Automatic Hierarchical Nesting of Partially Observable Markov Decision Processes for Task Planning in Service Robotics

por

Sergio Arredondo Serrano

Tesis sometida como requerimiento parcial para obtener el grado de

Maestro en Ciencias en el área de Ciencias Computacionales

por el

Instituto Nacional de Astrofísica, Óptica y Electrónica
Noviembre, 2019
Tonantzintla, Puebla

Supervisada por:

Dr. Luis Enrique Sucar Succar

Agradecimientos

En primer lugar, me gustaría agradecer al CONACyT por otorgarme la beca no. **634966** sin la cual jamás hubiera podido llevar a cabo mis estudios de maestría. A mis profesores de clase, por proveerme de herramientas que resultaron indispensables en el desarrollo de mi trabajo de tesis. A mis cuatro revisores, les agradezco su profesionalismo al criticar a mi trabajo, contribuyendo a llevar el producto final a un mejor término. A mi asesor, el Dr. Luis Enrique Sucar, por la dedicación y paciencia con la que me guió durante el desarrollo de la investigación, así como por dar un ejemplo de ética profesional y calidad humana.

A mi familia, por creer en este proyecto y apoyarme incondicionalmente desde el principio. A mi *amá* y *apá*, por sus palabras de aliento que en muchas ocasiones fueron la gasolina que me mantuvo andando. A mi hermano, por celebrar mis éxitos, estar conmigo en los tiempos difíciles, seguirme la curas y nunca dejarme abajo.

Por compartir camino conmigo y hacer broma de la adversidad, agradezco a Ricardo, Josué, Aco, Oswualdo, Madrid, Bruno, Jessica, Kenpa, Diana, Jonathan, Jeymy, Juan, Loreth, Freddy, Anibal, Ángel, Estefanía y Carlos, después de todo, *¿somos una banda que no?, hay que apoyarnos...*

Abstract

A wide variety of approaches have been proposed to address the problem of task planning in robotics, from which partially observable Markov decision processes (POMDP) stand out due to their capacity to model the uncertainty of actions and keep track of the state of the world by means of a partially observable representation of it. Nonetheless, there are some drawbacks inherent to the use of POMDPs, such as designing a representation that models as best as possible a particular problem, along with the complexity that represents to find a good policy for POMDPs with large state spaces. Therefore, in order to mitigate these challenges, in this thesis we propose an architecture for task planning oriented towards service robot applications, that combines a knowledge representation scheme and POMDPs to build a hierarchy of actions that enables the decomposition of problems into several smaller ones. The knowledge representation defines a list of parameters, so that domain specific information can be encoded by a designer, and used by the architecture to automatically generate and execute plans to solve tasks. Using the hierarchy of actions to generate plans, the system is able to exploit the structure of the environment and ignore those regions in the state space that are irrelevant for a specific task. To evaluate the proposed architecture, a mobile robot navigation domain is employed as case study. Experimental results show that, in scenarios with moderate uncertainty, the architecture is able to perform both reliably and time efficiently, as it generates plans in a time that is several orders smaller than baseline methods.

Keywords: Task Planning, Hierarchical POMDPs, Service Robotics, Declarative Programming, General Architecture.

Resumen

Una amplia variedad de enfoques han sido propuestos para abordar el problema de planificación de tareas en robótica, entre los cuales destacan los procesos de decisión de Markov parcialmente observables (POMDP por sus siglas en inglés) debido a su capacidad para modelar la incertidumbre en las acciones y realizar un seguimiento del estado del mundo mediante una representación parcialmente observable del mismo, lo cual es particularmente importante en robótica. No obstante, existen algunas desventajas inherentes al uso de los POMDPs, tales como el diseño de una representación que modele lo mejor posible un problema en particular, así como la complejidad que representa encontrar una buena política para POMDPs con espacios de estado grandes. Así, con el objetivo de mitigar estas dificultades, en esta tesis presentamos una arquitectura para la planificación de tareas orientada a aplicaciones de robótica de servicio, que combina un esquema de representación de conocimiento y POMDPs para construir una jerarquía de acciones que permite la descomposición de problemas en varios más pequeños. La representación del conocimiento define una lista de parámetros que permite que un diseñador codifique información específica del dominio, y su utilización por parte de la arquitectura para generar y ejecutar, de manera automática, planes con el objetivo de resolver tareas. Utilizar la jerarquía de acciones para planificar permite que el sistema aproveche la estructura del entorno e ignore regiones del espacio de estados que son irrelevantes para una tarea en específico. Para evaluar la arquitectura propuesta, un dominio de navegación de un robot móvil es empleado como caso de estudio. Resultados experimentales muestran que, en escenarios de incertidumbre moderada, la arquitectura es capaz de desempeñarse de manera confiable y eficiente, dado que genera planes en un tiempo que es menor en varios órdenes de magnitud al requerido por otros métodos base.

Palabras clave: Planificación de Tareas, POMDP Jerárquico, Robótica de Servicio, Programación Declarativa, Arquitectura General.

Contents

Agradecimientos **iii**

Abstract **v**

Resumen **vii**

1 Introduction **1**

1.1 Motivation 1

1.2 Problem description 4

1.3 Research Questions 4

1.4 Hypothesis 4

1.5 Objectives 5

1.6 Scope and limitations 5

1.7 Description of the proposed method 6

1.8 Contributions 9

1.9 Document organization 10

2 Theoretical framework **11**

2.1 ASP: Answer Set Programming 11

2.1.1 SPARC 14
2.1.1.1 Directives and sort definitions 15
2.1.1.2 Predicate Declarations 17
2.1.1.3 Program rules 17
2.1.1.4 Answer sets 18
2.1.2 Action Language for transition diagrams 19
2.1.3 Action Language in SPARC 21
2.2 Markov Decision Processes 25
2.2.1 Policies 27
2.2.2 Partially Observable Markov Decision Processes 28
2.3 Hierarchical Reinforcement Learning 30
2.3.1 Abstract actions 31
2.3.2 State abstraction 32
2.3.3 Optimality 33
2.4 Chapter Summary 34

3 Related work **35**
3.1 Architectures applied towards service robotics 35
3.2 Hierarchical approaches for solving MDPs and POMDPs 44
3.3 Discussion 48
3.4 Chapter Summary 51

4 Proposed method **53**
4.1 General overview 53
4.2 Knowledge base construction 56

4.2.1 General knowledge 56
4.2.1.1 Basic modules 56
4.2.1.2 Domain dynamics 59
4.2.1.3 Hierarchical function 65
4.2.2 Specific knowledge 66
4.3 Architecture initialization 67
4.3.1 Construction of bottom POMDP 67
4.3.2 Hierarchy of actions 70
4.3.2.1 State space tree 70
4.3.2.2 Concrete and abstract components 72
4.3.2.3 Modeling abstract actions 73
4.4 Architecture operation 80
4.4.1 Planning 80
4.4.1.1 Relevant sub-space 80
4.4.1.2 Hierarchical policy 81
4.4.2 Plan execution 85
4.4.2.1 Hierarchical policy execution 85
4.4.2.2 Local policy execution 87
4.5 Chapter Summary 91

5 Experiments and results **95**

5.1 Navigation domain as study case 95
5.2 Experiment parameters 97
5.2.1 Baseline methods and failure criteria 97

5.2.2 Control parameters 100
5.2.3 Independent variables 102
5.2.4 Dependent variables 103
5.2.5 Statistical parameters 104
5.3 Experiment 1 106
5.3.1 Objective 106
5.3.2 Hypothesis 106
5.3.3 Results 106
5.4 Experiment 2 111
5.4.1 Objective 111
5.4.2 Hypothesis 111
5.4.3 Results 111
5.5 Experiment 3 116
5.5.1 Objective 116
5.5.2 Hypothesis 116
5.5.3 Results 116
5.6 Experiment 4 121
5.6.1 Objective 121
5.6.2 Hypothesis 121
5.6.3 Results 121
5.7 Experiment 5 125
5.7.1 Objective 125
5.7.2 Description 125
5.7.3 Results 126

5.8 Discussion 130
5.9 Chapter Summary 135

6 Conclusions and future work **137**

6.1 Conclusions 137
6.2 Contributions 138
6.3 Future Work 139

List of Figures

1.1 Task planning in a household domain 2

1.2 Knowledge base structure 9

2.1 MDP model for the interaction between an agent and the world . . . 26

2.2 POMDP model for the interaction between an agent and the world . 29

2.3 Example of a task hierarchy 32

2.4 Ranking of different types of optimality for policies in hierarchical systems 34

3.1 Multi-layer representation from [Hanheide et al., 2011] 37

3.2 Task hierarchy from [Pineau and Thrun, 2002] 46

3.3 Hierarchical POMDP from [Theocharous et al., 2001] 48

4.1 Example of a navigation domain scenario 54

4.2 Methodology followed by the proposed architecture to solve task planning problems 55

4.3 Multi-resolution representation of a navigation domain environment . 71

4.4 Hierarchical representation of a navigation domain environment . . . 72

4.5 Example of a hierarchical policy 82

5.1 Example of an environment of two buildings, generated for a configuration in experiment 1 102

5.2 Success ratio scores in experiment 1 108

5.3 Average relative error scores in experiment 1 108

5.4 Average path relative cost scores in experiment 1 109

5.5 Average planning time scores in experiment 1 109

5.6 Confidence intervals for differences found in experiment 1 110

5.7 Success ratio scores in experiment 2 112

5.8 Average relative error scores in experiment 2 113

5.9 Average path relative cost scores in experiment 2 113

5.10 Average planning time scores in experiment 2 114

5.11 Confidence intervals for differences found in experiment 2 115

5.12 Success ratio scores in experiment 3 118

5.13 Average relative error scores in experiment 3 118

5.14 Average path relative cost scores in experiment 3 119

5.15 Average planning time scores in experiment 3 119

5.16 Confidence intervals for differences found in experiments 3 and 4 . . . 120

5.17 Success ratio scores in experiment 4 123

5.18 Average relative error scores in experiment 4 123

5.19 Average path relative cost scores in experiment 4 124

5.20 Average planning time scores in experiment 4 124

5.21 Map generated using 2D SLAM 128

5.22 Arrangement of markers employed to track the robot's position . . . 128

5.23 Tracking of the robot's position within the robotics laboratory 129

5.24 Gaussian probability density function plots 129

5.25 Failure ratio of the proposed architecture in experiment 3 133

List of Tables

3.1 Comparison of task planning architectures 44

5.1 Comparison of results obtained by HP in an environment made of 11 buildings 134

Chapter 1

Introduction

1.1 Motivation

Recently, service robots have shown that they can be very helpful with a variety of tasks, usually found in domestic environments such as offices and households, that involve cleaning chores and taking care of people. By having a service robot in charge of assisting elderly people, for instance, it would mean a solution to a 24 hours a day caring problem, one that does not need to rest, gets distracted nor bored while working. Whereas for domestic chores, a service robot would take care of repetitive labors, such as doing the dishes, cleaning surfaces or receiving packages. This would free people from these tasks so they could invest all of their time in creative work and recreational activities. In general, society can benefit from service robotics systems by putting them in charge of tasks that either require full attention from the worker for long continuous periods of time, or consist of repetitive and non-creative duties, pretty similar to what robotic arms have done for the manufacturing industry.

Since service robots are developed towards domestic domains with a user-oriented approach, there is a series of challenges that ought to be overcome before a robot operates at a degree of autonomy that enables it to handle any scenario that might arise during work. Among the set of skills a service robot is expected to have, goal reasoning, human-robot interaction and acting are the ones that standout in the context of interacting and assisting to people. Goal reasoning is a monitoring function at the highest level of mission management [Ingrand and Ghallab, 2017],

and its main objective is to keep track of the feasibility and relevance of the robot's goals, as well as to create new goal if required for the task at hand. Human-robot interaction studies algorithms, techniques, models, and frameworks necessary to build robotic systems that engage in social interactions with humans [Thomaz et al., 2016]. With regards to acting, it refers to the refinement of planned actions into commands appropriate for the current context and reacts to events [Ingrand and Ghallab, 2017]. Furthermore, generating sequences of actions can be studied as two types of planning problems: motion planning and task planning, being the latter the topic this thesis focuses on.

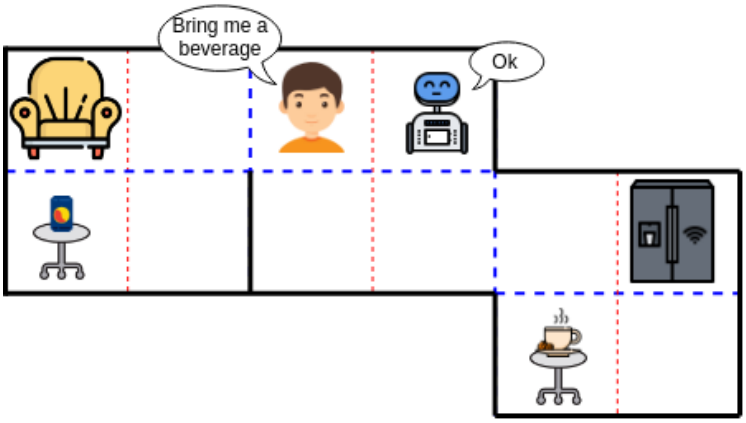

Figure 1.1: The purpose of developing service robots is to assist people in activities commonly found in indoor domains, such as households and offices. The problem of task planning is concerned with generating plans that an agent (in this case the robot) can follow to solve a particular task. However, because people will not provide the details of how tasks can be solved, even activities as simple as bringing a beverage require from a robot certain degree of knowledge of the environment in order to perform them.

While motion planning studies problems such as computing collision-free paths among moving obstacles and performing motions to produce certain relations between objects [Latombe, 2012], task planning produces a high level plan whose whose steps requiring motions can be handled by a specific motion planner [Ingrand and Ghallab, 2017]. Plans are modeled either as a sequence of actions or a policy, however, uncertainty of action outcomes and partial observability of the environment's true state are aspects that must be considered while planning in robotics. Thus, *Partially Observable Markov Decision Processes* (POMDP) have been widely employed for task planning problems since they are capable of modeling an environment's uncertainty and partial observability [Foka and Trahanias, 2007, Zhang

et al., 2013, Kim et al., 2018, Pusse and Klusch, 2019], however, one of the major drawbacks POMDPs have is the high computational cost required to compute an optimal policy (as results from a complexity analysis performed by [Papadimitriou and Tsitsiklis, 1987] suggest that finding an ϵ-optimal policy for a POMDP is PSPACE-complete).

In order to mitigate the computational burden of computing POMDP policies, two main type of approaches have been studied: approximate solving algorithms and problem decomposition. Although, works using the former approach have shown results that compete with those of other exact methods [Pineau et al., 2003, Kurniawati et al., 2008], they still have their limitations, whereas proposals following the latter approach have explored methods that leverage independence relations between variables to transform the original problem into a set of small POMDPs, that can be solved independently, and executed concurrently [Corona-Xelhuantzi et al., 2009] or sequentially [Foka and Trahanias, 2007]; nevertheless, the structure of such decomposition has to be acquired from either, data or an expert in the domain.

Therefore, because of the task diversity inherent to service robotics, extracting a problem's structure from data seems unfeasible. On the other hand, since service robots are designed to perform tasks people already handle with certain degree of expertise, by employing a scheme that defines the dynamics of a skill set as a list of logical and probabilistic parameters, and by integrating various skills into a single planning process based on their individual definitions, people could pass the underlying structure in tasks to a robot. Hence, in this thesis a general task planning architecture is proposed, based on the premise that by obtaining knowledge on individual skills from a human expert, it is possible to automatically integrate them into a hierarchy of actions (modeled as POMDPs) that is able to solve any task that can be solved through a sequence of actions included in the skill set definitions. Thus, with the integration of a knowledge base as a representation scheme to a method for the construction of a hierarchy of actions, the architecture addresses task diversity and automatically designs the POMDPs required to solve a specific task. Furthermore, given that POMDPs are being used to model abstract actions, state uncertainty and partial observability are taken into account while generating and executing plans.

1.2 Problem description

In the previous section it has been emphasized how important it is for a service robot to be able to behave in a robust way within its environment, of how difficult it is to model a POMDP and how this problem has been addressed in different ways. Excepting approximate algorithms, the rest of the discussed approaches use domain information in some way to reduce the POMDP's size (*i.e.* the cardinality of its state, observation and action spaces), while looking to decrease as little as possible its effectiveness in task planning. In a similar way, this thesis addresses the following problem:

In task planning for service robotics, the characteristics of the environment that represent a greater challenge to find a plan that successfully solves a given task are its size, uncertainty in outcomes of events, partial observability of its true state and task diversity. Since the computational complexity of plan search is a non-decreasing function of the environment's size and the task diversity, while its uncertainty and partial observability preclude to guarantee the success of a plan, there is a need for a task planning methodology that simultaneously addresses plan search complexity, uncertainty and partial observability in order to suffice the short response time and high degree of reliability required in planning for tasks related to service robotics applications.

1.3 Research Questions

- Can domain specific knowledge be employed to build a hierarchy of POMDPs?
- Can recursively optimal POMDP policies perform as effective as a global optimal POMDP policy in task planning problems for service robotics?

1.4 Hypothesis

In task planning problems for service robotics applications, using domain specific knowledge along with a recursive formulation allows to build hierarchies of POMDPs

that perform as effectively and more efficiently than a standard POMDP.

1.5 Objectives

The **general objective** of this research is:
To develop an architecture to perform task planning for service robotics applications using domain specific knowledge and a hierarchical representation of the domain based on hierarchies of POMDPs.

The **specific objectives** are:

1. Define a representation for domain knowledge.
2. Design a structure for a knowledge base to organize and represent domain information.
3. Implement a method to convert a symbolic description of the environment into a POMDP.
4. Develop a method to build a hierarchy of POMDPs to model a hierarchy of actions.
5. Develop a method to execute a hierarchy of actions in order to solve a task.

1.6 Scope and limitations

This research is delimited by the following conditions:

- This work focuses on solving task planning, that is, motion planning is out of the scope of this thesis.
- The scenarios where the architecture is evaluated are expected to be indoor and static environments, *i.e.*, once a map of the the environment is built, the position and orientation of things that are part of the map (for instance furniture) shall not be changed.

- A closed world is assumed, that is, a description of the entire environment at some degree of detail is expected to be provided.
- For the architecture to receive task requests from a user, it is assumed there is an external module in charge of parsing requests to a format the architecture understands as a goal state, the robot knows its current state at the moment a task is requested, and requests will not be issued while the robot is solving a task.
- To evaluate the proposed architecture, the navigation domain is considered as the evaluation scenario. Despite other domains could also be addressed with our architecture, such as object manipulation or object recognition, the state spaces in navigation domains have a high degree of structure and tend to be large in real world scenarios, even when the state space is discretized. Thus, the navigation domain is a good fit to evaluate the impact a hierarchical approach with several levels of abstraction might have in problems with a large space.

1.7 Description of the proposed method

The proposed architecture for service robotics task planning is constituted by two main elements: declarative programming and probabilistic graphical models. Declarative programming is used to represent knowledge of the domain over which planning will take place. This representation includes a set of objects, relations, predicates and functions, that altogether make up the knowledge base required to specify the states, actions and observations of a POMDP. Moreover, as an additional form of *a priori* knowledge, a collection of *basic modules* (which are capable each of solving a specific type of task) is required. In order to use a *basic module* in a task planning problem, they are modeled in the knowledge base with a set of states, actions and observations that they are capable of modifying, performing, and perceiving, respectively.

Once a designer has provided a symbolic description of the environment and the robot's skill set, using action language theory, a non-deterministic transition diagram is built to model a POMDP. The POMDP represents the robot in the environment as a dynamic system, and is employed as starting point by a recursive method to build

a hierarchy of POMDPs, in a bottom-up way. The hierarchy of POMDPs enables the agent to probabilistically model task planning problems in an efficient way, as the structure of the hierarchy implicitly provides information of what aspects of the environment can be safely omitted for a specific task, and still be able to solve it.

The proposed method can be divided in three main phases: the knowledge base construction (**KBC**), architecture's initialization (**AI**) and architecture's operation (**AO**) phase, all three are described below.

1. [**KBC**] **Knowledge base construction**. A designer creates an instance of a knowledge base by providing two types of knowledge (see Fig. 1.2): general knowledge, which is associated with the robot's skill set and the aspects of the environment that it is capable of modifying and perceiving through such skills, while specific knowledge is related to facts that are true for a particular environment in which the robot will operate.

    - General knowledge: Description of the domain's dynamics, the robot's basic modules (each one representing a skill it is capable of doing), relations that can be used to abstract into a hierarchy those objects the robot will interact with, and statistics (in the form of probability distribution functions) that describe the probability each basic module has of generating a set of possible outcomes.
    - Specific knowledge: Information of the particular environment in which the robot will operate, *i.e.* a list of facts that are true within such environment. These facts serve as values of the variables that are relevant for the task planning problem and are modeled in the domain's dynamics description. For instance, a list of the rooms that constitute the agent's working environment, as well as their spatial arrangement.

2. [**AI**] **Construction of hierarchy of actions**. From the symbolic description of the environment and the basic modules, a POMDP is built to model the dynamics of the environment at the concrete level in the hierarchical representation. Then, the hierarchy of POMDPs is built that will be required later on in the **AO** phase.

3. [**AO**] **Planning on a task basis**. When a task request is received, the following steps are performed:

   (a) A subregion of the state space is selected, based on the goal state for that task and the robot's current state.

   (b) For each level in the hierarchy of the selected subregion, a local policy is computed to reach the goal state at every level of resolution in the subregion hierarchy.

   (c) The local policies are sequentially executed in a top-down order.

4. **[AO]** Step **3** is performed for every task request received in the future.

In order to evaluate the proposed architecture, a mobile robot navigation domain is employed as study case. Four experiments were designed to evaluate the effectiveness (percentage of successful runs) and efficiency (time required to plan and amount of actions executed to reach a goal state) of the architecture and two baseline methods, for different configurations of uncertainty of the agent's true state, and size of the environment. The first two experiments are constituted by several scenarios that vary in uncertainty, while the last two vary in size. Experimental results show that the proposed architecture is more prone to fail than both baseline methods as uncertainty increases. However, in scenarios with moderate values of uncertainty and large environments, our architecture is able to perform near optimal (in terms of steps taken to reach the goal state) and generate plans in a time that is several orders smaller than other baseline methods.

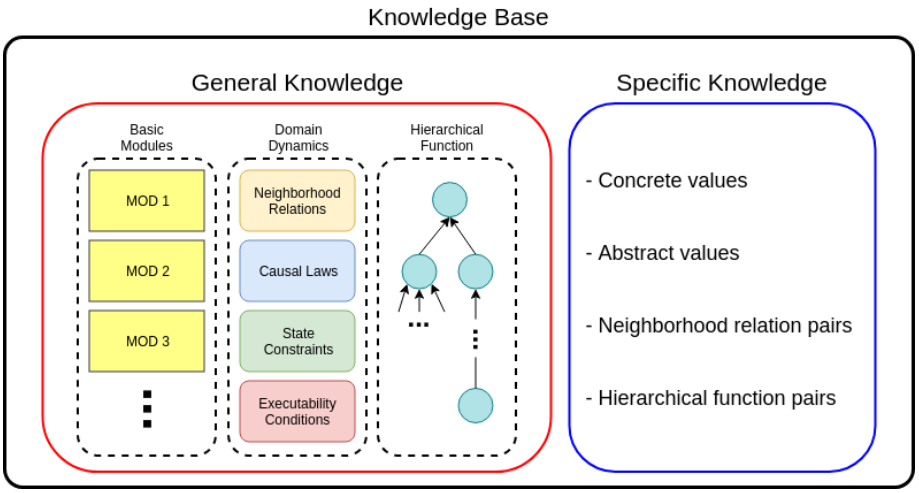

Figure 1.2: Knowledge base composed by general and specific knowledge. The general knowledge is constituted by a description of the domain dynamics that model through a set of rules an agent's interaction with the the environment, a set of basic modules descriptions (each one represents a skill set domain) and a hierarchical function that represents a relation by which the architecture builds a hierarchical representation of the state space. The specific knowledge is formed by lists of particular objects that describe the specific environment (in which the robot will operate) at a concrete and abstract level, as well as the pairs of objects that are neighbors by some action, or are directly related by the hierarchical function.

1.8 Contributions

The main contributions of the work presented in this research are:

1. A general framework for hierarchical task planning, capable of integrating new skills into a planning problem, without having to modify the description of those skills that already are part of the system.

2. A method that automatically builds an arbitrarily deep hierarchy of POMDPs from a hierarchical representation of the state space and a POMDP that models the bottom level of such representation.

3. A methodology to generate and execute a multi-resolution plan in a sub-region of the original state space, employing its hierarchical representation and a hierarchy of POMDPs.

1.9 Document organization

The rest of the document is organized as follows. In chapter 2 is presented the base theoretical framework used throughout this document. The related work to this research is analyzed in chapter 3. Next, in chapter 4 the proposed architecture is described in detail. Then, the experimental results are shown in chapter 5, and finally the conclusions and future work are presented in chapter 6.

Chapter 2

Theoretical framework

In this chapter, we present the basic theory necessary to understand the method proposed by this research. We start by covering Answer Set Programming as a representation paradigm for the knowledge base of our architecture, followed by Markov Decision Processes as way to model reinforcement learning problems, such as decision-theoretic planning. Then, Partially Observable Markov Decision Processes are presented. Finally, we end this chapter with a brief description of the main concepts in Hierarchical Reinforcement Learning.

2.1 ASP: Answer Set Programming

Based on the stable model semantics of logic programming [Gelfond and Lifschitz, 1988], answer set programming (ASP) is a type of declarative programming. Among the most common applications for ASP, there are all kind of difficult search problems, mainly because in ASP search problems can be translated into computing stable models with the help of answer set solvers [Lifschitz, 2008]. Being ASP a declarative programming language, contrary to an imperative language (whether it is procedural or object-oriented) in which a program consists of a list commands that must be executed in a given order, an ASP program is made of a collection of rules and objects that define a problem configuration [Gelfond and Kahl, 2014]. Thus, instead of indicating a system *how* a problem should be solved, ASP programs describe *what* a solution to the problem looks like.

In Answer Set Prolog, which is an instance of an ASP language, a program Π is constituted by a signature Σ and a list of rules. The signature is the program's alphabet, while the rules describe the problem the program is designed to solve. These concepts are presented below.

Definition 1 *A **Signature** Σ of an ASP program Π is a four-tuple $< \mathcal{O}, \mathcal{F}, \mathcal{P}, \mathcal{V} >$ of sets, that contain the objects, functions, predicates and variables that can be used within the program.*

Definition 2 *The following cases are **Terms**.*

1. *Let p be an object constant and X a variable, then both p and X are **Terms**.*
2. *Let $t_1, ..., t_n$ be terms and f a function symbol of arity n, then $f(t_1, ..., t_n)$ is a **Term**.*

Definition 3 *Let $t_1, ..., t_n$ be terms and p a predicate symbol of arity n, then an expression of the form $p(t_1, ..., t_n)$ is an **Atomic Statement** or **Atom**.*

Definition 4 *Let $p(t_1, ..., t_n)$ be an atom and $\neg p(t_1, ..., t_n)$ its negation, then both $p(t_1, ..., t_n)$ and $\neg p(t_1, ..., t_n)$ are **Literals**.*

Definition 5 *Let $p(t_1, ..., t_n)$ be an atom and $t_1, ..., t_n$ terms that are all constant objects or functions with no arguments that are variables or have variables as argument, then both $p(t_1, ..., t_n)$ and its negation $\neg p(t_1, ..., t_n)$ are **Ground Literals**.*

Definition 6 *Let l be a literal, the **Default Negation** of l, **not** l, expresses that it is not believed that l is true.*

Definition 7 *Let l_1 and l_2 be literals, the **Epistemic Disjunction** of l_1 and l_2, l_1 **or** l_2, expresses that l_1 is believed to be true or l_2 is believed to be true.*

Contrary to classical logics, *i.e.* those in which predicates can take one of two possible values (usually *true* and *false*), in Answer Set Prolog a literal can be believed to be *true*, *false* or neither. That is, upon scenarios with incomplete information a system can stay neutral with respect to what it believes of a given literal for which there is no evidence. Thus, the default negation of a literal l is true if $\neg l$ is true, or if there is no evidence about l at all. Moreover, the epistemic disjunction of a collection of literals $l_0, ..., l_m$ forces the system to believe that exactly one of those literals is true, which differs from the definition of the classical disjunction $\vee$. While the classical disjunction outputs a *true* or *false*, the epistemic disjunction outputs a literal, which the system believes to be true.

Definition 8 *Let $l_0, l_i, l_{i+1}, ..., l_m, l_{m+1}, ..., l_n$ be literals, then a statement of the form*

$$l_0 \text{ or } ... \text{ or } l_i \leftarrow l_{i+1}, ..., l_m, \text{ not } l_{m+1}, ..., \text{ not } l_n.$$

*is a **Rule**, where the epistemic disjunction on the left-hand side of the rule, called **Head**, is said to be believed by the program if the conjunction of the literals and default negations in the right-hand side of the rule, called **Body**, is believed.*

Definition 9 *Let $l_0, ..., l_m$, not $l_{m+1}, ...,$ not l_n be the body of a rule, then a rule with an empty head of the form*

$$\leftarrow l_{i+1}, ..., l_m, \text{ not } l_{m+1}, ..., \text{ not } l_n.$$

*is a **Constraint**.*

Definition 10 *Let l_0 or ... or l_i be the head of a rule, then a rule with an empty body of the form*

$$l_0 \text{ or } ... \text{ or } l_i.$$

*is a **Fact**.*

Similar to the distinction between classical and epistemic disjunction, ASP rules should not be read as a logical implication in the classical sense, but rather as a statement in which if the system believes all the literals and default negation of literals in a rule's body to be true, then it is forced to believe one of the literals in its

head to also be true. Furthermore, when the literals in the body of a constraint are believed to be true, since this rule has an empty head the system is forced to believe *nothing*, which could be interpreted as a dead end point for a syllogism. Whereas for facts, since they do not have a body that must be believed in the first place, their head is automatically believed to be true. Hence, based on the concepts of literals and rules, is built the definition of answer set.

Definition 11 *Let* Π *be an ASP program with signature* Σ *and* S *a consistent set of ground literals from* Σ*, such that* S *is consistent with the rules of* Π *and* S *is minimal, i.e., there is no proper subset of* S *that is consistent with the rules of* Π*, then* S *is an* ***Answer Set*** *of* Π*.*

Informally, the definition for the answer set of an ASP program Π says that, an answer set represents a belief (in the form of a set of literals) that believes in both the body and head of every rule in Π, does not believe in contradictions and follows the *Rationality Principle*, that says: *"Believe nothing you are not forced to believe"* [Gelfond and Kahl, 2014].

2.1.1 SPARC

SPARC is a declarative programming language that is an extension of CR-Prolog, which is a version of Answer Set Prolog with consistency restoring rules [Balduccini and Gelfond, 2003]. SPARC integrates sorts to CR-Prolog to enable programmers to explicitly define sorts, as well as properties and relationships between them. Furthermore, by explicitly separating the sort definitions from their properties and relations, SPARC promotes programmers to define rules that describe general properties of a domain without referring to a domain in particular [Balai et al., 2013b].

In this section, the syntax and semantics (from [Gelfond and Kahl, 2014, Balai et al., 2013b]) for SPARC are presented. The syntax is constituted by the definition of directives, sorts, predicates and rules, while its semantics are an extension to the answer set definition of Answer Set Prolog programs.

2.1.1.1 Directives and sort definitions

A SPARC program is constituted by four consecutive sections. The first section, called *directives*, is made of a collection of statements with the following format

$$\#const \ < identifier > = < natural\_number > .$$
$$\#maxint \ = < natural\_number > .$$

The second section starts with the keyword *sorts* followed by a list of sort definitions of the form

$$sort\_name \ = \ sort\_expression.$$

where *sort_name* is an identifier preceded by the pound sign (#), while *sort_expression* represents a list of strings called *sorts*. The *sort_expression* component can be take any of the following six forms.

1. **Numeric range:**

   $$number_1..number_2$$

   where $number_1$ and $number_2$ are natural numbers such that $number_1 \leq number_2$ holds. This expression is equivalent to the set of integer numbers in the closed interval bounded by $number_1$ and $number_2$. For example, given the definition $\#sort = 1..3$ then $\#sort$ is constituted by the set of numbers $\{1, 2, 3\}$.

2. **Identifier range:**

   $$id_1..id_2$$

   where id_1 and id_2 are identifiers that start with lowercase letter. Also, id_1 must be lexicographically smaller or equal than id_2, and id_1 must have a smaller length than id_2. For example, given the definition $\#sort = a..d$ then $\#sort$ is constituted by the set of letters $\{a, b, c, d\}$.

3. **Set of ground terms:**

   $$\{t_1, ..., t_n\}$$

   A set of ground terms may consist of any of the following elements, numbers and identifiers are ground terms; If f is an identifier and $\alpha_1, ..., \alpha_n$ are ground terms, then $f(\alpha_1, ..., \alpha_n)$ is a ground term. For example, $\#sort1 = \{f(a), a, b, 2\}$.

4. **Set of records:**

$$f(sort\_name_1(var_1), ..., sort\_name_n(var_n)) \ : \ condition(var_1, ..., var_n)$$

where f is an identifier, every $sort\_name_i$ occurs in one of the preceding sort definitions and the condition on variables is defined as follows. If var_i and var_j occur in the sequence $var_1, ..., var_n$ and $\odot \in \{>, <, \leq, \geq\}$, then $var_i \odot var_j$ is a condition on $var_1, ..., var_n$; if $\mathcal{C}_1$ and $\mathcal{C}_2$ are conditions on $var_1, ..., var_n$ and $\oplus \in \{\cap, \cup\}$, then $\mathcal{C}_1 \oplus \mathcal{C}_2$ is a condition on $var_1, ..., var_n$. Finally, if $\mathcal{C}$ is a condition on $var_1, ..., var_n$, then $not(\mathcal{C})$ is a condition on $var_1, ..., var_n$. For example, given a pair of sort definitions, $\#s = 1..2.$ and $\#sf = f(s(X), s(Y), s(Z))$, then the sort $\#sf$ consists of the set of records $\{f(1,1,1), f(1,1,2), f(1,2,1), f(1,2,2), f(2,1,1), f(2,1,2), f(2,2,1), f(2,2,2)\}$, where X, Y, Z are variables and $s(\cdot)$ is predicate that is automatically generated by the SPARC compiler for sort $\#s$. Predicate $s(\cdot)$ is true only if its input argument is an element of the sort $\#s$, that is why $\#sf$ is made of every possible permutation of length 3 that can be built with elements of $\#s$.

5. **Set theoretic expression:**

   - $\#sort\_name$
   - An expression of the form (3), denoting a set of ground terms.
   - An expression of the form (4), denoting a set of record.
   - $(S_1 \bigtriangledown S_2)$, where $\bigtriangledown \in \{+, -, *\}$ (stand for union, difference and intersection, respectively), and both S_1 and S_2 are set theoretic expressions.

   For example, given a pair of sort definitions $\#sort1 = \{a, b, 2\}$ and $\#sort2 = \{1, 2, 3\} + \{a, b, f(c)\} + f(\#sort1)$, then $\#sort2$ consists of the following ground terms $\{1, 2, 3, a, b, f(c), f(a), f(b), f(2)\}$.

6. **Concatenation:**

$$[b\_stmt_1]...[b\_stmt_n]$$

where every b_stmt_i is a *basic statements*, which is defined as follows. Statements of the forms (1), (2) and (3) are basic; also, statements of the form (5) if,

   - It does not contain sort expressions of the form (4), denoting sets of records.

- None of the curly brackets occurring in S contains a record.
- All sorts occurring in S are defined by basic statements.

For example, given a pair of sort definitions $\#sort1 = \{b\}$ and $\#sort2 = [\#sort1][1..100]$, then $\#sort2$ consists of identifiers $\{b1, b2, ..., b100\}$.

2.1.1.2 Predicate Declarations

The third section of a SPARC program begins with the keyword *predicates*, which is followed by statements of the form,

$$pred\_symbol(\#sort\_name_1, ..., \#sort\_name_n)$$

where *pred_symbol* is an identifier and $\#sort\_name_1, ..., \#sort\_name_n$ are sorts defined in the previous section of the program. Additionally, multiple declarations containing the same predicate symbol are not allowed, 0-arity predicates must be declared as $pred\_symbol()$, and for any sort name $\#s$ the system automatically includes the declaration $\#s(\#s)$. Predicates automatically included for a sort are used to verify if an element belongs to that sort. For instance, let $\#sort1 = \{a\}$ and $\#sort2 = \{b\}$ be two sorts of a SPARC program, then $\#sort1(b)$ and $\#sort2(a)$ would be false, while $\#sort1(a)$ and $\#sort2(b)$ would turn to be true.

2.1.1.3 Program rules

The fourth section of a SPARC program starts with the keyword *rules*, which is followed by standard ASP rules (as described in section 2.1), enhanced by consistency restoring rules (cr-rule), that are of the form:

$$l_0 \overset{+}{\leftarrow} l_1, ..., l_k, not\ l_{k+1}, ..., not\ l_n \tag{2.1}$$

where ls are literals. A cr-rule, is a special type of rule that is only believed by the program to be true if there is no way to obtain a consistent set of beliefs using solely standard ASP rules. A cr-rule enables the integration of indirect exceptions to defaults of ASP programs. Take the following ASP program as an example.

$$
\begin{aligned}
p(X) &\leftarrow c(X), not\ \neg p(X).\\
q(X) &\leftarrow p(X).\\
c(x).&\\
\neg q(x).&
\end{aligned}
$$

Given observation $c(x)$, and that $p(x)$ has not been observed, the body of the first rule is believed, and consequently $p(x)$. Then, due to the second rule, $q(x)$ is also believed to be true, however, it contradicts observation $\neg q(x)$. By adding to the program above a cr-rule of the form $\neg p(X) \stackrel{+}{\leftarrow} c(X)$, its consistency would be restored, since by including the cr-rule is the only way for the program to generate non-empty answer sets. Furthermore, cr-rules are employed to model the *contingency axiom* for rules that represent defaults over objects.

The *contingency axiom* states that *"Any element of class c can be an exception to the default d(X), but such a possibility is very rare and, whenever possible, should be ignored"*. That is, it is always possible for a default rule (such as the first one in the program above) to be wrong about an object having a property when there is no information about it. Thus, cr-rules make possible for an ASP program to retract about all the consequences derived from mistakenly assuming something.

2.1.1.4 Answer sets

A set of ground literals S is an answer set of a SPARC program Π, with regular rules, only if S is an answer set of an ASP program consisting of the same rules. However, in order to define the semantics of a general SPARC, it is necessary to define it as a CR-Prolog program (after all, SPARC is a sorted version of CR-Prolog). Thus, a CR-Prolog is defined by a tuple of four elements: a) A signature, b) a collection of regular ASP rules, c) a collection of rules of the form of Eq. 2.1, and d) a partial order defined on the sets of cr-rules. According to [Gelfond and Kahl, 2014], the inference engine of CR-Prolog supports only two relations:

1. $\leq_1$: $R_1 \leq_1 R_2$ holds if and only if $R_1 \subseteq R_2$.
2. $\leq_2$: $R_1 \leq_2 R_2$ holds if and only if $|R_1| \leq |R_2|$.

Now, let $\alpha(r)$ denote a regular ASP rule obtained from a consistency restoring rule r by replacing $\stackrel{+}{\leftarrow}$ by $\leftarrow$; α is expanded in a standard way to a set R of consistency restoring rules, *i.e.* $\alpha(X) = \{\alpha(r) : r \in R\}$. Also let Π_r and Π_{cr} be sets of regular and consistency restoring rules, respectively, of a CR-Prolog program Π, then abductive support and the semantics for a CR-Prolog program are defined as follows.

Definition 12 *A minimal (with respect to one of the partial orders $\leq_1$ or $\leq_2$) collection R of cr-rules of Π, such that $\Pi_r \bigcup \alpha(R)$ has at least one answer set, is called an* ***abductive support*** *of Π.*

Thus, a set A is called an answer set of a CR-Prolog program Π if A is an answer set of a regular program $\Pi_r \cup \alpha(R)$ for some abductive support R of Π. Therefore, if A is an answer set for a CR-Prolog program, then it also is an answer set for a SPARC program.

2.1.2 Action Language for transition diagrams

Transition diagrams offer a feasible alternative to model systems that change over time. A state transition diagram is a directed graph employed to model objects and systems that have a finite amount of possible states. In a transition diagram graph, the vertices represent states the system can reach, while the directed arcs denote transitions, that correspond to certain event that triggers that transition from the starting to the ending state [Attenborough, 2003]. Moreover, in a transition diagram with a single agent system (for which it is assumed that the agent is the only entity able to modify the state of the system), arcs will model the agent's actions.

Given that action languages are formal models that serve as a tool to describe the behavior of dynamic systems, as well as their respective transition diagram [Gelfond and Kahl, 2014], they seem to be a natural approach to model the interaction of an agent with its environment if the rules that govern such interactions can be provided. Thus, the syntax for an action language is briefly described below.

An action language $\mathcal{AL}$ is defined by a sorted signature containing three special sorts: statics, fluents and actions. Statics and fluents are properties that are

employed to describe a domain (*e.g.*, the robot's location, or its battery charge status). The difference between statics and fluents is that the value of a fluent can be changed over time, while statics' values cannot, furthermore, fluents are divided into two subcategories: defined and inertial fluents. The difference between them is that the value of a defined fluent depends on the value of other value of other fluents, while the value of an inertial is independent to other fluents (*e.g.*, for a program that has fluents *battery_level* and *battery_discharged* that describe the level of charge of a robot's battery and whether it is discharged or not, the latter would depend on the value of the former). Both statics and fluents are also called domain properties, and a domain literal is a domain property p or its negation $\neg p$. If a domain literal l is formed by a fluent, then it is called a fluent literal, otherwise, it is a static literal. With regards to the sort of actions, it will hold the set of actions an agent can perform. In an action language $\mathcal{AL}$, the following statements are allowed:

1. *Causal laws:*

$$a \textbf{ causes } l_{in} \textbf{ if } p_0, ..., p_m \tag{2.2}$$

2. *State Constraints:*

$$l \textbf{ if } p_0, ..., p_m \tag{2.3}$$

3. *Executability Conditions:*

$$\textbf{impossible } a_0, ..., a_k \textbf{ if } p_0, ..., p_m \tag{2.4}$$

where a_i is an action, l is an arbitrary domain literal, l_{in} is a literal formed by an inertial fluent, $p_0, ..., p_m$ are domain literals, $k \geq 0$, and $m \geq -1$ (in case that $m = -1$, then keyword **if** is omitted). Thus, a system description SD of $\mathcal{AL}$ is a collection of $\mathcal{AL}$ statements [Gelfond and Kahl, 2014]. Furthermore, non-deterministic causal laws can be incorporated to describe stochastic transitions, *i.e.*, transitions that have a set of several possible ending states. To do so, a defined fluent must be added to Eq. 2.2, leading to Eq. 2.5. For non-deterministic rules, a defined fluent *df* is required to define the set of literals that qualify as a possible outcome after performing action a in a stochastic domain, such that every possible outcome is a literal, from $\{l_0, ..., l_q\}$, that satisfies *df*. The defined fluent can take as input arguments some of the literals that are, either part of the rule's premise or not; this will be up to the knowledge

the designer has on which fluents influence the effect of action a in the real world.

$$a \textbf{ can cause } l_0 \textbf{ or}...\textbf{or } l_q \textbf{ if } p_0, ..., p_m \textbf{ such that } df \tag{2.5}$$

2.1.3 Action Language in SPARC

In order to define a system description of $\mathcal{AL}$ in SPARC, in [Gelfond and Kahl, 2014] a formulation to encode the signature and statements of a system description is provided. Such notation, that converts an action language system description to an ASP program, has its equivalent for a sorted language such as SPARC, which is described below.

The encoding $\Pi(SD)$ of a system description SD consists of a SPARC program that results from the encoding of the signature of SD and rules obtained from the statements of SD.

- **Encoding of the signature:** Let $sig(SD)$ be the encoding of the sorted signature of SD.
  - For each constant symbol c of sort $sort\_name$ other than fluent, static or action, $sig(SD)$ contains a sort definition $\#sort\_name$ such that $sort\_name(c)$ is true.
  - For every static g of SD, $sig(SD)$ contains the sort definition $\#static$ such that $static(g)$ is true.
  - For every inertial fluent f of SD, $sig(SD)$ contains the sort definition $\#inertial\_fluent$ such that $inertial\_fluent(f)$ is true.
  - For every defined fluent f of SD, $sig(SD)$ contains the sort definition $\#defined\_fluent$ such that $defined\_fluent(f)$ is true.
  - For every action a of SD, $sig(SD)$ contains the sort definition $\#action$ such that $action(f)$ is true.
- **Encoding of the statements:** Since only two time steps (0 and 1) are required to encode a transition, then for the encoding of statements, the pair

of time steps is encoded with the following sort definitions:

$$\#const\, n \;=\; 1.$$

$$\#step \;=\; 0..n.$$

Moreover, to simplify the description of the encoding, the notation of $h(l, i)$ and $occurs(a, i)$ are introduced, where l is a domain literal, a an action and i a time step. If f is a fluent, $h(l, i)$ denotes $holds(f, i)$ if $l = f$, or $\neg holds(f, i)$ if $l = \neg f$, at time step i. Also, $occurs(a, i)$ denotes that action a occurred at time step i. Thus, the statements of SD are encoded as follows:

– For every causal law

$$a \textbf{ causes } l \textbf{ if } p_0, ..., p_m$$

$\Pi(SD)$ contains

$$\begin{aligned} h(l, I+1) \leftarrow\; & h(p_0, I), ..., \\ & h(p_m, I), \\ & occurs(a, I), \\ & I < n. \end{aligned} \tag{2.6}$$

where Eq. 2.6 says that if at instant I the set of literals $\{p_0, ..., p_m\}$ are true, action a is performed and instant I is not that last instant of time in the SPARC program, then the fluent l will be true at instant $I+1$.

– For every non-deterministic causal law

$$a \textbf{ can cause } l \textbf{ if } p_0, ..., p_m \textbf{ such that } df$$

$\Pi(SD)$ contains

$$\begin{aligned} 1\{h(l, I+1) \;:\; h(df, I)\}1 \;\leftarrow\; & h(p_0, I), ..., \\ & h(p_m, I), \\ & occurs(a, I), \\ & I < n. \end{aligned} \tag{2.7}$$

Similar to Eq. 2.6, Eq. 2.7 states that if the elements of the rule's body are true at instant I, then every fluent l that satisfies the defined fluent at

instant I is a possible outcome at instant $I+1$. In SPARC, the notation $a\{l\ :\ c\}b$ describes the epistemic disjunction of every set of literals with cardinality greater than or equal to a and less than or equal to b, where every element l of each set satisfies property c.

– For every state constraint

$$l \textbf{ if } p_0, ..., p_m$$

$\Pi(SD)$ contains

$$h(l, I) \leftarrow h(p_0, I), ..., h(p_m, I). \tag{2.8}$$

where Eq. 2.8 states that if the set of literals $\{p_0, ..., p_m\}$ is true at instant I, then literal l will also be true at instant I.

– $\Pi(SD)$ contains the Closed World Assumption for defined fluents:

$$\neg h(F, I) \leftarrow defined\_fluent(F), not\ h(F, I). \tag{2.9}$$

where Eq. 2.9 says that if a defined fluent F was not observed to be true at instant I, then the system will assume that it is false at instant I.

– For every executability condition

$$\textbf{impossible } a_0, ..., a_k \textbf{ if } p_0, ..., p_m$$

$\Pi(SD)$ contains

$$\neg occurs(a_0, I)\ or\ ...\ or\ \neg occurs(a_k, I) \leftarrow h(p_0, I), ..., h(p_m, I) \tag{2.10}$$

where Eq. 2.10 states that if the set of literals $\{p_0, ..., p_m\}$ is true at instant I, then any of the actions in the set $\{a_0, ..., a_k\}$ can be executed at instant I.

– $\Pi(SD)$ contains the inertia axiom:

$$\begin{aligned} h(F, I+1) \leftarrow\ & inertial\_fluent(F), \\ & h(F, I), \\ & not\ \neg h(F, I+1), \\ & I < n. \end{aligned} \tag{2.11}$$

$$\begin{aligned} \neg h(F, I+1) \leftarrow\ & inertial\_fluent(F), \\ & \neg h(F, I), \\ & not\ h(F, I+1), \\ & I < n. \end{aligned} \tag{2.12}$$

where Eq. 2.11 and Eq. 2.12 state that the value of an inertial fluent F at instant $I+1$ will remain the same as from instant I, if the value of F was not observed to change from instant I to $I+1$.

– $\Pi(SD)$ contains Closed World Assumption for actions:

$$\neg occurs(A, I) \leftarrow\ not\ occurs(A, I). \tag{2.13}$$

Similar to Eq. 2.9 for defined fluents, Eq. 2.13 states that if action a was not observed to occur at instant I, then the system will assume that action a did not occur at instant I.

An important remark has to be made, although Eq. 2.5 and Eq. 2.7 are not part of the definitions provided in [Gelfond and Kahl, 2014] (the main source document for section 2.1.3) they are supported in SPARC and I believe this is the correct section to present them as they will be required in chapter 4.

Thus, after defining the notation to encode a system description into a SPARC program $\Pi(SD)$, what remains is to specify what constitutes a transition within the transition diagram represented by SD. Let $h(\sigma_0, 0)$ be an initial state and $occurs(a, 0)$ an action, then their encodings are:

$$h(\sigma_0, 0) = \{h(l, 0) : l \in \sigma_0\}$$
$$occurs(a, 0) = \{occurs(a_i, 0) : a_i \in a\}$$

Also, let $\Pi(SD, \sigma_0, a) = \Pi(SD) \cup h(\sigma_0, 0) \cup occurs(a, 0)$, then a transition in the transition diagram represented by the encoding $\Pi(SD)$ is defined as follows.

Definition 13 *Let a be a nonempty collection of actions and σ_0 and σ_1 be states of the transition diagram $T(SD)$ defined by a system description SD. A state-action-state triple $< \sigma_0, a, \sigma_1 >$ is a **transition** of $T(SD)$ if and only if $\Pi(SD, \sigma_0, a)$ has an answer set A such that $\sigma_1 = \{l : h(l, 1) \in A\}$.*

2.2 Markov Decision Processes

Markov decision processes are a framework employed to model, and eventually solve (*i.e.* compute its policy), sequential decision problems in systems whose state changes over time. An MDP is focused on problems that satisfy the following properties: a) time is discretized into instants of time, b) the system is controlled by an agent, and c) the agent always knows with certainty its current state. Every time an agent interacts with the world by executing an action, the state of the world changes and the agent receives a reward (which depends on the world's state at the moment the action was performed) as a scalar real value. The main goal of an MDP is to maximize the expected reward in the long run, thus, by assigning reward values to *state-action* pairs a designer can model the desired behavior it expects from the agent [Puterman, 2014].

A Markov decision process is formally defined by a tuple $< S, A, \Phi, R >$, where

- S is a finite set of states $\{s_1, ..., s_n\}$;
- A is a finite set of actions $\{a_1, ..., a_m\}$;
- $\Phi : S \times A \times S \rightarrow [0, 1]$ is the state transition function that specifies a probability distribution, for each state and action, over all states, where $\Phi(s, a, s')$ is the probability of ending in state s' after executing action a while being in state s; and

- $R : S \times A \to \mathbb{R}$ is the reward function, where $R(s, a)$ is immediate reward the agent receives after performing action a in state s.

Once the four tuple of an MDP problem has been fully-defined, what follows is to solve the MDP, however, since its main objective is to maximize its expected utility in the long run, it is necessary to define first what the MDP shall consider *long term*. From this question, MDPs are classified into two categories: a) finite horizon and b) infinite horizon. For finite horizon problems, the process is bounded by a fixed finite known duration, whereas for infinite horizon problems we do not know how long the decision making process will last.

Knowing how much time is left to finish a task is an important aspect that should influence the behavior of a rational agent, lets take the example of a basketball player agent. In basketball, there is a 24 seconds period of possession within which the players of the team that has the ball are forced to make a shoot, or otherwise, the ball is given by the referee to the other team and the timer resets. If an agent had the ball and there was plenty of time, the most intelligent thing to do would be to wait until one of its teammates is in a good position to receive a pass and has a high probability of scoring. However, if there was only a few seconds remaining, the agent should shoot regardless of how far it is from the basket. It might have a low probability of scoring, but it still beats its team chances to score if it decides to make a pass to one of its teammates that are being guarded.

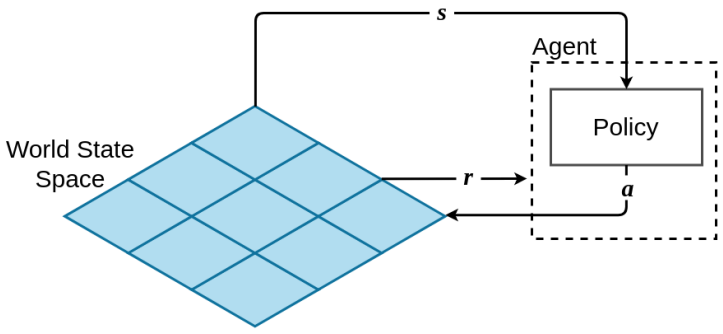

Figure 2.1: In the MDP framework, after the agent executes an action (***a***) it will immediately receive a reward (***r***) and update its state (***s***). In order to select an action, the agent's policy will revise its state and decide which action turns to be the best one.

Once we have an MDP tuple defined, and we know whether it is a finite or infinite horizon problem, the MDP is solved and a policy is obtained. A policy can be seen as a function that tells the agent what is the best possible action to be made, based on its current state. Hence, policies for finite horizon MDPs are known to be non-stationary with respect time, while optimal policies for infinite horizon problems, under some circumstances, can show a stationary behavior [Sucar, 2015].

2.2.1 Policies

Let a decision rule be a function $d_t : S \rightarrow A$ that chooses an action based on an state for a given instant. Decision rules are namely classified by two aspects: how information from previous instants of time is included, and by the way they select actions. Hence, there are four different types: history dependent and deterministic (HD), history dependent and randomized (HR), Markovian and deterministic (MD), and Markovian and randomized (MR); where history dependent rules integrate information on past states and actions to select an action, while Markovian rules depend only on their current state (as any system that has the Markovian property). A deterministic decision rule d_t selects an action based on some historic data $d_t(h_t)$ (history dependent) or its current state $d_t(s_t)$ (Markovian). On the other hand, a randomized decision rule specifies a probability distribution function over the set of actions, also, based on historic data $q_{d_t(h_t)}(\cdot)$ or its current state $q_{d_t(s_t)}(\cdot)$ [Puterman, 2014].

A policy for an MDP, is a Markovian deterministic decision rule (since it bases its decision solely on the agent's current state, in a deterministic way), and also a function $\pi : S \rightarrow A$ that selects an action $a_i \in A$ for every state $s_j \in S$, where A and S are the sets of actions and states of the MDP, respectively. Furthermore, an optimal policy π^* selects the *best* action, *i.e.* that maximizes the expected reward in the current state. For an infinite horizon MDP, with a discount factor of $0 \leq \gamma < 1$, the optimal policy is described by equation 2.15, which is obtained from equation 2.14 (Bellman equation [Bellman, 1957]).

$$V^{\pi}(s) = max_a \left\{ R(s,a) + \gamma \sum_{s \in S} \Phi(s,a,s') V^{\pi}(s') \right\} \tag{2.14}$$

$$\pi^*(s) = argmax_a \left\{ R(s,a) + \gamma \sum_{s \in S} \Phi(s,a,s') V^{\pi}(s') \right\} \tag{2.15}$$

$$V_t(s) = max_a \left\{ R(s,a) + \gamma \sum_{s \in S} \Phi(s,a,s') V_{t-1}(s') \right\} \tag{2.16}$$

Among the methods used to find an optimal MDP policy, value iteration is one of the basic methods. It starts by initializing the value of every state with 0, and iteratively updates state values with equation 2.16 for every state and action. This update operation is repeated until $|V_t(s) - V_{t-1}(s)| < \epsilon$ is true for every state, for certain convergence threshold ϵ. Furthermore, as an alternative to value iteration, policy iteration is a policy solving algorithm that, although has greater computational complexity, tends to converge in fewer iterations than value iteration [Sucar, 2015].

2.2.2 Partially Observable Markov Decision Processes

Full observability of the system's state is an assumption that holds for many sequential decision problems, thus, an MDP is employed in such cases. For instance, a dice player agent could be modeled as an MDP since the outcome of throwing a dice is uncertain, however, the agent can know with certainty the state that is being reached, say the sum of a couple of dices. Yet, there are domains that do not comply this assumption, being robotic systems one of those. For these domains, there is an extension to the MDP framework, known as partially observable Markov decision processes (POMDP). A POMDP is an MDP that relaxes the full-observability assumption for states, and does not know with certainty its true state. A POMDP employs a probability distribution function over the state space, called belief state, to keep a track of its location within the belief space, *i.e.*, where it *believes* it is [Kaelbling et al., 1998].

A partially observable Markov decision process is formally defined by a tuple $< S, A, \Phi, R, O, \Omega, B_0 >$, where

- S is a finite set of states $\{s_1, ..., s_n\}$;

- A is a finite set of actions $\{a_1, ..., a_m\}$;
- $\Phi : S \times A \times S \rightarrow [0, 1]$ is the state transition function that specifies a probability distribution, for each state and action, over all states, where $\Phi(s, a, s')$ is the probability of ending in state s' after executing action a while being in state s;
- $R : S \times A \rightarrow \mathbb{R}$ is the reward function, where $R(s, a)$ is the immediate reward the agent receives after performing action a in state s;
- O is a finite set of observations $\{o_1, ..., o_l\}$;
- $\Omega : S \times A \times O \rightarrow [0, 1]$ is the observation function that specifies a probability distribution, for each state and action, over all observations, where $\Omega(s, a, o)$ is the probability of perceiving observation o after taking action a and state s is reached; and
- B_0 is the initial state distribution that describes the probability of being in each state at the first instant of time.

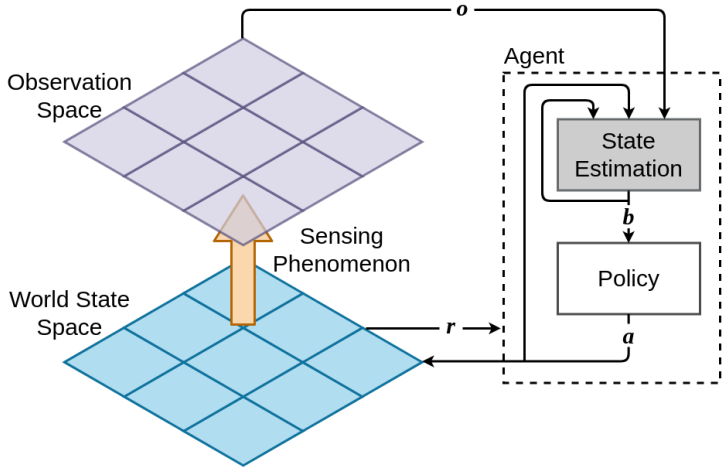

Figure 2.2: In the POMDP framework, after the agent executes an action (***a***), it will immediately receive a reward (***r***) and perceive an observation (***o***), by means of measuring the environment with some sort of sensor. In order for the agent's policy to select the best action, first, it updates its belief state distribution (***b***), using the latest action, observation and the current belief distribution.

Instead of tracking the agent's state, the belief state is updated at every time step, based on the its latest action, observation and current belief state, using equation

2.17, where η is a normalizing coefficient. Moreover, according to [Kaelbling et al., 1998] if the belief state is properly computed, then there is no information that previous observations and actions could provide, so that the agent would gain more knowledge of its current state.

$$b_{t+1}(s') = \eta\Omega(s', a, o) \sum_{s \in S} \Phi(s, a, s') b_t(s) \tag{2.17}$$

With regards to the policy of a POMDP, just like an MDP policy, it is a Markovian deterministic decision rule, however, instead of selecting an action given its current state, it determines which of the policy's α-vectors bounds the region (in the belief space) that contains the point equivalent to the current belief state, then, it selects the action associated to that α-vector [Pineau et al., 2003]. In fact, there are also versions of value iteration and policy iteration [Braziunas, 2003] for POMDPs, however, because computing optimal policies for POMDPs becomes computationally intractable as the problem grows, several methods have been proposed and have shown to provide near optimal solutions by taking an approximate approach, for instance, PBVI [Pineau et al., 2003] (which is employed in the research presented in this document), PERSEUS [Spaan and Vlassis, 2005], SARSOP[Kurniawati et al., 2008].

2.3 Hierarchical Reinforcement Learning

Reinforcement learning (RL) is concerned with problems that model a dynamic system, *i.e.* changes over time, as a set of states and actions, where the objective is to have an agent to learn a behavior that, through actions, controls the system's state in a way that maximizes certain criterion of optimality [Kaelbling et al., 1996]. Because MDPs use states and actions to model the interaction between an agent and its environment, they have become standard framework to model RL problems, among which we find decision theoretic planning to be one of them.

With regards to algorithms used to solve an MDP, there are two main types: model-based and model-free. This dichotomy arises from the assumption that the transition-state and reward functions, Φ and R respectively, are available to the algorithm (model-based). On the other hand, model-free algorithms estimate these

functions by means of interacting with the environment. However, both approaches have difficulties to scale for large and complex problems.

Hierarchical reinforcement learning (HRL) aims at using the structure of a problem to decompose it into several sub-problems, that are individually easier to solve and whose solutions can be combined to solve the original problem, commonly known as the *divide and conquer* strategy [Hengst, 2012]. Particularly, HRL focuses on systems that show a hierarchical structure, that is, they are constituted by several sub-problems that might also have a hierarchical structure. According to [Polya, 1945], hierarchical systems have the *nearly decomposable* property which refers to the fact that intra-component links are stronger than inter-component linkages, therefore, enabling the transformation of a hierarchical problem into an equivalent hierarchy of several, hopefully smaller, problems.

Since HRL is concerned on specifying a useful hierarchical decomposition of an RL problem, and RL problems are partially defined by a set of actions and a set of states, the concepts of abstract action and state abstraction present two ways in which the decomposition of the problem can take place, each offering different advantages that should be analyzed to determine if they suit a particular problem, both are described below.

2.3.1 Abstract actions

Abstract actions are defined as temporally persistent actions, given that when they are invoked, a sequence of several actions takes place leading to a multi-step duration action. In the context of HRL, abstract actions are modeled as policies, that are invoked by a policy (parent action) and in turn invoke other policies (child action). For RL problems that are decomposed into several MDPs, if an MDP incorporates abstract actions, it is known as a semi Markov Decision Process (SMDP) [Puterman, 1990], which is a formalism to model abstract actions as a generalization of primitive actions (those in the original MDP's set of actions) by adding the variable of time to describe the amount of steps an action lasts. Thus, primitive actions are a special type of abstract action that always last one time-step.

From the parent-child relationship between abstract actions arises a structure that represents the behavior of the agent, as shown in Fig. 2.3. Moreover, if such structure is made of SMDPs, it is called a task hierarchy [Dietterich, 2000], in which nodes with children nodes are abstract actions, while those with no children are primitive actions, since they cannot invoke other actions. Furthermore, if there is knowledge available on the problem at hand, a designer could use it to define a task hierarchy that might help to learn the overall policy faster than a standard MDP would.

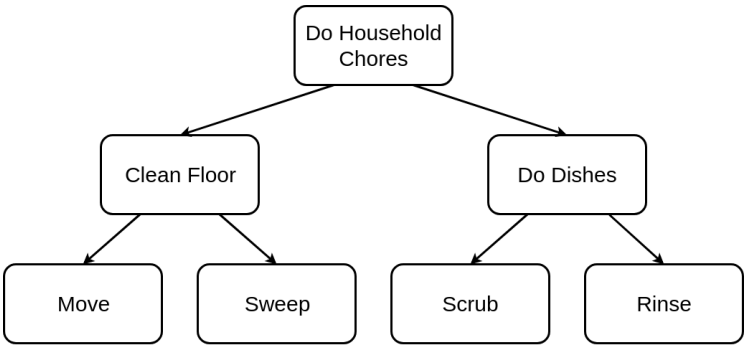

Figure 2.3: By means of a parent-child relation, a task hierarchy describes the tasks that must be solved (child tasks) before attempting to solve more general and complex tasks (parent tasks). For instance, in an apartment that had no furniture, just dirty dishes and a dusty floor, cleaning chores would consist of sweeping the floor and washing the dishes. However, in order to solve these last two tasks, a robot must first be able to sweep, move through the apartment, scrub and rinse dishes.

2.3.2 State abstraction

An abstracted state space is smaller than the original one (in which every state the system can take is included), and leads to RL problems with a smaller complexity. Since the reduction or abstraction of the state space could endanger the possibility of finding a solution for the task at hand, *i.e.* by omitting relevant information about the system's state or a state that is necessary to eventually transit to a goal state, in [Dietterich, 2000] are presented two conditions under which state abstraction can be performed: eliminating irrelevant variables and funneling.

- **Eliminating Irrelevant Variables**: Variables that do not have an effect on the learning process of a policy for a given task can be omitted, since the state-transition and reward signal do not depend in any way to the value of

such variables. For instance, learning to grasp a glass bottle in a kitchen during summer, should be exactly the same to grasping the same bottle during winter, thus, the variable that describes the current season could be safely removed from the learning process. Thus, by ignoring an irrelevant variable, the states that differ only in the value of the ignored variable are grouped into a single block, leading to a reduced state space in which each block is a state.

- **Funneling**: For decomposed problems in which abstract actions tend to finish their execution within a small set of states after they are invoked from an element of a large set of initial states, then the original state space can be reduced to a space in which each state corresponds to the set of resulting states of each abstract action. Because the starting state does matter as much as the ending state, modeling only resulting states maintains the amount of information required to finish the overall task, while it might significantly reduce the complexity of the problem.

2.3.3 Optimality

As good as HRL is in terms of reducing a problem's complexity, it also has a major drawback. Given that by decomposing a task into several smaller ones, it is necessary to specify what a goal is for each one of these new sub-tasks, which in the context of the original problem can be seen as sub-goals. Each sub-goal is modeled in a way that induces an optimal behavior within its sub-task, however, depending on the problem's decomposition, this same behavior might or might not be optimal in the context of the original problem, since sub-goals incorporate information relevant to their respective sub-task. Therefore, three different criteria of optimality are defined to describe the quality of local policies in a hierarchy. These concepts are presented below and Fig. 2.4 summarizes how they rank against each other.

- **Hierarchically optimal (HO)**: A hierarchically optimal policy for MDP M is a policy that achieves the highest cumulative reward among all policies consistent with the given hierarchy [Dietterich, 2000]. That is, local policies that seek to maximize the hierarchy's overall reward, which strongly depends on how sub-goals are modeled.

- **Recursively optimal (RO)**: A recursively optimal policy is one that seeks to

maximize the reward within its local sub-task, regardless of what its resulting state would mean in a broader scheme for the global problem.

- **Hierarchical greedy optimality (HGO)**: Since the conditions that make a given abstract action the best option might change along the way during its execution, by committing to finish it might induce a sub-optimal behavior. Thus, [Dietterich, 2000] defines a hierarchical greedy execution as the process of constantly interrupt the execution of an abstract action to evaluate at in-between time steps if there is a better abstract action that should take control.

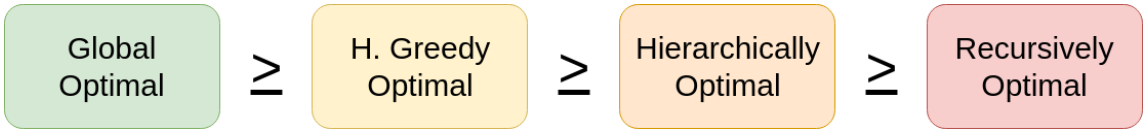

Figure 2.4: Since RO policies consider only local information they cannot do better than HO. According to [Hengst, 2012], although HGO does not offer any guarantee for global optimality (GO), it does guarantee to be no worse than HO, thus, it is safe to rank HGO right between GO and HO.

2.4 Chapter Summary

In this chapter, the basic theory on answer set programming, partially observable Markov decision processes and hierarchical reinforcement learning has been presented, which is employed in the work of this thesis document. By using an ASP language such as SPARC in tandem with an action language, it is possible to model stochastic dynamic systems as non-deterministic transition diagrams from a system description, encoded as an ASP program made of sorts, predicates and rules. Also, the POMDP framework has been introduced as a tool to model planning problems in partially observable environments, as well as some approximate model-based algorithms to compute its policy. Finally, basic theory on hierarchical reinforcement learning, including hierarchical system, abstract action, state abstraction and types of optimality for policies within hierarchical structures, have been presented to show how standard reinforcement learning concepts are incorporated in hierarchical solutions.

Chapter 3

Related work

Task planning in service robotics is a problem that has been addressed in a variety of approaches that differ mainly in the representation used to model the problem, as well as the way the planning system interacts with the domain. In order to present an analysis of the most related research, the reviewed literature has been grouped in two sections. First, related work on architectures designed for service robotics applications is presented. Next, hierarchical approaches for MDPs and POMDPs are summarized. This chapter is concluded with a discussion of the reviewed literature, with the purpose to establish where this work presented stands compared to the analyzed research.

3.1 Architectures applied towards service robotics

Since in domestic robotics domains, task planning problems can encompass a wide variety of tasks, it is necessary to endow service robots with a representation model that integrates into a single framework all these tasks in order to generate plans that solve all of them. According to [Ingrand and Ghallab, 2017], approaches for the specification of deliberation models for planning and acting can be grouped in two categories: single model and multiple model. While the former comprises into a single representation both, descriptive models (those employed to the describe the scenario's state) and operational models (those used to specify how actions should be performed), the latter approach uses separate representations, which leads to

a design philosophy that promotes modularity, as several operational models are supported. In this section, a collection of architectures are presented, which employ the multiple model approach in tandem with a diversity of planning techniques to address the problem of task planning in service robotics.

In [Galindo et al., 2008], in order to represent the environment they use semantic maps, which combine hierarchical spatial information and semantic knowledge, which is employed as the search space for task planning problems. Such structures model spatial connectivity between locations and semantic labels for them. By assigning a label to each location, the architecture can infer implicit information such as those objects that are most likely to be there. With its inference mechanism, the system is able to discard locations that are irrelevant for a particular planning problem, thus improving planning efficiency with respect to not using inference.

One of the greatest challenges robots must face when solving task planning problems is reasoning with incomplete information and the partial observability of the environment. For instance, [Weser et al., 2010] address both challenges by defining a set of possible exceptions which are handled either by re-planning or planning for the purpose of gathering missing information. Also, in order to fill for incomplete information the architecture is able to integrate assumptions that, if wrong, the actual scenario is treated as an exception and re-planning is performed. Alternatively, [Chen et al., 2010] deal with incomplete information by integrating to their planning architecture a natural language processing (NLP) module to gather domain related information via spoken dialog. The architecture has a knowledge base where new information is stored and used for future task planning problems, by means of non monotonic inference, using answer set programming (ASP). One of the major contributions of this work is that, thanks to the expressive power of ASP, the system is able to acquire new causal knowledge through dialogs, which demonstrate to increase the types of tasks it can solve. Furthermore, [Hanheide et al., 2011] propose an architecture that integrates a *conceptual layer* (very similar to the semantic knowledge graph from [Galindo et al., 2008], see Fig. 3.1) that models commonsense knowledge with probabilistic relations among concepts represented in a graph; such relations are computed using co-occurrences from the locations database provided by the *Open Mind Indoor Common Sense* project. They use Bayesian inference to update the probability of relations when new evidence is perceived. What is more,

they use a switching planning system that interleaves between a deterministic classical and a decision theoretic planner; by integrating a POMDP, their architecture is able to execute actions that have several possible outcomes.

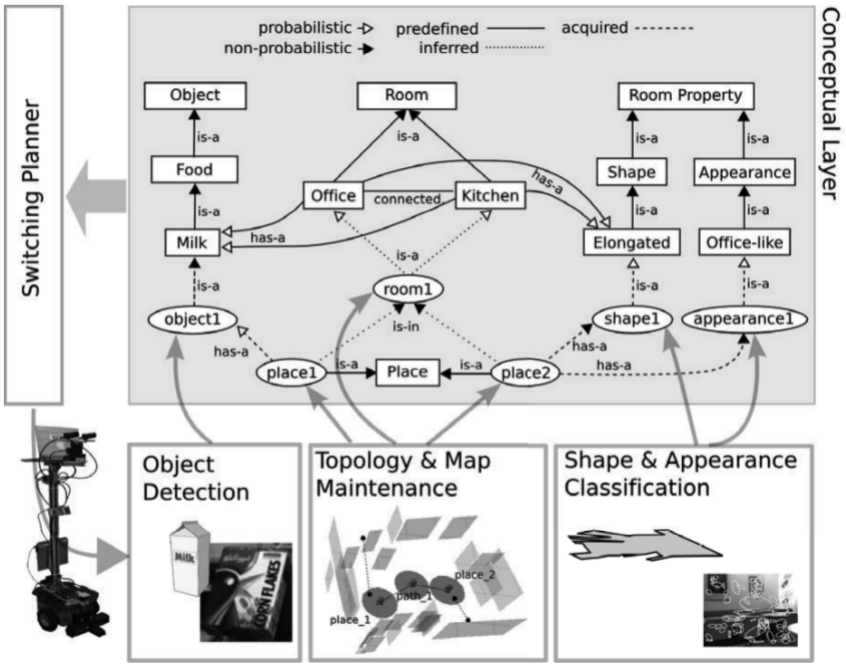

Figure 3.1: Image taken from [Hanheide et al., 2011], shows the conceptual map that integrates into a single framework concepts and instances of objects that describe the environment at different layers of abstraction. Elements in the conceptual map are linked with deterministic and probabilistic relations, that altogether are employed to represent and reason about commonsense knowledge. As new sensory data is perceived, the relational representation is compiled into a chain graph and Bayesian inference is preformed. For plan execution, the planner starts executing a sequence of actions until the probability of the outcome goes below 0.95, then it switches to decision-theoretic planning.

Besides being robust against incomplete information of the world's state, efficiency in planning turns to be essential, since service robotics applications are user-event-driven (*i.e.*, most of them are triggered by an action executed by the human user) and short response times are required. In [Kaelbling and Lozano-Pérez, 2011] this issue is addressed by combining symbolic and geometrical planning, and using a hierarchical top-down approach to decompose abstract actions into more primitive ones. They model the domain's states with conjunctions of fluents, and actions as causal rules. The main advantage this architecture has over classical planners is that it constructs a plan at an abstract level of the hierarchy and commits to it, then

proceeds to plan and execute actions to complete the first task, without building the remaining of the plan. However, this approach relies on the assumption that the execution of a first step will not preclude the execution of following steps is true, which might not always be the case. As a matter of fact, the hierarchical approach by which problems are decomposed into more simple ones, has been adopted by many authors in the design of general purpose architectures, probably because it provides an elegant way to solve subtasks separately and combine their outputs into a single solution. Such is the case of [Keller et al., 2012], which propose an architecture for a planning application, which in turn is part of a larger service robot system. They use a symbolic representation of the domain over which plans of high-level actions are built, while the details of how these actions are performed are encapsulated in independent modules called *semantic attachments*, that take care of the information required to execute a particular action. In this way, by having separate models for the planning and execution components, the architecture supports the integration of new skill sets without modifying those that are already part of the system.

Given that the representations employed for task planning to model the world in which the agent is confined, are actually an abstraction (a simplification that does not include aspects considered irrelevant for the task at hand), it is possible to leave out some phenomena that might influence variables that are part of the model. Because the representation does not have the elements to explain nor predict such phenomena, unexpected events are usually modeled as uncertainty. One way planning architectures deal with uncertainty is by re-planning when the representation of the state of the world they have is inconsistent with the observed evidence, just like [Weser et al., 2010] do. However, depending on the amount of variables that must be considered, re-planning can become a computationally expensive process (in terms of time) that service robotics application cannot afford. Instead, some authors go beyond rectifying a failed plan, by including probability distributions of the agent's actions in the planning process. For instance, in [Zhang et al., 2012] a non-monotonic logic programming language paradigm (ASP, for knowledge representation and inference), and a hierarchical structure of POMDPs (for modeling the uncertainty of sensors) are combined and evaluated in indoor domains to complete the task of locating objects. They also merge the POMDP belief state with a biased distribution generated from the information in the knowledge base of where the object might be found; showing that the more the biased distribution is trusted,

the less it takes to find an object, however, the accuracy also decreases (as a consequence of the uncertainty in the state of the world, objects will not always be where the prior knowledge base indicates). Furthermore, another interesting application of non-monotonic representations can be found in [Pineda et al., 2017], where a knowledge base is specified as a hierarchy of classes using a non-monotonic language, in order to support defaults and exceptions. Their system was evaluated in a real service robot, within a scenario in which the robot takes petitions from the user about what object she desires to be brought to her. They show how their reasoning system takes advantage of the inheritance of properties, among objects in the hierarchy of classes, to adapt to unexpected situations (such as an object being tipped over, hence, precluding its grasping) and still be able to comply a request.

Extending the previous work, [Zhang et al., 2014] generalize the logic and probabilistic based task planning architecture to a two-level scheme, in which at the high level answer set programming is used for representation of the domain and reasoning, while at the low level abstract actions from the high level are implemented as POMDPs. Both levels are coupled using a formalization based on an *action language* [Gelfond and Kahl, 2014]. With this coupling, plans for any domain, that is fully described in ASP, can be implemented as series of POMDPs, in this way they take advantage of the best of both classical and decision-theoretic planning. Also, in [Zhang et al., 2015] the authors evaluate how by integrating default knowledge into an architecture that combines declarative programming and probabilistic models can improve its accuracy on locating objects within an office environment. Furthermore, their system is able to terminate the search of an object based on a beta distribution (models the target existence), whose prior is computed using the evidence in the knowledge base and the amount of times the desired object was found to exist or not in previous searches in other similar domains. This work shows how a task planning architecture can use historical knowledge to improve the behavior of a robot, *i.e.* by deciding that a particular search task cannot be solved, without having to visit every existing location.

In [Sridharan, 2016], they integrate to an architecture, that uses a declarative programming along with POMDPs, the capacity to learn new knowledge about its environment, as a reinforcement learning problem. In this way, the system is able to learn previously unknown domain rules. The architecture's learning skills were

evaluated on multiple simulated trials in which the robot had to arrange objects in specific configurations. At the beginning of the experiments some relevant axioms were not given to the robot on purpose, which eventually led to failed plans. However, once the architecture learned the initially missing rule (*e.g.* that bigger books should not be stacked on smaller ones), and added them to its domain description, it was able to generate successful plans. Alternatively, [Chen et al., 2016] propose a general purpose architecture that uses a *plan-execute-monitor-re-plan* control loop that enables the system to continuously update its representation of the world's current state via sensory data and human-robot interaction. In this way, the robot increases the amount of facts stored in its knowledge base (which can be seen as a form of learning), that are used to generate future plans that are consistent with the latest version of the world's state. From experimental results, they show that despite the architecture does not integrate probabilistic models, by means of continuously sensing and consulting information from humans, they can mitigate uncertainty to such degree that a symbolic representation and planning system is sufficient for a service robot designed to take orders from people. Furthermore, in [Zhang et al., 2017] a logic-probabilistic-based architecture shows to be robust against exogenous phenomena without including it in its model. When the low level components of the system, those in charge of performing the concrete actions, perceive an observation that leads to facts that are inconsistent with the description of the world in the knowledge base, the world's description is updated, a new set of possible states is computed, and used to build on-the-fly an MDP that considers the latest evidence into its planning process. The architecture's adapting capability is evaluated in a navigation problem, which has a random walker in the hall as exogenous phenomena. In this experiment it is shown that by including new evidence in the construction of the MDPs (which perform the planning between two point in the environment), the system significantly reduces the time consumed in execution, in comparison to not adapting to evidence of exogenous events.

In addition to using sensory information to update an architecture's knowledge base to adapt to external changes of the environment that were not initially modeled, other approaches have been proposed to handle the large environments service robots are usually immersed in. That is, since architectures that use a symbolic representation and deterministic planning rely on hand-crafted descriptions of the problem's domain, by automating the acquisition of domain knowledge a system

would not be limited by the designer's expertise. For instance, [Lu et al., 2017] integrate to a service robot task planning architecture a mechanism to automatically build causal rules from an online repository of semantic dictionaries (*FrameNet*[1]). A meta-language is presented to formalize the semantic roles of common verbs found in the repository, then the descriptions of actions are converted from meta-language to causal rules in answer set programming (which is the language used for symbolic representation and planning). Once the system has built a set of abstract actions (that are implemented by other low level actions the robot was initially endowed with) from *FrameNet*, it is ready to receive task petitions issued by a user. On the other hand, although the amount of actions the system is capable of building depends greatly on the semantic parser used to retrieve frames (which is how actions are grouped in *FrameNet*), through experimentation the authors show that by combining the answer set programming planner and their retrieval method, they significantly improve the system's capability to generate plans in comparison to not using semantic online information.

Another approach that has revealed to be a feasible alternative to tackle the complexity of planning in service robotics applications, is modeling the environment as an open world, *i.e.* instead of assuming that the elements of a domain can be listed, the system starts with a knowledge base whose size is increased as the agent gathers new observations via sensory input; such approaches require of methods that enable planning with incomplete information and a high degree of uncertainty. Such is the case of [Hanheide et al., 2017], which proposes an architecture that operates within open and uncertain worlds by using a series of assumptions to fill in for the missing information at planning time. Their architecture makes a distinction between the knowledge that is known to be true by fact, and the assumptions about the value of unknown variables. The values assumed to be true are used to guide the deterministic planning process when relevant information has not been discovered yet, which results in a sequence of actions that are executed afterwards; if an action has several possible outcomes, then a POMDP is built for that section of the planning problem. In this way, the architecture confronts uncertainty and, what is more, is capable of generating explanations for failed plans. From their experiments, the architecture was observed to be robust on exploration tasks, however, it only succeeded half of the runs on searching for an object in the environment, because in some cases planning

[1] https://framenet.icsi.berkeley.edu/fndrupal/

took too long, while in others errors in the robot's sensors caused the failure. Altogether, the system demonstrated a robust behavior for exploration tasks, whereas search tasks with few information still constitute a difficult problem.

In addition to being able to handle incomplete information, partial observability, uncertainty, and large environments, the generality of task planning architectures is also an important parameter due to the high degree of task diversity and variability found in service robotics applications [Ingrand and Ghallab, 2017]. For instance, [Köckemann et al., 2018] propose to use general purpose domains (GPD) as an approach for task planning in real-world robot systems. They claim that it is easier to extend a general domain to a particular context than creating a new one; which is more likely to hold when several tasks will be addressed with the same robotic platform. The problem of domain reasoning is described as determining a specific domain to be used by a task planner given a GPD and the current context. In other words, domain reasoning can be seen as a function that maps a GPD to a particular domain, as a result of performing a series of modifications to the original one, *e.g.* variable substitution, removing unwanted constraints and operators, and structure generation/alteration; thus offering an alternative to address the problems of scalability and extensibility in the deployment of robotic systems. Meanwhile, [Lima et al., 2018] use an architecture composed of a planning coordinator and a set of automatons that implement high level actions, in this way the details related to a particular task, such as navigation or object manipulation, are hidden within each independent automaton; another important consequence of separating the task and motion planning is that the size of the task planner's search space can be significantly reduced in comparison when these problems share the search space. The planning coordinator, which in turn is an automaton, implements a control loop that generates a plan whenever new facts are added to the knowledge base, or the current plan failed. Once a plan is generated, by means of deterministic planning, each of its actions are executed by one of the high level action automatons.

Furthermore, [Sridharan et al., 2018] propose a general purpose architecture that combines the advantages of non-monotonic logical inference and probabilistic probabilistic reasoning in a single framework. A two-level hierarchical representation of the domain is used in order to reduce the computational cost of the deterministic planning (which occurs at a coarse resolution level), while the uncertainty associ-

ated to the output of sensors, as well as executing actions that change the state of the environment, is handled by a POMDP at the fine resolution level. The coarse and fine levels in the hierarchical representation are coupled by a methodology that converts a coarse transition diagram (expressed in an action language) into a fine resolution version, from which a POMDP is derived. Among the main contributions of this work stands out that they detail the formalization used along with the action language in order to generate a description of a domain that can be used in their hierarchical representation. Through a set of experiments, they show how their architecture significantly improves its performance in comparison with a simple POMDP in terms of planning time, amount of executed actions, and success rate.

In Table 3.1 we present a brief comparison of the reviewed architectures. Among the reviewed features, those architectures that were designed for general purpose and perform some sort of hierarchical planning are highly desirable, since they would not be restricted to a domain in particular and, probably, would scale better than a system that does not employ a hierarchical approach in the face of large problems. By employing a non-monotonic representation, the system does not need to revise for consistency in its knowledge base as new knowledge is added. Moreover, decision-theoretic planning enables robust planning in uncertain domains, while deterministic planning provides explainability on the agent's actions. In section 3.3 a discussion is presented, which includes an analysis on how the proposed architecture stands in comparison to the reviewed architectures, with respect to the features presented in Table 3.1.

Table 3.1: Comparison of task planning architectures, which are summarized by the aspects reviewed in section 3.1. The second and third columns specify if an architecture integrates deterministic and probabilistic planning techniques, respectively. The fourth column indicates if some form of hierarchical decomposition is performed to the planning problem, while the fifth and sixth columns show those architectures that incorporate a non-monotonic representation language and are not restricted to a particular type of task planning problem.

Author	Deterministic planning	Decision-theoretic planning	Hierarchical planning	Non-monotonic representation	General purpose
[Galindo et al., 2008]	x		x		
[Weser et al., 2010]	x		x		x
[Chen et al., 2010]	x		x	x	x
[Hanheide et al., 2011]	x	x	x		x
[Kaelbling and Lozano-Pérez, 2011]	x		x		
[Keller et al., 2012]	x				x
[Zhang et al., 2012]	x	x	x	x	
[Zhang et al., 2014]	x	x	x	x	x
[Zhang et al., 2015]	x	x	x	x	
[Sridharan, 2016]	x	x	x	x	
[Chen et al., 2016]	x			x	x
[Zhang et al., 2017]		x		x	
[Lu et al., 2017]	x			x	
[Hanheide et al., 2017]	x	x	x		x
[Köckemann et al., 2018]					x
[Lima et al., 2018]	x		x		x
[Sridharan et al., 2018]	x	x	x	x	x
Our architecture		x	x	x	x

3.2 Hierarchical approaches for solving MDPs and POMDPs

Given that MDPs provide a framework to model sequential decision making problems that encompass events with uncertainty in their outcome, they have become the standard model for planning problems in robotics. Among its variants, POMDPs stand out due to their capacity to plan in scenarios where full observability of the state is not met. However, as the size of the problem's state space increases, to compute the policy of either of them becomes intractable. In an effort to reduce the computational burden that represents computing policies, among others, hierarchical approaches have been proposed, which consist in decomposing the problem into several smaller sub-problems, and arranging them into a hierarchy, so that the solution for the original problem can be obtained by combining the solutions to the sub-problems. Hierarchical approaches offer advantages such as solving smaller (and thus easier) problems, policies can be reused across different contexts, and state un-

certainty can be decreased (in case of POMDPs that perform state abstraction). In this section, research related to hierarchical MDPs and POMDPs is presented, in which some form of hierarchical decomposition is performed.

[Hauskrecht et al., 1998] propose a hierarchical model that employs MDPs along with macro-actions in order to reduce the time required to converge to an optimal solution. Among the most important consequences of employing macro-actions is that the state space of the MDP is a subset of the original problem, due to each macro-action is designed to transit the agent between a particular pair of states from the original state space. Once the original state space has been divided in subregions (state abstraction), those states that are adjacent to a subregion to which they do not belong are called *peripheral states*, and are the ones that constitute the MDP's set of states. Furthermore, since the MDP's state space can be drastically smaller than the problem's original space (which includes every state the system can take), this framework offers an alternative to address large problems, however, it has the major drawback that the point of convergence of the learned value function strongly depends on the design of the macro-actions. Given that the ending state of each macro-action is hand crafted, there is a risk that the agent reaches a state for which there is no permutation of macro actions that can take it to the goal state from there.

[Pineau et al., 2001] use an action decomposition approach. Instead of structuring the state space, they define a hierarchy of tasks in which leaf nodes represent concrete actions, while internal nodes correspond to abstract actions defined by a set of child nodes, which in turn may be abstract or concrete actions, see Fig. 3.2. In this framework, each subtask is implemented as a POMDP whose actions are the child nodes in the task hierarchy, while its sets of states and observations correspond to those of the original problem. For each POMDP, a local optimal policy is computed; this is done in a bottom-up approach, by first solving those subtask that have only concrete actions as children, and then propagating upwards the parameters for the transition, observation and reward functions of its parent subtask by means of the children policies. Also, in order to reduce the size of the overall hierarchy, on a subtask basis they perform a reduction of the state space to those states that are relevant. Moreover, in [Pineau and Thrun, 2002], which is an extension of the previous work, they keep the action decomposition approach to build a hierarchy of

POMDPs, however, among the upgrades made since the last version, the clustering of states and observations is the one that stands out. They extend the algorithm for model minimization of [Dean and Givan, 1997] to also consider the probability of perceiving observations when evaluating stability between clusters. Furthermore, in order to minimize the set of observations, they keep those observations that have a probability greater than zero of being perceived after one of the POMDP's actions is executed, and reaches one of the states that remained after the state space minimization. The framework is evaluated on three simulated domains, as well as in a real world service robotic platform. Overall, their architecture appears to suit information contingent problems, it also showed a good performance after operating for two days and guiding six different elderly people. During this period of experimentation, the robot had to handle high levels of uncertainty by interacting through dialog with the assisted people.

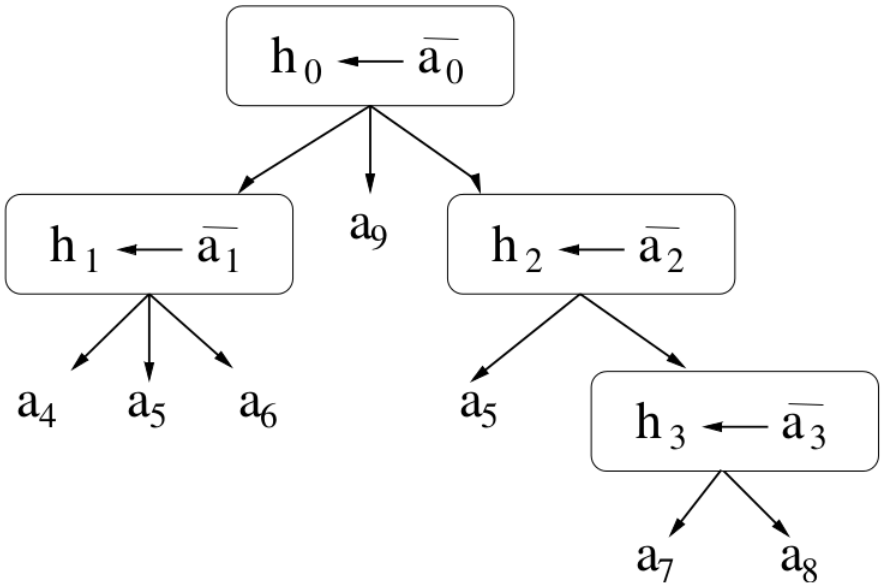

Figure 3.2: Image taken from [Pineau and Thrun, 2002] shows an example of how tasks are represented with a task hierarchy that decomposes the action space. In this example, the original task is decomposed into the set of sub-tasks $\{h_0,h_1,h_2,h_3\}$, each one with their respective set of actions, which may contain abstract ($\{a_0,a_1,a_2,a_3\}$) or concrete ($\{a_4,a_5,a_6,...,a_9\}$) actions.

Furthermore, besides using action decomposition to take advantage of the structure of partially observable domains related to service robotics, state abstraction has

also been employed to exploit the underlying structure of such environments. For instance, [Theocharous et al., 2001] propose a framework based on Hierarchical Hidden Markov Models (HHMMs) [Fine et al., 1998] to build a hierarchy of POMDPs, see Fig. 3.3. In this approach, HHMMs are extended by adding a set of actions to them, which result in a hierarchical POMDP, also the hierarchical Baum-Welch algorithm (used to learn HHMMs) is extended to suit hierarchical POMDPs learning. Therefore, instead of using a dynamic programming approach to compute the hierarchical POMDP's policy, a set of sequences of observations are used to train the POMDPs at the bottom of the hierarchy. In their experiments, they observed that hierarchical POMDPs that trained its low level sub-models separately, converge faster (and sometimes to a higher value) than the flat model and the hierarchical model that trained all its low level sub-models simultaneously. Also, [Theocharous and Mahadevan, 2002] extend their previous framework by integrating multiple entry and exit states to the concept of abstract state, which suits well for abstract states with several neighbor states, such as corridor intersections. By having a multi-resolution representation of the environment, and keeping track of the agent's belief state at each level, as shown in their evaluation results, the architecture is able to make good decisions because it selects which macro-action will be executed at a level in the hierarchy where the entropy of its true location is low (due to the small amount of states at that level).

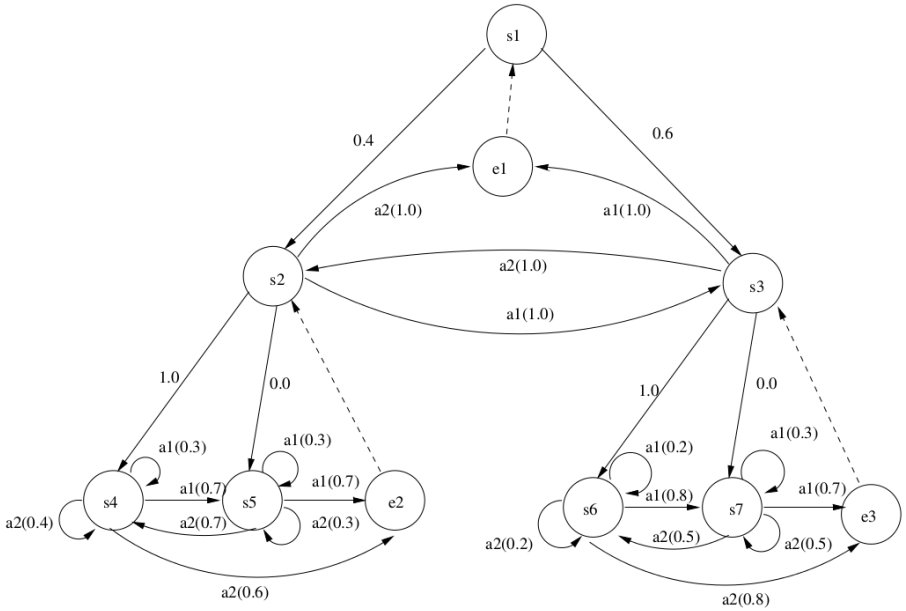

Figure 3.3: Image taken from [Theocharous et al., 2001], shows an example of a hierarchical POMDP that is result of extending the HHMM framework with a set of actions. A hierarchical POMDP has production states (leaf nodes), internal states (states that represent a stochastic process) and end-states (terminate the execution of a stochastic process and return control to their parent node). During execution, to map the belief state to an action, heuristics such as *Most Likely State* (MLS) and *Q-MDP* are employed. Furthermore, in order learn a policy, instead of using Dynamic Programming techniques, the model is trained from a set of sequences of observations, similar to how Hidden Markov Models are learned.

3.3 Discussion

Each work analyzed in this chapter addresses task planning in robotics by proposing a way to either, mitigate the computational cost of computing policies, reduce the agent's state uncertainty, or model a wider variety of tasks. In real scenarios, service robots are expected to bear all these challenges simultaneously, hence the need to develop systems that integrate a general representation that task planning architectures usually employ, as well as the hierarchical decomposition performed by works presented in section 3.2. Thus, this section presents a discussion on the aspects (found in the analyzed literature) that are essential for the development of a general task planning system oriented towards service robotics applications, and which of those are part of the work presented in this thesis.

One of the main components of the proposed architecture is a knowledge base that contains information about the dynamics of the service robot's domain, a set of descriptions for each skill the robot is capable of executing, information of the underlying structures present in the domain, and a description of the particular environment in which the robot will operate. The knowledge base is employed to store a hierarchical representation of the environment's state space, and build a hierarchy of POMDPs that enable the agent to create a plan to solve a particular task a user may request; for which the architecture was not specifically designed for.

In the case of [Pineau et al., 2001, Pineau and Thrun, 2002], despite they take advantage of the hierarchical structure of the task at hand, and use a hierarchy of POMDPs to handle uncertainty while executing a plan, their architecture does not automatically generate a plan for a task other than the one represented by the hierarchy task network from which the hierarchy of POMDPs was built. Meanwhile, even though the architecture proposed by [Theocharous et al., 2001, Theocharous and Mahadevan, 2002] does not depend on a specific task decomposition (since it performs state abstraction), it does rely on the assumption that a set of training examples on how to perform the task at hand will be available, which is difficult to satisfy as the environment gets larger and the amount of actions available increases, since the set of sequences that can possibly be perceived will increase even faster. Also, besides having trouble to plan for a task without the specific details of it beforehand, neither of both approaches (task decomposition and HHMM) shows a methodology to integrate new skills in the planning process, which is necessary in general purpose service robotics, given its high degree of task diversity [Ingrand and Ghallab, 2017].

With regards to the reviewed task planning architectures that are constituted by several components (for instance, knowledge base, multiple modules, multi-layered structure, etc.), most of them integrate a deterministic planning component which enables their systems to generate plans as a static sequence of actions. These type of plans can be useful when used with a symbolic representation, however, for a service robot they are insufficient due to the uncertainty on the effects of the agent's actions when it interacts with its environment. Even though some of these architectures [Weser et al., 2010, Chen et al., 2010, Chen et al., 2016] include methods to mitigate uncertainty (by constantly evaluating the state of the world in order to re-plan)

they present a disadvantage compared to those works that do include probabilistic planning techniques (such as MDPs and POMDPs) in terms of efficiency, *i.e.*, despite computing a policy for an MDP or POMDP in many real world scenarios has a high computational cost, this step is usually performed only once during the planning process. On the other hand, a pure deterministic planning system might be computationally more expensive, in the long term, as larger and more uncertain problems are addressed, given that the amount of re-planning steps will increase.

Moreover, having a compact representation of the search space (such as hierarchies) for planning systems that are constantly requested to generate plans for different tasks turns to be essential. However, from the reviewed works that use decision-theoretic planning, a subset of them employ a hierarchical structure of the problem in an effort to keep the planning search space as compact as possible. Furthermore, from those that implement both decision-theoretic and hierarchical techniques in their architecture, only two of them, [Hanheide et al., 2017] and [Sridharan et al., 2018] which are extensions to [Hanheide et al., 2011] and [Zhang et al., 2014] respectively, are intended to serve as general purpose planning systems (within the service robotics realm), *i.e.* they propose architectures that are able to model other skills that were not shown in their study cases or experiments.

Regarding the two most related works, the assumption they do about the completeness of their knowledge about the world sets our architecture closer to one of them than the other. While [Hanheide et al., 2017] assume an open world, *i.e.* they do not expect to have a description of the entire environment in any form, and thus use assumptions to guide the planning process, on the other hand, [Sridharan et al., 2018] do assume that the domain description stored in the knowledge base, at some resolution, encompasses the whole environment. However, given that so far no works have been found that indicate that planning for closed worlds is a sub-problem of doing so for open worlds, we consider them as two variants of task planning, and therefore regard [Sridharan et al., 2018] as the most closely related work to ours.

While our proposal shares a fair amount of similarities with [Sridharan et al., 2018], there are some key differences, in which our architecture takes into consideration aspects related to the size of the environment; such features are described below:

- When available, the proposed architecture uses relations among classes of objects to automatically build deep hierarchical representations of the state space, while the architecture in [Sridharan et al., 2018] is restricted to a two level hierarchical representation. As shown by one of our experiments, hierarchical planning systems with deeper representations seem to scale better in terms of planning time, which is an important advantage of our architecture.

- In the proposed architecture, a recursive definition for abstract actions is proposed, which exploits hierarchical structures of the environment to decompose the task planning problem, by automatically building a hierarchy of POMDPs from an initial POMDP and a hierarchy that abstracts its state space. While in [Sridharan et al., 2018] abstract actions are only invoked in the coarse level, and employed to execute concrete actions, in our architecture abstract actions can either invoke concrete or abstract actions. Hence, as shown by experimental results, our architecture is able to achieve higher success ratio scores (in problems with large state spaces) than a hierarchical planning system with two levels (such as the one proposed by [Sridharan et al., 2018]), by distributing the computational burden of planning among the models that constitute the hierarchy of POMDPs.

Since we do not have access to the code for the implementation of [Sridharan et al., 2018], nor they evaluated their architecture in a public benchmark problem, we implemented a hierarchical planning system in order to replicate theirs as close as possible. Despite such implementation is not an exact copy, it does meet two of the main aspects we are interested in: i) at coarse level deterministic planning is performed, and ii) it models and executes each action in the coarse level as a POMDP. Therefore, this implementation and a standard POMDP are employed as baseline to compare the performance of the proposed architecture, and evaluate how much impact it is to use deeper hierarchies, in terms of efficiency and effectiveness.

3.4 Chapter Summary

In this chapter, it has been analyzed the most related research (to the proposed architecture) on task planning for service robotics, which address the problem within

a wide variety of approaches that range from deterministic planning, decision theoretic planning, commonsense reasoning, combination of the previous three, and hierarchies of MDPs and POMDPs using state or action abstraction.

Furthermore, by comparing it with the most closely related works ([Sridharan et al., 2018, Hanheide et al., 2017]) it has been established that the proposed architecture, which combines several forms of domain knowledge to plan for service robotics applications, differs from others in that it is capable of constructing arbitrarily deep hierarchies of POMDPs, while it simultaneously enables the integration of several skills into the same planning problem without having to modify skills that are already part of the architecture. In this sense, the problems of large environments, uncertainty, partial observability and task diversity are addressed within a single framework in a way that has not been found in the reviewed literature so far.

Chapter 4

Proposed method

4.1 General overview

The proposed architecture for task planning is based on the idea that humans have information about structures present in their indoor environments, that follow certain organization scheme and are valid for most of the indoor scenarios. Furthermore, having knowledge of how objects are organized, *e.g.* spatially, temporarily or causally, can be exploited for planning given that it would help to mitigate the uncertainty on the outcomes from events. In task planning, whether it is deterministic or probabilistic, the fewer amount of actions that can be executed in a single state, the better. In other words, since the amount of possible transitions among the set of all states constitutes the size of the search space in a planning problem, if there is information about which transitions are inconsistent to the description of a planning problem in particular, this information could be leveraged to decrease the size of the search space and, consequently, reduce the time required to find a successful plan.

Thus, in order to mitigate the burden that large search spaces represent, the proposed architecture addresses the task planning problem in service robotics by integrating a knowledge representation scheme and a probabilistic planning model. The knowledge representation is employed to capture the available knowledge about the environment, and is exploited to probabilistically generate and execute plans.

The methodology followed by the proposed architecture, which is summarized in Fig. 4.2, is segmented in three main phases: a) knowledge base construction, b) architecture's initialization and c) architecture's operation. In the first phase, a human is required to endow the architecture with information about the robot's skill sets and the environment in which it will operate. This information is encoded into the knowledge base using a scheme that encompasses features of the domain that are necessary to describe the dynamics of the environment. Next, during initialization, the architecture builds a hierarchy of actions that will be invoked during the operation phase in order to execute plans.

For instance, in the example of a navigation domain scenario shown in Fig. 4.1, a service robot is asked to go to specific locations in the environment, which is discretized into twelve possible locations (cells). To accomplish this type of tasks, the designer should endow the architecture with a basic module (see section 4.2.1.1) that describes the robot's navigation skill set and a hierarchical description of the environment (see section 4.2.1.3), which in this example four levels of granularity are employed: cell, section, room, and building. This basic module should include actions that enable the robot to transit between cells and monitor its location. Thus, the architecture could use the description of the environment and the robot's actions to generate plans that solve specific task requests, such as going to cell **C5** or to room **R3**. For illustration purposes, this example scenario is revisited along this chapter as each one of the three phases are described in detail.

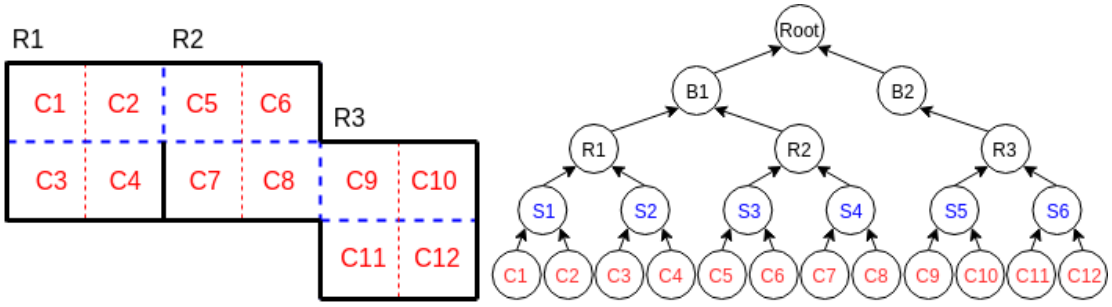

Figure 4.1: Example of a navigation domain constituted by twelve cells (C). As the hierarchical description of the environment in the right shows, the cells are grouped (abstracted) into six sections (S), which are also grouped into three rooms (R), that in turn are grouped into two buildings (B). This hierarchical description enables the proposed architecture to decompose planning problems, so they can be solved as a set of smaller tasks.

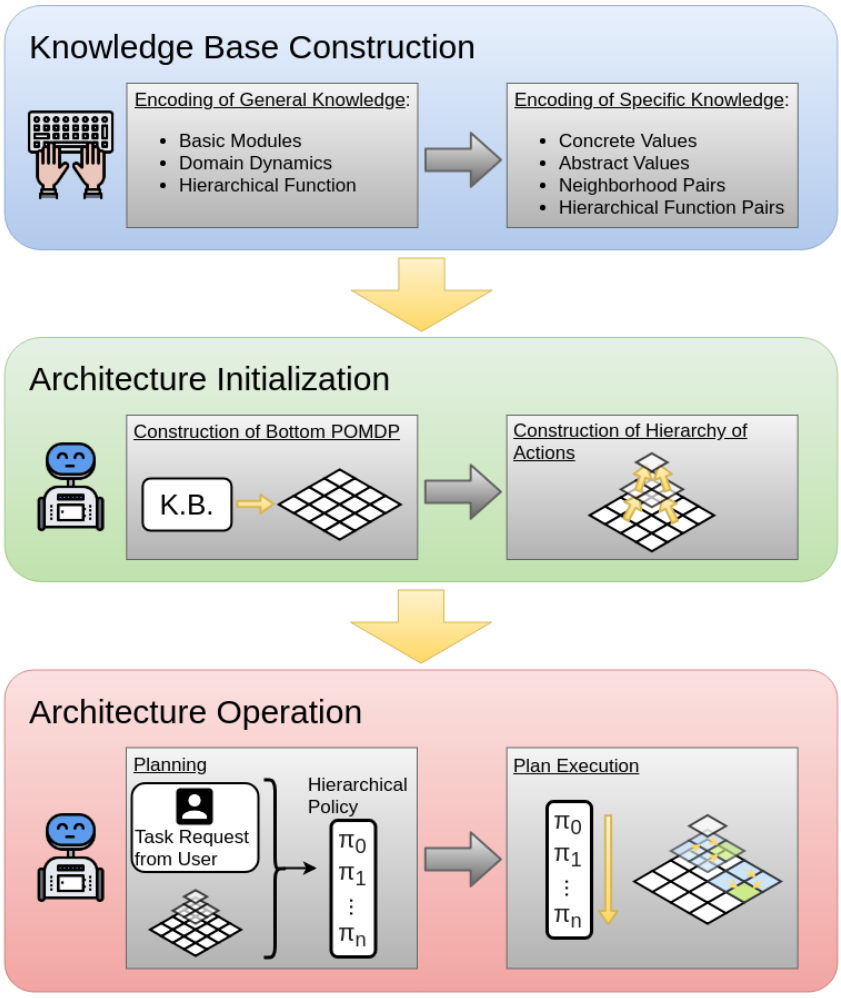

Figure 4.2: The proposed architecture follows a three phase methodology to solve task planning problems. First, a human designer encodes general information about the robot's domain, as well as about the specific environment in which the robot will operate. Next, the architecture initializes by building a POMDP that models the environment at a concrete level, and builds a hierarchy of POMDPs from it. Both phases need to be performed only once, whereas the third phase will be carried out every time the robot receives a task request. It consists of two steps, first, based on a task request and the hierarchy of POMDPs, a hierarchical policy is built as a sequence of POMDP policies $(\pi_0, ..., \pi_n)$ to model a plan for the respective task. Finally, the policies in the hierarchical policy are executed in a top-down way to gradually bring the agent closer to the goal state that represents the solution to the task at hand.

4.2 Knowledge base construction

During the knowledge base construction, the information related to the domain, within which the robot will operate, is encoded and organized following a criterion of specificity. The knowledge base is constituted by two main blocks of information: general and specific knowledge. The general knowledge stores information associated to the robot's skill sets, *i.e.* the aspects of the environment it is capable of modifying and perceiving through its actuators and sensors. On the other hand, specific knowledge represents facts that are true for a particular environment and may not hold for other scenarios. Hence, the criterion of specificity allows the architecture to reuse a knowledge base in different environments and avoid constructing it from scratch, by redefining the specific knowledge component, so that it matches the new scenario. As for the representation language, SPARC [Balai et al., 2013a] is employed to construct the knowledge base as a system description (section 2.1.3) defined by a sorted ASP program containing a collection of sorts, relations, rules and facts.

4.2.1 General knowledge

The general knowledge is constituted by three parts that altogether define the system description for the actions a robot can perform. First, for each skill set, a basic module description will contain the list of actions the robot can perform by means of such skills, the set of variables these actions modify, the observations that can be perceived after a variable is modified, as well as the probability distributions of possible outcomes for each action. Second, a set of rules that define the actions' effects, under which situations they cannot be executed, and inherent constraints of the domain. Finally, a function that describes the hierarchical structure in which one of the variables could be abstracted.

4.2.1.1 Basic modules

A basic module is defined by six components: i) four sets (actions, state variables, variable value sets and variable observations sets) and ii) two probability distribution functions (a transition and an observation function). In order to integrate a basic

module into the knowledge base, all six components must be specified by the designer in the form of SPARC sorts and statements, each one of them is described in detail below.

- **Actions**: Set of actions the module can perform, where each action must modify at most one state variable. Actions are listed in the $\#action$ sort.

- **State variables**: Set of variables that can be modified by at least one of the module's actions. Each variable is specified as an inertial fluent and must be in the $\#inertial\_fluent$ sort.

- **State variable values**: For each state variable, a unique sort (for instance $\#var0\_values$) must be defined. This sort will list the set of values the variable can hold. If a set of values depends on the particular environment, then the elements of its sort must be specified when the specific knowledge is encoded.

- **State variable observations**: For each state variable, a unique sort (for instance $\#var0\_obs$) must be defined. This sort will contain the set of observations that can be perceived when an action that modifies the state variable is performed. Similar to the variable's values, if the set of observations depends on the particular environment, then it must be specified during the encoding of the specific knowledge.

- **Transition probability distribution**: For each action and each value its respective variable can take, a probability distribution must be provided to describe the action's transition probabilities. The distributions can specify the probabilities for particular state transitions, *e.g.* $< door\_is\_open, close\_door, door\_is\_closed, 0.65 >$, or transitions defined by a neighborhood relation, *e.g.* $< move\_to\_left, at\_left\_of, 0.65 >$, where $move\_to\_left$ is an action and at_left_of is a binary relation defined over the set of values the variable can take. That is, the relation-based definition specifies a transition $< X, move\_to\_left, Y, 0.65 >$ for every pair (X, Y) such that $(X, Y) \in at\_left\_of$ (neighborhood relations are described in section

4.2.1.2).

- **Observation probability distribution**: For each action and each value its respective variable can take, a probability distribution must be provided to describe the action's observation probabilities. Similar to transition distributions, an observation distribution can specify probabilities for tuples with particular state-observation values, or for tuples defined with observation neighborhood relations, *e.g.* $< scan, ObservableWith3D, 0.55 >$ would specify the probability of 0.55 for the observation tuple $< X, scan, Y, 0.55 >$ for every state-observation pair (X, Y) such that $(X, Y) \in ObservableWith3D$ (observation neighborhood relations are described in section 4.2.1.2).

The transition and observation probability values must be provided by the designer of the knowledge base. Since these probabilities correspond to events associated to a single variable, one could estimate them by performing a series of experiments in which actions are executed several times, and the resulting state and observation of each trial is registered, as suggested by [Sridharan et al., 2018].

Recalling the example scenario from Fig. 4.1, the description in SPARC of a basic module for navigation would be specified as:

$$\#cell = c[1..12].$$
$$\#loc\_obs = \#cell.$$
$$\#loc\_values = \#cell.$$
$$\#action = \{up, down, left, right\}.$$
$$\#inertial\_fluent = loc(\#loc\_values).$$

Whereas the transition and observation distributions would be modeled by neighborhood relations as:

- Transition distributions

$$up = \{< up, above, 0.9 >, < up, current\_cell, 0.1 >\}$$
$$down = \{< down, below, 0.9 >, < down, current\_cell, 0.1 >\}$$
$$left = \{< left, at\_left, 0.9 >, < left, current\_cell, 0.1 >\}$$
$$right = \{< right, at\_right, 0.9 >, < right, current\_cell, 0.1 >\}$$

- Observation distributions

$$\begin{aligned} ACTION = \{ & < ACTION, above, 0.1 >, \\ & < ACTION, below, 0.1 >, \\ & < ACTION, at\_left, 0.1 >, \\ & < ACTION, at\_right, 0.1 >, \\ & < ACTION, current\_cell, 0.6 > \} \end{aligned}$$

  where the transition and observation probability values are given by the designer of the knowledge base, and the method employed to compute such probabilities is out of the scope of this research. Furthermore, since the observation distribution in this example is the same for every action, just substitute the actual action name instead of $ACTION$ to obtain its distribution.

This basic module description indicates that the agent can perform four actions to modify the state variable *loc*. The values and observations for this state variable are the cells in the environment, whereas the transition distribution of every action has two possible ending states: staying in the same cell and reaching the target cell. The observation distribution is the same for the four actions, and consists of a 4 connected neighborhood. Moreover, from the transition and observation distributions employed in this example, one can tell that actions have a considerable chance of success (0.9), while the precision of the robot's sensing device seems to be relatively good, as it has a probability of 0.6 of returning the correct observation.

4.2.1.2 Domain dynamics

The domain dynamics description encompasses the relations that characterize the structure of an environment, as well as the rules that define the stochastic effects of

actions, inherent domain constraints and the scenarios under which actions cannot be executed. Encoded as SPARC statements (see section 2.1.3), The domain dynamics description is constituted by five components: neighborhood relations, deterministic causal laws, non-deterministic causal laws, state constraints and executability conditions. This section starts by introducing the definitions of neighborhood relation and observation neighborhood relation, followed by the description of its components.

Definition 1 (Neighborhood relation) *Let A_i be an action defined in a basic module, V_j the state variable action A_i can modify, $Val_j = \{v_0, ..., v_k\}$ the set of values V_j can take, and $N(A_i)$ a binary relation defined over Val_j, then we call $N(A_i)$ the neighborhood relation of action A_i.*

Definition 2 (Observation neighborhood relation) *Let A_i be an action defined in a basic module, V_j the state variable action A_i can modify, $Val_j = \{v_0, ..., v_k\}$ the set of values V_j can take, $Obs_j = \{o_0, ..., o_l\}$ the set of observations that can be perceived after modifying V_j, and $N(A_i)$ a binary relation defined over $Val_j \times Obs_j$, then we call $N(A_i)$ the observation neighborhood relation of action A_i.*

It is worth noting that it is not necessary for neighborhood relations to have any property other than being binary, given that their purpose is to model the connectivity between values for a given action. For example, to specify the transition and observation distributions for the actions at the end of section 4.2.1.1, every neighborhood relation is defined over the sort *#cell*, since both values and observations are defined by this sort. Hence, the domain dynamics components are presented.

- **Neighborhood relation**: For each action A_i in each basic module, define a neighborhood relation that represents how the value of its respective variable V_j changes when A_i is performed, in other words, which pairs of values from Val_j are neighbors by means of executing A_i. In a similar fashion for observation neighborhood relations, one must specify the relations that associate values with observation given an action. Also, the pairs of particular values that comply a given neighborhood relation are specified in the specific knowledge. The relation must be defined in a SPARC program as an element of the *#static_fluent* sort, specifying the sorts it receives as arguments. For

instance, to specify the relation *above* in the navigation scenario, one would use the following SPARC code:

$$\#static\_fluent = above(\#cell, \#cell).$$

- **Deterministic causal law**: For each action in each basic module, a deterministic causal law (see section 2.1.3) is defined in SPARC to model the effect such action would have in a deterministic environment. Let ai, $nai(\#val, \#val)$, and $varj(\#val)$ be the SPARC definitions of an action, its neighborhood relation, and its variable, respectively. Also, let $X1$ and $X2$ be variables that can be substituted by elements of the sort $\#val$, I be a variable that represents the instant of time at which a fluent has certain value or an action occurs and n be the last instant of time included in the SPARC program (see section 2.1.3 for the encoding of statements). Then, the deterministic causal law for action ai is defined by

$$\begin{aligned} h(varj(X2), I+1) \leftarrow\ & h(varj(X1), I), \\ & h(nai(X1, X2), I), \\ & occurs(ai, I), \\ & I < n. \end{aligned} \tag{4.1}$$

  For instance, to specify the deterministic causal law of the action *up*, one would use the SPARC code presented by Eq. 4.6, that states that such action will change the robot's location to the cell that is above of its current position. Moreover, the meaning of variables $X1$, $X2$, I and n presented for Eq. 4.1 is the same for Eqs. 4.2, 4.3, 4.4, 4.5 and 4.6, while in Eqs. 4.7, 4.8, 4.9, 4.10 and 4.11 the SPARC variables $X1$ and $X2$ are substituted by elements from sort $\#cell$.

- **Non-deterministic causal law**: Similar to its deterministic counterpart, a non-deterministic causal law is intended to model the stochastic effects of an action, based on several neighborhood relations, where the set of relations must be specified by the designer, based on its knowledge of the domain. Thus, let ai, $nai(\#val, \#val)$, $varj(\#val)$, $aif(\#val)$, and $X3$ be the SPARC definitions of an action, its neighborhood relation, its variable, the defined fluent that

describes the set of possible outcomes, and a SPARC variable that can be substituted with an element from sort $\#val$, then the non-deterministic causal law for action ai is defined by

$$\begin{aligned} 1\{h(varj(X3), I+1) \ : \ h(aif(X3), I)\}1 \leftarrow & \ h(varj(X1), I), \\ & \ h(nai(X1, X2), I), \\ & \ occurs(ai, I), \\ & \ I < n. \end{aligned} \tag{4.2}$$

where, the defined fluent $aif(\#val)$ is defined by a collection of SPARC rules of the form

$$\begin{aligned} h(aif(X2), I) \leftarrow & \ h(varj(X1), I), \\ & \ h(nak(X1, X2), I). \end{aligned} \tag{4.3}$$

where $nak(\#val, \#val)$ is a neighborhood relation defined over values in sort $\#val$. Thus, if the transition and observation distributions are defined with neighborhood relations, those relations must be included in the definition of fluent $aif(\#val)$ (for a description of the syntax of non-deterministic rules in SPARC, see the description of Eq. 2.7). Furthermore, Eq. 4.8 presents an example that specifies the non-deterministic causal law of action *up*, which includes the current cell and the one above it as possible outcomes. That is, Eq. 4.8 incorporates the possibility of having the robot to drift in its current position when attempting to move upwards.

- **State constraints**: State constraints are used to model inherent properties of the domain that are always true, regardless of the particularities of an environment. Thus, let $var(b)$, and $\{vari(c), ..., vark(d)\}$ be the SPARC code for an inertial fluent, and an arbitrary large set of inertial fluents with particular values, then a state constraint that states $var(b)$ will be true anytime the set $\{vari(c), ..., vark(d)\}$ is true is defined by

$$\begin{aligned} h(var(b), I+1) \leftarrow & \ h(vari(c), I), ..., \\ & \ h(vark(d), I). \end{aligned} \tag{4.4}$$

and the same can be done for properties that are false, that is

$$\begin{aligned} \neg h(var(b), I+1) \ \leftarrow & \ h(vari(c), I), ..., \\ & \ h(vark(d), I). \end{aligned} \tag{4.5}$$

For instance, the fact that a robot cannot be at two different locations simultaneously should be encoded as state constraints in a domain where the robot's spatial location is relevant for planning (in Eq. 4.10 this restriction is specified for the navigation scenario of Fig. 4.1).

- **Executability conditions**: Executability conditions state under which scenarios an action cannot be performed. Therefore, for each action a in each basic module, a collection of SPARC rules of the following form, specify conditions in which a cannot be executed.

$$\begin{aligned} \neg occurs(a, I) \ \leftarrow & \ h(vari(c), I), ..., \\ & \ h(vark(d), I) \end{aligned} \tag{4.6}$$

where $\{vari(c), ..., vark(d)\}$ is the set values that prohibit the execution of a. Furthermore, the designer can define as many executability conditions are required for each action. For example, Eq. 4.11 states the restriction that the robot cannot perform more than one action at a time.

In regards to our example from Fig. 4.1, the domain dynamics description would look like:

- Neighborhood relations

$$\begin{aligned} \#static\_fluent = & above(\#cell, \#cell) + \\ & below(\#cell, \#cell) + \\ & at\_left(\#cell, \#cell) + \\ & at\_right(\#cell, \#cell) + \\ & current\_cell(\#cell, \#cell). \end{aligned}$$

where these five relations work for values and observations, since both are defined by the set of cells.

- Deterministic causal law

$$\begin{aligned} h(loc(X2), I+1) \leftarrow\ & h(loc(X1), I), \\ & h(above(X1, X2), I), \\ & occurs(up, I), \\ & I < n. \end{aligned} \tag{4.7}$$

The deterministic causal law for the other three actions is similar to this one, what changes is the neighborhood relation used in the body of the rule, for instance, for action *down* the *below* relation would be used.

- Non-deterministic causal law

$$\begin{aligned} 1\{h(loc(X3), I+1)\ :\ h(up\_df(X3), I)\}1 \leftarrow\ & h(loc(X1), I), \\ & h(above(X1, X2), I), \\ & occurs(up, I), \\ & I < n. \end{aligned} \tag{4.8}$$

where the defined fluent *up_df* specifies the possible outcomes, that must be consistent to the ending states in the action's transition distribution specified in the basic module. Thus, *up_df* is specified with the following rules:

$$\begin{aligned} h(up\_df(X2), I) \leftarrow\ & h(loc(X1), I), \\ & h(above(X1, X2), I). \\ h(up\_df(X2), I) \leftarrow\ & h(loc(X1), I), \\ & h(current\_cell(X1, X2), I). \end{aligned} \tag{4.9}$$

That is, Eq. 4.9 specifies as possible outcomes the cell above and the current cell, as it is plausible for the robot to reach the cell above with action a, as it could also drift and stay in its current cell. Thus, the non-deterministic causal law for the other actions could be specified by substituting the action and the *above* neighborhood relation by its respective relation. Moreover, Eqs. 4.8 and 4.9 also define observation distributions, with the difference that a set of observation neighborhood relations is employed in the definition of the defined fluent.

- State constraint

$$\begin{aligned} \neg h(loc(X2), I) \ \leftarrow & \ h(loc(X1), I), \\ & X1 \,!= \, X2. \end{aligned} \tag{4.10}$$

  This state constraint says that it is impossible for the robot to be at several locations simultaneously.

- Executability conditions

$$\begin{aligned} \neg occurs(X2, I) \ \leftarrow & \ occurs(X1, I), \\ & X1 \,!= \, X2. \end{aligned} \tag{4.11}$$

  This condition states that the robot cannot execute several actions at once.

4.2.1.3 Hierarchical function

In order for the architecture to build a hierarchy of concrete and abstract actions, it is necessary to abstract the state space into a hierarchical representation. Therefore, a function that establishes the links between objects, to build a hierarchy, must be defined. Let E be the set containing the values a state variable var_i can have, as well as the more abstract form of those values, then a function $F : E \rightarrow E$ is said to abstract the state space with respect to variable var_i, if for every input argument x then $F(x)$ returns a more abstract form of x. In this way, F represents a hierarchy whose leaf nodes are the values var_i can take.

In SPARC, a hierarchical function is specified as a static fluent with two input arguments, that is, $hf(\#h\_sort, \#h\_sort)$, where $\#h\_sort$ is the sort defined as the union of the sorts defined for each level of the hierarchy. For instance, in the navigation example from Fig. 4.1, a hierarchical function that represents the relation of *is in* between locations at different levels of resolution would be defined as

$$\#static\_fluent = is\_in(\#location, \#location)$$

where

$$\#cell = c[1..12].$$
$$\#section = s[1..6].$$
$$\#room = r[1..3].$$
$$\#building = b[1..2].$$
$$\#location = \{root\} + \#building + \#room + \#section + \#cell.$$

These sort definitions correspond to the example scenario presented in Fig. 4.1, which includes the twelve cells that are distributed in six sections, three rooms and two buildings.

4.2.2 Specific knowledge

As aforementioned, the specific knowledge refers to a collection of facts that are true for a particular environment in which the robot will operate. In the knowledge base, the designer must specify four types of facts: particular values for state variables, abstract forms of particular values, pairs of values that are in a neighborhood relation and pairs of values that are in the hierarchical function.

- **Concrete values**: For each state variable V_i whose set of values was not specified in the definition of the basic module, the designer must provide the set of values Val_i variable V_i can take within the particular environment, as well as the set Obs_i of observations that can possibly be perceived when variable V_i is modified.

- **Abstract values**: Values that represent the internal nodes in the hierarchical representation of the state space.

- **Neighborhood relation pairs**: For each neighborhood relation $N(A_i)$ defined over the set of values Val_j, every pair of values from Val_j that are in $N(A_i)$ must be specified.

- **Hierarchical function pairs**: For every value in the set that defines the domain of the hierarchical function F, specify its parent value as a pair (a, b) where $F(a) = b$.

For the navigation example, the concrete values are specified by the list of cells, whereas the abstract values are the lists of sections, rooms and buildings. The pairs of values that are in the neighborhood relation in this example are every pair of cells that comply with any of the five neighborhood relations defined in the domain dynamics, for instance, $current\_cell(c1, c1)$, $below(c1, c3)$, $at\_right(c1, c2)$. With regards to the hierarchical function pairs, in this scenario the following pairs should be included: $is\_in(c1, s1)$, $is\_in(s1, r1)$, $is\_in(r1, b1)$, $is\_in(r2, b1)$, and so on.

Once the specific knowledge of a particular environment has been encoded, the architecture is able to build the stochastic transition diagram that describes how the system may change after the agent performs an action. The next section starts by introducing the construction of the POMDP that will model the environment at the bottom level in the hierarchy of actions.

4.3 Architecture initialization

In this section, the procedure followed to initialize the architecture is presented. The initialization sets the architecture ready to receive task requests and solve them, and consists of two main steps: a) building the bottom POMDP, and b) building the hierarchy of actions. After specifying a base POMDP (section 4.3.1) and a hierarchical representation for its state space (section 4.3.2.1), a recursive formulation is employed to build a hierarchy of abstract actions (section 4.3.2.3).

4.3.1 Construction of bottom POMDP

In order to build a hierarchy of actions, it is necessary to first have a POMDP built from which the hierarchy will start its construction (that will be referred to as base POMDP or bottom POMDP indistinctly). In section 2.2.2 it is mentioned that a POMDP is defined by a tuple $M =< S, A, \Phi, R, O, \Omega, B_0 >$, however, since the purpose of the base POMDP is to describe the dynamics of the environment and not to model a particular planning problem, its construction consists in specifying all of the parameters in M except the reward function R and the initial belief distribution B_0.

The construction of the bottom POMDP takes place by using the sets of values, observations and actions, specified in the basic modules and specific knowledge, to define the S, A, and O parameters, while Φ and Ω are built from the probability distributions specified for each action. The details on the definition of each parameter are presented below.

- S: Let $V = \{v_0, ..., v_i, ..., v_n\}$ be the set containing the set of values v_i of every state variable defined in a basic module, then, the set of states S of the bottom POMDP is defined by the cross product of every element in V, that is

$$S = v_0 \times ... \times v_i \times ... \times v_n \tag{4.12}$$

- A: Let $B = \{b_0, ..., b_m\}$ be the set of all basic modules defined in the knowledge base, and $a(b_i)$ the set of actions defined in basic module b_i, then the set of actions A for the bottom POMDP is defined by the union of all the sets of actions defined in each basic module, that is

$$A = \bigcup_{b_i \in B} a(b_i) \tag{4.13}$$

- O: Let $B = \{b_0, ..., b_m\}$ be the set of all basic modules defined in the knowledge base, $sv(b_i)$ the set of state variables defined in basic module b_i, and $o(sv)$ the set of observations defined for state variable sv, then the set of observations O for the bottom POMDP is defined by the union of all the sets of observations defined in each basic module, that is

$$O = \bigcup_{b_i \in B} \bigcup_{sv \in sv(b_i)} o(sv) \tag{4.14}$$

- Φ: Let $B = \{b_0, ..., b_m\}$ be the set of all basic modules defined in the knowledge base, $a(b_i)$ the set of actions defined in basic module b_i, $t(a)$ the set of transition probability distributions of action a, p the probability of transiting from state s_j to s_k after executing a, and $s \in S$ a state of the bottom POMDP defined as an n-tuple, where the l-th element of s, $s[l]$, is a value for the l-th state variable. Then the transition function Φ for the bottom POMDP is defined by the set of every tuple $< s_j, a, s_k, p >$ that satisfies one of the following cases:

1. **Case 1**: There is a state transition $t \in t(a)$, specified as a particular-value transition, such that $t =< s_j[l], a, s_k[l], p >$, where p is the probability of transiting to value $s_k[l]$ from $s_j[l]$ after executing a, $b_i \in B$ is a basic module, $a(b_i)$ returns the set of actions defined in b_i, and $a \in a(b_i)$ is an action that modifies the l-th state variable.

2. **Case 2**: There is a state transition $t \in t(a)$, specified as a neighborhood defined transition, such that $t =< a, N, p >$, where $(s_j[l], s_k[l]) \in N$, p is the probability of transiting to value $s_k[l]$ from $s_j[l]$ after executing a, $b_i \in B$ is a basic module, $a(b_i)$ returns the set of actions defined in b_i, $a \in a(b_i)$ is an action that modifies the l-th state variable, and N is a neighborhood relation defined over the set of values for the l-th state variable.

- Ω: Let $B = \{b_0, ..., b_m\}$ be the set of all basic modules defined in the knowledge base, $a(b_i)$ the set of actions defined in basic module b_i, $z(a)$ the set of observation probability distributions of action a, $s \in S$ a state of the bottom POMDP defined as an n-tuple, where the l-th element of s, $s[l]$, is a value for the l-th state variable, and $o \in O$ an observation of the bottom POMDP. Then the observation function Ω for the bottom POMDP is defined by the set of every tuple $< s_j, a, o_k, p >$ that satisfies one of the following cases.

  1. **Case 1**: There is an observation transition $z \in z(a)$, specified as a particular-value transition, such that $z =< s_j[l], a, o_k, p >$, where p is the probability of perceiving o_k when value $s_j[l]$ is reached after executing a, $b_i \in B$ is a basic module, $a(b_i)$ returns the set of actions defined in b_i, and $a \in a(b_i)$ is an action that modifies the l-th state variable.

  2. **Case 2**: There is an observation transition $z \in z(a)$, specified as a neighborhood defined transition, such that $z =< a, N, p >$, where p is the probability of perceiving o_k when value $s_j[l]$ is reached after executing a, $(s_j[l], o_k) \in N$, $b_i \in B$ is a basic module, $a(b_i)$ returns the set of actions defined in b_i, $a \in a(b_i)$ is an action that modifies the l-th state variable, and N is an observation neighborhood relation defined over the set of values and observations for the l-th state variable.

Hence, the tuple $BP =< S, A, O, \Phi, \Omega >$ models the environment according to the domain knowledge encoded in the knowledge base. This tuple will serve as starting point for the construction of the hierarchy of abstract actions.

4.3.2 Hierarchy of actions

From the bottom POMDP, a hierarchy of POMDPs is built in a bottom-up approach. At every level in the hierarchical description for the bottom POMDP's state space, starting at the bottom level going all the way to the top, POMDPs are modeled over subsets of the full state space to transit to neighbor subsets. Besides enabling the system to transit between sub-regions of states, for each POMDP a pair of probability distributions (transition and observation) are computed so that they can be employed as an action in a POMDP at the immediate upper level.

4.3.2.1 State space tree

For a given hierarchical structure that describes a particular environment, in order for it to be employed it must abstract the state space in disjoint subsets of states, whose union results in the complete state space, at every level in the hierarchical structure. If this requirement is met, the architecture employs such hierarchical representation, which we call *State Space Tree* (SST), see Fig. 4.4 which corresponds to the SST of the building shown in Fig. 4.3. Within this representation, as explained later in section 4.4.1.1, one can define a *Relevant Sub-Space* (RSS) to bound a sub-region from the original state space, in order to reduce (at the operation phase) the size of the state space that needs to be considered to solve a specific task. The building process of an SST for a state space using a hierarchical function F is described below.

Let F be the hierarchical function that performs abstraction with respect to the i-th state variable, and S the set of states of the bottom POMDP, where each state s is defined as an n-tuple and the i-th element $s[i]$ is a value for the i-th state variable. For every $s \in S$, the parent state $P(s)$ of s in the SST hierarchy is the resulting n-tuple from setting the i-th element of s with $F(s[i])$. After creating the set parent

states $P(S)$, the process is repeated to generate $P(P(S)), ...$ and so on until the level at which the root of the hierarchy is located is reached. Furthermore, neighborhood relations for states in the bottom POMDP are propagated upwards in the SST.

Definition 3 (Neighbor states) *Let $N(A_i)$ be the neighborhood relation of action A_i, V_j the state variable action A_i can modify, $States(P)$ and $Actions(P)$ the sets of states and actions in POMDP P, and $s \in States(P)$ be a state of a POMDP P, defined as a tuple $s =< v_0, ..., v_k, ..., v_n >$ where $s[k] = v_k$ is the value for the k-th state variable defined in the knowledge base. For a pair of states $s_k, s_l \in States(P)$, if there is an action $A_i \in Actions(P)$ such that $(s_k[j], s_l[j]) \in N(A_i)$ or $(s_l[j], s_k[j]) \in N(A_i)$, then s_k and s_l are said to be neighbor states.*

Once every state has a parent state assigned in the SST, by means of the hierarchical function, neighborhood relations are propagated as follows: for every pair of neighbor states, if they have different parents in the SST, then their parents become neighbors. However, in order to build an SST a single variable is employed to perform state abstraction, therefore, if the designer can provide more than one hierarchical function, only one of them can be used at a time.

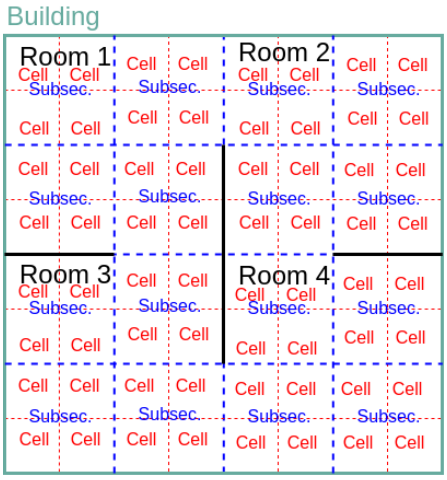

Figure 4.3: Hierarchical description of a navigation domain environment, with four levels of resolution: buildings, rooms, subsections, cells (from top to bottom in the hierarchy). This particular is constituted by a single building with four rooms, each with four subsections which in turn have four cells each.

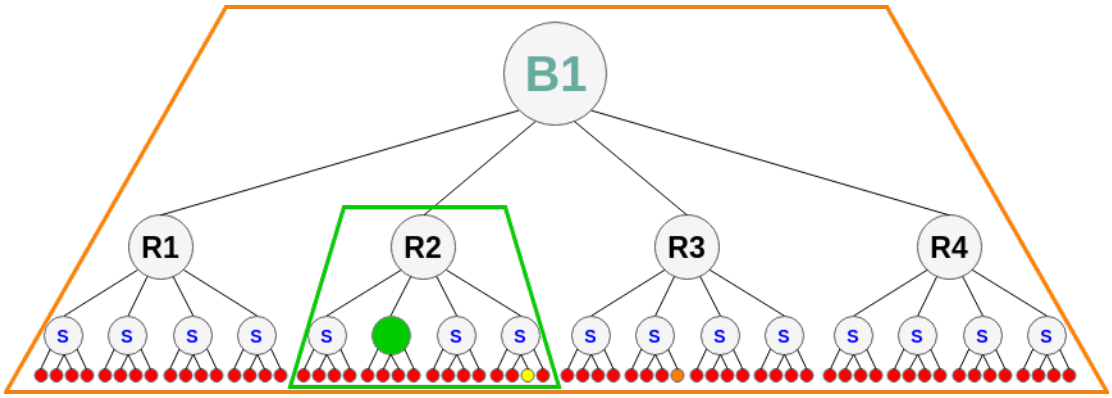

Figure 4.4: Hierarchical representation of the environment depicted in Fig. 4.3, in the form of a *State Space Tree*. The polygons enclose the *Relevant State Space* for different pairs of initial and goal states. The initial state is depicted by the yellow leaf node, whilst the green and orange nodes represent two different goal states, located at different resolutions.

4.3.2.2 Concrete and abstract components

After the state space has been abstracted into a hierarchy, resulting in the environment's SST, the architecture proceeds to build abstract actions that will serve to transit the agent between abstract states. However, before the method for the construction of abstract action is presented, the differences between concrete and abstract components are introduced. As states are used to model the possible scenarios the agent might encounter in the environment, in a hierarchical representation, there are two types of states.

- **Concrete states**: Concrete states have the highest resolution possible, that is, is the finest representation in which the agent can reason about the state space.

- **Abstract states**: Abstract states are constituted by a collection of states with higher resolution, which are its children in the SST, that can either be concrete or abstract states. Abstract states can be seen as clusters of states with a higher resolution.

The dichotomy between concrete and abstract actions depends on whether the computation of a policy is required to model them.

- **Concrete actions**: Concrete actions are executed from concrete states, and generate concrete observations.

- **Abstract actions**: Abstract actions are modeled as POMDP policies, and invoked from abstract states.

On the other hand, the concept of clustering elements of a higher resolution to form more abstract ones is not transferable to observations, instead, these are modeled as follows.

- **Concrete observations**: A concrete observation o is associated with concrete states and actions; having a probability $\Omega(s, a, o)$ greater than zero of being perceived for at least one pair (s, a), where s and a are a concrete state and action, respectively.

- **Abstract observations**: Abstract observations are defined over each level of the SST, assigning an abstract observation for each abstract state, in order to model the partial observability at any level in the hierarchy of the state space. For more detail on how abstract observations are defined, see section 4.3.2.3.

4.3.2.3 Modeling abstract actions

In order to model an abstract action as a POMDP within a hierarchy of actions, it is necessary to define the set of parameters that characterize the POMDP, as well as those required at a higher level in the hierarchy to incorporate it as an action in another POMDP. This section starts by introducing the formulation used to define a POMDP that models a specific abstract action at a given depth in the hierarchy, followed by a definition for abstract action's parameters required by POMDPs at a higher level, and ends with the algorithm used to build the hierarchy of actions, based on the definition of abstract action.

Let s_i^d and s_j^d be two abstract states located at depth d in a SST and $C(s)$ the set of states that are children of state s in the SST. An abstract action a_{ij}^d designed to transit from state s_i^d to s_j^d is modeled as a POMDP using the following formulation:

1. S_{ij}^d: Set of states from the immediate lower level in the SST that are relevant for abstract action a_{ij}^d, defined by

$$S_{ij}^d = \{s \mid s \in C(s_i^d)\} \bigcup \{s \mid s \notin C(s_i^d), \exists s_k \in C(s_i^d), neighbors(s, s_k)\} \quad (4.15)$$

Equation 4.15 says that the POMDP's set of states is the union of the children states of s_i^d and the set of states adjacent to some child of s_i^d that are not part of its children.

2. A_{ij}^d: Set of actions that can be executed at the immediate lower level in the SST that are relevant for abstract action a_{ij}^d.

$$A_{ij}^d = \{a \mid \exists s_k, s_l \in S_{ij}^d, \Phi^{d+1}(s_k, a, s_l) > 0\} \quad (4.16)$$

Equation 4.16 states that A_{ij}^d is made of actions defined at level $d+1$ that have a probability greater than zero of transiting between states that are in S_{ij}^d.

3. O_{ij}^d: Set of observations from the immediate lower level in the SST that are relevant for abstract action a_{ij}^d, defined by

$$O_{ij}^d = \{o \mid \exists s \in S_{ij}^d, \exists a \in A_{ij}^d, \Omega^{d+1}(s, a, o) > 0\} \quad (4.17)$$

Equation 4.17 says that the set of observations is made of those that have a probability greater than zero of being perceived after executing any of the actions in A_{ij}^d and a state contained in S_{ij}^d is reached.

4. Φ_{ij}^d: The transition function simply takes the probability values defined in Φ^{d+1} (transition function of the immediate lower level in the SST) for every pair of states in S_{ij}^d, that is

$$\forall s_k, s_l \in S_{ij}^d, \forall a \in A_{ij}^d, \Phi_{ij}^d(s_k, a, s_l) = \Phi^{d+1}(s_k, a, s_l) \quad (4.18)$$

Furthermore, for every pair of starting state and action that are in S_{ij}^d and A_{ij}^d, respectively, that has a transition probability distribution with ending states that are not in S_{ij}^d, such probability distribution is normalized over the ending states that are in S_{ij}^d.

5. Ω^d_{ij}: The observation function is defined similarly to the transition function. It takes the probability values defined in Ω^{d+1} for all the elements in S^d_{ij}, A^d_{ij}, and O^d_{ij}.

$$\forall s \in S^d_{ij}, \forall a \in A^d_{ij}, \forall o \in O^d_{ij}, \Omega^d_{ij}(s,a,o) = \Omega^{d+1}(s,a,o) \tag{4.19}$$

6. R^d_{ij}: Because the abstract action is designed to transit to a particular abstract state s^d_j, the reward function must model this behavior by assigning a large positive reward to transitions that end in any state contained in $C(s^d_j)$, whilst a large negative reward for transitions that end in states that are not contained in $C(s^d_j)$ nor $C(s^d_i)$, that is, those that end in a children state of a neighbor of s^d_i that is not s^d_j. Hence, the reward function is defined as follows,

$$\forall s \in S^d_{ij} \wedge s \in C(s^d_j),\ R^d_{ij}(\cdot,\cdot,s) = \Re^+ \tag{4.20}$$

$$\forall s \in S^d_{ij} \wedge s \notin C(s^d_i) \bigcup C(s^d_j),\ R^d_{ij}(\cdot,\cdot,s) = \Re^- \tag{4.21}$$

$$\forall s \in C(s^d_i),\ R^d_{ij}(\cdot,\cdot,s) = -1 \tag{4.22}$$

Once the tuple $< S^d_{ij}, A^d_{ij}, O^d_{ij}, \Phi^d_{ij}, \Omega^d_{ij}, R^d_{ij} >$ for an abstract action a^d_{ij} is fully defined, a POMDP solving algorithm is employed to find a policy. It is worth noting that this notation is valid for abstract actions built at any level in the SST, since the recursive formulation uses the components of the immediate lower level in the SST to build the local POMDP (which is defined over a subregion of the state space), regardless of whether they are concrete or abstract.

At this point, the abstract action a^d_{ij} is ready to be invoked, however, because a^d_{ij} is intended to be used as an action by a POMDP in the immediate higher level of the SST, it is necessary to also define the following parameters:

- Set of observations at depth d that can be perceived after a^d_{ij} is executed.

- Transition probability between abstract states at depth d when a^d_{ij} is executed.

- Observation probability of perceiving observations at depth d after a_{ij}^d is executed.

In order for an abstract action to be invoked from a POMDP policy, the parameters that describe how such action interacts with the state and observation spaces, at its respective level, must be defined. These parameters are formulated as follows.

- O^d: Set of abstract observations. Let d be the depth of the SST at which an abstract action a^d, from the set A^d, can be executed, and S^d the set of abstract states at depth d. The set of abstract observations at depth d is made up by one observation o^d for each abstract state $s^d \in S^d$.

- Φ^d: Transition function at depth d. For an abstract action a_{ij}^d at depth d designed to transit from state s_i^d to s_j^d, the transition probability distribution defined over abstract states at depth d is given by,

$$\Phi^d(s_i^d, a_{ij}^d, s_j^d) = sim\_prob(s_j^d) \tag{4.23}$$

$$\Phi^d(s_i^d, a_{ij}^d, s_i^d) = sim\_prob(s_i^d) \tag{4.24}$$

$$\forall s_k^d \in S^d \mid neighbors(s_i^d, s_k^d), s_j^d \neq s_k^d, \Phi^d(s_i^d, a_{ij}^d, s_k^d) = sim\_prob(s_k^d) \tag{4.25}$$

$$\forall s_k^d \in S^d \mid \neg neighbors(s_i^d, s_k^d), s_i^d \neq s_k^d, \Phi^d(s_i^d, a_{ij}^d, s_k^d) = 0.0 \tag{4.26}$$

$$\forall s_k^d \in S^d \mid s_i^d \neq s_k^d, \Phi^d(s_k^d, a_{ij}^d, \cdot) = 0.0 \tag{4.27}$$

Eq. 4.26 and 4.27, represent the probabilities of transiting from a state different than s_i^d, and to a state that is not adjacent to s_i^d, respectively. As for Eq. 4.24, 4.23, and 4.25, represent the probability of staying in s_i^d, transiting to the goal state and to non-goal neighbor states, which are estimated by simulating the policy computed for the POMDP that models abstract action a_{ij}^d.

That is, let N be the amount of simulations performed for the policy computed from the POMDP that models abstraction action a_{ij}^d, $SimCount(s^d)$ the amount of simulations that ended at a child of abstract state s^d, and $Neig(s_i^d)$ the set of states that are neighbors of s_i^d, then the transition probabilities estimated from simulation for abstract action a_{ij}^d are given by

$$\begin{aligned} &\forall s^d \in \{s_i^d\} \bigcup Neig(s_i^d), \\ &sim\_prob(s^d) = \frac{SimCount(s^d)}{N} \end{aligned} \tag{4.28}$$

- Ω^d: Observation function at depth d. For an abstract action a_{ij}^d at depth d designed to transit from state s_i^d to s_j^d, the probability of perceiving an abstract observation after executing a_{ij}^d is given by,

$$\Omega^d(s_k^d, a_{ij}^d, o_k^d) = \begin{cases} \Phi^d(s_i^d, a_{ij}^d, s_k^d) & \text{if } s_k^d \neq s_i^d \\ 0 & \text{if } s_k^d = s_i^d \end{cases} \tag{4.29}$$

$$\Omega^d(s_k^d, a_{ij}^d, o_i^d) = 1 - \Omega^d(s_k^d, a_{ij}^d, o_k^d) \tag{4.30}$$

where o_k^d is the observation associated to state s_k^d. The heuristic used to model the observation distribution is: when an abstract action finishes its execution, the agent should always be relatively doubtful about its success of moving away from the initial state, because in case that it did not leave s_i^d it could simply fix it by executing a_{ij}^d again. Thus, the observation probability distribution is made of two elements: o_k^d and o_i^d, when $s_k^d \neq s_i^d$. In this way, the higher the probability a state has of being reached from state s_i^d with action a_{ij}^d, the more the agent will believe that it has been reached when the associated observation o_k^d is perceived. Therefore, if the agent remains doubtful about transiting out of s_i^d, it will invoke a_{ij}^d again and become more confident about not being in s_i^d. Thus, the execution of action a_{ij}^d might take place several times until the agent is confident enough of reaching the abstract state s_j^d. If an abstract action policy is relatively good, it should not have problems to eventually reach its goal state s_j^d.

Hence, in order to build an abstract action at level d, it is necessary to specify a pair of abstract states (starting and ending states) and to have a tuple $< S, A, O, \Phi, \Omega >$ modeling the environment at level $d + 1$ in the SST. Algorithm 1 describes the procedure followed to build a hierarchy of abstract actions, starting from the environment's bottom POMDP (see section 4.3.1). Let BP be the bottom POMDP, SST the state space tree of a particular environment, and N the amount of times abstract actions should be simulated to estimate their transition and observation distributions, then Algorithm 1 will return a vector of tuples H, where each of its tuple model the environment at a particular level in the SST, and the actions in tuple $H[i]$ invoke actions from $H[i-1]$. Thus, the environment's hierarchy of actions is given by the actions in each tuple of H.

In Algorithm 1, H is initialized with the bottom POMDP, next, for every level $i = [depth(SST) - 1, depth(SST) - 2, ..., 2, 1]$ (starting in the one above the concrete level, all the way up to the level below the root node) a tuple $< S, A, O, \Phi, \Omega >$ is built. In lines **9-10**, the set of abstract states is extracted from the SST and the set of observations is defined for those states, for level i, whereas in lines **11-13** the set of actions, transition and observation functions are initialized as empty sets. Then, for every ordered pair of neighbor states in S, in line **16**, using tuple from level $i+1$, an abstract action a is built to transit from s_0 to s_1, followed by line **17** that simulates N times action a to estimate its transition (t) and observation (z) distributions. In lines **18-20**, a, t, and z are included in the sets of all actions, state transitions and observation transition, respectively. Once the outer *for* loop has terminated, the tuple $< S, A, O, \Phi, \Omega >$ that models the environment at level i, is added to H.

Recalling example from Fig. 4.1, the process to build the hierarchy of actions would start by constructing the actions to transit between sections. For instance, to transit from section $s1$ to $s2$, cells $c1, c2, c3, c4, c5$ would make the state space of the abstract action, where cells $c3, c4$ would be the goal cells. For abstract action to transit from $s1$ to $s3$, only the goal state would be changed (to $c5$). This procedure would be performed for every pair of sections that are neighbors, and when done, the construction of abstract actions to transit between rooms would be next. This process would be also performed to transit between buildings.

Algorithm 1 Construction of hierarchy of actions

▷ $\boldsymbol{BP}$: Bottom POMDP
▷ $\boldsymbol{SST}$: State space tree of a particular environment
▷ $\boldsymbol{N}$: Amount of times abstract actions should be simulated to estimate their parameters
procedure BUILDHIERARCHYACTIONS(BP, SST, N)
    $i \leftarrow depth(SST) - 1$
    $H \leftarrow [\,]$
    $Append(H, BP)$
    while $i > 0$ **do**
        $S \leftarrow States(SST[i])$
        $O \leftarrow GenerateObs(S)$
        $A \leftarrow \emptyset$
        $\Phi \leftarrow \emptyset$
        $\Omega \leftarrow \emptyset$
        for all $s_0 \in S$ **do**
            for all $s_1 \in NeighborStates(s_0)$ **do**
                $a \leftarrow BuildAbstractAction(s_0, s_1, H[length(H) - 1])$
                $t, z \leftarrow EstimateTZ(a, S, O, N)$
                $A \leftarrow A \bigcup \{a\}$
                $\Phi \leftarrow \Phi \bigcup t$
                $\Omega \leftarrow \Omega \bigcup z$
            end for
        end for
        $Append(H, < S, A, O, \Phi, \Omega >)$
        $i \leftarrow i - 1$
    end while
    return H
end procedure

So far, it has been described how to formulate abstract actions, as a POMDP policy, designed to transit between states at any depth in the SST. However, any of these abstract actions by themselves are insufficient for solving tasks in which the goal state is not a neighbor to the agent's initial state. In the next section, the architecture's operation phase is introduced, which is constituted by methods that build and execute global policies that are capable of solving any task, through the execution of abstract and concrete actions.

4.4 Architecture operation

At the operation phase, the architecture is ready to receive task requests. Every time a request is received, the architecture starts a two part procedure to solve the task at hand. First, by bounding the state space based on the robot's current state and the goal state (section 4.4.1.1), planning is performed to generate a task specific plan, called hierarchical policy (section 4.4.1.2). Then, the hierarchical policy is executed in a top-down approach until the goal state is reached.

4.4.1 Planning

A hierarchical policy is a sequence of standard POMDP policies defined over several levels of the SST. The system takes advantage of the hierarchical representation by generating plans over a reduced version of the state space. Below is described how the state space is bounded to a *Relevant sub-space* (RSS), followed by the method that builds a hierarchical policy based on the task's RSS.

4.4.1.1 Relevant sub-space

Once a task request is issued to the agent, the architecture proceeds to bound the state space to the smallest sub-region that still allows the agent to reach the goal state. The following elements are used to generate such sub-region known as *Relevant Sub-Space*, based on definition 4.

1. The particular environment's *State Space Tree*.

2. The goal state.

3. The robot's state at the time the request is made.

Definition 4 (Relevant Sub-Space) *Let* $\boldsymbol{S}$ *be the initial state and* $\boldsymbol{G}$ *the goal state for a given task request, then the sub-tree whose root node is the deepest common ancestor of* $\boldsymbol{S}$ *and* $\boldsymbol{G}$ *in SST is the task's Relevant Sub-Space.*

For instance, in Fig. 4.4 the RSS for the task that has the green goal state (enclosed by a green polygon) reduces significantly the state space, since the initial (yellow leaf node) and goal states are contained in *Room-2*. However, in the case of the orange goal state, its RSS remains the same as the SST because the only ancestor it shares with the initial state is SST's root node.

4.4.1.2 Hierarchical policy

By defining a sequence of policies, one for each level in the RSS hierarchy, the agent is be able to determine the best action at each resolution of the environment. Before specifying the hierarchical policy structure, the concept of a *hierarchical state* [Hengst, 2004] is introduced, which has been modified to fit into the context of the RSS representation, also, an extension of this definition that enables to specify a *hierarchical state* down to a given depth is presented.

Definition 5 (Hierarchical state) *A hierarchical state is defined by a sequence of states, one for each level of the RSS hierarchy, such sequence starts with RSS's root node.*

Definition 6 (D-hierarchical state) *A D-hierarchical state is a hierarchical state specified from depth 0 all the way to depth D, over an RSS hierarchy.*

Thus, let G be a goal state located at depth d in the RSS hierarchy, and G^H the d-hierarchical state that contains G as the sequence's last element. We define a *hierarchical policy* over G^H as follows.

Definition 7 (Hierarchical Policy) *Let $S^H = [s_0, ..., s_D]$ be a D-hierarchical state with length $D+1$, and $\Pi^H = [\pi_0, ..., \pi_{D-1}]$ a sequence of policies with length D, where the i-th policy in Π^H was computed for a POMDP modeled over the children states of the i-th state in S^H, then Π^H is a Hierarchical Policy defined over the D-hierarchical state S^H.*

For instance, *hierarchical policy* $\Pi^H = [\pi_0, \pi_1]$ is defined over *2-hierarchical state* $S^H = [B1, R1, S3]$, both shown in Fig. 4.5. It is worth noting that a *hierarchical policy* will have one less element than the *D-hierarchical state* G^H over which it is defined, because a policy is required to guide the agent towards each state in G^H except for its first state, which is RSS' root node. Thus, after having defined the *D-hierarchical state* and *hierarchical policies*, the algorithm employed to build a *hierarchical policy* is presented below.

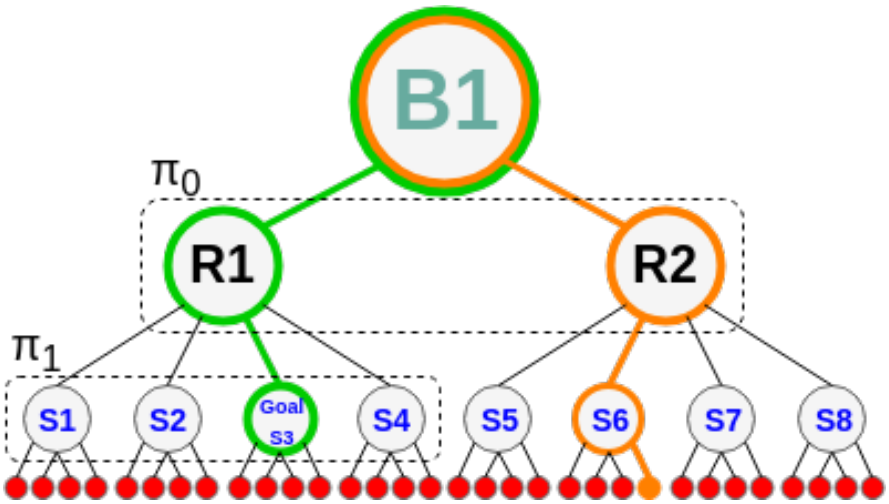

Figure 4.5: 2-hierarchical and 3-hierarchical states are depicted as green and orange traces, respectively. Also, *hierarchical policy* Π^H is defined over the green 2-hierarchical state. For each policy in Π^H, a dotted rectangle encloses the sub-state space over which the policy executes its actions.

Let G be a goal state for which the relevant sub-space RSS is defined, d the level in RSS at which G is located and H the vector of tuples that model the environment at every level of the SST, except to level 0, then Algorithm 2 builds a *hierarchical policy* that enables the agent to reach goal state G that is located at depth d in the task's RSS hierarchy. In line **6**, G^H is initialized with the *d-hierarchical state* that

Algorithm 2 Construction of hierarchical policy

▷ $\boldsymbol{G}$: Goal state
▷ $\boldsymbol{RSS}$: Relevant sub-space for G
▷ $\boldsymbol{d}$: Depth level at which G is located in RSS
▷ $\boldsymbol{H}$: Vector of tuples that model the environment at every level of the SST
procedure HIERARCHICALPOLICY(G, RSS, d, H)
    $G^H \leftarrow DHierarchicalState(G, d)$
    $\Pi^H \leftarrow [\,]$
    $parent\_node \leftarrow G^H[0]$
    $i \leftarrow 1$
    while $i < length(G^H)$ **do**
        $j \leftarrow length(H) - (RootLevel(RSS) + i)$
        $S \leftarrow Children(parent\_node)$
        $S \leftarrow S \bigcup \{ps \mid ps \notin S, \exists s \in S, neighbors(ps, s)\}$
        $A \leftarrow RelevantActions(S, H[j][1])$
        $O \leftarrow RelevantObservations(S, H[j][2])$
        $\Phi \leftarrow H[j][3]$
        $\Omega \leftarrow H[j][4]$
        $R \leftarrow GoalBasedReward(S, A, G^H[i])$
        $\pi \leftarrow SolvePolicy(S, A, O, \Phi, \Omega, R)$
        $Append(\Pi^H, \pi)$
        $parent\_node \leftarrow G^H[i]$
        $i \leftarrow i + 1$
    end while
    return Π^H
end procedure

contains G. In line **7**, *hierarchical policy* Π^H is initialized as an empty sequence, while line **8** initializes *parent_node* with the RSS' root node. Once the procedure has entered the while loop, in every cycle a POMDP is formulated and solved by doing the following:

- **Line 11**: Since the first tuple in H corresponds to the last level in the environment's SST, and RSS is a sub-tree of the SST, then index j is computed so it corresponds to level i in RSS, where $RootLevel(R)$ returns the level in the SST at which the root of R is located.

- **Lines 12-13**: Defines the set of states S as the union of *parent_node*'s children in RSS hierarchy with the set of peripheral states (those that are not children of *parent_node* and are adjacent to some of its children, such peripheral states are not necessarily in RSS).

- **Lines 14-15**: *RelevantActions* selects the set of actions that have a probability greater than zero of transiting the agent between (at least) a pair of states in S, while *RelevantObservations* selects those observations that have a probability greater than zero of being perceived after executing any of the actions in A while reaching any state in S.

- **Lines 16-17**: The transition and observation functions are extracted from tuple $H[j]$, which models the environment at level i in the RSS.

- **Line 18**: A reward function is defined over sets S and A, in which transitions that end in local goal state $G^H[i]$ have a large positive reward, transitions that end in any other peripheral state have a large negative reward, while any other transition has a reward value of -1.

- **Lines 19-20**: *SolvePolicy* returns a policy computed for the formulated POMDP, which is subsequently added to the Π^H sequence.

- **Line 21**: *parent_node*'s value is updated with the next local goal state, *i.e.* $G^H[i]$.

4.4.2 Plan execution

Since a *hierarchical policy* is constituted by several standard policies, it is necessary to define how they pass control to each other in order for the agent to reach the goal state.

4.4.2.1 Hierarchical policy execution

Let G^H be a *D-hierarchical state* that represents the agent's goal, and Π^H a *hierarchical policy* defined over G^H in the RSS for the pair of states g_D and s_0, which are the last state in G^H and the agent's initial state, respectively. Then the task of executing a plan represented by Π^H is reduced to determine at any time step which of the policies in Π^H must have control until the agent believes it has reached its goal state g_D. Thus, this section starts by defining a condition called *Hierarchical contain* for a pair of *D-hierarchical states*, followed by the description of the *hierarchical policy* execution algorithm.

Definition 8 (Hierarchical contain) *Let hs_a and hs_b be two D-hierarchical states with lengths $length(hs_a)$ and $length(hs_b)$, where $length(hs_a) \leq length(hs_b)$, and every state in hs_a is also in hs_b. Then we say that hs_a hierarchically contains hs_b.*

In other words, we say that hs_a hierarchically contains hs_b if hs_b is a *D-hierarchical state* defined at a greater depth than hs_a, and the last state in hs_a is an ancestor node to the last state in hs_b with respect to the SST hierarchy. Thus, reaching a goal state is equivalent to having a goal's *D-hierarchical state* to hierarchically contain the agent's current *hierarchical state*. The algorithm for the execution of a *hierarchical policy* is presented below.

Algorithm 3 executes *hierarchical policy* Π^H to take the agent from an initial *hierarchical state* s^H to a state that is hierarchically contained in G^H. To do so, the algorithm performs the following steps:

Algorithm 3 Execution of a hierarchical policy

```
1: ▷ Π^H: Hierarchical policy
2: ▷ s^H: Hierarchical state of initial state s
3: ▷ G^H: D-hierarchical state of goal state G
4: procedure ExecuteHierarchicalPolicy(Π^H, s^H, G^H)
5:     B ← init(s^H)
6:     Z ← initNull(length(s^H))
7:     A ← initNull(length(s^H))
8:     i ← 0
9:     while i < length(G^H) do
10:        if s^H[i] ≠ G^H[i] then
11:            while i < length(G^H) do
12:                ExecuteLocalPolicy(Π^H[i − 1], B, Z, A, i)
13:                i ← i + 1
14:            end while
15:        else
16:            i ← i + 1
17:        end if
18:    end while
19: end procedure
```

- **Lines 5-7**: At line **5** a list of belief state vectors is initialized from the *hierarchical state* s^H. Each belief state vector (one for each level in the RSS hierarchy) is initialized by assigning a probability of 1 to the agent's state at the beginning of the task, and 0 for every other states. As for lines **6** and **7**, they initialize vectors Z and A of empty elements, one element for each level in the RSS hierarchy. These vectors are required for the **ExecuteLocalPolicy** procedure to keep track of the latest observations perceived and actions performed by the agent at each level of RSS.

- **Lines 9-10**: When the procedure has entered the first while loop, it compares the states in G^H and s^H until it finds the shallowest level at which G^H and s^H differ. At this level, the *hierarchical policy* starts its execution.

- **Line 11-12**: The $(i-1)$-*th* policy of Π^H is invoked to take the agent to the i-*th* state of G^H. The **ExecuteLocalPolicy** procedure finishes when the executed policy has reached its local goal state. This step is repeated until the last

policy in Π^H is executed, meaning that the agent has reached the goal state.

In general, what Algorithm 3 does is to use each policy in Π^H to move the agent within the sub-space contained by the abstract state the policy has for goal. For instance, in Fig. 4.5 in which the orange trace represents the agent's initial state, the first policy the **ExecuteHierarchicalPolicy** procedure would invoke is π_0, since G^H and s^H differ in the room level of the RSS. After policy π_0 is done, it is safe to say that the agent is within **room 1**, however, there is uncertainty with respect to the subsection of **room 1** the agent is located in. Then, the algorithm proceeds to invoke policy π_1, thus, after π_1 is done we can assure that the agent is within **subsection 3**, which is the goal state, and **ExecuteHierarchicalPolicy** terminates its execution. Algorithm 3 can be seen as the main controller of the system that is in charge of determining the order in which sub-policies are invoked, however, the detail of how these local policies invoke the actual actions that move the agent to the goal state is hidden within the **ExecuteLocalPolicy** procedure, which is described in the next section.

4.4.2.2 Local policy execution

Similarly to the execution of a *hierarchical policy* Π^H, special care must be taken when executing policies of Π^H because they are not defined over the full state space, but over sub-regions of it. For this reason, a method that handles the update of the global belief state with observations returned by abstract actions is required.

In general, Algorithm 4 executes a local policy by invoking actions and updating the local POMDP's belief state; while simultaneously keeps track of the latest observation and action, perceived and performed respectively, at every level in the RSS, that might be used by other local policies invoked at the same level. In fact, the input parameters for Algorithm 4 are assumed to be passed *by reference*, which means that changes made to them within one call to the procedure, will remain after the procedure has returned control to the point from where it was called.

The **ExecuteLocalPolicy** procedure receives as input parameters:

Algorithm 4 Execution of local policy

▷ $\boldsymbol{\pi}$: Local policy
▷ $\boldsymbol{B}$: List of initial belief state vectors
▷ $\boldsymbol{Z}$: Vector of the last observations perceived at each level in the RSS
▷ $\boldsymbol{A}$: Vector of the last actions performed at each level in the RSS
▷ $\boldsymbol{d}$: Depth in the RSS hierarchy at which local policy π executes
procedure EXECUTELOCALPOLICY(π, B, Z, A, d)
    $b \leftarrow null$
    if $Z[d] \neq null$ **and** $A[d] \neq null$ **then**
        $z_{space} \leftarrow getObsSpace(\pi)$
        $b \leftarrow updateBelief(uniformBelief(z_{space}), Z[d], A[d])$
    else
        $b \leftarrow B[d]$
    end if
    $a \leftarrow getAction(\pi, b)$
    while $a \neq terminate\_execution$ **do**
        $z \leftarrow null$
        if a is concrete **then**
            $z \leftarrow executeAction(a)$
        else
            $z \leftarrow ExecuteLocalPolicy(a, B, Z, A, d + 1)$
        end if
        $b \leftarrow updateBelief(b, z)$
        $Z[d] \leftarrow z$
        $A[d] \leftarrow a$
        $a \leftarrow getAction(\pi, b)$
    end while
    $s \leftarrow getMostLikelyState(b)$
    $z^{d-1} \leftarrow getAssociatedObservation(s)$
    return z^{d-1}
end procedure

- π: Local policy to be executed.

- B: List of initial belief state vectors, which is used by a local policy at depth d only if it is the first time such depth has been reached by a recursive call of **ExecuteLocalPolicy**.

- Z: Vector that holds the latest observation perceived at every level in the RSS hierarchy. If a given depth d has already been visited by a **ExecuteLocalPolicy** call, then for future local policies located at depth d, they will use the observation and action stored in $Z[d]$ and $A[d]$ to generate their initial belief state.

- A: Vector that holds the latest action performed at every level in the RSS hierarchy.

- d: Depth in the RSS hierarchy at which local policy π executes.

As for the main steps of Algorithm 4, these are detailed below.

- **Lines 8-13**: The initial belief state for policy π is built; whether it uses the latest observation and action, perceived and performed, at depth d to update a uniform belief state, or the global initial belief state B.

- **Line 14**: Gets the first action from policy π and initial belief state b.

- **Lines 16-21**: Once the procedure has entered the while loop (which is finished by the policy invoking the *terminate_execution* action), the current selected action a is invoked. If a is a concrete action, it is executed by invoking the correspondent *basic module* (see section 4.2.1.1). Whereas for abstract actions, are executed as a local policy by the **ExecuteLocalPolicy** procedure.

- **Lines 22-25**: With the observation z returned by executing action a, the local belief state b is updated, observation z is stored in Z, action a is stored in A, and the next action to be executed is obtained from local policy π.

- **Lines 27-29**: Once the *terminate_execution* is called and breaks the while loop, the observation associated to the most likely state (according to b) is returned by the procedure, in case the local policy being executed is an abstract action.

Thus, the recursive definition of Algorithm 4 makes possible to invoke both, policies from a *hierarchical policy* Π^H, and abstract actions. Moreover, since the procedure takes care of passing control among local policies, one must only worry about deciding what action should be taken at the end of the execution of the local policy, which in the context of the planning architecture, the **ExecuteHierarchicalPolicy** procedure is responsible of making this decision in line **12** of Algorithm 3.

Lets take for example the task request depicted by the yellow and orange cells in Fig. 4.4. The hierarchical policy to solve this task would be made of three policies, defined over the rooms, subsections and cells. Thus, the execution of such hierarchical policy would be as follows:

1. The local policy defined over rooms would be executed and stopped until **R3** was reached.

2. Local policy defined over sections in **R3** would then be executed, since at this point the robot is already in **R3**, and it would stop when the agent entered the subsection containing the orange cell.

3. Finally the third local policy would be executed to make sure that the robot reaches the orange cell, given that it already is within the same subsection.

4.5 Chapter Summary

To summarize, the key-points that constitute the proposed planning architecture are presented below.

- This planning scheme relies on two assumptions: the state of the robot is known at the moment it starts executing a task, and the designer can provide domain specific knowledge necessary to describe the domain's dynamics.

- A knowledge base is employed to represent the description provided by the designer for: the skill sets the robot is endowed with, the dynamics of the environment, a hierarchical function and a characterization of the particular environment in which the robot will operate.

- A hierarchical representation of the state space (SST) is employed to abstract the original state space and eventually decompose tasks. Such structuring enables to define sub-regions at which local policies can be executed.

- The POMDP formulation is used to model abstract actions, which are recursively built upon lower level components (less abstract actions, states and observations).

- To address a task request issued by the user, the following steps are performed:

    1. **Knowledge base construction**: A designer provides domain specific knowledge which is required to perform planning.

    2. **Architecture initialization**: The architecture extracts from the knowledge base information related to the domain dynamics and its hierarchical structure to build a hierarchy of actions, which are modeled as POMDP policies.

3. **Architecture operation**: After performing only once steps 1 and 2, if the agent receives a task request it will perform the following operations.
   (a) Based on the initial and goal state, an RSS is specified.

   (b) A hierarchical policy is built over the D-hierarchical goal state used to specify the RSS.

   (c) The local policies that constitute the hierarchical policy are executed in a top-down way, until the goal state is reached, which terminates the execution of the hierarchical policy.

   (d) Steps **a-c** are repeated for every future task request.

With the proposed architecture described in this chapter, the following contributions are presented:

1. A general framework for hierarchical task planning, capable of integrating new skills into a planning problem, without having to modify the description of those skills that already are part of the system.

2. A recursive definition for abstract actions (as POMDPs) that enables the construction of arbitrarily deep hierarchies of POMDPs for task planning problems, starting from a standard POMDP and a hierarchical representation of its state space.

3. A methodology to generate and execute a multi-resolution plan in a sub-region of the original state space, employing its hierarchical representation and a hierarchy of POMDPs.

In the next chapter, a set of experiments are presented with the purpose of evaluating the performance of the proposed architecture, in comparison to two baseline methods, in terms of efficiency and effectiveness. The experiments are designed to

estimate the impact the size of the environment, and the agent's state uncertainty have in the planning systems' performance, which are believed to be the variables that can most increase the difficulty of generating a plan and executing it, respectively.

Chapter 5

Experiments and results

A set of five experiments were performed under several scenarios, that vary in the size of the environment and state uncertainty of the agent, to evaluate the planning architecture's effectiveness and efficiency. Furthermore, two baseline methods (standard POMDP and a two level hierarchical planner) were tested and their results compared to the ones obtained by the proposed architecture. This chapter starts by introducing the navigation domain that was used as study case, followed by a description of the parameters that specify an experimental configuration, *i.e.* the control variables, evaluation metrics and statistical model employed to evaluate the obtained results. Next, the configuration and results of each experiment are presented, followed by a discussion on the experimental results and finally the chapter's summary.

5.1 Navigation domain as study case

In order to evaluate the proposed architecture, a mobile robot navigation domain was selected as test scenario. Since this type of domain has an inherent high degree of structure, and real-world problems tend to have large environments, it seemed a good option to study the impact a hierarchical approach might have in the performance of a task planning system. To model the bottom POMDP of the environment, the space was discretized into a grid of square cells (with identical dimensions) aligned horizontally and vertically, while the agent's spectrum of actions was constituted

by movements that led to any cell, horizontally or vertically, adjacent to its current location. The probability distributions for the state transition and observation functions were modeled to have two and nine possible outcomes, respectively. The details on the bottom POMDP parameters are presented below.

- S: There is a state for each cell in the environment.

- A: There is an action to move the agent to each of its horizontal and vertical neighbor cells, that is, $\{move\_up, move\_down, move\_left, move\_right\}$.

- O: There is an observation for each cell in the environment.

- Φ: The transition probability distribution of each action has two possible outcomes, defined by the following neighborhood relations and probabilities.

  $move\_up = \{< move\_up, above, 0.9 >, < move\_up, current\_cell, 0.1 >\}$
  $move\_down = \{< move\_down, below, 0.9 >, < move\_down, current\_cell, 0.1 >\}$
  $move\_left = \{< move\_left, at\_left, 0.9 >, < move\_left, current\_cell, 0.1 >\}$
  $move\_right = \{< move\_right, at\_right, 0.9 >, < move\_right, current\_cell, 0.1 >\}$

  The first transition in each set stands for the action's goal cell, while the transitions defined by the *current_cell* neighborhood relation model the possibility of staying in the agent's current cell.

- Ω: The observation probability distribution of each action is modeled with the same set of nine observation neighborhood relations, *above*, *below*, at_left, at_right, $above\_left$, *above_right*, $below\_left$, *below_right*, *current_cell*. A 3×3 Gaussian kernel centered, with respect both axes, to the transition's ending state is modeled with the set of observation neighborhood relations. In contrast to the state transition distributions, because the agent's state uncertainty is one of the control variables, the specific observation probabilities are not fixed, but rather depend on the standard deviation used to compute the discrete Gaussian

for each experimental configuration. The particular observation probabilities are calculated with Eq. 5.1.

$$g_{x,y} = \eta \frac{e^{\frac{-r^2}{2\sigma^2}}}{2\pi\sigma^2} \tag{5.1}$$

where $g_{x,y}$ is the probability of perceiving the observation located at column x and row y (with respect to the kernel's origin), r is the Euclidean between (x, y) and the kernel's center coordinate (expressed in standard deviations), σ is the standard deviation and η a normalization factor. Moreover, by using a Gaussian distribution to model the observation function it is possible to establish a comparison between the synthetic scenarios generated for experiments $1 - 4$ and a real robot, as shown in experiment 5 (see section 5.7).

The hierarchical function, employed to abstract the state space, considered four levels of resolution (from less to more abstract): cells (concrete level), subsections, rooms and buildings. The amount of levels a hierarchical function has is result of how much knowledge the designer is able to provide about the environment's structure. For the experiments of this thesis, it is believed that four levels are enough to study the impact state abstraction might have in the effectiveness and efficiency of a task planning system.

5.2 Experiment parameters

This section describes the baseline methods employed for comparison purposes, parameters that define an experimental configuration, as well as the metrics employed to measure the experiments' independent and dependent variables. At the end of this section, the details on the statistical model used to evaluate the experimental results are presented, followed by a description of the sampling technique performed.

5.2.1 Baseline methods and failure criteria

In order to measure the impact state abstraction has in task planning, two baseline methods were employed as reference for comparison purposes. The first one is a

standard POMDP [Kaelbling et al., 1998] (equivalent to our architecture's bottom POMDP), while the second baseline is a method that performs a single step of state abstraction to solve tasks in a hierarchical representation with two levels, which is an implementation we did in an attempt to replicate the planning architecture proposed by [Sridharan et al., 2018]. Despite this implementation omits several features of the original work, it does maintain three properties that establish it as a middle point between the standard POMDP and our proposal: employs a hierarchical representation of the state space of two levels, performs deterministic planning in the top level and abstract actions (those in the top level) are performed as POMDP policies in the bottom level. Both baseline methods are described in detail below.

- **Standard POMDP**: The standard or flat POMDP (FP), is a POMDP that shares the same parameters than the bottom POMDP (see section 4.3.1), that is, S, A, O, Φ and Ω. In every simulation run, the flat POMDP defines its reward function, by assigning a large positive reward for transitions that end in the goal cell for that run, and then its policy is computed.

- **Two level planner**: The two level planner baseline (TLP), in its initialization step, abstracts the state space in a hierarchy with two levels: i) the concrete level (cells) and ii) the top-level, that has a state for each building in the environment. The process of state abstraction is followed by computing two policies to transit between each pair $(Building_i, Building_j)$, one from $Building_i$ to $Building_j$, and the other one for the opposite direction. After the initialization step has been finished for an environment, the following steps are performed in every simulation run.

  1. For a pair $(cell_0, cell_{goal})$, where $cell_0$ is the initial position and $cell_{goal}$ the goal cell, the method builds a sequence $P = [p_0, ..., p_n]$ of policies (computed in the initialization step), that will take the agent from its current building to $Building_{goal}$ (the one that contains $cell_{goal}$).

  2. Computes a policy P_{final} for a POMDP whose state space is defined as the set of cells contained in $Building_{goal}$.

  3. Executes in sequence the policies in P, followed by the execution of P_{final}.

Furthermore, since having a policy wondering around for an undetermined amount of steps was a possibility, failure criteria were defined for each one of the evaluated methods, including the proposed architecture.

- **Standard POMDP failure criteria**

  1. The amount of steps taken surpasses the POMDP's state space size, *i.e.* $|S|$.

  2. The policy terminates its execution in a cell different to the goal cell.

- **Two level planner failure criteria**

  1. The amount of steps taken by a single policy p_i, computed to transit from $Building_i$ to $Building_j$, surpasses its POMDP state space size, *i.e.* the amount of cells in $Building_i$ plus the amount of cells in $Building_j$ that are adjacent to a cell in $Building_i$.

  2. A policy p_i, computed to transit from $Building_i$ to $Building_j$, terminates its execution in a cell that is not contained in $Building_j$.

  3. Policy P_{final} takes the agent to a cell that is not contained in $Building_{goal}$.

  4. Policy P_{final} terminates its execution in a cell different to $cell_{goal}$.

- **Proposed architecture failure criteria**

  1. The amount of steps taken by a policy, whether is an abstract action or an element from a hierarchical policy, surpasses the size of its POMDP state space.

  2. A policy, whether is an abstract action or an element from a hierarchical policy, takes the agent to a state that was not modeled in its POMDP.

3. An abstract action returns an observation that has a probability of 0.0 of being returned, according to the observation probability distribution estimated for that abstract action.

4. A hierarchical policy terminates its execution in a cell different to $cell_{goal}$.

Hence, during a simulation run, if a method entered to any of its failure scenarios, the simulation is stopped and considered to be a failed run for that method. Furthermore, from now on, the standard POMDP, two level planner and proposed architecture, will be referred as FP, TLP and HP, respectively.

5.2.2 Control parameters

The parameters that define an experimental configuration can be classified as those with a fixed value and those whose value is modified across the set of experiments.

- **Fixed parameters**
  - POMDP solving algorithm: In order to compute the policy for any POMDP (FP, POMDPs in TLP, abstract actions and elements from a hierarchical policy in HP), Point-based value iteration [Pineau et al., 2003] (PBVI) is used, which is an approximate solving algorithm.

  - Number of belief points: Instead of computing an exact solution, PBVI solves a POMDP for finite set of belief points, where the larger the set of belief points is, the better the computed solution will approximate the exact solution. For every FP and POMDPs in TLP, **250** belief points were employed, while for every POMDP in HP **75** were used. As explained in [Pineau et al., 2003], for any belief set B and horizon n, PBVI generates an estimate V_n^B. The denser B samples the belief simplex (whose size depends on the POMDP's state space size), V_n^B converges to V_n^*, the true value function. Therefore, a larger set of belief points is employed for FP and TLP than for HP, since their simplex will always be larger than the ones in HP. However, given that no heuristic

has been found in literature to select the amount of belief points based on the problem's dimension, the criterion followed to select these values was that they should be large enough so that each method could find good policies in the first experimental configurations, but also that they were small enough so that the total amount of simulations could be finished in a reasonable amount of time.

- Discount factor and horizon: For the computation of every policy for the three evaluated methods, a discount factor of **0.95** and a horizon of **10** were used.

- Number of simulations for parameters estimation: To estimate the transition and observation distributions of an abstract action (see section 4.3.2.3) 100 simulations were performed for each abstract action.

- Number of cells connecting rooms: When an environment is generated, the pairs of cells that connect adjacent rooms are randomly selected, and the amount of pairs is equal to the room's side length (in cells) divided by two. For instance, in Fig. 5.1 the pairs $\langle c4, c5\rangle$, $\langle c28, c29\rangle$ were randomly selected to connect the pair of upper rooms in the leftmost building.

- Number of cells connecting buildings: Similar to the cells that connect rooms, the amount of pairs of cells connecting adjacent buildings is equal to the subsection height (in cells). However, the set of connecting pairs were always arranged as an uninterrupted sequence, whose position was randomly selected.

- **Variable parameters**

  - Number of buildings: Amount of buildings that constitute an environment, where all buildings in an environment have the same dimensions. The relative position of a building to an adjacent one was randomly selected.

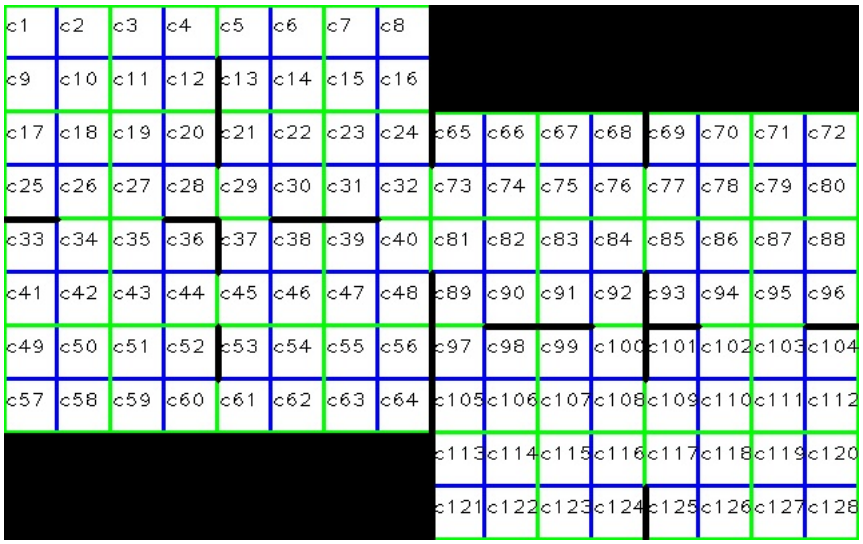

Figure 5.1: Example of an environment made of two buildings, which was employed for the fourth experimental configuration in experiment 1 (see section 5.3). In this environment, subsections (green squares) have a dimension of 2, and the same goes for rooms and buildings.

- Subsection dimension: The width and height dimension (in cells) every subsection has.

- Room dimension: The width and height dimension (in subsections) every room has.

- Building dimension: The width and height dimension (in rooms) every building has.

- Standard deviation for Ω: The value of σ used to compute the discrete Gaussian kernel to model the observation distribution (see Eq. 5.1).

5.2.3 Independent variables

In order to characterize the agent's state uncertainty and the environment's size, which are the independent variables in every experiment, a pair of variables that take into account the values of the variable parameters are employed.

- Size of the environment: The size of the environment is measured by the total amount of cells.

- State uncertainty: Since the agent's state uncertainty is expected to be larger as the kernel size and standard deviation increase, Shannon entropy for a Gaussian discrete kernel is used instead, given by Eq. 5.2. Hence, since the same observation distribution is used across an environment, the Shannon entropy value for such distribution describes the amount of uncertainty an agent is expected to face during a task.

$$H(g) = -\sum_{x=0}^{width(g)-1}\sum_{y=0}^{height(g)-1} g_{x,y} log_2(g_{x,y}) \tag{5.2}$$

  where $g_{x,y}$ is the probability of perceiving the observation located at coordinate (x, y) within kernel g.

5.2.4 Dependent variables

In the same way independent variables describe the environment properties of interest, the purpose of dependent variables is to measure the effect independent variables have in the phenomenon that is being studied, *i.e.*, the planning methods. For the set of experiments presented in this chapter, efficiency and effectiveness are the phenomenon's properties of interest, and their operational definitions are presented below.

- Efficiency in planning: The time required to compute a plan (policy) measured in seconds. For FP is the time required to compute its policy, for TLP is the time required to determine the sequence of policies, and compute the final building policy. Whereas for HP, is the total time required to compute all the policies that constitute the hierarchical policy.

- Efficiency in execution: During execution, efficiency is measured by means of the path relative cost, which is the amount of actions required to take the agent from the initial cell to the goal cell, compared to the shortest path between

them, that is:

$$Path\,relative\,cost = \frac{actions_{exe}}{length_{opt}} \tag{5.3}$$

where $length_{opt}$ is the length of the shortest path between the initial and goal cell of a task, and $actions_{exe}$ is the amount of concrete actions performed by the agent. Moreover, since the paths of simulations in which the evaluated method does not reach the goal state might be shorter than the optimum path (because of the early termination), the path relative cost is computed only for runs in which the goal state is reached, in order that the average of this metric does not misinform about the method's actual performance.

- Effectiveness: The purpose of measuring the effectiveness of a planning method is to establish how reliable it is in solving tasks. To measure effectiveness in planning, success ratio and relative error are employed, given by the following equations.

$$Success\,ratio = \frac{N_{success}}{N_{total}} \tag{5.4}$$

$$Relative\,error = \frac{error\_dist}{length_{opt}} \tag{5.5}$$

where N_{total} is the total amount of simulations executed in an experimental configuration, $N_{success}$ the amount of simulations in which the goal cell was reached, and $error\_dist$ the length of shortest path between the agent's final location and the goal cell for a simulation run. Hence, while Eq. 5.4 expresses the proportion of simulations in which a model succeeded, Eq. 5.5 describes how far the agent was from the goal cell at the end of a simulation, relative to how far it was at the beginning of it (which is a normalized value).

5.2.5 Statistical parameters

With regards to the parameters employed to sample data and evaluate the experimental results, these are described below.

- Sample size: In order to detect a difference of 0.2 and 0.3 standard deviations, between two means at the 5% significance level with statistical power 90%,

samples of size 233 and 525, respectively, are employed for different experiments.

- Statistical model: Since the sample sizes are not large, the t distribution is employed to find a difference between two means.

- Sampling procedure: Let B_0 and B_n be the sets of cells contained in the leftmost and rightmost buildings in the environment, respectively. Then, for a sample size of N, Algorithm 5 is performed to sample the set of initial-goal state pairs for an experimental configuration.

Algorithm 5 Procedure to sample pairs of initial and goal states

```
▷ N: Sample size
▷ B_0: Set of cells contained in the leftmost building
▷ B_n: Set of cells contained in the rightmost building
procedure SAMPLE(N, B_0, B_n)
    i ← 0
    S ← [ ]
    while i < N do
        s_0 ← SampleState(B_0 ⋃ B_n)
        s_goal ← NULL
        if s_0 ∈ B_0 then
            s_goal ← SampleState(B_n)
        else
            s_goal ← SampleState(B_0)
        end if
        Append(S, < s_0, s_goal >)
        i ← i + 1
    end while
    return S
end procedure
```

In Algorithm 5, sub-routine $SampleState(A)$ returns an element randomly selected (with replacement) from set A, and $Append(V, t)$ appends tuple t at the end of vector V. Hence, Algorithm 5 guarantees that the initial and goal cells will be selected from opposite ends of the environment.

5.3 Experiment 1

Experimental setting:

- Number of buildings: 2
- Subsection dimension: 2
- Room dimension: 2
- Building dimension: 2
- Standard deviation: [0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6]
- Amount of simulations: 233

5.3.1 Objective

To determine if the magnitude of uncertainty in the agent's state decreases the effectiveness and efficiency of HP as much as it does to FP or TLP.

5.3.2 Hypothesis

If the state uncertainty increases, then HP will consistently behave as equal or more effectively, and more efficiently, than FP and TLP.

5.3.3 Results

In Figs. 5.2, 5.3, 5.4, and 5.5 the success ratio and average scores on relative error, path relative cost, and planning time for FP, TLP, and HP obtained in each experimental configuration in experiment 1 are shown. In addition, in Fig. 5.6 the 95% and most significant (widest interval that does not include 0) confidence intervals for the mean differences found between HP and the two baselines are shown.

Each plot in Figs. 5.6, 5.11 and 5.16 shows the confidence intervals found for the difference of means between a pair of methods (FP, TLP or HP), with respect to a certain metric (path relative cost, planning time or relative error). For each experimental configuration (horizontal axis), two confidence intervals are plotted: the 95% (since it is a standard practice to report this confidence interval) and the most significant interval (*i.e.*, the widest interval that does not cover the value 0) in blue and green, respectively. Also, the left axis shows the value for the center and ends of each interval, while the right axis shows the percentage of confidence (which is plotted as a green bar) of each most significant interval. Therefore, the wider a green interval is, it means that there is more confidence that the difference between the pair of methods for that configuration is not due to randomness.

From Figs. 5.2 and 5.3 one can observe that, although HP and TLP obtained a similar relative error (according to Fig. 5.6f), TLP dominated HP and FP in terms of effectiveness across the overall set of trials, by obtaining the highest success ratio and lowest relative error. For the first couple of trials ($\sigma = 0.2, 0.4$), HP showed to be more effective than FP, however, at $\sigma = 0.6$ HP experimented the largest decrease, with respect to $\sigma = 0.4$, among the three methods and from there on its success ratio kept going lower. With regards to the methods' efficiency, Figs. 5.4 and 5.5 show that TLP obtained the lowest value for the path relative cost for most of the trials, while HP reported the lowest planning time over the full set of trials.

In view of HP not reporting the highest success ratio and, simultaneously, the lowest planning time over the full set of trials, it is concluded that there is insufficient evidence to support the following hypothesis: *If the state uncertainty increases, then HP will consistently behave as equal or more effectively, and more efficiently, than FP and TLP*. Moreover, in section 5.8 a discussion that covers the results found in each experiment is presented.

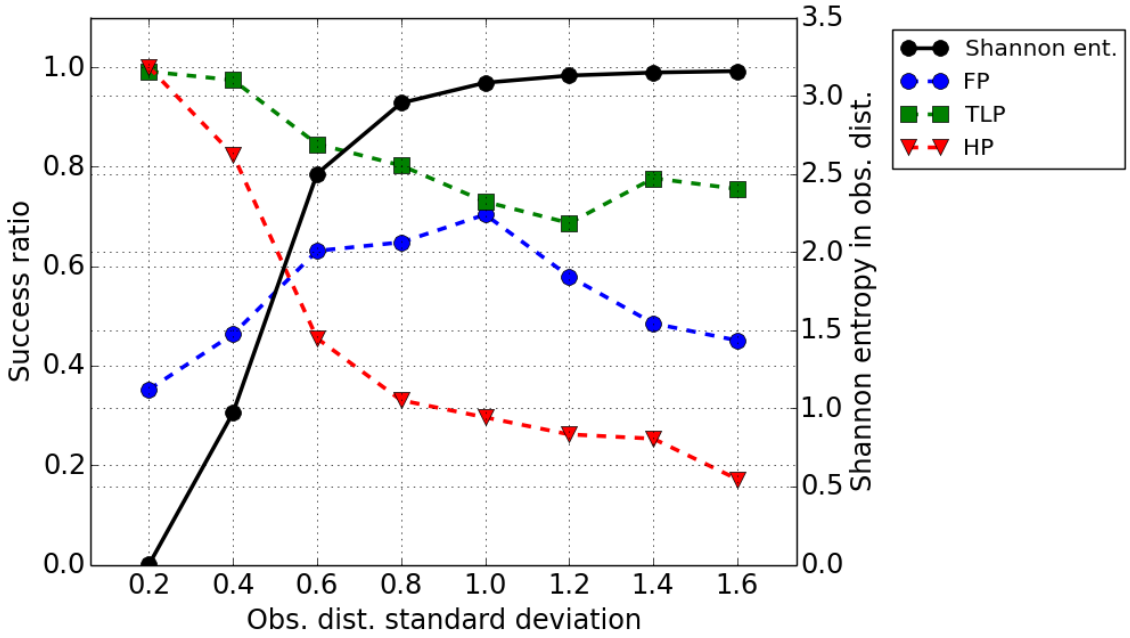

Figure 5.2: Success ratio (left axis) and Shannon entropy in Gaussian kernel (right axis) in experiment 1.

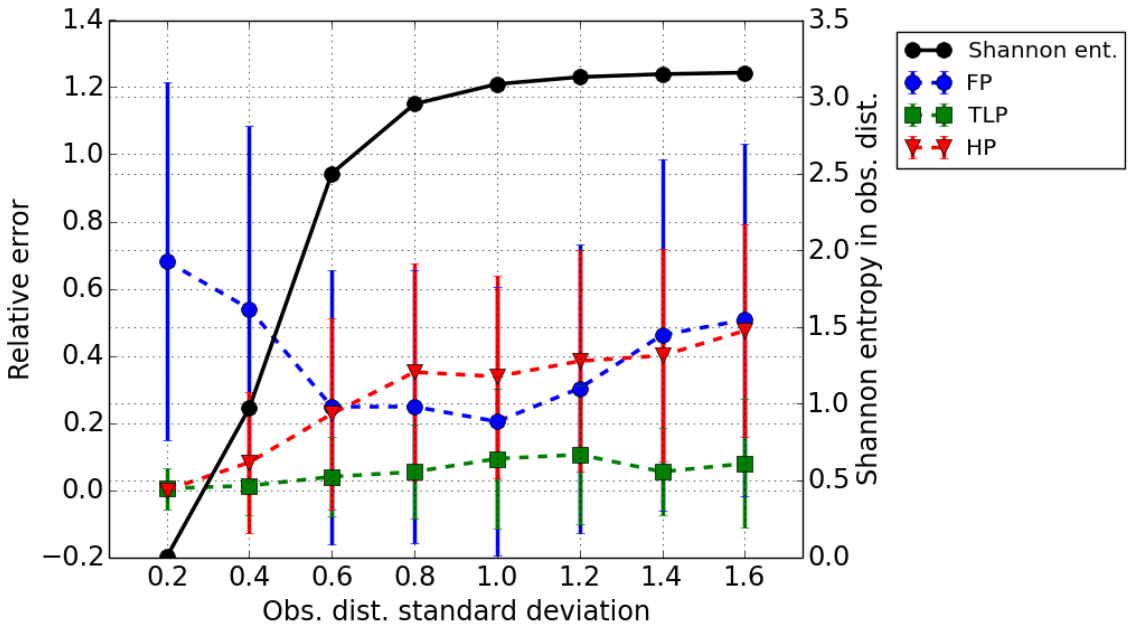

Figure 5.3: Average relative error (left axis) and Shannon entropy in Gaussian kernel (right axis) in experiment 1.

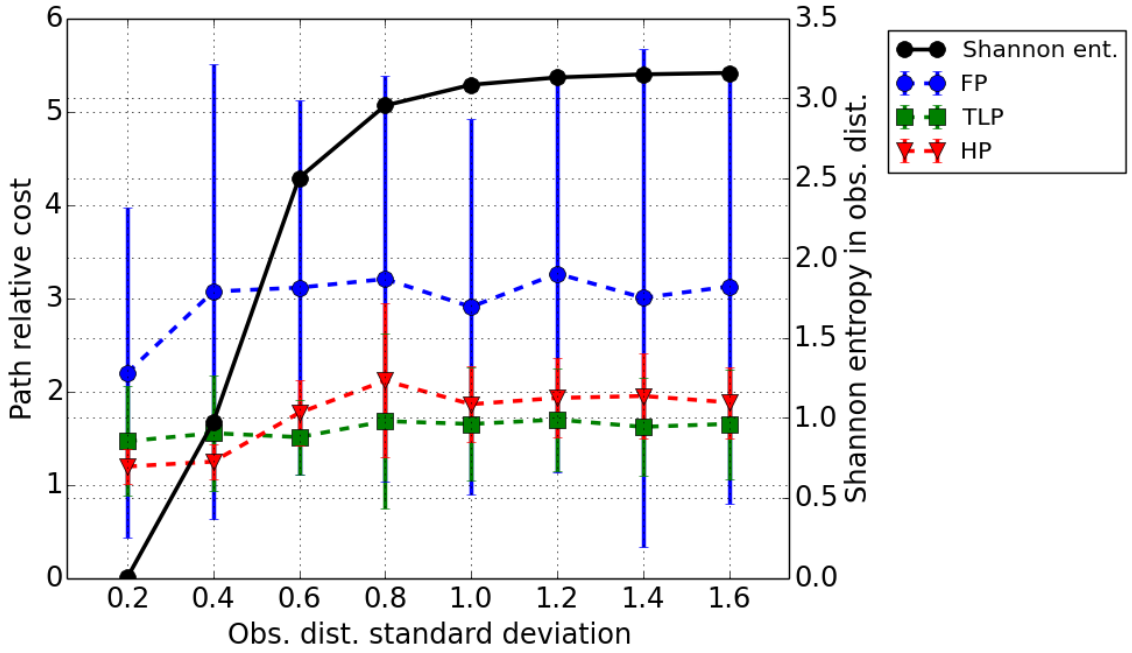

Figure 5.4: Average path relative cost scores (left axis) and Shannon entropy in Gaussian kernel (right axis) in experiment 1.

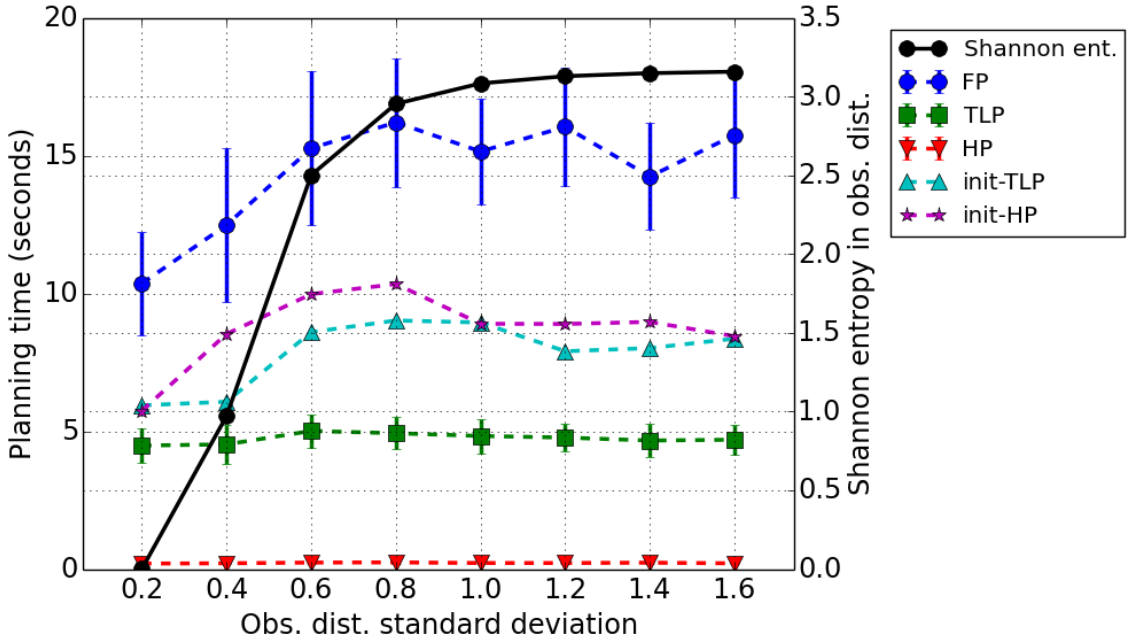

Figure 5.5: Average planning time (left axis) and Shannon entropy in Gaussian kernel (right axis) in experiment 1, where init-TLP and init-HP stand for the time required for the initialization phases of TLP and HP, respectively.

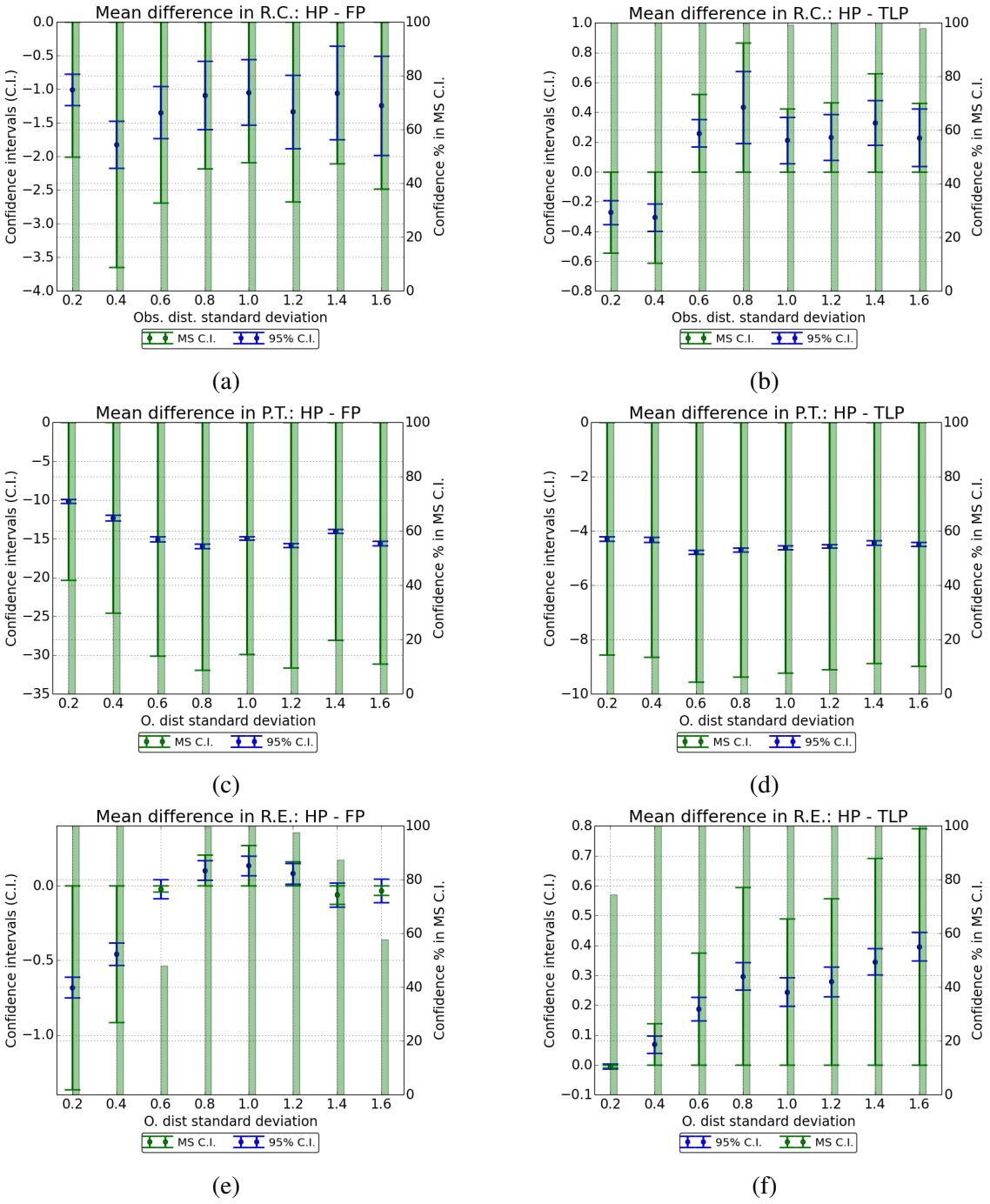

Figure 5.6: Confidence intervals (C.I.) for the mean differences (HP - FP) (left column) and (HP - TLP) (right column), in experiment 1, of the path relative cost (R.C.), planning time (P.T.), and relative error (R.E.), where blue intervals have 95% confidence, while the green ones are the most significant (M.S.) intervals (the widest one that does not include 0 in it). Left axes are for intervals and right axes for the % of confidence of the M.S. intervals.

5.4 Experiment 2

Experimental setting:

- Number of buildings: 2
- Subsection dimensions: 4
- Room dimension: 2
- Building dimension: 2
- Standard deviation: [0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6]
- Amount of simulations: 233

5.4.1 Objective

To determine if by increasing the state space of abstract actions modeled with concrete elements, the effectiveness of HP increases with respect to when smaller state spaces are defined.

5.4.2 Hypothesis

If the state uncertainty increases and an HP $H1$ uses larger subsections than another HP $H2$, then $H1$ will consistently behave as equal or more effectively than $H2$.

5.4.3 Results

In Figs. 5.7, 5.8, 5.9, and 5.10 the success ratio and average scores on relative error, path relative cost, and planning time for FP, TLP, and HP obtained in each experimental configuration in experiment 2 are shown. In addition, in Fig. 5.11 the 95% and most significant (widest interval that does not include 0) confidence intervals for the mean differences found between HP and the two baselines are shown.

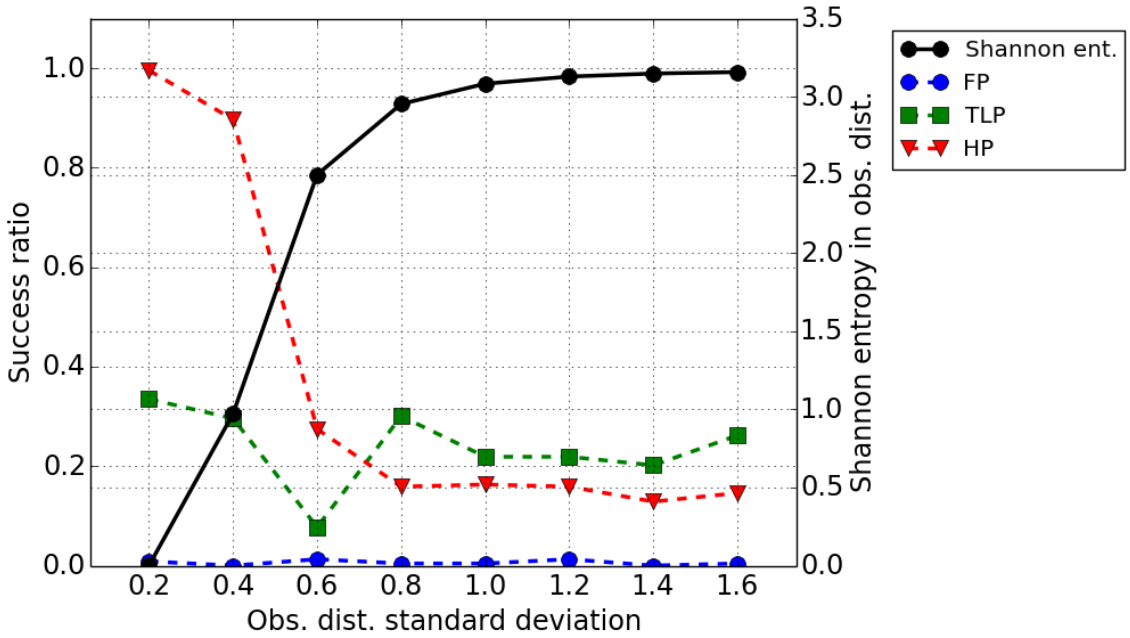

Figure 5.7: Success ratio (left axis) and Shannon entropy in Gaussian kernel (right axis) in experiment 2.

From Figs. 5.7 and 5.8, it is shown that for the first three trials HP obtained the largest success ratio, but it moved to the second best for the remaining trials. However, in terms of relative error, HP showed the lowest score in the overall set of trials, whereas FP reported the lowest success ratio over the whole experiment and the largest relative error for most of the trials. With regards to efficiency, HP showed the lowest value for path relative cost for most of the trials, while the lowest planning time in the whole experiment. It is worth noting that the confidence intervals for the differences between HP and FP with respect to the path relative cost (see Fig. 5.11a) could not be computed, since FP did not reach the goal state more than once.

Given that the version of HP with larger subsections failed to report a higher success ratio than the version with smaller subsections across the full set of experimental configurations, it is concluded that there is insufficient evidence to support the following hypothesis: *If the state uncertainty increases and an HP $H1$ uses larger subsections than another HP $H2$, then $H1$ will consistently behave as equal or more effectively than $H2$.* Moreover, in section 5.8 a discussion that covers the results found in each experiment is presented.

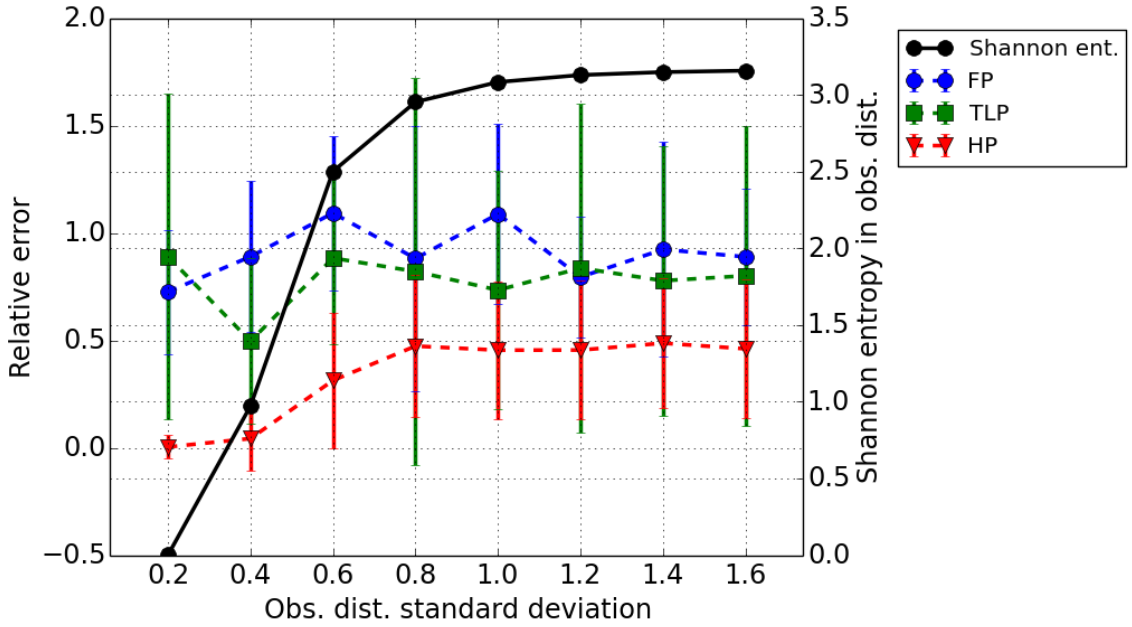

Figure 5.8: Average relative error (left axis) and Shannon entropy in Gaussian kernel (right axis) in experiment 2.

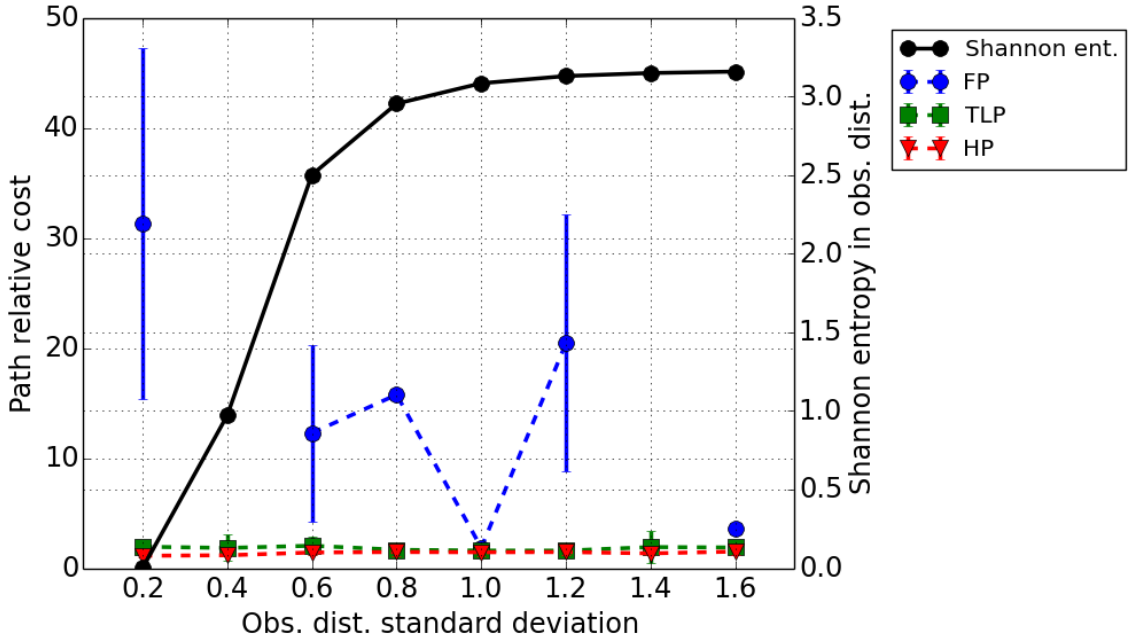

Figure 5.9: Average path relative cost scores (left axis) and Shannon entropy in Gaussian kernel (right axis) in experiment 2.

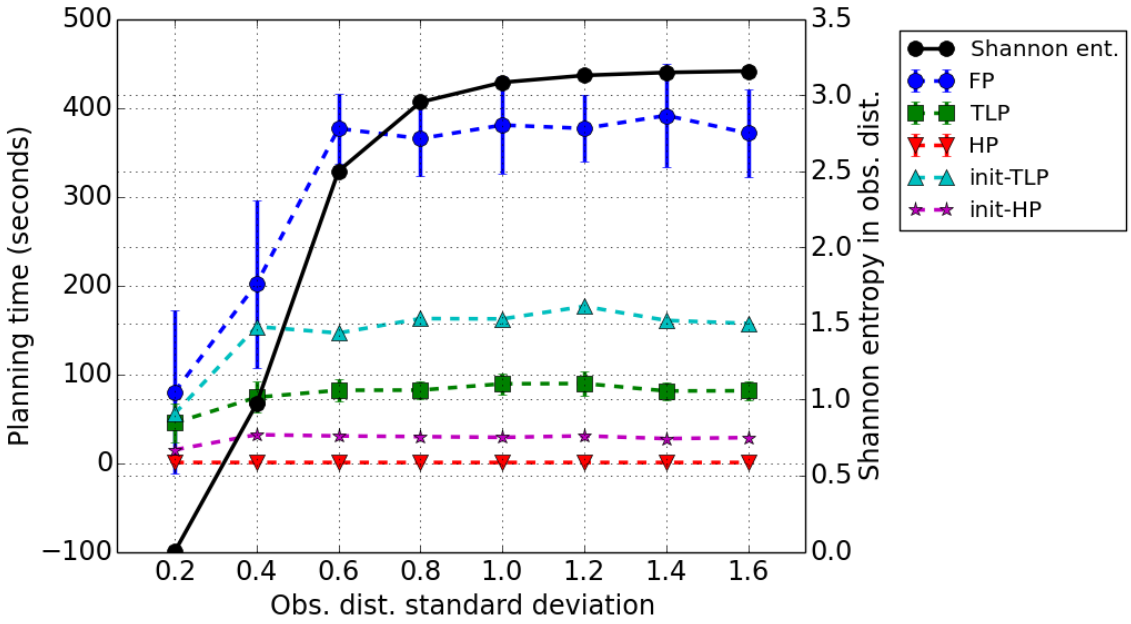

Figure 5.10: Average planning time (left axis) and Shannon entropy in Gaussian kernel (right axis) in experiment 2, where init-TLP and init-HP stand for the time required for the initialization phases of TLP and HP, respectively.

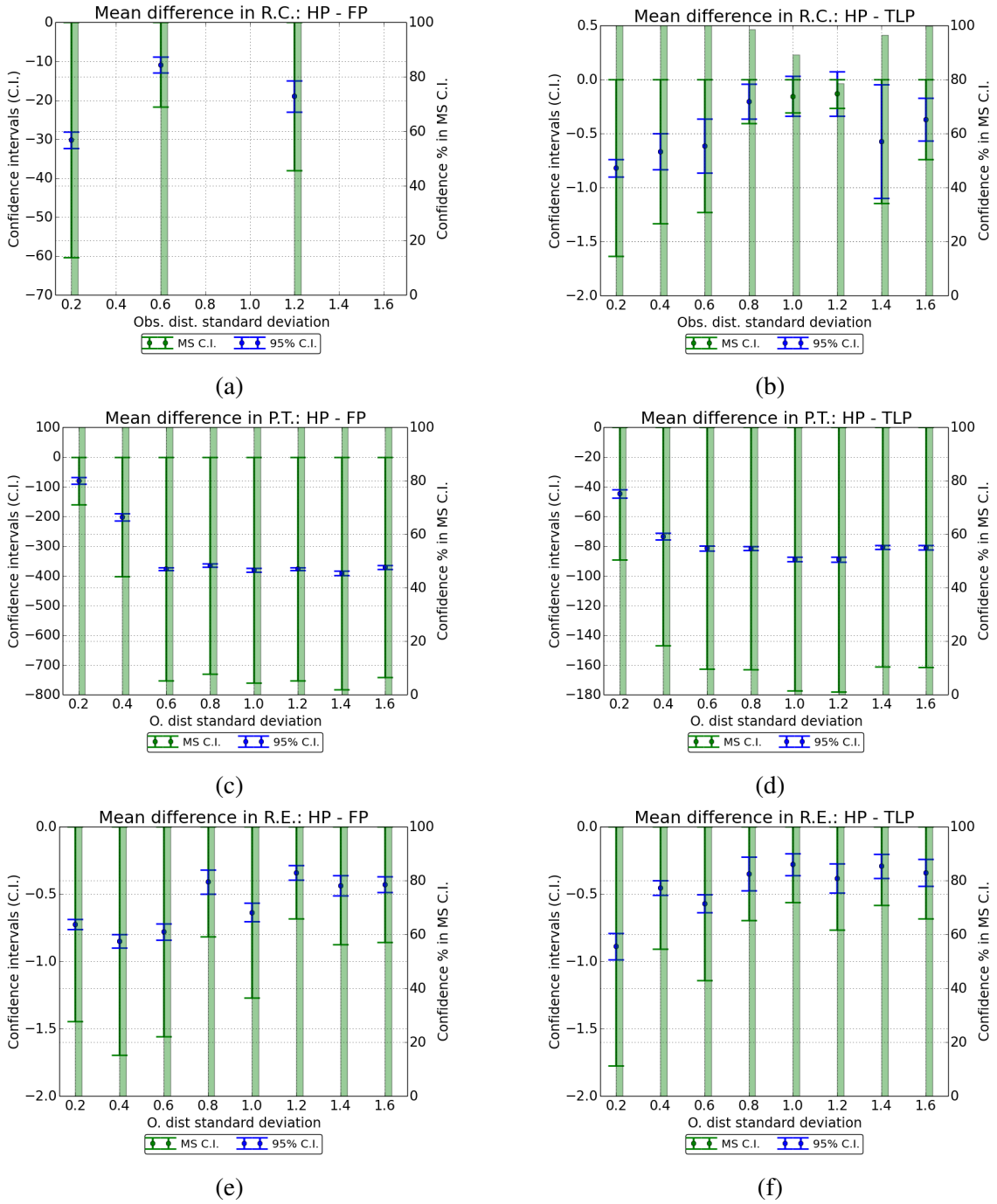

Figure 5.11: Confidence intervals for the mean differences (HP - FP) (left column) and (HP - TLP) (right column), in experiment 2, of the path relative cost (R.C.), planning time (P.T.), and relative error (R.E.), where blue intervals have 95% confidence, while the green ones are the most significant (M.S.) intervals (the widest one that does not include 0 in it). Left axes are for intervals and right axes for the % of confidence of the M.S. intervals.

5.5 Experiment 3

Experimental setting:

- Number of buildings: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
- Subsection dimensions: 2
- Room dimension: 2
- Building dimension: 2
- Standard deviation: 0.2
- Amount of simulations: 525

5.5.1 Objective

To determine if the magnitude of the environment's size decreases the effectiveness and efficiency of HP as much as it does to TLP.

5.5.2 Hypothesis

If the state space size increases, then HP will consistently behave as equal or more effectively, and more efficiently, than a TLP.

5.5.3 Results

In Figs. 5.12, 5.13, 5.14, and 5.15 the success ratio and average scores on relative error, relative error, and planning time for FP, TLP, and HP obtained in each experimental configuration in experiment 3 are shown. In addition, in Fig. 5.16 the 95% and most significant (widest interval that does not include 0) confidence intervals for the mean differences found between HP and TLP for experiments 3 and 4 are shown.

From Figs. 5.12 and 5.13, it is shown that for the first half of trials, HP obtained a near perfect performance, however, from the 10^{th} trial its success ratio dropped to, and remained in, the bottom of the scale. As for TLP, for the first two trials it obtained the same performance than HP, nevertheless, despite it showed several ups and downs, it never reported a success ratio as low as HP did for the second half. With regards to efficiency, Figs. 5.14 and 5.15 show that for the first half of trials, HP obtained the lowest value for path relative cost, while for the second half it could not be calculated, since HP never reached a goal cell. Furthermore, the initial planning for both HP and TLP grew linearly with respect to the amount of buildings, while their average iterative planning time (the one that is not the initial planning time) remained constant, where HP reported the lowest planning time over the whole experiment. It is worth noting that the confidence intervals for the differences between HP and TLP with respect to the path relative cost (see Fig. 5.16a) could not be computed, since HP did not reach the goal state even once for the second half of the experimental configurations.

Given that HP failed to report a higher success ratio and, simultaneously, a lower planning time than TLP over the full set of experimental configurations, it is concluded that there is insufficient evidence to support the following hypothesis: *If the state space size increases, then HP will consistently behave as equal or more effectively, and more efficiently, than a TLP.* Moreover, in section 5.8 a discussion that covers the results found in each experiment is presented.

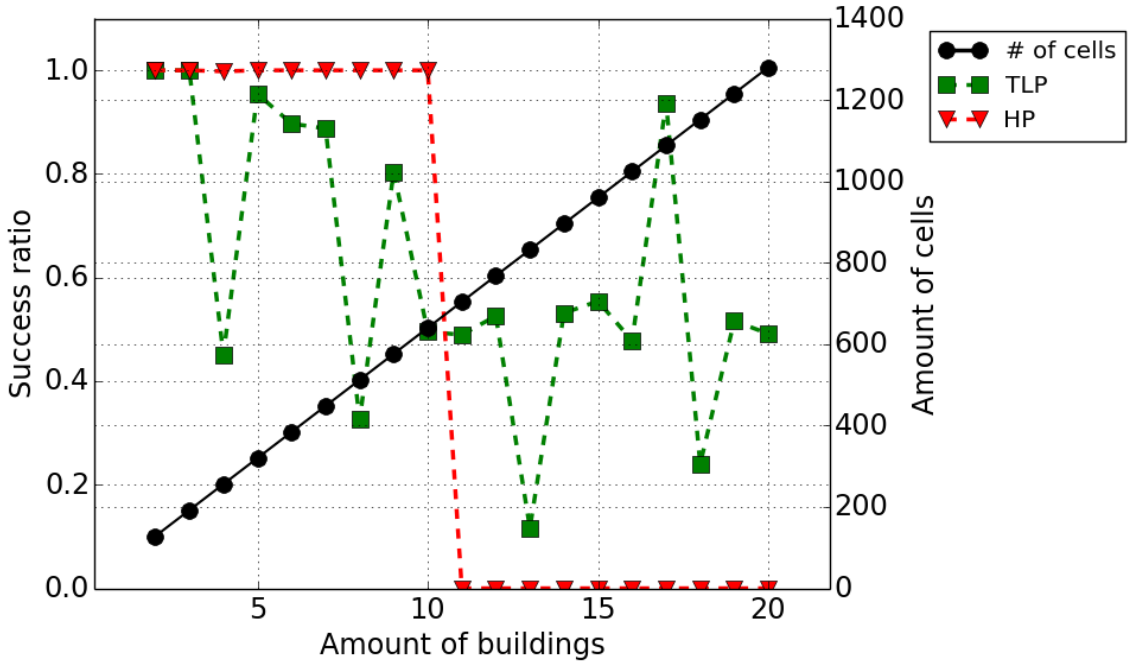

Figure 5.12: Success ratio (left axis) and total amount of cells in the environment (right axis) in experiment 3.

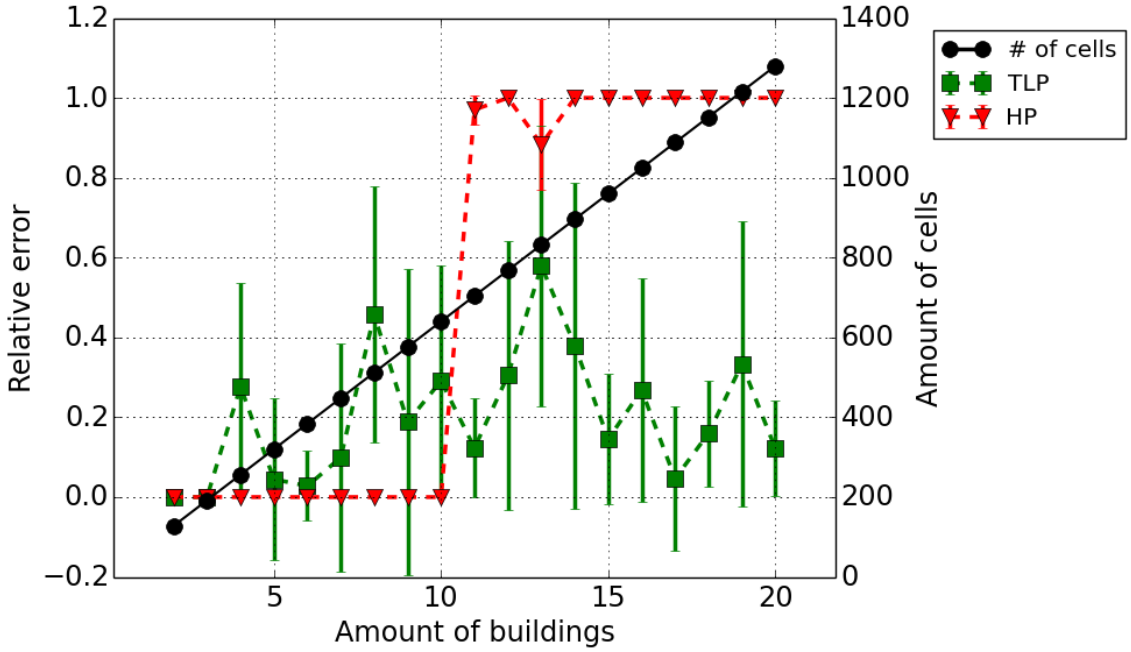

Figure 5.13: Average relative error (left axis) and total amount of cells in the environment (right axis) in experiment 3.

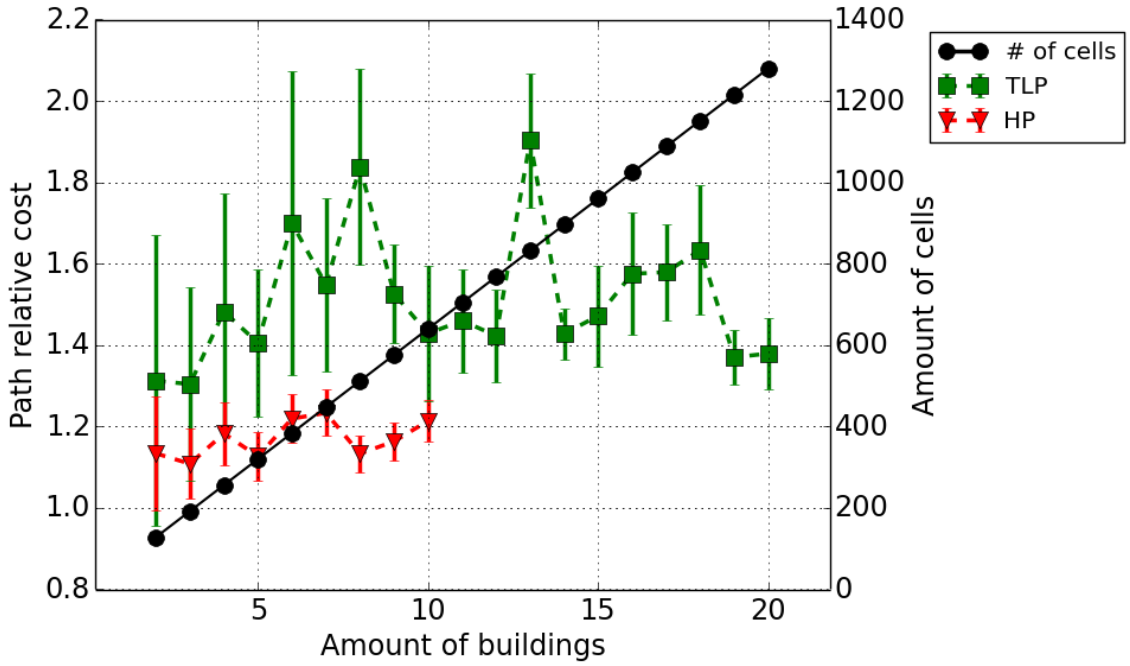

Figure 5.14: Average path relative cost scores (left axis) and total amount of cells in the environment (right axis) in experiment 3.

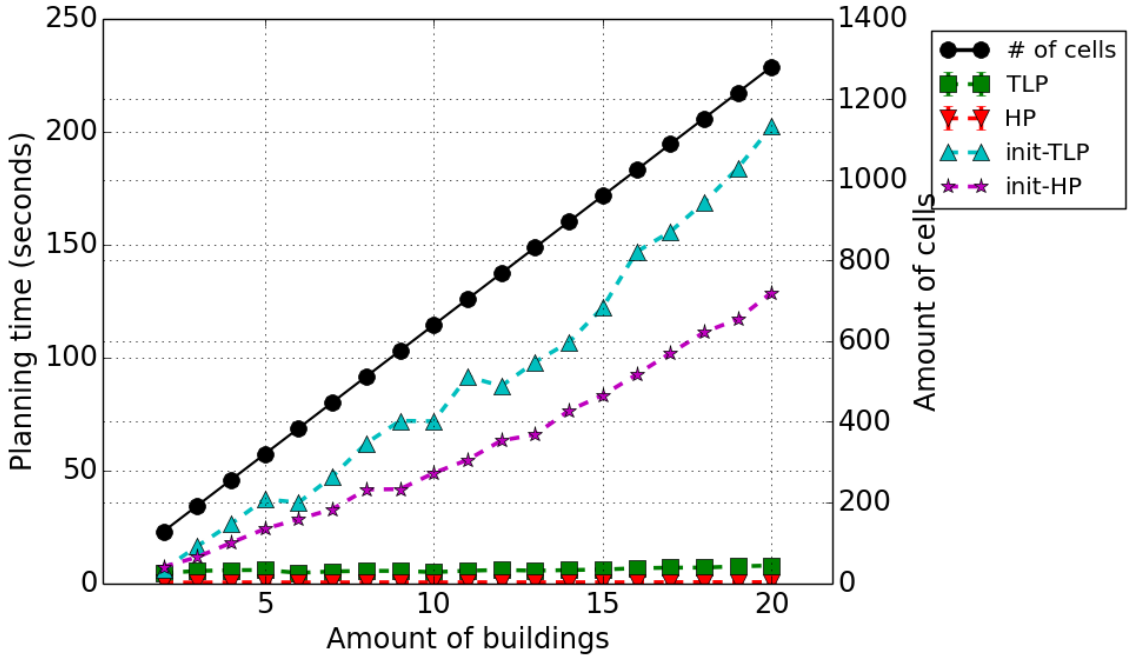

Figure 5.15: Average planning time (left axis) and total amount of cells in the environment (right axis) in experiment 3, where init-TLP and init-HP stand for the time required for the initialization phases of TLP and HP, respectively.

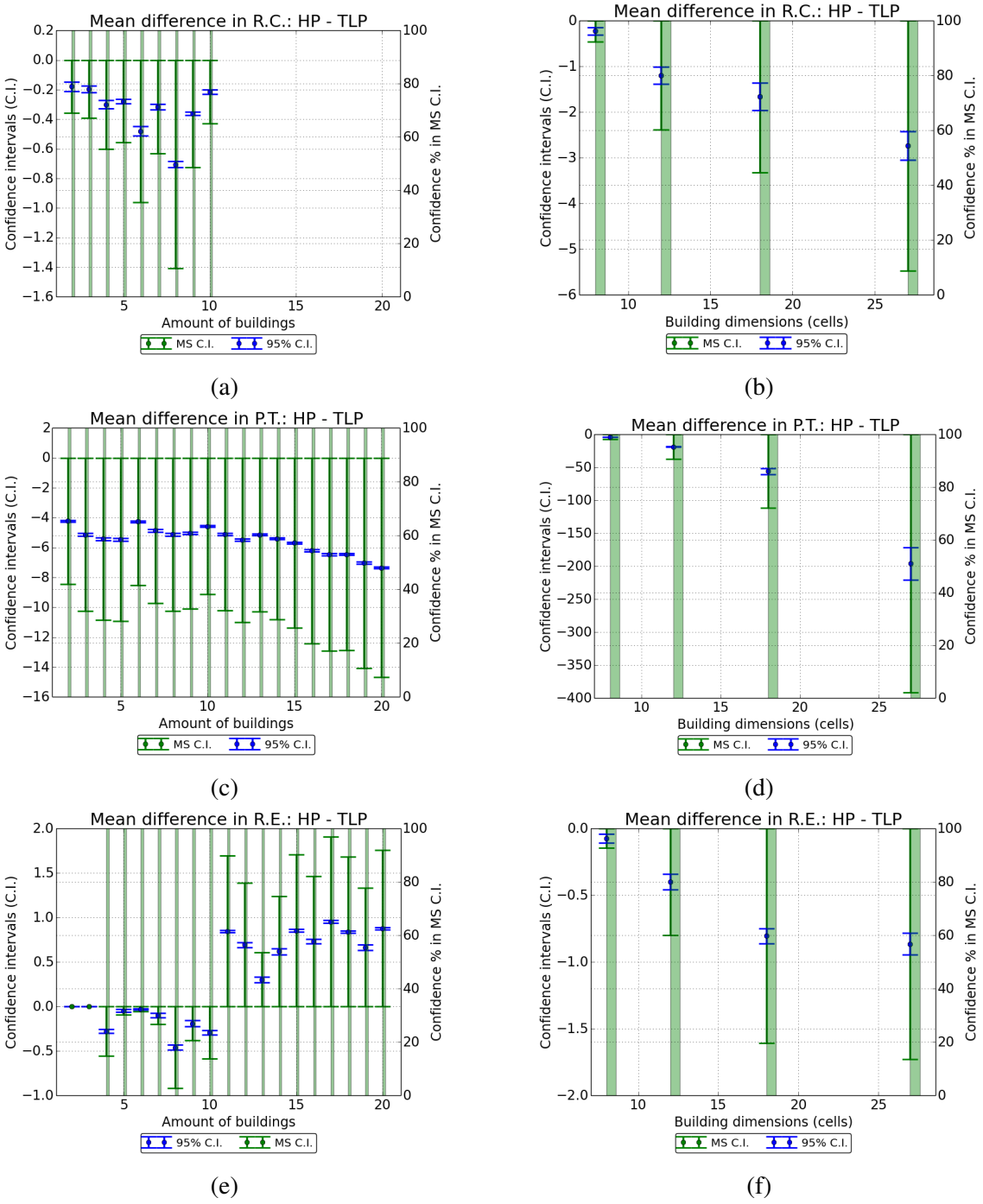

Figure 5.16: Confidence intervals for the mean differences (HP - TLP) in experiment 3 (left column) and experiment 4 (right column) of the path relative cost (R.C.), planning time (P.T.), and relative error (R.E.), where blue intervals have 95% confidence, while the green ones are the most significant (M.S.) intervals (the widest one that does not include 0 in it). Left axes are for intervals and right axes for the % of confidence of the M.S. intervals.

5.6 Experiment 4

Experimental setting:

- Number of buildings: 2
- Subsection dimensions: [2, 3, 3, 3]
- Room dimension: [2, 2, 3, 3]
- Building dimension: [2, 2, 2, 3]
- Standard deviation: 0.2
- Amount of simulations: 233

5.6.1 Objective

To determine if by increasing the environment's size in a way that does not increase the amount of abstract states in TLP, but does in HP, increases the difference between HP and TLP in terms of effectiveness and efficiency.

5.6.2 Hypothesis

If the subsections, rooms and buildings' dimensions increase, then the difference in effectiveness and efficiency between HP and TLP will increase.

5.6.3 Results

In Figs. 5.17, 5.18, 5.19, and 5.20 the success ratio and average scores on relative error, relative error, and planning time for TLP, and HP obtained in each experimental configuration in experiment 4 are shown. In addition, in Fig. 5.16 the 95% and most significant (widest interval that does not include 0) intervals for the mean differences found between HP and TLP for experiments 3 and 4 are shown.

Figs. 5.17 and 5.18 show that, in terms of effectiveness, HP obtained a near perfect performance over the full set of experimental configurations, while TLP showed a good performance in the first trial, and considerably decreased towards the following configurations. With regards to the efficiency, as shown in Figs. 5.19 and 5.20, TLP reported the highest path relative cost and planning time in every trial. Moreover, unlike the planning time results obtained in experiment 3 (see section 5.5), that maintained a constant behavior across the full set of trials, Fig. 5.20 shows that TLP's planning time not only grows, but it does to such magnitude that even HP's initial planning time is lower than TLP's average planning time in the last three configurations.

Given that HP succeeded in constantly reporting a success ratio and planning time that kept moving away from the ones reported by TLP, as the size of the environment increased, it is concluded that there is sufficient evidence to support the following hypothesis: *If the subsections, rooms and buildings' dimensions increase, then the difference in effectiveness and efficiency between HP and TLP will increase.* Moreover, in section 5.8 a discussion that covers the results found in each experiment is presented.

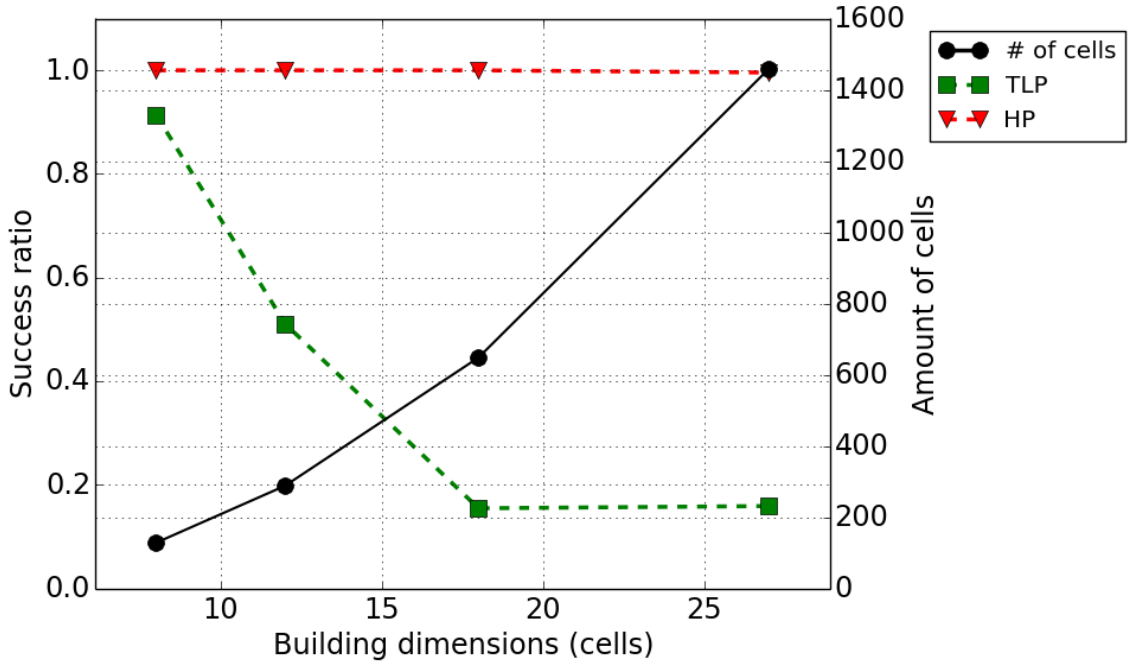

Figure 5.17: Success ratio (left axis) and total amount of cells in the environment (right axis) in experiment 4.

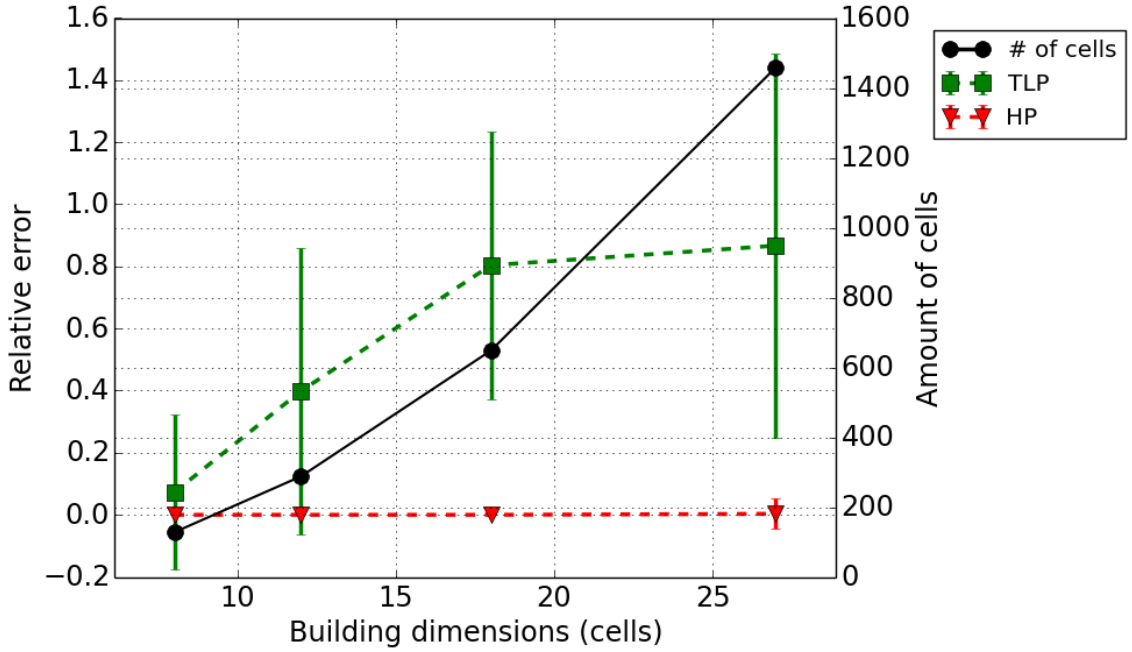

Figure 5.18: Average relative error (left axis) and total amount of cells in the environment (right axis) in experiment 4.

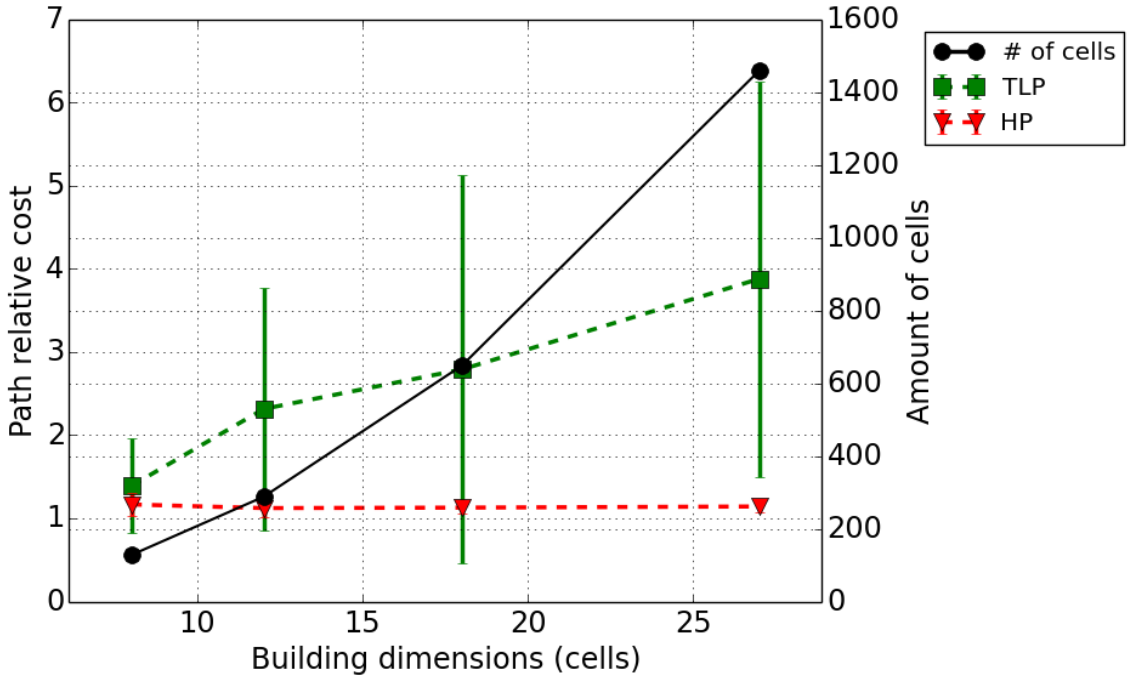

Figure 5.19: Average path relative cost scores (left axis) and total amount of cells in the environment (right axis) in experiment 4.

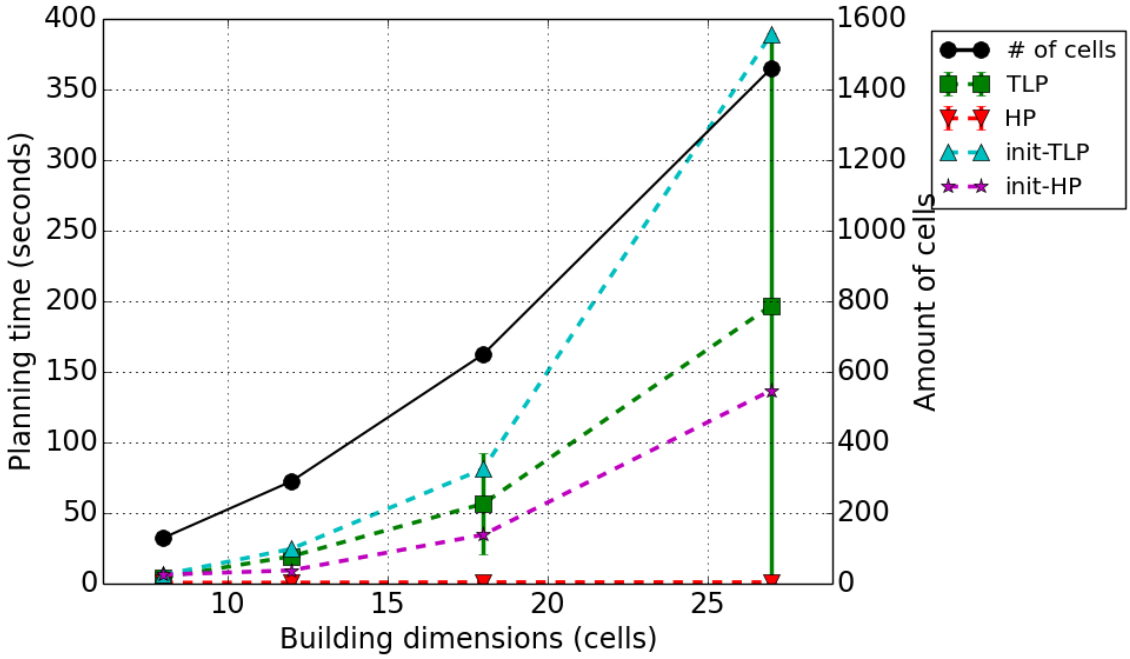

Figure 5.20: Average planning time (left axis) and total amount of cells in the environment (right axis) in experiment 4, where init-TLP and init-HP stand for the time required for the initialization phases of TLP and HP, respectively.

5.7 Experiment 5

5.7.1 Objective

In order to gain insight on how the proposed architecture would behave in a real system, such as the RB1 service robot[1] available in the robotics laboratory in INAOE, in this experiment the error of the RB1's localization system is estimated and compared to the experimental configurations employed in the previous experiments. In this way, the uncertainty in the robot's localization system can be used to extrapolate the results found in simulation to the real world.

5.7.2 Description

Currently, the RB1 robot uses the 2D SLAM implementation of the Active Vision Group [Seib et al., 2016]. This implementation enables the robot to generate a description of the environment (known as map) to locate itself within it, as shown in Fig. 5.21. To measure the error in the robot's 2D SLAM system, a set of five VICON VANTAGE cameras[2] were set in the room to track the robot's position and serve as ground truth, because of their high precision.

To estimate the error in the robot's 2D SLAM system, the following steps were performed:

1. A map of a section of the robotics laboratory, from INAOE, was generated. Then, a collection of ten starting points and a target point were set in the map, as shown in Fig. 5.21.

2. A set of four markers were mounted on the robot's head, in this way, the VICON cameras could detect its position within their range of view at any time, as shown in Figs. 5.22 and 5.23.

[1] https://www.robotnik.eu/manipulators/rb-one/
[2] https://www.vicon.com/hardware/cameras/vantage/

3. The robot was commanded to move (once) from each starting point to the target point (using the navigation system also implemented by [Seib et al., 2016]). At the end of every run, the robot's position reported by the VICON system ($C^i_{x,y}$) was registered.

4. The standard deviation was computed for the set of final positions $C = \{C^1_{x,y}, ..., C^i_{x,y}, ..., C^{10}_{x,y}\}$ using Eq. 5.6.

$$\sigma = \sqrt{\frac{1}{10-1}\sum_{i=1}^{10}\left(EucDist(C^i_{x,y}, Avg_{x,y})\right)^2} \tag{5.6}$$

where $Avg_{x,y}$ is the average coordinate from the positions in C and $EucDist(\cdot,\cdot)$ is a function that returns the Euclidean distance between two points.

5.7.3 Results

After performing the ten runs, a standard deviation of $0.0993673 \approx 0.1$ (meters) was obtained. Furthermore, given that in Eq. 5.1 (which models the observation distribution) the distance of a cell to the center of the kernel is expressed in standard deviations, and assuming that the error in the robot's localization system follows a Gaussian distribution, then it is possible to match an observation distribution, from previous experiments, to a setting for the RB1 robot.

In experiment 1, it was shown that the proposed architecture obtained success ratio scores > 0.8, for standard deviation values of 0.2 and 0.4 (see Fig. 5.2). In order to extrapolate the performance of HP observed in simulation, instead of modifying RB1's standard deviation (≈ 0.1 meters), the distances by which adjacent cells should be separated to replicate the observation distribution for standard deviation values of 0.2 and 0.4 (shown in Fig. 5.24) were computed.

Eq. 5.8 (derived from Eq. 5.7 which is a variation of Eq. 5.1 that was employed to compute the probability values in the observation distribution, but omits the normalization factor η) computes the euclidean distance (r in standard deviations)

between the center of a Gaussian built with σ standard deviations, and a point in space that has a probability of p.

Let $G_{0.2}$ and $G_{0.4}$ be two Gaussian functions with σ values of 0.2 and 0.4, respectively (shown in the left and middle columns in Fig. 5.24). Then, $p_{0.2}$ and $p_{0.4}$ are probabilities of a point located at 1 standard deviation apart from the center of $G_{0.2}$ and $G_{0.4}$.

$$p = \frac{e^{\frac{-r^2}{2\sigma^2}}}{2\pi\sigma^2} \tag{5.7}$$

$$r = \sqrt{-2\sigma^2 \ln(2\pi p\sigma^2)} \tag{5.8}$$

Using $p_{0.2}$ and $p_{0.4}$, with $\sigma = 1$, in Eq. 5.8 the distance values of $r_{0.2} \approx 4.31$ and $r_{0.4} \approx 1.61$ were computed, which are expressed in standard deviations. By multiplying $r_{0.2}$ and $r_{0.4}$ by 0.1 (which is RB1's actual standard deviation), $d_{0.2} = 0.431$ and $d_{0.4} = 0.161$ were obtained (in meters). Therefore, for a standard deviation of 0.1 meters in its localization system, the RB1 robot (with HP in charge of planning) is expected to behave similarly to how HP did in those configurations with a standard deviation of 0.2 and 0.4, by descretizing environments into a grid of adjacent cells separated by 0.431 and 0.161 meters, respectively.

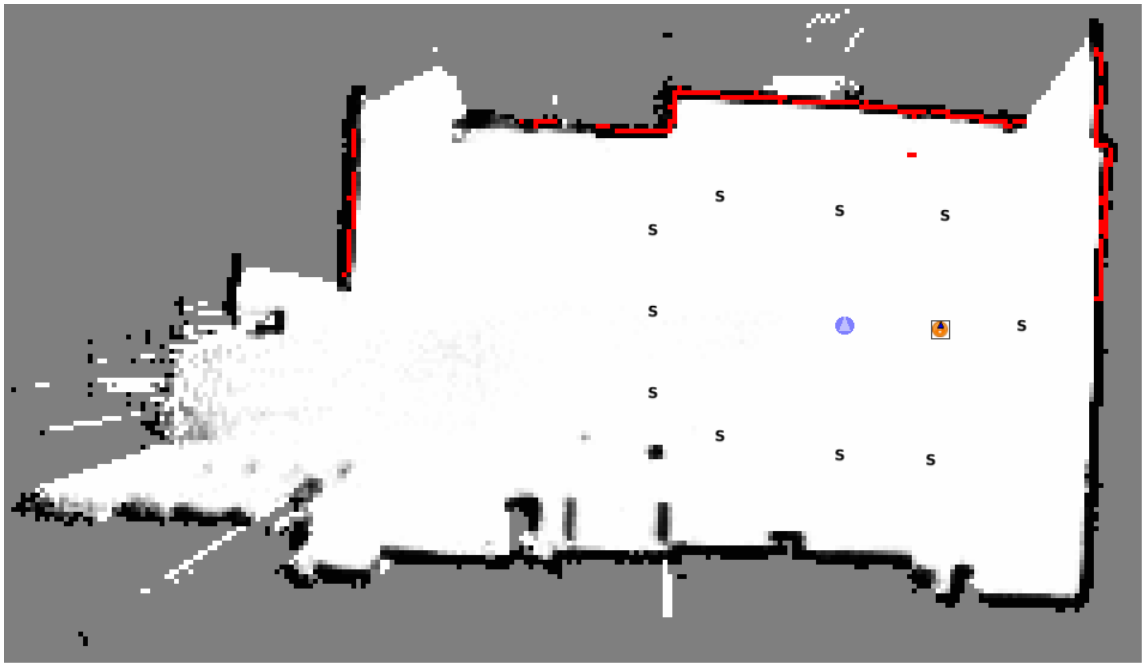

Figure 5.21: Map of a section of the robotics laboratory in INAOE. The map was generated using a laser sensor and the 2D SLAM implementation of the Active Vision Group [Seib et al., 2016]. In this map, the yellow marker corresponds to the target location that the robot was instructed to go, the letters "S" indicate the 10 starting points from which the robot moved to the target, and the purple circle shows the robot's location at the moment the picture of the map was taken.

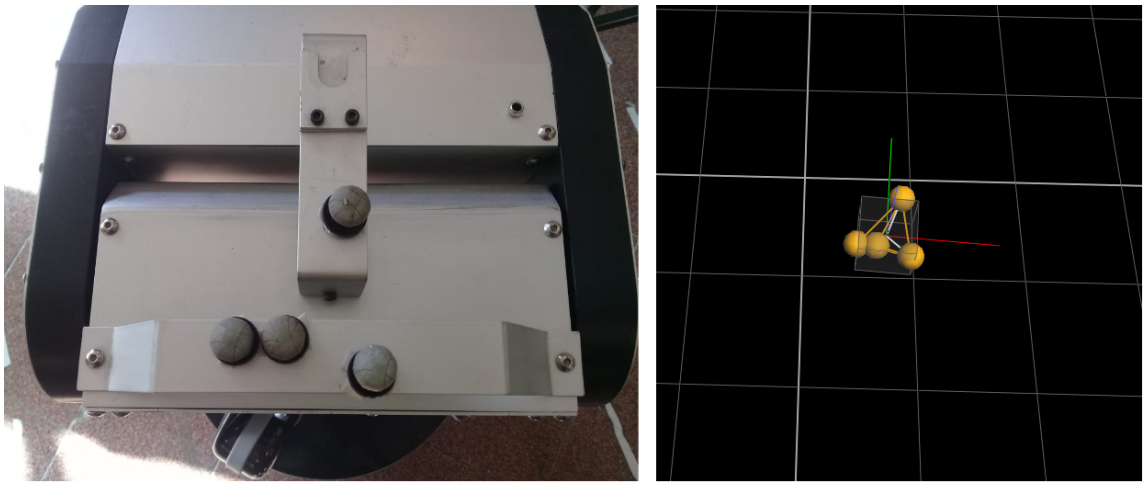

Figure 5.22: The image on the left shows the arrangement of markers set on the robot's head, so that the VICON cameras could detect them at any moment. On the right side are shown the markers being tracked, from which the VICON system creates a coordinate system that represents the robot's true position.

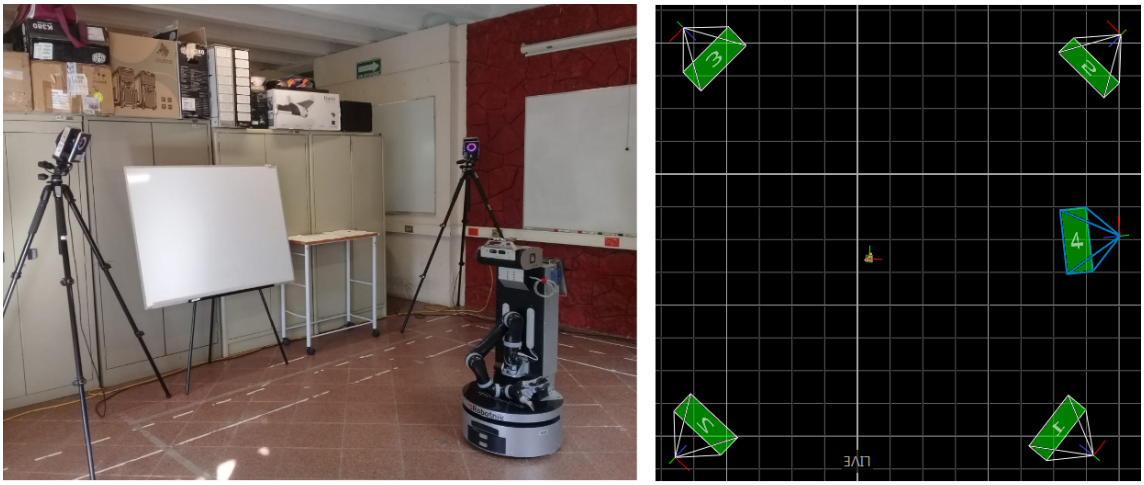

Figure 5.23: The image on the right side shows the robot's position being tracked within the robotics laboratory, which corresponds to its location in the image on the left.

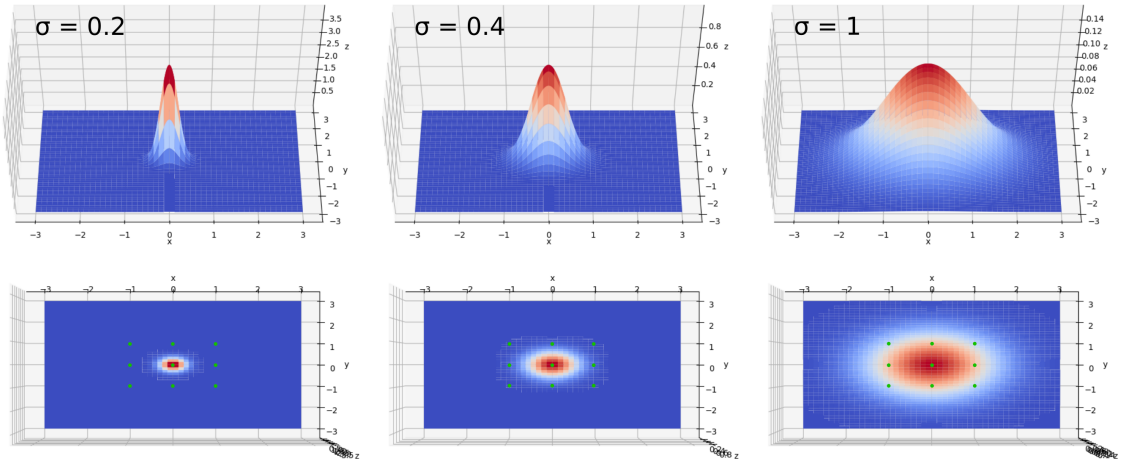

Figure 5.24: Two dimensions Gaussian probability density function plots with different standard deviation values (from left to right: 0.2, 0.4 and 1 standard deviations, respectively). Plots in the bottom row correspond to the top view of the plots in the row on the top. The green dots represent the center of the cells in a 3×3 kernel (employed to model the agent's observation distribution in previous experiments), which are horizontally and vertically separated by 1 standard deviation.

5.8 Discussion

In experiment 1, as the differences found with respect to time planning (illustrated in Figs. 5.6c and 5.6d) suggest, HP is significantly more efficient in planning time regardless of the agent's state uncertainty. With regards to path relative cost, HP reports the lowest value only for the first two experimental configurations and from the third one it ranks in second place. As for effectiveness, unlike the baseline methods, HP is the only one that consistently decreases its success ratio and increases its relative error as entropy increases. Also, the Shannon entropy seems to have a proportional effect in the success ratio of HP (as Fig. 5.2 shows), as both show a steep slope in the interval [0.4,0.6], and slowly and consistently converge.

Thus, because of the trends shown by the success ratio and differences between HP and TLP in terms of relative error (see Figs. 5.2 and 5.6f), results in experiment 1 do not support the hypothesis of HP consistently behaving as effectively and more efficiently than FP and TLP. Therefore, it is safe to state that the performance of HP suffers the most, compared to the baseline methods. Given that HP executes concrete actions through policies computed to model abstract actions, and the set of states over which an abstract action invokes actions is a subset of the environment's state space (as Eq. 4.15 specifies), there is a possibility that the agent transits to a state that is not in the abstract action's state space.

Furthermore, as the entropy in the observation distribution increases, so does the probability for the agent to perceive observations that do not represent its true state. Hence, by combining a high degree of uncertainty of its current state with a small state space, in HP, it is likely that the agent will perform actions that take it out of its state space before it reaches the goal state. Moreover, despite that the agent might also take sub-optimal actions in FP and TLP, due to the large state space in which it operates, it has a larger margin of error to get back in track towards its goal state. Therefore, the smaller margin for error would explain why HP was the method with the lowest success ratio for most of the trials, and was the motivation for experiment 2, which evaluates if by enlarging tha abstract actions' state space, HP reports a better success ratio.

The success ratio and planning time results for HP in experiment 2 support two observations (that also appeared in results from experiment 1): first, uncertainty does not appear to have any effect in planning time of HP, and second, uncertainty seems to have a direct negative effect in HP's success ratio. As Fig. 5.10 shows, HP has a flat trace for both, initial and iterative planning times over the whole set of values for standard deviation. Whereas in Fig. 5.7, HP shows again a trace that looks like a horizontally mirrored version of the Shannon entropy trace. Thus, even in a scenario with larger sub-sections, HP struggles to maintain the high success ratio score (reported in configurations with low entropy) in configurations with a Shannon entropy of 2.5 or higher. Furthermore, as Fig. 5.8 illustrates, similar to the relative error results from experiment 1, HP seems to stabilize at a relative error of 0.5. Therefore, since the success ratio and relative error performance of HP were pretty much replicated in experiment 2, this suggests that increasing the sub-section dimension does not mitigate the negative effects uncertainty has over HP. Thus, results from experiment 2 do not support the hypothesis that larger sub-sections improve the effectiveness of HP.

However, despite the performance of HP did not improve in a scenario of larger sub-sections, it did remain pretty similar to experiment 1, which cannot be said for FP and TLP. With regards of FP, its success ratio remained near to 0 across the full set of experimental configurations, whilst TLP suffered a significant drop in success ratio, compared to its results from experiment 1. Even though TLP reported a higher success ratio score than HP for most of the experimental configurations, in Figs. 5.8, 5.9, 5.11f, and 5.11b is shown that HP obtained the smallest relative error for every configuration and the smallest path relative cost for most of them. This seems to suggest that the environment's size does not affect the performance of HP, at least not as much it does to FP and TLP.

In experiment 3, HP showed the lowest initial and iterative planning (see Figs. 5.15 and 5.16c) across the full set of experimental configurations. With regards to the first half of the trials, HP reported a success ratio of 1 and a relative error of 0 for every configuration in the range $[2, 10]$ with exception of $amount\,of\,buildings = 3$. Furthermore, the path relative cost of HP, similar to its score on effectiveness, shows a consistent behavior (see Figs. 5.14 and 5.16a). However, the performance of HP completely drops for the second half of experimental configurations, while TLP

maintains the erratic behavior shown in the whole experiment. Therefore, given that HP completely fails, for the second half of trials, to reach the goal cell even once, then the results from experiment 3 do not support the hypothesis that HP consistently behaves as effectively and more efficiently than TLP as the state space size increases.

Recalling the description of failure criterion 2 for HP, it is caused by the agent transiting to a state that is not part of the current state space. As Fig. 5.25 shows, all of HP's failures occurred in the second half of the experimental configurations, were caused by failure of type 2. By having HP always failing because a not modeled state was reached, while simultaneously barely moving away from its starting position (as shown in Fig. 5.13), could be caused by policies (at the most abstract level in the SST, *i.e.* where buildings are states) consistently executing abstract actions that were built to be invoked from an abstract state different to the current one. Furthermore, in [Pineau et al., 2003] they state in the definition of the convergence and error bound for PBVI that with a denser sampling of belief points, the estimated value function converges to the true value function, which means that HP would benefit by using a set of belief points larger than the 75 employed in these experiments.

However, given that decrease in performance of HP occurred abruptly, it is likely that the horizon value, employed to compute every policy, caused the poor decision making at the buildings level in the SST. Since the pair of initial and goal states are sampled from the pair of buildings at the ends of the environment (see section 5.2.5), using a horizon of 10 would explain why the policy does not know what the best action is while the agent is 10 transitions away form the goal state.

Hence, in order to determine if any of our hypotheses can explain the sudden failure of HP in the scenario of 11 buildings, two additional configurations of HP were evaluated using: i) a set of 250 initial belief points, and ii) a horizon of 11. Both configurations were simulated the same amount of times than those in experiment 3 (525 runs), and the results are shown in Table 5.1, along with the results of the original configuration. At first, by comparing the results between the top and middle row, it is clear that by increasing the amount of initial belief points does improve

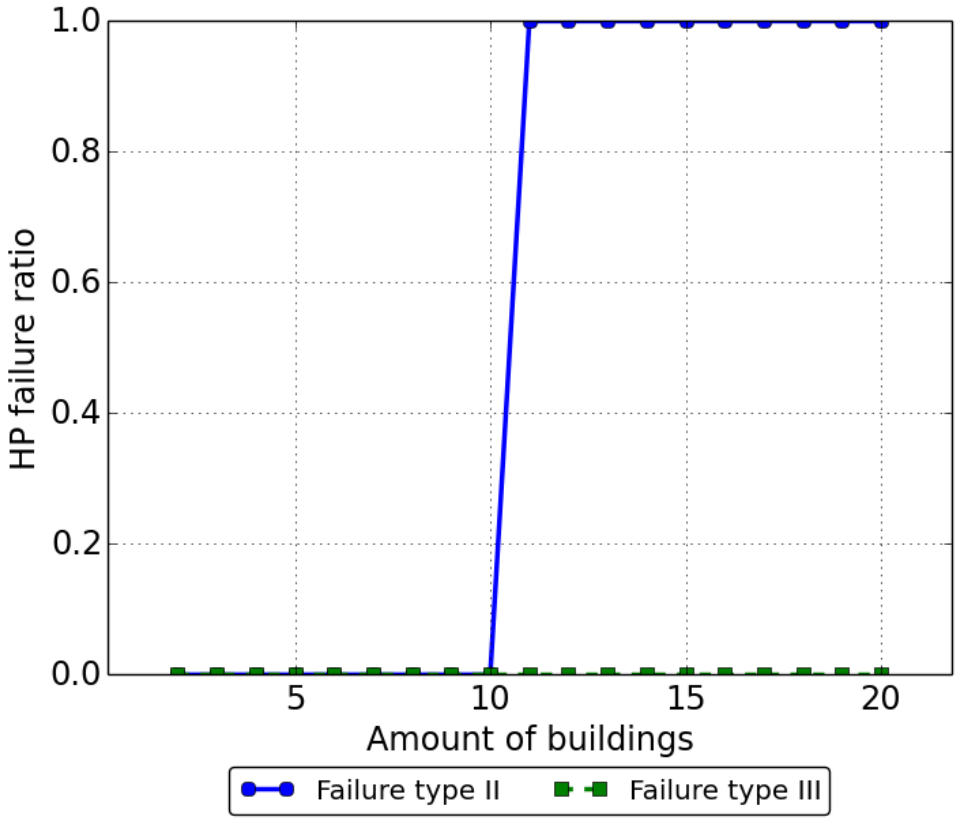

Figure 5.25: Proportion of simulations terminated as failed ones as a consequence of HP meeting cases of failure criteria 2 and 3, described in section 5.2.1.

Table 5.1: Comparison of the success ratio and average relative error, path relative cost and planning time, after 525 simulations, obtained by HP in an environment made of 11 buildings using different configurations of horizon and amount of belief points in PBVI, where the first row corresponds to the configuration used in experiment 3.

Amount of Buildings	# of Belief Points in PBVI	Horizon in PBVI	Success Ratio	Relative Error	Path Relative Cost	Planning Time (sec.)
11	75	10	0	0.97 ± 0.04	- -	0.35 ± 0.04
11	250	10	0	0.96 ± 0.03	- -	0.86 ± 0.16
11	75	11	1	0	1.15 ± 0.04	0.28 ± 0.05

the performance of HP, since it was not able to reach the goal cell not even once. Thus, it is safe to discard a greater amount of belief points as a possible solution. Nevertheless, with regards to the results in the bottom row HP obtained similar to those in experiment 3 up to the scenario of 10 buildings. These results suggest that, in fact, HP required of a larger horizon in order to successfully traverse from one end of the environment to the other. However, given that HP consumes an amount of time significantly smaller than FP and TLP, one could still afford to increase the horizon in the PBVI algorithm if necessary.

In experiment 4, HP significantly outperforms TLP in all four evaluation metrics, as it reported success ratio scores of 1, 1, 1 and 0.99. Moreover, considering the fact that the state space of the fourth configuration (1458 cells) is larger than the last configuration in experiment 3 (1280 cells) gives insight on how large state spaces with a high degree of connectivity among its states are still manageable, if the information about those connections is available, which is the main idea behind the proposed architecture in this document: to take advantage of knowledge about domestic environments a designer might have, to enable a service robot to perform its tasks in an efficient and reliable way.

From experiments 1 and 2, the proposed architecture showed to be more susceptible to fail reaching the goal state than FP and TLP, as the uncertainty in the agent's state increased, however, as results from experiment 5 indicate, in a real setting a robot can behave reliably if the space is discretized into a grid of cells separated by at least 0.25 meters, which seems to be a feasible condition for most service robotics applications. With regards to experiment 3, HP was exposed to be sensitive to the horizon employed to solve its policies. Whereas experiment 4 exhibited the capacity

of HP to exploit known structures of the domain and perform near optimal in such environments. Although the proposed architecture has turned out to be a feasible task planning method for large environments, whose structure is known and show a low degree of state uncertainty, it still suffers from several shortages.

The proposed architecture could benefit from using larger horizons and sets of belief points to compute its policies, as the planning time reported in these experiments are still way lower than both baselines. However, the main reason HP assumes the initial state is known with certainty, and why it performed so bad in experiments 1 and 2, is due to its inability to recover from scenarios with a considerable amount of uncertainty, unlike a standard POMDP. Therefore, by extending the algorithms that build and execute hierarchical policies (Algorithms 2 and 3, respectively) to handle uncertainty at abstract levels better than it currently does, the proposed architecture would benefit the most, as it already manages well large spaces with low state uncertainty.

5.9 Chapter Summary

In this chapter, a set of four experiments designed to evaluate the shortcomings, as well as the strengths of the proposed architecture in a navigation domain, were presented. Results suggest that the proposed architecture is significantly more sensitive and prone to fail than the two baseline methods, as the state uncertainty of the environment increases. However, the architecture also showed to be consistently more effective and efficient than the baseline methods in large environments with low uncertainty. From the results obtained, insight has been gained on which aspects of the architecture need to be improved in order to overcome its greatest challenge, state uncertainty.

Chapter 6

Conclusions and future work

The manufacturing of platforms for service robotic applications has been in a growing streak in sales[1]. Thus, it is expected for service robots to be involved in the day to day activities of all type of indoor environments, *e.g.* households, offices and hospitals. However, for a service robot to be able to assist in domestic activities, autonomy is a key feature expected from it. The research area of task planning involves problems related to autonomously solving tasks through the execution of a sequence of actions, what is more, uncertainty, partial observability and large spaces are the main challenges service robots face in task planning problems. Probabilistic models are widely used for task planning problems, due to their ability to deal with state uncertainty and partial observability (*e.g.* POMDP), however, they lack the capacity to bear with large state spaces. In this document, the presented architecture addresses all three drawbacks by integrating a knowledge base (to represent domain specific information) with a method that uses information from the knowledge base to build a hierarchy of actions that can be used to solve tasks in large environments.

6.1 Conclusions

The presented architecture for task planning in service robotics applications, enables a robot to automatically plan and execute a series of actions to solve tasks. Based on domain specific knowledge encoded in its knowledge base, and a hierarchy of

[1] `http://ifr.org/news/why-service-robots-are-booming-worldwide`

POMDPs that model a structure of sub-tasks that can be reused over several tasks, the architecture decomposes a task into set of smaller ones that are relatively easier to solve. By exploiting the information encoded in the knowledge base, the proposed architecture is able to segment the state space into a multi-resolution representation and build abstract actions that enable the robot to plan with actions that operate at different levels of granularity. In this way, the architecture makes possible to start planning at the most coarse resolution and increase the detail in planning as the robot gets closer to its goal state.

From the results obtained in the presented set of experiments, the proposed architecture showed to be more prone at failing than the baseline methods as the uncertainty of the agent's state increased. However, as the size of the environment increased where the uncertainty was relatively low, our architecture outperformed both baselines in terms of efficiency and effectiveness, showing a significant lower planning time and a consistent path relative cost across different environment sizes. The proposed architecture offers a hierarchical task planning solution for robotics that, despite it generates recursively optimal policies, by means of a recursive formulation it builds abstract actions starting from a concrete model of the environment and a hierarchical structure for one of the state variables. As the proposed architecture clearly has its shortcomings (to endure uncertainty), it is an important step towards task planning systems that fulfill the low response times and high reliability required in service robotics domains.

6.2 Contributions

The main contributions of the work presented in this research are:

1. A general framework for hierarchical task planning, capable of integrating new skills into a planning problem, without having to modify the description of those skills that already are part of the system.

2. A method that automatically builds an arbitrarily deep hierarchy of POMDPs from a hierarchical representation of the state space and a POMDP that models the bottom level of such representation.

3. A methodology to generate and execute plans in a sub-region of the original state space, employing its hierarchical representation and a hierarchy of POMDPs.

6.3 Future Work

In this document, a task planning architecture that combines a logic symbolic representation of the domain with a method to build hierarchies of POMDPs has been presented. However, so far it has been evaluated in a single domain. Thus, there are several aspects of the architecture that remain to be improved and could be considered as future work, which are listed below:

- **Evaluate on multi-domain scenarios.** Since most domestic domains encompass a large diversity of tasks, the first aspect to work on the current research is to evaluate how reliable the architecture behaves as the amount of domains involved in a task increases. Also, a potential alternative to deal with multi-domain problems could be to employ a factorized representation of the individual domains' state spaces and develop a mechanism that interleaves actions from different domains to solve a task.

- **An adjustable hierarchy of POMDPs.** One of the most restricting aspects of our architecture is that, since the hierarchy is built once at the initialization phase, it does not have the capability to adjust to changes in the environment. Thus, we believe that by updating the knowledge base with observations to modify the structure of the probabilistic planner (as done by [Zhang et al., 2017]), which in this case is the hierarchy of POMDPs, the architecture could become more robust to facts that were not originally included in the database.

- **Start with uncertainty.** In all the experiments the initial state was provided so that the hierarchical policy could be built. Thus, we would like to relax this assumption so that the architecture could plan even if it does not know its initial state. One way to overcome this drawback would be to develop a method, that would be executed every time before the planning phase, that performs a sequence of actions with the purpose of sampling observations, hypothesize on what its state is, and stop until the agent's state uncertainty reaches a given threshold.

- **Integrate reactive behavior.** Among the limitations of this research, the restriction that task requests shall only be issued to the robot if it is currently solving a task, is unrealistic in the context of real scenarios. In real domestic environments, the robot is expected to be able to be aware of its surroundings and, more importantly, of the needs of human users. Thus, the current architecture could benefit by integrating a layer of reactive control, maybe by assigning priority to reactive events and tasks that are deliberately solved, so that the robot can effectively redirect its attention to the most important activity at any moment.

Bibliography

[Attenborough, 2003] Attenborough, M. P. (2003). *Mathematics for electrical engineering and computing*. Elsevier.

[Balai et al., 2013a] Balai, E., Gelfond, M., and Zhang, Y. (2013a). Sparc-sorted asp with consistency restoring rules. *arXiv preprint arXiv:1301.1386*.

[Balai et al., 2013b] Balai, E., Gelfond, M., and Zhang, Y. (2013b). Towards answer set programming with sorts. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 135–147. Springer.

[Balduccini and Gelfond, 2003] Balduccini, M. and Gelfond, M. (2003). Logic programs with consistency-restoring rules. In *International Symposium on Logical Formalization of Commonsense Reasoning, AAAI 2003 Spring Symposium Series*, volume 102.

[Bellman, 1957] Bellman, R. E. (1957). Dynamic programming.

[Braziunas, 2003] Braziunas, D. (2003). Pomdp solution methods. *University of Toronto*.

[Chen et al., 2016] Chen, K., Yang, F., and Chen, X. (2016). Planning with task-oriented knowledge acquisition for a service robot. In *IJCAI*, pages 812–818.

[Chen et al., 2010] Chen, X., Ji, J., Jiang, J., Jin, G., and Wang, Feng an2d Xie, J. (2010). Developing high-level cognitive functions for service robots. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 989–996. International Foundation for Autonomous Agents and Multiagent Systems.

[Corona-Xelhuantzi et al., 2009] Corona-Xelhuantzi, E., Morales, E. F., and Sucar, E. (2009). Executing concurrent actions with multiple markov decision processes. In *Adaptive Dynamic Programming and Reinforcement Learning, 2009. ADPRL'09. IEEE Symposium on*, pages 82–89. IEEE.

[Dean and Givan, 1997] Dean, T. and Givan, R. (1997). Model minimization in markov decision processes. In *AAAI/IAAI*, pages 106–111.

[Dietterich, 2000] Dietterich, T. G. (2000). Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303.

[Fine et al., 1998] Fine, S., Singer, Y., and Tishby, N. (1998). The hierarchical hidden markov model: Analysis and applications. *Machine learning*, 32(1):41–62.

[Foka and Trahanias, 2007] Foka, A. and Trahanias, P. (2007). Real-time hierarchical pomdps for autonomous robot navigation. *Robotics and Autonomous Systems*, 55(7):561–571.

[Galindo et al., 2008] Galindo, C., Fernández-Madrigal, J.-A., González, J., and Saffiotti, A. (2008). Robot task planning using semantic maps. *Robotics and autonomous systems*, 56(11):955–966.

[Gelfond and Kahl, 2014] Gelfond, M. and Kahl, Y. (2014). *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge University Press.

[Gelfond and Lifschitz, 1988] Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080.

[Hanheide et al., 2017] Hanheide, M., Göbelbecker, M., Horn, G. S., Pronobis, A., Sjöö, K., Aydemir, A., Jensfelt, P., Gretton, C., Dearden, R., Janicek, M., et al. (2017). Robot task planning and explanation in open and uncertain worlds. *Artificial Intelligence*, 247:119–150.

[Hanheide et al., 2011] Hanheide, M., Gretton, C., Dearden, R. W., Hawes, N. A., Wyatt, J. L., Pronobis, A., Aydemir, A., Göbelbecker, M., and Zender, H. (2011). Exploiting probabilistic knowledge under uncertain sensing for efficient robot behaviour. In *Twenty-Second International Joint Conference on Artificial Intelligence*.

[Hauskrecht et al., 1998] Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T., and Boutilier, C. (1998). Hierarchical solution of markov decision processes using macro-actions. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 220–229. Morgan Kaufmann Publishers Inc.

[Hengst, 2004] Hengst, B. (2004). Model approximation for hexq hierarchical reinforcement learning. In *European Conference on Machine Learning*, pages 144–155. Springer.

[Hengst, 2012] Hengst, B. (2012). Hierarchical approaches. In *Reinforcement learning*, pages 293–323. Springer.

[Ingrand and Ghallab, 2017] Ingrand, F. and Ghallab, M. (2017). Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 247:10–44.

[Kaelbling et al., 1998] Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134.

[Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285.

[Kaelbling and Lozano-Pérez, 2011] Kaelbling, L. P. and Lozano-Pérez, T. (2011). Hierarchical task and motion planning in the now. in 2011 ieee icra.

[Keller et al., 2012] Keller, T., Eyerich, P., and Nebel, B. (2012). Task planning for an autonomous service robot. In *Towards Service Robots for Everyday Environments*, pages 117–124. Springer.

[Kim et al., 2018] Kim, J. W., Choi, G. B., and Lee, J. M. (2018). A pomdp framework for integrated scheduling of infrastructure maintenance and inspection. *Computers & Chemical Engineering*, 112:239–252.

[Köckemann et al., 2018] Köckemann, U., Khaliq, A. A., Pecora, F., and Saffiotti, A. (2018). Domain reasoning for robot task planning-a position paper. In *Proceedings of the Planning and Robotics Workshop at ICAPS 2018 (PlanRob)*.

[Kurniawati et al., 2008] Kurniawati, H., Hsu, D., and Lee, W. S. (2008). Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. In *Robotics: Science and systems*, volume 2008. Zurich, Switzerland.

[Latombe, 2012] Latombe, J.-C. (2012). *Robot motion planning*, volume 124. Springer Science & Business Media.

[Lifschitz, 2008] Lifschitz, V. (2008). What is answer set programming?. In *AAAI*, volume 8, pages 1594–1597.

[Lima et al., 2018] Lima, O., Ventura, R., and Awaad, I. (2018). Integrating classical planning and real robots in industrial and service robotics domains.

[Lu et al., 2017] Lu, D., Zhou, Y., Wu, F., Zhang, Z., and Chen, X. (2017). Integrating answer set programming with semantic dictionaries for robot task planning. In *IJCAI*, pages 4361–4367.

[Papadimitriou and Tsitsiklis, 1987] Papadimitriou, C. H. and Tsitsiklis, J. N. (1987). The complexity of markov decision processes. *Mathematics of operations research*, 12(3):441–450.

[Pineau et al., 2003] Pineau, J., Gordon, G., Thrun, S., et al. (2003). Point-based value iteration: An anytime algorithm for pomdps. In *IJCAI*, volume 3, pages 1025–1032.

[Pineau et al., 2001] Pineau, J., Roy, N., and Thrun, S. (2001). A hierarchical approach to pomdp planning and execution. In *Workshop on hierarchy and memory in reinforcement learning (ICML)*, volume 65, page 51.

[Pineau and Thrun, 2002] Pineau, J. and Thrun, S. (2002). An integrated approach to hierarchy and abstraction for pomdps.

[Pineda et al., 2017] Pineda, L. A., Rodríguez, A., Fuentes, G., Rascón, C., and Meza, I. (2017). A light non-monotonic knowledge-base for service robots. *Intelligent Service Robotics*, 10(3):159–171.

[Polya, 1945] Polya, G. (1945). How to solve it; a new aspect of mathematical method.

[Pusse and Klusch, 2019] Pusse, F. and Klusch, M. (2019). Hybrid online pomdp planning and deep reinforcement learning for safer self-driving cars. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 1013–1020. IEEE.

[Puterman, 1990] Puterman, M. L. (1990). Markov decision processes. *Handbooks in operations research and management science*, 2:331–434.

[Puterman, 2014] Puterman, M. L. (2014). *Markov Decision Processes.: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.

[Seib et al., 2016] Seib, V., Memmesheimer, R., and Paulus, D. (2016). A ros-based system for an autonomous service robot. In *Robot Operating System (ROS)*, pages 215–252. Springer.

[Spaan and Vlassis, 2005] Spaan, M. T. and Vlassis, N. (2005). Perseus: Randomized point-based value iteration for pomdps. *Journal of artificial intelligence research*, 24:195–220.

[Sridharan, 2016] Sridharan, M. (2016). Towards an architecture for representation, reasoning and learning in human-robot collaboration. In *2016 AAAI Spring Symposium Series*.

[Sridharan et al., 2018] Sridharan, M., Gelfond, M., Zhang, S., and Wyatt, J. (2018). Reba: A refinement-based architecture for knowledge representation and reasoning in robotics.

[Sucar, 2015] Sucar, L. E. (2015). Probabilistic graphical models. *Advances in Computer Vision and Pattern Recognition. London: Springer London. doi*, 10:978–1.

[Theocharous and Mahadevan, 2002] Theocharous, G. and Mahadevan, S. (2002). Approximate planning with hierarchical partially observable markov decision process models for robot navigation. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292)*, volume 2, pages 1347–1352. IEEE.

[Theocharous et al., 2001] Theocharous, G., Rohanimanesh, K., and Maharlevan, S. (2001). Learning hierarchical observable markov decision process models for robot navigation. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, volume 1, pages 511–516. IEEE.

[Thomaz et al., 2016] Thomaz, A., Hoffman, G., Cakmak, M., et al. (2016). Computational human-robot interaction. *Foundations and Trends® in Robotics*, 4(2-3):105–223.

[Weser et al., 2010] Weser, M., Off, D., and Zhang, J. (2010). Htn robot planning in partially observable dynamic environments. In *2010 IEEE International Conference on Robotics and Automation*, pages 1505–1510. IEEE.

[Zhang et al., 2017] Zhang, S., Khandelwal, P., and Stone, P. (2017). Dynamically constructed (po) mdps for adaptive robot planning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[Zhang et al., 2012] Zhang, S., Sridharan, M., and Bao, F. S. (2012). Asp+ pomdp: Integrating non-monotonic logic programming and probabilistic planning on robots. In *2012 IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL)*, pages 1–7. IEEE.

[Zhang et al., 2014] Zhang, S., Sridharan, M., Gelfond, M., and Wyatt, J. (2014). Integrating probabilistic graphical models and declarative programming for knowledge representation and reasoning in robotics. In *Planning and Robotics (PlanRob) Workshop at ICAPS, Portsmouth, USA*.

[Zhang et al., 2013] Zhang, S., Sridharan, M., and Washington, C. (2013). Active visual planning for mobile robot teams using hierarchical pomdps. *IEEE Transactions on Robotics*, 29(4):975–985.

[Zhang et al., 2015] Zhang, S., Sridharan, M., and Wyatt, J. L. (2015). Mixed logical inference and probabilistic planning for robots in unreliable worlds. *IEEE Transactions on Robotics*, 31(3):699–713.