PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

KIN MAX PIAMOLINI GUSMÃO

MULTIAGENT GOAL RECOGNITION AS SAT

Porto Alegre
2023

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

MULTIAGENT GOAL RECOGNITION AS SAT

KIN MAX PIAMOLINI GUSMÃO

Dissertation submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Felipe Rech Meneguzzi
Co-Advisor: Prof. Dr. Ramon Fraga Pereira

Porto Alegre
2023

Ficha Catalográfica

G982m Gusmão, Kin Max Piamolini

Multiagent goal recognition as SAT / Kin Max Piamolini Gusmão. – 2023.

79 f.

Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Felipe Rech Meneguzzi.
Coorientador: Prof. Dr. Ramon Fraga Pereira.

1. Artificial intelligence. 2. Goal recognition. 3. Multiagent goal recognition. 4. Boolean satisfiability. 5. SAT. I. Meneguzzi, Felipe Rech. II. Pereira, Ramon Fraga. III. , . IV. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

KIN MAX PIAMOLINI GUSMÃO

MULTIAGENT GOAL RECOGNITION AS SAT

This Dissertation has been submitted in partial fulfillment of the requirements for the degree of Master in Computer Science of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on August 29th, 2023.

COMMITTEE MEMBERS:

Prof. Dr. André Grahl Pereira (PPGC/UFRGS)

Prof. Dr. Duncan Dubugras Alcoba Ruiz (PPGCC/PUCRS)

Prof. Dr. Ramon Fraga Pereira (The University of Manchester, Manchester, UK - Co-Advisor)

Prof. Dr. Felipe Rech Meneguzzi (PPGCC/PUCRS - Advisor)

I dedicate this dissertation to my mother, Jucymere Maria Piamolini Gusmão, and my father, João Carlos Gusmão Junior, who never ceased to believe in my potential, to support me on everything I ever did, and to motivate me towards my goals.

“If you put your mind to it, you can accomplish anything”
(“Doc” Emmett L. Brown from *Back to the Future*)

ACKNOWLEDGMENTS

First, I would like to thank the Pontifical Catholic University of Rio Grande do Sul for providing me with the knowledge foundation I needed for this Master's. I have been a student at PUCRS since I was a Computer Science B.Sc. student and I am very proud to call it my *alma mater*.

I would also like to thank CAPES/PROEX for providing me with the scholarship without which I would not have been able to attend to this Master's program.

I want to thank my parents, and my family in general, for always believing in me and giving me the support I needed during the difficult times of my Master's research. This project would not have been possible without their love and support.

I want to thank my co-advisor, Professor Ramon Fraga Pereira, from The University of Manchester, for all the learning and support over these past years. My first contact with planning and goal recognition was through one of Ramon's papers with Professors Felipe Meneguzzi and Nir Oren, and that paper put me in the research path I have followed up to this point.

Finally, I want to thank my advisor, Professor Felipe Meneguzzi, for all the support he has given me since I started working with him, in 2019, still as a Computer Science B.Sc. student. I could not thank him enough for everything I have learned from him on AI, planning, goal recognition, scientific writing, and how to be a researcher in general. I would not be who I am or where I am today if it were not for him. Additionally, I would like to thank him for allowing me to use figures and practical examples from his lecture slides in this dissertation.

RECONHECIMENTO DE OBJETIVOS EM AMBIENTES MULTIAGENTE COMO SAT

RESUMO

O reconhecimento de objetivos e planos tem sido foco de muitas pesquisas na área de inteligência artificial e planejamento automatizado, uma vez que as aplicações são diversas, como cuidado de idosos, detecção de intrusão, entre outras. Complementarmente, o campo de reconhecimento de objetivos em ambientes multiagente tem crescido muito nos últimos anos, pois em muitos casos não há um único agente atuando, mas um grupo de agentes no mesmo ambiente que podem ou não estar cooperando como um time em direção a um objetivo comum ou a vários objetivos diferentes. Existem muitas soluções para problemas de reconhecimento de objetivos, em cenários de agente único e multiagente, que se baseiam em técnicas de planejamento e na teoria de domínio de planejamento. No entanto, há poucos trabalhos existentes para reduzir um problema de reconhecimento de objetivo a um problema SAT e usar um solucionador SAT padrão para resolvê-lo. Esse tipo de solução é, entretanto, relativamente comum na área de planejamento automatizado. Nesta dissertação, desenvolvemos uma abordagem de reconhecimento de objetivos em ambientes multiagente baseada em testes de satisfatibilidade e avaliamos empiricamente o desempenho de tal abordagem com um conjunto de problemas de reconhecimento de objetivos em ambientes multiagente. Os resultados mostram que nossa abordagem atinge uma precisão de reconhecimento razoável, embora com desempenho limitado em tempo de execução.

Palavras-Chave: inteligência artificial, reconhecimento de objetivos, reconhecimento de objetivos multiagente, satisfatibilidade booleana, SAT.

MULTIAGENT GOAL RECOGNITION AS SAT

ABSTRACT

Goal and plan recognition have been the focus of much research within the field of artificial intelligence and automated planning, since the applications are manyfold, such as elder care, intrusion detection, among others. Complementary, the field of multiagent goal recognition has grown a lot in the past years, as in many cases, there is not a single agent acting but a group of agents in the same environment who may or may not be co-operating as a team towards a common goal or multiple different goals. There are many solutions for goal recognition problems, both in single-agent and multiagent scenarios, that leverage on planning techniques and planning domain theory. However, there is little work on the field on reducing a goal recognition problem to a SAT problem and using a standard SAT solver to solve it. Meanwhile, this kind of solution is fairly common in the automated planning field. In this dissertation, we develop a multiagent goal recognition approach based on satisfiability testing, and empirically evaluate our approach's performance with a set of multiagent goal recognition problems. Results show that our technique achieves reasonable recognition precision, albeit at limited runtime performance.

Keywords: artificial intelligence, goal recognition, multiagent goal recognition, boolean satisfiability, SAT.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ACRONYMS

CDCL – Conflict-Driven Clause Learning

CMAP – Centralized Multi-Agent Planning

FD – Fast-Downward Planning System

FF – Fast-Forward Planning System

HSP – Heuristic Search Planner

MAGR – Multiagent Goal Recognition

MAP – Multiagent Planning

MAPR – Multi-Agent Planning by Plan Reuse

MAPRAP – Multiagent Plan Recognition as Planning

MA-PDDL – Multiagent PDDL

MA-STRIPS – Multiagent STRIPS

OMT – Optimization Modulo Theory

PDDL – Planning Domain Definition Language

P-MAPRAP – Probabilistic Multiagent Plan Recognition as Planning

PMR – Plan Merge by Reuse

RPG – Relaxed Planning Graph

SAT – Boolean Satisfiability Problem

SMT – Satisfiability Modulo Theory

STRIPS – Stanford Research Institute Problem Solver

VSIDS – Variable State Independent Decaying Sum

CONTENTS

1. INTRODUCTION

Correctly inferring an agent's goal based on observations of their actions has multiple real-world applications [45]. A few examples are elder-care applications [20], intrusion detection systems [21], exploratory domain models [43, 46], among many others. Additionally, there are many cases where we might not want to identify the intent of a single agent but of a group of agents who might or might not be cooperating toward a common goal.

Recent approaches to both single-agent and multiagent goal recognition have managed to obtain good accuracy in the recognition process employing different techniques [51, 48, 2, 3, 63, 64]. The use of SAT solvers for goal recognition tasks has been, however, scarce over the years, with the most prominent example being the use of Weighted MAX-SAT by Zhuo *et al.* for multiagent scenarios [62, 63, 64]. This lack of SAT-based approaches differs from planning, the overarching area, which extensively uses satisfiability approaches [28, 31, 16, 54, 15].

Given the performance of SAT-based approaches while solving planning problems that allow multiple parallel actions [54], we aim to explore further the usage of SAT-based solutions for planning and goal recognition applications in both single-agent and multiagent scenarios. In this work, we develop a multiagent goal recognition algorithm that relies on satisfiability testing to perform the recognition.

We leverage SAT-based planning techniques and soft constraints enabled by the Z3 theorem prover [14] to develop a recognizer capable of analyzing all possible team-goal mappings given a set of agents and a set of goal hypotheses, scoring these mappings based on the observed team traces (the actions performed on the environment) and selecting the most likely correct mappings based on their scores. The resulting algorithm performs well regarding recognition precision for most problems in our dataset, but the recognition time is a significant drawback to our approach.

We organize the remainder of this manuscript as follows. First, we go over the theoretical background around planning, planning as satisfiablity, multiagent planning, and goal recognition, in Chapters 2, 3, 4, and 5, respectively. We present our definition of a multiagent goal recognition problem and our multiagent goal recognition approach, in Chapter 6. Then, in Chapter 7, we discuss our experimentation strategy and the results obtained from the experiments conducted with our approach's implementation. Chapter 8 discusses the related work in the multiagent goal recognition field. Finally, in Chapter 9, we conclude this dissertation and add our final remarks, with possible future work.

2. CLASSICAL PLANNING

Classical planning, or as we refer to it from now on, simply planning, is the task of obtaining a sequence of actions (a plan) that can achieve a goal state starting from some initial state [22]. Figure 2.1 depicts an example of a planning environment with multiple possible goals, where we might want to derive a plan to direct our robot agent to one of the goals. Classical planning is a subset of a wider research field called automated planning, which includes other planning approaches such as *hierarchical task network* planning [17]. Automated planning has applications in many fields, such as video-game AI [32], autonomous systems [26], and many others. In this chapter, we outline the field of classical planning. We start by going over the formalism and notation in Section 2.1. We discuss the main problem description languages in Section 2.2. Finally, in Section 2.3, we discuss the main frameworks and algorithms.

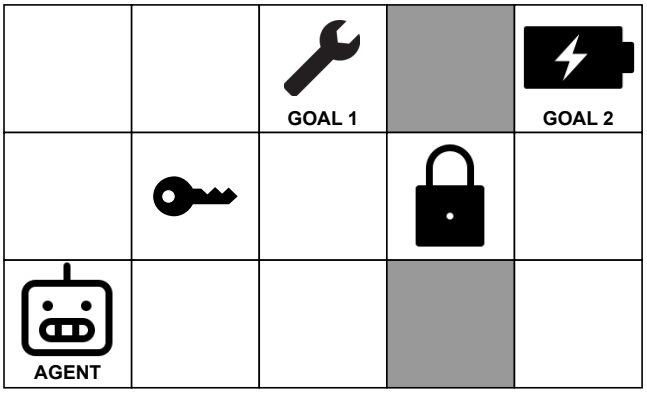


Figure 2.1: Planning environment example

2.1 Formalism and Notation

To explain the classical planning formalism, we provide examples from the classical planning domain *Blocks World* [37]. In this planning domain, the agent sees itself in an environment where uniquely identified blocks are displayed above an immovable table. These blocks may be clear or stacked on each other, in any order, at the beginning of the process. The agent can pick up clear blocks, *i.e.* blocks that have no other blocks on top of them, and put them down on the table or on top of another block, stacking them up. The agent must move the blocks to achieve its goal, a specific positioning of some or all of the blocks. Figure 2.2 depicts an example of an initial state and a goal state in the *Blocks World* domain. In a planning task, we must obtain a sequence of actions (a plan) for the agent to execute and transform the initial state into the goal state.

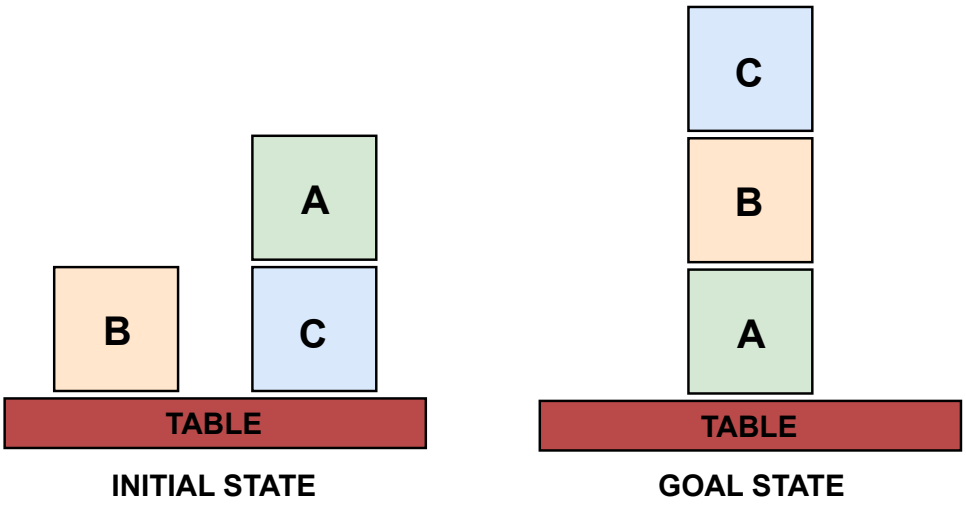


Figure 2.2: *Blocks World* initial and goal state examples.

There are two basic structures in planning: facts and actions. Facts are ground predicates associated to zero or more terms $(\tau_1, \tau_2, ..., \tau_n)$. If we take the *Blocks-World* planning domain as an example, we could have a fact `(on a b)`, where the predicate `on` is associated to the terms `a` and `b`, representing the fact that block `a` is on top of block `b`, or `not (on a b)`, representing the fact that that block `a` is not on top of block `b`. A *state* comprises a conjunction of facts and describes the environment state at a given time step.

Actions are operators with instantiated free variables, where each variable represents an object of the environment that action modifies and/or needs to modify the state. An operator a is a tuple $a = \langle name(a), pre(a), eff(a) \rangle$, where $name(a)$ is the name or description of the operator, $pre(a)$ are the preconditions of the operator, *i.e.* facts that must be true before the agent executes the operator and $eff(a) = eff(a)^+ \cup eff(a)^-$ are the effects of the operator, where $eff(a)^+$ is an add-list of facts, *i.e.* facts that become true once the operator executes, and $eff(a)^-$ is a delete-list of facts, *i.e.* facts that become false once the operator executes.

In *Blocks-World*, we obtain the action `(unstack a b)` by instantiating operator `unstack`'s free variables to blocks `a` and `b`, representing the action to remove block a from the top of block b. In this example, the action's precondition is that block `a` must be currently on top of block `b`, while the action's effect is that block `a` is no longer on top of block `b` once the agent performs the action. We say that an action is applicable at a given state S if its preconditions are satisfied in that state, *i.e.* $\forall p \in pre(a), p \in S$. When an agent performs an action at a given state S, it generates a new state S', where $S' = S \backslash eff(a)^- \cup eff(a)^+$.

A classical planning domain definition is a tuple $\Xi = \langle \Sigma, \mathcal{A} \rangle$, where Σ is a finite set of ground facts and $\mathcal{A}$ is a finite set of instantiated operators, *i.e.* all facts and actions possible to occur in the planning domain. A planning instance is a triple $T_P = \langle \Xi, \mathcal{I}, G \rangle$, where Ξ is the planning domain definition, $\mathcal{I}$ is the initial state, and G is the goal state.

Finally, a plan is a sequence of actions $\pi = \langle a_1, a_2, ..., a_n \rangle$ that modify the initial state $\mathcal{I}$ into the goal state G.

2.2 Problem Description Languages

The first standardized language for describing planning problems is STRIPS, named after the planner that first used it, the Stanford Research Institute Problem Solver [19]. Using STRIPS, one can define the initial and goal states and a set of operators. Each operator has a name, a set of preconditions, a set of effects and, optionally, a set of variable constraints.

Later, the first and still to this day most widely used language for defining planning domains and problems, the Planning Domain Definition Language (PDDL), would be introduced in 1998 [41]. Currently, PDDL is the standard language for defining planning domains and problems. It is mostly derived from STRIPS, maintaining all of its features, while having greater description capabilities that allows one to define such things as object types, negated preconditions, conditional effects, among other features, being considered an evolution to STRIPS.

The definition of a full planning instance in PDDL comprises two files: a domain file and a problem file. The domain file contains the information needed to define a planning domain: a list of fact predicates and a list of action operators. Listing 2.1 depicts an example of a simple definition of the *Blocks World* planning domain.

```
1  (define (domain blocks)
2
3   (:requirements :strips)
4   (:constants-def table)
5
6   (:predicates (on ?a ?b)
7                (block ?b)
8                (clear ?b) )
9
10  (:action move
11   :parameters (?b ?x ?y)
12   :precondition (and (on ?b ?x)
13                      (clear ?y)
14                      (clear ?b)
15                      (block ?b)
16                      (block ?y)
17                 )
18
19          :effect (and (on ?b ?y)
20                       (clear ?x)
21                       (not (on ?b ?x))
```

```
                    (not (clear ?y))
                )
 )

 (:action moveToTable
  :parameters (?b ?x)
  :precondition (and (on ?b ?x)
                     (clear ?b)
                     (block ?b)
                     (clear ?x)
                )

  :effect (and (on ?b table)
               (clear ?x)
               (not (on ?b ?x))
          )
 )
)
```

Listing 2.1: Simple *Blocks World* domain definition

The file defines a list of predicates applied to free variables, preceded by the "?" symbol. Once those variables get instantiated to actual objects, we obtain a ground predicate, or a fact. We see predicates to verify that a given block is on top of another given block, to check whether a given object is a block, and to check whether a block is clear, *i.e.* if there are no blocks on top of it.

The definition also contains a list of operators, each denoted by the reserved word "action". Each action definition must include an action name, the list of variables that action takes, a list of preconditions regarding the variables in the form of a conjunction of facts, and a list of effects, also as a conjunction of facts. The positive facts represent the add-list of effects, while the negated ones represent the delete-list. Once we instantiate the operator's variables to actual objects, we obtain a ground operator, or an action.

Having the domain defined, we can then define the planning instance, or the problem. The problem must be defined in a separate problem file, containing information such as the objects over which the facts and actions from the domain will be applied, the initial state and the goal state. Listing 2.2 depicts a problem definition for the previously defined domain.

```
(define (problem pb1)
 (:domain blocks)
 (:objects a b c table)
 (:init (on a table)
        (on b table)
        (on c a)
        (block a)
```

```
8           (block b)
9           (block c)
10          (clear b)
11          (clear c))
12   (:goal (and (on a b)
13                (on b c)
14          )
15   )
16  )
```

Listing 2.2: *Blocks World* problem definition example

The first thing in the problem definition must be the problem name, defined in line 1. The next definition is the planning domain of the problem, as defined in line 2. It must also define the list of objects in the instance, as defined in line 3, the initial state, and the goal state. The initial state (line 4) is defined by a list of facts over the defined objects, while the goal state (line 12) is a conjunction of facts over the objects. This problem Listing describes the initial and goal states from Figure 2.2 in PDDL.

2.3 Frameworks and Algorithms

Most of the older algorithms used to solve planning problems use forward or backward search strategies. STRIPS is one of the first and better known algorithms at the beginning of planning research [19]. Like its predecessors, STRIPS uses backward search, but reduces the search space by applying some strategies. One of these strategies is only considering sub-goals that are preconditions to the last operator added to the plan and committing to execute a given operator, not backtracking over this commitment if the current state satisfies all the operator's preconditions. Apart from being one of the first algorithms to apply planning-specific strategies to search algorithms, STRIPS also defined the first standard language for defining planning problems.

Years later, another planner would become a landmark in the planning research field. The Graphplan planner [7] uses a graph to annotate problem information, which helps prune sub-goals in the backward search done in the solution phase of the algorithm. In this graph, every even level is a fact level, where each node represents a ground fact, composing the state at that time step. Complementary, every odd level is an action level, where each node represents a ground action that is applicable at that time step, *i.e.* its preconditions are satisfied at that time step.

The first level contains every fact in the initial state, while the last level contains every fact in the goal state. Every fact node connects to the actions in the next level to which it is a precondition, while every action node connects to the facts in the next

level that are an effect for that action. We call this structure a *planning graph*. Figure 2.3 illustrates the general structure of a planning graph.

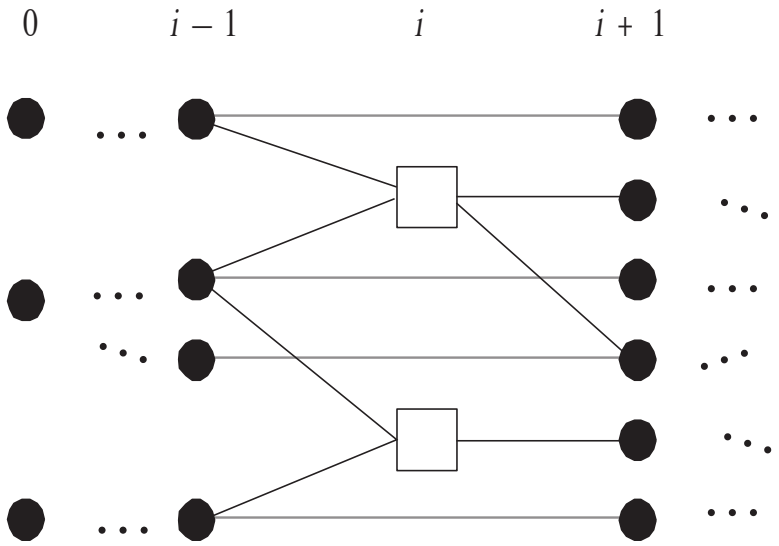


Figure 2.3: Planning graph general structure

With the planning properly graph set up, Graphplan then starts the search process. The planner performs a backward-chaining level-by-level strategy. First, it starts by analyzing the last level of the graph (for convenience, we call the current level t), searching for the goal state predicates. It then goes to level $t-1$, searching for the actions that have those predicates as add-effects. Once it finds those actions, it takes the preconditions for these actions as sub-goals, and the recursive search process goes to the level prior to that, treating it as level t now. The planner repeats this process until it reaches the first graph level, then it derives a plan from the actions selected to fulfill the sub-goals at each recursion. If an action fails to fulfill a sub-goal, Graphplan will try to select another one until that sub-goal is fulfilled. If it comes to a point that some sub-goal from level t could not be fulfilled by any actions in level $t-1$, the planner returns a failure, as the goal is not achievable.

There is also a relaxation of the planning graph structure where the delete-list of effects is ignored in the relation between the facts and actions. We call this relaxed structured a *relaxed planning graph* (RPG). Graphplan brought major improvements to the planning process that are still referenced in recent works, such as the ability to represent mutual exclusion between actions, the possibility of having parallel actions when they are not mutually exclusive, memoization capabilities obtained by having the actions fixed at their respective time steps, and not reducing instantiations at search time, as the planner generates the entire graph before the search process begins.

Some planning approaches use satisfiability testing to solve planning problems. These approaches convert a planning problem into a boolean formula, use SAT solvers to find a valid model for that formula, and then extract the plan from the found model. This kind of approach first appeared in Kautz and Selman's work [28, 31, 27, 29], with the SATPLAN and BLACKBOX planners. We discuss these approaches in details in the next chapter.

Other planning approaches leverage on heuristic search, such as the Heuristic Search Planner (HSP) [8] and the Fast-Forward planning system (FF) [25]. Both use heuristic search based on a delete-free relaxation of planning problems, where one ignores the delete-list of effects.

HSP was one of the planners in the AIPS 98 planning competition [37], and it showed that heuristic-based planners can be competitive against the existing Graphplan [7] and SAT planners such as BLACKBOX [29]. The work that introduces HSP [8] presents three variations of HSP. The first one is a hill-climbing planner with additive heuristic that, even though showed to be competitive against the other planners in the competition, was not optimal nor complete. To solve the completeness issue, they present a variation called HSP2, that uses best-first search instead of hill-climbing, and it not only showed to be better at solving the planning problems, but also obtained a better runtime performance than the hill-climbing version. Finally, they present a third version, called HSPr, which does a backward search instead of the forward one performed by the first two variants, performing a regression from the goal state. They discuss that this prevents the full heuristic recomputation at every single state, resulting in a much better runtime performance.

The influence of HSP in the field derived the Fast-Forward planning system (FF) [25]. FF is widely based on HSP, but has a major increase in performance, being the best planner in the AIPS 2000 planning competition [4]. The main differences from HSP are the heuristic, that takes into account positive interactions between facts, using enforced hill-climbing instead of hill-climbing for a search method, and pruning action nodes based on identifying the ones that are more helpful on reaching the goal.

These planners kept deriving more high-achieving planners, such as the Fast-Downward planning system (FD) [24] and the LAMA planner [52]. FD is also a forward search heuristic planner, but it translates the planning task into what the call a multivalued planning task, and solve it by hierarchically decomposing the task to compute the heuristic function. The Fast-Downward planning system was the best performing planner in the classical track of the 4th International Planning Competition at ICAPS 2004. A few years later, the LAMA planner was introduced, based on the FD planning system, but using the FF heuristic combined with a heuristic based on planning landmarks, *i.e.* necessary facts or actions that must be achieved or executed to achieve a given goal. LAMA was the best performing planner in IPC 2008's sequential satisfying track.

3. PLANNING AS SATISFIABILITY

One of the approaches to solving a planning problem is converting it into a SAT instance. For this, one must properly encode the planning problem into a SAT problem, which allows solving the planning problem with a regular SAT solver. In this Chapter, we go over the theoretical background involving the Boolean Satisfiability Problem (SAT) and some solving techniques, in Section 3.1, different encodings for planning as satisfiability, in Section 3.2, different approaches in SAT solving for planning, in Section 3.3, and some planners based on SAT, in Section 3.4.

3.1 Boolean Satisfiability Problem (SAT)

The Boolean Satisfiability Problem (SAT) is the problem of checking whether a given boolean formula is satisfiable or not. In other words, the solver needs to check if there is a model, *i.e.* a truth-value assignment for each variable in the formula, that makes that formula true. An example is the formula below:

$$(p \vee \neg q \vee r) \wedge (\neg p \vee \neg q \vee \neg r) \wedge (\neg p \vee q \vee \neg r)$$

The formula above is satisfiable, and a valid model for it is $\{p = \mathit{False}, q = \mathit{False}, r = \mathit{True}\}$. By contrast, the formula below is an example of an unsatisfiable formula, as no model exists that makes the formula true:

$$(p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee q) \wedge (\neg p \vee \neg q)$$

SAT is probably the most widely studied problem in computational logic, as it was the first problem to be proven NP-complete, through the Cook-Levin theorem [11, 36]. This not only means that SAT is in NP time complexity class, *i.e.* it can be solved in polynomial time by a non-deterministic Turing Machine, but also means that any problem in NP can be reduced in polynomial time to a SAT problem. This is extremely useful, as we don't need to know how to solve every problem in NP. We can just convert it to a SAT problem and solve it with a standard SAT solver.

One of the first procedures to be used in SAT solving is the Davis-Putnam procedure [13, 12], or DPLL, named after the initials of the authors for the latter paper, Davis, Putnam, Logemann, and Loveland. It takes a boolean formula as input and outputs a boolean value that represents whether the formula is satisfiable or not, and the found model that satisfies the formula, in case it is satisfiable. Some implementations only re-

turn the boolean value, but we assume implementations that also return the model, as the algorithm is capable of doing so.

The algorithm works as a recursion over the formula. It starts by propagating all unit clauses, *i.e.* it selects the variables that appear alone in clauses and assigns truth-values to them to satisfy those clauses. This is called the *unit propagation* step. It also removes from the formula all the clauses that were satisfied by the current truth-value assignment. Finally, it selects a variable to assign a truth-value to, and applies that value to the formula. It recursively tries to assign the value “true”, and then “false”. The variable selection depends on the implementation, and can be as simple as a random selection, or have a more clever procedure such as selecting the variable with most occurrences among the clauses. On the next recursion, the unit propagation step will remove the unit clauses created by the truth-value assignment of the chosen variable, and every clause satisfied by it. If all variables in a clause are assigned, but the clause is not yet satisfied, the clause is called an *empty clause*, still remaining in the formula. Conversely, when a clause is satisfied, it is removed from the formula. Each recursion, the algorithm checks if the formula is empty, which makes it trivially satisfied, and also checks for the presence of any empty, *i.e.* unsatisfiable, clauses. If there is an empty clause, the formula is not satisfied with that model, and it backtracks to the previous recursion step to choose a different variable assignment.

Although the DPLL procedure is still widely used, the current state-of-the-art procedure for SAT solving is an algorithm commonly called Conflict-Driven Clause Learning (CDCL), originally developed in [39, 40], under the name GRASP. The CDCL algorithm uses DPLL as a base, but adds major improvements to it. For example, it adds clause conflict learning, as it can learn exactly which truth-value assignment caused the conflict that led to the formula being unsatisfiable with that model. Furthermore, it adds non-chronological backtracking. The DPLL procedure can only backtrack to the previous recursion level each time, which may lead to the algorithm still exploring paths that will lead to unsatisfiability. The clause learning performed by CDCL allows it to backtrack directly to the assignment that led to the conflict. Other improvements may vary depending on the implementation, such as analyzing isolated clauses separately, in parallel, and using heuristics to make a better choice on the variable to assign next.

As a heuristic example, we take the Variable State Independent Decaying Sum (VSIDS) heuristic, first implemented as a part of the Chaff SAT solver [44]. The heuristic works by computing a score for each variable, then CDCL chooses the one with the highest heuristic score. The initial score is the number of literal occurrences of the variable. Then, for each new conflict clause found, the solver increments the score for all the variables that belong to the clause, and periodically divides it by a constant.

There are also variants of the SAT problem. For instance, there is the MAX-SAT problem, an optimization variation of SAT, where we want to find a model to maximize

the number of satisfied clauses. We can even weight the clauses differently, creating a Weighted MAX-SAT problem. Moreover, we can generalize the SAT problem even further as a Satisfiability Modulo Theory (SMT) problem. An SMT problem is a satisfiability problem that allows descriptions using logical quantifiers, arithmetic operations, data structures such as arrays, lists, among others. Both DPLL and CDCL work as SMT solvers besides their original SAT origin. MAX-SAT also has its own generalization, called Optimization Modulo Theory (OMT).

3.2 Problem Encodings

In [28], Kautz and Selman introduce the concept of planning as SAT, and develop a way to convert a planning problem into a SAT problem, by representing it as a set of axioms. The axioms determine the properties of the planning domain and instance, such as the property that if the preconditions of an action hold, then the action achieves its effects, describe which propositions a given action does not affect when executed, and rule out anomalous models, such as the possibility of an action executing despite its preconditions being false, as well as guaranteeing that one, and only one action occurs at a time. This initial problem encoding is referenced in future work as a *linear encoding*.

A later work by Kautz and Selman [31], published in 1996, discusses over three possible encodings for planning as satisfiability. Besides revisiting their linear encoding [28], they describe two additional problem encodings: one based upon GraphPlan [7] and a third state-based problem encoding. They describe the GraphPlan encoding as a direct conversion of a planning graph as defined in [7] into a boolean formula, as they can fully represent the graph through axioms that guarantee that:

- The initial state holds at layer 1, and the goals hold at the highest level;
- Each fact at level i implies the disjunction of all the operators at level $i - 1$ that have it as an add-effect;
- Operators imply their preconditions;
- Conflicting actions are mutually exclusive.

The advantage of this encoding is that it allows partially ordered plans, as the subgraph that serves as a solution to a planning problem is only partially-ordered, where the actions at a given level can be executed at any order given they are non-conflicting. This encoding is, however, less expressive than the linear encoding.

Finally, they [31] describe a third encoding called a state-based encoding. This encoding takes advantages from both previous encodings, being able to represent partially-ordered plans, as well as having a greater expressiveness if compared to the GraphPlan

encoding. The core of this encoding is defining axioms to ensure that each state is valid, by expressing the actions that could account for every possible state change *i.e.* the effects imply the action. Basically, the axioms define that, if a fluent changed its value between two time steps, then it means that an action that has this fluent in its effects (add or delete) must have been executed at that time step.

In this state-based encoding the axioms that represent the planning instance can be classified into groups of formulas, each representing a group of definitions and constraints. We consider the plan steps for this encoding, where, for an $n-1$-step plan, step 0 is the step before executing the first action, while step n is the step after executing the final action. Equation 3.1 encodes the initial state in a way that guarantees that every fact that belongs to that state holds at step 0 and every other fact does not.

$$\bigwedge_{f \in \mathcal{I}} f_0 \wedge \bigwedge_{f \notin \mathcal{I}} \neg f_0 \tag{3.1}$$

Equation 3.2 encodes the goal state by the conjunction of all fluents (facts) that must hold at time step n. This ensures that all facts from the goal state must hold in the state generated by executing the final action in the plan.

$$\bigwedge_{f \in G} f_n \tag{3.2}$$

Equation 3.3 ensures that an action must be applicable in the current state if we want to execute it, *i.e.* its preconditions must hold at the current time step. This equation also ensures that the action's effects hold at the next time step, meaning that we modified the state by executing the action. In other words, any given action implies its preconditions and effects.

$$\bigwedge_{a \in \mathcal{A}} \left(a_i \implies \left(\bigwedge_{p \in pre(a)} p_i \wedge \bigwedge_{e \in eff^+(a)} e_{i+1} \wedge \bigwedge_{e \in eff^-(a)} \neg e_{i+1} \right) \right) \tag{3.3}$$

The axioms also guarantee that an action *only* changes the facts that are in its effects, by ensuring that any state change must be related to a suitable action being executed. This means that an action can only modify what its definition describes. These are called *explanatory frame axioms*, and are encoded through Equation 3.4.

$$\bigwedge_{f \in \Sigma} \left(\left(\neg f_i \wedge f_{i+1} \implies \left(\bigvee_{a \in \mathcal{A} \mid f_i \in eff^+(a)} a_i \right) \right) \wedge \left(f_i \wedge \neg f_{i+1} \implies \left(\bigvee_{a \in \mathcal{A} \mid f_i \in eff^-(a)} a_i \right) \right) \right) \tag{3.4}$$

Depending on the application, we might need an axiom to ensure that only one action occurs at each time step, for each and every $a, b \in \mathcal{A}$. This is called a *complete exclusion axiom*, and is defined by Equation 3.5.

$$\bigwedge_{a,b \in \mathcal{A},\, a \neq b} \neg a_i \vee \neg b_i \tag{3.5}$$

In other applications, we might want to allow actions that are not mutually exclusive to be executed in parallel, at the same time step. There are multiple definitions for mutual exclusion in actions across different work, such as considering mutually exclusive actions that cannot be executed in every possible ordering [31], or considering actions that can be executed in at least one order to be non-mutually exclusive [16]. If we consider the case where a group of actions is not mutually esclusive if they are not pairwide mutually exclusive, *i.e.* they can be executed in any given order, we need to extend the mutual exclusion axiom to account for that, and we do so through Equation 3.6. It is the same equation as Equation 3.5, but it only adds axioms for actions that are pairwise mutually exclusive, which is represented by the symbol $\bowtie$.

$$\bigwedge_{a,b \in \mathcal{A},\, a \neq b,\, a \bowtie b} \neg a_i \vee \neg b_i \tag{3.6}$$

Finally, depending on the application as well, we might define yet another axiom to ensure that at least one action occurs at each time step. This is useful when we do not want to account for a no-op action, and do not want any idle time steps. We call this axiom a *full frame axiom*, and it is defined by Equation 3.7.

$$\bigvee_{a \in \mathcal{A}} a_i \tag{3.7}$$

This kind of formulation assumes a planning problem with a bounded horizon. This means that each set of formulas is meant to find a valid plan with a length (or horizon) n that reaches the goal state. The horizon is iteratively incremented until a valid model is found for the formula.

To illustrate the way the formulas above work, we take an example of a planning instance where a robot must move between different locations. Listing 3.1 depicts the domain we work on. In this domain, we have robots and locations, and the only predicate we have, defined in line 4, states that a certain robot is at a certain location. Similarly, we have only one action, defined in lines 6 through 17, which represents a robot r moving from an origin location o to a destination location d. The precondition for this method, defined in line 12, states that robot r must be at location o for the action to be executed, while the effects for this method, defined in lines 13 through 16, define that, once the action is executed, robot r is no longer at location o, and is now at location d.

```
(define (domain robots)
  (:requirements :strips :typing)
  (:types robot location)
  (:predicates (at ?r - robot ?l - location))

  (:action move
    :parameters (
      ?r - robot
      ?o - location
      ?d - location
    )
    :precondition (and (at ?r ?o))
    :effect (and
      (not (at ?r ?o))
      (at ?r ?d)
    )
  )
)
```

Listing 3.1: Robots domain definition

Having the domain defined, we then define the planning instance itself through Listing 3.2. In this problem definition, we define one robot object, $r1$, and two location objects, $l1$ and $l2$, in lines 3 through 7. Then, in line 8, we state that, in the initial state, robot $r1$ is at location $l1$, and in line 9 we state that our goal is for robot $r1$ to be at location $l2$.

```
(define (problem pb1)
  (:domain robots)
  (:objects
    r1 - robot
    l1 - location
    l2 - location
  )
  (:init (at r1 l1))
  (:goal (and (at r1 l2))
  )
)
```

Listing 3.2: Robots problem definition

If we encode this example with the axioms presented above, with a planning horizon of 1, we get the set of axioms representing that planning instance. Equation 3.8 encodes the initial state IS in step 0, while Equation 3.9 encodes the goal state in step 1, the final step of our horizon, as per Equations 3.1 and 3.2, respectively. Equation 3.10 encodes the action formula A for our horizon of 1 time step, as per Equation 3.3, while Equation 3.11 encodes the explanatory frame axioms EFA for our horizon, as per Equa-

tion 3.4. Then, Equation 3.12 encodes the exclusion axiom *EA*, and Equation 3.13 encodes the full frame axiom *FFA*, as per Equations 3.6 and 3.7, respectively. Finally, Equation 3.14 encodes the full formula Ω, as the conjunction of Equations 3.8 through 3.13, which is the formula we need to find a model to.

$$IS = at(r1, l1, 0) \wedge \neg at(r1, l2, 0) \tag{3.8}$$

$$GS = at(r1, l2, 1) \tag{3.9}$$

$$\begin{aligned} A = (move(r1, l1, l2, 0) \Rightarrow at(r1, l1, 0) \wedge at(r1, l2, 1) \wedge \neg at(r1, l1, 1)) \wedge \\ (move(r1, l2, l1, 0) \Rightarrow at(r1, l2, 0) \wedge at(r1, l1, 1) \wedge \neg at(r1, l2, 1)) \end{aligned} \tag{3.10}$$

$$\begin{aligned} EFA = (\neg at(r1, l1, 0) \wedge at(r1, l1, 1) \Rightarrow move(r1, l2, l1, 0)) \wedge \\ (\neg at(r1, l2, 0) \wedge at(r1, l2, 1) \Rightarrow move(r1, l1, l2, 0)) \wedge \\ (at(r1, l1, 0) \wedge \neg at(r1, l1, 1) \Rightarrow move(r1, l1, l2, 0)) \wedge \\ (at(r1, l2, 0) \wedge \neg at(r1, l2, 1) \Rightarrow move(r1, l2, l1, 0)) \end{aligned} \tag{3.11}$$

$$EA = \neg move(r1, l1, l2, 0) \vee \neg move(r1, l2, l1, 0) \tag{3.12}$$

$$FFA = move(r1, l1, l2, 0) \vee move(r1, l2, l1, 0) \tag{3.13}$$

$$\Omega = IS \wedge GS \wedge A \wedge EFA \wedge EA \wedge FFA \tag{3.14}$$

Once the SAT solver finds a model for formula Ω with the given horizon, we then need to extract the plan from the model. As per to Equations 3.6 and 3.7, we know that for each time step i from 0 through $n - 1$, there will be at least one ground action fluent with a truth-value of *true*. Those fluents represent the actions executed at the $i - th$ time step of the plan. The time steps define the order of the plan, while actions in the same time step are partially order, as they can be executed at any given order. We could say that a valid model for the example above would be setting the following literals to “true”, and all other literals to “false”:

- $at(r1, l1, 0)$
- $move(r1, l1, l2, 0)$

- $at(r1, l2, 1)$

The model above satisfies all the formulas we have defined, and sets to "true" the literals representing each state, and the actions performed. In this case, with a horizon of 1, we have only two states and one action performed. Given that model we know that a valid solution for the planning problem we defined would be a one-step plan containing only the action $move(r1, l1, l2)$.

To address the issue of the resulting large problem encodings, which tends to increase the solving time, the authors [31] discuss that a way to mitigate that issue is to reduce the predicate arity. In that kind of strategy, if there is, for instance, a quaternary predicate `move(x,y,z,i)` that represents that an object `x` moves from `y` to `z` at time `i`, we can instead represent it as 3 binary predicates `object(x,i)`, `source(y,i)`, and `dest(z,i)`, which reduces the size of the encoding. Having the encoding defined, they generate SAT instances for bounded increasing planning horizons until the SAT solver finds a solution.

In later work [16], the authors develop a new encoding that relaxes the representation even more. In this encoding, they take plan parallelism even further, allowing parallel actions given that the actions can be laid in some order where they do not conflict with each order, *i.e.* there is, at least, one valid ordering for the actions. This differs from GraphPlan parallelism in [31], where every possible ordering of the actions needs to be valid. There is a further relaxation of this technique, described in [61], where a group of actions does not need to have all of its preconditions satisfied to be applied in parallel, given that the actions can satisfy each other's preconditions using their effects in some ordering.

3.3 SAT Solving for Planning as SAT

The work by Kautz and Selman [28, 31] uses a SAT solving approach that solves SAT-encoded planning problems sequentially, one at a time, for bounded increasing horizon lengths (1, 2, 3, 4, 5, ...). Work by Rintanen *et al.* [55] introduces two parallel algorithms, which we call algorithms A and B. Algorithm A simultaneously solves SAT instances for bounded horizons of size 1 through n. If it finds that the formula is satisfiable, it returns the plan, and the algorithm terminates. Otherwise, it starts a solver for the shortest plan length not searched yet, always solving n SAT instances in parallel. Algorithm B also runs a predefined number of SAT solvers in parallel, but the CPU time it assigns to solver with horizon t is the result of the multiplication between the time assigned to the instance with horizon $t - 1$ and a constant $g < 1$, decreasing the CPU time as the horizon grows. It also restricts the number of parallel SAT solvers depending on memory availability.

Since some approaches use the CDCL algorithm for SAT solving [61, 54], there is research on planning-specific heuristics to replace standard heuristics, such as VSIDS.

A recent one is the Variable Selection to Satisfy Goals and Subgoals heuristic [53]. This heuristic is based on the principle that each goal literal has to be made true by an action, and that for that action to be executed, its preconditions must be made true by some previous action, or be true at the initial state. It then performs a backwards search to find the earliest time point where a goal literal becomes true and remains true to the end of the plan. Finally, it chooses the action that makes that literal true.

3.4 SAT-Based Planners

Kautz and Selman [28] developed the SATPLAN [28, 31, 27] system, updating it through different papers as they found better problem encodings. The initial version implemented the linear encoding from their first work [28]. They further refined it, implementing the state-based encoding [31]. Kautz and Selman also developed the BLACKBOX planner [29], which works by creating a GraphPlan-style [7] planning graph with a bounded horizon, translating the constraints into a set of clauses, solving it with a regular SAT solver and, if a solution is found, translating the solution into a plan and pruning unnecessary actions. If a solution is not found, it increases the horizon and tries again. This planner obtained great results in the 1998 AIPS planning competition [37], obtaining the best results in plan length across the problems, while being one of the fastest planners.

The current state-of-the-art SAT planner is Jussi Rintanen's Madagascar planner [54]. This planner leverages the parallel actions encoding described in [61] and GraphPlan's binary mutexes. The authors describe more than one version for the planner, with different configurations, which allows the user to experiment with different execution settings and use the most efficient one for their application. The versions vary the horizon length the planner considers when searching for plans with bounded horizons, as well as the heuristic to use with CDCL algorithm. They also describe two heuristics currently implemented in the planner: the VSIDS heuristic [44] and the Variable Selection to Satisfy Goals and Subgoals heuristic [53].

4. MULTIAGENT PLANNING

The formalism we discussed so far assumes there is either a single agent acting in the environment, or at least a centralized decision-making agent generating plans. However, in many cases, there is not only one agent, but a group of agents that must cooperate to complete a given task. In that case, the planner must take that into account and find the most efficient set of actions for a group of agents to execute and reach the common goal. This is called a multiagent planning (MAP) task. Figure 2.1 depicts an example of a multiagent planning environment with two robot agents and two possible goals, where we might want to derive a plan to direct both robots to one of the goals, where they shall cooperate to achieve it, or direct each robot to a different goal. In this chapter, we summarize background for multiagent planning, defining the formalism and notation in Section 4.1, the language used to describe a multiagent planning problem, in Section 4.2, and relevant frameworks and algorithms, in Section 4.3.

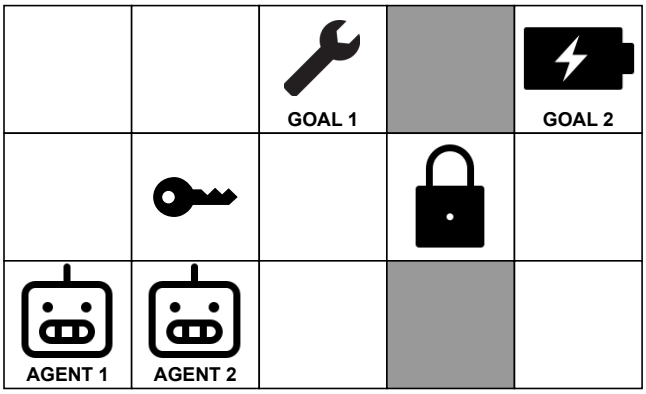


Figure 4.1: Multiagent planning environment example

4.1 Formalism and Notation

For convenience, we use the definitions from [59]. To define a MAP task, we use the MA-STRIPS definition [10], which extends the previously discussed STRIPS definition to multiagent tasks, as this is the main formalism adopted by the work in this field. MA-STRIPS shares most definitions with the standard STRIPS definition, such as the definition of *states*, *facts*, and *actions*, as well as the definition of preconditions, effects, and state applicability for actions.

We define a MAP task as a tuple $T_{MAP} = \langle \Phi, \Sigma, \{\mathcal{A}^i\}_{i=1}^{k}, \mathcal{I}, G \rangle$, where:

- Φ is a finite set of k agents;
- Σ is a finite set of facts;

- $\mathcal{A}^i$ is the finite set of actions for agent i. We define the full set of actions for T_{MAP} as $\mathcal{A}$, where $\mathcal{A} = \cup_{\forall i \in \Phi} \mathcal{A}^i$;
- $\mathcal{I}$ is the initial state;
- G is the common goal state.

The solution for a MAP task is an ordered sequence of actions, *i.e.*a plan, that, when applied to initial state $\mathcal{I}$, generates a state that contains the goal state G. In the standard MA-STRIPS definition, a solution plan must be a totally-ordered set of actions [10], while other works describe a plan as a set of action sequences (one sequence per agent) [35] or as a partially-ordered sequences of actions [60].

We can divide the facts Σ in a MAP task into *public* and *private* facts. Private facts are internal to a given agent $i \in \Phi$, and can only be used and affected by actions in $\mathcal{A}^i$. Public facts are accessible to all agents. We denote agent's i private (internal) facts as Σ^i_{int} and the public facts as Σ_{pub}. We distribute the task to the agents in Φ as a set of *local views*, where each agent has its own local view of the task. The local view for agent i of the MAP task T is denoted as a 4-tuple $T_i = \langle \Sigma^i, \mathcal{A}^i, \mathcal{I}^i, G \rangle$, where:

- $\Sigma^i = \Sigma^i_{int} \cup \Sigma_{pub}$ is the set of facts accessible to agent i;
- $\mathcal{A}^i$ are the available actions for i;
- $\mathcal{I}^i \in \Sigma^i$ is the set of facts belonging to the initial state $\mathcal{I}$ that are accessible to i;
- G is the common goal state. All facts in the common goal state must be accessible to all agents in Φ.

4.2 Problem Description Language

Just as single-agent planning has its standard language for defining domains and tasks (PDDL), so does MAP. In MAP, the most widely used language for defining a task is Multiagent PDDL (MA-PDDL) [33, 34]. MA-PDDL is very similar to standard PDDL, being nothing but a multiagent extension to standard PDDL. We discuss the differences below.

In MA-PDDL, we can define tasks with both *factored* and *unfactored* privacy. Unfactored privacy tasks have a domain definition file and a single task file that serves all the agents. Factored tasks have a domain definition file and an individual task file for each agent, representing the agent's local view. This is done by adding the `:factored-privacy` or `:unfactored-privacy` property to the `requirements` section. Listing 4.1 depicts an example of a domain definition in a factored definition taken from [59].

This example represents the *Transport Agent* domain. In this domain, transport agencies that work in different geographical areas must transport packages from locations within their areas. There is also a factory that resides in the intersection of the agencies area and needs raw material to manufacture a product. The transport agencies must transport the packages of raw material from their current location to the factory, so that the factory can use it to manufacture the final product. A transport agency can only transport a package to the factory if the initial package location resides within its area.

As we can see, the domain definition is fairly similar to a single-agent PDDL domain file. The main difference we see is in the `:factored-privacy` key word, in line 3, which we have already discussed, and the `:private` key word within the `:predicates` section, in line 21. The `:private` key word states that the predicates within that scope will not be shared across the agents, meaning that each transport agency will not disclose any information neither about the topology of their work area nor about their trucks.

```
1  (define (domain transport-agency)
2    (:requirements
3      :factored-privacy
4      :typing
5      :equality
6      :fluents
7    )
8    (:types
9      transport-agency
10     area
11     location
12     package
13     product - object
14     truck
15     place - location
16     factory - place
17   )
18   (:predicates
19     (manufactured ?p - product)
20     (at ?p - package ?l - location)
21     (:private
22       (area ?ag - transport-agency ?a - area)
23       (in-area ?p - place ?a - area)
24       (owner ?a - transport-agency ?t - truck)
25       (pos ?t - truck ?l - location)
26       (link ?p1 - place ?p2 - place)
27     )
28   )
29   (:action drive
30     :parameters (?ag - transport-agency ?a - area ?t - truck ?p1 - place
   ?p2 - place)
31     :precondition (and
```

```
                        (area ?ag ?a)
                        (in-area ?p1 ?a)
                        (in-area ?p2 ?a)
                        (owner ?a ?t)
                        (pos ?t ?p1)
                        (link ?p1 ?p2)
                    )
      :effect (and
                (not (pos ?t ?p1))
                (pos ?t ?p2)
              )
    )
    [...]
  )
```

Listing 4.1: Excerpt from *Transport Agency* domain described in MA-PDDL

Having the domain defined, we can define the task itself. In this example, since it is a factored definition, we have a task file for each agent. Listing 4.2 shows the MA-PDDL task file for agent 1. Firstly, we see that it is not so different from a single-agent problem definition. Secondly, we see that only facts regarding the transport agency 1 are described in the initial state, as the ones regarding any other transport agencies are private to those agencies. Finally, we see that the goal is to have the final product manufactured, as the agencies must work together to make sure that the factory has the necessary raw material to do so. We assume that the factory is another agent and that it will manufacture the product if the transport agencies provide the raw material.

```
(define (problem ta1)
  (:domain transport-agency)
  (:objects
    ta1 - transport-agency
    ga1 - area
    l1 l2 sf - place
    p - package
    fp - product
  )
  (:init
    (area ta1 ga1)
    (pos t1 l1)
    (owner t1 ta1)
    (at p l1)
    (link l1 l2)
    (link l2 l1)
    (link l1 sf)
    (link sf l1)
    (link l2 sf)
    (link sf l2)
```

```
21        (in-area l1 ga1)
22        (in-area l2 ga1)
23        (in-area sf ga1)
24      )
25      (:goal (manufactured fp))
26    )
```

Listing 4.2: Excerpt from *Transport Agency* domain described in MA-PDDL

4.3 Frameworks and Algorithms

Since this work is related to SAT-based techniques, an important planner to mention is μ-SATPLAN [15], which first distributes the goals among the agents and then iteratively feeds each agent with the solution from the previous agent as input. Each agent solves the task using the regular SATPLAN [27] planner, and the agents progressively solve the entire task. Still, this work does not bring anything new into the SAT planning field itself, as it just breaks the task down into single-agent ones and then uses the regular single-agent SATPLAN planner to solve them.

As we see from the approach above, as well as from other approaches, it is common in multiagent planning to just translate a problem into a single-agent one, or to break it down into multiple single-agent ones, and use common single-agent planners to solve them. Some examples of such approaches are Multi-Agent Planner by Plan Reuse (MAPR) [9], Plan Merge by Reuse (PMR) [38], and Centralized Multi-Agent Planning (CMAP) [18]. MAPR has a similar approach to μ-SATPLAN, where it distributes the goals to the agents and each agent receives the solution from the previous agent as input. The difference is that each agent uses the LAMA [52] planner to solve its task. PMR also uses the LAMA planner in as similar way as MAPR, but instead of making the agents solve the task sequentially, it parallelizes the agents, making each of them generate a plan for the distributed goals, and then merges the plan in post-processing. Finally, CMAP also uses LAMA, but it completely translates the MAP task into a standard single-agent planning task, and then uses LAMA to solve it.

5. GOAL RECOGNITION

Goal recognition is the task of correctly identifying an agent's goal by observing its interactions with the surrounding environment [58]. Such observations may translate into actions performed by the agent or properties of the environment during the agent's actions. We define a goal recognition problem over planning domain theory in the same manner as Ramirez and Geffner [50, 51] defined a plan recognition problem. Figure 5.1 illustrates an example of a goal recognition environment. In this example, we observe the agent's movements, depicted in blue, and since we can see that the agent is moving towards the wrench, we could infer that the wrench is the agent's goal. In this chapter, we outline the goal recognition formalism and notation, in Section 5.1, discuss relevant approaches for solving goal recognition problems, in Section 5.2, and expand the formalism from Section 5.1 to multiagent scenarios in Section 5.3.

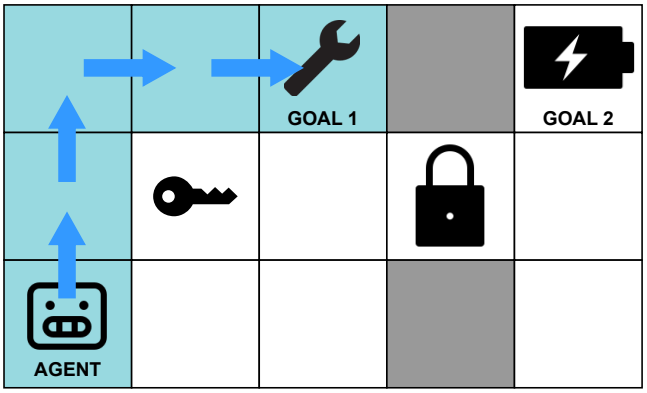


Figure 5.1: Goal recognition environment example

5.1 Formalism and Notation

Research on goal recognition is closely related to that on plan recognition, a variant where one desires to not only recognize the agent's goal, but also the plan this agent is executing to achieve it and that explains the observations. Earlier approaches to plan recognition [30, 23] are based on plan libraries, where the recognizer assumes that the agent was following a plan taken from a previously defined plan library. The algorithms are dependent of this plan library, and the frameworks generate the plans before the recognition process starts. The recognizer then tries to find which plan from the library explains the observations. This was the common approach until a paper by Ramirez and Geffner [50] was published, where the authors interpret a plan recognition problem as a problem over a domain theory, or a planning domain theory, using the formalism and techniques of planning as a base to formulate and solve plan recognition problems. Their work

paved the way to many other works, and shaped the state-of-the-art of how we tackle plan and goal recognition problems. For this reason, this is the kind of approach we focus on this work. Analogously, the majority of goal and plan recognition solutions that are based on planning domain theory use the PDDL language [41] as a base to define the problems.

A goal recognition problem over domain theory can be defined as a tuple $T_{GR} = \langle \Xi, \mathcal{I}, \mathcal{G}, \mathcal{O} \rangle$, in which $\Xi = \langle \Sigma, \mathcal{A} \rangle$ is a planning domain definition; $\mathcal{I}$ is the initial state; $\mathcal{G}$ is the set of goal state hypotheses, which includes the correct intended goal state G^* (*i.e.*, $G^* \in \mathcal{G}$); and $\mathcal{O} = \langle o_1, o_2, ..., o_n \rangle$ is an observation sequence of executed actions, where each observation $o_i \in \mathcal{A}$.

The solution to a goal recognition problem is the hidden goal G^* achieved by the sequence of observations $\mathcal{O}$. We can have either full or partial observations, meaning that we may be observing all the actions performed by the agent or only a subset (contiguous or not) of such actions. Furthermore, observations may contain noise within them, meaning that the observation sequence may include fake or spurious actions, that were not actually performed by the agent. In real-world applications, faulty sensors may cause noise in observations, for instance.

5.2 Frameworks and Algorithms

When we deal with single-agent goal recognition, we see no relevant work on SAT-based approaches. Given that, we focus this section on discussing other relevant goal recognition approaches. We focus on goal recognition approaches based on planning domain theory, as this is the approach we use in our multiagent goal recognition framework, besides being the most common kind of approach in the field since Ramirez and Geffner's work [50].

In [50], the authors develop the formalism to treat a goal recognition problem as a problem over domain theory, instead of using a previously generated plan library and looking for a plan included in the library, which can be costly and unfeasible in many applications. In one of the approaches, the authors use a planner to obtain the cost to each goal, using that cost as a maximum when reusing the planner with the observations added to the goal state, checking if there is an optimal plan that can reach the goal through the observations. As mentioned before, this work is one of the most relevant works in the field, as it changed the way we interpret plan recognition problems.

Later work by the same authors [51] introduces a way to discover the correct intended goal by computing a probability distribution over the set of goal hypotheses. In this later work, the authors compute the probability of each goal hypothesis given the observations. To do that, they use a Bayesian model, and use the cost difference between plans

that follow the observations versus plans that deviate from the observations. To generate those plans, they use off-the-shelf classical planners, such as HSP [8] and LAMA [52].

A different approach [48] is based on landmark analysis. In this work, the authors develop two heuristics to analyze the landmarks for each goal hypothesis, and compare the achieved landmarks to the total landmarks for each of them. The first heuristic computes the ratio between achieved and total landmarks to provide a score for each goal hypothesis. The second heuristic computes the same ratio, but assigns a higher value to landmarks that are landmarks to fewer goal hypotheses, since achieved landmarks that are not shared across different goals provide more information on the agent's intention.

More recent approaches rely on operator-counting constraints [49] to compute heuristic values for goal hypotheses, considering scenarios with and without the observations, and on reinforcement learning [1], by computing a Q-function for each goal hypothesis, and measuring its distance to the observation sequence. Both approaches obtain state-of-the art performance and are competitive with previous goal recognition approaches.

5.3 Multiagent Goal Recognition

In Chapter 4 we discussed over another prism of planning problems where one might want to plan for multiple agents to cooperate as a team in pursuit of a given goal. Conversely, we also have interest in identifying when multiple agents are cooperating as a team, and which goal they are pursuing. This derives an expansion of goal recognition called *multiagent goal recognition* (MAGR).

Figure 5.2 illustrates an example of a multiagent goal recognition environment. In this example, we observe agents 1 and 2's movements, depicted in blue and green, respectively. Since we can see that agent 1 is moving towards the wrench, while agent 2 is moving towards the battery, we could infer that each agent is its own one-agent team, and that the wrench is agent 1's goal, while the battery is agent 2's goal. In this chapter, we describe the formalism and notation for multiagent goal recognition.

To define a MAGR problem, we extended the definition of a single-agent goal recognition problem. The main difference is that we have a set of agents, instead of a single one. This alone adds another variable to the problem, as we have to analyze which agent executed each observed action. Additionally, the agents may cooperate in teams, where each team is pursuing a different goal. Our objective is to find out how are these teams organized, *i.e.* how many teams are there, who (which agents) are the members of each team, and which goal each team is pursuing. We call each of the team-goal combinations a team-goal mapping.

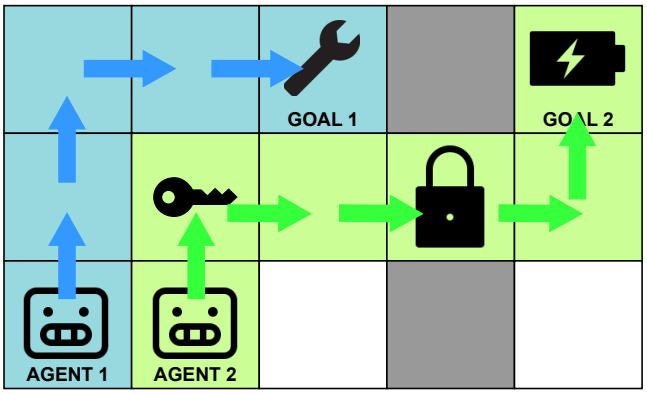


Figure 5.2: Multiagent goal recognition environment example

The different work in MAGR diverge in how they define a task. They have a common ground, such as defining the actions with names, preconditions and effects, defining the possible facts and the initial state, defining the list of agents involved in the process, and defining the observations, or team traces. What the MAGR papers we discuss here mostly differ is in how they treat the possible goals the agents pursue. For instance, the work by Zhuo *et al.* [62, 63, 64] clearly defines a set of goal hypotheses for the recognizer to analyze, while in the work by Argenta and Doyle [2, 3], the authors define the set of hypotheses as the set of all possible goals, not defining a specific set. For convenience, we define a MAGR task as close as to the definition we gave to a single-agent one, with the appropriate adaptations taken from the papers we discuss here.

We define a MAGR task as a tuple $T_{MAGR} = \langle \Xi, \Phi, \mathcal{I}, \mathcal{G}, \mathcal{O} \rangle$, where $\Xi = \langle \Sigma, \mathcal{A} \rangle$ is the domain definition, with the set of facts Σ and ground actions $\mathcal{A}$. $\Phi = \phi_{i=1}^{k}$ is the set of agents, where $k > 0$ and where each agent is a part of one, and only one team, where a team is a set of one or more agents. We denote a given team i as T_i, and the set of all possible teams as $\mathcal{T}$, where $|\mathcal{T}| = 2^k - 1$, since we consider the power set of agents, excluding the empty team. We denote the hidden subset of correct teams as $\mathcal{T}^*$, where $\mathcal{T}^* \subseteq \mathcal{T}$. $\mathcal{I}$ is the initial state. $\mathcal{G}$, where $|\mathcal{G}| = l$, is the set of goal hypotheses, where $l \geq k$, meaning there is a possibility that each agent is its own team pursuing its own goal. Each goal hypothesis might or might not be a correct intended goal for one of the acting teams, and every correct intended goal for each of the acting teams is included in $\mathcal{G}$. We denote the hidden subset of correct intended team goals as $\mathcal{G}^*$, where $\mathcal{G}^* \subseteq \mathcal{G}$. If we combine a team and a goal hypothesis, we obtain a team-goal mapping, or simply a mapping. We denote the set of all possible team-goal mappings as $\mathcal{M}$ and the hidden subset of correct team-goal mappings as $\mathcal{M}^*$, where $\mathcal{M}^* \subseteq \mathcal{M}$, which is the resulting correct combination of $\mathcal{T}^*$ and $\mathcal{G}^*$. $\mathcal{O} = \{\mathcal{O}_i\}_{i=1}^{k}$ are the team traces, which are a sequence of observed actions mapped to the agent who executed them. The solution to a MAGR task is the hidden subset of correct team-goal mappings $\mathcal{M}^*$.

One of the main issues in solving a multiagent goal recognition problem with this definition is the number of possible team-goal mappings. If we consider problems with smaller numbers of agents and hypotheses, for instance two agents and two goal hypotheses, that is not a major issue, as we have only six possible mappings. But if we increase the number of agents, even if we keep the minimal number of hypotheses, equal to the number of agents, the number of mappings increases exponentially, as we see from the graph in Figure 5.3. In this graph, we see how the number of possible mappings grows as we increase the number of agents and goal hypotheses. In this case, we consider that the number of goal hypotheses is equal to the number of agents, as it is the minimum in our definition, but there might be more hypotheses than agents, aggravating that issue. Even for as few as 4 agents, we end up having at least 60 possible mappings, and with 10 agents, we have at least $10,230$ possible mappings.

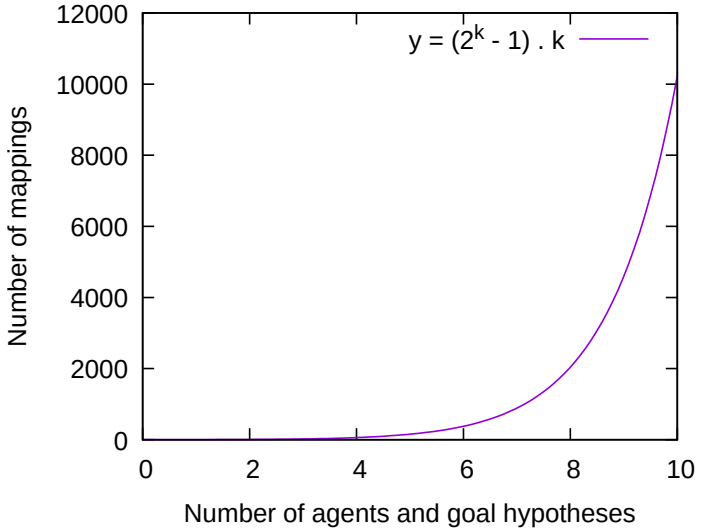


Figure 5.3: Number of mappings as the number of agents and goal hypotheses grow

If we formally define the example from Figure 5.2, the planning domain definition Ξ comprises all the facts Σ that represent the current position of the objects and agents and all the actions $\mathcal{A}$ that represent the agent's movements. The set of agents Φ is $ag1, ag2$, including agents 1 and 2. The initial state is the state representation of the initial position of all objects and agents as depicted in the figure. The set of goal hypotheses $\mathcal{G}$ is a set $g1, g2$ containing the two possible goals, 1 and 2. Finally, the sequence of observations $\mathcal{O}$ is the sequence of movements performed by each agent, as depicted by the arrows in blue and green in the figure.

6. MULTIAGENT GOAL RECOGNITION AS SAT

In this chapter, we develop the architecture of our approach for multiagent goal recognition using MAX-SAT and SAT-based planning. In Section 6.1, we develop the SAT-based planner we use to aid our multiagent goal recognition approach. We then develop our multiagent goal recognition approach, and explain it in detail, in Section 6.2.

6.1 A SAT-Based Planner

Just as in other goal recognition approaches both in single-agent [50, 51] and multiagent [2, 3] scenarios, our multiagent goal recognition approach uses a planner as a tool while recognizing the correct team-goal mappings. We discuss how exactly we use that planner in Section 6.2. In this Section, we present and discuss the SAT-Based planner we use in our recognition process.

As the focus of this work are SAT-based techniques, we naturally use an implementation of a SAT-based planner to aid the recognition process. We leverage an existing implementation provided by the advisor. This planner uses the encoding presented in detail in Section 3.2, using Formulas 3.1 through 3.4. Since we want to consider the possibility of parallel actions to reduce the number of time steps in our plan, and we do not want to account for idle time steps, we also use Formulas 3.6 and 3.7. From now on, we refer to this specific set of formulas as *encoding axioms*.

We solve a planning instance with this planner through Algorithm 6.1. This algorithm receives the planing domain definition, through the list of facts Σ and ground actions $\mathcal{A}$, the initial state $\mathcal{I}$, the goal state G, and the maximum planning horizon $\mathcal{H}$ we want to analyze. For each planning horizon H from 0 up to the maximum planning horizon $\mathcal{H}$, it encodes the planning instance in a boolean formula using the *encoding axioms* for all time steps from 0 up to the current step, in line 3. In line 4, it adds a minimizer to the formula, to try and minimize the sum of na, the number of actions literals set to true across all time steps i in the resulting model. Since the action literals that are set to true in the model represent the plan itself, minimizing them consequently minimizes the parallel plan length, *i.e.* its cost. The cost of a plan is the sum of the costs of the actions that compose that plan. Since in this work all actions have a unit cost, the cost is equal to the number of actions, or the length of the plan. If the resulting formula is satisfiable, *i.e.* the SAT solver finds a valid model for the formula, it extracts the plan from the found model, in line 6. It does that by extracting the action literals that were set to true. The time step that labels that action literal represents the time step in which the action was executed, so we have an ordering for the actions. It then returns the extracted plan, in line 7. If the SAT solver is not able to find a valid model, the algorithm verifies the next horizon, until the maximum

horizon $\mathcal{H}$ is reached. If it reaches the maximum horizon with no found model, it returns an error in line 10, as there is no solution for that planning instance within that maximum planning horizon $\mathcal{H}$.

1: **function** Plan($\Sigma, \mathcal{A}, \mathcal{I}, G, \mathcal{H}$)
2: **for** $H \in [0, \mathcal{H}]$ **do**
3: $\Omega \leftarrow$ conjunction over Equations 3.1 through 3.4, 3.6, and 3.7 for time steps 0 through H
4: $\Omega \leftarrow \Omega \wedge \mathit{minimize} \sum_{i=0}^{H} na_i$
5: **if** Ω is satisfiable **then**
6: $\pi \leftarrow$ extract plan from found model
7: **return** π
8: **end if**
9: **end for**
10: **return error**
11: **end function**

Algorithm 6.1: Algorithm to solve a planning instance with SAT

Although this method is capable of deriving plans for planning tasks, building the entire formula from scratch for every planning horizon can be a major performance issue. Given that the horizon only increases, we can reuse already encoded previous horizons, and only add the current horizon to the formula. For this reason, we implement a variation of the solving algorithm using what we call incremental solving. Through incremental solving we can create a base formula with the initial state and just update it with the goal state and remaining axioms for the current step, not needing to restart the entire process at each step. We also use a resource to push and pop states into our satisfiablity tester. With these features, we can push a temporary state in our solver, add constraints to our formula, test the satisfiability, and then simply pop back to the previous state. This is specially useful to incrementally solve the problems, as we can add the goal state in each horizon in a temporary state, and then pop back to a state with no goal state to prepare for the next iteration.

```
1: function Plan(Σ, 𝒜, ℐ, G, ℋ)
2:     Ω ← initial state encoded through Equation 3.1
3:     for H ∈ [0, ℋ] do
4:         if H > 0 then
5:             Ω ← Ω ∧ time step H encoded through Equations 3.3, 3.4, 3.6, and 3.7
6:         end if
7:         Push state to encode goal state
8:         Ω ← Ω ∧ goal state for time step H encoded through Equation 3.2
9:         Ω ← Ω ∧ minimize Σ_{i=0}^{H} na_i
10:        if Ω is satisfiable then
11:            π ← extract plan from found model
12:            return π
13:        end if
14:        Pop state to prepare for the next horizon
15:    end for
16:    return error
17: end function
```

Algorithm 6.2: Algorithm to incrementally solve a planning instance with SAT

As we can see from Algorithm 6.2, we no longer encode the entire formula for each time step, but rather leverage previously encoded horizons, incrementally adding each horizon to our formula. Additionally, we push a temporary state before adding the goal state to the formula, as we do not want it permanently there. We pop that temporary state if the current formula is not satisfiable, so we can add the goal state to the formula for the next planning horizon. We still minimize the number of parallel actions in the plan and extract the plan from the found model the same way we do in Algorithm 6.1.

6.2 Recognition Process

In the previous Section, we presented the SAT-based planner we use to aid our multiagent goal recognition approach. In this Section, we go over the recognition process itself, how we deal with the goal hypotheses and the possible team configurations, how we analyze the team traces, and how we use the planner we discussed in Section 6.1.

Algorithm 6.3 depicts the multiagent goal recognition process of our approach. We use the same formalism and notation for a multiagent goal recognition task as defined in Section 5.3. The algorithm receives the set of facts Σ and ground actions $\mathcal{A}$ from the planning domain, the initial state $\mathcal{I}$, the set of goal hypotheses $\mathcal{G}$, the set of agents Φ, the team traces $\mathcal{O}$, a threshold Θ, and a maximum horizon $\mathcal{H}$. It returns a set of accepted

mappings, where the mappings included in this set are the ones considered the most likely to be the correct ones according to the algorithm.

1: **function** Recognize(Σ, $\mathcal{A}$, $\mathcal{I}$, $\mathcal{G}$, Φ, $\mathcal{O}$, $\mathcal{H}$, Θ)
2: $\mathcal{T} \leftarrow \mathcal{P}(\Phi) - \emptyset$
3: $\mathcal{M} \leftarrow$ all combinations between $\mathcal{T}$ and $\mathcal{G}$
4: **for** $M \in \mathcal{M}$ **do**
5: $O \leftarrow \bigcup_{i \in M.team} \mathcal{O}_i$
6: EvaluateMapping(Σ, $\mathcal{A}$, $\mathcal{I}$, M, O, $\mathcal{H}$)
7: **end for**
8: Compute mappings scores through Equations 6.1 and 6.2
9: $\mathcal{AM} \leftarrow$ mappings in $\mathcal{M}$ whose scores are up to Θ% lower than the best score
10: **return** $\mathcal{AM}$
11: **end function**

Algorithm 6.3: Algorithm to recognize the team-goal mappings for a multiagent goal recognition task

Since we need to analyze each possible team-goal mapping to infer which ones are the most likely correct ones, we first need to generate them. To generate the teams, we generate the power set of agents, and exclude the empty set, as we have no interest in an empty team, in line 2. We then generate each mapping as each possible combination between the teams and the hypotheses, in line 3.

Once we have all the mappings, we can then iterate over them to analyze each of them separately, as we do in line 3. For each mapping, we filter the team traces in line 4, to only consider the observed actions that were executed by members of the current mapping's team. We then evaluate the mapping through the *EvaluateMapping* procedure, in line 5, which assigns a cost to each mapping.

After all mappings have been analyzed, we can compute the scores for them and return the most likely correct ones. For that, we first compute the score for each mapping using Equations 6.1 and 6.2. In Equation 6.1, we perform an inverse normalization of the mapping's cost. We perform this inverted normalization because a higher cost means a worse mapping, and we want higher scores to represent better mappings. In this Equation, *max* is the highest cost across all mappings, while *min* is the lowest score across all mappings. Once we have this preliminary score computed, we use Equation 6.2 to transform the scores into a probability distribution, by dividing each score by the sum of all scores. We do that as it is more meaningful to display the probability of a given mapping being the correct one instead of just outputting a score.

$$score = \frac{max - cost}{max - min} \tag{6.1}$$

$$score = \frac{score}{\sum_{i=0}^{|\mathcal{M}|} score_i} \tag{6.2}$$

Finally, we select the mappings with the highest scores as the accepted ones, in line 8, and return them, in line 9. To select the mappings, we use their scores and the threshold Θ. The threshold is a relaxation of the algorithm's mapping selection criteria, and it tells the algorithm how worse, when compared to the best-scoring mapping, a mapping can be to be considered acceptable. The algorithm will accept any mappings that have a score up to Θ% lower than score of the best-scoring mapping.

To evaluate each mapping individually, we use the *EvaluateMapping* procedure, depicted by Algorithm 6.4. This algorithm receives the facts Σ and ground actions $\mathcal{A}$, the initial state $\mathcal{I}$, the current mapping being evaluated M, the filtered observations for the mapping O, and the maximum planning horizon $\mathcal{H}$. We base our mapping evaluation process on Ramirez and Geffner's work [51], where they compare the cost of a plan that deviates from the observations against the cost of a plan that follows the observations. The algorithm iterates over each horizon H from 0 up to the maximum horizon $\mathcal{H}$. For each horizon, it encodes the planning formula using Formulas 3.1 through 3.4, 3.6, and 3.7 for all horizons from 0 up to H, considering the goal for mapping M, in line 3. Then, in line 4, it adds the minimizer to the formula, to minimize the number of parallel actions in the extracted plan. This planning and minimization step uses is the same we see in Algorithm 6.1.

Once the planning formula is encoded, the algorithm deals with the traces. For that, it pushes a temporary state, and then adds soft constraints to avoid the observed traces, in lines 5 and 6. These constraints state that the SAT solver should avoid setting literals that represent the traces to "true" in the model. Since they are soft constraints, the solver will try to satisfy them, but will still return a model that doesn't satisfy them if they cannot be satisfied for that horizon, making this a MAX-SAT problem. If the solver finds a model, the algorithm stores the model's plan in $\bar{\pi}$, in line 8, returns to the original state, in line 9, and then adds constraints to find a model that follows the traces, in lines 10 and 11. For that, it does the opposite of what it did in line 6, and adds soft constraints in line 10 to state that the solver should try to set the literals that represent the traces to "true" in the model. In line 11, it adds hard constraints to enforce the traces ordering. What it states in these constraints is that, if a given trace a comes after a given trace b in the sequence of traces, then they must be in that same order in the plan. However, it only adds these constraints for actions that are pairwise mutually exclusive, as we want to allow parallel actions.

After all constraints have been set, the solver tests the formula for satisfiability, and extracts the plan from it into π, in line 13. It then computes the cost for that mapping, in line 14, and resumes back to the *Recognize* procedure. To compute the cost, it uses the cost difference between π and $\bar{\pi}$, inspired by Ramirez and Geffner's work [51], and it also adds the number of traces in O that were not included in π, represented by $|O - \pi|$, and the number of actions in π that were not observed in O, represented by $|\pi - O|$, inspired by Shvo *et al.*'s work [56].

As we have discussed in Section 5.3, the number of mappings grows exponentially as we increase the number of agents, as depicted in Figure 5.3. Given that, and the way our approach works, the main component that makes solving a multiagent goal recognition problem costly in our approach is the fact that we have to repeat this analysis for each possible team-goal mapping.

1: **procedure** EvaluateMapping($\Sigma, \mathcal{A}, \mathcal{I}, M, O, \mathcal{H}$)
2: **for** $H \in [0, \mathcal{H}]$ **do**
3: $\Omega \leftarrow$ conjunction over Equations 3.1 through 3.4, 3.6, and 3.7 for time steps 0 through H
4: $\Omega \leftarrow \Omega \wedge \mathit{minimize} \sum_{i=0}^{H} na_i$
5: Push state to add trace-avoiding constraints
6: $\Omega \leftarrow \Omega \wedge$ soft constraints to avoid the traces
7: **if** Ω is satisfiable **then**
8: $\bar{\pi} \leftarrow$ extract plan from found model
9: Pop state to add trace-following constraints
10: $\Omega \leftarrow \Omega \wedge$ soft constraints to follow the traces
11: $\Omega \leftarrow \Omega \wedge$ hard constraints to enforce trace ordering
12: **if** Ω is satisfiable **then**
13: $\pi \leftarrow$ extract plan from found model
14: $M.\text{cost} \leftarrow (\mathit{cost}(\pi) - \mathit{cost}(\bar{\pi})) + |O - \pi| + |\pi - O|$
15: **return**
16: **end if**
17: **end if**
18: **end for**
19: **end procedure**

Algorithm 6.4: Algorithm to evaluate a team-goal mapping and assign a cost to it

In Section 6.1 we present Algorithm 6.2, a variation of Algorithm 6.1 that uses incremental solving as a way to increase runtime performance. In that same manner, we also have a version of the *EvaluateMapping* procedure that leverages incremental solving, with the same runtime performance goal in mind. Algorithm 6.5 depicts this alternative

way to evaluate a mapping. As we can see, we use the exact same strategy as the one depicted in Algorithm 6.2, encoding a base formula and incrementing it for each time step. We also need to push temporary states to add the goal state, as well as the trace-following constraints, to prepare for a possible next iteration.

1: **procedure** EvaluateMapping($\Sigma, \mathcal{A}, \mathcal{I}, M, O, \mathcal{H}$)
2: $\Omega \leftarrow$ initial state encoded through Equation 3.1
3: **for** $H \in [0, \mathcal{H}]$ **do**
4: **if** $H > 0$ **then**
5: $\Omega \leftarrow \Omega \wedge$ time step H encoded through Equations 3.3, 3.4, 3.6, and 3.7
6: **end if**
7: Push state to encode goal state
8: $\Omega \leftarrow \Omega \wedge$ goal state for time step H encoded through Equation 3.2
9: $\Omega \leftarrow \Omega \wedge \mathit{minimize} \sum_{i=0}^{H} na_i$
10: Push state to add trace-avoiding constraints
11: $\Omega \leftarrow \Omega \wedge$ soft constraints to avoid the traces
12: **if** Ω is satisfiable **then**
13: $\bar{\pi} \leftarrow$ extract plan from found model
14: Pop back to state with no trace constraints
15: Push state to add trace-following constraints
16: $\Omega \leftarrow \Omega \wedge$ soft constraints to follow the traces
17: $\Omega \leftarrow \Omega \wedge$ hard constraints to enforce trace ordering
18: **if** Ω is satisfiable **then**
19: $\pi \leftarrow$ extract plan from found model
20: $M.\text{cost} \leftarrow (\mathit{cost}(\pi) - \mathit{cost}(\bar{\pi})) + |O - \pi| + |\pi - O|$
21: **return**
22: **else**
23: Pop back to state with no trace constraints
24: Pop back to state with no goal
25: **end if**
26: **else**
27: Pop back to state with no trace constraints
28: Pop back to state with no goal
29: **end if**
30: **end for**
31: **end procedure**

Algorithm 6.5: Algorithm to incrementally evaluate a team-goal mapping and assign a cost to it

6.3 A Working Example

In this section, we discuss an example of a multiagent goal recognition problem that can be solved using the approach discussed in the previous section. In this hypothetical problem, we have two agents in a *Blocks World* environment. The agents can pick any given free block (a block that has no blocks stacked on top of it) up from the table, put it down directly above the table, or stack it on top of any other free block above the table, or unstack any free block that is sitting on top of another block. So far we hav our problem's domain and the set of agents Φ defined, where $\Phi = \{ag1, ag2\}$. The next thing we need is the initial state and the goal hypotheses. Figure 6.1 depicts these problem components. In this example, we have three blocks, labeled A, B and C. In the initial state, all three blocks sit free on the table, there are no stacked blocks. As per the definition in Section 5.3, since we have two agents operating on the environment, we must have at least two goal hypotheses. Given that, we define our two goal hypotheses also in Figure 6.1, labeled as goal hypotheses 1 and 2.

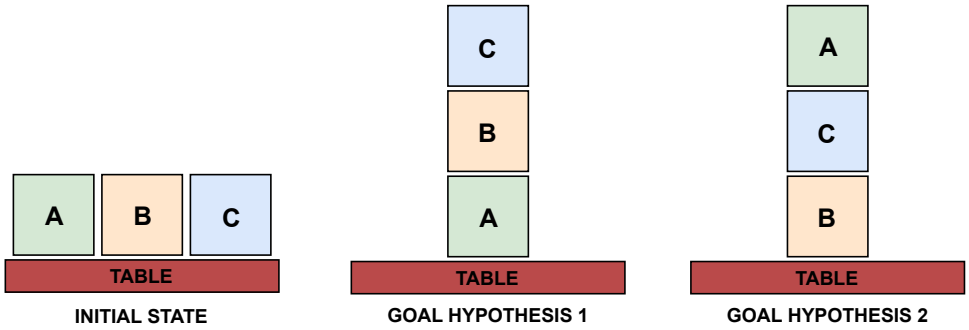


Figure 6.1: Initial state and goal hypotheses

The last component we need for our problem are the team traces. For that, we first define how the agents are organized and which goal hypothesis they are pursuing. In this example, there is only one team, composed by both agents *ag*1 and *ag*2, and they are pursuing goal hypothesis 1 together, which makes goal hypothesis 2 a fake hypothesis. For this setup, we have the following observed actions as team traces depicted in Listing 6.1. We see that agent *ag*1 picks block B up, while agent *ag*2, picks block C up. Agent *ag*1 then stacks block B on top of block A, and agent *ag*2 stacks block C on top of block B, creating the block stack that represents goal hypothesis 1. In this example, we consider a problem with full observability, where we observe the interactions of the agents with the environment in their entirety.

```
1  (pickup ag1 b)
2  (pickup ag2 c)
3  (stack ag1 b a)
4  (stack ag2 c b)
```

Listing 6.1: Sequence of team traces

Now that we have every aspect of our problem defined, we can start the solving process. It is important to note that even though we know the correct team-goal mapping while designing the problem, our approach does not know it, and will analyze every possible mapping to infer the correct one.

To solve this problem, we use Algorithm 6.3. The first thing the algorithm does is generating every possible mapping. Since we have two agents and two goal hypotheses, we have six possible team-goal mappings, labeled 1 through 6, depicted in Table 6.1.

Mapping	Team	Goal
1	$ag1$	1
2	$ag2$	1
3	$ag1, ag2$	1
4	$ag1$	2
5	$ag2$	2
6	$ag1, ag2$	2

Table 6.1: Every possible team-goal mapping in our example problem, labeled 1 through 6

The algorithm then starts to analyze each mapping separately. For convenience, we only discuss here the in-depth analysis of one of these mappings. We analyze mapping 3, the correct one. To analyze a single team-goal mapping, we use Algorithm 6.5.

The algorithm must generate two plans: one plan that tries to avoid the team traces, and one plan that tries to follow the traces, maintaining the order of the mutually-exclusive traces. Both agents can pick blocks B and C at the same time, but the one of them must stack block B on top of block A before the other one can stack block C on top of block B. This means that even though we have four actions, we can execute them in only 3 time steps, so we need at least four time steps for a valid goal-achieving plan to be executed, given that time step 0 is the one that represents the initial state, before any action has been executed, and time step 3 represents the state after the last action in the plan has been executed. The algorithm needs to iterate through every horizon encoding the formula and checking for satisfiability, until it finally finds a model in horizon 3. We depict the plan that avoids the traces in Table 6.2, while the plan that follows the traces is depicted in Table 6.3. We see that both plans have the exact same length, having the same number of time steps and parallel actions. Hence, the cost difference between them is zero.

Now the algorithm compares the trace-following plan from Table 6.3 to the team traces from Listing 6.1. We see that the trace-following plan includes all actions observed in the team traces, and does not include any non-observed actions. Given that, we can

Action Count	Action	Time Step
1	`pickup ag2 b`	0
2	`pickup ag1 c`	0
3	`stack ag2 b a`	1
4	`stack ag1 c b`	2

Table 6.2: Plan that avoids the traces

Action Count	Action	Time Step
1	`pickup ag1 b`	0
2	`pickup ag2 c`	0
3	`stack ag1 b a`	1
4	`stack ag2 c b`	2

Table 6.3: Plan that follows the traces

safely say that mapping 3 has a cost of zero. Algorithm 6.5 then returns to Algorithm 6.3, which will perform the same process with every other mapping. At the end, we have every team-goal mapping with an associated cost. For this example, given the way Algorithm 6.5 works, the cost for each mapping would be as depicted in Table 6.4.

Mapping	Team	Goal	Cost
1	$ag1$	1	4
2	$ag2$	1	4
3	$ag1, ag2$	1	0
4	$ag1$	2	8
5	$ag2$	2	4
6	$ag1, ag2$	2	4

Table 6.4: Cost for each mapping

With the cost for each mapping, the algorithm would then compute the score for each mapping, performing the inverted normalization from Equation 6.1. These scores are depicted in Table 6.5.

Mapping	Team	Goal	Score
1	$ag1$	1	0.5
2	$ag2$	1	0.5
3	$ag1, ag2$	1	1.0
4	$ag1$	2	0.0
5	$ag2$	2	0.5
6	$ag1, ag2$	2	0.5

Table 6.5: Score for each mapping

Finally, the algorithm computes the probabilistic score by dividing each score by the sum of all scores, following Equation 6.2. We depict the resulting scores in Table 6.6.

We see that mapping 3 achieved the highest score, a score of 0.3333. If we consider a threshold value of 0%, then Algorithm 6.3 would return only mapping 3 as an accepted mapping. So we see that, for this example, the algorithm would correctly infer the team-goal mappings.

Mapping	Team	Goal	Score
1	$ag1$	1	0.1667
2	$ag2$	1	0.1667
3	**ag1, ag2**	**1**	**0.3333**
4	$ag1$	2	0.0
5	$ag2$	2	0.1667
6	$ag1, ag2$	2	0.1667

Table 6.6: Score for each mapping

7. EXPERIMENTS AND RESULTS

In the previous chapter, we have discussed our multiagent goal recognition approach in detail, and presented a runtime performance enhancement we have added to it, using incremental solving. Now we wish to evaluate the recognition and runtime performance of the recognizer we implement our approach on, and verify whether it is actually able to recognize the correct team-goal mappings in reasonable time, and whether our enhancement made a performance impact on it. Moreover, we present yet another performance enhancement that changes the way we parse our problems. In this chapter, we discuss the SMT solver we use to encode and solve our problems, in Section 7.1. In Section 7.2, we describe our recognizer's problem input format, and in Section 7.3, the problems we use to evaluate our recognizer. In Section 7.4, we describe the second performance enhancement we implement in our recognizer. Finally, we discuss the benchmarking strategy and the metrics we use, in Section 7.5, and the actual results, in Section 7.6.

7.1 SMT Solving with Z3

In this Section, we present and discuss Z3, the SMT solver we use in our work. Z3 is an SMT solver developed by Microsoft Research [14]. It is a state-of-the-art SMT solver, offering a range of different tools, from the usual logical structures needed for SAT-solving purposes, to arithmetic operations, arrays, bitvectors, regular expressions, strings, sequences, and even custom datatypes.

The solver allows the user to define custom solving strategies, which is particularly useful given that existing solving strategies may not suit all problem classes, allowing the user to use Z3 to solve different kinds of problems. In the documentation [42], they define these strategies as *tactics*. A tactic receives a goal (a set of formulas to solve) or a set of goals as an input, and may simplify the formula in a certain way, or eliminate some variables, for instance. The solving tactics can be combined using *tacticals*, which are commands one can use to compose tactics. Using tacticals to combine different tactics, the user can define the order in which Z3 executes the tactics, so that it feeds the subgoals generated by one tactic into another tactic, parallelize the solving of goals with different tactics, and even treat a possibly failing tactic by applying another one.

Besides its SMT capabilities, Z3 also has optimization features added to it [6], making it an OMT solver as well. These features allow the user to perform arithmetic optimization, such as trying to minimize or maximize certain properties in the model, such as the number of literals with a "true" truth-value, for example. The user can also define soft constraints, which are constraints the solver will try to satisfy, but not satisfying

them does not mean the whole formula is unsatisfiable. Soft constraints might also have a weight associated to them, which serves the purpose of informing Z3 on which soft constraints it should prioritize when trying to satisfy them.

Z3 uses the language defined by SMT-LIB [5] as its input language. The SMT-LIB description language is based on LISP, and it was designed with the purpose of standardizing the input and output languages used in SMT solvers. The solver also provides a set of APIs for multiple programming languages, such as C, C++, .NET, Java, ML/OCaml, and Python, besides distributing Javascript bindings. These APIs allow the user to use the SMT and optimization features of Z3 to its full extent inside a program written in any of those languages, importing Z3 as a library and calling functions to add constraints or test satisfiability, for example.

In this work, we use Z3's Python API. More specifically, we solve our formulas using the default `Optimize` class, which exposes all the methods we need to work with the optimization module and soft constraints. We use both hard and soft constraints with the default `Optimize` class, with no custom tactics, as well as a minimizer to minimize the number of literals set to true in the model. We also use the available `push` and `pop` methods, through which we can push a disposable state, add soft and hard constraints, and then pop it in a stack-like mechanism when we do not want those constraints applied to our optimizer anymore. By calling the `pop` method, we restore the optimizer state to the one we had before calling the `push` method.

7.2 Problem Input Format

Before discussing the dataset of problems we use to evaluate our recognizer, we must first discuss how we define all the properties of a multiagent goal recognition problem in a way the actual implementation of the approach can parse it and solve it. In this section, we go over the input format for the problems our multiagent goal recognizer can solve.

In Section 5.3, we have defined the formalization of a multiagent goal recognition problem we use in this work. Naturally, we use that formalization as a basis for our recognizer's input format. Given that, we use PDDL [41] as an input language for some of the files in the problem description. We choose to not use MA-PDDL [34], as the recognizer has access to a whole view of the environment, and we do not need to factor the problem into multiple sub-problems.

Since we need multiple different information to define an entire problem, we break the problem description into a set of files, more specifically, seven different files. Each of these files contains a part of the problem definition.

As in a planning problem, the first thing we need is a planning domain definition. We represent the domain and all of its possible facts and actions with a PDDL domain file, following the standard PDDL syntax for domain definition, like the one we present in Listing 2.1. It represents the Ξ portion of a multiagent goal recognition problem defined in Section 5.3, and we name it `domain.pddl`. Listing 7.1 is an excerpt from the actual *Blocks World* domain definition PDDL file we use in our experiments.

```
(define (domain ma-blocksworld)
  (:requirements :strips :negative-preconditions)
  (:types agent block)
  (:predicates (clear ?x) (onTable ?x)
               (holding ?ag ?x) (on ?x ?y)
               (equal ?x ?y) (handempty ?ag)
               (agent ?ag)
  )

  (:action pickup
    :parameters (?ag ?ob)
    :precondition (and
                    (clear ?ob)
                    (onTable ?ob)
                    (agent ?ag)
                    (handempty ?ag)
                    (not (agent ?ob))
                  )
    :effect (and
              (holding ?ag ?ob)
              (not (clear ?ob))
              (not (onTable ?ob))
              (not (handempty ?ag))
            )
  )

  [...]
)
```

Listing 7.1: Excerpt from a multiagent goal recognition *Blocks-World* domain file

The main difference we notice from the *Blocks World* domain file from Listing 2.1 is that we have predicates that define that a given object is an agent, and that a given agent is free to pickup blocks. These are necessary as we need to uniquely represent the ability of each agent acting at any time step. In the same manner, we have a parameter in each action that represents the agent executing that action, as we need to identify the action's executor.

As in planning solutions, we also provide our recognizer with a PDDL file representing the problem itself, with the same structure as the one presented in Listing 2.2.

There are, however, key differences in the problem file for a multiagent goal recognition problem when compared to an automated planning problem one. Since we analyze the different goal hypotheses, we leave the goal state section as a placeholder, so we can programmatically replace it with each hypothesis. Similarly, we analyze each agent combination as a team, so we leave the list of team objects (agents) as a placeholder, doing the same with the list of team atoms, which are predicates that assert facts about the agents, such as the fact that a given object is an agent, or the fact that an agent is not holding any blocks at that moment. For that reason, we do not call it a problem file, but rather a template file, as it serves as a template for the problem files we generate during the recognition process with each team-goal mapping, and we name it `ma-template.pddl`. This file also contains the initial state $\mathcal{I}$ for our problem. Listing 7.2 depicts one of the template files we use in our experiments.

```
(define (problem problem-p0)
  (:domain ma-blocksworld)
  (:objects a b c d e f
            <TEAM-OBJS>
  )
  (:init (equal a a) (equal b b) (equal c c)
         (equal d d) (equal e e) (equal f f)
         (onTable f) (on a f) (on c a) (clear c)
         (onTable b) (clear b) (onTable e)
         (on d e) (clear d)
         <TEAM-ATOMS>
    )
  (:goal (and
    <HYPOTHESIS>
  ))
)
```

Listing 7.2: *Blocks World* multiagent goal recognition template file

As we have left placeholders for our agents and their atoms in the template file, we need to provide a list of the agents, representing Φ in the problem definition, and the list of all the atoms that are associated with each agent. For this, we have two separate files, one for the list of agents, and one for the list of atoms associated with each agent. The list of agents is simply a list of agent identifiers separated by new line characters. The list of team atoms is a list of predicates with a placeholder `<AGENT>` for the agent name, as we must replicate them for each agent in the agents list with their identifiers. Listings 7.3 and 7.4 represent these files, named `agents.dat` and `team-atoms.dat`, respectively.

```
ag1
ag2
ag3
ag4
```

Listing 7.3: List of agents

```
(agent <AGENT>) (handempty <AGENT>)
```

Listing 7.4: List of team atoms

In our formal definition of a multiagent goal recognition file, we also define a set of goal hypotheses as $\mathcal{G}$, where that set contains all the goals being actually pursued by each team, besides possibly containing fake goal hypotheses. Naturally, we have a separate file for that, named `hyps.dat`. This file is simply a list of possible goal states separated by new line characters, as depicted in Listing 7.5.

```
(on c a),(on f c)
(on b e),(on d b)
(on d b),(on e d)
(on a c),(on f a)
```

Listing 7.5: List of goal hypotheses

The final component we must define for a multiagent goal recognition problem is the sequence of team traces $\mathcal{O}$, which we analyze to determine which team-goal mappings are the most likely correct ones. We also need a separate file for that, named `obs.dat`, which is simply a list of observed actions separated by new line characters. In our PDDL definition, we defined that the first parameter of each action is the agent that performed that action, and that applies for the actions in the team traces, so we know exactly which agent executed each action. Listing 7.6 is an example of a list of team traces.

```
(unstack ag1 c a)
(putdown ag1 c)
(unstack ag1 a f)
(putdown ag1 a)
(pickup ag1 c)
(stack ag1 c a)
(pickup ag1 f)
(stack ag1 f c)
(unstack ag4 d e)
(pickup ag2 e)
(stack ag4 d b)
(stack ag2 e d)
```

Listing 7.6: List of team traces

With the listings above, we can provide our recognizer with all the information needed to solve a multiagent goal recognition task, as defined in Section 5.3. Still, to determine whether our recognizer correctly recognized the team-goal mappings, we must have the correct result defined somewhere, so that we can compare the result returned

by the recognizer against a ground-truth. For this reason, we define yet another file, named `realTeamHyp.dat`, containing the correct result, to be used as a ground-truth after the recognition process is complete. It is important to note that we do not use this file during the recognition process, and that it only serves the purpose of knowing whether our recognizer was correct or not, for benchmarking purposes. Listing 7.7 depicts an example of this file, containing a list of the correct goal hypotheses, each associated to a list of agents (a team), separated by new line characters.

```
1 ag1: (on c a),(on f c)
2 ag2,ag3,ag4: (on d b),(on e d)
```

Listing 7.7: List of correct team-goal mappings

Having all of those files properly defined, we can then feed them into our recognizer. To do this, we compress them into a single `tar.bz2` file, and feed that file into the recognizer.

7.3 Dataset

To benchmark our solution, our initial need is a set of problems to solve using our approach and extract performance metrics. Our initial idea was to use the dataset from other works on multiagent goal recognition [2, 3, 63, 56], or adapt an existing single-agent goal recognition dataset if we could not obtain or adapt one from related work in the field. Due to time constraints, we could not obtain a dataset from previous work, and adapting a single-agent one showed to be a lengthy process, so we choose to generate a small set of multiagent goal recognition problems from scratch to serve as a benchmark dataset for our approach.

We only generate problems from one problem domain: *Blocks World*, due to time constraints. As we have discussed in Section 2.1, in the single-agent *Blocks World* domain, the agent sees itself in an environment with different blocks over a table, each block with a unique identifier, and the agent must move and stack the blocks to have them in a specific goal position. In a multiagent scenario, the environment and goal remain the same, but there are multiple agents moving blocks simultaneously, and these agents are organized in teams, where each team has a different goal position for the blocks. It might be the case where the teams' goals are conflicting, for example when one team wants a given block in a certain position while another team wants that same block in another position. In our problems, that does not happen. Each team has goals regarding a separate set of blocks, and the initial state guarantees that no two teams must move each other's blocks.

For all problems, we use a configuration of 4 agents, labeled `ag1` through `ag4`, divided into 2 teams. The teams might have 2 agents each, or 1 and 3 agents, randomly

varying this along the problems. All problems have 6 blocks in the environment, labeled `a` through `f`. Each team's goal is related to 3 of the 6 blocks, and the goals' blocks do not intersect. Each goal is a stack of blocks in a specific order, and we randomly assign them to the teams. The reason we use 3 blocks per team is that each team might have 1, 2, or 3 agents, and so each agent would have at least one block to handle.

Once we have the teams and the correct intended goal for each team defined, we can then define the initial state, which we do by randomly shuffling the goal stack ordering and randomly splitting the ordering. With this random generation, we are able to get different orderings that might be harder or easier depending on the goal state. Additionally, we guarantee that the ordering for the initial state is not the same as the on for the goal state. Finally, we randomly split the ordering to obtain a wider variation of initial states. With this random splitting, we could have the blocks not completely stacked up at the initial state, and having one of them clear above the table. For example, if the ordering we select for our initial state is $\langle a, b, c \rangle$, we could have the 3 blocks stacked in that order, or have block a clear over the table and blocks b and c stacked in that order, or have blocks a and b stacked in that order and block c clear above the table.

The next step is generating the goal hypotheses. For that, we randomly shuffle the goal states block ordering from both teams until we have 2 orderings that are different from the correct goal ones. Since the correct intended goals must be included in the hypotheses list, the final product of our hypothesis generation process is a set of 4 hypotheses, where 2 of them are the correct intended goals for each team, while the other 2 are fake hypotheses. It is important to note that even though we ensure that the fake hypotheses differ from the correct ones, we do not ensure that same differentiation from the initial state. This means that it might be the case that the initial state satisfies (or partially satisfies) one or more fake hypotheses. This contributes to harder problems, especially ones where we have lower observation degrees, *i.e.* we are only able to observe a small subset of the actions performed by the agents, as it is much harder to disambiguate between the correct and fake hypotheses when the fake one is already satisfied, and we do not see the agents doing much to change that scenario.

The final component we must have in a problem are the team traces, the observed actions performed by the agents. We use the SAT-based planner defined by Algorithm 6.2 to generate the plans where we take the traces from, and we generate the plans separately for each team. For each team, we feed the planner with the facts and actions regarding that team's members and the blocks in the environment, the initial state, and the goal for that team. Since the planner allows parallel actions, and it tries to find a plan within each horizon from 0 up to the configurable maximum horizon, it will try to have the agents perform parallel actions as much as possible to reach the goal within the least time steps possible. This makes it easier to have all agents on the team performing actions. However, even though we prevent the initial state from completely satisfying

the goal state, there are cases where the initial state partially satisfies the goal state. In those cases, one of the agents may not perform any actions in the plan, as only part of the team is needed to achieve the goal in minimum time steps. This also contributes to harder problems, as the recognizer would not be able to differentiate between a mapping with or without that idle agent, even if we have full observability over the agents actions.

With the steps above, we have our problems with full traces, or 100% observability degree, where we observe the interaction of the agents with the environment in its entirety. For our experiments, we also want problems with partial traces, where we only have access to a subset of the agents' interactions, which might be contiguous or not. For that, we randomly select traces from the fully observed problem to compose the traces of the partially observed ones, while keeping their order intact. The number of traces selected depends on the observability degree for the problem we are generating, and is a percentage of the number of traces in the fully observed problem. In this work, we use problems with full traces, with an observability degree of 100%, and partial traces with four observability degrees: 10%, 30%, 50%, and 70%.

We initially generate 30 problems with full observations, and for each of those problems, we generate three partially observed problems for each partial observability degree. We do this to variate the selected actions in each problem. Due to time constraints to obtain the results and the recognizer taking too long to run each problem, we had to reduce the dataset. As a result, we randomly selected 10 of the 30 generated problems. Additionally, we only used the first observation variation for each problem. As a result, we have 10 problems in each of the five observability degrees.

7.4 Using the PDDL Parser to Increase runtime performance

As we have discussed in Section 6.2, we evaluate each team-goal mapping separately, and compare their individual scores to select the most likely correct ones. To do so, we use the `template.pddl` file discussed in Section 7.2 to generate a complete PDDL planning problem file for each mapping, replacing the `<TEAM-OBJS>` tag for the agents identifiers, the `<TEAM-ATOMS>` tag for the predicates regarding each agent in the initial state, and the `<HYPOTHESIS>` tag for the mapping's goal.

While the team objects change with the different teams, we fixate other objects, such as the blocks in *Blocks-World* domain, appearing both in the objects list and in the initial state. For most mappings, however, some objects are irrelevant, neither being a part of the mapping's goal, nor being related to a relevant object, by being under or over it, for example. This causes irrelevant objects to be parsed and included in the set of facts Σ, the set of ground actions $\mathcal{A}$, and the initial state $\mathcal{I}$, increasing the encoding length, and hence the recognition time, as there are more literals for the solver to consider. With that

in mind, we develop a way to filter out irrelevant objects, so they are not encoded in the formula, thus reducing the encoding length and recognition time.

To filter out the irrelevant objects for the mapping, we first select all the relevant objects. These are the agents themselves, and any objects present in the mapping's goal and in the team traces. Once we have selected the initial relevant objects, we then search the initial state for facts that have at least one relevant object, and add all the objects in that fact to our set of relevant objects. We iterate over the initial state multiple times, until no new objects are added to the set of relevant objects. Once that process is complete, we have a set containing all the relevant objects we must encode, and we can ignore any objects that are not a part of that set. We then filter the initial state, removing facts that contain no relevant objects.

7.5 Benchmarking our Approach

In the previous Section, we discuss the problem dataset used to empirically evaluate the performance of our multiagent goal recognition approach. In this Section, we discuss how exactly we use that dataset to evaluate our approach. We use Section 7.5.1 to discuss the benchmark strategy, how, and in what settings we run the problems to evaluate our approach. We discuss the metrics we use to evaluate our approach in Section 7.5.2. Finally, we discuss the hardware and software environment we use to run our recognizer during the experiments in Section 7.5.3.

7.5.1 Benchmark Strategy

We have a few different settings to run the problems and, for each of these settings, we run each problem on the dataset once. The dataset already provides us with problems with different observability degrees, which allows us to understand how the amount of available team traces affects our recognizer's capability of correctly recognizing the team-goal mappings.

In Sections 6.2 and 7.4, we have discussed two runtime performance enhancements we implement in our recognizer. The former is the incremental version of the mapping evaluation algorithm, while the latter is the exclusion of irrelevant objects when parsing the PDDL file that represents each mapping. We want to evaluate how much each of these enhancements affect the runtime performance of our approach. To evaluate this, we run our recognizer with none of these enhancements, then we run it with only the incremental solving enhancement, to evaluate how that enhancement alone affects the

performance, and then we run it with both enhancements, to evaluate how the object filter affects the performance.

Finally, as we have discussed in Section 6.2, our approach receives a threshold Θ as a parameter. What that threshold does is making our approach more indulging when accepting mappings as correct based on their scores. Although we use 0% as our default threshold, we also wish to understand how different thresholds affect our recognizer's results, so we experiment with three other threshold values: 10%, 20%, and 30%. We only run the threshold experiments with both enhancements, as we only use the threshold in our approach when we select the accepted mappings based on their scores, not being related to the enhancements.

7.5.2 Metrics

To evaluate the performance of our multiagent goal recognition approach, we also define a set of metrics. The combination of these metrics will provide us with the information needed to understand our approach's performance in the different settings we experiment in, and understand how it behaves in these settings.

The first metric we use is the recognition accuracy. This metric represents the average percentage of correct team-goal mappings in the set of accepted mappings returned by the recognizer. For example, if we have a problem with 2 teams, each following a goal, our recognizer's accuracy could be 100%, if both correct mappings are included in the accepted mappings set, 50% if only one of the correct mappings is included in the set, and 0% if none of the correct mappings are included in the set. This metric shall inform us on whether our recognizer is actually including the correct mappings in its set of accepted mappings.

Even though we can infer the recognizer's correctness with the accuracy metric, it does not provide us with a precision information. If we only use the accuracy metric, we would have an accuracy of 100% if our recognizer only returns the correct mappings as the accepted ones, but we would also have an accuracy of 100% if the recognizer returns all possible mappings as accepted. For this, we use an additional metric, called *spread in* $\mathcal{M}$ ($\mathcal{S}$ in $\mathcal{M}$), or simply *spread*. The spread is the average cardinality of the set of accepted mappings. This allows us to evaluate whether our recognizer's high accuracy is just an effect of it returning too many mappings, or whether it is actually being precise in its process. What we want ideally is an accuracy of 100% and a spread that is equal to the cardinality of the set of correct mappings ($|\mathcal{M}^*|$).

We also use additional metrics to aid our performance evaluation process, such as the average number of mappings analyzed in each problem ($|\mathcal{M}|$), the average number of correct team-goal mappings in each problem ($|\mathcal{M}^*|$), the average number of team traces

in each problem $|\mathcal{O}|$, the average number of agents in each problem ($|\Phi|$), the average number of goal hypotheses in each problem ($|\mathcal{G}|$), and the number of problems in each setting ($|P|$). Finally, we use a time metric, that represents the average time in seconds our recognizer took to solve a problem.

7.5.3 Hardware and Software Environment

We run our experiments in a computer owned by Pontifical Catholic University's (PUCRS) Autonomous Systems Lab (LSA), for convenience, availability, and performance reasons. The machine we run our experiments on has the following hardware characteristics:

- Intel® Xeon® CPU E5-2620 v3 @ 2.40GHz
- Nvidia® Quadro® K2200 GPU
- 16 GB DDR4 RAM Dual Channel (2×8) @ 1866 MHz

The system OS is GNU/Linux Ubuntu 20.04.5 LTS. We write the entire code for our recognizer in Python 3, and we run it using Python 3.8.10.

7.6 Results

In the previous sections of this chapter, we have presented the problems we use to evaluate our approach, our benchmark strategy, the metrics we use, and the environment we run our recognizer on. In this section, we present and discuss the results obtained from our experiments.

Our first goal is to evaluate how the performance enhancements we implement affect the runtime performance. What we want to verify is that the enhancements actually improve the recognition time, while maintaining the recognition performance results unchanged, as they should only make the process faster, but should not affect any other metrics, such as accuracy and spread in $|\mathcal{M}|$.

In Table 7.1, we can see the results for three different enhancement settings in the rows. In the first row, we have no enhancement at all, in the second row, we have the incremental solving enhancement only, as described in Sections 6.1 and 6.2, and in the third row, we have both incremental solving and the object filtering enhancement, as described in Section 7.4, combined. All results in this table were obtained with a 0% threshold. In the columns, we see the different metrics, as the average number of problems $|P|$, the average number of agents $|\Phi|$, the average number of goal hypotheses $|\mathcal{G}|$,

the average number of team-goal mappings analyzed $|\mathcal{M}|$, the average number of correct team-goal mappings $|\mathcal{M}^*|$, the observability degree, represented by the percentage of traces analyzed, the average number of traces analyzed $|\mathcal{O}|$, the average accuracy percentage, the average spread in $|\mathcal{M}|$ ($\mathcal{S}$ in $\mathcal{M}$), and the average recognition time in seconds.

First, we see that the first metrics match the problems we have discussed in Section 7.3. We have 10 problems, each with 4 agents and 4 goal hypotheses. This generates 60 possible team-goal mappings, as we take the power set of the agents, minus the empty set, which is 15, and multiply it by the number of goal hypotheses (4). These 4 agents are always organized in 2 teams, resulting in 2 correct team-goal mappings. Finally, we have problems with full observability, and 4 degrees of partial observability: 10%, 30%, 50%, and 70%.

What we see is that, except for the recognition time (last column to the right), all other metrics remain exactly the same across all enhancement settings. This is exactly what we expected, as the enhancements should not affect the recognition performance, but only the runtime performance. Regarding runtime performance, we see that the enhancements do produce considerable results. If we compare the recognition time between a setup with no enhancements with the incremental solving one, we see that the recognition time dropped from around $56,000$ seconds to around $20,000$ seconds. This means that our recognizer ran around 2.8 times faster with this enhancement only. We see that the runtime performance with the "Incremental Solving" and the "No Enhancements" settings remains approximately the same across all observability degrees.

On the other hand, if we compare the results with no enhancements to the results with the object filtering, we see that the recognition time changes with the observability degree. This is expected, since a higher observability degree, means more team traces, having more team traces generally means having more relevant objects to consider, and having more relevant objects means having a larger formula, which takes longer to find a model to. We see that with 100% observability, the recognition time is around $43,000$ seconds, meaning that the recognizer runs 1.3 times faster. With 10% observability, the enhancement is even more noticeable, where the recognition time drops to around $18,000$ seconds, meaning that it ran around 3.1 times faster.

Now if we add the object filter enhancement, we achieve even better results. As we can see, the object filter reduces the recognition time specially when we have less traces to analyze. This is due to the fact that it only keeps objects that are related to the goal state and the team traces. Logically, if we have less team traces, we have fewer objects to account for, the encoding gets smaller, and the recognition time drops. We see it dropping to around $16,000$ seconds with full observability, meaning the recognizer ran 3.5 times faster if compared to the "No Enhancements" setting. When we deal with fewer team traces, the performance increases even more. With 10% observability, for example,

we see the recognition time drop to around $7,000$ seconds, representing a recognition speed increase of 8 times when compared to the "No Enhancements" setting. Still, runtime performance is a major issue in our recognizer, as even $7,000$ seconds is still too much time for a recognizer to analyze a single problem, mainly if compared to other multiagent plan and goal recognition solutions [2, 3, 63, 56], that can run problems in a few minutes.

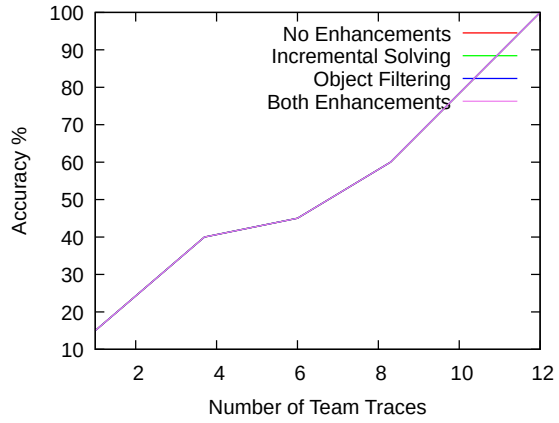
Enhancement	$\|P\|$	$\|\Phi\|$	$\|\mathcal{G}\|$	$\|\mathcal{M}\|$	$\|\mathcal{M}^*\|$	Obs %	$\|\mathcal{O}\|$	Acc %	$\mathcal{S}$ in $\mathcal{M}$	Time (s)
No Enhancements	10	4	4	60	2	10	1.0	15.0%	7.8	55840
						30	3.7	40.0%	4.9	56435
						50	6.0	45.0%	2.7	56239
						70	8.3	60.0%	2.8	56566
						100	12.0	100.0%	2.5	55744
Incremental Solving	10	4	4	60	2	10	1.0	15.0%	7.8	20210
						30	3.7	40.0%	4.9	20273
						50	6.0	45.0%	2.7	20270
						70	8.3	60.0%	2.8	20232
						100	12.0	100.0%	2.5	20296
Object Filtering	10	4	4	60	2	10	1.0	15.0%	7.8	18313
						30	3.7	40.0%	4.9	25028
						50	6.0	45.0%	2.7	39024
						70	8.3	60.0%	2.8	41252
						100	12.0	100.0%	2.5	42884
Both Enhancements	10	4	4	60	2	10	1.0	15.0%	7.8	6936
						30	3.7	40.0%	4.9	9781
						50	6.0	45.0%	2.7	14633
						70	8.3	60.0%	2.8	15369
						100	12.0	100.0%	2.5	16002

Table 7.1: Experimental results comparing our different settings with performance enhancements, with no enhancements, with the object filtering only, with incremental solving only, and with both enhancements

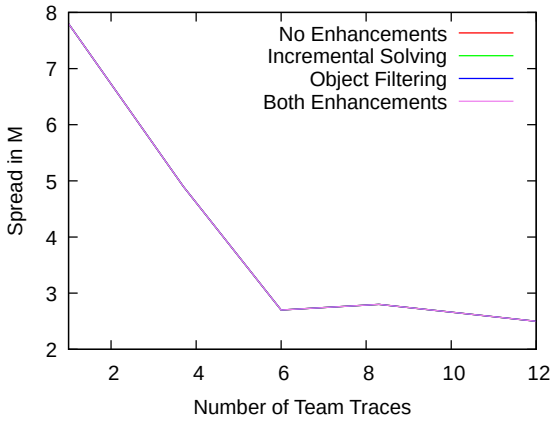
We illustrate the results from Table 7.1 in the three graphs from Figure 7.1. In each graph, there is a curve for each enhancement setting we evaluate. Figure 7.1a shows how the accuracy percentage changes as we increase the number of traces. Figure 7.1b displays the same for spread in $\mathcal{M}$, while Figure 7.1c does it for the recognition time. The graphs show us what we have already seen in the table, with the accuracy and spread being exactly the same for all enhancement settings, while the time is what the enhancements really affect. We can really see on the rightmost graph how the "Object Filtering" enhancement makes the recognition time change with the observability degree, while settings without the filtering maintain the same recognition time across all observability degrees.

Now that we have seen that our runtime performance enhancements actually increased the runtime performance of our recognizer without affecting its recognition performance, we can evaluate our approach's recognition performance alone, with different threshold values. Table 7.2 displays the same metrics as Table 7.1, but now we analyze different threshold values in the rows, and we only use the "Both Enhancements" performance setting.

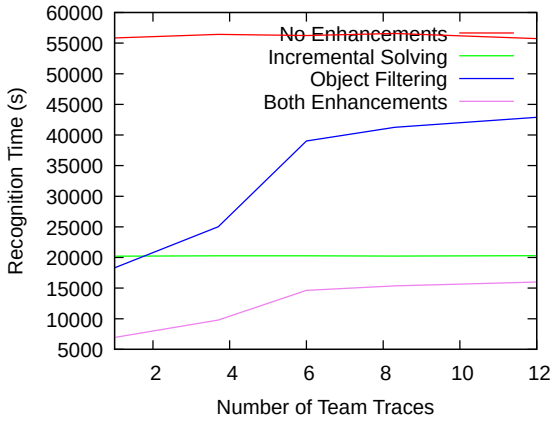
We can see that when have a 0% threshold and a 100% observability degree, we have an almost perfect scoring recognizer, with 100% average accuracy, meaning that the recognizer included all correct mappings in the accepted mappings for all problems,



(a) Accuracy % among different performance enhancement settings



(b) Spread in $\mathcal{M}$ among different performance enhancement settings



(c) Recognition time among different performance enhancement settings

Figure 7.1: Accuracy percentage, Spread in $\mathcal{M}$, and recognition time for different performance enhancement settings

with a spread of 2.5. This means that the cardinality of the set of accepted mappings was, in average, 2.5, which is good considering we have 2 correct mappings, even though a perfect score would have been a spread of 2. The reason behind this slightly higher spread is in some problems where, due to the initial state partially satisfying the goal state, one or more of the agents in the team did not need to perform any actions. This results in our recognizer not being able to disambiguate between two mappings with or without those agents. For that reason, it ends up scoring them both equally, and accepting both of them, increasing the spread. This is a known limitation of our approach that is intrinsic to its design and to the way we analyze the team traces.

Another known limitation of our approach lies on satisfied incorrect mappings. It might be the case in some problems that the initial state partially, or even fully satisfies one or more of the fake goal hypotheses. In this case, the found plan for the mappings with those goals would have no actions, as the goal state is already satisfied, and the cost for that mapping is reduced to the amount of team traces. In problems with lower

observability, this becomes a huge problem, as in a setting with high observability our recognizer tends to assign a lower the score to the incorrect mappings, as it identifies more unobserved actions in the plan. This results in the recognizer accepting incorrect mappings and rejecting the correct ones, and being less precise in its response.

These limitations become more apparent when we deal with lower observability, as we see from the results. We can see that the lower the observability degree, the lower the accuracy and, in general, the higher the spread we obtain. We are able to increase the accuracy by using thresholds, but that comes with a cost. Since the thresholds simply relax the recognizer into accepting worse-scoring mappings, we obtain better accuracy by making it more indulgent and accepting more mappings, but that results in a higher spread in $\mathcal{M}$ as well, as we can see from Table 7.2. Even when we deal with problems where those limitations do not appear, having a lower observability degree will affect the recognition process hugely, as our approach depends on the team traces to properly score the team-goal mappings.

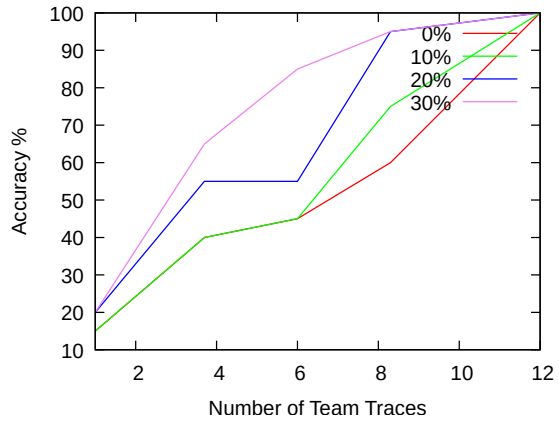
It is difficult to compare our results to existing multiagent goal and plan recognition solutions as we have not used the same dataset or even problem generation process as other works. Still, even with 0% threshold, we were able to obtain accuracy percentages that roughly match the observability degrees, while keeping our spread low if compared to the number of total mappings analyzed. As mentioned before, we were able to increase the accuracy through the use of thresholds, but at the cost of increasing the spread, thus decreasing the recognizer's general precision.

Θ	$\lvert P\rvert$	$\lvert\Phi\rvert$	$\lvert\mathcal{G}\rvert$	$\lvert\mathcal{M}\rvert$	$\lvert\mathcal{M}^*\rvert$	**Obs %**	$\lvert\mathcal{O}\rvert$	**Acc %**	$\mathcal{S}$ in $\mathcal{M}$	**Time (s)**
0%	10	4	4	60	2	10	1.0	15.0%	7.8	6936
						30	3.7	40.0%	4.9	9781
						50	6.0	45.0%	2.7	14633
						70	8.3	60.0%	2.8	15369
						100	12.0	100.0%	2.5	16002
10%	10	4	4	60	2	10	1.0	15.0%	7.8	6780
						30	3.7	40.0%	5.7	9540
						50	6.0	45.0%	3.5	14474
						70	8.3	75.0%	4.5	15264
						100	12.0	100.0%	2.5	15977
20%	10	4	4	60	2	10	1.0	20.0%	17.7	6803
						30	3.7	55.0%	10.9	9569
						50	6.0	55.0%	5.3	14414
						70	8.3	95.0%	8.7	15230
						100	12.0	100.0%	6.4	15911
30%	10	4	4	60	2	10	1.0	20.0%	19.3	6869
						30	3.7	65.0%	17.1	9572
						50	6.0	85.0%	12.8	14490
						70	8.3	95.0%	10.3	15228
						100	12.0	100.0%	10.1	15956

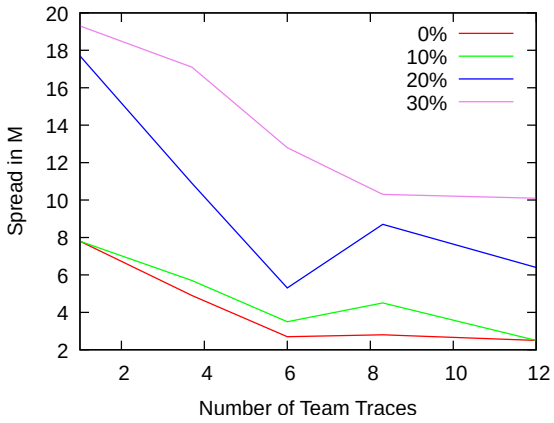
Table 7.2: Experimental results comparing different threshold values, with no threshold, 10% threshold, 20% threshold, and 30% threshold

Again, as a better way of visualizing the results from Table 7.2, we have the three graphs in Figure 7.2. They display the same information as the graphs in Figure 7.1, but regarding the different threshold values instead of the different enhancement settings. We see in the graphs that the threshold does not affect the recognition time, but is does affect accuracy and spread. We also see the accuracy increasing as the observability degree

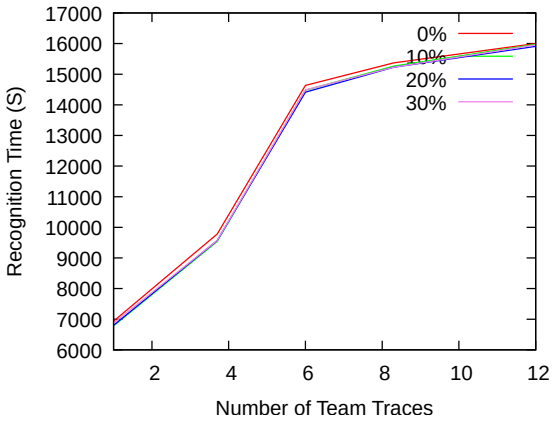
increases, while the spread drops. Regarding the spread, we see an anomaly where the spread for 8 traces, the 50% observability mark seems to drop before increasing again for the 70% observability mark. For a 20% threshold, we even see the minimum spread being achieved at the 50% observability mark. This is caused by an anomaly in some problems where, depending on which traces are among the 50% we can observe, some mappings end up having a cost of zero due to none of their agents performing actions in the traces. Add that to the problems where a fake hypothesis is satisfied by the initial state, and we have those specific mappings with a cost of zero while the closest one has a higher enough cost to not be included in the set of accepted mappings, hence reducing the spread, even though the accepted mapping is not the correct one.



(a) Accuracy % among different threshold values



(b) Spread in $\mathcal{M}$ among different threshold values



(c) Recognition time among different threshold values

Figure 7.2: Accuracy percentage, Spread in $\mathcal{M}$, and recognition time for different threshold values

Moreover, there were some problem settings we did not experiment with, such as problems where teams compete for resources. In this kind of problems, two separate teams could desire to add the same block to their different stacks. We did not experiment with that kind of problem due to the way we generate the team traces. Since we generate

the traces individually for each team, completely isolated from one another, we would experience difficulties when merging the team plans to achieve the observations, as the team plans would conflict with each other.

8. RELATED WORK

In this chapter, we discuss work on multiagent goal and plan recognition that is strongly related to our own work. While we do not compare these approaches directly, given the different assumptions and problem formalisations, we compare our work to these in qualitative terms. We divide this section into the different kind of approaches. In Section 8.1, we discuss a classic "plan recognition as planning" approach inspired by Ramirez and Geffner's work [50]. In Section 8.2, we discuss probabilistic approaches for multiagent goal and plan recognition. Finally, in Section 8.3, we discuss SAT-based approaches for goal and plan recognition, as it is the focus of this work.

8.1 A Classical "Plan Recognition as Planning" Approach

In a 2016 work by Argenta and Doyle [2], the authors develop a recognizer called Multiagent Plan Recognition as Planning (MAPRAP) inspired by the one developed by Ramirez and Geffner [50], using a planner to compute the cost to achieve each goal and then comparing that cost to the cost of a plan that incorporates the observations. If the plan cost increases with the observations, then that plan does not explain the observations, and hence that is not the correct one. They describe two approaches: $MAPRAP^A$ and $MAPRAP^B$. $MAPRAP^A$ maps all possible teams to all possible goals and tests the cost for each one of them, removing the mapping when the observations increase the cost. $MAPRAP^B$, on the other hand, starts with a single team, containing all agents, for each goal, and prunes the agents from the composition when their actions increase the cost. This makes so that $MAPRAP^B$ reduces the number of runs per goal as the agent/team ratio increases.

8.2 Probabilistic Approaches

Following the work above, the same authors extend the MAPRAP recognizer in a probabilistic way to not only return the goals and plans identified as the possibly correct ones, but to rank them using a likelihood score [3]. They call this new recognizer Probabilistic Multiagent Plan Recognition as Planning (P-MAPRAP). They do this by computing a likelihood score for each possible interpretation (an interpretation maps every agent to a team and every team to a goal) and storing them in a priority queue based on said score. Before reading the first observable, they compute a baseline score for each interpretation in the list without considering observations. They iterate over the list of observables computing the cost with the observations, just like MAPRAP. They compute the

likelihood score using the cost difference, and store the interpretations back in the priority queue, repositioning inside the queue according to the score. If the new top interpretation does not include the current observations, they remove it from the queue and repeat the process until it does. This guarantees that the recognizer only considers the most likely interpretations.

In a different work [56], the authors propose three probabilistic approaches to multiagent plan recognition problems. In this work, they transform the goal recognition problems into temporal planning problems, *i.e.* problems where each action has a defined temporal duration in the environment. The first approach is based on Ramirez and Geffner's probabilistic plan recognition work [51], using the cost difference between a plan with and without the observations as a proxy to compute the probability of the observations given each goal, and then using this probability to compute the probability of each goal hypothesis being the correct one given the observations.

The second approach differs from the first by not using this cost difference to compute the probabilities, and by using a diverse planner, as in [57]. In a diverse planning problem, the objective is to obtain a set of m plans that are, at least, at a distance of d away from each other [57]. In this approach, the authors compute the probability of each plan generated by the diverse planner being the followed one given the observations, and use this probability as a proxy to compute the probability of each goal hypothesis being the correct intended one given the observations. One interesting feature to note is that during the probability computation, they add penalties for unexplained and missing observations in the plans. The third and final approach, the one that achieved the best results, is a hybrid one between the first two, where they use a diverse planner, but merge the generated plans.

8.3 SAT-Based Approaches

The work by Zhuo *et al.* [62, 63, 64] is one of the greatest references in the field. The reason it directly relates to our work is the fact that all of these approaches work by building a set of constraints from the planning problem and solving them using a weighted MAX-SAT solver. The main difference between them is that [62] uses team plan libraries as inputs, and [64] works with incomplete team traces and action-models, which would be analogous to working with incomplete domain models [47]. Since we are focusing on goal recognition over a domain theory instead of plan libraries, and we are not focusing on goal recognition over incomplete domain models, we focus on discussing [63].

In [63] the authors develop a recognizer that, given a partially-observed team trace, a set of action models (defined as STRIPS actions), an initial state, and a set of goal hypotheses, outputs a set of team plans with the maximum likelihood to achieve some

goal among the hypotheses. They use what they call a likelihood function to compute the likelihood of a set of team plans achieving a goal. The set of team plans must respect a series of properties, which they ensure by building sets of constraints.

First, they build a set of candidate activities by instantiating the actions using the observations and the initial and goal states. Then, they build a set of hard constraints to ensure certain conditions: that the set of team plans is a partition of the traces, covers all the observed activities, and transforms the initial state into a goal state. With the hard constraints generated, they generate a set of soft constraints based on the likelihood function. Finally, they group all the constraints and solve them using a MAX-SAT solver. They do that for each goal hypothesis, and the recognizer considers the one with the highest result from the MAX-SAT solver to be the correct intended one. The recognizer then converts the found solution for the MAX-SAT problem into a plan and returns it as well. The results are promising, but the authors limit the evaluation to simpler problem domains.

9. CONCLUSION

In this dissertation, we have provided extensive theoretical background over planning and goal recognition in both single-agent and multiagent settings. We have also provided extensive background on SAT-based planning and the existing techniques for it. We have presented a multiagent goal recognition approach that relies on satisfiability analysis to correctly infer the team-goal mappings, and conducted experiments to evaluate its performance.

Our approach is able to achieve satisfactory results regarding recognition performance considering the different observability degrees, even when considering a 0% threshold for accepting mappings based on their scores. On top of that, we have developed two runtime performance enhancements that largely improve the recognition time. Still, runtime performance remains a major issue for our approach. Other limitations to our approach lie on harder problems where a one or more agents in a team do not perform any actions, and problems where the initial state satisfies (partially or fully) one or more fake goal hypotheses. In addition to that, we have not experimented with problems where the teams must compete for resources, due to our problem generation process.

For future work, we would like to better experiment with our approach, as time constraints limited our experimentation capabilities. We would like to generate more diverse problems, with varying number of teams and agents, besides generating problems from domains other than *Blocks World*. We would also like to experiment with other runtime performance enhancements we did not have a chance to experiment with, such as parallelizing Algorithm 6.3 to analyze multiple mappings at a time. Another future work would be to add more constraints to our formula, where those constraints could account for other domain properties, such as landmarks. Finally, we would like to experiment our mapping analysis process with non-SAT approaches already used in single agent goal recognition, such as landmark-based techniques [48], and evaluate how well a state-of-the-art single-agent goal recognition technique performs in a multiagent setting.

REFERENCES

[1] Amado, L.; Mirsky, R.; Meneguzzi, F. “Goal Recognition as Reinforcement Learning”. In: Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI-22), 2022, pp. 9644–9651.

[2] Argenta, C.; Doyle, J. “Multi-Agent Plan Recognition as Planning (MAPRAP)”. In: Proceedings of the 8th International Conference on Agents and Artificial Intelligence (ICAART 2016), 2016, pp. 141–148.

[3] Argenta, C.; Doyle, J. “Probabilistic Multi-Agent Plan Recognition as Planning (P-MAPRAP): Recognizing Teams, Goals, and Plans from Action Sequences”. In: Proceedings of the 9th International Conference on Agents and Artificial Intelligence (ICAART 2017), 2017, pp. 575–582.

[4] Bacchus, F. “AIPS 2000 Planning Competition: The Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems”, *AI Magazine*, vol. 22, 2001, pp. 47–56.

[5] Barrett, C. W.; Stump, A.; Tinelli, C. “The SMT-LIB Standard Version 2.0”. In: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (SMT ’10), 2010.

[6] Bjørner, N. S.; Phan, A.-D.; Fleckenstein, L. “νZ - An Optimizing SMT Solver”. In: Proceedings of the 21st International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2015), 2015, pp. 194–199.

[7] Blum, A. L.; Furst, M. L. “Fast Planning Through Planning Graph Analysis”. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI’95), 1995, pp. 1636–1642.

[8] Bonet, B.; Geffner, H. “HSP: Planning as Heuristic Search”, *Artificial Intelligence*, vol. 129–1, June 2001, pp. 5–33.

[9] Borrajo, D. “Multi-Agent Planning by Plan Reuse”. In: Proceedings of the 2013 International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS’13), 2013, pp. 1141–1142.

[10] Brafman, R.; Domshlak, C. “From One to Many: Planning for Loosely Coupled Multi-Agent Systems”. In: Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS 2008), 2008, pp. 28–35.

[11] Cook, S. A. “The Complexity of Theorem-Proving Procedures”. In: Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC’71), 1971, pp. 151–158.

[12] Davis, M.; Logemann, G.; Loveland, D. “A Machine Program for Theorem-Proving”, *Communications of the ACM*, vol. 5-7, July 1962, pp. 394–397.

[13] Davis, M.; Putnam, H. “A Computing Procedure for Quantification Theory”, *Journal of the ACM*, vol. 7–3, July 1960, pp. 201–215.

[14] de Moura, L.; Bjørner, N. “Z3: An Efficient SMT Solver”. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008), 2008, pp. 337–340.

[15] Dimopoulos, Y.; Hashmi, M. A.; Moraitis, P. “μ-SATPLAN: Multi-Agent Planning as Satisfiability”, *Knowledge-Based Systems*, vol. 29, May 2012, pp. 54–62.

[16] Dimopoulos, Y.; Nebel, B.; Koehler, J. “Encoding Planning Problems in Nonmonotonic Logic Programs”. In: Proceedings of the 4th European Conference on Planning (ECP’97), 1997, pp. 169–181.

[17] Erol, K.; Hendler, J.; Nau, D. S. “Semantics for Hierarchical Task-Network Planning”, Technical Report, Department of Computer Science, University of Maryland, USA, 1994.

[18] Fernández, S.; Borrajo, D. “MAPR and CMAP”. In: Proceedings of the Competition of Distributed and Multi-Agent Planners (CoDMAP), on the Workshop on Distributed and Multiagent Planning (DMAP), on the 25th International Conference on Automated Planning and Scheduling (ICAPS 2015), 2015, pp. 1–3.

[19] Fikes, R. E.; Nilsson, N. J. “STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving”, *Artificial Intelligence*, vol. 2–3, 1971, pp. 189–208.

[20] Geib, C. W. “Problems with Intent Recognition for Elder Care”. In: Proceedings of the 18th AAAI Conference on Artificial Intelligence (AAAI-02), 2002, pp. 13–17.

[21] Geib, C. W.; Goldman, R. P. “Plan Recognition in Intrusion Detection Systems”. In: Proceedings of the DARPA Information Survivability Conference and Exposition II (DISCEX’01), 2001, pp. 46–55.

[22] Ghallab, M.; Nau, D. S.; Traverso, P. “Automated Planning - Theory and Practice.” Elsevier, 2004.

[23] Goldman, R. P.; Geib, C. W.; Miller, C. A. “A New Model of Plan Recognition”. In: Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI’99), 1999, pp. 245–254.

[24] Helmert, M. “The Fast Downward Planning System”, *Journal of Artificial Intelligence Research (JAIR)*, vol. 26–1, July 2006, pp. 191–246.

[25] Hoffmann, J. “FF: The Fast-Forward Planning System”, *AI Magazine*, vol. 22–3, September 2001, pp. 57–62.

[26] Karpas, E.; Magazzeni, D. “Automated Planning for Robotics”, *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 3–1, 2020, pp. 417–439.

[27] Kautz, H. “Deconstructing Planning as Satisfiability”. In: Proceedings of the 21st AAAI Conference on Artificial Intelligence (AAAI-06), 2006, pp. 1524–1526.

[28] Kautz, H.; Selman, B. “Planning as Satisfiability”. In: Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI’92), 1992, pp. 359–363.

[29] Kautz, H.; Selman, B. “BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving”. In: Proceedings of the Workshop on Planning as Combinatorial Search on the 4th International Conference on Artificial Intelligence in Planning Systems (AIPS’98), 1998.

[30] Kautz, H. A.; Allen, J. F. “Generalized Plan Recognition”. In: Proceedings of the 5th AAAI Conference on Artificial Intelligence (AAAI-86), 1986, pp. 32–37.

[31] Kautz, H. A.; Selman, B. “Pushing the Envelope: Planning, Propositional Logic and Stochastic Search”. In: Proceedings of the 13th AAAI Conference on Artificial Intelligence (AAAI-96), 1996, pp. 1194–1201.

[32] Kelly, J.-P.; Botea, A.; Koenig, S. “Offline Planning with Hierarchical Task Networks in Video Games”. In: Proceedings of the 17th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-21), 2021, pp. 60–65.

[33] Komenda, A.; Štolba, M.; Kovács, D. “The International Competition of Distributed and Multiagent Planners (CoDMAP)”, *AI Magazine*, vol. 37–3, Fall 2016, pp. 109–115.

[34] Kovacs, D. L. “Complete BNF Definition of MA-PDDL with Privacy”. Accessed: September 09, 2022, Source: http://agents.fel.cvut.cz/codmap/MA-PDDL-BNF-20150221.pdf, 2015.

[35] Kvarnström, J. “Planning for Loosely Coupled Agents Using Partial Order Forward-Chaining”. In: Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS 2011), 2011, pp. 138–145.

[36] Levin, L. A. “Universal Sequential Search Problems”, *Problems of Information Transmission*, vol. 9–3, 1973, pp. 265–266.

[37] Long, D.; Kautz, H.; Selman, B.; Bonet, B.; Geffner, H.; Koehler, J.; Brenner, M.; Hoffmann, J.; Rittinger, F.; Anderson, C. R.; Weld, D. S.; Smith, D. E.; Fox, M.; Long, D. “The AIPS-98 Planning Competition”, *AI Magazine*, vol. 21–2, June 2000, pp. 13–34.

[38] Luis, N.; Fernández, S.; Borrajo, D. “Plan Merging by Reuse for Multi-Agent Planning”. In: Proceedings of the 2nd ICAPS Workshop on Distributed and Multi-Agent Planning, 2014, pp. 38–44.

[39] Marques Silva, J.; Sakallah, K. “GRASP: A New Search Algorithm for Satisfiability”. In: Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design (ICCAD’96), 1996, pp. 220–227.

[40] Marques-Silva, J.; Sakallah, K. “GRASP: A Search Algorithm for Propositional Satisfiability”, *IEEE Transactions on Computers*, vol. 48–5, February 1999, pp. 506–521.

[41] McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; Wilkins, D. “PDDL — The Planning Domain Definition Language”. In: Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems (AIPS’98), 1998.

[42] Microsoft. “Online Z3 Guide”. Accessed: May 12, 2023, Source: https://microsoft.github.io/z3guide/.

[43] Mirsky, R.; Gal, Y. K.; Shieber, S. M. “CRADLE: An Online Plan Recognition Algorithm for Exploratory Domains”, *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8–3, May 2017, pp. 1–22.

[44] Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; Malik, S. “Chaff: Engineering an Efficient SAT Solver”. In: Proceedings of the 38th Annual Design Automation Conference (DAC’01), 2001, pp. 530–535.

[45] Oh, J.; Meneguzzi, F.; Sycara, K. “Probabilistic Plan Recognition for Proactive Assistant Agents”. In: *Plan, Activity, and Intent Recognition: Theory and Practice*, Sukthankar, G.; Goldman, R. P.; Geib, C.; Pynadath, D. V.; Bui, H. H. (Editors), Elsevier, 2014, pp. 275–288.

[46] Oh, J.; Meneguzzi, F.; Sycara, K.; Norman, T. J. “Prognostic Normative Reasoning”, *Engineering Applications of Artificial Intelligence*, vol. 26–2, February 2013, pp. 863 – 872.

[47] Pereira, R.; Meneguzzi, F. “Goal Recognition in Incomplete Domain Models”. In: Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI-18), 2018, pp. 8127–8128.

[48] Pereira, R. F.; Oren, N.; Meneguzzi, F. “Landmark-Based Heuristics for Goal Recognition”. In: Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI-17), 2017, pp. 3622–3628.

[49] R de A Santos, L.; Meneguzzi, F.; Pereira, R. F.; Pereira, A. “An LP-Based Approach for Goal Recognition as Planning”. In: Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI-21), 2021, pp. 11939–11946.

[50] Ramírez, M.; Geffner, H. “Plan Recognition as Planning”. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09), 2009, pp. 1778–1783.

[51] Ramírez, M.; Geffner, H. “Probabilistic Plan Recognition Using Off-the-Shelf Classical Planners”. In: Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-10), 2010, pp. 1121–1126.

[52] Richter, S.; Westphal, M. “The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks”, *Journal of Artificial Intelligence Research*, vol. 39–1, September 2010, pp. 127–177.

[53] Rintanen, J. “Heuristics for Planning with SAT”. In: The 16th Annual International Conference on the Principles and Practice of Constraint Programming (CP 2010), Cohen, D. (Editor), 2010, pp. 414–428.

[54] Rintanen, J. “Madagascar: Scalable Planning with SAT”. In: The 2014 International Planning Competition (IPC), 2014, pp. 66–70.

[55] Rintanen, J.; Heljanko, K.; Niemelä, I. “Planning as Satisfiability: Parallel Plans and Algorithms for Plan Search”, *Artificial Intelligence*, vol. 170–12, September 2006, pp. 1031–1080.

[56] Shvo, M.; Sohrabi, S.; McIlraith, S. A. “An AI Planning-Based Approach to the Multi-Agent Plan Recognition Problem (Preliminary Report)”. In: Proceedings of the AAAI 2017 Workshop on Plan, Activity, and Intent Recognition (PAIR), 2017.

[57] Sohrabi, S.; Riabov, A. V.; Udrea, O. “Plan Recognition as Planning Revisited”. In: Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI-2016), 2016, pp. 3258–3264.

[58] Sukthankar, G.; Goldman, R. P.; Geib, C.; Pynadath, D. V.; Bui, H. H. “Plan, Activity, and Intent Recognition: Theory and Practice”. Elsevier, 2014.

[59] Torreño, A.; Onaindia, E.; Komenda, A.; Štolba, M. “Cooperative Multi-Agent Planning: A Survey”, *ACM Computing Surveys*, vol. 50–6, November 2017, pp. 1–32.

[60] Torreño, A.; Onaindia, E.; Sapena, O. “An Approach to Multi-Agent Planning with Incomplete Information”. In: Proceedings of the The 20th European Conference on Artificial Intelligence (ECAI-12), 2012, pp. 762–767.

[61] Wehrle, M.; Rintanen, J. “Planning as Satisfiability with Relaxed ∃-Step Plans.” In: Proceedings of the 20th Australian Joint Conference on Artificial Intelligence (AI 2007), 2007, pp. 244–253.

[62] Zhuo, H.; Li, L. “Multi-Agent Plan Recognition with Partial Team Traces and Plan Libraries”. In: Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-2011), 2011, pp. 484–489.

[63] Zhuo, H.; Yang, Q.; Kambhampati, S. “Action-Model Based Multi-agent Plan Recognition”. In: Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS’12), Pereira, F.; Burges, C.; Bottou, L.; Weinberger, K. (Editors), 2012, pp. 368–376.

[64] Zhuo, H. H. “Recognizing Multi-Agent Plans When Action Models and Team Plans Are Both Incomplete”, *ACM Transactions on Intelligent Systems and Technology*, vol. 10–3, May 2019, pp. 1–24.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br