TÉCNICO LISBOA

Learning User Profiles for Automatic Test of Games

Luís Miguel Martins Fernandes

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor(s): Prof. Manuel Fernando Cabido Peres Lopes
Prof. Rui Filipe Fernandes Prada

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Rui Filipe Fernandes Prada
Member of the Committee: Prof. Francisco António Chaves Saraiva de Melo

November 2021

Acknowledgments

I would like to thank everyone who was essential in this academic phase.

First, I would like to thank my supervisors Prof. Manuel Lopes and Prof. Rui Prada for support and guidance trough all Thursday meetings.

Secondly, to my family.

Third, to Pedro for providing the game and traces where agents will be tested. And the traces used to train those agents.

A big thanks to everyone.

Resumo

Com o aumento da complexidade dos jogos de video, está cada vez mais difícil e caro, testar todas as componentes do jogo de forma a garantir a sua qualidade. Para resolver esse problema, a indústria está se movendo em direcção à automação do processo de teste.

Esta tese tem como foco o desenvolvimento de um agente que terá a capacidade de testar diferentes recursos de um jogo. Além dessa capacidade, nosso agente poderá demonstrar diferentes tipos de comportamento durante o jogo. Esses comportamentos geralmente são gerados através da utilização de funções ou heurísticas, estas são criadas manualmente pelos designers do jogo. Em vez desse método, propomos uma alternativa em que os comportamentos serão gerados com base na observação de um jogador, desta forma vamos criar perfis mais semelhantes ao comportamento humano do jogador.

O que conseguimos aqui foi a criação de um agente que foi capaz de jogar diferentes níveis de um jogo, que ele aprendeu apenas com as observações dos jogadores. Ele foi até capaz de jogar um nível que ele nunca havia visto antes, e ele não precisou de nenhum treino extra.

Palavras-chave: Perfis de jogadores, Aprendizagem por Reforço Inverso, Teste Automatizado, Entropia Máxima, Aprendizagem

Abstract

Since the increase of the complexity of video games, its getting harder and expensive to fully test, and assure the quality of the game components. To solve this issue, the industry is moving towards the automation of the testing process.

This thesis focuses on the development of an agent that will have the capability of testing different video game features. In addition to this capability, our agent will be able to have different types of behaviour when playing through the game. These behaviors are usually generated through the use of functions or heuristics that are manually created by the game designers. Instead of this method, we proposed a method that will generate the behaviours based on observing a player, this way we are going to create profiles that are closer to the human player behaviour.

What we achieved here the creation of an agent that was capable of play different levels of a game, that he learn only from player observations. This was even capable of playing a level that he never saw before, and he was able to do this without requiring any extra training.

Keywords: Player profiles, Inverse Reinforcement Learning, Automated Testing, Max Entropy, Apprenticeship learning

Contents

List of Tables

List of Figures

Nomenclature

Greek symbols

γ	MDP discount factor.
π	Policy

Subscripts

(s, a)	MDP state action pair.

Superscripts

A	MDP action space.
P	MDP Transition Probability.
R	MDP Reward function.
S	MDP state space.

Chapter 1

Introduction

1.1 Motivation

One of the most important parts of software development, in particular for video game development, is the testing process. In this process the various features and concepts of the game are evaluated, considering a set of defined metrics. This is done with the goal of ensuring the quality of the video game, regarding technical quality and experience.

Typically, this process is performed manually by groups of people, hired by the developer company or, in some cases, by members of the development team. One problem with this manual approach is the process scalability, since the results obtained are related to the number of tests performed and who performs them. Also, to increase the test coverage, it may be necessary to test the game with different players. This scalability greatly affects the time and resources spent during the testing phase.

To solve the scalability problem, the testing process is starting to be automated. Meaning the tests are now being done by software agents, rather than people. This automation introduced many advantages to the testing process. Regarding the technical quality, with test automation, it allows for a reduction in the feedback cycle of new video game features and accelerates their validation. By increasing the number of tests, the coverage of the testing process will also increases. This will help to test all of video game features, and their overall quality.

1.1.1 Learning Player Profiles

With the reduction of the number of real users that perform the testing process, it will be necessary for agents, to have behaviours that are similar to real users. There are various techniques of creating these said agents. One of those techniques is based on collecting player data. Then use this data in order to make the agent learn the profiles/personas of players, based on their observable behavior. This learning process is defined as apprenticeship learning. Considering that the agent will try to infer the goal of the player and not try to copy it directly. This works as an alternative to having to manually create policy or heuristics for agents to follow.

To carry out this type of learning, the approach followed in this thesis is based on Inverse Reinforcement Learning(IRL).

With these agents, it will be possible to see how different types of players interact on a level. What a player considers most important to play. And finally how do certain levels allow for different types of gameplay. For example, a certain level can cause a player to explore more or be more aggressive.

1.1.2 Profile clustering

As mentioned before, the approach analysed here is based on observed data. Although, after we gather the said data, we cannot use it directly on the IRL algorithm. Since different players play differently. This will cause disturbance in the agent learning, and will result in very poor results.

So first we need to organize the player data. To do this, we are going to use and analyse different clustering approaches. This will create clusters of players, where we will have a different type of behavior in each one.

Then we are going to use the defined player clusters in the IRL algorithm to train the agent.

1.2 Objective

With this thesis, we aim to create agents capable of testing video games and are capable of following different profiles. To replicate these profiles, we are first developing a clustering tool that will be able to group the player data. Then we are going to present an approach based on the Inverse Reinforcement Learning framework. This approach will take the resulting clusters and then it will take the behaviour that exists within that cluster. With that behaviour it will generate a reward police that the behaviour is trying to maximize. Then we are going to generate a policy that replicates the observed behaviour.

In the end, we will be analysing the performance of these agents, by comparing them to their respective player data. With this we will see if using this approach is possible to replicate player profiles.

1.3 Thesis Outline

As mentioned before, this thesis is composed of three major parts following the introduction.

On Chapters 2 and 3, we explain the theory behind this area of research. We briefly present some important concepts such as: Clustering algorithms; We explain in detail IRL, then we delve into a specific IRL algorithm that we are interested in for this project: Max Entropy IRL. Finally, we present some work done in areas that are relevant to the topic of this thesis.

In Chapter 4, we will analyse the different cluster approaches, present the implementation of the proposed architecture, In Chapter 5 we present the method that we will use to evaluate our approach.

In Chapters 6 and 7 we will discuss obtained results, from which we will draw conclusions regarding the effectiveness of the approach followed.

Chapter 2

Background

In this section, we will give a more detailed explanation, regarding what is the method of agent learning, how the Inverse Reinforcement problem can be formulated, bases behind the analysed algorithm, and how such algorithm is capable of generating/capturing a player profile.

2.1 Important Concepts

2.1.1 Apprenticeship learning

As mentioned before, the propose of this thesis is to create an agent, who would be able to learn from human behaviour. This type of learning is usually called Apprenticeship learning. This method via inverse reinforcement learning, will try to infer the goal of the expert behaviour. In other words, it will learn a reward function from observations, which can then be used in reinforcement learning to generate policies. For example, if it discovers that the goal is to hit a nail with a hammer, it will ignore the blinks and scratches from the expert, as they are irrelevant to the goal.

2.1.2 Markov Decision Process

The base for reinforcement learning and inverse reinforcement learning is the Markov Decision Process formulation.

A (finite-state) Markov Decision Process (MDP) can be represented as the following tuple $M = (S, A, \{P_{sa}\}, \gamma, R)$ where S is S is a finite set of states; A is a set of actions; P_{sa} is a set of state transition probabilities (here, P(s,a) is the state transition distribution upon taking action a in state s); $\gamma \in [0, 1]$is a discount factor; and $R : S \mapsto A$ is the reward function,

2.1.3 Max Entropy theorem

The principle of maximum entropy states that the most appropriate distribution, to model a given set of data, is the one with highest entropy among all those that satisfy the constrains of our prior knowledge.

2.2 Clustering

2.2.1 Definition

Is an unsupervised learning technique where the goal is to divide data points into a number of groups. Such that the data points in the same groups, are more similar to other data points in the same group, than those in other groups.

Therefore, when providing data, that is described by a set of variables, the goal is to identify the constraints that put each record in a specific cluster. In some manner, the goal is just to recognize what are the most important variables, and what are the values assigned to them in each cluster.

We can say that the act of clustering, is similar to classifying. But, in clustering we are discriminating among records, and in classification, we are discriminating as a function of the target variable.

In the book Hartigan [1] about clustering, the term is general for formal, planned, purposeful, or scientific classification.

This technique is used on a variety of areas for example:

- Economy - where customer segmentation has been its most paradigmatic case.
- Biology - clustering can be applied in phylogenetics for identifying groups of similar organisms, and in transcriptomics, to recognize groups of genes with related expression patterns.

2.2.2 Clustering example

Here we present a clustering example, this one represents how clustering will work in our approach.

When playing a video game, human players have different types of inner goals. This had been shown in several studies like Nacke et al. [2]. In this paper they proposed the BrainHex that has the following categories: Achiever, Conqueror, Daredevil, Mastermind, Seeker, Socialiser and Survivor. In the case of paper, players filled out a questionnaire to find the category they belong to. Instead of this approach we are going to group the players based on player performance metrics.

In this scenario, the measure of similarity, is usually computed between each pair of players, afterwards clusters of similar player patterns are constructed.

Here the classification technique is based on first selecting one or two important variables by expert judgement. And then classifying them according to these variables, in this case player performance metrics.

In addition, we can use clustering to see what performance metrics are the most important or discriminant. Then we check what are the patterns or the player profiles assigned to the players in each one of the clusters.

For example using only the metric of (reaching the end of a level), we are going to have two groups of players. Those who reach the end and those who don't. If we add more performance metrics it might be possible to find more interesting groups of players as such: we can find players who did not reach the level, but collect all important items in the level.

2.2.3 Algorithm types

As we can see, this technique is based on concept of similarity. Which may vary from algorithm to algorithm. In this thesis, we chose to analyze 3 different algorithms, that follow different models.

Connectivity models: As the name suggests, these models are based on the notion that the data points closer in data space exhibit more similarity to each other than the data points that are lying farther away. These models can follow two approaches. In the first approach, they start with classifying all data points into separate clusters and then aggregating them as the distance decreases. In the second approach, all data points are classified as a single cluster and then partitioned as the distance increases. Also, the choice of distance function is subjective. These models are very easy to interpret but lack scalability for handling big datasets. Examples of these models are hierarchical clustering algorithm and its variants.

Centroid models: These are iterative clustering algorithms in which the notion of similarity is obtained through the distance of a data point to the centroid of clusters. K-Means clustering algorithm is a popular algorithm that falls into this category. In these models, the number of clusters required at the end, has to be mentioned beforehand, which makes it important to have prior knowledge of the dataset. These models run interactively to find the optimal solution.

Distribution models: These clustering models are based on the notion of probabilities of all data points in the cluster belonging to the same distribution (For example: Normal, Gaussian). These models often suffer from overfitting. A popular example of these models is Expectation-maximization algorithm which uses multivariate normal distributions.

2.2.4 K-Means algorithm

As mentioned in the previous section K-means is a centroid based (or a distance-based) algorithm. The goal of this algorithm is to partition n points into k clusters, where each point belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster. The first time this algorithm was proposed was at Bell Labs in 1957, in relation to the field of signal processing by Stuart Lloyd as a technique for pulse-code modulation.

Now we are going to present an example of the k-means algorithm working.

Let us consider P as set of 8 points as shown in figure 2.1. Now we are going to apply the algorithm to create clusters for these points.

Figure 2.1: Set of points P

Step 1: Choose the number of clusters k.

The first step of the algorithm is to pick the number of clusters, k.

Step 2: Select random k points to be the centroids.

For this situation we want only 2 clusters (k=2). In figure 2.2 we can see the 2 centroids represented by the color red and green.

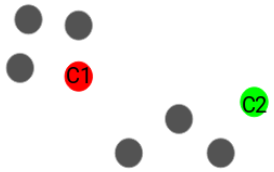


Figure 2.2: Random centroids

Step 3: Assign the points to the cluster with the closest centroid.

Once the centroids have been initialized, we can assign each of the points to the cluster with the closest centroid. In figure 2.3 we can see that the points that are closer to the red point, are assigned to the red cluster, while the points closer to the green point are assigned to the green cluster.

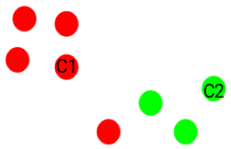


Figure 2.3: Cluster assignment

Step 4: Recompute the centroids for the new formed clusters.

Now, that we have assigned all points to either cluster, we now need to compute the centroids of these new formed clusters.

In figure 2.4 we present the new cluster centroids as red and green crosses.

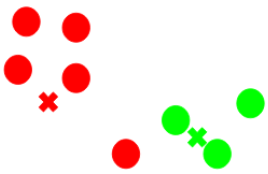


Figure 2.4: New centroids

Step 5: Repeat steps 3 and 4

Now we repeat steps 3 and 4 until we satisfy the stopping criteria. In figure 2.5 we present the final result.

Figure 2.5: Final K-means centroids

As we can see, this algorithm can be divided into 2 steps or phases:

- Assignment step - where we assign points to the cluster with the closest centroid
- Update step - where we update the centroids of the new formed clusters

To define the stopping criteria, we can use one of the three stopping criteria to stop the K-means algorithm:

1. Centroids of newly formed clusters do not change
2. Points remain in the same cluster
3. Maximum number of iterations are reached

In short, we can stop the algorithm if the centroids of new formed clusters are equal to the previous ones. So after several iterations, if the centroids for all clusters aren't changing, we can assert that the algorithm is not learning any new pattern, and we can stop it's training. Another sign that we should stop the training process, is if the points remain in the same cluster even after training the algorithm for multiple iterations. Finally, a simple way to stop the training is to impose a maximum number of iterations on the training process. For example, we can set the maximum number of iterations to 100. Thus, the process will repeat for 100 iterations before stopping.

Unfortunately, this algorithm has some limitations: The fact that the number of clusters, k is an input parameter. An inappropriate choice of k may lead to poor results. One way to solve this, before performing k-means, it is important to run diagnostic checks to determine the number of clusters in the dataset. Another way is to execute the algorithm multiple times with different values of k, then comparing the results.

Other important limitation of k-means is its cluster model. The concept is based on spherical clusters that are separable so that the mean converges towards the cluster center. The clusters are expected to be of similar size, so that the assignment to the nearest cluster center is the correct assignment.

In algorithm 1 we present the pseudo code for the k-means algorithm, both the assignment step, and update step.

Algorithm 1 K-means algorithm

procedure K-MEANS(P, k) ▷ data set of points $P = \{P_1..P_k\}$, k number of clusters
  $choose\ k\ initial\ centers\ C = \{c1, ..., c_k\}$
  while $stopping\ criterion\ has\ not\ been\ met$ **do**
    for $i = 1, .., N$ **do** ▷ assignment step:
      $find\ closest\ center\ c_k \in C\ to\ instance\ p_i$
      $assign\ instance\ p_i\ to\ set\ C_k$
    end for
    for $i = 1, .., k$ **do** ▷ update step:
      $set\ c_i\ to\ be\ the\ center\ of\ mass\ of\ all\ points\ in\ C_i$
    end for
  end while
  return C
end procedure

2.2.5 Expectation–Maximization algorithm

As mentioned in the previous section the Expectation–Maximization algorithm or EM for short, is a model based on distribution.

Before explaining the algorithm, we first need to define what EM is. Em consists on a iterative optimization method, where the goal is to estimate some unknown parameter Θ, given measurement data U. Although, we don't receive some hidden variables J, which we need to integrate. We mainly want to maximize the posterior probability of the parameter Θ given the data U, marginalizing over J as the following equation:

$$\Theta^* = argmax_\Theta \sum_{J \in J^n} P(\Theta, J|U)\,. \tag{2.1}$$

The intuition behind EM is an old one: alternate between estimating the unknowns Θ and the hidden variables J. This idea has been around for a long time. However, instead of finding the best J $\in$ J given an estimate Θ at each iteration, EM computes a distribution over the space J. The algorithm was initially presented in the article Hartley [3]. A good reference to EM and its applications can also be found in Tanner [4].

We can derive EM in many ways, one of the most insightful being in terms of lower-bound maximization as shown in Neal and Hinton [5], Minka [6]. In Minka [6], the author derives the algorithm on that basis. And presents all important steps and equations.

Following that derivation, we can present the EM as an iterative algorithm, that alternates between 2 steps/modes: The first step that can be interpreted as constructing a local lower-bound on the posterior distribution. Since the bound is expressed as an expectation, the first step is named the "expectation-step" or E-step. On the second step, defined as the "maximization-step" or M-step, the bound is optimized, with the aim of improving the estimate for the unknowns.

This algorithm is most commonly used in clustering with a mixture model. In most of the cases, defined as a Gaussian mixture model or (GMM). Here a finite mixture model is defined as being the density for x which has the form of a weighted sum of component densities:

$$p(x, \Theta) = \sum_{c=1}^{k} P(\Theta, J|U)\,. \tag{2.2}$$

Gaussian because, if p(x—c, Θ) is Gaussian, then p(x—Θ) is a weighted sum of K Gaussians.

It is important to mention that here we are treating c, as a random variable whose value we don't know. If the value of c is known, then the density for x is just the cth component density. Hence, c is called the hidden assignment for x to one of the component densities. If we have several independent samples xi, then each has its own assignment variable c_i.

There are two main independent assumptions implicit in the finite mixture model. First, if Θ is known, then the observed data points are statistically independent. Second, if Θ is known, then the hidden assignments are independent.

$q(c1..c_N) = q1(c1)..q_N(c_N)$

Since c_i is only taken on values $j = 1..K$, the functions $q_i\ (c_i)$ can be represented by the N × K matrix Q where $q_{ij} = q_i(j)$. Furthermore, the independence of the data points xi, leads to $\log p(X, c, \Theta) = P_i \log p(xi, ci, \Theta)$.

Taking into account some of the equations presented in Minka [6] we can simplify the EM algorithm into:

- **E-Step**. From equation(8), compute

$$q_{ij} = \frac{p(x_i|c_i = j, \Theta)p(c_i = j|\Theta)}{\sum_j p(x_i|c_i = j, \Theta)p(c_i = j|\Theta)} = p(c_i = j|x_i, \Theta)\,. \tag{2.3}$$

- **M-Step**. From equation(13), maximize over Θ:

$$\sum_{ij} q_{ij} \log p(x_i, c_i = j, \Theta)\,. \tag{2.4}$$

In simpler words:

- **E-Step**. Estimate the missing variables in the dataset.
- **M-Step**. Maximize the parameters of the model, in the presence of the data.

To present a further discussion of EM applied to a mixture model, we reference Bishop et al. [7].

As we can see, the technique used in EM is similar to the K-Means technique. Since both are divided into 2 steps, that are related to the assignment step and update step, respectively. Nonetheless, each approach will reach different ends. In the case of k-means, we obtain a hard clustering solution, where a point belongs specifically to one, and only one cluster. In EM we obtain a soft solution, where a point has different probabilities of belonging to different clusters.

In terms of limitations, the EM also has some:

- Similar to k-means, it is necessary to know the number of desired clusters before we start.
- In EN we don't have the guarantee that we will find the globally best solution.
- The algorithm is slow for large datasets.

2.2.6 Hierarchical algorithm

Hierarchical clustering is another cluster technique that we decided to explore. The most dominant approach for this algorithm, has been the Agglomerative strategy. This strategy consists of a "bottom-up" approach, where each observation will start in its own cluster, and pairs of clusters are merged as one moves up the hierarchy.

Before we start this algorithm, there are things that we need to consider first, beyond the number of clusters we want.

Firstly, since we want to group the data, we need a way to measure each element size, and their distances relative to each other in order to decide which elements belong to a group. This choice of

distance metrics, should be made based on the theoretical concerns of the domain under analysis. In other words, the distance metric needs to define similarity in a way that is sensible for the field of study. The most used metrics are Euclidean, Manhattan and Chebyshev distances. Where the Euclidean distance is generally the most used one. In the example shown, the metric we are going to use, is Euclidean.

Secondly, we need to determine from where we are going to compute the distance. This is defined as the linkage criteria, which has different approaches. For example, in the single-linkage, the distance is computed between the two most similar parts of a cluster. In the complete-linkage, are taken in consideration the most different parts of a cluster. Or we can use the average-linkage that uses the center of the clusters.

Similarly to the distance metrics, when choosing a linkage criteria, we should analyze the domain of application.

Now we are going to present an example of how the Hierarchical algorithm works. As we mentioned before, the Agglomerative Hierarchical clustering, starts by treating each observation as a separate cluster. Then, it repeatedly executes the following two steps:

1. identify the two clusters that are closest together.
2. merge the two most similar clusters.

This iterative process continues until all the clusters are merged. Now let us illustrate an example of Hierarchical clustering. In figure 2.6 we present a set of points.

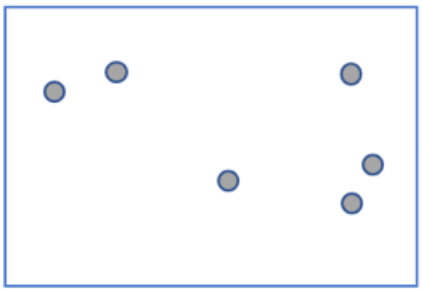
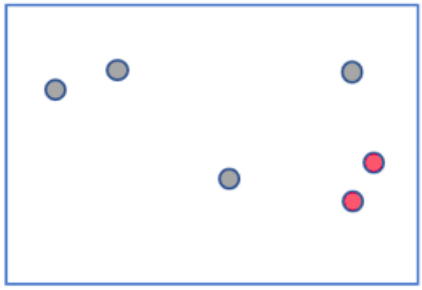
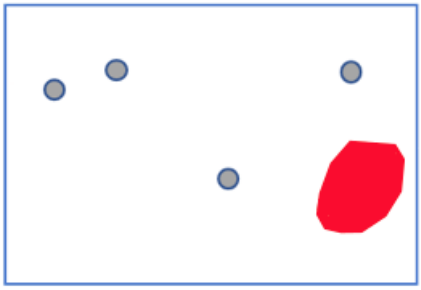


Figure 2.6: Set of points for Hierarchical

First, we identify the closest clusters

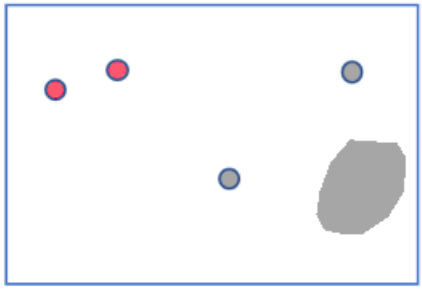


(a) select the closest clusters

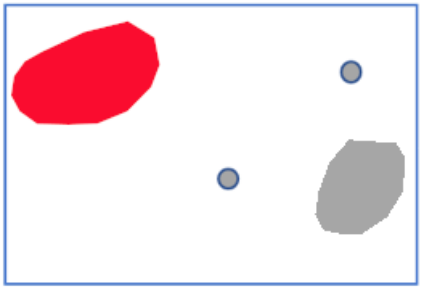


(b) merge them

Figure 2.7: group the closest clusters.

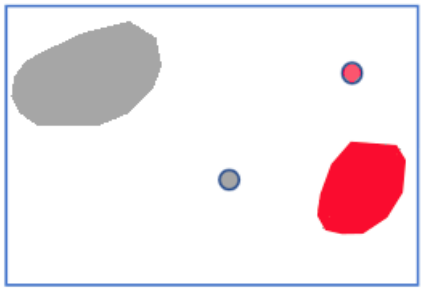


(a) select the closest clusters

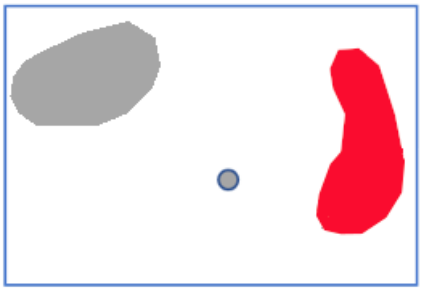


(b) merge them

Figure 2.8: group the closest clusters.

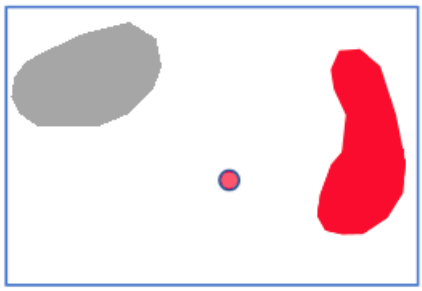


(a) select the closest clusters

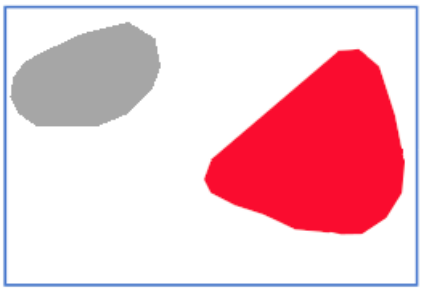


(b) merge them

Figure 2.9: group the closest clusters.

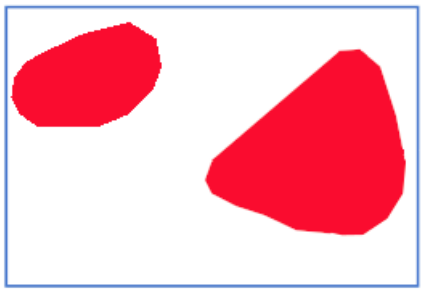


(a) select the closest clusters



(b) merge them

Figure 2.10: group the closest clusters.



(a) select the closest clusters

(b) merge them

Figure 2.11: group the closest clusters.

With this algorithm, we obtain what is called a dendrogram. This consists of a diagram, that represents the hierarchical relationship between points. In figure 2.12 we show the dendrogram obtained after applying the hierarchical algorithm to the points in figure 2.6 now represented by letters.

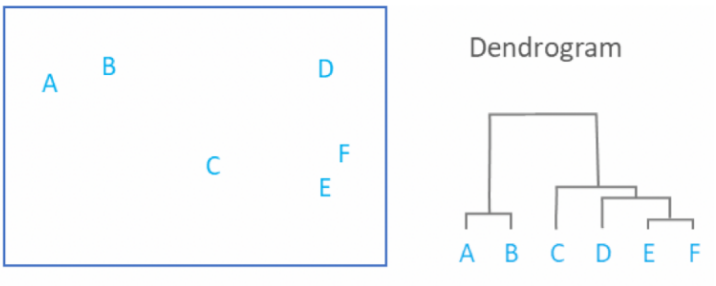


Figure 2.12: hierarchical dendrogram

In order to interpret the dendrogram, it is important to pay close attention to the height at which any two objects are joined together. As we saw in figure 2.7, points E and F are the closest ones. Thus, in the dendrogram, the height of the link that joins the points together is the smallest one. This way, with the height of the dendrogram we can establish the order in which the clusters were joined.

Seeing that, this algorithm does not use random centroids for its computation. Saying that, we can affirm that hierarchical clustering consists of a deterministic process, meaning cluster solution won't change when the algorithm is executed multiple times, with the same input data.

However, it is important to mention that the dendrogram consists of a summary of the distance matrix. And thanks to that summary, some information might be lost. For example, in the dendrogram it is suggested that point C is closer to D than B, but we can see that in scatterplot that this is not true.

2.3 IRL

2.3.1 History

The main objective in a traditional reinforcement learning (RL) scenario, is to obtain a decision process to produce a behavior in a way to maximize some predefined reward function.

In the inverse reinforcement learning initial defined by Russell [8], the goal is flipped, instead of learning a behaviour. The objective is to learn the goals, values or rewards of an agent, by observing his behavior. This is shown in figure 2.13.

The main purpose of this approach was to change the perspective of the traditional RL tasks. In RL, the agent aims to maximize a known reward function. With IRL, the main objective of the agent is to find to the reward function based on observed behaviour.

Meaning in the IRL framework, the task is to extract an approximation of a human's reward function, on a given task, through a set of human-generated driving data. This framework is often used in relation to tasks where the role is difficult to define.

For example, the task of driving a car. Here we can define a reward that will capture the basic behaviour of a responsible driver, such as avoiding pedestrians, and stopping at red lights. Unfortunately,

with this manual approach, we will need an exhaustive list of all the behaviors we'd like to consider. With IRL we can extract this complex reward function from human driving data.

Figure 2.13: RL vs IRL.

2.3.2 Inverse reinforcement learning formulation

An IRL task can be defined ruffly as the following:

Given:

- 1) measurements of an agent's behaviour over time, in a variety of circumstances
- 2) if needed measurements of the sensory inputs to the agent[1]
- 3) if available, a model of the environment

Obtain: The reward function being optimized

It is important that every solution to the IRL problem within a finite state space must meet the following inequality[1]:

$$(T_{\pi^*} - T_a)(I - \gamma T_{\pi^*})^{-1} r^{\pi^*} \geq \gamma^{-1}(r^a - r^{\pi^*}) \,. \tag{2.5}$$

Where $\geq$ denotes vector inequality, T_a are the state transition probabilities when executing action a, π^* is the optimal policy that represents the agent's behaviour, and $ra = (r(s,a))_{|S|}$ is a state-reinforcement vector for the action a , i.e., the reinforcement for each state where the action was already chosen. This inequality characterises the set of all reinforcement functions, for which a given policy π^* is optimal.

Later in Ng and Russell [9] an alternate approach based on Inverse Reinforcement Learning was proposed. Here the proposed strategy consists on matching feature expectations (Equation 2.6) between an observed policy and a learner's behavior.

$$\sum_{Path\zeta_i} P(\zeta_i) f_{\zeta_i} = \check{f} \tag{2.6}$$

[1]these measurements are an extra, they serve to get a better understanding of the circumstances in which the agent finds himself

[1]This inequality is an extension of the inequality $(T_{\pi^*} - T_a)(I - \gamma T_{\pi^*})^{-1} r_{\pi^*} \geq 0$ presented by Ng and Russell [9]. The original inequality considers the restriction $r(s,a) = r(s,b) = r(s)\ \forall a, b \in A, \forall s \in S$.

Then was demonstrated that with this matching it was possible for an agent to achieve the same performance, as if the agent were in fact solving an MDP with a reward function linear in those features.

2.3.3 Inverse reinforcement learning analysis

The advantage of using this framework, is that the reward function has more value to an agent, since the reward function can be migrated from task to task. When applying the framework to video games, we can plug video game runs performed by players on to the first item, then we can give the representation of the game environment. By doing this, the agent will get the reward function behind the players actions, and their limitations that will be represented in the different runs that the algorithm will receive. This will result in a better understating of a player profile. Although, in this approach, it will still be necessary manual testing of the game, to retrieve the user maneuvers. However, in the end, the overall number of manual tests will be lesser than, the situation where the tests are all done manually. Considering that, once the agent obtains the rewards of the player, it can automatically test the game as a player.

Nevertheless, there is a problem when using this framework, when we try to extract a reward from an expert's observations. There will be many different reward functions that the expert may be trying to maximize.

2.3.4 Max Entropy IRL

Inspired by Ziebart et al. [10], this approach uses a probabilistic model that helps to find the reward function based on partial trajectories. As the name implies, this IRL approach is based on the Principle of Maximum Entropy. This principle is based on the premise that when estimating the probability distribution, you should select that distribution which leaves you with the largest remaining uncertainty (i.e., the maximum entropy) consistent with your constraints. This way makes it harder to introduce any additional assumptions or biases into the calculations.

The use of this principle to find a solution for the IRL problem, was proposed in [10, 11]. Here the principle of maximum entropy is employed in order to resolve the ambiguity, when choosing a distribution over decisions. We provide efficient algorithms for learning and inference, for deterministic MDPs. We rely on an additional simplifying assumption to make reasoning about non-deterministic MDPs tractable. The resulting distribution consists of a probabilistic model that normalizes globally over behaviors.

The main difference with this approach is, instead of using a optimal policy like in the previous approach, we are going to create a model based on a set of player behaviours (or runs), to obtain a reward function for the agent. In Abbeel and Ng [11] they applied the maximum entropy to two types of path distributions: Deterministic and Non-Deterministic. The one we are going to analyse is the deterministic path, since in most cases the video game scenario tends to be deterministic.

Similar to the distributions of policies, many different distributions of paths match feature counts, when any demonstrated behavior is sub-optimal. Any one distribution from among this set, may exhibit a preference for some of the paths, over others that are not implied by the path features. We employ the principle of maximum entropy, which resolves this ambiguity by choosing the distribution that does not

exhibit any additional preferences beyond matching feature expectations (Equation 2.6).

The resulting distribution over paths for deterministic MDPs, is parameterized by the reward weights Θ (Equation 2.7). Under this model, plans with equivalent rewards, have equal probabilities, and plans with higher rewards are exponentially more preferred.

$$P(\zeta_i|\Theta) = \frac{1}{Z(\Theta)} e^{\Theta^T f_{\zeta_i}} = \frac{1}{Z(\Theta)} e^{\sum_{s_j \in \zeta_i} \Theta^T f_{s_j}} \tag{2.7}$$

Given parameter weights, the partition function, Z(Θ), will always converge for finite problems and infinite problems that have discounted reward weights. In infinite problems that have zero-reward absorbing states, the partition function might not converge even if the rewards of all states are negative. Nonetheless, if we give expert trajectories that consist in a finite number of steps, then the reward weights maximizing entropy must be convergent.

Now considering that paths in these MDPs are determined by the action choices of the agent and the random outcomes of the MDP. The distribution over paths must take randomness into account.

So the maximum entropy distribution of paths is used, and is being conditioned on the transition distribution, T, and constrained to match feature expectations (Equation 2.6). Considering the space of action outcomes, T and an outcome sample, o, that specifies the next state for every action. The MDP is deterministic given o with the previous distribution (Equation 2.7) over paths compatible with o.

Then the indicator function, $I_{\Theta \in o}$ is 1 when Θ is compatible with o and 0 otherwise. Computing this distribution shown in Equation 2.8 is generally intractable. However, assuming that transition randomness has an effect on behavior and that the partition function is constant for all o $\in$ T , then we obtain a approximate distribution that can be traced over paths as presented in Equation 2.9.

$$P(\zeta_i|\Theta) = \frac{1}{Z(\Theta)} e^{\Theta^T f_{\zeta_i}} = \frac{1}{Z(\Theta)} e^{\sum_{s_j \in \zeta_i} \Theta^T f_{s_j}} \tag{2.8}$$

$$P(\zeta_i|\Theta) = \frac{1}{Z(\Theta)} e^{\Theta^T f_{\zeta_i}} = \frac{1}{Z(\Theta)} e^{\sum_{s_j \in \zeta_i} \Theta^T f_{s_j}} \tag{2.9}$$

This distribution over paths generates a stochastic policy (i.e., a distribution over the available actions of each state) if the partition function of Equation 2.9 converges. Then the probability of an action is weighted by the expected rewards of all paths that started with that specific action.

Now to learn from the demonstrated behavior, we have to maximize the entropy of the distribution over paths subject to the feature constraints collected from observed data.

$$\Theta^* = argmax L(\theta) = argmax \sum_{examples} log P(\zeta|\Theta, T) \tag{2.10}$$

This function is convex for deterministic MDPs and the optimal can be found by using a gradient-based optimization method. The gradient here consists in the difference between expected empirical feature counts and the learner's expected feature counts, these can be expressed in terms of expected state visitation frequencies, D_{s_i} as represented in Equation 2.11.

$$\nabla L(\Theta) = \tilde{f} - \sum_{\zeta} P(\zeta|\Theta, T) f_{\zeta} = \tilde{f} - \sum_{s_i} D_{s_i} f_{s_i} \tag{2.11}$$

If we have the expected state frequencies, then this gradient can easily be computed for optimization.

The most directly approach for computing expected state frequencies is to enumerate each possible path. Unfortunately, with exponential growth of paths in the MDP, makes enumeration-based approaches computationally infeasible.

Instead, the algorithm proposed computes the expected state occupancy frequencies efficiently based on a technique that is similar forward-backward algorithm used in Conditional Random Fields or value iteration that is used in Reinforcement Learning. The algorithm can be represented as:

Algorithm 2 MaxEnt algorithm

Backward pass

1: $Z_{s_i,0} \leftarrow 1$

2: Recursively compute for N iterations

$Z_{a_i,j} \leftarrow \sum_k \mathrm{P}\ (s_k|s_i, a_{i,j})\ e^{\mathrm{reward}(s_i|\Theta)}\ Z_{s_k}$

$Z_{s_i} \leftarrow \sum_{a_{i,j}} Z_{a_{i,j}}$

Local action probability computation

3: $P(a_{i,j}|s_i) \leftarrow \frac{Z_{a_{i,j}}}{Z_{s_i}}$

Forward pass

4: $D_{si,t} \leftarrow P(s_i = s_{initial})$

5: Recursively compute for t = 1 to N

$_{D}si,t+1 \leftarrow \sum_{a_{i,j}} \sum_k D_{s_k,t} P(a_{i,j}|s_i) P(s_k|a_{i,j}, s_i)$

Summing frequencies

6: $D_{si} \leftarrow \sum_t D_{s_{i,t}}$

However, there is a problem with this approach. Since it is dependent on several player runs, it's prone to be affected by noise and imperfect behaviours.

Chapter 3

Related work

In this section we will analyse some of the relevant work, done in the areas connected to the topic of this thesis. Those important areas are: automated playtesting, player profile representation in agents and IRL implementation in video game agents.

3.1 Automated Playtesting

As mentioned before, in order to decrease the time needed to do manual testing, many developers started using automated playtesting. This method aims for the reduce of the time taken to complete certain tasks. These tasks instead of being done by people, will be done by autonomous agents powered by artificial intelligence.

3.1.1 Path-search methods

There are multiple ways to build the agents. One of the aforementioned approaches, consists on the use of tree-search algorithms, such as the case of MCTS, and A*. With this type of algorithms, the agents have the capability of testing many different combination of actions, allowing for a strong evaluation of distinct outcomes. Silva et al. [12], designed agents to playtest certain parts of the game Sims Mobile [13]. For the design of the agents, they take different questions proposed by the developers and testers, and use them to build distinct agent goals. Also, they used the A* algorithm, in a way to choose the best action, in order for the agent to be closer of achieving said goal. An example of generating a goal based on a question can be seen as follows: They used a question for example *how does object acquisition, impact career progress?*, then they generate the following goal of *Achieve the max level of a career*.

To measure if the goal is accomplished, they used quantitative metrics like the career level, and career experience points. This process of $question \rightarrow goal \rightarrow metric$ was also used to check how imbalance is the relation between different game characters and the effects of objects in career progress. The results of the experience prove to be valuable, giving useful information to designers, helping them to make changes in order to improve the game.

One addition that can be done to path-search methods, is to add the use of heuristics Mugrai et al. [14]. In Holmgård et al. [15], the agents were developed by combining the MCTS algorithm with different designed heuristics in order to create different player personas. The agents were created for the game MiniDungeons 2 [16]. The main goal of the personas, is to create in the tree-search a bias to visit certain nodes, allowing the agent to follow a different playstyle, similar to human players. This is accomplished thanks to the use of different heuristics and metrics, for example *Steps taken*, *Proximity to Exit*, *Treasures Opened*, *Monsters Slain*, among others.

In the case of this paper, they achieve different playstyles for example: the *Runner* that aims to exit the dungeon as quickly as possible, for this persona, the more important metrics, are the Steps taken and the Proximity to Exit. Other persona, is the *Treasure Collector* that aims to collect all treasures in the game. For this persona, the more important metric is the Treasures Opened. After the experiments with the distinct personas, the authors stated that this concept is useful for the test of specific core priorities for games, that have a similar scope and size of MiniDungeons 2.

3.1.2 Reinforcement Learning

Despite the fact that tree-search algorithms and manually made heuristics, are considered more financially manageable, the evolution of AI, invited developers started to change to other approaches, that are faster at runtime and are capable of obtaining decent results in most of the playtesting scenarios. One of those is Reinforcement Learning (RL). Sriram [17], introduces a RL approach to platform games. The agents created were using Deep RL, since it was an approach, with proved results as the case of Mnih et al. [18] where Deep RL is used to play atari games. The problem of this approach is that the algorithms are computation heavy in the training step, and they are not very simple to design, since Deep RL requires the use of neural nets.

One simpler approach was proposed by Holmgard et al. [19]. Here they introduce a RL approach through the use of the Q-Learning algorithm. With this algorithm, they created generative agents as model capable of making decisions similar to human players, in a two-dimension rogue-like dungeon game. With this simple algorithm, helped by the the addition of manually designed heuristics, it was possible to change reward function of the agents. This made possible for the agents to have different priorities, resulting in different player personas.

3.1.3 Inverse Reinforcement Learning

The main problem with this previous approach is: RL takes a long time to develop, and to train the agents. Also most of the time is used by the developers to manually define a reward function. To solve the last, Russell [8], proposed the Inverse Reinforcement Learning (IRL). With this, the agent can learn the reward function from human behaviour. Complementing RL with the IRL technique, the agent can have a more human-like behaviour.

3.2 Player profiles in agents

More importantly, that being able to play a game, when testing, is essential for the agent, be able to simulate different player behaviours, to test all features with different goals. Furthermore, it's important when testing the game, to simulate a player that has some type of limitation.

There are multiple ways to allow the agents to simulate a behaviour similar to a human player.

3.2.1 Heuristics

One of possible approaches is by manually design heuristics that change the degree of importance of game variables and states. As mentioned in the previous section, Holmgård et al. [15], achieved the creation of personas, through the use of manually defined heuristics, and a different set of metrics. These metrics make the agents, prefer certain nodes of the tree search algorithm.

Following the RL option, the heuristic approach can still be used. Holmgard et al. [19] applied similar heuristics to the Q-learning algorithm. These heuristics change the importance of certain variables, with the goal of modifying the reward function of the agent.

3.2.2 Inverse Reinforcement Learning

As mentioned previously, instead of manually design the heuristics, it is possible to make the agent perform apprenticeship learning via IRL, where we induced the agent to extract the reward\goal, from the expert behaviour. In [8, 11], it's showed how with IRL, an agent can obtain the reward function for a given environment. With the obtained reward, the agent collects a underlying knowledge of the activity\game, that is under analysis. Later, the reward can then be used in reinforcement learning. With this, the agent will learn based on a person underlying goal, therefore the agent can learn the goals of distinct types of players, accomplishing an human-like behaviour in the agent.

In Tastan and Sukthankar [20], using IRL with a linear solver, they create an agent with skills to play the competitive game Unreal Tournament[21]. They designed a bot that would play the game with similar behaviours to a human player, possessing three different modes of interaction with the game such as: exploration where the main objective is to gather resources, when the bot encounters an enemy it switches to the attack mode. Then in attack mode the agent swaps to targeting mode, to fire\attack the sighted opponent.

In addition, to learn from human experience with IRL, it is also possible to make the agent capable of predicting player actions. In Rastogi et al. [22], it's proposed with using the max entropy IRL model, an agent that is capable of predicting the actions of a certain player.

Even more important than learning a player behaviour, is to be able to learn their underlying motivation. In Wang et al. [23], the authors proposed a model that is going to be added to the IRL framework. They proposed the Multi-Motivation Behavior Modeling (MMBM). This model aims to retrieve the reward function of players, but taking into consideration the motivation theory in Yee [24]. In the MMBM model, the goal of the agents is, instead of just maximizing one reward value based on only one single motiva-

tion, for instance, achieving high scores, the model maximizes a combination of multiple rewards values, based on multifaceted motivations.

With this model, the aim of the authors was to extent the IRL framework, with the aim of uncovering the multi-dimensional reward mechanism. In the first stage, their model first quantifies each dimension of the reward signal individually, with the help of the motivation theory. Subsequently, these individual signals are then combined with the assumption that each player that appeared in the trajectory, is acting with the main goal of optimizing its objective.

A great advantage of the approach adopted here, was the fact that the MMBM model doesn't require a simulation of the tested environment. Providing to big and complex games, a feasible way of extracting player motivations, as were the case for the authors, they used World of Warcraft Avatar History [1] dataset from World of Warcraft game [25], and the model was showed to be able to collect distinct reward functions, and different value structures among various player groups.

3.2.3 Discussion

The studies mentioned previously, presented some of the work done in the field area of this thesis. Also some of those will be implemented in a similar manner here.

In the field of automatic playtesting complemented with player profiles, there are many different studies focused on several approaches, namely, heuristic-based agents and IRL. With these, it is possible to create an agent that is going to perform in a similar way to a human player. One advantage of adding a player profile to the automated test agents is the increasing of test coverage, since the agent might execute actions that are not considered in the optimal solution proposed by a learning algorithm.

[1] gameplay data from 70,000 users, spanning over a three-year period

Chapter 4

Implementation

In this section it is explained the approach taken to resolve the defined problem. This section is dived into 5 subsections. First, we will provide specifications for the software used in this thesis. Then in the other subsections, the other parts of the solution are going to be analysed. Namely, the testing environment, the different clustering algorithms, the agent proprieties, in addition to their Profiles.

4.1 System Architecture

In this section we present a representation for our approach.

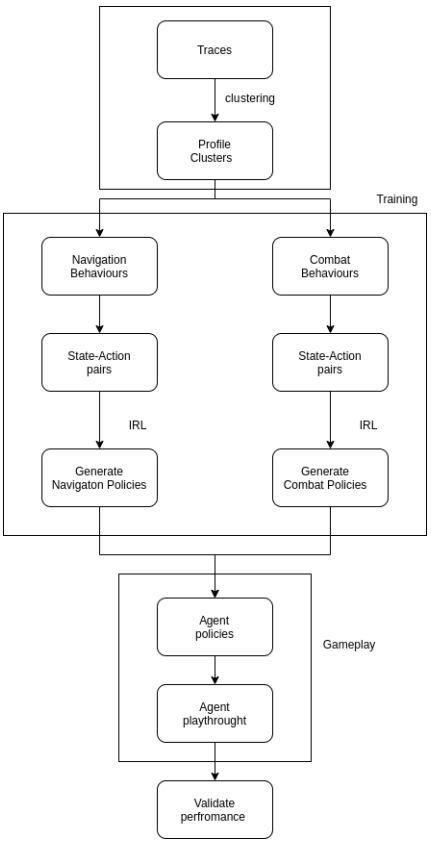


Figure 4.1: Tool architecture.

From figure 4.1 we see that our approach can be divided into 3 parts:

- clustering: where the traces are group according to their profiles.
- training: where the irl is used to generate agent policies.
- gameplay: where we test the different polices.

4.2 System Requirements

The most important software used in this thesis is the programming language Python, since, all of important computations, are simplified using Python modules. We give special attention to the Numpy module that facilitates matrix operations and computations. The clustering algorithms used in this thesis also come from a python module, in this case sklearn. We used this module due to the variety of different types of clustering algorithms provided by the module, and also, it has methods to perform clustering analysis. The game environment was also built in Python, through the use of the Pygame engine. In addition, to present some of the results in a more visual way, Matplotlib module was used.

4.3 Game Environment

As mentioned in the acknowledgements section, this thesis is based on a game developed by Pedro Fernandes. In Figure4.2 we can see a screenshot of the game environment.

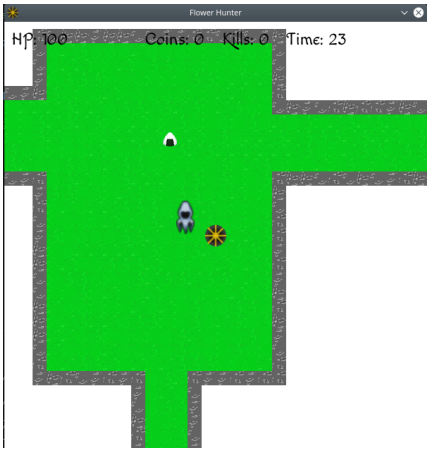


Figure 4.2: Infinite Game.

The game is named infinite game. It consists on a 2d top down inspired by the classic Legend of Zelda games [26]. In this game the player can explore a level with the main objective of reaching the flower at the end. When navigating through the level, the player will encounter enemies. Then the player can fight these enemies or escape from them. Finally, in the game, the player can collect optional coins.

The game is divided into 3 levels, which have same layout as we show in figure 4.3. In addition to the layout of the levels, the player's starting position, and the objective's position, are the same. The main differences between the levels are related to the number of objects that the player can interact with. In

this case, the number of enemies, the number of coins, and finally the health pickups. Furthermore, this also has an affect on the difficulty of each level. Making the level 1 the easiest one, and the level 3 the hardest. In table 4.1 with present the different between all levels, regarding their constituent elements.

Table 4.1: Game level elements.

Level	enemies	coins	rice
1	1	2	1
2	10	25	10
3	32	9	2

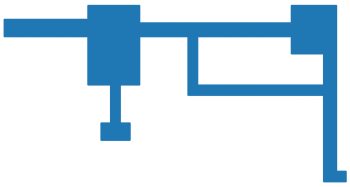


Figure 4.3: Infinite Game level layout.

There were several reasons for choosing this game as an analysis environment. The first reason is related with the simplicity of the game. Since, in terms of mechanics the game isn't very complex, and it's world has only 2 dimensions. This makes it easier for us to create a symbolic representation of it, mainly available actions, and possible agent states.

Secondly, there is already some available data. Since this game is part of the iv4XR project, and is being used for other project components, we already have access to player data, that was collected in previous experiments. The data by having different people playing the game, who play the levels in a random order. The said data is composed of the following:

- For each level, we have 90 player traces, making a total of 270 traces.
- Each player trace is dived into 3 parts:

  position all positions the player has gone through.

  actions all actions the player performed.

  preceptor traces of relevant game data.

As for the relevant data, this is all the data the preceptor captures:

- distance to closest enemy
- distance to closest food_item
- distance to closest money
- number of enemies in view

- number of food_item in view
- number of money in view
- sum of enemy values
- sum of food_item values
- sum of money values
- sum value slash distance enemies
- sum value slash distance food_item
- sum value slash distance money
- seconds since seeing enemy
- seconds since seeing food_item
- seconds since seeing money
- distance to objective
- current life
- current money
- current kills
- total damage done

In order to train the agents, we are going to use all available traces. However, is very likely that within the traces are players that are playing the game in different ways. With the recovery of these said profiles in mind, instead of using the data directly, we are going to perform clustering, with the goal of producing smaller datasets.

4.4 Clustering

As mentioned, all algorithms used here, came from the sklearn module. These consist in the **KMeans** clustering, **GaussianMixture** EM clustering and hierarchical **AgglomerativeClustering**. When performing the clustering as we mentioned in section 2.2 one common problem on the 3 algorithms, is that we first need to define a number of clusters k. Basically, k needs to be an input value, and it is hard to define a value for k. Therefore, to overcome this problem, we decided to run the 3 algorithms several times, with different k values. Then, compare the different results according to specific metrics in particular MSE and Silhouette Score. So, first we define an array with the various values for k. $array = [2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]$ We decided on these values since we want an odd

number of clusters, also we don't need very large values of k, since as we only have 270 records, with values greater than 30 we will have clusters with few elements

Now we define the relevant features that are going to be used for clustering. For this we decided to use 4 that we defined as the most relevant when establishing a player profile. Those are:

- Distance to objective
- Remaining life
- Number of coins collected
- Number of enemies defeated

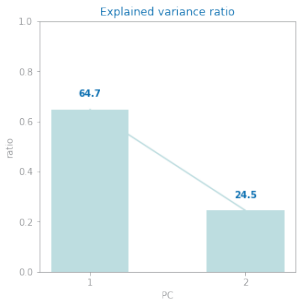
We considered these 4 to be most important, because with the *Distance to objective*, we can check if a player reach the end. By considering the *Remaining life*, we can check if a player evades the enemies, and if he doesn't how he deals with the enemies. With the analysis of the *Number of coins collected*, we can infer the percentage of map that was explore by the player. With the *Number of enemies defeated*, we can analyse how aggressive a player is.

As an alternative solution, since some profiles might have similar behaviour between 3 levels, we decided to normalize 2 of the features. In this case, the number of coins and the number of enemies, since these vary from level to level. For example, in level 1 there is only 1 enemy, but in level 2 there are 10. With that in mind, if a player kills all enemies on both levels, the "scores" will be very different (1 and 10 respectively), although the behavior is very similar. So, with the normalization, both scores will equal to 1.

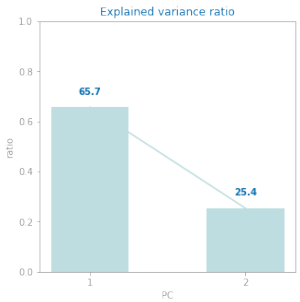
The points for which the cluster algorithms were used, are represented in figure 4.5 with and without normalization.

Since the data has 4 dimensions, we decided to test if it was possible to reduce the dimension space, mainly to simplify the readability of the different clustering solutions. To do this we decided to test Principal Component Analysis (PCA), to see if with only 2 variables it is possible to represent most of the data.

As shown in figure 4.4 with 2 features, PCA covers 89% of the not normalized data, and 91% of the normalized data.



(a) No normalization



(b) Normalized

Figure 4.4: PCA variance.

(a) No normalization

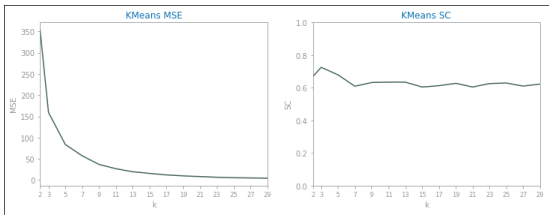
(b) Normalized

Figure 4.5: Level traces points after PCA.

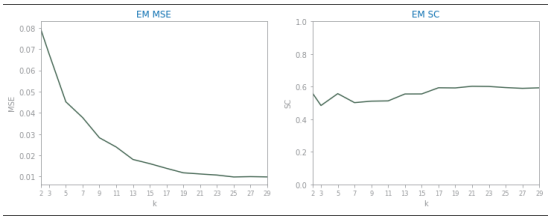
Now we select the metrics that we are going to use with purpose of comparing the different algorithms. We decided on two metrics, the Mean Square Error (MSE) and the Silhouette Score. Both metrics are already implemented in the sklearn module.

MSE focus on the distance of points to the centroid of clusters. It's the same principle used for optimization in the K-means algorithm. In this metric, we first compute the centroid of each cluster. Then we sum all of the Euclidean distances between all cluster points to its centroid. So, the lower the MSE, the better the solution. The Silhouette Score or Silhouette Coefficient, is related to cluster cohesion. Here the values are fixed between -1 and 1. Where 1 represents clusters, which are well apart from each other, and clearly distinguished. Oppositely, the value -1 means that the clusters are assigned in a wrong way. Thus we will prefer a Silhouette Score close to 1.

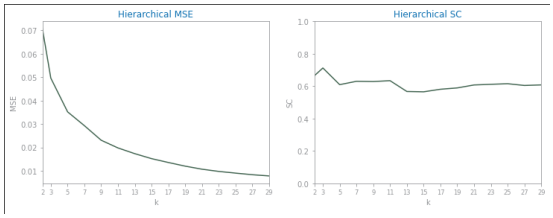
In figure 4.6 we present the metric results for the algorithms, according to different values of k for the not normalized data.



(a) K means scores



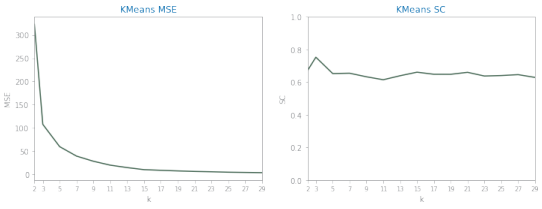
(b) EM scores



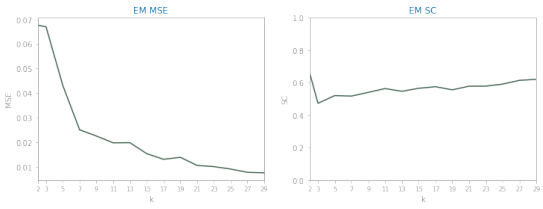
(c) Hierarchical scores

Figure 4.6: Clustering scores. MSE on the left, Silhouette Score(SC) on the right

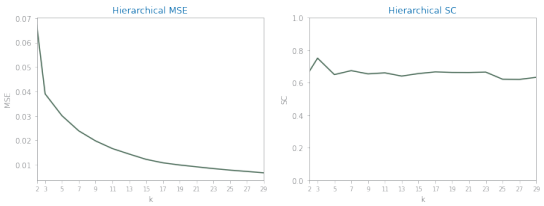
Now for the metric results with the normalized data, whose are represented in figure 4.6 for the different algorithms, according to different values of k.



(a) K means scores



(b) EM scores



(c) Hierarchical scores

Figure 4.7: Clustering scores. MSE on the left, Silhouette Score(SC) on the right

From the figure 4.6 we can draw some conclusions to choose the best cluster solution. Through the Silhouette Score, we can affirm that the best solutions tend to use a small value for k. In the case of k-means and hierarchical, that value is 3, whereas in EM the value is 5. As for the MSE the obtained results are similar. From the values we can see that the 3 solutions have what is defined as the elbow, close to these same values (3 to 7).

With these values in mind, the clustering solution that we choose was the one with 5 clusters. In figure 4.8 we illustrate the chosen solution. This one was EM, since it presented a better cohesion of the 5 clusters, although it had some outliers.

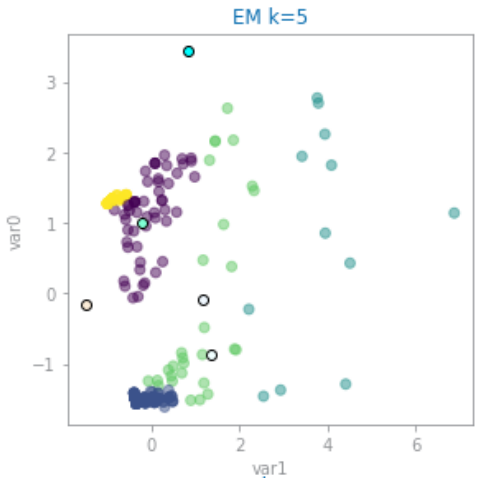


Figure 4.8: EM solution.

However, the fact that these cluster solutions tend to prefer low values for k, may reveal that they may be separating the data by levels. We can see signs of this in both k-means and hierarchical solutions. Where one of the clusters encompasses all points of level 1. A similar effect also happens with level 3. This happens as expected, since the 3 level have very different numbers of elements. In this situation even thought a player might have the same behaviour in all levels, his score will be different. For example, in level 1 there is only 1 enemy, but in level 3 there are 30. With that in mind, if a player has an aggressive behaviour where he kills all enemies on both levels, his score will 2 values (1 and 30 respectively) that are far apart.

For the normalized data, we decided to also use the EM solution. From the scores presented in figure 4.7 the clustering solution that we choose was the one with 11 clusters. In figure 4.9 we illustrate the chosen solution.

4.5 Agent Structure

For the creation of agents, we decided to follow an approach similar to Tastan and Sukthankar [20]. We decided to divide the agent's behavior into two main components, so that the agent has two types of behaviours when playing trough the level. These two modes are: the **navigation mode** and the **combat mode**. This division was created for 2 main reasons:

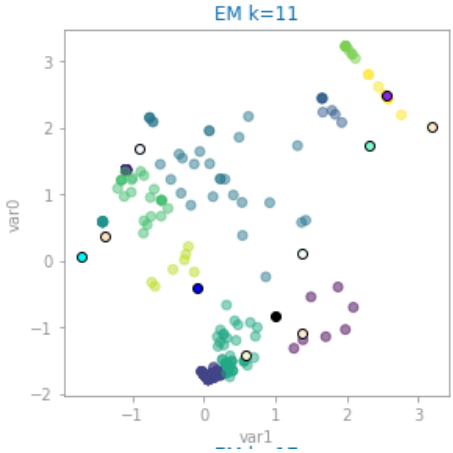


Figure 4.9: EM solution.

- Since the levels have enemies whose position is not fixed, it would make somewhat complex, represent agent states if we consider the various enemy positions.
- This division simplifies the combination of different behaviors. For example it will be possible to have an aggressive agent, who wants to explore the level or go straight to the end.

The transition between these two modes is mainly related to the distance between the agent and the closest enemy. In essence, the agent will switch to the combat mode when it gets close to an enemy.

4.5.1 Navigation mode

This mode is related to how the agent navigates through the different levels. Here, the objective was to represent a level using a GridWorld approach. In this approach, the state space of the MDP will match all valid positions of a level.

To define MDP state space, we decide to take the level structure, defined as a csv file. In figure 4.10 we show a part of the csv map file. Then we divided the map into different areas as later shown in figure 4.11 (a). This division was done to reduce the size of the state space.

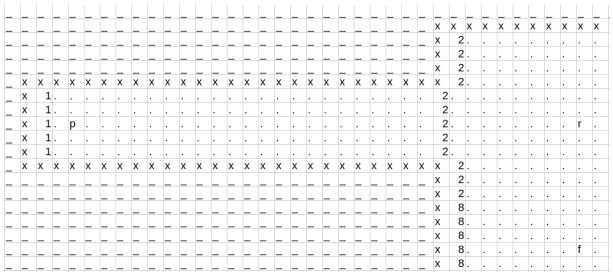


Figure 4.10: CSV map file.

To extract the states from the file, the method used was as follows. First, we store the positions of all important cells of the file. These are represented as ".", "P", numbers, among others. These cells, when they are registered, are stored in lists according to the area where they belong. Secondly, we obtain the cell position that corresponds to the player's starting position, identified in the file by P. Then

for each area we normalize the values of the positions that belong to its list, by subtracting the position components to those of P. For example if P is (10, 10) and A is (14, 10), then after normalization P is (0,0) and A is (4,0). Finally, we create a Python dictionary where each key corresponds to an area id and the value points to a list of all positions of the said area.

It is important to mention that the positions that are extracted from the csv do not correspond to a specific position on the game map. Instead they encompass several positions on the map.

The final state space has a total value of 1285 states with the following area division:

Table 4.2: States per area.

Area	NºStates
area 1	125 states
area 2	197 states
area 3	126 states
area 4	202 states
area 5	255 states
area 6	105 states
area 7	147 states
area 8	128 states

For the action space, we consider 5 simple actions:

- The four directions for movement UP, DOWN, LEFT and RIGHT.
- An ”empty” action STAY since when analyzing the player’s data, we saw that the player is sometimes in the same position.

As for the transition matrix between states, we define that all of the actions as deterministic. Meaning an action performed in a specific state can only transit to only one state. Because in a video game scenario, actions with an associated error don’t usually exist. As an error we can consider selecting the action RIGHT and instead of going right we go left.

Although, this solution has a problem since the MDP created depends on a level layout, it is not possible to use what is learned on one level on another one that has a different layout. This doesn’t happen here, since the 3 levels that we are going to be analysing have the same layout as shown in figure 4.11 (a).

4.5.2 Combat mode

For the structure of combat MDP, we decided on a more abstract solution. Since during combat, the most important features, are the location of the enemy and the current player’s life. We define the state space as a combination of these 2 elements, giving a total of 12 states as shown in table 4.3.

For the available actions we decided on 5 possible actions:

- moving towards the enemy

Table 4.3: Combat mode states.

dist to enemy \ Life %	0-29	30-59	60-89	90-100
≤ 40	1	2	3	4
≥ 40 and ≤ 80	5	6	7	8
≤ 100	9	10	11	12

- moving away from the enemy
- move towards and attack the enemy
- attack the enemy
- stay

With these actions, it is possible to replicate several behaviors that are related to combat. Such as, we can have one that only escapes from enemy, or one that goes straight to the attack.

4.6 IRL

For the implementation of the IRL algorithm, we followed the Max Entropy as it was presented in Ziebart et al. [10]. Inspired also by Alger [27] that had already an implementation of this same algorithm defined here as MaxEnt. To his code [1] , we made some modifications to deal with the concept of level areas. These modifications consist of changing the transition matrix from (State, Action, State) to (Action, State, State), we also add a limit for the maximum reward value and the capability of generating trajectories from a profile folder to a specific area. To execute the MaxEnt IRL the most important function to consider is the following:

```
def irl(feature_matrix, n_actions, discount, transition_probability,
trajectories, epochs, learning_rate, area_mdp_id, profile_id):
```

Where the attributes correspond to the following

- **feature_matrix**: matrix for state features
- **n_actions**: number of available actions
- **discount**: discount factor of the MDP
- **transition_probability**: a (Action, State, State) matrix
- **trajectories**: a set of trajectories based on observed behaviour
- **epochs**:the number of epochs used to train
- **learning_rate**:the learning rate of algorithm between 0 and 1

[1] available in https://github.com/MatthewJA/Inverse-Reinforcement-Learning

- **area_mdp_id**: id of the current area under analysis

- **profile_id**: id of the current profile under analysis

For this project we decided on using an Identity matrix for the feature matrix, for the training epochs, we decided on a value of 60, we chose this value because we saw that in examples a value of 50 was used, so since we have small areas we decided to increase this value. In relation to trajectories, we define a trajectory as being a list of following: (state, action, reward), here the state is the cell position, the action is one of the five defined in section 4.5.2, the reward is the value extracted from a basic reward function where most of the states have a value equal to 0, the cells where are coins have a value of 0.5 and the final objective has a value of 1.

When we are executing the IRL, we can see the advantages of separating the original map into several areas. With this division, we can provide small components for IRL training, for example: thanks to the division the MDPs are smaller, meaning smaller feature and transition matrices. This also had an impact on the size of the trajectories, which also became smaller. Furthermore, it is possible to split the IRL into several processes, since the result of the IRL of an area is independent of the others, and does not cause read and write conflicts.

(a) Map layout

(b) Learning areas.

Figure 4.11: Division of the game environment into learning areas.

One other element that has a big impact on the results, and on the time taken by the MaxEnt algorithm, is the size of the player trajectories. After some analysis, we can see that in the trajectories there are many (state, action, reward) pairs that are duplicated. This results on bigger trajectories and affects the reward value for the duplicated states. To solve this, we decided on crop the number of duplicates. We allow the existence of them, but we create a limit for the maximum number of duplicates that can exist, for this project, we decided on a maximum of 4 duplicates.

After running the MaxEnt algorithm, we get a reward function for an area and a certain profile. To obtain the final policy, it is necessary to run one more algorithm to convert the reward into a policy. In this case, the algorithm is value iteration. This consists in a simple algorithm that iterates over a given reward function and generates a policy that maximizes said reward. Although it is computationally heavy, this algorithm also benefits from the area division since the mdps will be smaller. Additionally, this algorithm is guaranteed to converge to the optimal values. With the value iteration we obtain a policy based on player behaviour for each area.

A problem with this approach is the obtained policies are deeply connected to the level structure, making it impossible to reuse learned polices in different levels or level areas.

Chapter 5

Validation

In this chapter, we will explain the validation method that we are going to use to validate our approach.

5.1 Approach

To test the approach analysed in this thesis we defined a process that is divided into 3 parts:

1. IRL results
2. Profiles vs Traces
3. New level

5.2 IRL results

In the first part, we are going to compare the learned policies to the player behaviour. This process will be based on 2 metrics: similarity and probability. In the case of similarity, for each state, we are going to analyse the average of player actions. Then compare it to the action in the agent policy. As present in the following algorithm

Algorithm 3 Similarity algorithm

```
1: procedure SIMILARITY(p, a)                ▷ The similarity between players p and agent a
2:     count ← 0
3:     for s in S do                         ▷ S corresponds to all states visited by the players
4:         if agent action in s = player action in state s then
5:             count ← count + 1
6:         end if
7:     end for
8:     similarity ← count / size_of(S)
9:     return similarity
10: end procedure
```

For the probability, the process is similar to the similarity, but, instead of counting the actions that are equal, we will calculate the probability of the agent performing the action sequence described by the

average player behaviour.

Algorithm 4 Probability algorithm

```
1: procedure PROBABILITY(p, a)          ▷ The probability of agent a perform like players p
2:     probability ← 1
3:     for s in S do                    ▷ S corresponds to all states visited by the players
4:         a ← player action in state s
5:         agent_probability ← probability of agent execute action a in state s
6:         probability ← probability * agent_probability
7:     end for
8:     return probability
9: end procedure
```

To validate these results, we consider similarity superior to 50% to be positive. For the probability we expect very low values, having said that we consider an value 1e-35 to be positive.

5.3 Profiles vs Traces

On the second part, we are going to compare the performance of the agent and compare it to the player behaviour. To do this, we are going to make the agent play the average level of the traces used to train it. For example, if profile 0 has the most traces of level 1 playthroughs, we should test it on level 1. Then we are going to measure its performance based on the following features: remaining life or HP, the number of coins collected, number of kills or enemies defeated, the time taken to complete the level, and the percentage of map exploration. In case the agent gets stuck in a part of the map, we will set a limit on the time taken.

5.4 New level

Finally, we are going to take the different profiles and test them on a new made level. This new level will have a similar layout to the previous ones. Here, we only change the number and position of enemies and coins. Then we are going to evaluate the agents using the previous performance features.

Chapter 6

Results

In this section we will analyze the results obtained for the different approaches taken. But first we will describe the baseline problem

6.1 Problem Description

As mentioned in the previous sections, the objective of this work is to create agents that are capable of having different profiles, making it possible for the agents to be able to play the game in different ways. It is also important to test the profile learned at the level used in the training, in order to compare it's performance with that of real players. Also it is important to test it in a level different from the one used in training.

For this thesis, we are going to present two solutions, where the main differences are related to the player data, that was provided to the clustering algorithms. With this, we are going to change the clusters that are provided to agent to preform the IRL. In the first solution, the player data is not normalized and we have a total of five possible profiles. In the second, the player data was normalized and it gave a bigger number of possible profiles namely 11 profiles.

We are going to analyse the learning process of the profiles, by comparing the obtained policies with the actions taken by players. Then, we are going to analyse the performance of profiles in the level used in the IRL process. Finally, we are going to test them in a new level.

First, we need to consider the level layout, as mentioned before, the level was divided into 8 areas. In figure 6.1 we present the final division of the level into areas and the labeling on them.

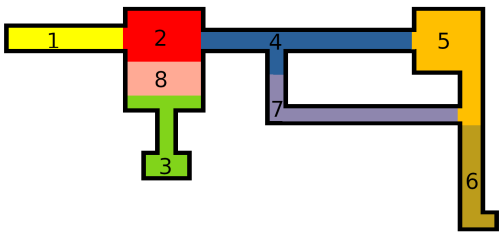


Figure 6.1: Level map areas

This is important when analysing the first obtain results.

6.2 Test with non normalized data

For solution 1, we decided to use the not normalized data. This approach gave 5 possible profiles for the agents. From the table 6.1, we can see that we have some interesting results in relation to navigation, namely: We can see that similarity between the agent and the player is in most cases below 50%, meaning that the algorithm shows problems in capturing how the player moves. And we can see that this happens more in bigger areas.

Also, we can see that the most difficult areas to learn are areas 3 and 8. We can also see that it is impossible for profile 2 to reach the end of a level, since the level objective is located in area 6. And the agent doesn't have results for this area, meaning we have an agent that can not complete any level. This happens because the traces used for this profile were all from the level 3, and as we can see in table 6.2, all players have not reached the end of the level. This means that the agent captured the behavior well.

From the table 6.2, as expected, we can see that only profile 4 didn't reach the level objective since its remaining HP is equal to 0. We can also see that most of the profiles captured here did not explore much of the level, most of them move directly to the objective since their map coverage is inferior to 40%. Although the performance of the agents was very similar to that of the player behaviour. We can also see that the agents tend to be more aggressive that the player, since in all cases the agents defeated more enemies than the players average.

From the table 6.3, when playing the new level, we see that the results are not as good. We can see that only 1 of the agents was able to finish the level, 1 of them died, and the other 3 became stuck. It is interesting that 1 of the 3 that got stuck is the profile that does not have knowledge of the area where the objective is, as was expected. One other that also got stuck, was the one that got lost in the previous experiment. In terms of percentage of the map explored, the values obtained here are very similar to the previous levels.

6.3 Test with normalized data

For solution 2, we decided to use the normalized data. Since, we see that with the not normalized data it is harder to compare the behaviours between levels, even though all levels have the same layout, they have different elements.

This approach gave 11 possible profiles for the agents. Since this was the second approach, we expect that some of the 10 profiles consist of previous profiles that were split. From Table 6.4, we can see this division effect, where in the previous solution we only have 1 profile that didn't complete the level, now we have 3 of them. From this table we also obtain positive results regarding navigation. We can see that the most difficult areas to learn, are still areas 3 and 8 since area 8 is the most opened one and area 3 is the most prone to stuck agents.

Table 6.1: Solution 1 Learning Results Table.

Profiles	Area 1		Area 2		Area 3		Area 4	
	Simp	Prob	Simp	Prob	Simp	Prob	Simp	Prob
Profile 0	40.3	7.18e-39	49.32	9.78e-105	38.79	4.71e-82	58.92	2.46e-90
Profile 1	62.5	3.88e-25	33.78	1.74e-127	31.03	1.17e-39	44.63	5.4e-78
Profile 2	48.9	1.77e-53	42.05	1.34e-154	41.03	5.38e-57	48.47	1.13e-113
Profile 3	45.83	2.39e-28	39.76	4.29e-98	35.14	7.49e-79	51.85	2.94e-104
Profile 4	58.93	4.53e-56	43.11	7.56e-104	37.84	6.97e-106	50.0	8.41e-214
Profiles	Area 5		Area 6		Area 7		Area 8	
	Simp	Prob	Simp	Prob	Simp	Prob	Simp	Prob
Profile 0	43.35	1.04e-109	59.59	2.59e-52	45.21	3.03e-85	29.41	3.13e-83
Profile 1	45.11	8.93e-117	65.31	3.79e-51	49.66	3.37e-80	30.91	8.75e-79
Profile 2	39.7	2.03e-48	—	—	79.63	7.86e-23	36.72	1.21e-98
Profile 3	57.41	1.53e-123	56.7	3.12e-49	53.74	1.67e-81	26.4	3.63e-94
Profile 4	36.97	1.28e-194	64.06	4.9e-30	51.03	2e-95	46.4	3.69e-81

Table 6.2: Solution 1 Base Performance Results Table.

agents	level	time	coins collect	enemies killed	remaining HP	map coverage
Profile 0	2	45	0	5	45	31.15%
Trace 0	2	42	5	1	20	20.99%
Profile 1	1	28	1	0	100	22.04%
Trace 1	1	30	1	0	100	21.18%
Profile 2	3	11	0	4	0	16.67%
Trace 2	3	24	0	1	0	15.34%
Profile 3	2	78	0	3	25	32.39%
Trace 3	2	53	8	2	90	25.61%
Profile 4	2	200	15	6	30	31.3%
Trace 4	2	97	18	5	30	32.46%

Table 6.3: Solution 1 Extra Performance Results Table.

agents	time	coins collect	enemies killed	remaining HP	map coverage
Profile 0	200	4	5	40	31.30%
Profile 1	26	8	4	70	24.45%
Profile 2	200	8	5	70	16.98%
Profile 3	25	3	3	0	23.29%
Profile 4	200	12	5	100	35.5%

From Table 6.5, we see that the performance of the agents shows similar results to that of the player average. As expected we see that 3 profiles didn't reach the level objective, since its remaining HP is equal to 0. An interesting observation taken from this results is that even though the agents map coverage is inferior to 40%, meaning that they did not explore much of the level, all of the agent values are slightly higher than players. Although, most values related to the collection of coins are smaller. Similar to the previous solution, we see that the agents tend to be more aggressive that the player, since

in all cases the agents defeated more enemies than players average.

From Table 6.6, when playing the new level, the results here are not as bad as the previous solution. We see here that 3 agents got stuck, 1 of them got stuck in the previous experiment, the other 2 are agents that don't possess knowledge of the area where the objective is, meaning they are incapable to finish the game. In addition to this, we also have different agents losing. But, these were trained based on the level 1 which only has 1 enemy, which can have an impact on the agent combat performance.

Table 6.4: Solution 2 Learning Results Table.

Profiles	Area 1		Area 2		Area 3		Area 4	
	Simp	Prob	Simp	Prob	Simp	Prob	Simp	Prob
Profile 0	52.87	1.44e-45	58.97	2.85e-56	38.93	1.81e-81	84.05	1.03e-57
Profile 1	58.70	1.29e-18	53.06	5.76e-45	41.67	2.29e-14	60.34	1.10e-54
Profile 2	54.12	3.14e-39	41.15	1.09e-140	44.87	8.08e-51	51.33	1.78e-97
Profile 3	80.77	1.34e-10	51.06	1.82e-79	44.44	8.56e-24	56.50	1.36e-95
Profile 4	51.02	6.23e-26	53.33	3.29e-69	34.88	2.66e-58	64.86	2.03e-96
Profile 5	61.19	1.58e-36	58.29	3.98e-96	26.19	9.31e-29	35.75	2.01e-117
Profile 6	74.00	2.53e-23	50.85	8.64e-72	32.79	6.578e-43	64.18	2.40e-31
Profile 7	96.00	4.37e-6	60.76	1.21e-40	41.38	3.81e-37	63.24	1.77e-69
Profile 8	91.67	1.87e-8	55.14	1.15e-53	38.46	5.48e-16	50.00	2.45e-124
Profile 9	53.33	3.62e-17	59.85	3.26e-70	31.30	5.19e-79	77.22	7.93e-80
Profile 10	68.57	1.54e-12	48.84	1.61e-52	37.88	3.27e-78	52.29	7.44e-68
Profiles	Area 5		Area 6		Area 7		Area 8	
	Simp	Prob	Simp	Prob	Simp	Prob	Simp	Prob
Profile 0	48.84	3.99e-110	79.17	8.53e-12	51.35	1.69e-55	53.15	2.26e-67
Profile 1	61.76	1.63e-83	82.65	4.45e-35	35.37	1.64e-101	50.00	1.30e-12
Profile 2	100.00	1.57e-4	—	—	79.63	8.41e-24	38.10	1.84e-94
Profile 3	73.56	3.33e-86	77.77	1.25e-37	78.91	4.79e-66	52.83	1.21e-66
Profile 4	36.51	1.94e-125	87.67	5.36e-24	72.80	6.94e-65	38.95	2.70e-63
Profile 5	40.48	9.56e-30	—	—	—	—	37.50	1.65e-76
Profile 6	79.69	1.01e-28	—	—	87.50	8.60e-48	54.17	8.24e-60
Profile 7	36.31	4.21e-107	82.02	1.94e-35	60.28	5.80e-65	43.75	1.27e-43
Profile 8	51.33	3.62e-66	76.77	2.49e-39	74.83	4.93e-68	45.71	2.55e-45
Profile 9	42.93	1.45e-17	65.91	3.53e-41	83.56	1.85e-57	37.40	1.64e-84
Profile 10	50.00	3.75e-19	69.23	2.65e-25	77.03	3.55e-30	52.56	1.95e-55

Table 6.5: Solution 2 Base Performance Results Table.

agents	level	time	coins collect	enemies killed	remaining HP	map coverage
Profile 0	2	200	10	1	90	22.04%
Trace 0	2	38	10	0	0	21.17%
Profile 1	1	28	1	0	100	21.57%
Trace 1	1	34	1	0	100	22.72%
Profile 2	3	13	0	5	0	17.92%
Trace 2	3	22	0	2	0	15.96%
Profile 3	2	45	0	5	65	31.30%
Trace 3	2	43	8	1	85	24.87%
Profile 4	1	27	0	0	100	23.83%
Trace 4	1	64	2	1	80	28.53%
Profile 5	3	26	0	8	0	22.51%
Trace 5	3	26	0	2	0	17.62%
Profile 6	2	89	0	7	0	34.34%
Trace 6	2	59	6	6	0	23.26%
Profile 7	1	64	0	0	100	35.11%
Trace 7	1	56	1	1	95	25.76%
Profile 8	1	28	0	0	100	23.75%
Trace 8	1	37	0	0	100	22.27%
Profile 9	2	46	0	5	35	31.07%
Trace 9	2	80	11	4	85	28.71%
Profile 10	2	62	0	4	30	32.23%
Trace 10	2	85	10	4	10	33.56%

Table 6.6: Solution 2 Extra Performance Results Table.

agents	time	coins collect	enemies killed	remaining HP	map coverage
Profile 0	200	10	5	10	28.42%
Profile 1	27	12	4	100	24.76%
Profile 2	121	11	5	0	26.55%
Profile 3	67	16	5	65	32.47%
Profile 4	38	10	4	0	27.18%
Profile 5	200	8	4	100	28.97%
Profile 6	200	18	6	70	33.25%
Profile 7	38	0	3	0	23.75%
Profile 8	28	17	5	70	24.76%
Profile 9	41	16	5	55	31.15%
Profile 10	78	18	6	5	36.36%

Chapter 7

Conclusions

In this section, we draw some conclusions from the work done and present some work to be done in the future.

7.1 Achievements

After analysing the obtained results, we can draw some conclusions regarding what was achieved on the work done during this thesis development. First, we demonstrate that it is possible for an agent, to learn only based on observed player behaviour. And this was enough to make agents play in a slightly different way among them. This difference can be seen in both solutions, for example in the test with the non normalized data, we have a profile that moves straight to the end objective, in opposition we have a profiles that cannot reach the end. It was also possible to make the agents play on a different level, other that the ones that were used to train it. Although, the disadvantage is that this new level needs to have the same layout has the previous ones.

Even though the model we used here was the simplest one, and has some limitations, making it not the best one. For example, it tends to make agents less interested in level exploration, and punish those who try to explore. Even thought with those constrains, the obtained results were positive, where most of the agents performed similar to the respective player averages.

The division of the agent behaviour into navigation and combat proved to be a good addition. This division helped to simplify the structure of agents, and had an impact on the learning process. Thanks to this when training the agents, we could divide the observed behaviour into parts, which allow the training to be focused on a specific part. Other good addition was the area division, and this helped by reducing the time taken by the agents to learn a profile. Although, it creates a new problem, when diving the map we might lose the level structure. In addition, to define these areas, it is necessary to be careful with the number of areas that are defined and the size of them. And this design might not be suitable to every video game.

When talking about the IRL algorithm, choosing the use of the MaxEnt algorithm turned out to be a good choice. This algorithm was already prepared to receive the player behaviour in the form of

a list of trajectories. Although, this algorithm has some limitations such as: It was necessary for the all the trajectories, to be of the same size, so it was necessary to add extra elements to some of the trajectories, creating duplicated ones. Another problem with this algorithm is the time it takes to provide results, during the research, we saw that the default value for the training epochs was 60, and this value with the original level structure simply takes a long time to obtain results. To solve this we created the separation of a level into areas.

One final mention is regarding the navigation model that we used. We decided to use a very simple and basic model, a gridworld approach, where each agent's state corresponds to a direct cell of the map. The use of this model had a particular impact on exploration, where some of the agents got stuck in some areas of the map. Furthermore, this model has the disadvantage of when we obtain a reward function for a level, we cannot reuse it on another level, unless this one has the same exact positions as the previous.

Thus, even thought that we have those problems and limitations with the tools used, it is commendable that the results obtained have been positive.

7.2 Future Work

Regarding the future of the work done in this thesis. The main goal now is to first create a new navigation model that doesn't depend fully on the level structure. To do this, we will have to change from a gridworld approach, to a more abstract one. We will have to make a state space similar to that of the combat mode, where the state is a combination of different variables. In this case we could use the distance to the different elements of the level(coins, objective, ...) and the percentage of the explored map. In addition, we will probably need to consider more complex actions instead of the basic ones.

With this change it might be possible to create a framework that can be used and reused on different levels, and possibly on different video games (considering that they are similar in terms of objectives and mechanics).

We can also experiment with different IRL algorithms to test if we can find an alternative to the MaxEnt, especially one that doesn't take much time, and simultaneously be more resilient to behaviour noise.

In addition to all of this, in the future we want to make the agents capable of testing different components of a game. For example if it can reach the end, if it can explore 100% of a level, or give a value for how difficult the level is.

Bibliography

[1] J. A. Hartigan. *Clustering algorithms*. John Wiley & Sons, Inc., 1975.

[2] L. E. Nacke, C. Bateman, and R. L. Mandryk. Brainhex: preliminary results from a neurobiological gamer typology survey. In *International conference on entertainment computing*, pages 288–293. Springer, 2011.

[3] H. O. Hartley. Maximum likelihood estimation from incomplete data. *Biometrics*, 14(2):174–194, 1958.

[4] M. A. Tanner. The data augmentation algorithm. In *Tools for Statistical Inference*, pages 90–136. Springer, 1996.

[5] R. M. Neal and G. E. Hinton. A view of the em algorithm that justifies incremental, sparse, and other variants. In *Learning in graphical models*, pages 355–368. Springer, 1998.

[6] T. Minka. Expectation-maximization as lower bound maximization. *Tutorial published on the web at http://www-white. media. mit. edu/tpminka/papers/em. html*, 7:2, 1998.

[7] C. M. Bishop et al. *Neural networks for pattern recognition*. Oxford university press, 1995.

[8] S. Russell. Learning agents for uncertain environments (extended abstract). In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, pages 101–103. ACM Press, 1998.

[9] A. Y. Ng and S. J. Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, page 663–670, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1558607072.

[10] B. D. Ziebart, A. Maas, J. A. Bagnell, and A. K. Dey. Maximum entropy inverse reinforcement learning. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, page 1433–1438. AAAI Press, 2008. ISBN 9781577353683.

[11] P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, 2004.

[12] F. d. M. Silva, I. Borovikov, J. Kolen, N. Aghdaie, and K. Zaman. Exploring gameplay with ai agents. *arXiv preprint arXiv:1811.06962*, 2018.

[13] EA. Sims mobile, 2010.

[14] L. Mugrai, F. Silva, C. Holmgård, and J. Togelius. Automated playtesting of matching tile games. In *2019 IEEE Conference on Games (CoG)*, pages 1–7. IEEE, 2019.

[15] C. Holmgård, M. Cerny Green, A. Liapis, and J. Togelius. Automated playtesting with procedural personas through mcts with evolved heuristics. *arXiv*, pages arXiv–1802, 2018.

[16] C. Holmgård, J. Togelius, A. Liapis, and G. N. Yannakakis. Minidungeons 2: an experimental game for capturing and modeling player decisions. 2015.

[17] V. Sriram. *Automated playtesting of platformer games using reinforcement learning*. PhD thesis, Northeastern University Boston, 2019.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[19] C. Holmgard, A. Liapis, J. Togelius, and G. N. Yannakakis. Generative agents for player decision modeling in games. 2014.

[20] B. Tastan and G. Sukthankar. Learning policies for first person shooter games using inverse reinforcement learning. In *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'11, page 85–90. AAAI Press, 2011.

[21] Epic-games. Unreal tournament, 1999.

[22] P. Rastogi, K. Sun, Z. Wang, and X. Li. Predicting player's actions through inverse reinforcement learning.

[23] B. Wang, T. Sun, and X. S. Zheng. Beyond winning and losing: modeling human motivations and behaviors using inverse reinforcement learning. *arXiv preprint arXiv:1807.00366*, 2018.

[24] N. Yee. Motivations for play in online games. *CyberPsychology & Behavior*, 9(6):772–775, 2006. doi: 10.1089/cpb.2006.9.772. URL `https://doi.org/10.1089/cpb.2006.9.772`. PMID: 17201605.

[25] Blizzard. world of warcraft, 2004.

[26] Nintendo. Legend of zelda, 1986.

[27] M. Alger. Inverse reinforcement learning, 2016. URL `https://doi.org/10.5281/zenodo.555999`.