Reasoning over hierarchical task workflows in robust robot planning

Master thesis

For Attainment of the Academic Degree

"Master of Science in Engineering"

Degree Program:

Mechatronics

Management Center Innsbruck

External supervisor:

Prof. Katia Sycara

Internal supervisor:

Dr. Sebastian Repetzki

Author:

Benedek Szulyovszky

1310620016

Declaration in lieu of oath

„I hereby declare, under oath, that this master thesis has been my independent work and has not been aided with any prohibited means. I declare, to the best of my knowledge and belief, that all passages taken from published and unpublished sources or documents have been reproduced whether as original, slightly changed or in thought, have been mentioned as such at the corresponding places of the thesis, by citation, where the extent of the original quotes is indicated. The paper has not been submitted for evaluation to another examination authority or has been published in this form or another."

Place, Date

Signature

Acknowledgement

This research work at The Robotics Institute of the Carnegie Mellon University was supported by Marshall Plan Scholarship to Benedek Szulyovszky. First of all special thanks to Yuqing Tang and Katia Sycara, for those many ideas, which guided me through all deep technical challenges and for being always motivated and helpful regarding to my research work. Special thanks to Sebastian Repetzki for the proofreading and for insightful comments.

Abstract

In the near future, robots will assist humans at work and home environments. To address this challenge, we formulate formal characterization and algorithms that enable to computationally implement and test the notions of robust robotic planning. This thesis attempts to combine human-like planning with high-level decision making, which successfully handles probabilistic behavior of the human environment. This research focuses on robotic planning over hierarchically ordered probabilistic task-workflows, which comprises most human activities such as many household activities. The Hierarchical Task Network planning reduces the searching space for the Markov Decision Process algorithms, which provide ability to formulate coherent and robust plans applying hard feasibility constrains and soft risk measures. Thus, the robot can proactively chose plans by making decisions over the trade-off problems between the cost and expected utility related to the robust operation.

Keywords: Artificial intelligence, High-level decision making, Robust planning, Machine learning, Probabilistic graphical models, Hierarchical planning

Table of Contents

1 Introduction **1**

2 Background and Related Work **3**

2.1 AI planning in general 3
2.2 Classical planning model 5
2.3 Hierarchical planning 6
2.4 Non-deterministic planning 7
2.5 Markov Decision Process 8
2.6 Robust planning 9

3 Hierarchical planning **10**

3.1 Chosen framework: SHOP2 10
3.2 Features of SHOP2 10
3.2.1 Elements of a Domain Description 11
3.2.2 Developing domain descriptions and problem for SHOP2 14
3.3 Example 15
3.4 Evaluation 17
3.5 Conclusion 17

4 Probabilistic hierarchical planning **19**

4.1 Formalization 19
4.2 Probabilistic hierarchical framework 21
4.3 MDP module 23
4.4 Algorithms 25
4.5 Q-learning 27
4.6 Probability assumptions 28
4.7 Conclusion 29

5 Robust planning **31**
5.1 Proposed solution 33
5.2 Soft constrain and hard feasibility constrain 34
5.3 Bisection search 35
5.4 Test case 36
5.5 Conclusion 37

6 Evaluation **38**
6.1 Modules 38
6.2 Robust reasoning over probabilistic hierarchical workflows 38
6.3 Evaluation of the modules separately 45
6.4 Conclusions 47

7 Discussion **48**
7.1 Planner as a black box model 48
7.2 Robustness 49
7.3 Limitation 50
7.4 Extension of the proposed model 51

8 Summary **52**

List of Algorithms **53**

List of Figures **54**

List of Code **55**

Bibliography **56**

Appendix A Appendix A **59**
A.1 Planing domain 59
A.2 Planing problem definition 64

Appendix B Appendix B **65**
B.1 HTN module 65
B.2 MDP module 78
B.3 Simulator module 82

Chapter 1

Introduction

There have been dynamically growing physical and cognitive capabilities in robotics. The doubtless demand for mobile robots to perform in uncertain and dynamic environments results increasing focus on intelligent and robust planning. The complexity of robot interaction with its environment is often under-estimated, since the variety for robot to chose appropriate actions in time has exponential increasing tendency, the planner has to deal with enormous data to specify each operation. A relatively simple human action such as grabbing of a mug could mean thousands of possible sequential and methodical variation in time. Therefore it is crucial to have expressive, but compact representation of a robot planner to handle challenges of this kind in a fast and accurate manner. Significant developments were made in building low level-level controllers and detect objects to enable robots to perform navigation and manipulation. Although manually sequenced instructions are not scalable because of the large variety of tasks and situations that can arise in unstructured and uncertain environments. To carry out increasingly complex tasks, robotic communities make strong efforts on developing robust and sophisticated high-level decision making models and implement them as planning systems. One of the most challenging issues is to find optimum in the trade-off between computational efficiency and needed domain expert engineering work to build a reasoning systems. One might handle the problem with using abstract models, which are close to human thinking, which has twofold advantages respect to its effect on domain expert work. On one side, the human-like models need less abstraction, which lead more intuitive and straightforward systems. On the other hand, human created big data on internet can be easily interpreted and transformed into these structures.

This thesis outlines an approach to integrate human-like planning and models, which are successfully functioning uncertain environments generating robust plans. Our contributions are as follows:

- Describe human-like high-level planning framework for reasoning over probabilistic task networks.
- Analyze notions of robustness respect to foresight in human environments and evaluate specific risk measures on our planning system.
- Give a discussion about the extension of our framework.

This thesis organized as follows. Chapter 2 surveys related work and give a short overview of the fundamental theory of relevant topics. Chapter 3 reviews Hierarchical Task Network planning, give formalism and illustrate it with an example. Then Chapter 4 gives formal description of the used probabilistic model and with pseudocodes, it explains our implementation. Chapter 5 investigates robustness regarding to the given problem and formally describes the proposed considerations and solutions for handling them. Chapter 6 describes the evaluation of the result of the proposed planning system. In Chapter 7 we give a discussion over extension of the planning model. Finally Chapter 8 gives a summary of the work.

Chapter 2

Background and Related Work

There are a great deal of related work for ours, we attempt to introduce the main concepts and contacting scientific works in this chapter.

2.1 AI planning in general

This section attempts to give an overview of relevant concepts of Artificial Intelligence (AI) planning. The following works [18, 20] extends this short review with detailed introduction and state of the art research concepts.

Autonomous planning is an extensive area of Artificial Intelligence. The most real world complex planning tasks can be successfully solved in various ways, the action sequences do not only vary by their order or length but by even more complex properties such as efficiency. In robotic planning there are many factors, which could describe the efficiency of a plan and can be functioning as optimization criteria for it, such as time, cost, energy or even more abstract and complex factors like fluency or safety. The goal of a plan can be described in various ways: reach a set of states that satisfies a given goal condition; or to perform a specific task; or a set of states that the world should be kept in or kept out of; or a partially ordered set of states that we might want the world to go through. The diversity planning circumstances results the need for incorporating relevant domain knowledge. The question which is a problem specific query is then arising: how much a-priori knowledge is need to be incorporated into a planning system. The question is twofold. If the planner does not possess enough background knowledge, the decisions might be not enough foresighted. On the other side, too much hard-coded knowledge limits the flexibility and learning capability of a system.

One motivation therefore is practice-related. Success of an automated planning highly depends on its ability to process and interpret given information which is necessary for efficient planning. Imagine a complex problem such as a company leading, enormous relevant information is needed for even a single decision. The ability to reason intelligently is inconceivable without relevant experience and background knowledge, but unnecessary information and inefficient processes makes the system slow. This abstract point leads to another motivation of automated planning. As a human being, our purpose is well described by our decisions. To make intelligent robots and characterize their behavior, one of the fundamental points is start to design decision making procedures.

In classical planning based on its conceptional model there are three components of the planning domain: (1) the planner, (2) the plan-execution agent, and (3) the world Σ in which the plan is executed as it is shown on Figure 2.1.

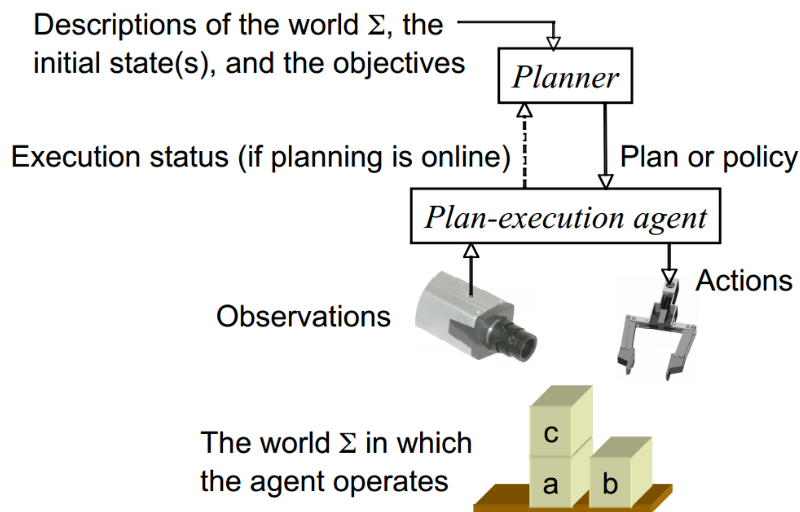

Figure 2.1: Conceptional model of AI planning [3]

One important part of the descriptions of Σ is the initial state of the environment and the plan-execution agent at the beginning of the planning procedure. The state is described with literals. A literal is a symbolic predicate applied to a list of arguments, which might be variables or even constants. Such a literal can be: at(car,garage) or grabbed(ball). Because the infinite number of possible arguments, the characterization of the world never attempt to be represented complete. There are list of arguments which has to be defined explicitly, but irrelevant information just make the representation unnecessarily complex. In all of the representations in this work the closed world

condition is hold, therefore the presumption that a statement is true is also known to be true. Conversely, what is not currently known to be true, is false. The goal states are described by conjunction of literals.

The planner output consists of either a plan, which means a linear sequence of actions or a policy, which is a set of state-action pairs with at most one action for each state.

2.2 Classical planning model

The previously described conceptional model gives a starting point to assess the following assumptions, which illustrate the framework of classical planning model:

- **Assumption 1** - State-transition model: Σ has finite set of states.
- **Assumption 2** - Fully observability: Σ is fully observable, the current state is always known.
- **Assumption 3** - Determinism: Σ is deterministic, from any state an applicable action take the world to another known state.
- **Assumption 4** - Static: Σ is static, it remains in the current state until an applicable action is applied.
- **Assumption 5** - Restricted goals: The goal of the planner is to produce a plan which take Σ into the goal state, where all goal conditions s_g are fulfilled.
- **Assumption 6** - Sequential plans: The solution plan is a linearly ordered finite sequence of actions.
- **Assumption 7** - Implicit Time: Actions have no duration, they are instantaneously take Σ to the next state.
- **Assumption 8** - Offline programming: The planner generates the plan from the initial state to the goal state without any execution by the agent.

This conceptual model gives a foundation to understand the basic concepts, but in the latter sections a few of these assumptions will be relaxed and the appropriate formalism of the problem will be detailed, which will be mathematically exact and compatible with the created algorithms.

2.3 Hierarchical planning

In Hierarchical Task Network (HTN) planning, similarly to classical AI planning, each state of the world is represented by a set of literals, and each action corresponds to deterministic state transitions. On the other hand, HTN planners have significantly different approach to generate plans in what they plan for, and how they plan for it.

The objective of the HTN planner is to generate a sequence of actions that will perform some higher-level activity or task. The description of the planning domain includes a set of operators, which are the primitive actions and also a set of methods, each of which is a prescription for how to decompose a task into subtasks. Figure 2.2 illustrates the basic approach of HTN planning with a simple example.

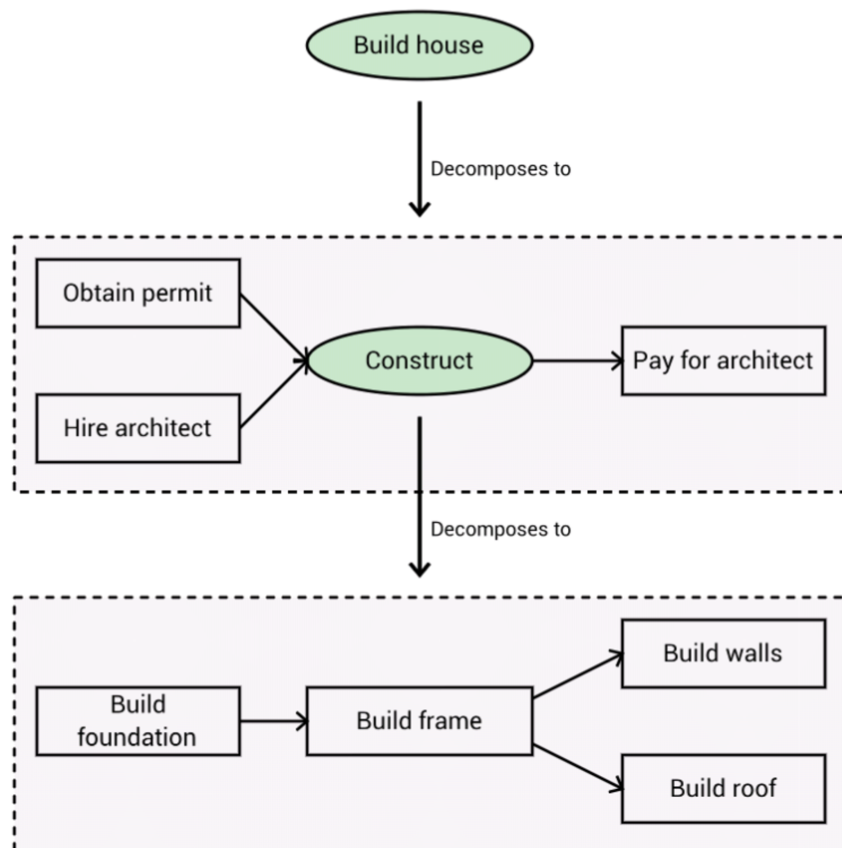

Figure 2.2: Conceptional model of HTN planning

The planning domain can be described as follows. The initial set of literals, similarly to classical AI planning represents the initial state. However, the goal states are expressed by the problem specification, which contains a list of tasks to accomplish. The planning process takes the first task from the problem specification and decomposes

it until the planner reaches a primitive operators which after execution takes the world to the resulting state. When the actual task is not a primitive operator, the planner chooses an applicable method to decompose this high-level task into lower subtasks and so on. When the method cannot be decomposed into legal sequence of primitive operators, the planner need to backtrack to a higher level and try another applicable method to solve the task problem.

HTN planning was first developed more than 35 years ago [21, 27]. Historically, most of the HTN-planning researchers have focused on practical applications. Examples include production-line scheduling [33], planning and scheduling for spacecraft [7, 15], evacuation planning [17].

2.4 Non-deterministic planning

Many successful algorithms were developed for challenging classical AI planning problems, but the above described assumptions proved to be strong limitations for many real problems in human environment. To extend the bounds of the model for further challenges, a few assumptions has to be relaxed.

The world is not predictable, actions has more than one possible outcomes, which means a better describing model has to allow non-determinism instead of the deterministic world assumption. In other words, an action can fail as it often does. The decision-theoretic optimal approach to handle non-determinism in a planning domain is to make a conditional plan, which can be in a form of a tree or a policy. This supports the interpretation of the different outcomes of a certain action. On the Figure 2.3 a binary decision tree shows a possible strategy to represent the non-deterministic outcomes of the actions.

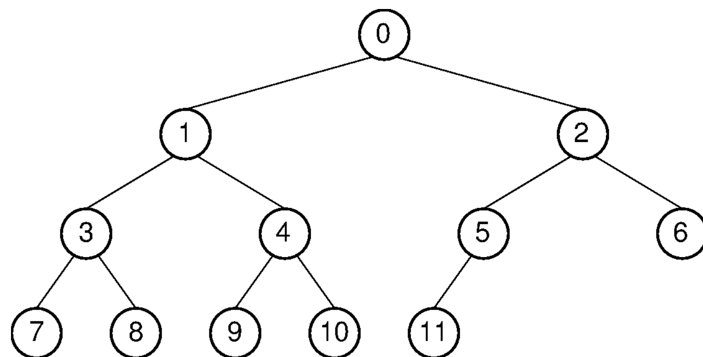

Figure 2.3: Non-determinism represented by binary decision tree

The process of constructing and reasoning over an exponentially increasing searching space of this kind can be especially computationally expensive. This problem is one of the general challenges in fields where searching spaces are constructed. Several successful research work has been done on this problem such as [8].

2.5 Markov Decision Process

Handling uncertainty is crucial in human environment. Two types of uncertainty are distinguished, one is related to the present and the other is coming from the unpredictability of future. In this thesis we focus on the second aspect, but we note that there are successful implementations of Partially Observable Markov Decision Processes (POMDPs) [23] in AI planning involving planning in the belief space rather than the underlying state space of the domain. The belief space is a space of probability distributions over the underlying world states, where a state estimator (such as a Bayesian or Kalman filter) maintains the current distribution over the underlying world states, based on the actions and observations [22].

Markov Decision Process (MDP) [5, 10] is a discrete time stochastic control process, which is applied for decision making problems, where the actions results outcomes with uncertainty. The model is an extension of Markov chains, in which the agent has choice of actions in the states and does not know how many actions has to be made before the world reach its goal state. If the process may go on forever the process has infinite horizon, however if the agent will stop eventually, the problem has indefinite horizon. In every state the agent has to decide which applicable action it wants to execute. This decision is made based on a policy. The solution for a MDP problem is the policy which defines the optimal actions for every state in the problem domain. MDP problems often solved by dynamic programming and reinforcement learning. It is used in various fields such as robotics, economics or game theory.

Many recent work attempt to formulate and solve problems of robot planning under uncertainty in probabilistic frameworks. Filling the gap between HTN and MDP, successful and efficient planning models were built [12, 26], which give foundation of our work. However, these existing works leaves several challenges open regarding to robustness.

2.6 Robust planning

Nowadays, autonomous mobile robotics has grown up for many challenging tasks. Delegating tasks for robots gives considerable help for people. But it is essential that the quality has to remain the same or even has to exceed human level. To do this, the robots has to remain to be robust for the changing of the environment and the planner has to adopt for unknown effects. There are extensive work on robust planning to investigate and formally implement notions of robustness.

The authors of [16] design algorithms to build models based on historical data of operation which can make the robot able to predict previously unknown events and states. [13] presents a Bayesian approach to learn flexible safety constraints and subsequently verifying whether plans satisfy these constraints. [4] describes a planning system, called HOTRiDE (Hierarchical Ordered Task Replanning in Dynamic Environments), which interleaves plan generation, execution, and repair in order to work in dynamic environment. [34] proposes solution for handling unexpected faults at run-time. They have developed methods for the localization and repair of faulty software components at run-time and the deliberative layer of the control system is aware of the lost capabilities of the system and adapt its decision-making.

[32] identifies three major meanings for resilience: foresight and avoidance of undesired events, coping with ongoing trouble and recuperation from occurrence of these events. [35] defines resilience, which completes the concept of robustness with the notion of proactive activity, which is a way of anticipating failures. In this work we will evaluate our planner system for notions of foresight and avoidance of undesired events in meaning of robustness.

Chapter 3

Hierarchical planning

3.1 Chosen framework: SHOP2

As it was introduced in the previous chapter, HTN planners are developed based on a well-known and successful approach to formulate plans in a similar manner as human does. The potential of this planner family is multilateral. On one side, since constructing HTN methods are analogous to write receipts for cooking, the human-based knowledge can be transferred easily into methods. This is not just comfortable and straightforward but solve several scalability issues of making more abstract planning models. Moreover, HTN planning has a favorable inherited property: due to its decomposition structure (methods) it prunes the search space, resulting computationally more efficient operation.

SHOP2, Simple Hierarchical Ordered Planner 2 [17], is a domain-independent planning system based on HTN planning. SHOP2 generates the steps of each plan in the same order that those steps will later be executed, so it knows the current state at each step of the planning process. This reduces the complexity of reasoning by eliminating a great deal of uncertainty about the world, thereby making it easy to incorporate substantial expressive power into the planning system. SHOP2 can do axiomatic inference, mixed symbolic/numeric computations, and calls to external programs [18].

3.2 Features of SHOP2

This section describes the basic features of SHOP2 based on [18].

3.2.1 Elements of a Domain Description

The description of a planning domain, consists of a set of methods, operators, and axioms. Below we describe each of these briefly.

Operators

Each operator indicates how a primitive task can be performed. The SHOP2 operators are very similar to PDDL [9] operators: each operator *o* has a head *head(o)* consisting of the operator's name and a list of parameters, a precondition expression *pre(o)* indicating what has to be true in the current state in order for the operator to be applicable, and a delete list *del(o)* and add list add(o) giving the operator's negative and positive effects. Like in PDDL, the preconditions and effects may include logical connectives and quantifiers. The operators also can do numeric computations and assignments to local variables. Just as in PDDL, no two operators can have the same name; thus for each primitive task, all applicable actions are instances of the same operator. Each operator also has an optional cost expression (the default value is 1). This expression can be arbitrarily complicated and can use any of the variables that appear in the operator's head and precondition. The cost of a plan is the sum of the costs of the operator instances.

```
1  ;dusting in the room
2  (:operator (!dusting ?perf ?loc)
3      ((at ?perf ?loc) (clean water) (at cleaner_cart ?loc))
4      ((clean water))
5      ((dusted ?loc) (dirty water))
6  )
```

Listing 1: Operator description

Listing 1 shows an operator description in SHOP2. After the *:operator* keyword in the head, the *!dusting* is the name and the next variables (starting with question mark) are the parameters of the operator. In line 3 the precondition literals are listed, with similar syntax the delete post-conditions and add post-conditions are in the fourth and fifth lines respectively. The dusting operator therefore defines the way of dusting a room by the performer with help of the cleaner cart. The precondition is the following: the cleaner cart and the performer have to be at the same location and for dusting the room there is need for clean water. Then the operator delete post-condition deletes

the clean water literal from the world state meaning the cleaning changed the state of the water and in the next line it adds new literals to the state meaning the room is now dusted and the water became dirty.

Methods

Each method indicates how to decompose a compound task into a partially ordered set of subtask, each of which can be compound or primitive. The simplest version of a method has three parts: the task for which the method is to be used, the precondition that the current state must satisfy in order for the method to be applicable, and the subtask that need to be accomplished in order to accomplish that task. In general, there may be several alternative ways of accomplishing *head(m)*. There may be more than one method whose head is *head(m)*, more than one set of variable bindings that satisfy *precond(m)*, more than one ordering consistent with *sub(m)*, or more than one possible way to accomplish some of the subtask in *sub(m)*. These alternatives produce branches in SHOP2's search space.

```
1  (:method (Moving ?perf ?perf_loc ?loc)
2     ;the room is directly accessable from the performer's loc,
3       ;so move operator is directly called
4     ((at ?perf ?perf_loc) (TransConnect ?perf_loc ?loc))
5     ((!move ?perf ?perf_loc ?loc))
6     ;the target location is not accessable, therefore an
7       ;intermediate location will be the first target location
8     ((at ?perf ?perf_loc) (not(TransConnect ?perf_loc ?loc))
9       (TransConnect ?perf_loc ?interm-loc))
10    ((TravelingThrough ?perf ?perf_loc ?interm-loc ?loc))
11 )
```

Listing 2: Method description

Listing 2 explains a method definition in SHOP2. This method gives instructions if the performer attempt to move from a location to another, but there is no direct connection between them, for example these rooms are not separated with door but the performer has to travel through a corridor. Similarly to the operator definition, in the head after the name there are the parameters. Then, the method branches are coming. The first branch of the *TravellingThrough* method describes the case when the performer attempt to move between two rooms connected by one additional

room. The connected rooms are described by an axiom *TransConnect*, which will be explained in the next section. After the preconditions in line 5, the move operator is called to take the performer to the intermediate location, from where it can go to the original destination. In the second branch the case is described, when there is no intermediate location which connects the current location and the destination. In this branch the *TravellingThrough* method is recursively called to find the way to the destination. Of course this greedy approach is not efficient way to find optimal path, but further method can refine the strategy together with this representative example for a SHOP2 method.

Axioms

The precondition of each method or operator may include conjunctions, disjunctions, negations, universal and existential quantifiers, implications, numerical computations, and external function calls. Furthermore, axioms can be used to infer preconditions that are not explicitly asserted in the current state. The axioms are generalized versions of Horn clauses, written in a Lisp-like syntax: for example, (:- head tail) says that head is true if tail is true. The tail of the clause may contain anything that may appear in the precondition of an operator or method.

```
1 ;definition of 'same'
2 (:- (same ?x ?x) nil)
3 ;definition of 'different'
4  (:- (different ?x ?y) ((not (same ?x ?y))))
5 ;interchangeability of the connect property of rooms
6 (:- (TransConnect ?x ?y) (or (connect ?y ?x) (connect ?x ?y)))
7 ;definition of directly connected rooms
8 (:- (TransConnect ?x ?z) ((connect ?x ?interm-loc)
9  (connect ?interm-loc ?z)))
```

Listing 3: Axiom description

Axiom description is shown in Listing 3, where there are four examples for better understanding. The first axiom defines what same mean in our problem. This is needed to define same locations. The next axiom similarly describes difference as the negate of the same property on two variables. The third and fourth axioms were used in the method description to help plan the moving strategy from one location to another. First in line 6 the interchangeability of the connect property is shown, which means

if room A is connected to room B, then B is also connected to A. The in the fourth axiom the intermediate location is used to define path between two locations.

3.2.2 Developing domain descriptions and problem for SHOP2

The first step in developing a domain description for SHOP2 is to formulate some abstract tasks and methods that constitute a reasonable problem-solving strategy. As an example, we will use a simplified version of *HouseCleaning* domain. The problem is to clean the entire house by doing certain household activities and use tools for these actions. Listing 4 shows the problem definition with describing the initial state with literals and the high-level tasks which are related to high-level methods, these are a set of abstract tasks, which can be done by apply further methods which decompose these task into primitive action, which are applied based on the operator descriptions. Once we have an abstract strategy similar to the one in Listing 4, we can implement it as a SHOP2 domain description consisting of methods, operators and axioms. For example, there are methods to decompose how the performer can clean in the kitchen which includes the *TravellingThrough* method in Listing 2. Then this method description is using axioms in Listing 3. Then arriving to the kitchen the performer can dust the room applying the dusting operator described in Listing 1.

```
{defproblem problem cleaning
2  (
3     ;initial states defined by ground logical atoms
4     (at cleaner_cart kitchen)
5     (at robot bath)
6     (connect liv bath)
7     (connect liv kitchen)
8     (clean water)
9  )
10    ;planning problems
11    (:unordered (HouseCleaning robot bath)
12                (HouseCleaning robot kitchen)
13                (HouseCleaning robot liv))
14 )
```

Listing 4: Problem description

As it has been seen in the previous sections, in SHOP2 the planning procedure is done with help of variables. Similarly to the modern programming languages, the

variables can take value. For example, *?perf-loc* can be be equal to all the location ground logical atoms such as: *liv* (living room), *bath* (bathroom) and *kitchen.* The variable matching is done exactly in similar manner as the planing, the high-level task define for example the robot as *?performer* then in every method and operator the *?performer* means the robot.

SHOP2 was implemented in Lisp programming language, but the its authors made a Java version (JSHOP2), which inherited most of the features from the original program. For easier understanding and debugging we build our domain in this version and use its interfaces to generate the legal plans. For detailed syntax and algorithm explanation see [11].

3.3 Example

In this thesis we are introducing an easy but representative example, the HouseCleaning domain to illustrate the results of JSHOP2 and the later details algorithms.

The main task in the HouseCleaning domain is to clean all the rooms of a house. The cleaning subtask is the following: the performer, which is the robot has to do the dusting, the vacuuming and the mopping in all different rooms. In order to add realistic ordering constrain, the mopping can only start after the vacuuming is done as well as the dusting has to be before the vacuuming. For each of these cleaning subtask the performer has to be in the certain room and the robot needs tools from the cleaner cart, which has to be in the same room where the performer is. Furthermore, after the dusting and mopping actions the water in the cleaning cart becomes dirty, therefore at latest before the next dusting or mopping action the water has to be changed. This means before execution of every cleaning subtask, a method make sure that every preconditions are fulfilled, namely the performer and the cleaner cart is in the room and the water is clean for relevant cleaning tasks. There are three rooms: living room, bathroom and kitchen, where the living room connects directly to bathroom and kitchen, therefore an axiom defines if two arbitrary rooms are directly connected or the performer has to go through on a third one when it wants to take the cleaner cart or just simple go from a room to another.

The Figure 5.1 illustrates the hierarchy of HTN methods and operators for the previously introduced example. The horizontal direction describes the level of hierarchy. While the full lined arrow gives ordering connection between the methods (oval

elements), the dotted line gives the related operator (rectangular elements) for the methods.

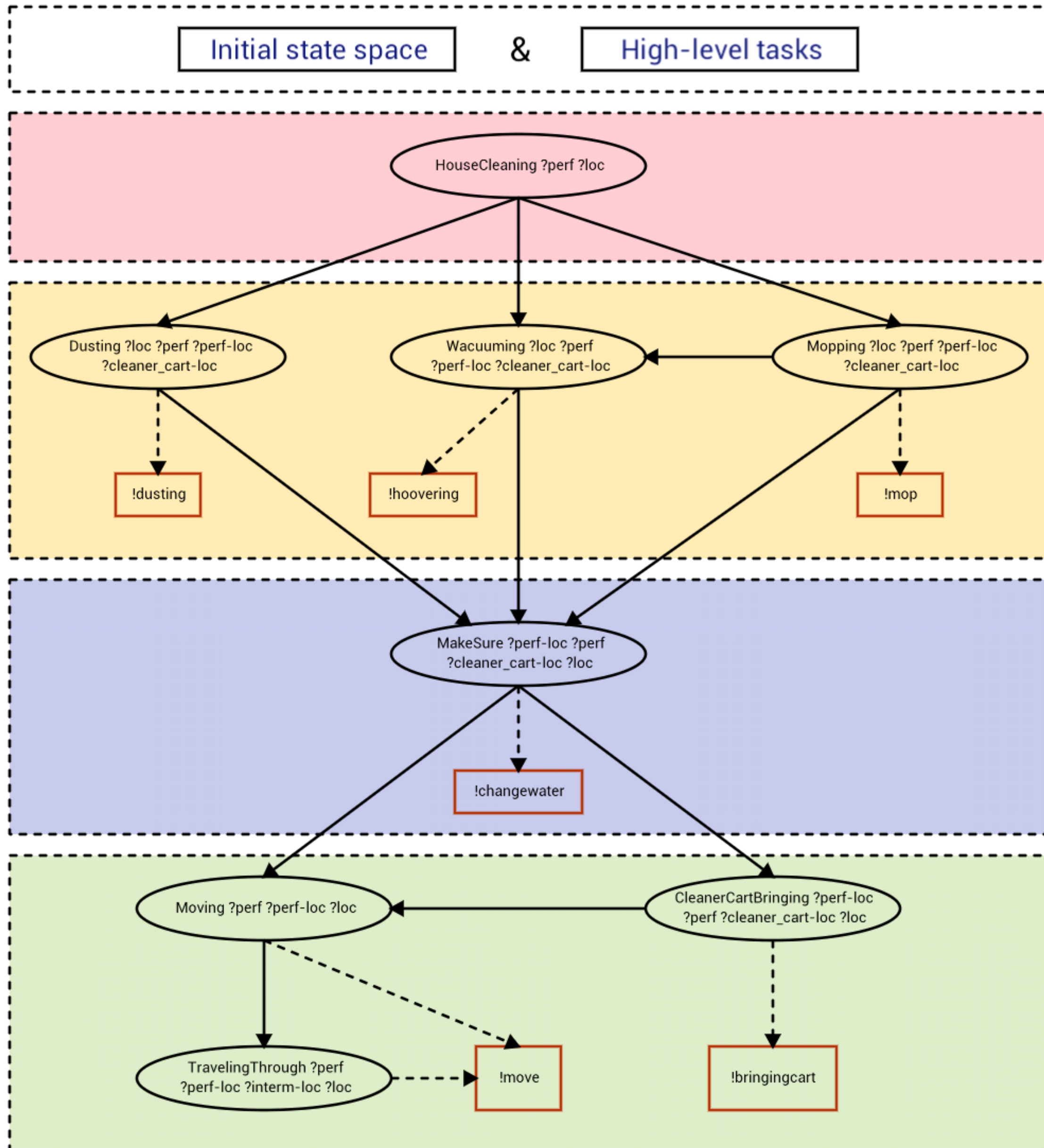

Figure 3.1: HouseCleaning domain

In Appendix A the HouseCleaning domain description and problem definition source codes are available. The used syntax has many similarity to Common Lisp and PDDL languages, but for better understanding comments gives further information, for detailed description see the Documentation of JSHOP2 [11].

3.4 Evaluation

Based on the previous sections the HouseCleaning domain was build and simulated for legal plans. However, JSHOP2 only could generate plan for a simplified model, because of the exponential blow up the problem of memory overflow terminates the simulation. Therefore different simplifications were made. For example with only two rooms, the planner find a few tens of plans depending on the problem description, which can vary on the performer and the cleaner cart initial locations. Or if one of the cleaning subtasks such as dusting or mopping were eliminated from the plans, JSHOP2 finds approximately 170 different successful plans. The different scenarios are basically varies on the robot and cleaner cart movements, depending on scheduling the cleaning subtasks in different rooms with different orderings.

SHOP2 plans for tasks in the same order that they will be executed, and thus it knows the current state at each step of the planning process. This reduces the complexity of reasoning by removing a great deal of uncertainty about the world. However this property is not compatible with our purpose, because for probabilistic inference, the planner has to reason over the set of all possibilities instead of having only choice for the next action. JSHOP2 planning procedure is sound and complete over a large class of planning problems [17].

However, JSHOP2 has excellent capabilities to generate hierarchical plans based the predefined problem domain, there are notions of robust probabilistic planning which cannot be evaluated in this framework. Since the Java implementation of SHOP2 does not contains all embedded functions which is inherited from the Lisp language, the forming heuristic functions can be problematic. On the other hand, to implement a probabilistic domain and apply MDP algorithms on it, is especially involved. Therefore to combine the advantages of HTN planning, using the already generated method and action libraries but having flexibility on design a probabilistic environment, we build our own planner and simulation environment in Java.

3.5 Conclusion

In this chapter Hierarchical Task Network planning were introduced more in detail with special focus on SHOP2 and its functionality. The basic approach of HTN, which is close to the human thinking and planning can be transferred into a formal planning framework. The literals exactly describe the world state. The high-level task

can be decomposed into subtasks with help of the methods, which define partially ordered task-workflow. The sequence of actions can be executed based on the operator descriptions, which define deterministic state transitions to accomplish all the tasks. With SHOP2 we formed our method and action libraries, which will be used in the next chapters combined with further tools to generate robust plans.

Chapter 4

Probabilistic hierarchical planning

To overcome of the limits of the framework used for HTN planning, we are formulating a simulation-based environment based on the MDP model. The combination of MDP and HTN planning gives flexibility to evaluate plans respect to the robustness.

4.1 Formalization

In this section we formally define the framework which will be used to handle uncertainty in our algorithms to generate plans. This work is built on [26].

Definition 1 *A non-deterministic state transition domain NSTD is a tuple* $M =< P, S, A, R >$ *where: (i)* $P = P_s \cup P_A$ *is a finite set of propositions resulting of the union of state and action propositions respectively; (ii)* $S \subseteq 2^{P_S}$ *is the finite set of all possible states; (iii)* $A \subseteq 2^{P_A}$ *is the finite set of primitive actions; and (iv)* $Tr \subseteq S \times A \times S$ *is the state-transition relation.*

We attempt to build a regression-based planning using logical representation of sets of world states, actions and the state transitions. We wish to retain the basic HTN structure of planning to achieve a goal, so we formulate first the framework for HTN planning.

Definition 2 *An HTN planning domain D is a tuple <T,M> where: (i) T is a finite set of task symbols which are composed by two disjoint sets: non-primitive tasks NT and primitive actions A; and (ii) M is a finite set of methods to refine the tasks in T.*

The non-primitive tasks NT can be decomposed into primitive actions A with help of methods.

Definition 3 *An HTN method* $m \subseteq M$ *is represented as a pair (NT, H), where NT is the non-primitive task to be decomposed and H is a task network which specifies how NT can be achieved by other tasks. If H defines non-primitive tasks further methods has to be called via the basic principle of recursion to continue the decomposition. The method m has preconditions precond(m), if precond(m)* $\subseteq s_t$ *then the method is applicable in* s_t.

Definition 4 *A task network H is a pair (T, C) where T is a finite set of tasks which can be non-primitive tasks NT and primitive actions A. C is a set of fully ordering constraints on the tasks T. If* $T \subseteq A$ *then H is a primitive task network.*

An action a is expressed by its preconditions *precond(a)*, its postcondition delete list *delete(a)* and its postcondition add list *add(a)*.

Definition 5 *A primitive action a is executable in* s_t *if* $precond(a) \subseteq s_t$ *and the execution of a in* s_t *takes the world immediately to the state* s_{t+1} *such that* $add(a) \subseteq s_{t+1}$ *and* $delete(a) \nsubseteq s_{t+1}$.

Definition 6 *A primitive task network* $H = <C,T>$ *is applicable on* s_t *iff (1) the preconditions of the first primitive action in T is executable* s_t.

Definition 7 *Let* $m_1, ..., m_n$ *be a sequence of methods that fully decomposes the set of HTN planning tasks T into task network H. Then, the decomposition is valid at state* s_0 *iff the resulting primitive task network H is applicable at* s_0.

In this way sequence of primitive actions $a_1, ..., a_n$ which is defined by the task network for the set of HTN planner tasks, results a sequence of state transitions $s_0, a_1, ..., s_{n-1}, a_{n-1}, a_n$, which is called trajectory of execution.

Now that the HTN framework can define all the applicable primitive actions in a given state respecting to the applicable methods, we formally define the MDP model.

Definition 8 *And MDP forms a tuple* $< S, A, app, PR, R >$, *where S and A are the finite set of states and actions respectively; app(*s_t*) is the set of all applicable actions in state* s_t*, defined by the HTN task network; for every* $a \subseteq app(s_t)$, $PR(s_t, a, s_{t+1})$ *is the probability distribution of the state transition* (s_t, a, s_{t+1})*; and for every* $s_t \subseteq S$, $R(s_t) \geq 0$ *is the reward for* s_t.

4.2 Probabilistic hierarchical framework

In this section we give an overview of the implemented algorithms with representative pseudo-codes. The most important parts of the original Java source code can be found in Appendix B.

Algorithm 1 HTN Search-space building algorithm

Require: initial state s_0, max depth D, high-level tasks $T(s_t)$
Ensure: fill up the *nodemap*, *edgemap* and *statetoactions* hashmaps

```
 1: procedure BuildingHTNSearchSpace(s_0, D, T(s_t))
 2:     add s_0 to the first hierarchical level
 3:     for each d = 1...D do
 4:         for each s_t in the current hierarchical level d do
 5:             if all tasks are achieved by s_t then
 6:                 continue
 7:             else
 8:                 for each applicable m decomposes T(s_t) do
 9:                     s_{t+m}, T(s_{t+m}) ← ApplyMethod(m, s_t)
10:                     add s_{t+m} and T(s_{t+m}) to d + 1 level
11:                 end
12:         end
13:     end
14: end procedure
```

Algorithm 1 shows the core iterative routine for building the HTN searching space. The basic approach is a modified iterative deepening search strategy, where the depth parameter can be defined as follows. For every certain state on the given hierarchical level, every applicable method are called, which results a new set of resulting states, which are then the next hierarchical level. The depth parameter therefore gives a hierarchical level, choosing the maximum depth for enough high all states on that given hierarchical level guaranteed to accomplish all the tasks. Therefore this top-down fashion search takes the initial state and builds up the possible next states inherently pruned by the methods. It means compared to purely combinatorial approach, the methods filter out the actions which will not achieve the given task even they are legal. The most trivial example is the moving action, which can lead to an infinite loop with completely legal decomposition. This implementation do not attempt to handle interleaving planning, in a later chapter there will be a short discussion about this extension.

Algorithm 2 implements a multiple level conditioning of the state. The certain worldstate has to fulfill not only the precondition of a method, but also the precondition of a branch and its first action in this way HTN and MDP has different set of applicable actions for every states. In the evaluation chapter we will highlight this behavior in more detail. If all these conditions are hold, then an action can be executed and the state can be updated. However, a method can be further decomposed into other methods to accomplish the task. In this case the state is not updated, but the tasks to be achieved respect to the state will be updated. Algorithm 2 also shows that the searching space built by Algorithm 1 only takes the last state of a method stack as a node into account. This would mean the distribution of searching space is not enough fine, but in the next algorithm we introduce another structure which will represent the fine searching space for the latter planning modules.

Algorithm 2 Apply a method

Require: current state s_t, method m
Ensure: resulting state s_{t+n} after applying m, updated task $T(s_{t+n})$

```
1:  procedure ApplyMethod(s_t, m)
2:      add s_t to the stack of states
3:      for each branch b of the method do
4:          if precond(b) ∈ s_t then
5:              if precond(first action a_1) ∈ s_t then
6:                  apply a_1 to get the next state s_{t+1}
7:                  UpdateSearchingSpace(s_t, a_1, s_{t+1}, m)
8:                  update T(s_{t+1})
9:                  add s_{t+1} to the stack of states
10:             else
11:                 continue
                ⋮
12:             if precond(a_n) ∈ last element of stack then
13:                 apply a_n to get the next state s_{t+n}
14:                 UpdateSearchingSpace(s_{t+n-1}, a_n, s_{t+n}, m)
15:                 update T(s_{t+n})
16:                 return s_{t+n}, T(s_{t+n})
17:             else
18:                 continue
19:     end
20:     return m failed
21: end procedure
```

Algorithm 3 updates the *nodemap*, *edgemap* and *statetoactions* hashmaps, which will be used as a compact representation of searching space for the MDP planning module.

Algorithm 3 Update the searching space for MDP module

Require: current state s_t, next state s_{t+1}, action a, method library m
Ensure: updated *node*, *edge* and *statetoactions* hashmaps

1: **procedure** UPDATESEARCHINGSPACE(s_t, s_{t+1}, a, m)
2:     **if** s_{t+1} is not in *nodemap* **then**
3:         add mapping from s_{t+1} to node n in *nodemap*
4:     **else**
5:         get the value n for the key s_{t+1}
6:     add mapping between s_t and s_{t+1} in *edgemap*
7:     add a to the applicable actions of s_t in *statetoactions*
8: **end procedure**

It can be easily seen that Algorithm 1 is complete and sound if the value of max depth parameter D is chosen to be enough high. Otherwise if there are still states in the last hierarchical level it means, not all of the plans terminated, therefore the value of the depth parameter has to be increased. After the appropriate number of iterations these algorithms provide a tree-structured search space which shows all the possible plans which can be apply for the initial state to complete the highest-level task. However, the goal of the HTN module is not to generate all the plans, but to map all the state-action pairs for the later MDP module. Therefore instead of determining the termination condition by defining a high depth value, it is enough to look on the dynamics of the stored mappings of the searching space. If the mapping are not increasing any more it means, the further plans will repeat already existing state-action sequences from already stored scenarios. In this way the searching space generation can be much faster, which bear a crucial part in simulation of human environments.

4.3 MDP module

The solution for an MDP problem is a policy, which maps at most one action for each state. Even there are more applicable actions in a state based on the HTN task network and MDP conditions, the optimal policy gives the one which has the highest expected utility to take the world to the desired state or accomplish all the tasks defined by the problem. This depends on the chosen optimization criteria. There are problems in

which the optimal plan is defined by its length or its highest probability to achieve its goal, but there are also different combination of criteria can be chosen such as probability to safeness. The policy then has to be trained based on these requirements to get its optimal form.

The learning procedure also has certain challenging aspects, which varies by methods. One of the most important factors is the speed of convergence, since the time need to be invested is strongly limited in many applications. Other aspect is the memory requirements of such a learning procedure. It is obvious that in human environment there can be enormous information which can have effect on the learning. Even if this data is highly reduced by an appropriate knowledge representation, a long plan can be easily intractable if all the history has to be saved. A third aspect is the optimal criterion of the learning algorithm. Many real world problem is simplified effectively by the appropriate representation to be handled well and an optimal plan can be found. However, in many cases there is no best plan, therefore the learning algorithm has to find if there is one, otherwise has to have considerations to handle the situation. Finally a fourth important aspect of chosen a learning algorithm for a model is a trade-off between fully automated procedure and supervised. In the second case, the domain expert has to contribute in the learning, which can make the procedure much faster and robust, but also result a natural bias for the system.

In the single-agent reinforcement learning model an agent interacts with its environment by choosing an action, which takes the world to a state which is rewarded if it reached certain goal conditions. The fundamental goal of the agent is to maximize its reward over the horizon. This can be finite, but even for infinite horizon a discount factor can result convergence for the algorithm. The reward can be a total sum of all received rewards but different formalization such the average reward can be also the goal of an agent. Therefore in our MDP framework the agent in state s_t chooses an action a which takes the world into the resulting state s_{t+1} and receive a real number representing a reward for choosing this particular a in s_t resulting s_{t+1}. In our work we relax this reward definition and make the reward only dependent on s_{t+1}. The policy which determines which action should be chosen is a collection of probability distributions over all actions for s_t. However, in our model resulting to the HTN approach this probability distribution is narrowed only for the applicable action in s_t. This result a sparse matrix for the policy distributions, which makes the learning much faster. The resulting action-selection policy can be interpreted as a transformation to an induced discrete-time Markov chain, which then results a more simplified model.

Fundamental tool to solve MDP algorithms is the state value function $V^{\pi}(s)$ given a policy π, which is the expected reward for a certain state. Similarly, Q-function $Q^{\pi}(s, a)$ defines the expected reward given the agent chooses action a in s_t following policy π, where the corresponding values are the Q-values. Generally, a deterministic policy is defined as $\pi : S \rightarrow A$. The problem is often that there is no a-priori knowledge of the optimal state function and Q-function therefore a technique is necessary to estimate them and find the optimal policy with their help. Dynamic programming methods are widely used for this purpose [5]. However, the components of the MDP have to be known such as the reward table and state transition model, which is not always the case. Monte Carlo and Inverse Reinforcement algorithms are appropriate for learning the rewards and find optimal policy based on experience, meaning of choosing an action and collecting the reward in each state [19].

4.4 Algorithms

Two classical dynamic programming methods for solving MDP problems are the value iteration and policy iteration.

To find the optimal policy, it is needed to calculate the optimal value function. The value iteration algorithm is to compute $V(s)$ which has to be proved to converge to its optimal value. The expression 4.1 to $V(s)$ is obtained by turning the Bellman optimality equation into an update role [5]:

$$V(s) \leftarrow \max_{a} \sum_{s'} Tr(s, a, s')[R(s, a, s') + \gamma V(s')], \tag{4.1}$$

where $Tr(s, a, s')$ is the state-transition model, $R(s, a, s')$ is the reward and γ is the discounting factor. γ is used to handle infinite horizon problems giving a weight for the newly acquired experience.

The value iteration is guaranteed to find optimal greedy policy in finite number of steps, even though the optimal value function might not have converged, which is often the case [6].

The Algorithm 4 shows the basic approach of value iteration. In this work we do not attempt to define sophisticated convergence criteria, we just state in the algorithm that the residual must be lower than a experimentally defined bound:

Another approach to find an optimal policy in a finite MDP is to directly update the policy itself instead of compute it from the state values. A simple algorithm to

Algorithm 4 Value iteration

Require: state s, residual δ, convergence criteria ϵ, state-transition model Tr, reward R, value function $V(s)$, action a, discount factor γ
Ensure: updated $V(s)$

1: **procedure** ValueIteration(s, δ, ϵ, Tr, R, $V(s)$, a, γ)
2: initialization of $V(s)$
3: **while** $\delta < \epsilon$ **do**
4: $\delta \leftarrow 0$
5: **for** all $s \in S$ **do**
6: $v \leftarrow V(s)$
7: $V(s) \leftarrow \max_a \sum_{s'} Tr(s, a, s')[R(s, a, s') + \gamma V(s')]$
8: $\delta \leftarrow \max(\delta, |v - V(s)|)$
9: **end**
10: **end**
11: **end procedure**

do that is alternating two steps: the *value iteration* and *policy iteration* step. The first step solves the linear equation system and computes the state value based on the actual policy which can be different from the optimal. Then in the second step it updates the policy.

Algorithm 5 Policy iteration

Require: state s, current value function $V^\pi(s)$, policy $\pi(s)$, state-transition model Tr, reward R, action a, discount factor γ
Ensure: updated $\pi(s)$

1: **procedure** PolicyIteration(s, $\pi(s)$, Tr, R, $V^\pi(s)$, a, γ)
2: initialization of $V(s)$ and $\pi(s)$
3: **while** policy is optimal **do**
4: $V^\pi(s) \leftarrow \sum_{s'} \pi(s, a) \sum_{s'} Tr(s, a, s')[R(s, a, s') + \gamma V^\pi(s')]$
5: **if** the policy is optimal **then**
6: **break**
7: **for** all $s \in S$ **do**
8: $b \leftarrow \pi(s)$
9: $\pi(s) \leftarrow \text{argmax}_a \sum_{s'} Tr(s, a, s')[R(s, a, s') + \gamma V(s')]$
10: **if** $b \neq \pi(s)$ **then**
11: policy is not optimal
12: **end**
13: **end**
14: **end procedure**

Policy Iteration often takes fewer iteration steps than value iteration but, on the other hand, value iteration consumes much less time in each iteration. The reason is the policy iteration has to solve a possibly large set of equations or use an iterative procedure similar to value iteration itself, and do it several times.

4.5 Q-learning

The chosen MDP learning method the Q-learning, proposed by [30] is a reinforcement learning technique. Many of the previously mentioned criteria are fulfilled by this method. It is an off-policy technique, which means independently from the used policy during the training it can find the optimal policy. This is very handy, when there is no prior information about the procedure and it can be combined with the benefit of HTN planning, which generates all the applicable actions for a state but it does not give preference for one. Q-learning is model-free, therefore it is well incorporable with our forward-chaining MDP planning algorithm and it has great memory efficiency due to the fact that the state transition history does not need to be saved. These class of algorithms for model-free problems are known as temporal difference methods [25]. Q-learning is guaranteed to find the optimal action-selection policy for any given (finite) Markov decision process (MDP) [10], which fulfills the above mentioned optimal criteria. This property is result of its basic approach to learn action-value functions by value iteration.

The update rule for Q-learning is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)) \tag{4.2}$$

which is proved to converge for the optimal policy Q* if these conditions are hold [31]:

- every state-action pair has to be visited infinitely often
- α must decay over time such that $\sum_{t=0}^{\infty} \alpha_t = \infty$ and $\sum_{t=0}^{\infty} {\alpha_t}^2 < \infty$.

The update rule is always performed based on a greedy and deterministic policy which is being improved. On the other hand, the actions which are chosen for the controller to train the optimal policy, are executed and sampled based on the action set which is determined by the HTN task network and MDP conditions. This is possible

because of the off-policy property of Q-learning, however, the sampling of for the policy training has strong impact on efficiency. The Q-learning is shown in Algorithm 6.

Algorithm 6 Q-learning

Require: state s, reward $R(s')$, discount factor γ, learning rate α
Ensure: optimal policy $\pi^*(s)$

1: **procedure** QLearning(s, $R(s')$, γ, α)
2: initialization of $Q(s, a)$
3: Initialization of s_0 as s
4: **while** the policy is optimal **do**
5: execute a
6: observe the resulting state s' and the respecting $R(s')$
7: $Q(s, a) \leftarrow Q(s, a) + \alpha(R(s') + \gamma \max_{a'} Q(s', a') - Q(s, a))$
8: **end**
9: **end procedure**

There are two parameters in the Q-learning algorithm: learning rate α and discount factor γ. α controls what is trade-off between the importance of the newly acquired information and the old value. Obviously, if it is 0 the agent does not learn from the new experience, otherwise if it is 1 the agent always take the most recent information into account. For our example we have to use low value such as 0.2 regarding to the probabilistic behavior of our model. γ determines the importance of the later rewards. Low value make the agent short-sighted, value which is close to 1 means the agent focus on the rewards in the future. Our chosen value is 0.8 respect to the value of α. The initialization of $Q(s, a)$ has also impact on the learning procedure. Since Q-learning is an iterative algorithm, initial condition is necessary for the Q table. It is often to initialize the $Q(s, a)$ table to full zero, because otherwise the so called optimistic initial conditions (values higher than zero) might lead to longer simulation because the special phenomenon when the initial value is higher than the updated.

4.6 Probability assumptions

There a few probability components in the existing framework, which has to be incorporated with mathematical assumptions into the algorithms.

First, we introduce a probability in HTN to choose a certain action. The applicable actions, which are defined by the HTN methods theoretically can be substitutionary

with each other. A method to be chosen has to only depend on its preconditions but not the order of the algorithmic call or any random behavior. Therefore the probability of choosing a certain applicable action in the given state has a probability PR_{HTN} defined as the following:

$$Pr_{HTN}(s, a) = 1/n, \tag{4.3}$$

where n is the number of all applicable actions in s, which is analogue with the number of applicable methods. This model could be extended with the probability of the likelihood of each methods, but in this work we assumes homogeneous distribution over the subjective chose of the applicable methods.

Second probability is coming from the MDP state-transition model, which represents the probability to execute a in s taking the world to s'. $Pr_{MDP}(s, a, s')$ is then defined with assuming Markov Property over the action sequence meaning the probability of each state-action transitions are independent. Formally:

$$Pr_{MDP}(s_n|a_{n-1}, s_{n-1}) = Pr(s_n|a_1, s_1, a_2, s_2, ..., a_{n-1}, s_{n-1}) \tag{4.4}$$

In our work we assumes Pr_{HTN} and Pr_{MDP} are independent, therefore with Markov property assumption, choosing a in s results s' with $Pr(s, a, s')$ as follows:

$$Pr(s, a, s') = Pr_{HTN} \times Pr_{MDP} \tag{4.5}$$

After introducing the Q-learning module, the Algorithm 7 shows how it is embedded into the existing probabilistic hierarchical framework. Algorithm 7 follows the same approach to Algorithm 1 with Algorithm 6, which results optimal policy for the defined search space by the HTN based searching algorithm.

4.7 Conclusion

In this chapter we introduced formal definitions for our planner. The HTN and MDP modules were introduced and implemented by the represented by pseudocodes. The main procedure consists of two conceptional parts. First, the searching space was built based on the HTN formalism and the MDP conditions. Then with the help of the tree representation of the possible state-transitions, the MDP algorithms result the solution policy. The Q-learning is guaranteed to converge to the policy, which gives those actions for every states, which has the highest expected utility. Probabilistic

behavior is captured by the learning algorithms to make the planner able to give the most probable plan as solution for the initial state.

Algorithm 7 Non-deterministic hierarchical MDP with Q-learning

Require: depth *d*, hashmap *nodemap*, hashmap *edgemap*, training iteration number *iter*, method library $m \in M$, action library $a \in A$, reward table $R(a')$, state-transition probability $Pr(s, a, s')$, highest-level task T

Ensure: optimal policy $\pi^*(s)$

1: **procedure** MainProcedure(d,$nodemap$,$edgemap$,$iter$,m,a,$R(s')$,$Pr(s, a, s')$,T)
2: initialization of $Q(s, a)$
3: **for** *iter* **do**
4: Initialization of s_0 as s
5: add s to the first hierarchical level
6: **for** each s in the current hierarchical level d **do**
7: **if** goal-condition is fulfilled on s **then**
8: **continue**
9: **else**
10: **for** each applicable m decomposes T **do**
11: a and $s' \leftarrow$ apply applicable actions
12: observe $R(s')$
13: $Q(s, a) \leftarrow Q(s, a) + \alpha(R(s') + \gamma \text{Pr(s,a,s')} \max_{\hat{a}} Q(s', \hat{a}) - Q(s, a))$
14: add s' to the list of states in the $d + 1$
15: **end**
16: **end**
17: **end**
18: **end procedure**

Chapter 5

Robust planning

Planning under uncertainty is handled by MDP successfully, various methods were developed to reasoning over probabilistic state-transition systems. However, robustness which is captured by foresight is not directly handled by the MDP formalism. In this chapter we are extending our model in a way to handle risk as a constrain for the planner. The extended model belongs to the family of Constrained Markov Decision Processes [2].

To define risk, [28] distinguishes between *accumulated* and *memoryless* risk measures. Accumulated risk is calculated respect to the whole action sequence from the starting state to the current state. In this way, a *total* risk has to be defined and updated after each state-transition to the evaluation and decision-making. *Memoryless* risk measure approach is analogue to Markov property and assumes the independence of the future expected risk from the past risk which was already taken into account. The model therefore assumes that even a risk factor has been associated with the past state-transitions, if the actions were successful, this risk is not relevant any more. To capture accumulated risk, it has to be incorporated into the state space and since risk is a probability distribution, the classical MDP has to be extended to Continuous MDP, which is undoubtedly more computational expensive [1]. Handling accumulated risk is still an ongoing research field, where different function approximation methods attempts to model and calculate the risk propagation over the action sequence. In [14] the authors present a sampling-based motion planner (CC-RRT), which generates optimal plans with handling process noise, localization error and uncertain environmental constraints. It guarantees probabilistic feasibility with modeling soft risk constraints and hard probabilistic feasibility bounds. They model the problem with a discrete-time system where the control problem defines cost functions to be optimized incorporating

risk. Both risk models capture aspects of risk handling, although the second one is closer to MDP approach, because in this formalism the Markov property is also hold. In this work we are focusing on memoryless risk measure.

In the previous chapter we formally described our probabilistic hierarchical planning problem with Definition 8, where the optimization problem attempts to find the optimal policy in a way:

$$\pi^* = arg\max_{\pi} E[\sum_{t=0}^{T} \gamma^t R(s_t)] \tag{5.1}$$

In this chapter we extend this model to a Constrained-MDP (CMDP) founding on the model in [29].

Definition 9 *A CMDP is a tuple <S, A, app, PR, R, C>, where S, A, app, PR and R are defined in the Definition 8 and C is a constrained-model. The constrain-model* $C(s) : S \rightarrow \mathbb{R}\ s \in S$ *defines a constrained penalty.*

The constrained penalty is used to formally describe risky zones, which the planner should handle carefully.

In this way our problem is defined as follows:

$$\pi^* = arg\max_{\pi} E[\sum_{t=t_0}^{T} \gamma^t R(s_t)] \tag{5.2}$$

$$s.\ t.\quad E[\sum_{t=t_0}^{T} C(s_t)] \leq \lambda \tag{5.3}$$

The constrain model in this way defines a non-negative value in the interval [0,1] for every state. We can define hard constrains with setting $C(s)$ to 0, which does not allows for the planner to reach the certain state. A value between 0 and 1 determines a soft feasibility constrain which describes a probability-like measure to violate the constrain respect to the given state.

To handle risk in the MDP model, the reward can be combined with a penalty measure, which is a compensating element in the Equation 5.4 to reduce the expected utility corresponding to s_t. Therefore the reward model is extended in the following way:

$$\hat{R}(s_t) = \beta \times R(s_t) + (1 - \beta) \times Pe(s_t, a), \tag{5.4}$$

where $Pe(s_t, a)$ is the penalty model and β is a positive number, which defines the trade-off between conservative and risky decisions by taking penalty into account when reward is collected by the agent. Lower β results more risky decisions, because the decreased importance of the reward compared to the cost. The behavior captured by this formula is way more complex and finding the balance between gain and cost in general needs further convex optimization work.

However, it can be seen intuitively, that this model led to losing the characteristic of both the reward and cost separately. There are many cases where the two measures can balance each other. In [29] it is shown that there are cases when there is no appropriate value to β, therefore there is not way to avoid the phenomenon that the system become either too conservative or make too risky decisions with using this formalism of risk.

The proposed solution captures both the characteristics of constrain and reward models.

5.1 Proposed solution

To extend our model to handle risk as a constrain, the previously introduced algorithm has to not only optimize the Q-value respecting to the reward but there is need to check for constrain feasibility beyond the horizon. For ensure feasibility, two value function have to be tracked $V_R(s)$, which is associated with the reward $R(s, a)$ and $V_C(s)$, which is connected with the constrain model $C(s)$. Therefore the forward searching algorithm has to optimize based the following set of equations:

$$\pi^*(s) = arg \max_{a \in a_C} Q_R(s, a) \tag{5.5}$$

$$a_C = \{a : Q_C(s, a) < \lambda\} \tag{5.6}$$

$$Q_R(s, a) = (1 - \alpha)Q_R(s, a) + \alpha(R(s') + \gamma Pr(s, a, s') \max_{\hat{a}} Q_R(s', \hat{a}) - Q_R(s, a)) \tag{5.7}$$

$$Q_C(s, a) = (1 - \alpha)Q_C(s, a) + \alpha(C(s') + \gamma Pr(s, a, s') \max_{\hat{a}} Q_C(s', \hat{a}) - Q_C(s, a)) \tag{5.8}$$

In Algorithm 8, $Q_C(s, a)$ is evaluated for the now extended model according to the above described mathematical model. The general goal of the planner is to maximize the expected reward which was received, while keep the feasibility constrain always under the defined threshold λ.

Algorithm 8 Non-deterministic hierarchical constrained MDP with Q-learning

Require: depth d, training iteration number $iter$, action $a \in A$, reward $R(s)$, state-transition probability $Pr(s, a, s')$, constrain model $C(s, a)$, feasibility bound λ
Ensure: optimal policy $\pi^*(s)$

```
 1: procedure PolicyLearning(d, iter, a, R(s), Pr(s,a,s'), C(s), λ)
 2:     initialization of Q_R(s,a) and Q_C(s,a)
 3:     for iter do
 4:         s ← s_0
 5:         add s to the first hierarchical level
 6:         for each state s at all hierarchical level d do
 7:             if goal-condition is fulfilled on s then
 8:                 continue
 9:             else
10:                 for each a ∈ app(s) based on HTN and MDP do
11:                     a, s' ← apply a
12:                     observe R(s') and C(s')
13:                     update Q_R(s,a) according to Equation 5.7
14:                     update Q_C(s,a) according to Equation 5.8
15:                     add s' to the list of states on level d+1
16:                 end
17:         end
18:     end
19:     a_C ← [a : Q_C(s,a) < λ]
20:     π*(s) ← arg max_{a∈a_C} Q_R(s,a)
21: end procedure
```

5.2 Soft constrain and hard feasibility constrain

The above described model eliminates those state-transitions, which might violate hard feasibility constrains. This means after convergence the policy eliminates those action which might lead to risky states. In this way we define a policy Π, which can be described as follows:

$$\Pi_i(s_t) = \{\pi(s_t) : E[\sum_{t=t_0}^{T} C(s_t)] < \lambda_i\} \tag{5.9}$$

On the other, the model allows state-action transitions which have a certain level of soft risk if it does not exceed the the hard constrain bound λ_i respect to Π_i. To use this information to evaluate Π_i we need to define first that constrain bound λ_{min} which results a policy for the given problem, in such a way that it allows at most one

action for each state. Now a constrain margin measure $\lambda_{margin}(\Pi_i)$ can be calculated for an arbitrary Π_i, according to Equation 5.10. It gives a measure about what is the margin of that specific policy Π_i given to the described problem.

$$\lambda^{\Pi_i}_{margin} = \lambda_i - \lambda_{min} \tag{5.10}$$

5.3 Bisection search

Algorithm 9 Bisection Search

Require: $\lambda interval : [min, max]$, tolerance TOL
Ensure: finding λ_{min} with TOL

```
 1: procedure BisectionSearch(min, max, TOL)
 2:     NumberOfFail = 0
 3:     λ ← (min + max)/2
 4:     PolicyLearning(λ, d, iter, a, R(s), Pr(s, a, s′), C(s))
 5:     SuccessFlag ← is there any legal plan based on the generated policy
 6:     if NumberOfFail > 2 then
 7:         Return 0
 8:     else if max − min < TOL and SuccessFlag then
 9:         Return λ_min
10:     else if SuccessFlag then
11:         NumberOfFail = 0
12:         Return BisectionSearch(min, λ, TOL)
13:     else
14:         NumberOfFail + +
15:         BisectionSearch(λ, max, TOL)
16: end procedure
```

To evaluate the solution policies as function of the constrain margin λ_{margin}, first λ_{min} has to be found . For this, bisection search strategy is used. The bisection of binary search is a root-finding method in mathematics and economics, which is widely used because its straightforward concept and its computational efficiency, which does not exceed $O(log(n))$. The basic idea is similar to the Guessing game, where the guesser can ask to get closer to the number which is figured out by the other. A wise chose is to half the possible interval, with asking is the number higher or lower than the middle value and so the possible choices only could be in one of the half intervals. The searching algorithm therefore needs an ordered list of items. In this work based on

the definition of λ^{Π}_{margin}, it can be easily seen that with increasing the value of λ_i, the distribution of the solution policy for each states will also increase respectively, allowing more actions to execute. Based on this analogy, Algorithm 9 gives an implementation with pseudo-code, which is used to find λ_{min}.

5.4 Test case

In the test case a modified HouseCleaning domain will help to evaluate the planner in the next chapter.

Figure 5.1: HouseCleaning domain

Figure 5.1 shows the domain setting. The house has three rooms: living room, bathroom and kitchen. The robot can access the bathroom and kitchen through stairs from the living room. There is a certain probability to failure to climb on the stairs, but we assume either the robot fail one time it can correct its action and manage to successfully reach the destination. There is a terrace which connects the bathroom and kitchen, which is a faster way between them. On the other hand, there is a certain

probability for failure and risk if the robot fail to successfully execute the action, it can stuck into that narrow path. This trade-off between risky but shortest plan versus low probability and longer plan will be detailed and evaluated with our planner. The robot has to sequence its actions based on the cleaning high-level tasks, which are the followings: dusting all the rooms, then vacuuming and mopping them. While the vacuuming needs only the presents of the performer and the cleaner cart, the dusting and mopping actions needs clean water in the cleaner cart, which has to be therefore changed after every of these two actions. There are taps for cleaning the water in every rooms, but there is a probability to fail and if it happens in the living room it gets wet which violates the safety constrains. This results another trade-off between efficiency versus safe operation. The house is modeled as a grid-space, where the state description is limited by the used literals.

5.5 Conclusion

In this chapter the planing model was extended by a penalty and a constrain-model, which attempt to capture notions of robustness. The penalty model gives possibility to find shortest plans by compensating the reward with a penalty for certain states and actions. To handle risks an additional Q-value is introduced to store the constrain value functions. The solution policy in this way will give that action for every state, which has the highest expected utility and not exceed the maximum risk level. In the next chapter the evaluation will give more details about the introduced trade-off problems.

Chapter 6

Evaluation

In this chapter we outline the capabilities of the proposed planner. Showing and evaluate the HouseCleaning domain for different test cases, the scenarios highlight the aspects the robust robot planning. The evaluation is in top-down fashion, we start to evaluate the final model first, then going deeper and evaluate the modules separately.

6.1 Modules

The program itself is broken into three pieces for better understanding and faster operation. The first module is related to the HTN planner. It builds up the state-action searching space based on the previously detailed algorithms in the first section of Chapter 4, creating a graph-based model with all the connecting state nodes. The outcome is a set of mappings for the training procedure in the second MDP module, which finds optimal policy based on the state-transition, reward, penalty and constrain models. Finally, the third module is a simulator, which uses the trained policy as input to chose actions in sequence and in this way create plans for the problem definition. The simulation combines the background information from the problem description and random behavior to illustrate the online execution with uncertain environment.

6.2 Robust reasoning over probabilistic hierarchical workflows

After building the HouseCleaning domain as described in the previous chapter, the initial state is defined as follows. The performer is in the bathroom, the cleaner cart is

in the kitchen and additionally the cleaning water is assumed to be clean initially. The high-level task is to clean all the rooms by dusting, vacuuming and mopping them.

The first test case shows a simple scenario, which can be used as a trivial solution for later comparisons. The MDP state-transition probabilities are shown in the Figure 6.1, which are simplified to relates only for actions. The probability values define how probable each operator is, the higher number is associated with higher chance for success. As it was introduced in Section 4.6, the state transition probability Pr_{MDP} has to be multiplied with the HTN probability Pr_{HTN}, which will be used as a combined probability in the update functions. This resulting combined probability table describes our state-transition model for the problem.

	Probability of success
Moving (with or without the cart)	
Between living room and kitchen	80%
Between living room and bathroom	80%
Between kitchen and bathroom	60%
Cleaning	
Dusting	95%
Vacuuming	95%
Mopping	95%
Changing the cleaning water	90%

Figure 6.1: State-transition probability Pr_{MDP}

The reward model defines a high number only for one state, when all the cleaning tasks are accomplished, therefore the house is totally clean. No intermediate rewards are used to make the learning as flexible as possible. The constrain and penalty models are not initialized in this model.

The Figure 6.2 shows the generated plan based on the converged policy. Note that since there is no intention for finding the shortest plan, the performer chooses the most probable actions in every state by their reward value functions. The most probable plan is to use the stairs in the living room when moving operation is needed between the kitchen and bathroom.

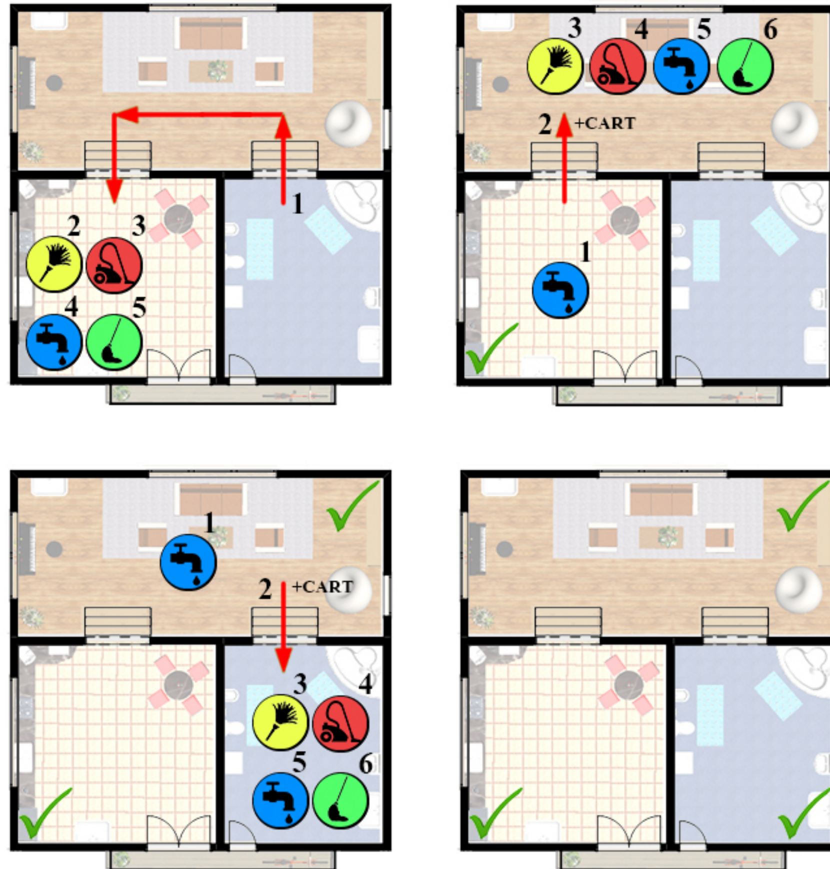

Figure 6.2: First test case - Most probable plan

The second test case is the extension of the first one with the penalty model, which is incorporated into the reward value functions of the reward model. By introducing a compensating term which reduces the value of the function for certain states-action pairs, penalties can be given for actions or states. Our goal is to train the policy for finding the shortest plan by avoiding as much moving operations as it possible (moving with or without the cleaning cart). Note that, handling penalty as a negative reward can make the planner to shortsighted or risky as it was introduced in Chapter 4. However, in our final formalism risk is handled by a separate value function, which allows us to neglect this side effect and low value for β as coefficient for the penalty values to balance the reward and chose always actions for the shortest plan.

Figure 6.3 shows the generated plan, in which the performer prefer to use the terrace to move between the bathroom and kitchen, instead of going through the living room, which is obviously a longer path.

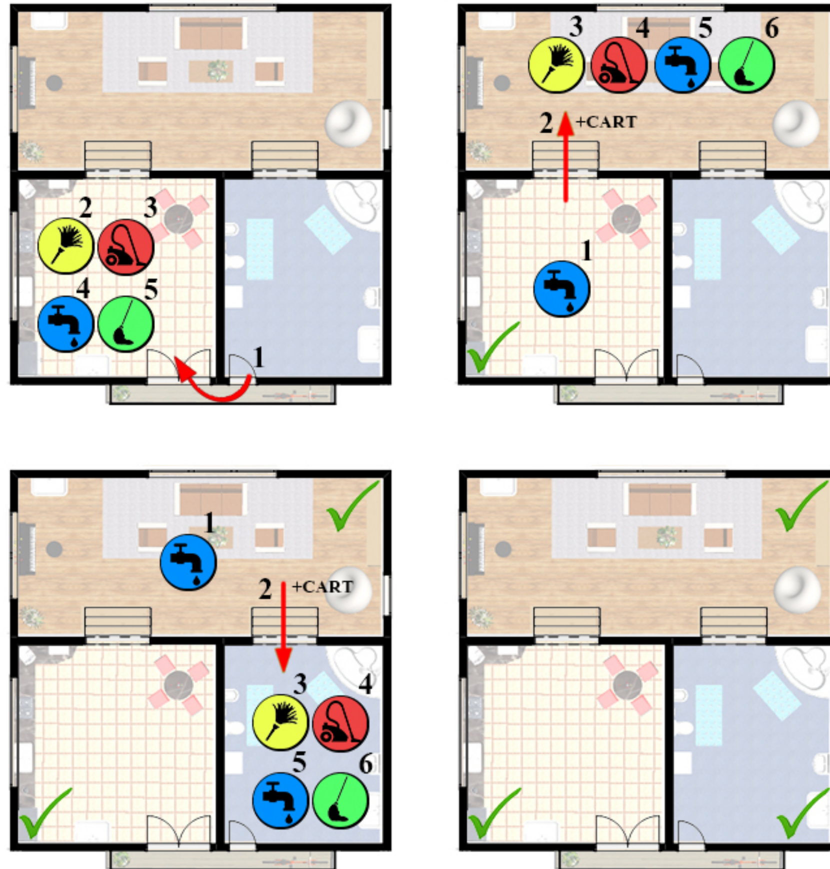

Figure 6.3: Second test case

In the third model the constrain model is also initialized. In the HouseCleaning domain we distinguish two abstract risks. One is associated with the navigation on the terrace, where due to the lack of enough space we assume the robot can stuck, which would lead to unsuccessful plan execution, because without human help, the cleaning tasks cannot be accomplished any more. In this way we highlight the difference between the probability of state-transition and related risk. Even the moving through the terrace is faster and more probable than uses the stairs for many times in the living room, but if failure occurs on the terrace it ends up to a fatal error. This is then much worse than having temporal errors because of the stairs which we assume cannot cause fatal error, only need for more trying.

The other risk-source is the cleaning water changing operator in the living room, due to the possible outcome of flooding the living room in case of operator failure. From this reason, to satisfy the feasibility constrain, the robot has to go to the kitchen

or bathroom to change the cleaning water any time it is needed. Figure 6.4 shows the constrain definition.

	Probability of risk
Stuck at the terrace	90%
Flood the living room	60%

Figure 6.4: Constrain model $C(s, a)$

Note that both risks are related to the feasibility bound, meaning there is a probability (based on feasibility bound) for facing to risk during execution of the plan. To capture this behavior the previously introduced λ can be adjusted. Figure 6.5 shows the scenario when both risks are introduced into the constrain model and the policy gives the following plan applying λ_{min}.

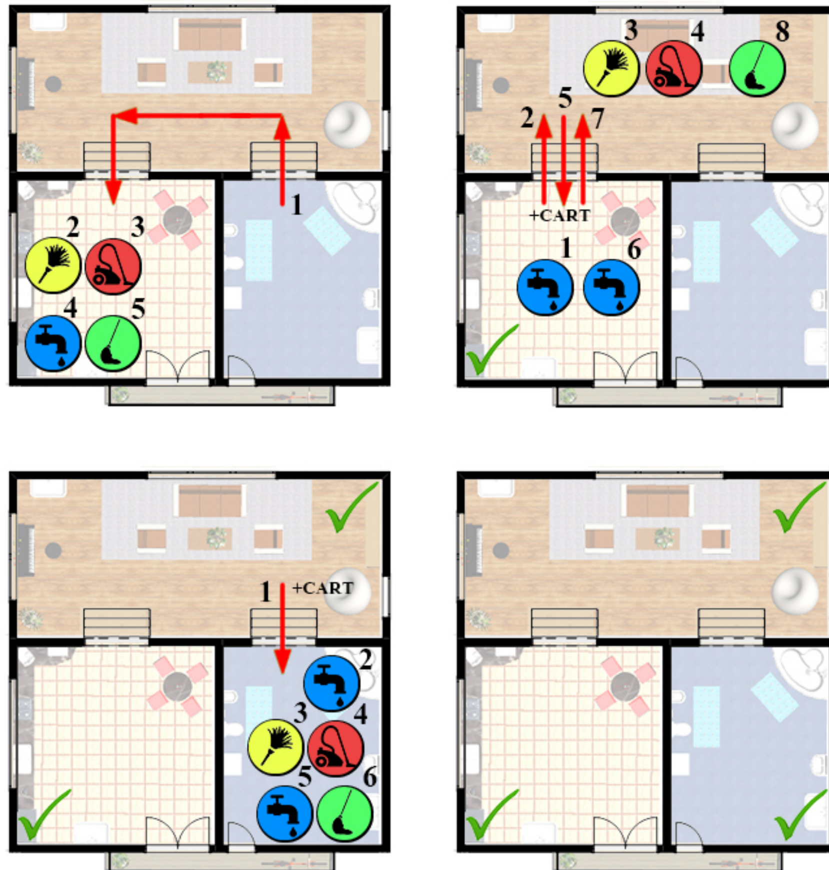

Figure 6.5: Third test case

One might see, to fulfill every feasibility requirements the robot has to chose not the most efficient and shortest plan, but has to take the possible risks into account with higher priority. This is analogue to the consideration about safety regulations in human-robot interaction, where the dangerous actions are eliminated even those could lead to higher efficiency.

Even this model first eliminate the risks and then optimize the plan for being the shortest and most probable, one might realize, there is a shorter version for execution without violating the constrains. Figure 6.6 shows the plan where the robot take the advantage of changing water in the bathroom and immediately execute the cleaning tasks there before going back and finish the living room. This is faster and more probable. The difference between the third and fourth test case is related to interleaving planning. However our framework is capable to generate the interleaving plan for the HouseCleaning domain, but this is not guaranteed for other problems as well, because designing interleaving behavior needs further intelligent strategies, which will be summarized in the next chapter.

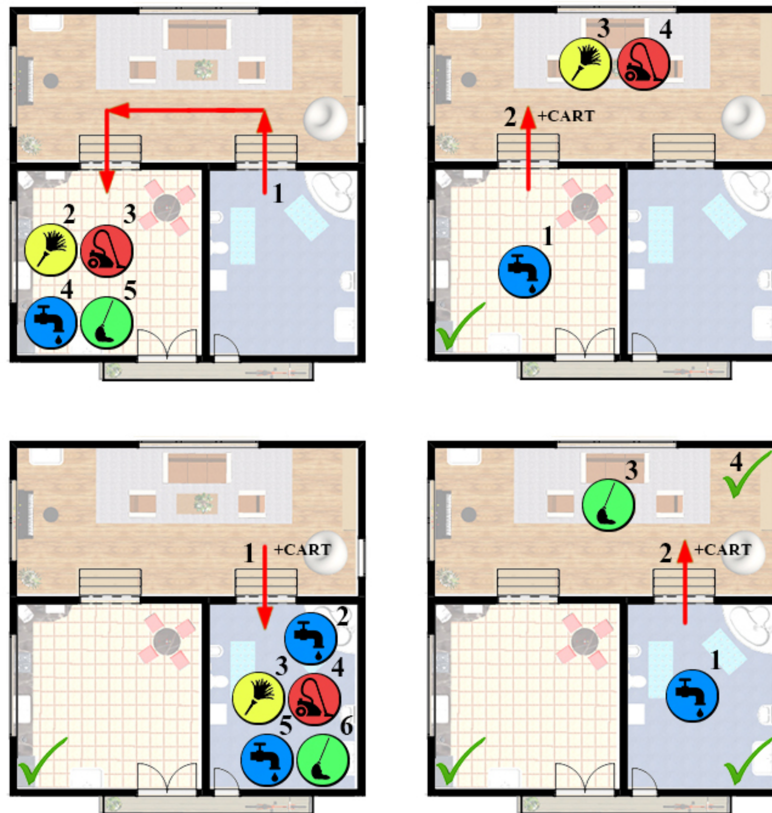

Figure 6.6: Fourth test case

The most important measure for the generated plans is the probability of success. To evaluate the generated policy Figure 6.7 summarizes the results of the simulator using different λ values. Every data point was evaluated based on 1000 plan executions. The upper graph shows the probability of success as function of λ, while under the average plan length can be seen.

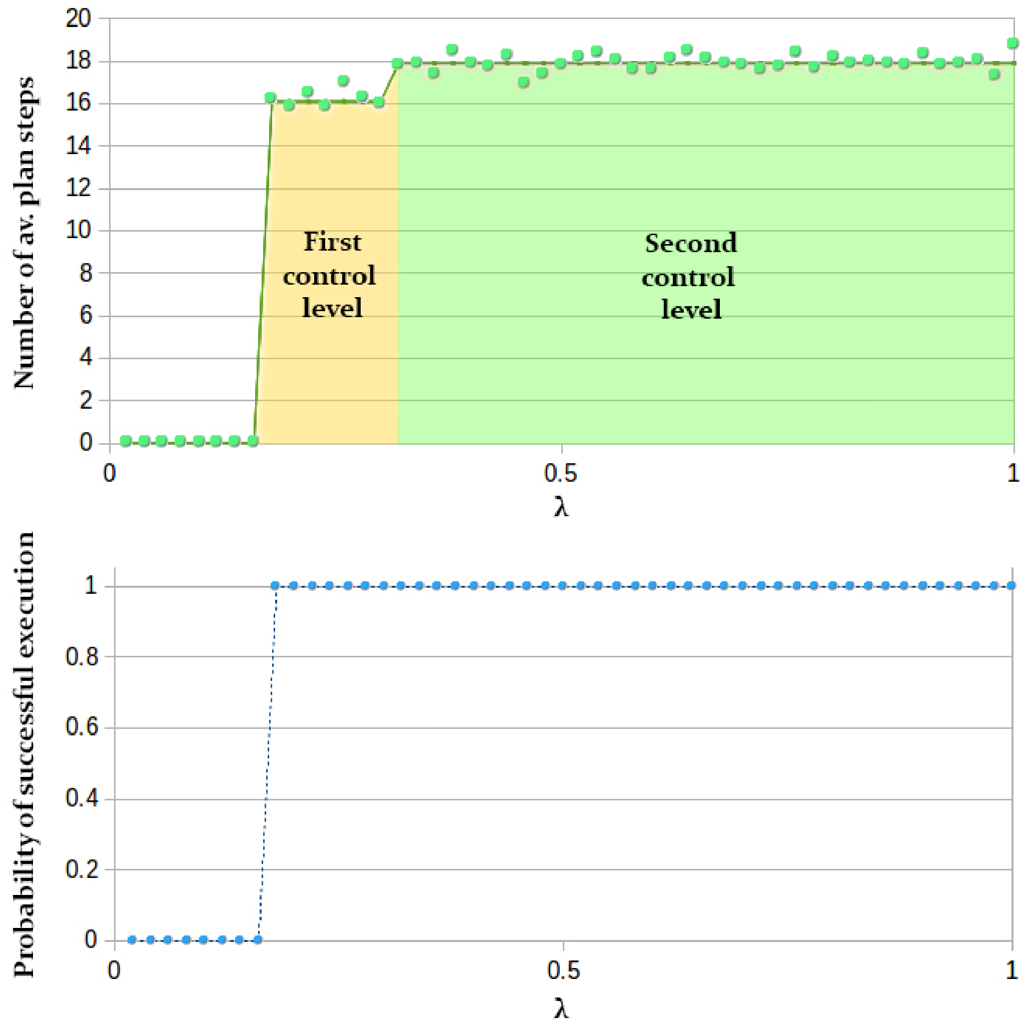

Figure 6.7: Length and probability of success of the generated plans

Similarly to the control engineering problems, one can define risk levels on the previous multilevel step-function. In our simplified domain, the human controller can chose, which risk level is acceptable for the cleaning plan and give preference in the trade-off between plan length and risk. Having shorter plan might cause risk with higher probability, while choosing long plan can eliminate all risk sources.

6.3 Evaluation of the modules separately

In this section both the two planning modules are evaluated separately. First the HTN module is under the scope. . HTN planner uses a ordered or unordered sequence of methods to decompose the task and define the sequence of actions, however the purely combinatorial approach investigate all the legal sequences of actions. Since this thesis focuses on hierarchical workflows, which covers most of the human activities, the HTN approach highly applicable, which reduce the searching space by introducing methods.

To illustrate the phenomenon with an example in our domain, let imagine the performer can move around in the house with or without the cleaning cart, which could lead to many different legal plans, but obviously this is not an optimal solution for the cleaning problem. Representing with numbers there are 16,384 different possible state configurations of the world, which lead to a few hundred thousands state-action pairs based on the purely combinatorial search. Regarding to the fact, that the performer can move infinitely between the rooms without doing cleaning task, this approach means infinite amount of legal plans. Compared to the HTN approach, there are approximately a several thousands of state-action pairs, because the methods gives instructions for the execution of the cleaning tasks. Since the methods are guaranteed to decompose the tasks to actions there are no more than a several hundreds of possible plans using HTN task network.

The CMDP module combines the probabilistic behavior with a risk sensitive reward based evaluation model. Compared to the HTN module, where the action sequences are defined by the methods, in the MDP module each actions has precondition and postcondition, which are evaluated based on the state-transition, reward, penalty and constrain models. The MDP module compared to the purely combinatorial searching strategy reduces the searching space dramatically as well. Figure 6.8 represents the difference by showing the confusion matrix of the policy based on HTN and MDP. It can be easily imaginable how sparse is this matrix compared to the full matrix related to the combinatorial approach.

On the right side of Figure 6.8 one can see that theoretically all actions can be executed at each state if the purely combinatorial approach is used for the training. On the left side, the confusion matrix shows that in case of MDP there are many actions which are not applicable because the preconditions are not fulfilled by the state. The reward value for each state-action pair is the result of the training procedure, which gives preferences for executing an action instead of another.

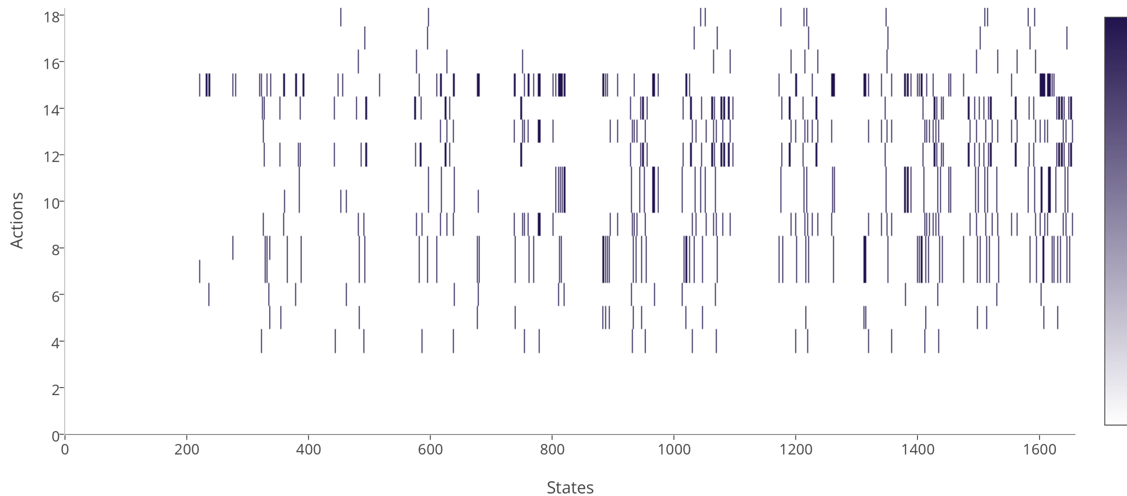

Figure 6.8: Sparse confusion matrix of HTN-MDP searching space

To specify the confusion matrix for our CMDP module, the reward function values are balanced with the penalties and scaled by the probabilities every time when the update function is called. The last element of the CMDP module is the matrix of risk values, which can be interpreted as separate table which eliminates all the risky state-transitions independently from its reward function value.

Figure 6.9 illustrates schematically the state-transition search space reduction by the different components of modules. The original state-transition searching space consists of all state-action pairs, this is the purely combinatorial searching space. The first set is defined by HTN, which gives state-action pairs, which are defined by the methods. The second set is determined by the MDP preconditions and postconditions. Finally the third set collects all the state-action pairs, which are not risky or with other words lead to constrained states. Mathematically while the HTN and MDP module define two surfaces respectively. These sets are assumed to be independent, therefore the summation of the two results a new surface, which is the controlled space, which lead to legal plans. The constrain model is then a third surface which can be interpreted on the same region as the controlled space surface and it cuts down those state-transition elements, which violates the feasibility constrain. Finally the resulting set of state-transition elements determines the optimal region, where based on λ the soft risk level can be adjusted.

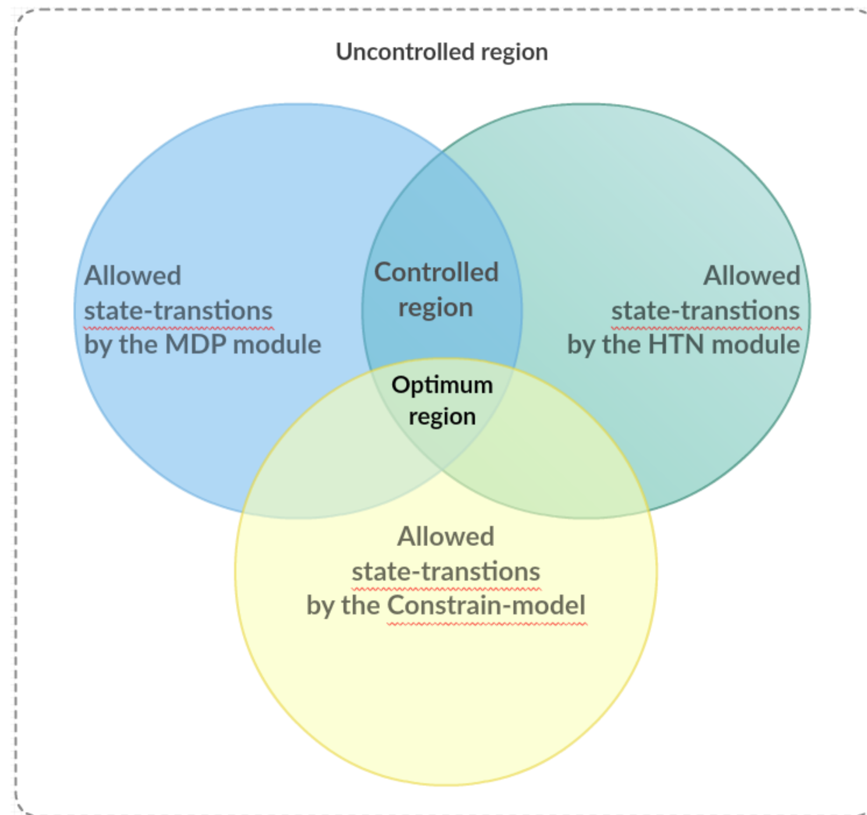

Figure 6.9: Sets

6.4 Conclusions

In this chapter four test cases were evaluated to test the capabilities of the proposed planner. The first case was to find the most probable plan without taking other models into account. To generate the most probable and shortest plan as a second test case, the penalty model was initialized for giving penalty for every moving actions. The third test case showed the planner capabilities by combining the constrain model together with the previous two. This plan gave the most probable and shortest plan taking risks into account. However interleaving behavior was not incorporated into our model directly we represented a fourth test case which could be generated for this specific problem, but in general further strategies has to be implemented to guarantee to find interleaving plan for other domains as well. In the next chapter this behavior together with other extensions and limitations will be discussed.

Chapter 7

Discussion

In this chapter we attempt to give a discussion over relevant selected topics: Position the proposed planner from a more general prospective, evaluate and analyze notions of robustness respecting to the planner, highlight limits of the planner and give proposals for extensions and future work.

7.1 Planner as a black box model

After going into details of each module and their functionality, in this section the possible interfaces and inputs of the model is discussed more in general.

The proposed high-level planner needs to interact with the low-level planning system meaning both a new input source and a new output channel for the proposed planner. After decomposing the tasks into actions, the low-level planner has to be called to execute them in an appropriate and precise way. The first task of the interface is the communication of the goals, the high-level actions and the constrains to the low level motion planner. Furthermore the motion planner has to estimate the time to be needed, the risk, preferences and further circumstances which has to be communicated back to the high-level decision making system. Finally, after the execution the motion planner has to report the results together with the motion feasibility, possible failures and their repair. This experience has to be evaluated and learnt to make the future planning more successfully. In this way the knowledge representation of the high-level planner has to be updated all the time. This means need for compact representation and efficient learning processes. On operator level, these experiences can be stored with help of suggesters, which give feedback for the high-level planner if the operator is called to increase the probability of successful online execution.

On the method level, we already highlighted the importance of human-like planning. One very important advantage of using human-like methods is their compatibility with many of human based knowledge storage. Take for instance, a receipt for a meal can be imported from a cooking book or webpage directly into the form of a HTN method. The process on the online based knowledge is strongly connected to different research areas such as Natural Language Processing, Big data or Social Network studies, which are connected but not part of our work.

Methods as personal preferences and way of thinking can be interpreted and associated with profiles for each human operator. When the robot plans based on its knowledge base, it will be biased by the human operator who adjusts the operation. The human experience and preferences can be stored by personal profiles. These can be used not only a specific individual, but can help to develop a common knowledge-base, which increase the service level for all the users. In this way, the probability and risk estimation can be more foresight and complex, adjusting for even special unique cases, which can be crucial in safety regulations and prevention of accidents.

7.2 Robustness

In our interpretation robust planing is captured by being prepared for risks and planing regarding to the probabilistic behavior of the actions. The proposed planner eliminates does state-transitions which might violates the safety constrains of the plans. This is analogue to those safety regulations in the everyday life human interaction, which is always kept in mind to avoid accidents. In the knowledge base however risk and similar abstract constrains has to be handled as flexible as it is possible. By defining an action to be risky or state to be risky only by itself regardless for other circumstance might lead the planner to generate to conservative plans. For a planner to be intelligent to distinguish risk is a hard challenge, which needs further robustness considerations to be incorporated. An important factor to design robust planner is the adaptability. This notion requires to have diverse and wide spectrum of the possible choices to find optimal solution for the given situation. However a naive representation is also can work robustly if it only have clear self-awareness of its capabilities and limits. Adjustable autonomy allows for even simple frameworks to find an appropriate control level, on which the system can guarantee robust operation.

7.3 Limitation

One of the most important limitation factors of our framework is about the symbolic representation of the world. However this simplify many real-world problems, in human environments with human-robot interaction the symbolic representation can be not tractable. Theoretically as the natural languages the world can be described by finite number of predicates and argument, but in planning problem definitions, this can be too demanding challenge for both the domain experts to design and the computer program to calculate with. Operators, methods hardly can be described by fix preconditions and postconditions, the variation of execution needs flexibility and many times continues world representation. STRIPS and PDDL formalism suffer from the demanding labeling need. There are several solutions for attribute-based representations to simplify and relax the definitions, but appropriate knowledge representation for human environment is still an emerging research field [24].

An important but limiting assumption the Markov property appears more times in our formalism. By allowing to assume that events and decisions are often independent from each other is correct for small scale systems and problems. However in real life problems the decisions and events are highly coupled even if the connection is far not obvious. Having effect on other people and their actions is part of our life. In a sophisticated planner representation therefore further considerations has to be taken into account. Above the local conditions, which allows to simplify the decision making procedure, global conditions have to be combined, which couple the system by introducing advanced abstract correlations. In classical physics it was proven that even the particles has seemingly independent movement, observing the whole system there are global conditions which introduce interconnection between the individual particles. To capture this behavior the partition coefficient and partition function is used in physics and graphical probabilistic model studies respectively. In our model the HTN probability inherits this property of the methods and decisions, however in later works these connection has to be investigated in more detail.

The earlier mentioned interleaving planning opens up great opportunity for further work as well. It is human practice to manage different task in the same time or accomplish tasks together by mixing the sequence of different procedures. People who can successfully do multitasking can achieve higher efficiency. In planning interleaving behavior can be handled in a naive way, making possible to do every action, which are legal, which take the planing procedure closer to the combinatorial approach, but this is

obviously computationally highly inefficient. Therefore further searching strategies has to be incorporated, probability models has to handle the not-logical thinking and often emotional-based decision making. This is not possible without introducing advanced plan and intention recognition.

Finally, a forth limitation aspect is a fundamental problem of all planning problems, which is related to the reduction of searching space. On one side for making prudent decisions many of the possibilities has to be analyzed, however this is introducing great amount of information. Then this space has to be reduced for that set, which consists only the appropriate solutions. The way of reducing the space is crucial. We implemented state of the art solutions to handle this challenge, however for more complex problems further strategies are needed. To extend and the searching capabilities various Monte-Carlo searching strategies can be used, which uses probabilistic, likelihood or mathematical conventions to chose actions for learning or analyzing.

7.4 Extension of the proposed model

As in the chapter 2 it was discussed uncertainty and probabilistic behavior can be considered related to the present and to the future. Markov Decision Process is an elegant formalism to handle uncertainty in the future and Partially Observable Markov Decision Processes are available to handle noise in the present state. Our framework captures only a few aspects of uncertain environment such as probabilistic state-transitions, but further aspects such as dynamic environment or adversarial interaction has to be investigated by help of multi-agent planning consideration or based on game theory.

A possible extension of our framework is its adaptation for human-robot interaction and multi-agent planning. In the near future robots will assist human in various tasks in their home. For this robot has to implement safe, fluent and efficient assistance. The key components of unobtrusiveness is timeless, meaning the robot should not introduce time delay for the operation; non-interruption to be autonomous and not interrupt human inappropriately; safety; predictability to make the human-robot interaction feasible and comfortable for humans; fluency and proactivity to make intelligent decisions and being capable to learn. A part of the above mentioned notions of unobtrusive assistance are fulfilled by the proposed planner such as safety and non-interruption, but many of them need for further investigation and formal implementation.

Chapter 8

Summary

In this thesis we introduced a human-like planner, which successfully handles uncertainty and generates robust plans for probabilistic hierarchical workflows. First, the Hierarchical Task Network planning approach was described, which decomposes high-level tasks into lower-level tasks and then generates sequence of actions. However this approach reduce the searching space enormously and give a human-like planning strategy, there are several open challenges regarding to robust robot planning. After introducing the Markov Decision Process framework, which is a reward based stochastic reasoning formalism, we extended this model with a feasibility bounded constrained model. Hard feasibility constrain and soft risk measures were described to evaluate the generated plans based on robustness. In the evaluation section, we highlighted the capabilities of our proposed planner. In the discussion chapter the different approaches were detailed together with the limitations and overcomes of our model.

In Artificial Intelligence Autonomous Planning there are still wide range of unsolved challenges. Different models and approaches performs differently depending on the environment and requirements. General solutions are highly computationally expensive, which today leads the planning communities to develop more specific solutions. We believe in the near future robots will assist humans unobtrusively in various household tasks, this thesis attempts to investigate notions of robust robot planning for bringing this vision closer.

List of Algorithms

1 HTN Search-space building algorithm 21
2 Apply a method 22
3 Update the searching space for MDP module 23
4 Value iteration 26
5 Policy iteration 26
6 Q-learning 28
7 Non-deterministic hierarchical MDP with Q-learning 30
8 Non-deterministic hierarchical constrained MDP with Q-learning 34
9 Bisection Search 35

List of Figures

2.1 Conceptional model of AI planning 4
2.2 Conceptional model of HTN planning 6
2.3 Non-determinism represented by binary decision tree 7

3.1 HouseCleaning domain 16

5.1 HouseCleaning domain 36

6.1 State-transition probability Pr_{MDP} 39
6.2 First test case - Most probable plan 40
6.3 Second test case 41
6.4 Constrain model $C(s, a)$ 42
6.5 Third test case 42
6.6 Fourth test case 43
6.7 Length and probability of success of the generated plans 44
6.8 Sparse confusion matrix of HTN-MDP searching space 46
6.9 Sets 47

List of listings

1 Operator description 11
2 Method description 12
3 Axiom description 13
4 Problem description 14

Bibliography

[1] Alpcan, T. and Başar, T. (2010). *Network Security: A Decision and Game-Theoretic Approach*. Cambridge University Press.

[2] Altman, E. (1999). *Constrained Markov decision processes*, volume 7. CRC Press.

[3] Au, T. C., Kuter, U., and Nau, D. (2009). Planning for interactions among autonomous agents. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 1–23.

[4] Ayan, N., Kuter, U., Yaman, F., and Goldman, R. (2007). HOTRiDE: Hierarchical ordered task replanning in dynamic environments. *Planning and Plan Execution for Real-World Systems - Principles and Practices for Planning in Execution: Papers from the ICAPS Workshop*.

[5] Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition.

[6] Bertsekas, D. P. (1987). *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[7] Estlin, T. A., Chien, S. A., and Wang, X. (1997). An Argument for a Hybrid HTN/Operator-Based Approach to Planning. In *Recent Advances in {AI} Planning, 4th European Conference on Planning,ECP'97, Toulouse, France, September 24-26, 1997, Proceedings*, pages 182–194.

[8] Gelly, S. and Silver, D. (2008). Achieving Master Level Play in 9 x 9 Computer Go. 1:1537–1540.

[9] Ghallab, D. M. M. (1998). Pddl - the planning domain definition language.

[10] Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA.

[11] Ilghami, O. (2006). Documentation for JSHOP2. *Department of Computer Science, University of*.

[12] Kaelbling, L. P. and Lozano-Perez, T. (2013). Integrated task and motion planning in belief space. *The International Journal of Robotics Research*, 32(9-10):1194–1227.

[13] Levine, G., Kuter, U., Rebguns, A., Green, D., and Spears, D. (2012). Learning and verifying safety constraints for planners in a knowledge-impoverished system. *Computational Intelligence*, 28(3):329–357.

[14] Luders, B. D., Karaman, S., and How, J. P. (2013). Robust sampling-based motion planning with asymptotic optimality guarantees. In *AIAA Guidance, Navigation, and Control Conference (GNC), Boston, MA*.

[15] M. Aarup, M. M. A. (1994). OPTIMUM-AIV: A knowledge-based planning and scheduling system for spacecraft AIV. *In Intelligent Scheduling*, pages 451–469.

[16] Molineaux, M., Aha, D. W., and Kuter, U. (2011). Learning event models that explain anomalies. In *Explanation-Aware Computing: Papers from the IJCAI Workshop*, Barcelona Spain. rAI Press, AAAI Press.

[17] Muñoz Avila, H., andDana S. Nau, D. W. A., Weber, R., Breslow, L., and Yaman, F. (2001). SiN: Integrating Case-based Reasoning with Task Decomposition. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, {IJCAI} 2001*.

[18] Nau, D., Au, T. C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404.

[19] Ng, A. Y. and Russell, S. (2000). Algorithms for Inverse Reinforcement Learning. *in Proc. 17th International Conf. on Machine Learning*, pages 663–670.

[20] Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach, 3rd edition*.

[21] Sacerdoti, E. D. (1975). The Nonlinear Nature of Plans. *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1*, pages 206–214.

[22] Shani, G., Pineau, J., and Kaplow, R. (2013). A survey of point-based POMDP solvers. *Autonomous Agents and Multi-Agent Systems*, 27(1):1–51.

[23] Smallwood, R. D. and Sondik, E. J. (1973). The optimal control of partially observable markov processes over a finite horizon. *Operations Research*, 21(5):1071–1088.

[24] Sung, J., Selman, B., and Saxena, A. (2014). Synthesizing Manipulation Sequences for Under-Specified Tasks using Unrolled Markov Random Fields.

[25] Sutton, R. S. (1988). Learning to Predict by the methods of temporal differences. *Machine Learning*, 3:9–44.

[26] Tang, Y., Meneguzzi, F., Parsons, S., and Sycara, K. (2011). Planning over MDPs through Probabilistic HTNs. In *Proceedings of the AAAI-11 Workshop on Generalized Planning*, page to appear.

[27] Tate, A. (1977). Generating Project Networks. *Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, August 1977*, pages 888–893.

[28] Undurti, A. and How, J. P. (2011). A decentralized approach to multi-agent planning in the presence of constraints and uncertainty. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 2534–2539. IEEE.

[29] Undurti, Aditya; How, J. P. (2010). An online algorithm for constrained pomdps. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 3966–3973. IEEE.

[30] Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK.

[31] Watkins, C. J. C. H. and Dayan, P. (1992). Q-Learning. 8(3-4):279–292.

[32] Westrum, R. (2006). All coherence gone: New Orleans as a resilience failure. *E. Hollnagel & E. Rigaud (Eds.), Proceedings of the Second Resilience Engineering Symposium*, pages 333–341.

[33] Wilkins, D. E. (1988). *Practical planning - extending the classical {AI} planning paradigm*. Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann.

[34] Wotawa, F. and Weber, J. (2010). AI-Planning in a Mobile Autonomous Robot with Degraded software capabilities.

[35] Zieba, S., Polet, P., Vanderhaegen, F., and Debernard, S. (2010). Principles of adjustable autonomy: a framework for resilient human-machine cooperation. *Cognition, Technology {&} Work*, 12(3):193–203.

Appendix A

Source code of JSHOP2

A.1 Planing domain

```
;Cleaning example v2.9
;22/04/2015
;Benedek Szulyovszky

(defdomain cleaning (

;OPERATORS

  ;moving operator from performer location to the chosen location
  (:operator (!move ?perf ?perf_loc ?loc)
     ;preconditions: perf at its loc, the target loc is different to the
       ;current loc, the target location is accesable, read the current-cost
     ((at ?perf ?perf_loc) (different ?perf_loc ?loc)
       (TransConnect ?perf_loc ?loc) (cost ?current-cost))
     ;delete list: performer's loc, current cost
     ((at ?perf ?perf_loc) (cost ?current-cost))
     ;add list: new loc, increased new current cost
     ((at ?perf ?loc) (cost (call + ?current-cost 3)))
  )

  ;bringing the cleaner cart from its loc to a certain loc
  (:operator (!bringingcart ?perf ?cleaner_cart-loc ?loc)
     ;preconditions: cleaner cart loc, performer loc, they are at
       ;different locations, current cost
     ((at cleaner_cart ?cleaner_cart-loc) (at ?perf ?cleaner_cart-loc)
       (different ?cleaner_cart-loc ?loc)
```

```
27      (cost ?current-cost))
28      ;delete-list: cart loc, performer loc, current cost
29      ((at cleaner_cart ?cleaner_cart-loc) (at ?perf ?cleaner_cart-loc)
30        (cost ?current-cost))
31      ;add-list: new performer loc, new cart location, updated cost
32      ((at ?perf ?loc) (at cleaner_cart ?loc) (cost (call + ?current-cost 2)))
33  )
34
35  ;hoovering in the room
36  (:operator (!hoovering ?perf ?loc)
37      ((at ?perf ?loc) (at cleaner_cart ?loc))
38      ()
39      ((wacuumed ?loc))
40  )
41
42  ;mopping in the room
43  (:operator (!mop ?perf ?loc)
44      ((at ?perf ?loc) (at cleaner_cart ?loc) (wacuumed ?loc) (clean water))
45      ((clean water))
46      ((mopped ?loc) (dirty water))
47  )
48
49  ;changing the water independently from the room
50  (:operator (!changingwater ?perf)
51      ((dirty water))
52      ((dirty water))
53      ((clean water))
54  )
55
56  ;drying the floor
57  (:operator (!drying ?perf ?loc)
58      ((mopped ?loc) (at ?perf ?loc) (at cleaner_cart ?loc))
59      ()
60      ((dried ?loc))
61  )
62
63  ;dusting in the room
64  (:operator (!dusting ?perf ?loc)
65      ((at ?perf ?loc) (clean water) (at cleaner_cart ?loc))
66      ((clean water))
67      ((dusted ?loc) (dirty water))
68  )
```

```
69
70 ;METHODS
71
72 ;----------TOP-LEVEL METHODS
73
74 (:method (HouseCleaning ?perf ?loc)
75     ()
76     (:unordered    ;(Dusting ?loc ?perf ?perf_loc ?cleaner_cart-loc)
77           (Wacuuming ?loc ?perf ?perf_loc ?cleaner_cart-loc)
78           (Mopping ?loc ?perf ?perf_loc ?cleaner_cart-loc)
79 ))
80
81 ;----------SECOND-LEVEL METHODS
82
83
84 (:method (Wacuuming ?loc ?perf ?perf_loc ?cleaner_cart-loc)
85     ;the certain room is already wacuumed
86     ((wacuumed ?loc))
87     (())
88     ;preparation is necessary before the hoovering could be made
89     ((at ?perf ?perf_loc) (at cleaner_cart ?cleaner_cart-loc)
90       (need-to-make-sure ?loc ?cleaner_cart-loc))
91     ((MakeSure ?loc ?perf ?perf_loc ?cleaner_cart-loc)
92       (!hoovering ?perf ?loc))
93     ;every precondition is given for executing the hoovering
94     (((not(wacuumed ?loc)) (at cleaner_cart ?loc) (at ?perf ?loc)))
95     ((!hoovering ?perf ?loc))
96 )
97
98 (:method (Dusting ?loc ?perf ?perf_loc ?cleaner_cart-loc)
99     ;the certain room is already dusted
100    ((dusted ?loc))
101    (())
102    ;preparation is necessary before the dusting could be made
103    ((at ?perf ?perf_loc) (at cleaner_cart ?cleaner_cart-loc)
104      (need-to-make-sure ?loc ?cleaner_cart-loc))
105    ((MakeSure ?loc ?perf ?perf_loc ?cleaner_cart-loc) (!dusting ?perf ?loc))
106    ;every precondition is given for executing the dusting
107    (((not(dusted ?loc)) (at cleaner_cart ?loc) (at ?perf ?loc)))
108    ((!dusting ?perf ?loc))
109 )
110
```

```
111 (:method (Mopping ?loc ?perf ?perf_loc ?cleaner_cart-loc)
112     ;the certain room is already mopped
113     ((mopped ?loc))
114     (())
115     ;the room is not wacuumed yet which is the precondition of the mopping
116     ((not(wacuumed ?loc)))
117     ((Wacuuming ?loc ?perf ?perf_loc ?cleaner_cart-loc)
118       (Mopping ?loc ?perf ?perf_loc ?cleaner_cart-loc))
119     ;preparation is necessary before the mopping and drying could be made
120     ((at ?perf ?perf_loc) (at cleaner_cart ?cleaner_cart-loc)
121       (need-to-make-sure ?loc ?cleaner_cart-loc))
122     ((MakeSure ?loc ?perf ?perf_loc ?cleaner_cart-loc)
123       (!mop ?perf ?loc) (!drying ?perf ?loc))
124     ;every precondition is given for executing the mopping and drying
125     ((wacuumed ?loc) (at cleaner_cart ?loc) (at ?perf ?loc))
126     ((!mop ?perf ?loc) (!drying ?perf ?loc))
127 )
128
129 ;-----------THIRD-LEVEL METHODS
130
131 (:method (Moving ?perf ?perf_loc ?loc)
132     ;the room is directly accessable from the performer's loc,
133       ;so move operator is directly called
134     ((at ?perf ?perf_loc) (TransConnect ?perf_loc ?loc))
135     ((!move ?perf ?perf_loc ?loc))
136     ;the target location is not accessable, therefore an
137       ;intermediate location will be the first target location
138     ((at ?perf ?perf_loc) (not(TransConnect ?perf_loc ?loc))
139       (TransConnect ?perf_loc ?interm-loc))
140     ((TravelingThrough ?perf ?perf_loc ?interm-loc ?loc))
141 )
142
143 (:method (TravelingThrough ?perf ?perf_loc ?interm-loc ?loc)
144     ;there is an intermediate location which connects the current loc
145       ;to the target, the performer goes there first
146     ((at ?perf ?perf_loc) (TransConnect ?perf_loc ?interm-loc) )
147     ((!move ?perf ?perf_loc ?interm-loc) (Moving ?perf ?interm-loc ?loc))
148     ;there is need for further intermediate locations
149     ((at ?perf ?perf_loc) (not(TransConnect ?perf_loc ?interm-loc))
150       (TransConnect ?perf_loc ?interm2-loc))
151     ((TravelingThrough ?perf ?perf_loc ?interm2-loc))
152 )
```

```
153
154   (:method (CleanerCartBringing ?loc ?perf ?perf_loc ?cleaner_cart-loc)
155      ;the performer is at the same room as the cleaner cart is,
156        ;therefore 'bringingcart' operation is called
157      ((same ?perf_loc ?cleaner_cart-loc) (different ?cleaner_cart-loc ?loc))
158      ((!bringingcart ?perf ?cleaner_cart-loc ?loc))
159      ;the performer is in another room, therefore he has to go first
160        ;to the cleaning cart loc to bring it
161      ((not(at cleaner_cart ?loc)) (not(at ?perf ?cleaner_cart-loc)))
162      ((Moving ?perf ?perf_loc ?cleaner_cart-loc)
163        (!bringingcart ?perf ?cleaner_cart-loc ?loc))
164   )
165
166   (:method (MakeSure ?loc ?perf ?perf_loc ?cleaner_cart-loc)
167      ;the performer is not in the certain room which has to be cleaned
168      ((not(at ?perf ?loc)) (at cleaner_cart ?loc))
169      ((Moving ?perf ?perf_loc ?loc) (MakeSure ?loc ?perf ?loc ?loc))
170      ;the cleaning cart is not in the room which has to be cleaned
171      ((different ?cleaner_cart-loc ?loc))
172      ((CleanerCartBringing ?loc ?perf ?perf_loc ?cleaner_cart-loc)
173        (MakeSure ?loc ?perf ?loc ?loc))
174      ;for the 'wacuuming' method the condition allows to neglect
175        ;the property of the water
176      ((not(wacuumed ?loc)) (dirty water) (dusted ?loc))
177      (())
178      ;if the water is dirty it has to be changed
179      (dirty water)
180      ((!changingwater ?perf))
181      ;if every condition is fullfield by this method the planer goes back
182        ;to the higher level task
183      ((at cleaner_cart ?loc) (at ?perf ?loc) (clean water))
184      (())
185   )
186 ;STATE AXIOMS
187
188   ;definition of 'same'
189   (:- (same ?x ?x) nil)
190   ;definition of 'different'
191    (:- (different ?x ?y) ((not (same ?x ?y))))
192   ;interchangeability of the connect property of rooms
193   (:- (TransConnect ?x ?y) (or (connect ?y ?x) (connect ?x ?y)))
194   ;definition of directly connected rooms
```

```
195   (:- (TransConnect ?x ?z) ((connect ?x ?interm-loc)
196    (connect ?interm-loc ?z)))
197   ;definition of 'make sure' which describes the need of
198    ;preparation for the certain cleaning task
199   (:- (need-to-make-sure ?loc ?cleaner_cart-loc) (not(at ?perf ?loc))
200    (different ?cleaner_cart-loc ?loc) (dirty water))
201 )))
```

A.2 Planing problem definition

```
1 (defproblem problem cleaning
2   (
3      ;initial states defined by ground logical atoms
4      (at cleaner_cart kitchen)
5      (at robot bath)
6      (connect liv bath)
7      (connect liv kitchen)
8      (clean water)
9   )
10     ;planning problems
11     (:unordered (HouseCleaning robot bath)
12                 (HouseCleaning robot kitchen)
13                 (HouseCleaning robot liv))
14  )
```

Appendix B

Java source code of the planner

B.1 HTN module

```
1  package edu.cmu.HTN;
2
3  import java.lang.reflect.InvocationTargetException;
4  import java.util.ArrayList;
5  import java.util.Arrays;
6  import java.util.List;
7  import java.util.Stack;
8
9  public class Methods {
10
11   public static List<String> moppingkitchen_method(List<String>
      ↪ inputstatelist, String methodname) throws IllegalAccessException,
      ↪ IllegalArgumentException, InvocationTargetException {
12
13   if (Worldstate_HTN.highleveltasks.containsAll(Arrays.asList("mopped
      ↪ kitchen"))&&
14          !Worldstate_HTN.highleveltasks.containsAll(Arrays.asList("
      ↪ makesure liv")) &&
15          !Worldstate_HTN.highleveltasks.containsAll(Arrays.asList("
      ↪ makesure bath")) &&
16          !Worldstate_HTN.highleveltasks.containsAll(Arrays.asList("
      ↪ makesure kitchen")) &&
17          !Worldstate_HTN.highleveltasks.containsAll(Arrays.asList("
      ↪ changing water"))
18              ){
19
```

```
20      if (inputstatelist.containsAll(Arrays.asList("mopped kitchen"))){
21          return null;
22      }
23
24      Stack<List<String>> statestack = new Stack<List<String>>();
25      statestack.push(inputstatelist);
26      List<String> result = new ArrayList<String>();
27
28      //branch-1
29      if (inputstatelist.containsAll(Arrays.asList("at perf kitchen", "
    ↪ at cart kitchen", "clean water"))){
30          result = Operators_HTN.moppingkitchen(statestack.peek());
31          operatorname = "moppingkitchen";
32          if (result == null) {
33              return null;
34          }
35          else {
36              Searchingforsuccessors.searchOnThePossibleActions(statestack
    ↪ .peek(), result, methodname, operatorname);
37              statestack.push(result);
38              if(result.containsAll(Arrays.asList("mopped kitchen"))){
39                  Worldstate_HTN.highleveltasks.remove("mopped kitchen");}
40              return statestack.peek();}
41      }
42
43      //branch-2
44      else if (inputstatelist.containsAll(Arrays.asList("clean water")
45              )) {
46          //first operator
47          Worldstate_HTN.highleveltasks.add("makesure kitchen");
48          result = makesure_kitchen(statestack.peek(), methodname);
49          if (result == null) {
50              return null;
51          }
52          else {
53              statestack.push(result);
54              return statestack.peek();
55              }
56      }
57
58      //branch-3
```

```
59        else if (inputstatelist.containsAll(Arrays.asList("dirty water")))
      ↪  {
60            Worldstate_HTN.highleveltasks.add("changing water");
61            return null;
62        }
63    }
64        return null;
65    }
66
67
68    public static List<String> makesure_kitchen(List<String>
      ↪ inputstatelist, String methodname) throws IllegalAccessException,
      ↪ IllegalArgumentException, InvocationTargetException {
69
70        if (Worldstate_HTN.highleveltasks.containsAll(Arrays.asList("
      ↪ makesure kitchen")) &&
71            !Worldstate_HTN.highleveltasks.containsAll(Arrays.asList("
      ↪ makesure liv")) &&
72            !Worldstate_HTN.highleveltasks.containsAll(Arrays.asList("
      ↪ makesure bath")) &&
73            !Worldstate_HTN.highleveltasks.containsAll(Arrays.asList("
      ↪ changing water")) &&
74            !Worldstate_HTN.highleveltasks.containsAll(Arrays.asList("
      ↪ movingfrombathtokitchen")) &&
75            !Worldstate_HTN.highleveltasks.containsAll(Arrays.asList("
      ↪ movingfromkitchentobath")) &&
76            !Worldstate_HTN.highleveltasks.containsAll(Arrays.asList("
      ↪ bringingcartfromkitchentobath")) &&
77            !Worldstate_HTN.highleveltasks.containsAll(Arrays.asList("
      ↪ bringingcartfrombathtokitchen"))
78                ){
79
80        Stack<List<String>> statestack = new Stack<List<String>>();
81        statestack.push(inputstatelist);
82        List<String> result = new ArrayList<String>();
83
84        //branch-0
85        if (inputstatelist.containsAll(Arrays.asList("at perf kitchen", "
      ↪ at cart kitchen"))){
86            Worldstate_HTN.highleveltasks.remove("makesure kitchen");
87            return null;
88        }
```

```
89
90      //branch-1 (the performer is in another room, but the cart is at
     ↪ the right place)
91      if (inputstatelist.containsAll(Arrays.asList("at perf liv", "at
     ↪ cart kitchen"))) {
92          //first operator
93          result = Operators_HTN.movingfromlivtokitchen(statestack.peek()
     ↪ );
94          operatorname = "movingfromlivtokitchen";
95          if (result == null) {
96              return null;
97          }
98          else {
99              Searchingforsuccessors.searchOnThePossibleActions(statestack
     ↪ .peek(), result, methodname, operatorname);
100             statestack.push(result);
101             if(result.containsAll(Arrays.asList("at perf kitchen", "at
     ↪ cart kitchen"))){
102                 Worldstate_HTN.highleveltasks.remove("makesure kitchen")
     ↪ ;}
103             return statestack.peek();}
104     }
105
106
107     //branch-2 (the performer is in another room, but the cart is at
     ↪ the right place)
108     else if (inputstatelist.containsAll(Arrays.asList("at perf bath",
     ↪ "at cart kitchen"))) {
109         //first operator
110         Worldstate_HTN.highleveltasks.add("movingfrombathtokitchen");
111         return null;
112
113     }
114
115     //branch-3 (the performer and the cart are in the same but not
     ↪ right room)
116     else if (inputstatelist.containsAll(Arrays.asList("at cart liv", "
     ↪ at perf liv"))) {
117         //first operator
118         result = Operators_HTN.bringingcartfromlivtokitchen(statestack.
     ↪ peek());
119         operatorname = "bringingcartfromlivtokitchen";
```

```
120             if (result == null) {
121                 return null;
122             }
123             else {
124                 Searchingforsuccessors.searchOnThePossibleActions(statestack
    ↪ .peek(), result, methodname, operatorname);
125                 statestack.push(result);
126                 if(result.containsAll(Arrays.asList("at perf kitchen", "at
    ↪ cart kitchen"))){
127                     Worldstate_HTN.highleveltasks.remove("makesure kitchen")
    ↪ ;}
128                 return statestack.peek();}
129         }
130 
131         //branch-4  (the performer and the cart are in the same but not
    ↪ right room)
132         else if (inputstatelist.containsAll(Arrays.asList("at cart bath",
    ↪ "at perf bath"))) {
133             Worldstate_HTN.highleveltasks.add("
    ↪ bringingcartfrombathtokitchen");
134             return null;
135 
136         }
137 
138         //branch-5 (the performer and the cart are in different room)
139         else if (inputstatelist.containsAll(Arrays.asList("at cart bath",
    ↪ "at perf liv"))) {
140             //first operator
141             result = Operators_HTN.movingfromlivtobath(statestack.peek());
142             operatorname = "movingfromlivtobath";
143             if (result == null) {
144                 return null;
145             }
146             else {
147                 Searchingforsuccessors.searchOnThePossibleActions(statestack
    ↪ .peek(), result, methodname, operatorname);
148                 statestack.push(result);}
149 
150             //second operator
151             Worldstate_HTN.highleveltasks.add("
    ↪ bringingcartfrombathtokitchen");
152             return statestack.peek();
```

```
153     }
154 
155     //branch-6  (the performer and the cart are in different room)
156     else if (inputstatelist.containsAll(Arrays.asList("at cart liv", "
    ↪ at perf bath"))) {
157         //first operator
158         result = Operators_HTN.movingfrombathtoliv(statestack.peek());
159         operatorname = "movingfrombathtoliv";
160         if (result == null) {
161             return null;
162         }
163         else {
164             Searchingforsuccessors.searchOnThePossibleActions(statestack
    ↪ .peek(), result, methodname, operatorname);
165             statestack.push(result);}
166 
167         //second operator
168         result = Operators_HTN.bringingcartfromlivtokitchen(statestack.
    ↪ peek());
169         operatorname = "bringingcartfromlivtokitchen";
170         if (result == null) {
171             return null;
172         }
173         else {
174             Searchingforsuccessors.searchOnThePossibleActions(statestack
    ↪ .peek(), result, methodname, operatorname);
175             statestack.push(result);
176             if(result.containsAll(Arrays.asList("at perf kitchen", "at
    ↪ cart kitchen"))){
177                 Worldstate_HTN.highleveltasks.remove("makesure kitchen")
    ↪ ;}
178             return statestack.peek();}
179     }
180 
181     //branch-7 (performer in the good room)
182     else if (inputstatelist.containsAll(Arrays.asList("at cart bath",
    ↪ "at perf kitchen"))) {
183         Worldstate_HTN.highleveltasks.add("movingfromkitchentobath");
184         return null;
185     }
186 
187     //branch-8 (performer in the good room)
```

```
188        else if (inputstatelist.containsAll(Arrays.asList("at cart liv", "
     ↪ at perf kitchen"))) {
189            //first operator
190            result = Operators_HTN.movingfromkitchentoliv(statestack.peek()
     ↪ );
191            operatorname = "movingfromkitchentoliv";
192            if (result == null) {
193                return null;
194            }
195            else {
196                Searchingforsuccessors.searchOnThePossibleActions(statestack
     ↪ .peek(), result, methodname, operatorname);
197                statestack.push(result);}
198
199            //second operator
200            result = Operators_HTN.bringingcartfromlivtobath(statestack.
     ↪ peek());
201            operatorname = "bringingcartfromlivtobath";
202            if (result == null) {
203                return null;
204            }
205            else {
206                Searchingforsuccessors.searchOnThePossibleActions(statestack
     ↪ .peek(), result, methodname, operatorname);
207                statestack.push(result);
208                if(result.containsAll(Arrays.asList("at perf kitchen", "at
     ↪ cart kitchen"))){
209                    Worldstate_HTN.highleveltasks.remove("makesure kitchen")
     ↪ ;}
210                return statestack.peek();}
211        }
212
213        }
214
215        return null;
216    }
217}
```

```
1 package edu.cmu.HTN;
2 import java.util.ArrayList;
3 import java.util.Arrays;
4 import java.util.HashMap;
```

```
import java.util.List;
import java.util.Map;
import java.util.Random;

public class Operators_HTN {

  private static List<String> preconditions = new ArrayList<String>();
  private static List<String> postconditions_add = new ArrayList<String
  ↪ >();
  private static List<String> postconditions_del = new ArrayList<String
  ↪ >();

  //moving from living room to bathroom
  public static List<String> movingfromlivtobath(List<String>
  ↪ inputstate) {

      String operatorname = "movingfromlivtobath";
      preconditions = Arrays.asList("at perf liv");
      postconditions_add = Arrays.asList("at perf bath");
      postconditions_del = Arrays.asList("at perf liv");
      return DeterministicUpdate(preconditions, inputstate,
  ↪ postconditions_add, postconditions_del, operatorname);
  }

  //vacuuming in bathroom
  public static List<String> vacuumingbath(List<String> inputstate) {

      String operatorname = "vacuumingbath";
      preconditions = Arrays.asList("at perf bath", "at cart bath", "not
  ↪  vacuumed bath");
      postconditions_add = Arrays.asList("vacuumed bath");
      postconditions_del = Arrays.asList("not vacuumed bath");
      return DeterministicUpdate(preconditions, inputstate,
  ↪ postconditions_add, postconditions_del, operatorname);
  }

  //water change
  public static List<String> changewater(List<String> inputstate) {

      String operatorname = "changewater";
      preconditions = Arrays.asList("dirty water");
      postconditions_add = Arrays.asList("clean water");
```

```
41      postconditions_del = Arrays.asList("dirty water");
42      return DeterministicUpdate(preconditions, inputstate,
   ↪ postconditions_add, postconditions_del, operatorname);
43   }
44
45   //mopping the bathroom
46   public static List<String> moppingbath(List<String> inputstate) {
47
48      String operatorname = "moppingbath";
49      preconditions = Arrays.asList("at perf bath", "at cart bath", "
   ↪ vacuumed bath", "not mopped bath", "clean water");
50      postconditions_add = Arrays.asList("mopped bath", "dirty water");
51      postconditions_del = Arrays.asList("not mopped bath", "clean water
   ↪ ");
52      return DeterministicUpdate(preconditions, inputstate,
   ↪ postconditions_add, postconditions_del, operatorname);
53   }
54
55   private static List<String> DeterministicUpdate(List<String>
   ↪ preconditions, List<String> inputstate,List<String>
   ↪ postconditions_add, List<String> postconditions_del,
56         String operatorname) {
57
58      if (inputstate.containsAll(preconditions)) {
59         Worldstate_HTN.stateUpdate(inputstate, postconditions_add,
   ↪ postconditions_del, operatorname);
60         return Worldstate_HTN.getState_actual();
61      }
62      else {
63         return null;
64         }
65   }
66
67 }
```

```
1 package edu.cmu.HTN;
2 import java.lang.reflect.InvocationTargetException;
3 import java.lang.reflect.Method;
4 import java.util.ArrayList;
5 import java.util.Arrays;
6 import java.util.Collections;
7 import java.util.HashMap;
```

```
8 import java.util.LinkedHashSet;
9 import java.util.List;
10 import java.util.Map;
11
12
13 public class Searchingforsuccessors {
14
15   //flags and indexes
16   public static int planindex = 0;
17   public static int successfulplans = 0;
18
19   //map from worldstate to executable actions
20   public static Map<List<String>, List<String>> mapfromstatestoactions
     ↪ = new HashMap<List<String>, List<String>>();
21   public static Map<Integer, List<String>> nodetable = new HashMap<
     ↪ Integer, List<String>>();
22   public static Map<Integer, List<Integer>> edgetable = new HashMap<
     ↪ Integer, List<Integer>>();
23   public static int transindex = 0;
24
25   private static Methods t = new Methods();
26   private static Class cls = t.getClass();
27   private static Method[] m = cls.getMethods();
28
29   public static ArrayList<List<String>> methodcall(List<String>
     ↪ inputstatelist) throws IllegalAccessException,
     ↪ IllegalArgumentException, InvocationTargetException{
30
31       ArrayList<List<String>> successors = new ArrayList<List<String>>()
     ↪ ;
32
33        //iteration through existing methods
34        call: for(int i = 0; i < Methods.numberofmethods; i++) {
35          Worldstate_HTN.highleveltasks = new ArrayList<String>(
     ↪ Worldstate_HTN.highleveltasks_last);
36          String methodname = m[i].toString().replace("public static java
     ↪ .util.List edu.cmu.probplan.Methods.", "")
37                  .replace("(java.util.List,java.lang.String) throws java.
     ↪ lang.IllegalAccessException,java.lang.IllegalArgumentException,
     ↪ java.lang.reflect.InvocationTargetException", "");
38          Object result = m[i].invoke(t, inputstatelist, methodname);
39          List<String> resultstate = (List<String>) result;
```

```
40
41          if (resultstate != null && resultstate.containsAll(
   ↪ Worldstate_HTN.getState_goal())) {}
42          else if (resultstate == null && !Worldstate_HTN.getTask().
   ↪ equals(Worldstate_HTN.highleveltasks_last)){
43              Collections.sort(inputstatelist);
44              successors.add(inputstatelist);
45              List<List<String>> tasks = new ArrayList<List<String>> (
   ↪ Worldstate_HTN.taskmap.get(inputstatelist));
46              tasks.remove(Worldstate_HTN.highleveltasks_last);
47              tasks.add(Worldstate_HTN.getTask());
48              for (List<String> goal: tasks){
49                  Collections.sort(goal);
50              }
51              tasks = new ArrayList<List<String>>(new LinkedHashSet<List<
   ↪ String>>(tasks));
52              Worldstate_HTN.taskmap.put(new ArrayList<String>(
   ↪ inputstatelist), tasks);
53          }
54          else if (resultstate != null) {
55              Collections.sort(resultstate);
56              if (!(Worldstate_HTN.taskmap.get(resultstate) == null)){
57                  for (List<String> goal : Worldstate_HTN.taskmap.get(
   ↪ resultstate)){
58                      if (goal.equals(Worldstate_HTN.getTask())){
59                          continue call;
60                      }
61                  }
62                  List<List<String>> tasks = new ArrayList<List<String>> (
   ↪ Worldstate_HTN.taskmap.get(resultstate));
63                  tasks.add(Worldstate_HTN.getTask());
64                  Worldstate_HTN.taskmap.put(new ArrayList<String>(
   ↪ resultstate), tasks);
65              }
66              else{
67                  List<List<String>> tasks = new ArrayList<List<String>>();
68                  tasks.add(Worldstate_HTN.getTask());
69                  Worldstate_HTN.taskmap.put(new ArrayList<String>(
   ↪ resultstate), tasks);
70              }
71              successors.add(resultstate);
72          }
```

```
73        }
74
75        return successors;
76    }
77
78    public static void searchOnThePossibleActions(List<String>
    ↪ inputstatelist, List<String> newstates, String methodname, String
    ↪ operatorname) throws IllegalAccessException,
    ↪ IllegalArgumentException, InvocationTargetException {
79
80
81        ArrayList<List<String>> leafs_states = new ArrayList<List<String
    ↪ >>();
82        List<Integer> nodes = new ArrayList<Integer>();
83
84            //check if the resulting states contains the goal states
85            if (newstates.containsAll(Worldstate_HTN.getState_goal())){
86                successfulplans++;
87                leafs_states.add(newstates);
88            }
89            else{
90                leafs_states.add(newstates);
91            }
92
93            //saving for tree
94             for (List<String> node : leafs_states){
95                if (Searchingforsuccessors.edgetable.get(savenewnode(
    ↪ inputstatelist)) != null)    {
96                    nodes = Searchingforsuccessors.edgetable.get(savenewnode(
    ↪ inputstatelist));
97                }
98                nodes.add(savenewnode(node));
99             }
100            nodes = new ArrayList<Integer>(new LinkedHashSet<Integer>(
    ↪ nodes));
101           edgetable.put(savenewnode(inputstatelist), nodes);
102
103           Collections.sort(inputstatelist);
104           List<String> possibleactions = new ArrayList<String>();
105           for (Map.Entry<List<String>, List<String>> entry :
    ↪ mapfromstatestoactions.entrySet()) {
106               if (entry.getKey().equals(inputstatelist)){
```

```
107                 possibleactions = entry.getValue();
108             }
109         }
110         possibleactions.add(operatorname);
111         possibleactions = new ArrayList<String>(new LinkedHashSet<
    ↪ String>(possibleactions));
112         mapfromstatestoactions.put(inputstatelist, possibleactions);
113
114 }
115
116
117 private static int savenewnode(List<String> inputstatelist) {
118
119     for (Map.Entry<Integer, List<String>> entry : nodetable.entrySet()
    ↪ ) {
120         if (equalLists(entry.getValue(),inputstatelist))
121             return entry.getKey();
122     }
123     nodetable.put(planindex, inputstatelist);
124     planindex++;
125     return planindex-1;
126 }
127
128 public static boolean equalLists(List<String> one, List<String> two){
129     if (one == null && two == null){
130         return true;
131     }
132
133     if((one == null && two != null)
134         || one != null && two == null
135         || one.size() != two.size()){
136         return false;
137     }
138     Collections.sort(one);
139     Collections.sort(two);
140     return one.equals(two);
141 }
142
143 }
```

B.2 MDP module

```
package edu.cmu.probplan;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Qlearning {

  public static Map<List<String>, Integer> inversenodetable = new
  ↪ HashMap<List<String>, Integer>();
  public static double[][] Q = new double[Main.nodemap.size()][
  ↪ Operators.numberofoperators+1];
  public static double[][] Q_c = new double[Main.nodemap.size()][
  ↪ Operators.numberofoperators+1];
  private static final DecimalFormat df = new DecimalFormat("#.####");

   private static final double alpha = 0.1;
   private static final double gamma = 0.9;

  public static void learn(int iterationindex) throws
  ↪ IllegalAccessException, IllegalArgumentException,
  ↪ InvocationTargetException, InterruptedException {

     Main.qlearning_start = true;
     newNodetable();
     Operators.probabilitybleInit();

     Operators t = new Operators();
     Class<? extends Operators> cls = t.getClass();
     Class[] argTypes = new Class[] { List.class };
     Method m;

     for (int j = 1; j<=iterationindex; j++){
        if((double) 100*j/iterationindex % 10 == 0){
```

```
35              System.out.println("Process at: " + (int) 100*j/
    ↪ iterationindex + "%");
36          }
37          for (Map.Entry<List<String>, List<String>> entry : Main.
    ↪ mapfromstatestoactions.entrySet()) {
38
39              for (String operatorname : entry.getValue()){
40                  try {
41                      m = cls.getMethod(operatorname, argTypes);
42                      m.invoke(t, entry.getKey());
43                  } catch (NoSuchMethodException e) {
44                      System.out.println("There is no such an operator: " +
    ↪ operatorname);
45                  }
46              }
47          }
48      }
49
50  }
51
52  public static void tableupdate(List<String> state_actual, String
    ↪ operatorname, List<String> newstates) {
53
54      Collections.sort(newstates);
55      Collections.sort(state_actual);
56        double p = probability(state_actual);
57        double p_action = Operators.probabilitytable.get(operatorname);
58
59        //Q_R value parameters
60        double q = Q[inversenodetable.get(state_actual)][Operators.
    ↪ operatortable.get(operatorname)];
61        double maxQ = maxQ(inversenodetable.get(newstates));
62        int r = Reward.rewardtable.get(newstates);
63        double penalty = Main.penaltycoef * Operators.penaltytable.get(
    ↪ operatorname);
64      //Q_R value update
65        double value = q + alpha * (r - penalty + gamma * p_action * p *
    ↪  maxQ - q); //(1-alpha)*
66        setQ(inversenodetable.get(state_actual), Operators.operatortable
    ↪ .get(operatorname), value);
67
68        //Q_C value parameters
```

```
69          double q_c = Q_c[inversenodetable.get(state_actual)][Operators.
    ↪ operatortable.get(operatorname)];
70          List<String> constraininput = new ArrayList<String>(newstates);
71          constraininput.add(operatorname);
72          double c = Constrain.constraintable.get(constraininput);
73          double maxQ_c = maxQ_c(inversenodetable.get(newstates));
74          //Q_C value update
75          double constrainvalue = (1-alpha)*q_c + alpha * (c + gamma *
    ↪ p_action * p * maxQ_c - q_c);
76          setQ_c(inversenodetable.get(state_actual), Operators.
    ↪ operatortable.get(operatorname), constrainvalue);
77      }
78
79      private static double probability(List<String> state_actual) {
80          double numberofchildren = 1;
81          numberofchildren = Main.edgemap.get(inversenodetable.get(
    ↪ state_actual)).size();
82          return 1/numberofchildren;
83      }
84
85      public static double maxQ(int s) {
86          double maxValue = Double.MIN_VALUE;
87          for (int i = 0; i < Q[s].length; i++) {
88              double value = Q[s][i];
89
90              if (value > maxValue)
91                  maxValue = value;
92          }
93          return maxValue;
94      }
95
96      public static double maxQ_c(int s) {
97          double maxValue = Double.MIN_VALUE;
98          for (int i = 0; i < Q_c[s].length; i++) {
99              double value = Q_c[s][i];
100
101             if (value > maxValue)
102                 maxValue = value;
103         }
104         return maxValue;
105     }
106
```

```
107 public static void setQ(int state, int action, double value) {
108         Q[state][action] = value;
109   }
110
111 public static void setQ_c(int state, int action, double value) {
112         Q_c[state][action] = value;
113   }
114
115 private static void newNodetable() {
116     for (Map.Entry<Integer, List<String>> entry : Main.nodemap.
    ↪ entrySet()) {
117         inversenodetable.put(entry.getValue(), entry.getKey());
118     }
119 }
120
121 public static int[] toIntArray(List<Integer> list){
122         int[] ret = new int[list.size()];
123         for(int i = 0;i < ret.length;i++)
124           ret[i] = list.get(i);
125         return ret;
126     }
127
128
129 // ----------------------policy evaluation
    ↪ -------------------------------------------------
130 public static Integer policy(int state) {
131         double maxValue = Double.MIN_VALUE;
132         Integer policyGotoState = 0; // do done! action, which will give
    ↪ error
133         for (int i = 0; i < Q[state].length; i++) {
134          if (Q_c[state][i] < Main.lambda){
135              if (Q[state][i] > maxValue) {
136                      maxValue = Q[state][i];
137                      policyGotoState = i;
138              }
139          }
140         }
141         return policyGotoState;
142   }
143
144 public static void Qconstrain() {
145         System.out.println("\nQ-constrain results:");
```

```
146         for (int i = 0; i < Q_c.length; i++) {
147             System.out.print("out from " + Main.nodemap.get(i) + ":  ");
148             for (int j = 0; j < Q_c[i].length; j++) {
149                 System.out.print(df.format(Q_c[i][j]) + " ");
150             }
151             System.out.println();
152         }
153     }
154 
155     // policy is maxQ(states)
156     public static void showPolicy() {
157         System.out.println("\nPolicy:");
158         for (int i = 0; i < inversenodetable.size(); i++) {
159             List<String> from = Main.nodemap.get(i);
160             for (Map.Entry<String, Integer> entry : Operators.
    ↪ operatortable.entrySet()) {
161                 if (entry.getValue() == policy(i)) {
162                     System.out.println("FROM: "+ from + " TO: "+ entry.getKey
    ↪ ());
163                     break;}
164             }
165 
166         }
167     }
168 
169 }
```

B.3 Simulator module

```
1  package edu.cmu.simulator;
2  
3  import java.io.FileInputStream;
4  import java.io.FileOutputStream;
5  import java.io.IOException;
6  import java.io.ObjectInputStream;
7  import java.io.ObjectOutputStream;
8  import java.lang.reflect.InvocationTargetException;
9  import java.lang.reflect.Method;
10 import java.text.DecimalFormat;
11 import java.util.Arrays;
12 import java.util.Collections;
```

```
13import java.util.HashMap;
14import java.util.List;
15import java.util.Map;
16import java.util.Random;
17import java.util.Map.Entry;
18import java.util.Stack;
19
20import edu.cmu.probplan.Qlearning;
21import edu.cmu.HTN.Searchingforsuccessors;
22
23
24public class Main_simulator {
25
26    public static double[][] Q ;
27    public static double[][] Q_c;
28    public static double lambda = 0.00;
29    public static Map<Integer, List<String>> nodemap = new HashMap<
      ↪ Integer, List<String>>();
30    public static Map<List<String>, Integer> inversenodemap = new HashMap
      ↪ <List<String>, Integer>();
31
32    //simulator interface settings
33    static boolean showtheplan = false;
34    static int iteration = 500;
35    static int resolution = 50;
36    static double increase = 0.02;
37
38    //saving for evaluation
39    public static double statistic_success = 0;
40    public static double statistic_length = 0;
41
42    public static void main (String[] arg) throws Exception {
43
44        readData();
45        evaluate();
46
47    }
48
49    public static void evaluate() throws IllegalAccessException,
      ↪ IllegalArgumentException, InvocationTargetException{
50        for (int k=0; k<resolution; k++){
51            lambda += increase;
```

```
52          statistic_success = 0;
53          statistic_length = 0;
54          for(int i=0; i<iteration; i++){
55              if(simulation()){
56                  statistic_success++;
57              }
58          }
59          statistic_success/=iteration;
60          statistic_length/=iteration;
61          System.out.println("\nAvarage length of the plans after " +
    ↪ iteration+ " iteration: " + statistic_length);
62          System.out.println("Probability of success after " +iteration+
    ↪ " iteration: " + statistic_success + "lambda: " + lambda);
63      }
64  }
65
66  public static boolean simulation() throws IllegalAccessException,
    ↪ IllegalArgumentException, InvocationTargetException {
67
68      Operators_simulator.operatortableInit();
69      Operators_simulator.probabilitybleInit();
70      Operators_simulator t = new Operators_simulator();
71      Class<? extends Operators_simulator> cls = t.getClass();
72      Class[] argTypes = new Class[] { List.class };
73
74      Stack<List<String>> plan = new Stack<List<String>>();
75      List<String> inputstate = StateTrans_simulator.getState_initial();
76      Collections.sort(inputstate);
77      plan.push(inputstate);
78
79      while (plan.peek().containsAll(StateTrans_simulator.getState_goal
    ↪ ()) == false){
80          if (inputstate.containsAll(Arrays.asList("at cart corridor"))){
81              Random randomGenerator = new Random();
82              if (randomGenerator.nextDouble() <= 0.8){
83                  return false;
84              }
85          }
86          Collections.sort(inputstate);
87
88          int operatornumber = decisionBasedOnPolicy(inversenodemap.get(
    ↪ inputstate));
```

```
89          String operatorname = null;
90          for (Map.Entry<String, Integer> entry : Operators_simulator.
    ↪ operatortable.entrySet()) {
91              if (entry.getValue() == operatornumber){
92                  operatorname = entry.getKey();}
93              }
94
95           Method m;
96          try {
97              m = cls.getMethod(operatorname, argTypes);
98          } catch (NoSuchMethodException e) {
99              return false;
100         }
101         plan.push((List<String>) m.invoke(t, inputstate));
102          inputstate = plan.peek();
103     }
104     statistic_length+= plan.size();
105
106     if (showtheplan == true){
107         System.out.println("\nPlan based on the policy: ");
108         for (int k = 0; k<plan.size();k++){
109             System.out.println(plan.get(k));
110         }
111     }
112     return true;
113
114 }
115
116 public static void readData() throws ClassNotFoundException,
    ↪ IOException{
117
118     FileInputStream Q_reward = new FileInputStream("/home/benny/
    ↪ workspace/ProbPlan non-det separated v6/Q_reward.dat");
119     ObjectInputStream ois = new ObjectInputStream(Q_reward);
120     Q = (double[][]) ois.readObject();
121
122     FileInputStream Q_constrain = new FileInputStream("/home/benny/
    ↪ workspace/ProbPlan non-det separated v6/Q_constrain.dat");
123     ObjectInputStream ois2 = new ObjectInputStream(Q_constrain);
124     Q_c = (double[][]) ois2.readObject();
125
```

```
126        FileInputStream nodemapfile = new FileInputStream("/home/benny/
    ↪ workspace/ProbPlan non-det separated v6/nodemap.dat");
127        ObjectInputStream ois3 = new ObjectInputStream(nodemapfile);
128        nodemap = (Map<Integer, List<String>>) ois3.readObject();
129
130        FileInputStream inversenodemapfile = new FileInputStream("/home/
    ↪ benny/workspace/ProbPlan non-det separated v6/inversenodemap.dat")
    ↪ ;
131        ObjectInputStream ois4 = new ObjectInputStream(inversenodemapfile)
    ↪ ;
132        inversenodemap = (Map<List<String>, Integer>) ois4.readObject();
133    }
134
135    public static int decisionBasedOnPolicy(Integer state) {
136          double maxValue = Double.MIN_VALUE;
137          Integer policyGotoState = 0; // default goto self if not found
138          for (int i = 0; i < Q[state].length; i++) {
139              double value = Q[state][i];
140
141              if (value > maxValue && Q_c[state][i]<lambda) {
142                  maxValue = value;
143                  policyGotoState = i;
144              }
145          }
146          return policyGotoState;
147     }
148
149
150 }
```