UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Matheus Ribeiro Furtado de Mendonça

Evolution of Reward Functions for Reinforcement Learning applied to Stealth Games

Juiz de Fora

2016

UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Matheus Ribeiro Furtado de Mendonça

Evolution of Reward Functions for Reinforcement Learning applied to Stealth Games

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Orientador:	Raul Fonseca Neto
Coorientador:	Heder Soares Bernardino

Juiz de Fora
2016

Ficha catalográfica elaborada através do programa de geração
automática da Biblioteca Universitária da UFJF,
com os dados fornecidos pelo(a) autor(a)

```
Mendonça, Matheus Ribeiro Furtado de.
   Evolution of Reward Functions for Reinforcement Learning
applied to Stealth Games / Matheus Ribeiro Furtado de
Mendonça. -- 2016.
   94 f. : il.


   Orientador: Raul Fonseca Neto
   Coorientador: Heder Soares  Bernardino
   Dissertação (mestrado acadêmico) - Universidade Federal de
Juiz de Fora, ICE/Engenharia. Programa de Pós-Graduação em
Ciência da Computação, 2016.


   1. Stealthy Path Planning. 2. Reinforcement Leaning. 3.
Genetic Algorithms. 4. Evolution of Reward Functions. I.
Neto, Raul Fonseca, orient. II. Bernardino, Heder Soares ,
coorient. III. Título.
```

Matheus Ribeiro Furtado de Mendonça

Evolution of Reward Functions for Reinforcement Learning applied to Stealth Games

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Aprovada em 3 de Março de 2016.

ACKNOWLEDGMENTS

First, I would like to thank my advisors, Raul and Heder, for their tremendous support and valuable teachings. Their help and advices allowed me to greatly enhance my knowledge and encouraged me to finish this work.

To my parents, Sônia and José Paulo, for their unconditional help during all these years and for their extremely valuable wisdom that helped to shape the person I am.

To Anna Claudia for sharing the best moments during these years, for being the perfect company, and for always supporting me. You clearly made things easier for me. I would also like to thank your family for the shared moments.

To my brothers, Philipe, Thiago and Pedro, and for my friends Luiz Eduardo, David, Eduardo and Allyson for the friendship and support.

I also thank UFJF, its staff members and CAPES for the continued support and resources.

"Do or do not. There is no try."

Master Yoda

RESUMO

Muitos jogos modernos apresentam elementos que permitem que o jogador complete certos objetivos sem ser visto pelos inimigos. Isso culminou no surgimento de um novo gênero chamado de jogos furtivos, onde a furtividade é essencial. Embora elementos de furtividade sejam muito comuns em jogos modernos, este tema não tem sido estudado extensivamente. Este trabalho aborda três problemas distintos: (i) como utilizar uma abordagem por aprendizado de máquinas de forma a permitir que o agente furtivo aprenda como se comportar adequadamente em qualquer ambiente, (ii) criar um método eficiente para planejamento de caminhos furtivos que possa ser acoplado à nossa formulação por aprendizado de máquinas e (iii) como usar computação evolutiva de forma a definir certos parâmetros para nossa abordagem por aprendizado de máquinas. É utilizado aprendizado por reforço para aprender bons comportamentos que sejam capazes de atingir uma alta taxa de sucesso em testes aleatórios de um jogo furtivo. Também é proposto uma abordagem evolucionária capaz de definir automaticamente uma boa função de reforço para a abordagem por aprendizado por reforço.

Palavras-chave: Planejamento de Caminhos Furtivos. Aprendizado por Reforço. Algoritmos Genéticos. Evoluçao de Funçoes de Reforço.

ABSTRACT

Many modern games present stealth elements that allow the player to accomplish a certain objective without being spotted by enemy patrols. This gave rise to a new genre called stealth games, where covertness plays a major role. Although quite popular in modern games, stealthy behaviors has not been extensively studied. In this work, we tackle three different problems: (i) how to use a machine learning approach in order to allow the stealthy agent to learn good behaviors for any environment, (ii) create an efficient stealthy path planning method that can be coupled with our machine learning formulation, and (iii) how to use evolutionary computing in order to define specific parameters for our machine learning approach without any prior knowledge of the problem. We use Reinforcement Learning in order to learn good covert behavior capable of achieving a high success rate in random trials of a stealth game. We also propose an evolutionary approach that is capable of automatically defining a good reward function for our reinforcement learning approach.

Keywords: Stealthy Path Planning. Reinforcement Leaning. Genetic Algorithms. Evolution of Reward Functions.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF SYMBOLS

S	Set of possible states in a Reinforcement Learning problem
A	Set of possible actions to be executed by the agent
π	Policy that controls which action will be executed for each state
$\pi*$	Policy that selects only the action that maximizes the expected future rewards
$r(s,a)$	Reward function for the action a executed while in state s
$Q(s,a)$	Q-Value of the state-action pair (s,a)

LIST OF ACRONYMS

RL Reinforcement Learning

GA Genetic Algorithm

EC Evolutionary Computing

DP Dynamic Programming

MC Monte Carlo

TD Temporal Difference

SRF Simple Reward Function

RBRF Rule-Based Reward Function

TBGA Table Based Genetic Algorithm

RBGA Rule Based Genetic Algorithm

CONTENTS

1 INTRODUCTION

Video games have faced a great evolution over the past years and now represents an industry with high economical value that is dedicated in creating high quality entertainment software (CRANDALL; SIDAK, 2006). The gaming industry have worked hard with the academic community in order to push the boundaries of several research areas and allow the creation of more realistic games. Computer Science played a major role in the popularization of video games through the development of different research areas, such as: computer graphics, high performance computing, artificial intelligence, computer networks, among others. Each of these areas are still under constant development due to the growing demands of the gaming industry.

Artificial Intelligence (AI) applied to games represents a major research area that is concerned in creating special controllers for artificial agents in order to allow them to behave realistically and autonomously in a game. This area is of great interest for the gaming industry due to the attractiveness of a believable and realistic AI in a game (SONI; HINGSTON, 2008; MENDONÇA et al., 2015). There are several studies regarding the application of AI techniques to different game genres, such as *First Person Shooter* (FPS) (GLAVIN; MADDEN, 2015), Real Time Strategy (RTS) (D'SILVA et al., 2005), fighting (MENDONÇA et al., 2015), etc.

Machine learning techniques applied to video games have been extensively researched through the past years, especially after the successful application of Reinforcement Learning to the Gammon game (TESAURO, 1992). Most research focus on using Artificial Neural Network (D'SILVA et al., 2005; SONI; HINGSTON, 2008) or Reinforcement Learning (GLAVIN; MADDEN, 2015; WENDER; WATSON, 2012, 2014) in order to learn through examples or through experience, creating more realistic agents with human behaviors that adapts to new situations. Other works also focus on how to improve the machine learning methods by using special algorithms in order to define certain learning parameters without any prior knowledge of the problem (SINGH et al., 2010; SCHRUM; MIIKKULAINEN, 2015). This helps to automate the learning process and represents an attractive solution for game developers.

1.1 PROBLEM DEFINITION AND OBJECTIVES

Machine Learning and video games have a mutual gain relationship, where video games provide an easy to manipulate testbed for complex machine learning problems, while it benefits with several methods capable of controlling intelligent agents in order to create a believable and realistic artificial intelligence for virtual agents. There are several game genres that benefits with machine learning methods, each with their own characteristics and challenges. Several of these genres have been extensively researched by the academic and gaming community, while others present little development with respect to intelligent behaviors. One such genre is called "Stealth", that presents stealthy agents that must complete a specific goal without being spotted by enemy patrols. Several works tackled the problem of stealthy path planning (GERAERTS; SCHAGER, 2010; MARZOUQI; JARVIS, 2003; JOHANSSON; DELL'ACQUA, 2010), which is a special path planning problem where the final path must balance distance traversed and covertness. To our knowledge, there has been no work related to the application of machine learning techniques to stealth games in order to create an artificial controller for stealthy agents. This represents a major setback for stealth games due to the lack of techniques capable of controlling a stealthy agent.

We present herein a reinforcement learning formulation for stealth games. We aim to allow stealthy agents to successfully interact with the environment in a believable and realistic way, adopting human-like strategies. We built a stealth game simulator that allows the creation of random environments with several obstacles and populated by a set of patrols. This simulator reproduces the basic concepts of a real stealth game, such as allowing the agent to hide from enemies and use stealthy path planning to reach a destination without being spotted by enemy patrols. Our objective is to propose a general approach for stealth games that can be adapted to real games.

Our Reinforcement Learning approach uses high-level actions in order to interact with the environment. Therefore, it is necessary to create specific algorithms for each of the possible actions. The most important of them is to allow the agent to move covertly through the environment. Hence, we also propose here a stealthy path planning approach which uses simple data structures commonly used in game development, namely, A* algorithm and Navigation Mesh. We tested three different reinforcement learning methods for our stealth game simulator: Q-Learning, Sarsa(λ), and Dyna-Q. The remaining domain-related aspects of the reinforcement learning formulation are also described and justified.

The second problem tackled in our work is related to the complexity of defining some parameters for reinforcement learning applications, such as: action set, state formulation, training parameters, and reward function. These parameters are usually defined based on an expert's knowledge, but this approach may prove infeasible for more complex problems. Thus, we propose an evolutionary approach in order to search for a good reward function without any prior knowledge of the problem.

1.2 CONTRIBUTIONS

Our research presents several contributions to different areas. The first contribution is an efficient stealthy path planning approach that uses the Navigation Mesh and A* algorithm, which are largely used in the gaming industry. Our second contribution is the modeling of a stealth game problem for a reinforcement learning approach in order to allow the stealthy agent to behave realistically in a general environment. Our third and last contribution is a proposal for the automatic definition of a reward function for reinforcement learning methods without any prior knowledge of the problem.

1.3 STRUCTURE

This dissertation is divided into 5 chapters. Chapter 2 presents the stealthy path planning problem. We also present our stealthy path planning approach and how it compares to one of the current state of the art algorithm. Chapter 3 shows how we can improve the odds of success of a stealthy agent by using Reinforcement Learning in conjunction with our stealthy path planning method. We then show in Chapter 4 how we can automate some aspects of our Reinforcement Learning formulation by using Evolutionary Computing. Finally, we present our conclusions and discussions in Chapter 5.

2 STEALTHY PATH PLANNING

This chapter will address the stealthy path planning problem in video games and virtual simulations. We start by presenting some of the related work. We then follow by describing the basic concepts of our stealth game simulator, which is the testing environment for our entire work. In the sequence, we detail an approach from the literature called the Dark Path Algorithm, used here to compare with our proposed method. We then describe our proposed stealthy path planning method. Finally, we present the experimental results obtained by using our method and the Dark Path Algorithm in the stealth game simulator.

2.1 RELATED WORK

Stealthy path planning is used in several different applications that cover three main domains: robotics (MARZOUQI; JARVIS, 2003; PARK et al., 2009; JARVIS, 2004), military vehicles (Unmanned Aerial Vehicle) (BORTOFF, 2000; MOHAN et al., 2008; WANG et al., 2010; HE; DAI, 2013), and video games (GERAERTS; SCHAGER, 2010; TREMBLAY et al., 2013; JOHANSSON; DELL'ACQUA, 2010). The former domain focuses on creating special algorithms for autonomous robot navigation while minimizing the exposure to observers. The second domain focuses on finding covert paths for aircrafts and UAVs (Unmanned Aerial Vehicle) in order to avoid enemy radars. The latter comprises the domain of this work and it is focused on creating stealthy paths through a virtual environment that is patrolled by enemy sentries. Although these three domains tackle a similar problem, their different contexts result in different approaches.

Teng et al. (1993) presented one of the pioneering works for stealthy path planning that focused on determining a covert path for vehicle navigation in a digital elevation map. The environment is represented by a grid in which each position is associated to an elevation value. The terrain is also patrolled by several dynamic sentries. The goal is to find a path from a starting point S to a destination point D that isn't exposed to the sentries. The agent uses the different elevation values to avoid visual contact with the observers. It is considered that a complete path from S to D is not feasible due to uncertainties of the adversaries predicted positions, given that each sentry moves constantly around the environment. Thus, the authors use several subgoals in order to achieve the final destination. To find a subgoal it is first defined the reachable points from the agent's current position and then the points that are visible to nearby observers are removed. The subgoal is then chosen to be the safest point reachable within a time step, where safety is defined as the nearest point to the destination which has a good observability of nearby observers, in order to update their predicted positions. The authors later extended their work by using parallel computing in order to calculate a stealthy path for multiple agents (TENG et al., 1992). Another approach for stealth-based path planning in digital elevation maps is presented by Ravela et al. (1994). The latter work uses harmonic potential fields in order to define a stealthy path. A method for reconstructing elevation terrains from images is also presented, and it is used for defining the terrain characteristics in real time.

When calculating covert paths for UAVs, it is usually necessary to avoid enemy radars. Thus, the sentries are considered static, given that a radar does not change its position. Also, the environment does not present any obstacle, since UAVs traverses through the air. To tackle this problem, Bortoff (2000) presents a method that builds a Voronoi diagram around each radar site and then calculates the minimum path in the resulting graph. The resulting path passes as far away as possible from the radars. Finally, virtual forces are applied to the path in an effort to create a smoother and more realistic path. A similar problem is addressed by Wang et al. (2010), where a stealthy path for an aircraft through a radar site is obtained by maintaining a constant azimuth during flight. Mohan et al. (2008) and He and Dai (2013) tackles a slightly different problem: exploring and covering a 3D world with multiple UAVs without being seen by enemy outposts. The former work determines the shorter stealthy path by maximizing a specific metric that balances greater terrain visibility and short distances traveled for all UAVs. The latter work uses Niched Genetic Algorithm (NGA) in order to determine the best shortest path that covers the entire environment.

Stealthy path planning is also widely used for mobile robots. In this type of application, the environment is usually represented by a 2D grid. One of the main work related to stealthy pathfinding for mobile robots is presented by Marzouqi and Jarvis (2003), in which the authors propose an algorithm for calculating stealthy paths in a 2D grid, called *Dark Path Algorithm*. This algorithm uses the base concept of the Distance Transform algorithm (JARVIS, 1984), which calculates the shortest path from one point to every other point in a 2D grid world, and a visibility map that indicates, for each grid cell, how many other cells has a clear view to it. The Dark Path Algorithm extends the Distance Transform by adding a visibility value to each cell in the grid when calculating the best path. Thus, that algorithm searches for a path that balances visibility and shortest distance. Jarvis (2004) later proposed an improvement to his former work (MARZOUQI; JARVIS, 2003) by adding enemy sentries in the environment. To deal with unknown environments, Marzouqi and Jarvis (2003) proposed an approach using the Dark Path Algorithm. It is considered that the robot is equipped with a sensor capable of detecting obstacles, and for each discovered obstacle the environment grid is updated, followed by an update to the visibility map. Stentz (2002) presents a variation of the D* algorithm (STENTZ, 1994), called CD* algorithm, for stealthy pathfinding for robots in unknown environments. The CD* algorithm considers that the robot is also equipped with sensors for detecting obstacles and each new obstacle detected updates the environment map. Birgersson et al. (2003) and Tews et al. (2004) addresses the problem of stealthy path planning in unknown environments by using potential fields. The environment is also represented by a 2D grid and the robot is also capable of detecting obstacles with its sensors. Each new obstacle generates a shadow area, which comprises the unobservable area by the sentry. The destination point and each shadow area are assigned as an attractive force and the obstacles generate a repulsive force. The latter work is also used in dynamic environments with moving obstacles and the robot is allowed to perform two basic actions: move through the potential fields or wait until a stealthy path is found. Finally, Marzouqi and Jarvis (2005) extends his Dark Path Algorithm to deal with unknown sentries location.

Virtual simulations and video games are also an important domain related to stealthy path planning, being one of the three domains which has received greater attention in the last years due to its demand for virtual agents with covert behaviors. That type of application usually considers that the environment is known a priori. Johansson and Dell'Acqua (2010) proposes a stealthy path planning in a 2D grid which presents several novel ideas that were used in our work. Path planning is performed by using the A* in a quad-tree which represents the environment. The environment is patrolled by several enemy sentries and their previous positions are stored in a probability map, which indicates the predicted areas being patrolled. A visibility map is then built through the probability map. The visibility map is used to weight the quad-tree transitions and guide the A* in order to find a covert path that passes through low visibility areas. Furthermore, the virtual agent adopts a higher speed when passing through visible areas and a lower speed when in shadowed areas. A different approach is proposed by Geraerts and Schager (2010) where a special structure called corridor maps (GERAERTS; OVERMARS, 2007) is used in order to calculate the visibility map of the environment. The A* method is then used on a special graph generated from the corridor map structure, allowing it to be faster when compared to an A* performed on a high resolution 2D grid. This method is thus capable of finding a covert path in an environment patrolled by several dynamic sentries in a static and known world. Also, the method was implemented to run on graphic cards (GPU) in order to allow it to be executed in real time.

Other work addresses different problems related to stealthy behavior that are equally important to the path planning phase. One important aspect of stealthy agents is not only the path chosen by it, but also how it performs this path. A stealthy path must not only avoid enemy sentries, but also behave covertly by passing through cover areas whenever possible. Rook and Kamphuis (2005); Kamphuis et al. (2005) and Coleman (2009) present two different approaches for stealthy aesthetics in covert paths. The former proposes a stealthy path planning method that calculates a path in a special structure called roadmap corridors. The path planning considers each vertex of the polygons which comprises the roadmap corridor and weights each point considering their exposure to enemies, proximity to a wall, and distance to an enemy threat. Thus, the resulting path passes near walls and covers, even when no threats are near. The latter work uses fractal models to modify a minimum path, in order to guarantee that it passes through cover areas and near walls whenever possible. It is important to note that this fractal model does not consider enemy sentries in the environment. Other approaches related to stealthy behaviors are also present in the literature and addresses other problems, for example, predicting stealthy behaviors in specific level designs in order to aid level designers to build viable environments for games (TREMBLAY et al., 2013). Although useful for stealthy behaviors and for game developers, these approaches are not the focus of our work and, thus, are not discussed further.

2.2 STEALTH GAME SIMULATOR

We created a simple stealth game simulator in order to perform the experiments and validate our approach. The simulator was built using C++ with the OpenGL library (SEGAL; AKELEY, 1994) and simulates a square shaped environment with several rectangular shaped walls patrolled by a variable number of sentries. The simulator also presents a virtual agent that must traverse the environment from one starting point to a destination.

The environment's size, minimum and maximum size of the walls, minimum and maximum number of walls, and the number of sentries are all set as parameters for the simulator, allowing for a wide range of possible configurations. The size and number of walls are selected randomly between their respective minimum and maximum parameters. Each wall is placed in a random position, and each wall must be at a minimum distance from each other. If there are no free space for a wall to be placed, we ignore it. Thus, the resulting environment for each run in the simulator is unique, allowing our tests to consider a more general stealth game environment and not only a fixed set. Figure 2.1 depicts two possible configurations with different minimum number of walls, resulting in a greater or lesser number of walls in the environment. For our experiments, we used only environments with a size of 500 world units.

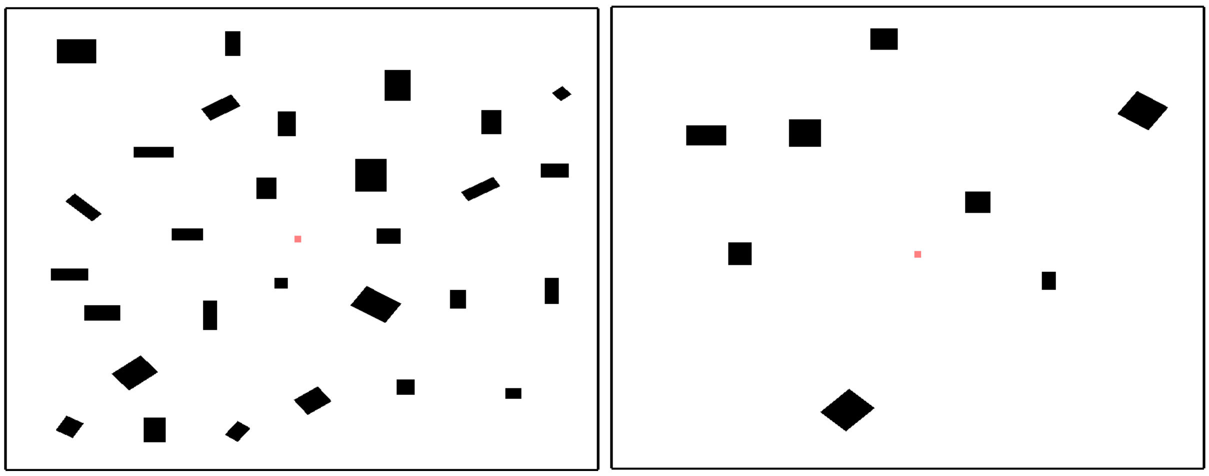
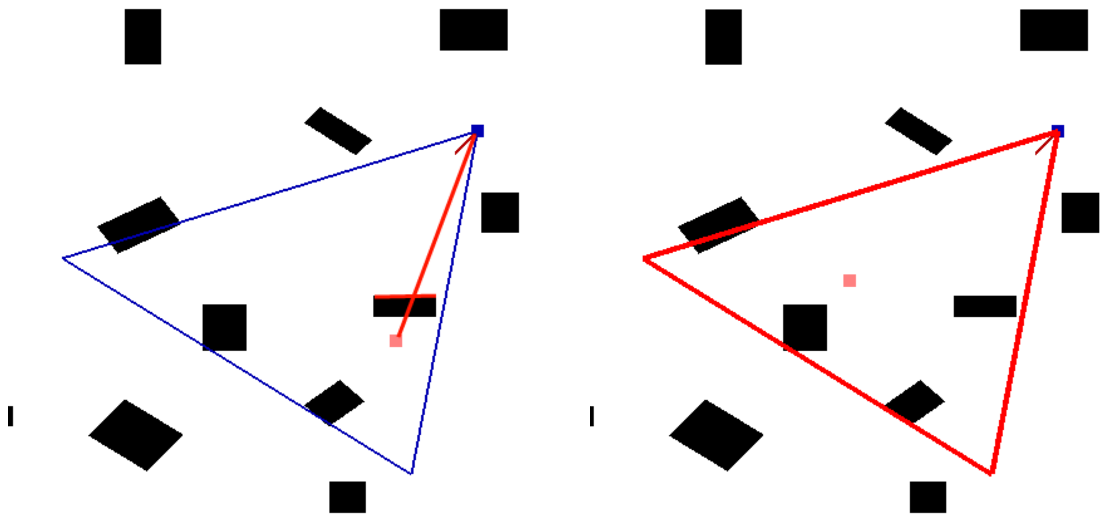


Figure 2.1: Different environment configurations. (a) shows an environment with a minimum number of walls set to 20, while (b) shows an environment with a minimum number of walls set to 1. The maximum number of walls for both environments was set to 100. The orange dot in both images represents the stealthy agent.

Each sentry moves around the environment between a starting point and a destination with a fixed speed through a minimum distance path. The starting and destination points for either the agent and sentries are randomly generated. Thus, the enemy sentries do not present any movement pattern. Also, the enemy sentries have a limited field of view that is shaped in a triangular form of 120 world units, as depicted in Figure 2.2. We have also implemented a simple system to detect if the sentry has a clear sight to the agent or if it is being occluded by an obstacle. To do this, we check if there exists an intersecting point between the line segment formed by the enemy's and agent's current position and any line segment which comprises the nearby walls. If at least one intersecting point exists, then the agent is being occluded by a wall and, thus, can not be seen by the enemy patrol (even when it is inside the sentry's field of view). This situation is illustrated in Figure 2.2.



(a) Agent hidden from the enemy (b) Agent detected by the enemy

Figure 2.2: The agent is capable of hiding from enemy patrols. (a) depicts a situation where the agent is hidden behind a wall. The red lines represent the intersecting line segments, one being the line formed by the agent's and patrol's location and the other is a wall's line segment. (b) shows a situation where the agent is detected by the enemy patrol, since no wall intersects with the enemy's line of sight.

Further informations regarding the simulator are described through the remaining of the text as they are deemed necessary.

2.3 DARK PATH ALGORITHM

The Dark Path Algorithm was developed by Marzouqi and Jarvis (2003) and later expanded in several other work of the same author (JARVIS, 2004; MARZOUQI; JARVIS, 2003, 2005, 2006). It represents one of the state of the art methods related to stealthy path planning and is used here for comparisons. The Dark Path Algorithm is used to determine a covert path from a starting point to a destination point in a 2D grid environment patrolled by several dynamic sentries. Although originally proposed for mobile robots, that method can be extended and adapted for video games and virtual simulations. In order to adjust the Dark Path Algorithm to our simulator, presented in Section 2.2, we used a 2D grid to represent the environment. Therefore, the agent is only allowed to move to one of its 8 neighbor cells. The grid is comprised of free spaces and obstacles, allowing each cell to assume one of these two types.

The Dark Path Algorithm is an extension of the Distance Transform (DT) method (JARVIS, 1984), a global path-planning algorithm capable of finding the shortest path from one cell to every other cell in a grid-based map. Each cell is associated with a distance value that indicates how far this cell is to the destination. Initially, the distance value of the destination cell is set to zero and the distance value of every other cell is set to a very large number. Then, the distance value of each free cell is updated according to

$$d(c) = min[d(c),\ min_{i=1}^{8}[d(n_i) + ED(n_i, c)]] \tag{2.1}$$

where it is associated the lowest distance value of the cell's neighbors plus the distance to reach its neighbor, but only if this value is lower than its actual distance value. In Equation 2.1, $d(c)$ represents the distance value of cell c, n_i represents the neighbor i of cell c and $ED(a, b)$ represents the Euclidean Distance from cell a to cell b. The min function returns the minimum between two values and $min_{i=1}^{8}$ returns the minimal value between the eight neighbors of cell c. The authors considered the distance from one cell to their horizontal or vertical neighbors equal to 1 and the distance to their diagonal neighbors equal to $\sqrt{2}$, approximating it to the Euclidean distance. The distance value of the obstacle cells are not updated. That update procedure is performed in what the authors call forward and reverse rasters, in order to reduce the complexity of the algorithm (MARZOUQI; JARVIS, 2003). The forward raster follows from left to right and from top to the bottom. The reverse raster follows from right to left and from bottom to the top. This process follows until no distance value is altered. The code for the Distance Transform is presented in Algorithm 1 (we consider the starting and ending indexes of a B sized array to be 0 and B-1, respectively). In Algorithm 1, the variable *Grid e* represents the environment, where $e[i,j] = 1$ if $cell(i,j)$ is an obstacle and $e[i,j] = 0$ otherwise. The forward pass in Algorithm 1 starts at line number 4 and ends at line number 10, while the backward pass starts at line number 11 and ends at line number 17. The variable N represents the size of the grid.

Algorithm 1 Distance Transform (Grid e, Cell goal)

```
 1: d[goal.x, goal.y] $\leftarrow$ 0;
 2: d[c.x, c.y] $\leftarrow$ 99999999 $\forall$ c $\neq$ goal;
 3: repeat
 4:   for $i = 1$ to $N - 1$ do
 5:     for $j = 1$ to $N - 1$ do
 6:       if e[i, j] = 0 then
 7:         d[i, j] $\leftarrow$ $min[d(c_{i,j}),\ min_{i=1}^{8}[d(n_i) + ED(n_i, c_{i,j})]]$;
 8:       end if
 9:     end for
10:   end for
11:   for $i = N - 2$ to 0 do
12:     for $j = N - 2$ to 0 do
13:       if e[i, j] = 0 then
14:         d[i, j] $\leftarrow$ $min[d(c_{i,j}),\ min_{i=1}^{8}[d(n_i) + ED(n_i, c_{i,j})]]$;
15:       end if
16:     end for
17:   end for
18: until d is not moddified;
19: return d;
```

The Distance Transform method is then used for generating a Visibility Map. The visibility map is responsible for indicating, for each cell, how many other cells are visible from it. High visibility values indicate that a cell is visible from many other cells. Lower visibility values indicates that a cell is more concealed. The visibility value for each cell is determined by executing the Distance Transform algorithm with the corresponding cell as the destination for an obstacle-free environment and then for the original environment with obstacles. Then, we compare the distance map calculated for the obstacle-free environment and for the original environment. If the distance from the target cell to other cell is greater in the original environment when compared to the obstacle-free environment, then the pair of cells are not visible to each other, since this difference indicates that there is an obstacle between these cells.

Jarvis (2004) indicated that certain anomalies occurs on tessellated spaces (such as our grid map) when calculating the visibility map. The main problem occurs when the distance value between two cells for the obstacle-free and original environments are the same but these cells are concealed from each other, that is, the straight line connecting the two cells intersects with a wall. This happens due to the diagonal moves, which creates a non-straight shortest path between two cells in the obstacle-free environment, allowing paths to go around small edges of walls without loosing sight of the destination cell. Thus, Equation 2.2 was proposed in order to determine if a cell is visible from the target cell.

$$Vis(i,j) = \begin{cases} 1, & \text{if} \dfrac{DT_{free}(i,j)}{DT_{original}(i,j)} > \beta \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

In Equation 2.2, $Vis(i,j)$ indicates if cell i is visible from cell j ($Vis(i,j) = 1$ if cells i and j are visible to each other and $Vis(i,j) = 0$ otherwise), $DT_{free}(i,j)$ represents the distance from cell j to cell i in the obstacle-free environment, $DT_{original}(i,j)$ represents the distance from cell j to cell i in the original environment and β is the *visibility coefficient*. When β is set to a value equal or greater than 1, then no cell will be visible to each other, since $DT_{original}(i,j) \geq DT_{free}(i,j)$. Figure 2.3 shows four visibility maps with different values of β and for different environments. Here, we have chosen $\beta = 0.99$. The code for generating the visibility map is presented in Algorithm 2, where the variable v indicates the visibility value for each cell of the environment.

By analyzing the Visibility Map Algorithm, presented in Algorithm 2, we notice that its time complexity depends of the grid dimension N, where the number of cells in the grid is given by N^2. The asymptotic complexity of this algorithm is given by $O(N^4)$. We have chosen $N = 100$ due to the algorithm's high complexity. Thus, our experiments were performed on a 100x100 grid environment.

After calculating the Visibility Map, the visibility value of cells observed by enemy sentries are updated to a very high value. This make these cells behave almost like obstacles for the Dark Path Algorithm, although the agent is still capable of passing through these cells, unlike what happens for obstacle cells. The original idea proposed by the authors was to use the visibility algorithm for each sentry location to determine which cells are visible to them. But, in order to adapt this algorithm to our simulator

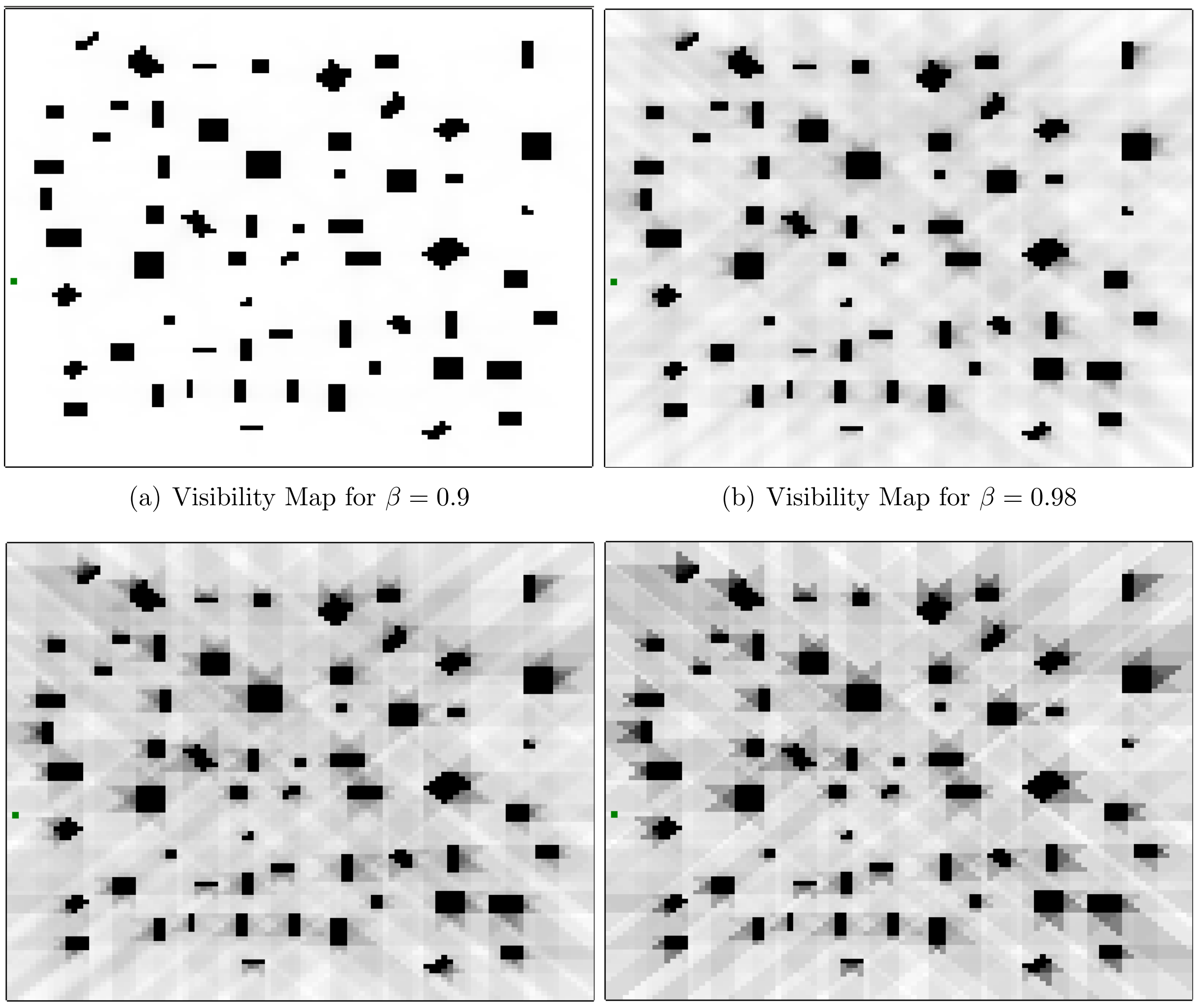
Algorithm 2 Visibility Map ()

```
1: v[x][y] ← 0 $\forall\ x, y$;
2: g1 ← obstacle-free environment grid;
3: g2 ← original environment grid;
4: for $i = 0$ to $N - 1$ do
5:   for $j = 0$ to $N - 1$ do
6:     if g2[i][j] is not an obstacle then
7:       d1 ← Distance Transform (g1, cell(i, j));
8:       d2 ← Distance Transform (g2, cell(i, j));
9:       for $a = 0$ to $N - 1$ do
10:        for $b = 0$ to $N - 1$ do
11:          if d1[a][b]/d2[a][b] $> \beta$ then
12:            v[i][j]++;
13:          end if
14:        end for
15:      end for
16:    end if
17:  end for
18: end for
19: return v;
```

and allow a fair comparison to our method, we use the visibility method presented in Section 2.2 (illustrated in Figure 2.2) to determine which cells are visible to each sentry. For each sentry, we use this method for every cell in order to determine which cells are inside the corresponding sentry's field of view and are not occluded by obstacles. The resulting visibility map with the enemy sentries is presented in Figure 2.4.

The Dark Path Algorithm uses informations regarding the distance and visibility values of each cell in order to determine a stealthy path to the destination. The algorithm starts by initializing the cost of the destination cell to its visibility value and a high number for the other cells. After initializing the cost value of every cell, we use the same steps used for the Distance Transform: update the cost value of each cell to the minimum value between its actual value and the cost of its neighbors summed with the Euclidean Distance and the visibility value of the corresponding neighbor. Marzouqi and Jarvis (2006) also use a stealth coefficient, η, to control the importance given to the visibility value of a cell. Higher η values results in paths which prioritize stealth over the distance traveled. The cost value update function is given by

$$CV(c) = min[CV(c),\ min_{i=1}^{8}[CV(n_i) + ED(n_i, c) + \eta V(n_i)]] \tag{2.3}$$


(a) Visibility Map for $\beta = 0.9$

(b) Visibility Map for $\beta = 0.98$

(c) Visibility Map for $\beta = 0.99$

(d) Visibility Map for $\beta = 0.999$

Figure 2.3: Visibility Maps generated with different β values in a 100x100 world grid. Black cells represent obstacles and the other gray-shaded cells represent free spaces, where darker cells are assigned to less visible cells. Since our environment presents several small obstacles, going around the obstacles takes only a few steps, that is $DT_{original}(i, j)$ is only slightly bigger than $DT_{free}(i, j)$. Thus, for $\beta = 0.9$, every cell can see every other cell, resulting in completely white free cells. When comparing the visibility maps with $\beta = 0.98$(b) and $\beta = 0.99$(c), we note that (c) presents less visible cells. For the visibility maps presented in (c) and (d) one can notice that there are no significant differences. The green dot represents the agent's location.

where $CV(c)$ represents the cost value of the Dark Path Algorithm for cell c and $V(c)$ represents the visibility value for cell c. It is important to note that η may assume small values, given that $V(c) >> ED(i, c)$. Figure 2.5 depicts the resulting paths for different values of η in a 100x100 grid. We have chosen $\eta = 10$, due to its greater stealth behavior.

The update process is performed using the forward and reverse rasters, just like the Distance Transform. This update process continues until no further changes occurs in the

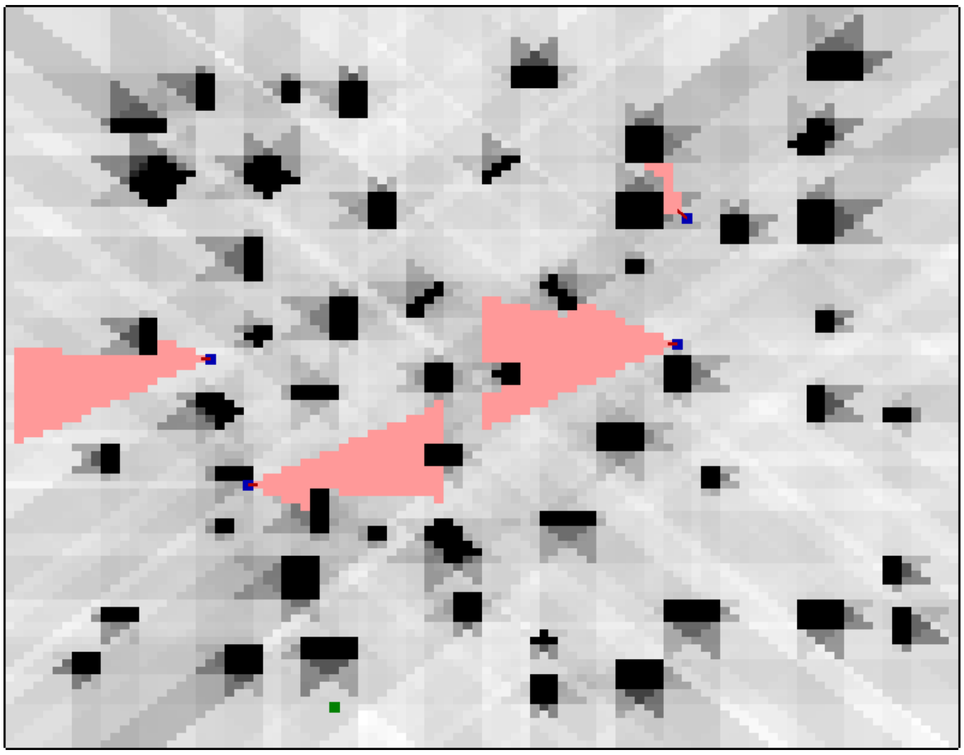


Figure 2.4: Final Visibility Map generated for a 100x100 grid and $\beta = 0.99$. The red cells are the ones visible by the enemy sentries. The blue dots represent the enemy positions.

cost value for every cell, that is, when Equation 2.3 returns only $CV(c)$ for every cell. The Dark Path Algorithm is presented in Algorithm 3.

2.4 STEALTHY PATH PLANNING USING NAVIGATION MESH

The quality of a path is directly related to how the environment is mapped and presented to the agent. By using a 2D grid, the agent is only allowed to move in eight possible directions, resulting in an unrealistic movement. Also, 2D grid representations are rarely used in modern commercial games due to the resulting path's quality and memory usage. Thus, we use Navigation Meshes to represent the environment, since this structure is widely used in commercial games and is able to efficiently represent a large environment with lower memory usage. We created a simple algorithm for generating special navigation meshes adapted to our stealthy path planning method.

In this section we present our proposed stealthy path planning method. We start by describing our algorithm for generating a special navigation mesh. In the sequence, we describe our stealthy path planning method and how it is performed in real time.

Algorithm 3 Dark Path (Grid e, Cell goal)

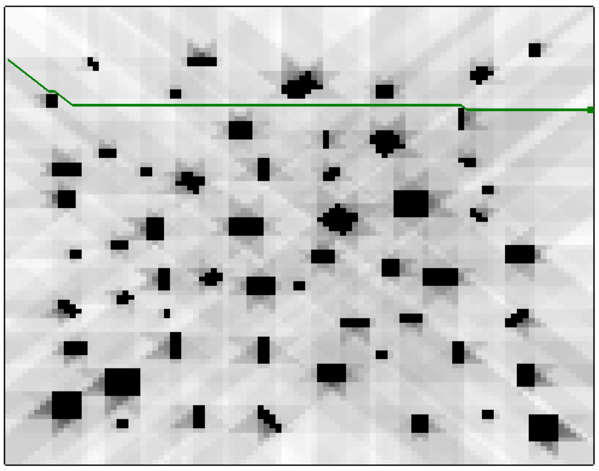
```
1: CV[goal.x, goal.y] ← V[goal.x, goal.y];
2: CV[c.x, c.y] ← 99999999 ∀ c ≠ goal;
3: repeat
4:    for $i = 1$ to $N - 1$ do
5:       for $j = 1$ to $N - 1$ do
6:          if e[i, j] = 0 then
7:             CV[i, j] ← $min[CV(c_{i,j}),\ min_{i=1}^{8}[d(n_i) + ED(n_i, c_{i,j}) + \eta V(n_i)]]$;
8:          end if
9:       end for
10:   end for
11:   for $i = N - 2$ to 0 do
12:      for $j = N - 2$ to 0 do
13:         if e[i, j] = 0 then
14:            CV[i, j] ← $min[CV(c_{i,j}),\ min_{i=1}^{8}[d(n_i) + ED(n_i, c_{i,j}) + \eta V(n_i)]]$;
15:         end if
16:      end for
17:   end for
18: until CV is not moddified;
19: return CV;
```

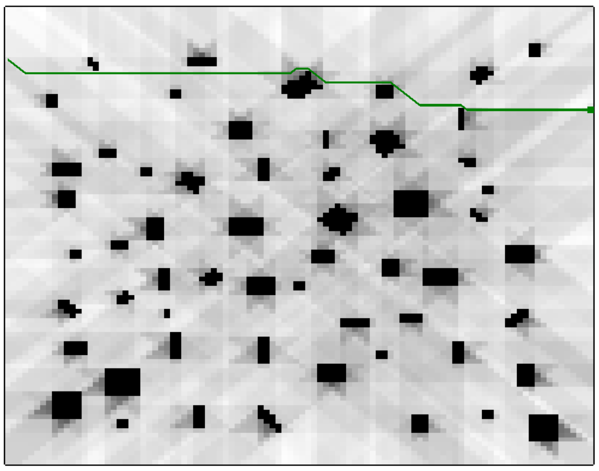
2.4.1 NAVIGATION MESH GENERATION

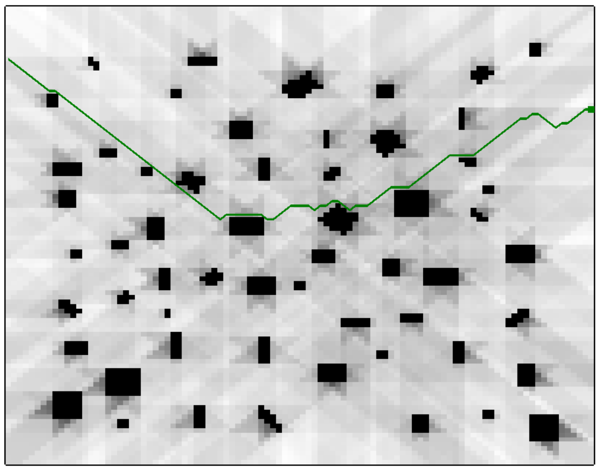
A navigation mesh is a set of convex polygons used to represent the walkable areas of an environment. Each polygon of the navigation mesh represents a traversable area where the agent can move freely inside. The agent is also allowed to move between adjacent polygons. Thus, we can build a graph that represents the connections between the navigation mesh's polygons. It is over this graph that we apply our stealthy path planning method.

Each polygon of a navigation mesh may be associated with some specific terrain features (water, forest, snow, etc.) which allows for an increased complexity of the path planning. We use a special navigation mesh with two types of polygons: (i) cover polygons and (ii) normal polygons. The cover polygons represent cover areas around an obstacle and the normal polygons represent the remaining free area. Figure 2.6 shows the two types of polygons present in our navigation mesh.

Building a navigation mesh means defining every polygon that comprises it. This can be done manually (a task usually accomplished by game designers during the environment creation step) or by using special algorithms for generating a navigation mesh automatically (KALLMANN, 2010; HALE et al., 2008). The literature doesn't present any algorithm for generating a navigation mesh with the characteristics necessary for our


(a) Path for $\eta = 0$


(b) Path for $\eta = 0.0001$


(c) Path for $\eta = 1$

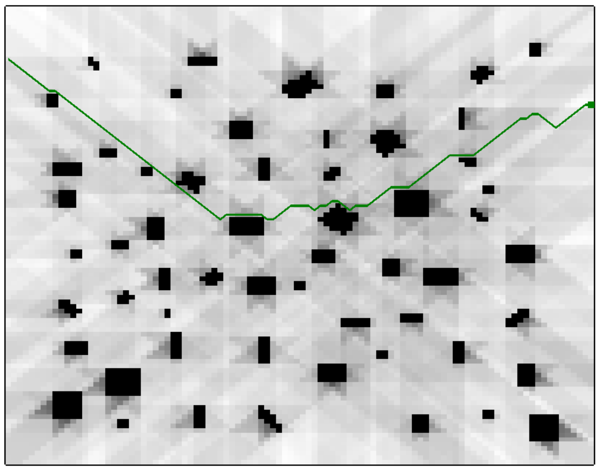

(d) Path for $\eta = 10$

Figure 2.5: Resulting paths for different stealth coefficient values. When $\eta = 0$ (a), the resulting path doesn't consider the visibility values and, thus, depicts a similar result as the original Distance Transform Algorithm. For $\eta = 0.0001$ (b), the resulting path prioritizes the distance over visibility values. The resulting paths for $\eta = 1$ and $\eta = 10$ presents a similar stealthy behavior, showing that values for η greater than 1 results in a similar behavior for a 100x100 grid environment. Both paths use cover in order to avoid high visibility areas.

method. Thus, we propose here an algorithm for creating a simple navigation mesh with the characteristics needed for our covert pathfinding method that receives the environment size and a list of obstacles as input. The procedure starts by creating the cover polygons. Since in our simulator all of the obstacles are rectangles, then 8 rectangles are created around each obstacle. For the normal polygons, we use a sweeping technique that starts by checking, from left to right, the first cover polygon encountered. A rectangle that covers the entire area is then created from the starting point of the sweep to the cover polygon found. The upper and lower regions of the cover polygon are swept following the

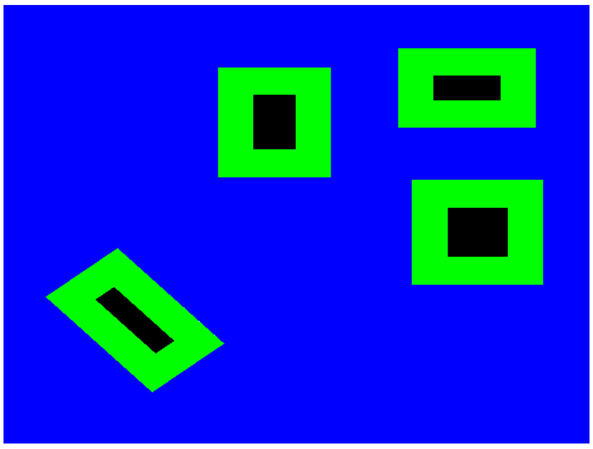
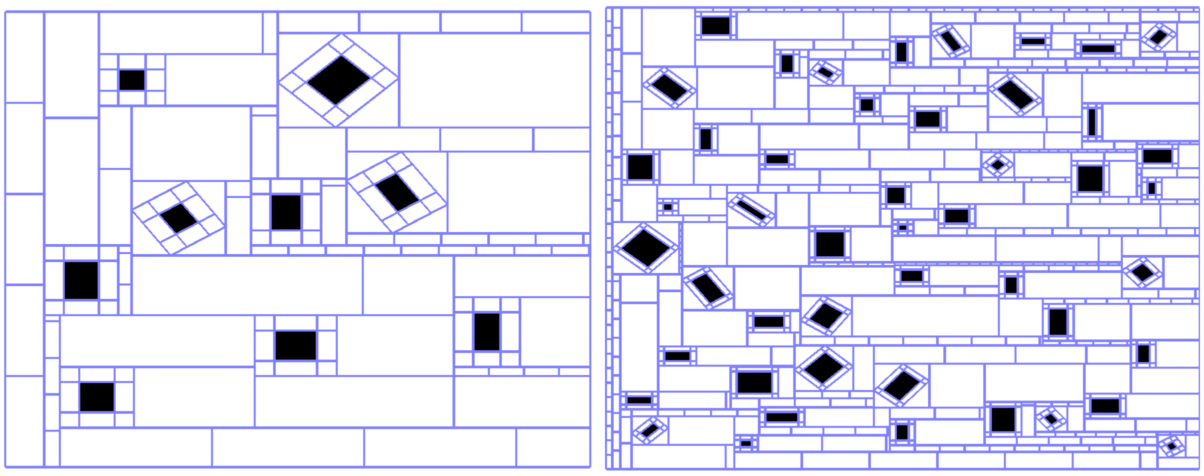


Figure 2.6: Polygon types. The obstacles are represented in black, the cover polygons are represented in green and the normal polygons are shown in blue. The exact boundaries of each polygon is not shown in this figure. It is important to note that the obstacles are not included in the navigation mesh.

same procedure. This is repeated until the end of the environment is reached and, thus, no more sweeps are executed.

The presence of stretched rectangles on the navigation mesh can influence the overall quality of the calculated paths, since these polygons present disproportional sizes. Thus, we don't allow the creation of rectangles with one side greater than three times the other side. This way, when a polygon created presents this characteristic, then it is broken into several smaller rectangles, contained within the original rectangle, that follows this rule. The resulting navigation mesh is presented in Figure 2.7.



(a) Navigation Mesh with 155 polygons (b) Navigation Mesh with 1004 polygons

Figure 2.7: Navigation mesh generated over a random terrain. (a) presents a navigation mesh with 155 polygons and (b) presents a navigation mesh with 1004 polygons.

Path planning techniques can't be used directly in a navigation mesh. Thus, a proper structure is required to execute the proposed stealthy path planning method. For this purpose, we create a graph that represents the connections between each polygon. The graph's vertexes represent each polygon and each edge represents the connection between two polygons. Each vertex is positioned at the center point of its corresponding polygon. Figure 2.8 shows the resulting graph of a random environment.

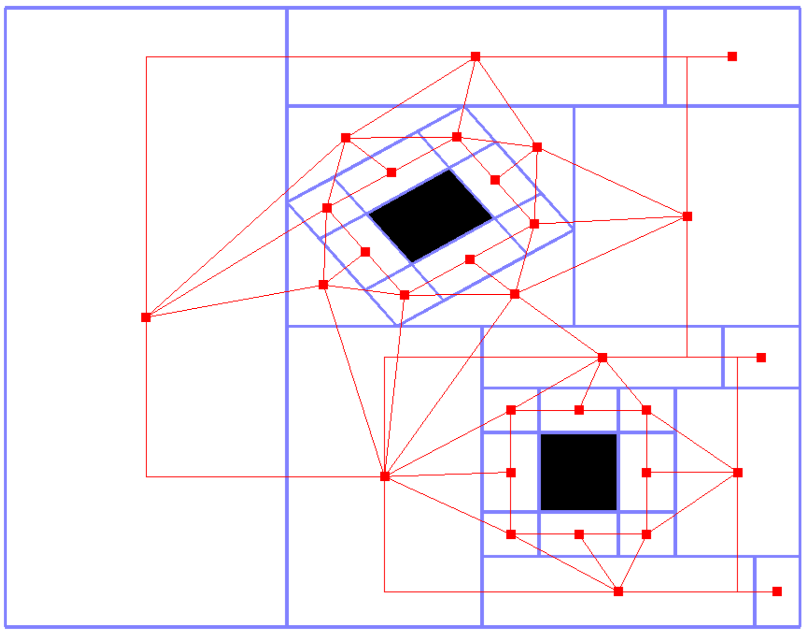


Figure 2.8: Graph of a navigation mesh built over a random terrain. Each node of the graph is given by a polygon (represented by the red dots). If two polygons are connected, then an edge is created between their respective nodes (represented by the red lines).

2.4.2 STEALTHY PATH PLANNING

Moving stealthily through an environment means moving between cover areas without being detected by patrolling agents. This behavior is usually observed by human players in stealth games, even when there are no patrols around. The Dark Path Algorithm also depicts this behavior, as shown in Section 2.3.

Our algorithm presents three main steps: (i) generating an initial stealthy path without considering the enemy patrols, (ii) path smoothening process in order to generate realistic paths, and (iii) path update in order to avoid the enemies. We use the A* algorithm (HART et al., 1968) over the graph presented in Figure 2.8 in order to calculate a stealthy path. The A* algorithm was chosen due to its efficiency, simplicity, and popularity for commercial games, making our approach more appealing for game developers. We then use a B-Spline in order to smooth the resulting path.

The weight of each edge of the graph represents the distance between the central points of the polygons (vertexes) that comprises the corresponding edge. We also apply penalties over an edge's weight in order to guarantee that the resulting path passes through cover areas. The penalties are applied according to the following rules:

- **Transition between normal areas:** penalty of 3 times the actual weight, since it is not desirable for the agent to walk in open areas;
- **Transition from normal to cover or from cover to normal areas:** penalty of 3 times the actual weight, since the distance walked in open area is generally greater than the distance walked under cover, due to the small size of cover polygons;

The heuristic method used for the A* algorithm represents the Euclidean Distance from the central point of a polygon to the destination point. This heuristic is admissible and thus guarantees the best path (based on the penalties and transition weights presented previously).

By using the mentioned penalties, the resulting path balances small distances with the low visibility. The amount of covertness can be adjusted by altering the penalties, allowing the simulation of several different behaviors. The output of the A* algorithm is a sequence of polygons to be visited, where the first polygon contains the starting point and the final polygon contains the destination point. Figure 2.9 shows the polygon path (green polygons) calculated in a random environment without enemy patrols, where the agent passes through covers in order to reach its goal.

After building a low resolution path formed by a sequence of polygons, a more accurate path must be generated in order to define the exact points which the agent will pass. We generate the exact path by using a quadratic B-Spline. B-Splines have already been used to generate smooth paths in another work from the literature (JUNG; TSIOTRAS, 2008).

The control points of the B-Spline are positioned according to the low resolution path outputted by the previous step. First, two control points are placed in the starting point and two control points are placed in the destination point. We use two points in these situations because quadratic B-Splines don't pass exactly through their first and last control points (and for any other control point). By placing two control points at each end of the B-Spline, we force it to pass at the starting and ending points. The

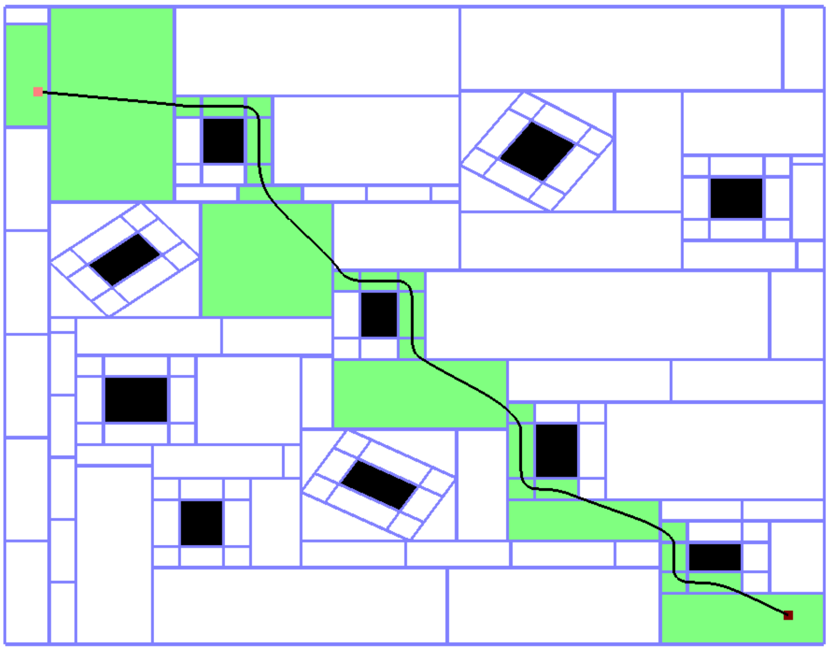


Figure 2.9: Resulting path in an enemy free terrain. The green polygons represent the path determined by the A* method and the black curve represents the smoothed path. The agent is represented by the orange dot at the top-end of the curve.

remaining control points are placed at a position inside each of the following polygons in the low resolution path, where the selected position minimizes the distance from the previous control point plus the distance to the central point of the next polygon (if the next polygon does not exist, then we consider the distance equal to zero). The tested positions inside each polygon were: each vertex of the polygon, the central point of the polygon, and the median points of each edge of the polygon. The B-Spline is represented by a set of 150 points. Figure 2.9 presents a smoothed path generated in a random environment. Figure 2.10 shows a smoothed path built using a B-Spline and its control points positioned along the polygons that belong to the path.

So far, we have detailed how to create a stealthy path in an enemy-free environment. In an environment patrolled by one or more enemies, it is also necessary to avoid their line of sight and stay at a safe distance from them in order to achieve a stealthy behavior. To avoid paths that passes close to an enemy, we penalize polygons whose center point is closer to an enemy within a 150 world units radius when compared to the center point of the actual polygon. Thus, the edge that connects the actual polygon and the penalized polygon is penalized by multiplying its current weight by 2. This penalty is applied separately for each enemy sentry. This penalization is responsible for adjusting the path to pass as further as possible from enemies.

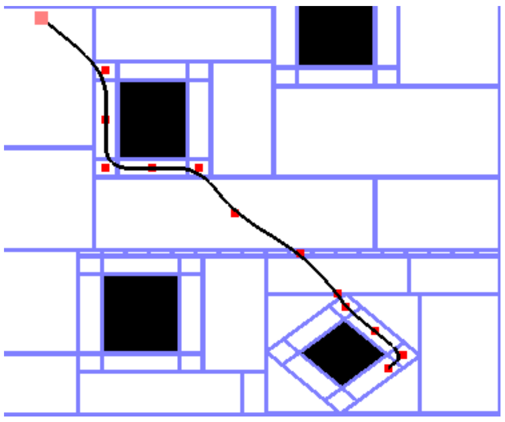


Figure 2.10: Smoothed path using a B-Spline. The red dots represent the control points of the B-Spline.

After building a path, we test each of the discretized points of the B-Spline to see if at least one of them is in an enemy's field of view. The stealthy agent is capable of detecting enemies within a radius of 220 world units. Thus, we only check the discretized points that are within the stealthy agent's radar area. If at least one point is visible to an enemy in the vicinity, the transition that led the agent to that point is penalized. This transition can be found by searching for the closest control point of the B-Spline. The transition is then defined as the transition between the corresponding polygon of the closest control point to the next polygon in the path. Through empirical evaluation, we found that the same penalty should be applied to the the transition followed by the penalized transition, since the agent detection may occur not only because of a single transition, but by the union of two consecutive transitions. Each transition is allowed to be penalized three times. After the third penalty the given transition becomes inviable and it is disregarded of the path planning step. Figure 2.11 illustrates the process: the dark blue dot represents the point where the agent enters into the enemy sight. Then, it is determined that the transition that led to this situation is the transition from polygon 1 to polygon 2. Thus, a penalty is applied to this transition, followed by a penalization of the following transition, given by polygons 2 and 3.

After applying these penalties, the A* algorithm is re-executed considering the new weights. The A* may be executed several times until a viable path is found. It is important to note that not all paths are tested, since the penalties used can remove transitions between two polygons, that is, remove edges from the graph. Hence, by removing edges from the graph, we can separate it into two or more components, making it impossible for the agent to reach the destination that is located in another component. Thus, the agent cancels the search.

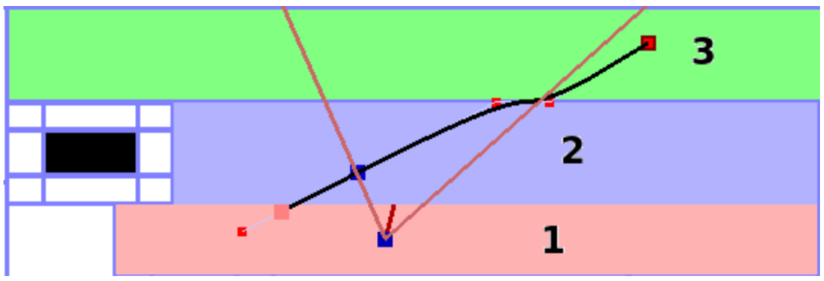


Figure 2.11: Illustration of the penalization process. The blue dot represents the patrolling agent and the orange dot is the stealthy agent. The dark blue dot represents the point where the agent became visible to the patrolling agent. A penalty is applied when the agent moves from polygon 1 to 2, followed by a penalty to the transition between polygon 2 to 3.

The resulting path is presented in Figure 2.12, where the black curve represents the stealthy agent's path and the red curves represent the paths of the patrolling agents. It is important to note that the patrolling agents move through a minimum cost path to reach their destination, as mentioned in Section 2.2. The agent's resulting path uses cover whenever possible, maintains a safe distance to enemy patrols, and stays out from their line of sight. This is all done in real time and by using simple and well known data structures and algorithms, such as navigation mesh and the A* algorithm.

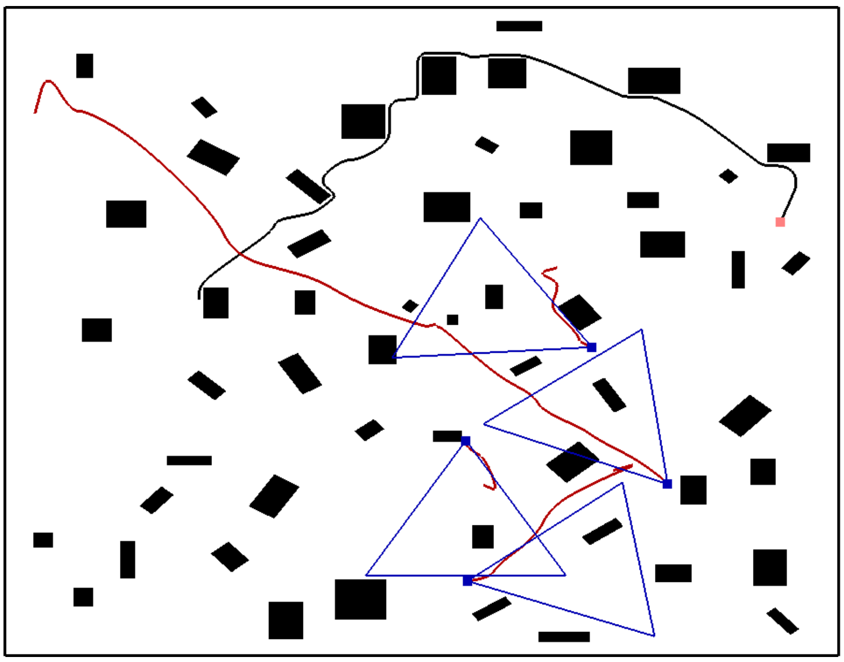


Figure 2.12: Path calculated in a random environment patrolled by 4 enemies. Each patrolling agent's path is represented by the red curve.

In this work it is not only considered the path planning phase but also the movement phase, in which the agent moves along the determined path. At each time step, the agent advances along the smoothed path with a given movement speed, which can vary between three possible values: sneaking, walking, and running. While moving, the agent emits footstep noises that depends on the current movement speed: the higher the speed, the greater the emitted sound.

Sneaking doesn't produce any sound and is used only when an enemy is too close to the stealthy agent. Walking produces a medium range sound with a 100 world units radius and it is used whenever the agent is in cover areas or when enemies are close enough to hear the running sound. Running is used whenever the agent traverses open areas and there are no enemies within a 170 world units radius. By moving faster in open areas, the agent avoids exposure and, by moving slower on cover areas, the agent spends more time in safe zones. Sneaking is used only in critical situations, when the agent is at a close range from an enemy. The agent always follows these restrictions, that is, it won't ever alert the nearby enemies through its footsteps noise. Each enemy sentry moves through the environment with a movement speed that is equal to 0.75 times the sneaking speed.

The stealth-based path planning method described previously uses enemies momentary locations to plan a viable path. Thus, the resulting path becomes obsolete as the agent and enemies move around the environment. This problem is bypassed by setting time intervals in which a new path is calculated, called the re-planning phase. Hence, when the path becomes outdated, it is replaced by a new and updated path that considers the latest changes in the environment. The chosen time interval is 0.07 seconds, defined based on empirical analysis.

2.5 EXPERIMENTAL RESULTS

We performed a set of experiments in order to verify the efficiency of the proposed stealthy path planning method, as well as how it performs in real-time executions. Every test performed used random environments, random moving patterns for the enemies, and random starting and destination points for the stealthy agent and for the enemies (as described in Section 2.2). The maximum number of allowed polygons in a mesh was limited to 2000, in order to guarantee real-time executions.

The number of enemies in the environment is strongly related to the stealthy agent's success rate. We performed executions of the proposed stealth simulator with different number of enemies in the environment in order to determine how the agent's success rate varies with the different number of enemies. We performed 40 sets of 1000 runs of the stealth simulator in order to achieve a success rate with a statistical certainty. The mean and standard deviation that resulted through these 40 sets of experiments were 66.9 and 1.62, respectively. Therefore, we can determine the agent's success rate through 1000 executions with a small error of ± 0.66 and a confidence level of 95%. The resulting success rate of the stealthy agent for different number of enemies through 1000 runs of the stealth simulator is presented in Table 2.1.

	1 Enemy	**2 Enemies**	**3 Enemies**	**4 Enemies**
Success Rate	95.0%	85.7%	78.7%	67.5%

Table 2.1: Results obtained after several executions of the stealth simulator in random environments with varying number of enemy sentries. It was executed 1000 executions for each fixed number of enemies.

Table 2.1 shows that the agent's success rate is related to the number of enemies in the environment, where the success rate increases as the number of enemies decreases. The agent's failures occur, most of the time, when the agent is surrounded by the enemies.

Our second experiment is to compare our stealthy path planning method with one of the state of the art methods present in the literature. Herein, we will compare our method with the Dark Path Algorithm, described in details in Section 2.3. We also tested how the number of enemies in the environment influences the Dark Path Algorithm's success rate. The environment's variables were set to the same values used for the tests of our stealthy path planning method. The only difference is the use of a grid (instead of a navigation mesh) for the Dark Path Algorithm due to that method's restrictions. We used a 100x100 grid for the experiments. Grids with higher dimensions were not used due to the excessive time demanded by the Dark Path Algorithm (see Section 2.3). We tested each method in 1000 different environments and retrieved their success rate. It is important to note that we used the same 1000 environments for both methods and for different number of enemies in order to perform a fair comparison between the methods. The comparison of the two stealthy path planning methods is presented in Figure 2.13.

Figure 2.13 shows that the proposed Navigation Mesh based method achieves higher success rates when compared to the Dark Path Algorithm. One of the main advantages of our method is that it allows the agent to move freely through the environment, without any

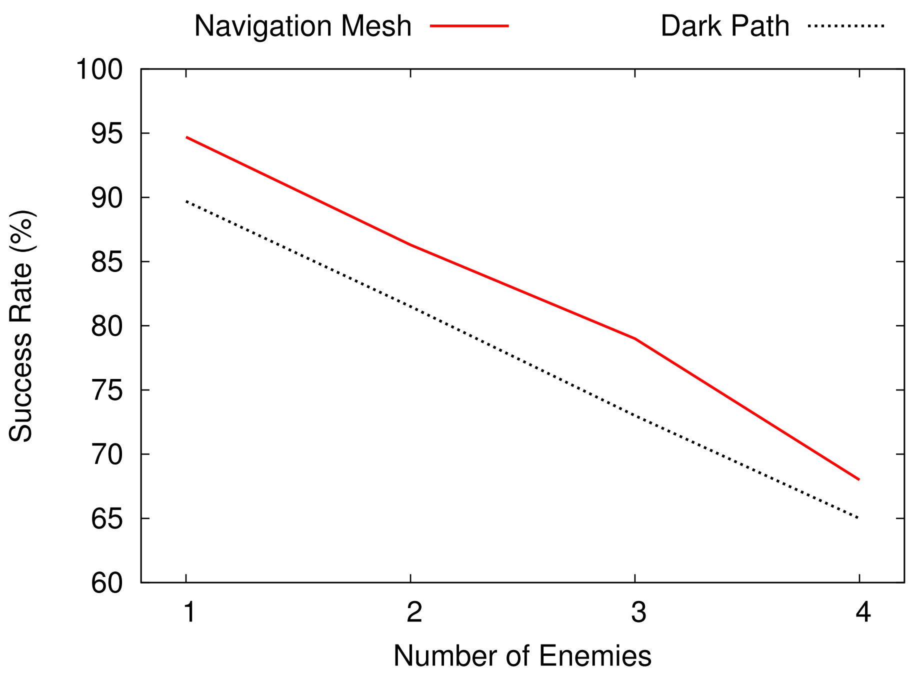
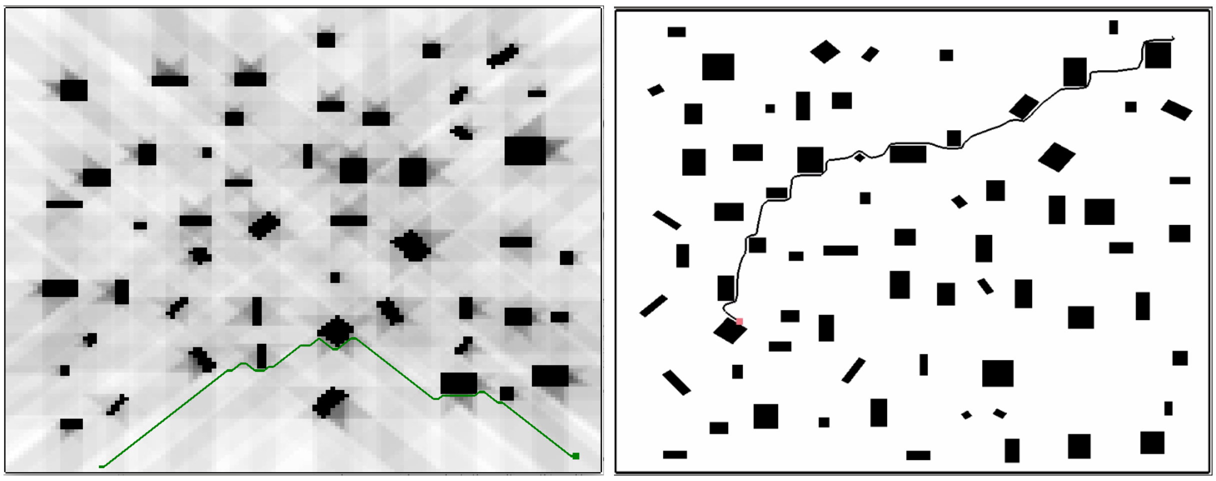


Figure 2.13: Comparison of the success rate for different numbers of enemies between our Navigation Mesh based stealthy path planning method and the Dark Path Algorithm.

movement restrictions. This allows the agent to traverse the environment more effectively. In contrast, the Dark Path Algorithm only allows the agent to move between grid blocks. Also, the Dark Path Algorithm considers the enemies' field of view as obstacles. Therefore, the resulting path doesn't try to maintain a safe distance to the enemies, which occurs in our method.



(a) Resulting path using the Dark Path Algorithm. (b) Resulting path using the Navigation Mesh approach.

Figure 2.14: Illustration of the resulting path generated by the Dark Path Algorithm, presented in (a), and by the proposed Navigation Mesh approach, presented in (b).

The main advantages of the proposed navigation mesh based stealthy pathfinding are the path's final quality and real-time performance. This method, as mentioned before, tries to maintain a safe distance to the enemies, just as a human player would. It also creates paths with realistic curves. These differences are presented in Figure 2.14. We also recorded two videos showing how the stealthy agent behaves by using different stealthy path planning techniques: the first video[1] shows the proposed method and the second video[2] shows the behavior of the Dark Path Algorithm.

Another important aspect of a stealthy path planning method is related to its performance. Our method uses the A* algorithm in a Navigation Mesh. Thus, the time complexity for finding a stealthy path is O(P + E), where P is the number of polygons in the navigation mesh and E is the number of edges in the resulting graph (which is the same as the number of connections between the polygons). Since the value of E is normally greater than P, then the time complexity can be considered O(E). The number of connections of each polygon is usually a value near 4. Thus, $P < E < P^2$, where $E \approx cN$ and c is a constant. Therefore, the time complexity of our method is linear in relation to the number of polygons in the environment (which is the same as the number of nodes in the graph). The time complexity of the Dark Path Algorithm is related to the grid dimension. The time complexity for the Dark Path Algorithm, previously presented in Section 2.3, is $O(N^4)$, considering a grid of size NxN. Figures 2.15 and 2.16 present a comparison between the time complexity of both algorithms. Note that a linear function can surpass the time demanded by the proposed method (Figure 2.15), while the time demanded by the Dark Path Algorithm can only be surpassed by a fourth-degree function (Figure 2.16). Although the proposed method presents a lower time complexity, it is important to note that it is executed several times during a simulation, while the Dark Path Algorithm is executed only once, before the simulation starts. But even then, to achieve a realistic environment represented by a large grid, the computational time demanded by the Dark Path Algorithm can be prohibitive.

[1]https://youtu.be/2AFqYN_eoAI
[2]https://youtu.be/RUBCG9ZuIVY

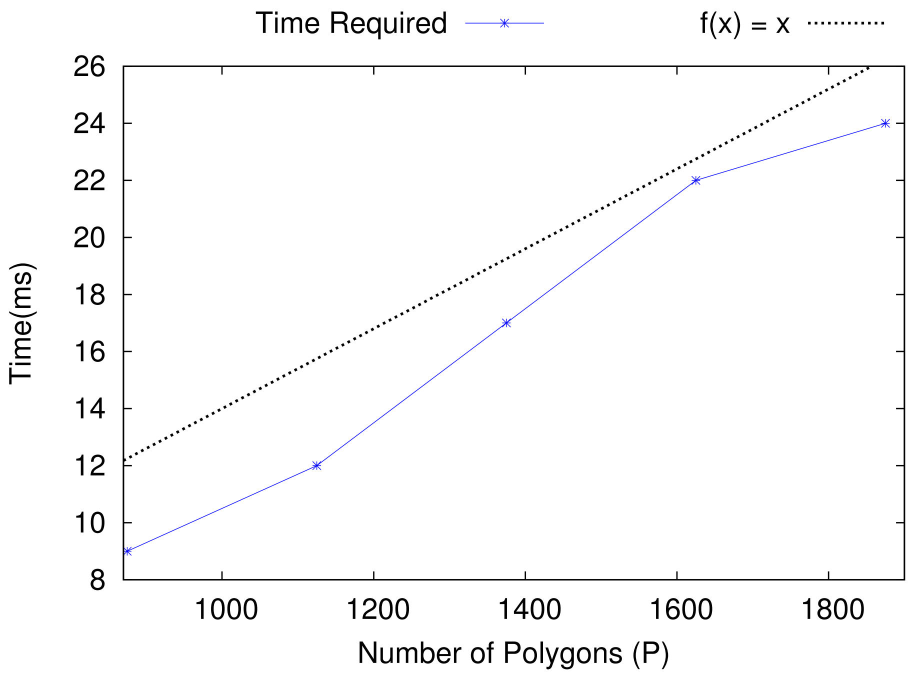


Figure 2.15: Time complexity for the Navigation Mesh approach. Note that the linear function $f(x)$ was able to surpass the time curve of the algorithm, where $f(x)$ is $O(x)$

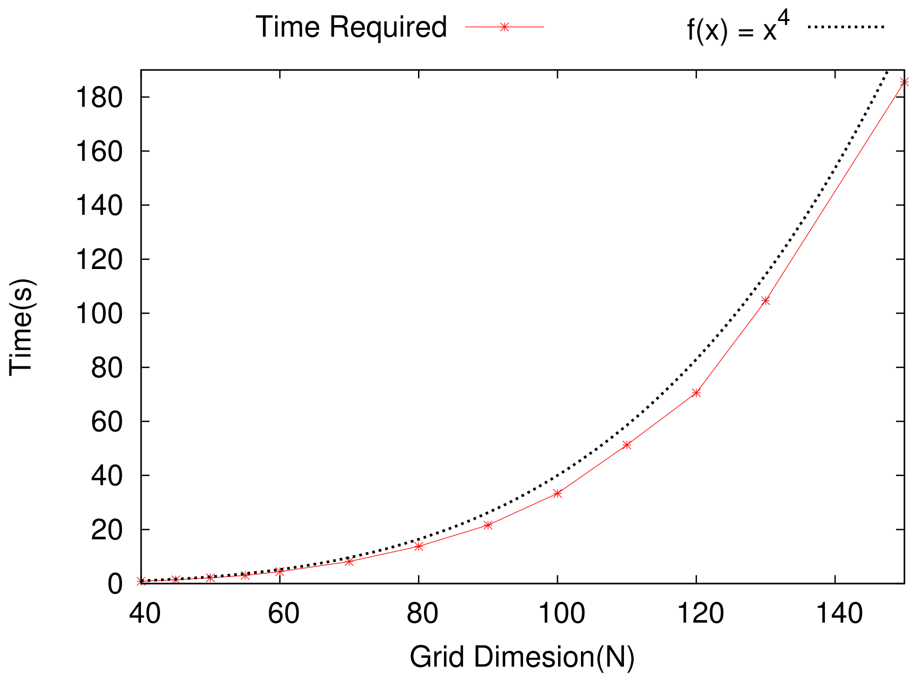


Figure 2.16: Time complexity for the Dark Path Algorithm. Note that only the fourth-degree function $f(x)$ is able to surpass the time curve of the algorithm, where $f(x)$ is $O(x^4)$

3 REINFORCEMENT LEARNING FOR STEALTH GAMES

We showed in the previous chapter how to use stealthy path planning in order to achieve a covert behavior. Although the agent was able to successfully evade the enemy patrols, it was only capable of performing one action: move toward the destination. This impacts directly over the realism and believability of the agent's behavior, since the agent is not capable of making smart decisions such as a human player would. Thus, we propose to use Reinforcement Learning (RL) in order to allow the agent to perform more actions and learn how to coordinate them, just as a human player.

We start this chapter by presenting some of the work related to the application of RL techniques to several game genres. The RL methods that we used in our work are described in the sequence. We then present the proposed RL approach for a stealth game, describing the defined state formulation, actions and reward function which were used. Finally, we present the experimental results using the previously presented stealth game simulator.

3.1 RELATED WORK

Reinforcement Learning applied to games was first introduced by Tesauro (1992), where the computer learned how to play Gammon at a master's level by using Temporal Difference techniques. His RL approach suppressed even his previous Artificial Neural Network approach (TESAURO, 1989). Modern games present several different difficulties when compared to board games, such as different game genres, greater number of agents to be accounted for and believable human-like behaviors. Thereby, different approaches are required in order to attend the demand of the gaming industry.

First Person Shooter(FPS) is a common game genre and the application of RL over this game type has been studied in previous works. In this genre, the player must move through an environment while shooting at its enemies and avoiding being shot. A common approach for FPS is to separate the game mechanics into modules and create special algorithms for each of these modules. The approach presented by McPartland and Gallagher (2011) uses two reinforcement learning modules: the first trains the agent on how to move around the environment without colliding with walls, while the second trains combat strategies. These modules are then connected through another RL layer that learns when to use each of the modules. A similar approach was used in a *capture the flag* game, presented by Hefny et al. (2008). Those authors used three modules: a high-decision module trained using RL, a path planning module, and a combat module trained by an Artificial Neural Network (ANN). Reinforcement learning can also be used to learn only specific controllers in a FPS game, as presented by Glavin and Madden (2015). The agent uses RL only to learn how to shoot by selecting the right weapon and aiming to specific spots at the right moment. The remaining controllers, such as navigation, is controlled by other techniques. Smith et al. (2007) used RL only to learn high-level decision.

Machine Learning techniques applied to Fighting games have also been extensively researched. The main reason is that fighting games provides a set of low level actions (walk forward, walk backwards, punch, etc.) that can be combined into a complex chain of basic actions. Therefore, learning a sequence of actions is very important to achieve success. Graepel et al. (2004) and Andrade et al. (2005) presented two similar approaches of RL for fighting games by using simple reward functions. The former work uses the Sarsa algorithm and the latter uses the Q-Learning algorithm. A more complex reward function is presented by Mendonça et al. (2015), where the aim was to create a fighter with a human-like behavior. That reward function allowed the agent to perform more complex chain of attacks, although it wasn't able to surpass a human player. Those authors also used an ANN in order to simulate a human behavior and compared the results obtained by the two machine learning approaches.

Real Time Strategy (RTS) games represent a more complex genre, where the player has to manage and build a base, comprised by several structures and units with different behaviors and abilities. Building RL controllers for RTS games requires a simplification to the game mechanics in order to reduce the number of states and action complexity. Wender and Watson (2012) used this simplification approach for a RTS game, where each unit's abilities were simplified to two high-level actions: fight and retreat. Therefore, each unit must only learn when to fight and when to retreat, where each action encodes special algorithms that are executed when an action is taken. In the work presented by Wender and Watson (2014), the authors were concerned in creating a navigation algorithm for each individual unit in a RTS game. The states was represented by a set of Influence Maps that abstracts environmental informations. With these influence maps, the agent learns how to move through the environment without colliding with objects and avoiding enemy units. The remaining layers of the RTS game were controlled by other techniques. Another approach, presented by Amato and Shani (2010), uses RL in order to learn when to use previously defined strategies. The state variables are also abstractions of the actual state, in order to reduce the learning complexity. The agent retrieves a set information (such as enemy's wealth, military power, etc.) and then determines which strategy is best suited for the current environment configuration.

A more general approach has also been proposed by Mnih et al. (2013), where it is presented a general artificial controller capable of playing several different games by analyzing the visual feedback from each game. That approach involved image processing in order to determine when to apply a positive or negative reward.

3.2 REINFORCEMENT LEARNING METHODS

Reinforcement Learning is characterized as a learning problem where a given agent must learn to interact with the environment such that its actions maximize a reward function (SUTTON; BARTO, 1998). An agent is any entity that interacts with the environment. When interacting with the environment, the agent may receive a reward signal, indicating that it accomplished a task or not. These rewards are sent according to a reward function, which decides when to send a positive or a negative reward to the agent. Therefore, the agent learns to interact with the environment through its experiences, by analyzing future rewards when performing a given action in a certain state. A Reinforcement Learning method is any algorithm capable of teaching an agent how to interact with an environment through its experience of past actions.

The agent is constantly interacting with the environment by performing actions. These actions may alter the environment, which sends a reward signal to the agent in response. The environment's characteristics at a given moment is represented by a state s. In the reinforcement learning problem, the environment is represented by a state map that indicates all possible states S, where $s \in S$. At any moment, the agent can perform an action a that is chosen within a set of all possible actions A. The agent chooses an action $a \in A$ while in state $s \in S$ based on its current policy π, that indicates which action must be performed for each state. The reward function $r(s, a)$ determines the reward signal that must be sent when the agent performs an action $a \in A$ while in state $s \in S$. The quality of a state-action pair (s, a) is given by the Q-Value $Q(s, a)$, which measures the expected future rewards after performing action a while in state s. Figure 3.1 summarizes the interaction between the agent and the environment.

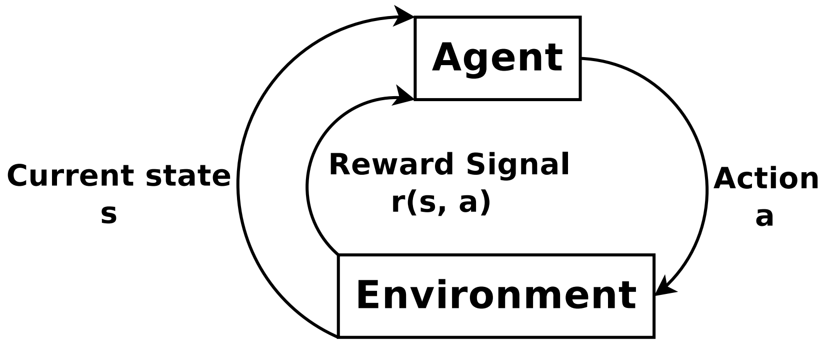


Figure 3.1: Interaction between agent and environment.

There are three main categories of Reinforcement Learning methods: Dynamic Programming (DP), Monte Carlo (MC), and Temporal Difference (TD) (SUTTON; BARTO, 1998). Dynamic Programming uses the best Q-Value of the possible next states (s') in order to update the Q-Value of the current state s. Therefore, DP methods require a distribution model $p(s'|s, a)$ that indicates the probability of reaching state s' after performing action a while in state s. The distribution model requires a full knowledge of how the environment works, which is very rare in practical situations. This way, DP methods can't be effectively used in real situations. Monte Carlo methods make use of experience in order to learn how to behave in an environment and thus, don't require a distribution model. Instead, they use a sequence of states, actions, and reward signals to learn. When the end of an episode is reached, each state-action pair (s, a) visited during the episode is updated according to the rewards received after it was visited. One disadvantage of MC methods is that the Q-Values are only updated during the end of an episode (a terminal state is reached). Temporal Difference methods, like MC methods, also uses experience in order to learn. The main difference between TD and MC is that the former approximates the expected future rewards following the state-action pair (s, a) by using the Q-Value $Q(s', a')$ of the next state. By using an approximation, TD methods update the Q-Values after each step. Practical applications has shown that TD methods converge faster than MC methods, although this statement lacks a mathematical proof (SUTTON; BARTO, 1998).

We applied three different TD methods in our stealth game simulator: *one step Q-Learning*, Sarsa(λ), and Dyna-Q. We chose TD methods due to their known efficiency and simple implementation. The TD methods adopted here also represents a great diversity among the TD methods: the *one step Q-Learning* is a very efficient *off-policy* algorithm, the Sarsa(λ) blends Temporal Difference and Monte Carlo characteristics, while the Dyna-Q represents a model based Temporal Difference method. We adopted a stochastic policy π, called ε-greedy policy, that selects random actions with a ε probability and actions with maximum expected future rewards with a $1 - \varepsilon$ probability, that is, $max_a Q(s', a)$, where s' is the next state after visiting the state-action pair (s, a). The remainder of this section will present more details regarding the chosen TD methods.

3.2.1 Q-LEARNING

One step Q-Learning (WATKINS, 1989) represents one of the main Temporal Difference learning method. It uses only the immediate reward signal given by $r(s, a)$ and the Q-Value of the next state $Q(s', a')$ in order to update the Q-Value of a state-action pair (s, a). Also, Q-Learning is an *off-policy* method: it approximates the expected future rewards for the state-action pair (s, a) by choosing the following state-action pair (s', a') using a different policy used for choosing actions. Here, we use a ε-greedy policy called π (described previously in section 3.2) to choose the action a to be executed during state s, while another policy, called $\pi*$, is used to select the next action of the following state in order to determine the value of $Q(s', a')$. Policy $\pi*$ chooses the optimal action for a given state s', that is, $max_a Q(s', a)$.

Q-Learning works by performing the following steps: (i) the environment informs the agent its current state $s \in S$; (ii) the agent uses its policy π to choose its following action $a \in A$; (iii) the environment returns the next state s' after carrying action a while in state s and the immediate reward signal given by $r(s, a)$; (iv) policy $\pi*$ is used for determining the optimal action a' of the next state s' ($Q(s', a') = max_a Q(s', a)$); (v) the Q-Value $Q(s, a)$ is updated according to

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha[r(s, a) + \gamma max_a(Q(s', a))] \tag{3.1}$$

These steps are executed until a termination state is reached, that is, the current episode is finished.

There are two rates accounted in the Q-Learning's update function: learning and discount rates. The learning rate (α) determines how much the immediate reward given by $r(s, a)$ and the approximate expected future rewards will be considered during the update process. Higher α values results in greater variations of the $Q(s, a)$ value during an update, while lower α values results in smaller variations. The discount rate (γ) determines how the expected future rewards ($Q(s', a')$) will influence the update process. Higher γ values makes the agent select actions with long-term rewards, that is, it prefers actions with higher future rewards instead of actions with higher immediate rewards. Lower γ values makes the agent select actions with higher immediate rewards $r(s, a)$. During training, the learning rate α should start with higher values, since during the initial stages of training, the expected future rewards are not accurate. At more advanced stages of training, the expected future rewards are more accurate, since the agent learned them during previous stages. Therefore, α is set to lower values (MILLINGTON; FUNGE, 2009). The learning and discount rates are set always to the following range: $0 \leq \alpha \leq 1$ and $0 \leq \gamma \leq 1$.

The pseudo-code of the *One step Q-Learning* is presented in Algorithm 4.

Algorithm 4 One step Q-Learning (S, A)

```
1: Initialize Q(s,a) for all $s \in S$ and $a \in A$;
2: repeat
3:    repeat
4:       $s \leftarrow$ current state;
5:       $a \leftarrow$ action chosen through policy $\pi$;
6:       Execute action $a$ and retrieve $r(s, a)$ and $s'$;
7:       $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[r(s, a) + \gamma max_a(Q(s', a))]$;
8:    until end of the episode
9: until number of episodes is reached
```

3.2.2 SARSA(λ)

Sarsa (RUMMERY; NIRANJAN, 1994) is a Temporal Difference method and it represents the *on-policy* version of the Q-Learning. This means that the Sarsa algorithm approximates the expected future rewards after visiting the state-action pair (s, a) by using the same policy π used for selecting an action to be performed, that is, the chosen action a' while in the following state s' is not necessarily optimal, it only follows the same policy π.

Thus, Sarsa updates a Q-Value $Q(s,a)$ by using $Q(s',a')$ instead of $max_a(Q(s',a))$, where a' represents the action selected by our ε-greedy policy π when in state s'. The update function of the Sarsa algorithm is given by

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha[r(s,a) + \gamma Q(s',a')] \tag{3.2}$$

Sarsa(λ) is a variation of the original Sarsa that blends Monte Carlo and Temporal Difference characteristics by using eligibility traces. Eligibility traces are associated to each state-action pair and they work as memory variables that indicate the last time a state-action pair (s,a) was visited. Whenever a state-action pair (s,a) is visited, its corresponding eligibility trace is incremented, informing that (s,a) was visited recently. The eligibility trace of the remaining (unvisited) state-action pairs at a given time step are all decreased. Eligibility traces are stored in a table $Z(s,a)$ that represents the corresponding eligibility traces for each state-action pair (s,a). At a given time step t, after performing action a_t while in state s_t, the table $Z(s,a)$ is updated according to

$$Z(s,a) = \begin{cases} \gamma\lambda Z(s,a) + 1, & \text{if } s = s_t \text{ and } a = a_t \\ \gamma\lambda Z(s,a), & \text{otherwise} \end{cases} \tag{3.3}$$

where γ represents the discount rate and λ represents the *trace decaying parameter*, whose value is limited to $0 \leq \lambda \leq 1$. This parameter corresponds to the decaying speed of the eligibility traces. Low values for λ result in faster decaying, while high values make the decaying process slower. The eligibility trace of a recently visited state-action pair must be incremented. Eligibility traces that are incremented are called cumulative traces, such as the one presented in Equation 3.3, since their values are not restricted. Instead of incrementing the traces, one can simply assign a value of 1 for every eligibility trace of state-action pairs recently visited. This is called trace replacement and it is given by

$$Z(s,a) = \begin{cases} 1, & \text{if } s = s_t \text{ and } a = a_t \\ \gamma\lambda Z(s,a), & \text{otherwise} \end{cases} \tag{3.4}$$

Here, we adopt the trace replacement technique.

After performing an action a_t while in state s_t, all state-action pairs are updated. The update function of the Sarsa(λ) algorithm considers the current Q-Value of a state-action pair (s, a), their corresponding eligibility trace, the immediate reward signal received after visiting (s_t, a_t), and the approximate expected rewards for (s_t, a_t) (given by $Q(s', a')$). The update rule first calculates a δ value according to

$$\delta = r(s_t, a_t) + \gamma Q(s', a') - Q(s_t, a_t) \tag{3.5}$$

and then updates every Q-Value $Q(s, a)$ according to

$$Q(s, a) = Q(s, a) + \alpha\delta Z(s, a) \tag{3.6}$$

An important aspect of the Sarsa(λ) is that it can behave like a full Temporal Difference method or a full Monte Carlo method, or a mixture of TD and MC. Its behavior is defined by the trace decaying parameter λ. When $\lambda = 0$, the eligibility traces of all non-recently visited state-action pairs will be 0, and 1 for the recently visited state-action pair (s_t, a_t). The result of Equation 3.6 for $\lambda = 0$ will be the same as the update function of the one step Sarsa (Equation 3.2). When $\lambda = 1$, the eligibility traces will remain 1 for every visited state-action pair (considering that the discount rate is also set to $\gamma = 1$) and thus, will update all state-action pairs visited during one episode, just like a Monte Carlo method. When $0 < \lambda < 1$, the visited state-action pairs will be updated with different intensities, depending on their eligibility traces.

The pseudo-code for the Sarsa(λ) algorithm is presented in Algorithm 5.

3.2.3 DYNA-Q

Dynamic Programming methods use a *distribution model* of the environment in order to update the Q-Values. The distribution model informs the probability of occurrence of all possible next states for each state. Hence, it requires a full knowledge of the environment's dynamics. Another approach for obtaining the environment's dynamics is by observing sequences of states during an episode and registering only one of the possible next states. This is called a *sample model.* Distribution and sample models are used to learn how the agent's actions impact on the environment and how the environment behaves in response to these actions. Sample models are less accurate than distribution models, but are easy to obtain and, thus, are more useful in real world applications. Distribution models, on the other hand, are more accurate, but given the stochastic nature of many real problems,

Algorithm 5 Sarsa(λ) (S, A)

```
 1: Initialize Q(s,a) for all $s \in S$ and $a \in A$;
 2: **repeat**
 3:     Z(s,a) = 0 for all $s \in S$ and $a \in A$;
 4:     **repeat**
 5:         $s \leftarrow$ current state;
 6:         $a \leftarrow$ action chosen through policy $\pi$;
 7:         Execute action $a$ and retrieve $r(s,a)$ and $s'$;
 8:         $Z(s,a) = 1$;
 9:         $\delta = r(s,a) + \gamma Q(s',a') - Q(s,a)$;
10:         **for** all $s \in S$ **do**
11:             **for** all $a \in A$ **do**
12:                 $Q(s,a) = Q(s,a) + \alpha\delta Z(s,a)$;
13:                 $Z(s,a) = \gamma\lambda Z(s,a)$;
14:             **end for**
15:         **end for**
16:     **until** end of the episode
17: **until** number of episodes is reached
```

it is hardly used.

Knowledge can be attained by *learning* through experience of past actions or by *planning*. Learning is the approach used in Monte Carlo and Temporal Difference methods, where the agent updates its Q-Values based on real experience. Planning, on the other hand, uses simulated experience through models in order to plan its next action, as done by Dynamic Programming methods. The Dyna-Q (SUTTON, 1991) uses planning and learning in conjunction by using sample models to learn how the environment behaves and by using Temporal Difference to learn.

The Dyna-Q algorithm observes sequences of states in order to build a model. It uses a table M to store the observed next state and the immediate reward received for any previously visited state-action pair (s,a). This way, the sample model is represented by table M, which is built by observing past samples. The sample model used allows the agent to predict possible outcomes for all previously visited state-action pairs, although it is not as accurate as a distribution model. This model is used to simulate experience by reconstructing a sequence of state-action pairs previously observed. The Dyna agent uses the Q-Learning update function (Equation 3.1) with its simulated experiences. The Q-Learning update function is also applied to current events: the agent selects an action using a given policy, observe the reward received and the following state, and update the Q-Value of the currently visited state-action pair (s,a) by using Equation 3.1.

The pseudo-code of the Dyna-Q algorithm is presented in Algorithm 6.

Algorithm 6 Dyna-Q (S, A)

```
1: Initialize Q(s,a) for all $s \in S$ and $a \in A$;
2: Initialize M(s,a) for all $s \in S$ and $a \in A$;
3: repeat
4:     repeat
5:         $s \leftarrow$ current state;
6:         $a \leftarrow$ action chosen through policy $\pi$;
7:         Execute action $a$ and retrieve $r(s, a)$ and $s'$;
8:         $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r(s,a) + \gamma max_a(Q(s',a))]$;
9:         $M(s,a) \leftarrow (r(s,a), s')$;
10:        for $i = 0$ to $N$ do
11:            $(s,a) \leftarrow$ randomly select a previously visited state-action pair;
12:            $r, s' \leftarrow M(s,a)$;
13:            $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma max_a(Q(s',a))]$;
14:        end for
15:    until end of the episode
16: until number of the episodes is reached
```

3.3 METHODOLOGY

Reinforcement Learning is very dependent on the application at hand. Several aspects of reinforcement learning methods are domain dependent and thus, require a careful formulation. The main domain dependent aspects of a RL method are: state formulation, set of possible actions and reward function. In this section, we will address to these three topics.

3.3.1 STATE FORMULATION

The state formulation, as already mentioned, defines which information is available to the agent during a decision making process. Therefore, we have to consider every important environmental characteristic. The state variables were defined based on an expert's knowledge, which analyzed important game aspects that are taken into account by a player during a decision making process. The state variables used are as follows:

- **Enemy Proximity:** represents the approximate distances between the stealthy agent and the enemies. The distances are discretized between three values: very near (distance of 140 world units or less), near (distance between 141 and 150 world units) or far (distance between 151 and 170 world units). This variable considers only enemies within a 170 world unit radius from the stealthy agent and it is discretized into 8 values that are presented in Table 3.1. As an example, consider the situation were there is only one enemy located at a ”very near“ distance from the stealthy agent. In this case, the value for the **Enemy Proximity** variable is 1;

- **Enemy Approaching:** indicates if the enemies are moving away or approaching the stealthy agent. This variable uses a discretized distance between the enemy and the stealthy agent. Therefore, we discretized this variable into 4 values that are presented in Table 3.2. The definitions of *very near*, *near*, and *far* were defined previously;

- **Enemy visible:** indicates if the nearest enemy is visible or not to the stealthy agent;

- **Nearest Enemy Distance:** represents the distance to the nearest enemy discretized into 2 values: close (distance of 60 world units or less) or not close (distance of 61 world units or more). If the nearest enemy is close, then the agent can eliminate it by using the “*Eliminate nearby enemy*” action (see Subsection 3.3.2 for more details);

- **Nearest Enemy Moving:** indicates if the nearest enemy is moving or not. This variable is important since when an enemy agent is standing still, it is possible that it starts moving to any direction. Therefore, it is important to wait and see where the enemy will move;

3.3.2 ACTION SET

With the state variables defined, the agent is now capable of perceiving the current environmental characteristics. The next step is to define the possible actions that the agent can execute for each state. It is important to consider which actions will contribute for the agent’s success when defining the possible actions. In a stealth game, it is desirable to allow the agent to hide from enemies for a given period, eliminate nearby patrols silently, and move toward the destination. Also, in order to reduce the learning complexity, we define high-level actions that, instead of performing a simple task, defines a goal that is

	Very Near	**Near**	**Far**
0	0	0	0
1	1	0	0
2	1	1+	0+
3	1	0	1+
4	2+	0+	0+
5	0	1+	0+
6	0	1+	1+
7	0	0	1+

Table 3.1: Possible values for the "Enemy Proximity" state variable. The columns indicates how many enemies are at each of the discretized distances, while the rows indicates the variable value for each configuration. The value X+ indicates that X or more enemies at a specific distance are approaching the enemy.

	Very Near	**Near**	**Far**
0	0	0	0
1	0	0	1+
2	0	1+	0+
3	1+	0+	0+

Table 3.2: Possible values for the "Enemy Approaching" state variable. The value X+ indicates that X or more enemies at a specific distance are approaching the enemy.

achieved by a separate algorithm. By doing this, we can define only a small set of high-level actions instead of several low-level actions (such as: move right, move left, move forward, etc.). The action set was also defined based on an expert's knowledge and it is comprised of the following actions:

- **Move toward the destination:** this is the only action that allows the agent to reach the destination, and therefore, is crucial to the agent's success. When this action is executed, the agent calculates a stealthy path using the Navigation Mesh Stealthy Path Planning method, presented in Chapter 2, and starts moving through the resulting path;

- **Hide:** the agent searches for the nearest cover area that is hidden from nearby enemies and moves to that location;

- **Eliminate nearby enemy:** the agent eliminates the nearest enemy using a silenced weapon. When the enemy is far from the agent, then this action may fail. Whenever the agent misses the shot, all enemies enter in an alert state, in which they move faster. Also, the enemy that the agent tried to eliminate will move to the agent's current location in order to identify the threat. We discretized the distance between the agent and the nearest enemy into four values: close (distance of 60 world units or less), medium (distance between 60 and 100 world units), far (distance between 100 and 150 world units) and very far (distance greater than 150 world units). The chance of successfully hitting the shot for each distance value are 100%, 50%, 10% and 0%, respectively;

- **Pursue the nearest enemy:** the agent walks toward the nearest enemy by using a minimum path. If the enemy is facing the agent, then this action will result in the agent walking right into the enemy's field of vision. Therefore, the agent must learn to use this action only when the nearest enemy is facing the opposite direction.

The state formulation defined here results in 256 different states. Since there are 4 different actions, the Q-Table is represented by a 256x4 table. For a more complex problem (such as the stealth game problem), it is important to carefully define the state and action sets in order to reduce the overall complexity of the learning algorithm. With the current state and action formulation, it is possible to define complex behaviors and allow the agent to behave realistically in any given environment.

3.3.3 REWARD FUNCTION

Another important element of any reinforcement learning method is the Reward Function. The reward function is responsible for delivering rewards and punishments to the agent based on past actions or states. Reward functions can focus only on previous states visited by the agent or on previous actions performed for each state. We will adopt here the approach where the reward is defined based on the state-action pairs visited.

Reward functions are domain dependent. Therefore, it is important to determine a set of rules that are considered good for the agent or a set of goals that the agent must complete in order to gain a reward. This also means defining a set os rules and goals that grant the agent punishments when completed.

We propose two different reward functions: a simple reward function that rewards the agent when it achieves success and punishes it otherwise (called here the Simple Reward Function), and a more complex reward function that rewards the agent for following a set of rules, as the one used by Mendonça et al. (2015) (called here the Rule-Based Reward

Function).

The Simple Reward Function (SRF) represents a straight-forward function that is used in many problems. It uses the basic concept of rewarding upon success and punishing when the agent fails to accomplish its goal, where success is achieved when the agent reaches its destination and failure occurs whenever the agent is detected by an enemy patrol, as mentioned previously. We used a maximum reward of +1 when the agent achieved success and a minimum reward of -1 when it failed. The only information needed for this function is the goal to be accomplished, that is, which states that indicates that the agent succeeded and which indicates that it failed. Therefore, the SRF is simple to define. The main problem is that it may fail to find an efficient policy for complex and stochastic environments. On the other hand, the Rule-Based Reward Function (RBRF) is good for problems where it is desirable to learn a behavior capable of achieving the goal instead of simply learning how to solve the problem directly. For example: in a stealth game with a non-stochastic environment, one could simply memorize the enemy patrols' pattern and avoid their paths. But if the environment is stochastic, it is impossible to memorize sequences of states, since it changes every time. In this case, it is desirable to learn a good behavior capable of generalizing for every environment, instead of learning how to achieve success in a specific scenario. The main drawback of the RBRF is that it relies on specific domain-related informations that are defined by an expert. Therefore, it depends on and expert's knowledge in order to define the reward rules. The RBRF also uses the SRF when the agent achieves success or when it fails. This way, the agent is capable of changing its tactic when the rules fail to guide it toward the goal.

Our Rule-Based Reward Function was designed to teach the agent how to behave in a generic environment. The rules were created based on an expert's knowledge, inspired by how human players perceive and react in a stealth game environment. The adopted rules are as follows:

- **Eliminate enemy when near it:** whenever the agent is close to an enemy (distance of 60 world units or less), then use the **"Eliminate nearby enemy"** action. This rule teaches the agent to eliminate its enemies whenever they are close (distance of 60 world units or less) and the chance of successfully eliminating the enemy is 100%. If this rule is followed, the agent receives a reward of 0.07. Otherwise, it receives a punishment of 0.04;

- **Move when there are no enemies around:** move toward the destination when the previous condition isn't met and there are no enemies within a 170 world units radius around the agent (this value is based on those ones presented in the "**Enemy Proximity**" state variable presented in Subsection 3.3.1). This rule prevents the agent from getting stuck by performing other actions when there are no enemies around (when all enemies are eliminated, then the agent must move toward the destination). If this rule is followed, the agent receives a reward of 0.07. Otherwise, it receives a punishment of 0.04;

- **Hide from enemies:** this rule teaches the agent to hide when none of the above conditions are met, and there is at least one enemy at a near distance (distance between 131 and 150 world units) and it is also moving closer to the agent or halted. This rule helps the agent to avoid entering in dangerous zones and waiting until the enemies are at a safe distance. If this rule is followed, the agent receives a reward of 0.07. Otherwise, it receives a punishment of 0.04;

- **Pursue enemies that are moving away:** if none of the above conditions are met, and there is only one enemy at a close distance ("Enemy Proximity" = 1) and it is moving away from the agent, then pursue it. This rule teaches the agent to pursue isolated enemies in an effort to approach and eliminate them without alerting the others. If this rule is followed, the agent receives a reward of 0.07. Otherwise, it receives a punishment of 0.04;

- **Move toward the destination:** if none of the above conditions are met, then reward the agent for just moving to the destination. This helps the agent to always move closer to the destination point. If this rule is followed the agent receives a reward of 0.07;

- **Reward upon success:** this is the reward based on the SRF, where the agent is rewarded with a maximum reward of 1.0 when it achieves success and a punishment of -1.0 when it is detected by the enemy patrols.

3.4 EXPERIMENTAL RESULTS

We performed several experiments in order to test the efficiency of using reinforcement learning for a stealth game. We compared the two reward functions applied to the stealth simulator: the Simple Reward Function (SRF) and the Rule-Based Reward Function (RBRF). Finally, we compared the results obtained by using RL with the results obtained by only using stealthy path planning, presented in Chapter 2.

Before performing any experiments, it is necessary to define how to test the quality of a given trained agent. To define a trained agent's quality, we perform several executions of the stealth simulator using random environments. The success rate attained will represent the quality of its training. Due to the stochastic nature of the stealth simulator, it is necessary to define the number of episodes to be executed while validating the agent that results in a success rate that is statistically valid. We ran 50 validation sessions with 1000 executions of the stealth simulator with a trained agent and we set the learning and exploration rates to zero ($\alpha = 0$ and $\varepsilon = 0$) in order to test only the current knowledge of the agent. These 50 sessions resulted in a mean and standard deviation equal to 79.68 and 1.32, respectively. The confidence interval for these 50 sessions was 79.68 ± 0.47 with a confidence of 95%. Although these results were obtained for a specific trained agent, it is consider here that similar results are obtained for different agents. Thus, all validations performed used 1000 executions of the simulator with the learning and exploration rates set to $\alpha = 0$ and $\varepsilon = 0$, resulting in a success rate with an error of approximately $\pm$ 0.47 and a confidence of 95%.

There are three main parameters that must be defined for RL methods: learning rate (α), discount rate (γ) and the exploration rate (ε). We performed preliminary tests in order to find a good set of parameters for our application. We used the Q-Learning method to test the α, γ and ε parameters. The learning parameter was set to a low value in order to reduce the variation of the Q-Values during an update and the exploration was also set to a low value to encourage exploitation over exploration. Therefore, we set $\alpha = 0.1$ and $\varepsilon = 0.1$. The discount rate is usually set to a high value in order to consider long-term rewards. But in the proposed RL formulation using the RBRF, the rewards received when succeeding or failing ended up degrading the final policy. This happens due to some failures that occur when the agent is hiding: sometimes, it chooses a bad hiding spot and ends up being found by enemy patrols. When γ is set to a high value, these failures ends up changing the initially defined strategy (defined by the set of rules of the RBRF) and, consequentially, achieving a lower success rate. Table 3.3 shows how the success rate varies with regard to the value of γ in an environment patrolled by 4 sentries. Therefore, we set $\gamma = 0.1$ in order to maintain our strategy. For the remaining of this work, except to specific tests that will be identified later, the training parameters used are $\alpha = 0.1$, $\gamma = 0.1$ and $\varepsilon = 0.1$. The *trace decaying parameter* λ of the Sarsa(λ) algorithm and the number of simulated experiences of the Dyna-Q algorithm (the N variable of Dyna-Q pseudo-code presented in Algorithm 6) were defined based on other experiments that will be shown later.

	$\boldsymbol{\gamma = 0.1}$	$\boldsymbol{\gamma = 0.5}$	$\boldsymbol{\gamma = 0.9}$
Success Rate	80.6%	77.7%	73.1%

Table 3.3: Success rate obtained in 1000 random executions of the stealth game simulator with 4 enemy patrols using the Q-Learning algorithm and with varied values of the discount rate (γ).

The first set of experiments was designed to test the efficiency of the SRF. As discussed previously, the SRF is best suited for deterministic environments, where the agent can learn the sequences of actions that led him to a successful state. If the discount rate is kept high, the agent can learn to carry actions that will only reward it in the long term. We used a specific scenario patrolled by 4 enemies where success couldn't be achieved by only moving toward the destination. The agent was then trained in this specific environment to learn how to obtain success by using other actions. We also used three different RL algorithms to test how each one behaves for this specific task. We performed several tests in order to determine the best parameters for all three methods. We start by analyzing the parameters of the Q-Learning algorithm: learning and discount rate. Since we want to create a deterministic environment, we fixed the exploration rate to $\varepsilon = 0$. We tested which combination of parameters converged faster to a winning policy. When the agent converges to a winning policy, it will not fail any more in the same environment. The results of the different parameter combinations are presented in Figure 3.2. The dots in each curve represents the point where the agent converged to a winning policy (represented by a diagonal line in Figure 3.2). It is possible to observe that the combination that converged faster was $\alpha = 0.2$ and $\gamma = 0.9$, taking 35 episodes to converge to a winning policy. Therefore, we used this parameter combination for the Q-Learning, Sarsa(λ) and Dyna-Q algorithms when using the SRF.

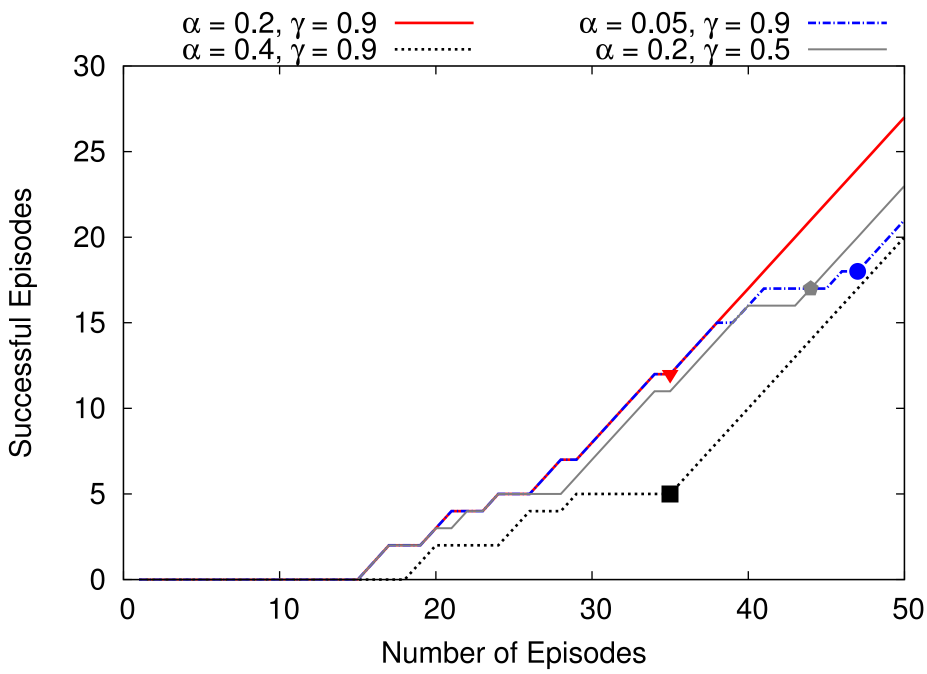


Figure 3.2: Parameters test for the Q-Learning algorithm in a fixed environment using the SRF.

The Sarsa(λ) algorithm also uses the decaying parameter λ. Several possible λ values were tested, but any value greater than zero resulted in the same results. The Sarsa(λ) updates every state-action pair visited during an episode. Therefore, when the agent fails, every state-action pair visited during an episode will be reduced, regardless of the value of λ. Likewise, when the agent achieves success, then every state-action pair visited will be incremented. When this happens, the method converges, since it will now choose this same sequences of actions and obtain success for the following episodes. Thus, the method will converge within the same number of episodes for any $\lambda > 0$.

The Dyna-Q algorithm uses the N parameter that determines how many simulated experiences are generated during each step. This parameter was variated to test which value makes the algorithm converge in less episodes. The results are presented in Figure 3.3. It shows that Dyna-Q converges faster when $N = 30$, taking 30 episodes until it converges to a winning policy. Thus, we adopted $N = 30$ for the Dyna-Q algorithm.

The final results for the SRF in a fixed environment are shown in Figure 3.4. These results show that Sarsa(λ) is capable of learning a winning police in only two episodes. Also, Sarsa(λ) converged faster than Dyna-Q and Q-Learning algorithms, which took 30 and 35 episodes to converge, respectively. The main reason for the faster convergence of the Sarsa(λ) is that it updates all state-action pairs visited in a given episode. This allows

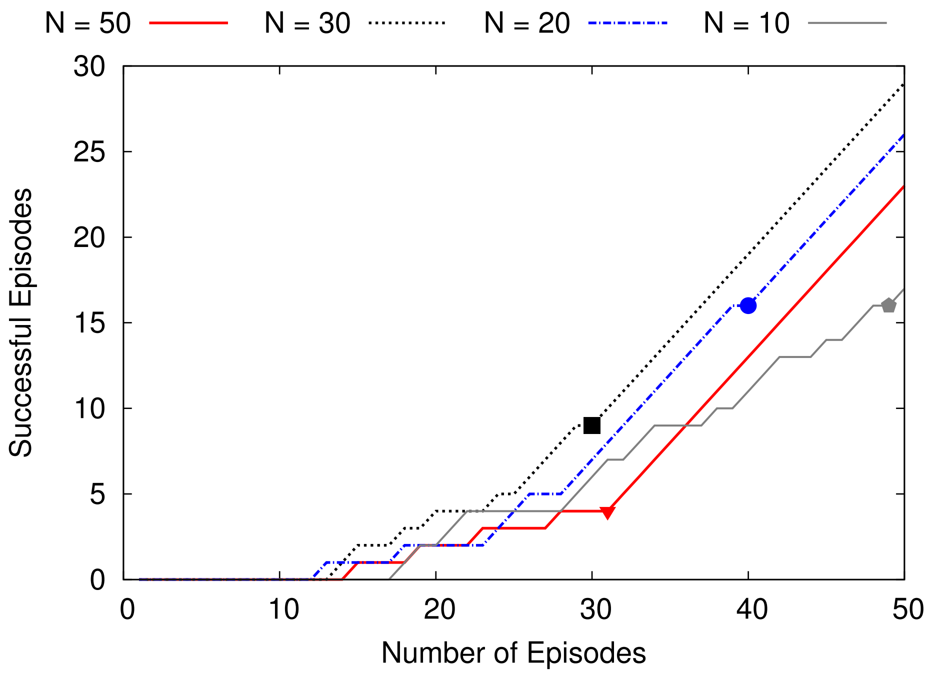


Figure 3.3: Parameters test for the Dyna-Q algorithm in a fixed environment using the SRF.

the agent to converge as soon as it achieves success for the first time.

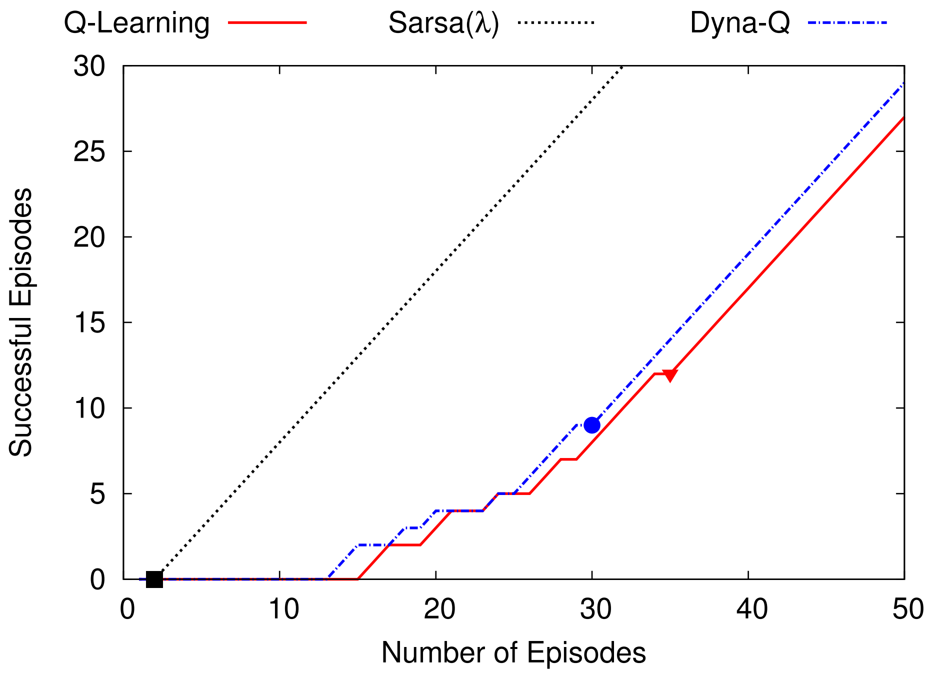


Figure 3.4: Number of episodes until convergence for Dyna-Q, Q-Learning and Sarsa(λ) in a fixed environment using the SRF.

The results presented in Figure 3.4 shows that the SRF is enough for a deterministic stealth game environment, where the agent must learn sequences of actions that it must execute in order to achieve success. The agent may act unrealistically, since its only goal is to achieve success, no matter how it behaves. Therefore, if its knowledge is used in another environment, it probably won't achieve success, since its knowledge is specific to the environment it trained in. If we use the SRF in a stochastic environment, it may not be able to effectively teach the agent how to attain success due to the restrictions that were already mentioned of the SRF. The goal in a stochastic environment is to teach a behavior to the stealthy agent, and not only a sequence of actions that is capable of achieving success. This is what happens in the stealth game simulator: the moving pattern of enemies and the environment are all randomly generated. When the agent succeeds in a stochastic environment, it may not have performed only good actions. It could have executed several bad decisions until it succeeded. Therefore, SRF isn't suited to the stealth game problem. We show this by using the SRF in a stochastic environment, where the moving pattern of enemies and the environment were all randomly generated between episodes. All the environments generated here were patrolled by 4 enemies. The results are presented in Figure 3.5.

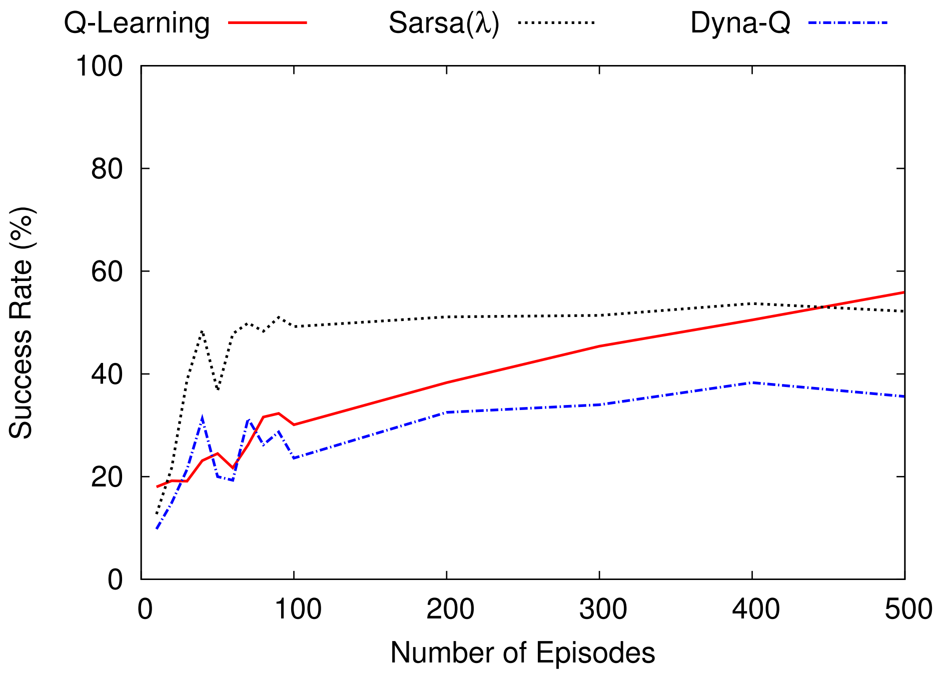


Figure 3.5: Convergence time for Dyna-Q, Q-Learning and Sarsa(λ) in stochastic environments using the SRF.

Figure 3.5 shows that none of the RL methods used here were capable of defining a good policy for the problem. These results were generated by training the agents a varied number of episodes (to a maximum of 500 episodes, although higher numbers were also tested) and then validating the agents after each training session. Therefore, Figure 3.5 shows how the number of training episodes influences the overall quality of an agent. Each training was performed with $\alpha = 0.1$, $\gamma = 0.1$ and $\varepsilon = 0.1$, as discussed at the beginning of this section. The λ parameter of Sarsa(λ) was set to $\lambda = 0.7$ and the N parameter of Dyna-Q was set to $N = 30$ (based on preliminary tests). Sarsa(λ) once again converged faster, although any of the methods were capable of effectively converging to a winning policy. We can see that Dyna-Q and Q-Learning required more episodes to converge to a stable policy. The Q-Learning algorithm eventually reached a similar success rate depicted by the Sarsa(λ) and then stabilized (although not shown in this figure, the success rate for Q-Learning stabilizes after 500 episodes during training). The Dyna-Q algorithm presented the worst performance between the three tested methods. The main reason for this behavior is due to the simulated experiences that does not effectively represent the environment's model. Therefore, these simulated experiences mislead the agent's learning process.

The RBRF, unlike the SRF, outlines behaviors that the agent must follow in order to receive a reward. We designed several tests to measure the performance of Dyna-Q, Sarsa(λ) and Q-Learning while using the Rule-Based Reward Function. One agent was trained for each of the RL algorithms used. The number of training episodes was varied for each agent and validated after its different training sessions. The results are presented in Figure 3.6.

The results depicted in Figure 3.6 show the success rates for each agent for different training sessions. It shows that all three reinforcement learning methods converge to an approximate success rate of about 79% after approximately 200 episodes of training. The success rate curves for all tested methods are similar. This happens due to the reward function adopted: the RBRF outlines specific behaviors that should be followed and delivers small reward signals for visiting several state-action pairs. The reward sent by obtaining success or failure presents little importance in this scenario, since the rules adopted already guarantee a high success rate. Thus, the agent's knowledge converges to a policy that obeys the outlined rules, regardless of the RL method used.

The agent's success rate depends on the number of enemies in the environment. We validated each agent with a varied number of enemies in the environment in order to measure how this parameter influences the agent's success rate. Each agent was trained

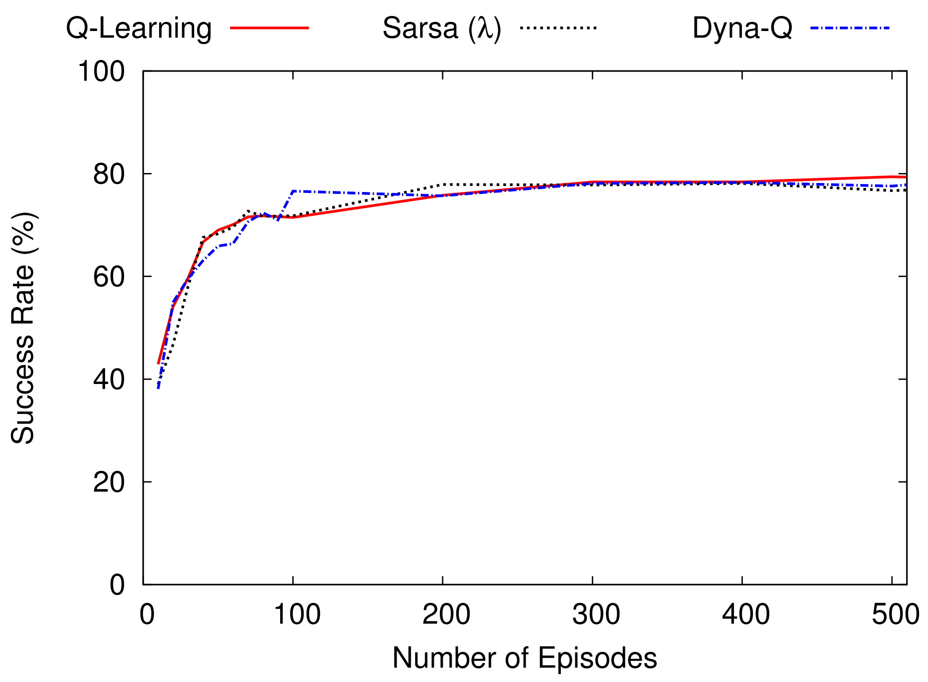


Figure 3.6: Success rate obtained by different RL algorithms using the RBRF in stochastic environments with varied number of episodes during the training stage.

in an environment patrolled by 4 enemies using one of the three RL methods. The trained agents were then validated in 1000 different environments. It is important to note that we used the same 1000 environments for all agents and for all different numbers of patrols. This set of 1000 environments is also the same used in the experiments of Figure 2.13 of Chapter 2. The results are presented in Figure 3.7, where one can see that all three RL methods performed equally for each number of enemies. Also, the success rate of all methods increases as the number of enemies decrease. The success rate varied from approximately 79% with 4 enemies to approximately 98% with 1 enemy.

Regarding the agent's behavior after training with any of the three methods used, it is evidenced that they effectively learned the rules outlined by the RBRF, as can seen in this video[1]. The agent pursues enemies that are near it and that are also moving away from it, eliminating them afterwards; it uses the *Hide* action when there are enemies approaching its location; walks to the destination when there are no enemies around; and it eliminates enemies that are very near it (when there is a 100% chance that the shot will hit its target). The agent usually fails when it is hiding from enemies and the hide action fails to find an efficient hiding spot. The agent also fails when it is surrounded by enemies in all directions. The RL presents these behaviors independently of the RL method used.

[1]https://youtu.be/ZaEZbhw9974

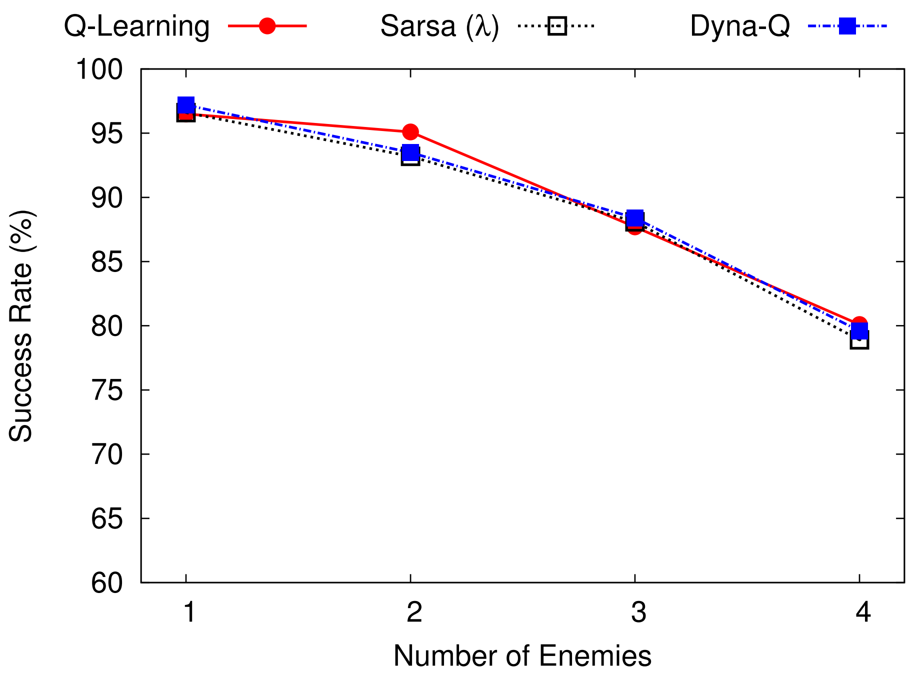


Figure 3.7: Success rate obtained for each RL method used in an environment patrolled by a varied number of enemies.

The training stage must visit as many states as it can in order to efficiently train the agent. If only 1 enemy is used during the training stage in the stealth game simulator, then the agent won't learn how to behave in an environment with more than 1 enemy. On the other hand, training the agent with 4 enemies will guarantee that all state-action pairs are visited a reasonable amount of times. This happens due to our state formulation: if there are several enemies in the environment but only one of them are near to the agent, then the state will inform that only 1 enemy is present. Also, the agent can learn how to eliminate enemies by using the *Eliminate* action, which reduces the number of enemies on the environment and helps the agent to learn state-action pairs with fewer enemies. To demonstrate this, we trained an agent using only 1 enemy in the environment using each of the RL methods adopted. We then validated the resulting agent against 1 and 4 enemies and the results are presented in Table 3.4. The results show that the agents trained with only 1 enemy performed poorly when validated with 4 enemies, although they present a good performance when validated with only 1 enemy. On the other hand, the agent trained with 4 enemies performed well when validated with either 1 or 4 enemies. Therefore, it is desirable to always use 4 enemies during the training stage.

The stealthy agent's success rate is somehow related to how it perceives the world and how it reacts to its changes. Using reinforcement learning in our stealth game si-

	Trained with 1 Enemy		Trained with 4 Enemies	
	1 Enemy	**4 Enemies**	**1 Enemy**	**4 Enemies**
Q-Learning	97.7%	59.4%	96.5%	80.1%
Sarsa(λ)	96.5%	55.5%	96.6%	78.9%
Dyna-Q	97.2%	59.7%	97.2%	79.6%

Table 3.4: Performance achieved by agents trained with only 1 enemy or 4 enemies in the environment and with different RL methods.

mulator provides the stealthy agent with new world perceptions and a more diverse set of actions, resulting in more complex and realistic reactions to the world changes. We tested how the application of RL methods to the stealth simulator affected the success rate of the stealthy agent when compared to the approach presented in Chapter 2. We used the Q-Learning method with the RBRF, since it presented a slightly better result among the tested methods, and compared it with the agent that only uses the proposed stealthy path planning method, presented in Chapter 2, in order to achieve the destination. The results presented in Figure 3.8 show that the RL approach achieves higher success rates. Therefore, we successfully improved the success rate of the stealthy agent by using reinforcement learning.

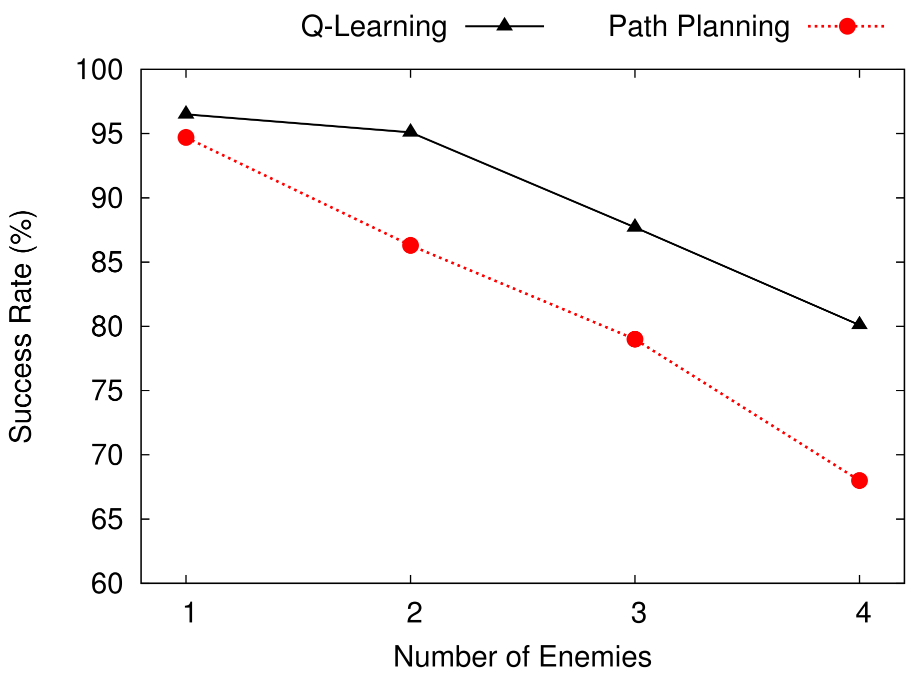


Figure 3.8: Success rate obtained for different number of enemies by using the Q-Learning method and the Navigation Mesh based stealthy path planning.

4 EVOLVED REWARD FUNCTIONS FOR STEALTH GAMES

Reinforcement Learning requires several parameters adjustment, such as state variables, action set, reward function and special rates ($\alpha, \gamma, \varepsilon, \lambda$, and N). Manually setting these parameters is time consuming and may require an expert's knowledge. One way to solve this problem is by using algorithms capable of defining such parameters automatically. Thus, we propose the use of evolutionary techniques in order to generate a reward function suited to the stealth problem without an expert's knowledge.

This chapter starts by reviewing some of the work related to evolved reward functions. The evolutionary methods used are detailed in the sequence. It is then showed how Evolutionary Computing was applied to evolve reward functions for the stealth game simulator.

4.1 RELATED WORK

Creating a reward function for a RL problem may be a difficult task, since it requires an expert's knowledge in order to hand-craft a good solution. It is possible to use search algorithms in order to find a good reward function without any prior knowledge of the problem. Singh et al. (2009) presented a work where a reward function was defined through Evolutionary Computing, requiring only the state formulation of the problem. Those authors presented a general framework for evolving reward functions that works as follows: define a state formulation for the problem, a set of possible environments, a set of possible reward functions, a history of states (or state-action pairs) generated while learning in a given environment using a given reward function, and a fitness function that takes into account the history of states. We can then search for a good reward function by measuring its fitness value using a fitness function. An extension of this work is then presented by Singh et al. (2010), where each reward function maps a numerical reward (positive or negative) for each state. Therefore, the reward function can be interpreted as a set of conditional statements and their respective reward. Another extension of that work is presented by Niekum et al. (2010) and Niekum et al. (2011), where the authors used Genetic Programming in order to evolve a population of reward functions. Each reward function was represented by a set of conditional statements and their respective reward, allowing for a more efficient search over large search spaces.

Evolutionary algorithms can also be used in order to evolve other aspects of a Reinforcement Learning problem, such as state variables, policy, action set, among others. When the number of possible state-action pairs is too large to fit in a table, it is necessary to use function approximation techniques in order to determine the value function of a given state-action pair. Whiteson and Stone (2006) uses the NEAT technique (STANLEY; MIIKKULAINEN, 2002) in order to evolve a set of function approximators based on Artificial Neural Network (ANN). The resulting method, called NEAT+Q, searches for a near optimal topology of an ANN capable of effectively defining the expected future rewards of each state-action pair. Another approach is presented by Girgin and Preux (2008), where genetic programming is used in order to evolve feature variables that define the current state. Each individual is comprised of a list of feature variables and their fitness corresponds to the performance of small RL trials using a state formulation based on the individual's feature variables.

4.2 EVOLUTIONARY COMPUTING

Evolutionary Computing (EC) represents a class of search algorithms for problem optimization that uses evolutionary processes in order to evolve a population and find an individual capable of solving a given problem. Each individual represents a candidate solution to the current problem and its representation is problem dependent. The population is comprised of a set of individuals. The process is divided between generations, where each generation is divided into: (i) evaluation, (ii) recombination, (iii) mutation and (iv) selection (ENGELBRECHT, 2007). The evaluation step is responsible for determining a *fitness* value for each individual that measures its capability in solving the problem. The recombination process is where two or more individuals are combined into one or more offspring. Recombination is repeated several times until a set of offspring is generated. Mutation is responsible for altering certain characteristics of an individual in order to explore its surroundings in the search space. The selection process is responsible for selecting the individuals that will be carried to the next generation and for selecting the individuals that will participate in the recombination process. Several generations are executed until a stopping criterion is met.

Evolutionary Computing is used for optimization problems, where search methods are adopted in order to find a good individual. Therefore, each individual is encoded with values for a set of variables from the problem being optimized. This structure is called *chromosome*, which is comprised by several *genes*. Each *gene* commonly represents a variable's value. The quality of an individual is measured by a fitness value that is calculated based on a fitness function, which depends of the problem.

The search process adopted by EC methods are based on Recombination and Mutation. Recombination is based on the evolutionary process called *Crossover*, where two chromosomes are recombined into two new chromosomes by swapping some of their genes. A general recombination process in EC takes two individuals (parents) and swaps some of their characteristics, resulting in their offspring. Therefore, the new individuals present characteristics from their parents. If two very distinct parents are selected to recombine, their offspring will probably present very different traits. For this reason, the recombination process helps to explore new regions in the search space.

Mutation in EC is also based on an evolutionary process, where random changes to a chromosome occurs. This process in EC occurs over any individual, resulting in a new individual with a similar chromosome, although with some mutated genes. Mutation plays a major role in many EC because it is the process responsible for inserting new values for genes, while the recombination only recombine the already existing values. Therefore, mutation is very important and it also helps to exploit an already explored region in the search space by making small changes to individuals in the population.

Selection represents the final step of a generation in a EC method. This process is responsible for selecting the population of the following generation, chosen between the individuals of the current generation and the new individuals generated through recombination and mutation. There are several selection methods: we can choose the fittest individuals, we can can simply replace the entire population with the new individuals (generated through mutation and recombination), etc.

4.2.1 GENETIC ALGORITHM

Genetic Algorithms (GA) were one of the first algorithmic models based on genetic systems (ENGELBRECHT, 2007). It was first presented by Fraser (1957), Bremermann

(1962) and Reed et al. (1967), although it only became popular after the extensive research presented by Holland (1975). Genetic Algorithms use a bit string or an array of characteristics in order to represent an individual. It also makes extensive use of the Selection and Recombination operators. Although mutation wasn't originally used in GA, it later became one of the operators of GA.

Genetic Algorithms usually represent a chromosome as a vector of characteristics. The early implementations of GA used only bit strings in order to represent an individual, although this was improved to a more general array of values in order to improve the expressive power of the method.

Recombination in GA was initially restricted to fixed point crossover, that is, the offspring would inherit a portion of a parent's traits until a specific point, and the remaining traits would be inherited from the other parent. Later implementations presented several new recombination methods, such as multiple points crossover, one-parent crossover, multi-parent crossover, and floating-point crossover. The recombination method used depends on the problem being optimized.

Parent selection is also a very important step in GA. It is responsible for selecting a set of parents for breeding. Selecting the right parents is essential in order to avoid local minimum or local maximum (depends on the problem's nature). In a minimization problem, a local minimum represents the best solution in a local region of the search space, although it is not necessarily the best global solution. The local minimum problem is represented in Figure 4.1. The local maximum problem is similar to the local minimum, although it applies to maximization problems. Therefore, selecting parents with different characteristics is important to escape from local regions, while recombination between similar parents is important to further explore a specific region. There are several parent selection techniques, such as tournament selection, random selection, and roulette selection.

The mutation operator in GA is performed over each individual generated through recombination. The mutation over each gene of an individual being mutated occurs with a probability p_m. Each mutated gene is assigned a new random value (in the simplest example). Mutation is important because it is responsible for inserting new genetic information in the population by randomly altering the existing genes.

The Selection operator in GA, as for other EC methods, is responsible for selecting the

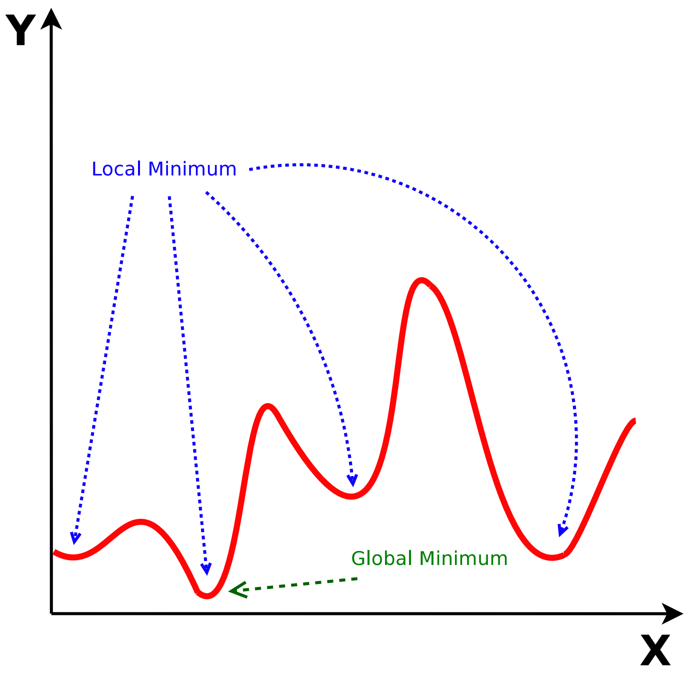


Figure 4.1: Example of Local and Global minima in a search space.

individuals to comprise the population of the following generation. As already mentioned for EC methods, there are several selection methods. A commonly adopted method in GA is elitism: this technique always selects the best individuals of the current population and passes it to the following generation. This helps the search process to maintain the best individuals in the population.

The GA operators (recombination, mutation, and selection) are not restricted and have already been used in many different forms, since these aspects of a GA are domain related. Therefore, each problem must be tackled differently. Algorithm 7 presents a general purpose genetic algorithm, without worrying with domain-related details.

Algorithm 7 Genetic Algorithm ()

```
1: Create an initial population $P$;
2: Calculate the fitness value for every individual $i \in P$;
3: repeat
4:     Select individuals for reproduction;
5:     Apply the Crossover operator and generate the offspring;
6:     Apply the Mutation operator;
7:     Calculate the fitness value of the new individuals;
8:     Select individuals to comprise the population P of the next generation;
9: until stopping criterion is met
```

4.3 METHODOLOGY

Using Reinforcement Learning for the stealth game problem allowed the stealthy agent to learn how to behave in any environment, as shown in Chapter 3. The main drawback presented is that building an efficient reward function for this problem is time-consuming and demanded an expert's knowledge. Thus, we used Evolutionary Computing in order to evolve an efficient reward function without any prior knowledge of the problem.

We chose Genetic Algorithms among the existing EC methods due to its simplicity and efficiency. The population in the GA is composed of several reward tables, each one representing an individual. The fitness of each individual consists on the success rate obtained over a few test episodes. We developed two different GA approaches for our stealth game problem and compared them in order to determine the best idea.

4.3.1 TABLE BASED GENETIC ALGORITHM

The first GA developed, called here *Table Based Genetic Algorithm (TBGA)*, evolves directly the reward table. The reward table is responsible for delivering a reward signal to the agent for every action executed. It works as a lookup table that takes into account the current state s of the agent and the action a performed, and returns the value stored at the position (s, a). Since there are 256 states and 4 possible actions, the reward table is represented by a 256x4 table (more details can be found in Section 3.3.2). Each individual in the population is then represented by a reward table. The fitness function of TBGA executes 100 simulations of the stealth game simulator using the Q-Learning algorithm and returns the success rate attained. Q-Learning was chosen based on its efficiency presented in Chapter 3 and we did not apply any reward for obtaining success or failure in order to test only the behavior depicted by the reward table. We used deterministic environments (randomly generated using the same seeds) for the fitness function in order to guarantee that every individual is tested in the same 100 environments. Thus, the fitness function always uses the same set of 100 environments (and enemies movement pattern) in order to determine the fitness of an individual. We also used 100 environments that are difficult to achieve success in an effort to guarantee that only the most fit obtain a high success rate.

Each generation of TBGA is divided in the same steps presented in Section 4.2: (i) evaluation, (ii) recombination, (iii) mutation, and (iv) selection. The population size was set to 40 individuals. TBGA also uses Elitism by always passing the 3 best individuals to the next generation.

The recombination process adopted for TBGA uses two parents to generate a single offspring. The recombination operates over each value of the new individual's reward table by assigning to it the weighted mean of the reward table's values of each parent. Each value of the offspring's reward table is assigned according to

$$R_{off}(s,a) = W \times R_{p1}(s,a) + (1-W) \times R_{p2}(s,a) \tag{4.1}$$

where W represents the weight associated to the first parent (we used $W = 0.85$), $R_{off}(s,a)$ represents the value at the position (s,a) of the offspring's reward table, and $R_{p1}(s,a)$ and $R_{p2}(s,a)$ represent the reward table's value at the position (s,a) of the first and second parents, respectively. The recombination process used for the TBGA is presented in Figure 4.2.

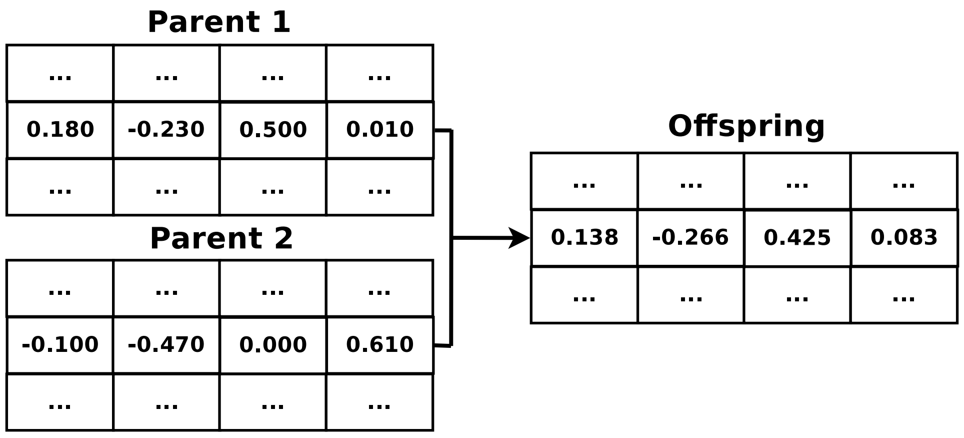


Figure 4.2: Example of the recombination process used for the TBGA. Note that only a portion of each individual is shown.

It is important to carefully select the pair of parents for the recombination process in order to guide the search process through different areas of the search space. Therefore, a Tournament selection method is used in order to select a pair of parents for breeding and allow all individuals to take part in the recombination process. For the first parent, two random individuals are selected from the population and the tournament operation selects the most fitted one. For the second parent, we select sequentially the first individual of the tournament in order to make possible that any individual participate at least of one tournament. The second individual of the second tournament is randomly selected. After performing two tournaments and selecting the parents, the crossover operation is performed and a new individual is generated. The number of crossovers performed during each generation is 34, resulting in 34 new individuals (the remaining individuals of the population are assigned according to the elitism and through the mutation of the elite, as described in the remainder of this section).

The mutation operator is applied over each value of an individual's reward table with a probability p_m (TBGA uses $p_m = 0.02$). Given that each individual in the genetic algorithm is a table of real values, we chose to apply random variations normally distributed over specific values of the reward table. This allows the mutation to perform subtle changes over several different values of a reward function without altering the main characteristics of an individual. We use the Box-Muller Transform (BOX; MULLER, 1958) in order to generate random numbers normally distributed and use these numbers as variation values for the current value being mutated. The mean of the Normal Distribution used for the Box-Muller Transform is set as the current value being mutated and the standard deviation is set as a parameter of the genetic algorithm, where higher values generates greater variations. The standard deviation of the Box-Muller Transform of TBGA was set to 0.6. The mutation process used for the TBGA is presented in Figure 4.3.

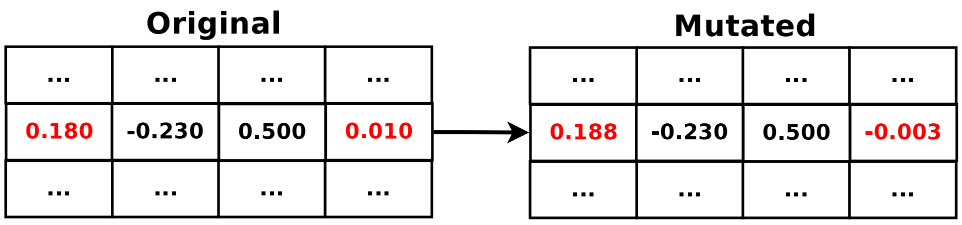


Figure 4.3: Example of the mutation process used for the TBGA. Note that only a portion of each individual is shown. The red values of the original individual are mutated using normally distributed variations.

Recombination is usually the process responsible for generating new individuals. Instead of only using recombination, we also generate new individuals using mutation. This helps the search process to further explore a specific region in the search space. TBGA generates new individuals by mutating the individuals from the elite and placing them in the next generation. Thus, 3 new individuals are created through mutation. The population of the following generation will then be comprised of the three best individuals of the current generation (elitism), three individuals generated through the mutation of the elite individuals, and 34 new individuals generated through recombination and mutation.

4.3.2 RULE BASED GENETIC ALGORITHM

A reward table can be created by applying several rules over the states and actions, such as we did for our Reinforcement Learning approach with the Rule-Based Reward Function, presented in Chapter 3. After defining a set of rules, we just need to pass through all states $s \in S$ and test all rules for each state, assigning the correct reward for each state-action pair visited. The following rule can be used to illustrate this situation: when the value of the state variable X is equal to Y, reward the agent for performing action Z (a state variable in this case will be any variable presented in Section 3.3.1 of Chapter 3). Thus, the agent will be rewarded for performing action Z with a large reward of 1.0 while giving a small punishment of -0.05 for executing any action $a \neq Z$ when the state variable X is Y. A large reward is adopted to accelerate the learning process, since the agent must learn quickly during the fitness function.

Evolving a reward table directly can be more difficult and time-consuming, since there are many possible values in a single individual. Another approach is to evolve a set of rules that defines a reward table instead of working directly with it. Therefore, each individual in the Rule Based Genetic Algorithm (RBGA) is represented by a set of simple rules similar to the one previously presented. Each individual may have a maximum number of rules, which depends upon the complexity of the problem to be optimized (we set the maximum number of rules to 20, given that this is enough to determine a good reward table for the stealth game problem). Each rule has a priority value assigned to it, where lower priority rules are applied first. Thus, we pass through every state $s \in S$ and check for each rule sequentially (following an ascending order in relation to the priority) when deriving the corresponding reward table of a given individual through its set of rules. An example of an individual is presented in Table 4.1.

RBGA follows the same basic structure of TBGA: each generation starts by evaluating each individual of the population, followed by the recombination process, mutation and selection. The population of the following generation of RBGA follows the same idea as TBGA: select the best three individuals of the current generation, mutate the elite of the

State Variable	Value	Action
Enemy Approaching	0	Move
Enemy visible	0	Move
Nearest Enemy Distance	0	Eliminate
Enemy Proximity	3	Hide

Table 4.1: Example of an individual of RBGA.

current generation, and recombine and mutate the individuals to generate the remaining individuals. The population size of RBGA was also set to 40 individuals and the elite size was set 3 individuals. Therefore, the difference between TBGA and RBGA lies in the individual's representation, recombination and mutation processes.

The fitness function works just like the one used for the TBGA: we perform 100 simulations using the Q-Learning algorithm and then return its success rate. We don't reward nor punish the agent for achieving success or failure.

The recombination process for RBGA also takes two parents in order to create a single offspring. The offspring is created by randomly selecting a set of rules from each parent. We pass through each rule of the set of rules of the first parent and assign it to the offspring with a probability of 50%. The same procedure is performed for the second parent. The recombination process ends when all rules of the second parent are visited or when the offspring reaches the maximum number of rules. It is important to note that a higher priority is assigned to the rules derived from the second parent, since they are added last (and are then associated to higher priority values). The recombination process used for the RBGA is presented in Figure 4.4. The parent selection mechanism follows the same tournament selection method described for TBGA.

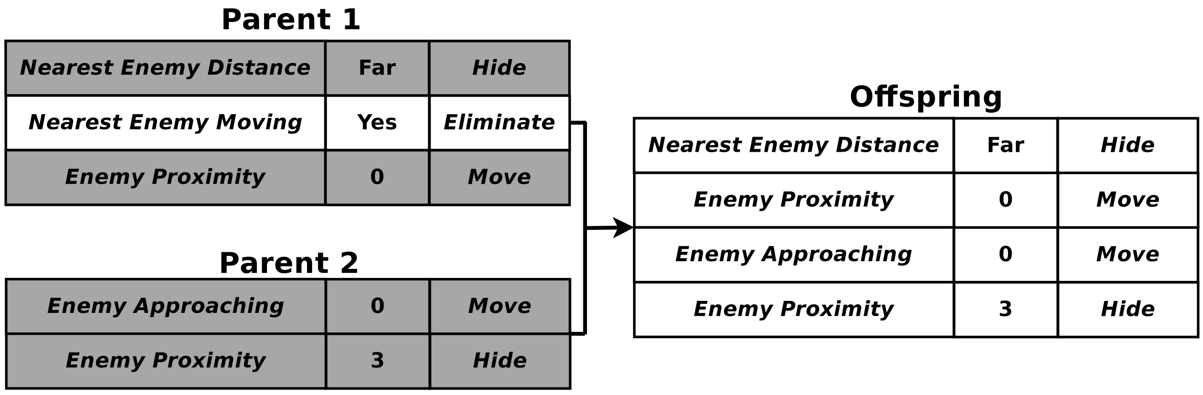

Parent 1
Nearest Enemy Distance | Far | Hide
Nearest Enemy Moving | Yes | Eliminate
Enemy Proximity | 0 | Move
Parent 2
Enemy Approaching | 0 | Move
Enemy Proximity | 3 | Hide
Offspring
Nearest Enemy Distance | Far | Hide
Enemy Proximity | 0 | Move
Enemy Approaching | 0 | Move
Enemy Proximity | 3 | Hide

Figure 4.4: Example of the recombination process used for the RBGA. The shaded rules were passed to the offspring.

Mutation in RBGA is applied to every new individual generated through recombination and to the elite. The mutation is applied to each rule individually. Changes are applied to each rule with a probability p_m, where the variation can occur over either the state variable, the value, or the action of the given rule. Thus, the mutation process is responsible for generating new rules from the already existing ones. We set $p_m = 0.1$ for RBGA, determined through preliminary tests. The mutation process used for the RBGA is presented in Figure 4.5.

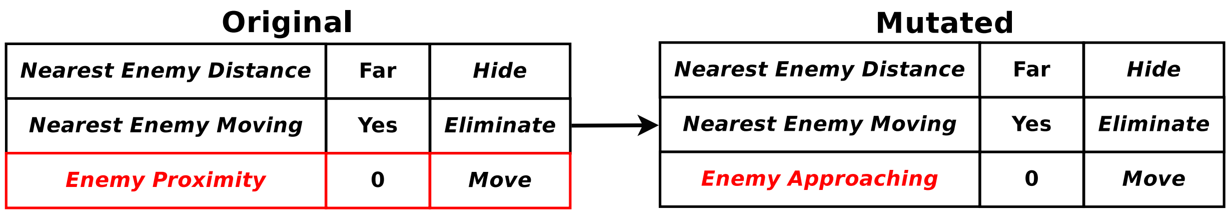


Figure 4.5: Example of the mutation process used for the RBGA. The original individual's rule highlighted in red represents the rule that will be mutated, where the red value represents the rule's region that will be effectively mutated.

4.4 EXPERIMENTAL RESULTS

We tested the two proposed Genetic Algorithm approaches to see if a good reward function could be found. Both algorithms are executed slowly due to the fitness function, that executes 100 episodes of the stealth game simulator (the same 100 environments and enemies' movement pattern were used in order to make the fitness function deterministic, as mentioned in Section 4.3). This allowed only a limited number of generations to be executed, although it was enough to study the behavior of both algorithms. 5 independent runs of TBGA and RBGA were performed. The convergence of Table Based Genetic Algorithm is present in Figure 4.6 and the convergence of Rule Based Genetic Algorithm is presented in Figure 4.7. Figure 4.6 and 4.7 present the fitness value of the best individuals.

It is easy to see through the analysis of Figure 4.6 that TBGA wasn't able to effectively define a good reward table, although it evolved from a very low success rate to a moderate fitness value. The best individual obtained presented a fitness value of 61. We tested how accurate this value was by testing the mean and confidence level of this reward table in 50 sets of 100 simulations of random environments. It achieved a success rate of $70.2\% \pm 1.43$ with a confidence of 95% for 100 random simulations. The fitness value is

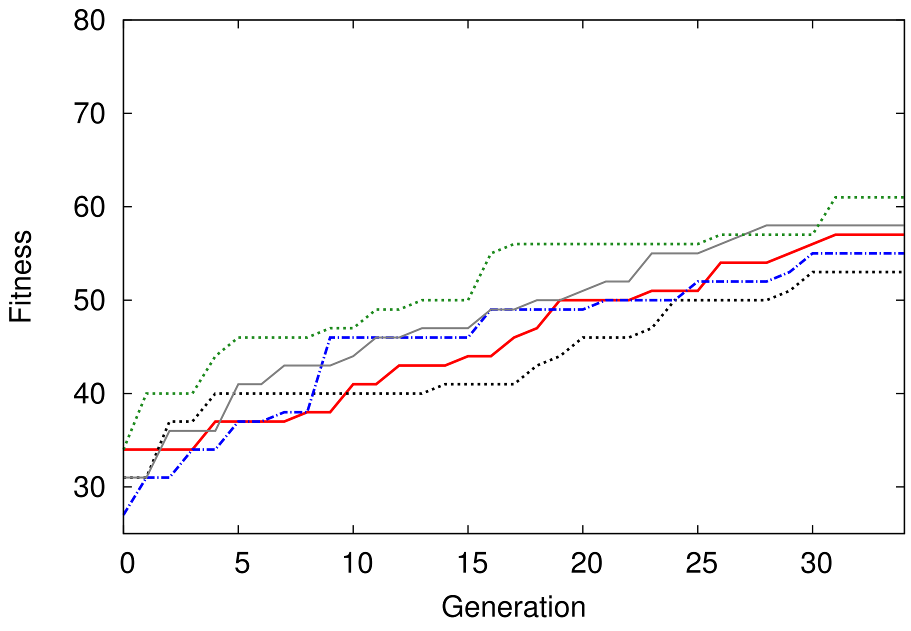


Figure 4.6: Convergence of 5 different instances of Table Based Genetic Algorithm, each with a random initial population.

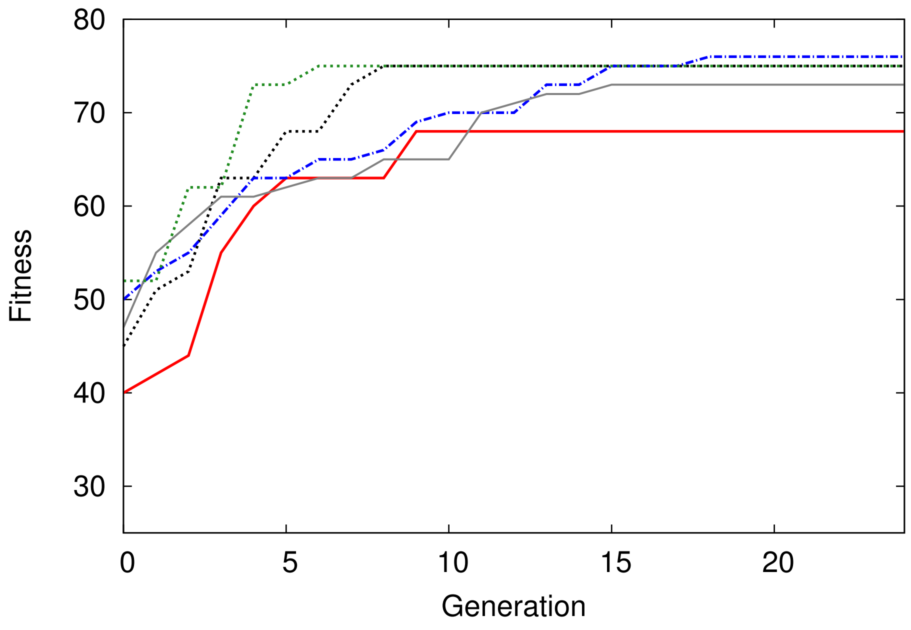


Figure 4.7: Convergence of 5 different instances of Rule Based Genetic Algorithm, each with a random initial population.

lower than its expected success rate in random simulations due to the hardness set to the 100 environments of the fitness function, as mentioned is Section 4.3. Since this value is not very inaccurate, we tested it through 1000 episodes with random environments and 4 enemies patrolling each scenario (as it was done for validation purposes, as mentioned

State Variable	**Value**	**Action**
Enemy Proximity	0	Move
Enemy Approaching	2	Move
Nearest Enemy Moving	No	Eliminate
Nearest Enemy Moving	Yes	Move
Nearest Enemy Distance	Far	Hide
Enemy visible	No	Hide
Enemy Approaching	0	Move

Table 4.2: Best individual found by RBGA.

in Section 3.4). The success rate achieved was $69.1\% \pm 0.47$ with a confidence of 95% for 1000 executions(see Section 3.4 of Chapter 3 for more details on the confidence level test).

Differently from TBGA, RBGA was capable of successfully finding a good reward table that reaches a high success rate and a more realistic behavior. The fitness value of the best individual found by RBGA was 75 (Figure 4.7), which is close to the results achieved by the Reinforcement Learning approach with the Rule-Based Reward Function. As mentioned for TBGA, the fitness function may not be completely accurate due to the low number of simulations (100) and the difficulty set to the environments of the fitness function. In this case, this reward function achieves a success rate of $76.0\% \pm 1.14$ with a confidence of 95% for 100 random simulations. Therefore, the best individual was tested for 1000 executions of the simulator with random environments and it achieved a success rate of $76.5\% \pm 0.47$ with a confidence of 95%. The set of rules adopted by this individual is presented in Table 4.2. It is important to note that the first rule has the lowest priority, while the last rule has the highest priority.

Figure 4.8 shows the mean fitness value of the best individuals for the 5 instances of TBGA and RBGA. One can see through this figure that both algorithms converged to a stable fitness value in only a few generations. As already mentioned, it was only possible to execute a limited number of generations due to time restrictions, but it is possible for the fitness value to be increased, although it could take several generations in order to achieve a small increment.

We then tested the best individuals found by TBGA and RBGA in the validation scenario, where they are used in 1000 executions with the same sequence of environments and variating the number of enemies (as done for the results presented in Chapter 2 and 3). The Q-Learning algorithm was used in order to train each agent with their respective reward table, found by the TBGA and RBGA. The success rate obtained by

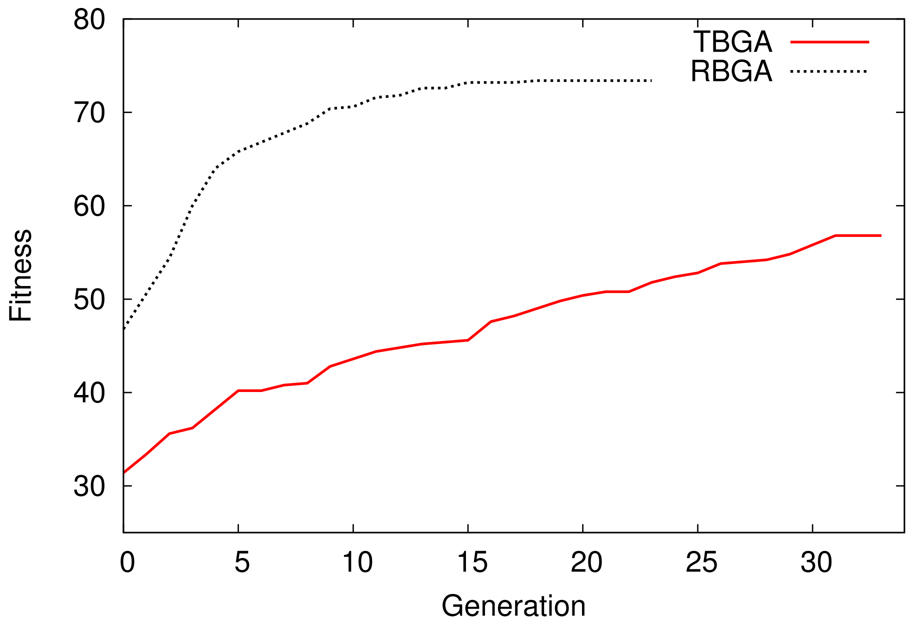


Figure 4.8: Mean fitness value of the best individuals of the 5 instances of the TBGA and RBGA measured for each generation.

these individuals were then compared with the stealthy path planning agent and the best reinforcement learning agent obtained here (Q-Learning). The results are shown in Figure 4.9, where we compare how well each agent performed in the 1000 validation environments with different number of enemies patrolling them. It is possible to notice that RBGA was capable of finding an individual with similar performance to that achieved by the Q-Learning agent. The behavior of the individual found by RBGA (shown in this video[1]) indicates that the agent rarely uses the eliminate action (only when the nearest enemy is halted), but it learned to hide correctly. The individual found by TBGA, on the other hand, was not capable of finding an efficient reward table, presenting a performance only slightly better than that obtained by the stealthy path planning agent. Regarding the behavior of the individual found by TBGA[2], it usually uses the **Move** action, just like the stealthy path planning agent, although it also tends to pursue the enemies and eliminate them when near.

One can notice through the results that we were capable of developing an effective Genetic Algorithm which finds a good reward function for the stealth game problem. Table Based Genetic Algorithm did not perform as well as expected, although it was able

[1]https://youtu.be/6unj0Ub0d7M
[2]https://youtu.be/7ks9WgCY1FY

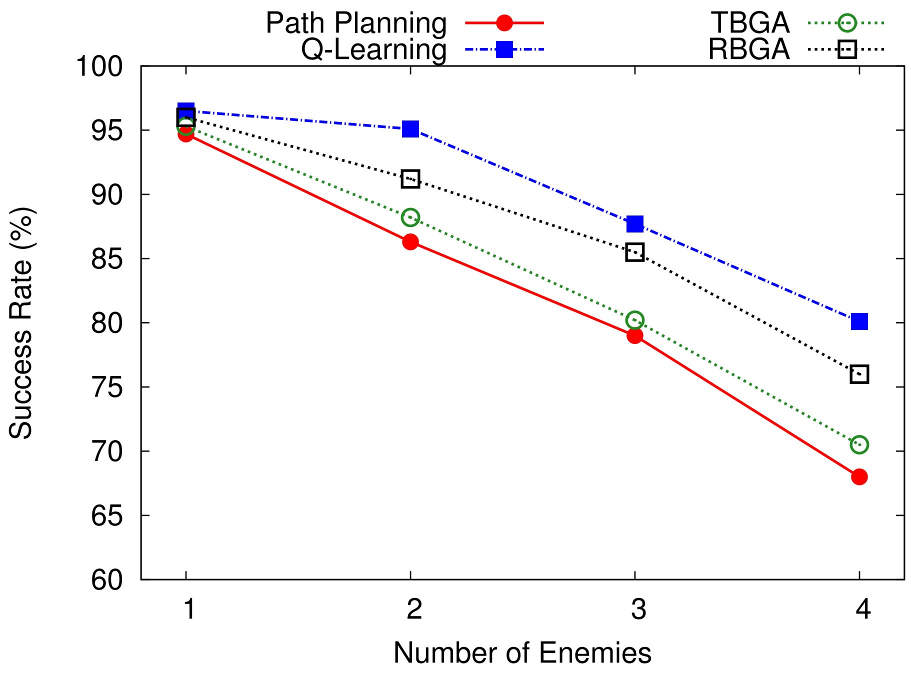


Figure 4.9: Success rate obtained for different number of enemies by using the Navigation Mesh based stealthy path planning, the Q-Learning method with the Rule-Based Reward Function, and Q-Learning using the reward function found by Table Based Genetic Algorithm and Rule Based Genetic Algorithm.

to evolve individuals from a very low fitness value. On the other hand, RBGA performed very well, finding a good behavior by evolving simple rules, allowing the definition of a good reward function without the need of an expert's knowledge.

5 CONCLUSION

We presented here how to effectively control a stealthy agent in stealth games. A stealth game simulator was developed in order to allow the implementation of different methods capable of controlling a stealthy agent. We proposed an efficient stealthy path planning method that allowed the generation of stealthy paths in a random environment patrolled by several enemy agents. Reinforcement learning was later used in conjunction with the proposed stealthy path planning method in order to allow the stealthy agent to perform different actions and behave realistically in the stealth game simulator. The reward function of the reinforcement learning method was defined using two approaches: based on specific strategies defined by an expert, and through evolutionary computing. The evolutionary method used a simple genetic algorithm in order to define a good reward function without any prior knowledge of the problem. Therefore, our approach represents a good alternative for commercial stealth games, in the sense that it uses simple algorithms and it is capable of finding a good behavior automatically.

Stealth has become a common element in modern games. One problem that developers face is the lack of techniques capable of controlling an agent that exhibits stealthy behaviors. The stealthy path planning presented here is capable of defining a stealthy path in real-time in an environment patrolled by several enemy agents. The environment is represented by a Navigation Mesh that separates cover areas from normal areas. The stealthy path is calculated using the A*. We then use a B-Spline in order to smooth the resulting path.

Reinforcement Learning represents a powerful tool that allows the agent to learn through experience. Although it presents a great potential for several real world applications, reinforcement learning is very difficult to be modeled for complex problems. Several of its aspects are very sensitive to the application at hand, such as the action set, the state formulation, the reward function, and the several parameters. These components must be carefully set in order to allow the reinforcement learning to properly work for a given problem. This work showed how we can effectively apply reinforcement learning to the stealth game problem. We used a set of high-level actions in order to better control the agent and to reduce the overall learning complexity by reducing the number of possible actions. One of these high-level actions consisted on defining a stealthy path to the destination point by using the stealthy path planning method proposed here. We also used other high-level actions that presents complex behaviors when executed. The state formulation consisted of important informations that were discretized in order to reduce the learning complexity of the agent. Finally, we created a reward function that delivers small rewards for accomplishing specific tasks defined by an expert and high rewards when succeeding or failing. This allows the agent to learn a specific behavior and also to adapt in case the specified strategy is not effective.

Defining a good reward function may be difficult and time consuming for complex problems, such as the stealth game problem. Therefore, we developed a genetic algorithm capable of evolving a reward function. Two different approaches were proposed: the first (Table Based Genetic Algorithm – TBGA) evolves a reward table that represents a reward function, while the latter (Rule Based Genetic Algorithm – RBGA) evolves a set of rules that defines a reward table. We showed that evolving a set of rules is more effective, resulting in a good reward function with only a few generations (approximately 30 generations). The main flaw of this method is that the reward function found by the RBGA is very restrictive: it delivers high reward signals when the agent completes a rule, not allowing the agent to change its strategy in case the enemies start behaving differently, for example. This problem can be solved by adjusting the reward values and inserting the Simple Reward Function in conjunction, in order to allow the agent to adapt in case the defined strategy is not good.

There are several real world problems that are much more complex than that presented here. Commercial stealth games, for example, are much more complex than the simulator that we built: the stealthy agent is usually capable of performing several other actions (even in a high-level), such as hiding an enemies' body to avoid attracting the attention of other patrols, use distraction techniques (throwing a rock to make the patrol move away from the agent's position, for example), and other weapons with specific behaviors. The state formulation might also become more complex if the stealth game uses a 3D world (which is the case of most commercial games). Therefore, the action and state formulation might become too large to fit in a Q-Table. The same may happen to the reward function. The table must then be replaced by any method capable of approximating a function (in this case, the future rewards and the reward functions). We could then use evolutionary methods in order to evolve a Neural Network, for example, that defines a good reward function without any prior knowledge of the problem (similar to what Whiteson and Stone (2006) propose).

We showed here that only rewarding or punishing an agent when it achieves success or failure may not be a good solution for some complex reinforcement learning problems. There might be cases where the agent performs several bad decisions during an episode, but it eventually ends up completing its objective. Hence, we could change the reward function to define scores to the agent's performance in order to achieve a good behavior. These scores can be defined by an expert through a set of rules, similar to the Rule-Based Reward Function presented here. Another possible approach is to use gameplay records of human players and measure the similarity of the stealthy agent's behavior with the behavior depicted by the player. Whenever the agent behaves similarly to the human player, it receives a reward, and if it behaves differently, it receives a punishment. This allows the agent to learn a reward function automatically by analyzing the gameplay of a human player.

REFERENCES

AMATO, C.; SHANI, G. High-level reinforcement learning in strategy games. In: **Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1**, 2010. (AAMAS 2010), p. 75–82.

ANDRADE, G.; RAMALHO, G.; SANTANA, H.; CORRUBLE, V. Automatic computer game balancing: A reinforcement learning approach. In: **Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems**, 2005. (AAMAS 2005), p. 1111–1112.

BIRGERSSON, E.; HOWARD, A.; SUKHATME, G. Towards stealthy behaviors. In: IEEE. **International Conference on Intelligent Robots and Systems. (IROS 2003). Proceedings IEEE/RSJ**, 2003. v. 2, p. 1703–1708.

BORTOFF, S. Path planning for uavs. In: **American Control Conference, 2000. Proceedings of the 2000**, 2000. v. 1, n. 6, p. 364–368 vol.1.

BOX, G. E.; MULLER, M. E. A note on the generation of random normal deviates. **The annals of mathematical statistics**, n. 29, p. 610–611, 1958.

BREMERMANN, H. J. Optimization through evolution and recombination. In: YOVITS, M. C.; JACOBI, G. T.; GOLSTINE, G. D. (Ed.). **Proceedings of the Conference on Self-Organizing Systems**, 1962. p. 93–106.

COLEMAN, R. Fractal analysis of stealthy pathfinding aesthetics. **Int. J. Comput. Games Technol.**, Hindawi Publishing Corp., New York, NY, United States, v. 2009, p. 21–27, jan. 2009.

CRANDALL, R. W.; SIDAK, J. G. Video games: Serious business for america's economy. **Entertainment Software Association Report**, 2006.

D'SILVA, T.; JANIK, R.; CHRIEN, M.; STANLEY, K. O.; MIIKKULAINEN, R. Retaining learned behavior during real-time neuroevolution. **Artificial Intelligence and Interactive Digital Entertainment**, American Association for Artificial Intelligence, 2005.

ENGELBRECHT, A. P. **Computational Intelligence: An Introduction**. 2nd. ed., 2007.

FRASER, A. S. Simulation of genetic systems by automatic digital computers I. Introduction. **Australian Journal of Biological Science**, v. 10, p. 484–491, 1957.

GERAERTS, R.; OVERMARS, M. H. The corridor map method: A general framework for real-time high-quality path planning: Research articles. **Comput. Animat. Virtual Worlds**, John Wiley and Sons Ltd., Chichester, UK, v. 18, n. 2, p. 107–119, maio 2007.

GERAERTS, R.; SCHAGER, E. Stealth-based path planning using corridor maps. **Computer Animation and Social Agents (CASA2010)**, 2010.

GIRGIN, S.; PREUX, P. Feature discovery in reinforcement learning using genetic programming. In: O'NEILL, M.; VANNESCHI, L.; GUSTAFSON, S.; ALCÁZAR, A. E.; FALCO, I. D.; CIOPPA, A. D.; TARANTINO, E. (Ed.). **Genetic Programming**, 2008, (Lecture Notes in Computer Science, v. 4971). p. 218–229.

GLAVIN, F. G.; MADDEN, M. G. Adaptive shooting for bots in first person shooter games using reinforcement learning. **IEEE Trans. Comput. Intellig. and AI in Games**, v. 7, n. 2, p. 180–192, 2015.

GRAEPEL, T.; HERBRICH, R.; GOLD, J. Learning to Fight. 2004.

HALE, H.; YOUNGBLOOD, M.; DIXIT, P. N. Automatically-generated Convex Region Decomposition for Real-time Spatial Agent Navigation in Virtual Worlds. In: **Proceedings of the 4th AAAI AI in Interactive Digital Entertainment Conference**, 2008.

HART, P.; NILSSON, N.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. **IEEE Transactions on Systems Science and Cybernetics**, v. 4, n. 2, p. 100–107, July 1968.

HE, P.; DAI, S. Stealth coverage multi-path corridors planning for uav fleet. In: **Mechatronic Sciences, Electric Engineering and Computer (MEC), Proceedings 2013 International Conference on**, 2013. p. 2922–2926.

HEFNY, A. S.; HATEM, A. A.; SHALABY, M. M.; ATIYA, A. F. Cerberus: Applying supervised and reinforcement learning techniques to capture the flag games. In: DARKEN, C.; MATEAS, M. (Ed.). **AIIDE**, 2008.

HOLLAND, J. H. **Adaptation in Natural and Artificial Systems**, 1975.

JARVIS, R. Distance transform based visibility measures for covert path planning in known but dynamic environments. In: **International Conference on Autonomous Robots and Agents**, 2004. p. 396–400.

JARVIS, R. A. Collision-free path trajectory using distance transforms. In: **National Conference and Exhibition on Robotics**, 1984.

JOHANSSON, A.; DELL'ACQUA, P. Knowledge-based probability maps for covert pathfinding. In: **Motion in Games**, 2010. p. 339–350.

JUNG, D.; TSIOTRAS, P. On-line path generation for small unmanned aerial vehicles using b-spline path templates. In: **AIAA Guidance, Navigation and Control Conference, AIAA**, 2008. v. 7135.

KALLMANN, M. Navigation queries from triangular meshes. In: BOULIC, R.; CHRYSANTHOU, Y.; KOMURA, T. (Ed.). **Motion in Games**, 2010, (Lecture Notes in Computer Science, v. 6459). p. 230–241.

KAMPHUIS, A.; ROOK, M.; OVERMARS, M. H. Tactical path finding in urban environments. In: **First International Workshop on Crowd Simulation**, 2005. p. 51–60.

MARZOUQI, M.; JARVIS, R. Covert path planning for autonomous robot navigation in known environments. In: CITESEER. **Proc. Australasian Conference on Robotics and Automation, Brisbane**, 2003.

MARZOUQI, M.; JARVIS, R. Covert path planning in unknown environments with known or suspected sentry location. In: **IEEE/RSJ International Conference on Intelligent Robots and Systems. (IROS 2005).**, 2005. p. 1772–1778.

MARZOUQI, M.; JARVIS, R. A. Covert robotics: Covert path planning in unknown environments. In: CITESEER. **Proceedings of the Australasian Conference on Robotics and Automation, Canberra, Australia**, 2003.

MARZOUQI, M. S.; JARVIS, R. A. New visibility-based path-planning approach for covert robotic navigation. **Robotica**, v. 24, p. 759–773, 11 2006.

MCPARTLAND, M.; GALLAGHER, M. Reinforcement learning in first person shooter games. **Computational Intelligence and AI in Games, IEEE Transactions on**, v. 3, n. 1, p. 43–56, March 2011.

MENDONÇA, M. R.; BERNARDINO, H. S.; NETO, R. F. Simulating human behavior in fighting games using reinforcement learning and artificial neural networks. In: **In Proceedings of the XIV SBGames**, 2015. p. 26–32.

MILLINGTON, I.; FUNGE, J. **Artificial Intelligence for Games**. 2nd. ed., 2009.

MNIH, V.; KAVUKCUOGLU, K.; SILVER, D.; GRAVES, A.; ANTONOGLOU, I.; WIERSTRA, D.; RIEDMILLER, M. Playing atari with deep reinforcement learning. **NIPS Deep Learning Workshop**, p. 1–9, 2013.

MOHAN, M.; SAWHNEY, R.; KRISHNA, K.; SRINATHAN, K.; SRIKANTH, M. Covering hostile terrains with partial and complete visibilities: On minimum distance paths. In: **IEEE/RSJ International Conference on Intelligent Robots and Systems. (IROS 2008)**, 2008. p. 2572–2577.

NIEKUM, S.; BARTO, A.; SPECTOR, L. Genetic programming for reward function search. **IEEE Transactions on Autonomous Mental Development**, v. 2, n. 2, p. 83–90, June 2010.

NIEKUM, S.; SPECTOR, L.; BARTO, A. Evolution of reward functions for reinforcement learning. In: **Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation**, 2011. (GECCO '11), p. 177–178.

PARK, J.-H.; CHOI, J.-S.; KIM, J.; LEE, B.-H. Roadmap-based stealth navigation for intercepting an invader. In: **IEEE International Conference on Robotics and Automation, ICRA**, 2009. p. 442–447.

RAVELA, S.; WEISS, R.; DRAPER, B.; HANSON, B. P. A.; HANSON, A.; RISEMAN, E. Stealth navigation: Planning and behaviors. In: **Proceedings of ARPA Image Understanding Workshop**, 1994. p. 1093–1100.

REED, J.; TOOMBS, R.; BARRICELLI, N. A. Simulation of biological evolution and machine learning. **Journal of theoretical biology**, Elsevier, v. 17, n. 3, p. 319–342, 1967.

ROOK, M.; KAMPHUIS, A. Path finding using tactical information. **Eurographics/ACM SIGGRAPH Symposium on Computer Animation**, 2005.

RUMMERY, G. A.; NIRANJAN, M. **On-Line Q-Learning Using Connectionist Systems**, 1994.

SCHRUM, J.; MIIKKULAINEN, R. Solving interleaved and blended sequential decision-making problems through modular neuroevolution. In: **Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2015)**, 2015. p. 345–352.

SEGAL, M.; AKELEY, K. **The OpenGL Graphics Interface**, 1994.

SINGH, S.; LEWIS, R.; BARTO, A.; SORG, J. Intrinsically motivated reinforcement learning: An evolutionary perspective. **IEEE Transactions on Autonomous Mental Development**, v. 2, n. 2, p. 70–82, June 2010.

SINGH, S.; LEWIS, R. L.; BARTO, A. G. Where do rewards come from. In: **Proceedings of the annual conference of the cognitive science society**, 2009. p. 2601–2606.

SMITH, M.; LEE-URBAN, S.; AVILA, H. Muñoz. Retaliate: Learning winning policies in first-person shooter games. In: **AAAI**, 2007. p. 1801–1806.

SONI, B.; HINGSTON, P. Bots trained to play like a human are more fun. In: **IEEE International Joint Conference on Neural Networks. IJCNN 2008. (IEEE World Congress on Computational Intelligence)**, 2008. p. 363–369.

STANLEY, K. O.; MIIKKULAINEN, R. Evolving neural networks through augmenting topologies. **Evolutionary Computation**, v. 10, n. 2, p. 99–127, 2002.

STENTZ, A. Optimal and efficient path planning for partially-known environments. In: **Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)**, 1994. v. 4, p. 3310 – 3317.

STENTZ, A. Cd*: A real-time resolution optimal re-planner for globally constrained problems. In: **AAAI/IAAI**, 2002. p. 605–612.

SUTTON, R. S. Planning by incremental dynamic programming. In: **In Proceedings of the Eighth International Workshop on Machine Learning**, 1991. p. 353–357.

SUTTON, R. S.; BARTO, A. G. **Introduction to Reinforcement Learning**. 1st. ed., 1998.

TENG, Y. A.; DEMENTHON, D.; DAVIS, L. S. Stealth terrain navigation for multi-vehicle path planning. In: SAOUDI, A.; NIVAT, M.; WANG, P. S. P.; BUNKE, H. (Ed.), 1992. cap. Parallel Image Processing, p. 185–205.

TENG, Y. A.; DEMENTHON, D.; DAVIS, L. S. Stealth terrain navigation. **IEEE Transactions on Systems, Man, and Cybernetics**, v. 23, n. 1, p. 96–110, 1993.

TESAURO, G. Neurogammon wins computer olympiad. **Neural Comput.**, MIT Press, Cambridge, MA, USA, v. 1, n. 3, p. 321–323, set. 1989.

TESAURO, G. Practical issues in temporal difference learning. p. 257–277, 1992.

TEWS, A.; MATARIC, M. J.; SUKHATME, G. S. Avoiding detection in a dynamic environment. In: **IROS**, 2004. p. 3773–3778.

TREMBLAY, J.; TORRES, P. A.; RIKOVITCH, N.; VERBRUGGE, C. An exploration tool for predicting stealthy behaviour. In: **Proceedings of the 2013 AIIDE Workshop on Artificial Intelligence in the Game Design Process, IDP**, 2013. v. 2013.

WANG, L.-X.; ZHOU, D.-Y.; ZHENG, R. A stealthy path planning method for aircraft by constant azimuth. In: **Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2010 International Conference on**, 2010. p. 497–503.

WATKINS, C. J. C. H. **Learning from Delayed Rewards**. Tese (Doutorado) — King's College, Cambridge, UK, 1989.

WENDER, S.; WATSON, I. Applying reinforcement learning to small scale combat in the real-time strategy game starcraft: Broodwar. In: **CIG**, 2012. p. 402–408.

WENDER, S.; WATSON, I. Combining case-based reasoning and reinforcement learning for unit navigation in real-time strategy game ai. In: LAMONTAGNE, L.; PLAZA, E. (Ed.). **Case-Based Reasoning Research and Development**, 2014, (Lecture Notes in Computer Science, v. 8765). p. 511–525.

WHITESON, S.; STONE, P. Evolutionary function approximation for reinforcement learning. **J. Mach. Learn. Res.**, JMLR.org, v. 7, p. 877–917, dez. 2006.