Tasks Recognition using AI Planning techniques

Miguel Doctor Yuste

Universidad de Alcalá

Escuela Politécnica Superior
Departamento de Automática

Máster en Ciencia y Tecnología desde el Espacio

Trabajo Fin de Máster

Tasks Recognition using AI Planning techniques

Autor: Miguel Doctor Yuste

Tutora: María Dolores Rodríguez Moreno

TRIBUNAL:

Presidente:

Vocal 1º:

Vocal 2º:

Calificación _____

Alcalá de Henares , a ____ de _____ de 2011

Trabajo Fin de Máster

Tasks Recognition using AI Planning techniques

Miguel Doctor Yuste

Ningún grupo puede actuar con eficacia si falta el concierto y la sinceridad;
ningún grupo puede actuar en concierto y sinceridad si falta la confianza;
ningún grupo puede actuar con confianza si no se halla ligado
por opiniones, afectos o intereses comunes.
Edmund Burke (1729-1797) Político y escritor irlandés.

A todos aquellos que confían en mí;
A todos aquellos que son sinceros conmigo;
A todos aquellos que me apoyan y aníman;
A todos aquellos que consideran mis logros como suyos propios.
En definitiva, a todos aquellos que me quieren.

Abstract

This Master Thesis describes an agent's intentions recognizer based on Artificial Intelligence (AI) planning techniques. The proposed architecture is able to recognize the agent's intentions from partial observations of its actions. In addition, we present different scenarios related to autonomous control systems where our application can be useful, such as goals prediction, performance monitoring, collaborative frameworks among agents or behaviours validation.

Plan recognition can be seen as the inverse operation of AI Planning. In many of the space planning problems, there are situations which we are interested in knowing what intentions are pursuit by the agent in order to help to achieve his goals, to compete with it or simply to oversee and monitor the work performed.

Our recognizer is based on a general-purpose STRIPS-based planner called GraphPlan. This planner uses a planning graph which stores the useful information for constraint search. Then, this information can be quickly propagated through the graph as it is being built. We have exploited this idea to develop a system able to deduce the agent's goals instead of generating a valid plan.

Resumen Ampliado

En el presente Trabajo Fin de Master, presentamos un reconocedor de objetivos basado en técnicas de planificación de Inteligencia Artificial. Más concretamente, nuestro sistema utiliza un conjunto de técnicas agrupadas bajo la denominación de "Plan Recognition" o reconocimiento de planes. Estas técnicas permiten inferir el plan, o secuencia de acciones, así como los objetivos perseguidos por un agente, a partir de la observación de una secuencia incompleta de las acciones que el propio agente está realizando. Los sistemas basados en "Plan Recognition" tienen una fuerte presencia en áreas relacionadas con los sistemas de control autónomo, tales como monitorización, detección precoz de errores o verificación y validación de sistemas. Además, están empezando a ser implementados en otras áreas de gran interes empresarial como por ejemplo: detección de intenciones comerciales, interfaces gráficas adaptativas o modelado de comportamiento humano.

El reconocimiento de planes puede ser visto como la operación inversa a la planificación. Mientras un sistema de planificación busca obtener la secuencia de acciones que permitan a un agente alcanzar un estado objetivo a partir de una situación inicial, los sistemas de reconocimiento de planes proporcionan el plan que presumiblemente está siguiendo el agente, recibiendo como entrada el estado inicial y una secuencia incompleta de las acciones realizadas por el mismo. Sin embargo, en muchos problemas de reconocimiento de planes, hay situaciones en las que es más interesante conocer los objetivos que persigue el agente, en lugar del posible plan que está llevando a cabo. El reconocedor descrito en el presente trabajo se centra en este último enfoque.

Nuestro reconocedor está basado en un planificador de proposito general tipo STRIPS llamado GraphPlan. Este planificador usa grafos de planificación (planning graphs) para mantener la información relacionada con las restricciones y las acciones llevadas a cabo por el agente. El uso de este tipo de grafos nos permite obtener un plan válido de manera eficiente. Nosotros hemos utilizado esta idea para desarrollar un reconocedor capaz de inferir los posibles objetivos perseguidos por el agente a partir de las acciones que va realizando.

El reconocedor implementado, el cual hemos llamado *JPlanRecognizer*, es una aplicación desarrollada en Java que recibe como inputs dos archivos PDDL (un dominio y un problema) y un conjunto de acciones observadas. *JPlanRecognizer* procesa los datos introducidos y genera como output un conjunto de objetivos compatibles con el plan, que el agente monitorizado, está llevando a cabo. A lo largo del documento hemos probado la eficacia de nuestra aplicación en dominios de diferente índole, los cuales van desde el seguimiento de un rover sobre la superficie de Marte, hasta el reconocimiento de intenciones entre usuarios de una red social.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Aims . . . 2

2 Artificial Intelligence Techniques **5**
- 2.1 Planning . . . 5
- 2.2 GraphPlan Review . . . 6
  - 2.2.1 Disjunctive progression stage . . . 7
  - 2.2.2 Plan extraction stage . . . 9
  - 2.2.3 A brief example . . . 9
- 2.3 Plan Recognition . . . 10
  - 2.3.1 Introduction to Plan Recognition . . . 11
  - 2.3.2 Plan Recognition Systems . . . 12
  - 2.3.3 Planning graph based architecture advantages . . . 12
  - 2.3.4 Research fields in Plan Recognition . . . 13

3 JPlanRecognizer a Tool to Recognize Goals **15**
- 3.1 Preliminary Definitions in Plan Recognition Problems . . . 15
- 3.2 Description of the algorithm . . . 18
  - 3.2.1 Acquisition Data . . . 22
  - 3.2.2 Constructor . . . 24
  - 3.2.3 Constrain Checker . . . 26
  - 3.2.4 u-mutex checker . . . 29
  - 3.2.5 ID-Constraint checker . . . 31
  - 3.2.6 ExtendState Procedure . . . 33

4 Complete Operation Examples **37**
- 4.1 Agent Cleaner Domain . . . 37
  - 4.1.1 Domain and Problem . . . 37
  - 4.1.2 Experimentation . . . 39
- 4.2 Social Networking Sites . . . 43
  - 4.2.1 Motivation . . . 43
  - 4.2.2 Domain and Problem presented . . . 44
  - 4.2.3 Conclusions . . . 47

5 Conclusions **49**
5.1 Conclusions 49
5.2 Future Works 49

Bibliography **51**

List of Figures

2.1 GraphPlan General Structure 6
2.2 Inconsistent Effects Example 7
2.3 Interference Example 7
2.4 Competing Needs Example 8
2.5 Negated literals 8
2.6 Inconsistency support 9
2.7 GraphPlan example: Note that for simplicity, some of the actions in layer 3 are omitted 10

3.1 Class diagram of JPlanRecognizer 17
3.2 Main modules of JPlanRecognizer 19
3.3 Sequence diagram of Main method 20
3.4 Sequence diagram of the extendPlanningGraph method 25
3.5 *PlanningGraph* produced by *Constructor* 26
3.6 Sequence diagram of *updateGraph* method 28
3.7 Partial *u-planning graph* updated with an observation 29
3.8 Method uMutexCompute example 30
3.9 u-mutex checker assign k-value=-1 31
3.10 Planning graph obtained at the end of the first iteration of the process 33
3.11 *ExtendState* method example 34
3.12 Planning graph generated by *JPlanRecognizer* for the Mars rover example 35

4.1 Results offered by BlackBox for the agent cleaner problem 40
4.2 Results offered by *JPlanRecognizer* for the agent cleaner problem: First step 40
4.3 Results offered by *JPlanRecognizer* for the agent cleaner problem: Second step 41
4.4 Results offered by *JPlanRecognizer* for the agent cleaner problem: Third step 41
4.5 Results offered by *JPlanRecognizer* for 3-steps and 4-steps examples . . 42
4.6 Results offered by *JPlanRecognizer* for different number of observations 43
4.7 Results offered by *JPlanRecognizer* for three observations 43
4.8 Results offered by *JPlanRecognizer* for Social Network problem 48

Chapter 1

Introduction

In this first chapter we describe the motivations to develop an agent's intention recognizer. We also explain what "Plan Recognition" is, supported by a daily situation example. Finally, we present the structure of the rest of the document.

1.1 Motivation

Space exploration applications offer a unique opportunity for the development and deployment of autonomous systems, due to limited communications, large distances, and high expenses of direct operations. The use of autonomy in space missions will greatly benefit space exploration. First, the operations cost can be significantly reduced as fewer human operators are required; an essential aspect of making future exploration missions cost effective and sustainable. Second, the use of autonomy will permit spacecraft to achieve more in less time, and to accomplish what is otherwise not possible. For example, autonomous Mars rovers could respond to interesting events like dust storms, while autonomous lunar construction rovers could operate continuously. However, autonomous systems are still rarely used in space exploration; this is in part due to reluctance to change and in part due to a technology gap in certifying the behaviour of such systems.

A number of complex systems currently deployed present a significant amount of autonomy, as in the case of the NASA rovers Spirit and Opportunity exploring the surface of Mars since January 2004. In the future there will be an increasing number of robots on the surface of the Moon or Mars. In this scenario, the development of autonomous supervision systems to check the correctness of the agent actions *in situ*, it is considered as an alternative to the current monitoring ground techniques.

There are several approaches that have been used to predict behaviours; we can mention:

- *Neuronal Networks*: are very useful to detect and predict behaviours based in patterns, but we need to spend a lot of time teaching the system. These systems analyse a large quantity of data during the "training" stage to extract patterns of behaviour, the knowledge obtained in this stage is then used to predict behaviours in similar situations. Thus, the predictions offered by these systems are based in

the previous data analysed. The main problem of these approaches is the need to train again to the neuronal network if we want that the system works with new patterns of protocols. For example, in a communication application with rovers, if we change the actions that the rover is able to perform, a neuronal networks based recognizer should be trained again to learn the behaviours resulting from the new available actions [RH95, YMMK04].

- *Data Mining & Clustering*: are based on data exploratory analysis. It means that it does not use inference techniques so we need a huge quantity of data to apply these methods. Another inconvenient is that the extracted information is linked to the data stored, then the obtained knowledge cannot be extrapolated to other kind of users not presented in the considered sample [Kan02, PM04].

- *Heuristic Rules & Probabilistic Algorithms*: can help to traditional search techniques to get better results [ACM+10, HYL08]. The knowledge about the problem beyond the definition could be used to improve the recognizers but they are not enough to recognize patterns with exactitude.

We propose instead to use plan recognition techniques [NGT04, Kau91]. The aim of these systems is to recognize the actions and goals of one or more agents from a series of partial observations on the agents actions and the environmental conditions. To illustrate more clearly what is the meaning of *plan recognition* we present the next situation [Car01]:

Suppose that someone asked you for the location of the Federal Express office and subsequently asked about the availability of delivery outside the country. You might reasonably infer that he or she wanted to quickly get an item to someone in another country and intended to do this using Federal Express for delivery. In doing so, you had inferred the goals of the other person and a portion of that person's plan for achieving those goals. This is often referred to as *plan recognition*.

1.2 Aims

Plan recognition systems have many limitations related to the need of manually input the plans that the systems are able to recognize. In this project we suggest a different approach: we propose to use modern techniques from classical Artificial Intelligence (AI) planning field and apply them to plan recognition problems [SY07, RG09]

The aim of the current Master Thesis is to offer a plan recognition approach based on graph planning techniques. That is, we introduce a technique to use the planning graph to recognize the agents' intentions. We explain the necessary modifications to adapt the standard planning graph called *GraphPlan* [KPL97] to recognize goals and we propose our own extension to improve its possibilities. To illustrate our work we present *JPlanRecognizer*, a Java developed application that implements a graph based plan recognition approach. In addition, two domains have been modelled to evaluate the effectiveness of our application.

The structure of this work is as follows:

- **Chapter 2. Artificial Intelligence Techniques**

  This chapter describes the principles of modern Planning approaches, focussing specially in GraphPlan. It offers a brief description of main features of GraphPlan and we mention the main advantages contributed by planning graph based architectures. At least, we show several fields where plan recognition systems have been successfully implemented.

- **Chapter 3. JPlanRecognizer a Tool to Recognize Goals**

  Chapter 3 explains the planning graph based implemented architecture: basis definitions, data structures and procedures implicated. It also shows the *JPlanRecognizer* design from a static and dynamic point of view. Finally we present an example of a rover in Mars to illustrate the application operation.

- **Chapter 4. Other application areas**

  Chapter 4 presents another example to show the effectiveness of the developed application. First we discuss the different answers offered by *JPlanRecognizer* when it analyse the same problem with different observations. In addition a Social Networking site domain is shown in this chapter to probe that our recognizer has many application domains' from space environments (as we have seen in the previous chapter) to current web sites.

- **Chapter 5. Conclusions**

  This chapter presents and discusses the conclusions achieved after the completion of this work. Further works are also outlined.

Chapter 2

Artificial Intelligence Techniques

We review the basis and the main techniques arisen around AI Planning techniques. We explain the principles of the planning and we list different approaches. After, we present a new discipline called "Plan Recognition" focusing on its relationship with traditional planning systems. The basics of "Plan Recognition" are mentioned and explained in this chapter. Finally, we discuss about other research areas where "Plan Recognition" techniques have been successfully applied.

2.1 Planning

In the last two decades the advances in computer science have been translated into tools and techniques to automate processes that until then were performed and supervised by humans. Without any doubt, AI is one of the computer science areas with more expectation created in the last years. AI is a scientific discipline which tries to operational human intellectual and cognitive capabilities in order to make them available through information processing systems. Within AI, one of the problem solving techniques that has gained more relevance is AI Planning.

AI Planning techniques select an ordered sequence of activities in order to achieve that an agent achieved one or more goals, satisfying a set of domain constraints. Therefore, a planner is a system that solves planning problems which are usually modelled on a specific language called Planning Domain Definition Language (PDDL) [MGH+98]. PDDL is an attempt to standardize planning domain and problem description languages. It was mainly developed to make the 1998/2000 International Planning Competitions possible. It was first developed by Drew McDermott in 1998 and later evolved with each International Planning Competition. The latest version of this language is PDDL3.1 [GL05].

Two inputs are required for a typical planner: a domain and a problem files.

- *Domain file*: It represents a description of the world. This description includes predicates, variables considered, elements, available actions with preconditions and effects, etc

- *Problem file*: It describes the initial state of the problem, the set of goals to achieve, and the instantiated objects.

After processing these two files the planner will produce an output. This output consists of a sequence of actions, denominated plan. This plan is a solution to a given problem, in consequence these actions can be performed by the agent to arrive from an initial state to the searched goal state. The way to obtain this plan is different depending on the techniques used by the planner. These techniques offer different features with impact in several important parameters such as time needed, completeness of the solution or memory space requirements. There are many kind of planning systems that perform their search based on different techniques and algorithms such us, Total Order Planning [BVV01, VR99], Partial Order Planning [PW92], Heuristic search planners [BG98, HN01]. In this project we focus on a concrete family of planners: Planners based on a planning graph, in particular GraphPlan [BF95].

2.2 GraphPlan Review

GraphPlan [BF95] is a planning system based on a structure called Planning Graph. A Planning Graph is a directed graph represented by levels, each one composed by two layers. Figure 2.1 shows the general structure of GraphPlan. Starting at level 0, the even layers contain information about the propositions that are reached — *fact layers*. The odd layers contain actions whose preconditions are present at the previous level — *action layers*. The edges represent the precondition and effect relations between facts and actions. This graph is much smaller than state transition graph, specifically, only polynomial in size and can be efficiently constructed for many planning problems. So in the worst case, it still takes an exponential time to extract a plan from the planning graph.

GraphPlan involves two states: disjunctive progression stage and plan extraction stage. Next subsections will explain them both in detail.

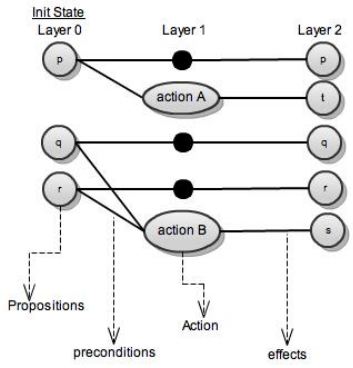

Figure 2.1: GraphPlan General Structure

2.2.1 Disjunctive progression stage

This phase builds the planning graph until all goals are reached. In level 0, the fact layer is made up of the initial state (*InitState*) problem facts (in Figure 2.1 is represented with the letters p, q ,r) , and the action layer contains all actions applicable to *InitState* (in the example is showed as *Action A* and *Action B*). The pseudo actions are also represented in level 1 and they are showed as a black point into the action layer. This process is repeated until a fact layer is reached that contains all goals.

During the layer-by-layer construction stage, another kind of important information about whether two fact literals can be in the same state and whether two actions can be executed concurrently is computed. This information is called mutex.

We say that exists a mutex relationship between two actions if any of the next conditions happen.

1. *Inconsistent effects*: An effect of one action is the negated literal of the other one. Figure 2.2 shows when this situation occurs.

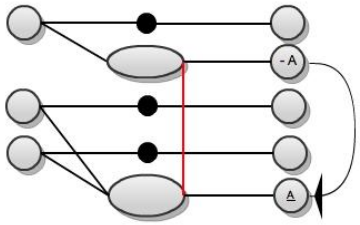

Figure 2.2: Inconsistent Effects Example

2. *Interference*: An effect of one action is the negated literal of a precondition of the other one as shown in figure 2.3

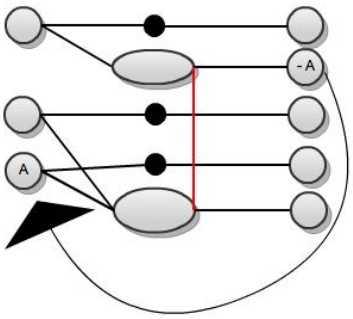

Figure 2.3: Interference Example

3. *Competing needs*: A pair of preconditions of both actions is mutex in the previous fact layer. Figure 2.4 shows an example of this situation.

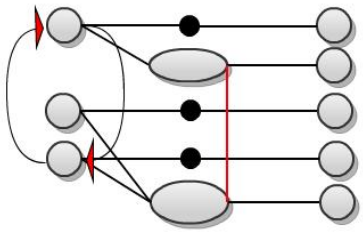

Figure 2.4: Competing Needs Example

A pair of fact literals is defined to be mutex if any of these conditions is accomplished.

1. *Negated literals*: An effect of one action is the negated literal of the other one. Figure 2.5 shows a pair of fact with a *Negated literal* mutually exclusive relation.

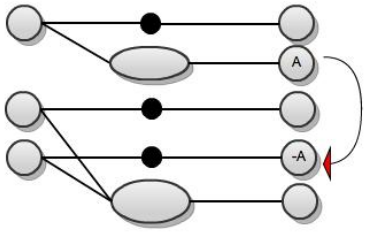

Figure 2.5: Negated literals

2. *Inconsistency Support*: Every pair of actions in the previous operator layer that achieves the literals is mutex. In figure 2.6 we illustrate this condition

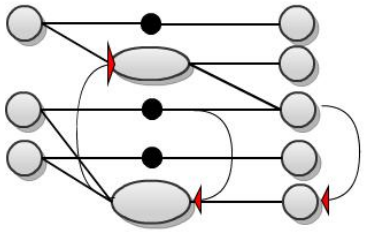

Figure 2.6: Inconsistency support

2.2.2 Plan extraction stage

Once the planning graph is completed , we can search for a solution. If the planning graph has stabilized but the goal facts are not achieved, we cannot extract a solution plan. If goal conditions have been satisfied and each pair of goal literals is not mutex, GraphPlan uses a backward-chaining strategy level by level. The process consists of: given a set of objectives in t, where t is the last graph level, find a set of actions in level *t-1* at reach these goals and there are not mutex between other actions previously selected. Then, the preconditions of these actions form a set of sub goals in *t-1*. If the new objectives of *t-1* can be gained in *t-1* steps, then the original objectives can be obtained in t steps. If it is impossible to reach them we change the considered action combination. If the solution is found, the planning graph yields a layered plan, else it continues to expand until stabilized.

2.2.3 A brief example

The example below is very simplistic but it allows us to make things more clear. The problem is defined in table 2.1:

Table 2.1: GrahpPlan Example

Actions:	
a	:preconditions(p) :effect(and (r) (q))
b	:preconditions(q) :effect (and (s) (not p))
Initial state:	(and (p) (q))
Goal:	(and (r) (s))

Applying the process previously described, we obtain the Planning Graph showed in figure 2.7:

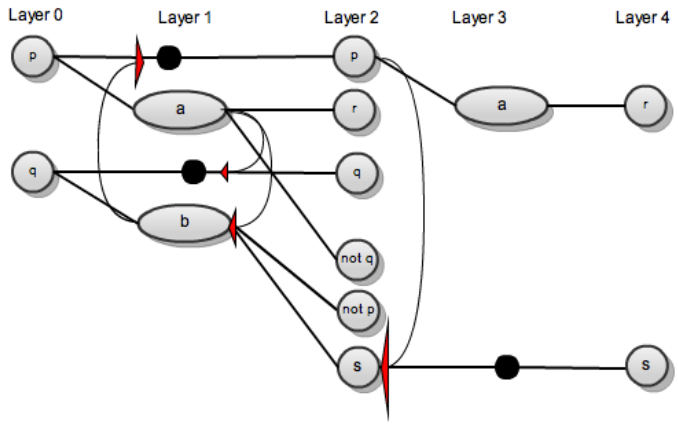

Figure 2.7: GraphPlan example: Note that for simplicity, some of the actions in layer 3 are omitted

Firstly, we observe the initial state facts placed in level 0. In level 1 the actions which preconditions are satisfied on initial state (a and b) and the pseudo-actions are located in it. A pseudo-action (also called "no_operation") is represented in the figure as a black point located in an action layer. It means that no action is performed over the previous layer fact linked, in consequence, this appears into the next fact layer without changes. In level 1 we find mutex relationships between a and b by *Interference*. In addition we see two more mutex relations: First one appears between p pseudo-action ("no_operation_p") and b, and second one between q pseudo-action and a. The plan extraction process is called when the graph is stabilized (this happens in layer 4). From the last level, it tries to find a set of non-mutex actions. This set contains the action a and the pseudo-action to maintain p ("no_operation_p"), to support the goal r. Then, the process makes p and s the sub-goals to achieve. After that, it determines the action b and the pseudo-action to maintain p to be the supporting actions set. In consequence a resulting valid plan is b, a.

2.3 Plan Recognition

As we have previously mentioned, AI Planning systems [HTD90] can be defined as the operation of selecting an order set of activities that the agent should perform to achieve one or more goals and respecting a set of environment constraints. In this way AI Plan Recognition systems [Car01] are very related to AI Planning systems but the point of view of how they approach the problem is different. Plan Recognition systems reason over the activities that the agent has previously performed and try to detect their goals or their not observed activities needed to complete the set of activities of the plan.

2.3.1 Introduction to Plan Recognition

Plan recognition (or activity recognition) is the inverse operation of AI Planning [NGT04, Kau91]. It aims to recognize the actions and goals of one or more agents from a set of tasks performed by a group of monitored agents. These agents modify the environmental conditions thanks to their behaviour, then these changes in the conditions should be also consider by the recognizer. This way of analysing the problem is called "abduction", and the conclusions obtained "explanations". Then, plan recognitions techniques can be considered a powerful tool for the problems we need to know the possible intentions of the agent while this one is trying to achieve it.

The plan recognition process requires at least two agents participants. The first one is called the "actor" and its goal is try to solve a problem making a set of ordered activities interacting with the environment. This role is played by the agents in planning and scheduling systems. The other one is called "observer" agent. An observer is capable to infer the goals pursued by the actor analysing his actions. In addition of the inputs required by a planning process (PDDL Domain file and PDDL Problem file) another input is needed in the plan recognition problems. This new input is an "observation" file. The content of this file is a list of observations with the actions performed by the actor and detected by the observer.

The relationship established between an actor and an observer generates a classification for the plan recognition problems [Wae96]:

- *Cooperative systems*: The observer uses the information extracted from the actor to help him to achieve their goals.
- *Competitive systems*: In these systems observers and actors have incompatible goals. The observer tries to get their goals which implies obstructing the plan of the actor.
- *Recognizer systems*: Actors don't need to take into account the observer. The observer only monitors or studies the possibilities derived from the actors tasks but it doesn't perform actions to change the environmental conditions. In this sense we can say that both independently develop their activities.

In all cases, both actors and observers work with many elements [AA07] that we consider very important to mention here to get a well understanding of the techniques described in this project, that is:

- *Actions*: are atomic operations performed over the agents' interface. They don't involve necessary changes in the environment, but a concrete sequence of them will be treated to generate a Task.
- *Tasks*: are atomic operations with meaning for the modelled domain. They are composed as a result of the execution of one or a set of basic actions, which are intended to get a common goal. These tasks could produce effects on the environment and change the domain status.
- *Goals*: are a set of environment conditions that one (or more) agents desire to achieve.
- *Plans*: are sequences of tasks the agents perform to accomplish their goals from an initial environment state.

2.3.2 Plan Recognition Systems

Perhaps the most well-known of the early plan recognition systems are those of Robert Wilensky [Wil78] and James Allen [AP86]. Wilensky's system was developed for story understanding, and the inference path served as an explanation for the occurrence of an action *A1* in the story. Wilensky's system selected the shortest inference path connecting an observed action to an expected goal, without considering the current focus of attention in the story. Allen's system assumed that an agent had one of a small number of top-level goals which could be deduced from a single utterance. Another researchers propose plan recognition systems using hierarchical model of plans [KA86]. These systems have many limitations related to the needs for input the plans library that the systems were able to recognize.

Other approaches has extended it introducing an AND/OR graph as the plan library [Les98]. However, such algorithms suffer the problem of acquisition and hand-coding of a large plan library. In order to reduce the manual library plan generation and face more realistic scenarios, some authors have attempted to use machine learning techniques [Bau98], pattern recognition using Bayesian inference [AZNB97], probabilistic reasoning, and partially observable Markov decision process (POMDP). However, other formalisms have been proposed for dealing with the uncertainty inherent in plan inference, most notably formal argumentation models and approaches based on probabilistic reasoning. Several researchers have captured plan recognition in a formal model of argumentation or abduction. Charniak and Goldman [CG91] constructed the first Bayesian plan inference system.

Some recent approaches have used techniques from classical planning into planning recognition [SY07, RG09]. But classical planning assumes that actions are known and deterministic, and the world is always perfectly known. The only changes allowed to occur in the environment are due to the actions the agent decides to execute. Unfortunately, agents acting in the real world usually face a highly dynamic and partially observable environment. Consequently, their beliefs about the current state of the world are limited, uncertain or simply incorrect.

2.3.3 Planning graph based architecture advantages

We consider a graph based architecture [Hon01] the best choice to work as engine of our intelligent tool proposal. The plan recognition system we present in this project uses the possibilities offered by the modern plan synthesis methods, specifically GraphPlan [BF95]. The systems based on planning graphs are very powerful and they provide many advantages:

- *Velocity*: Graph based systems are faster recognizing agent's goals than others systems based on plan libraries.

- *Non-limit recognizable plans*: GraphPlan based systems don't need a hand-coding plan library to determine the possible plans to recognize, this feature allow us to detect new plans without adding them in explicit way. Otherwise, with plan library systems (i.e. hierarchical systems) the task to introduce new plans in the recognizer is a very cost process.

- *Planning graph as a tool to store information*: The dynamically built graph provides agent's candidate plans. Running the nodes of the graph we can detect relevant information such as, incompatible actions, observed actions or reachable goals.
- *Domain Independent*: This system can be used in many applications, for example commercial intention recognizer, pattern behaviour modeller, space, etc

However this kind of recognizers usually generates also some constraints:

- *Fully observable domain*: Actions happened in the scenario are always observed by the agent.
- *All the actor's actions are relevant*: All the actor's actions influence the observed plan and the set of determined goals.

2.3.4 Research fields in Plan Recognition

Once defined the main features of the plan recognition techniques, we are going to mention the fields which these techniques have been successfully implemented.

Since the 1980s, this research field has captured the attention of several computer science communities due to its strength in providing personalized support for many different applications and its connection to many different fields of study such as medicine, human-computer interaction or sociology. Despite the importance and ubiquity of plan recognition techniques, there has not been a great deal of computational research on this problem.

The research areas where plan recognition techniques have been applied getting interesting results are, for instance, *military simulations and defences issues*. These fields have quickly evolved using different approaches such as incorporating learning machines with intelligent agents in the development of mental models of other agents. An example of this approach is a system capable to recognize executing plans of the combat aircrafts [AFFH86], providing the ability of customizing the behaviour of the aircraft in function of the actions taken by the opponent, trying to anticipate their movements thanks to this behaviour recognition system. The intrusion detection process is a field wherein progress in plan recognition techniques have been well received [GG01]. The ability to identify hostile agents by their behaviour is a powerful tool. This is possible because we don't search the actor responsible of the attack, but rather the actions composing the attack. Another interest application field is the prevention and early identification of terrorist activities analysing suspicious tasks and making hypotheses from other sources of information [JFLLM05].

A different direction taken by plan recognition research is the development of intelligent user interfaces. The approaches in this topic are very diverse. Some researchers try to predict the user preferences using quantitative approaches to create the possible plans and plan recognition techniques to identify them [Bau98] but the current trend is focused on identifying the user's behavior when he interacts with the system. Furthermore this capability to model users has attracted the interest of many commercial sectors producing significant results in domains such as prediction of costumer's commercial intentions [HYL08], students modelling and human-computer interaction [Wae96].

In general, plan recognition techniques have been applied in many different areas with diverse goals, getting good results. So what should we do? The answer is simple: we need to design architectures that allow us to implement intelligent features based on plan recognition techniques. This new point will automatically provide new functionalities, new security systems and customizable interfaces according to user's preferences and profiles.

Chapter 3

JPlanRecognizer a Tool to Recognize Goals

In this chapter we address the main goal of this project: The development of a Plan Recognizer application in Java, based on planning graph techniques. We call this application *JPlanRecognizer*. First, we define in detail the main elements that are part of the application, offering a static design view of the tool through a class diagram. Then, we offer a description of the main methods of the application and we propose our own extension. All definitions and procedures are supporting by an operation example over a simplistic domain to clarify the process. Finally, we show the output for the proposed domain generated by *JPlanRecognizer*.

3.1 Preliminary Definitions in Plan Recognition Problems

Like GraphPlan, our method applies to STRIPS-like planning domains. In these domains, operators have preconditions, add-effects and delete-effects, all of which are conjunctions of propositions, and have parameters that can be instantiated by objects in the world. The two involved agents in a plan recognition problem ("actor" and "observer") have the *same knowledge* about the planning domain. In addition all observed actions can be incomplete or unrelated but they have to be *correct*. It means that if one observed action is accepted like input is because it has really happened. Furthermore the environment is *static*, it means that only the actor's actions are capable to change the planning environment. Finally, we assume an *explicit notion of discrete time step*. Then, although the observed actions are incomplete and the observer doesn't know how many actions have been executed, it always knows how many time steps there are and he knows in which time step a given action is in.

In a planning problem, a valid plan is formed by a set of actions and specified times in which each one is carried out. It takes into account that two or more actions can occur at the same time step satisfying only a rule: they cannot be mutually exclusive. From an initial state, we execute a set of actions that achieve the goal conditions satisfied at the

final time step. In our plan recognition problem, for partial observations, the definition should be slightly changed. In a plan recognition problem, at a given time step, we are not always sure that whether a given proposition is true, whether a given action is really executed, and whether a given plan is really valid. So we introduce several concepts to represent propositions, actions, and plans [SY07].

- *u-proposition*: While a proposition in a planning problem is an element used to describe the world, in plan recognition an *u-proposition* is a clause with the form *(p, k)* being p a planning proposition and k an element from the set $K \in \{0, 1, -1\}$. k is called the value of the *u-proposition* and it denotes if the *u-proposition* p is unknown ($k = 0$), true ($k = 1$) or false ($k = -1$) in a concrete state.

- *u-state*: A set of *u-propositions*. It's a partial description of the world.

- *u-action*: An action is a representing set of state transitions, in consequence a *u-action* consists of preconditions, a set of *u-propositions*, characterizing the state where the action is applicable in, and another set of *u-propositions* as the effects. For a *u-action* (a, k), we say k is the value of the *u-action*. If its value is 1, it represents that the action has really happened; if its value is -1, it represents that it didn't happen for sure; if its value is 0, it represents that whether it happened is unknown.

- *u-goal*: We name *u-goal* to a map from the set of *u-propositions*, which describes the goal conditions to $K = \{-1, 0, 1\}$. We say a *u-goal's* value is 1 if and only if all the u-propositions in the set have value 1. We say a *u-goal's* value is -1 if and only if at least one of the u-propositions in the set has value -1. For the rest of the cases the *u-goal's* value is 0.

- *u-causal link*: Let a_i and a_j be two *u-actions* at time steps i and j respectively, where $i < j$. There exists a *u-causal link* between a_i and a_j, if and only if one of the effects of a_i has the literal p, and the execution of the a_j requires p. If there exists a *u-causal link* between a_i and a_j, we write it as $(a_i < a_j, k)$, where $k \in K, K = \{-1, 0, 1\}$. If both actions have value 1, then the *u-causal link* is set with $k = 1$. If at least one of the actions have the value -1, the k-value is $k = -1$, otherwise the *u-causal link* will be set with $k = 0$.

- *u-valid plan*: Let g be a *u-goal* recognized, with k-value 0 or 1. How to reach g from the initial state s_0 is formalized as a set $\{A, E, C, k\}$, which formalizes a valid plan. A is a set consisting of *u-actions*, E denotes the set of *u-casual link* relations in A and we store in C the temporal and logical constraints. Finally k represents the value of the plan. A plan is valid if and only if the *u-goals* are achieved after the actions are executed from s_0 in an order consistent with C and the plan's value is 0 or 1. A plan's value is defined as following the same rule that the others "u-" elements. If all the *u-casual link* and g have the value 1, then the plan's value is 1; if at least one of the u-causal link or g has the value -1, the plan's value is -1; else the plan has the value 0.

- *u-planning graph*: It is a graph alternatively layered. First layer (as level 0) and the other even levels contain nodes with propositions. Propositions are boolean formulas

from first-order logic that refer that one aspect is accomplished in a concrete state in the environment. Then, the graph proposition nodes describe the set up of the domain in a determinate time, so we set in the level 0 the initial state. Then, we locate the action nodes into odd levels. Actions are the way to change the status of the environment by an agent. An action requires that its preconditions (propositions that represents the necessaries conditions to allow the agent to execute this action) have been located in the previous proposition layer to be set in an action layer. Furthermore, the action nodes are connected with their preconditions and effects (propositions that represents modifications that occurs in the environment after the action's execution) nodes by edges. Action layers include actions and pseudo-actions. These last nodes are linked to a previous layer proposition node with the same name but adding the prefix "*nop_*". These actions don't modify the state, they just keep the state of the proposition to the next proposition layer. The only difference between the *u-planning* graph and a planning graph is that each proposition node is labelled with some *u-proposition* rather than some proposition, and each action node is labelled with some *u-action* rather than some action.

Once we have described the main elements of our approach, we show in figure 3.1 the high level static design implemented for our application *JPlanRecognizer*. For simplicity we have deleted the prefix "u-" for the concrete classes of the class diagram (Action, Proposition, PlanningGraph, etc), but we have respected it for the *abstract* classes to avoid problems with the names if we decide to implement or to instance elements from a Planning Graph application in the future.

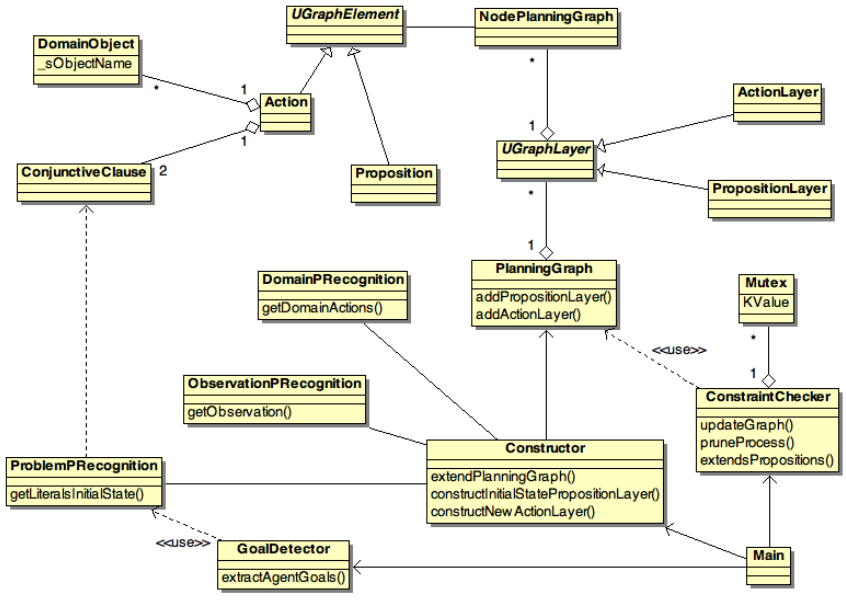

Figure 3.1: Class diagram of JPlanRecognizer

As we can observe in figure 3.1, the main class of the model is the *PlanningGraph* class. It is composed of many elements extended from *UGraphLayer*. This abstract class allow us to work with action layers as the same way that if it were a proposition layer. This design decision has been also applied to *UGraphElement* class to make the application more flexible.

Each layer consists of elements called *NodePlanningGraph*. The objects of this class can encapsulate actions or propositions and each one stores the information about the *"edges"* linked with the element encapsulated. For instance, if we analyse a node that encapsulates an action, this *NodePlanningGraph* records the effects and the preconditions compatibles with the actions, it means there are edges (implemented using references) between the considered *NodePlanningGraph* and the nodes that cover its preconditions and its effects in previous and next graph proposition layers respectively.

We notice that the *u-state* element is represented in the application with the class *Conjunctive Clause*. The instantiated objects of this class allow us to manage first-order-logic formulas, which facilitates the work with this kind of structures.

The inputs of the recognizer are obtained through special classes used like interface:

- *Domain PDDL file*: This input is parsed and managed by an instantiated object of the class *DomainPRecognition*. This file processes the PDDL domain and generates objects using the classes *Action, Proposition, UGraphElement, DomainObject and ConjunctiveClause*. The method responsible of this task is "getDomainActions". For simplicity, we have not represented this "use" relationship in the diagram.

- *Problem PDDL file*: To extract the initial state and the possible goals of the observed agent, *JPlanRecognizer* implements the class *ProblemPRecognition*. This class uses an object from *ConjunctiveClause* class to represent the initial state and the goals states.

- *Observed PDDL file*: While the two previous inputs are common to planning problems, this one is exclusive of the plan recognition problems as mentioned in chapter 2. In this case the class in charge of creating the list of observed actions is *ObservationPRecognition* through the method *getObservation*.

In the next section we explain more in detail other important classes presents in the model such as *Constraint Checker* or *Mutex*. These classes are very related to the constraint process and we consider more appropriated to describe them together. In addition the class *GoalDetector* will be also explained in following sections. This class encapsulates the functionality in charge of interpreting the results offered by the recognizer.

3.2 Description of the algorithm

The architecture of the developed system consists of four components as we can see in figure 3.2:

- *Acquisition Data*: This module encapsulates the classes related to information extraction from PDDL files as we have mentioned in the previous section. The main component of this module is an external jar library called *PDDL4J*, which is available at *http://sourceforge.net/projects/pdd4j/*.

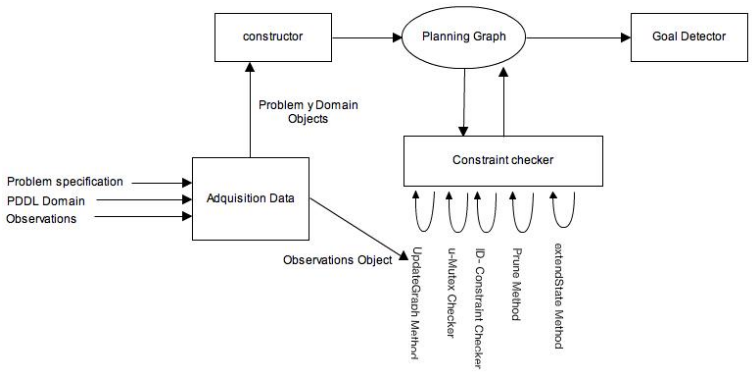

Figure 3.2: Main modules of JPlanRecognizer

- *Constructor*: It is responsible for generating the Planning Graph object using the domain and the problem information from the PDDL files by the *Acquisition data* module. Unlike GraphPlan, the *Constructor* only builds a partial *u-planning graph*. It means that the module just adds one action layer and one proposition layer at one time step.

- *Constraint Checker*: This module assumes responsibility for updating the planning graph according to the observation (or observations) given as an input in each time step. This task is performed by the methods (*UpdateGraphMethod*, *u-Mutex Checker*, *ID-Constraint Checker*, *PruneMethod* and *ExtendStateMethod*) showed in figure 3.2. We explain in detail these method in next subsections.

- *Goal Detector*: It analyses the planning graph built when the other modules have finished their tasks. *Goal Detector* compare the propositions present in the last proposition layer of the *u-planning graph* with the possible proposed goals in the problem specification file. If the module determines that some goals are achieved, it finishes the plan recognition process and sends an output with the goal or goals reached.

As we can see, our approach is different from [SY07] because of we use their techniques but we cannot purse the same goals. While the objective of [SY07] is to deduce the plan followed by the agent, we are more interested in detecting their goals as soon as possible. For this reason, we do not need the *Plan extractor* module that they require. Furthermore we update some methods and introduce others to help us to early detect the agents goals

The high-level description of the implemented algorithm is shown in figure 3.3 through a sequence diagram. Looking at the picture we distinguish three stages:

- *First stage*: The static class *Main* makes an instance of each one of the following classes related to the *Acquisition data* module; *ProblemPRecognition, DomainPRecognition and ObservationPRecognition* obtain three objects called “problem”, “domain” and “observation” respectively. These objects extract and store the informa-

tion about the plan recognition problem from the input files as we have previously mentioned. Afterwards, the *Main* class invokes objects from other two classes; *Constructor* and *ConstraintChecker* using as parameters the objects "problem", "domain" and "observation" as we see in figure 3.3.

- *Second stage*: Once the five objects are created the second stage begins its work. In this stage the constructor creates an *u-partial planning graph* using the method *extendPlanningGraph*. This method uses the initial state provided by the object "problem" to create the first layer of the graph (if it is the first time is called). In addition it adds two more layers: an action layer and another proposition layer, so we obtain a partial *u-planning graph*. This partial *u-planning graph* is then fed by the object from the class *ConstraintChecker* which calls the method *UpdateGraph* to update the *u-planning graph* with the observed actions, provided by the object "observation". Then, the methods *U-Mutex Checker*, *ID-Constraint Checker*, *PruneMethod* and *ExtendStateMethod* are invoked, as we show in 3.3, to keep the coherence among the planning graph and the updated actions through the observation process. We explain in detail these methods in the next subsections. This stage-cycle runs until the observations list is empty. This repetitive task is represented by the block "loop" in the diagram.

- *Third stage*: Finally, we create an object from the class *Goal Detector* and call the method *extractAgentGoals* using the final u-planning graph as a parameter. This method checks the last proposition layer and it will execute the appropriated runtime.

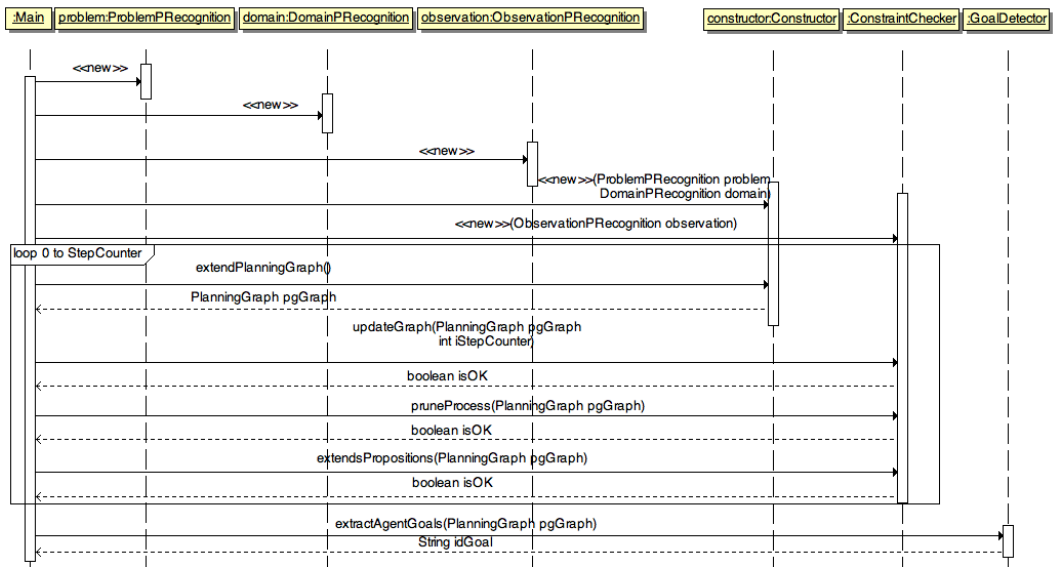

Figure 3.3: Sequence diagram of Main method

This runtime has not been implemented in the recognizer because it will depend on the tool where the recognizer will be installed. For instance, if we want to monitor a rover, this module should implement the procedures that the rover has to execute in an emergency case or if the rover is driven to a hole it has to change the direction. Note that the extended planning graph used to represent propositions, actions and goals is dynamically constructed, so we do not suffer the problems of hand coding of large plan libraries. This is one of the reasons we have chosen a planning graph implementation for our plan recognizer.

To explain more in detail the different stages of the algorithm we are going to use a simplified example to illustrate each of the involved processes. In table 3.1 we present a domain where a rover located in Mars has to obtain some rock samples and take some pictures.

This domain describes a simplified problem in which a rover is located over the Mars surface and it has to perform some tasks. First, it has to get pictures of the environment using a camera, and it also has to analyse the composition of the rocks thanks to a detector. The rover cannot move but it can check the correct working of the two instruments at any time. The rover has to check duly at least one instrument each day. This condition is represented with the proposition "maintenance". Once the rover checks any instrument, this one is disabled (dirty or "not cleanDetector" for the rock composition detector and if the rover checks the camera, this one will be "not pointing") but the rover has no maintenance tasks pending for this day (so, the proposition maintenance will be negated or "not maintenance"). For simplicity, there are no parameters in this problem.

The initial state of the problem is that the detector and the camera are clean and pointing to the target and no check process has been perform on the current day. Note that positive literals that do not appear in the initial state description are assumed to be false. The rover can perform the following actions (see table 3.1).

- *analyzeRock*: The precondition is that the detector is clean and the effect is that the rock is analysed.
- *takePicture*: The precondition is that the camera is pointing to the target and the effect is that the picture has been obtained.
- *checkDetector*: This action has no preconditions and the effects are that the rover does not need to perform more maintenance tasks during the day and, in consequence the detector is dirty due to the test process performed.
- *calibrateCamera*: The precondition list of this action is empty (the rover can perform calibrateCamera action at any time) and the effects are that the rover has performed the daily check task and it does not need do it again during the current day. In addition, the camera is not pointing to any target.

There are three goals in the problem. The first is that the rover gets a rock analysis, a picture and it has not any maintenance task pending; the second is that the rover wants to get a picture and to perform some maintenance tasks, but it has not taken a rock sample yet; the third is that the rover only wants to take a rock sample and perform at least one maintenance task ("checkDetector" or "calibrateCamera"). Once defined the actor agent behaviour is the turn of the observer; he knows that the rover is analysing a rock

Table 3.1: A basic Mars Rover Domain. For simplicity there are no parameters

Actions:	
analyzeRock	:preconditions (cleanDetector) :effect (analizedRock)
takePicture	:preconditions (pointing) :effect (picture)
checkDetector	:preconditions: :effect (and (not cleanDetector) (not maintenance))
calibrateCamera	:preconditions: :effect (and (not pointing) (not maintenance))
Initial state:	(and (cleanDetector) (pointing) (maintenance))
Goals:	
g1:	(and (picture) (not maintenance) (analizedRock))
g2:	(and (picture) (not analizedRock) (not maintenance))
g3:	(and (not picture) (analizedRock) (not maintenance))
Observations:	
step 0: step 1:	analyzeRock takePicture

in first step and detects that the rover is taking a picture in the second step. What is the goal the agent is trying to achieve? In the rest of the chapter we will use this domain to illustrate each procedure implemented by our recognizer and demonstrate that the goal pursued by the agent is "g". Note that our plan recognition system allows the agent to perform many actions at the same time.

3.2.1 Acquisition Data

The *Acquisition data* module aims to obtain the necessary inputs to execute the recognizer from PDDL files. We difference two parts in this module. The first one is an

external jar library called *PDDL4J*. *PDDL4J* is an open source library to facilitate java implementation of planners based on PDDL. The library contains a parser of the last version of PDDL. This library allow us to manipulate files in PDDL format and to extract information from them. The second part aims to convert the pddl data obtained using the library, in classes and objects of our application. This design in two layers facilitates the change of the parser used (in our case *PDDL4J*) for a new one in the future. So our recognizer has a partial independence of the parser used to work with PDDL. As we can see in figure 3.1 the application has many classes to store the information related to the domain and the problem to solve. We explain them next:

- *DomainObject*: The created objects of this class represent the instantiated objects of the problem. In figure 3.1 for simplicity we only show the main attribute, called *_sObjectName* but there are also others important attributes as *_sType* (to work with PDDL domains using the requirement *:typing*) or the *_sIdObjectGeneric* used to perform the action instantiation process.

- *DomainPRecognition*: This class stores the information of the actions available in the domain. All action names are stored in the object instantiated from this class, in addition the preconditions and the effects of each action are also represented in this object.

- *ProblemPRecognition*: This class provides to our application, the elements from the PDDL problem file. That is, the initial state, the PDDL objects and goals. The objects from this class have mainly two important methods called *getLiteralsInitialState()* and *getOperators(String sType)*. The first one returns a conjunctive clause representing the initial state of the problem, and the other one returns an arrayList with the operators instantiated in the problem file of the type passed as parameter.

- *ObservationPRecognition*: This class aims to provide the actions that the observer detects. This observation is introduced to the application using PDDL syntax. The main elements stored are the total number of steps and an *ArrayList* with the observations seen by the recognizer.

We have explained that we use the PDDL4J external library to extract information from PDDL files. We have also mentioned the classes of our application that store the data provided by the PDDL4J library, however we don't know, what is the bridge between these two layers. This bridge is the class *PDDLFilesParser*. This class (it does not appear in figure 3.1 for simplicity) aims to extract all important data from the PDDL files using the classes provided by PDDL4J and encapsulate this information in objects of the classes we have just described.

As we are analysing a simplified version of the Mars Rover Domain without parameters, the *Acquisition data* module does not use the class *DomainObject* because there are no parameters in the example. The other classes (*DomainPRecognition*, *ProblemPRecognition* and *ObservationPRecognition*) store and process important data, for instance actions with its effects and preconditions, initial state, goals and observations.

3.2.2 Constructor

As we have mentioned in the algorithm description, the first module that will be called once the *Acquisition data* module has obtained the domain and problem inputs is the *Constructor*. In figure 3.4 we show the sequence diagram that represents the behaviour of the method named *extendPlanningGraph*. This method is available to the objects instantiated from the *Constructor* class. The figure illustrates the relation among *Constructor* class and the classes *DomainPRecognition* and *ProblemPRecognition*, which obtain the recognizer inputs as we have mentioned in previous subsection. We also observe that, as mentioned in the high level description of the algorithm, this method is called at each iteration of the algorithm as we show in figure 3.4 using the "alt" block. This block represents the next behaviour:

- The first time that the method is called, it obtains the initial state propositions from the object "problem" of the class *PropblemPRecognition*. Then it creates a new instance named "pgGraph" of the class *PlanningGraph*. In addition, this first time, the method builds a proposition graph layer with the initial state propositions and it stores this layer in a variable named *plNewLayer*. This task is performed by the method *constructInitialStatePropositionLayer* of the *Constructor* class. Note that the obtained proposition layer from the method *constructInitialStatePropositionLayer* contains all their propositions with the $k - value = 1$, because we are sure these propositions are true at the time to start to solve the plan recognition problem.

- Other times it does not create the "pgGraph" object, so the method only gets the last proposition layer of the graph. A reference of this layer is stored in a variable named *plNewLayer*. In this case the method called is $get\_LayerFrom\_alLevelGraph$, which is defined in the *Constructor* class.

Once we have set some values to the variable *plNewLayer*, the next step followed by the method is to add this new layer to the graph. We represent this, in the sequence diagram plotted in figure 3.4, with the method *addPropositionLayer* of the object "pgGraph". Then, a new action graph layer is made and added to the partial planning graph using the methods *constructNewActionLayer* of the *Constructor* class and *addNewActionLayer* of the object "pgGraph". The method *constructNewActionLayer* must create a layer with all actions whose preconditions are satisfied in the previous proposition layer. The method also includes in the layer, pseudo-actions and actions without preconditions. In consequence all the actions in this layer have to be linked with, at least, one precondition of the previous layer, except for actions without preconditions. This task is performed by the method *setEdgesBetweenNewLayer* which links the nodes setting edges between each pair of nodes that require it.

Finally, the method generates a new propositions layer. This layer has to be compatible with the actions of the last action layer, what it means that all propositions must be the effects of the actions located in the previous action layer, including pseudo-actions. This implies that all propositions in this proposition layer have to be linked with at least one action of the previous action level. To perform this last task the methods called are *constructNewGenericPropositionLayer* and *addNewPropositionLayer* as we display in figure 3.4.

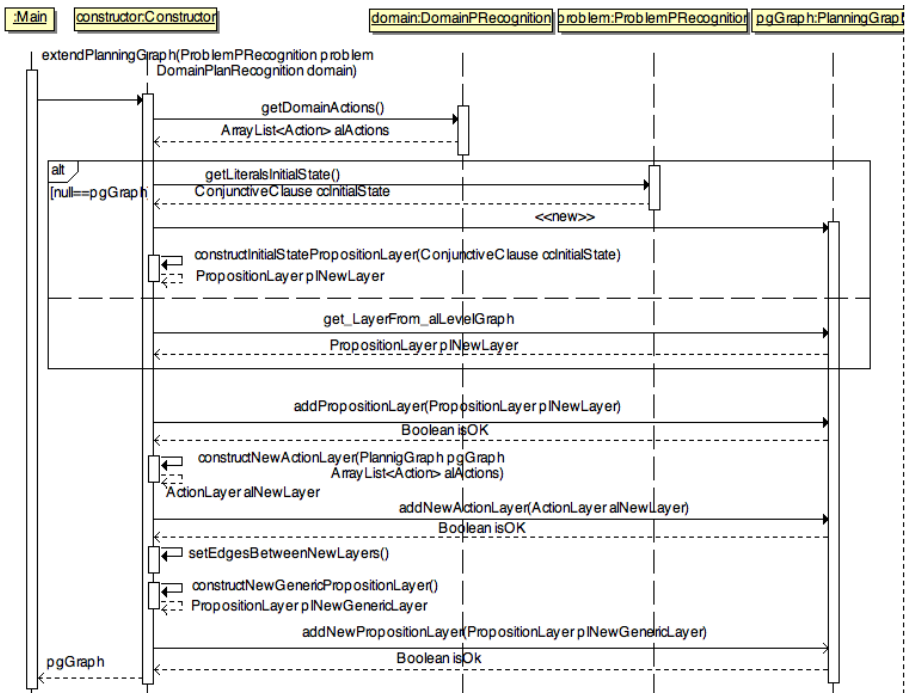

Figure 3.4: Sequence diagram of the extendPlanningGraph method

The difference between extending a *u-planning graph* in plan recognition problems and extending a planning graph in planning problems is that, our graph is only extended to the next proposition layer while a planning graph will be extended until it is stabilized or the goal conditions are all satisfied. Furthermore we don't do any mutex relations computation while constructing the *u-planning graph* because we delegate this task to the *Constraint Checker* module.

In figure 3.5 we show the *u-planning graph* obtained using the Mars rover problem of Table 3.1.

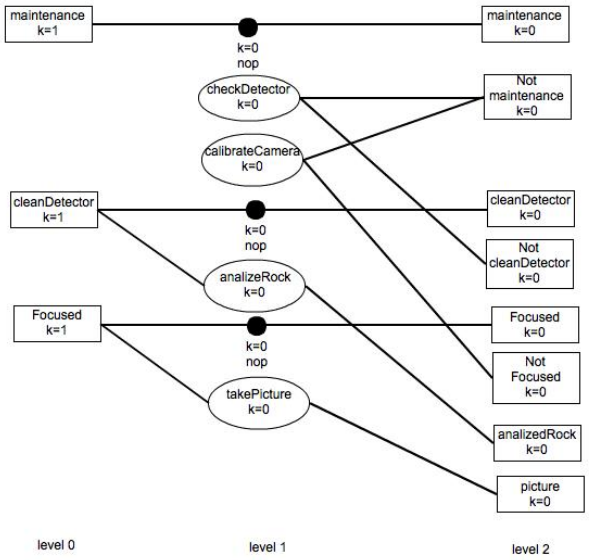

Figure 3.5: *PlanningGraph* produced by *Constructor*

Analysing the figure 3.5 we note that the level 0 has been created by the method *constructInitialStatePropositionLayer*, because all the nodes in that layer have its k-value set to 1. In level 1 we find actions which preconditions are in level 0 (e.g "analizeRock"), pseudo-actions (showed as black points) and actions without preconditions ("checkDetector" and "calibrateCamera"). In this stage all of them must have their k-value set to 0. Level 2 consists of nodes that are effects of the actions at level 1. The black lines represent edges linking actions nodes with propositions nodes and vice versa.

3.2.3 Constrain Checker

This module is responsible of detecting the wrong nodes in the graph to keep the consistency among true nodes and possible or discarded nodes. To achieve this task we use the mutex relations between the nodes of the planning graph. This technique was introduced in chapter 2 for a *GraphPlan* planning example. The mutex relations in a plan recognition problem are sightly different to the mutex definitions mentioned in chapter 2. For this reason we introduce a new property for the mutex: the *mutex k-value*.

We say that exists a mutex relationship between two actions into a **u-planning graph** if any of the next conditions are accomplished.

1. *u-Inconsistent effects*: An effect of one action is the negated literal of the other one and its k-value is defined as follows: For any mutex of this kind found in the *u-planning graph* its k-value will be 1 if and only if (iff) at least one of the two actions

has the k-value set to 1; It will be 0 iff both actions have the value 0; otherwise the value of the mutex is -1.

2. *u-Interference*: An effect of one action is the negated literal of a precondition of the other one. The k-value of these kind of mutex will be set following the next rule. The k-value will be 1 iff at least one of the two actions have the k-value 1; It will be 0 iff both the actions have the value 0; in other case the value of the mutex is -1.

3. *u-Competing needs*: A pair of preconditions of both actions is mutex in the previous fact layer. The rule followed to define the mutex k-value in these mutex is the same as the other cases. The k-value will be 1 iff at least one of the two actions have the k-value set to 1; It will be 0 iff both actions have the value 0; in other case the value of the mutex is -1.

We also can find mutex between proposition nodes into a **u-planning graph**, so we say that two propositions are mutually exclusive if any of the next situations are met.

1. *u-Negated literals*: An effect of one action is the negated literal of the other one. The k-value will be 1 iff at least one of the two actions have the k-value 1; It will be 0 iff both actions have the value 0; in other case the value of the mutex is -1.

2. *u-Inconsistency Support*: Every pair of actions in the previous operator layer that achieves the literals is mutex. The k-value will be 1 iff at least one of the two actions have the k-value 1; It will be 0 iff both actions have the value 0; in other case the value of the mutex is -1.

The mutex relations are a very powerful allied tool to reduce the possible goals pursued by an agent, because for each action observed we will delete all actions incompatible with it at the step time studied. Their effects and preconditions will be pruned too, in consequence the graph will be decreasing their size observation by observation, and the possible goals will be bounded.

With the mutually exclusive relations adapted to plan recognition problems we pass to describe the process performed by our implemented recognizer.

Once a partial planning graph is constructed at any time step, the plan recognizer checks the observation input to find whether there is any action observed at that time step. When an action is observed, the constraint checker is called to update the values of the nodes in the planning graph. The method invoked is *updateGraph*. In figure 3.6 we show the sequence diagram that represents the behaviour of this method. In the diagram we present the objects implicated in the process. First one, the object "checker" is an instance of the class *ConstraintChecker* and it is invoked from the Main method of the application. The second one is the object from *PlanningGraph* class created by the *Constructor* and called "pgGraph" as we have mentioned in previous subsection. The first stage of the method displayed in the picture requests information about the two last layers of the object "pgGraph" using the method $get\_LayerFrom\_alLevelGraph$. Once the checker receives these layers, it starts the second stage consisting of the invocation of two methods:

- *updateTheActionsObserved*: This method reads all the observed actions for a concrete time step. These observations are provided by the object of the class *ObservationPRecognition*, which we mentioned in the subsection about the *Acquisition Data* module. Once the method has a list with the actions, it searches them in the action layer passed as argument. Then, when a layer node with an action in the observations list is found, the method sets to 1 the k-value of the node. When an action is in the observation list, it means that this action has actually happened and in consequence we update it with $k - value = 1$ in the graph.

- *u-mutexCompute*: This method performs the different types of mutex computations for a plan recognition problem. It creates a data structure to store objects of the class *Mutex*. This class is showed in figure 3.1. The objects from this class built by this method encapsulates a k-value according to mutex definitions for a *u-planning graph* mentioned at the beginning of the subsection. In this sense, we will find in the structure mutex relation objects with values 1,0 or -1 in their k-value fields.

The third stage stars when the mutually exclusive relations are computed and stored into a data structure. As we can see in figure 3.6 this stage consists of the execution of two methods called *u-mutexConstraintChecker* and *ID-ConstraintChecker*. These ones are enclosed into a loop displayed in the diagram as a "while" box. The task of the methods is to update the k-values of the planning graph nodes following a set of rules. The loop is responsible to repeat the execution of these methods until the graph is stabilised, in other words, until no node is modified during an iteration. The rules followed by these methods to update different nodes are explained in detail in the next two subsections.

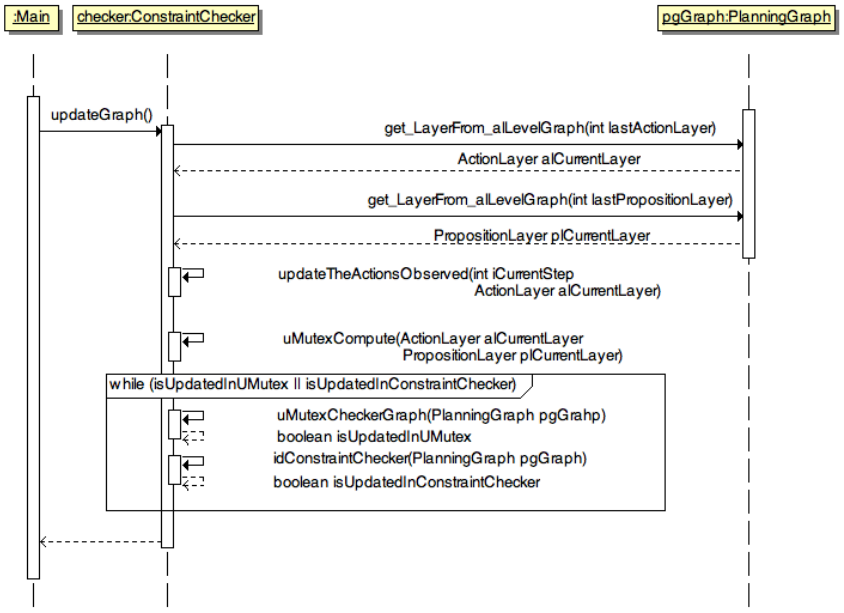

Figure 3.6: Sequence diagram of *updateGraph* method

To illustrate the method *updateTheActionsObserved*, we show its effects in the example proposed in table 3.1. As we see in the example, the observed action in the first time step is "analizeRock". Then, this method updates with the k-value to 1 the node "analizeRock" in the level 1 of the graph. This update is displayed in red color. In consequence their effects and their preconditions are also updated, so the obtained graph is showed in figure 3.7.

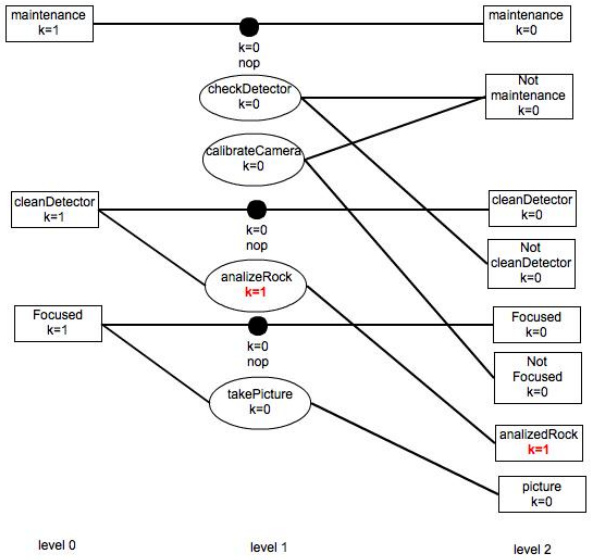

Figure 3.7: Partial *u-planning graph* updated with an observation

When all observed actions in the current time step are updated, the next step is to detect and delete all actions and propositions incompatible with the action observed. To perform this task we introduce two new procedures *u-mutex checker* and *ID-Constraint checker* which we describe in the followings subsections.

3.2.4 u-mutex checker

The main idea of the method *u-mutex checker* is to use the mutex relations we have defined to reason over the *u-planning graph*. Given a partial u-planning graph as the one shown on figure 3.7, the method *uMutexCompute* computes all the mutex in the current action level (level 1) and the next proposition level (level 2). Once it has finished, it performs a backward checking with the previous pair of levels to update the mutex relations. In this case, it does nothing because the current action level (level 1) is the first action level of the graph. In figure 3.8 we show with red lines all mutex relations between each pair of nodes found in the partial planning graph that appears in figure 3.7.

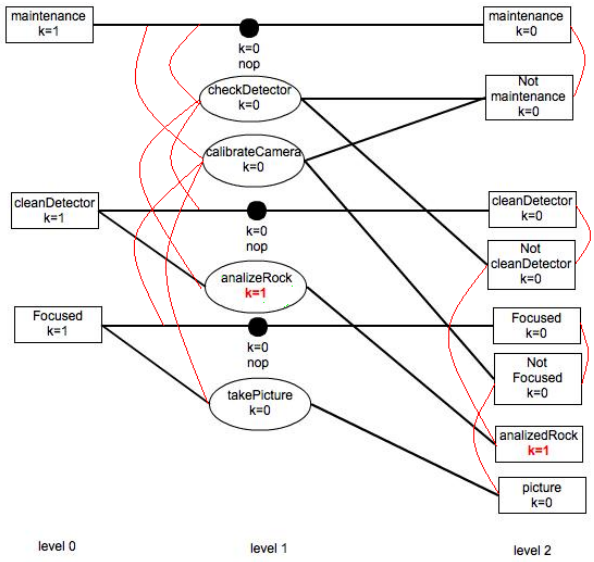

Figure 3.8: Method uMutexCompute example

Analysing figure 3.8, in level 1 obviously we don't find mutex between these propositions because we know all of them are true proposition in the initial state. In level 2 we apply the mutex definitions provided. Since level 2 is an action level we compute mutex relations of the next categories *u- Inconsistent effects*, *u-Interference* and *u-Competing needs*. In consequence we find six mutex relations. For instance we find an *u-Inconsistent effects* mutex between "checkDetector" and "nop_maintenance" with k-value =0 or the *u-Interference* mutex between "checkDetector" and "analizeRock". In this level we cannot detect mutex of the type *u-Competing Needs* because the previous level is the initial state. Advancing to the level 3 we compute the mutex into a proposition layer, thus we only compute the next type of mutex: *u-Negated literals* or *u-Inconsistency Support*. Examples of the first type are the mutex between "maintenance" and "not maintenance", "pointing" and "not pointing" or "cleanDetector" and "not cleanDetector". All of them have the k-value = 0. In the other case we find two more mutex as we see in the picture. The most relevant of them is the mutex between "not cleanDetector" and "analysingRock" because the k-value for this mutex is 1 according to the definitions previously mentioned.

After computing all mutex relations in the partial planning graph, the method *u-mutex checker* runs through the list of mutex and when it finds something with k-value=1, it gets the action or proposition implicated in the mutex with k-value=0 and updates it with the value -1 in the graph as we see in figure 3.9.

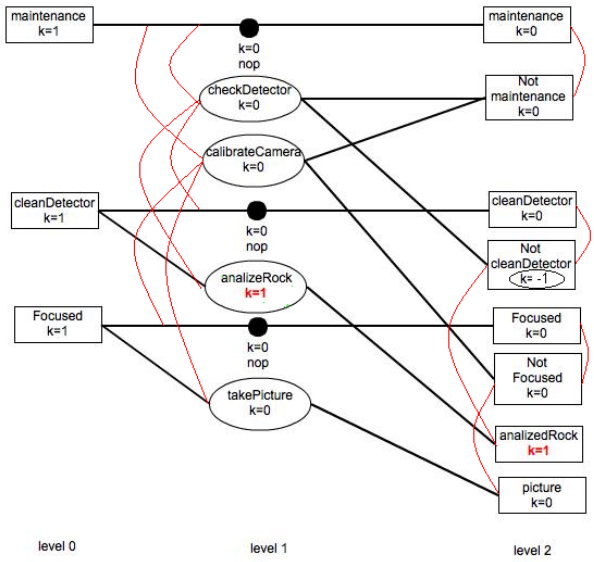

Figure 3.9: u-mutex checker assign k-value=-1

Looking at figure 3.9 we observe that the node "not cleanDetector" is mutex with "analizedRock" in level 2. As "analizedRock" has k-value=1, the mutex relation between them is also set to $k - value = 1$, then the method *u-mutexChecker* gets the node "not cleanDetector" and updates its k-value to -1. Having updated the nodes in the proposition level (level 2), the method performs the same task with the previous action level. Then it updates the action node "checkDetector" with $k - value = -1$, because this node is mutex with the observed action "analizeRock".

3.2.5 ID-Constraint checker

The mutex relations defined are incomplete, in fact, it is almost impossible to find all the mutually exclusive relationships because determining all mutex is as expensive as finding a legal plan. Fortunately the mutually exclusive relations implemented can help us to prune the graph. To advance in the prune process of the graph far away of the *u-mutex checker* has done, we introduce four constraints definitions to keep the coherence between the partial planning graph and the observations that it has received.

- *I-action constraint*: For each action, if one of its precondition literal or effect literal is found to have value -1, the action is impossible to happen.

- *I-proposition constraint*: For each action with p as one literal of its precondition in a previous action level or p as one literal of its effects (pseudo-actions included), with $k - value = -1$, we deduce that p is impossible to happen so we set to -1 its k-value.

- *D-action constraint*: Let p be a proposition with value 1. If there is only one action where p appears in its precondition or in its effect, the action is determined to happen.
- *D-proposition constraint*: For each action with k-value = 1, all the literals that appear in its preconditions and its effects are set to the value 1.

Once we have defined the I and D constraints for actions and propositions we next explain the algorithm.

1 The first step of the *ID-constraint checker* is the same as the *u-mutex checker*, it means that we need to execute *u-mutex checker* before calling the *ID-constraint checker*.

2 According to the *I-proposition constraint*, we update the last proposition level. We use the *I-action constraint* to refresh the k-values of the last action layer. After that, the method processes the current action level assigning the value 1 to the nodes that achieve the *D-action constraint*. Finally, the literals that appear in these actions' preconditions and effects are also updated according to *D-proposition constraint*.

3 If no action's value is updated in step 2, it continues to check previous pair of proposition and action layer, otherwise go back to step 2.

4 Repeat the steps above until the value of each node in the *u-planning graph* does not change any more.

The whole procedure seems to cost much, but in fact we only need to update the values of those propositions with $k - value = 0$ and those actions with the $k - value = 0$. To show the performance of this method we need to advance to the next iteration of our Mars Rover example, because in this first time step any node accomplishes the conditions checked by the *ID-Constraint Cheker*. In consequence the planning graph obtained at the end of the first time step iteration is presented in figure 3.10. The only difference with the planning graph shown in figure 3.9 is that the nodes evaluated with $k - value = -1$ have been deleted from the planning graph. This task is performed by a generic method previously mentioned called *PruneMethod*, for this reason the nodes "checkDetector" and "not cleanDetector" have been deleted from the picture.

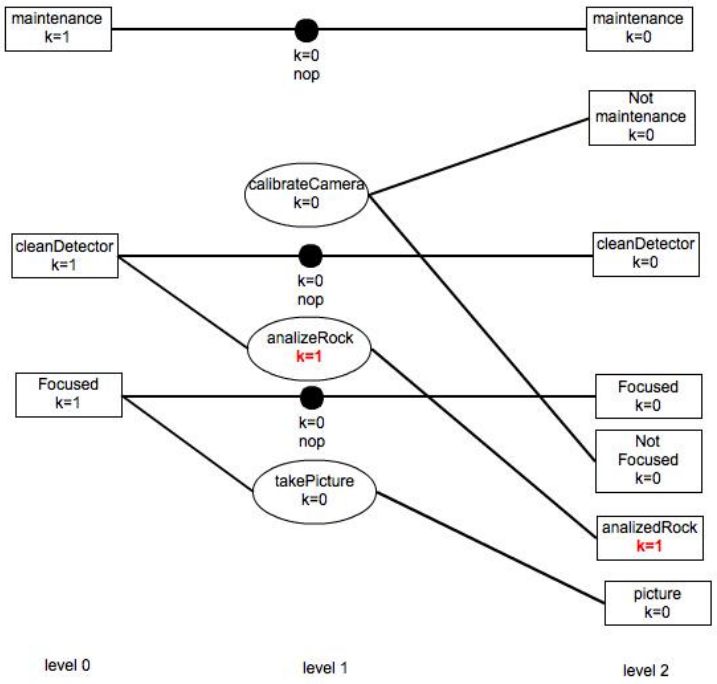

Figure 3.10: Planning graph obtained at the end of the first iteration of the process

In figure 3.10 we show the graph obtained after all constraint processes. We see that we have deleted the "checkDetector" action applying *I-action constraint*. In addition the process marks with the *I-proposition constraint* the proposition "not cleanDetector" in level 2 with k-value=-1. If we advance to the next iteration and we update the graph with a new observation: "takePicture". It produces that through *D-proposition constraint* the proposition "focused" in the level 2 is set with $k - value = 1$. This can be checked in figure 3.11.

3.2.6 ExtendState Procedure

Comparing our approach and the implementation from [SY07], many differences arise due to the different goals they pursued. While [SY07] use the recognizer to complete a partial plan sequence, we are only interested to limit the possible goals achieved by the agent. For this reason we have discarded two modules (*the guesser* and the *plan extractor*) as we have mentioned in previous chapters, and we add a new procedure called *extendState Method*.

We introduce this method because we have observed that applying only the previous constraint methods we find actions with k-value = 0 (mainly pseudo-actions) that link two equals propositions with different k-values (1 in the minor level and 0 in the next one). Obviously if no action with k-value = 0 is mutex with this pseudo-action, this one

should be labelled with k-value = 1. In consequence, the other linked proposition will be labelled with 1 by the other previous constraint methods in the next iteration.

In figure 3.11 we show an example of how this method runs. In this case the observation of the first step is "analyzeRock" and the observation for the next step is double. We observe the actions "checkDetector" and "takePicture" at the same time in the step number 2. Their nodes are updated with k-value=1 and in the next proposition layer "not maintenance", "not cleanDetector" and "picture" are also updated with k-value=1. At the end of this iteration the extendState method is called to update the pseudo-action "nop_true_analyzeRock". This action is mutex with any other action at the same level evaluated with 0, then this pseudo-action has sure happened (in color red in the picture). Consequently, the "analyzedRock" proposition in the level 4 will be updated with k-value = 1.

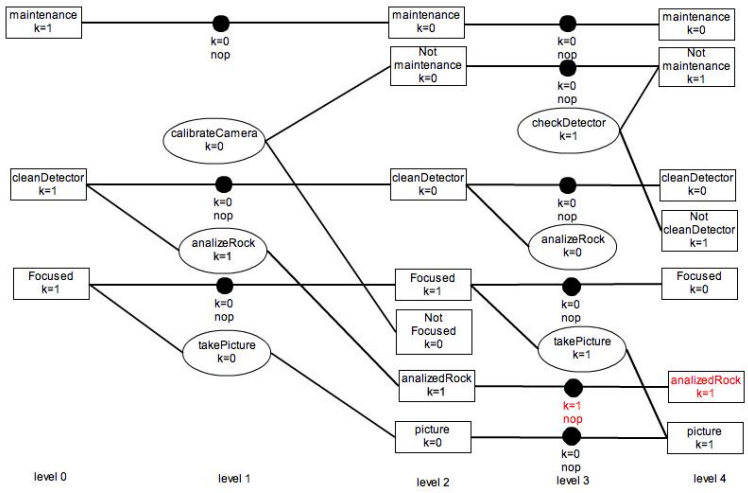

Figure 3.11: *ExtendState* method example

All procedures of the recognizer have been explained in this chapter. In the next one, we show that our approach can be used in several domains, for instance we will use it to detect the on-line marketing intention of the social networking site members. So to conclude, we show in figure 3.12 the output produced by *JPlanRecognizer* for the Mars rover domain showed in table 3.1. In the figure we see the k-values of the nodes when the algorithm arrives to the last observation input of the problem. The nodes with $k - value = -1$ are not shown because they have been pruned by the prune method mentioned previously. In this step our system calls the *Goal Detector* module which determines that one of the sets of goal pursued by the agent (in this case "g1") is in the last level achieved by the recognizer.

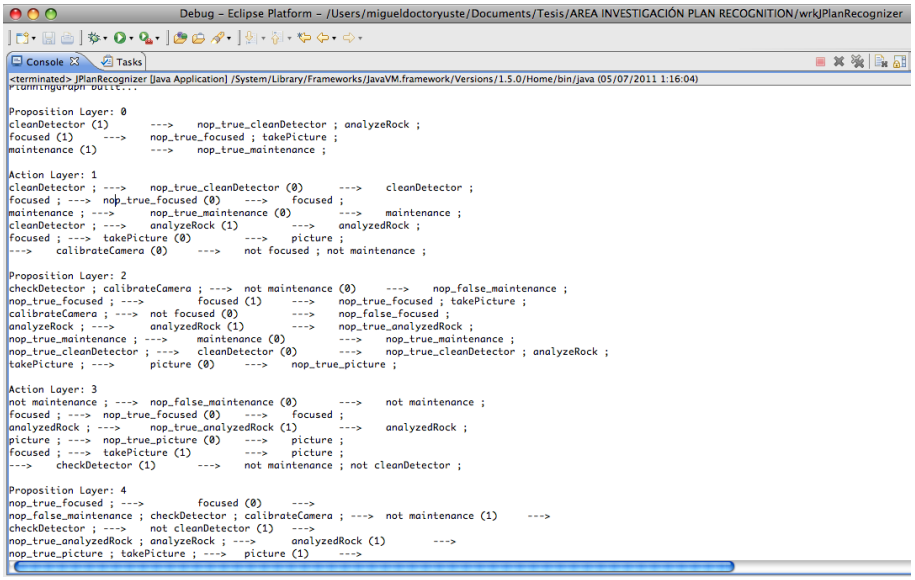

Figure 3.12: Planning graph generated by *JPlanRecognizer* for the Mars rover example

Chapter 4

Complete Operation Examples

In this chapter we will show our *JPlanRecognizer* working at well-known PDDL domains. Then, we will explain the results generated by our application and how we can reduce the possible goals to achieve if we increase the number of the observed actions. Furthermore, we will probe that our approach is valid to recognize goals in domains of different areas, so we have developed a PDDL domain that represents the behaviour of the costumers into a social Networking Site.

4.1 Agent Cleaner Domain

The *Agent Cleaner Domain* is a simplified version of the famous Shakey's world domain [FN71]. We have developed this domain to test our recognizer into a simple and easy scenario. In this domain we have a robot able to move around a set of rooms. This robot can open the doors and clean the dirties rooms. To move between two rooms, the robot needs a door connecting them. In addition, the robot needs to open the door and clean the room if it was dirty, before leaving the room.

4.1.1 Domain and Problem

Our simplified version of the *Agent Cleaner Domain* and the problem to solve consists of:

- Six predicates:
  - *door ?x ?y*: It indicates that exists a door between the room *?x* and the room *?y*.
  - *door_open ?x ?y*: It means the door between the rooms is open.
  - *door_closed ?x ?y*: This predicate represents that the door is closed.
  - *agent_in ?x*: It indicates that the robot is located in the room *?x*.
  - *dirty ?x*: When a room is dirty we will use this predicate.
  - *clean ?y*: As we have previously mentioned the rooms have two possible states: dirty or clean. Then, this predicate represents that the room have been cleaned.

- Five available actions for the robot.
  - *to_clean ?x*: This action is used by the robot to clean a dirty room. The robot needs to be located in the room *?x*, which is dirty. The room *?x* will be clean when this action was executed.
  - *to_open_the_door_pull ?x ?y*: This action will be called by the robot when the door between the rooms *?x* and *?y* was closed, and the robot was in the room *?x*.
  - *to_open_the_door_push ?x ?y*: This action is similar to the previous action but in this case, the robot is in the room *?y* and it wants to open the door to access to the room *?x*.
  - *go_to_next_room ?x ?y*: When the room is clean, and the door is open, the robot can leave the room *?x* to arrive to room *?y*.
  - *go_to_previous_room ?x ?y*: If the robot is in the room *?y* and it wants to arrive to *?x*, it will call this action instead of *go_to_next_room*.
- Two rooms called *h1* and *h2*. These rooms are dirty and they are connected through a closed door.
- The initial situation shows the robot located in the room *h1*, with the door closed.
- The goal pursued by the agent is to arrive to the room *h2* after to cleaning the room *h1*.

The complete PDDL domain and problem files are presented as follow:

```
(define (domain ag_cleaner)
        (:requirements :strips :equality)
        (:predicates (door ?x ?y)
                     (door_open ?x ?y)
                     (door_closed ?x ?y)
                     (agent_in ?x)
                     (dirty ?x)
                     (clean ?x))

    (:action to_clean
        :parameters (?x)
        :precondition (and (dirty ?x)
                           (agent_in ?x))
        :effect (and (clean ?x)
                     (not (dirty ?x))))

    (:action to_open_the_door_pull
        :parameters (?x ?y)
        :precondition (and (door ?x ?y)
                           (agent_in ?x)
                           (door_closed ?x ?y)
                           (clean ?x))
        :effect (and (door_open ?x ?y)
                     (not (door_closed ?x ?y))))

    (:action to_open_the_door_push
        :parameters (?x ?y)
        :precondition (and (door ?x ?y)
                           (agent_in ?y)
                           (door_closed ?x ?y)
                           (clean ?y))
        :effect (and (door_open ?x ?y)
```

```
                    (not (door_closed ?x ?y))))
    (:action go_to_next_room
        :parameters (?x ?y)
        :precondition (and (door ?x ?y)
                           (agent_in ?x)
                           (door_open ?x ?y)
                           (clean ?y))
        :effect (and (agent_in ?y)
                     (not (agent_in ?x))))
    (:action go_to_previous_room
        :parameters (?x ?y)
        :precondition (and (door ?x ?y)
                           (agent_in ?y)
                           (door_open ?x ?y)
                           (clean ?y))
        :effect (and (agent_in ?x)
                     (not (agent_in ?y)))))


(define (problem problem_cleaner)
        (:domain ag_cleaner)
        (:objects h1 h2)
        (:init (agent_in h1)
               (door h1 h2)
               (dirty h1)
               (dirty h2)
               (door_closed h1 h2))
        (:goal (and (clean h1)
                    (agent_in h2))))
```

Once we have explained and shown the domain and the problem developed, we will test this domain using a planner called *Blackbox* [KS98]. *Blackbox* is a planning system that works by converting PDDL problems into Boolean satisfiability problems (SAT), and solving them with a variety of state-of-the-art satisfiability engines. The results offered by the planner are shown in figure 4.1.

We can see that the goal is achieved in three steps performing the actions: *to-clean h1, to-open-the-door-pull h1 h2*, and *go-to-next-room h1 h2*, then, this is the plan that our monitored agent has intention to perform. Now it's time to test the effectiveness of our recognizer through two experiments over this problem. These experiments are performed in the next subsection.

4.1.2 Experimentation

First we are going to demonstrate that *JPlanRecognizer* keeps the possible predicates to achieve in the last proposition level thanks to the method (added to modify the initial architecture proposed in [SY07]) *extendsProposition* as we have mentioned in the previous chapter. Then, we propose to know the possible goals achieved by the agent with only one observation (or observed action) happened in the first iteration: "to-clean h1". The total steps considered are three to facilitate the well understand of the example:

- First Step: The recognizer extends the graph with the possible actions. These actions have to be compatible with the initial state and with the action "to-clean h1" observed by the recognizer. Then, the method *extendsProposition* extends the k-values of the propositions indicating which of them will sure happen. In figure

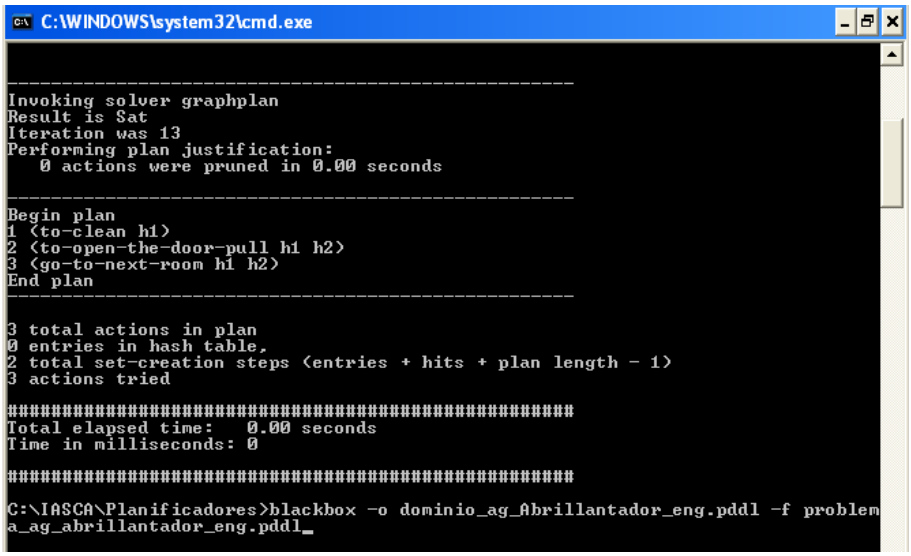

Figure 4.1: Results offered by BlackBox for the agent cleaner problem

4.2 we observe that the last achieved proposition level guarantees that the room h1 will be clean (and "not dirty" obviously). The rest of the propositions are sure (with the $k - value = 1$) because there are no possible actions to be mutex with them. It is also interesting to observe that the pseudo action "nop_true_dirty h1" will be pruned due to its $k - value = -1$. It is obvious because we have observed the action "to_clean h1" and they are mutually exclusive.

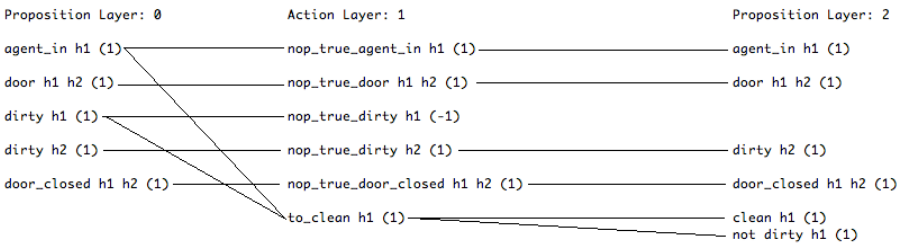

Figure 4.2: Results offered by *JPlanRecognizer* for the agent cleaner problem: First step

- Second Step: In this step, the recognizer does not receive any observed action. Then the recognizer set as possible all the actions which preconditions are satisfied in the second proposition layer. In consequence the final proposition level in this step is shown in figure 4.3. In this level we see that the only action that the agent can perform is "to_open_the_door_pull h1 h2" but we are not sure if the action has happened, for this reason we find in the picture three propositions with

their k-values = 0. The figure 4.3 shows that the k-value of the pseudo action "nop_true_door_closed h1 h2" is not propagated by the method "extendsMethod" because if the action "to_open_the_door_pull h1 h2" has happened, the door will be open. In consequence, the proposition "door_closed h1 h2" sets its $k - value = 0$. The rest of the process continues to maintain the uncertainty about this step.

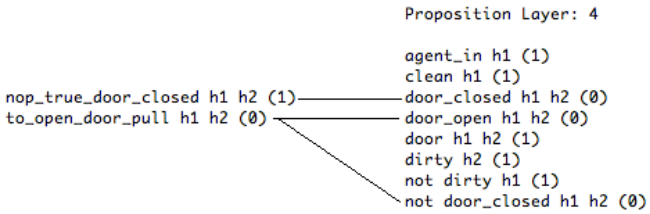

Figure 4.3: Results offered by *JPlanRecognizer* for the agent cleaner problem: Second step

- Third Step: In the last step, the recognizer offers in their last proposition level all the possible predicates that the agent can achieve. The propositions with $k - value = 1$ are mandatory achieved by the agent. The propositions with $k - value = 0$ are possible but not sure. In consequence, any combination of the set of propositions with $k - value = 1$ with any subset composed with propositions with k-value=0, is likely to be the goal agent intention. In figure 4.4 we show the last level of our graph, and we see that our application predicts that with the information received, at the end of three time steps: The room *h1* will be clean ("not dirty" in consequence), in addition our recognizer assure us that the room *h2* will be dirty and that there is a door between *h1* and *h2*. In addition, *JPlanRecognizer* says that the mentioned door could be open or closed and the agent could be in any room. If we review the agents intentions shown in the PDDL problem file we check that goal pursued (clean h1 and agent_in h2) is compatible with the results offered by our application (marked in red in figure 4.4).

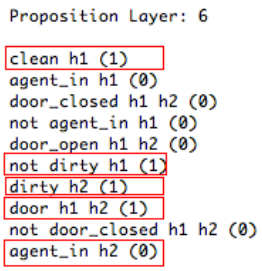

Figure 4.4: Results offered by *JPlanRecognizer* for the agent cleaner problem: Third step

Once we have demonstrated that our application is useful to offer a set of possibles agent intentions, we propose two questions that we will answer through two experiences:

- *Are the results offered by the application related to the total step analysed?*: We mean that if we introduce four steps instead of three as we have done, we will get different results. Observing the figure 4.5, we see that the result generated for the "4-steps" example considers that the room *h2* can be dirty or not, while the "3-steps" example assure that it will be dirty. The explanation is related to the number of actions that the agent can execute. Using only three steps it is impossible to clean the room *h2* from the initial state, because the robot needs to clean the first room, open the door and arrive to the room *h2* to satisfy the preconditions of the action "to_clean h2". In consequence, if we only have three steps, we are sure that the second room will be dirty. But if we have four time steps to perform four actions, we could execute "to_clean h2" when the robot arrives to the second room. Then, in this last example, we cannot know if the room will be clean or dirty.

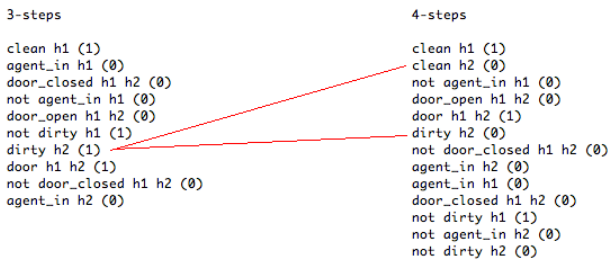

Figure 4.5: Results offered by *JPlanRecognizer* for 3-steps and 4-steps examples

- *Is the recognizer more accurate if the number of observations increase?*: The answer to this question can be obvious: "yes", but we have developed an example which results are presented in figure 4.6. In this figure, we compare the last proposition level obtained by the recognizer, in a three steps execution to one, two or three observations respectively. The observations are the three actions generated by the *Blackbox* planner and shown in figure 4.1. As we can see when we increase the observations, the number of propositions in the last level is always the same or minor. In contrast, with the number of propositions with $k - value = 1$, occurs the same, but in inverse way. Thus, we get more propositions with $k - value = 1$ if we increase the number of observations.

To justify that all experiments have been successfully performed with *JPlanRecognizer*, we present in the figure 4.7 the output generated by the tool for the problem of the three observations used in the last experiment.

```
----- 3 steps with 1 observation -------||------- 3 steps with 2 observations -------||-------- 3 steps with 3 observations ------

     10 propositions/4 with k-value=1              9 propositions/ 6 with k-value=1              8 propositions/ 8 with k-values=1

     clean h1 (1)                                  clean h1 (1)                                  clean h1 (1)
     agent_in h1 (0)                               agent_in h1 (0)                               not agent_in h1 (1)
     door_closed h1 h2 (0)                         not agent_in h1 (0)                           door_open h1 h2 (1)
     not agent_in h1 (0)                           door_open h1 h2 (1)                           not dirty h1 (1)
     door_open h1 h2 (0)                           not dirty h1 (1)                              dirty h2 (1)
     not dirty h1 (1)                              dirty h2 (1)                                  door h1 h2 (1)
     dirty h2 (1)                                  door h1 h2 (1)                                not door_closed h1 h2 (1)
     door h1 h2 (1)                                not door_closed h1 h2 (1)                     agent_in h2 (1)
     not door_closed h1 h2 (0)                     agent_in h2 (0)
     agent_in h2 (0)
```

Figure 4.6: Results offered by *JPlanRecognizer* for different number of observations

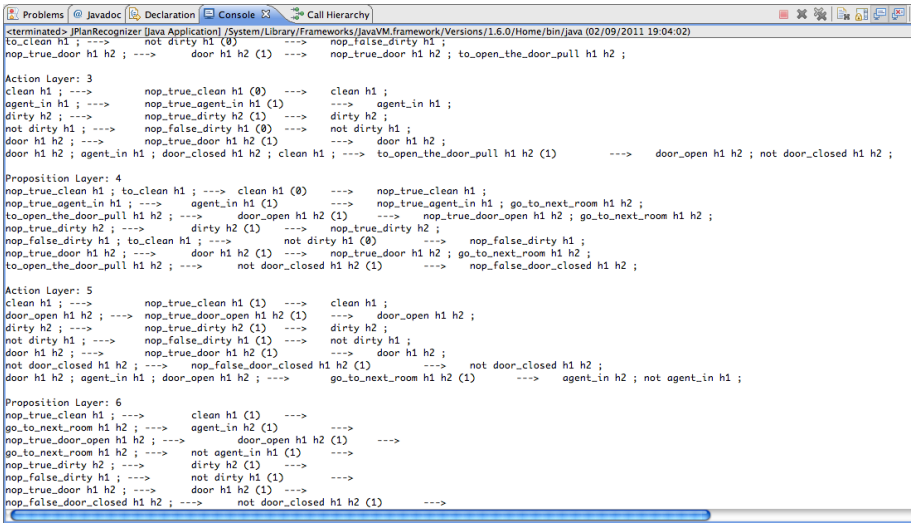

Figure 4.7: Results offered by *JPlanRecognizer* for three observations

4.2 Social Networking Sites

4.2.1 Motivation

Internet is the most powerful communication tool created until now. The variety of channels and methods to exchange information among internauts is as different as web designers and developers can imagine.

In this scenario, a new way of social interaction has born: we are talking about the social networking sites such as Facebook, Twitter, Linkedln or Tuenti. These systems are able to move out all the social experience to the web universe.

These social applications have immensely grown in the last 10 years and many of them are expected to grow to 1 billion participants by 2012 [KNT06]. Recently, many companies from different business areas have demonstrated interest in these platforms to support their business.

The aim of this subsection is to show how *JPlanRecognizer* can be used to the problem of detecting customer's intention, analysing the actions performed in a social networking domain modelled for this project. We have previously treated this problem in [DMMn+11]

4.2.2 Domain and Problem presented

The following PDDL domain is a subset of the domain we have developed to represent a social network environment. The actions and the predicates described here, are necessary to understand the operation example that we will present.

```
(define (domain Social_Network)
        (:requirements :strips :equality)
        (:predicates (online ?user)
                     (friend_of ?user ?friend)
                     (profile_visited ?user ?friend)
                     (chat_enable ?user ?friend)
                     (element_like ?user ?element ?friend)
                     (friendship_request_pending ?emisor ?receptor)
                     (status ?user ?message)
                     (send_privated_material ?emisor ?receptor))

    (:action connect
        :parameters (?x)
        :precondition (and (not (online ?x)))
        :effect (and (online ?x)))

    (:action disconnect
        :parameters (?x)
        :precondition (and (online ?x))
        :effect (and (not (online ?x))))

    (:action visit_profile
        :parameters (?x ?y)
        :precondition (and (online ?x)
                           (friend_of ?x ?y))
        :effect (and (profile_visited ?x ?y)))

    (:action set_private_connection
        :parameters (?x ?y)
        :precondition (and (online ?x)
                           (friend_of ?x ?y)
                           (not (chat_enable ?x ?y)))
        :effect (and (agent_in ?y)
                     (chat_enable ?x ?y)))

    (:action remove_private_connection
        :parameters (?x ?y)
        :precondition (and (online ?x)
                           (friend_of ?x ?y)
                           (chat_enable ?x ?y))
        :effect (and (not (chat_enable ?x ?y))))

    (:action rate_element
        :parameters (?x ?elem ?y)
        :precondition (and (online ?x)
                           (friend_of ?x ?y)
                           (profile_visited ?x ?y))
        :effect (and (element_like ?x ?elem ?y)))

    (:action accept_friendship
        :parameters (?emi ?rec)
        :precondition (and (friendship_request_pending ?emi ?rec)
                           (online ?rec))
        :effect (and (friend_of ?emi ?rec)
                     (not (friendship_request_pending ?emi ?rec))))

    (:action block_friendship
        :parameters (?x ?y)
        :precondition (and (friend_of ?x ?y)
                           (not (chat_enable ?x ?y))
                           (online ?x))
        :effect (and (not (friend_of ?x ?y))))

    (:action request_friendship
        :parameters (?x ?y)
```

```
    :precondition (and (not (friendship_request_pending ?x ?y))
                       (not (friend_of ?x ?y))
                       (online ?x))
    :effect (and (friendship_request_pending ?x ?y)))

(:action update_status
    :parameters (?x ?sms)
    :precondition (and (online ?x))
    :effect (and (update_status ?x ?sms)))

(:action send_privated_material
    :parameters (?x ?y)
    :precondition (and (chat_enable ?x ?y)
                       (online ?x))
    :effect (and (private_material_delivered ?x ?y))))
```

This domain covers the most common tasks that a user can perform in any social networking site, such as Facebook or Tuenti. We present a universe with eleven actions and eight predicates. We will now explain each action an each predicate to understand the final example.

- Eight predicates are proposed in this domain:
  - *(online ?user)*: It indicates that the user ?user is logged into the web site platforms.
  - *(profile_visited ?user ?friend)*: It means that the monitored user (*?user*) has visited the public profile page of an other users represented as ?friend.
  - *(friend_of ?user ?friend)*: This predicate represents that the client ?user has a list of friends that contains the member ?friend.
  - *(chat_enable ?user ?friend)*: It indicates that a private channel of communication (for example a chat window or a webcam connection) is set between *?user* and *?friend*.
  - *(element_like ?user ?element ?friend)*: When a user finds an item (picture, notice, song...) owned by other client (*?friend*) and the user likes this item or he agrees with it, it can express this situation pushing the "I like it" button linked to this item. In consequence everybody will know that the *?user* likes the item *?element* property of *?friend*.
  - *(friendship_request_pending ?emisor ?receptor)*: As we have previously mentioned, the clients of the web site, may be related through a relation of friendship. But to set this link between two users is necessary that someone send to the other one a friendship request. Finally, the receptor of this request can answer in affirmative or negative way. This situation is represented with this predicate.
  - *(status ?user ?message)*: Every user of any social networking site has a public message to show to all his friends. This predicate covers this functionality.
  - *(send_privated_material ?emisor ?receptor)*: Mostly of the members of a social networking site share information with all their friends, but it is also possible to exchange pictures or songs with only one of their friends. To model this case we have defined the current predicate.

- Eleven actions are available for the social networking site users. We explain them in detail as follows:
  - *connect ?x*: This action is used by the user *?x* to access to the social networking site. Then the user will be logged and he will can start to use the site.
  - *disconnect ?x*: This action is used by the user *?x* to leave the social networking site.
  - *visit_profile ?x ?y*: This action could be performed by a logged user *?x* to see the information about one of his friends *?y*.
  - *set_private_connection ?x ?y*: When two friends are logged in, they can set a private chat or a voice call to have a private conversation.
  - *remove_private_connection ?x ?y*: When one user wants to end a private chat with other friend, it can use this action.
  - *rate_element ?x ?y ?z*: If the user *?x* wants to let his friends that he likes something *?y* of the other friend *?z* it will execute this action.
  - *request_friendship ?x ?y*: if the user *?x* wants to add another member of the site *?y* to his friends-list, he needs to execute this action and to wait that *?y* accept the request.
  - *accept_friendship ?x ?y*: It is the action that the user *?x* has to execute to accept the request sent by the user *?y*.
  - *block_friendship ?x ?y*: If the user *?x* wants to delete one friend *?y* from his friend-list, it can invoke this action, in consequence the user *?y* won't be able to visit his profile again.
  - *update_status ?x ?sms*: Action called to update the public message *?sms* of the user *?x*.
  - *send_privated_material ?x ?y*: If the user *?x* wants to send any picture to other member *?y*, he needs to set a private connection channel and use this action.

Once we have explained the domain, we are going to present the recognize problem used for this example[1]:

```
(define (problem problema_SocialNetwork)
        (:domain Social_Network)
        (:objects user1 user2 user3 user4 user5
             statusSMS1 cocheNuevo statusSMS3 statusSMS4
             statusSMS5 picture1)
        (:init
                  (friend_of user1 user2)
                  (friend_of user2 user1)
                  (status user2 cocheNuevo)
                  (online user2)
                  (online user3))
```

[1] This PDDL code is not standard. We have developed an extension to be able to introduce many goals in the same PDDL problem file. In the code we see two possible goals that the recognizer is able to detect "g1" and "g2". We can also see the observations. In this case, we have a four time step observation. At the first and third time step, the recognizer does not receive any input (empty). At the second step it detects that the *user1* is connected. Finally, at the last step, the recognizer detects that the *user1* executes the action *rate_element*.

```
(:goal
     ; goals for g1
          (and (online user1)
               (profile_visited user1 user2)
               (friend_of user1 user2)
               (element_like user1 cocheNuevo user2)
               (status user2 cocheNuevo))
     ; goals for g2
          (and (send_privated_material user1 user2)
               (not (friend_of user1 user2)
               (friendship_request_pending user2 user1)))
(:observation
          (4 (empty) (connect user1) (empty)
          (rate_element user1 cocheNuevo, user2))))
```

- Many users could be instantiated, in our example we have created five: *user1*, *user2*, *user3*, *user4* and *user5*.
- Each user has an item related to its status message. They are respectively: *statusSMS1*, *statusSMS3*, *statusSMS4* and *statusSMS5*. The status message for *user2* is *cocheNuevo*.
- The initial state presents two logged users: *user2* and *user3* with the predicates *online user2* and *online user3*. In addition, *user2* has the message "cocheNuevo" on his status message.
- The problem presented is the next one: We want to know if *user1* pursued the goal "g1" or he wants to achieve the goal "g2". The first one represents that *user1* is interested in the status message of his friend, then we can suppose that he could be interested in buying a car like *user2*. The other one represents a situation where *user1* sents a private material (for instance: a picture or a movie...) and now *user1* does not keep the friendship with him, so the predicate "not (firend_of user1 user2) is part of the goal. In addition, the *user2* continues to send him requests to recovery the access to his profile. In some cases this behaviour could represent an extortion situation, because *user2* has material that he can use to blackmail the *user1*.
- The problem implies the analysis of the problem at least for 4 steps, and we can see in the previous PDDL code that the recognizer only have two observations. The first step is empty or unknown. In the second step it receives that the *user1* is logging and the last step shows us that *user1* has positively rated the status message of *user2*.

4.2.3 Conclusions

The results offered by *JPlanRecognizer* show that the last proposition level has the next predicates with $k - value = 1$: *(online user1)*, *(friend_of user1 user2)*, *(element_like user1 cocheNuevo user2)*, *(status user2 cocheNuevo)* and *(profile_visited user1 user2)*. Thus, "g1" is a possible goal pursued by *user1*. In contrast we don't find the predicate *(send_privated_material user1 user2)* in this last level, which it is a sufficient condition to discard "g2" as goal pursued by *user1*. The output of *JPlanRecognizer* for this problem is shown in figure 4.8.

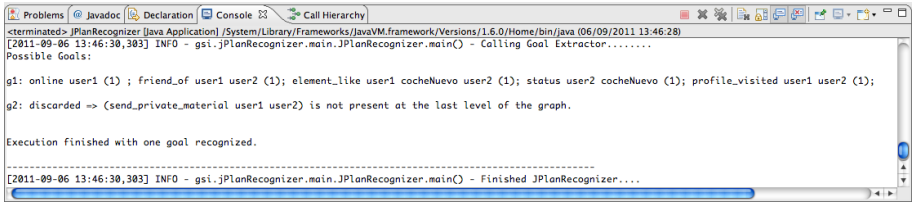

Figure 4.8: Results offered by *JPlanRecognizer* for Social Network problem

In this example we have shown that our recognizer is useful to work with different types of domains that can be modelled and represented with the PDDL language, then it could be considered a powerful tool to predict actions or behaviours of many kind of agents.

Chapter 5

Conclusions

In this chapter we summarize the conclusions of this Master Thesis as well as further future works.

5.1 Conclusions

The Plan Recognition problem has been widely investigated in the past to try to recognize the planning sequence followed by an agent. In these project, we present a plan recognition methodology and an application to incrementally detect the goals pursued by an agent in several contexts such as a Mars Rover domain or a social networking site. We have shown, using very simplistic domains, how these techniques can help us to increase the autonomy of the rovers or satellites, setting mechanisms to monitor their behaviours and detect if the rover has changed the goals set from the Earth. This point opens the mind to develop a new types of rovers whose goals were to surveillance other rovers.

We have also shown how to make that commercial companies improve their advertising strategies offering their products only to costumers that are interested in some topics. For instance, we have developed an example where the system recognizes that one user was interested in the car bought by one of his friends. This situation could be used by a company to offer a sale to that user. Furthermore, our tool could be used to detect extortion cases or bad-uses of the social networking site as we have presented in the same example.

5.2 Future Works

The next steps to advance in the development of *JPlanRecognizer*, point in many different ways:

- *JPlanRecognizer should offer us an answer with probabilistic data*

  Our application can tell us if one goal could be achieved by an agent, but if two or more goals are compatible with the last offered proposition level, the tool cannot tell us if the first one is more likely than the other one. Then, we are going to

modify the application to use the k-value field of the nodes to store probabilistic values instead of values into the set $K = (1, 0, -1)$. In consequence, our recognizer will work with PPDDL [LY04] instead of PDDL.

- *JPlanRecognizer should be tested in a real platform*

  We propose to develop the necessary components in our recognizer to detect the intentions of the users inside a social network platform. We will mainly need two elements: An interface to transform the user actions into PDDL actions and another interface to complete the results offered by *JPlanRecognizer* with recorded data about previous users with the same behaviour.

- *Compare JPlanRecognizer against others recognizers*

  We need to compare our approach against state of the art techniques in plan recognition in different benchmark problems. We will mainly focus on time performance results to predict the goals.

Bibliography

[AA07] Marcelo Armentano and Analía Amandi. Plan recognition for interface agents. *Artificial Intelligence Review*, 28:131–162, 2007. 10.1007/s10462-009-9095-8.

[ACM+10] Gilbert Ahamer, Adrijana Car, Robert Marschallinger, Gudrun Wallentin, and Fritz Zobl. *Heuristics and Pattern Recognition in Complex Geo-Referenced Systems*, pages 300–318. InTech, 2010.

[AFFH86] Jerome Azarewicz, Glenn Fala, Ralph Fink, and Christof Heithecker. Plan recognition for airborne tactical decision making. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, AAAI '86, pages 805–811, Philadelphia, PA, USA, 1986.

[AP86] James F. Allen and C. Raymond Perrault. *Analyzing intention in utterances*, pages 441–458. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986.

[AZNB97] David W. Albrecht, Ingrid Zuckerman, Ann E. Nicholson, and Ariel Bud. Towards a bayesian model for keyhole plan recognition in large domains. In *Proceedings of the Sixth International Conference on User Modeling*, UM '97, pages 365–376, Chia Laguna, Sardinia, Italy, 1997.

[Bau98] Mathias Bauer. Acquisition of abstract plan descriptions for plan recognition. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, AAAI '98, pages 936–941, Madison, Wisconsin, USA, 1998.

[BE08] Danah M. Boyd and Nicole B. Ellison. Social network sites: Definition, history, and scholarship. *Journal of Computer-Mediated Communication*, 13(1):210–230, 2008.

[BF95] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1):1636–1642, 1995.

[BG98] Blai Bonnet and Hector Geffner. HSP: Heuristic Search Planner. In *AIPS-98 Planning Competition*, Pittsburgh, PA, USA, 1998.

[BHKL06] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference*

on Knowledge Discovery and Data mining, KDD '06, pages 44–54, Philadelphia, PA, USA, 2006.

[BVV01] Daniel Borrajo, Sira Vegas, and Manuela Veloso. Quality-based learning for planning. In *Working notes of the IJCAI '01 Workshop on Planning with Resource*, pages 9–17, Seattle, WA, USA, 2001.

[Car01] Sandra Carberry. Techniques for plan recognition. *User Modeling and User-Adapted Interaction*, 11(1–2):31–48, March 2001.

[CG91] Eugene Charniak and Robert Goldman. A probabilistic model of plan recognition. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, AAAI '91, pages 160–165, Anaheim, California, USA, 1991.

[DMMn+11] Miguel Doctor, Ángel Moreno, Pablo Muñoz, Daniel Díaz, and María Dolores R-Moreno. Intelligent social networks. In *Proceedings of the International Conference on Web Intelligence, Mining and Semantics*, WIMS '11, pages 69:1–69:8, Sogndal, Norway, 2011.

[FN71] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

[GG01] Christopher W. Geib and Robert P. Goldman. Plan recognition in intrusion detection systems. In *DARPA Information Survivability Conference and Exposition*, volume 1 of *DISCEX II '01*, Anaheim, California, USA, June 2001.

[GL05] Alfonso Gerevini and Derek Long. Plan constraints and preferences in PDDL3. Technical Report 2005-08-07, Department of Electronics for Automation, University of Brescia, Brescia, Italy, 2005.

[HN01] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:2001, 2001.

[HNC95] Donna L. Hoffman, Thomas P. Novak, and Patrali Chatterjee. Commercial scenarios for the web: Opportunities and challenges. *Journal of Computer-Mediated Communication*, 1(3):29–53, 1995.

[Hon01] Jun Hong. Goal recognition through goal graph analysis. *Journal of Artificial Intelligence Research*, 15:1–30, 2001.

[HTD90] James Hendler, Austin Tate, and Mark Drummond. AI Planning: systems and techniques. Technical report, College Park, MD, USA, 1990.

[HYL08] Derek Hao Hu, Qiang Yang, and Ying Li. An algorithm for analyzing personalized online commercial intention. In *Proceedings of the 2nd International Workshop on Data Mining and Audience Intelligence for Advertising*, ADKDD '08, pages 27–36, Las Vegas, Nevada, 2008.

[JAGS01] James B. D. Joshi, Walid G. Aref, Arif Ghafoor, and Eugene H. Spafford. Security models for web-based applications. *Commun*, 44:38–44, February 2001.

[JFLLM05] Peter A. Jarvis, Teresa F. Lunt, and Karen L. Myers. Identifying Terrorist Activity with AI Plan-Recognition Technology. *AI Magazine*, 26(3):73–81, 2005.

[KA86] Henry A. Kautz and James F. Allen. Generalized plan recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, AAAI '86, pages 32 – 38, Philadelphia, PA, USA, 1986.

[Kan02] Mehmed Kantardzic. *Data Mining: Concepts, Models, Methods and Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[Kau91] Henry A. Kautz. A Formal Theory of Plan Recognition and its Implementation. In *Reasoning About Plans*, pages 69–125, San Mateo, CA, USA, 1991. Morgan Kaufmann Publishers.

[KNT06] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and evolution of online social networks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data mining*, KDD '06, pages 611–617, Philadelphia, PA, USA, 2006. ACM.

[KPL97] Subbarao Kambhampati, Eric Parker, and Eric Lambrecht. Understanding and extending graphplan. *Recent Advances in AI Planning*, pages 260–272, 1997.

[KRRT99] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Trawling the web for emerging cyber-communities. In *Proceedings of the Eighth International Conference on World Wide Web*, WWW '99, pages 1481–1493, Toronto, Canada, 1999.

[KS98] Henry A. Kautz and Bart Selman. BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving. In *Proceedings of the Working Notes of the Workshop on Planning as Combinatorial Search*, pages 58–60, Pittsburgh, PA, USA, 1998.

[Les98] Neal Lesh. *Scalable and Adaptive Goal Recognition*. PhD thesis, Computer Science & Engineering, University of Washington, 1998.

[LY04] Michael L. Littman and H.L.S. Younes. PPDDL1.0: An Extension to PDDL for Expressing Planning Domains with Probabilistic Effects. Technical Report CMU-CS-04-167, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 2004.

[Met95] Robert Metcalfe. Metcalfe's law: A network becomes more valuable as it reaches more users. *Infoworld*, (17), 1995.

[MGH+98] Drew Mcdermott, Malik Ghallab, A. Howe, Craig A. Knoblock, A. Ram, M. Veloso, Daniel S. Weld, and David E. Wilkins. PDDL—The planning domain definition language. Technical Report DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, Connecticut, USA, 1998.

[NGT04] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[PM04] Sankar K. Pal and Pabitra Mitra. *Pattern Recognition Algorithms for Data Mining: Scalability, Knowledge Discovery, and Soft Granular Computing*. Chapman & Hall, Ltd., London, UK, 2004.

[PW92] Scott J. Penberthy and Daniel S. Weld. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning*, KR '92, pages 103–114, Cambridge, MA, USA, 1992.

[RG09] Míquel Ramírez and Hector Geffner. Plan recognition as planning. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI '99, pages 1778–1783, Pasadena, CA, USA, 2009.

[RH95] Brian D. Ripley and Nils L. Hjort. *Pattern Recognition and Neural Networks*. Cambridge University Press, New York, NY, USA, 1st edition, 1995.

[SBF98] Rudi Studer, Richard V. Benjamins, and Dieter Fensel. Knowledge engineering: principles and methods. *Data & Knowledge Engineering*, 25:161–197, March 1998.

[SKK00] Michael Steinbach, George Karypis, and Vipin Kumar. A comparison of document clustering techniques. In *KDD 2000 Workshop on Text Mining*, KDD '2000, Boston, MA, USA, 2000.

[SP09] Ben Shneiderman and Jennifer Preece. The Reader-to-Leader Framework: Motivating Technology-Mediated Social Participation. *AIS Transactions on Human-Computer Interaction*, 1(1):13–32, March 2009.

[SY07] Jigui Sun and Minghao Yin. Recognizing the agent's goals incrementally: planning graph as a basis. *Frontiers of Computer Science in China*, 1:26–36, 2007.

[TG98] Harry C. Triandis and Michelle J. Gelfand. Converging measurement of horizontal and vertical individualism and collectivism. *Journal of Personality and Social Psychology*, 74(1):118–128, 1998.

[VR99] Vincent Vidal and Pierre Régnier. Total order planning is more efficient than we thought. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, AAAI '99, pages 591–596, Orlando, Florida, USA, 1999.

[Wae96] Annika Waern. *Recognising Human Plans: Issues for Plan Recognition in Human - Computer Interaction.* PhD thesis, Royal Institute of Technology, 1996.

[Was94] Stanley Wasserman. *Social network analysis: methods and applications.* Cambridge University Press, 1994.

[Wel99] Daniel S. Weld. Recent Advances in AI Planning. *AI Magazine*, 20:93–123, 1999.

[Wil78] Robert Wilensky. Why John married Mary: Understanding stories involving recurring goals. *Cognitive Science*, 2(3):235–266, 1978.

[YMMK04] Hongyu Yang, Joseph Mathew, Lin Ma, and Vladis Kosse. Matching pursuit feature based neural network pattern recognition of ball bearing faults. In *International Conference of Maintenance Societies 2004*, pages 1–8, Sydney, Australia, 2004.