Plan and Goal Recognition with Application to Real-Time Strategy Games

by

LORTHIOIR, Guillaume

Dissertation

submitted to the Department of Informatics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

The Graduate University for Advanced Studies, SOKENDAI
September 2021

Acknowledgements

First and foremost, I am extremely grateful to my supervisor, Prof. Katsumi Inoue, for his invaluable advice, continuous support, and patience during my Ph.D. study. His knowledge and plentiful experience have encouraged me in all the time of my academic research and daily life.

I would like to give a special thanks to Dr. Florian Richoux, who was like a second supervisor for me and kept providing his help and advice even after departing from our lab. His expertise in my field was most welcome, and I really appreciated his frankness when giving me feedback about my work.

I would also like to thank my jury members for their comments, discussion, and attending to my defense. Thanks to my friends and colleagues at Inoue lab and the "International Affairs and Education Support Team". It is their kind help and support that have made my study and life in Japan a wonderful time. Finally, I would like to express my gratitude to my family and my girlfriend. Without their tremendous understanding and encouragement in the past few years, it would be impossible for me to complete my study.

Abstract

Plan and goal recognition are challenging problems that were introduced by the AI community a few decades ago. Plan recognition is the task of inferring the plan, and by extension, the goal, which accounts for the observed behavior of an agent. On the other hand, goal recognition only focuses on inferring the goal of an agent when observing its behavior, and it can be seen as a sub-problem of plan recognition. There is a wide variety of applications to these tasks, for intelligent and robot assistants, automatic driving, suspect behaviors monitoring, multi-agent cooperation, or competition like for games, for instance, to obtain adaptive AIs. Although plan and goal recognition problems have existed for a while in AI, their complexity has hindered their application to real-world scenarios. But recently, with the improvement in terms of computation power and machine learning approaches, these restrictions start to fade.

In this thesis, we study a method for plan recognition in Real-Time Strategy games (RTS) and another method to infer an agent's potential goals to be able to do plan recognition on this agent. The purpose of the first method is to infer the plan and goal of the opponent player in real-time in an RTS game, especially during the first minutes of the game, as they are usually critical and often determine the outcome of a game. This method is based on plan recognition as planning, a new paradigm proposed by Miquel Ramirez and Hector Geffner in 2009 and changed the way to tackle the plan recognition problem. Furthermore, we used heuristic planning and pruning to make our method effective for real-time recognition. The second method aims at inferring the potential goals of an agent after observing its behavior. The purpose is not to find precisely the goal that the agent is trying to achieve but rather to generate a set of candidates as the potential goals that the agent might achieve. We will see later that this set of candidates is needed for many current plan recognition approaches. Our approach is particular, as other approaches often take this set of candidates for granted, an assumption that we found not realistic in many cases.

In the first chapter, we will provide background about the plan and goal recognition problems. We will explain why we use games as a testbed and why especially RTS games and StarCraft, and then describes the organization of the thesis. Chapter 2 introduces background knowledge that is needed for the rest of the thesis. Then chapter 3 presents the online plan recognition that we developed for the RTS game StarCraft, the different experiments that have

been done to evaluate our method, and the possible improvements and extensions of the method. The second contribution of the thesis is presented in chapter 4 where we introduce the problem of inferring an agent's goals. It is not only related to games but much more general. We detail our problem and its formalization, present the experiments that we conduct to evaluate this goal recognition method, and then the possible extensions of the method. Chapter 5 concludes the thesis with perspectives of this work, discussion on how to combine both contributions and conclusion.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Being able to infer the strategy that your opponent will use against you in a Real-Time Strategy game (RTS) is one of the critical skills that human players should master to improve their level in the game. Professional players use scouting for this purpose, and the better they do it, the higher their winning chances are. The problem is, where it is almost an automatism for a human player to do strategy inference using its past knowledge about the game, it is a complex problem for an AI player to do the same, especially with partial observations of the game state. We developed a method to infer in real-time the goal that the opponent player is trying to achieve in an RTS game and the most likely plan for the opponent to reach it. Here, what we call the player's plan is the set of buildings it will create and in what order. The goals are usually a certain amount of specific resources (resources including building and units, not only natural resources) that the player will try to reach as fast as possible, for example, a small group of military units in order to perform a quick attack on the opponent's base. This method considers uncertainty, which is represented by the "fog of war" in RTS games, and hides the actions performed by your opponent. Our work is based on the game StarCraft: Brood War[1] which is a canonical RTS game released in 1998, but our method could be adapted to other RTS games without too much difficulty. For a more general purpose, we also developed a goal recognition method; this method has the particularity to build a set of possible goals for the agent, where the other approaches start with a given set of possible goals and will select the most likely one. We do not predict which goal is the most likely one with our method, but we build the set of possible goals of the agent. Knowing that most of the current plan recognition approaches need a set of possible goals to work , we could combine such a plan recognition

[1]StarCraft and its expansion StarCraft: Brood War are trademarks of Blizzard Entertainment^{TM}

method, like the one we develop for RTS games, in order to have a complete framework for plan and goal recognition.

This chapter will introduce the plan and goal recognition problems and detail related work about them in section 1.1. In section 1.2 we discuss the application of plan and goal recognition in the field of game AI, what has been done, and why it is an interesting testbed. Finally, section 1.3 will detail the contributions and the organization of the thesis.

1.1 Plan and Goal Recognition in AI

Home automation, robotics, intelligent personal assistants will soon be interacting with humans daily. Nevertheless, for this, we need artificial intelligence able to understand and interpret human intentions. It is not trivial. Even currently, the knowledge about an agent's intentions is important, in the case of cooperation or competition in multi-agent systems, for an intelligent assistant that tries to guide or help a human user, or just even for marketing, when trying to target some specific customers. The ISS-CAD problem [1] illustrates this problem; a free-flying robot observes an astronaut performing a task on the International Space Station and tries to help him. In hazardous areas where humans require a robot's assistance, plan recognition can also be used to allow the robot to act as a scout for the human by guessing in which direction the human will move [2]. This problem is called "plan recognition" (or intention recognition sometimes). It is a challenging problem with many applications, and that has been studied for some time in AI research.

The plan recognition problem was first introduced by Schmidt *et al.* [3] in 1978. They designed their BELIEVER system, a psychological theory of how humans observers understand the actions of others, to understand how descriptions of observed actions are utilized to attribute intentions, beliefs, and goals to the actor. Their research is really oriented on the psychology of the problem but allowed other approaches to emerge later, as they provide a formal description of the process that they implemented in their BELIEVER system. Kautz and Allen [4] introduced a new theory of plan recognition that handled concurrent actions, shared steps between actions, and disjunctive information. The theory allows one to draw conclusions based on the class of possible plans being performed. It employs circumscription to transform a first-order theory of actions into an action taxonomy which can be used to logically deduced the complex actions an agent is performing. The idea is that every observed action is part of one or more top-level plans. Using this restriction plan recognition becomes a task of

nonmonotonic deduction where you need to find the top-level plan that includes most of the observed actions. While being recognized for its clarity, this work has several flaws. It does not provide information about how more likely a plan can be than another that includes similar actions. The distinction between top-level plans and the rest makes it impossible for some actions to be top-level plans just by themselves. Set minimization as a principle for abduction is wrong. Charniak and Goldman noted these flaws and presented a formal model of the plan recognition process based upon probability theory that deals with them [5]. Their model consists of a knowledge-base of facts about the world expressed in a first-order language and rules for using that knowledge-base to construct a Bayesian network. The network is then used to find the plans with the highest probability according to the agent's observed actions. A few years later they extended their model in a journal article [6], one of their main contributions is the fact that they are able to decide to what degree the evidence supports any particular plan hypothesis thanks to their Bayesian network. They also introduced a probabilistic model for plan recognition at a time when people believed such a model would be far too complex and time-consuming in terms of computation to treat the plan recognition problem. Continuing in this direction, Goldman *et al.* [7] introduced a new abductive, probabilistic theory of plan recognition. This theory is centered around a model of plan execution when the previous methods have been based on plans as formal objects or on rules describing the recognition process. Their model accounts for phenomena omitted from most previous plan recognition theories: like the cumulative effect of a sequence of observations of partially-ordered, interleaved plans and the effect of context on plan adoption. Their model even allows exogenous effect like actions being executed by another agent than the one we are focusing on. It uses a plan library compiling the possible plans that the agent might be following and specify probabilities for a restricted set of hypotheses about the agent's behavior to provide a framework for abductive plan recognition. Following these works, many approaches based on probabilistic models emerged. Some are based on probabilistic grammars, like Pynadath and Wellman [8], arguing that Bayesian networks will not support the huge amount of observations that one could get in a real-world plan recognition problem, and that probabilistic grammar can handle it. Approaches based on Hidden Markov Model (HMM) started to be used as it is quite handy to model evolving environments under uncertainty and it was already used for speech recognition for several years. The works of Bui *and all.* are probably the most famous for this [9, 10]. They use HMM to be able to perform online-plan recognition as they demonstrate that such an ap-

proach scale reasonably well for some real-world plan recognition problem. Geib and Goldman [11] use also a probabilistic algorithm based on plan tree grammars that could address several issues in the field like, the execution of multiple plans, partially ordered plans, and failing to observe actions. To handle uncertainty and structured representations Singla and Mooney [12] mix probabilistic and logical approaches, as they argue that previous approaches mostly use logical methods that do not handle uncertainty or purely probabilistic methods that do not handle structured representations. Geib and Kantharaju designed an incremental supervised learning algorithm [13] to extract the Combinatory Categorial Grammars (CCGs) that were used by Geib and Goldman [11] for plan recognition. The algorithm learns CCGs that capture the structure of plans from a set of successful plan execution traces, which could alleviate the time-consuming task of manually encoding the CCGS. However, this is just a start, and more experiments need to be performed to increase the algorithm's performance and deal with complex knowledge domains. Though, many of these approaches rely on plan libraries in order to infer the agent's plan. Ramirez and Geffner [14] shown that one can ignore the plan libraries by using planning, with enough knowledge about the environment and the agent, plans matching the observations can be built using a planner. Following this assumption, Ramirez and Geffner use POMDPs [15] to perform goal recognition as they still need to solve this part of the plan recognition problem. Inverse planning is another way to deal with this problem without using a plan library, as shown by Baker *et al.* [16, 17, 18] when trying to understand the human way of thinking when performing plan recognition but it suffers from the same limitation as POMDP since it does not scale, most of the real-world situations have a too large state space for these methods.

After the 2000s, the concept of goal recognition started to become widespread, and several works focused their attention on it instead of tackling the whole problem of plan recognition in which goal recognition is included. It does make sense for some topics in which the real concern is the goal that the agent is trying to achieve and not the way it will achieve it. Hong [19] proposed a method based on graph analysis, it first builds a goal graph to represent the observed actions, the state of the world, and the achieved goals. Then, analyze the goal graph at each time step to recognize the goals that are consistent with the observed actions so far. This method does not need a plan library and works in polynomial time and polynomial space but, often, cannot decide exactly which goal was followed if there was only one goal that interested the agent, instead, it will return several possible goals. It does not take into account

uncertainty too. In fact, after the work of Ramirez and Geffner [14, 15] most of the works have been focused on goal recognition instead of plan recognition. Though, as stated before, their method does not scale on large domains due to the calls of the planner that can be too costly if there are many possible goals. E-Martín *et al.* [20] address this problem using cost estimates and a plan graph to infer probability estimates for the possible goals of the agent. They prune the plan graph according to the observations to reduce the computation time of their method. Their approach yields faster results than Ramirez and Geffner when still providing high-quality solutions. Later, other approaches based on Ramirez and Geffner paradigm but using heuristics to decrease the computation time due to planning have been introduced. As the one of Vered and Kaminka [21] that is applied to continuous spaces where most of the previous approaches focus on discrete ones. Or Pereira *et al.* [22] that use landmark-based heuristics, they define landmarks as facts or actions that must occur in the plan when trying to achieve a specific goal. Pereira *et al.* extended their idea in a journal article [23]. In their approach, goals are ranked according to the percentage of landmarks that the agent has achieved, with each goal having its specific landmarks, the goals with the highest percentage of achieved landmarks being the most likely ones. They performed extended evaluations of their approach against the state-of-the-art methods and obtained similar or often better results. Höller *et al.* presented their approach for Plan and Goal Recognition as Hierarchical-Task Network(HTN) planning [24]. HTN planning creates plans by decomposing tasks into smaller and smaller tasks, with the smallest task being primitive actions. Their approach uses unmodified, off-the-shelf HTN planners, which greatly help for generalization. The approach scales well with large sets of possible goals (thousands) and results in high recognition rates. However, it is based on the same strong assumption as [14], which are that the agent is rational, and so follows optimal plans and that the agent's actions are fully observable, so there is no noise. Another problem, similar to [14], is that the possible goals returned by the goal recognition algorithm are not ranked, so we do not know which one is the most likely. Recently, Amado *et al.* [25] used deep autoencoders to overcome the need for expert knowledge to design the goal recognition problem by learning from a flow of images this domain theory and performing automated planning and goal recognition. That is an important step forward for automated goal recognition. However, their method does not provide as good results as non-automated ones, and it needs to observe all possible transitions of the domain in order to infer its encoding, which can be impossible to obtain. Mirsky *et al.* introduced two new problems, the "goal recognition design for plan libraries"(GRD-PL) problem and the

"plan recognition design"(PRD) problem [26]. Both of these problems are pretty related to the "goal recognition design" (GRD) problem previously introduced by Keren *et al.* [27]. The GRD problem aims at designing domain theory that improved the goal recognition process in this domain. Especially, it aims to reduce the ambiguity between different possible goals for the agent by removing some actions of the domain theory without altering the agent's ability to reach the different goals, thus improving the accuracy of goal recognition in this domain theory. The GRD-PL and PRD problems introduced by Mirsky *et al.* are similar, but this time focus on designing a plan library that improves the recognition process by making the different plans within the library as different from each other as possible according to a specific metric defined in their paper. However, they make two strong assumptions that reduce the domain of application of their work which are that actions are deterministic (an observation is mapped to exactly one action in the plan library), and the agent and the system are fully observable. [28] and [29] are two good surveys of the different approaches that have been developed for plan recognition and goal recognition, recently, Van-Horenbeke and Peer [30] did a survey that emphasized the problem of plan, and goal recognition as a whole. Aha detailed the different applications and prospects of goal reasoning [31], which could give good hints about how the knowledge of one's goal could be used.

1.2 Game AI: a Challenging Domain of Application

1.2.1 Plan and Goal Recognition in Games

Games have often been seen as milestones for the progress of AI, that was the case when Deep Blue [32] defeated the world chess champion, Kasparov, in a six-game match in 1997. That was incredible at the time, the chess game having an estimated state-space complexity of 10^{47}. In 2015 AlphaGo [33] won against Go world's best players, the game of Go being played on a board of 19x19 tiles, the state-space complexity is estimated to be around 10^{170}, a staggering number. Recently, a new step has been taken with AlphaStar [34], an AI able to reach the top 1% of the best players in the world of StarCraft 2, a real-time strategy game that was seen as far more challenging for AIs than the game of Go. For several reasons including that the state-space complexity is far bigger than Go, there is a lot of uncertainty about the opponent's moves, and decisions have to be made in real-time. Games have also proven to be very good test beds for plan recognition, after the first works about probabilistic models for plan recognition, Albrecht *et al.* [35] developed a model based on dynamic Bayesian networks

applied to a multi-player Dungeon adventure game with thousands of possible actions and locations. Their approach allows the use of incomplete, sparse, and noisy data. They use a rather simple representation of the game where a player is following one quest at a time and performing actions at different locations, so they try to infer what is the current quest followed by the player and what will be its next action and location. Quests are achieved by performing specific actions in specific locations. Ontañón *et al.* [36] developed an online Case-Based Planning architecture that uses expert traces to learn how to plan strategies in RTS games and implemented a bot in the RTS Wargus to test their architecture against the basic AI of the game. Later, Ontañón acknowledged that, although promising, this approach struggles to generalize due to the large variety of situations that can arise in RTS games. Ontañón explored the possibilities of scaling up Monte Carlo Tree Search algorithms to RTS games to compute optimal policy to play the game [37]. He uses μRTS as a testbed and a fully observable version of the game. It is unsure whether this method could scale on much more complex RTS games such as StarCraft. Ontañón and Buro also introduced a new approach called Adversarial Hierarchical-Task Network Planning based on Hierarchical-Task Network (HTN) planning and the minimax game tree search algorithm [38]. Their method allows adversarial planning for RTS games. It has been evaluated μRTS where it shows that the branching factor of the method scale better comparing to other state-of-the-art search algorithms for RTS games while still providing very efficient plans. However, as stated before, μRTS is a very simple game compared to real RTS games, and this without adding uncertainty to the balance. Kabanza *et al.* [39, 40] discuss about the algorithmic challenges of performing plan recognition in games. Schadd *et al.* [41] describe opponent modeling through hierarchically structured models of the opponent behaviour using Spring RTS (Total Annihilation clone) for experiments. Weber and Mateas [42] evaluated several machine learning algorithms on replays that were labeled with strategies, and found that the best way to do strategy prediction would be to use a mix of different algorithms, as some are more efficient for early-game predictions but are supplanted by others for mid-game predictions. They also provide their dataset for further comparisons. Synnaeve and Bessière [43, 44] use Bayesian models to predict the build order of the opponent player in StarCraft, their method works in real-time and handle uncertainty. Both of these approaches need to be trained on replays and have their efficiency decreasing in a linear way, at best, according to noisy data. Synnaeve and Bessière sum up the different approaches based on Bayesian models that they developed for RTS games in a journal paper [45]. Their other

contributions (aside from the plan recognition method) are a method for units micro-managing during a fight with the opponent. Another method to predict tactical moves like where, when, and how the opponent can attack, and vice versa for the player. And finally, a model to perform armies clustering and predict the output of a fight (which side will win). Later, with the explosion of Deep Learning, approaches based on it have appeared, Min *et al.*[46] use LSTM to predict in which order the player will accomplish the different goals in an Open World Serious Game, later, they incorporate player gaze traces in the data and saw an improvement of the predictive power of the model[47], but their model does not take into account uncertainty as it is intended to accompany the player through the game. Synnaeve *et al.* [48] present new encoder-decoder and use it to predict the evolution of the position of the different units and buildings in StarCraft, thus helping to predict tactical moves from the opponent. Geib *et al.* introduced a model for cooperative behavior based on the interaction of plan recognition and automated planning [49]. One agent called *supporter* observes another agent, the *initiator* and will try to infer its plan. The *supporter* will ask the *initiator* if its plan is this one, and if so, the *supporter* will starts negotiation and ask for tasks that could be achieved in order to help the *initiator*. Plan recognition is used to guess the goal of the *initiator* and planning to find tasks to help the *initiator*. They used a single representation for plans and actions during the whole process. This representation is the one based on CCGs introduced by Geib and Goldman[11]. Since their model is based on a cooperative agent that has a complete vision of the *initiator* actions, they do not deal with uncertainty and make wrong inference about the *initiator* plan is not so penalizing as the *supporter* will ask confirmation about the plan that was inferred. If the *initiator* says it is not the correct plan, then the *supporter* will keep asking with the other possible plans that it has inferred. Kantharaju *et al.* [50] developed a greedy version of the algorithm introduced in [13] as the original algorithm is exhaustive and cannot scale for plans including more than three actions. They tried to learn CCGs for plan recognition in μRTS. Although they successfully learned CCGs, there are still some scalability problems when the plans are too complex, and the learned CCGs are not yet optimal to perform plan recognition, as they acknowledge it at the end of the paper. Then they tried to combine this work with Monte-Carlo Tree Search (MCTS) [51] in order to make it scale better to RTS games. MCTS is used to perform plan recognition thanks to the CCGs learned by the algorithm from [50]. However, in their work, Kantharaju *et al.* consider that the actions of the player are fully observable and that the timing of each action is also known. That is not the case when dealing

with a game such as StarCraft, when trying to infer the plan of the opponent player, only part of the actions are observed, and most of the time, we do not know when the observed actions have been performed. Kantharaju *et al.* also worked on player modeling, they tried to trace player knowledge in a parallel programming educational game [52] in order to generate procedural levels that are adapted to the player's current level of knowledge. They combine machine learning to domain knowledge rule to detect if specific skills have been applied to solve specific problems and conclude that the player has learned the skills. When this work is not directly related to plan recognition in games, player modeling is.

Ontañón *et al.* made a very interesting survey [53] about AI for RTS games. They detail the different AI challenges posed by RTS games, work that has been done in the field to tried to address them, and challenges that have yet to be solved.

1.2.2 Real-Time Strategy Games

RTS game is a well-known type of game where the players have an aerial view of a map and usually need to gather natural resources, build structures and units, and defeat their opponents by beating their army and destroying their base. StarCraft is one of the most famous RTS games, it was released in 1998. Millions of players played the game and it was one of the very first games to have international competitions with big prize pools. Many replays (recording of a game between different players) are available online and for free, to help to improve your skill by learning from the others. Thanks to this and the complexity of the game, researcher that were interested in the field developed different tools to interact with the game, performed data-mining and provide interesting dataset. Most of the competitive matches in StarCraft are 1 versus 1 player. Each player selecting a race to control before the game, there are three races in StarCraft, Protoss, Terran, and Zergs. Each with their very own play style. Protoss units are very powerful but expensive, where Zerg units are weak but cheap and usually try to overwhelm their opponent with sheer numbers, Terran units are situated in between. Nevertheless, there are common points for all the races which are that natural resources are gathered by *workers*, and used to build *buildings*, train *units* and research *technological advances*. That is why the players need to manage natural resources carefully. In most RTS games (StarCraft included), each unit and building has a sight range that provides the player with a view of the map. Parts of the map not in the sight range of the player's units are under a fog of war and the player ignores what is and happens there. This means that players have to deal with uncertainty as they do not know what their opponent is doing without sending units to scout it, and even

then, it is difficult to gather important information as the opponent will usually try to kill the scout and hide its units or buildings. Figure 1.1 shows the screen of a Terran player a the beginning of a game, we can see one building and another in construction, some units gathering the minerals. On the top left corner, we can see the screen that becomes darker, that is where the fog of war starts, and the player cannot see anything in the area beyond except if it moves a unit there. In the bottom left corner, there is the minimap which is a small miniature of the game map, where buildings and units are represented as small colorful dots, in this case, the vast majority of the minimap is black because the player did not explore it yet. The white rectangle on the minimap represents the area that the player is currently seeing on its screen. In StarCraft, there are two kinds of natural resources, and each of them is required to build

Figure 1.1: Example of the screen of a player

specific buildings or units. Likewise, there are different types of military units, each of them has its own strengths and weaknesses (some can fly, are fast or slow, etc...), and can be trained in different buildings. Players, therefore, have many strategies at their disposal, they often follow

a certain pattern in the way they will manage natural resources and buildings. This pattern is called "Build Order" and gives a good hint about the strategy that a player will follow, like if it will performs a very early attack to harass its opponent or if it will play defensively and just developed a strong economy to attack later. Figure 1.2 shows some basic units and building for the Protoss player. The Nexus is one of the most important buildings, it allows to produce Probes, Probes are the workers of Protoss and are used to gather natural resources and build buildings. The Gateway is the building that allows the Protoss to produce basic military units such as the Zealot. At the beginning of each game, the players start with a building that allows the production of workers (Nexus for Protoss) and five workers (Probes for Protoss). Then they are free to expand their base as they want with the final goal of beating their opponent, who usually starts at the opposite side of the map.

Figure 1.2: Basics units and buildings for Protoss

All these parameters make it very hard to develop AI players that can compete with humans, especially when long-term planning and tactical decisions are that important in the game. It is something that humans will learn with experience by playing the game, but it is far more complex for AIs. To offset this difficulty games developers will often provide the AIs with unfair bonuses like having extra resources and a complete view of the map, ignoring the fog of war and thus uncertainty. This unfairness will often conduct the players to exploit the weakness of AIs (the behaviors for which it does not know how to answer) at the expense of fun, which is, one of the main points of games. A good plan recognition module for the AIs would probably make them more interesting if they could use this knowledge to their advantage and not just using unfair bonuses. Despite the fact that different methods exist for plan recognition in RTS games, as we saw in the previous section, not all of them are taking into account uncertainty and operating in real-time, and for the ones that do, there is still room for improvement in

terms of accuracy. In addition, with the emergence of deep learning, more and more methods tend not to be concerned with this problem anymore and seek to develop a global optimal policy that will look for each case what could be the best action from the AI at each time step. Which can provide really good results like for [34] but cannot be extended to other games, just even other RTS games, without changing a lot of parameters and training the model again for a huge cost. Also, there is the problem of updates, which can happen often for recent games, and in this case, without a proper transfer learning system, the model should be learned again. In contrast, our method could easily be used for other RTS games by adapting the planner that we use and the pruning.

1.3 Organization of the Thesis

There are two main contributions presented in this thesis. The first one is an online plan recognition method for RTS games. As we saw in the previous chapter, RTS games are complex environments, and most of the current plan recognition methods do not scale with the state space of such games. Moreover, for the few approaches that could scale, like the approach of Höller *et al.* [24], they often do not deal with partial observations and assume that the observed actions are ordered. However, it is not the case for most RTS games, we only get partial observations, and these observed actions are not always ordered. The plan recognition method introduced in this thesis handles all these problems and compared to previous work like [42, 43, 44] that use machine learning to perform strategy prediction in StarCraft, our method does not use machine learning and thus does not need a set of training data, instead, it requires expert knowledge. Therefore, it makes it easier to use for games without much data from the players as the developers have expert knowledge about their game. Our method uses heuristic planning to compute in real-time different plans that the player might follow in order to achieve different goals. Then, using the partial observation that we get during the game, we will refine the different plans according to these observations, prune some plans that we consider as very unlikely and select the plan and goal that is the most likely to be followed by the player.

The second contribution is a goal recognition method that, compared to most of the existing goal recognition methods that predict the most likely goal of the observed agent from a set of possible goals, builds this set of possible goals by observing the agent trying to achieve them. Often, previous works assume that this set of possible goals for the agent is given for the goal recognition problem or even plan recognition. However, it is usually not the case, especially in

complex or unknown environments. For example, when we are not sure about all the actions that the agent can execute or if the number of possible goals is just too important. That is why we developed a method that can infer the set of possible goals of the agent. Our method allies concept learning with a representation of the environment based on propositional logic that allows us to generate a hypothesis in the form of a DNF representing the agent's possible goals. This contribution is not only related to games but much more general. We observed an agent performing some task until it eventually achieves its goal. By repeating this kind of observation, we will get a dataset that we divided into a set of positive examples, which corresponds to the states where the agent achieved a goal, and a set of negative examples, which corresponds to the states where it did not. Using the set of positive examples, we will create the concept that represents the agent's goal and refine this concept with the set of negative examples.

This thesis is organized as followed. Chapter 2 introduces background knowledge that is needed for the rest of the thesis, section 2.1 described the main plan recognition approach from which our method is inspired, and section 2.2 the definitions associated with this approach, section 2.3 details what concept learning is. Chapter 3 presents the online plan recognition that we developed for the RTS game StarCraft in section3.1 and 3.2. Section 3.3 presents the experimental protocol that was used to evaluate our method, the results, and analysis of this evaluation, and section 3.4 details the possible improvements of this method and some discussion. The second contribution of the thesis is presented in chapter 4 where we introduce the problem of inferring an agent's goals in section 4.1, its formalization, and the algorithm we developed to solve it. Section 4.2 presents the experiments that we conduct to evaluate this goal recognition method and then section 4.3 the possible extensions of the method. Chapter 5 concludes this thesis with section 5.1 summarizing the contributions of this thesis, section 5.2 discussing about how to combine our two method in a single framework, and section 5.3 detailing further perspectives.

Chapter 2

Background

In this chapter, we introduce the basic to understand the method presented in this thesis. The method for plan recognition in StarCraft is inspired by the work of Ramirez and Geffner [14] so we will detail their method in section 2.1, then, in section 2.2 we introduce some definitions from their papers that we use.

2.1 Plan Recognition as Planning

In their paper [14], Ramirez and Geffner assume that the agent they observe is perfectly rational and always follows optimal plans. The observations are sequences of actions, and even if some actions could be missing because of noise, they assume that the order is the same as it was when performed by the agent. Following these assumptions, the problem of plan recognition is then converted into solving a planning problem and a goal recognition problem. They have a set of possible goals, design an optimal plan to reach each goal from the initial position of the agent, and then according to the observations, they remove the plans and goals that do not include them. For instance, if we look at Figure 2.1, here we have a navigation problem. The agent starts at position "A" and has three possible goals, to reach "C", "I" or "K". By using planning, three plans will be generated, one to reach each possible goal by the shortest way, from "A". Later, we observe the transition of the agent from "A" to "B", all the plans are kept as they all include this transition. Then, the transition from "F" to "G" is observed, which leaves out the goal to reach "C" and its associate plan since the optimal plan to reach "C" from "A" is just to go straight. Only the goals "I" and "K" are kept, and the respective plans to reach them.

As stated in section 1.1, despite being fairly simple and convenient, this method has some

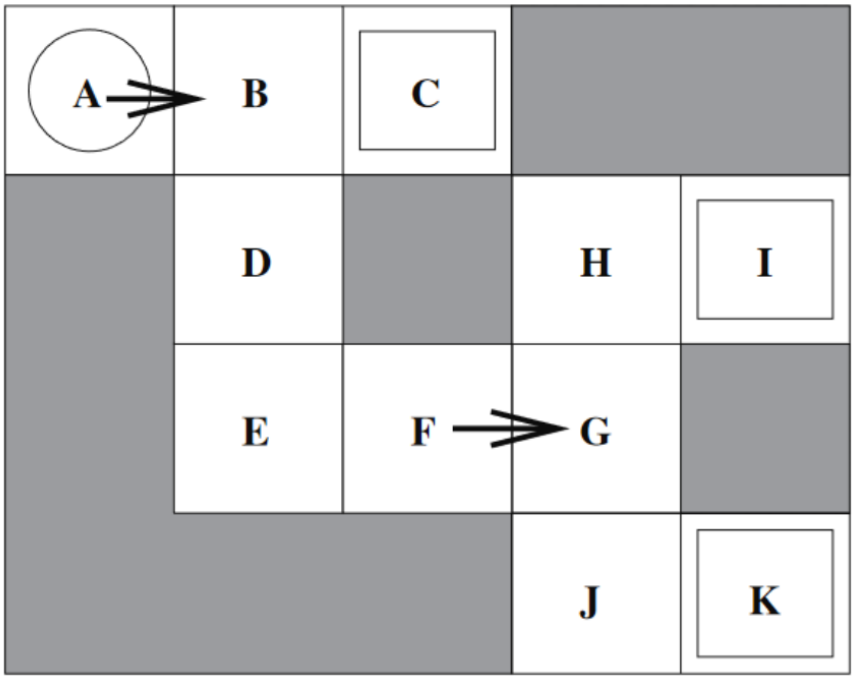


Figure 2.1: Plan recognition for navigation

flaws. First, the assumption that the agent is rational and always follows optimal plan is quite strong and does not work in our case since the agent is a human player and can make mistakes, especially in such a game where there are a lot of things to plan in real-time. We assume that the human player does not always follow an optimal plan to reach its goal and can make mistakes. The fact that observed actions are assume ordered also does not apply to our case. We observe consequences that we link a posteriori to actions, but without being able to determine their order. Hopefully, many actions have some preconditions that can help to have an estimation of the order in which they have been performed. The last, important flaw, that we actually observe in the example is that their method does not provide any information about which goal is more likely than the other ones. It just filters the ones not compatible with the observations. Their work on POMDPs [15] deals with most of these flaws but does not scale for an RTS game such as StarCraft. This is why the approach described in this thesis follows the "plan recognition as planning" paradigm but differs from the original one. Instead, our approach is similar to more recent ones based on heuristics search to reduce the computation time. It is an online method that collects observations about actions performed by the opponent player

(a building or a unit that has been built) and refines the plans previously computed in order to match the observations. It uses a sub-optimal planner that computes a sub-optimal plan for each goal and uses pruning along the process to remove the plans and goals that are not considered anymore. Pruning is done by using heuristic search, we use this also to choose which goal is the most likely to be followed by the opponent player, when the cost of a plan exceeds a certain threshold this plan and goal are removed from the set of candidates.

2.2 Definitions

Here are the definitions from [14] that we use too. The third definition is modified, as in our case, the observations are not always ordered.

2.2.1 Planning Problem

A planning problem is a tuple $P = \{F, I, A, G\}$ *where* F *is the set of fluents,* $I \subset F$ *and* $G \subset F$ *are the initial and goal situations, and* A *is a set of actions "a" with preconditions, add, and delete lists Pre(a), Add(a), and Del(a) respectively, all of which are subsets of* F*.*

Here, we assume that for each action $a \in A$ there is a non negative cost $c(a)$ and the cost of a sequential plan $\pi = a_1, ..., a_n$ is $c(\pi) = \sum c(a_i)$. A plan π is *optimal* if its cost is minimum. The player is assumed to be rational, so to try to follow optimal plans, which means that our planner should generate optimal plans or at least plans that tend to be optimal in order to infer the player's plan. As we explained before, even if the player is trying its best, it is fairly possible that it makes mistakes as the game is not easy, so this is why the plans do not have to be optimal, but rather approximate optimal ones.

2.2.2 Plan Recognition Problem

The plan recognition problem is then defined as a triplet $T = \{P, \mathcal{G}, O\}$ *where* $P = \{F, I, A\}$ *is a planning domain,* $\mathcal{G}$ *is the set of possible goals* G*,* $G \subset F$*, and* $O = o_1, ..., o_m$ *is an observation sequence with each* o_i *being an action in* A *(in our case observations are also consequences of actions but it is trivial to make the conversion to the action made).*

2.2.3 Satisfiability of an Action Sequence

We say that a an action sequence $\pi = a_1, ..., a_n$ *satisfies an observation sequence* $O = o_1, ..., o_m$ *if there is a function* f *mapping the observation indices* $j = 1, ..., m$ *into action indices* $i =$

$1, ..., n$, *such that* $a_{f(j)} = o_j$ *and* $Pre(a_{f(j)}) \in \pi$ *with* $\forall a_k \in Pre(a_{f(j)})\ k < f(j)$.

The solution to the plan recognition problem is then given by the goals that admit the shortest plans compatible with the observations.

2.3 Concept Learning

Concept learning, also known as category learning, is defined by Bruner *et al.* [54] as "the search for and listing of attributes that can be used to distinguish exemplars from non-exemplars of various categories". Merrill and Tennison [55], based on component display theory, argued that concept formation focuses on attributes and examples, where Rosch and Lloyd [56] suggested that the natural concepts in everyday life are learned through examples rather than abstract rules. As expected from something as hard as a "concept" to define, many other researchers tried to give their own definition of the thing, and we will not go further into the details as it will be beyond the scope of this thesis.

Generally speaking, in a concept learning task, the learner is trained to classify objects by learning from a set of example objects and their class labels. The learner simplifies what has been observed by condensing it in the form of an example. This simplified version of what has been learned is then applied to future examples. Because the task is somehow similar to a two-class classification problem, a classification algorithm could be used to estimate how accurate a concept learning algorithm is. We could train both algorithms with the same data and then, using a set of testing data, see how the concept learn by the concept learning algorithm is consistent with it and how the classifier will classify this testing set.

Chapter 3

Online Plan Recognition for RTS Games

In this chapter of the thesis, we detail the plan recognition method that we developed for StarCraft. Using the definitions introduced earlier, in section 3.1 we explain the specific planning problem that we have to solve and how we solve it. Section 3.2 describes the plan recognition process under this planning domain. Section 3.3 presents the experimental protocol that we used to evaluate our method, the results, and analysis of this evaluation. At last, section 3.4 details the possible improvements of this method and some discussion.

3.1 Heuristic Planning in RTS Games

In our case, the fluents composing F are all the resources that a player can get. Here we see resources in a general way, which means that a resource can be the natural resources of the game (minerals and gas that workers can gather) or more advanced resources like military units or buildings. Naturally, advanced resources are harder to get and will need several actions from the player to get them. The goal of a player will be to gather a certain amount of resources in the shortest possible time. When we want to reach a goal state in G with the planner, we can partially give this goal state since there are many possible plans and goal states, but some resources are more important than others in this case. For example, if the goal of the player is to have eight zealots (military units) in order to attack its opponent as quickly as possible, one way to do it could be building workers that will gather minerals, then building a gateway with one of these workers and then producing eight zealots. However, it is pretty slow to use only one gateway, so another plan could be to build two gateway and then produce the eight

zealots. It leads to two different goal states because the amount of gateway is not the same in each plan, but the main goal, which is to produce eight zealots, is not changed. So we just need to specify the relevant resources that the player wants to get, and the planner will build the goal state by itself. For instance, Figure 3.1 shows the case of the goal corresponding to a Protoss strategy called "Fast DT rush". This strategy aims to send as fast as possible three or four Dark Templars (DT) units to kill a maximum of the workers of the opponent before it can react and thus striking a killing blow to its economy. DT that we can observe on Figure 3.2 are invisible units for the opponent and need to do some special actions in order to reveal them. So it is crucial for the opponent to anticipate this strategy from the Protoss player. Back to Figure 3.1, on the left is a possible goal state for this strategy, on the right are the only resources we need to specify for the planner in order to reach this goal state. The resources colored in yellow are preconditions of the ones in red so we do not need to mention them to the planner, it will generate them in order to fulfill the condition leading to this goal state. Gateway is also a precondition to produce dark templars, but in this case, we specify it to the planner because we want precisely two gateways, and not only one.

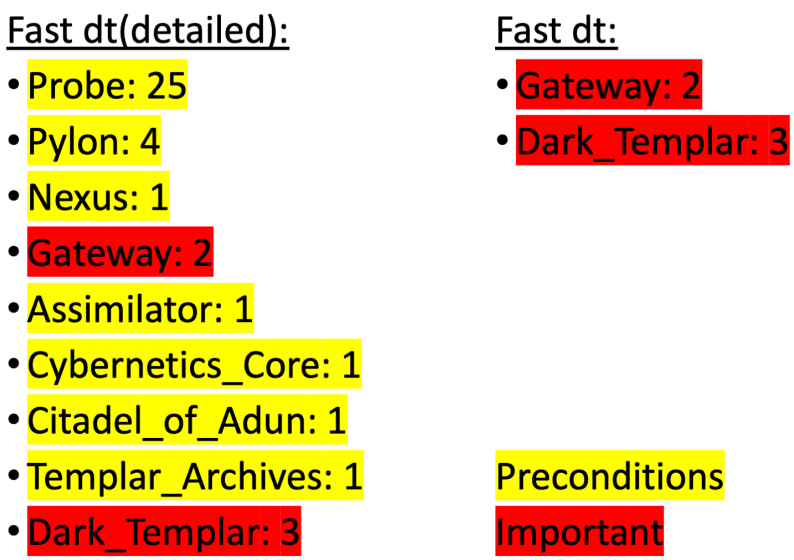


Figure 3.1: Possible goal state for a "Fast DT rush" strategy

Specific planners have been developed for RTS games. The one developed by Chan *et al.* [57] is interesting for us as it runs in a low polynomial time and offers the same performance

Figure 3.2: The dark templar, a Protoss military unit

as an experimented human player. They use sequential planning based on means-end analysis to generate a plan that will reach the goal with a minimum number of actions and natural resources, and next, they use scheduling to adapt this plan to concurrent actions that can be performed in the game. We adapted this one for our planning problem, but other planners could be used to see the planner's impact on our method. Algorithm 1 is the sequential planner based on mean end analysis from Chan *et al.* [57] and using the definitions previously introduced: S is the current set of resources owned by the player and G is a set of values g_i that we want to reach for specific resources i, R_i being the current amount of each resource. The algorithm operates by selecting a sub-goal (in this case, the sub-goal is to reach the desired amount for one of the resources in the goal), decreasing the difference between the initial state and the goal state, and then executing the necessary actions to achieve the sub-goal. Then from the new state which satisfies the sub-goal, the process is recursively applied until we reach the goal state. The output: $Plan$ is a sequence of actions where each action will start after the previous one in the sequence is completed.

Algorithm 1: MEA

input : S, G
output: $Plan$

if $\forall i, R_i \geq g_i$ **then**
  return [ ];
end
$R_i \leftarrow$ some resource where $R_i < g_i$ in S;
$A_i \leftarrow$ action to produce R_i;
$r_i \leftarrow$ amount of R_i in S;
$\alpha \leftarrow$ units of R_i produce by A_i;
$k \leftarrow roundUp((g_i - r_i)/\alpha)$;
$Act \leftarrow$ sequential plan with A_i repeated k times;
$G_p \leftarrow$ [ ];
for $prec_i$ *in* $Pre(A_i)$ **do**
  if $prec_i$ *not satisfy in* S **then**
    $G_p \leftarrow add(prec_i)$;
  end
end
$Precon \leftarrow MEA(S, G_p)$;
$Plan \leftarrow concatenate(Precon, Act)$;
update S with $Del(a)$ and $Add(a) \forall a \in Plan$;
return $concatenate(Plan, MEA(S, G))$

The algorithm 2 is the scheduler also described by Chan *et al.* [57], where $Plan$ is a sequential plan where the actions are sorted by their starting times in increasing order. For

each action $A_i \in Plan$, we check the earliest possible time to start this action in $Plan$ while still fulfilling the preconditions $Pre(A_i)$. Assuming that A_i starts at time t, and the state $R^-(t)$ is the resource state at time t before the effects of all actions that end at time t are added to the resource state, and $R^+(t)$ is the resource state after the effects are added. Obviously, $Pre(A_i)$ are fulfilled by $R^+(t)$. If they are also fulfilled by $R^-(t)$, this means the fulfillment of $Pre(A_i)$ is not due to any actions that end at time t, and we can now move action A_i to start earlier than t, to the previous decision epoch (time where an action starts or ends).

Algorithm 2: Scheduler

input : $Plan(sequential)$
output: $Plan(parallel)$

```
for A_i in Plan do
    t ← starting time of A_i;
    R⁻(t) ← resources state before the effect of actions finishing at t;
    while Pre(A_i) satisfied by R⁻(t) do
        move A_i at t in Plan;
        t ← previous decision epoch;
    end
end
return Plan
```

Algorithm 3 is the complete planning algorithm. It is a modified version of the one in [57] that adds more intermediate goals, as in the original version, they just add the best intermediate goal. It first computes a sequential plan to reach the goal G with MEA and then adds concurrent actions to this plan with $Scheduler$, and the scheduler will also return a set of bottleneck resources that were used a lot in the plan (like some specific building or units), these resources become a bottleneck when they are the preconditions of a specific threshold number of actions. Using this set of resources, we will generate intermediate goals to increase the amount of these resources, and we will check if we can decrease the makespan of the basic plan to reach G by first reaching some of these intermediate goals. We keep incorporating intermediate goals until we cannot improve the makespan of the plan anymore.

3.2 Plan Recognition Method

In this plan recognition problem, we observe the opponent player. First, we will generate the possible plans that the opponent player could use, using Algorithm 3, for each goal, we generate one plan to reach it. We will then get incremental observations about the opponent player's

Algorithm 3: Online planner

input : S, G
output: $Plan$

$Plan \leftarrow Schedule(MEA(S, G))$;
$int\_G \leftarrow (G_1, ..., G_n)$ (a list of intermediate goals to create additional resource);
$bestGoalsList \leftarrow [\,]$
while *There is improvement* **do**
  for G_i *in* int_G **do**
    $P_0 \leftarrow MEA(S, G_i + bestGoalsList)$;
    $S' \leftarrow$ state after executing P_0 from S;
    $P_1 \leftarrow MEA(S', G)$;
    $Plan_i \leftarrow Schedule(concatenate(P_0, P_1))$;
    if *makespan of* $Plan_i <$ *makespan of* $Plan$ **then**
      $Plan \leftarrow Plan_i$;
      $best\_goal \leftarrow G_i$;
    end
  end
  $bestGoalsList \leftarrow add(best\_goal)$;
  $int\_G \leftarrow remove(best\_goal)$;
end
return $Plan$

actions since the method is designed to work online. For each new observation, we check if it satisfies the current set of plans. For each plan, we will check if the observation was part of the plan. As we say before, observations are consequences of actions; for instance, if we observe "Marine" at 2 minutes of the game, we deduct that the action "Build Marine" has been done and that the action "Build Barrack" that fulfill the precondition of "Build Marine" has been done too, and both of them before 2 minutes, but we do not know when exactly. For simplicity, we consider that another similar observation, "Marine", is the consequence of a different action "Build Marine", even if this observation happens later. It makes sense since the human player will remember where it has observed something and can eventually tell if the unit it observes is a new one or one that has been observed before. So, we check if the action was part of the plan. If it is the case, then the plan is not changed. Otherwise, we incorporate this action in the plan by computing a new plan going through this action. Next, we use a heuristic search to choose which goal is the most likely to be followed by the opponent player and eliminate the irrelevant ones to avoid high computational complexity. In previous work, the cost difference between the plan to reach a goal when including the observations and the optimal plan to reach the same goal has often been used as a way to estimate the likelihood of a goal [15, 20, 21]. Based on this

previous work, we also use a heuristic that follows the same principle. The heuristic that we use is then based on a cost that we will compute for each plan; this cost is a three-dimensional vector where these dimensions are respectively the cost of the plan in terms of minerals, gas, and time. For a plan $\pi = a_1, ..., a_n$ where action a_i has a starting time t_i^s and an ending time t_i^e, we have $c(\pi) = (c_m(\pi), c_g(\pi), c_t(\pi))$ with $c_m(\pi) = \sum c_m(a_i)$, $c_g(\pi) = \sum c_g(a_i)$, and $c_t(\pi) = max(t_i^e)$. Then we use a three dimensions space with the dimensions being respectively the costs in minerals, gas, and time, and for each possible goal g_i that the opponent player is trying to achieve, we will project into this three dimensions space the coordinate of the cost of the plan π_i (plan to reach g_i)) that we compute previously and the cost of the plan π_i^{opt} the optimal plan to reach g_i. Here we give more importance to the time, as we believe that a player will always prioritize a faster plan regardless of the cost in terms of minerals or gas, as is it often the case in RTS games, so a plan will be considered as optimal if it achieves its goal in the shortest amount of time. In our case, the optimal plan is the first one that has been computed to reach the goal g_i without modifying it to include the observations. We then draw a sphere of center $c(\pi_i^{opt})$, and radius $r_i = \delta \cdot d(c(\pi_i^{opt}), origin)$ with $origin$ being the coordinate of the origin point in the three dimensions space, $d(a, b)$ the three dimensions distance between the point a and b, and $\delta \in [0, 1[$. We only keep the goal g_i and the plan π_i in our set of candidates if $d(c(\pi_i^{opt}), c(\pi_i)) < r_i$. In other words for each plan π_i we compute this sphere of radius r_i and check if the $c(\pi_i)$ coordinate point is included in the sphere or not to decide if we should remove π_i and its associate goal g_i from our set of candidates. The process keeps going for each new observation, gradually restricting the set of candidates. The plan and the goal that have the minimum distance $d(c(\pi_i^{opt}), c(\pi_i))$ are considered to be those currently being followed by the opponent player. The parameter δ is a constant fixed at the beginning of the whole process, it represents the sensitivity of the pruning, low value of δ means more pruning but potentially less accurate result. We chose to set $r_i = \delta \cdot d(c(\pi_i^{opt}), origin)$ in order to handle the fact that for a costly goal, the margin of error to reach it is assumed higher than for a less costly goal, which means that the longer a plan is, the more different it could be from the actual plan of the opponent player (assuming these two plans reach the same goal). In Figure 3.3 we can see an example of how this heuristic works for a goal g_i with a cost of opt_i for the optimal plan to reach g_i. The plan to reach this goal is pruned if the projection of its cost in the three-dimensional space is not included in the sphere of radius r_i.

One drawback of this pruning method, though, is that the cost of a plan in terms of minerals

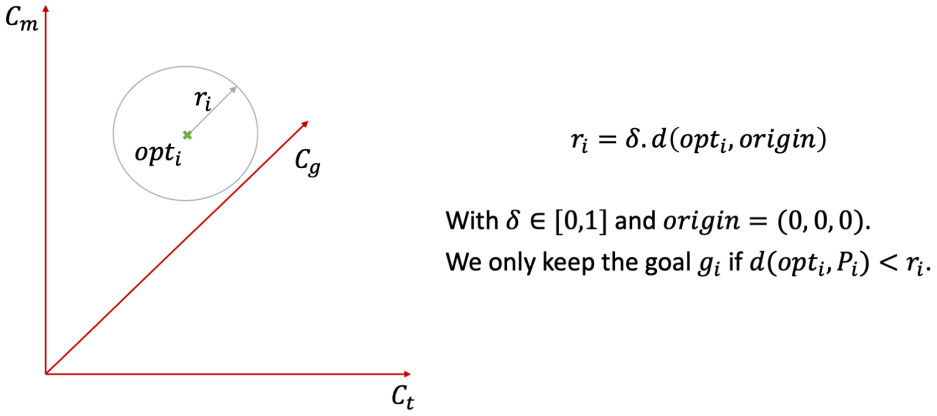


Figure 3.3: Visualization of the three-dimensional cost heuristic

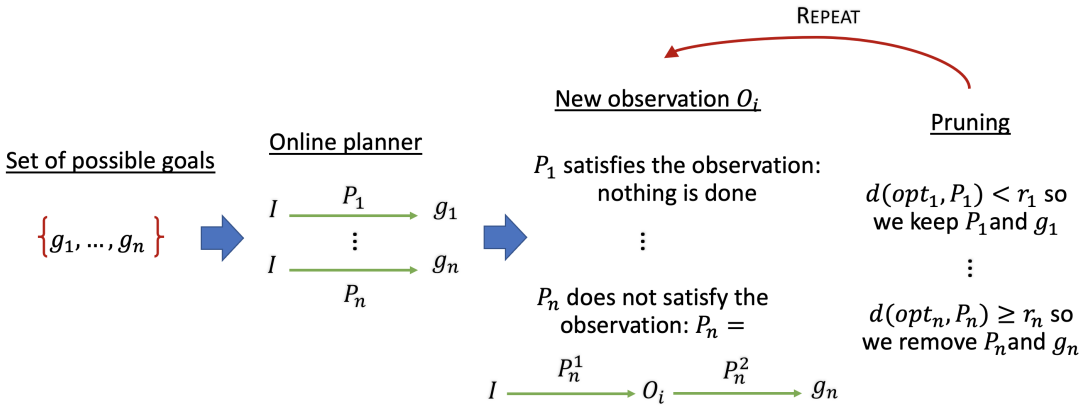


Figure 3.4: Illustration of the plan recognition process

and gas does not evolve linearly with time. We can multiply the cost in terms of time by a specific factor, so it keeps up with the two other costs. However, this factor has to be adjusted for each prediction horizon as the cost in minerals of a plan often increases exponentially with time. That can lead to cases where our planner could build a plan for a specific goal that takes twice as much time as the optimal plan to reach this goal. However, because the cost in terms of minerals is much higher than the cost in terms of time of the plan, it will not be pruned as the radius r_i depends on the three different costs, and one of these costs can be insignificant compared to the others. We used a second heuristic to offset this and compare the performance of both heuristics in the next section. The second heuristic is also based on the cost of the plan in terms of time, mineral, and gas, but this time for each cost, we will check that the value is within a certain ratio δ of the cost of the optimal plan. For instance, taking the same plans as before, π_i and π_i^{opt} the optimal plan to reach g_i, and $\delta = 0.2$. The difference $|c(\pi_i) - c(\pi_i^{opt})| <= \delta \cdot c(\pi_i^{opt})$, and this for the three different cost. In other words, with $\delta = 0.2$, it means that the cost of the plan π_i should not differ from more than 20% of the cost of the plan π_i^{opt} in terms of time, minerals and gas. Then for all the remaining plans after pruning, we will choose the most likely to be followed by the agent using the mean squared error: $min(|c_t(\pi_i) - c_t(\pi_i^{opt})|^2 + |c_m(\pi_i) - c_m(\pi_i^{opt})|^2 + |c_g(\pi_i) - c_g(\pi_i^{opt})|^2)$, the plan minimizing this error will be the chosen one. The mean square error allows us to penalize more the plans that will differ from the optimal ones by having just one of the costs far from the optimal one. Instead, we will prefer plans with a slight difference for each cost as supplementary action often increases the plan's cost in more than just one way. Noted that for this heuristic, we could take a vector δ that will specify how far we could be from the optimal for each cost (30% for the cost in minerals and gas and only 15% for the cost in time, for instance) but for now we just use single value δ. For clarity we will call the first heuristic "three-dimensional heuristic" and the second one "mean-square heuristic".

Figure 3.4 is a Schema that represent our full plan recognition process, we start from a set of possible goals $\{g_1, ..., g_n\}$, our planner will generate a plan P_i and an optimal plan opt_i for each goal g_i. For each new observation we check if the different plans P_i satisfy this observation, in Figure 3.4 the plan P_n does not satisfy the observation O_i and has to be modified, it will be the concatenation of the plan P_n^1 that will reach an intermediate state including O_i from the initial state I and another plan P_n^2 that will reach g_n from this intermediate state. This modification of P_n will increase the distance $d(opt_n, P_n)$ in this example, and that will cause P_n and g_n to

be removed from the possible plans and goals during the pruning.

3.3 Evaluation

3.3.1 Methodology

Labeling Data:

We use a set of replays from iCCup users [1] available on the personal website of Gabriel Synnaeve. These replays are store in the format of log files where we can see each action that has been executed by the players, with the ID of the player performing the action, the location of the action, and at what time it has been performed. And this from the beginning to the end of the game, knowing that one replay corresponds to one game between two players. There are three civilizations that can be played in StarCraft: Protoss, Zergs, and Terrans, each with its own gameplay. Here we focus on inferring the opening strategy that the Protoss player will use, opening because we stop using the replay after that, the in-game time is superior to six minutes. Since only the replays, including a Protoss player, interest us, we used a data set of 445 replays of Protoss and Protoss (PvP) matches, 2408 Protoss and Terran (PvT) matches, and 2027 Protoss and Zergs (PvZ) matches, so a total of 4880 replays. We used clustering to group together the replays where the Protoss player used a specific strategy. Then for each cluster, the replays are labeled with the name of the strategy corresponding to the cluster. The labeling is performed on each matchup (a matchup is a match between two civilizations) distinctively as one strategy will not be executed in the same way depending on the matchup. The first labeling method that we used was the same as [44], expectation-maximization (EM) using the R package Mclust [58] and using the same features and label as them. Replays are labeled with the statistical appearance of key features with a semi-supervised method. The selection of the features along with the opening labels is the supervised part of the labeling method. The knowledge of the features and openings comes from expert play and the StarCraft liquipedia[2], a wiki of the game based on knowledge from thousands of players. The features are the times at which a specific action should be performed by the player if the player intends to follow a specific strategy. For example, the feature corresponding to the strategy that we saw earlier, "Fast DT rush" is the early apparition of Dark Templar units in the game. One replay can get several labels first, but then the most likely one is chosen among these labels.

[1] http://emotion.inrialpes.fr/people/synnaeve/iccup_users.7z
[2] https://liquipedia.net/starcraft/Category:Protoss_Build_Orders

We invite the reader to check the paper of Synnaeve and Bessière to see the method more in detail. Table 3.1 details the different labels and features corresponding. We also used

Table 3.1: Labels and features for the expectation-maximisation clustering method

Labels	**Features**
speedzeal	LegEnhancements, GroundWeapons+1
fast_dt	DarkTemplar
nony	Dragoon, SecondGateway, SingularityCharge
reaver_drop	Reaver, Shuttle
corsair	Corsair
templar	PsionicStorm, HighTemplar
two_gates	SecondGateway, Gateway, Zealot

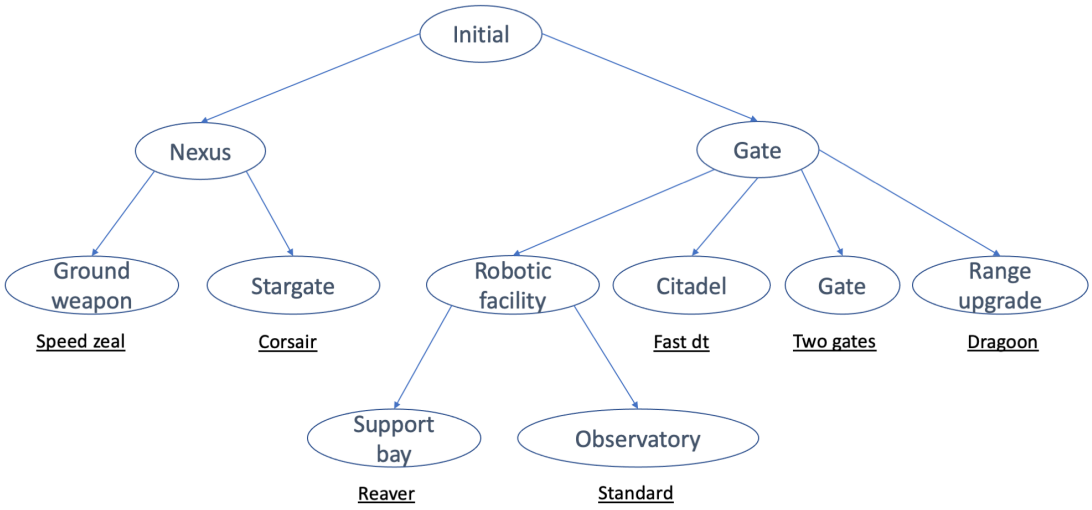


Figure 3.5: Graph for the rule set labeling method

rule set labeling like [42], but with different rules and labels. In this case, for each replay, we start at the root of the graph shown in Figure 3.5 and the first action appearing in the replay corresponding to one of the children nodes will make us going into this node. We repeat this process until reaching a leaf, and each leaf corresponds to a label. We did this to see if we can obtain more balanced data as with EM labeling, some labels are clearly predominant. It turned out that even with the rule set labeling, some labels are still predominant, although less than using EM. It is probably due to the fact that some strategies are more popular among

players than others. For both of the labeling methods, there were some replays that did not get any labels. We removed them from the dataset. Each replay is labeled with only one label, and each label corresponds to one and only one goal that we designed manually according to the label and expert knowledge from the website liquipedia [3]. The goals are the ones use as input for our algorithm. Finally, after filtering the data in order to get a balanced dataset for experiments, we only keep 800 replays for PvT replays, 800 for PvZ replays, and 400 for PvP replays. The different labels and their distribution in the dataset use for experiments can be seen in figure 3.6 for EM labeling and Figure 3.7 for the rule set labeling. After seing the poor distribution of the labels when using EM method for clustering, we decided to only use rule set labeling. We focus on opening strategies because they are the most relevant, after a certain time in the game strategy inference is less useful as the players have enough natural resources to do whatever they want, but in the first three to eight minutes using its natural resources in the smartest way is really important.

Metrics:

We use five different metrics to evaluate the performance of our method:
- The precision, which can be seen as the ability of the classifier not to label as positive a sample that is negative.

$$precision = \frac{Tp}{Tp + Fp} \tag{3.1}$$

- The predictive accuracy, it is the fraction of successfully predicted goals, and it is calculated as follows:

$$predictive\ accuracy = \frac{Tp + Tn}{Tp + Tn + Fn + Fp} \tag{3.2}$$

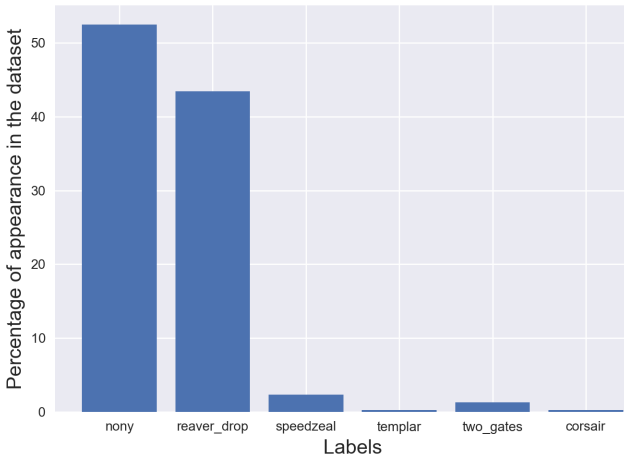
- The recall, which is the fraction of positive examples that are successfully classified as such by the algorithms, we calculated as follows:

$$recall = \frac{Tp}{Tp + Fn} \tag{3.3}$$

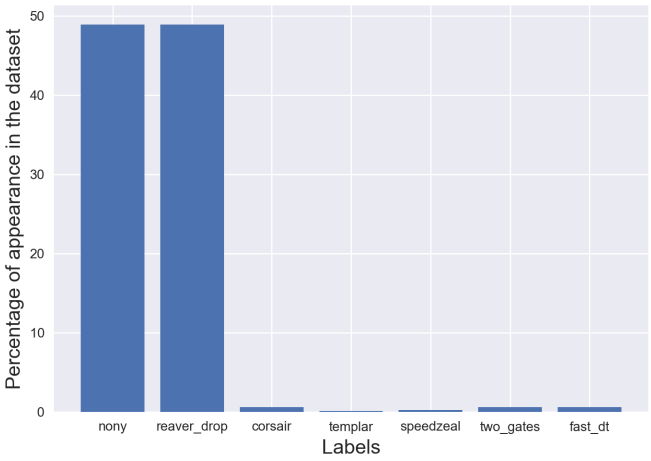
- The F-1 score, which can be interpreted as a weighted harmonic mean of the precision and recall, here is the formulas to compute the F-1 score:

$$F - 1\ score = \frac{2}{recall^{-1} + precision^{-1}} = 2 \cdot \frac{recall \cdot precision}{recall + precision} \tag{3.4}$$

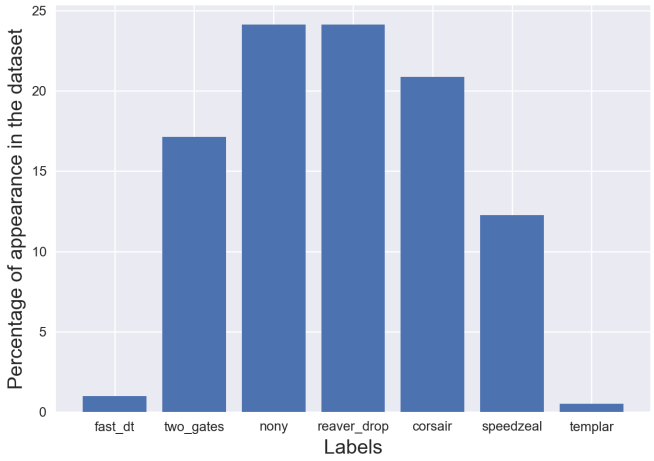
[3] https://liquipedia.net/starcraft/Category:Protoss_Build_Orders



(a) PvP

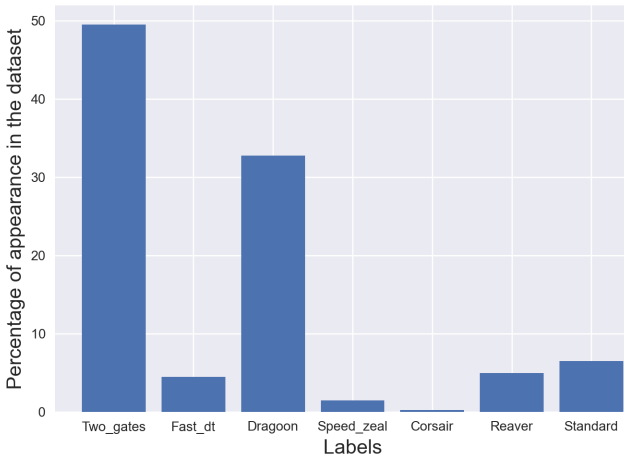


(b) PvT

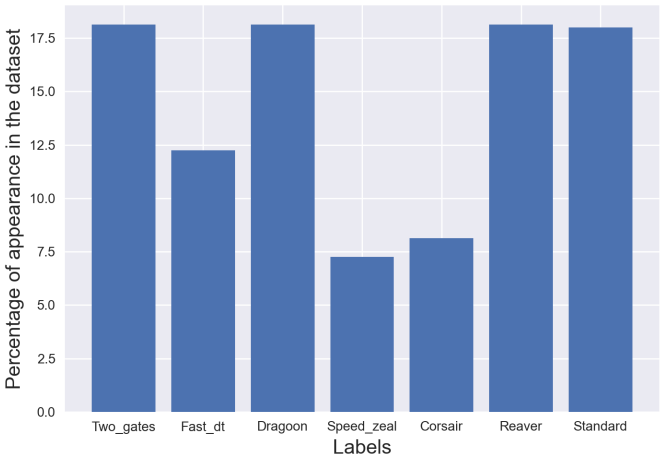


(c) PvZ

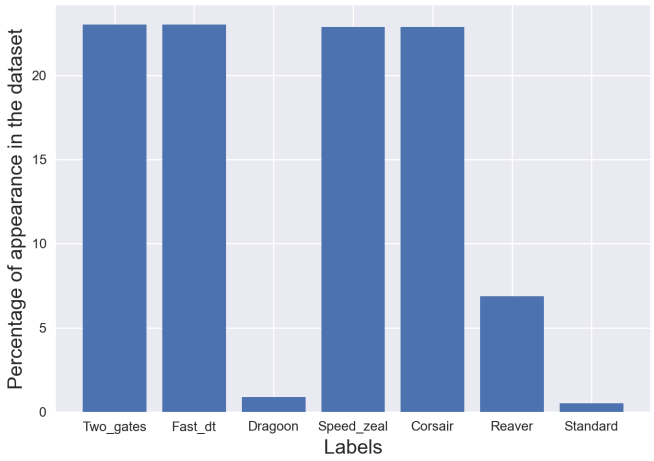
Figure 3.6: Labels distribution for the three matchup using EM



(a) PvP



(b) PvT



(c) PvZ

Figure 3.7: Labels distribution for the three matchup using rule set

- The number of missing buildings in the plan returned by our method compared to the actual plan of the Protoss player (we only count buildings and not units as units could have been made to adapt to a specific event in the game, where buildings are more relevant for the build order of the player).

Where Tp, Tn, Fp, and Fn are respectively, True Positive, True Negative, False Positive, and False Negative. The precision, recall, and F-1 score are computed for each label, and then a weighted average is computed according to the proportion of each label in the testing dataset.

We simulate the fog of war (uncertainty) by varying the number of actions observed, ranging from 80% of observed action until 20%. We could have directly use the actions that were only observed by the player playing against the Protoss player since they are recorded in the replay but it is harder to control the noise in this way. The observations that we get from the replay are all the buildings and units that the Protoss player built except worker units, as they are less relevant in our case. We get the observation as follows: first, we take the list of all the actions that the opponent player has done before six minutes in the game or until it gets its height first military units. We remove part of these actions according to the noise (if we use 30% noise, then we remove 30% of them). We simulate scouting, usually two or three times for one replay. The scouting is spread over the replay's duration, so for a replay that we stopped at six minutes, we will first get observation at two minutes, then at four minutes, and finally at six minutes. All the remaining actions (after applying the noise) that have been done before the first phase of scouting will be labeled with a timing corresponding to this phase of scouting, the ones that have been done between the first and second phase of scouting will get a label corresponding to the second phase of scouting, and the other ones will correspond to the timing of the last phase of scouting. Figure 3.8 illustrates this process, on the top is the list of actions that have been done by the player, in this case, we assume that we could only scout 50% of these actions (corresponding to 50% of noise) so half of the actions are removed randomly. Then, seeing that the sequence of actions lasts for 180 seconds (timing of the last action), scouting is performed at each third of the interval so, 60, 120, and 180 seconds. It means that for each action done before 60 seconds we observe its consequence during the first scouting phase so at 60 seconds and so on for the other actions. The list at the bottom of the figure is the actual list of observations that will be used for the plan recognition method. This process models the fact that because of the fog of war we do not know precisely when the actions have been done since the consequence is often observed later, except in some lucky case when the opponent player happens to start

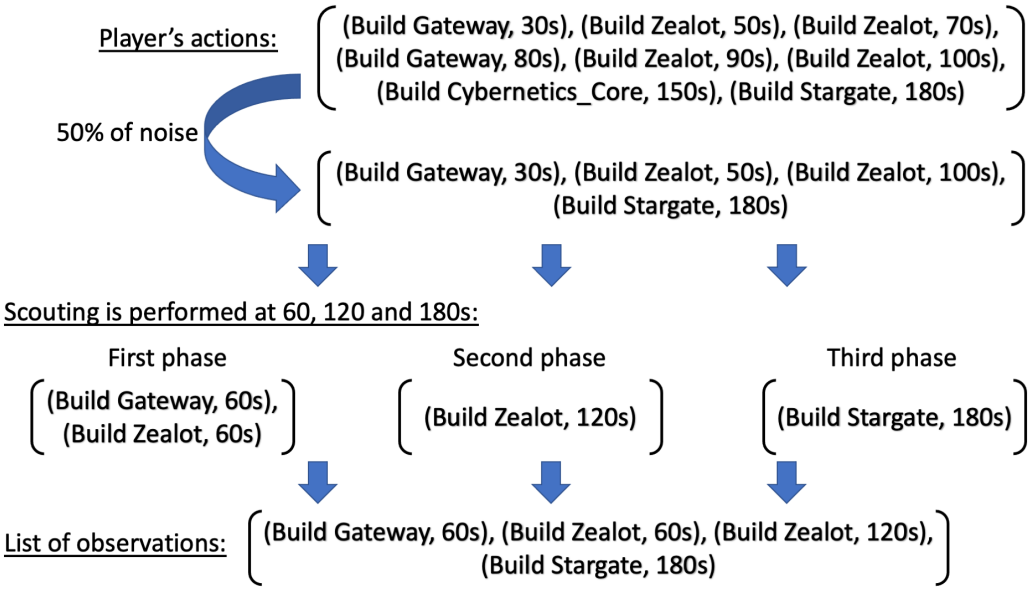


Figure 3.8: Simulation of scouting

an action in the line of sight of one of our units, then, we can see it directly. Note that, *Build Gateway* is a precondition of *Build Zealot* and knowing that *Build Zealot* takes 20 seconds, we could infer that the timing of the action *Build Gateway* was before 40 seconds instead of 60 seconds, we use such a process to extract complementary information from the observations. From the list of observations that we get, we infer all the actions that should have been done to fulfill the precondition of the observed actions with a timing corresponding to their respective duration and the order in which they should be executed. For instance, in figure 3.8 we have *Build Stargate* at 180 seconds in the final list of observations; this action takes 44 seconds to be performed and has *Build Core* as a precondition, so we will infer *Build Core* at 136 seconds. The timing of some observed actions can be refined too, as we said with *Build Gateway* that could take place at 40 seconds in the example instead of 60. By collecting the observations like this, we actually end up with a situation very close to what could happen in a real game of StarCraft as a player need to scout the opponent base to see what the opponent is doing and cannot be sure about when a building or a unit that it has scout has been built. The player can only perform inference in the same way as we do and use its past knowledge about the game to estimate the timing of the action. Where Weber and Mateas [42], and Synnaeve and Bessière [44], that conducted similar experiments on the same dataset did not use such a process and

instead just randomly remove some observation to simulate noise but kept the exact timing for the remaining ones. That could bias the result, as it is not quite what happens in a real game, and their methods are based on learning specific timing for each action from the replays, which indeed works better when keeping the exact timing of the observed action. We will see this in more detail later.

Finding the Good Parameters δ:

We first conducted some experiments to determine which value for δ we should use for both heuristics. To this end, we use a set of 200 PvT replays and 20% to 80% of noise. Here we just measured the predictive accuracy and execution time of our plan recognition algorithm. Then, we compute a ratio $\frac{accuracy}{exe_time}$, intuitively, the higher the value of the ratio is, the better, as it means the accuracy increases or the execution time decrease. This ratio makes sense since the predictive accuracy does not vary too much for the different values of δ, so we cannot get a case with 100% of predictive accuracy and 5 seconds of execution times that will give the same ratio as 50% of predictive accuracy and 2.5 seconds of execution time, but 100% of accuracy is indeed more interesting. The results have been normalized by taking the lowest value among the ratio as lower bound 0 and the highest one as upper bound 1. Of course, the normalization is different for both heuristics. Figure 3.9 shows the result for the three-dimensional heuristic. We can see that a $\delta = 0.3$ seems to provide the best results. In fact, for $\delta = 0.2$, the pruning is too strong, and thus the accuracy decreases too much, $\delta = 0.3$ provides almost the same accuracy as with higher values of δ but with a faster execution time. For $\delta \geqslant 0.5$ the results become similar most of the time. In the case of the mean-square heuristic, $\delta = 0.2$ provide the best ratio, as shown in figure 3.10. The predictive accuracy is a bit better with $\delta = 0.3$, but the increase in terms of execution time is more significant than this gain. Past $\delta = 0.3$ mainly the execution time increase and not the accuracy as it seems that the relevant plans are often kept, and only the useless ones are pruned.

Based on these results, we decided to take $\delta = 0.3$ for the three-dimensional heuristic and $\delta = 0.25$ for the mean-square heuristic for the following experiments. When using the three-dimensional heuristic, we also use a coefficient 3 to multiply the cost in times $c_t(\pi)_i$ and a coefficient of 2 to multiply the cost in gas $c_g(\pi)_i$ of each plan π_i in order to compensate for the very high cost in minerals of the plan.

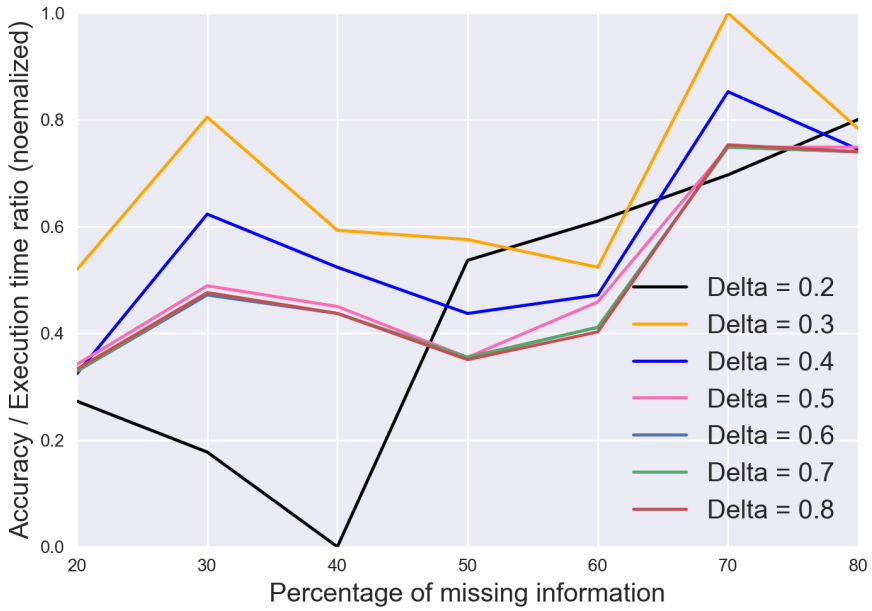


Figure 3.9: Comparison of the different values of delta for the three-dimensional heuristic

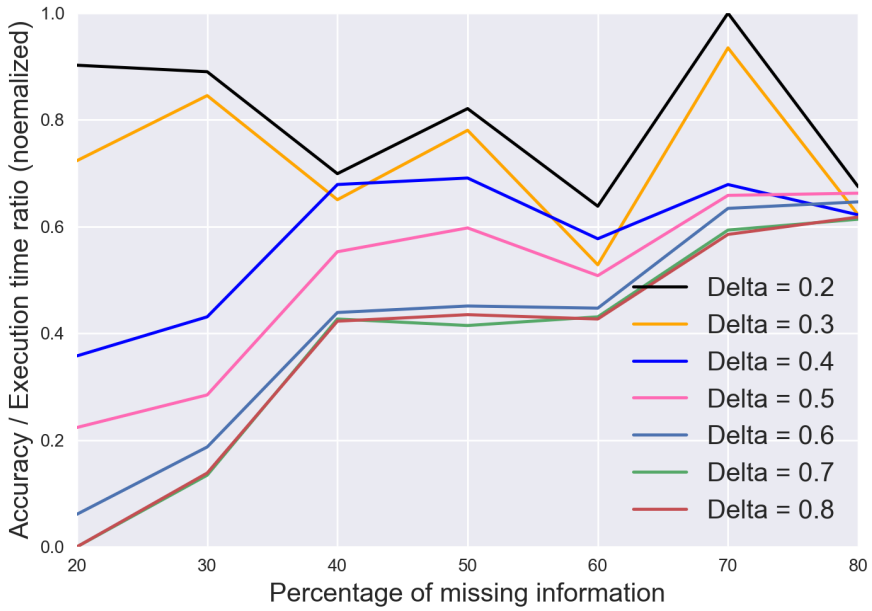


Figure 3.10: Comparison of the different values of delta for the mean-square heuristic

Algorithms for Comparison:

In their papers, Weber and Mateas [42] shown that k-NN algorithms provide the best predictive accuracy with noisy data, so we used this algorithm for comparison. We also use the Random forest algorithm to get another baseline. For both of these algorithms, we use the Python implementation from the library scikit-learn [59]. For k-NN, we use $k = 2$ neighbors to be robust against noise and still keep a good accuracy, and then the default setting proposed by scikit-learn. For Random forest, we used 28 trees, and fixed the size of the random subsets of features to consider when splitting a node to 11, plus, we did not put any limitation on the depth of the trees. These parameters for the two algorithms come from empirical experiments; we found out that these parameters provide the best results with the right balance between performance with high and low noise. These algorithms use a similar input as in [42], a feature vector with the timing of the first apparition of each unit and building during the replay and the second apparition of some specific buildings (like Gateway). We use 75% of the dataset for each matchup as training data and the 25% remaining as testing data to perform validation. However, as we said before, the feature vectors that are used to perform validation are built from noisy observations (from the scouting simulation) and thus do not contains the exact timing of the actions that have been observed but rather the timing that deduced with our inference process. Where in their paper Weber and Mateas do have the exact timing for the actions that they observed. For our method, we provide results with both heuristics introduced before in order to see if one performs better than the other.

3.3.2 Results

Because we randomly remove some observations from the replay in order to simulate noise, the predictions can be very good or very bad on the same replay file, depending on the observations that have been removed. Thus, to offset this stochasticity, we do five runs on each replay for the PvT and PvZ dataset and ten runs for the PvP dataset. Each data point on graphs resulting from the experiment corresponds to the average of one thousand predictions.

The first important thing to notice is that the predictive accuracy (that we simply call accuracy) has the same value as the recall for all matchups. So here, the recall that we compute is actually the weighted recall, which is a weighted average of the recall for each label, and it happens that it is the same value that the accuracy in our case. This is something that can happen in a multi-classification problem. Figure 3.11 shows the precision, recall, accuracy, and

F-1 score for the PvT matchup with the rule-set labeling. It is actually the matchup where our algorithm performs the closest to k-NN for almost all the metrics except the precision where K-NN is a little better. We can see that Random forest does perform worse than the other algorithms on this specific matchup. For this matchup, the different metrics lower when the noise increases, and this in the same way for all the algorithms. It is obviously a hard prediction problem as none of the algorithms reach 50% of predictive accuracy even when there are not that many missing observations. The PvT dataset is also the most balanced one in terms of strategies. This puts all algorithms on an equal footing as the prior distribution of the different strategies does not have a relevant impact on the prediction in such a case. Both the three-dimensional heuristic and the mean-square heuristic give similar results, with a slight edge for the three-dimensional heuristic.

Figure 3.12 shows the precision, recall, accuracy, and F-1 score for the PvZ matchup with the rule-set labeling. For this matchup, we can see that Random forest performs better than the other algorithms. When we reach more than 70% of noise, though, our algorithm does perform in a similar fashion as Random forest. This dataset is less balanced than the previous one, and we could see in figure 3.7 that four strategies really dominate the dataset in terms of distribution (Two_gates, Fast_dt, Speed_Zeal, and Corsair). It happens that, for this specific matchup, the plans that players will follow to achieve their strategy are often similar in terms of action for the strategies Two_gates, Fast_dt, and Corsair. What will change is the execution order of these actions, and this is something our algorithm struggle with when Random forest and k-NN have less problem since they are trained based on the timing of the actions. So for this matchup, Random forest and k-NN do outperform our method.

As for the PvP matchup, displayed in Figure 3.13, we observe that when the data are not well balanced, our algorithm does not perform as well as Random forest and k-NN in terms of accuracy, though this changes when we reach 50% of noise, in fact, the accuracy of our algorithm increases with the noise. This is not shown on the graph, but when we take the two best predictions of our algorithm instead of only the first one, the accuracy starts around 70% with low noise and then slowly decreases to 60% when we reach the maximum amount of noise. It means that we observe this strange result of accuracy increasing with noise because our algorithm hesitates between two goals that probably have a very similar plan to reach each of them, and our algorithm will often pick up the wrong goal. Then, when the noise increases, our algorithm will fail less often at choosing the right goal between these two candidates. The

confusion most likely appears between the goals Two_gates and Speed_Zeal since they are very similar in terms of resources to achieve for the player in this matchup. The precision, though, follows a more usual pattern and decreases with noise for our algorithm, and our algorithm actually achieves 50% more precision on average than k-NN or Random forest. When looking at the F1-score, our algorithm does perform better than k-NN and Random forest as soon as we reach more than 30% of noise, which is expected a the F1-score is a kind of average of the recall and precision.

An important fact is that the PvT matches are the ones with the most "pure" strategies. By pure, we mean that the player playing this game almost only performed actions related to the strategy that they were following, where for the other matches, like PvZ or PvP, the players often had to perform actions to react to the behavior of the opponent and these kinds of actions are not part of their strategy most of the time. It could disturb our algorithm by leading it to confuse the strategy the player is following with another strategy that includes these "parasite" actions. It is a reason why the accuracy of our algorithm increases with the noise for the PvZ and PvP matchup. As the noise increases, the number of parasite actions decreases, and our algorithm makes fewer mispredictions. Another factor that could explain these strange curves is pruning. In fact, when the noise is low, and there are a high amount of parasite actions, our algorithm often prunes some plans and goals too early as these plans and goals might be more consistent with the observations later on. Sometimes, the actual plan and goal are prune because of parasite actions, and indeed, the prediction will be wrong. However, it happens less often as the noise increases, thus decreasing the number of parasite actions. Even though the percentage of these parasite actions will stay the same among the observations since we randomly remove actions from the list of observations. Since we get fewer observations, the absolute number of these parasite actions will decrease in the plan inferred by our algorithm, and we are less likely to prune the actual plan and goal before the final prediction. It does not happen for the PvT matches, in fact, in these matches, it is often the Protoss player that takes the first initiative, so the Protoss player is often left free to follow its strategy without interruption, where for PvP and PvZ the other player might take the initiative more often and thus forcing the Protoss player to react and perform actions not related to its initial strategy. In addition, k-NN and Random forest have an edge when the labels are unbalanced. These two methods naturally handle priors about label distribution since they learn it from the training data. It gives an advantage to these two methods in the PvZ and PvP matchup. Our method

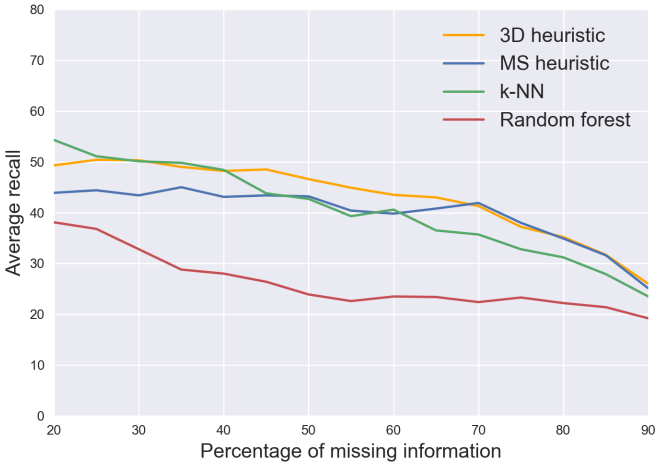
does not take this into account as we believe that prior are suggest to change according to many factors, especially in games where there are some trends in some specific strategies sometimes. By including such a prior, the result in the PvZ and Pvp matchup would surely be better than currently for our method. Moreover, it could also fix this strange phenomenon of having an increasing accuracy with noise for the PvP matchup. More details on the results can be seen in tables 3.2, 3.3 and 3.4, since the accuracy and the recall are shown to be equal in our experiments, we only put the accuracy in the tables.

Table 3.2: Average precision, accuracy, and F-1 score for the PvT matchup using rule-set labeling

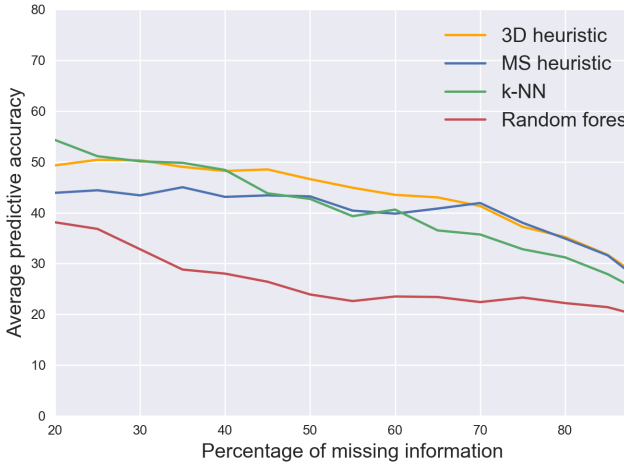
	Precision				***Recall***				***F-1 score***			
Noise	3D	MS	k-NN	Random forest	3D	MS	k-NN	Random forest	3D	MS	k-NN	Random forest
20.0	42.3	40.9	58.3	54.4	49.3	43.9	54.3	38.1	43.0	39.1	51.2	39.6
25.0	40.2	41.2	55.9	54.1	50.4	44.4	51.1	36.8	43.2	39.4	48.2	38.8
30.0	42.0	42.7	55.8	54.1	50.3	43.4	50.1	32.8	44.0	40.1	46.9	35.1
35.0	37.5	38.1	56.7	56.0	49.0	45.0	49.8	28.8	42.0	40.7	46.8	30.7
40.0	43.5	41.1	54.8	51.5	48.2	43.1	48.4	28.0	41.7	39.0	45.1	29.2
45.0	41.1	38.6	50.9	50.0	48.5	43.4	43.8	26.4	42.2	39.6	39.9	27.3
50.0	41.7	39.8	49.2	45.1	46.6	43.2	42.7	23.9	40.6	39.4	38.0	23.2
55.0	36.5	36.1	46.6	50.8	44.9	40.4	39.3	22.6	39.4	37.2	34.3	22.4
60.0	35.5	35.0	51.1	57.4	43.5	39.8	40.6	23.5	37.8	36.4	35.6	23.2
65.0	35.7	35.3	45.7	55.1	43.0	40.8	36.5	23.4	37.5	36.9	31.1	21.6
70.0	37.2	37.8	50.4	46.9	41.3	41.9	35.7	22.4	36.3	37.9	30.7	18.2
75.0	36.7	37.4	43.4	36.1	37.2	38.0	32.8	23.3	32.8	34.0	26.7	17.8
80.0	35.0	34.6	39.7	42.8	35.2	34.9	31.2	22.2	30.2	30.5	24.0	16.0
85.0	35.3	34.7	42.5	51.3	31.7	31.6	27.9	21.4	26.1	26.6	20.1	13.3
90.0	37.0	35.5	32.0	32.9	26.0	25.1	23.5	19.2	18.3	18.6	13.5	10.7

Most of the results show that our method performs better than Random forest or k-NN when the noise reaches 50% or more, otherwise Random forest and k-NN perform a bit better. The fact that the replays are balanced also seems to benefit our method that struggles when the replays are not balanced because our method does not use any prior about the opening strategies distribution. Our method is more sensitive to the others to the appearance of actions that are not directly related to the current strategy of the player. However, on the other hand, it is also less sensitive to the timing of actions than usual classifiers like Random forest and k-NN, this is why the performance of our method is better than the others when the noise is important. Also, an essential factor to consider is that our method does not need that much training data. We could directly learn the possible goals from expert knowledge, for these

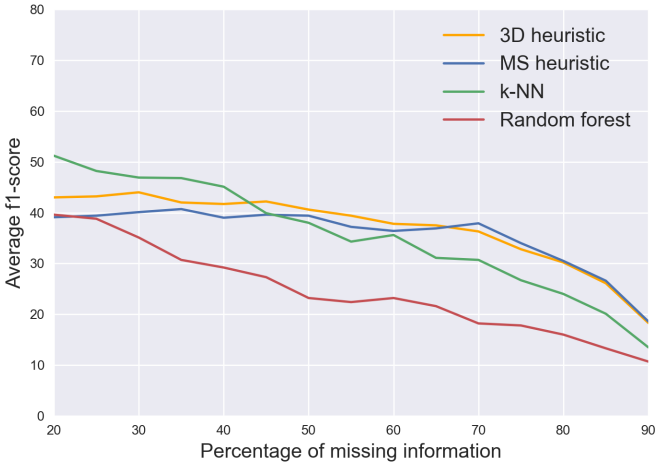
(a) Precision



(b) Recall

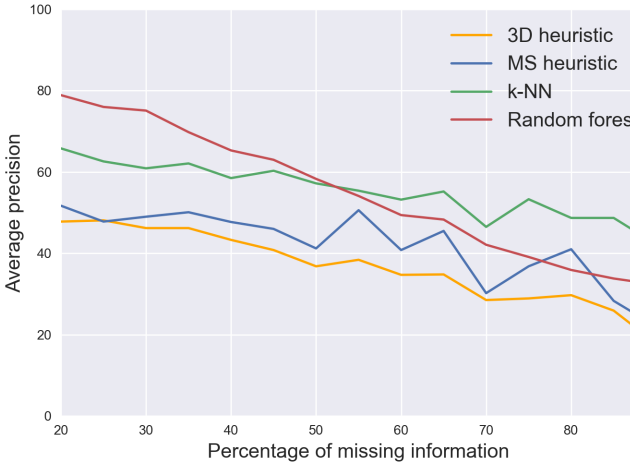


(c) F-1 score

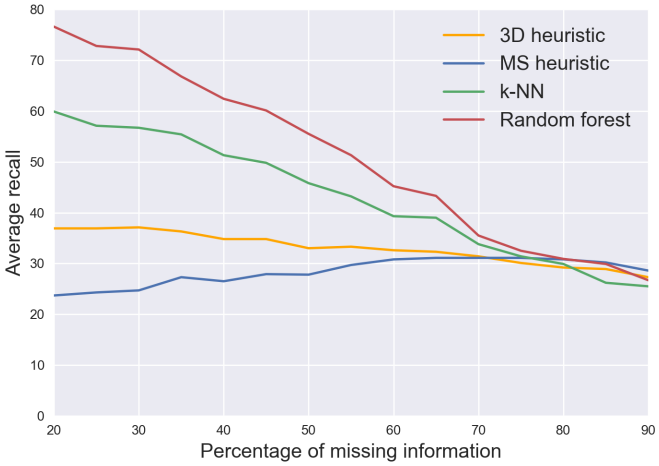


(d) F-1 score

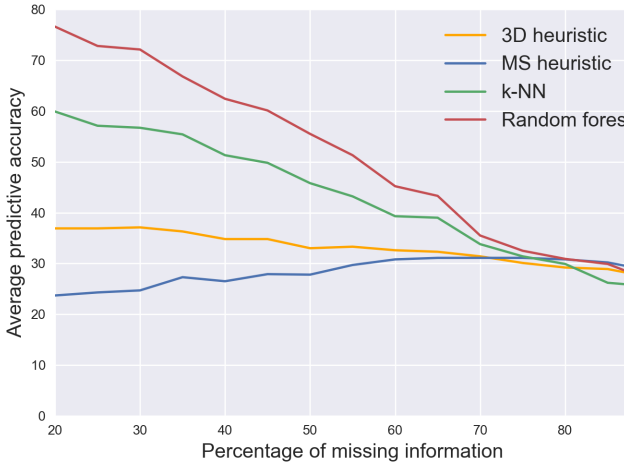
Figure 3.11: Average precision, recall, accuracy, and F-1 score for the PvT replays, at 5 to 6 minutes in-game time, using rule set labeling.



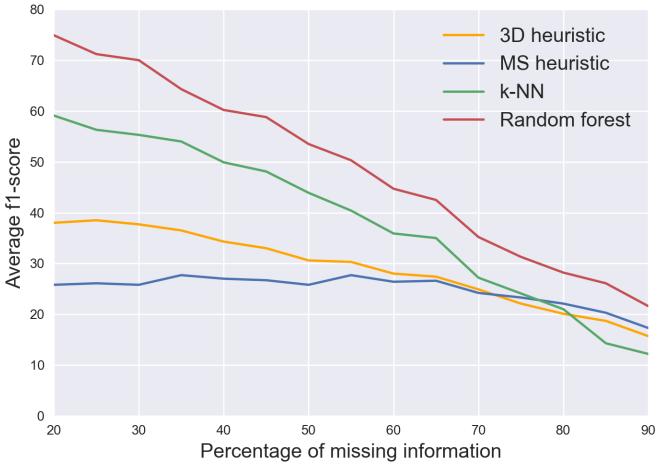
(a) Precision



(b) Recall



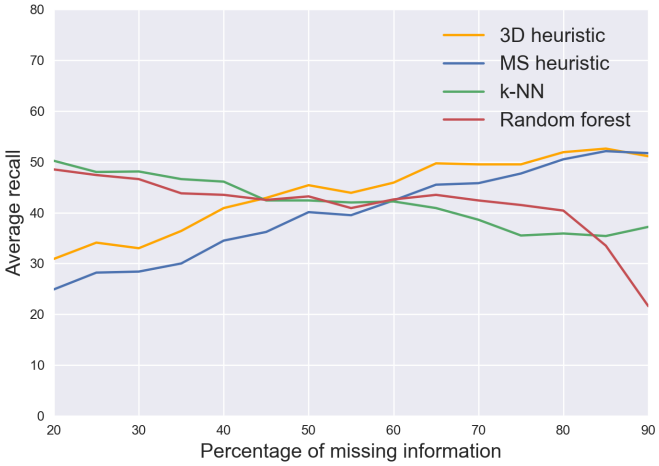
(c) F-1 score



(d) F-1 score

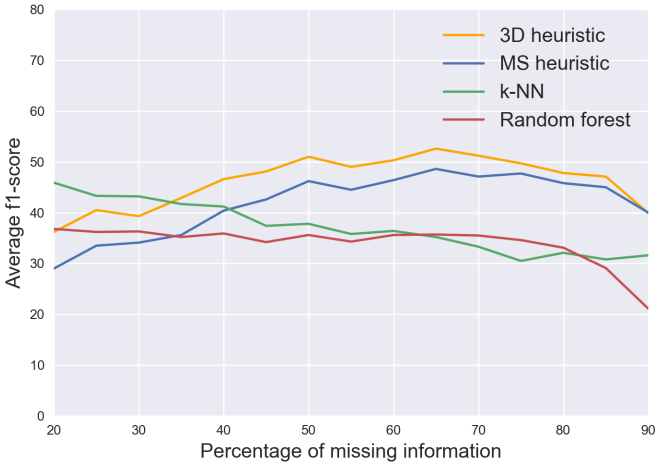
Figure 3.12: Average precision, recall, accuracy, and F-1 score for the PvZ replays, at 5 to 6 minutes in-game time, using rule set labeling.

(a) Precision



(b) Recall

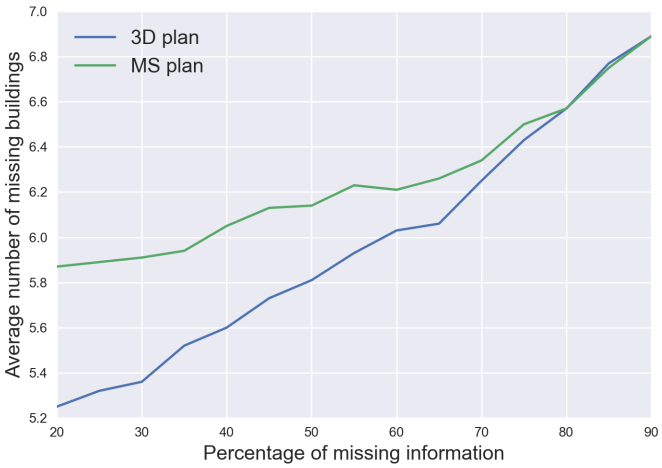
(c) F-1 score



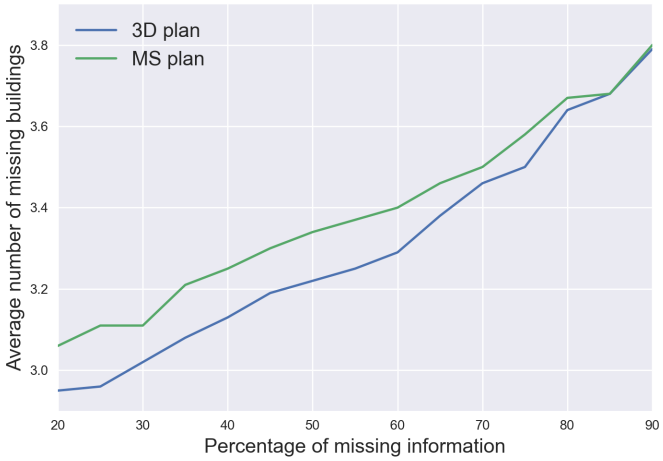
(d) F-1 score

Figure 3.13: Average precision, recall, accuracy, and F-1 score for the PvP replays, at 5 to 6 minutes in-game time, using rule set labeling.

(a) PvT



(b) PvZ



(c) PvP

Figure 3.14: Number of missing buildings in the most likely plan that we inferred compared to the actual plan that the Protoss player followed, at 5 to 6 mins in-game time, using rule set labeling.

Table 3.3: Average precision, accuracy, and F-1 score for the PvZ matchup using rule-set labeling

	Precision				***Recall***				***F-1 score***			
Noise	3D	MS	k-NN	Random forest	3D	MS	k-NN	Random forest	3D	MS	k-NN	Random forest
20.0	47.8	51.7	65.8	78.9	36.9	23.7	59.9	76.6	38.0	25.8	59.1	74.9
25.0	48.1	47.8	62.6	76.0	36.9	24.3	57.1	72.8	38.5	26.1	56.3	71.2
30.0	46.2	49.0	60.9	75.1	37.1	24.7	56.7	72.1	37.7	25.8	55.3	70.0
35.0	46.2	50.1	62.1	69.8	36.3	27.3	55.4	66.8	36.5	27.7	54.0	64.3
40.0	43.3	47.7	58.5	65.3	34.8	26.5	51.3	62.4	34.3	27.0	49.9	60.2
45.0	40.8	46.0	60.3	63.0	34.8	27.9	49.8	60.1	33.0	26.7	48.1	58.8
50.0	36.8	41.2	57.2	58.3	33.0	27.8	45.8	55.5	30.6	25.8	43.9	53.5
55.0	38.4	50.6	55.4	54.1	33.3	29.7	43.2	51.3	30.3	27.7	40.4	50.3
60.0	34.7	40.8	53.2	49.4	32.6	30.8	39.3	45.2	28.0	26.4	35.9	44.7
65.0	34.8	45.5	55.2	48.3	32.3	31.1	39.0	43.3	27.4	26.6	35.0	42.5
70.0	28.5	30.2	46.5	42.1	31.4	31.1	33.8	35.5	24.9	24.2	27.2	35.2
75.0	28.9	36.8	53.3	39.1	30.1	31.1	31.4	32.5	22.1	23.3	24.1	31.3
80.0	29.7	41.0	48.7	35.9	29.2	30.8	29.9	30.9	20.1	22.1	21.0	28.2
85.0	25.9	28.3	48.7	33.8	28.9	30.2	26.2	29.9	18.7	20.3	14.3	26.1
90.0	17.8	22.0	42.2	32.3	27.3	28.6	25.5	26.7	15.7	17.3	12.2	21.6

experiments, we did both. We design each possible goal by computing the average buildings and units that the player has after 5 to 6 minutes for each label, and we refine it with expert knowledge. However, we can do without the training data, we just need to know the main units and building that the player is aiming for and estimate how many. The performance of our method could decrease a bit but will still be far better than Random forest or k-NN without training data. There is no significant difference between the three-dimensional heuristic and the mean-square heuristic, though the three-dimensional one does perform better most of the time. This slight difference is due to the fact that for some of the goals (especially Two_gates), the cost of the plan in terms of gas is very low. Suppose there is an observation of an action that increases this cost. In that case, even if it is just one action, it might multiply the cost by two, so the mean-square heuristic will prune this possible goal where it will not be prune by the three-dimensional heuristic. However, we needed training data to get the good parameters for the three-dimensional heuristic as the plan's cost in terms of minerals and gas is not increasing linearly with time, as stated before. So we needed the training data to find the good factors to multiply each cost and make them equivalent. Nevertheless, that is not the case if we use the mean-square heuristic, we developed this heuristic to offset this problem.

For the plan inference, we can see the result in Figures 3.14 for the case of PvT there are

Table 3.4: Average precision, accuracy, and F-1 score for the PvP matchup using rule-set labeling

	Precision				***Recall***				***F-1 score***			
Noise	3D	MS	k-NN	Random forest	3D	MS	k-NN	Random forest	3D	MS	k-NN	Random forest
20.0	61.3	59.6	44.2	34.4	30.9	24.9	50.2	48.5	36.2	29.0	45.9	36.8
25.0	64.4	62.9	41.4	33.2	34.1	28.2	48.0	47.4	40.5	33.5	43.3	36.2
30.0	61.8	62.6	41.0	35.4	33.0	28.4	48.1	46.6	39.3	34.1	43.2	36.3
35.0	62.6	62.6	39.6	31.0	36.4	30.0	46.6	43.8	42.9	35.6	41.7	35.2
40.0	63.3	62.4	39.6	36.9	40.9	34.5	46.1	43.5	46.6	40.4	41.2	35.9
45.0	60.2	62.4	35.0	32.5	42.9	36.2	42.4	42.5	48.1	42.6	37.4	34.2
50.0	62.3	64.4	36.4	35.4	45.4	40.1	42.4	43.2	51.0	46.2	37.8	35.6
55.0	58.9	59.7	34.0	29.8	43.9	39.5	42.0	40.9	49.0	44.5	35.8	34.3
60.0	58.1	58.3	33.4	37.0	45.9	42.4	42.2	42.6	50.3	46.4	36.4	35.6
65.0	57.3	57.7	32.1	36.9	49.7	45.5	40.9	43.5	52.6	48.6	35.2	35.7
70.0	54.2	53.6	33.2	45.9	49.5	45.8	38.6	42.4	51.2	47.1	33.3	35.5
75.0	51.6	52.2	28.1	36.9	49.5	47.7	35.5	41.5	49.7	47.7	30.5	34.6
80.0	50.9	50.6	30.6	36.8	51.9	50.5	35.9	40.4	47.8	45.8	32.1	33.1
85.0	52.8	55.7	30.0	41.4	52.6	52.1	35.4	33.5	47.1	45.0	30.8	29.1
90.0	47.6	56.8	31.2	21.9	51.1	51.7	37.2	21.6	39.9	40.0	31.6	21.1

Table 3.5: Average new plans computation time(*Average time*), whole plan recognition method average computation time (*Average total time*), minimum observed computation when computing new plans (*Min time*), maximum observed computation time when computing new plans (*Max time*), in seconds for all of them.

Average time	***Average total time***	***Min time***	***Max time***
1.882	7.260	1.411	6.116

on average less than 3.6 missing buildings in the plan generate by our method compared to the actual one of the Protoss player until that we reach 60% of noise. Then we miss up to 4.2 buildings when reaching 90% of noise. Knowing that the length of the plan followed by the player is around 11.25 buildings on average, it means that we infer between 68% in the best case to 63% of the plan in the worst case. In the case of PvZ, we miss 5.2 buildings with low noise and up to 6.9 when the noise reaches 90%. For this matchup, the Protoss player follows plans composed of 13.8 buildings on average, so we infer from 63 to 50% of the plan depending on the noise with the three-dimensional heuristic. The mean-square heuristic provided slightly worse results for this matchup, especially when the noise is under 70%. For PvP, the plans that the player follows are shorter and composed of 10.5 buildings on average. In this case, we miss

between 2.9 and 3.8 buildings, which means that we infer between 73 and 64% of the player's plan. In most cases, the buildings that we are missing are not key features of the player's strategy but rather some building built to react to an action of the opponent or buildings that have been build multiple times when our planner estimated that it was too much. Of course, this is just an average among all the goal predictions; it means that when our method correctly infer the goal of the observed player, the plan that we infer is also much closer to the actual one than what we can see on the graph. On the other hand, when the goal prediction is incorrect, the inferred plan is farther from the actual one than the average shown on the graph.

Table 3.5 shows the computation time of our method when we compute the first plans at the beginning, or when we update the current plans in order to satisfy the observation, this operation takes on average 1.882 seconds, the whole method from the generation of the first plans to the last observation checking takes on average 7.260 seconds (sum of the computation time of all the pruning/planning operation during a run). The longest observed computation time for one pruning/planning operation was of 6.116 seconds when the shortest was of 1.411 seconds. The computations were done with a Ryzen 7 2700X CPU with 8 cores running at 3.7GHz. Our performances could be optimized a lot, also we were using Python but C++ will definitely decrease the computation time, this computation time is still acceptable for real-time use. Optimizing the implementation and using a faster programming language could easily take less than a second for our method to return the result when we ask for the predicted plan. For usage in bot competition, when the computation time is limited to intervals of 100ms, it can be hard to divide the computation like this, but not impossible. However, for use in games for the purpose of playing against humans, then there is not such a limitation and one second is pretty similar to what a human could achieve when performing strategy inference after collecting enough clues. Furthermore, RTS games do not require immediate reactions like other games. Most of the actions take some time to perform, so our method has a very viable execution time as the actions to take after performing the inference (building the good buildings and units) take much more time.

3.4 Possible Improvements and Discussion

3.4.1 Multi-Horizons Planning

With the planner we are currently using, it might be complicated to try to predict the player's plan at ten minutes horizon right from the start of the game. The computation time could

be a bit too high for online use. So instead, what we plan to do, is to make predictions at multiple horizons, little by little. For a prediction at ten minutes horizon, for example, we can predict first the plan of the player for the five first minutes, then, when we reach five in-game minutes, we can make a prediction for the five next minutes, that will correspond to an indirect ten minutes prediction from the beginning of the game. In fact, it makes sense to follow such a process, as players often follow a similar process when they plan their strategy during the game. They often plan a first "opening strategy" (the one we used for the experiments) that they aim to achieve at four to six in-game minutes, and then, if the game does not end here because one of the two players was victorious, they will plan another strategy that usually depends on the one they used as an opening. Each opening could lead to some more advanced strategy at a later stage in the game, with some advanced strategy being easier to achieve with a specific opening strategy before. We could select the two or three most likely plans to be followed by the player that our method predicts at five minutes horizon. Use the goal states of each of these plans as new initial states for our planner, and for the new possible goals, use the advanced strategy that players usually try to achieve as a follow-up of the corresponding opening strategy. This would indirectly include prior probabilities in our model as the plan that we infer at ten minutes horizon will also depend on the ones that we inferred at five minutes horizon.

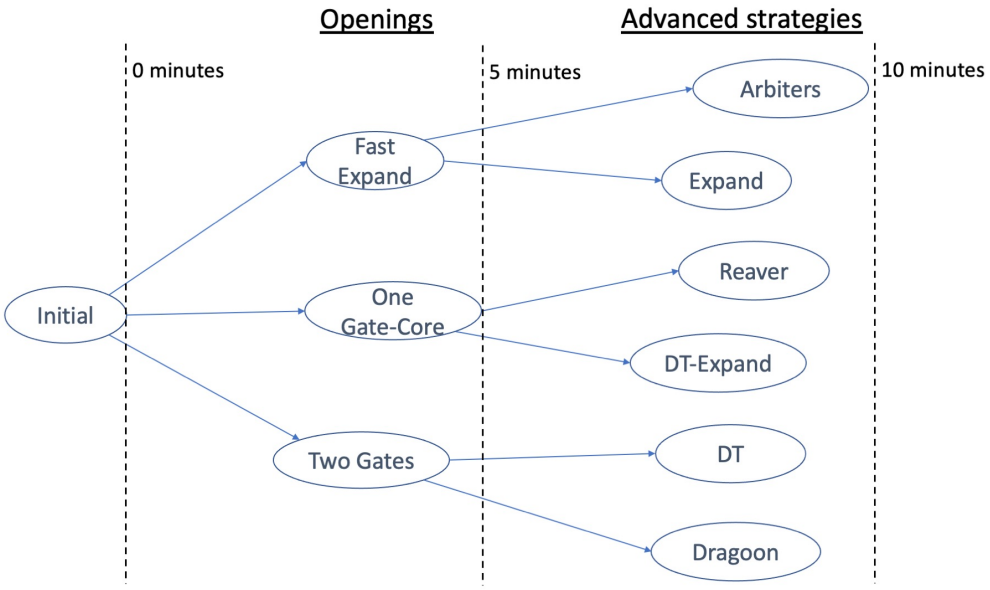


Figure 3.15: One possible strategy tree for the Protoss player in PvT matchup

Figure 3.15 illustrates this. It is a strategy tree of some possible openings for the Protoss player when playing against a Terran player. We can see three different openings that the player will usually achieve between four to five minutes, "Fast Expand", "One Gate-Core", and "Two Gates". Moreover, as explained before, each of them leads to different advanced strategies that the player usually achieves between eight to ten minutes. So if we take an example related to Figure 3.15, we will first perform plan recognition at five minutes horizon, say that our method returns "Fast Expand" as the most likely goal and "One Gate-Core" as the second most likely one. Next, we discard "Two Gates", and now, to perform plan recognition at ten minutes horizon, we take two initial states, one that corresponds to the state after performing all the actions in the plan that was inferred for the goal "Fast Expand", and the other one in a similar fashion but for the goal "One Gate-Core". The new possible goals will be "Arbiters", "Expand", "Reaver", and "DT-Expand". Indeed, "Arbiters", for example, could still be a possible goal for the player when using the "Two Gates" opening in theory, but it is such a not optimal choice that it is really unlikely that the player would do that (except to mislead its opponent but then it is a very hard problem to infer this), so we represent the strategy tree with the most likely transitions for the player.

3.4.2 New Planner and Heuristic

The efficiency of the method strongly relies on the planner that we use. Here the planner could be improved or even change for another one, especially the scheduling part could be improved to get better plans as in many cases the plan is not optimal. The approach of Churchill and Buro [60] could be a good candidate, but we have to keep a reasonable computing time and to keep in mind that the planner should match with the player skills in order to make reasonable inferences. We could also try another heuristic for the pruning, and the goal selection, especially the landmarks heuristic developed by Pereira *et al.* [22] looks rather promising if we can correctly extract the landmarks of each possible goals. That should be possible when having enough expert knowledge about the game, like StarCraft's case.

3.4.3 Discussion

We introduced a method for plan recognition in RTS games that has proven to be very robust to noise while still providing satisfying results. The method's execution time is good enough to allow real-time use, and the fact that this method does not rely on training with replay files makes it easy to adapt to any RTS games following similar rules as StarCraft. It is pretty

robust to noise, and although the precision , accuracy, and recall are not perfect, it does predict often enough the opening of the opponent player. We need to do more experiments, especially to compare it to the approach of Synnaeve and Bessiere [44] that has a very good predictive accuracy. However, they did not use the same scouting approach to get the observation, so the results may be different using our experiment parameters as Bayesian models are pretty sensitive to the timing of observations. Knowing that the efficiency of our method strongly relies on the planner that we use, and we could definitely obtain better accuracy with a better planner but generate good plans in a short time for RTS games stay a challenging problem. The more precise the possible goals are, the better the method will work as the different plans will be more refine and different from each other, but if we want really precise goals, then we may need some analysis on the replay, and we lose one good point compared to the existing approaches which is that we do not need to do data mining. To conclude this study, our approach allows strategy inference in RTS games, is robust to noise, and offers fair performances. We hope that it could be used to improve AIs in this kind of game and make them more interesting to play against.

Chapter 4

Goal Recognition

The second part of the thesis starts here. In this part, we will study a method that has been developed to perform goal recognition. The application field is wider here and not only related to games. Current approaches in goal recognition have not yet tried to apply concept learning to a propositional logic formalism. We developed a method for inferring an agent's possible goal by observing this agent in series of successful attempts to reach its goal and using concept learning on these observations. We propose an algorithm, LFST (Learning From Successful Traces), to produce concise hypotheses about the agent's goal. We show that, if such a goal exists, our algorithm always provides a possible goal for the agent, and we evaluate the performance of our algorithm in different settings. We compare it to another concept learning algorithm that uses a formalism close to ours, and we obtain better results at producing the hypotheses with our algorithm. We introduce a way to use assumptions about the agent's behavior and the dynamics of the environment, thus improving the agent's goal deduction by optimizing the potential goals' search space.

As we saw in chapter 1, section 1.1, much previous work has attempted solving the plan recognition problem using a set of pre-determined goals for the agent and guessing which one is the most likely. It is the case in the work of Lang for example [61]. Often, they assume that this pre-determined set of goals is given, but it is usually not the case. Even approaches that remove the use of plan libraries like Ramirez and Geffner [62, 15], or deep learning like Min *et al.* [46] still assume that they know the set of possible goals of the agent and need it to work. This assumption, made in previous work and which we do not take for granted, motivated our work. What if we do not have information about the agent and its preferences? If the agent evolves in a complex environment, it could potentially have many goals. We wanted to see to what

extent it was possible to generate this set of potential goals for the agent. Furthermore, this, using simple but effective tools, which are propositional logic and concept learning. Besides, there are only a few approaches using propositional logic for goal recognition. This lack of literature has reinforced our interest in this formalism. Hong [63] uses propositional logic, but he combines this to a graph representation, and like for previous approaches, he already knows the agent's goals.

Section 4.1 will explain how we formalize the concept learning task of inferring an agent's goals from observations of successful scenarios in section 4.1.1 and detail our algorithm LFST in section 4.1.2. In section 4.2 we present an experimental study of the proposed algorithm where we compare it to two other algorithms, [64] and [65] (an improved version of [66]) and show that we often get better results with our algorithm. We will explain how to use assumptions about the environment and the agent decision process to improve the results of LFST in section 4.3, section 4.3.1. We will detail two ways to extract this information from the data and a way to consider the agents' preferences regarding their goals in section 4.3.2.

4.1 Inferring an Agent's Goals Using Concept Learning

4.1.1 Problem Formalization

We want to infer an agent's possible goals by observing a series of successful attempts by the agent to achieve them. We, therefore, assume a training process in different scenarios during which we observe the agent performing actions until it achieves one of its goals. We will model these observed scenarios as a set of traces describing the succession of states of the environment as well as the actions of the agent. We consider environments without exogenous events and with discrete-time: an action performed by the agent always leads to a state change. The state of the environment is modeled as a series of attributes with discrete values: N variables var_i for $i \in \{1, \dots, N\}$ taking values in domains $D_j = \{val_1^i, \dots, val_{N_i}^i\}$. An atomic representation is construct by converting all the couples $var_i = val_j^i$ into atoms $var_i^{val_j^i}$, we denote by $\mathcal{L}$ this set of atoms. We define a *state* S as a set of atoms $var_i^{val_j^i}$ from $\mathcal{L}$ with each var_i only appearing once, and we define a *trace* as a sequence of couples (S_i, a_i) with $S_i \subset \mathcal{L}$ being a state and a_i an action (taken from $\mathcal{A}$, a finite set of actions). Traces can be of different lengths as they rely on the number of actions performed by the agent to reach its goal. We will not go into the details of the environment's dynamics, but they can be abstracted away using a function *next* from $2^{\mathcal{L}} \times \mathcal{A}$ to $2^{2^{\mathcal{L}}}$ which will, given a state S and an action a provides the set

of potentially reachable states from S by performing a. If the environment is deterministic, $next(S, a)$ will correspond to a single state. Since they come from observations, the traces are assumed to follow this dynamic, meaning that if the index i is not the last one of the trace, we have: $S_{i+1} \in next(S_i, a_i)$. We define successful traces using a special action `success` that has no effects ($\forall S, next(S, \texttt{success}) = \{S\}$), performed by the agent whenever it reaches its goal. Thus, a successful trace is a trace $T = (S_0, a_0), \ldots, (S_k, a_k)$ with $a_k =$`success` and $a_i \neq$`success` for $i < k$. A successful trace is therefore a trace ending in the first state where the agent achieves its goal. Given a trace $T = (S_0, a_0), \ldots, (S_k, a_k)$, we denote the last state S_k of this trace by $endS(T)$ and the set of intermediate states $\{S_0, \ldots, S_{k-1}\}$ by $intS(T)$. Our problem takes as input a set of successful traces: $\Sigma = \{T_0, \ldots, T_l\}$ and we seek to infer from it a hypothesis about the agent's goals expressed as a propositional formula over $\mathcal{L}$, this hypothesis must be satisfied by specific states (when interpreting states as a conjunction of atoms) if and only if a goal is achieved in these states. It is assumed here that the goal does not depend on the way it is reached, only the final state matters. The agent only needs to reach a state satisfying its goal. In order to cover several goals, we want the hypothesis written as a DNF (Disjunctive Normal Form). More exactly, we want a hypothesis of the form $H = C_0 \vee C_1 \vee ... \vee C_m$ with each $C_i = x_0 \wedge ... \wedge x_n$ being a conjunction of atoms from $\mathcal{L}$. Then, given $H = C_0 \vee C_1 \vee ... \vee C_m$, a state $S = \{y_0, \ldots, y_N\}$ satisfies H if and only if $\bigwedge_{y_i \in S} y_i \models H$, that is, if and only if there exists $i \in \{0, \ldots, m\}$ such that $C_i \subseteq S$ (an assignment $\mathcal{A}$ is a model of a formula F if F is true under this assignment, $\mathcal{A} \models F$). Even without knowledge about the dynamics of the system or the agent's behavior, these observations provide a series of states where it is known whether or not the agent's goals are being reached. In fact, we can build the set of successful states $S_{positive}$ by including in it all the end-states of successful traces from Σ, such that, $S_{positive} = \{endS(T) | T \in \Sigma\}$. In the same way, we can build the set of unsuccessful states $S_{negative}$ taking the union of all the intermediate states, such that, $S_{negative} = \bigcup_{T \in \Sigma} intS(T)$. The problem of inferring an agent's goals is then equivalent to a concept learning problem with the states belonging to $S_{positive}$ as positive examples and the states belonging to $S_{negative}$ as negative ones. We said that a hypothesis H is consistent with our data if all the elements of $S_{positive}$ satisfy it and none of the elements of $S_{negative}$ satisfy it. We are looking for such a hypothesis as the output of our problem. We designed an algorithm, LFST, to address this problem and produce the hypothesis sought. LFST can be found in section 4.1.2.

Our assumptions are strong, and we are aware of this, though we do not think they are

absurd. Of course, usually, an agent will not stop acting after reaching its goal, but a significant change in its behavior could be observed. This significant change could hint about when the goal was reached, and then we could find the successful state. For instance, in the field of game AI, especially for Real-Time Strategy game such as StarCraft the player will usually follow a specific pattern called "Build Order", it is a specific series of actions that will lead the player to a specific state in the game (a number of units and buildings at disposal) which allow the player to use different strategies. There are many strategies in this game and as many Build Orders. Players will tend to wait to finish their Build Order before launching an offensive. If we want to learn the different Build Orders used by a player by analyzing different games that this player played, we have to find the moment when the offensives started to determine when the Build Order was completed and then use our method to learn it. We did not use this method to generate the possible goals of the player in the first part of the thesis, the reason why is that we use expert knowledge as we knew it was available. Of course, it will always be more precise than using our method to generate possible goals for the player. However, this is related to the fact that there is a huge database of expert knowledge about StarCraft, and it is not the case for many other RTS games where our method then would be helpful. Now, we will introduce some definitions adapted from [64] to make the rest of the thesis easier to understand.

Definitions

- A *prototype* is a conjunction of atoms from $\mathcal{L}$ (for example $p_i = x_0 \wedge x_1 \wedge x_2$), the states of the world S being also conjunctions of atoms from $\mathcal{L}$ they are prototypes, there are prototypes of different sizes.

- We say that a prototype p_1 is *more general* than a prototype p_2 if and only if the set of atoms describing p_1 is a subset of the set of atoms describing p_2. A prototype p_1 is *more specific* than a prototype p_2 if and only if the set of atoms describing p_2 is a subset of the set of atoms describing p_1.

- A prototype p_1 *covers* a prototype p_2 if and only if p_1 is more general than p_2, written $cover(p_1, p_2)$. On the other hand, a prototype p_1 rejects a prototype p_2 if and only if p_2 is not more general than p_2.

4.1.2 Learning From Successful Traces

As shown in the previous section, we can model the problem of inferring an agent's goal from the observation of a set of successful scenarios as a concept learning problem by converting the set of successful traces Σ that we observed into the sets of states $S_{positive}$ and $S_{negative}$. In fact, the goal of the agent will be the concept that will be learned. To solve this problem, we will produce a hypothesis H in the form of a DNF and such that H is consistent with our data (i.e., satisfied by all the states in $S_{positive}$ and none of the states in $S_{negative}$), as described in section 4.1.1. In other words, for any state $S_i \in S_{positive}, \exists p_j \in H$ a prototype such as $cover(p_j, S_i)$ and for every state $S_k \in S_{negative}, \nexists p_j \in H$ such as $cover(p_j, S_k)$. Given this, using an existing symbolic concept learning algorithm such as MGI, Ripper, or ID3 [64, 67, 66] would be possible. With an output close to the form we are looking for, MGI might be a good candidate. However, being a bottom-up algorithm, the least general generalization of some positive examples will provide the different conjunctive statements of the disjunctive hypothesis. Knowing that we will probably have many more negative examples than positive examples in our specific case, we adapted this algorithm using different biases that will favor the generation of general terms. We named this concept learning process LFST (Learning From Successful Traces) and described it in Algorithm 4.

Algorithm 4: Learning From Successful Traces

Input: $S_{negative}$, $S_{positive}$
Parameter: $listOfPotentialGoals$ = list(), i = integer, $hypothesis$ = list()
Output: $hypothesis$

for state in $S_{positive}$ **do**
  if $\exists p \in hypothesis$ such as $cover(p, state)$ **then**
    skip to next $state$
  else
    $listOfPotentialGoals$ = [list of all the prototypes covering $state$]
    $p_{current}$ = the most general prototype $p \in listOfPotentialGoals[i]$ such as p covers as many states from $S_{positive}$ as possible)
    while $\exists S \in S_{negative}$ such as $cover(p_{current}, S)$ **do**
      $listOfPotentialGoals[i]$.remove($p_{current}$)
      $p_{current}$ = the most general prototype $p \in listOfPotentialGoals[i]$ such as p covers as many states from $S_{positive}$ as possible)
    end while
    $hypothesis$.append($p_{current}$)
  end if
end for

Two sets are given as input of the algorithm, the set of positive examples $S_{positive}$ and the set of negative examples $S_{negative}$, they are used for concept learning. We first use one of the positive examples to generate a potential goal that corresponds to a prototype covering this positive example. We start from the most general prototypes by selecting those that cover as many positive examples as possible. We then check if the potential goal is valid or not by using the set of negative examples and checking that none of this set's elements are covered by the prototype representing the potential goal. If it is the case, then the potential goal is valid. We keep repeating this process until, for every positive example, there is at least one potential goal that covers it. In the end, our algorithm will return a DNF corresponding to the one that we described in section 4.1.1. Our algorithm will always finish, and in the worst-case generate a new potential goal for each of the positive examples and go through the whole set of negative examples for each potential goal. Therefore, the complexity of LFST in terms of time is $O(|S_{positive}| * |S_{negative}|)$ according to the number of positives and negatives examples, which is a bit less than being quadratic in terms of the size of the whole trace, which means that using more data too learn the hypothesis does not increase the computing time of LFST too much. However, complexity is 2^N according to the number of variables that describe the environment because of the potential goals generation phase. Then the final complexity is in $O(|S_{positive}| * |S_{negative}| * 2^N)$.

Works based on PAC-learning [68] of DNF should also be mentioned, especially Jackson and his Harmonic Sieve algorithm [69, 70], they rely on learning with membership queries and Fourier analysis. They use an oracle to get information about the DNF value they want to learn for specific assignments of atoms and compute a Boolean function close to the DNF thanks to this information. Our case's main problem is that we do not have an oracle that could be questioned on the value of the DNF sought for a specific assignment of atoms. We do have positive examples, but they are much more specific than the prototypes constituting the DNF. Without an oracle, these algorithms could not be used to learn the agent's set of goals, and we did not use them as possible candidates to compare to LFST.

4.2 Experiments

To show that the choice to use our algorithm was well-founded, we conducted experiments. Since our approach is quite different from the current ones in goal recognition, it is not fair to compare results of LFST with those of another goal recognition algorithm for the same data

set. What we are doing here could be seen as identification and classification, so it is more natural to compare LFST with algorithms that could also generate a hypothesis about the possible goals of the agent, or that could classify if states are goal states or not according to what has been learned before. That is why we compare our method to other concept learning algorithms and not to traditional plan recognition algorithms since, as mentioned before, we are not tackling the same problem. Where traditional plan recognition approaches based on probabilistic models or deep learning like [15, 46] need a set of possible goals and will then, according to the observations that they get from the agent, compute which goal and plan have the highest probability to be followed by the agent, we do not compute such a thing. Instead, we will try to generate this set of possible goals used by the other approaches. We do not put probabilities on the goals; we generate the possible ones according to the data that we observed. In fact, our method could be seen as a pre-process for traditional plan recognition approaches. We are trying to generate the set of goals that the agent is aiming at, and this by using concept learning. Therefore, the question of the legitimacy of our LFST algorithm compared to already existing algorithms arises. These experiments are here to determine whether we made the right choice or not. We compare LFST to MGI since, as we said before, the output of MGI is quite close to the one of LFST, and our formalism can be applied to it. We do also use C4.5 [65] as a comparison to see how well LFST performs compared to a well-known decision tree. The comparison with C4.5 shows if the hypothesis generated by LFST is still consistent with the rest of the data that have not been used for learning. The hypothesis can act as a classifier as it should theoretically cover the remaining positive examples and reject the negative ones.

4.2.1 Experimental Protocol

For MGI, we use precisely the same process as LFST as the two algorithms are pretty similar. We use as input the sets of negative and positive examples MGI will use the positive examples to learn the concept to be learned from them and refine this concept using the negative examples. In the end, MGI will return a hypothesis of the very same style as LFST. C4.5 being a decision tree used for classification, does not produce any formula. Instead, it returns a "positive" or "negative" answer when we ask if the input corresponds to the learned concept or not. As for the two previous algorithms, the set of positive examples is used to learn the concept (goals of the agent) and the set of negative examples to learn what is not the concept, basically, all the positive examples belongs to the class "positive" and the negative ones to the class "negative". The decision tree is built thanks to this, and then using holdout method, the remaining part

of the positive and negative examples that were not used to build the tree is given as input to C4.5 to see how it classifies them ("positive" or "negative"). Figure 4.1 is an example of how we train C4.5 for one of our environment (421World described below), each state of the environment correspond to one input with each variable being a feature for C4.5, and the last feature indicates if the state of the world is negative or positive since we want to learn to classify them. The input is almost the same as for LFST and MGI. Cross-validation is used for the three algorithms; in the case of LFST and MGI, we will check if the hypothesis generated thanks to the learning data is still consistent with the remaining data. We use one agent and three different environments; two are deterministic, and the last one is non-deterministic. For each environment, the agent's goals and its decision model are different. The first environment is a grid of size $16 * 16$ that we have named "GridWorld", the second environment is a game in which the agent has to move some blocks, we named it "CubeWorld", and the last environment is a dice game named "421World".

GridWorld is then, a grid of size $16 * 16$ with some randomly placed walls between its cells. In this environment, the goal of the agent is reaching a dead-end, a dead-end being a cell surrounded by exactly three walls. We assume that for each generated grid there exists a dead-end and that this dead-end is reachable by the agent from its different starting positions in the grid. In GridWorld the agent can perform five different actions, it can move one cell top, down, left, right, or stay at its current position. Walls are impassable obstacles for the agent, a wall between two cells will prevent the agent to move from a cell to the other, the agent cannot get out of the grid either, even if there are no walls along the edge of the grid.

GridWorld is a deterministic environment since the agent's actions cannot fail, meaning that for each move that is not prohibited in its current position, the agent will necessarily arrive at the corresponding position. The successful traces are generated by implementing the agent as follows. First, the agent uses the A^* algorithm to compute a path that reaches a dead-end from its starting position. Then the agent begins its journey. Each state and action is recorded in the trace, knowing that only one action is performed by the agent at each time step. Using the formalization described earlier to design the trace of the agent's journey, we will use the following variables to build our atomic set $\mathcal{L}$: a variable *pos* related to the agent's position in the environment, the variables upW, $rightW$, $downW$, and $leftW$ respectively related to the presence of walls at the top, right, down, and left of the agent. For the experiments in GridWorld, we generated eight different grids. Given that, depending on the inlet position,

there are four possible configurations for dead-ends, we assume that each of these configurations appears at least once among all the grids. We generated multiple traces for each instance of a grid, depending on the agent's starting position.

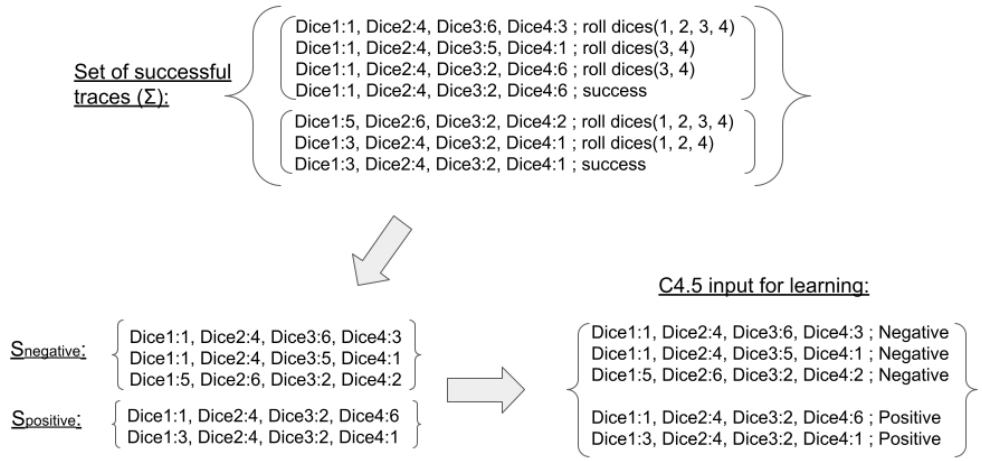


Figure 4.1: Illustration of C4.5 learning setup with our model

The second environment, CubeWorld, is a game with six blocks numbered from 1 to 6 and three columns. The agent has to place block 1 between block 2 and 3 with block 2, which has to be at the column's base, and block 3 above block 1. The agent wins the game as soon as this configuration is reached on one of the columns. Knowing that a block cannot be moved if another block is above this one. The agent can stack as many blocks as wanted on a column. We implemented a naive algorithm to allow the agent to solve this problem and thus obtain many solutions. Figure 4.3 is an illustration of CubeWorld. Each block movement caused by the agent corresponds to a time step. To design the agent's trace in this environment, we use a set of variables describing the result of each action performed by the agent. For each of the six blocks, we have a variable corresponding to the column on which the block is situated $blockPos$, and another variable indicates the height on which it is located $blockH$, which corresponds to coordinates on a plan.

The third and last environment, 421World, is a game of chance with four dices (six-sided dices), and the goal of the agent is to get a combination with at least one "4", one "2", and one "1". The agent can choose to keep the value of some dices; for example, if it gets a "4", it can keep it and roll only the three remaining dices. The game ends when the agent obtains the winning combination "421", the dices can be rolled without limitation. This environment

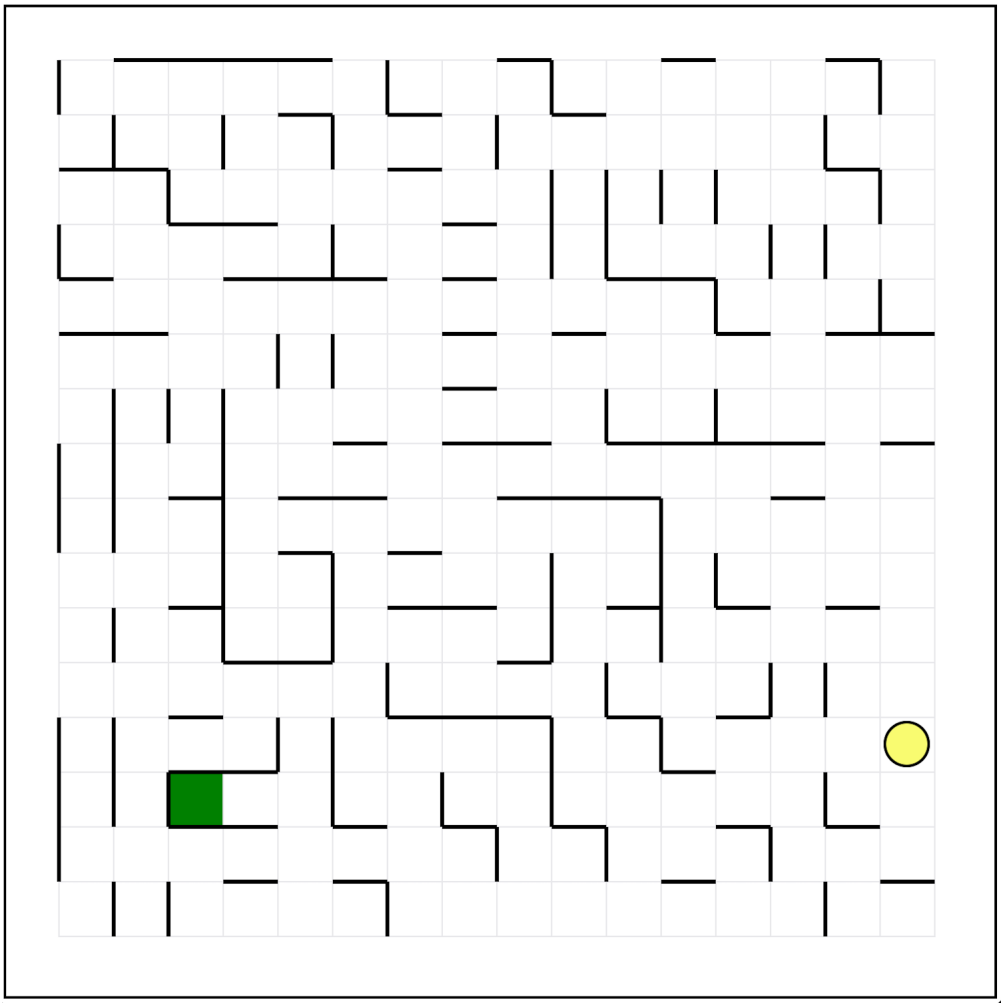


Figure 4.2: GridWorld example

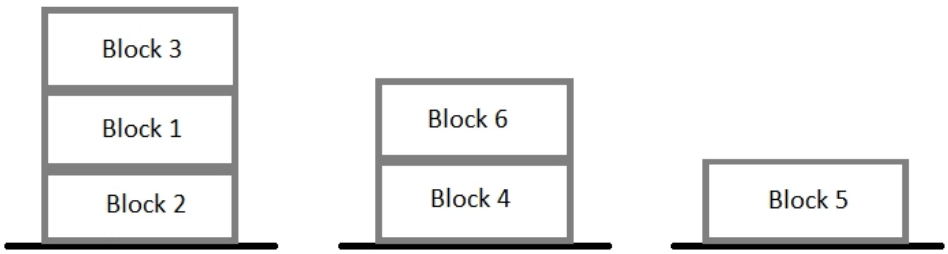


Figure 4.3: CubeWorld winning example

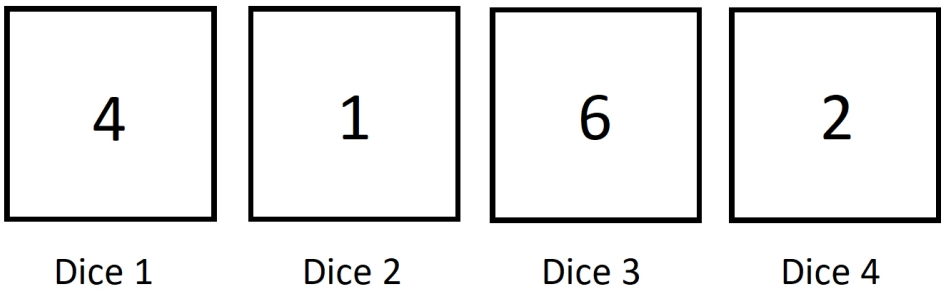


Figure 4.4: 421World winning example

is non-deterministic since the agent cannot know the result of the roll of the dices. Figure 4.4 is an example of a winning combination. Each dices rolling corresponds to a time step. Designing the trace of the agent for this environment is done in the same way as before; a set of variables describes the result of each action of the agent. For each of the dice, there is a variable $diceValue$, which corresponds to the number that the agent obtains after rolling it.

For our experiments, we have an initial set of data with negative and positive examples collected from the generated traces. They are used as input for LFST, MGI, and C4.5. Then from this set of data, the holdout method is used for the validation; we use 70% as a training set and the 30% remaining as a test set to measure the performances of the algorithms. We also vary the size of the initial set of data to simulate some noise (missing states in the traces). In the beginning, 100% percent of the data is used, then we will decrease by two percent the percentage of data used at each simulation, gradually moving from 0% of missing data to 90%. For each data range, we run two hundred times the three algorithms, taking randomly the amount of data needed in our dataset and computing the average of the results. However, the same percentage of data is used for positive and negative examples. For instance, if we use 80% of the data, we will randomly pick 80% of the positive examples and 80% of the negative ones. For GridWorld environment, we have done the experiments with a set of 128 positive examples and a set of 1791 negative ones, which have been extracted from a set of 128 successful traces. For CubeWorld environment, we have done the experiments with a set of 100 positive examples and a set of 559 negative ones, which have been extracted from a set of 100 successful traces. In 421World, we have done the experiments with a set of 200 positive examples and 882 negative examples; this results from 200 successful traces.

4.2.2 Measures

Different types of measures are used to evaluate our results. The first measure is a *syntactic distance* between the actual agent's goals and the hypotheses returned by the algorithms. It shows the similarities between the hypotheses and the target goals. However, this measure is only used to compare LFST and MGI as C4.5 does not produce a hypothesis.

The *syntactic distance* is defined as follows: the symmetrical distance is computed from each goal in the hypothesis to each of the agent's actual goals. Then, for each of the goals of the hypothesis, the minimum of this symmetrical distance is taken and will define the distance between this goal and the agent's goal set. Afterward, the distance between the agent's set of goals and the hypothesis is computed by summing the distance of each of the goals of the hypothesis to the agent's goal set that has been computed previously. For example if our hypothesis is $H = (a \wedge c \wedge d) \vee (b \wedge d)$ and the real set of goals is $G = (a \wedge d) \vee (b \wedge c \wedge d)$, we have:

$$d(H, G) = d((a \wedge c \wedge d), G) + d((b \wedge d), G) \tag{4.1}$$

$$d(H, G) = min(Card(\{c\}), Card(\{a, b\})) + min(Card(\{a, b\}), Card(\{c\})) \tag{4.2}$$

$$d(H, G) = min(1, 2) + min(2, 1) = 1 + 1 = 2 \tag{4.3}$$

Where $Card(X)$ is the function cardinal which is equal to the number of elements in X.

The second measure is the predictive accuracy of the three algorithms, it is the fraction of successfully classified examples, and it is calculated as follows:

$$predictive\ accuracy = \frac{Tp + Tn}{Tp + Tn + Fn + Fp} \tag{4.4}$$

Where Tp, Tn, Fp, and Fn are respectively, True Positive, True Negative, False Positive, and False Negative.

The last one is the recall, which is the fraction of positive examples that are successfully classified as such by the algorithms. This measure is the same as in section 3.3, and is calculated with equations 3.3.1.

4.2.3 Results

As shown by the experiment, the average syntactic distance between the agent's actual goals and the results provided by LFST is pretty good because this average is a little less than 9 in the worst-case for GridWorld, as we can see in Figure 4.5. Knowing that for GridWorld,

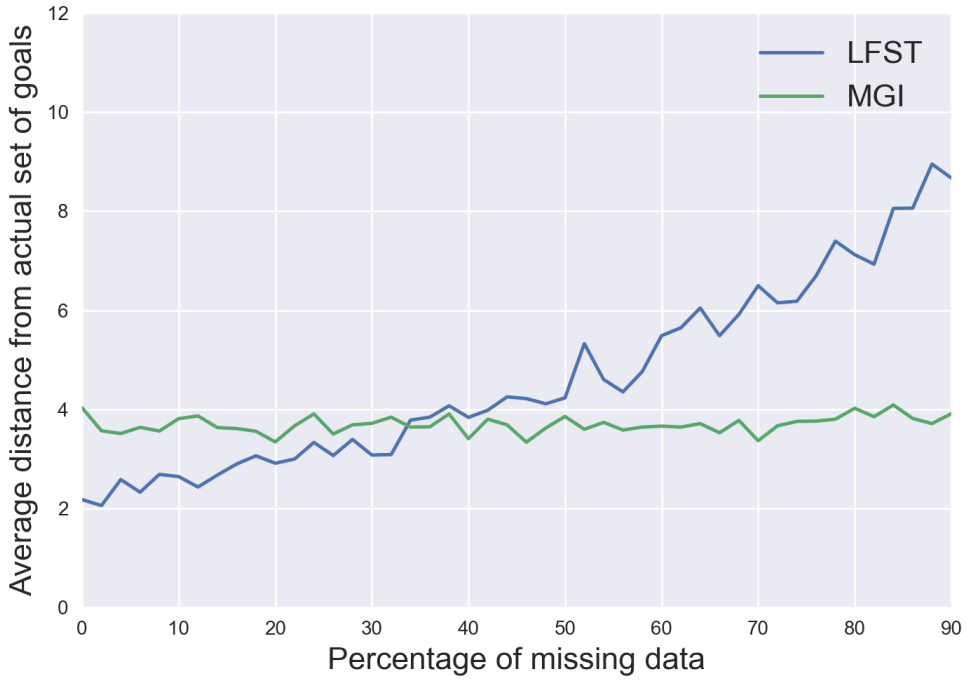


Figure 4.5: Comparison of the average distance from the actual goals in GridWorld

the agent has four goals composed of three atoms each, which means that on average, in the worst-case, LFST still finds almost one-third of the atoms composing each real goal. Moreover, even when using as few as twenty percent of the initial data set, the results are still correct. Indeed, the average syntactic distance being around 5. It means that almost two-thirds of the atoms composing each real goal of the agent are still deducted even in this case. MGI is more constant and thus stays to an average distance of 4 from the actual set of goals, being less precise than LFST until sixty percent of missing data where LFST gets worse. This difference can be explained by the fact that MGI will generate more specific prototypes than LFST. Thus, even if the actual goal is a conjunction of three atoms, MGI will generate a prototype of four or five atoms that cover the goal, and since they do not cover any negative examples, it will stop there. LFST will generate more general prototypes that the negative examples will invalidate, so the prototypes will start to be more and more specific and eventually as specific as the actual goals. Nevertheless, when there is not enough data, the prototypes which are too general will not be invalidated anymore due to the lack of negative examples, so the average syntactic distance from the actual goals will increase. However, as we can see in Figure 4.6, the

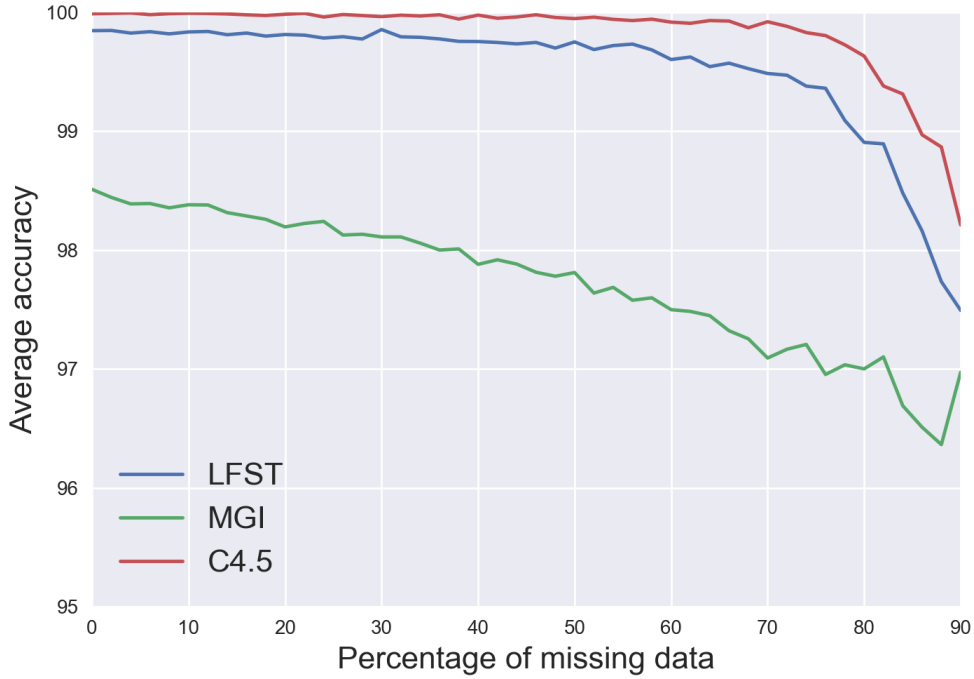


Figure 4.6: Comparison of the average accuracy in GridWorld

accuracy of LFST is always better than the one of MGI and remains on average around 99.7%. As expected, the accuracy of both algorithms is decreasing with the quantity of data used. The difference between the two algorithms is that LFST will rarely fail to classify a positive example. Thus it will produce very few False Negatives when MGI fails more often to classify a positive example. C4.5 has excellent accuracy in this environment, being even slightly better than LFST and reaching 100% of accuracy, though it is only a small gain compare to LFST. However, the most significant difference in efficiency for the algorithms is the recall; Figure 4.7 shows that the value for the recall of LFST and C4.5 is much better than the one of MGI, which is just around the average (50%). The reason for this result is the same as before since the recall is highly dependent on the number of False Negatives.

As for CubeWorld, in this environment, the agent has three goals of six atoms, so the syntactic distance cannot exceed 18, and we can see in Figure 4.8 that LFST obtains a bit less than 10 in the worst-case, so we infer more than one-third of the goals in this case. The value of the distance for MGI is, on average, around 9, which is almost the worst value for LFST, so here also LFST is more accurate. The accuracy and the recall, Figure 4.9 and 4.10 are also

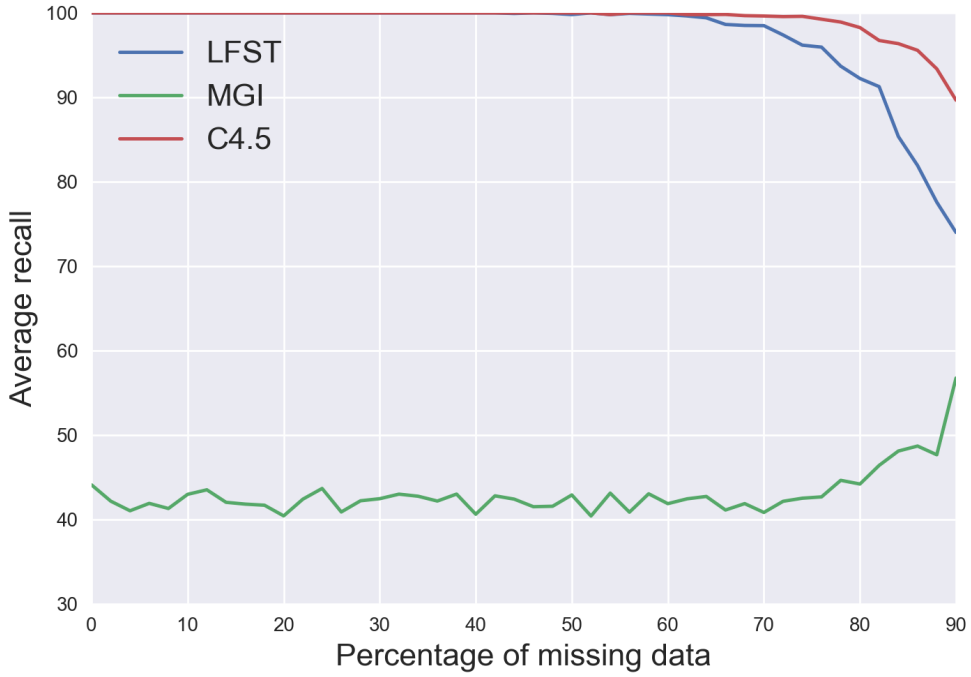


Figure 4.7: Comparison of the average recall in GridWorld

better here with LFST than with MGI, and here LFST also has better accuracy than C4.5, but it is a difference on the same scale as in the previous environment, relatively slight. The difference is more pronounced for the recall value, where LFST is also better than C4.5.

In 421World, since the agent's goal is to obtain a dice combination of 421, the dice's order and the fourth one's value do not matter, which implies that there are many winning combinations, 24 to be exact. So the agent has 24 goals that correspond to all the winning combinations. This environment is then the one that could potentially give the worst results. Though, as we can see in Figure 4.11 the syntactic distance from the actual goals is relatively low for both algorithms. The accuracy and recall Figure 4.12 and 4.13 are less good than for the other environment, especially the recall for MGI is very low in this environment. We can observe a difference this time between LFST and C4.5, LFST being on average 10% more accurate than C4.5.

In these three environments, the performances of LFST are better than that of MGI. We also observe that LFST and C4.5 seem to have the same performances here, with a slight advantage for LFST, being better in two among three environments. Our guess was that C4.5 performs

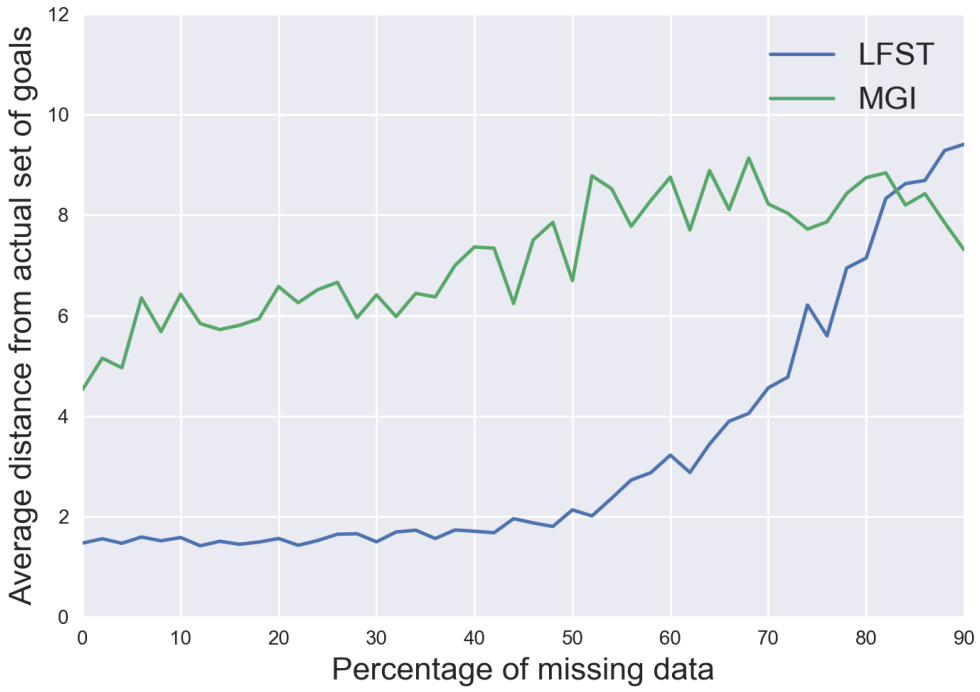


Figure 4.8: Comparison of the average distance from the actual goals in CubeWorld

better in environments without a big difference between the number of positive and negative examples, though the results between CubeWorld and 421World do not necessarily validate this hypothesis. Nevertheless, that might be related to 421World itself that seems to be rather challenging for all the algorithms. However, if we have to compare the algorithms' running time when the number of variables to describe the environment increases. In that case, we can see a difference between LFST and MGI, as shown by Figures 4.14 and 4.15, for GridWorld, there are five variables to describe the environment, and the two algorithms seems to be on the same order of computation time as the curves on 4.14 are almost indistinguishable from each other. For CubeWorld, though, where we have twelve variables to describe the environment, MGI is faster than LFST; this can be seen in Figure 4.15, MGI is on average seven times faster than LFST. The difference between the execution time of both algorithms is due to the fact that MGI uses a heuristic search to find the prototypes that will constitute the result when LFST does not. We chose CubeWorld and GridWorld to show these comparisons because one of the environments has the highest number of examples used as learning data, and the other the highest number of variables. We can see that for LFST, increasing the number of variables to describe the

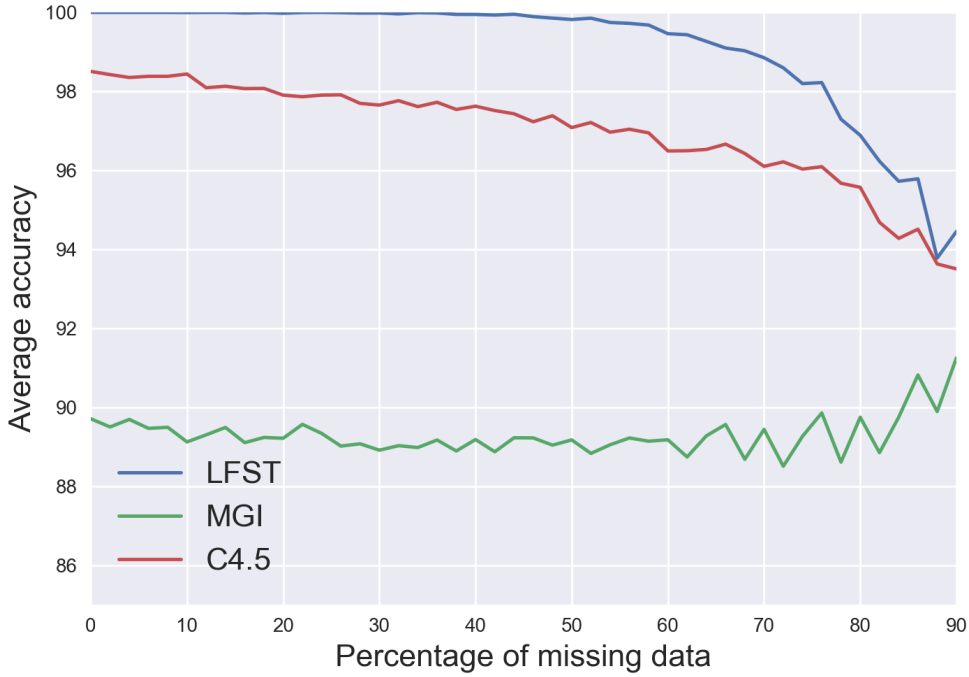


Figure 4.9: Comparison of the average accuracy in CubeWorld

environment has a much more substantial impact on the computation time than increasing the number of examples, as it was expected according to the $O(|S_{positive}| * |S_{negative}| * 2^N)$ complexity of the algorithm. Still, the algorithm is pretty fast for these environments, and we could always play with the representation of the environment to avoid having a too high number of variables describing it. The experiments being done on an old CPU i7 two cores at 3.3GHz and using Python as the programming language. We cannot see C4.5 in Figures 4.14 and 4.15, this is because C4.5 take much time compared to the two other algorithms to learn the concept (build the decision tree) 1.9 to 0.6 seconds on average for GridWorld and 1.5 to 0.5 seconds for CubeWorld, which makes it difficult to see the difference between MGI and LFST. One might argue that the learning time is not essential to learn concepts since it is just learned one time, though we believe that if it has to be part of some real-time system, it is better to have a fast algorithm. We do think we made the right choice using our algorithm LFST for this problem since it performs as well if not better than traditional concept learning algorithms, and compared to C4.5, LFST does produce the learning concept, thing that C4.5 is unable to do. The tables 4.1, 4.2, and 4.3 are here to help the reader to visualize the accuracy and recall

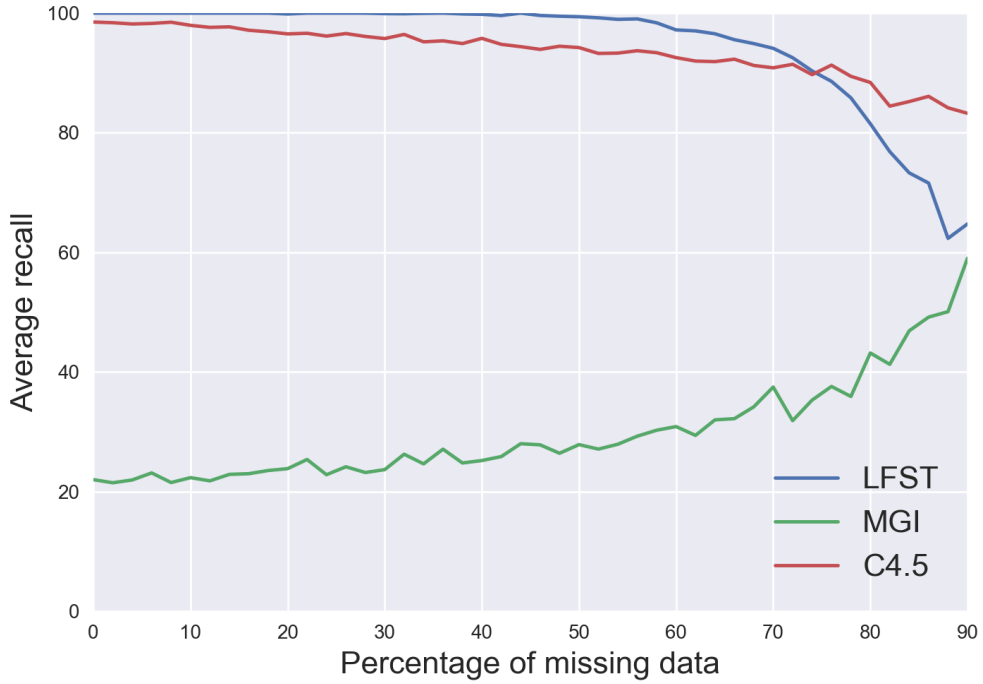


Figure 4.10: Comparison of the average recall in CubeWorld

of the different algorithms on the three environments (only half of the data points have been included in order to reduce their size).

4.3 Extensions and Discussion

4.3.1 Inferring the Agent's Model and Environment Rules From Data

In previous sections, the input Σ, the set of successful traces, is simply reduced to two different sets of states $S_{negative}$ and $S_{positive}$. By doing so, we are not taking advantage of the information included in Σ about the order in which the states are explored and the actions that the agent performs at each step. By ignoring these aspects, we have the advantage of being able to infer possible goals without more assumptions about the agent than the ones induced by successful traces how successful traces are defined, that is, as soon as its goal is reached, the agent stops (with a `success` action) and this goal depends only on the current state of the environment. However, if the dynamics of the environment are known, it seems relevant to derive some information from what the agent has chosen to do, compared to what the agent could have done. It is only possible if we are given some insight into how the agent chooses its actions and

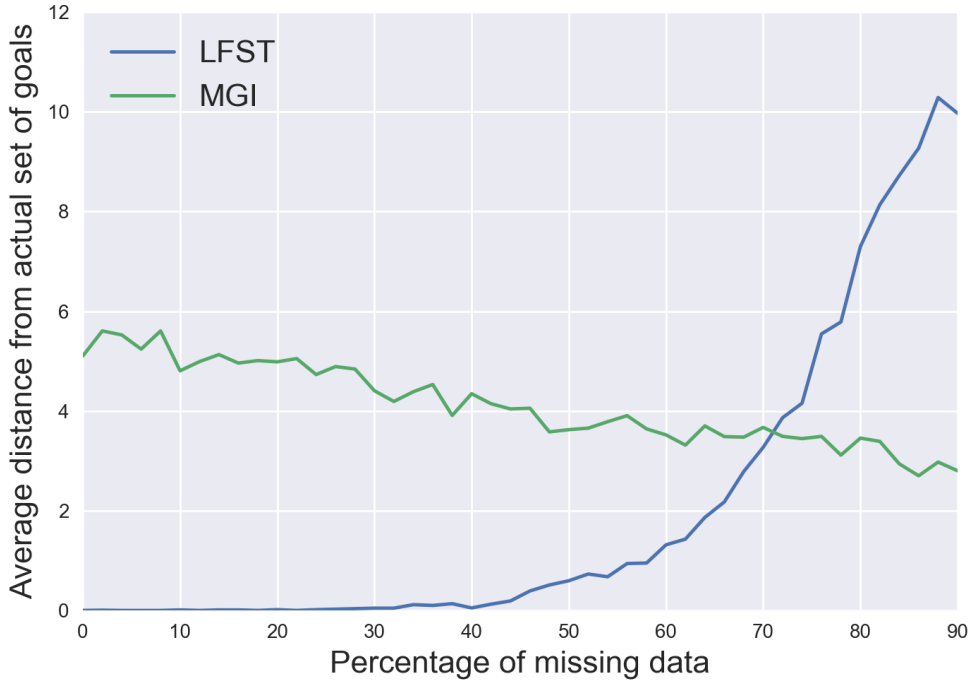


Figure 4.11: Comparison of the average distance from the actual goals in 421World

takes decisions.

For instance, if we consider a rational omniscient agent's case, we know that the observed agent has complete knowledge about its environment and its rules and uses a planner to compute the shortest plans to reach its goal. In such a case, by knowing the environment dynamics, we can use it to generate more negative examples, for each successful trace $T \in \Sigma$ of size K, all the negative examples that are reachable from the state i in $K - i$ steps can be generated and added to the set $S_{negative}$. We can compute this using K_i times the function $next$ with every possible action to get all the reachable states. Using the same process with weaker assumptions, we consider now the case of a deterministic environment for which it is assumed that the agent is smart enough to reach its goal with only one move when its goal is reachable in one move (it cannot fail since the environment is deterministic). This amounts to considering that the agent can plan at least one step ahead, i.e., to understand its actions' direct consequences and choose its actions accordingly. In such a case, when also knowing the environment and its dynamics, many negative examples can be generated. Indeed, if the action model ruling the effect of the agent's actions is known, we can deduce from each state of the world the states that can be

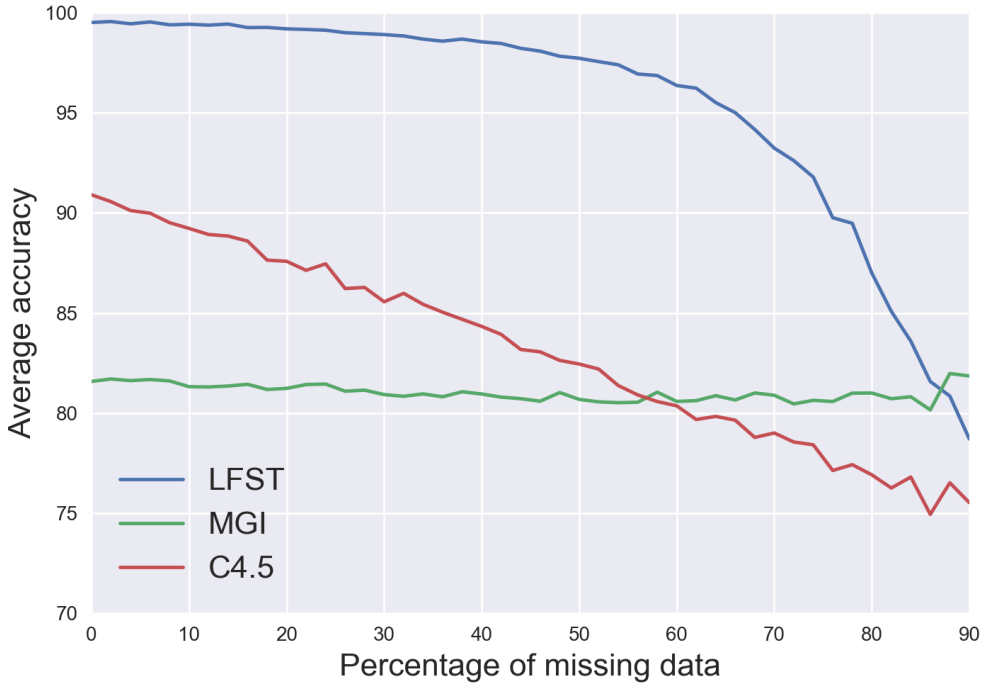


Figure 4.12: Comparison of the average accuracy in 421World

reached by the agent at the next time step. Next, if we know that at the time-step t the agent did not reach its goal, it means that all the states that were reachable by the agent at the time step $t-1$ are not final ones. Furthermore, all these intermediate states can thus be regarded as negative examples, allowing us to refine the results of LFST by using more learning examples. In such a case, we modify LFST by this pre-processing that is applied to each $T \in \Sigma$ in order to compute additional negative examples. This process improves learning by expanding the initial input. It allows us to learn precise goals when using fewer original examples.

Figure 4.16 shows us the advantage of having these kinds of assumptions about the agent and environment. For this graph, the same experimental protocol as GridWorld is used but this time with a grid of size $4*4$, which means that we have 192 positive examples and 776 negative ones that are used as input to LFST when no assumptions about the agent and environment are made. Then, we make two assumptions about the agent: the agent action model is known, and the agent can plan at least one step ahead so if its goal is reachable at step $t-1$, at step t it will reach it. These are not strong assumptions, and they allow us to generate more data from the 192 itineraries that have been followed by the agent throughout the simulation. In

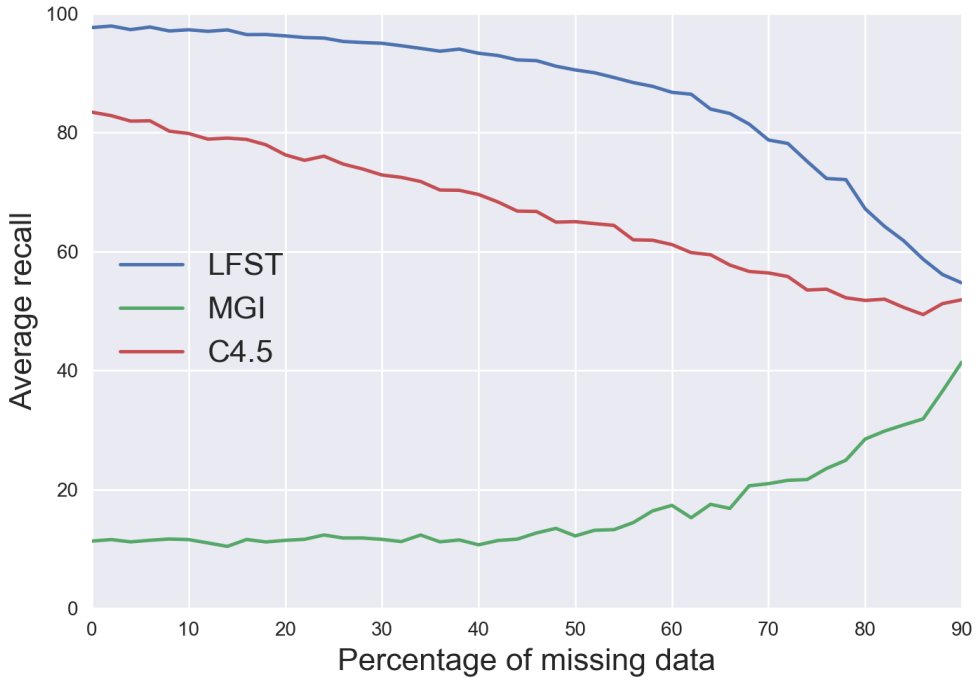


Figure 4.13: Comparison of the average recall in 421World

total, we get 2074 negative examples thanks to this data generation, which is almost three times more than what we have without making assumptions about the agent. We can see in Figure 4.16 that, as we expected, it allows LFST to generate more precise hypotheses about the agent's goals since the result is much closer to the actual goals than when we do not make assumptions. As we can see in Figure 4.16, it allows us to be closer to the actual goals than when we use no assumptions. With enough knowledge about the agent's action model, we are able to use the method of [14]. This method matches our method quite well since it needs a set of goals to find which ones are the more likely to be sought by the agent by using planning. So we can first generate this set of goals by using LFST and then refine our set of possible goals with [14]. That allows us to correct some goals that could have been wrongly inferred by LFST. However, the combination of these two methods is not trivial. Indeed, the definition of a "goal" is not exactly the same between LFST and [14] since they define a goal as a final state where we define it as a conjunction of atoms which is included in this final state. It means that to check the validity of a goal generated by LFST, we need to go through all the traces where this specific conjunction appears in the description of a state of the world assume that in these

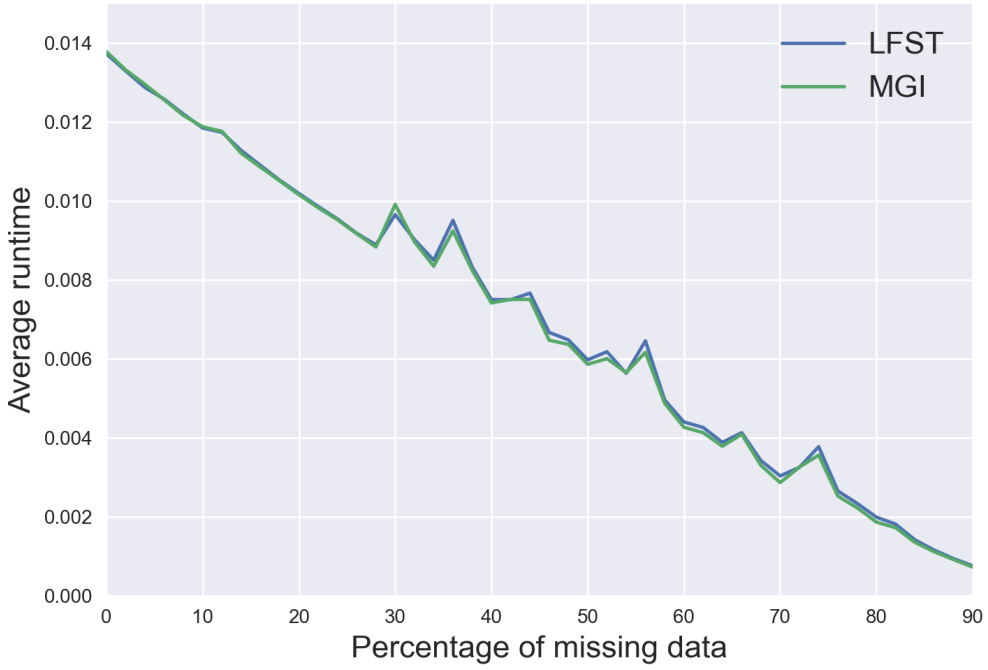


Figure 4.14: Comparison of the average runtime in GridWorld (in seconds)

traces, this state is a final state and then apply [14].

4.3.2 Discussion

We have two interesting methods that we wish to exploit to deduce the agent's action model and the environment rules. The first one is to use LFIT [71] which is a framework for learning normal logic programs from transitions of interpretations, which means that, since we use a logical representation of the world, we will be able to infer the possible action model of the agent and the rules of the environment. In fact, to use LFIT, we just need a trace of the evolution of the states of the world in chronological order, which are the same data that we use to infer the agent's goals. It means that we do not need an important modification of the data collection process to incorporate LFIT into our method, which is a good point. After processing these data with LFIT, we will obtain a set of rules that correspond to the possible transition from a state of the world to another. We will then have for each state of the world (observed previously in the collected data) the different actions that the agent did when it was in this state and the states of the world that it reached after these actions. So we can infer a potential action model of the agent and the dynamic of the environment based on the

Figure 4.15: Comparison of the average runtime in CubeWorld (in seconds)

observed transitions. Unfortunately, LFIT is not very robust against the lack of data, and the noise, especially the data need to cover as many transitions as possible between all the different possible states of the world. It is not very convenient because usually, the agent we observe is not acting randomly, and so, many possible transitions will never be observed. We also plan to use SMILE [72] another method that we can use to learn the action model of the agent. This method uses the same kind of data as LFIT, so here also the modification of the data collection process to use SMILE, and our method is not too cumbersome. Even if SMILE is usually used for a multi-agent system because it is, in fact, the different agents who will learn information about the other agents of the system, we can just create an agent "Observer" who will learn the action model of the observed agent. SMILE is more robust to the lack of data than LFIT but not as effective in some cases. So we will choose which method to use based on the quantity of data obtained during the observation phase. We already tried SMILE and LFIT with our data to extract the agent's action model and the rules of the environment. We obtained promising results, and so, we want to continue in this way. Nevertheless, we do not have combined our method and these two algorithms yet.

Figure 4.16: Comparison of the average syntactic distance from the actual goals in function of the percentage of missing data with and without assumptions on the agent.

To improve our method and provide some more interesting hypotheses about the agents' goals, we thought about incorporating a preferences function about these ones' goals. It means that our hypothesis will provide us with a set of possible goals for an agent and ordered these goals according to the agent's preferences. We want to start first with a simple preference model; by simple, we mean that if the agent prefers to reach the goal A rather than the goal B, its preferences will not consider the difficulty to reach these two goals. The process of inferring agents' preferences will take place after the goals inference in our method. Because, of course, we need to know the agents' goals to be able to order them. So once we have the agent's goals, we also need to know the agent's action model, then, for each intermediate state of the trace of the agent, we can see if there are other reachable goals that the one reached in the final state of the trace and if so, compare the number of time that a goal was preferred to another one. If between two goals, one has always been preferred, and if it happened several times, we can assume a strict preference between them $A > B$. Otherwise, the preference will be more moderate $A \succeq B$. Given an order of priority $>_p$ about an agent's goals, we can translate the fact $A >_p B$ in a logic formula by assuming that if the agent finally reaches the goal B,

it is because the goal A was not reachable. Then we can translate this in the logic formula $A \vee \neg A \wedge B$ (where A and B are atoms conjunctions) by using our language $\mathcal{L}$ described in section 4.1.1. Likewise, if we have the preference $A >_p B >_p C$ on the agent's goals we can write this $A \vee \neg A \wedge B \vee \neg A \wedge \neg B \wedge C$. With such a translation, we could write all the possible strict orders of preference on the agent's goals, and since the writing still represents a DNF, we can modify a bit our algorithm and return a disjunction of such DNF as a result and thus obtain the possible goals of the agent and its preferences for these goals. If the agent is assumed rational, then such a simple linear model should work as transitivity (if $A >_p B$ and $B >_p C$ then $A >_p C$), antisymmetry (if $A >_p B$ and $B >_p A$ then $A = B$) and connexity ($\forall A, B$ we have $A >_p B$ or $B >_p A$) should be supported by the agent's preference model. Also, it could be hard to observe connexity, so we could have several partial orders instead. However, suppose the agent's preferences are not rational (like a human, for example). In that case, this linear model will not work anymore, especially in human case the transitivity of preference is not always true, as shown by the "Condorcet paradox" or "voting paradox", which is that, when voting, majorities prefer, for example, candidate A over B, B over C, and yet C over A. If no cycles about the preference have been observed in the data, we can still assume that we have several partial orders; each of them could be represented as a DNF as described before. We still get a disjunction of DNF where goals will be ordered in their respective partial order (DNF), and the different partial orders could not be compared. Note that, if the goals A and B are not in relation but we have $A >_p C$ and $B >_p C$, we could write it $A \vee B \vee \neg A \wedge \neg B \wedge C$. We can start to work on this extension before completing the one in the previous section. Even if we need the agent's action model, we can still assume that we know it and start the experiments like this.

With our formalization, we do not take into account the cost of the actions for the agent and thus the fact that some goals could be achieved more often than others. If all the goals are achieved at least once, then our method should be able to infer them or build a hypothesis covering the state of the world corresponding to these goals. However, if one particular goal is never reached by the agent (because of its cost, for instance) during the observation process, it will be impossible to infer it with our current method as it is pretty challenging to learn from what has not been observed, especially if there is an important number of possible states of the world. We could potentially deal with this problem using a higher-level representation of the goals that we inferred once our first hypothesis was generated by LFST. A representation

that could allow us to create a global concept for similar goals, and then we can look for states of the world covered by this concept and are not find in the set of negative examples used for learning. For instance, taking the case of GridWorld experiments, we saw that there are four possible dead-ends. Let say that one of them was never reached by the agent, but LFST infer the three other kinds of dead-ends as possible goals. When looking at the hypothesis returned by LFST, we could create the concept "dead-end", which is that the agent is surrounded by three walls. We then check that there are no negative examples that reject this concept and keep it if it is not rejected. The concept "dead-end" will then cover even the fourth kind of dead-end that the agent did not reach when we were observing it. It could help us dealing with unobserved goals, but it needs human assistance or a way to generate concepts from a low-level representation of the environment.

Finally, the experiments' results have shown that the main weakness of LFST is the execution time. We plan to improve the search for new potential goals since our method is currently quite exhaustive. A heuristic research method, as used for MGI, could significantly improve the execution time. However, the difficulty is then to find a good heuristic function. The clues that we have about the agent's goals at the beginning are mainly in the positive examples set $S_{positive}$, then it will not be irrelevant to focus our attention on it. We thought about using our syntactic distance to cluster these positives examples and group together those with a minimum distance from each other. Then, for each cluster, compute the cardinal of the intersection of all the positive examples within this cluster. We can expect that each cluster corresponds to a possible goal. Thus, computing the cardinal of the intersection within the cluster could provide an idea of the size in terms of atoms of the possible goal corresponding to this cluster. Then we can look for prototypes around this size to cover the positive examples in this cluster instead of starting from the smallest possible prototypes as we currently do.

Table 4.1: Average accuracy and recall for the three algorithms in GridWorld

	Accuracy			***Recall***		
Percentage of missing data	LFST	C4.5	MGI	LFST	C4.5	MGI
0	99.85	99.99	98.51	100.0	100.0	44.1
2	99.85	99.99	98.44	100.0	100.0	42.17
4	99.83	99.99	98.39	100.0	100.0	41.04
6	99.84	99.98	98.39	100.0	100.0	41.91
8	99.82	99.99	98.36	100.0	100.0	41.3
10	99.84	99.99	98.38	100.0	100.0	42.98
12	99.84	99.99	98.38	100.0	100.0	43.52
14	99.81	99.99	98.32	100.0	100.0	42.04
16	99.83	99.98	98.29	100.0	100.0	41.82
18	99.8	99.97	98.26	100.0	100.0	41.69
20	99.81	99.98	98.19	100.0	100.0	40.43
22	99.81	99.99	98.22	100.0	100.0	42.41
24	99.78	99.96	98.24	100.0	100.0	43.68
26	99.8	99.98	98.13	100.0	100.0	40.89
28	99.78	99.97	98.13	100.0	100.0	42.22
30	99.86	99.96	98.11	100.0	100.0	42.46
32	99.79	99.98	98.11	100.0	100.0	43.0
34	99.79	99.97	98.06	100.0	100.0	42.76
36	99.78	99.98	98.0	100.0	100.0	42.18
38	99.76	99.94	98.01	100.0	100.0	43.01
40	99.75	99.98	97.88	100.0	100.0	40.63
42	99.75	99.95	97.92	100.0	100.0	42.8
44	99.74	99.96	97.88	99.93	100.0	42.41
46	99.75	99.98	97.81	100.0	100.0	41.51
48	99.7	99.96	97.78	99.94	100.0	41.57
50	99.75	99.95	97.81	99.8	100.0	42.91
52	99.69	99.96	97.64	100.0	100.0	40.41
54	99.72	99.94	97.69	99.79	99.8	43.13
56	99.73	99.93	97.58	99.94	100.0	40.87
58	99.68	99.94	97.6	99.87	100.0	43.04
60	99.6	99.92	97.5	99.79	100.0	41.88
62	99.62	99.91	97.48	99.64	99.8	42.45
64	99.54	99.93	97.45	99.42	99.79	42.73
66	99.57	99.93	97.32	98.63	99.82	41.12
68	99.53	99.87	97.25	98.51	99.67	41.88
70	99.49	99.92	97.09	98.49	99.63	40.84
72	99.47	99.88	97.17	97.38	99.56	42.16
74	99.38	99.83	97.21	96.18	99.59	42.52
76	99.36	99.8	96.95	95.95	99.25	42.68
78	99.09	99.73	97.03	93.69	98.92	44.64
80	98.91	99.63	97.0	92.23	98.26	44.2
82	98.89	99.38	97.1	91.27	96.73	46.41
84	98.48	99.31	96.69	85.36	96.36	48.11
86	98.16	98.97	96.51	81.93	95.57	48.7
88	97.73	98.87	96.36	77.58	93.37	47.67
90	97.49	98.21	96.97	74.0	89.67	56.75

Table 4.2: Average accuracy and recall for the three algorithms in CubeWorld

	Accuracy			***Recall***		
Percentage of missing data	LFST	C4.5	MGI	LFST	C4.5	MGI
0	100.0	98.5	89.7	100.0	98.51	21.99
2	100.0	98.42	89.5	100.0	98.4	21.47
4	100.0	98.35	89.69	100.0	98.19	21.94
6	100.0	98.38	89.46	100.0	98.28	23.11
8	100.0	98.38	89.49	100.0	98.48	21.51
10	100.0	98.44	89.12	100.0	97.95	22.32
12	100.0	98.09	89.3	100.0	97.62	21.79
14	100.0	98.13	89.49	100.0	97.71	22.87
16	99.98	98.07	89.1	100.0	97.13	22.98
18	100.0	98.07	89.23	100.0	96.88	23.52
20	99.98	97.9	89.21	99.86	96.53	23.83
22	100.0	97.86	89.56	100.0	96.63	25.36
24	100.0	97.9	89.34	100.0	96.16	22.82
26	99.99	97.91	89.01	100.0	96.58	24.14
28	99.98	97.69	89.07	100.0	96.1	23.18
30	99.98	97.65	88.91	99.92	95.75	23.67
32	99.96	97.76	89.03	99.89	96.44	26.25
34	99.99	97.61	88.98	99.96	95.2	24.63
36	99.98	97.72	89.17	100.0	95.37	27.09
38	99.95	97.54	88.89	99.87	94.91	24.78
40	99.95	97.62	89.18	99.8	95.78	25.19
42	99.93	97.51	88.87	99.59	94.78	25.85
44	99.95	97.43	89.22	100.0	94.39	28.0
46	99.89	97.23	89.22	99.61	93.94	27.81
48	99.85	97.38	89.04	99.49	94.47	26.4
50	99.82	97.08	89.17	99.4	94.24	27.85
52	99.85	97.21	88.83	99.22	93.26	27.1
54	99.74	96.97	89.05	98.96	93.32	27.91
56	99.72	97.04	89.22	99.03	93.71	29.26
58	99.68	96.95	89.14	98.38	93.38	30.26
60	99.46	96.49	89.17	97.19	92.56	30.87
62	99.43	96.49	88.74	97.03	91.98	29.39
64	99.27	96.53	89.27	96.55	91.89	31.99
66	99.1	96.66	89.56	95.56	92.3	32.19
68	99.03	96.43	88.68	94.9	91.25	34.17
70	98.85	96.1	89.44	94.11	90.86	37.47
72	98.6	96.21	88.5	92.55	91.44	31.85
74	98.2	96.03	89.26	90.34	89.71	35.29
76	98.22	96.09	89.85	88.59	91.31	37.58
78	97.29	95.67	88.61	85.83	89.43	35.9
80	96.89	95.57	89.75	81.5	88.41	43.17
82	96.23	94.68	88.85	76.84	84.45	41.28
84	95.72	94.27	89.75	73.29	85.21	46.87
86	95.78	94.51	90.81	71.58	86.08	49.17
88	93.78	93.63	89.89	62.33	84.17	50.08
90	94.45	93.5	91.25	64.75	83.25	59.0

Table 4.3: Average accuracy and recall for the three algorithms in 421World

	Accuracy			***Recall***		
Percentage of missing data	LFST	C4.5	MGI	LFST	C4.5	MGI
0	99.51	90.9	81.58	97.67	83.43	11.31
2	99.56	90.56	81.7	97.93	82.85	11.55
4	99.45	90.12	81.62	97.32	81.94	11.17
6	99.54	89.99	81.67	97.76	81.98	11.44
8	99.4	89.51	81.6	97.09	80.24	11.65
10	99.43	89.22	81.32	97.3	79.85	11.55
12	99.38	88.92	81.3	97.03	78.9	10.99
14	99.43	88.84	81.35	97.27	79.08	10.41
16	99.26	88.59	81.44	96.48	78.86	11.57
18	99.27	87.64	81.18	96.5	77.95	11.16
20	99.2	87.58	81.23	96.27	76.26	11.43
22	99.17	87.13	81.42	95.99	75.34	11.6
24	99.13	87.46	81.45	95.9	76.04	12.33
26	99.0	86.22	81.09	95.33	74.7	11.82
28	98.96	86.28	81.14	95.15	73.88	11.82
30	98.91	85.56	80.92	95.03	72.87	11.59
32	98.84	85.98	80.84	94.61	72.47	11.22
34	98.68	85.43	80.95	94.17	71.78	12.32
36	98.58	85.04	80.81	93.69	70.34	11.18
38	98.69	84.68	81.06	94.04	70.3	11.5
40	98.55	84.33	80.95	93.34	69.59	10.68
42	98.47	83.93	80.79	92.97	68.34	11.4
44	98.22	83.18	80.72	92.23	66.81	11.63
46	98.08	83.06	80.59	92.1	66.74	12.68
48	97.83	82.63	81.02	91.17	64.95	13.44
50	97.73	82.45	80.68	90.54	65.02	12.17
52	97.56	82.2	80.56	90.07	64.69	13.12
54	97.4	81.37	80.52	89.27	64.4	13.23
56	96.94	80.91	80.54	88.42	61.97	14.41
58	96.86	80.58	81.04	87.78	61.9	16.38
60	96.37	80.36	80.58	86.78	61.17	17.31
62	96.23	79.68	80.62	86.45	59.82	15.23
64	95.51	79.83	80.87	83.96	59.46	17.49
66	95.01	79.64	80.65	83.22	57.73	16.79
68	94.17	78.78	81.0	81.43	56.64	20.6
70	93.24	79.0	80.89	78.76	56.4	20.96
72	92.61	78.55	80.45	78.18	55.8	21.52
74	91.79	78.41	80.63	75.16	53.52	21.66
76	89.76	77.13	80.57	72.28	53.67	23.51
78	89.48	77.42	80.99	72.1	52.22	24.9
80	87.0	76.91	81.0	67.19	51.78	28.46
82	85.08	76.25	80.72	64.23	51.99	29.8
84	83.59	76.8	80.81	61.8	50.59	30.85
86	81.57	74.94	80.16	58.71	49.4	31.86
88	80.85	76.52	81.97	56.13	51.24	36.53
90	78.71	75.52	81.85	54.72	51.89	41.4

Chapter 5

Conclusion

This last chapter concludes the thesis. The section 5.1 summarize the different contributions of the thesis. Section 5.2 discusses merging the two approaches that have been presented in this thesis into one single framework. Finally, section 5.2 show future perspectives of this work.

5.1 Contribution of the Thesis

Plan and goal Recognition has a wide variety of applications. But, unfortunately, there are still several serious problems that slow down the use of goal recognition in large-scale real-world situations (computation time, action space, etc...). However, in this thesis, we presented a method to solve plan recognition in RTS games. As we showed in the introduction, RTS games are very complex games, and solving plan recognition in such games and in real-time brings us closer to real-world applications. Even when restricting the application field to games, it is actually a very important economic sector with a rise in the demand for adaptive AIs in order to make games more entertaining for humans. We saw that our method handles missing and delayed observations. Compared to most of the current plan recognition approaches, we also deal with not ordered actions and interleaved plans. Our method predicts the goal of the observed player and also infers its plan to reach it. The performance is similar to machine learning methods that have been used for the same purpose (goal recognition) in previous work. However, our method has the advantage that it could work without training data but only by using expert knowledge. We proposed extending this method by including multi-horizons prediction, which means predicting the player's plan at five minutes horizon, and when we reach this horizon, predict the new plan at ten minutes horizon based on the previous prediction and continue for the next horizons. This will lower the computation time compared to trying to

predict at ten minutes horizon from the start. It also makes much sense since the human players usually do the same. They plan an opening strategy for the five first minutes of the game, and if it was not enough to win the game, then they transition to a middle-term strategy. We also discussed using another planner, like the one of Churchill and Buro [60] as the one we used has been developed for simpler games and struggles with the state space of more complex RTS games like StarCraft, and using different pruning heuristic, like a heuristic based on landmarks similarly to Pereira *et al.* [22]. However, many plans share similar actions but achieve different goals in our case, so it might be harder to find landmarks that are unique to each goal; a solution could be to use the timing of the actions.

We then presented our goal recognition method and showed how different it is from previous approaches in the field of goal recognition. Because few approaches are focused on learning the agent's set of goals, they often consider it given in the problem. Besides, our formalization of the problem also allows us to use concept learning. We use our algorithm, LFST, to control some generalization bias, and we have shown that it was a legitimate choice in chapter 4.2 since our algorithm performs better than MGI and even C4.5. We could also make a version with a heuristic search of the prototype to improve the execution of LFST since it is the only point where MGI is better than LFST. We also showed that we could drastically improve the goals deduction process's efficiency by making assumptions about the agent and its environment. This is why we plan to use LFIT or SMILE to infer the agents' models and the environment's rules to exploit such assumptions. This and the integration of the agents' goals preferences in our model will allow us to infer more refined hypotheses about the agent's goals. Also, as we said in section 4.3.1 if we have good knowledge about the agent and its environment, we can combine our method to this one [14], and we might obtain rather promising results. One drawback is that, in reality, it is hard to know when an agent reaches its goal. It is actually the main weakness of our method, and we need to overcome this weakness. For this, dynamic learning throughout the agent's actions could be efficient and would be more realistic. It could also be effective in the case of an agent that changes its goal along the way or in a case where the agent repeats the same cycle of actions, cases that we cannot solve yet with our method.

5.2 Combining Plan and Goal Recognition

In the work presented in this thesis, we did not use LFST to generate the possible goals that were used in chapter 3 for our plan recognition method. Instead, we used expert knowledge as

we had them at hand, and it is more accurate than trying to infer the possible goals with LFST. However, we could combine the two methods to have one single framework. LFST would be used first to generate the set of possible goals for the player, and then this set will be used to perform plan recognition in RTS games. A good point that makes this implementation very easy is that we use propositional logic to represent the state of the environment with both of our methods, so we do not need any translation to combine them as they used the very same formalism. We said that we have a lot of expert knowledge for the case of StarCraft, so we do not need to try to infer the player's goals. However, if we want to have the player's goals at a specific time horizon of the game, says twelve minutes, expert knowledge can be missing. So we can take all the replays files and stop them at twelve in-game minutes. Then we can use our algorithm LFST to build a hypothesis about the player's goals at twelve minutes by fixing the state of the environment that the player reached at that time as a goal state (positive example in LFST formalism), and the previous states as intermediate ones (negative examples). Indeed it will require some preprocessing as some replays might include misleading information, like if the player was attacked by the opponent and had fewer buildings and units at twelve minutes than before. We could also want to infer the player's goals right before it starts an offensive move to be able to anticipate it. In this case, the player's goals would be different armies composition. We can analyze the replay files and stop them when we observe the first offensive move of the player, and then we set as goal state the state of the world as it was just before this move. Figure 5.1 illustrates the complete process of using first LFST on the replay files to infer the player's possible goals; the replays are converted into positive and negative examples as it is the input that LFST needs. Then, using the set of possible goals that have been output by LFST, we can use them as input for our plan recognition method, and we will infer the most likely plan and goal for the player.

Combining our two methods is even more relevant if we apply it to other games as we may not have much expert knowledge. In such a case, it might be hard to find possible goals for the agent in order to perform plan recognition. However, by using our goal recognition algorithm LFST, we could design a set of possible goals for the agent and use it for plan recognition. Furthermore, most of the RTS games follow the same rules as StarCraft, so our method could be directly applied to them; we should just adjust the planner that we use. Finally, as we explained before, most of the current methods in goal recognition predict the most likely goal among a given set of candidates. Thus, by combining our goal and plan recognition method, we

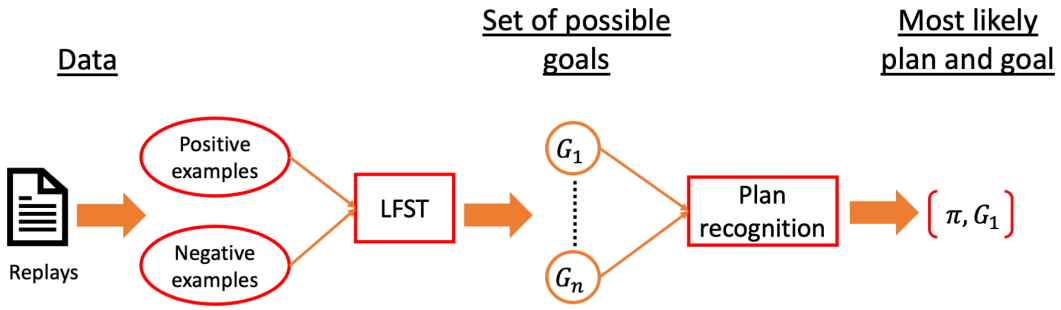


Figure 5.1: Illustration of the full goal recognition and plan recognition process

could get a very complete method as we will be able to generate this set of candidates and then perform plan recognition to infer the most likely plan and goal of the observed agent. To the best of our knowledge, there are no existing approaches that do this complete plan recognition process as they often use a given set of candidates.

5.3 Further Perspectives

The fact that we succeed at performing goal recognition in real-time in an environment as complex as an RTS game, even though the results are not perfect, is an important step forward to solve the plan and goal recognition problem, as it brings us closer to solve in real-time real-world problems. Future perspectives for this thesis are first to implement the multi-horizons plan recognition system introduce in section 3.4.1. It can be done relatively easily as there is nothing to change in the plan recognition method. The only thing we need is to perform statistical analyses about the differents strategy transitions usually performed by the players and then modify the inputs and outputs of our plan recognition algorithm. The next step would be to try another pruning heuristic. As we said before, the heuristic from Pereira *et al.* [22], based on landmarks uniqueness, looks very promising. However, compared to the environments that Pereira *et al.* used for experiments, in RTS games, it is harder to find landmarks that more

or less unique for each goal, especially as all the actions that players perform are not always related to their goal. What could work well and that we plan to try is to use a combination of our current heuristic and the landmarks heuristic as the latter is very fast to run. Using a kind of consensus between them, if both heuristics agree to prune or keep a goal, then we do it. Otherwise, if both heuristics disagree, we will follow the one that puts more weight in the balance (the one for which the threshold for pruning or keeping has been exceeded the most). The landmarks heuristic could help when our current heuristic hesitates between two goals with rather similar plans if the landmarks of these two goals are different. On the other hand, our heuristic could help if the player performed the landmark of some goal without actually trying to achieve this goal. And then merging our goal recognition and plan recognition method together as described previously.

Plan and goal recognition are very interesting and challenging problems to solve; however, one question arises: to what end shall we solve these problems. Knowing the plan and goal of an agent is indeed a good thing, but if this knowledge is not used, then one could argue that the efforts invested into earning it are not really worth it. Our idea behind performing plan recognition in RTS games was to be able to develop adaptive AI that could entertain the human player more than hardcoded AI as players are currently used to. Indeed developing a complete bot to play RTS games is very intensive work and would take much time, but we can instead build the differents module that will compose such a bot. We wanted the AI to be adaptive in the way that knowing the strategy of the human player, the AI could adapt its own strategy in order to counter the player more efficiently. So what we could work on next would be a way to decide which strategy to use, knowing the player's strategy. Some works already treated similar problems, like Antuori and Richoux [73] that use constrained optimization to manage their unit production in order to counter the opponent. An idea could be to maximize the following function:

$$max\ f(x) : x_1u_1 + ... + x_nu_n \tag{5.1}$$

Under the constraints:

$$\begin{aligned} x_1c_1^m + ... + x_nc_n^m &\leqslant R_m \\ x_1c_1^g + ... + x_nc_n^g &\leqslant R_g \\ x_1c_1^t + ... + x_nc_n^t &\leqslant T \end{aligned} \tag{5.2}$$

Where x_i are buildings or units, u_i the utility of the building or unit x_i according to the strategy currently used by the opponent player, this utility would be learned from replays, c_i^m, c_i^g and,

c_i^t respectively the cost in terms of minerals, gas and time of x_i. And finally, R_m, R_g, T the amount of minerals, gas, and time that are at disposal. The last constraint about the time is not totally exact as some action might be performed in parallel. In fact, it should be the longest sequence of actions that should not exceed T. Solving this constraint optimization problem is equivalent to finding the best buildings and units to generate during the game to counter the current strategy of the opponent player. However, we expect the task of learning the utility function from the replays to be relatively complex, so we also thought about another way to adapt the strategy to the opponent which is, by using a decision tree. Instead of learning from the replay a utility function for each building and unit, we will learn the utility function for each strategy against the other strategies.It can be easily done by looking at how successful each strategy is (looking for the win rate ratio of each strategy against the others and even using expert knowledge), and then we can build a decision tree similar to figure 5.2. We can see that the first node corresponds to the strategy to adopt, each strategy having a cost; this cost is the transition cost from the current strategy to the new one that we wish to adopt. Each strategy has its own probability of failure or success against the opponent's strategy and has its own gain in case of success and losses in case of failure. We say that some strategies, hard or soft, counter other strategies, which implies that in some cases, the gain or losses could be relatively low (soft counter), but the opposite can happen too (hard counter). We need to choose the strategy that solves the following equation.

$$argmax_i \ (p_s^i \cdot gains(S_i) - \overline{p_s^i} \cdot losses(S_i)) - cost(S_i) \tag{5.3}$$

These were the two ways we are currently considering to solve the problem of adapting the agent's strategy to the opponent's strategy once it has been inferred through plan recognition. Thus we could make use of the information that we got about the opponent and take an edge in the game. Constraint optimization and decision trees have the advantage of being used for many problems and not only in the case of RTS games if we want to extend our method to other fields.

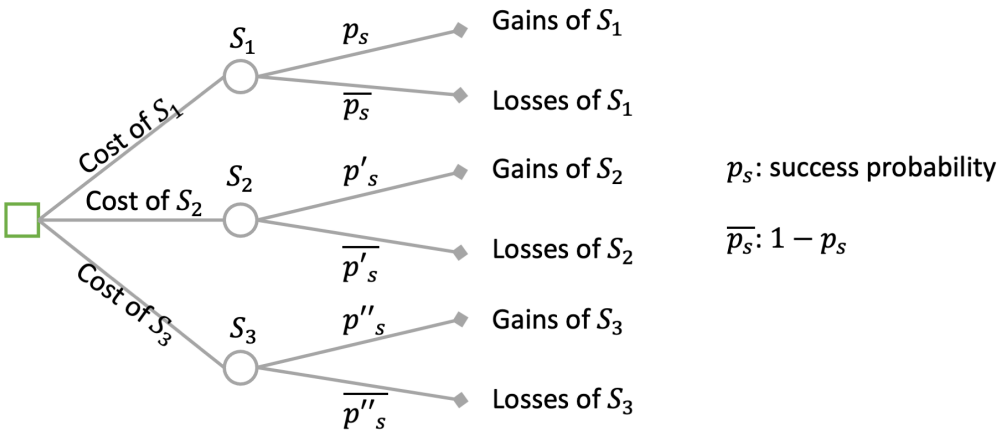


Figure 5.2: Example of decision tree to choose the strategy to adopt

Bibliography

[1] Y. E-Martin, M. D. R-Moreno, and D. E. Smith, "Practical goal recognition for iss crew activities," in *IJCAI the International Workshop on Planning and Scheduling for Space 2015 (IWPSS)*, (Buenos Aires, Argentina), 2015.

[2] D. Ognibene, L. Mirante, and L. Marchegiani, "Proactive intention recognition for joint human-robot search and rescue missions through monte-carlo planning in pomdp environments," in *International Conference on Social Robotics*, pp. 332–343, Springer, 2019.

[3] C. F. Schmidt, N. S. Sridharan, and J. L. Goodson, "The plan recognition problem: An intersection of psychology and artificial intelligence," *Artificial Intelligence*, vol. 11, no. 5, pp. 45–83, 1978.

[4] H. A. Kautz and J. F. Allen, "Generalized plan recognition.," in *AAAI*, vol. 86, p. 5, 1986.

[5] E. Charniak and R. P. Goldman, "A probabilistic model of plan recognition," in *AAAI*, 1991.

[6] E. Charniak and R. P. Goldman, "A bayesian model of plan recognition," *Artificial Intelligence*, vol. 64, no. 5, pp. 53–79, 1993.

[7] R. P. Goldman, C. W. Geib, and C. A. Miller, "A new model of plan recognition," in *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, 1999.

[8] D. V. Pynadath and M. P. Wellman, "Probabilistic state-dependent grammars for plan recognition," in *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, 2000.

[9] H. H. Bui, S. Venkatesh, and G. West, "Policy recognition in the abstract hidden markov model," *Journal of Artificial Intelligence Research*, vol. 17, pp. 451–499, 2002.

[10] H. H. Bui, "A general model for online probabilistic plan recognition," in *IJCAI*, vol. 3, pp. 1309–1315, Citeseer, 2003.

[11] C. W. Geib and R. P. Goldman, "A probabilistic plan recognition algorithm based on plan tree grammars," *Artificial Intelligence*, vol. 173, no. 11, pp. 1101–1132, 2009.

[12] P. Singla and R. J. Mooney, "Abductive markov logic for plan recognition.," in *AAAI Proceedings of the twenty-fifth national conference on Artificial intelligence 2011*, pp. 1069–1075, 2011.

[13] C. Geib and P. Kantharaju, "Learning combinatory categorial grammars for plan recognition," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[14] M. Ramırez and H. Geffner, "Plan recognition as planning," in *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)*, 2009.

[15] M. Ramırez and H. Geffner, "Goal recognition over pomdps: Inferring the intention of a pomdp agent," in *IJCAI*, pp. 2009–2014, Citeseer, 2011.

[16] C. L. Baker, J. B. Tenenbaum, and R. R. Saxe, "Goal inference as inverse planning," in *Proceedings of the Annual Meeting of the Cognitive Science Society*, vol. 29, 2007.

[17] C. L. Baker, R. Saxe, and J. B. Tenenbaum, "Action understanding as inverse planning," *Cognition*, vol. 113, no. 3, pp. 329–349, 2009.

[18] C. L. Baker, J. Jara-Ettinger, R. Saxe, and J. B. Tenenbaum, "Rational quantitative attribution of beliefs, desires and percepts in human mentalizing," *Nature Human Behaviour*, vol. 1, no. 4, pp. 1–10, 2017.

[19] J. Hong, "Goal recognition through goal graph analysis," *Journal of Artificial Intelligence Research*, vol. 15, pp. 1–30, 2001.

[20] Y. E-Martín, M. D. R-Moreno, and D. E. Smith, "A fast goal recognition technique based on interaction estimates," in *Proceedings of the 24th International Conference on Artificial Intelligence*, pp. 761–768, 2015.

[21] M. Vered and G. A. Kaminka, "Heuristic online goal recognition in continuous domains," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pp. 4447–4454, 2017.

[22] R. Pereira, N. Oren, and F. Meneguzzi, "Landmark-based heuristics for goal recognition," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2017.

[23] R. F. Pereira, N. Oren, and F. Meneguzzi, "Landmark-based approaches for goal recognition as planning," *Artificial Intelligence*, vol. 279, p. 103217, 2020.

[24] D. Höller, G. Behnke, P. Bercher, and S. Biundo, "Plan and goal recognition as htn planning," in *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 466–473, IEEE, 2018.

[25] L. Amado, R. F. Pereira, J. Aires, M. Magnaguagno, R. Granada, and F. Meneguzzi, "Goal recognition in latent space," in *2018 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2018.

[26] R. Mirsky, K. Gal, R. Stern, and M. Kalech, "Goal and plan recognition design for plan libraries," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–23, 2019.

[27] S. Keren, A. Gal, and E. Karpas, "Goal recognition design," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 24, 2014.

[28] S. Carberry, "Techniques for plan recognition," *User Modeling and User-Adapted Interaction*, vol. 11, no. 5, pp. 31–48, 2001.

[29] G. Sukthankar, C. Geib, H. H. Bui, D. V. Pynadath, and R. P. Goldman, *Plan, Activity, and Intent Recognition*. Elsevier, 2014.

[30] F. A. Van-Horenbeke and A. Peer, "Activity, plan, and goal recognition: A review," *Frontiers in Robotics and AI*, vol. 8, p. 106, 2021.

[31] D. W. Aha, "Goal reasoning: Foundations, emerging applications, and prospects," *AI Magazine*, vol. 39, no. 2, pp. 3–24, 2018.

[32] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[33] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[34] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[35] D. W. Albrecht, I. Zukerman, and A. E. Nicholson, "Bayesian models for keyhole plan recognition in an adventure game," *User modeling and user-adapted interaction*, vol. 8, no. 1, pp. 5–47, 1998.

[36] S. Ontanón, K. Mishra, N. Sugandh, and A. Ram, "On-line case-based planning," *Computational Intelligence*, vol. 26, no. 1, pp. 84–119, 2010.

[37] S. Ontanón, "Combinatorial multi-armed bandits for real-time strategy games," *Journal of Artificial Intelligence Research*, vol. 58, pp. 665–702, 2017.

[38] S. Ontanón and M. Buro, "Adversarial hierarchical-task network planning for complex real-time games," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[39] F. Kabanza, P. Bellefeuille, F. Bisson, A. R. Benaskeur, and H. Irandoust, "Opponent behaviour recognition for real-time strategy games," in *Proceedings of the 5th AAAI Conference on Plan, Activity, and Intent Recognition*, AAAIWS'10-05, pp. 29–36, AAAI Press, 2010.

[40] E. Ha, J. P. Rowe, B. W. Mott, and J. C. Lester, "Goal recognition with markov logic networks for player-adaptive games.," in *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 2011*, 2011.

[41] F. Schadd, E. Bakkes, and P. Spronck, "Opponent modeling in real-time strategy games," in *Proceedings of the GAME-ON 2007*, pp. 61–68, 2007.

[42] B. G. Weber and M. Mateas, "A data mining approach to strategy prediction," in *2009 IEEE Symposium on Computational Intelligence and Games*, pp. 140–147, Sep. 2009.

[43] G. Synnaeve and P. Bessiere, "A bayesian model for plan recognition in rts games applied to starcraft," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 6, 2011.

[44] G. Synnaeve and P. Bessiere, "A bayesian model for opening prediction in rts games with application to starcraft," in *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pp. 281–288, IEEE, 2011.

[45] G. Synnaeve and P. Bessiere, "Multiscale bayesian modeling for rts games: An application to starcraft ai," *IEEE Transactions on Computational intelligence and AI in Games*, vol. 8, no. 4, pp. 338–350, 2015.

[46] W. Min, B. Mott, J. Rowe, B. Liu, and J. Lester, "Player goal recognition in open-world digital games with long short-term memory networks," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pp. 2590–2596, AAAI Press, 2016.

[47] W. Min, B. Mott, J. Rowe, R. Taylor, E. Wiebe, K. Boyer, and J. Lester, "Multimodal goal recognition in open-world digital games," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 13, 2017.

[48] G. Synnaeve, Z. Lin, J. Gehring, D. Gant, V. Mella, V. Khalidov, N. Carion, and N. Usunier, "Forward modeling for partial observation strategy games-a starcraft defogger," *Advances in Neural Information Processing Systems*, vol. 31, pp. 10738–10748, 2018.

[49] C. Geib, J. Weerasinghe, S. Matskevich, P. Kantharaju, B. Craenen, and R. Petrick, "Building helpful virtual agents using plan recognition and planning," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 12, 2016.

[50] P. Kantharaju, S. Ontañón, and C. W. Geib, "Extracting ccgs for plan recognition in rts games.," in *KEG@ AAAI*, 2019.

[51] P. Kantharaju, S. Ontanón, and C. W. Geib, "Scaling up ccg-based plan recognition via monte-carlo tree search," in *2019 IEEE Conference on Games (CoG)*, pp. 1–8, IEEE, 2019.

[52] P. Kantharaju, K. Alderfer, J. Zhu, B. Char, B. Smith, and S. Ontanón, "Tracing player knowledge in a parallel programming educational game," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 14, 2018.

[53] S. Ontanón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A survey of real-time strategy game ai research and competition in starcraft," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 5, no. 4, pp. 293–311, 2013.

[54] J. S. Bruner and G. A. Austin, *A study of thinking*. Transaction publishers, 1986.

[55] M. D. Merrill and R. D. Tennyson, *Concept teaching: An instructional design guide*. Educational Technology, 1977.

[56] E. Rosch and B. B. Lloyd, *Cognition and categorization*. Citeseer, 1978.

[57] H. Chan, A. Fern, S. Ray, N. Wilson, and C. Ventura, "Online planning for resource production in real-time strategy games.," in *ICAPS*, pp. 65–72, 2007.

[58] L. Scrucca, M. Fop, T. B. Murphy, and A. E. Raftery, "mclust 5: clustering, classification and density estimation using Gaussian finite mixture models," *The R Journal*, vol. 8, no. 1, pp. 289–317, 2016.

[59] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[60] D. Churchill and M. Buro, "Build order optimization in starcraft," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 6, 2011.

[61] J. Lang, "A preference-based interpretation of other agents' actions," in *International Conference on Automated Planning and Scheduling 2004*, vol. 4, (Whistler), pp. 33–42, AAAI, 2004.

[62] M. Ramırez and H. Geffner, "Probabilistic plan recognition using off-the-shelf classical planners," in *Proceedings of the Conference of the Association for the Advancement of Artificial Intelligence (AAAI 2010)*, pp. 1121–1126, Citeseer, 2010.

[63] J. Hong, "Goal recognition through goal graph analysis," *J. Artif. Int. Res.*, vol. 15, pp. 1–30, July 2001.

[64] M. Henniche, "Mgi: An incremental bottom-up algorithm," in *Australian New Zealand Intelligent Information Systems Conference 1994*, (Brisbane, Queensland, Australia), pp. 347–351, IEEE, 1994.

[65] S. L. Salzberg, "C4. 5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993," 1994.

[66] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.

[67] W. W. Cohen, "Trees and rules with set-valued features," in *AAAI Proceedings of the thirteenth national conference on Artificial intelligence 1996*, vol. 1, (Portland, Oregon, USA), pp. 709–716, AAAI, 1996.

[68] L. G. Valiant, "A theory of the learnable," *Communications of the ACM*, vol. 27, no. 11, pp. 1134–1142, 1984.

[69] J. C. Jackson, "An efficient membership-query algorithm for learning dnf with respect to the uniform distribution," *Journal of Computer and System Sciences*, vol. 55, no. 3, pp. 414–440, 1997.

[70] N. H. Bshouty, J. C. Jackson, and C. Tamon, "More efficient pac-learning of dnf with membership queries under the uniform distribution," *Journal of Computer and System Sciences*, vol. 68, no. 1, pp. 205–234, 2004.

[71] K. Inoue, T. Ribeiro, and C. Sakama, "Learning from interpretation transition," *Machine Learning*, vol. 94, no. 1, pp. 51–79, 2014.

[72] G. Bourgne, A. El Fallah Segrouchni, and H. Soldano, "Smile: Sound multi-agent incremental learning," in *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pp. 164–171, ACM, 2007.

[73] V. Antuori and F. Richoux, "Constrained optimization under uncertainty for decision-making problems: Application to real-time strategy games," in *2019 IEEE Congress on Evolutionary Computation (CEC)*, pp. 458–465, IEEE, 2019.