph

It consists in relaxing the problem to obtain all the possible plans up to a pre-established length. This technique uses a novel planning graph to reduce the amount of search needed to find the solution from a straightforward exploration of the state space graph.

In this graph, the nodes are possible states and the edges indicate the reachability through a certain action.

Commonly the solution of this relaxed problem is used to guide the search process. By using graphplan we can see the goals that can be reached, pruning as many of them as possible thanks to incompatibility information.

Figure 3.7 shows the resolution of the Sussman anomaly problem with planning graph.

### Hierarchical Task Network

Hierarchical Task Network (HTN) is planning paradigm that solves a planning task by applying a successive goal decomposition. A solution to an HTN problem is then an executable sequence of primitive tasks that can be obtained from the initial task network by decomposing compound tasks into their set of simpler tasks, and by inserting ordering constraints.

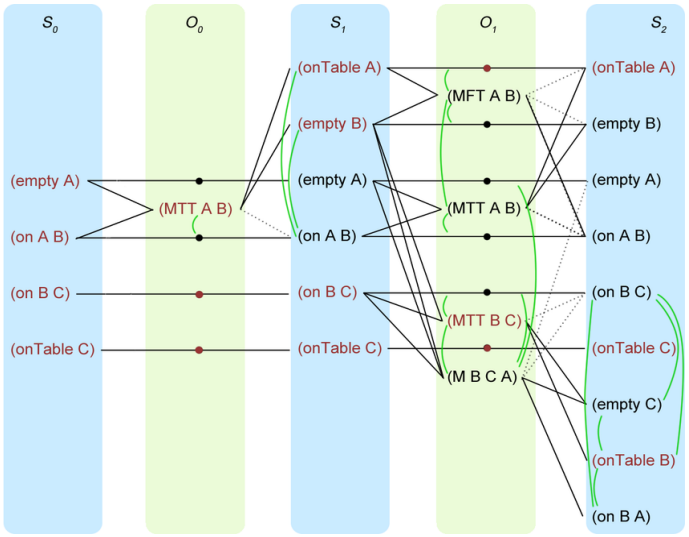



**Figure 3.7:** Planning graph solution of Sussman Anomaly

Constraints among tasks are expressed in the form of networks, called (hierarchical) task networks. A task network is a set of tasks and constraints among them. Such a network can be used as the precondition for another compound or goal task to be feasible.

The input of an HTN planner includes a set of actions and a set of methods to indicate how a task can be decomposed. Hence, HTN progressively decomposes tasks until only primitive or executable actions remain.

## 3.3 Iterative-Width algorithm

The idea of *search for novelty* is first introduced in the work [19] as a search technique that ignores the objective of the search and searches for behavioral novelty. Specifically, a *novelty search algorithm* searches with no objective other than continually finding novel behaviors in the search space. Yet because many points in the search space collapse to the same point in behavior space, it turns out that the search for novelty is computationally feasible.

*Iterative-Width* (IW) is a novelty-based pruned breadth-first search (BFS) that uses a set of **atoms** (i.e., pairs of state variables with their corresponding associated value) to represent a state and prunes states that do not satisfy a given **novelty** condition.

In IW, a state is composed of a set of **state variables**.

$$V = \{v_1, v_2, \ldots, v_N\}$$

Each state variable $v_j \in V$ has a finite and discrete **domain** $D_{v_j}$ that defines the possible values of that variable. A **state** is a total assignment of values to the state variables

The **IW(i) algorithm** is an implementation of a standard BFS starting from a given initial state $s_0$, but that prunes any state that is considered not *novel* where the novelty condition is defined as follows:

**Definition 3.3.1. State novelty.** When a new state $s$ is generated, IW($i$) contemplates all $n$-tuples of atoms of $s$ with size $n \leq i$. The state is considered *novel* if at least one tuple has not previously appeared in the search, otherwise the state is pruned.

Assuming that the $N$ state variables have the same domain $D$, IW($i$) visits at most $O((N \times |D|)^i)$ states. A key property of the algorithm is that while the number of states is exponential in the number of atoms, IW($i$) runs in time that is exponential in only in $i$. In particular, IW(1) is linear in the number of atoms, while IW(2) is quadratic. IW($i$) is then a blind search algorithm that eventually traverses the entire state-space provided that $i$ is large enough.

To understand the algorithm, let us present as example a simple search task. The objective of the *counters problem* is defined as finding a given number with a predefined number of counters. It is a simple search task that allows us to explain the different concepts of the IW algorithms and illustrate the potential of these algorithms.

In the *counters problem* the state variables are integers numbers each representing the value of a counter. For example, if we have 3 counters, we will need 3 variables $x_1$, $x_2$ and $x_3$. The initial state is by convention the situation where all counters are fixed to 0, in our example, $x_1 = 0$ , $x_2 = 0$ , $x_3 = 0$. The goal or final state will be to reach a predetermined number. For our example goal will be $x_1 = 3$ , $x_2 = 3$ , $x_3 = 3$. To transit between different states we define a function per counter that increments the counter value in one.

$$f_1 \rightarrow x_1 \mathrel{+}= 1 \ , \ f_2 \rightarrow x_2 \mathrel{+}= 1 \ , \ f_3 \rightarrow x_3 \mathrel{+}= 1$$

In IW($i$) atoms are a subset of variables of size $i$ with an specific value; for example in the initial state of our example task, there are three true atoms of size 1 $\{(x_1 = 0), (x_2 = 0), (x_3 = 0)\}$. For IW(2) there are three atoms of size 2, $\{(x_1 = 0, x_2 = 0), (x_1 = 0, x_3 = 0)(x_2 = 0, x_3 = 0)\}$ and so on. The number of possible atoms increases exponentially in function of $i$, for IW(1) there are 30 possible atoms whereas for IW(2) the number raises to 300 and for IW(3) reaches 1000 atoms.

Figure 3.8 shows the trace of the IW(1) in our *counters problem* example. *Nodes in red* are pruned because they do not satisfy the novelty condition. That is, other nodes of the tree previously discovered the atoms that appear in the pruned node. For example, the first red node has three atoms $x_0 = 1$, $x_1 = 1$ and $x_2 = 0$; the first atom of this node appears in the parent node, the second atom appears in the second node of previous level, and the last atom comes from the root node. Hence, the node $\{x_0 = 1, x_1 = 1, x_2 = 0\}$ does not provide any new atom and does not satisfy the novelty condition for IW(1). *Green nodes* represent atoms that satisfy an individual goal (we remind that the goal of the problem in our example is to reach $\{x_0 = 3, x_1 = 3, x_2 = 3\}$). We can observe that IW(1) always finds individuals goals in an optimal path.

Since we are not able to solve a problem with IW(1), we would increment the width (number of parameters) and come up with more complex goals. IW(2) will account for two simultaneous goals at the cost of augmenting the number of produced nodes. And IW(3) would solve the complete problem achieving the goal of the problem; i.e., the number 333.

The same problem can be solved by running IW(1) until one objective is achieved, and then starting the algorithm again in that state to find the subsequent objectives (because in this particular problem goals are serializable). The first execution allows the IW(1) algorithm to reach the goal $x_0 = 3$ as in the Figure 3.8. Starting from the first green node, the second goal will be reached in the second iteration. And the last iteration will eventually achieve the global goal. With this serial implementation of the IW(1) algorithm, we can achieve the given three objectives in one same execution. This is possible provided that the objectives of the problem are independent. That is, when the achievement of one objective does not interfere in the accomplishment of another objective. In the case that the objectives are dependent to each other, it will be needed to increase the width level

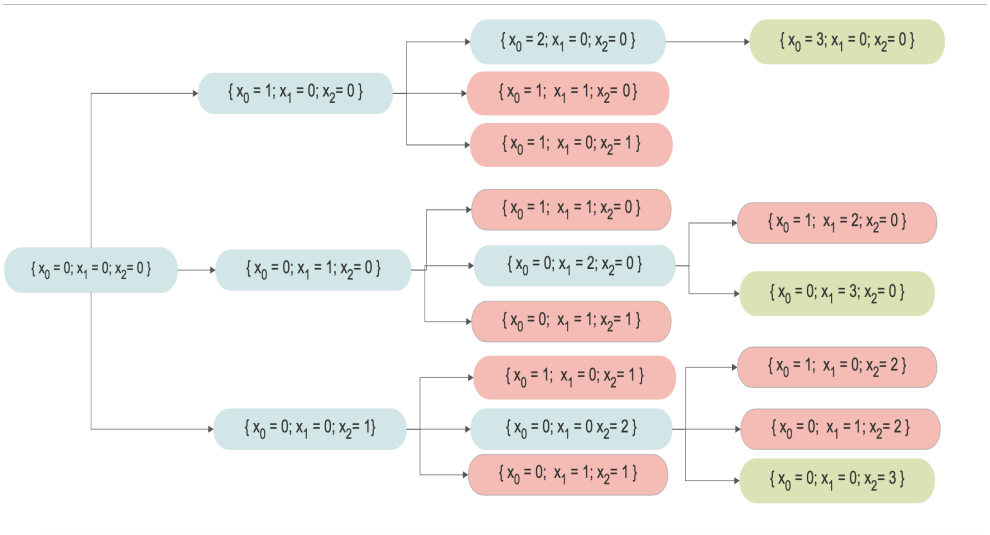



**Figure 3.8:** Trace of IW(1) to the *counters problem*.

of the algorithm to achieve them. For instance, in the game *Bait* game, it is necessary to first grasp a key in order to open a door. With IW(1) we will be able to reach the key or the door, but we may not achieve the two objectives. However, if we try first to reach the key, we will succeed in the second objective, opening the door.

To summarize, by using IW($i$), we can obtain as many objectives as the value of the width equal to the $i$ parameter. On the other hand, with a serialized implementation of IW we can reduce the $i$ param to achieve independent objectives.

# CHAPTER 4
# New novelty-based technique for videogames

As commented in Chapter 1, the aim of this research project is to develop a game agent in the context of GVG-AI using a search technique that draws on a new notion of width. In this chapter, we will explain the design and implementation details of our game agent based on an Iterative Width (IW) algorithm.

## 4.1 Abstract representation of the game states

This section details the representation of the state of a game as an abstraction of the game screen. Specifically, we will explain the representation of states handled by the IW algorithm as well as the method used by the game agent to generate the nodes of the search space.

To choose this representation, we must remind that IW($i$) generates nodes proportional to the number of atoms and runs in time that is exponential in $i$ only. On the other hand, the time limit provided by the GVG-AI competition to execute an action is 40ms. Hence, we must find a balance between an accurate representation that would allow us a more exhaustive search, with a more abstract representation, which allows us to search further in the tree and perform more complex actions.

The controller uses the grid component provided by the GVG-AI framework to represent a state of the *game state observation*. This component divides the game screen in a grid of cells represented with $x$ and $y$ coordinates that denote the horizontal and vertical position of the cell, respectively. Each cell contains information about all the sprites that are positioned at this point of the map. For each sprite the component provide us the following information: an unique Id, the category of the sprite (if it is our agent, an enemy an static element.) and a type inside that category.

To define the atoms of states, one representation may be create a boolean feature for each element in each position in the map. That is make an atom true where an element with unique Id it is moved to a new position in the search exploration process. The problem of this representation is that the set of Ids change dynamically throughout the game being created and destroyed numerous sprites. As a result a big number of nodes are be created and this negatively affected the outcome of our algorithm.

Instead, we opt for a representation where a state is composed of a set of boolean features as the ones just representing whether an object of a certain category is in a given cell of the grid. This reduces the complexity of our node representation, generating a good

balance between precision in the state abstraction and the depth that can be achieved with that representation.

The figure 4.1 shows an example of representation for the *Survive zombies* game, where several atoms are displayed. A boolean variable in our IW algorithm is represented by a composition of $(posx, posy, type)$ along with its associated value (TRUE,FALSE). A state will be represented by an atom $(posx, posy, type)$ =TRUE if an element of type $type$ is located in position $(x, y)$. Otherwise the state will be represented by the atom $(posx, posy, type)$ =FALSE.


**Figure 4.1:** State representation

In addition, we need extra game information to control our algorithm. First, we extract whether the actual state is a winning state. This property is included as part of the goal of our IW algorithm and it is used as the stop condition for the algorithm to finalize. Another important property is the *game over*. We should include this property and penalize the algorithm when a path reaches a game over state. Finally, we also collect the *reward property*. This property does not form part of the IW algorithm but it is used to choose the best action when IW is not able to reach the goal of the game. The last property of the game extracted to implement our controller is the reward.

At each movement, the agent collect all previous information to generate the initial state of the IW search. Also, the agent re-starts the novelty table and begins the IW exploration. The agent uses the simulator to pull out the stats of the nodes and decomposes each state into its constituent atoms to upgrade the novelty table and pruning all states that not satisfy the novelty condition.

To collect this information the state observation provides a bidimensional array where each element of the array contains a list of all observations positioned in that cell. By iterating over this list, we obtain the character types of the cell. This allows us to update the novelty table and check the atom that provides the component of novelty, if this is the case, to continue exploring the branch.

## 4.2 Preprocess

As the *characteristic_extraction* class completely explores the states with all its sprites, we have included some methods that have allowed us to know the peculiarities of the framework. Including at this part a debugging task that allow us discover for example that when the agent dies change its position to the cell (0,0) or than some movable objects, apparently resources to be collected, disappear when our avatar collide with them without becoming part of the available resources of our avatar.

Another problem of the IW algorithm is that it makes a deterministic search simulation that may be an optimistic approximation in stochastic games of the GVG-AI framework and cause some unexpected outputs. To avoid this problem we implement a previous process to the IW simulation. We named to this process **look ahead** and consist in simulating the first steps of our avatar many times to take statistics of the danger of each action. Experimentally we conclude that a single step repeated 5 times avoid the game over in a computational time adequate to continue expanding the search tree.

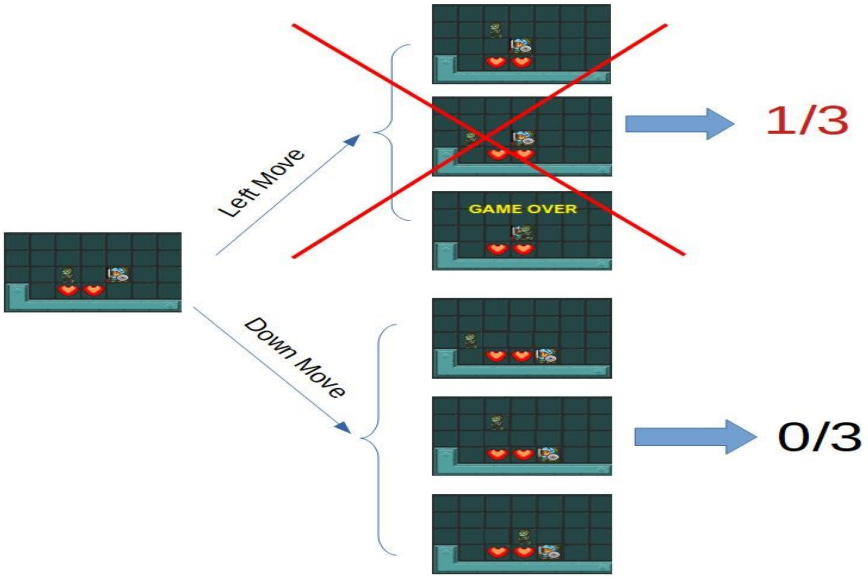



**Figure 4.2:** Look ahead of the first action in game survive zombies to prevent dangerous actions.

For illustration, we can see the figure 4.2 where it is observed that the left movement causes death once out of 3 simulations. While the downward movement, for example does not cause our death in any case. After the simulation made in the look ahead step, we would conclude that the movement to the left is less safe, so we would prune the corresponding part of the tree.

To conclude with the preprocessing methods, in this research is included a simple agent to understand better the games involved in the framework and to be able to classify them according to their similarities and differences. We developed an agent that recollects information about the characteristics of the game. This agent recollect information about which games have enemies that try to kill us, what games are puzzle types, in which games we complete the IW node expansion.... That information, recollected in the tables of appendix A allow us to understand better, which characteristics are necessaries in a game for the rightful behaviour of our algorithm. This table allows to analyze the characteristics of games where IW not produce the best results. That information will be useful to improve our algorithm and perform future researches where a metaheuristic based in this type of games will determine the best algorithm for use at each moment.

## 4.3 Iterative Width

Due to the little computational time available for action selection in the GVG-AI competition (40 ms) we consider only IW(1) because it just requires linear time in the number of atoms. Unlike heuristic search algorithms, IW algorithms do not exploit the goals to guide the search process (i.e. goals are only used as the conditions to finalize the search). Further, in our implementation, we will use rewards as an objective to maximize (since there is usually not enough time to conclude search in just 40 ms). More precisely, we will interleave IW(1) explorations to find the best possibles rewards, with actions chosen according to the maximum reward obtained with these explorations.

Next, we present our adaptations of the IW algorithm to build an agent for the games of the GVG-AI competition.

### 4.3.1. Basic approximation

Our first goal was to elaborate a simple agent based on Iterative Width capable of playing multiple GVG-AI games. To do so, we included the IW(1) search algorithm as the backbone of our agent, with these basic tuning:

1. The agent runs the **IW(1) search as an anytime function** at each moment to decide the next action to be executed. To this tend, an action selection function is needed that is based on choosing the action that contains the descendant with the highest score in the game.

2. A **tie-breaking mechanism** is added to the IW(1) search for two or more actions that have the same score. Our agent implements a random function over the actions available to guarantee that the same action is not ever chosen.

3. A **Look ahead** is implemented to make the agent avoid actions that may cause its death. Because the stochastic nature of some games, in a particular simulation the agent may not be dead after applying a dangerous action.

Now we describe in detail the behavior of our basic agent. In each step of the game, the *Basic Approximation* initializes the IW(1) novelty list to decide which nodes to prune for the IW(1) explorations. Given that the time limit per move in the GVG-AI competition (40ms) is very short, it is important an efficient implementation of the search loop. For this reason, we decided to implement the novelty table reserving a large space into a big boolean array with a dimension equal to the number of possible cells multiplied by the different types of characters, twenty types. This ensures that we check whether an atom has already been explored in a constant time.

Before starting the IW(1) exploration, the agent checks what actions are unsafe in the actual state. To do this, it makes use of the *look ahead* mechanism explained in the previous point. Safe actions are the only ones included in the search tree. Then, the agent begins the IW(1) exploration to determine the more desirable paths. The agent runs IW(1) as an anytime algorithm providing a time parameter in which the search must terminate. For this we use the 95% of the competition time to explore the nodes. The tree is explored as a BFS algorithm expanding completely the actual level of the tree, and subjecting each new node to the novelty analysis. If a node does not pass the novelty condition, it will be pruned. Each node is formed with the forward model provided by the simulator. To use it, we need the previous state and the action that we will simulate. Therefore, it is necessary to save in each node, either the current state, in order to generate new states, or

the action that caused it, in order to undo the path and generate the complete path that forms that state.

When the IW(1) time is exhausted, the best node is chosen and its parent root action is returned. In other words, the node with the best reward obtained in the simulator is the objective of our agent. To achieve that node we must back-propagate the node reward throw its parent to the root node. The action that has generated the branch that gives us the greatest reward is the result output by the IW(1) exploration.

We also introduced a mechanism for tie-breaking at games that have scarce rewards like *Sokoban*. Let's take as an example a fictitious game in which the score only changes when a victory is reached. Let's imagine that our avatar aims to reach a point located in the lower right corner, but far enough so that the exploration of IW(1) does not achieve it. In this example our avatar would expand the nodes with the movements in the following order: **left, up, right** and **down**. After exploring the map in the given time, IW(1) will not find an action with a reward higher than 0 so the agent will return the first explored action; that is, it will conclude that the best solution is going left. This conclusion is repeated until reaching the left margin. At this point the algorithm discards the left action for not being within the possible actions, so the best solution would be the next available action, i.e. moving upwards. As a consequence, the avatar will end up in the upper left corner in just a few seconds, finding itself stuck in a loop from which it cannot exit.

The solution to overcome this issue is simple: exploring the actions in a random order. This avoids to repeatedly choose the same action when all actions have the same reward, and thus avoid the agent making the same movements endlessly and wasting the time without exploring different parts of the map. We pre-compute some random steps, and reuse them in a loop of "random" movements.

Subsequently, we introduce enhanced versions of our *basic approximation* with the aim of improving its performance.

### 4.3.2. Reward Shaping plus Discount Factor

For our second version of the IW agent, we designed two techniques that will potentially overcome some of the limitations observed in the basic approximation. These improvements are:

1. **Reward Shaping**: the application of this technique will enhance the selection of nodes that account for subgoals of the game which are otherwise inaccessible with the basic approximation.
2. **Discount Factor**: the application of this technique will give more preference to the nodes closer to the root node which are scored with a high reward. Discount Factor is introduced with the purpose of achieving goals sooner, that is, with fewer actions.

These improvements are detailed below.

#### Reward Shaping

When playing games like *Sokoban* or *Portals*, we observed that some changes occur along the game playing that might be exploitable for improving the performance of these games. The idea lies in rewarding these changing situations, which go unnoticed in IW(1), with a higher score. To this end, we implemented a *Reward Shaping* function. This function returns a reward value which is calculated as a combination of the score obtained from the current state plus an increment if one of the following events occurs:

- **Type 1:** This type of event detects when movable objects of the game are removed, and this change is not reflected in the score of the node. This happens, for instance, in games like *Bait*, which is represented in Figure 4.3. In this game, the avatar grasps a key that is later needed to open a door, where the key is the movable object detectable in this type of event. When this happens, we increase the score of the node by one point.

- **Type 2:** This event detects when the type of the avatar changes. Some games like the *Real Portals* include different types of the main avatar, and these types are used to decide the results of the actions; for example, the action 'use' in *Real Portals* creates a different kind of portal depending on the avatar's type. A different usage of the avatar's type is to restrict access to some parts of the map for a particular avatar type. In this case, the avatar must eventually change its type in order to achieve the game objective. When the avatar changes its type, we increase the score of the node by five points.

The increments of one and five points for the events of Type 1 and Type 2, respectively, were experimentally chosen. We observed that events of Type 1 tend to yield more false negative results than events of Type 2.

Reward Shaping aims at improving the agent performance in complex games where a victory usually requires going through different stages, achieving a sub-goal at each stage. Our hypothesis is that there is more chance to reach a sub-goal when either an event of Type 1 or Type 2 occurs in the game. Reward Shaping gives more priority to the nodes in which these events will happen, so the search is guided by these events as well as by the score increments provided by the game.

One of the games that is altered by this improvement is the *Bait*, which is represented in Figure 4.3. In this game our avatar must reach a key needed to later open a door. The basic approximation of IW(1) will return the same reward for all actions since none is able to reach the key and get to the door before being pruned for novelty. The fact of not finding any reward in the search tree makes the agent choose the next action with a random criterion. With this random component, it might be possible for the agent to reach a state from which it is impossible to win the game. For example, at figure 4.3, the agent might randomly choose to execute action-down twice. The second time, the action would move the box over the key, being impossible to move the box to access the key at the rest of game. Since we cannot get the key, we cannot win the game. So we would have lost the game without any negative reward to indicate it. In contrast, with the Reward Shaping version, the agent will get a reward when he reaches the position where the key is located and this helps his actions be guided by this sub-objective.

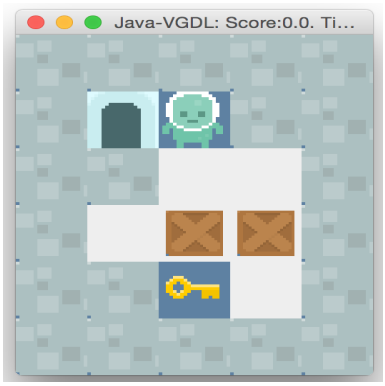



**Figure 4.3:** Screenshot of *Bait*

The games influenced by the Reward Shaping function are reflected in the tables of Appendix A with the value True (T) in the "Reward shaping" row.

### Discount Factor

To improve the performance of the game agent we included a discount factor to calculate the node score. The discount factor is used to penalize the nodes that are found at deeper levels in the search tree and favour rewards obtained in the upper levels of the search tree.

The Discount Factor penalty is used to discard, for one same objective, the paths that have more actions. That is, if we have to move to an adjacent left cell to get a reward is preferable to reach it with a single action (action-left) that with three actions (action-up, action-left, action-down). This can be seen in Figure 4.4 where the bottom node has a reward of 0.99 while the top node on the right has a reward of 0.97 because more actions are needed to reach this node. Minimizing the number of actions to reach one particular states gives the agent extra time to continue increasing the reward.

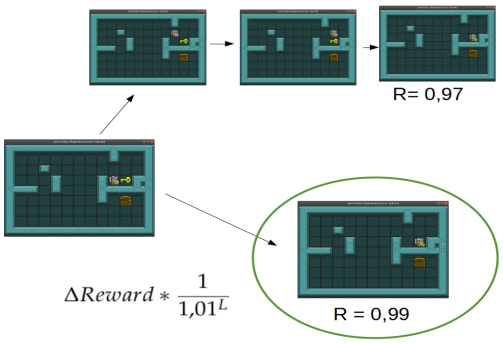



**Figure 4.4:** Example of discount factor function

Another reason for including this penalty stems from stochastic games where each action involves a certain random factor, which accumulates along a chain of actions. Because of this, deeper states are more unlikely than states close to the root. Thanks to the Discount Factor, the agent will prefer to reach close rewards that have more certainty of being reached rather than distant rewards which are more uncertain.

It is also worth noting that the chosen Discount Function should not penalize lower levels to a large extent. Otherwise, the agent will always opt for the closest rewards, forgetting the deeper levels of the search tree. The function chosen for our agent is:

$$\Delta Reward * \frac{1}{1.01^L}$$

where $L$ is the level of the node in the search tree. As an example, exploring a game with a node in the first level with an increment of 100 reward units will result in $100 * \frac{1}{1.01^1} = 99.01$ while considering a node in the ninth level that reports an increment of 105 reward units results in $100 * \frac{1}{1.01^9} = 96.01$ despite the reward is 5 points higher.

### 4.3.3. Iterative Width (3/2)

The last variant implemented in the IW algorithm is called IW(3/2), a version halfway between IW(1) and IW(2) where some *informative* atoms are handpicked to build 2-atom tuples.

In our case we decided to include together with the atoms chosen for the basic version some variables with which to form two-atom tuples. These *informative* variables are:

- The score of the game in the current state.
- The type of the main avatar.

In addition, we also contemplate whether a sprite of some kind is included in a cell using a combination of the type and the category of each element, which gives more precision to the abstraction of the state.

In the IW(3/2) version we consider a novelty state if it provides a new score or if it provides a new avatar type. This is particularly useful in games like *Real Portals* where it is necessary to change the avatar type because each type create a different kind of portal. At the game agent must create different types of portals for get a key and resolve the puzzle game. The IW(3/2) version allows the avatar to achieve complex rewards without increasing the complexity towards a quadratic algorithm like in IW(2).

This version uses tuples of two atoms to update the table of novelties. The tuples are formed by:

- One atom whose variable belongs to the set of variables used in the basic implementation. That is, the atom indicates if an element of a certain type is in a specific position.
- The other atom must contain an *informative* variable like the one described above (*score of the game* or *type of main avatar*).

Thanks to this enhancement, the agent can go back inside the search space once it finds a reward or when it reaches a position that makes it change its type. This additional advantage enables solving problems composed of these two properties, thus improving the operation of the IW(1) especially in those cases in which the IW(1) expands completely the search tree in the given time. These games are identified in the row "Ends exploration" of the appendix A.

In other words, the use of a pair of atoms to check the novelty condition makes more unlikely that a node will be pruned, so the search performs a more detailed analysis, which allows the agent to achieve goals out of range in the previous versions based on IW(1). The games affected by the IW(3/2) version are reflected in the tables of Appendix A with a value higher than 1 in the row "Avatar types". The other *informative* variable take into account for the IW(3/2) is the score of the game. Therefore, the number of times that the score changes during the game will have impact on the development of the search algorithm. That is to say, a game in which the score is not updated until the victory, will not be affected by this variable of the IW(3/2), whereas a game that updates the score continuously, will delay the pruning to a greater extent, contemplating many nodes that until now were not considered. All this information is reported in the row "Rewards" of the Apendix A where are reflected the number of different scores detected by the exploration agent throughout each game.

# CHAPTER 5
# Results

In this chapter we will present the results obtained by a game agent that implements our adaptations of the IW(1) algorithm. To this end, we have created a series of experiments to test our agent on various different games of the GVG-AI competition and we will discuss the strengths and weaknesses of our approach compared to baseline algorithms.

## 5.1 Game-set description

For the experimental evaluation, we used 30 games from three training sets of the GVG-AI competition. The GVG-AI site provides, for each game, description files associated to five game levels that feature different configurations of the initial state and a different difficulty level. We run 25 experiments distributed across the five game levels for every game.

The 30 games used in the experimental evaluation are split into three different game sets. Following we describe the general features of the games comprised in each of the three sets.

The games in **Game Set 1** are characterized for being a kind of reactive games that require agents endowed with reflexes and react abilities. This set was used as (the only) training set during at the Computational Intelligence and Games (CIG) competition at 2014. This game set include games like:

- *Aliens*: in this game the agent controls a ship at the bottom of the screen shooting aliens that come from space. The agent must dodge the bullets at all times while trying to eliminate enemy ships. When the agent kills an enemy, the game updates the score. This score represents a reward that tells the agent the progress made towards the global goal. A screenshot of the game is shown in Figure B.1a.

- *Frogs*: the objective of this game (represented in figure B.1e) is for the avatar (a frog) to cross a road and a river in order to reach the finish line. The path is full of obstacles; there are cars on the road that must be avoided, and trunks floating on the river which can be used by the avatar to hop over them and move along the river. There is no intermediate reward in this game, the only score outcome is at the end of the game.

The **Game Set 2** was used as the validation game set for the CIG 2014 competition. The games comprised in this set are very much alike the games of the Set 1 but slightly more complex. Hence, while the gist of the games in this set also lies in dodging obstacles

to reach a finish location, the games are usually more difficult than the games in Set 1 and require longer solution paths. Some examples of this games are:

- *Camel race*: in this game the avatar must get the finish line before any other camel competing in the race. The race consists of a long straight circuit, moving from the left side where the starting line is located to the right side where the finish line is. The raceway also presents some barriers that the agent must avoid. Unlike other games, the width dimension of the map in the *Camel race* is considerably enlarged (see Figure B.3a). Specifically, while the map dimensions in the rest of games are usually 12x26 or 12x31 squares, the map in the *Camel race* is 10x50 squares. This makes the distance between the finish line and the start to be significantly longer and so a longer solution path.

- *Firecaster* (B.1e): in this game the avatar must find its way to the exit by burning wooden boxes down. In order to be able to shoot, the avatar needs to collect ammunition (mana) scattered around the level. Flames spread, being able to destroy more than one box, but they can also hit the avatar. The avatar has health, that decreases when a flame touches him. If health goes down to 0, the player loses. When a box is destroyed the score of the game increases regardless how directly it contributes to a win. The agent will burn boxes all along until it finds the long path to the exit, when it will receive the highest reward.

The games of the **Game Set 3**, which was used for the 2018 CIG competitions, are all puzzles. Therefore this type of games are less reactive and more deliberative. They typically require more complex solutions, combining the achievement of several sub-objectives within the game. Among the games of this third set, we can find:

- *Bait*: in order to reach the goal of this game, the avatar must collect a key first. The player can push boxes around to open paths. There are holes in the ground that kill the player, but they can be filled with boxes (and both the hole and the box disappear). The player can also collect mushrooms that give points. There are not intermediate rewards in this game, a solution path to this game must include an action to pick up a key and actions to move along the map. A screenshot that shows an image of the first level of this game can be seen at B.5a.

- *Real portals*: the avatar must find the exit of the level. It is armed with a gun that allows to open portals in the walls of the level. Two types of portals can be opened (entrance or exit portals), and the avatar must change its weapon in order to open one type or another. The avatar can travel through these portals, shooting through them or pushing rocks. There can also be a key that opens locks on the way to the exit. Once the avatar has the key, it must create the portals to reach the exit. And all this while changing the weapons for each kind of portal. As we can see in figure B.6a the weapons are represented by red and green scepters, depending on whether they create entrance or exit portals.

## 5.2 Comparative analysis of the Basic Approximation with MCTS

In this section we will compare the performance of the Basic Approximation of our IW algorithm (see section 4.3.1) with a Monte Carlo tree search (MCTS). For this comparison, we used the MCTS implementation provided by the GVG-AI framework. As commented in Section 4.3.1, the basic approximation of the IW(1) algorithm includes the essentials of

the Iterative Width behaviour and allows us to compare with a MCTS method in similar terms. With this comparative analysis we expect to cover the main aspects of the algorithm, understanding its functionality as well as its strengths and weaknesses so as to be able to suggest later improvements accordingly to the results.

We define three different settings for all the games used in the experimental evaluation. The first setting gives the agent a 40 ms deadline to return an action as specified in the rules of the GVG-AI competition. In the other two settings, we relax the time limit and we set a deadline of 300ms and 1 sec respectively. Our purpose is to understand how the time limitation affects the functioning and performance of the algorithms.

### 5.2.1. Results for Game Set 1

The results obtained for the games in Set 1 are displayed in Table 5.1.

**Table 5.1:** Problems solved with MCTS and our Basic Approximation of IW(1) in Game Set 1

| Time | 40ms | | 300ms | | 1s | |
|---|---|---|---|---|---|---|
| Game | IW(1) | MCTS | IW(1) | MCTS | IW(1) | MCTS |
| Aliens | 25 | 25 | 25 | 25 | 25 | 25 |
| Boulderdash | 2 | 0 | 4 | 1 | 3 | 0 |
| Butterflies | 24 | 21 | 25 | 24 | 25 | 24 |
| Chase | 1 | 0 | 11 | 0 | 7 | 1 |
| Frogs | 18 | 5 | 25 | 0 | 25 | 4 |
| Missile Command | 25 | 15 | 25 | 15 | 25 | 19 |
| Portals | 16 | 4 | 20 | 3 | 21 | 6 |
| Sokoban | 4 | 4 | 7 | 11 | 8 | 10 |
| Survive Zombies | 11 | 11 | 13 | 11 | 13 | 12 |
| Zelda | 11 | 6 | 13 | 6 | 8 | 4 |
| Total | 137 | 91 | 168 | 96 | 160 | 105 |

Table 5.1 shows the number of problems solved in each game out of a total of 25 problem instances per game for both the MCTS and our basic implementation of IW(1). As we can see in the table, IW(1) clearly outperforms the MCTS in the three settings.

The IW(1) shows an outstanding behaviour in three games across the three settings, namely the *Aliens* (Figure B.1a in Appendix B), the *Missile Command* (Figure B.1f in Appendix B) and the *Butterflies* (Figure B.1c in Appendix B) games. In these three games, IW(1) is able to solve all of the problems except for one instance in the *Butterflies* game within 40ms, while the MCTS shows a weaker performance in the *Missile Command* and slightly lower than IW(1) in the *Butterflies*. These three games are distinguished for being highly reactive games where the avatar needs to react nimbly and early rewards help eliminate the enemies.

We can also observe that IW(1) is much better suited for the games *Frogs* (Figure B.1e in Appendix B) and *Portals* (Figure B.2a in Appendix B) than MCTS, which only wins in 20% of the executed problems. The purpose of both games is to find the finish line while moving along a path hampered by a number of obstacles that may cause the death of the game agent. In the IW(1) algorithm, the application of the novelty pruning allows the agent to expand through the grid map and to eventually find the finish line. In contrast, the behaviour of the MCTS algorithm is rather different. Since these games feature multiple paths where the avatar can die, the nodes in MCTS that explore the hazardous zones of the map receive a lower score than nodes that explore the safety

zones. The avatar needs, however, to get into the danger zones to be able to progress towards the finish line but then this results in a lower score and so the avatar backs off. This happens successively along the multiple simulations of the MCTS to update the score of a node, meaning the algorithm is stuck in a local minimum. This is the reason that explains the performance of MCTS is significantly worse in these two games.

There are, however, other games like *Chase* (Figure B.1d in Appendix B) and *Zelda* (Figure B.2d in Appendix B) where IW(1) is not able to get such good results as in the previous games but yet it performs better than MCTS. Both algorithms show difficulties in solving these two games because a rather long sequence of actions is needed to reach the goal. In the case of *Chase* this is due to the sprites controlled by the computer run off as rapidly as our avatar and hence the agent has to chase and corner them in order to get them caught. The complexity of *Zelda* stems from the fact that the avatar needs to apply two actions in order to move in one particular direction: a first rotation movement to point at the right direction and a second action that effectively moves the avatar. The rotation requirement increases the number of action combinations thus augmenting considerably the size of the search tree. Despite the complexity of these two games, the superior performance of IW(1) is due to the novelty pruning, which enables focusing the search toward deeper nodes (objectives) of the tree.

Conversely, the MCTS exhibits a better performance in the *Sokoban* game, solving four more problems when running the algorithms with a 300ms deadline and two more problems under a 1 sec deadline. A solution to this game involves complex actions that are likely to be found in branches of the tree which IW(1) prunes because they do not bring enough novelty when only 1-atom tuples are considered. The simulation of the MCTS algorithm, however, is able to get rewards that suitably focus the search in the right direction towards the goal.

It is also worth noting that IW(1) reaches the sweet spot in the 300ms setting solving 168 problems. While giving additional time to IW(1) does not bring an improvement in terms of total problems solved, the MCTS gets to solve 9 more problems (from 96 solved instances to 105) with a 1 sec deadline. The reason why IW(1) does not get to solve more problems when the deadline is extended to 1 sec is because the strong novelty pruning largely reduces the size of the search tree, which implies the algorithm already exhausts the search space when using only 300ms. In contrast, a longer time favours MCTS because it can run more simulations and obtain more accurate predictions for each node.

### 5.2.2. Results for Game Set 2

The results obtained for the games in Set 2 are displayed in Table 5.2.

When comparing the results of the Game Set 2 in Table 5.2 with the results of the Game Set 1 in Table 5.1, we observe that the number of wins for Game Set 2 is noticeable lower for both algorithms. This is due to the games of this second set generally involve more complex solutions that require searching deep down into the search tree. For example, in the 40ms setting, IW(1) wins 137 games in Game Set 1 to 102 wins in Game Set 2. A similar trend is observed in all the settings and for both algorithms, IW(1) and MCTS.

Due to the higher complexity of the games in Game Set 2, we also observe another difference with respect to Game Set 1. While the sweet spot of IW(1) in Set 1 is reached with 300ms, the sweet spot in Set 2 occurs when the deadline is set to 1 second. This means that more time benefits IW(1) in more complex games. Specifically, the increment in the number of problems solved when switching from 300ms to 1sec is as follows: MCTS

**Table 5.2:** Comparing MCTS with our basic implementation of IW(1) in Game Set 2.

| Time | 40ms | | 300ms | | 1s | |
|---|---|---|---|---|---|---|
| Game | IW(1) | MCTS | IW(1) | MCTS | IW(1) | MCTS |
| Camel Race | 2 | 2 | 25 | 3 | 25 | 1 |
| Digbug | 0 | 0 | 0 | 0 | 0 | 0 |
| Firestorms | 9 | 1 | 18 | 4 | 19 | 5 |
| Infection | 24 | 25 | 25 | 25 | 24 | 25 |
| Firecaster | 0 | 1 | 0 | 1 | 0 | 0 |
| Overload | 15 | 6 | 12 | 3 | 15 | 7 |
| Pacman | 0 | 0 | 0 | 0 | 0 | 0 |
| Seaquest | 13 | 25 | 18 | 24 | 18 | 24 |
| Whackamole | 21 | 24 | 18 | 23 | 21 | 25 |
| Eggomania | 18 | 0 | 18 | 3 | 19 | 5 |
| Total | 102 | 84 | 134 | 86 | 141 | 92 |

solves 8.5% more problems in Game Set 1 and 6.5% more in Game Set 2 while IW(1) goes from -4.7% in Game Set 1 to 5% in Game Set 2.

Analyzing the particularities of both algorithms in the games, we observe in Table 5.2 that IW(1) solves only two problems in the game *Camel Race* (Figure B.3a in Appendix B) with a 40ms deadline but gets to solve all of the instances when more time is given. Again, this is an indication, as commented above, that more time largely favours IW(1), which is the case for almost all of the games when the deadline is 1sec.

On the other hand, the MCTS algorithm performs very poorly in the *Camel Race* across the three settings. As we mentioned before, the width of the screen in this game is significantly larger than in other games (about 50 squares between the finish line and the starting line) so the goal line is found at a depth level 50 of the search tree and the search space comprises $4^{50}$ possible combinations to reach the goal. On top of that, since there are no intermediate rewards in this game, the agent plays randomly. This explains that the MCTS simulations will rarely achieve a victory; only in those cases where the camel approaches by chance close enough to the goal it will be capable of reaching the finish mark. In contrast, the IW(1) algorithm expands linearly across the grid map generating a number of nodes similar to the number of squares on the map.

Table 5.2 also shows that MCTS is superior to IW(1) in two games, the *Seaquest* (Figure B.4c in Appendix B) and *Whackamole* (Figure B.4d in Appendix B). The game mechanics in these two scenarios is similar; the avatar is chased by enemies who may cause its death, and while the avatar dodges his enemies, it picks up items that increase the score in spite of this putting him in danger. IW(1) is more prone to pick up the items and raise the score of the node despite the existence of enemies nearby who can kill the avatar. However, the multiple simulations of the MCTS in a node reveal the probability that the avatar dies if a particular action is taken; this value is backpropagated to the ancestor nodes, penalizing the branches that statistically show a higher probability of dying even though reporting a raising reward. In other words, the simulations of the MCTS return an accurate estimate on how dangerous an action is thanks to the backpropagation carried out by the descendant nodes which penalize enormously the dangerous actions despite the achievable reward. In contrast, the one-action lookahead of our basic approximation of IW (explained in section 4.2) only prevents the agent from dying if death occurs in the next action, which is clearly not enough in some cases. These arguments explain the superior behaviour of MCTS in the games *Seaquest* and *Whackamole*.

Table 5.2 also shows that both algorithms perform very poorly in games such as *Digbug* (Figure B.3b in Appendix B), Pacman (Figure B.4b in Appendix B) or *Firecaster* (Figure B.3e in Appendix B) where the result for both controllers is close to 0. These are complex games whose solutions involve linking several actions together and are found at a deep level in the search tree. Additionally, the score that the player obtains in these games is not an indication of how close the agent is to a win. The score is a positive reinforcement which helps the agent understand the mechanics of the game and be aware of how well it is doing so far but not a value that shows how easy/difficult is for the agent to win the game. Indeed, an agent can be highly rewarded after executing an action even if this action guides the agent to a dead-end node. In short, the reward in these games is not necessarily correlated with progressing towards the goal and the complexity of the solutions implies that not every action is a good move towards winning the game.

Consequently, due to the lack of helpful information to advance to the goal, the search process may select actions that increment the score and makes the agent fall in a local maximum from which it can not exit. Thereby when the action selected by the agent achieves a sub-objective that increases the score but does not imply a step towards winning the game, the agent may end up either eventually been killed by the enemies (dead-end node) or stuck in a local point of the game. In this latter case, the only thing the agent can do is to play randomly while waiting for the deadline to expire and the game officially declared to be over.

### 5.2.3. Results for Game Set 3

The results obtained for the games in Set 3 are displayed in Table 5.3.

This set comprises the most difficult games, all of them puzzle games that require more elaborated solutions. Unlike the other two sets, selecting an action in a game of Set 3 involves a more sophisticated reasoning process. Let's see this with an example of the *Real Sokoban* game represented in Figure B.6a in Appendix B. The goal of this game is to push all boxes to the circle-shaped targets or platforms. In the particular situation shown in Figure B.6a, the only action that leads the player to a victory is to move the box on the left because any other action will eventually lead the agent to a dead-end situation from which it will lose the game. For instance, if the first move of the agent is to push the upper box upwards, the box will be moved to the first row next to the top wall. Then the rest of possible actions for this box will be to move it to the left or to the right but none of these actions will ever get the box in one of the platforms and the agent will end up in a dead-end node. Obviously, the agent is unaware of this result when selecting the box to push.

The explanation of the above paragraph gives an idea why Game Set 3 is particularly tricky. The reward or score of these games, if any, does not supply an informative feedback about the implications of certain moves in the accomplishment of the final goal. This type of situations are only avoidable with the use of heuristics that would provide the cost of the solution to reach the goal. But heuristic-based search is not affordable in interactive video-games like the ones treated here.

Table 5.3 shows that the best results are obtained for the games *Modality* (Figure B.5e in Appendix B) and *Painters* (Figure B.5f in Appendix B). The small-size grids of these two games, 5x7 and 4x3 respectively, enable the agent to explore a large portion of the search tree.

Most of the victories in Game Set 3 are achieved in the first level of the games, which was included by the developers to easily test the game agent. This is the case of the game *Bait* (Figure B.5a in Appendix B) where the size of the grid in the first level of the game

**Table 5.3:** Comparing MCTS with our basic implementation of IW(1) in Game Set 3.

| Time | 40ms | | 300ms | | 1s | |
|---|---|---|---|---|---|---|
| Game | IW(1) | MCTS | IW(1) | MCTS | IW(1) | MCTS |
| Bait | 2 | 3 | 3 | 5 | 1 | 5 |
| Bolo Adventures | 0 | 0 | 5 | 0 | 5 | 0 |
| Brain Man | 0 | 2 | 1 | 2 | 0 | 1 |
| Chips Challenge | 3 | 5 | 4 | 4 | 2 | 4 |
| Modality | 10 | 5 | 12 | 5 | 10 | 7 |
| Painters | 19 | 19 | 15 | 21 | 18 | 20 |
| Real Portals | 0 | 0 | 0 | 0 | 0 | 0 |
| Real Sokoban | 0 | 0 | 0 | 0 | 0 | 0 |
| The Citadel | 5 | 2 | 3 | 5 | 5 | 8 |
| Zen Puzzle | 5 | 1 | 7 | 4 | 7 | 6 |
| Total | 44 | 37 | 50 | 46 | 48 | 51 |

is considerably smaller than the rest of levels (5x6 cells in the first game level and 14x9 in the second game level). In *Chips Challenge* (Figure B.5d in Appendix B), *The Citadel* (Figure B.6c in Appendix B) or *Zen Puzzle* (Figure B.6d in Appendix B), the first game level is relatively simpler than the rest of levels. For instance, in the game *Chips Challenge* the first level does not include hurdles like water or fire that require to pick up specific items to overcome them.

Another characteristic of the games in this set is that MCTS achieves more victories when switching from 300ms to 1sec while IW(1) solves fewer problems. Particularly noticeable is the fact that some games achieve worse results in IW(1) when more time is given. This is the case, for instance, of the *Bait*, *Chips Challenge* or *The Citadel* games, which get to solve two less problems when comparing the results of 300ms versus 1sec. As commented in Chapter 4, we introduce a random exploration of the actions in the IW(1) algorithm in order to avoid the agent to systematically choose the first applicable action when all actions obtain the same reward (including the case of a value 0 when the game does not report intermediate rewards). In these games, the agent selects actions randomly until it is close enough to the solution, moment at which the rewards usually become more informative and help the agent focus the search towards the goal.

This random behaviour occurs both in IW(1) and MCTS, having a higher impact in IW(1). This explains that the number of times a game level is solved will vary accordingly to the execution, which in turn explains the small drop in the number of solved problems when the deadline is extended.

In summary, we have observed that our algorithm, despite obtaining good results, also exhibits some issues when tackling the more complex games in Game Set 3 where a solution typically involves the achievement of a series of intermediate steps or subgoals. Specifically, as we commented in Chapter 4, the number of sub-objectives that the IW($i$) algorithm can address increases with the value of $i$.

The refinements over IW(1) presented in sections 4.3.2, 4.3.2 and 4.3.3 are precisely intended to overcome the appointed limitations of IW(1). To this end, the next two sections, Section 5.3 and Section 5.4, describe the improvements accomplished with the refinements explained in Chapter 4.

## 5.3 Basic approximation versus Reward Shaping

We developed an enhanced version of the Iterative Width algorithm that includes **Reward Shaping** (see section 4.3.2) as an attempt to improve the reward function by introducing new elements of the environment when calculating the score of the node. Our hypothesis is that a more informative score might be helpful to better guide the search towards the goal.

For example, imagine a game in which the agent needs to pick up a key for opening a door and this operation increments the score of the node. The reward shaping version of IW, IW(RS) hereafter, is intended to output a reward that reflects that grasping the key is an action on the right track towards a victory. This way, the IW(RS) algorithm will favour nodes where the key is grasped as this operation represents a first sub-objective that needs to be achieved. Subsequently, the algorithm will focus the agent towards the door to eventually reach the final goal of the game.

In addition, we apply a **Discount Factor** refinement (see Section 4.3.2) to obtain higher scores in nodes located in upper levels of the tree (nodes closer to the root node). The Discount Factor is intended to accelerate the search process for achieving the rewards of the game.

We will refer to IW(RS) as the version of IW(1) that implements both the Reward Shaping and Discount Factor. As we will see in the results our hypothesis that IW(RS) improves performance is highly dependent on the characteristics of each of the games.

The setup of this experiment is the same used in the experiments of the previous section; that is, 25 executions per game divided into 5 different game levels with three different time limits (40ms, 300ms and 1 sec). We will compare the results of IW(RS) with the Basic Approximation of IW(1).

### 5.3.1. Results for Game Set 1

The results obtained for the games in Set 1 are displayed in Table 5.4.

**Table 5.4:** Comparing IW(RS) versus our basic implementation of IW(1) in Game Set 1.

| Time | 40ms | | 300ms | | 1s | |
|---|---|---|---|---|---|---|
| Game | IW(1) | IW(RS) | IW(1) | IW(RS) | IW(1) | IW(RS) |
| Aliens | 25 | 25 | 25 | 25 | 25 | 25 |
| Boulderdash | 2 | 2 | 4 | 2 | 3 | 2 |
| Butterflies | 24 | 25 | 25 | 25 | 25 | 25 |
| Chase | 1 | 5 | 11 | 7 | 7 | 6 |
| Frogs | 18 | 18 | 25 | 25 | 25 | 25 |
| Missile Command | 25 | 23 | 25 | 25 | 25 | 25 |
| Portals | 16 | 18 | 20 | 20 | 21 | 21 |
| Sokoban | 4 | 4 | 7 | 4 | 8 | 7 |
| Survive Zombies | 11 | 13 | 13 | 14 | 13 | 14 |
| Zelda | 11 | 8 | 13 | 10 | 8 | 9 |
| Total | 137 | 141 | 168 | 157 | 160 | 159 |

In general terms, we can say that IW(RS) does not improve the results of IW(1) in Game Set 1. In view of the figures shown in Table 5.4 we cannot conclude that either algorithm is more effective for this game set. Indeed the results obtained with IW(RS) are not particularly impressive, showing only a very modest improvement in the 40ms

setting with a total of four more solved problems. It is also noticeable the performance drop in the 300ms setting. We should though note that Game Set 1 is mostly composed of games that require reactive rather than deliberative skills to be solved, which is likely to be reason why the improvement implemented in IW(RS) has such a small impact in this game set.

### 5.3.2. Results for Game Set 2

The results obtained for Game Set 2 are shown in Table 5.5. We can observe a high equality of results except in the 300ms setting where IW(RS) outperforms IW(1). This is justifiable as follows. IW(RS), which is more costly to compute than IW(1), will most likely take more than 40ms to be effectively applied. This would explain that non-appreciable differences are observable in this setting. On the other hand, since the enhanced reward value of IW(RS) is helpful to better discriminate between nodes, IW(RS) will be able to explore nodes which are more informative within a fixed time. This clearly favors the 300ms scenario. However, extending the time for reasoning to 1sec does not bring any clear advantage with respect to IW(1); that is, the potential benefit of IW(RS) is not noticeable in this setting because the objectives deducible by IW(RS) in one-second time slot are also reachable by IW(1) within the same time. In other words, IW(RS) is worthy when no much time is available and we are interested in exploiting as much information as possible about the context within the given time.

The reason why the gain from using IW(RS) is more significant in the results of Game Set 2 than in the results of Game Set 1 in Section 5.3.1 may be explained for the higher complexity of the games in the Set 2.

According to the general results obtained in this game set, we can conclude that the answer to the question as to whether it is worth applying sophisticated and costly reasoning methods such as IW(RS) is found in a trade-off between the problem complexity and the reasoning time.

**Table 5.5:** Comparing IW(RS) versus our basic implementation of IW(1) in Game Set 2.

| Time | 40ms | | 300ms | | 1s | |
|---|---|---|---|---|---|---|
| Game | IW(1) | IW(RS) | IW(1) | IW(RS) | IW(1) | IW(RS) |
| Camel Race | 2 | 3 | 25 | 25 | 25 | 25 |
| Digdug | 0 | 0 | 0 | 0 | 0 | 0 |
| Firestorms | 9 | 13 | 18 | 25 | 19 | 25 |
| Infection | 24 | 25 | 25 | 25 | 24 | 25 |
| Firecaster | 0 | 0 | 0 | 0 | 0 | 0 |
| Overload | 15 | 15 | 12 | 15 | 15 | 12 |
| Pacman | 0 | 0 | 0 | 0 | 0 | 0 |
| Seaquest | 13 | 14 | 18 | 20 | 18 | 21 |
| Whackamole | 21 | 16 | 18 | 19 | 21 | 17 |
| Eggomania | 18 | 16 | 18 | 19 | 19 | 19 |
| Total | 102 | 102 | 134 | 148 | 141 | 144 |

It is also worth commenting the impact of the reward function in some individual games. We can see in Table 5.5 that while IW(RS) reports better results than IW(1) for the game *Firestorms* (Figure B.3c in Appendix B) , the opposite occurs in the game *Whackamole* (Figure B.4d in Appendix B).

The Reward Shaping function in both games draws upon events of **Type 1** (removal of movable objects – see Section 4.3.2), which are interpreted as a reward by the agent. The difference though lies in that while the events of Type 1 are helpful in *Firestorm* to reach a victory in the game, the opposite occurs in *Whackamole*. That is, a higher score value does not mean being closer to a win in the *Whackamole* game. This illustrates one of the downsides when applying the Reward Shaping to the General Video Game framework. The point is that it is difficult to find events that are beneficial to all games, since an event observed in one specific type of game can be detrimental when applied to the context of another game. Nevertheless, the two generic events, **Type 1** and **Type 2** explained in section 4.3.2, selected for the reward function have proven to be useful in a large majority of games.

### 5.3.3. Results for Game Set 3

The results obtained for the games in Set 3 are displayed in Table 5.6. As a whole, we do not observe a major improvement of IW(RS) over IW(1). While it was expected that IW(RS) would perform better in Game Set 3 (this set comprises the most difficult and deliberative games), the design of the generic reward functions highly influences the results in this set.

**Table 5.6:** Comparing IW(RS) versus IW(1) in Game Set 3.

| Time | 40ms | | 300ms | | 1s | |
|---|---|---|---|---|---|---|
| Game | IW(1) | IW(RS) | IW(1) | IW(RS) | IW(1) | IW(RS) |
| Bait | 2 | 5 | 3 | 5 | 1 | 5 |
| Bolo Adventures | 0 | 0 | 5 | 5 | 5 | 5 |
| Brain Man | 0 | 2 | 1 | 2 | 0 | 1 |
| Chips Challenge | 3 | 5 | 4 | 4 | 2 | 5 |
| Modality | 10 | 5 | 12 | 5 | 10 | 5 |
| Painters | 19 | 19 | 15 | 20 | 18 | 17 |
| Real Portals | 0 | 0 | 0 | 0 | 0 | 0 |
| Real Sokoban | 0 | 0 | 0 | 0 | 0 | 0 |
| The Citadel | 5 | 4 | 3 | 1 | 5 | 3 |
| Zen Puzzle | 5 | 6 | 7 | 6 | 7 | 8 |
| Total | 44 | 46 | 50 | 48 | 48 | 49 |

Looking at the breakdown of Table 5.6, we can affirm that no significant differences exist between both versions. However, a detailed look at each game will help us understand the advantages and disadvantages of IW(RS) in this game set.

The data of Table A.3 in Appendix A show that almost every game of the set features events of the two types considered in the design of the Reward Shaping function (Type 1 and Type 2). However, three games are particularly affected by the rewards:

- In the games *Bait* and *Chips Challenge* , represented in the figure B.5d of Appendix B, IW(RS) obtains better results (5 wins in one game level in almost all of the experiments in both games). As we can see in Table A.3, *Bait* has rewards of **Type 1** and **Type 2** and *Chips Challenge* has rewards of **Type 1**. In both cases, the reward results from the collected items, an action which is also necessary for winning the games.

- In contrast, the number of wins with IW(RS) in the game *Modality* is significantly lower than with IW(1). Unlike the previous games, the reward in this game is only

given by events of **Type 2**, which is determined by the four different types of avatar that exist for this game (see Table A.3). The avatar always chooses the node with the highest score returned by the reward shaping and it consequently gets stuck in a loop switching its type repeatedly and without possibility of advancing to reaching the goal of the game.

## 5.4 Basic approximation versus Iterative Width (3/2)

In this section we present the results obtained with the intermediate version between IW(1) and IW(2), IW(3/2), explained in Section 4.3.3. This new version tests the novelty condition using a combination of two atoms, one of them being a special (informative) variable as reflected in the section 4.3.3. The features selected for IW(3/2) have been chosen through expert knowledge of the games, particularly for their relevance in searching a path towards the goals. Our hypothesis is that some of the nodes in the solution paths feature score increments and/or changes in the type of the main avatar. By introducing these elements in a 2-atom tuple for checking the novelty of new nodes, we allow the algorithm to expand the search in those branches in which variations of these two features occur. In practice, this entails finding a solution to a more deliberative objective. That is, the addition of a new atom in the tuples will bring the possibility to reach joint objectives as long as they involve a variation in the atoms.

The other atom of the 2-atom tuple is one regular atom like the used in the basic approximation of IW(1), that is a boolean atom that indicates if an element of a particular type is found in a particular grid position. The pair of atoms is evaluated to check the novelty condition, and prune the corresponding nodes if necessary. The details of this implementation are in section 4.3.3.

The computational cost of IW(3/2) is, obviously, higher than the other two previous versions of IW but in return the game agent will be able to find paths that reach combinations of two sub-goals. The improvement is designed to increase the victories in the deliberative games and, therefore, it will have less relevance in reactive games.

In this section, we will compare the results obtained by the agent in IW(3/2) with the basic approximation of IW(1). Again, the setup of this experiment is composed of 25 executions per game divided into 5 different game levels and using three different time limits.

### 5.4.1. Results for Game Set 1

The results obtained for the games in Set 1 are shown in Table 5.7

There are no significant differences when comparing the basic approximation of IW(1) with IW(3/2) in the first game set. This is because Game Set 1 is mostly composed of reactive games with a lot of feedback in the form of increments at the game score. This gives us enough information for our IW(1) algorithm. Therefore the additional information provided for the IW(3/2) algorithm is redundant and does not provide significant improvements.

We can emphasize the game *Chase*. In this game, the IW(3/2)-based agent, achieves a higher number of victories thanks to the enhanced vision of the search tree provided by this algorithm. This enables the agent to improve the strategy for capturing its enemies, contemplating more deliberative routes to corner rivals.

**Table 5.7:** Comparing IW(3/2) with IW(1) in Game Set 1.

| Time | 40ms | | 300ms | | 1s | |
|---|---|---|---|---|---|---|
| Game | IW(1) | IW(3/2) | IW(1) | IW(3/2) | IW(1) | IW(3/2) |
| Aliens | 25 | 24 | 25 | 25 | 25 | 24 |
| Boulderdash | 2 | 1 | 4 | 6 | 3 | 7 |
| Butterflies | 24 | 25 | 25 | 25 | 25 | 25 |
| Chase | 1 | 5 | 11 | 10 | 7 | 10 |
| Frogs | 18 | 21 | 25 | 25 | 25 | 25 |
| Missile Command | 25 | 23 | 25 | 24 | 25 | 23 |
| Portals | 16 | 15 | 20 | 19 | 21 | 20 |
| Sokoban | 4 | 6 | 7 | 9 | 8 | 4 |
| Survive Zombies | 11 | 9 | 13 | 15 | 13 | 15 |
| Zelda | 11 | 8 | 13 | 12 | 8 | 8 |
| Total | 137 | 137 | 168 | 170 | 160 | 161 |

**Table 5.8:** Comparing IW(3/2) with IW(1) in Game Set 2.

| Time | 40ms | | 300ms | | 1s | |
|---|---|---|---|---|---|---|
| Game | IW(1) | IW(3/2) | IW(1) | IW(3/2) | IW(1) | IW(3/2) |
| Camel Race | 2 | 1 | 25 | 25 | 25 | 25 |
| Digdug | 0 | 0 | 0 | 0 | 0 | 0 |
| Firestorms | 9 | 9 | 18 | 25 | 19 | 24 |
| Infection | 24 | 25 | 25 | 25 | 24 | 25 |
| Firecaster | 0 | 0 | 0 | 0 | 0 | 0 |
| Overload | 15 | 16 | 12 | 16 | 15 | 16 |
| Pacman | 0 | 0 | 0 | 0 | 0 | 0 |
| Seaquest | 13 | 15 | 18 | 11 | 18 | 16 |
| Whackamole | 21 | 17 | 18 | 18 | 21 | 21 |
| Eggomania | 18 | 11 | 18 | 20 | 19 | 19 |
| Total | 102 | 94 | 134 | 140 | 141 | 146 |

### 5.4.2. Results for Game Set 2

The results obtained for the games in Set 2 are shown in Table 5.8

In the second set is noticeable the importance of a good balance between the reliability representing the states and the increment in the reasoning time that this reliability brings. In other words, more detailed states (more atoms taken into account to calculate the novelty, or a higher number of variables), provide us with more information and therefore more possibilities of making better decisions. However, we must take into account the time limit, so the increment in complexity will make the search algorithm explore fewer nodes. This may prevent the search algorithm from reaching a node with a reward, causing the agent to adopt a random behaviour.

This characteristic is clearly seen in the experiment carried out with the games of the Set 2 for a 40 ms time limit, where the algorithm IW(3/2) obtains fewer victories due to the computational cost of adding the new 2-atom tuples. In a short time-frame like 40ms, the application of IW(3/2) entails a lower number of explored nodes, hindering a deeper exploration of the search tree and causing that the agent not to be able to find distant rewards. In contrast, with a time limit of 300ms, the IW(3/2)-based agent is able to achieve more victories than the IW(1) basic approximation. The wider time slot allows the agent

**Table 5.9:** Comparing IW(3/2) with IW(1) in Game Set 3.

| Time | 40ms | | 300ms | | 1s | |
|---|---|---|---|---|---|---|
| Game | IW(1) | IW(3/2) | IW(1) | IW(3/2) | IW(1) | IW(3/2) |
| Bait | 2 | 5 | 3 | 9 | 1 | 7 |
| Bolo Adventures | 0 | 0 | 5 | 5 | 5 | 5 |
| Brain Man | 0 | 0 | 1 | 1 | 0 | 2 |
| Chips Challenge | 3 | 8 | 4 | 9 | 2 | 6 |
| Modality | 10 | 5 | 12 | 5 | 10 | 5 |
| Painters | 19 | 24 | 15 | 23 | 18 | 25 |
| Real Portals | 0 | 0 | 0 | 0 | 0 | 0 |
| Real Sokoban | 0 | 0 | 0 | 0 | 0 | 0 |
| The Citadel | 5 | 4 | 3 | 6 | 5 | 8 |
| Zen Puzzle | 5 | 5 | 7 | 5 | 7 | 6 |
| Total | 44 | 51 | 50 | 63 | 48 | 64 |

to explore a higher number of nodes. In addition, the use of 2-atom tuples augment the precision in the search, exploring more branches that would have been pruned pruned otherwise with IW(1). The larger exploration occurs at all levels, from the branches close to the root as to deeper branches of the tree. This justifies the increase of victories obtained for both 300ms and 1 second.

It is also worth commenting the impact of the algorithm in some individual games. We can see in Table 5.8 that while IW(3/2) reports better results than IW(1) for the game *Firestorms* (Figure B.3c in Appendix B) , the opposite occurs in the game *Seaquest* (Figure B.4c in Appendix B). In the case of the *Firestorms*, the IW(3/2) version allows exploring a larger part of the tree finding more elaborated solutions that make the agent get the victory in more cases. In contrast, in the case of *Seaquest*, the big number of increments of score (see Table A.2) causes the search tree grow up more in width than in depth. That is, the changes of the game score will make that hardly any node will be pruned. So the algorithm will have a behavior similar to a BFS. This causes the search algorithm to not reach the deeper levels of the tree causing the submarine representing the avatar to be unable to surface in time and remain without oxygen.

### 5.4.3. Results for Game Set 3

The results obtained for the games in Set 3 are displayed in Table 5.8

Regarding Game set 3, which comprises the most complex games that require in some cases complex strategies to reach the partial objectives, we find that the algorithm IW(3/2) provides significant improvements. This version of IW increases the wins by 7, 13 and 16 regarding the victories obtained by the IW(1) in the different time sets.

In this set, games like *Painters* are improved to the point of winning in almost all instances.

The games *Bait*, *Chips Challenge* and *The Citadel* improve their results in comparison with the basic approximation. Due to the prune of IW(3/2) is less aggressive than in the basic approximation, the IW(3/2) is able to expand more nodes of the tree search. This exploration allows the agent to reach more complex objectives such as those included in these three mentioned games where some goals are dependent on other sub-goals.

## 5.5 Conclusions

From the results described in this chapter we have extracted certain characteristics about the different versions of the agents. This Section is devoted to summarize these characteristics in order to establish the relevant points of each type of algorithm as well as the types of games in which it specializes.

Firstly, comparing IW(1)-based agent with MCTS-based agent we can conclude that IW is more suitable to face games of the difficulty proposed by the GVG-AI competition. In general terms, IW(1) outperfroms MCTS across the three time settings.

Specifically, we can draw different conclusions depending on the types of game evaluated:

- The IW(1) algorithm specializes in reactive games, which require reflexes on the part of the agent, games in which the system provides enough rewards to contribute to the search. However this type of games do not represent a problem for other algorithms of General Video Game as is the case of the MCTS so it does not represent a significant improvement.
- The IW(1) algorithm is ideal for games with simple objectives that do not require the realization of several sub-objectives, even if they require a distant solution in the search space (such as the games of Set 2). This algorithm, with very little time, expands the search tree to the point of obtaining diverse states, covering great part of the map and also covering a lot of different new variables into the states of the game. On the other hand, if there are no rewards, the MCTS algorithm will show a random behavior and perform contradictory movements within the search space. The IW(1) algorithm can be expanded to a distant target and thus achieve victory in some games.
- When confronted with complex objectives, especially in puzzle games such as in Set 3, the IW(1) algorithm might prune the branches that contain the solution, as it does not satisfy the condition of novelty. In contrast, the MCTS will always have access to that branch. So theoretically, with enough time, it would reach the right solution. Despite this, for the three time limits, the behavior of both game agents is fairly similar, slightly better IW(1) for the shortest times (40ms and 300ms) and the MCTS for the highest time (1 sec.).
- On studying the games, we observed that in some cases the increments of score are not a sufficiently informative indicator towards the objectives of the game. As an example, the action of collecting some items – which is required in some games in order to win - is not reflected as an increase in the score so an agent guided by the score will not benefit from this action. In addition, in many games the increase in score is not a direct indicator that progress is being made. Some of the examples in which this happens are:

  – Puzzle games in which items are collected to improve the score as a complement to solve the main problem like in *Brainman*.
  – Games in which a positive reinforcement is given to understand that the game is about making that event although that event is not beneficial for the resolution of the problem. This is the case of the *Firecaster* where destroying any box increases our score.

  In these cases it is easy for the agent to be guided by a local minimum and may end up losing the game or reaching dead spots.

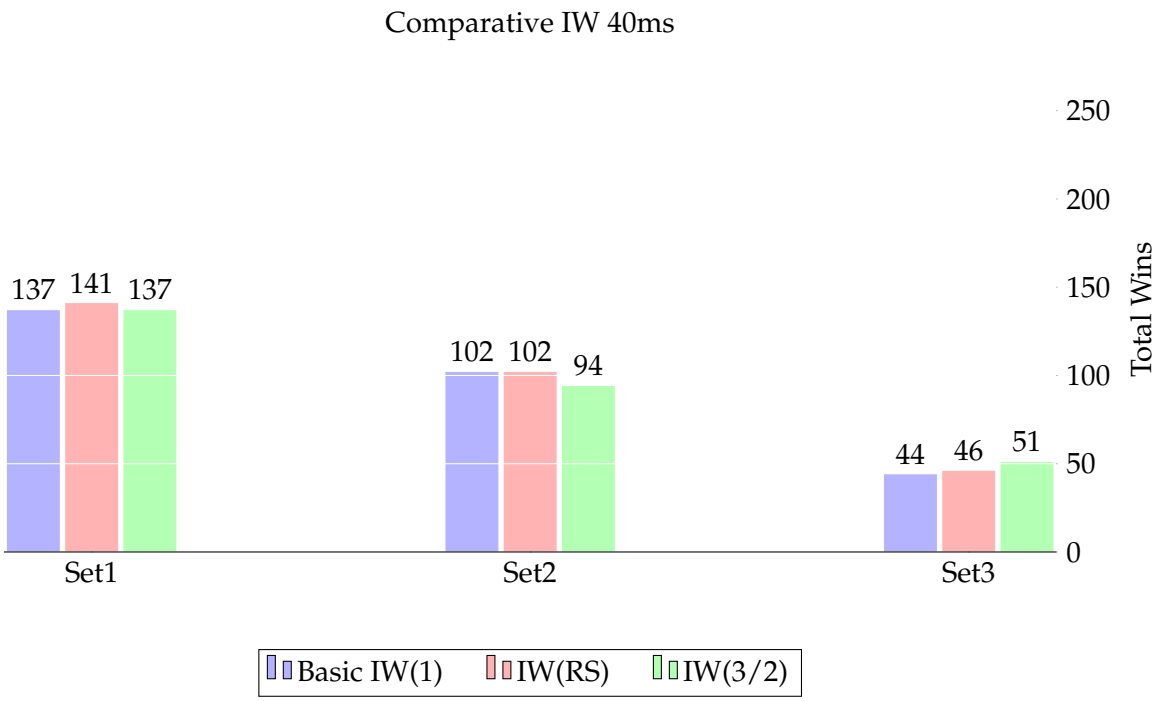



**Figure 5.1:** Comparison of total victories between IW(1), IW(RS) and IW(3/2) in a time-frame of 40 milliseconds

- We also observed that IW(1) completes the search tree in certain games. For this reason, it is feasible to implement improvements that increase its complexity and allow taking advantage of the exceeding time. These improvements would be thought to be used exclusively for those games in which the search tree is completely explored since to use it in the rest of cases, could be counterproductive.

Covering those needs, in this project we have implemented the improvements of IW(RS) and IW(3/2). Figures 5.1, 5.2 and 5.3 show a comparison of the algorithms based on the total results for each set.

From these tables we can see that the IW(RS) does not represent a significant improvement. In most of the experiments, the results remain at the same rate as the basic version of IW(1). The only mentioned differences are observed within 300ms (Figure 5.2) where set 1 gets 11 victories less than the basic version and set 2 gets 14 victories more. This could support our hypothesis that the reward shaping version, prioritizes the nodes with rewards, making the rewards more accessible despite being at a deep level of the search tree so it gives better results in the set2 whose rewards are usually far from the initial node. It also confirms that any increase in complexity involves a cost, as it damages the number of observable nodes per unit of time. Therefore, in a real application of these algorithms, we should be selective with the problems to which we can and cannot apply these improvements.

It is also necessary cautiously choose the events that can be recognized as subjective rewards. Remember that the goal within the GVG-AI is to make agents capable of adapting to new environments. In this context, an event that can be considered positive in one game might negatively influence another.

In summary, IW(RS) can help us to make accessible certain objectives using a single atom to check novelty condition. Objectives, that are not accessible by the characteristics of the game using IW(1). However, we must include generic events to avoid an agent specialized in a certain type of games.

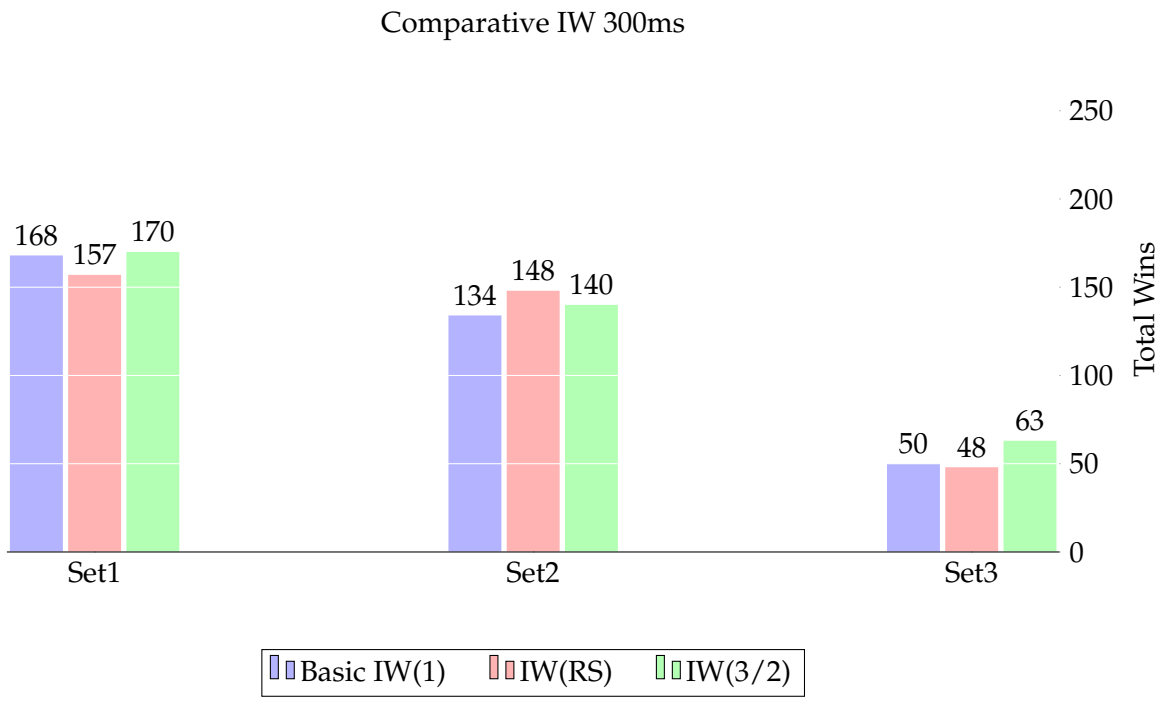



**Figure 5.2:** Comparison of total victories between IW(1), IW(RS) and IW(3/2) in a time-frame of 300 milliseconds

On the other hand, the IW(3/2) version obtains great results. From the results in Table 5.1 it can be concluded that the ideal type of game where to include this improvement are the puzzle type. In this figure we can visualize how the games of the set 3 ( most of them puzzle games) increase their victories. On the other hand, the set 1, which is composed of reactive games, remains constant. Finally, the games of the set 2, which require longer solution paths, suffer a reduction in the number of victories.

In addition, by increasing the time limitation, the increase of victories in the set 3 is favored by this new version. Going from 15.9% more victories for 40ms experiments to 26% more for 300ms set-time and 33.3% more victories with IW(3/2) in 1 second set-time.

Unlike the *reward shaping* version, the IW(3/2) version allows the search algorithm to go deeper into the search tree compared to the basic version. This is thanks to the fact that 2 properties or atoms are compared simultaneously at the time of verifying if they bring novelty to the search algorithm. This makes it less likely that a new state will be pruned. This then increases the number of explored branches producing more complex behaviors.

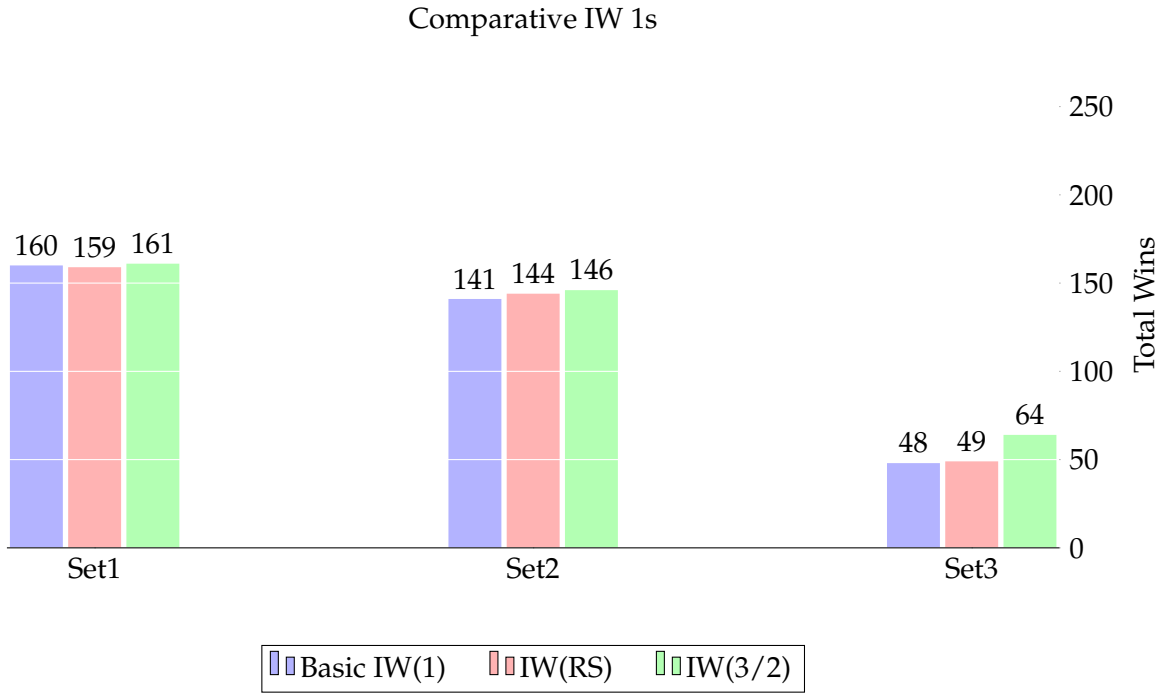



**Figure 5.3:** Comparison of total victories between IW(1), IW(RS) and IW(3/2) in a time-frame of 1 second

# CHAPTER 6
# Future work

This chapter discusses several research directions that can be followed in the near future to improve the performance of IW-based agents at video-game playing.

**Exploiting game structure**

First we propose to exploit knowledge about the particular structure of the games, like the information reported in appendix A. As discussed in chapter 4.2, this project implemented a simple agent capable of collecting representative information of the games as it plays. The information collected by that agent has been compiled in the tables of the appendix A and they have served us as support, to be able to relate and understand the contributions included both for the IW-based agents. However, the collected data is not exploited in the decision making process of the implemented IW-based agents. For instance, we could using the most appropriate algorithm for each game according to its structure.

We believe that the information gathered in these tables can serve as characteristics for the use of machine learning techniques to classify the games according to their type, as well as choose the best applicable strategy for the game. In more detail, what is proposed in this section is to use clustering algorithms to detect similar games, according to the features included in those tables, as well as more characteristics that are considered interesting. These clusters might differentiate the games between deliberative or reactive, might join those games in which it is required an objective more elaborated like collecting an item to use it to obtain a goal in the game, might group those games in which there are enemies and it is required that the agent dodges them with agility. In summary, similar games would be grouped to be treated with the same algorithm. The objective of these clusters is taking advantage of the processing of 1 second that allows the GVG-AI for each game to extract its characteristics of new games and classify it according of the models extracted of these clusters. To do this we can use a K-Nearest Neighbors classifier or some kind of linear classifier (We can check experimentally which offers better results).

Finally, once we have the type of game we're dealing with, we should be able to choose the algorithm that's most beneficial to the game. To do this, we can make use of the results tables, checking the average number of victories that each type of game has for each of the agents developed in this project (or even adding other types of agent like a more refined version of the MCTS). With this correspondence between game types and the best strategy to be used by our new agent, it is trivial to choose the right strategy for the new game we are facing.

### Planning with pixels

Another possible research direction is to use IW for general video game playing but, using as state atoms features that are extracted directly from the screen. This approach has acquired relevance in recent years due to its great results in the field of General Video Game [6]. It is based on the philosophy that a human being, when facing a video game, does not have access to the variables in Ram memory or the methods provided by an API such as the ones used in this project. Instead, the human faces a game through a screen, which transmits the relevant information of the game. With the same philosophy, the input received by an agent oriented to planning with pixels, is the content of the screen, i.e. the color that represents each of the pixels of the screen. In the case of the games we are facing, based on an Arcade video console, the domain of each pixel has 128 values, representing the full range of colors.

The first change to be applied is the use of different atoms to implement the IW search. In planning with pixels, the atoms are extracted directly of **screen**. The node will be represented by a number of variables equal to the resolution of screen (Number of width pixels $\times$ Number of height pixels). Each variable has a value between 0 and 127 representing the color of this pixel in the game. The number of different atoms is so large that could make the search process intractable. For this reason there are some techniques, aimed at reducing the amount of atoms we use for the IW search, such as *Convolutional neural networks* or the *B-PROST* features [20].

### Improving the any-time behaviour

While IW(1) is a linear time algorithm that has been shown effective at the GVG-AI competition, its performance when actions have to be taken in a few milliseconds is limited by the underlying breadth-first search. IW(1) can search much deeper than a regular breadth-first search over a limited time window, but if the window is small, nodes that are beyond a certain depth will not be explored either. This issue is evidenced at games like *camel race*, *chase*, *frogs* or *firestorms*, where the performance of IW(1) for 40ms is much lower than the performance of the same algorithm but for 300ms and 1 sec time windows.

Recently an alternative to IW(1), called Rollout IW(1) has been developed [6], that does not have this limitation and has better any-time behavior than IW(1). In addition, Rollout IW(1) asks less from the simulator: while tree search algorithms like IW(1) need the facility of expanding nodes, i.e., of applying all actions to a node the rollouts in Rollout IW(1) apply just one action per node.

### Integrating planning and learning

Last but not least, IW algorithms are pure exploratory planing algorithms that do not implement any kind of learning of the previous decision making episodes. In other words, if our IW-based agent plays twice the same game it will start again the IW(1) searches from scratch.

A promising research direction here is to integrate IW algorithms with machine learning techniques, for instance within a reinforcement learning framework similar to the popular game playing algorithms *Alphago*, *Alphazero* and *Alphastar* [21].

# Bibliography


[1] Alpha. https://deepmind.com/research/alphago/. Accessed: 2018-03-25.

[2] The gvg-ai competition. http://www.gvgai.net/. Accessed: 2018-09-30.

[3] The gvg-ai framework state observation. http://www.gvgai.net/forwardModel.php. Accessed: 2019-03-30.

[4] Mario a. http://www.marioai.org/. Accessed: 2018-03-25.

[5] Stockfish. https://stockfishchess.org/. Accessed: 2018-03-25.

[6] Wilmer Bandres, Blai Bonet, and Hector Geffner. Planning with pixels in (almost) real time. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[7] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[8] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2):201–240, 2002.

[9] Mat Buckland. *Programming game AI by example*. Jones & Bartlett Learning, 2005.

[10] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[11] Michael Chung, Michael Buro, and Jonathan Schaeffer. Monte carlo planning in rts games. In *CIG*. Citeseer, 2005.

[12] N. Cole, S. J. Louis, and C. Miles. Using a genetic algorithm to tune first-person shooter bots. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*, volume 1, pages 139–145 Vol.1, June 2004.

[13] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. Sas+ planning as satisfiability. *Journal of Artificial Intelligence Research*, 43:293–328, 2012.

[14] Ercüment Ilhan and A Şima Etaner-Uyar. Monte carlo tree search with temporal-difference learning for general video game playing. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 317–324. IEEE, 2017.

[15] Andreas Junghanns and Jonathan Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1-2):219–251, 2001.

[16] Richard Kaye. Minesweeper is np-complete. *The Mathematical Intelligencer*, 22(2):9–15, 2000.

[17] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[18] Richard E Korf. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI/IAAI*, pages 700–705, 1997.

[19] Joel Lehman and Kenneth Stanley. Exploiting open-endedness to solve problems through the search for novelty. *Artificial Life - ALIFE*, pages 329–336, 01 2008.

[20] Yitao Liang, Marlos C Machado, Erik Talvitie, and Michael Bowling. State of the art control of atari games using shallow reinforcement learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 485–493. International Foundation for Autonomous Agents and Multiagent Systems, 2016.

[21] Chris J Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in go using deep convolutional neural networks. 2015.

[22] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games*, 5(4):293–311, 2013.

[23] Juan Ortega, Noor Shaker, Julian Togelius, and Georgios N Yannakakis. Imitating human playing styles in super mario bros. *Entertainment Computing*, 4(2):93–104, 2013.

[24] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon M Lucas. General video game ai: Competition, challenges and opportunities. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[25] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

[26] Tom Schaul. A video game description language for model-based or interactive learning. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

[27] Alexander Shleyfman, Alexander Tuisov, and Carmel Domshlak. Blind search for atari-like online planning revisited. In *IJCAI*, pages 3251–3257, 2016.

[28] 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, 2016.

[29] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[30] GJ Sussman. A computer model of skill acquisition, volume 1 of. *Artificial Intelligence Series*, 1975.

[31] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

[32] Julian Togelius, Noor Shaker, Sergey Karakovskiy, and Georgios N Yannakakis. The mario ai championship 2009-2012. *AI Magazine*, 34(3):89–92, 2013.

APPENDIX A

# Games analysis

**Table A.1**

| Games | Aliens | Boulderdash | Butterfies | Chase | Frogs | Missile Command | Portals | Sokoban | SurvZombies | Zelda |
|---|---|---|---|---|---|---|---|---|---|---|
| Avatar types [1] | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| Reward shaping[2] | F | F | F | F | T | F | F | F | T | T |
| Dead[3] | T | T | F | T | T | F | T | F | T | T |
| Portals [4] | F | T | F | F | F | F | T | F | F | T |
| Resources [5] | F | T | F | F | F | F | F | F | F | F |
| NPC [6] | 1 | 2 | 1 | 2 | 0 | 1 | 1 | 0 | 2 | 1 |
| Movable [7] | 1 | 1 | 0 | 0 | 3 | 0 | 2 | 1 | 1 | 0 |
| Inamobibles [8] | 1 | 1 | 1 | 2 | 4 | 3 | 6 | 3 | 4 | 2 |
| From Avatar [9] | T | T | F | F | F | T | F | F | F | T |
| Rewards [10] | 71 | 6 | 11 | 7 | 2 | 5 | 2 | 1 | 25 | 9 |
| Num Sprites [11] | 7 | 10 | 5 | 6 | 10 | 6 | 12 | 5 | 8 | 8 |
| Ends exploration [12] | 0,07 % | 0,00 % | 1,20 % | 0,00 % | 10,00 % | 2,00 % | 0,00 % | 99,00 % | 0,00 % | 99,00 % |
| Max Nodes[13] | 345 | 135 | 232 | 248 | 148 | 310 | 260 | 368 | 392 | 300 |

[1]Number of different Main avatar's type in the game. (Reward shaping of Type 2 if > 1)
[2]Games with events captured as a reward in the reward shaping version. (Reward shaping of Type 1 or Type 2)
[3]Games in wich the avatar can die.
[4]Games with portal elements.
[5]Games with resources provided by the GVG-AI framework.
[6]Number of Non-Player-Character types in game.
[7]Number of movable sprites types in game.
[8]Number of inamobibles sprites types in game.
[9]Games with elements of the game or sprites made by the avatar like shots.
[10]Number of changes in the game score.
[11]Number of different sprite types in game.
[12]Percentage of movements in wich the avatar ends the exploration search in IW(1).
[13]Max number of nodes observed in game for experiments in 40ms.

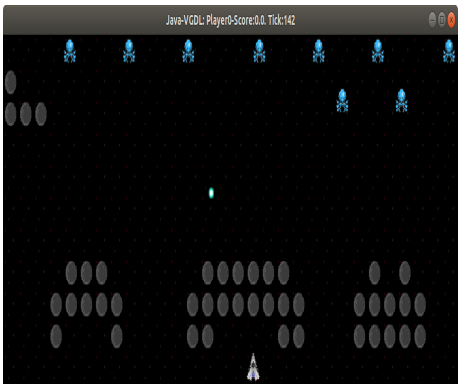
**Table A.2**

| Games | Camel Race | Digbug | Firestorms | Infection | Firecaster | Overload | Pacman | Seaquest | Whackamole | Eggomania |
|---|---|---|---|---|---|---|---|---|---|---|
| Avatar types | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| Reward shaping | T | T | T | T | F | F | F | T | T | F |
| Dead | T | T | T | F | T | T | T | T | T | T |
| Portals | T | T | T | T | T | F | T | T | T | F |
| Resources | F | F | T | F | T | T | F | F | F | F |
| NPC | 1 | 0 | 0 | 3 | 0 | 1 | 4 | 2 | 1 | 2 |
| Movable | 3 | 3 | 1 | 0 | 1 | 0 | 0 | 2 | 2 | 1 |
| Inamobibles | 2 | 4 | 1 | 2 | 4 | 5 | 5 | 2 | 1 | 3 |
| From Avatar | F | T | F | T | T | T | F | T | F | T |
| Rewards | 2 | 16 | 8 | 35 | 15 | 17 | 10 | 63 | 71 | 8 |
| Num Sprites | 9 | 11 | 7 | 10 | 10 | 10 | 14 | 11 | 7 | 7 |
| Ends exploration | 5,00 % | 1,00 % | 4,00 % | 1,00 % | 99,00 % | 0,81 % | 0,00 % | 0,00 % | 97,00 % | 94,00 % |
| Max Nodes | 208 | 165 | 300 | 290 | 245 | 255 | 66 | 440 | 564 | 360 |

**Table A.3**

| Games | Bait | Bolo Adventures | Brain Man | Chips Chall. | Modality | Painters | R. Portals | R. Sokoban | The Citael | Zen Puzzle |
|---|---|---|---|---|---|---|---|---|---|---|
| Avatar types | 2 | 1 | 1 | 1 | 4 | 1 | 2 | 1 | 1 | 1 |
| Reward shaping | T | T | T | T | T | T | T | F | T | T |
| Dead | F | T | T | T | T | F | T | T | T | T |
| Portals | F | F | F | F | T | F | T | F | F | F |
| Resources | 0 | 0 | 0 | 5 | 0 | 0 | 1 | 0 | 0 | 0 |
| NPC | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Movable | 2 | 4 | 3 | 1 | 1 | 2 | 2 | 2 | 1 | 1 |
| Inamobibles | 4 | 4 | 6 | 9 | 5 | 2 | 8 | 2 | 3 | 3 |
| From Avatar | F | F | F | F | F | F | T | F | 0 | F |
| Rewards | 7 | 1 | 12 | 23 | 2 | 31 | 10 | 3 | 2 | 34 |
| Num Sprites | 8 | 10 | 10 | 17 | 11 | 5 | 16 | 7 | 6 | 6 |
| Ends exploration | 0.81 | 0.02 | 0.99 | 0.03 | 0.99 | 0.94 | 0.99 | 0.99 | 0.96 | 0.99 |
| Max Nodes | 152 | 232 | 284 | 284 | 96 | 72 | 185 | 180 | 284 | 588 |

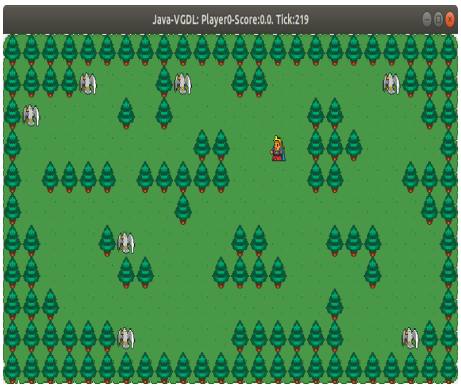
# APPENDIX B
# Games




**(a)** *Aliens*

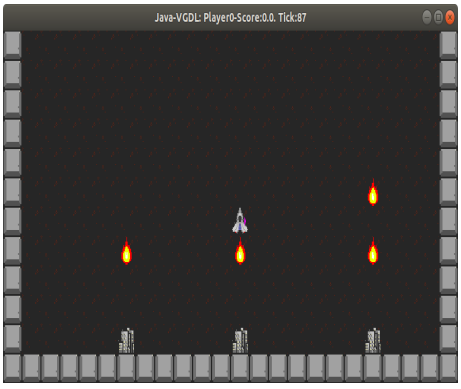

**(b)** *Boulderdash*


**(c)** *Butterflies*



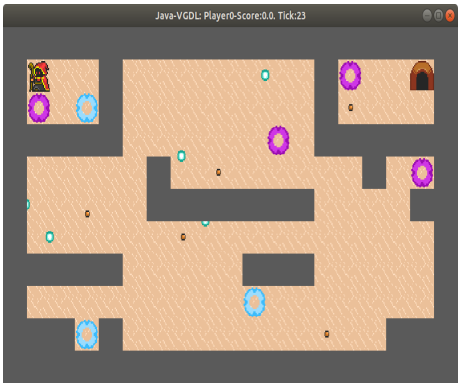

**(d)** *Chase*


**(e)** *Frogs*

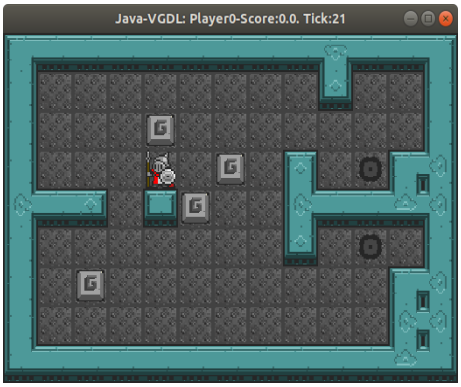



**(f)** *Misilecommand*

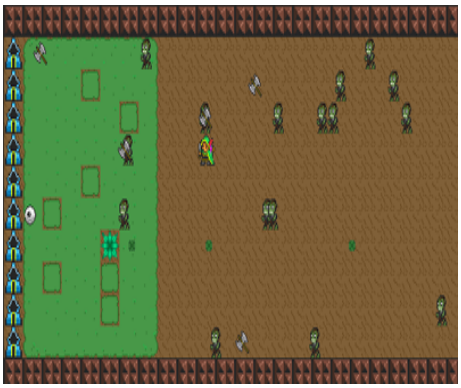
**Figure B.1:** Game Set 1




**(a)** *Portals*



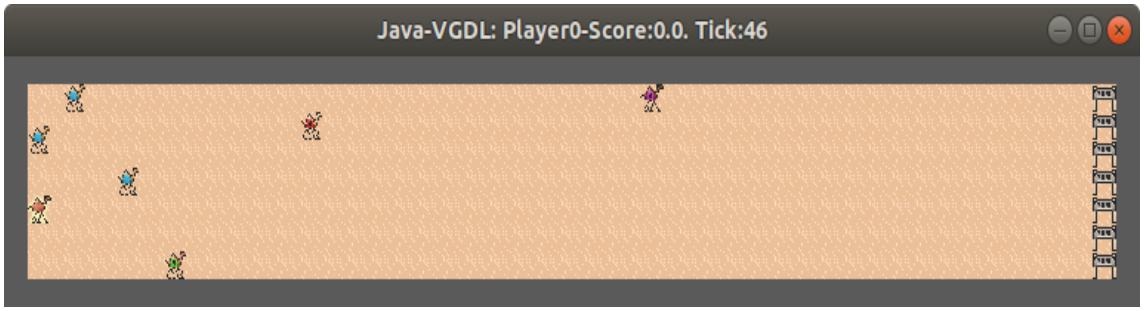

**(b)** *Sokoban*



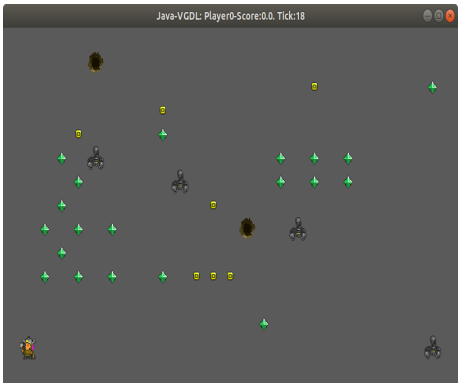
**(c)** *Survivezombies*


**(d)** *Zelda*

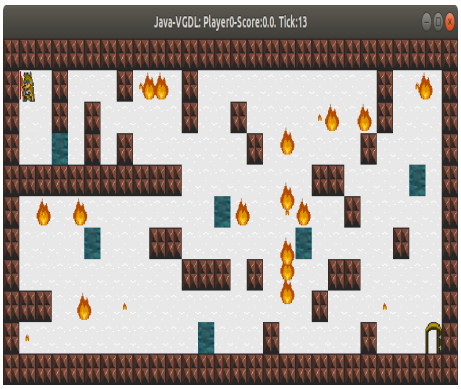
**Figure B.2:** Game Set 1



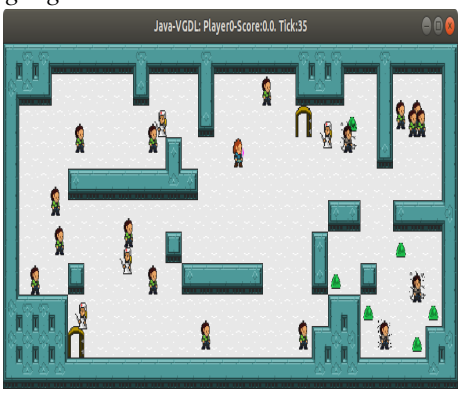

**(a)** *Camel Race*



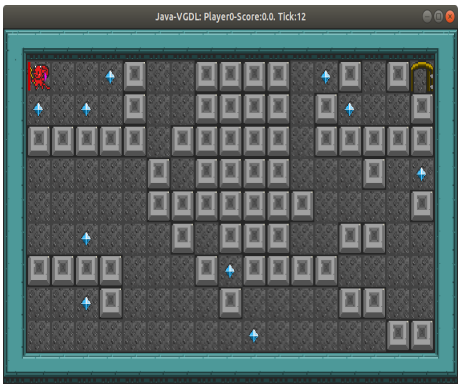

**(b)** *Digdug*




**(c)** *Firestorms*

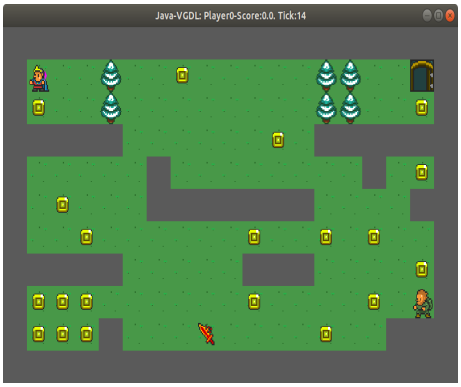



**(d)** *Infection*

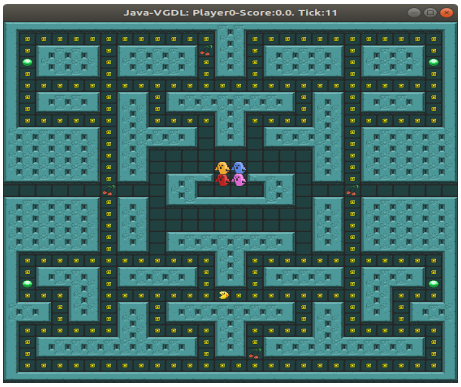



**(e)** *Firecaster*

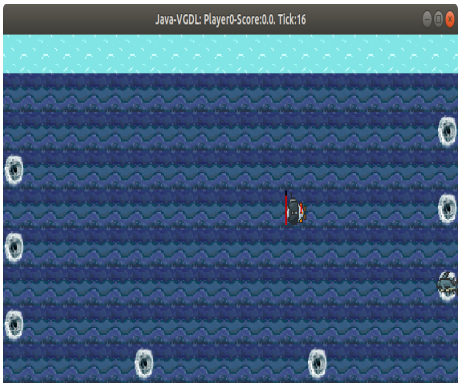
**Figure B.3:** Game Set 2




**(a)** *Overload*

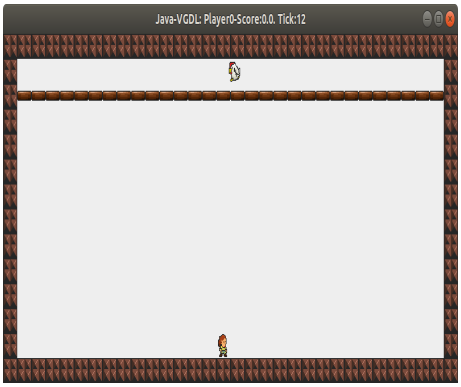



**(b)** *Pacman*




**(c)** *Seaquest*

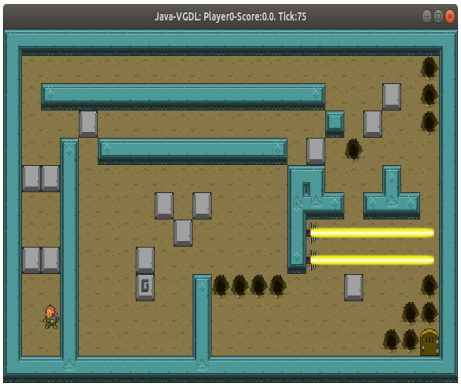

**(d)** *Whackamole*




**(e)** *Eggomania*

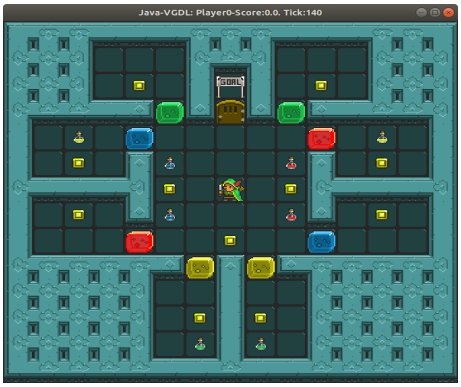
**Figure B.4:** Game Set 2


**(a)** *Bait*

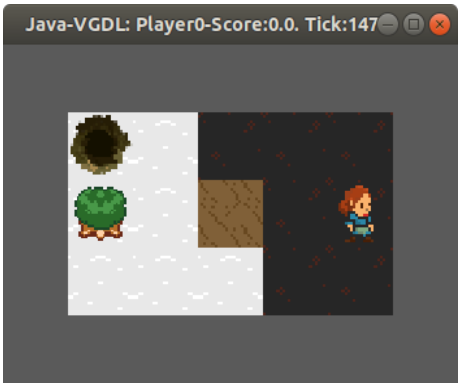



**(b)** *Bolo Adventures*


**(c)** *Brainman*




**(d)** *Chips Challenge*




**(e)** *Modality*

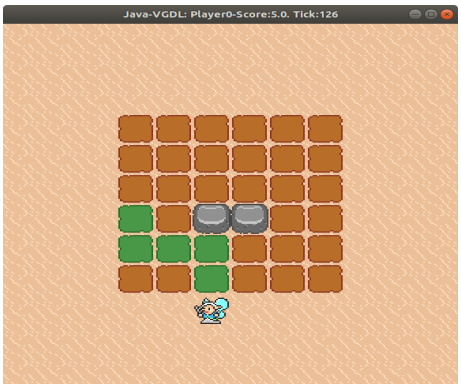

**(f)** *Painter*

**Figure B.5:** Game Set 3

**(a)** *Real Portals*


**(b)** *Real Sokoban*


**(c)** *The Citadel*




**(d)** *Zen puzzle*

**Figure B.6:** Game Set 3