Hierarchical Task Recognition and Planning in Smart Homes with Partial Observability

by

Dan Wang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2017

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Older adults with cognitive impairment have significantly burdened their families and the society due to costly caring and waste of labors. Developing intelligent assistant agents (IAAs) in smart homes that can help those people accomplishing activities of daily living (ADLs) independently has attracted tremendous attention, from both academia and industry. Ideally, IAAs should recognize older adults' goals and reason about further steps needed for the goals.

This paper proposed a goal recognition and planning algorithm to support an IAA in smart home. The algorithm addresses several important issues. First it can deal with *partial observability* by Bayesian inference for step recognition. Even advanced sensors are not guaranteed to be 100% reliable. Besides, due to limited accessibility or privacy, not all attributes of physical objects can be measured by sensors. The proposed algorithm can reason about ongoing goals with some sensors missing or unreliable. Second, the algorithm reasons about concurrent goals. For everyday life, a person is typically involved in multi-tasks by switching back and forth. Based on the context, the proposed algorithm can assign a step to the correct goal and keep tracks of the goal's ongoing status. The context involves status of ongoing goals inferred from a recognition procedure, and desired next steps and tasks, which are obtained through a planning procedure. Last but not least, the algorithm can handle incorrectly executed steps. For older adults with cognitive impairment, executing unrelated or wrong steps towards certain goals is common in their daily life. A module is designed to hand wrong steps by detecting and then prompt the person with correct steps.

The algorithm is based on Hierarchical Task Network (HTN), of which the knowledge base is composed of methods (for tasks) and operators (for steps). Such hierarchical modeling of tasks and steps enables the algorithm to deal with partially ordered subtasks and alternative plans. Furthermore, the preconditions of methods and operators enable to generate feasible hints of next steps and tasks by considering uncertainties in belief states.

In the experiment, a simulator is designed to simulate the virtual sensors and a virtual human executing a sequence of steps predefined in a test case. The algorithm is tested on many simulated easy or difficult cases. For example single goal and correct steps are easy test cases. Having multiple goals with wrong steps makes the problem more difficult. Also cases of sensors missing are experimented. The results shows that the algorithm works very well on simple cases, achieving nearly 100% accuracy. Even for the hardest cases, the performance is acceptable when sensor reliabilities are above 0.95. Test cases with missing sensors also provide meaningful guideline for setting up sensors for an intelligent assistant agent.

Acknowledgements

First of all, I'm very grateful to my supervisor Prof. Jesse hoey. Jesse helped me a lot in my study and thesis. Whenever I got lost in my project, Jesse is always there to hep and give insightful and concrete advice. He helped and supported students in all possible ways. I enjoyed very much working with Jesse for the last two years. Time flies and I will never forget how much time and care he invested on my research.

I also would like to thank Prof. Kate Larson and Prof. Ian McKillop for reading my thesis. Your effort helps to improve this thesis.

When I came to Waterloo, I got to know many lab mates, including Areej Alhothali, Josh Jung, Shehroz Khan, Deepak Rishi, Zhengkun Shang, Haiyu Zhen, Aarti Malhotra, and Aron Li. I learned a lot from you and of course we had a lot of fun!

Lastly, I want to thank my family for their continuous support over the years.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Nowadays, more and more older adults suffer from cognitive impairments, which cause great difficulties in doing activities of daily living (ADLs) [19]. For example, Alzheimer's disease degenerates people' capability to remember things and think clearly. As a consequence, the older adults have more difficulty in taking care of themselves in ADLs, such as cooking, laundry, etc.

Cognitive impairments impact significantly the older adult, his/her family as well as the society. In general, people suffer from psychological stresses when they are confused with simple things which was not the case before. Some typical symptoms include anxiety, rage, and even violence, which accelerate cognitive deterioration. For the families, the life quality is reduced remarkably due to costly treatment of Alzheimer's disease. According to the Alzheimer's association report in 2016 [3], families with Alzheimer's disease members have substantial financial burden, which forces people to "take money out of their retirement savings, cut back on buying food, and reduce their own trips to the doctor" [1]. For the society, the care of older adults with Alzheimer's and other dementia diseases leads to occupation of labors. As reported by the Alzheimer's association [3], "in 2015, more than 15 million family members and other unpaid caregivers provided an estimated 18.1 billion hours of care to people with Alzheimers and other dementias, a contribution valued at more than $221 billion" [1]. This report also predicts that by 2050, there would be about 1 million new cases of Alzheimer's per year.

[1] Quote from page 459, Alzheimer's Association. "2016 Alzheimer's disease facts and figures." Alzheimer's & Dementia 12.4 (2016): 459-509.

For the above mentioned reasons, developing intelligent assistance agents (IAAs) to help the older adults with cognitive impairments on ADLs becomes urgent. Recently, both academia and industry have spent tremendous efforts in this direction. An IAA is expected to help the older adults to accomplish ADLs without caregivers, so as to free caregivers from repeated and time-consuming caring. Besides, the older adults also benefit from the interactive process by gaining independence and self-confidence.

In a smart home, IAAs play the role of intelligent real-time reminders. It means that the agent prompts the older adult whenever he/she is confused in ADLs. To achieve this, IAAs should at least be capable of: gathering signals from sensors, being aware of situations [16], recognizing ongoing goals, presenting effective assistances and alerting caregivers [36]. Ideally, the prompts should take into account older adults' current awareness and emotional states, so older adults move forward comfortably and smoothly.

Due to limitations of sensors and privacy concerns, not all attributes of objects in the environment can be monitored by sensors. For example, it is not feasible to attach a sensor to a person's hands for detecting if his/her hands have soap. Furthermore, even though some sensors are available, they are not 100% reliable. Thus IAAs should also cope with partial observability due to missing or unreliable sensors.

Older adults with cognitive impairments commonly execute ADLs with irrational, repeated and disordered steps. IAAs are required to identify these improper behaviors. Geib et al. [24] discussed several critical considerations of goal recognition for the older adults. These include abandoning plans, hostile agents, observations of failed actions, partially ordered plans, multiple concurrent goals, actions used for multiple effects, and failure to observe.

Although an assistance agent aims to help older adults with cognitive impairments as much as possible, aggressive prompts have the opposite consequences. According to Hoey et al. [33], smart home assistance should be as passive as possible, so as to maintain the independent feeling of older adults. Researchers in social psychology also argue that task instructions without excessive emphasizing on the memory component of a task can improve the older adults performance on the task [19]. Hence it is undesirable to always present the most detailed instructions for an older adult with Alzheimer's or other dementias.

An IAA should be customizable and generalizable. An IAA helps older adults achieve some goals. The recipe of a daily task varies from one person to another, for two reasons. Firstly, each person has his/her own preference and habits for ADLs. For example, to make a cup of tea, some people are used to fill the cup with hot water first, while others are used to dropping a tea bag first. Instructions consistent with habits make people feel

comfortable. Secondly, each home has its own appliances, which lead to different ways to achieve ADLs. Ideally, IAAs should customize its assistance according to personal specifications and preferences.

Based on all the above discussions, an IAA to help older adults with Alzheimer's disease or other dementias on their ADLs should address the following aspects.

- Tolerate partial observability caused by missing and unreliable sensors.
- Recognize concurrent goals.
- Detect improper steps and rectify the older adult from mistakes.
- Present hints or prompts of various detail levels.
- Customize for different personal specifications and preferences.

1.2 Objectives

This work proposes an algorithm for IAAs to address the above-mentioned issues. It should have the following functionalities.

- **Tolerance to partial observability**. The input of the algorithm is sensor measurements, from which the algorithm infers steps happening. Inaccurate sensor measurements lead the algorithm to produce incorrect step recognition. This will further affect goal recognition, leading to wrong hints for the next tasks and steps.
- **Step recognition**. Step recognition precedes goal recognition. Given sensor measurements, the algorithm should give the probabilities of steps occurring. Considering the large number of possible steps, to reduce complexity, the algorithm extracts only steps related to the context.
- **Goal recognition**. As a core part of an IAA, the algorithm definitely should be able to recognize the person's purpose or intent based on observations. Besides, the algorithm should know how much progress has been made towards the goals. Furthermore, the algorithm should also work when the person is multitasking. To be precise, the algorithm matches observations to goals, monitors the ongoing process of each goal, and derives the probability of a goal in the progress.

- **Planning**. A planning process generates a sequence of steps for achieving a given goal. It is needed in this work because whenever the older adult get confused, the agent needs to reason about the correct next tasks and steps. The steps should be feasible w.r.t. constraints. Given the status of an ongoing goal, the planning process should prompt in different detail levels.

- **Exception handling**. As mentioned before, mistakes are common for older adults with Alzheimer's disease and other dementias when accomplishing ADLs. The algorithm should identify mistakes made by the older adults and rectify them by giving instructions.

- **Customizable activities**. The ADLs that need help varies from person to person. For example, some older adults have problems with making coffee, while some can't take a shower independently. Furthermore, even for the same activity, different person has different preferences. The algorithm in this work aims to be activity independent. The caregivers can simply set up the agent based on activities that their older adults need help with and also the older adults' preferences.

This work proposes the **HTN-GRP-PO** algorithm, which stands for "hierarchical task network based goal recognition and planning algorithm with partial observability". The algorithm adopts the hierarchical paradigm in *Hierarchical Task Network* (HTN) planning [20], which generates feasible plans for predefined tasks by recursively decomposing composite tasks using a knowledge base. The modeling of ADLs with a HTN framework has two advantages. First, the hierarchical nature of HTNs enables the algorithm to provide prompts of different detail levels. Second, the knowledge base allows to set up specific goals and preferences for the older adults.

Although the algorithm aims at recognizing ongoing goals, in the step level, probabilities of steps occurring are inferred through a Bayesian network. To reduce computational complexity, only steps related to the context are considered. The proposed hierarchical goal recognition algorithm can handle concurrent goals. It tracks the status of all the ongoing goals, updates the beliefs on goals based on new observations, monitors steps just happened, and reports wrong steps. This framework works with given hierarchical task networks which is predefined in a knowledge base. As such, the algorithm shares the strengths of HTNs, including the ability to deal with ordered / unordered subtasks, alternative ways to achieve a goal, and preconditions of tasks and steps.

The planning part of the algorithm is designed to obtain the desired next steps and tasks in order to achieve the recognized goals. It is based on HTN planning. The generated

hierarchical hints will be presented when necessary. Basically, the planning process decomposes tasks which are ready to be implemented by applying methods in the knowledge base. A method is applied to a task only when all its preconditions are satisfied. Thus hints for the desired next steps and tasks must be consistent with the context.

Exception handling in the algorithm is motivated by re-planning in HTN, which recreates feasible plans when the execution of plans goes wrong. In the case of wrong steps, the algorithm figures out the associated hierarchical task network and repairs the network. The output of exception handling are hints for the next steps and tasks to execute, so as to rectify wrong steps.

A simulator which sets up virtual sensors and simulates the occurring of steps predefined in a test case is designed for the experiment. No real older adults are involved in the experiment. After each simulated step, the algorithm reasons about the ongoing goals and the correct next tasks and steps for the ongoing goals. The outputs of each step are compared with the ongoing goals and the next tasks and steps of the corresponding test case, so as to compute output accuracy. According to experiments, the proposed algorithm's performance depends on how challenging is the test case. Basically, issues like unreliable or missing sensors, concurrent goals, and wrong steps make the problem more difficult. The experiment results indicate that when only one kind of aforementioned issues presents, the algorithm performs very well, with accuracy above 95%. When two kinds of issues present, the accuracy reduces to 85%-95%. When all of the issues present, the performance becomes unacceptable, with accuracy under 80%.

Despites the performances on some hard case being less than satisfying, meaningful suggestions are given. The results shows that the priors of goals have impacts on the performance. Thus, priors of goals should be carefully set up. Besides, the experiments also indicate which sensors are important for a specific scenario. Those sensors should be highly reliable and not missing. The experiments on test cases with sensors missing provide meaningful guidance for setting up smart home environments properly.

1.3 Thesis Overview

The remaining of this thesis is structured as follows.

- Chapter 2 discusses related works of this paper and reviews state-of-the-art.
- Chapter 3 defines the problem that this work aims to solve. The recognition and planning problem are mathematically defined. Problem properties are analyzed. Also notations and terminologies are given.

- Chapter 4 details the proposed HTN-GRP-PO algorithm. Section 4.1 explains the terminologies and associated data structures. Section 4.2 outlines the algorithm by a simple example. The following six sections Section 4.3 to Section 4.8 detail six modules of the algorithm respectively. Summary of the algorithm is given in Section 4.9.

- Chapter 5 conducts experiments to evaluate the algorithm's performance. It starts with the experimental scenario and knowledge base. The simulator for the experiment is designed in Section 5.2. Section 5.3 provides a list of test cases. Experiment results and discussions are presented in the last section.

- Chapter 6 concludes and discusses some interesting future work.

Chapter 2

Related Work

Various terminologies including goal recognition, plan recognition, intent recognition, and activity recognition, refer to the reasoning of what people are doing and what they will do next. Activity recognition is more about recognizing ongoing activities by analyzing low-level data gathered from physical sensors. Machine learning is widely used for activity recognition, see surveys [40, 59]. In contrast, goal, plan, or intent recognition aims to identify high-level goals by reasoning about observed low level steps. In our work, we focus on the latter kind of goal recognition.

Kautz et al. defined goal recognition as finding possible top level tasks (goals) to explain a set of observed steps [39]. Cohen et al. [14] classified goal recognition into two categories according to whether the older adult knows he/she is observed, namely *intended* or *keyhole* goal recognition respectively. In *intended* goal recognition, the person being observed collaborates with IAA. However in *keyhole* goal recognition, the person is not and doesn't know he/sher is being monitored. Note that the target users of IAAs in this work are the older adult with cognitive impairments, who does not have a complete knowledge of the domain and are likely to perform erroneous steps. So this work falls to the category of keyhole goal recognition, where users are not actively collaborating with the IAA.

The definition of goal recognition, which involves both high level goals and low level steps, already indicates its hierarchical nature. Such hierarchy is of multiple levels. It is also standard to use a knowledge base for goal recognition, despite differences in formulations. Considerations of goal recognition includes, but is not limited to, the number of actors, the number of goals, goals priors, partially and totally ordered steps, the likelihoods of explanations and erroneous steps.

In [1], goal recognition approaches are classified as either single layer approach or hierarchical approach. For single layer approaches, the reasoning process matches the observations directly to goals. The observations are usually raw data from sensors. However, in hierarchical approaches, the reasoning process recognizes the highest level goals together with inner level sub-goals. In this literature review, we adopt this taxonomy and discuss both non-hierarchy and hierarchy approaches, see Section 2.1 and Section 2.2 respectively. A brief summary of related work is given in Section 2.3.

2.1 Non-Hierarchical Approaches

Non-hierarchical approaches do not consider complex hierarchy of tasks. So it is indeed activity recognition rather than goal recognition. Often it boils down to a classification problem. Many machine learning techniques have been applied to activity recognition based on sensor data. Usually sufficient training data are needed for this kind of approach.

2.1.1 Bayesian Network (BN)

Blaylock and Allen [6] proposed a **Bayesian inference** based approach to recognize ongoing goals. Necessary prerequisite of their work is a given *plan corpus*, which stores goals and the corresponding plans an agent executes to achieve them. Given a sequence of observed steps, the probability of each goal is computed using Bayesian inference. Conditional probabilities are learned from the given plan corpus. This method is not scalable to large scale problems and complex tasks with hierarchical structures. Furthermore, collecting plan corpora might be extremely difficult in some domains.

Chen et al. [11] proposed a data mining framework for goal recognition in a smart home setting. The inputs of the framework are raw annotated data from sensors which are attached to objects in a three-bedroom apartment. In the feature extraction stage, two popular feature selection criteria, called minimal redundancy & maximal relevance and decision tree based on information gain, are used. They utilized four popular machine learning methods to predict what is happening, namely Bayesian network (**BN**), Artificial Neural Network (**ANN**), Sequential Minimal Optimization and LogitBoost.

2.1.2 Artificial Neural Network (ANN)

In [23], the authors applied artificial neural network (ANN) for activity recognition. The network is trained with standard back-propagation (BP). Using different features selected, ANN achieved competitive performance compared to other techniques.

Recently, Min et al. [43] proposed deep LSTM (Long Short Term Memory) network that can handle noisy and non-optimal behaviors of digital game players. LSTM is a special Recurrent Neural Network (RNN) for sequential temporal data. The proposed deep LSTM outperforms competing methods include single layer LSTM [44], n-gram encoded feed-forward neural network [42] and Markov logic network [30].

2.1.3 HMM and CRF

Hidden Markov Model (HMM) [67, 51] and Conditional Random Fields (CRF) are similar and popular probabilistic models used for activity recognition.

Valina HMMs are limited in modeling state durations. Van Kasteren et al. [63] claims that the modeling of accurate state durations is important for ADLs. For this motivation, Hidden semi-Markov models (HSMM) and semi-conditional random fields (SMCRFs) are proposed. The results shows that HSMM is much better than HMM. However, the performance differences between SMCRF and CRF are not significant.

Such probabilistic models deal with noise and uncertainty in a principled manner [63]. However, they don't scale well for complex network and relations, which is a major drawback of this kind of method.

2.1.4 Other Machine Learning Techniques

Fahad et al. [28] proposed a two level classification approach for activity recognition. First, they use the Lloyd's clustering algorithm [41] to separate activities into groups. Similar activities are in one group. Then activities inside a cluster are classified by Evident Theoretic K-Nearest Neighbor [18].

Statistical learning method can learn discriminant features without prior knowledge. However, such methods rely on large amounts of training data. Also it does not work for hierarchical tasks where complex relations and preferences exist among subtasks.

In summary, non-hierarchical methods focus on identifying activities based on sensor data. The advantage is their ability to reason purely based on data gathered from sensors.

This paper aims at developing an IAA for older adults with cognitive impairments. The non-hierarchical approaches have the following drawbacks.

- The inference from data to activity, without considering intermediate level tasks, makes status tracking not feasible. As a result, next steps cannot be predicted.
- Up to now, the accuracy of most of those methods are far less than 90%, which is not good enough for helping older adults with cognitive impairments.
- These methods are computationally expensive, particularly for complex problems.
- Non-hierarchical approaches need training data, which are not easy to obtain for some activities.

2.2 Hierarchical Approaches

The earliest work on goal recognition with a hierarchical approach has concentrated on story understanding. BELIEVER system by Schmidt et al. in 1978 [58] follows a hypothesize and revise strategy. It is a "top down" inference system which retrieves a single plan based on observation. Another way to understand story is *script* based [65]. Scripts describe regularities of the world. For instance, a classroom typically has tables, chairs, and blackboard. Based on scripts, the dominant reasoning paradigm in AI is adopted. Those methods need strong assumptions and can hardly handle multiple plans or goals.

A significant milestone in goal recognition is the conceptual framework published on SCIENCE in 1986 by Kautz and Allen [39]. In this work, they proposed the hierarchy format to represent the top level tasks and low level steps, and the concept of decomposition. This concept has became popular and standard. Their deductive inference follows a "bottom up" strategy to consider multiple explanations for inputs. Knowledge hierarchies are represented as axioms using the second order logic. There are two important assumptions in their work. Firstly, all the known ways to perform an goal are the only ways to perform that goal. Secondly, the given decompositions for a task are the only decompositions. Most of the knowledge based approaches follow these two assumptions.

Advantages of hierarchical approaches are summarized in [1]. Firstly, hierarchical approaches are especially suitable for recognizing high level tasks with complex structures. Secondly, they are suitable for interactions with humans. Thirdly, they can easily incorporate prior knowledge into the representation. Finally, with such a framework, less training data are needed.

Following Kautz and Allen's conceptual framework [39], many techniques have been developed. Here we review **Hierarchical Task Network (HTN)** based and **ontology** based hierarchical approaches, which are closely related to this work.

2.2.1 HTN-based Approach

HTN, a terminology in planning, is firstly proposed by Sacerdoti and Earl in 1975 [56]. In HTN planning, high-level (composite) tasks are recursively decomposed into simpler subtasks by domain artifacts called methods. The decomposition process continues until so called primitive tasks, which represent concrete steps to execute, are obtained. Primitive steps are realized by the domain artifact operator. Hierarchical modeling such as HTNs are believed to be a natural representation for complex cognitive models [13]. Researchers have utilized HTN framework to goal recognition based on a plan execution procedure and probability theory.

Chen et al. [12] followed the similar idea in HTN planning and constructed a hierarchical structure for composite and atomic activities and gestures. With such hierarchical structure, resolution based automated reasoning is adopted to recognize composite activity. The input of their algorithm is video streams. Firstly, a list of low level gestures are identified. Secondly, atomic actions are derived based on the identified gestures. Finally, high level goals are obtained using resolution based reasoning. Their framework seems general but does not take uncertainties into consideration.

Due to powerful representation of HTN knowledge base, HTN-based approaches are usually capable of dealing with goals with complex structures, such as partially ordered subtasks, concurrent goals, and so on. The seminal work by Goldman et al. [29] proposed HTN for goal recognition. The proposed framework is called PHATT (Probabilistic Hostile Agent Task Tracker). The framework can deal with partially ordered subtasks, overloaded steps (involved in the implementation of multiple tasks), contextual influence on choices of steps & goals and observation failures. The recipes of task implementation are coded in a task library using the HTN format described in [21]. Based on the domain knowledge, a series of rules, which adopt the Poole's logic of probabilistic Horn Abduction (PHA) [53], are proposed to reason about posteriors of steps occurring, the probabilities of goals, and the pending step sets.

Older Adults often forget the goals they are engaged in. Considering this, Geib added an additional module to PHATT to identify abandoned goals [24]. The probability a goal is abandoned is computed based on the observed action sequences which do not contribute to that goal. The proposed algorithm is tested on several examples. It turns out the time

complexity increases significantly with the increasing number of observations. The authors also mentioned that goals with similar steps increase the complexity of the problem.

In [26], three series of experiments are done to analyze how the impact factors influence the running time of PHATT [29]. The analyzed factors include tree depth, number of roots, different order constraints and early closed plans. The results show that the ordering constraints has the most significant effects on the algorithm's runtime, followed by the number of roots, and followed by the actual depth of the plan trees. Follow-up work [25] summarized previous work and integrated a constraint reasoning module into PHATT to consider parametrized actions and temporal constraints among actions. The main focuses of constraint reasoning are action durations and beginning time point of actions.

The PHATT framework proposed by Geib and Goldman is very powerful. It addresses several difficult issues in goal recognition, including multiple goals, partially ordered subtasks, temporal constraints, abandoned plans, and failure to observe steps. The main function of PHATT is to compute the conditional probability of a goal given the observation. The observation of PHATT are steps without uncertainty.

However, in reality, the steps are recognized based on sensor measurements, and usually sensors are not 100% reliable. Given data gathered with uncertainties, only probabilities of occurrence for steps can be obtained, rather than a deterministic step. For example, given a list of sensor notifications, we can say that we are 90% sure that action $a1$ has occurred based on the given sensor reliabilities and the current context.

Another limitation of PHATT is that it doesn't consider the influence of the current state on the selection of expansions. There may be more than one way to accomplish a task. Which way to chose really depends on the current state. Some ways might be unfeasible under a specific situation. Because PHATT does not consider the uncertainties in the step level, its state is deterministic. Such assumption is unrealistic in practice due to partial observability.

The algorithm proposed in this work is closely related to PHATT. The main difference is that we addressed the uncertainties in the step level and the influence of belief states on the selection of decomposition paths. The probability of a precondition being satisfied is used for goal recognition and task decomposition.

2.2.2 Ontology-based Approach

Ontology-based approaches are also knowledge-driven. They highlight the modeling of activities and behaviors with rich semantics. However, ontology-based modeling have drawbacks including incompleteness, inflexibility, and lack of adaption. Considering this, Okeyo

et al. [47] proposed a novel approach to learn and evolve activity models based on "seed" ADL ontologies. The algorithm analyzes data logs from sensors and evolves an activity model according to the predefined evolving condition. Their work does not recognize ongoing tasks. It is designed to generate customizable knowledge bases for users in a smart home setting.

According to [48], in ontology-based approach, activities are characterized into atomic activity, simple activity, and composite activity. A simple activity is defined as a sequence of ordered actions and a composite activity as two or more simple activities occurring within a given time interval. Composite activities are formulated using both ontological and temporal modeling formalisms.

In [49], two types of composite activities, concurrency & interleaved and sequential activities are handled. They exploit ontological reasoning for simple activity recognition and rule-based temporal inference to support the recognition of composite activities. Only when two or more simple activities are identified, can they infer composite activities.

To reason about temporal constraints among subtasks of a composite task, Okeyo et al. [46] proposed a hybrid ontological and temporal approach to model composite activities. The corresponding approach is based on their previous work [49]. They choose the 4D-fluents approach described in [4] to represent temporal knowledge. Personal preferences are also considered. For a specific activity model, several activity instances with different step order preference are provided.

The modeling of knowledge in our work is similar to those ontology based approaches. However, the considered problems are different. Firstly, this paper considered partially ordered relations of subtasks. It is insufficient to only involve the preferred order of a specific person like in [46]. In fact, even for a specific person, the order of steps to achieve an activity might change. Secondly, alternative ways of implementing a specific task under specific state is considered. Usually, there might be many different ways to achieve a composite task. Which way to choose depends on the current situation. Considering alternative branches make the system more flexible and applicable. Lastly, we consider partial observability of sensors, which are not addressed by their work.

2.2.3 Other Approach

Some other early works using hierarchical trees include parsing and Bayesian inference. Huff and Lesser [35] utilized a parser to derive a parse tree so as to explain the observations. This method has two main drawbacks. Firstly, it is over commitment unless it generates all possible explanations. Secondly, every step in the plan must be observed. In other

words, it can not deal with partial observations. Vilain [64] also formulated a deductive reference plan recognition method into a parsing problem. However, the parsing strategy is not good at dealing with partially ordered subtasks because of the explosion of grammar size.

For specific observations, there are multiple explanations. Charniak and Goldman [10] choose the most likely interpretation using Bayesian updates. In their work, the domain knowledge includes two parts. The predicate-calculus-like representations are used to explain the basic logic (relations) of domain knowledge. Each relationship in the domain knowledge has a Bayesian network, telling the conditional probability distribution. During reasoning, the predicate-calculus like representations construct a reasonable network through "introducing evidence", "Up-existential" and "Down-existential". The Bayesian networks specify the probabilities for the whole network. The Bayesian updates on all of the generated networks decides to what extent the evidence supports a plan hypothesis.

2.3 Goal Recognition in Smart Homes

Chen et al. [11] utilized adopted popular machine learning techniques to recognize ongoing activities based on sensor data. The targeted activities includes: cook, watch TV, use computer, groom, sleep, and bed to toilet. The best accuracy is about 90%. The activities considered in this work are quite simple, which do not involve complex hierarchical structures. Also, in order to identify what is happening, a complete sensor observation are needed in their work. In other words, an activity is identified after it has been finished. This does not work in an IAA where the agent is supposed to recognize ongoing goals before completion.

Fahad et al. [28] proposed a clustering based classification algorithm. The activities considered includes toilet, shower, breakfast, and so on. Such activity recognition method only cares about what the activity is. It doesn't care about how the activity is accomplished, which way to accomplish is chosen, or what are the next steps. As a consequence, these work is not that suitable for an IAA which helps the older adult to complete ADLs indepedently.

Bouchard et al. [8] used the lattice theory and action description logic to stress intra-dependencies of goals. Some goals share steps and goal recognition. For example, one person "GoToKitchen" and wants to "GetWater" to drink and then "StartWashing". Step "GoToKitchen" is shared by the two goals "PrepareTea" and "WashDish". In order to handle this problem, the authors proposed variable plans, which is used for concrete extra-

plans generation. Based on observed actions, the existing plans, and the current environment state, variables in a plan are substituted by concrete actions using the proposed plan composition process. The generated plans are guaranteed to be consistent.

In order to help older adults with cognitive impairments in smart home, Rafferty et al. [54] proposed an intelligent agent architecture based on the beliefs, desires, and intentions (BDIs) paradigm [57] and intention recognition (IR) techniques [2]. Specifically, in BDIs, beliefs represent the agents perception of the world, desires stores a library of the person's goals, and intentions are goals an person is pursuing. Beliefs and desires are modeled by ontologies. The IR process involves predicting the most likely goal based on observed steps. The system proposed seems powerful. However, various issues include multiple goals, partially ordered subtasks, and partial observability are addressed in their work.

Another important issue the above-mentioned works does not address is the goal recognition with multiple occupants. In [5], the authors give a survey on multiple occupant goal recognition in a smart home environment. They conclude that graphical probabilistic algorithms are very popular in the modeling and recognizing of activities with multiple occupancy. Besides, they declared that taking data association and interaction into account is a major issue for goal recognition in the context of multi-occupancy.

2.4 Related Work Summary

Due to limited representation of complex semantics of ADLs, single layered data-based approaches are not suitable for assistance agents. The hierarchical knowledge-based approach is more powerful in both the representations of complex relations and the reasoning about ongoing goals. Hierarchical-based approach narrows the search space for goal recognition by goal library and the tracking context. Hence complex problems become feasible.

To summarize, the issues addressed in the above-mentioned hierarchical knowledge-based approaches include: multiple goals, partially ordered subtasks, overloaded steps, abandoned plans, temporal constraints and multiple occupants. In our work, we only addressed some of them, including multiple goals and partially ordered subtasks. However, considering our unique application scenario, we also addressed issues like wrong steps due to cognitive impairments and partial observability due to unreliable or missing sensors.

Our algorithm is a hierarchical knowledge based approach. The knowledge base is expressed in methods and operators using similar formats described in SHOP2 [45], which is a well-known open source HTN planner. We also adopt the plan execution concept in [29]. The algorithm combines goal recognition with planning. As a result, it can not only

recognize what the older adult is trying to do, but also what are the proper next steps and tasks in order to achieve the recognized goals. Consequently, the algorithm is able to guide older adults with cognitive impairments to accomplish daily tasks by providing prompts in different levels. Our algorithm reasons from sensor data rather than steps. In this way, partial observability are considered in the step recognition procedure and the belief state updates process. Plan feasibility is guaranteed by checking the preconditions of methods and operators during planning.

Chapter 3

Problem Description

This chapter describes and defines problems addressed in this work. The definition part, Section 3.1, starts with a terminology summary, followed by the Hierarchical Task Network (HTN) planning problem and the goal recognition problem. After that, the goal recognition and planning problem is defined, which is the focus of this work. The problem is divided into categories in Section 3.2 based on several problem properties.

3.1 Definitions

3.1.1 Terminology Summary

Terminologies occur in the definitions which are also used throughout this paper are summarized as follows for reference use.

- *obj*: Object. An object refers to a concrete thing in the environment, such as a door, a cup, a chair and so on.
- *att*: Attribute. An attribute describes a specific property of an object that matters to the problem. For example, a door can have an attribute of *open-state*, a cup can have attributes includes *location*, *has-water*, and so on.
- *sensor*: Sensor. A sensor is used to measure the value of an attribute. This work assumes that one sensor measures one attribute, with a 1-to-1 binding relation.

- *obs*: Observation. A series of sensor readings which indicates values of attributes of objects.
- t: Task. A composite activity which cannot be achieved by one step. For example, *wash-hand* is a task which includes 5 steps: *turn-on-faucet*, *use-soap*, *rinse-hand*, *turn-off-faucet*, and *dry-hand*.
- *st*: Step. An atomic or primitive action which can be achieved through one step, such as *sit-down*, *turn-on-light*, and so on.
- g: Goal. A goal is a special composite task. It refers to a task that in the highest level, acting as the intent or purpose. Usually a series of steps need to be executed in order to achieve a goal. A task is a goal or not depends on the considered problem. For example, *make-tea* can be a goal if the user only needs help on *make-tea*. However if the user cares about *make-breakfast*, *make-tea* becomes a lower level subtask (subgoal) rather than a goal.
- G: Goals. G stands for all the goals in the problem. $g \in G$.
- D: Knowledge base for the problem. $D = (O, M)$. M is a set of methods, and O is a set of operators.
- m: A method. $m \in M$. A method matches to a task. It describes the preconditions, subtasks, and alternative branches to decompose the task.
- o: An operator. $o \in O$. An operator matches to a step. It describes the preconditions and effects of the step.
- *subt*: A subtask. As mentioned before, methods can be decomposed into lower level subtasks. A subtask in a method can be a method or an operator.
- $effect$: Effect. The outcome of a step, indicating the value changes of related attributes.
- *precondition*: Precondition. The prerequisites that a method or operator can be applied to make a goal, a task or a step happen. A precondition contains a list of items with the format of $(obj, att, value)$.
- s: State. A description of the real environment, including all related objects and their attributes' status.

- bs: Belief state. The agent's belief on the current environment based on all previous observations and reasonings.
- $PROB$: Goal distribution. After each iteration of reasoning, $PROB$ is the goal recognition result, indicating the likelihood of happening of all goals.
- PS: Pending set. After each iteration of reasoning, PS is the possible next tasks and steps in order to achieve $PROB$. It has multiple levels. Each element in PS includes level information, task or step name, and its probability. The probabilities act as the priors for the next reasoning iteration.
- PS_{step}: Step level pending set. It includes the step name and its probability. $PS_{step} \in PS$.
- $carryOn_{\pi_{executed}}$: The carry-on effects of the executed step sequence $\pi_{executed}$.

The problem definitions of HTN planning, goal recognition and goal recognition & planning are given in Subsection 3.1.2, Subsection 3.1.3, and Subsection 3.1.4, respectively.

3.1.2 HTN Planning

Definition 3.1.1 (**HTN Planning Problem**)**.** An HTN planning problem is a four tuple

$$P^p = (s, g, D, \pi).$$

The superscription p indicates that this is a planning problem. s is the initial state,consisting of a set of fluents that are true in the environment. g is the goal the generated plan trying to achieve. The HTN planning domain knowledge

$$D = (O, M),$$

includes a set of methods M and a set of operators O [27]. A method decomposes a composite task into subtasks, while an operator describes a primitive step. A solution to the problem

$$\pi = [st_1, st_2, st_3, ..., st_n],$$

is a step list which is executable in initial state s. Goal g should be reached after sequentially executing the steps in π.

Definition 3.1.2 (**Method**). A method is a list with the format of

$$m = (mName, precondition\ [\,], subtasks\ [\,], parent\ [\,], (startStep\ [\,])).$$

$mName$ is the task name that m can be applied to. *precondition* and *subtasks* are two lists with the same length. $(precondition[i], subtasks[i])$ is one of the decomposition branches when applying m to a task. It means that when $precondition[i]$ is satisfied in s, the corresponding task can be decomposed into $subtasks[i]$ by applying method m. *parent* specifies all methods whose *subtasks* contains $mName$. As one can see, *subtasks* specifies the top-down relationship while *parent* specifies the bottom-up relationship. *startStep* stores the beginning steps of a goal. It is optional since *startStep* is present only when m stands for a goal.

Each item in *precondition* is a list of fluents with the format of

$$precondition[i] = [(obj_1, att_1, value_1), (obj_2, att_2, value_2), ...].$$

It describes the required values of attributes of related objects to make m feasible. Each item in *subtasks* is a list of subtasks

$$subtasks[i] = (subt_1, subt_2, subt_3, ...).$$

$subt_i$ has $precedent(subt_i)$ and $decedent(subt_i)$, specifying the order of subtasks. $subt_i$ stands for either a method or a step. $subt_i$ should be executed after the completion of all tasks and steps in $precedent(subt_i)$ and must be completed before the start of any task or step in $decedent(subt_i)$.

Figure 3.1 is an example of a method written in JSON format. In the example, the corresponding task's name is *prepare-hot-water*. Both *precondition* and *subtasks* have two items. Thus there are two ways to accomplish the task. The required values of attributes are clearly stated in *precondition*. The (*person-1*, *ability*, 0.6) indicates that in order to complete the task, the person's awareness ability should be above 0.6. We evaluate a person's ability using a value between [0, 1]. The larger the better. The decomposition result, *subtasks* is listed with order information. The *parent* list tells the parent methods of *prepare-hot-water*. In this example, *prepare-hot-water* can be generated by decomposing both *make-tea* and *make-coffee*. The *startStep* is an empty list since *prepare-hot-water* is not a goal in the knowledge base.

Figure 3.1: Method example in JSON format


```
{
    "type":"method",
    "m_name":"prepare_hot_water",
    "precondition":[{
        "faucet_1":{
            "state":"off",
            "location":"kitchen"
        },
        "person_1":{
            "location":"kitchen",
            "ability": 0.6
        },
        "kettle_1":{
            "has_water":"no",
            "switch":"off",
            "water_hot":"no"
        }
    },
    {
        "kettle_1":{
            "has_water":"yes",
            "switch":"off",
            "water_hot":"no"

        },
        "person_1":{
            "location":"kitchen",
            "ability":0.6
        }
    }],
    "subtasks":[{
        "kettle_1_add_water":{
            "pre":[],
            "dec":["kettle_1_heat_water"]
        },
        "kettle_1_heat_water":{
            "pre":["kettle_1_add_water"],
            "dec":[]
        }
    },
    {
        "kettle_1_heat_water":{
            "pre":[],
            "dec":[]
        }
    }],
    "parent":["make_tea", "make_coffee"],
    "start_action":[]
}
```

Definition 3.1.3 (**Operator**). An operator is a three tuple

$$o = (oName, precondition\ [\,], effect\ [\,], parent\ [\,]).$$

$oName$ is the step name that o can be applied to. Similar to that in a method, *precondition* describes the circumstance in which the step can be executed. $effect$ is a list of fluents which comes to true after executing step $oName$. It has similar format to that of *precondition*. *parent* specifies all methods whose *subtasks* contains $oName$.

Figure 3.2 is an example of an operator written in JSON format. The operator matches to step *turn-on-faucet-1*. All attributes that occur in $effect$ must exist in *precondition*. The *parent* tells that both *wash-hand* and *kettle-1-add-water* need step *turn-on-faucet-1*.

Figure 3.2: Operator example in JSON format

```
{
    "type":"step",
    "st_name":"turn_on_faucet_1",
    "precondition":{
        "faucet_1":{
            "state":"off",
            "location":"kitchen"
        },
        "person_1":{
            "location":"kitchen",
            "ability": 0.6
        }
    },
    "effect":{
        "faucet_1":{
            "state":"on"
        }
    },
    "parent":["wash_hand", "kettle_1_add_water"]
}
```

According to the definitions of "method" and "operator", one can summarize the following characteristics of the knowledge base in the HTN planning.

- It can represent multiple ways to decompose a task or a goal.
- The methods of goals and tasks can share subtasks.

- It supports ordered, unordered, and disordered relationships of subtasks.
- The method for a goal has "start-action" information.
- All methods and operators have "parent" information.
- It provides knowledge pieces in "method" and "operator" format. A complete decomposition path, which contributes to a complete plan for the goal, is constructed during the planning process using the knowledge pieces.

Definition 3.1.4 (**Step Execution**)**.** A step st_i can be executed in state s if the corresponding operator o_i satisfies

$$o_i(precondition) \subset s,$$

The successor state after executing st_i in state s is

$$\theta(st_i, s) = s \vee o_i(effect),$$

where symbol "$\vee$" means making every fluent in $o_i(effect)$ true in s.

Definition 3.1.5 (**Carry-on Effect**)**.** Given the initial state s_0 and the already executed step sequence

$$\pi_{executed} = [st_1, st_2, st_3, ..., st_i],$$

the current state s_i can be calculated using

$$s_i = \theta(st_i, \theta(st_{i-1}, ..., \theta(st_1, s_0))).$$

Then the carry-on effects of $\pi_{executed}$ is

$$carryOn_{\pi_{executed}} = s_i - (s_0 \cap\ s_i).$$

The carry-on effects is used for wrong steps handling in the algorithm.

3.1.3 Goal Recognition

Definition 3.1.6 (**Goal Recognition Problem**)**.** A goal recognition problem is a tuple

$$P^r = (bs, obs, G, prior, D, PROB).$$

The superscription r indicates that this is a recognition problem. bs is the initial belief state of the real environment (values of attributes of related objects). obs is a series of

observations from sensors. G is a set of goals that matter to the problem, with $G = \{g_1, g_2, ..., g_n\} \subset M$. *prior* provides the priors of goals. D is the knowledge base which is the same as that defined in Definition 3.1.1. $PROB = \{g_1 : p_1,\ g_2 : p_2,\ ...,\ g_n : p_n\}$ is a probability distribution over G, which is the recognition result of the problem.

In Definition 3.1.6, *obs* comes from sensor measurements. A sensor is attached to an object to identify the value of a specific attribute of that object. For example a location sensor is attached to a person to identify the person's current location. Object attribute and sensor are 1-to-1 binding in this work. One sensor is used to observe one attribute and vice versa. As a result, the object name and object attribute together can determine a sensor. Definition 3.1.7 presents the official definition of sensors used in this work.

Definition 3.1.7 (**Sensor**)**.** A sensor is a five tuple

$$sensor = (obj, att, value, value\_num, reliability).$$

obj and *att* determine which sensor it is and describe the function of the sensor. *value* is the current sensor reading and *value_num* tells how many different values the sensor has. *reliability* indicates the degree that the sensor can be relied on. $reliability = 0.9$ means that the sensor gives the correct measurement with 90% percentage. The recognition problem considered in this work has **partial observability** since the sensors are not totally reliable.

Because of sensor unreliability, the agent could never know the real state of the environment. Thus the agent holds a belief state which is updated based on the observations and the reasoning procedure after each step. Definition 3.1.8 provides the format of belief states in this work. Figure 3.3 is an example of a belief state item which shows the beliefs on values of attributes of object $faucet\_1$. As one can see in Figure 3.3, for each attribute of an object, the belief state item holds a complete distribution of all the possible values.

Definition 3.1.8 (**Belief State**)**.** A belief state of an attribute is a tuple

$$bs_i = (obj, att, att\_val\_prob\_pair\ [\,]),$$

with *obj* and *att* specifying the attribute. *att_val_prob_pair* is a list, providing the complete distribution of all the values of the attribute.

Figure 3.3: Belief State example in JSON format

3.1.4 Goal Recognition and Planning

The purpose of this work is to help older adults with cognitive impairments to implement daily tasks independently. On the one hand, the agent should be able to recognize the older adult's intent, which is a goal recognition problem. On the other hand, the agent needs to tell what are the correct next steps so as to provide hints when the older adult gets stuck, which is a planning problem. Thus the problem considered in this work is a combination of goal recognition and planning. Therefore, the definition of the goal recognition and planning problem is given in Definition 3.1.9.

Definition 3.1.9 (**Goal Recognition and Planning Problem**)**.** A goal recognition and planning problem is a tuple

$$P^{rp} = (bs, obs, G, prior, D, PROB, PS),$$

where $(bs, obs, G, prior, D, PROB)$ is the goal recognition problem defined above, and $(bs, PROB, D, PS)$ is the planning problem. PS is the hierarchical pending set of the planning problem. It shows the next needed tasks and steps together with their probabilities in order to achieve $PROB$. Given the recognition result $PROB$, PS is generated through the planning process.

Unlike a pure planning problem, the result of the planning part in Definition 3.1.9, PS, only presents the next tasks and steps rather than a complete step sequence π. There are two considerations. Firstly, the recognition result $PROB$ is just a belief on the older adult's intent of the assistance agent. With more observations, the belief might change dramatically. Secondly, the assistance agent cannot control the older adult's next step

Table 3.1: A Goal Recognition and Planning Problem Example

Variable	Value
bs_0	Stores a list of JSON objects as shown in Figure 3.3. Each JSON object tells the belief states of an physical object's attributes.
obs_1	(*faucet-1*, *state*, *on*). This observation comes from sensor (*faucet-1*, *state*), whose reading at time point 1 is *on*.
G	(*wash-hand*, *make-coffee*, *make-tea*)
prior	{*wash-hand*: 0.333, *make-coffee*: 0.333, *make-tea*: 0.333}
D	Contains a list of methods as shown in Figure 3.1 and a list of operators as shown in Figure 3.2. A detailed explanation can be find in Appendix A and Appendix B.
$PROB_0$	{*wash-hand*: 0.333, *make-coffee*: 0.333, *make-tea*: 0.333}
PS_0	{level-0: {*turn-on-faucet-1*: 0.666, *switch-on-kettle-1*: 0.333}}
startStep	m(*wash-hand*): [*turn-on-faucet-1*], m(*make-tea*): [*turn-on-faucet-1*, *switch-on-kettle-1*], m(*make-coffee*): [*turn-on-faucet-1*, *switch-on-kettle-1*]

and cannot guarantee that the older adult will do a correct next step. Therefore, it is not necessary to generate a complete plan for the recognition result $PROB$. The feasible next tasks and steps are enough to guide the older adult to proceed forward. Furthermore, the partial planning process makes it easy to repair from the older adult's wrong steps and to change the recognition result with further observations. The generated PS has a hierarchical format. According to the user's awareness status, the assistance agent can choose hints at a proper level to help the user proceed smoothly and independently.

Table 3.1 shows an example of a goal recognition and planning problem. Time point 0 indicates the start point. The priors of the three goals are set to equal. Initially, $PROB_0$ is the same as *prior*. PS_0 has one level, containing the beginning steps of goals and corresponding probabilities. Row *startStep* shows the beginning steps of the three goals. *turn-on-faucet-1* has probability 0.666 because all the three goals can begin with *turn-on-faucet-1*, while *make-tea* and *make-coffee* can also begin with *switch-on-kettle-1*. So 0.333 + 0.1665 + 0.1665 = 0.666. obs_1 indicates that *faucet-1* becomes *on* at time point 1, and the change is monitored by sensor (*faucet-1*, *state*).

3.2 Problem Categories

The goal recognition and planning problem in this work is classified into eight categories based on three properties of the recognition problem. Firstly, according to the number of goals that the executed steps account for, the problem can be **single** goal recognition or **multiple** goals recognition. Secondly, according to if the executed steps contain wrong steps or not, the problem can be goal recognition **with / without** wrong steps. Note that during plan execution, a **wrong step** is a step which violates the order constraints between steps in π or does not belong to solution π at all. Lastly, according to if a sensor is present with a reliability or is simply missing, the problem can be goal recognition with **sensor reliability** or with **missing sensors**. Table 3.2 shows the eight categories of problem.

Table 3.2: Problem Categories

Sensor Config.	Single Goal		Multiple Goals	
	Correct Step	Wrong step	Correct Step	Wrong step
Reliability	**p1**	**p2**	**p3**	**p4**
Missing Sensor	**p5**	**p6**	**p7**	**p8**

In the "Sensor Config." column of Table 3.2 , "Reliability" means that all related sensors are present with a reliability. "Missing Sensor" means that some sensors are missing and the agent knows about which ones are missing. The other sensors are still present with a reliability. Wrong steps can be divided into two types: **non-related** wrong steps and **related** wrong steps. Definition 3.2.1 is an explanation of related wrong steps.

Definition 3.2.1 (**Related Wrong Step**)**.** Assume that there are two consecutive steps: st_i and st_{i+1}. st_i is a correct step while st_{i+1} is a wrong step. The carry-on effect of the executed sequence which ends with step st_i is $carryOn_{\pi_{executed}}$. The fluents in both $carryOn_{\pi_{executed}}$ and $effect_{st_{i+1}}$ are items with the format of $[obj, att, value]$. Then st_{i+1} is a related wrong step if and only if $effect_{st_{i+1}}$ has

$$[obj_m, att_n, value_x],$$

and $carryOn_{\pi_{executed}}$ has

$$[obj_m, att_n, value_y],$$

while

$$value_x \neq value_y.$$

If a wrong step is not a related wrong step, then it is an non-related wrong step.

As one can imagine, **p1** in Table 3.2 is the easiest problem, while **p8** is the hardest one. The proposed algorithm is designed to handle all those problems. Experiments will be done on each category to evaluate the performance of the algorithm.

Chapter 4

The Algorithm

The goal recognition and planning problem is defined and explained in Chapter 3. This chapter presents the algorithm, HTN-GRP-PO, which stands for "hierarchical task network based goal recognition and planning algorithm with partial observability". This chapter is divided into nine sections. Based on the example in Table 3.1, Section 4.1 explains the data structures and terminologies and Section 4.2 describes the framework of the algorithm in a high level and presents interrelationships among different modules. The details of each module are given in Section 4.3 to Section 4.7. Section 4.9 summarizes the proposed algorithm.

4.1 Data Structures and Terminologies

Definition 4.1.1 (**Algorithm Iteration**)**.** Denote sensor measurements at time t as obs_t, one iteration of the algorithm is the change of goal recognition and planning from

$$P_t^{rp} = (bs_{t-1}, obs_t, G, prior, D, PROB_{t-1}, PS_{t-1})$$

to

$$P_{t+1}^{rp} = (bs_t, obs_{t+1}, G, prior, D, PROB_t, PS_t).$$

An iteration is triggered by any change of sensor measurements. It updates the problem, produces the new goal recognition result, and gives hint for further tasks and steps. In this way, the assistance agent builds and corrects its beliefs on the ongoing goals based on observations. Changes of sensor measurements comes from step occurrences. This work

assumes that whenever sensors report any change of measurements, *only* one step happens. In other words, simultaneous steps are not considered in this work.

Table 4.1 shows an iteration which changes problem P_0^{rp} (same as that in Table 3.1) to problem P_1^{rp}. Sensor reliabilities are 0.9 in this example. The associated partial knowledge base of this iteration is shown in Figure 4.1. G, $prior$, and D are neglected in Table 4.1 because of no change. Note that PS_1 in Table 4.1 has more than one levels. However, only the step level (level 0) is shown to save space. One can easily see the changes from P_0^{rp} to P_1^{rp}. obs_1 is not the outcome but the trigger of an iteration. The iteration in Table 4.1 is triggered by obs_1. Similarly, obs_2 will trigger the next iteration. The algorithm has a basis of explanation, which is defined in Definition 4.1.2. Each iteration generates and updates explanations for observations so far and then computes $PROB$ and PS based on explanations.

Definition 4.1.2 (**Explanation**)**.** An explanation explains the observations so far by tracking ongoing goals and providing possible paths to proceed towards those goals from the tracked ongoing status. It is a tuple

$$expla = (prob, forest\ \ [\,], pendingStep\ \ [\,], startGoal\{\}).$$

The likelihood of this explanation is *prob*, indicating to which degree we can rely on this explanation. $forest$ is a list of goal networks (Definition 4.1.6) looks like

$$forest = [goalNet_1, goalNet_2, ..., goalNet_n],$$

recording the progress statuses of ongoing goals in the explanation. *pendingStep* provides the correct next steps suggested by the explanation, having format

$$pendingStep = [st_1, st_2, st_3, ...].$$

startGoal records which goals have been started in this explanation. It's a dictionary like

$$startGoal = \{g_1 : True,\ g_2 : False,\ g_3 : True,\ ...,\ g_m : False\}.$$

For each $\{g_x : True\} \in startGoal$, there is a corresponding goal network $goalNet_y \in forest$ with $goalNet_y(goalName) := g_x$.

Given a series of observations *obs*, there might be multiple explanations explaining *obs*. All those explanations are stored in $ExplaSet$ which is defined in Definition 4.1.3.

Definition 4.1.3 (**Explanation Set**). All the explanations of $obs = \{obs_1, obs_2, ..., obs_t\}$ are stored in a queue,

$$ExplaSet = Queue[expla_1, expla_2, ..., expla_n].$$

Every $expla_i \in ExplaSet$ is generated based on an explanation in $ExplaSet_{prev}$. $ExplaSet_{prev}$ is the $ExplaSet$ of the previous algorithm iteration. The number of explanations in $ExplaSet$ is denoted by $Expla_{num}$.

For example, after the iteration indicated in Table 4.1, there are three explanations, $ExplaSet = [expla_1, expla_2, expla_3]$, which are shown in Table 4.2. Each of them is a complete explanation of observation $obs = \{obs_1\}$. According to Table 4.2, $expla_1$ believes that *wash-hand* is ongoing and the supposed next step is *use-soap*. Note that in this example, each explanation only has one ongoing goal. As a result, their $forest$ only contains one element. For each explanation, the length of $forest$ equals to the number of its ongoing goals. The structure of a goal network in $forest$ is defined in Definition 4.1.6.

Definition 4.1.4 (**Tree**). A tree in this work is a hierarchical task network with a list of functions, including

- $tree.root()$: to get the root of the tree.
- $tree.create\_node(tag, id, parent, data)$: to add a new node to $tree$.
- $tree.paste(id, childTree)$: to append $childTree$ to the id node in $tree$.
- $tree.leaves()$: to return all the leaves of $tree$.

Definition 4.1.5 (**TreeNode**). A tree node in this work is a tuple

$$treeNode = (id, tag, level, data)$$

with

$$data = [completeness, readiness, precedent, decedent].$$

id is the unique identifier of the node and $name$ is the corresponding task or step name. $level$ is an integer, telling in which level the node is. $completeness$ tells if the task or step has been completed. $readiness$ indicates if the precedents of this node have been completed. $precedent$ and $decedent$ are the predecessors and successors of the node.

Definition 4.1.6 (**Goal Network**)**.** A goal network tracks the ongoing status of a goal and provides the possible paths to achieve the goal. It belongs to an explanation and is a six tuple

$$goalNet =$$

$$(goalName, tree, expandProb, pendingGoalNet, completeness, executeSequence).$$

tree is a hierarchical task network as defined in Definition 4.1.4 and Definition 4.1.5, which tracks the ongoing status of goal $goalName$. It is a hierarchical task network which is constructed with knowledge base (methods and operators) based on the recognized steps using *obs*. Nodes in *tree* stand for either completed / ongoing tasks or completed steps in order to achieve $goalName$. $expandProb$ tells the probability of the partial plan reflected in *tree* being chosen. *completeness* is a binary variable, indicating if the goal has been finished or not. $executeSequence$ contains the order of already executed steps and their carry on effects (refer to Definition 4.1.8). $pendingGoalNet$ stores the possible ways (decomposed goal network, Definition 4.1.7) to proceed towards $goalName$ from the ongoing status indicated in *tree*. It comes from decomposing *tree*.

Definition 4.1.7 (**Decomposed Goal Network**)**.** Given $goalNet$, each item in its $pendingGoalNet$ is a decomposed goal network indicating one path to proceed towards the goal. It's a three tuple

$$decompGoalNet = (decompTree, decompProb, pendingStep\ [\,]).$$

$decompTree$ is a hierarchical task network as defined in Definition 4.1.4 and Definition 4.1.5, containing all the nodes in $goalNet(tree)$ and the newly added nodes by decomposing ready tasks in $goalNet(tree)$. The details of the decomposing process is described in Subsection 4.6.2. $decompProb$ is the product of probabilities of selected branches' preconditions are satisfied during decomposition. $pendingStep$ is a step list representing the correct next steps suggested by $decompoTree$.

Definition 4.1.8 (**Execute Sequence**)**.** Given a goal network $goalNet$, its execute sequence is a tuple

$$executeSequence = (stepSequence, carryOn),$$

where $stepSequence$ is the queue (ordered) of already finished steps in $goalNet(tree)$, $carryOn$ is the carry on effects of steps in $stepSequence$. $carryOn$ is a dictionary with each item having the format of

$$\{[obj, att, value] : st_i\},$$

where st_i is the latest step in $stepSequence$ that makes $[obj, att, value]$ true.

Table 4.1: The Outcome of an Algorithm Iteration

P_0^{rp}		P_1^{rp}	
Variable	Value	Variable	Value
bs_0	(*faucet-1*, *state*, {*off*: 0.999, *on*: 0.001})	bs_1	(*faucet-1*, *state*, {*off*: 0.0001, *on*: 0.9999})
obs_1	(*faucet-1*, *state*, *on*)	obs_2	[(*hand-1*, *soapy*, *yes*), (*hand-1*, *dry*, *no*)]
$PROB_0$	{*wash-hand*: 0.333, *make-coffee*: 0.333, *make-tea*: 0.333}	$PROB_1$	{*wash-hand*: 0.3574, *make-coffee*: 0.3213, *make-tea*: 0.3213}
PS_0	{level-0: {*turn-on-faucet-1*: 0.666, *switch-on-kettle-1*: 0.333}}	PS_1	{level-0: {*use-soap*: 0.357, *add-water-kettle-1*: 0.643}}

Figure 4.1: Part of a Knowledge Base

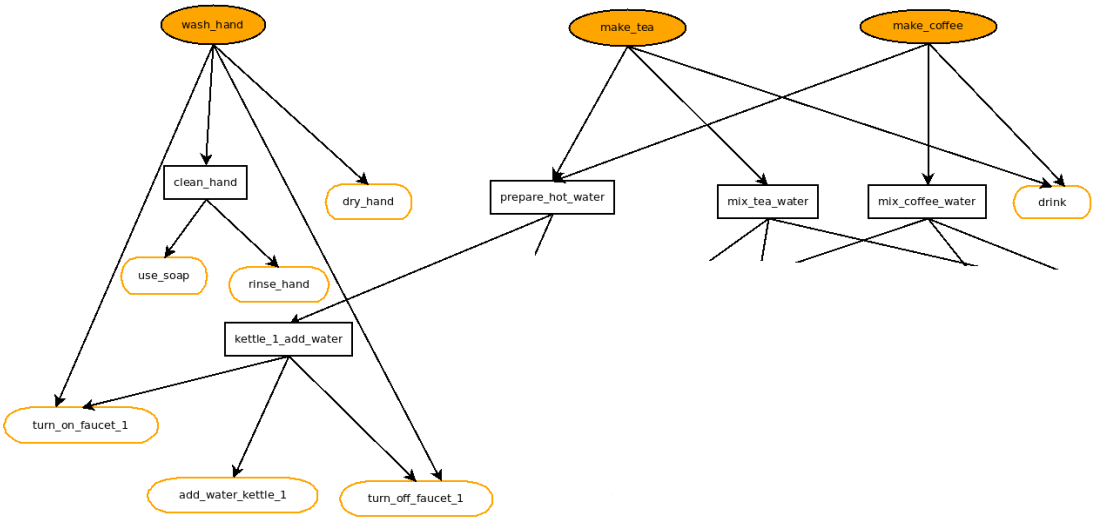


Table 4.2: Explanations after the Iteration shown in Table 4.1

Variable	$expla_1$	$expla_2$	$expla_3$
$prob$	0.3574	0.3213	0.3213
$forest$	$[goalN_1]$, refer Table 4.3	$[goalN_2]$	$[goalN_3]$
$pendingStep$	[*use-soap*]	[*add-water-kettle-1*]	[*add-water-kettle-1*]
$startGoal$	*wash-hand*: ***True***, *make-tea*: $False$, *make-coffee*: $False$	*wash-hand*: $False$, *make-tea*: ***True***, *make-coffee*: $False$	*wash-hand*: $False$, *make-tea*: $False$, *make-coffee*: ***True***

Table 4.3: Goal Network for $expla_1$ in Table 4.2

Variable	$goalN_1$
$goalName$	*wash-hand*
$tree$	$tree_1$, see Figure 4.2
$expandProb$	1.0
$pendingGoalNet$	$[decompGN_1]$ see Figure 4.2
$completeness$	$False$
$executeSequence$	*turn-on-faucet-1*, (*faucet-1*, *state*, *on*)

Figure 4.2: $tree_1$ and $decompGN_1$ in $goalN_1$

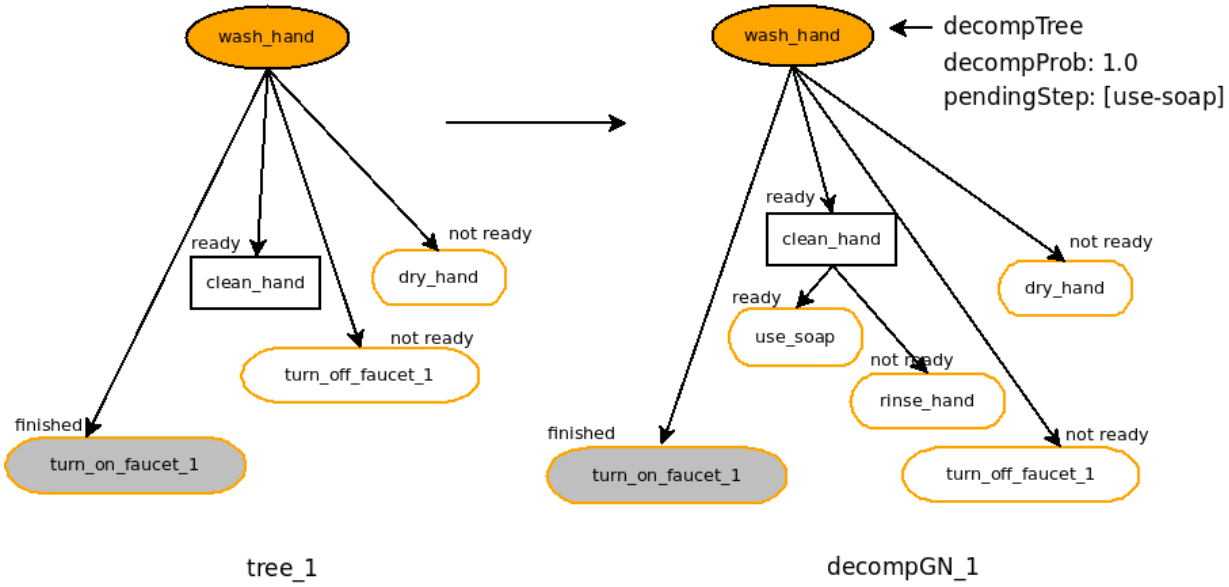


Table 4.3 shows the only goal network for the $expla_1$ in Table 4.2. The $goalName$ indicates that $goalN_1$ records the progress status of *wash-hand*. $tree_1$ is shown in Figure 4.2, indicating that *turn-on-faucet-1* has been finished. $expandProb = 1.0$ tells that all the already decomposed tasks in $tree_1$ only have one branch. $decomGN_1$, the only element in $goalN_1(pendingGoalNet)$ is also shown in Figure 4.2. It is derived from $tree_1$ by decomposing composite task *clean-hand*. Only $decompGN_1$ is derived means that there is only one way to accomplish *clean-hand*. This also explains $decompProb = 1.0$. Note that if there are two ways to accomplish *clean-hand*, $goalN_1(pendingGoalNet)$ would have two decomposed goal networks. The $goalN_1(completeness)$ is $False$ because *wash-hand* has not been completed. *turn-on-faucet-1* is the only step in $executeSequence$.

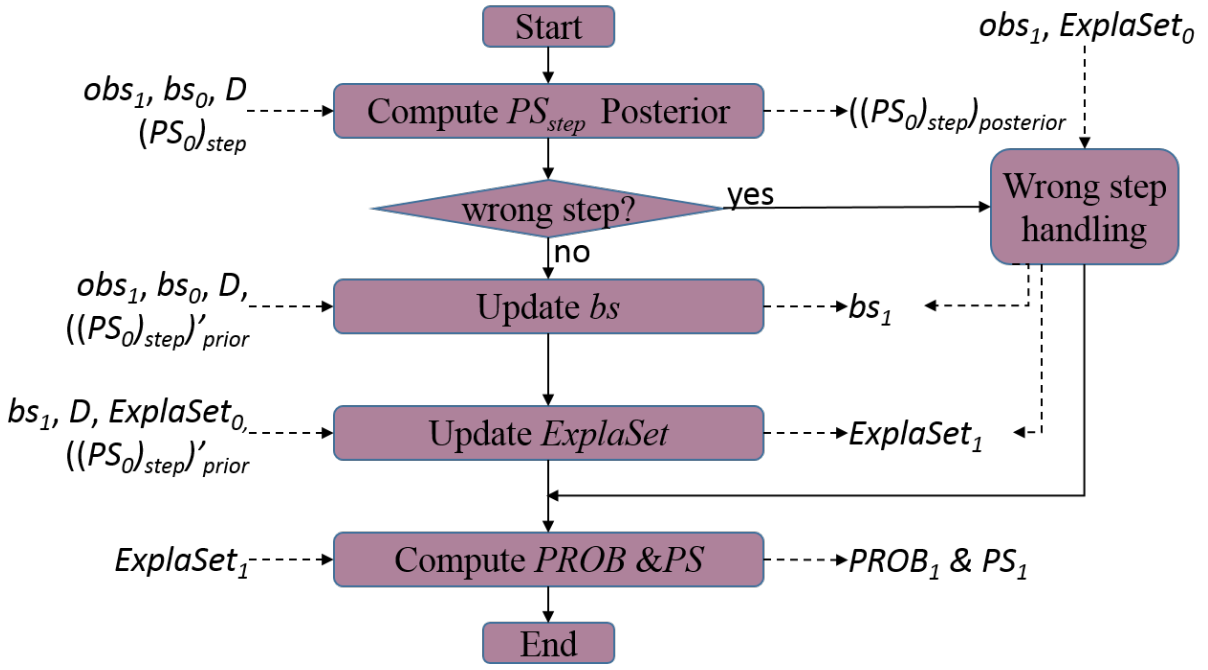
In summary, an iteration of the algorithm deals the problem with a basis of **Explanation**s. All the explanations are stored in an **Explanation Set**. An explanation has $forest$ storing a list of **Goal Network**. A $forest$ with length bigger than one means that in the explanation, multiple goals are in progress simultaneously. Each goal network in $forest$ explains the progress point of a goal. A goal network has $pendingGoalNet$ storing a list of **Decomposed Goal Network**. Based on the goal network, each decomposed goal network stands for a specific way to move forward from the progress point in order to achieve the goal. The **Execute Sequence** of a goal network records the already finished steps and their carry on effects of that goal.

4.2 The HTN-GRP-PO Algorithm

This section outlines the proposed algorithm. As defined in Definition 4.1.1, any change of sensor measurements triggers an iteration. In this section, we outline the algorithm using an iteration. Considering the iteration shown in Table 4.1, the reasoning procedures for it is shown in Figure 4.3, including the inputs and outputs of each module. Note that when the agent is started, an initialization module is executed to initialize $ExplaSet_0$, $PROB_0$, and PS_0 based on $D = (O, M)$ and the initial belief state bs_0. One can understand that this module creates the P_0^{rp} in Table 4.1. Details about this module is shown in Section 4.3.

An iteration starts with **Compute** PS_{step} **Posterior** module, which is the step recognition process adopting Bayesian inference (see Equation 4.1 on page 41). It computes the **posterior** of a step in PS_{step} that has occurred. For example, the iteration shown in Table 4.1 computes posteriors of steps in PS_0 level-0 using bs_0, obs_1, D and probabilities in PS_0 level-0. We call those posteriors $(PS_{step})_{posterior}$, which is the step recognition result of an iteration. In this example, $((PS_0)_{step})_{posterior}$ ={*turn-on-faucet-1*:0.8919, *switch-on-kettle-1*:0.0001}. Note that the sum of step probabilities in $((PS_0)_{step})_{posterior}$ is 0.892. It

Figure 4.3: An Algorithm Iteration



indicates that the probability of a wrong step has happened is 0.118. If the probability of a wrong step has happened is too high (e.g. 0.75), the algorithm will report a wrong step. Details of this module is presented in Section 4.4 with the criteria of detecting a wrong step.

If $(PS_{step})_{posterior}$ indicates that the just happened is a **correct step**, the algorithm drops the "a wrong step has happened" branch and normalizes $(PS_{step})_{posterior}$ to get $(PS_{step})'_{prior}$, which is step priors for belief state update. This procedure is necessary because the algorithm drops the "wrong step" branch and goes into a new stage. **Updates *bs*** using Bayesian inference based on *obs*, D, $(PS_{step})'_{prior}$, and bs_{last} (see Equation 4.8 on page 44). For example, the iteration shown in Table 4.1 computes bs_1 using obs_0, bs_0, $((PS_0)_{step})'_{prior}$ and D in the "Update *bs*" module. The high probability of (*faucet-1*, *state*, *on*) comes from two things. First, the high probability of *turn-on-faucet-1* in $((PS_0)_{step})'_{prior}$ which is almost 1.0. Second, the precondition and effects information indicated in the knowledge base which further supports the belief that *turn-on-faucet-1* has happened. Section 4.5 details how to update the belief state during an iteration.

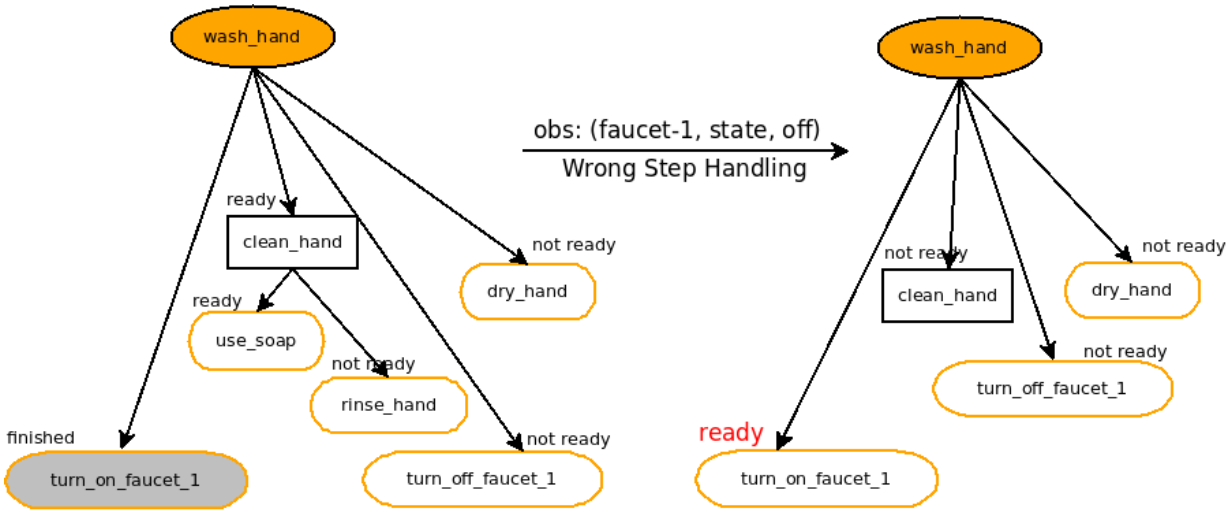
After updating belief state, the program **Updates *ExplaSet*** using D, bs, $(PS_{step})'_{prior}$ and $ExplaSet_{last}$. Each existing explanation in $ExplaSet_{last}$ ($ExplaSet_0$) will be updated to several new ones based on a given step and stored in the new $ExplaSet$ (the $ExplaSet_1$). The update includes two procedures: recognition and decomposition. Given a step st, the **recognition** procedure adopts a new $goalNet$ to represent the new ongoing status of the corresponding goal and computes the new explanation probability. The probability of a new explanation is the product of the following: the probability of the base explanation, the $expandProb$ of the new $goalNet$, and the posterior of the input step (see Equation 4.12 on page 56). The creation of the new $goalNet$s has two cases.

Case 1, st starts a new goal. In this case, there is no base $goalNet$ for creating a new one. A bottom up procedure as described in Subsection 4.6.1 is used to create a new $goalNet$ from scratch. Take the iteration in Table 4.1 as an example. Given st = *turn-on-faucet-1*, when creating $goalN_1$ for $expla_1$ in Table 4.2, the bottom up procedure is used to create $tree_1$ as shown in Figure 4.2. Note that case 1 enables the algorithm to handle concurrent goals. Case 2, st continues an ongoing goal. In this case, just choose a proper decomposed goal network from the given $goalNet(pendingGoalNet)$ as the new $goalNet$. For example, given $expla_1$ shown in Table 4.2 and st = *use-soap*, $decompGN_1$ (Figure 4.2, right) will replace $goalN_1$, becoming the new $goalNet$ in the new explanation.

The **decomposition** procedure creates $pendingGoalNet$ for a $goalNet$ through a decomposition procedure as described in Subsection 4.6.2. Take $goalN_1$ in Table 4.3 as an example. It's $pendingGoalNet$ is obtained through a decomposition procedure. The decomposition result is shown in the right part of Figure 4.2. When applying methods for decomposition, the probability that a precondition is satisfied is computed (see Equation 4.11 on page 51) and accumulated to derive $decompProb$, which indicates to which degree the corresponding decomposition path is feasible in bs. The decomposition process ends when every leaf in $tree$ is either a node standing for a step or a node standing for a task satisfying $node(data)(readiness) == False$. The "update $ExplaSet$" module is the most import part of the proposed algorithm. One can find a structural and detailed description for this module in Section 4.6.

If $(PS_{step})_{posterior}$ indicates that what just happened is a **wrong step**, the program will go into the **Wrong Step Handling** module. This module rectifies existing explanations so as to restore them from the wrong step. An visualization example of wrong step handling is shown in Figure 4.4. Assume that an explanation contains ongoing status of *wash-hand* as shown in the left tree of Figure 4.4. So the desired next step is *use-soap*. However, a wrong step is reported during the computation of $(PS_{step})_{posterior}$. Even though the algorithm does not know which wrong step has happened, the observation indicates that the effect of step *turn-on-faucet-1* has been destroyed by the wrong step. The wrong step handling

Figure 4.4: A Wrong Step Handling Example



module rectifies the ongoing status of *wash-hand* to the point as shown in the right tree of Figure 4.4. Consequently, the algorithm will remind the older adult to do *turn-on-faucet-1* again. One can refer to Section 4.7 for details of handling wrong steps.

The last step of an iteration is **Compute *PROB* and *PS***, which depends purely on the latest *ExplaSet*. For example, the iteration in Table 4.1 computes $PROB_1$ and PS_1 based on the three explanations shown in Table 4.2. The probability of goal g in $PROB$ is the sum of probabilities of explanations whose *startGoal* contains g. The probability of a task t (or step st) in PS is the sum of probabilities of explanations whose $forest$ contains a node standing for t (or st) with *completeness* being false while *readiness* being true. For details please refer to Section 4.8.

In the following sections, detailed implementations of the related modules are given one by one from Section 4.3 to Section 4.7. Those sections also show how the proposed algorithm is designed to deal with the problems raised in Section 3.2.

4.3 Agent Initialization

When the assistance agent is started, the initialization process is executed to initialize *ExplaSet* without any observation. This process is executed only once. The pseudo code of the initialization process is shown Algorithm 1.

Algorithm 1 Initialize $(M, G, prior)$

1: $ExplaSet \leftarrow Queue()$
2: $expla \leftarrow Explanation()$
3: $pendingStep \leftarrow Dict()$
4: **for each** $g \in G$ **do**
5: $\quad m_g \leftarrow m$, where $m \in M$ and $m(mName) == g$
6: $\quad$ **for each** $st \in m_g(startStep)$ **do**
7: $\quad\quad pendingStep[st] \leftarrow pendingStep[st] + prior(g)$
8: $\quad$ **end for**
9: **end for**
10: $expla(prob) \leftarrow 1$
11: $expla(pendingStep) \leftarrow pendingStep$
12: $ExplaSet \leftarrow ExplaSet.add(expla)$
13: **return** $ExplaSet$

After initialization, $ExplaSet$ has one $expla$ with probability 1. This is reasonable since without observations the algorithm believes that nothing is happening. The $expla(pendingStep)$ contains all the start steps of all the goals in G. Even though there are no observations, the algorithm gets to know the start steps purely based on the knowledge base.

In Algorithm 1, line 5 selects the correct method from M for goal g. Lines 6-8 build $pendingStep$ by adding new $\{st : prior(g)\}$ pairs or updating existing $\{st : prob\}$ pairs, where $prob$ stands for the accumulated prior of step st. The initialization process is such so that:

- A seed explanation should be created without observations.
- All the start steps of all the goals should be stored in $pendingStep$, which will be used in the oncoming algorithm iteration.
- Start steps related to a goal that has a higher prior should also have a higher probability in $pendingStep$ than others.

4.4 Compute PS_{step} Posterior

Deriving the posteriors for steps in PS_{step} is the step level recognition in the algorithm, which is the very first step of an iteration (refer to Figure 4.3). The posterior of a step

is calculated based on a standard Bayesian network shown in Figure 4.5 and the step's corresponding operator in the knowledge base.

In Figure 4.5, st_t stands for step st happening at time point t. The states at the last time point s_{t-1} and st_t together contribute to the current state s_t. s_t produces the observation at time point t, which is obs_t.

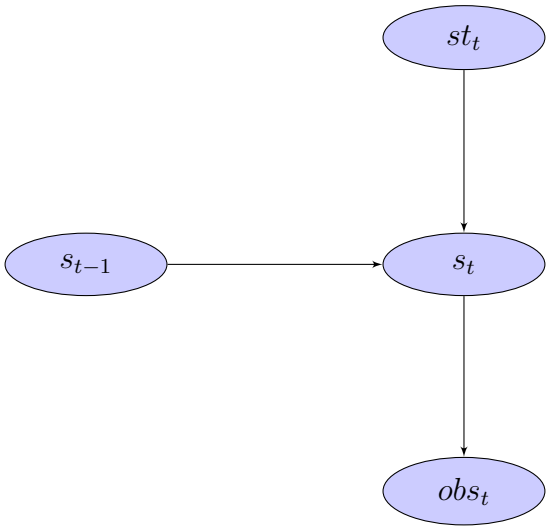


Figure 4.5: Bayesian Network for Step Posterior

$$
\begin{aligned}
p(st_t|obs_t) &= \frac{p(st_t, obs_t)}{p(obs_t)} \propto p(st_t, obs_t) \\
&= \sum_{s_t} \sum_{s_{t-1}} p(st_t, s_t, s_{t-1}, obs_t) \\
&= \sum_{s_t} \sum_{s_{t-1}} p(s_t|s_{t-1}, st_t) \times p(obs_t|s_t) \times p(s_{t-1}) \times p(st_t)
\end{aligned}
\tag{4.1}
$$

Based on Figure 4.5, Equation 4.1 computes the posterior of st_t given s_{t-1} and obs_t. Note that the s_{t-1}, s_t, and obs_t in Equation 4.1 only contain attributes relating to st_t, so as to reduce problem complexity. Items in s_{t-1}, s_t and obs_t have the format of $[obj, att, value]$. Equation 4.1 is applied to every step in PS_{step} to get step posteriors. $p(st_t)$ is the prior of step st_t in PS_{step}.

In Equation 4.1, given s_{t-1} and s_t, the conditional probability $p(s_t|s_{t-1}, st_t)$ is defined in Equation 4.2. The probability 0.999 is used because when the precondition of st_t is not satisfied, it is usually impossible to happen.

$$p(s_t|s_{t-1}, st_t) = \begin{cases} 0.999, & \text{if } st_t(precondition) \subset s_{t-1} \text{ and } \theta(st_t, s_{t-1}) \subset s_t \\ 0.001, & \text{otherwise} \end{cases} \tag{4.2}$$

Given s_t and its corresponding observation obs_t, the conditional probability $p(obs_t|s_t)$ is calculated using Equation 4.3. $attItem_{st_t}$ stands for the related attributes to step st_t. Each $item \in attItem_{st_t}$ is a pair (obj, att), specifying an attribute. Thus $sensor_{item}$ stands for the sensor monitoring the attribute. $(s_t)_{item}$ targets at the item in s_t which describes the status of the attribute.

$$\begin{aligned} p(obs_t|s_t) &= \prod_{item \in attItem_{st_t}} p(sensor_{item}) \\ p(sensor_{item}) &= \begin{cases} sensor_{item}(reliability), & \text{if } sensor_{item}(val) == (s_t)_{item}(val) \\ 1 - sensor_{item}(reliability), & \text{otherwise} \end{cases} \end{aligned} \tag{4.3}$$

Given s_{t-1}, $p(s_{t-1})$ is calculated using Equation 4.4. bs_{item} is the belief state distribution of the attribute specified by $item$. $(s_{t-1})_{item}(val)$ specifies which attribute value to choose in the belief state distribution.

$$p(s_{t-1}) = \prod_{item \in attItem_{st_t}} bs_{item}[(s_{t-1})_{item}(val)] \tag{4.4}$$

In the following, an example is used to explain how the posterior of a step is obtained using Equation 4.1. Assume that in Figure 4.5, st_t is *turn-on-faucet-1* as shown in Figure 3.2. Thus the related attributes occurring in $st_t(precondition)$ and $st_t(effect)$ is a list with length 4, which is shown in Equation 4.5. Assume that each of the attribute has two possible values, then the total number of **state instances** related to step st_t is $2^4 = 16$. One possible state instance is shown in Equation 4.6.

$$\begin{aligned} attItem_{st_t} = &[(faucet\text{-}1, state), (faucet\text{-}1, location), \\ &(person\text{-}1, location), (person\text{-}1, ability)] \end{aligned} \tag{4.5}$$

$$\begin{aligned} stateInstance_{st} = &[(faucet\text{-}1, state, \textbf{on}), (faucet\text{-}1, location, \textbf{kitchen}), \\ &(person\text{-}1, location, \textbf{kitchen}), (person\text{-}1, ability, 0.6)] \end{aligned} \tag{4.6}$$

When summing over in Equation 4.1, s_t and s_{t-1} can choose one of the 16 state instances. Thus the total number of combinations of s_t and s_{t-1} is $16 \times 16 = 256$. The sum in Equation 4.1 will enumerate all possible combinations of s_t and s_{t-1}.

By applying Equation 4.1 to every step in PS_{step}, step posteriors $(PS_{step})_{posterior}$ is obtained. The algorithm detects wrong steps based on $(PS_{step})_{posterior}$. A step is reported as a wrong step if the probability of $otherHappen$ is higher than a threshold $otherHappenThresh$. Equation 4.7 explains the wrong steps detection process. The computation of $otherHappenProb$ is simply deducting the sum of posteriors of steps in PS_{step} from 1. $otherHappenProb$ tells the probability of occurrence of some unknown steps which do not belong to PS_{step}. Comprehensive experiment results show that $otherHappenThresh = \textbf{0.75}$ is a good value for almost all of the problem categories defined in Table 3.2.

$$\begin{aligned} otherHappenProb &= 1 - \sum_{st \in PS_{step}} (PS_{step})_{posterior}(st) \\ otherHappen &= \begin{cases} True, & \text{if } otherHappenProb > otherHappenThresh \\ False, & \text{otherwise} \end{cases} \end{aligned} \tag{4.7}$$

4.5 Belief State *bs* Update

According to Figure 4.3, "Update *bs*" is executed when the algorithm believes that a correct step has occurred. Firstly, $(PS_{step})_{posterior}$ is normalized by 1 to get $(PS_{step})'_{prior}$, since the algorithm drops the "wrong step handling" branch and goes into the handling of a correct step. The formula to update belief state is shown in Equation 4.8, which is applied to attributes relating to the current iteration one by one.

$$
\begin{aligned}
p(s_t|obs_t) &= \frac{p(s_t, obs_t)}{p(obs_t)} \propto p(s_t, obs_t) \\
&= \sum_{s_{t-1} \in (att_i)_{value}} \sum_{st'_t \in (PS_{step})'_{prior}} p(s_t, st'_t, s_{t-1}, obs_t) \\
&= \sum_{s_{t-1} \in (att_i)_{value}} \sum_{st'_t \in (PS_{step})'_{prior}} p(s_t|s_{t-1}, st'_t) \times p(obs_t|s_t) \times p(s_{t-1}) \times p(st'_t)
\end{aligned}
\tag{4.8}
$$

For example, here we update the belief state for attribute att_i. We assume that the possible values of att_i are $(att_i)_{value} = [v_1, v_2]$ (only binary attributes are considered in this work). In Equation 4.8, the sum over s_{t-1} enumerates values in $(att_i)_{value}$. The sum over st_t enumerates values in $PS_{step})'_{prior}$. $p(v_1|s_{t-1}, obs_t)$ and $p(v_2|s_{t-1}, obs_t)$ are computed separately using Equation 4.8 and normalized over 1 to be the new belief state on attribute att_i.

In Equation 4.8, $p(st'_t)$ is the probability of st'_t in $(PS_{step})'_{prior}$. $p(s_{t-1})$ is the algorithm's belief that att_i has value s_{t-1} after the reasoning of the last iteration. Assume that $s_{t-1} = v_1$ and $s_t = v_2$, the conditional probability of $p(s_t|s_{t-1}, st_t)$ is obtained using Equation 4.9. The conditional probability of $p(obs_t|s_t)$ is obtained using Equation 4.10.

$$
p(v_2|v_1, st_t) = \begin{cases} 0.999, & \text{if } v_1 \in st'_t(precondition) \text{ and } v_2 \in st'_t(effect) \\ 0.001, & \text{otherwise} \end{cases}
\tag{4.9}
$$

$$
p((obs_t)_{att_i}(value)|v_2) = \begin{cases} sensor_{att_i}(reliability), & \text{if } sensor_{att_i}(value) == v_2 \\ 1 - sensor_{att_i}(reliability), & \text{otherwise} \end{cases}
\tag{4.10}
$$

Since Equation 4.8 is only applied to related attributes, the first step of "Update bs" is to obtain all the related attributes of the current iteration. A related attribute presents in the $effect$ of at least one of the steps in PS_{step}. The procedure of obtaining those related attributes is depicted in Algorithm 2. It returns a list of attributes with the format of $obj\_att = [att_1, att_2, att_3, ...]$. Algorithm 3 updates the belief states of those related attributes. In Algorithm 3, the for loop in line 4 sums over s_{t-1} and the for loop in line 5 sums over st_t.

Algorithm 2 ObtainRelatedAttribute (PS_{step})

```
obj_att ← Set()
for each st ∈ PS_step do
    o ← o ∈ O and o(oName) == st
    for each e ∈ o(effect) do
        obj_att.add(e)
    end for
end for
return obj_att
```

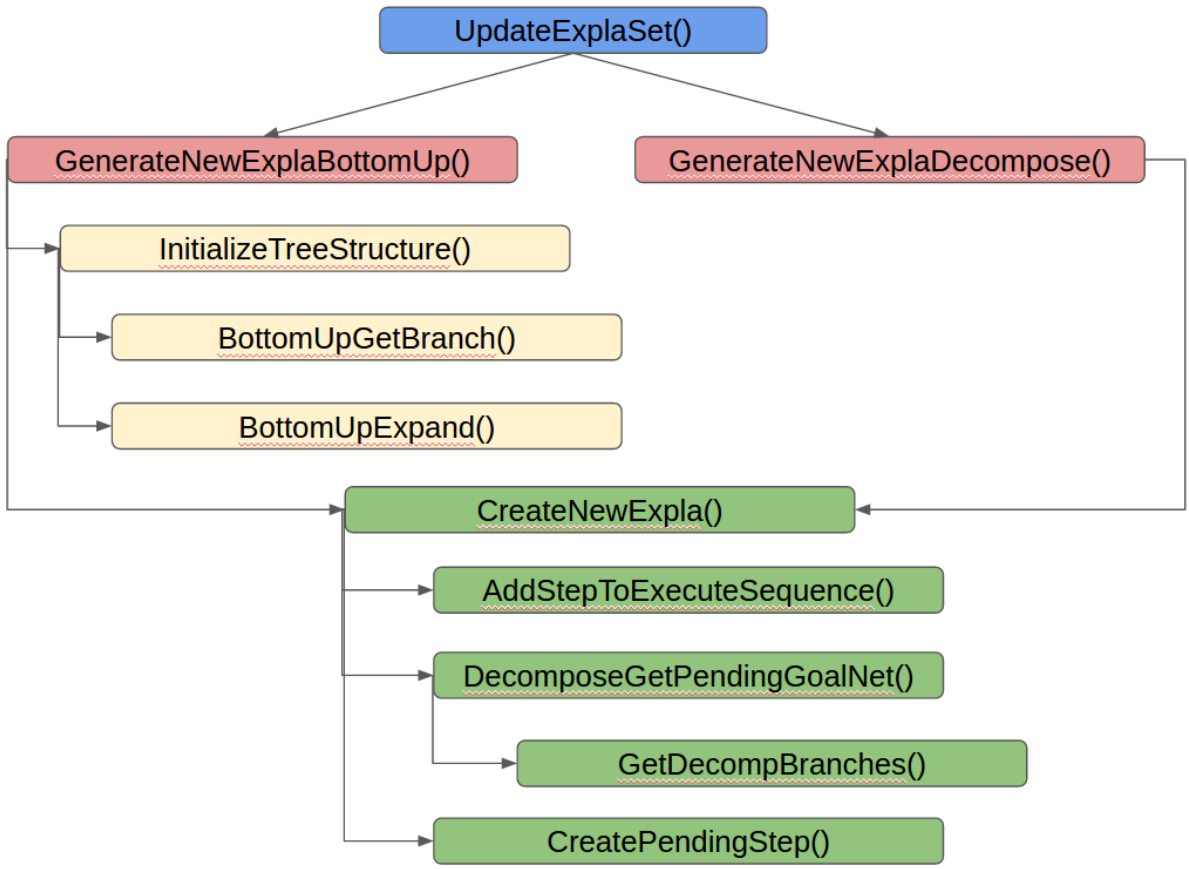
Algorithm 3 UpdateBeliefState ($obj\_att, (PS_{step})'_{prior}, obs_t, bs_{t-1}, sensor$)

```
for each att ∈ obj_att do
    for each s_t ∈ att_value do
        p(s_t) ← 0
        for each s_{t-1} ∈ att_value do
            for each st_t ∈ PS_step do
                pp ← p(s_t|s_{t-1}, st_t) × p(sensor_att(value)|s_t) × p(s_{t-1}) × (PS_step)'_prior[st_t]
                p(s_t) = p(s_t) + pp
            end for
        end for
        bs[att][s_t] ← p(s_t)
    end for
    normalize bs[att]
end for
return
```

4.6 Explanation Set Update

The most important module of the proposed HTN-GRP-PO algorithm is "update *ExplaSet*". *ExplaSet* is updated in this module based on two principles: the step recognition result $(PS_{step})'_{prior}$ which is obtained after dropping the "wrong step handling" branch, and the updated belief state bs_t which is obtained from the "update *bs*" module. The update on *ExplaSet* is a process which combines procedures of goal recognition and planning. It consists of several functions as shown in Figure 4.6. The blue rectangle *UpdateExplaSet*() is the module interface.

Figure 4.6: *ExplaSet* Update Breakdown



Algorithm 4 gives the pseudo code of function *UpdateExplaSet*(). The *newExplaSet* is created based on *ExplaSet* and will be assigned to *ExplaSet* at the end of the algorithm. A set of new explanations can be generated based on an existing $expla \in ExplaSet$ and a

Algorithm 4 UpdateExplaSet $(ExplaSet, bs_{t-1}, bs_t, (PS_{step})'_{prior}, D)$

1: $newExplaSet = Queue()$
2: **for each** $expla \in ExplaSet$ **do**
3: **for each** $(st : st_{prob}) \in ((PS_{step})'_{prior}$ **do**
4: $new\_expla \leftarrow$ ***GenerateNewExplaBottomUp***$(st, st_{prob}, expla, bs_{t-1}, D)$
5: $newExplaSet.extend(new\_expla)$
6: $new\_expla \leftarrow$ ***GenerateNewExplaDecompose***$(st, st_{prob}, expla, bs_t, D)$
7: $newExplaSet.extend(new\_expla)$
8: **end for**
9: **end for**
10: $ExplaSet \leftarrow newExplaSet$
11: **return** $ExplaSet$

step st which has probability of occurrence st_{prob} (line 2-3). There are two ways to create new explanations given $expla$ and $(st : st_{prob})$.

$GenerateNewExplaDecompose()$, which is the right red rectangle in Figure 4.6, creates new explanations by exploring on an existing $goalNet$ in $expla(forest)$ (Algorithm 4, line 6). Each new explanation is generated by function $CreateNewExpla()$, which is the green part in Figure 4.6.

$GenerateNewExplaBottomUp()$, which is the left red rectangle in Figure 4.6, creates new explanations by adding a new $goalNet$ to $expla(forest)$ (Algorithm 4, line 4). It means that according to the explanation the old adult just starts a new goal with step st. We firstly generate new $goalNet$s based on st using $InitializeTreeStructure()$, which is a bottom up initialization process as presented in the yellow rectangles in Figure 4.6. Then we generate new explanations based on the newly created $goalNet$s using $CreateNewExpla()$.

In Algorithm 4, the generated new explanations is attached to $newExplaSet$ (line 5 and line 7). After the two loops on $ExplaSet$ and $(PS_{step})'_{prior}$, $ExplaSet$ is replaced by the newly generated $newExplaSet$ (line 10). The returned $ExplaSet$ will be used to calculate $PROB$ and PS of this iteration. As indicated in the green part in Figure 4.6, the two ways to generate new explanations share the function $CreateNewExpla()$. The shared part is where the goal recognition and planning procedures are combined together. Detail explanations of each function in Figure 4.6 is given in the following subsections.

4.6.1 Bottom Up Initialization

This subsection explains how to generated new explanations using the bottom up procedure. The pseudo code of function $GenerateNewExplaBottomUp()$ is shown in Algorithm 5. Given the input $expla$ and (st, st_{prob}), it returns a list of new explanations, new_explas. Each explanation in the returned new_explas holds a newly added $goalNet$. It indicates that in the explanation, a new goal has been just started with st. Therefore, the $GenerateNewExplaBottomUp()$ can be applied only when the input $expla$ has a goal satisfying: the goal has not been started (line 4) and the goal's $startStep$ contains st (line 6). Otherwise, no new explanations will be created in this procedure.

In Algorithm 5, before creating new explanations (line 8-14), new $goalNet$s are created through function $InitializeTreeStructure()$ (line 7). $InitializeTreeStructure()$ creates new $goalNet$s using a bottom up expansion process starting with st. Those $goalNet$s account for goals which can start with step st. The given $expla$ can be updated into a new explanation with each $goalnet \in goalNets$ using procedure $CreateNewExpla()$ (line 11). The Algorithm 10 in Subsection 4.6.2 demonstrates the details of the $CreateNewExpla()$ procedure.

Tree Structure Initialization

Given bs_{t-1}, D and st, $InitializeTreeStructure()$ generates new $goalNet$s for goals which can start with st. Step st matches to a leaf node in a $goalNet(tree)$. Starting with the leaf node, the $goalNet(tree)$ is constructed through a bottom up expansion procedure. Parents nodes are iteratively added to the root of the current tree, until the root of the tree reaches the highest goal level. The pseudocode of $InitializeTreeStructure()$ is shown in Algorithm 6. Note that bs_{t-1}, rather than bs_t, is used in the bottom up procedure because st occurred in bs_{t-1}. Thus a bottom up path to a goal that is possible in bs_{t-1} rather than bs_t is needed.

The returned value of Algorithm 6 is a list of $goalNet$s rather than a single one, all of which are built based on st. There two major reasons. Firstly, there might be more than one goals starting with step st. Secondly, there are multiple paths from st to one goal because of different branches of methods. Algorithm 6 implements a breadth-first-search (BFS) in order to obtain all those possible $goalNet$s. The candidate tree structures are the mid products of creating $goalNet$s. The candidate tree structures and their probabilities of being chosen are stored in $tempForest$. Initially there is only one tree which contains one node standing for st (line 2). The corresponding $expandProb$ is 1 (line 3). The expansion process of a tree ends when the corresponding method of the root of the tree does not have any parents(line 10-11 and 23). Inside the while loop (line 5), candidate trees are checked one by one (line 8) to see if it needs further bottom up expansion.

Algorithm 5 GenerateNewExplaBottomUp $(st, st_{prob}, expla, bs_{t-1}, D)$

```
1: new_explas = List()
2: tempStartGoal ← expla(startGoal)
3: for each g ∈ tempStartGoal do
4:    if g(started) == False then
5:       m_g ← m ∈ M with m(mName) == g
6:       if st ∈ m_g(startStep) then
7:          goalNets ← InitializeTreeStructure(st, bs_{t-1}, D)
8:          for each goalnet ∈ goalNets do
9:             if tempStartGoal[goalnet(goalName)](started) == False then
10:               tempStartGoal[goalnet(goalName)](started) ← True
11:               new_expla ← CreateNewExpla(expla, st, st_prob, goalnet, D)
12:               new_explas.add(new_expla)
13:            end if
14:         end for
15:      end if
16:   end if
17: end for
18: return new_explas
```

Algorithm 6 InitializeTreeStructure (st, bs_{t-1}, D)

$goalNets \leftarrow List()$; $tempForest \leftarrow Queue()$; $tree \leftarrow Tree()$
$tree.create\_node(id = st, tag = st, data(completeness) = True)$
$tree(prob) \leftarrow 1$
$tempForest.enqueue(tree)$
while $length(tempForest) > 0$ **do**
  $len \leftarrow length(tempForest)$
  for i in range(len) **do**
    $theTree \leftarrow tempForest.dequeue()$
    $rootName \leftarrow theTree.root()(tag)$
    $m_{root} \leftarrow m \in M$ with $m(mName) == rootName$
    $parentList \leftarrow m_{root}(parent)$
    if $length(parentList) > 0$ **then**
      for each $p \in parentList$ **do**
        $m_p \leftarrow m \in M$ with $m(mName) == p$
        $BottomUpBranches \leftarrow$ ***BottomUpGetBranch***$(m_p, rootName, bs_{t-1})$
        for each $branch \in BottomUpBranches$ **do**
          $tempTree \leftarrow theTree$
          $tempTree(prob) \leftarrow branch(prob) \times theTree(prob)$
          $tempTree \leftarrow$ ***BottomUpExpand***$(p, tempTree, branch(subtasks))$
          $tempForest.enqueue(tempTree)$
        end for
      end for
    else $\{length(parentList) == 0\}$
      $newGoalNet \leftarrow$ ***New goalNet*** $(rootName, theTree, theTree(prob))$
      $goalNets.append(newGoalNet)$
    end if
  end for
end while
return $goalNets$

If the tree has reached a goal (line 23, the *parentList* has length 0), the expansion process ends. Thus a *newGoalNet* will be created (line 24) and added into the returned *goalNets* (line 25). Otherwise, new candidate trees are created by adding one more layer to the top of *theTree* (line 13-22) and saved into *tempForest* (line 20) waiting for further expansion. For each $p \in parentList$, several new candidate trees with root p can be generated. The number of newly generated candidate trees using p depends on the number of different branches containing the current root in method m_p. Line 15 in Algorithm 6 gets those branches and their probabilities of being chosen in bs_{t-1} using function *BottomUpGetBranch*().

The details of *BottomUpGetBranch*() is shown in Algorithm 7. With each $branch \in BottomUpBranches$, a new candidate tree is created by adding the parent node p and the subtasks in *branch* to the top of *theTree* using *BottomUpExpand*() (line 19), which is explained in Algorithm 8. The new probability is obtained by multiplying the newly added *branch*(*prob*) and the previous *theTree*(*prob*) (line 18).

As explained in Definition 4.1.6, in order to create a *goalNet*, one needs to specify six parameters. However, the *InitializeTreeStructure*() function only specifies the first three variables, which are *goalName*, *tree*, and *expandProb* (Algorithm 6, line 24). Given a tree *theTree* which has reached to a goal and is ready to be used to create a new *goalNet*, the tag name of its root becomes *goalName*; the tree itself becomes *tree*; the probability of this specific expansion path is selected, which is *prob*, becomes *expandProb*. The others will be specified in function *CreateNewExpla*() which is described in Subsection 4.6.2.

Getting bottom up branches

Function *BottomUpGetBranch*() is used to return branches of m_p whose subtasks containing *subt*, based on the given method m_p, subtask *subt*, and belief state bs_{t-1}, The pseudo code is shown in Algorithm 7. Each element in the returned *branches* (line 15) has *prob* and *subtasks*, where *prob* explains to which degree the branch's precondition is satisfied in bs_{t-1}, and *subtasks* shows the subtasks of that branch. The probability of a precondition *prec* being satisfied in bs_t is calculated using Equation 4.11.

$$prob(prec)_{bs_t} = \prod_{(obj_i, att_i, value_i) \in prec} bs_t[obj_i][att_i][value_i] \tag{4.11}$$

In Algorithm 7, the degrees of satisfaction for preconditions in m_p are computed and normalized in lines 1-5. The loop on branches of m_p selects branches whose subtasks contain the given *subt* (line 7-8). Those branches are added into the returned *branches* with their *subtasks* and the probabilities of precondition satisfaction (lines 9-12).

Algorithm 7 BottomUpGetBranch $(m_p, subt, bs_{t-1})$

```
1: prob ← List()
2: for each prec ∈ m_p(precondtion) do
3:    prob.append(prob(prec)_{s_{t-1}})
4: end for
5: normorlize prob
6: branches ← List()
7: for (i = 0; i < len(m_p(subtasks)); i++) do
8:    if subt ∈ m_p(subtasks)[i] then
9:       branch ← Dict()
10:      branch(prob) ← prob[i]
11:      branch(subtasks) ← m_p(subtasks)[i]
12:      branches.append(branch)
13:   end if
14: end for
15: return branches
```

Bottom up Expansion

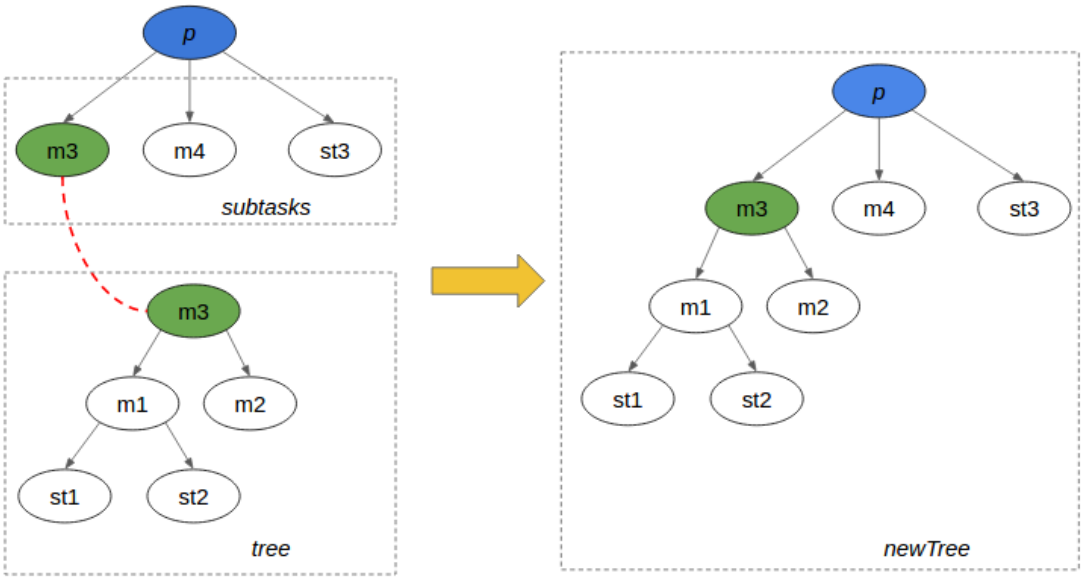
Algorithm 8 is the pseudo code for $BottomUpExpand()$. The inputs include the new root tag p, the current tree structure $tree$, and the chosen branch's subtask list $subtasks$. Figure 4.7 is a visualization of function $BottomUpExpand()$.

Firstly a single node tree, $newTree$, is created using p (line 1-3). Secondly, the input $tree$ is attached to the $newTree$ (line 9), together with the decedent and precedent constraints of $tree$'s root node (line 7-8). The joint point is the root of $tree$ and the subtask in $subtasks$ which has the same name as $tree.root$. For example, in Figure 4.7 the joint node is $m3$. Finally, a loop on $subtasks$ is used to add the other children of p into $newTree$(line 12-17). In Figure 4.7, the other children are $[m4, st3]$. The generated $newTree$ will be returned to Algorithm 6, waiting for the next expand iteration.

Algorithm 8 BottomUpExpand $(p, tree, subtasks)$

1: $newTree \leftarrow Tree()$
2: $data \leftarrow Data(completeness = False, readiness = True)$
3: $newTree.create\_node(tag = p, id = parent, data = data)$
4: $currentTreeRoot \leftarrow tree.root()$
5: **for each** $subt \in subtasks$ **do**
6: **if** $subt == currentTreeRoot(tag)$ **then**
7: $currentTreeRoot(data)(precedent) \leftarrow subt[pre]$
8: $currentTreeRoot(data)(decedent) \leftarrow subt[dec]$
9: $newTree.paste(currentTreeRoot(id), tree)$
10: **end if**
11: **end for**
12: **for each** $subt \in subtasks$ **do**
13: **if** $subt \neq currentTreeRoot(tag)$ **then**
14: $newData \leftarrow Data(precedent = subt[pre], decedent = subt[dec])$
15: $newTree.create\_node(tag = subt, id = subt, parent = newTree.root(), data = newData)$
16: **end if**
17: **end for**
18: **return** $newTree$

Figure 4.7: BottomUpExpand



4.6.2 Top Down Decomposition

As shown in Algorithm 4 and Figure 4.6, another way to create new explanations is to use function $GenerateNewExplaDecompose()$, whose pseudo code is depicted in Algorithm 9. Our main idea is to replace one of the $goalNet$s in $expla$ with a new one. We regard it as a top down decomposition process in the sense that the $newGoalNet$ is generated from the old $goalNet$ using a top down decomposition procedure.

Algorithm 9 GenerateNewExplaDecompose $(st, st_{prob}, expla, bs_t, D)$

```
1:  new_explas = List()
2:  for each gN ∈ expla(forest) do
3:    for each decompGoalnet ∈ gN(pendingGoalNet) do
4:      if st ∈ decompGoalnet(pendingStep) then
5:        newTree ← decompGoalnet(decompTree)
6:        g ← newTree.root()(tag)
7:        expProb ← decompGoalnet(decompProb)
8:        newGoalNet ← New goalNet (g, newTree, expProb, gN(executeSequence))
9:        new_expla ← CreateNewExpla(expla, st, st_prob, newGoalNet, D, gN)
10:       new_explas.append(new_expla)
11:     end if
12:   end for
13: end for
14: return new_explas
```

The key point of this function is that st proceeds to one of the goals that has started in $expla$, leading to the evolvement of $expla$. Before revising $expla$ to a new explanation, two questions need to be answered. **Firstly**, which goal does st contribute to? **Secondly**, which decomposition path is chosen if st is executed in order to proceed towards the goal?

Thanks to the $pendingGoalNet$ parameter in a $goalNet$ structure (Definition 4.1.6), the algorithm is capable of dealing with those two questions. A $goalNet(tree)$ represents the progress point of a goal. The $pendingGoalNet$ of a $goalNet$ stores all the decomposition paths based on $goalNet(tree)$, which are generated through the planning process of the last iteration. Each decomposition path reaches to steps by executing which can proceed towards the goal from the goal's progress point.

Consequently, for the first question, the goal should have a decomposition path whose pending steps contains st. For the second question, the path containing pending step st is the chosen decomposition path. With those two points, lines 2-4 in Algorithm 9 loop and

Algorithm 10 CreateNewExpla $(expla, st, st_{prob}, newGoalNet, D, oldGoalNet)$

1: $stNode \leftarrow newGoalNet(tree).get\_node(st)$
2: $stNode(completeness) \leftarrow True$
3: $newGoalNet(tree).updateCompleteness()$
4: $newGoalNet(tree).updateReadiness()$
5: $nES \leftarrow$ ***AddStepToExecuteSequence***$(newGoalNet(executeSequence), st, O)$
6: $newGoalNet(executeSequence) \leftarrow nES$
7: $new\_prob \leftarrow st_{prob} \times newGoalNet(expandProb) \times expla(prob)$
8: $pGNets \leftarrow$ ***DecomposeGetPendingGoalNet*** $(newGoalNet(tree), D, bs_t)$
9: $newGoalNet(pendingGoalNet) \leftarrow pGNets$
10: $newForest \leftarrow expla(forest)$
11: $newForest.remove(oldGoalNet)$
12: $newStartGoal \leftarrow expla(startGoal)$
13: **if** $newGoalNet(completeness) == True$ **then**
14: $\quad newStartGoal[newGoalNet(goalName)](started) = False$
15: **else** $\{newGoalNet(completeness) == False\}$
16: $\quad newForest.append(newGoalNet)$
17: **end if**
18: $new\_expla \leftarrow$ ***New Explanation*** $(new\_prob, newForest, [], newStartGoal)$
19: $new\_expla.CreatePendingStep()$
20: **return** new_expla

find the target goal and the decomposition path *decompGoalnet*. The $newGoalNet$ is created based on *decompoGoalnet* (line 5-8), where the *tree* and *expandProb* are correspondingly initialized with *decompGoalnet*(*decompTree*) and *decompGoalnet*(*decompProb*). With $newGoalNet$, a new explanation is created in function $CreateNewExpla()$.

New Explanation Creating

Function $CreateNewExpla()$ updates an explanation by replacing one of the explanation's *goalNet*s, which is $oldGoalNet$, with the input $newGoalNet$. The reason for doing this is to make the explanation account for the input *st*. The pseudo code of this function is in Algorithm 10. On the one hand, this function updates the input explanation's probability and the corresponding *goalNet* to explain the input *st*, which is a goal recognition logic. On the other hand, this function generates $pendingGoalNet$ for the updated *goalNet* to provide the correct next tasks and steps considering the new progress status of the target goal, which is a planning process.

In order to create a new explanation, there are six sequential steps, most of which modify the $newGoalNet$.

- Step 1: mark st as completed in $newGoalNet$. This includes changing the completeness status of the corresponding node in $newGoalNet(tree)$ to $True$ (line 2); updating the other nodes' completeness and readiness status (line 3-4); and adding st into the $newGoalNet(executeSequence)$ (line 5-6). The criteria of updating the completeness and readiness statuses for nodes are straight forward. For completeness, a $treeNode$ can be marked as $completeness = True$ only when all its children are completed. For readiness, a $treeNode$ can be marked as $readiness = True$ only when all its precedents are completed.

- Step 2: compute the probability of the new explanation using Equation 4.12 (line 7). The new probability is the product of the posterior of st, the newly added branching factor when choosing st as the next step, and the probability of $expla$.

$$new\_expla(prob) = st_{prob} \times goalnet(expandProb) \times expla(prob) \tag{4.12}$$

- Step 3: decompose $newGoalNet(tree)$ to generate $newGoalNet(pendingGoalNet)$ (line 8-9), which is a planning process as shown in Algorithm 12. Before executing the planning process, $newGoalNet(tree)$ has been revised in line 1-4.

- Step 4: Update $startGoal$ and replace $oldGoalNet$ with $newGoalNet$ in $newForest$ (lines 10-17). As explained in lines 13-17, if $newGoalNet$ indicates that the goal has been finished with the newly added st, the goal's ongoing status should be set to $False$ (line 13). In this case, $newGoalNet$ should not be added into $newForest$.

- Step 5: Initialize the new explanation with all the prepared information (line 18).

- Step 6: Update $pendingStep$ for the new explanation. Function $CreatePendingStep()$ (line 19) is adopted. It simply collects all the $decompGoalNet(pendingSteps)$ from each $goalNet(pendingGoalNet)$.

Function $AddStepToExecuteSequence()$, which is used in Algorithm 10 lines 5, includes 2 steps. The corresponding pseudo code is shown in Algorithm 11. Firstly, the given $step$ is appended into the $stepSequence$ list in line 1. Secondly, in lines 3-5, the $carryOn$ is extended based on the effects of $step$. Function $DecomposeGetPendingGoalNet()$ is more complicated and is explained in the following paragraphs.

Algorithm 11 AddStepToExecuteSequence ($executeSequence, step, O$)

1: $executeSequence(stepSequence).add(step)$
2: $o \leftarrow o \in O \ \ and \ \ o(oName) == step$
3: **for each** $(obj_i, att_i, value_i) \in o[effect]$ **do**
4: $\quad executeSequence(carryOn).update((obj_i, att_i, value_i))$
5: **end for**
6: **return** $executeSequence$

Tree Decomposition

Given a progress point of a goal indicated by $tree$, $DecomposeGetPendingGoalNet()$ is used to explore paths to proceed towards the goal from that point. Each path is stored in a $DecompGoalNet$. A path is obtained by applying methods to tasks whose corresponding nodes in $tree$ is a leaf node with readiness status $True$. Methods decompose those tasks into subtasks and expand the tree in a top down direction. This exploring contributes to the next steps & tasks hint. The process of tree decomposition ends when all the leaves in $tree$ are one of the following. This two criteria guarantee that the planning process will reach to the lowest step level.

- it's a node standing for a step
- it's a node standing for a task satisfying $node(data)(readiness) == False$.

Given a leaf node $treeNode_i$ standing for task t, with $treeNode_i(data)(readiness) == True$, the corresponding method $m\_node \in M$ should be applied to decompose t. The decomposed subtasks are added into the tree structure. Because of the multiplicity of a method's branches, the planning process will produce multiple new tree structures with a base tree. Each new tree stands for one decomposition path to reach the lowest step level.

Algorithm 12 explains the BFS implementation of the top down planning process. $treeQueue$ stores $queueItem$s which need further decomposition. Each $queueItem$ contains a network with a base of $tree$ and a probability telling the likelihood of the decomposition path is chosen. Initially, $treeQueue$ contains one item with $tree$ as the network and probability 1 (line 2-6). The while loop on $treeQueue$ in lines 7-32 is the BFS based planning. During each iteration, pop out an element $thisTree$ from $treeQueue$ (line 8) and check if the tree has finished the planning process. The check process is another loop on leaves of $thisTree(tree)$ (line 11). Each leaf is inspected according to the criteria mentioned above to see if the node needs further decompositions (line 12). The planning process on $thisTree$ ends only when all its leaves do not need further decompositionss.

Algorithm 12 DecomposeGetPendingGoalNet $(tree, bs_t, D)$

$pendingGoalNets \leftarrow List()$
$treeQueue \leftarrow Queue()$
$queueItem \leftarrow Dict()$
$queueItem[tree] \leftarrow tree$
$queueItem[prob] \leftarrow 1$
$treeQueue.enqueue(queueItem)$
while $treeQueue$ is not $empty$ **do**
  $thisTree \leftarrow treeQueue.dequeue()$
  $leaves \leftarrow thisTree[tree].leaves()$
  $flag \leftarrow True$
  for each $node \in leaves$ **do**
    if $node(data)(readiness) == True$ and $node$ stands for $method$ **then**
      $m_{node} \leftarrow m \in M$ with $m(mName) == node(tag)$
      $flag \leftarrow False$
      $decompBranches \leftarrow$ ***GetDecompBranches***(m_{node}, bs_t)
      for each $branch \in decompBranches$ **do**
        $newQueueItem \leftarrow Dict()$
        $newQueueItem[tree] \leftarrow thisTree[tree]$
        $newQueueItem[tree].create\_nodes(node(tag), branch(subtasks))$
        $newQueueItem[prob] \leftarrow thisTree[prob] \times branch(prob)$
        $treeQueue.enqueue(newQueueItem)$
      end for
      break
    end if
  end for
  if $flag == True$ **then**
    $thisTree[tree].udpateReadiness()$
    $pendS \leftarrow leaf.tag$ with $leaf \in thisTree[tree].leaves()$ and $leaf(data)(readiness) == True, leaf(data)(completeness) == False$
    $newDGNet \leftarrow$ ***New DecompGoalNet*** $(thisTree[tree], thisTree[prob], pendS)$
    $pendingGoalNets.append(newDGNet)$
  end if
end while
return $pendingGoalNets$

During each iteration of the while loop in line 7, there are two cases. Case 1, the popped out $thisTree$ do not need further decompositions ($flag == True$). Then a new $decompGoalNet$ is created in lines 26-31 and is added into the return list in line 30. The $pendingStep$ takes all leaves in $thisTree(tree)$ standing for steps with readiness status true but completeness status false (line 28).

The other case is that the check process of $thisTree$ (line 11) detects a leaf node who needs further decompositions. The decomposition process for that node is executed by applying the corresponding method m_{node} in lines 15-22. Firstly, all branches of m_{node} is obtained through $GetDecompBranches()$ (line 15). Details of this function is provided in Algorithm 13. Secondly, for every $branch \in decompBranches$, creating a $newQueueItem$ by adding $branch(subtasks)$ into $thisTree(tree)$ (line 18-19) and updating the decomposition probability (line 20). The probability is obtained by multiplying the previous accumulated probability and the newly added $branch(prob)$. The newly created $newQueueItem$ is added into $treeQueue$, waiting for the next check process.

Algorithm 13 is the pseudo code for function $GetDecompBranches()$. The returned $branches$ is a list. Each branch contains its subtasks and the probability of the branch's precondition is satisfied in bs_t. The degree of satisfaction for a precondition is calculated using Equation 4.11

Algorithm 13 GetDecompBranches (m_{node}, bs_t)

1: $branches \leftarrow List()$
2: **for** $(i = 0; i < len(m_{node}(precondition)); i++)$ **do**
3: $\quad branch \leftarrow Dict()$
4: $\quad prec \leftarrow m_{node}(precondition)[i]$
5: $\quad branch(prob) \leftarrow prob(prec)_{s_t}$
6: $\quad branch(subtasks) \leftarrow m_{node}(subtasks)[i]$
7: $\quad branches.append(branch)$
8: **end for**
9: normalize on $branch(prob) \in branches$
10: **return** $branches$

4.7 Wrong Steps Handling

As shown in Figure 4.3 and Equation 4.7, if a wrong step is detected, the program switches into the wrong step handling process. This module repairs the goal networks of an explanation from the wrong step and recovers the impacts of the wrong step. As indicated in Definition 3.2.1, wrong steps can be classified as related wrong steps or non-related wrong steps. Non-related wrong steps do not lead to any change of an explanation. In order to retrieve an explanation from a related wrong step, a repairing process on its $goalNet$s is needed which will be explained later.

Algorithm 14 details how to handle a related wrong step. The inputs of function $HandleWrongStep()$ do not include a specific step. Although a wrong step is reported, the algorithm does not know which step it is. Therefore in this module, the algorithm needs to recover from an unknown related wrong step according to the differences between obs_{t-1} and obs_t, which is called $sensorNotif$. The crucial part of function $HandleWrongStep()$ is the $goalNet$ repairing process in lines 5-18.

Given an $expla$ in $ExplaSet$, the process is applied to every $goalNet$ in $expla(forest)$. The fluents in the carry on effects, which are no longer true due to the wrong step's effects, should be repaired. The effects to be repaired for $expla$ are collected and stored in $explaRepairSummary$ (line 3). A $explaRepairSummary$ together with its explanation's probability are added into $repairSummary$ which belongs to $ExplaSet$ (line 20-22). $repairSummary$ contains the impacted effects and their probabilities (the degree of being impacted) of all explanations. Following the repairing process, line 24 updates the belief state to bs_t using $repairSummary$. Finally, with the amended belief state st_t and $goalNet$s, lines 25-29 generates the new $pendingGoalNet$ for every $goalNet$.

Given $sensorNotif$, in Algorithm 14 we propose the following procedures to repair a $goalNet$ from the unknown wrong step.

- Find out the affected steps which have been executed in order to achieve the goal. In line 5, affected steps are obtained by comparing $sensorNotif$ with the carry on effects of $goalNet$.
- For nodes of affected steps, change their completeness status to $False$ (line 7).
- For nodes whose completeness or readiness status have been changed, update the completeness and readiness status for nodes which relate to them (line 8). Function $UpdateCompletenessAndReadiness()$ includes three steps. Given an affected node: firstly, all of its parents' completeness are changed to $False$; secondly, all of its

decedents' readiness and completeness are changed to $False$; thirdly, if the node is not leaf but has a $False$ readiness status, remove the node and its children (the subtree) from $goalNet(tree)$. Please notice that, affected nodes are not only the ones standing for the affected steps, but also the ones whose completeness or readiness statuses have been changed.

- Remove fluents which have been destroyed by the wrong step from the carry on effects. Line 11 removes the affected steps from $goalNet(executeSequence)(stepSequence)$, so as to create the new $carryOn$ which does not contain the effects that are no longer true.

- Get the impacted effects of the goal. In the algorithm, those impacted effects is stored in $effectRepairSummary$ in line 12. It is obtained by comparing the new $carryOn$ and the old one. The $goalNet$'s effect repair summary is added to $explaRepairSummary$ in line 13.

- If a $goalNet$'s execute sequence has length 0 (all steps were affected and removed), remove it from $expla(forest)$ (line 15-18).

Generally, a related wrong step rewinds the progress status of a goal by destroying the executed steps' carry on effects. The wrong step handling module manipulates this rewinding process and forces the explanation to go back to a restored progress point, from where one can continue towards to the goal. It is not necessary to repeat the destroyed step sequence. Once the explanation has been repaired, it will allow all possible ways to achieve the goal from the restored point. This is important since the older adult might choose another way to proceed forward.

Algorithm 14 HandleWrongStep ($sensorNotif, ExplaSet$)

$repairSummary \leftarrow Dict()$
for each $expla \in ExplaSet$ **do**
  $explaRepairSummary \leftarrow Dict()$
  for each $goalNet \in expla(forest)$ **do**
    $affectedStep \leftarrow GetAffectedSteps(sensorNotif, goalNet(executeSequence))$
    for each $st \in affectedStep$ **do**
      $goalNet(tree).get\_node(st)(data)(completeness) \leftarrow False$
      $goalNet(tree) \leftarrow UpdateCompletenessAndReadiness(goalNet(tree), st)$
    end for
    $oldExecuteSequence \leftarrow goalNet(executeSequence)$
    $goalNet.UpdateExecuteSequence(affectedStep)$
    $effectRepairSummary \leftarrow goalNet.GetRepairSummary(oldExecuteSequence)$
    $explaRepairSummary.extends(effectRepairSummary)$
    $exeS \leftarrow goalNet(executeSequence)(stepSequence)$
    if $length(exeS) == 0$ **then**
      $expla(startGoal)[goalName](started) \leftarrow False$
      $expla(forest).remove(goalNet)$
    end if
  end for
  for each $(obj_i, att_i, value_i) \in explaRepairSummary$ **do**
    $repairSummary[obj_i][att_i][value_i] + expla(prob)$
  end for
end for
Update bs_{t-1} to bs_t based on $repairSummary$
for each $expla \in ExplaSet$ **do**
  for each $goalNet \in expla(forest)$ **do**
    $goalNet(pendingGoalNet) \leftarrow DecomposeGetPendingGoalNet(goalNet(tree), D, bs_t)$
  end for
end for
return $ExplaSet$

4.8 Derivation of $PROB$ and PS

As shown in Figure 4.3, "calculate $PROB$&PS" is the last executed module of an iteration. The goal recognition result $PROB$ and hints for the next tasks and steps PS are generated purely based on $ExplaSet$, which is the one newly generated in the same iteration. The pseudo code for this calculation is shown in Algorithm 15.

Algorithm 15 GeneratePROSandPS ($ExplaSet$)

```
1:  PROB ← Dict()
2:  PS_task ← Dict()
3:  PS_step ← Dict()
4:  for each expla ∈ ExplaSet do
5:    for each goalNet ∈ expla(forest) do
6:      gName ← goalNet(goalName)
7:      PROB[gName] ← PROB[gName] + expla(prob)
8:      for each treeNode ∈ goalNet(tree) and treeNode(data)(completeness) ==
        False and treeNode(data)(readiness) == True do
9:        nodeName ← treeNode(tag)
10:       if treeNode is step then
11:         PS_step[nodeName] ← PS_step[nodeName] + expla(prob)
12:       else {treeNode is task}
13:         pprob ← PS_task[nodeName]['prob']
14:         PS_task[nodeName]['prob'] ← pprob + expla(prob)
15:         PS_task[nodeName]['level'].add(treeNode(level))
16:       end if
17:     end for
18:   end for
19: end for
20: PS ← (PS_step, PS_task)
21: return PROB, PS
```

Basically, the probability of goal g in $PROB$ is the sum of probabilities of explanations whose $forest$ contains a $goalNet$ for goal g (Algorithm 15, line 6-7). The probability of a task t(or step st) in PS is the sum of probabilities of explanations whose $forest$ contains a node standing for t (or st) with $completeness$ being false while $readiness$ being true (Algorithm 15, line 8-17). The $completeness$ and $readiness$ constraints guarantee that only tasks and steps ready to go are added into PS.

When computing PS, a node standing for a step is stored in PS_{step} with its probability (Algorithm 15, line 10-11). A node standing for a task is stored in PS_{task} with its probability and level information (Algorithm 15, line 12-15). The step probabilities in PS_{step} will act as the priors of steps when calculating step posteriors in the next iteration.

The calculation of $PROB$ and PS seems simple given $ExplaSet$. The key point is how to generate the new $ExplaSet$ during each iteration. Four complicated modules are needed to generate $ExplaSet$. Details of those modules are given in the following subsections.

4.9 Algorithm Summary

The proposed HTN-GRP-PO algorithm has partial observability in the sense that the sensors may have reliabilities less than 1 or even missing. For the algorithm, a missing sensor has no difference from a sensor with reliability 0.5. Thus, the algorithm treats missing sensors in the same way as for sensors of reliability 0.5. In this work, the partial observability is considered in the following aspects.

- The step recognition process (Section 4.4) computes the posterior of a step occurring with the consideration of sensor reliabilities.
- We take sensor reliability into account when updating the belief states (Section 4.5).
- Explanation construction (Subsection 4.6.1 and Subsection 4.6.2) determines the probability of an explanation and the probability of a bottom up or top down path to be chosen by utilizing the degree of satisfaction for preconditions of knowledge base pieces (methods and operators). To compute the probability of a precondition is satisfied, distributions in the belief state are adopted.

As declared in Section 3.2, the proposed algorithm can deal with recognition problems with multiple goals. This capability lies in that one explanation is able to account for the simultaneous progresses on multiple goals. As defined in Definition 4.1.2, every goal network in $forest$ reveals an ongoing goal. A $goalNet$ provides the correct next steps and tasks to continue towards the goal. However, the probability of its corresponding explanation affects the final weights of those steps and tasks in the recognition result $PROB$ and the planning result PS. Furthermore, the algorithm can recover from side effects of wrong steps thanks to the carefully designed wrong step handling module in Section 4.7.

The proposed algorithm solves the goal recognition and planning problem defined in this work. The planning process is reflected in the generation of $goalNet(pendingGoalNet)$, which is implemented in Algorithm 12. A $goalNet(tree)$ records the progress status of its goal. $goalNet(pendingGoalNet)$ contains networks obtained through decomposing $goalNet(tree)$. Given an intermediate process towards a goal, there are many ways to move forward. $goalNet(pendingGoalNet)$ explains all those possible ways. Whenever $goalNet(tree)$ changes, $goalNet(pendingGoalNet)$ needs to be reproduced. This is because for different progress points the ways to move forward are different.

An iteration only computes posteriors of occurrences for steps in PS_{step}. The motivation is to reduce computing complexity. This is reasonable since PS_{step} contains necessary steps relating to the context. In the following chapter, a series of experiments are conducted to evaluate the effectiveness and performance of the proposed algorithm.

Chapter 5

Experiments

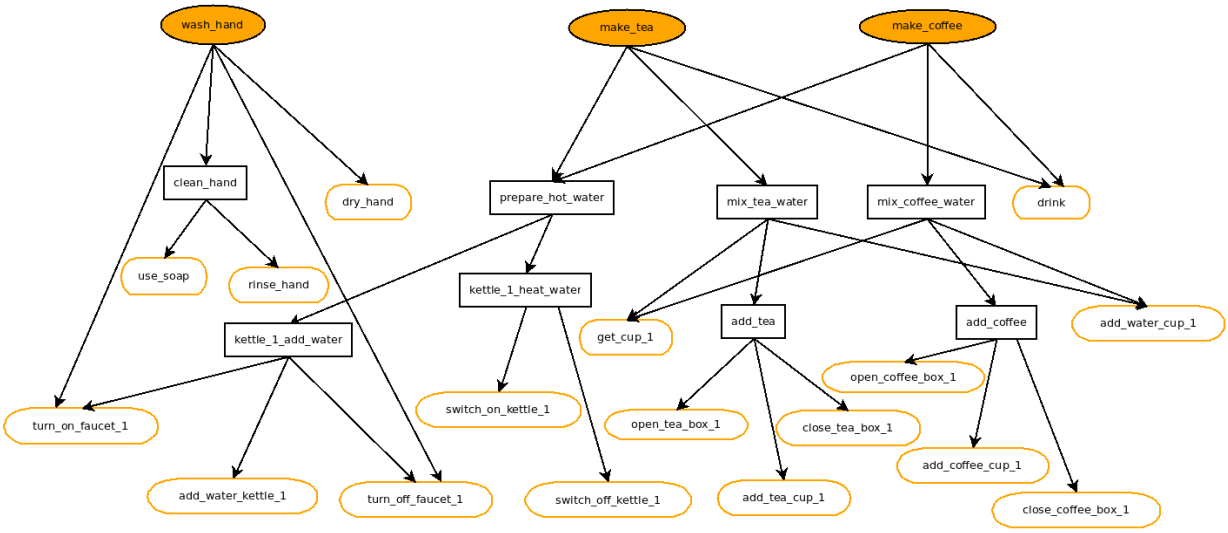
The effectiveness and performance of the proposed hierarchical task recognition algorithm with partial observability is evaluated in this chapter through test cases. The occurring of steps in a test case is simulated by a simulator and no real older adults with cognitive impairments are involved. Section 5.1 describes the scenario and the knowledge base of the experiment study, which relates to three common daily tasks. Section 5.2 presents the simulator, which simulates the changes of real environment. In the third part, Section 5.3, a list of test cases for each problem category is given. In Section 5.4, experiment results on each category of problems are shown with detailed discussions.

5.1 Scenario, Knowledge Base and Sensors

5.1.1 Scenario

Helen is an old woman with mild Alzheimer's disease. She has problems doing three simple daily tasks in the kitchen: washing hands, making a cup of tea, and making a cup of coffee. Her caregiver reports some of her common mistakes. When washing hands, she might forget to use soap or turn the faucet off, or repeatedly rinse her hands. Similar issues happen when making a cup of tea or coffee. The caregiver hopes an intelligent assistance agent can help Helen complete those simple tasks independently.

Figure 5.1: The Hierarchical Task Network for Experiment



5.1.2 Knowledge Base

There are three goals for the scenario above: *wash-hand*, *make-tea* and *make-coffee*. This section shows the domain knowledge ($D = (O, M)$) for these three goals. Although methods and operators in knowledge base are individual pieces, they implicitly indicate a hierarchical plan graph, which is presented in Figure 5.1. In the graph, the root nodes stand for goals G. Leaf nodes are the lowest level steps. Other internal nodes are inner level tasks. Each goal or task node corresponds to a method in M, and each step node corresponds to a step in O.

Figure 5.1 reflects some features of knowledge base. Firstly, different goals might share lower level knowledge. For example, both *make-tea* and *make-coffee* share the task *prepare-hot-water*. Secondly, there might be multiple ways to achieve a goal, depending on the current environment. For example there are two ways to *prepare-hot-water*. If *kettle-1* contains water, only *kettle-1-heat-water* is needed. If *kettle-1* does not contain water, sequential task *kettle-1-add-water* and *kettle-1-heat-water* are needed. However, the order information among subtasks cannot be shown in the graph. One can refer to Appendix A and Appendix B for details of preconditions, subtasks and effects of M and O.

One advantage of the proposed algorithm is that it only explores parts of the network rather than the whole thing. On the one hand, when reasoning about what is going on,

Table 5.1: Sensors Used in the Experiment
(Initial values are in **boldface**)

SensorID	*Obj*	*Att*	*Value*
1	hand_1	soapy	**no**, yes
2	hand_1	dirty	**yes**, no
3	hand_1	dry	**yes**, no
4	faucet_1	state	**on**, yes
5	faucet_1	location	**kitchen**, washroom
6	person_1	location	**kitchen**, washroom
7	person_1	ability	**0.6**, [0, 1]
8	kettle_1	has_water	**no**, yes
9	kettle_ 1	switch	**off**, on
10	kettle_1	water_hot	**no**, yes
11	cup_1	location	**cabinet**, table
12	cup_1	has_water	**no**, yes
13	cup_1	has_tea	**no**, yes
14	cup_1	has_coffee	**no**, yes
15	tea_box_1	location	**table**, cabinet
16	tea_box_1	open	**no**, yes
17	coffee_box_1	location	**table**, cabinet
18	coffee_box_1	open	**no**, yes

only related goals are explored. For example, if the step *switch-on-kettle-1* is observed, only *make-tea* and *make-coffee* will be explored. On the other hand, when generating the possible next steps, only branches with a satisfied precondition will be explored. Those designs contribute to more efficient algorithm by pruning unnecessary search space.

5.1.3 Sensors

According to the experiment scenario and the given knowledge base, virtual binary sensors (Table 5.1) are set up for the sake of simulation. There are 18 sensors altogether. The *Value* column shows all the possible readings of the sensors, with boldface the initial values. Sensor reliability is not shown in the table because it is a variable and will be specified for

different experiment. For easy reference, sensors are referred to by their $SensorIDs$ as in Table 5.1 in this chapter.

Sensor's *obj* and *att* should be consistent with terms in preconditions and effects of methods and operators. For example, $sensor_4$, whose *obj* is *faucet-1* and *att* is *state*, provides measurement for item (*faucet-1*, *state*, *off*), which is the first precondition item in step *turn-on-facuet-1* as shown in Figure 3.2.

$sensor_7$ measures an older adult's ability using a value between [0, 1]. $sensor_7 = 0$ means an older adult has no ability to ADLs at all, while $sensor_7 = 1$ means an older adult has perfect ability to ADLs. Strictly speaking, $sensor_7$ is not a sensor since it is a manual input. In the precondition of a method or operator, there is required ability to execute the corresponding task or step. The algorithm compares the requited ability to the older adult's real ability to determine if he/she has the ability to accomplish the task or step.

5.2 Simulator

The simulator simulates real environment state changes that results from virtually executed steps. The occurring of simulated steps is controlled by the simulator. No real human are involved in the experiment. Whenever a simulated step happens, the simulator firstly simulates the update of real state according to the effects of the step, and then simulates the change of sensor measurements based on the simulated real state and sensor reliability.

5.2.1 Real State Update

The *state* in the simulator is grouped by object. Equation 5.1 stands for the real state of obj_i, which contains obj_i's attributes that matter.

$$State_i = \{obj\_type, obj\_name, obj\_att\_value\_pair[]\} \tag{5.1}$$

When a step happens, *state* is updated following the effects of the step. We assume that whenever a step happens its effects (outcomes) will come true in the real environment. Thus, a simulated step will result in changes in *state* same as effects of the step. For example, the initial state of obj_j is

$$state_j = \{faucet, faucet\_1, [(state, off), (location, kitchen)]\}.$$

If that step *turn-on-faucet-1* (Table B.3) happens, the state would change into

$$state_j = \{faucet, faucet\_1, [(state, \textbf{on}), (location, kitchen)]\},$$

since step *turn-on-faucet-1* has an effect changing $(faucet, state, off)$ to $(faucet, state, on)$.

5.2.2 Sensor Reading Update

The *value* of a sensor depends on the status of the corresponding attribute. For any sensor s, $s(value)$ reports the correct attribute status with probability $s(reliability)$ and reports the opposite wrong status with probability $1 - s(reliability)$. Assume that object j's state is

$$state_j = \{faucet, faucet\_1, [(state, off), (location, kitchen)]\}.$$

Step *turn-on-faucet-1* (Table B.3) happens, changing $(faucet\_1, state)$ from off to on. Note that $sensor_4$ Table 5.1 monitors this attribute and we assume that the reliability of $sensor_4$ is 0.9. In order to update the *value* for $sensor_4$, firstly, the simulator generates a random number in range [0, 1]. Then $sensor_4(value)$ would be set to on if the generated random number falls in range [0, 0.9), otherwise set to off.

5.3 Experiment Test Cases

Each test case is list of steps in the order of execution. It accounts for one single goal or multiple goals. Noisy wrong steps can exist in the list. Given a test case, the simulator simulates step by step changes of objects' states and sensor measurements, with consideration of sensor reliability. The algorithm also reasons about $PROB$ and PS step by step. This section presents test cases for each problem category as shown in Table 5.2. All test cases are based on the knowledge base given in Subsection 5.1.2, which contains three goals: *wash-hand*, *make-tea*, and *make-coffee*. Note that case 4 is missing in Table 5.2. This is because case 4 contains shared steps of goals, where an executed step accounts for multiple goals. Thus it does not belong to any problem category in Table 5.2. It is designed to explain why the proposed algorithm in cannot solve problems with shared steps. Details about this is given in Subsection 5.3.5.

5.3.1 Single Goal Correct Step

Table 5.3 shows correct step sequences for achieving a single goal. They are cases 1-3, aiming at *wash-hand*, *make-tea*, and *make-coffee*, respectively. Goal *wash-hand* is easy to

Table 5.2: Test Cases for Problem Categories

Sensor Config.	Single Goal		Multiple Goals	
	Correct Step	Wrong step	Correct Step	Wrong step
Reliability	Case 1-3	Case 7-10	Case 5-6	Case 11-12
Missing Sensor	Case 1-3	Case 7-10	Case 5-6	Case 11-12

implement, with only five steps. However, *make-tea* and *make-coffee* are more complicated, with 11 total steps each. The first six steps of case 2 and case 3 are the same. They differ from step 7.

Table 5.3: Single Goal Correct Step Case 1-3

Step Num.	Case 1 *wash-hand*	Case 2 *make-tea*	Case 3 *wash-coffee*
1	turn-on-faucet-1	turn-on-faucet-1	turn-on-faucet-1
2	use-soap	add-water-kettle-1	add-water-kettle-1
3	rinse-hand	turn-off-faucet-1	turn-off-faucet-1
4	turn-off-faucet-1	switch-on-kettle-1	switch-on-kettle-1
5	dry-hand	switch-off-kettle-1	switch-off-kettle-1
6		get-cup-1	get-cup-1
7		open-tea-box-1	open-coffee-box-1
8		add-tea-cup-1	add-coffee-cup-1
9		close-tea-box-1	close-coffee-box-1
10		add-water-cup-1	add-water-cup-1
11		drink	drink

5.3.2 Multiple Goals Correct Step

Test cases 5 and 6 in Table 5.4 are cases for multiple goals with correct steps. The steps in those cases indicate that an old adult person works on many goals simultaneously by

switching back and forth. In Table 5.4, steps in **bold** format account for the goal *wash-hand* and steps in normal format account for the goal *make-coffee*. Based on Table 5.4, one can easily figure out how the older adult switch between *wash-hand* and *make-coffee*.

Table 5.4: Multiple Goals Correct Step Case 5-6
(steps for *wash-hand* are in **boldface**)

Step Num.	Case 5 *wash-hand, make-coffee*	Case 6 *wash-hand, make-coffee*
1	**turn-on-faucet-1**	turn-on-faucet-1
2	**use-soap**	add-water-kettle-1
3	**rinse-hand**	turn-off-faucet-1
4	**turn-off-faucet-1**	switch-on-kettle-1
5	turn-on-faucet-1	**turn-on-faucet-1**
6	**dry-hand**	**use-soap**
7	add-water-kettle-1	**rinse-hand**
8	turn-off-faucet-1	**turn-off-faucet-1**
9	switch-on-kettle-1	**dry-hand**
10	switch-off-kettle-1	switch-off-kettle-1
11	get-cup-1	get-cup-1
12	open-coffee-box-1	open-coffee-box-1
13	add-coffee-cup-1	add-coffee-cup-1
14	close-coffee-box-1	close-coffee-box-1
15	add-water-cup-1	add-water-cup-1
16	drink	drink

5.3.3 Single Goal with Wrong Step

Case 7-10 are test cases aiming at a single goal but with wrong steps, which are shown in Table 5.5. Wrong steps in those cases are marked with underlines. As explained in Definition 3.2.1, wrong steps are divided into non-related wrong steps and related wrong steps, depending on whether a wrong step changes the carry-on effects of its previous steps.

Case 7 is a step sequence to achieve the goal *wash-hand*, involving one related wrong step *turn-off-faucet-1*. The older adult *turn-off-faucet-1* after *use-soap*, forgetting to *rinse-*

Table 5.5: Single Goal Wrong Step Case 7-10
(wrong steps have underlines)

Step Num.	Case 7 *wash-hand*	Case 8 *wash-hand*	Case 9 *wash-hand*	Case 10 *make-tea*
1	turn-on-faucet-1	turn-on-faucet-1	turn-on-faucet-1	turn-on-faucet-1
2	use-soap	use-soap	use-soap	turn-off-faucet-1
3	turn-off-faucet-1	use-soap	use-soap	turn-on-faucet-1
4	turn-on-faucet-1	use-soap	turn-off-faucet-1	add-water-kettle-1
5	use-soap	rinse-hand	turn-on-faucet-1	turn-off-faucet-1
6	rinse-hand	turn-off-faucet-1	use-soap	switch-on-kettle-1
7	turn-off-faucet-1	dry-hand	rinse-hand	switch-off-kettle-1
8	dry-hand		rinse-hand	get-cup-1
9			dry-hand	open-tea-box-1
10			turn-off-faucet-1	open-tea-box-1
11				close-tea-box-1
12				open-tea-box-1
13				add-tea-cup-1
14				close-tea-box-1
15				add-water-cup-1
16				drink

hand. This is a related wrong step because it changes the faucet state *on* to *off*, which is a carry on effect of the previous steps. The step sequence in case 8 is also for goal *wash-hand*, containing two non-related wrong steps. The older adult repeats *use-soap* for a long time. Although using soap many times is not correct, the step itself does not have any impact on the carry-on effects of the previous steps. As a result, it will not affect the preconditions of later steps.

Case 9 is for the goal *wash-hand* with two non-related wrong steps (step 3, 8) and one related wrong step (step 4). Step 3 and step 8 repeat their previous step and should not have any side effects. For step 4, the older adult *turn-off-faucet-1* too early without *rinse-hand*. Note that cases 7-9 are sequences to achieve the goal *wash-hand*. In case 7 and 8, the order of the last two steps is *turn-off-faucet-1* and then *dry-hand*. However, in case 9, *dry-hand* is ahead of *turn-off-faucet-1*. They are both correct because these two

Table 5.6: Multiple Goals Wrong Step Case 11-12
(steps for *wash-hand* are in **boldface**, wrong steps have underlines)

Step Num.	Case 11 *wash-hand, make-coffee*	Case 12 *wash-hand, make-tea*
1	**turn-on-faucet-1**	turn-on-faucet-1
2	**use-soap**	add-water-kettle-1
3	**rinse-hand**	turn-off-faucet-1
4	**rinse-hand**	switch-on-kettle-1
5	**turn-off-faucet-1**	**turn-on-faucet-1**
6	turn-on-faucet-1	**turn-off-faucet-1**
7	**dry-hand**	**turn-on-faucet-1**
8	add-water-kettle-1	**use-soap**
9	turn-off-faucet-1	**use-soap**
10	switch-on-kettle-1	**rinse-hand**
11	switch-off-kettle-1	**rinse-hand**
12	get-cup-1	**turn-off-faucet-1**
13	open-coffee-box-1	**dry-hand**
14	add-water-cup-1	switch-off-kettle-1
15	close-coffee-box-1	get-cup-1
16	open-coffee-box-1	open-coffee-box-1
17	add-coffee-cup-1	add-coffee-cup-1
18	close-coffee-box-1	close-coffee-box-1
19	drink	add-water-cup-1
20		drink

steps are unordered steps and either one can be executed first. Case 10 is a step sequence for goal *make-tea*. It contains one non-related wrong step (step 10) and two related wrong steps (steps 2 and 11).

5.3.4 Multiple Goals with Wrong Step

Cases 11 and 12 in Table 5.6 are test cases with multiple goals and wrong steps. Both the two sequences account for the goal *wash-hand* and *make-coffee*. Steps for *wash-hand* are in **bold** format and wrong steps are underlined. In case 11, step 4 is a non-related wrong step relating to the goal *wash-hand*, step 15 is a related wrong step for *make-coffee*. In case 12, there are two non-related wrong steps and 1 related wrong step for *wash-hand*.

5.3.5 Multiple Tasks With Shared Step

Case 4 does not belong to any problem category in Table 3.2 and no experiment is run with it. It is used to explain a limitation of the proposed algorithm. Case 4 provides an execution sequence for achieving goals *wash-hand* and *make-coffee*. These two goals have a shared step which is present in ***bold and italic*** format in Table 5.7. The shared step *turn-on-faucet-1* contributes to both *make-coffee* followed by step 2, and *wash-hand* followed by step 3. When comparing case 4 with case 5 in Table 5.4, one can see that in case 5 *turn-on-faucet-1* is done twice, one for *wash-hand*, another for *make-coffee*. However in case 4, *turn-on-faucet-1* is done once but accounts for two goals, which is a shared step.

The algorithm proposed in this paper can only assign one step to one goal. So *turn-on-faucet-1* in case 4 is assigned to either goal *make-coffee* or goal *wash-hand*, but not to both. If it is assigned to *make-coffee*, the start step for *wash-hand* never happens. As a result, the algorithm recognizes step 3 *use-soap* as a wrong step. Consequently, the proposed algorithm cannot handle shared steps between goals.

Table 5.7: Multiple Tasks With Shared Correct Step Case 4
(the shared step is in **boldface**)

Step Num.	Case 4
1	***turn-on-faucet-1***
2	add-water-kettle-1
3	use-soap
4	rinse-hand
5	turn-off-faucet-1
6	dry-hand
7	switch-on-kettle-1
8	switch-off-kettle-1
9	get-cup-1
10	open-coffee-box-1
11	add-coffee-cup-1
12	close-coffee-box-1
13	add-water-cup-1
14	drink

5.3.6 Sensor Missing Cases

Sensor missing cases are designed to evaluate the algorithm's robustness to missing sensors and to figure out what kinds of sensors are crucial for goal recognition. My assumption is that the importance of a sensor strongly depends on its related steps. If the attribute that a sensor measures could be affected by a step which **starts** a goal, the sensor is important. If the attribute that a sensor measures could be affected by a step who has **multiple effects** (the step affects multiple attributes), the sensor is not very crucial. Based on this assumption, four sensor missing categories are given in Table 5.8. The category name is simply taking the first letter of the property. For example "S-E-S-M" stands for "Single Effect Start step sensor Missing".

Table 5.8: Sensor Missing Category

Category Name	Single Effect Step	Multiple Effects Step
Start Step	S-E-S-M	M-E-S-M
Non-start Step	S-E-N-S-M	M-E-N-S-M

Table 5.9: Sensor Missing Cases
(**boldface** decimals are sensor reliabilities)

Missing Sensor ID	Explanation	Category	Sensor Missing Cases Name	
			Other Sensor Reliability: **0.9**	Other Sensor Reliability: **0.8**
4	faucet-1, state	S-E-S-M	M-1	M-7
2	hand-1, dirty	M-E-N-S-M	M-2	M-8
10	kettle-1, water-hot	M-E-S-M	M-3	M-9
9	kettle-1, switch	M-E-S-M	M-4	M-10
8	kettle-1, has-water	S-E-N-S-M, M-E-N-S-M	M-5	M-11
13	cup-1, has-tea	S-E-N-S-M	M-6	M-12

With sensor missing category defined in Table 5.8, 12 cases with single sensor missing are designed by changing the missing sensor and the reliability of other sensors. In order

to distinguish between the test cases defined before, missing sensor cases start with capital letter "M". In Table 5.8, the "Missing Sensor ID" column shows the sensor ID indicated in Table 5.1. Note that $sensor_8$ belongs to two categories. This is because it relates to two steps, one is a single effect non-start step (*add-water-kettle-1*), and another is a multiple effects non-start step (*add-water-cup-1*).

5.3.7 Desired Output for Test Cases

The proposed algorithm aims to solve goal recognition and planning problem defined in Definition 3.1.9. So the output for each simulated step should include two parts: the goal distribution $PROB$ and the correct next tasks or steps PS. The desired output for each simulated step is as follows.

- In $PROB$, the ongoing goals should have higher probabilities of happening than others after one or two steps.
- In PS, the correct next steps should have higher probabilities than the others.
- In PS, the correct next composite tasks in inner levels should have higher probabilities than the others.
- For test cases of multiple goals, the algorithm should correctly assign each step to the correct goal.
- For problems with wrong steps, the algorithm should report both related and non-related wrong steps.
- For detected related wrong steps, the algorithm should be able to correctly target to the affected goals and rectify their ongoing status so as to give correct hint of next steps.

5.4 Experiment Results

This section presents experiment results for all the problem categories with detailed discussions. It includes three parts: performance evaluation criteria, results and discussions of each problem category, and a summary to the experiment section.

5.4.1 Performance Evaluation Criteria

The proposed algorithm recognizes the older adult's intents based on observations and gives proper hints when necessary. Note that hints are not necessarily of the lowest step level. It can be inner level composite tasks. For example in case 1 (Table 5.3), the algorithm believes that the older adult is trying to *wash-hand* after step *use-soap*. The hint can be the highest level goal *wash-hand*, or the intermediate level task *clean-hand*, or the lowest level step *rinse-hand*.

Each iteration should produce correct $PROB$ and PS. Given a test case, it is easy to decide if $PROB$ is correct or not. However PS can be partially correct when only some of its levels are correct. It is quite likely that the hint in a higher level is correct while it is wrong in the lowest level. However, if the hint in the lowest step level is correct, PS must be correct. To simplify evaluation, we measure the performance of PS in a strict way. PS is correct only when its lowest step level is correct. For cases with wrong steps, this criterion is also applicable because if the hint in the step level is correct, the algorithm must have reported and repaired the wrong step.

Note that to help an older adult with a cognitive impairment, recognizing his/her intent and providing proper hints are equally important. Thus, when measuring performance, the goal recognition result $PROB$ and the planning result PS are considered with equal weights. Assume that the number of steps in a test case is N, the number of iterations that the algorithm correctly recognizes the ongoing goals is $PROB_C$ ($PROB$ correct), and the number of iterations that the algorithm provides the correct hint is PS_C (PS correct). The performance is computed using Equation 5.2. Because of the strict criterion on PS, the obtained performance using Equation 5.2 is worse than the real performance of the algorithm. However, if the performance is acceptable under this evaluation, the real performance of the algorithm must be satisfying.

$$Performance = \frac{0.5 \times PROB_C + 0.5 \times PS_C}{N} \times 100\% \tag{5.2}$$

In the following subsections, the performance on each problem category is presented one by one. Sensor reliability has four values $[0.99, 0.95, 0.90, 0.80]$. For instance, sensor reliability 0.8 means that all the 18 sensors in Table 5.1 have reliability 0.8. Sensor missing has 12 cases M1-M12. Each test case is run under a specific sensor configuration for 20 times. The average performance is computed as the final performance.

Each algorithm iteration computes the goal recognition result $PROB$ and planning result PS (the correct next tasks and steps) based on explanations stored in the current

$ExplaSet$ (refer to Section 4.1). In general, the number of explanations ($Expla_{num}$) that can explain the observation series so far $obs = [obs_1, obs_2, obs_3, ...]$ should not be large. During reasoning, many incorrect explanations with relatively low probabilities are generated due to the partial observability. A large $Expla_num$ indicates that many noisy explanations exist which can consequently affect the correctness and perfectness of $PROB$ and PS. When an iteration updates $ExplaSet$ (refer Section 4.2), it reasons about every explanation in $ExplaSet$. To avoid too much calculation, explanations with probability smaller than **0.001** are removed when running experiments.

5.4.2 Results on Test Cases with Changing Sensor Reliabilities

The average accuracies of all the test cases with changing reliabilities is presented in Table 5.10, for which we conclude:

- The performances positively correlate with sensor reliabilities. When sensor reliabilities reduce, the average accuracies of test cases deteriorate as well.
- The easiest problem category **p1**, which targets problems with single goal and correct steps, has the best performance. The average accuracies are very high even when sensor reliabilities are only 0.8.
- The hardest problem category **p4**, which targets problems with multiple goals and wrong steps, has the worst performance. The accuracies are acceptable only when sensor reliabilities are above 0.95. This result is reasonable since the algorithm has to deal with noisy sensors, multiple goals and wrong steps.
- The other two categories, **p2** and **p3**, have similar performances, which are acceptable when sensor reliabilities are above 0.9.
- When the sensor reliabilities are above 0.95, the average accuracies of all the test cases are very high.
- Since the methods in the knowledge base contains unordered subtasks, the results also indicate that the algorithm is capable of dealing with unordered steps and tasks.

The results in Table 5.10 demonstrate the proposed algorithm's capacity to solve the goal recognition and planning problem described in Chapter 3. Our algorithm can efficiently handle issues including partial observability, wrong steps, unordered steps, and simultaneous goals.

Table 5.10: Average Performances on Test Cases
(**boldface** decimals are sensor reliabilities)

Case Num.	**0.99**	**0.95**	**0.90**	**0.80**
Case 1	100%	97%	95%	93%
Case 2	100%	99%	99%	97%
Case 3	100%	100%	98%	98%
Case 5	99%	99%	90%	79%
Case 6	100%	99%	93%	86%
Case 7	100%	98%	93%	**44%**
Case 8	100%	99%	98%	96%
Case 9	100%	96%	94%	**59%**
Case 10	100%	92%	83%	**62%**
Case 11	100%	90%	70%	66%
Case 12	100%	94%	79%	69%

Figure 5.2: The *PROB* Output for Case 1 (*wash-hand*)

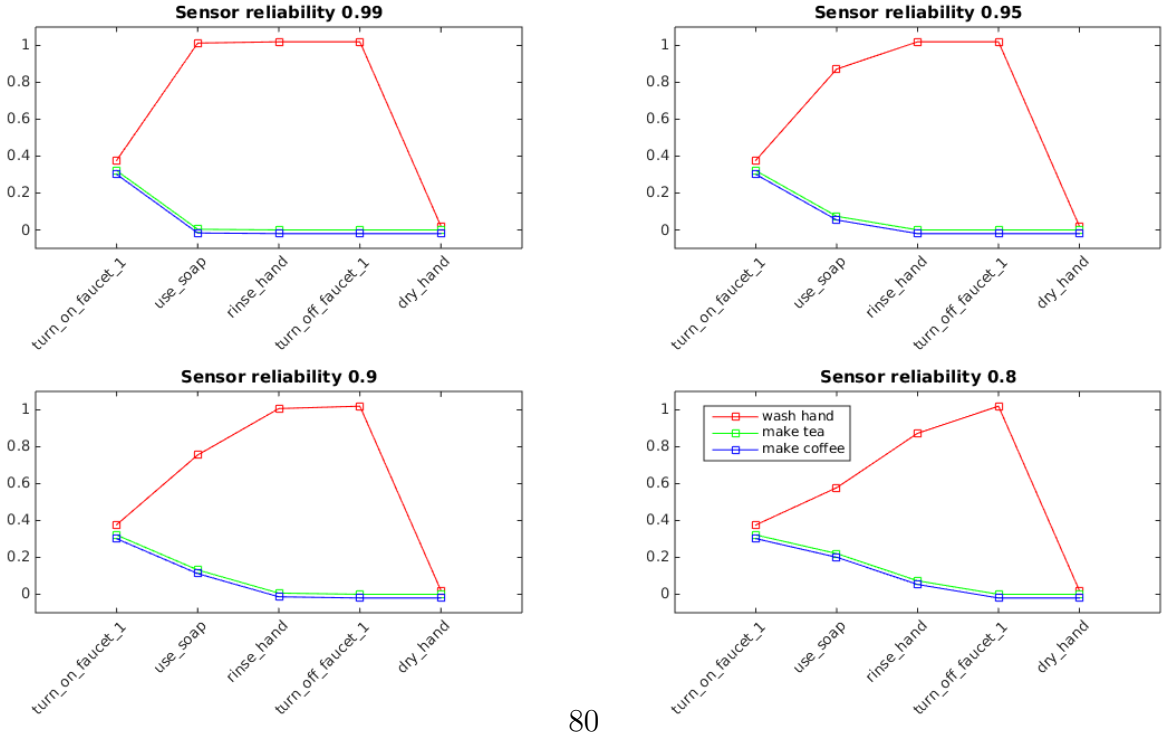


Figure 5.3: The *PROB* Output for Case 2 (*make-tea*)

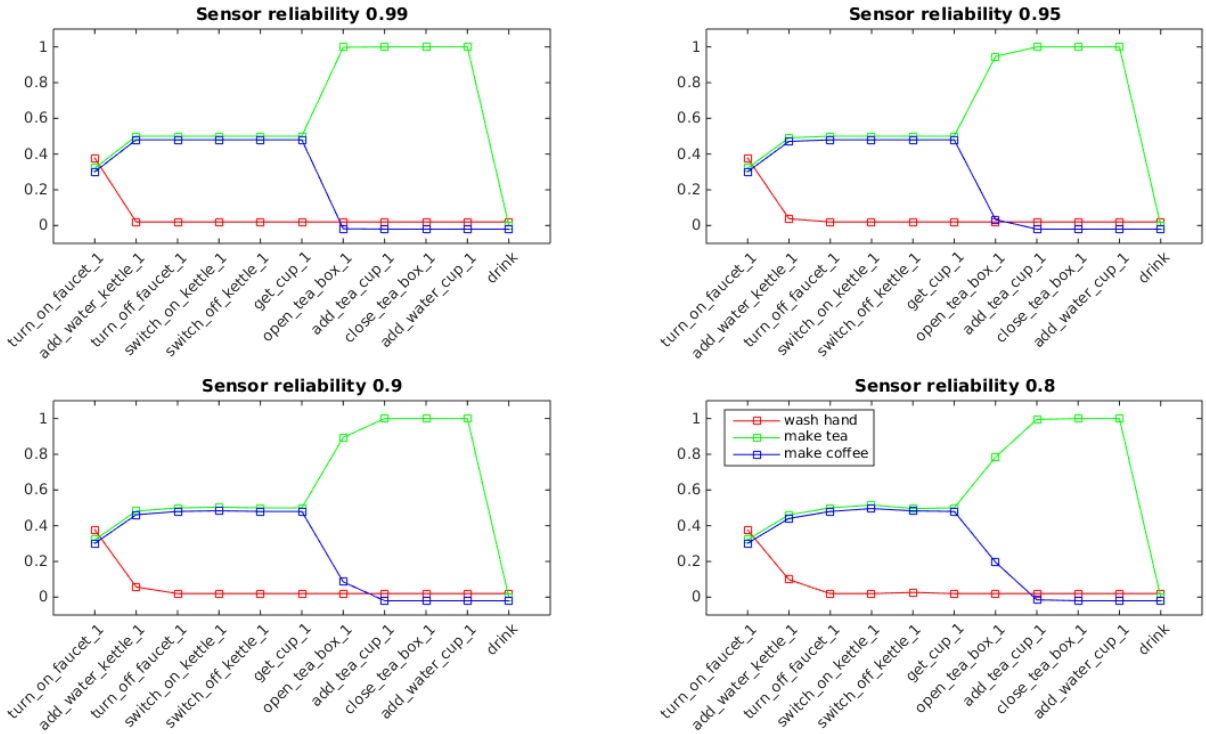


Figure 5.4: The *PROB* Output for Case 5 (*wash-hand*, *make-coffee*)

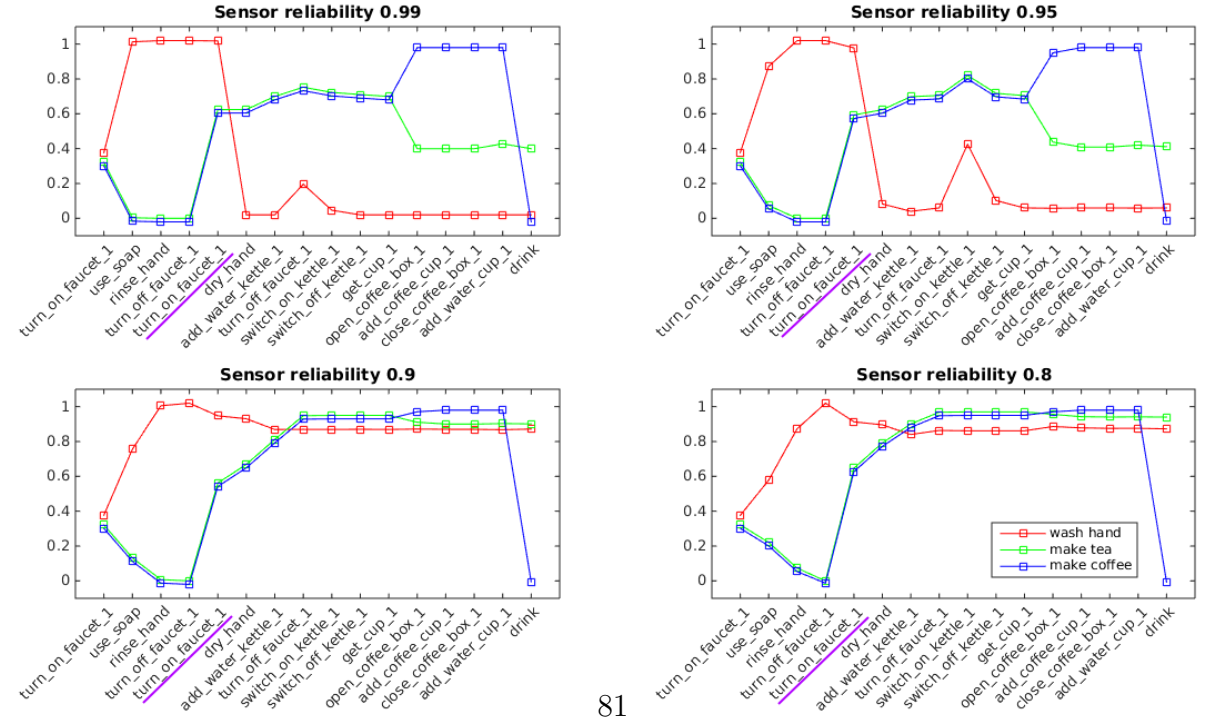


Figure 5.5: The *PROB* Output for Case 9 (*wash-hand*)

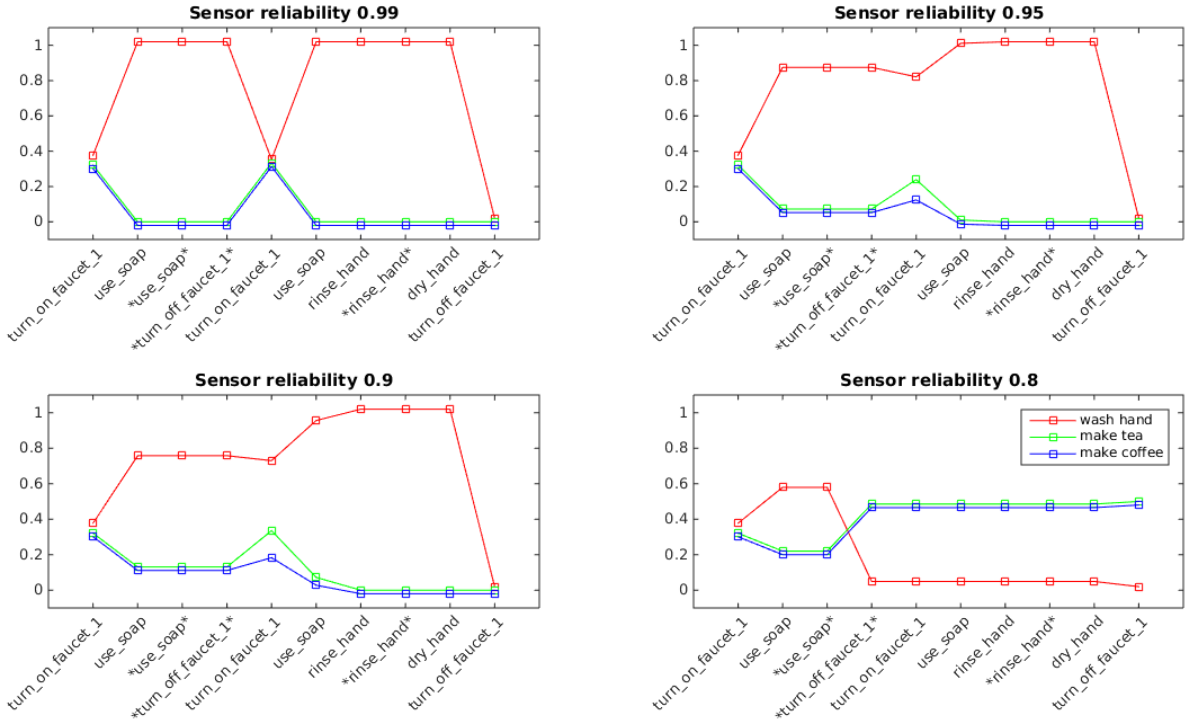
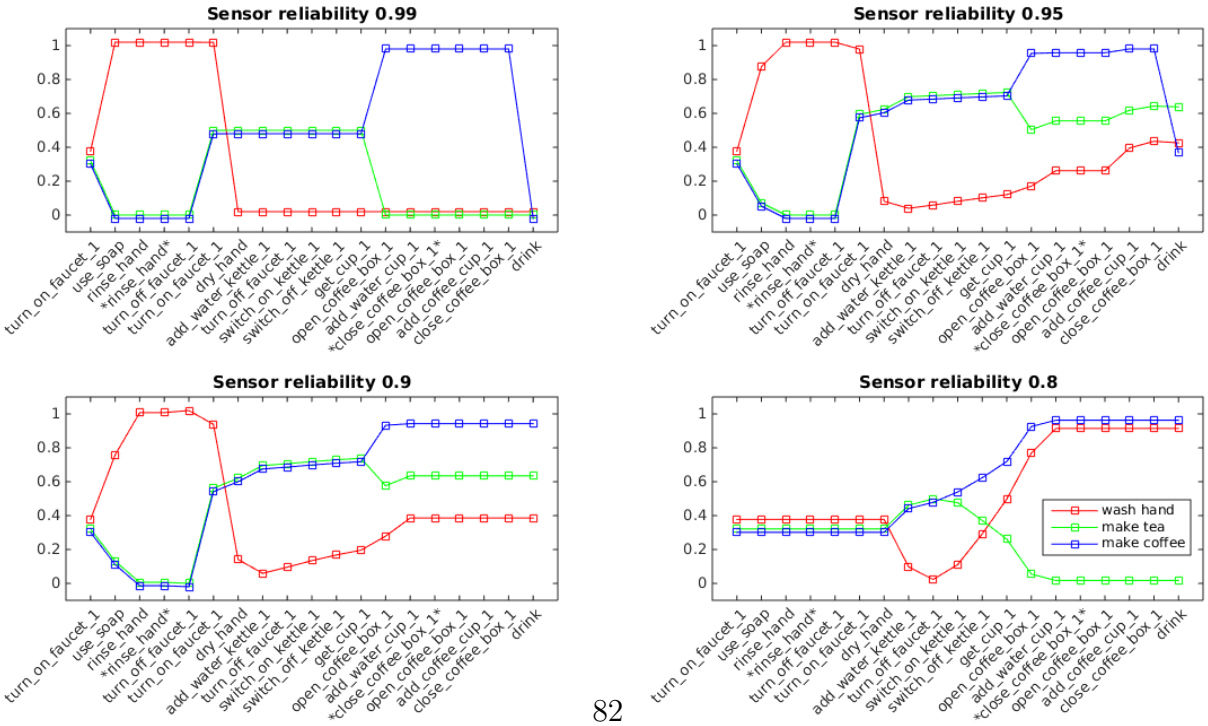


Figure 5.6: The *PROB* Output for Case 11 (*wash-hand*, *make-coffee*)



The Influence of Sensor Reliabilities on $PROB$

In order to see how sensor reliabilities affect reasoning, we pick the $PROB$ and PS_{step} outputs of case 1, case 2, case 5, case 9, and case 11 in one time run. The $PROB$ distributions after each step of those cases are shown in Figure 5.2, Figure 5.3, Figure 5.4, Figure 5.5, and Figure 5.6, respectively. Their corresponding PS_{step} outputs are shown in Appendix C start from page 126. According to those figures, we have the following conclusions.

- The convergence of $PROB$ is correlated with sensor reliability. In those figures, when the sensor reliability is 0.99, $PROB$ changes quickly. For example, the probability of *wash-hand* in Figure 5.2 with sensor reliability 0.99 jumps to 0.99 in the second step. When the sensor reliability reduces, $PROB$ becomes smoother. It indicates that the algorithm updates $PROB$ gradually with more observations.

- The probabilities of ongoing goals outweigh the probabilities of non-happening goals after the second or third steps. The algorithm can correctly recognize the ongoing goals very quickly.

- The probabilities of goal *make-tea* and *make-coffee* align with each other until step *get-cup-1* in case 2, case 5 and case 11. By referring to the knowledge base in Subsection 5.1.2, one can see *make-tea* and *make-coffee* have the same step sequence before *get-cup-1*. Furthermore, the priors of goals are set to equal. That's why the probabilities of those two goals align with each other. The alignments in plots for case 1 and case 9 also come from the even distribution of goal priors.

- The probability of a goal drops to 0.0 when it is finished. According to all those figures, the probabilities of ongoing goals reach to 1.0 after sufficient observations and drop to 0.0 when they are finished. This is reasonable since a finished goal should not be regarded as an ongoing one.

- For case 5 and case 11, which have multiple goals, the probabilities of non-happening goals in $PROB$ are very high when the sensor reliabilities are less than 0.9. Those include plots in Figure 5.4 with sensor reliability 0.9 and 0.8 and plots in Figure 5.6 with sensor reliability 0.95, 0.9, and 0.8. Such unsatisfying $PROB$ distributions indicate significant noises caused by low sensor reliabilities and multiple goals.

Further explanation for Figure 5.4. In case 5 the first 4 steps are for goal *wash-hand*. No matter what the sensor reliability is, the probability of *wash-hand* reaches to 1.0

after step 4 *turn-off-faucet-1*, while the probabilities of *make-tea* and *make-coffee* reduce to 0.0. After step 5 *turn-on-faucet-1*, which is for *make-tea* or *make-coffee*, the probability of *wash-hand* stays the same. The probabilities of *make-tea* and *make-coffee* jump to above 0.5. This distribution indicates that the algorithm believes *wash-hand* and *make-tea* (or *make-coffee*) are in progress. The curve of *make-tea* cannot distinguish from that of *make-coffee* because the first six steps of those two goals are the same. When referring to Table 5.11 on page 84, it turns out this uncertainty does not affect PS_{step} output because both *dry-hand* and *add-water-kettle-1* have very high probability in step 5. The probability of *add-water-kettle-1* is high because *add-water-kettle-1* is the desired next step of both *make-tea* and *make-coffee*.

Table 5.11: Pending Set for Case 5 with Sensor Reliability 0.90
(The First 8 steps)

Step Num.	Step Name	PS_{step} for This Step
1	turn-on-faucet-1	add-water-kettle-1: 0.6426 use-soap: 0.3574
2	use-soap	turn-off-faucet-1: 0.2644 rinse-hand: 0.7356
3	rinse-hand	turn-off-faucet-1: 0.9853 switch-on-kettle-1: 0.0147 dry-hand: 0.9853
4	turn-off-faucet-1	turn-off-faucet-1: 0.4253 dry-hand: 0.5747
5	turn-on-faucet-1	add-water-kettle-1: 0.9274 dry-hand: 0.9274
6	dry-hand	turn-off-faucet-1: 0.9107 add-water-kettle-1: 0.4079 dry-hand: 0.9107
7	add-water-kettle-1	turn-off-faucet-1: 0.5765 switch-on-kettle-1: 0.4235 add-water-kettle-1: 0.1987 dry-hand: 0.8489
8	turn-off-faucet-1	turn-off-faucet-1: 0.8461 switch-on-kettle-1: 0.9964 add-water-kettle-1: 0.0503 switch-off-kettle-1: 0.0036 dry-hand: 0.8497

In Figure 5.4, when the sensor reliabilities are 0.9 or 0.8, the probability of *wash-hand* does not drop much after step *dry-hand*. It indicates that the algorithm fails to recognize the step *dry-hand*. The probability of *wash-hand* in the following steps is maintained. The PS_{step} output (Table 5.11, page 84) also explains this mistake. After step 6, *dry-hand* is always presented in the PS_{step}. Furthermore with sensor reliability 0.9 or 0.8, after step *open-coffee-box-1*, although the probability of *make-coffee* is the highest, the probability of *make-tea* is still very high. The high probability of *make-tea* comes from the sum of probabilities of noisy explanations whose ongoing goals contain *make-tea*.

Further explanation for Figure 5.5. Case 9 contains wrong steps which are marked with * in Figure 5.5. As one can see, $PROB$ does not change when a wrong step happens. This is because when a wrong step is recognized, the algorithm repairs the $forest$ of an existing explanation rather than changing its probability. Evidences of successfully dealing with those wrong steps can be found in Table 5.12, which shows the PS_{step} output of case 9 in one time run with reliability 0.9. Step 3 and 8 are two non-related steps, and the algorithm gives the PS_{step} prompts same as the ones of the last step. Similarly, step 7 and step 8 have the same PS_{step}. Step 4 is a related wrong step which violates the effect (*faucet-1, state, on*). The algorithm repairs this wrong step and suggests the older adult to execute *turn-on-faucet-1* again.

When sensor reliability is 0.8 (the forth plot in Figure 5.5), the algorithm recognizes *make-tea* & *make-coffee* as the ongoing goals, which is wrong. As the plot shows that the algorithm gets lost after the wrong step *turn-off-faucet-1*. With sensor reliability 0.8, the probabilities of *make-tea* and *make-coffee* are 0.22 after step 3 *use-soap*. This contributes to the high prior for *turn-off-faucet-1* after *use-soap*, which is about 0.45. Consequently the algorithm believes that the wrong step *turn-off-faucet-1* is correct and votes *make-tea* or *make-coffee* as the ongoing goals. After step 4, the algorithm always detects the correct steps as wrong steps and never recovers.

Further explanation for Figure 5.6. The $PROB$ distribution with sensor reliability 0.99 is perfect. With sensor 0.95, the probabilities change in the correct direction with noise. As the plot shows, the noise is significant since the probabilities of non-happening goals, *wash-hand* and *make-tea*, keep going up in the later steps of the test case, reaching to 0.4 and 0.6, respectively. However, the algorithm still works because the probabilities of the correct ongoing goals outweigh the probabilities of the other goals. For sensor reliability 0.9 and 0.8, one can easily see the horizontally straight lines in the later steps. It means the algorithm gets lost and does not update the explanations any more. That's why

Table 5.12: Pending Set for Case 9 with Sensor Reliability 0.90

Step Num.	Step Name	PS_{step} for This Step	
1	turn-on-faucet-1	add-water-kettle-1:	0.6426
		use-soap:	0.3574
2	use-soap	turn-off-faucet-1:	0.2621
		rinse-hand:	0.7379
3	use-soap	turn-off-faucet-1:	0.2621
		rinse-hand:	0.7379
4	turn-off-faucet-1	turn-on-faucet-1:	1.0
5	turn-on-faucet-1	add-water-kettle-1:	0.4779
		use-soap:	0.71
6	use-soap	turn-off-faucet-1:	0.0981
		add-water-kettle-1:	0.0221
		rinse-hand:	0.9019
		use-soap:	0.0342
7	rinse-hand	turn-off-faucet-1:	1.0
		dry-hand:	1.0
8	rinse-hand	turn-off-faucet-1:	1.0
		dry-hand:	1.0
9	dry-hand	turn-off-faucet-1:	0.8999
		dry-hand:	0.1001
10	turn-off-faucet-1		

performances with sensor 0.9 and 0.8 are very bad. In this test case improper priors and wrong steps cause the algorithm's failure. Detailed explanations are given in the following sub-subsections. The experiment result on case 11 shows that the algorithm's ability to tolerate sensor noises is quite limited when dealing with problems with multiple goals and wrong steps. If the sensors are reliable enough (with reliability above 0.95), the algorithm can work very well.

Table 5.13: Pending Set for Case 1 with Sensor Reliability 0.90

Step Num.	Step Name	PS_{step} for This Step
1	turn-on-faucet-1	add-water-kettle-1: 0.6426 use-soap: 0.3574
2	use-soap	turn-off-faucet-1: 0.2644 rinse-hand: 0.7356
3	rinse-hand	turn-off-faucet-1: 0.9871 switch-on-kettle-1: 0.0129 dry-hand: 0.9871
4	turn-off-faucet-1	turn-off-faucet-1: 0.1011 dry-hand: 0.8989
5	dry-hand	

The Influence of Priors and Wrong Sensor Readings on $PROB$ and PS

According to Table 5.3, test case 1 is much simpler than case 2 and 3. But the overall performance on case 1 shown in Table 5.10 is not as good as that on case 2 and 3. By looking into the wrong recognition outputs of case 1, it turns out error occurs when related sensors of step 2 *use-soap* didn't give the proper notification. Whenever this happens, the PS_{step} after *use-soap* is

$$PS_{step} = [\textit{rinse-hand} : 0.2367, \textit{turn-off-faucet-1} : 0.7633].$$

The $PROB$ after *use-soap* is

$$PROB = [\textit{wash-hand} : 0.2367, \textit{make-tea} : 0.3817, \textit{make-coffee} : 0.3817].$$

This result is regarded as wrong because the goal *wash-hand* and the next step *rinse-hand* are supposed to have the highest probabilities.

The wrong output comes from the algorithm's incorrect belief that the second step that happened is *add-water-kettle-1*. However, the incorrect belief is reasonable for the following reasons. Firstly, although *use-soap* happens, its related sensor didn't present the proper notification; Secondly, the prior of *add-water-kettle-1* is much higher than *use-soap*. When referring to Table 5.13 in page 87, which shows PS_{step} of case 1 in one time run with sensor reliability 0.9, one can see after step *turn-on-faucet-1*, PS_{step} is

$$[\textit{add-water-kettle-1} : 0.6426, \textit{use-soap} : 0.3574].$$

After *turn-on-faucet-1*, each goal has a probability of about 0.333 since the priors of the three goals are set to equal. Furthermore, the desired next step for both *make-tea* and *make-coffee* are *add-water-kettle-1*. That's why *add-water-kettle-1* has a higher prior after *turn-on-faucet-1*. The missing notification and the high prior of *add-water-kettle-1* lead to the higher posterior of *add-water-kettle-1*, which convinces the algorithm to believe that *add-water-kettle-1* has happened.

The Influence of Wrong Steps on Accuracies

According to Table 5.10, the accuracies of case 7, 9 and 10 with sensor reliability 0.8 are extremely bad. The main reason for the bad performances is the **related wrong steps** in those three cases. With sensor reliability 0.8, the sensors are more likely to give wrong measurements, making the algorithm mistakenly report a correct step as wrong, or vice versa. Once a wrong step is reported mistakenly, the algorithm will repair the tree structure to a worse status. As a result, no matter how correct the later measurements are, the algorithm keeps giving wrong step reports in the later steps and can not go back to the correct direction. That's why the accuracies on case 7, 9, and 10 with sensor reliability 0.8 are below 60%. The bad performances on case 11 and case 12 with sensor reliability 0.9 and 0.8 have the similar reason.

In the contrast, the accuracies of case 8, which only contains **non-related wrong steps**, are very good with any sensor reliability. The result proves that related wrong steps create much more noises than non-related ones, which aligns with our assumptions in the problem description section.

The $Expla_{num}$ of Different Problem Categories

As mentioned before, the number of explanations is also an important measure of the reasoning result. The more explanations, the more noisy the algorithm. Too many noisy explanations might drag the algorithm away from correct reasoning. The average $Expla_{num}$ of case 2, case 9, case 5 and case 11 are shown in Figure 5.7a, Figure 5.7b, Figure 5.7c, and Figure 5.7d, respectively. According to those figures, we have the following observations.

- In General, the average $Expla_{num}$ increases when sensor reliability decreases. This indicates that low sensor reliability leads to more noises to the recognition process.
- For test cases with wrong steps, case 9 in Figure 5.7b and case 11 Figure 5.7d, a higher sensor reliability does not guarantee a smaller number explanations. For example,

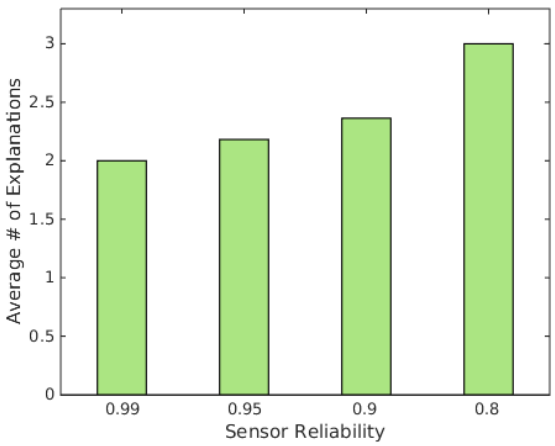
in Figure 5.7b, the average $Expla_{num}$ with sensor reliability 0.8 is smaller than that with sensor reliability 0.9. This is because the algorithm gets lost in some step and keeps reporting wrong steps for the later steps without adding new explanations.

- For recognition problems with multiple goals, the algorithm generates more noisy explanations than the ones with a single goal. Case 2 and case 9 are test cases with only one goal. Case 5 and case 11 are test cases with two goals. As shown in Figure 5.7, the average $Expla_{num}$ of case 5 and case 11 reach to about 30, which are much more than those of case 2 and case 9. Despite many noisy explanations, the algorithm still works because usually noisy explanations have very small probabilities.

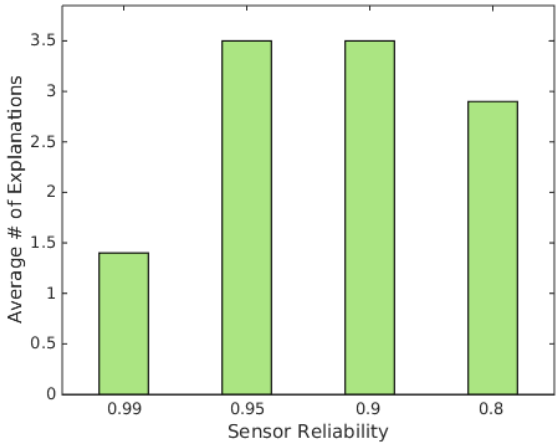
- Wrong steps slightly increase the number of explanations. In Figure 5.7, case 9 and case 11 contain wrong steps. Their explanation numbers are slightly bigger than those of case 2 and case 5, which do not have wrong steps. Whenever a wrong step is detected, the algorithm will repair existing explanations rather than creating new ones. Thus wrong steps do not increase the number of explanation to a large degree.

- By referring to Figure 5.7 and the accuracy results in Table 5.10, we can concludes that a large number of explanations lead to low accuracies. When the correct explanations cannot compete with the noisy ones, the algorithm will present the wrong reasoning result.

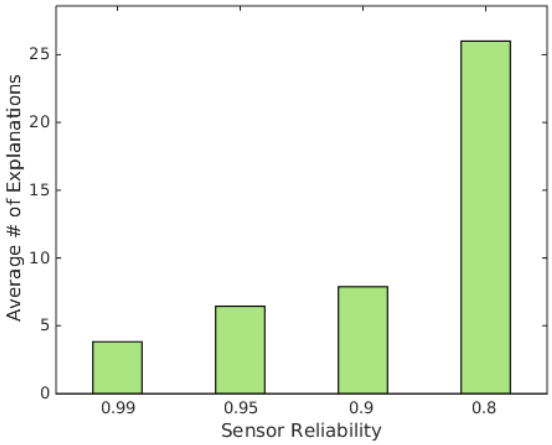
- Note that explanations with probability smaller than 0.001 are deleted during the experiments. When this parameter is 0.00001, the number of explanations for test case 11 reaches to 430. We delete noisy explanations with small probabilities to save computation time.



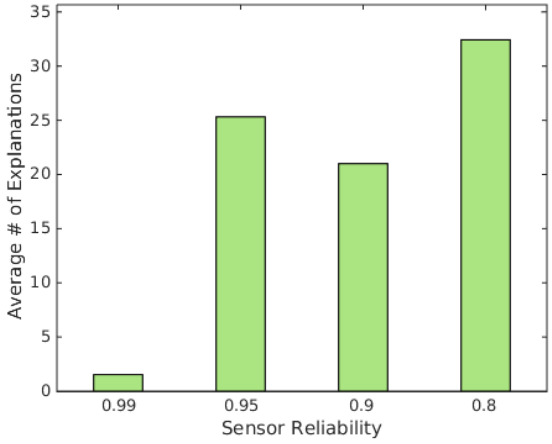
(a) Case 2



(b) Case 9



(c) Case 5



(d) Case 11

Figure 5.7: Explanation Numbers with Different Sensor Reliabilities

5.4.3 Results on All Test Cases with Sensor Missing

This section presents the algorithm's performance on all the test cases with missing sensors. Table 5.14, Table 5.15, Table 5.16 and Table 5.17 are the experiment results on the four categories of problems with missing sensor cases, which are p5-p8. We obtained the following conclusions.

- Compared with experiment results in Table 5.10, the overall performances with sensor missing are not as good as those without sensor missing. The difference is not significant for problems with single goal and correct steps. However, for the others, the performances deteriorate greatly when there is a missing sensor.
- Performances on problems with single goal and correct steps (Table 5.14) are the best. The problems with multiple goals and wrong steps have the worst performances (Table 5.17). Accuracies on problems with single goal and wrong steps (Table 5.16) are higher than those on problems with multiple goals and correct steps (Table 5.15). This means with missing sensors, the algorithm can hardly distinguish among different goals. This is reasonable since without sensor measurements, the algorithm can not prune the non-happening goals.
- Performances on M1-M6 are better than those on M7-M12. It indicates that higher reliabilities of non-missing sensors contribute to higher accuracies. For problems **p6**, **p7**, and **p8**, the accuracies with non-missing sensors' reliabilities 0.8 (M7-M12) are not acceptable.
- Missing sensor category "S-E-S-M" and "M-E-S-M", which match to M1 and M4, have relatively worse accuracy, such as [M1, case 5]. This means missing sensors relating to a start step of a goal have profounder impacts on the algorithm performance than the other categories of sensor missing.
- According to results in Table 5.14, performances on problem category **p5** with sensor missing cases M1-M6 is almost the same as those with sensor missing cases M7-M12. This means for problems with single goal and correct steps, the performance is not sensitive with sensors reliabilities.

Table 5.14: Average Performance on Case 1-3 with Sensor Missing

Case Num.	M1	M2	M3	M4	M5	M6
1	100%	94%	94%	95%	**85%**	97%
2	98%	99%	97%	99%	99%	99%
3	100%	100%	100%	99%	98%	100%
Case Num.	M7	M8	M9	M10	M11	M12
1	93%	91%	95%	95%	**73%**	97%
2	98%	97%	97%	99%	96%	98%
3	99%	99%	98%	100%	97%	99%

Table 5.15: Average Performance on Case 5-6 with Sensor Missing

Case Num.	M1	M2	M3	M4	M5	M6
5	77%	92%	92%	84%	83%	93%
6	82%	89%	93%	93%	90%	91%
Case Num.	M7	M8	M9	M10	M11	M12
5	67%	75%	77%	75%	57%	74%
6	82%	85%	90%	84%	89%	86%

Table 5.16: Average Performance on Case 7-10 with Sensor Missing

Case Num.	M1	M2	M3	M4	M5	M6
7	56%	94%	91%	96%	39%	95%
8	99%	100%	98%	98%	77%	98%
9	55%	93%	96%	97%	32%	97%
10	90%	85%	70%	67%	81%	71%
Case Num.	M7	M8	M9	M10	M11	M12
7	56%	50%	57%	57%	35%	59%
8	45%	98%	98%	94%	61%	94%
9	55%	53%	51%	60%	38%	53%
10	69%	61%	58%	49%	75%	71%

Table 5.17: Average Performance on Case 11-12 with Sensor Missing

Case Num.	M1	M2	M3	M4	M5	M6
11	75%	67%	61%	74%	57%	78%
12	82%	89%	83%	62%	83%	76%
Case Num.	M7	M8	M9	M10	M11	M12
11	71%	72%	66%	65%	42%	63%
12	69%	63%	61%	64%	70%	68%
Case Num.	M13	M14	M15	M16	M17	M18
11	71%	87%	87%	89%	82%	87%
12	86%	90%	85%	67%	91%	90%

Further explanation for Table 5.14. Generally speaking, the accuracies for case 1-3 are very high, except cases [M5, case 1] and [M11, case 1], which are shown in a **bold** format. In M5 and M11, $sensor_8$, which monitors attribute (*kettle-1, has-water*), is missing. As a result, the algorithm cannot obtain any information about the happening of step *add-water-kettle-1*. Table 5.18 is used to explain the non-satisfying performance on case 1 with M5 and M11. After step *turn-on-faucet-1*, the prior for *add-water-kettle-1* is about 0.6426, much higher than that of *use-soap*. Since the observations of *add-water-kettle-1* is missing, the algorithm can not prune the likelihood that *add-water-kettle-1* has happened. Due to the high prior, the algorithm prunes to believe that the second step that happened is *add-water-kettle-1*.

The upper part of Table 5.18 shows the $PROB$ and PS_{step} after the step *use-soap* when the related sensors of *use-soap* present correct measurements. After step *use-soap*, the probability of *wash-hand* improves a little, and the probabilities of the other two goals reduce a little, which are correct. However, the PS_{step} is wrong since *rinse-hand* is supposed to have a higher probability than *turn-off-faucet-1*. This indicates that the correct sensor measurements from *use-soap* cannot compete with the high prior of *add-water-kettle-1* when missing $sensor_8$. The lower part of Table 5.18 shows the $PROB$ and PS_{step} after step *use-soap* when the related sensors of *use-soap* failed to present correct measurements. In this case, the algorithm is more confident to believe that *add-water-kettle-1* has happened, which contribute to goal *make-tea* or *make-coffee*. Consequently, the probability of *wash-hand* reduces to 0.0613, while the probabilities of *make-tea* and *make-coffee* increase to 0.4693, which is totally wrong.

Table 5.18: Step output for case 1 with M5

Step Name	wash-hand	make-tea	make-coffee	Pending Set
turn-on-faucet-1	0.3574	0.3213	0.3213	use-soap: 0.3574 add-water-kettle-1: 0.6426
use-soap	0.3701	0.315	0.315	turn-off-faucet-1: 0.6299 rinse-hand: 0.3701
Step Name	wash-hand	make-tea	make-coffee	Pending Set
turn-on-faucet-1	0.3574	0.3213	0.3213	use-soap: 0.3574 add-water-kettle-1: 0.6426
use-soap	0.0613	0.4693	0.4693	turn-off-faucet-1: 0.9387 rinse-hand: 0.0613

Further explanations for Table 5.16. The performances on problem category **p6** are not stable. For example under M1, the accuracies of case 7 and 9 are about 55%, while the accuracies of case 8 and 10 are above 90%. Note that both case 7 and 9 have the related wrong step *turn-off-faucet-1*. Meanwhile the related sensor $sensor_4$ (*faucet-1, state*) is missing in M1. Consequently, for case 7 and 9, the algorithm mistakenly recognizes the wrong step as a correct step and never goes back in the later steps.The fluctuating performance on case 7-10 indicates that for test cases with wrong steps, the algorithm can never come back in case it gets lost.

Besides, performances on case 7 and 9 under missing sensor case M5 are extremely bad. The high prior of *add-water-kettle-1* plus the missing sensor $sensor_8$ (*kettle-1, has-water*) convince the algorithm to believe that *add-water-kettle-1* has happened. When the wrong step *turn-on-faucet-1* is inputed, the algorithm further confirms that *make-coffee* or *make-tea* is happening, which is totally wrong. Another reason for the overall bad performance under M5 is that this sensor relates to more steps than the other sensors.

Further explanations for Table 5.17. The results in Table 5.17 indicate that the algorithm can hardly handle problems with multiple goals, wrong steps and missing sensors. This is reasonable since the algorithm has to deal with noises from missing sensors, sensor reliabilities, multiple goals and wrong steps, which are too much. Case 11 and case 12 are run on another set of missing sensor cases M13-M18, where the reliabilities of other sensors are 0.95. It turns out the performance is improved comparing to that with M1-12.

5.4.4 Experiment Results Summary

Based on all the experiment results and case-by-case discussions, a summary of the performance of the proposed algorithm is given in this section. For experiment results with sensor reliability changes, we have the following conclusions:

- The algorithm can easily handle recognition problems with single goal and correct steps.
- If sensor reliabilities are above 0.9, the algorithm have relative stable and high accuracies on recognition for multiple goals & correct steps and for single goal & wrong steps.
- The algorithm can correctly distinguish related and non-related wrong steps. For related wrong steps, it can target at the corresponding hierarchical tree structure and repair the explanation from the wrong step so as to provide the desired next steps PS_{step}.
- The algorithm can deal with multiple tasks & wrong steps with sensor reliability equal or above 0.95.
- The algorithm can handle unordered tasks and steps.
- The accuracies of the goal recognition result $PROB$ (a distribution over goals) and the planning result PS (the desired next tasks and steps) are positively correlated with sensor reliabilities.
- Generally, the probabilities of ongoing goals outweigh the probabilities of the other goals after 2 or 3 steps.
- With more noises (e.g. sensor reliability, multiple goals, wrong steps), the number of explanations increase greatly. Multiple goals lead to dramatic increasing of explanation numbers.
- The algorithm usually makes mistakes when a step with lower prior in PS_{step} happens while the related sensor does not give correct measurements. In this case the algorithm tends to believe that the step with higher prior in PS_{step} has happened.
- For wrong steps, if the algorithm get lost, it can never go back.

According to experiment results with missing sensors, we have the following suggestions for setting up a smart home environment.

- Sensors related to start steps of goals should not be missing.
- If a step related to multiple sensors, one of the sensors is missing can be tolerated by the algorithm.
- If the caregiver notices that the old adult repeatedly make mistakes on some steps, the sensors related to those steps should not be missing.
- If a sensor relates to many steps, it should not be missing.
- For a step at the very beginning of a goal which has a high prior, its related sensors should not be missing.

Chapter 6

Conclusion and Future Work

The number of older adults with cognitive impairments increases dramatically in recent years, which brings significant burden to the person himself/herself, their families and also the society. We propose a goal recognition and planning algorithm to support IAAs to help older adults with cognitive impairments complete ADLs independently. Addressed issues in the algorithm include partial observability due to unreliable or missing sensors, concurrent goals, incorrectly executed steps, and partially ordered plans. The algorithm is supposed to enable IAAs to liberate caregivers from repeated and cumbersome care giving works.

6.1 Contribution Summary

6.1.1 Combining Goal Recognition and Planning

This work integrates goal recognition and planning into one process, so as to satisfy the required abilities of an IAA in a smart home setting. In order to teach an older adult with cognitive impairments who does not know how to finish a task, the agent needs to figure out what the older adult is trying to do, monitor the progress status of the task, and provide correct guidance when necessary. The proposed algorithm is a HTN framework based goal recognition and planning process. The recognition procedure and planning procedure are highly coupled together. The HTN framework reduces the searching space for goal recognition. The planning procedure generates the desired next steps to proceed towards the recognized ongoing goals.

Typically the inputs of a goal recognition problem are steps, rather than raw data collected from sensors. In this work, the goal recognition contains a standard Bayesian network based step recognition process which works with sensor measurements. This step recognition process is added to make the algorithm a complete solution for IAAs in smart homes. However, the proposed algorithm can integrate with any action recognition algorithm as long as it can recognize steps with raw sensor data.

Thanks to the hierarchical nature of HTN knowledge base, the hints for the next steps and tasks provided by the planning process are in multiple levels. Thus, the agent can choose a proper level of hints to present to the older adult according to his/her mood and cognitive status. This is very important in the sense that we want the older adult to keep a sense of independence.

6.1.2 Complex Problem Properties

This work considers several important properties of goal recognition: partial observability, concurrent goals, and wrong steps.

Partial Observability

Partial observability means sensors are not 100% reliable or totally missing. Thus sometimes sensors do not give the proper measurements of attributes. The partial observability has direct impact on the step recognition and belief state update. Typical goal recognition algorithms start from step inputs, so they usually do not consider wrong sensor measurements. In this work, the algorithm firstly takes such uncertainties into consideration in the step recognition and belief state update. After that, uncertainties are propagated to goal recognition by computing the degree of satisfaction for preconditions of methods and operators.

Concurrent Goals

The proposed algorithm has the capacity to track multiple ongoing goals. The algorithm can assign a recognized step to the correct decomposition path of the corresponding goal and update the progress status of that goal. In general, recognition problems with multiple goals have a step sequence input, and reason about which goals are ongoing. However, in this work, the problem is even harder. Besides correctly recognizing which goals are

ongoing, the algorithm has to assign each step to the correct execution path of the correct goal so as to correctly track the goal.

Wrong Steps

Another big issue for older adults with cognitive impairments is that they are prone to make mistakes when accomplishing ADLs. A wrong step handling module is proposed in this work to deal with incorrectly executed steps. Similar to re-planning, which is used to handle exceptions in plan execution, the algorithm simulates the pull back effects of the wrong step and rectifies the ongoing status of a goal. As a result, the agent is able to guide the older adult to repair the impact of wrong steps and proceed to the goal.

6.2 Conclusion

The effectiveness of the proposed algorithm is reflected in the experiment results, which are satisfying. For simple recognition problems with single goal and correct steps, the performance is almost 100% as long as sensor reliabilities are above 0.8. Even for the hardest kind of problem, which has multiple goals and wrong steps, the performance is acceptable when sensor reliabilities are above 0.95. Besides, the conducted experiments with sensor missing produce a meaningful guidance on how to set up sensors to help older adults complete ADLs using the proposed algorithm.

6.3 Limitations and Future Work

The proposed algorithm cannot solve recognition problems with shared steps, where an executed step works for more than one goals. However, this phenomena is common in our daily life. An interesting topic would be extending the algorithm to tackle this kind of problem.

The adopted knowledge base is believed to offer the caregivers the freedom to specify their own target goals and preferences, which is customizable. However, great efforts and education are needed to set up a correct knowledge base. A promising way to do this is adopting machine learning techniques to learn those hierarchical knowledge base for common ADLs. At the same time, providing some freedoms for caregivers to configure their preference, such as the order of subtasks, the preferred ways to accomplish a goal.

Currently, goal priors conditional probabilities are set according to the user's experience. A more precise and practical way is to learn customized parameters using history data. With customized parameter, on the one hand, the algorithm can provide more accurate recognition and planning result. On the other hand, the algorithm could easily observe abnormal behaviors of the older adult so as to remind the caregiver pay attention to potential worsening status of the older adult.

References

[1] Jake K Aggarwal and Michael S Ryoo. Human activity analysis: A review. *ACM Computing Surveys (CSUR)*, 43(3):16, 2011.

[2] J Allen, H Kautz, R Pelavin, and J Tennenberg. A formal theory of plan recognition and its implementation. *Reasoning About Plans*, pages 69–126, 1991.

[3] Alzheimer's Association et al. 2016 alzheimer's disease facts and figures. *Alzheimer's & Dementia*, 12(4):459–509, 2016.

[4] Sotiris Batsakis and Euripides GM Petrakis. Sowl: a framework for handling spatio-temporal information in owl 2.0. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, pages 242–249. Springer, 2011.

[5] Asma Benmansour, Abdelhamid Bouchachia, and Mohammed Feham. Multioccupant activity recognition in pervasive smart home environments. *ACM Computing Surveys (CSUR)*, 48(3):34, 2016.

[6] Nate Blaylock and James Allen. Corpus-based, statistical goal recognition. In *IJCAI*, volume 3, pages 1303–1308, 2003.

[7] Blai Bonet and Hector Geffner. Planning under partial observability by classical replanning: Theory and experiments. 2011.

[8] Bruno Bouchard, Sylvain Giroux, and Abdenour Bouzouane. A smart home agent for plan recognition of cognitively-impaired patients. *Journal of Computers*, 1(5):53–62, 2006.

[9] Sandra Carberry. Incorporating default inferences into plan recognition. In *AAAI*, pages 471–478, 1990.

[10] Eugene Charniak and Robert P Goldman. A bayesian model of plan recognition. *Artificial Intelligence*, 64(1):53–79, 1993.

[11] Chao Chen, Barnan Das, and Diane J Cook. A data mining framework for activity recognition in smart environments. In *Intelligent Environments (IE), 2010 Sixth International Conference on*, pages 80–83. IEEE, 2010.

[12] Shuwei Chen, Jun Liu, Hui Wang, and Juan Carlos Augusto. A hierarchical human activity recognition framework based on automated reasoning. In *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*, pages 3495–3499. IEEE, 2013.

[13] Dongkyu Choi and Pat Langley. Learning teleoreactive logic programs from problem solving. In *International Conference on Inductive Logic Programming*, pages 51–68. Springer, 2005.

[14] Philip R Cohen, C Raymond Perrault, and James F Allen. Beyond question answering. *Strategies for natural language processing*, pages 245–274, 1981.

[15] Diane J Cook and Sajal K Das. How smart are our environments? an updated look at the state of the art. *Pervasive and mobile computing*, 3(2):53–73, 2007.

[16] Diane J Cook, Hani Hagras, Vic Callaghan, and Abdesalam Helal. Making our environments intelligent. *Pervasive and Mobile Computing*, 5(5):556–557, 2009.

[17] Samuel Falcon Davis-Mendelow, Jorge A Baier, and Sheila McIlraith. Making reasonable assumptions to plan with incomplete information: Abridged report. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[18] Thierry Denoeux. A k-nearest neighbor classification rule based on dempster-shafer theory. *IEEE transactions on systems, man, and cybernetics*, 25(5):804–813, 1995.

[19] Olivier Desrichard and Catalina Köpetz. A threat in the elder: the impact of task-instructions, self-efficacy and performance expectations on memory performance in the elderly. *European Journal of Social Psychology*, 35(4):537–552, 2005.

[20] Kutluhan Erol, James Hendler, and Dana S Nau. Htn planning: Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128, 1994.

[21] Kutluhan Erol, James A Hendler, and Dana S Nau. Umcp: A sound and complete procedure for hierarchical task-network planning. In *AIPS*, volume 94, pages 249–254, 1994.

[22] Oren Etzioni, Steve Hanks, Daniel S Weld, Denise Draper, Neal Lesh, and Mike Williamson. An approach to planning with incomplete information. *KR*, 92:115–125, 1992.

[23] Hongqing Fang, Lei He, Hao Si, Peng Liu, and Xiaolei Xie. Human activity recognition based on feature selection in smart home using back-propagation algorithm. *ISA transactions*, 53(5):1629–1638, 2014.

[24] Christopher W Geib. Problems with intent recognition for elder care. In *Proceedings of the AAAI-02 Workshop Automation as Caregiver*, pages 13–17, 2002.

[25] Christopher W Geib and Robert P Goldman. A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence*, 173(11):1101–1132, 2009.

[26] Christopher W Geib and S Harp. Empirical analysis of a probabilistic task tracking algorithm. In *Proceedings of Workshop on Agent Tracking, Autonomous Agents and MultiAgent Systems (AAMAS)*. Citeseer, 2004.

[27] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[28] L Gillani Fahad, Syed Fahad Tahir, and Muttukrishnan Rajarajan. Activity recognition in smart homes using clustering based classification. In *Pattern Recognition (ICPR), 2014 22nd International Conference on*, pages 1348–1353. IEEE, 2014.

[29] Robert P Goldman, Christopher W Geib, and Christopher A Miller. A new model of plan recognition. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 245–254. Morgan Kaufmann Publishers Inc., 1999.

[30] Eunyoung Ha, Jonathan P Rowe, Bradford W Mott, and James C Lester. Goal recognition with markov logic networks for player-adaptive games. In *AIIDE*, 2011.

[31] David R Heise. *Expressive order: Confirming sentiments in social actions*. Springer Science & Business Media, 2007.

[32] Jesse Hoey and Marek Grzes. Distributed control of situated assistance in large domains with many tasks. In *ICAPS*, 2011.

[33] Jesse Hoey, Pascal Poupart, Axel von Bertoldi, Tammy Craig, Craig Boutilier, and Alex Mihailidis. Automated handwashing assistance for persons with dementia using video and a partially observable markov decision process. *Computer Vision and Image Understanding*, 114(5):503–519, 2010.

[34] Jesse Hoey, Tobias Schröder, and Areej Alhothali. Affect control processes: Intelligent affective interaction using a partially observable markov decision process. *Artificial Intelligence*, 230:134–172, 2016.

[35] Karen Huff and Victor Lesser. Knowledge-based command understanding: An example for the software development environment. *Computer and Information Sciences Technical Report*, pages 82–6, 1982.

[36] A Hwang and J Hoey. Diy smart home: narrowing the gap between users and technology. In *Proceedings of the Interactive Machine Learning Workshop, 2013 International Conference on Intelligent User Interfaces*, 2013.

[37] Leslie Pack Kaelbling. Hierarchical learning in stochastic domains: Preliminary results. In *Proceedings of the tenth international conference on machine learning*, volume 951, pages 167–173, 2014.

[38] Henry A Kautz. *A formal theory of plan recognition*. PhD thesis, Bell Laboratories, 1987.

[39] Henry A Kautz and James F Allen. Generalized plan recognition. In *AAAI*, volume 86, page 5, 1986.

[40] Oscar D Lara and Miguel A Labrador. A survey on human activity recognition using wearable sensors. *IEEE Communications Surveys and Tutorials*, 15(3):1192–1209, 2013.

[41] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[42] Wookhee Min, Eunyoung Ha, Jonathan P Rowe, Bradford W Mott, and James C Lester. Deep learning-based goal recognition in open-ended digital games. In *AIIDE*, 2014.

[43] Wookhee Min, Bradford Mott, Jonathan Rowe, and James Lester. Deep lstm-based goal recognition models for open-world digital games. 2017.

[44] Wookhee Min, Bradford Mott, Jonathan Rowe, Barry Liu, and James Lester. Player goal recognition in open-world digital games with long short-term memory networks. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 2590–2596, 2016.

[45] Dana S Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. Shop2: An htn planning system. *J. Artif. Intell. Res.(JAIR)*, 20:379–404, 2003.

[46] George Okeyo, Liming Chen, and Hui Wang. Combining ontological and temporal formalisms for composite activity modelling and recognition in smart homes. *Future Generation Computer Systems*, 39:29–43, 2014.

[47] George Okeyo, Liming Chen, Hui Wang, and Roy Sterritt. Ontology-based learning framework for activity assistance in an adaptive smart home. In *Activity Recognition in Pervasive Intelligent Environments*, pages 237–263. Springer, 2011.

[48] George Okeyo, Liming Chen, Hui Wang, and Roy Sterritt. A hybrid ontological and temporal approach for composite activity modelling. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pages 1763–1770. IEEE, 2012.

[49] George Okeyo, Liming Chen, Hui Wang, and Roy Sterritt. A knowledge-driven approach to composite activity recognition in smart environments. In *Ubiquitous Computing and Ambient Intelligence*, pages 322–329. Springer, 2012.

[50] Nuria Oliver, Ashutosh Garg, and Eric Horvitz. Layered representations for learning and inferring office activity from multiple sensory channels. *Computer Vision and Image Understanding*, 96(2):163–180, 2004.

[51] Donald J Patterson, Dieter Fox, Henry Kautz, and Matthai Philipose. Fine-grained activity recognition by aggregating abstract object usage. In *Wearable Computers, 2005. Proceedings. Ninth IEEE International Symposium on*, pages 44–51. IEEE, 2005.

[52] Martha E Pollack. *Generating expert answers through goal inference*. SRI International. Artificial Intelligence Center, 1983.

[53] David Poole. Probabilistic horn abduction and bayesian networks. *Artificial intelligence*, 64(1):81–129, 1993.

[54] Joseph Rafferty, Chris D Nugent, Jun Liu, and Liming Chen. From activity recognition to intention recognition for assisted living within smart homes. *IEEE Transactions on Human-Machine Systems*, 2017.

[55] Ira J Roseman and Craig A Smith. Appraisal theory. *Appraisal processes in emotion: Theory, methods, research*, pages 3–19, 2001.

[56] Earl D Sacerdoti. A structure for plans and behavior. Technical report, DTIC Document, 1975.

[57] Fariba Sadri. Ambient intelligence: A survey. *ACM Computing Surveys (CSUR)*, 43(4):36, 2011.

[58] Charles F. Schmidt, NS Sridharan, and John L. Goodson. The plan recognition problem: An intersection of psychology and artificial intelligence. *Artificial Intelligence*, 11(1-2):45–83, 1978.

[59] Muhammad Shoaib, Stephan Bosch, Ozlem Durmaz Incel, Hans Scholten, and Paul JM Havinga. A survey of online activity recognition using mobile phones. *Sensors*, 15(1):2059–2085, 2015.

[60] Geetika Singla, Diane J Cook, and Maureen Schmitter-Edgecombe. Incorporating temporal reasoning into activity recognition for smart home residents. In *Proceedings of the AAAI workshop on spatial and temporal reasoning*, pages 53–61, 2008.

[61] Thad Starner and Alex Pentland. Real-time american sign language recognition from video using hidden markov models. In *Motion-Based Recognition*, pages 227–243. Springer, 1997.

[62] Douglas L Vail, Manuela M Veloso, and John D Lafferty. Conditional random fields for activity recognition. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 235. ACM, 2007.

[63] TLM Van Kasteren, Gwenn Englebienne, and Ben JA Kröse. Activity recognition using semi-markov models on real world smart home datasets. *Journal of ambient intelligence and smart environments*, 2(3):311–325, 2010.

[64] Marc B Vilain. Getting serious about parsing plans: A grammatical analysis of plan recognition. In *AAAI*, pages 190–197, 1990.

[65] Robert Wilensky. Planning and understanding: A computational approach to human reasoning. 1983.

[66] Andrew D Wilson and Aaron F Bobick. Recognition and interpretation of parametric gesture. In *Computer Vision, 1998. Sixth International Conference on*, pages 329–336. IEEE, 1998.

[67] Daniel H Wilson and Chris Atkeson. Simultaneous tracking and activity recognition (star) using many anonymous, binary sensors. In *International Conference on Pervasive Computing*, pages 62–79. Springer, 2005.

[68] Shuai Zhang, Sally McClean, Bryan Scotney, and Chris Nugent. Learning under uncertainty in smart home environments. In *Engineering in Medicine and Biology Society, 2008. EMBS 2008. 30th Annual International Conference of the IEEE*, pages 2083–2086. IEEE, 2008.

APPENDICES

Appendix A

Methods in Knowledge Base

Table A.1: Method *clean-hand*

	Num.	**Object**	**Attribute**	**Value**
Precondition 1	1	faucet-1	state	on
	2	hand-1	dirty	yes
	3	hand-1	soapy	no
	4	person-1	location	kitchen
	5	person-1	ability	0.6
	Num.	**Name**	**Precedent**	**Decedent**
Subtasks 1	1	use-soap	None	[rinse-hand]
	2	rinse-hand	[use-soap]	None

Table A.2: Method *wash-hand*

	Num.	**Object**	**Attribute**	**Value**
Precondition 1	1	faucet-1	state	off
	2	hand-1	dirty	yes
	3	hand-1	soapy	no
	4	person-1	location	kitchen
	5	person-1	ability	0.6
	Num.	**Name**	**Precedent**	**Decedent**
Subtasks 1	1	turn-on-faucet-1	None	[clean-hand]
	2	turn-off-faucet-1	[clean-hand]	[dry-hand]
	3	dry-hand	[turn-off-faucet-1]	None
	4	clean-hand	[turn-on-faucet-1]	[turn-off-faucet-1]

Table A.3: Method *kettle-1-heat-water*

	Num.	**Object**	**Attribute**	**Value**
Precondition 1	1	kettle-1	water-hot	no
	2	kettle-1	has-water	yes
	3	kettle-1	switch	off
	4	person-1	location	kitchen
	5	person-1	ability	0.6
	Num.	**Name**	**Precedent**	**Decedent**
Subtasks 1	1	switch-on-kettle-1	None	[switch-off-kettle-1]
	2	switch-off-kettle-1	[switch-on-kettle-1]	None

Table A.4: Method *kettle-1-add-water*

	Num.	**Object**	**Attribute**	**Value**
Precondition 1	1	faucet-1	state	off
	2	faucet-1	location	kitchen
	3	kettle-1	water-hot	no
	4	kettle-1	has-water	no
	5	kettle-1	switch	off
	6	person-1	location	kitchen
	7	person-1	ability	0.6
	Num.	**Name**	**Precedent**	**Decedent**
Subtasks 1	1	turn-on-faucet-1	None	[add-water-kettle-1]
	2	turn-off-faucet-1	[add-water-kettle-1]	None
	3	add-water-kettle-1	[turn-on-faucet-1]	[turn-off-faucet-1]

Table A.5: Method *prepare-hot-water*

	Num.	**Object**	**Attribute**	**Value**
Precondition 1	1	faucet-1	state	off
	2	faucet-1	location	kitchen
	3	kettle-1	water-hot	no
	4	kettle-1	has-water	no
	5	kettle-1	switch	off
	6	person-1	location	kitchen
	7	person-1	ability	0.6
Subtasks 1	**Num.**	**Name**	**Precedent**	**Decedent**
	1	kettle-1-heat-water	[kettle-1-add-water]	None
	2	kettle-1-add-water	None	[kettle-1-heat-water]
Precondition 2	**Num.**	**Object**	**Attribute**	**Value**
	1	kettle-1	water-hot	no
	2	kettle-1	has-water	yes
	3	kettle-1	switch	off
	4	person-1	location	kitchen
	5	person-1	ability	0.6
Subtasks 2	**Num.**	**Name**	**Precedent**	**Decedent**
	1	kettle-1-heat-water	None	None

Table A.6: Method *add-tea*

	Num.	**Object**	**Attribute**	**Value**
Precondition 1	1	tea-box-1	open	no
	2	tea-box-1	location	table
	3	cup-1	has-coffee	no
	4	cup-1	has-tea	no
	5	cup-1	location	table
	6	person-1	location	kitchen
	7	person-1	ability	0.6
	Num.	**Name**	**Precedent**	**Decedent**
Subtasks 1	1	close-tea-box-1	[add-tea-cup-1]	None
	2	add-tea-cup-1	[open-tea-box-1]	[close-tea-box-1]
	3	open-tea-box-1	None	[add-tea-cup-1]

Table A.7: Method *add-coffee*

	Num.	**Object**	**Attribute**	**Value**
Precondition 1	1	cup-1	has-coffee	no
	2	cup-1	has-tea	no
	3	cup-1	location	table
	4	coffee-box-1	open	no
	5	coffee-box-1	location	table
	6	person-1	location	kitchen
	7	person-1	ability	0.6
	Num.	**Name**	**Precedent**	**Decedent**
Subtasks 1	1	open-coffee-box-1	None	[add-coffee-cup-1]
	2	add-coffee-cup-1	[open-coffee-box-1]	[close-coffee-box-1]
	3	close-coffee-box-1	[add-coffee-cup-1]	None

Table A.8: Method *mix-tea-water*

	Num.	**Object**	**Attribute**	**Value**
Precondition 1	1	tea-box-1	open	no
	2	tea-box-1	location	table
	3	kettle-1	water-hot	yes
	4	kettle-1	has-water	yes
	5	kettle-1	switch	off
	6	cup-1	has-coffee	no
	7	cup-1	has-tea	no
	8	cup-1	location	cabinet
	9	cup-1	has-water	no
	10	person-1	location	kitchen
	11	person-1	ability	0.6
	Num.	**Name**	**Precedent**	**Decedent**
Subtasks 1	1	add-tea	[get-cup-1]	None
	2	add-water-cup-1	[get-cup-1]	None
	3	get-cup-1	None	[add-tea,add-water-cup-1]

Table A.9: Method *mix-coffee-water*

	Num.	**Object**	**Attribute**	**Value**
Precondition 1	1	kettle-1	water-hot	yes
	2	kettle-1	has-water	yes
	3	kettle-1	switch	off
	4	cup-1	has-coffee	no
	5	cup-1	has-tea	no
	6	cup-1	location	cabinet
	7	cup-1	has-water	no
	8	coffee-box-1	open	no
	9	coffee-box-1	location	table
	10	person-1	location	kitchen
	11	person-1	ability	0.6
Subtasks 1	**Num.**	**Name**	**Precedent**	**Decedent**
	1	add-coffee	[get-cup-1]	None
	2	add-water-cup-1	[get-cup-1]	None
	3	get-cup-1	None	[add-coffee,add-water-cup-1]

Table A.10: Method *make-tea*

	Num.	**Object**	**Attribute**	**Value**
Precondition 1	1	kettle-1	switch	off
	2	person-1	location	kitchen
	3	person-1	ability	0.6
Subtasks 1	**Num.**	**Name**	**Precedent**	**Decedent**
	1	mix-tea-water	[prepare-hot-water]	[drink]
	2	drink	[mix-tea-water]	None
	3	prepare-hot-water	None	[mix-tea-water]

Table A.11: Method *make-coffee*

	Num.	**Object**	**Attribute**	**Value**
Precondition 1	1	kettle-1	switch	off
	2	person-1	location	kitchen
	3	person-1	ability	0.6
	Num.	**Name**	**Precedent**	**Decedent**
Subtasks 1	1	drink	[mix-coffee-water]	None
	2	prepare-hot-water	None	[mix-coffee-water]
	3	mix-coffee-water	[prepare-hot-water]	[drink]

Table A.12: Operator *use-soap*

	Num.	**Object**	**Attribute**	**Value**
Precondition 1	1	kettle-1	switch	off
	2	person-1	location	kitchen
	3	person-1	ability	0.6
	Num.	**Name**	**Precedent**	**Decedent**
Subtasks 1	1	drink	[mix-coffee-water]	None
	2	prepare-hot-water	None	[mix-coffee-water]
	3	mix-coffee-water	[prepare-hot-water]	[drink]

Appendix B

Operators in Knowledge Base

Table B.1: Operator *use-soap*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	hand-1	dirty	yes
	2	hand-1	soapy	no
	3	person-1	location	kitchen
	4	person-1	ability	0.6
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	hand-1	soapy	yes
Parent Task	[clean-hand]			

Table B.2: Operator *rinse-hand*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	faucet-1	state	on
	2	hand-1	dirty	yes
	3	hand-1	soapy	yes
	4	person-1	location	kitchen
	5	person-1	ability	0.6
	Num.	**Name**	**Precedent**	**Decedent**
Effect	1	hand-1	dry	no
	2	hand-1	dirty	no
	3	hand-1	soapy	no
Parent Task	[clean-hand]			

Table B.3: Operator *turn-on-faucet-1*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	faucet-1	state	off
	2	faucet-1	location	kitchen
	3	person-1	location	kitchen
	4	person-1	ability	0.6
	Num.	**Name**	**Precedent**	**Decedent**
Effect	1	faucet-1	state	on
Parent Task	[wash-hand ,kettle-1-add-water]			

Table B.4: Operator *turn-off-faucet-1*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	faucet-1	state	on
	2	faucet-1	location	kitchen
	3	person-1	location	kitchen
	4	person-1	ability	0.6
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	faucet-1	state	off
Parent Task	[wash-hand ,kettle-1-add-water]			

Table B.5: Operator *dry-hand*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	hand-1	dry	no
	2	hand-1	dirty	no
	3	person-1	location	kitchen
	4	person-1	ability	0.6
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	hand-1	dry	yes
Parent Task	[wash-hand]			

Table B.6: Operator *switch-on-kettle-1*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	kettle-1	water-hot	no
	2	kettle-1	has-water	yes
	3	kettle-1	switch	off
	4	person-1	location	kitchen
	5	person-1	ability	0.6
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	kettle-1	water-hot	yes
	2	kettle-1	switch	on
Parent Task	[kettle-1-heat-water]			

Table B.7: Operator *switch-off-kettle-1*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	kettle-1	water-hot	yes
	2	kettle-1	switch	on
	3	person-1	location	kitchen
	4	person-1	ability	0.6
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	kettle-1	switch	off
Parent Task	[kettle-1-heat-water]			

Table B.8: Operator *add-water-kettle-1*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	kettle-1	water-hot	no
	2	kettle-1	has-water	no
	3	kettle-1	switch	off
	4	person-1	location	kitchen
	5	person-1	ability	0.6
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	kettle-1	has-water	yes
Parent Task	[kettle-1-add-water]			

Table B.9: Operator *get-cup-1*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	cup-1	has-water	no
	2	cup-1	has-tea	no
	3	cup-1	location	cabinet
	4	cup-1	has-coffee	no
	5	person-1	location	kitchen
	6	person-1	ability	0.6
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	cup-1	location	table
Parent Task	[mix-tea-water ,mix-coffee-water]			

Table B.10: Operator *open-tea-box-1*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	tea-box-1	open	no
	2	tea-box-1	location	table
	3	person-1	location	kitchen
	4	person-1	ability	0.6
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	tea-box-1	open	yes
Parent Task	[mix-tea-water]			

Table B.11: Operator *add-tea-cup-1*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	tea-box-1	open	yes
	2	tea-box-1	location	table
	3	cup-1	has-coffee	no
	4	cup-1	has-tea	no
	5	cup-1	location	table
	6	person-1	location	kitchen
	7	person-1	ability	0.6
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	cup-1	has-tea	yes
Parent Task	[mix-tea-water]			

Table B.12: Operator *close-tea-box-1*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	tea-box-1	open	yes
	2	tea-box-1	location	table
	3	person-1	location	kitchen
	4	person-1	ability	0.6
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	tea-box-1	open	no
Parent Task	[mix-tea-water]			

Table B.13: Operator *add-water-cup-1*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	kettle-1	water-hot	yes
	2	kettle-1	has-water	yes
	3	kettle-1	switch	off
	4	cup-1	has-water	no
	5	cup-1	location	table
	6	person-1	location	kitchen
	7	person-1	ability	0.6
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	kettle-1	water-hot	no
	2	kettle-1	has-water	no
	3	cup-1	has-water	yes
Parent Task	[mix-tea-water ,mix-coffee-water]			

Table B.14: Operator *open-coffee-box-1*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	coffee-box-1	open	no
	2	coffee-box-1	location	table
	3	person-1	location	kitchen
	4	person-1	ability	0.6
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	coffee-box-1	open	yes
Parent Task	[mix-coffee-water]			

Table B.15: Operator *add-coffee-cup-1*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	cup-1	has-coffee	no
	2	cup-1	has-tea	no
	3	cup-1	location	table
	4	coffee-box-1	open	yes
	5	coffee-box-1	location	table
	6	person-1	location	kitchen
	7	person-1	ability	0.6
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	cup-1	has-coffee	yes
Parent Task	[mix-coffee-water]			

Table B.16: Operator *close-coffee-box-1*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	coffee-box-1	open	yes
	2	coffee-box-1	location	table
	3	person-1	location	kitchen
	4	person-1	ability	0.6
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	coffee-box-1	open	no
Parent Task	[mix-coffee-water]			

Table B.17: Operator *drink*

	Num.	**Object**	**Attribute**	**Value**
Precondition	1	cup-1	has-water	yes
Effect	**Num.**	**Name**	**Precedent**	**Decedent**
	1	cup-1	has-water	no
	2	cup-1	has-tea	no
	3	cup-1	has-coffee	no
Parent Task	[make-coffee ,make-tea]			

Appendix C

Pending Set Output for Cases

Table C.1: Pending Set for Case 1 with Sensor Reliability 0.90

Step Num.	Step Name	PS_{step} for This Step
1	turn-on-faucet-1	add-water-kettle-1: 0.6426 use-soap: 0.3574
2	use-soap	turn-off-faucet-1: 0.2644 rinse-hand: 0.7356
3	rinse-hand	turn-off-faucet-1: 0.9871 switch-on-kettle-1: 0.0129 dry-hand: 0.9871
4	turn-off-faucet-1	turn-off-faucet-1: 0.1011 dry-hand: 0.8989
5	dry-hand	

Table C.2: Pending Set for Case 2 with Sensor Reliability 0.90

Step Num.	Step Name	PS_{step} for This Step	
1	turn-on-faucet-1	add-water-kettle-1:	0.6426
		use-soap:	0.3574
2	add-water-kettle-1	turn-off-faucet-1:	0.9632
		rinse-hand:	0.0368
3	turn-off-faucet-1	switch-on-kettle-1:	1.0
4	switch-on-kettle-1	switch-on-kettle-1:	0.0073
		turn-on-faucet-1:	0.0073
		switch-off-kettle-1:	0.9927
5	switch-off-kettle-1	get-cup-1:	1.0
6	get-cup-1	add-water-cup-1:	1.0
		open-coffee-box-1:	0.5
		open-tea-box-1:	0.5
7	open-tea-box-1	add-coffee-cup-1:	0.0997
		add-tea-cup-1:	0.8888
		add-water-cup-1:	0.9885
		open-coffee-box-1:	0.0057
		open-tea-box-1:	0.0057
8	add-tea-cup-1	close-tea-box-1:	0.9964
		add-tea-cup-1:	0.0036
		add-water-cup-1:	0.9964
9	close-tea-box-1	close-tea-box-1:	0.0032
		add-water-cup-1:	0.9968
10	add-water-cup-1	drink:	1.0
11	drink		

Table C.3: Pending Set for Case 5 with Sensor Reliability 0.90

Step Num.	Step Name	PS_{step} for This Step	
1	turn-on-faucet-1	add-water-kettle-1:	0.6426
		use-soap:	0.3574
2	use-soap	turn-off-faucet-1:	0.2644
		rinse-hand:	0.7356
3	rinse-hand	turn-off-faucet-1:	0.9853
		switch-on-kettle-1:	0.0147
		dry-hand:	0.9853
4	turn-off-faucet-1	turn-off-faucet-1:	0.4253
		dry-hand:	0.5747
5	turn-on-faucet-1	add-water-kettle-1:	0.9274
		dry-hand:	0.9274
6	dry-hand	turn-off-faucet-1:	0.9107
		add-water-kettle-1:	0.4079
		dry-hand:	0.9107
7	add-water-kettle-1	turn-off-faucet-1:	0.5765
		switch-on-kettle-1:	0.4235
		add-water-kettle-1:	0.1987
		dry-hand:	0.8489
8	turn-off-faucet-1	turn-off-faucet-1:	0.8461
		switch-on-kettle-1:	0.9964
		add-water-kettle-1:	0.0503
		switch-off-kettle-1:	0.0036
		dry-hand:	0.8497
9	switch-on-kettle-1	turn-off-faucet-1:	0.8491
		dry-hand:	0.8491
		add-water-kettle-1:	0.0504
		switch-off-kettle-1:	1.0
10	switch-off-kettle-1	turn-off-faucet-1:	0.8438
		add-water-kettle-1:	0.0501
		dry-hand:	0.8501
		get-cup-1:	0.9937
		switch-on-kettle-1:	0.0063
		switch-off-kettle-1:	0.0063

Continued on next page

Table C.3 – continued from previous page

Step Num.	Step Name	PS_{step} for This Step	
11	get-cup-1	open-tea-box-1:	0.5
		turn-off-faucet-1:	0.8491
		add-water-kettle-1:	0.0504
		dry-hand:	0.8491
		open-coffee-box-1:	0.5
		add-water-cup-1:	1.0
12	open-coffee-box-1	add-coffee-cup-1:	0.8889
		open-tea-box-1:	0.006
		turn-off-faucet-1:	0.8395
		add-tea-cup-1:	0.0991
		add-water-kettle-1:	0.0499
		dry-hand:	0.8515
		switch-on-kettle-1:	0.012
		open-coffee-box-1:	0.006
		add-water-cup-1:	1.0
13	add-coffee-cup-1	add-coffee-cup-1:	0.0057
		turn-off-faucet-1:	0.8443
		close-coffee-box-1:	0.9943
		add-water-kettle-1:	0.0502
		dry-hand:	0.85
		switch-on-kettle-1:	0.0057
		add-water-cup-1:	1.0
14	close-coffee-box-1	close-coffee-box-1:	0.0038
		turn-off-faucet-1:	0.846
		add-water-kettle-1:	0.0503
		dry-hand:	0.8497
		switch-on-kettle-1:	0.0038
		add-water-cup-1:	1.0

Continued on next page

Table C.3 – continued from previous page

Step Num.	Step Name	PS_{step} for This Step	
15	add-water-cup-1	turn-off-faucet-1:	0.8266
		drink:	0.966
		close-coffee-box-1:	0.0036
		add-water-kettle-1:	0.0487
		dry-hand:	0.8479
		switch-on-kettle-1:	0.0277
		add-water-cup-1:	0.0304
16	drink	turn-off-faucet-1:	0.8366
		switch-on-kettle-1:	0.0148
		drink:	0.0148
		add-water-kettle-1:	0.0497
		dry-hand:	0.8514

Table C.4: Pending Set for Case 9 with Sensor Reliability 0.90

Step Num.	Step Name	PS_{step} for This Step
1	turn-on-faucet-1	add-water-kettle-1: 0.6426 use-soap: 0.3574
2	use-soap	turn-off-faucet-1: 0.2621 rinse-hand: 0.7379
3	use-soap	turn-off-faucet-1: 0.2621 rinse-hand: 0.7379
4	turn-off-faucet-1	turn-on-faucet-1: 1.0
5	turn-on-faucet-1	add-water-kettle-1: 0.4779 use-soap: 0.71
6	use-soap	turn-off-faucet-1: 0.0981 add-water-kettle-1: 0.0221 rinse-hand: 0.9019 use-soap: 0.0342
7	rinse-hand	turn-off-faucet-1: 1.0 dry-hand: 1.0
8	rinse-hand	turn-off-faucet-1: 1.0 dry-hand: 1.0
9	dry-hand	turn-off-faucet-1: 0.8999 dry-hand: 0.1001
10	turn-off-faucet-1	

Table C.5: Pending Set for Case 11 with Sensor Reliability 0.95

Step Num.	Step Name	PS_{step} for This Step
1	turn-on-faucet-1	add-water-kettle-1: 0.6426 use-soap: 0.3574
2	use-soap	turn-off-faucet-1: 0.1448 rinse-hand: 0.8552
3	rinse-hand	turn-off-faucet-1: 1.0 dry-hand: 1.0

Continued on next page

Table C.5 – continued from previous page

Step Num.	Step Name	PS_{step} for This Step
4	rinse-hand	turn-off-faucet-1: 1.0 dry-hand: 1.0
5	turn-off-faucet-1	turn-off-faucet-1: 0.0496 dry-hand: 0.9504
6	turn-on-faucet-1	add-water-kettle-1: 0.9591 dry-hand: 0.9591
7	dry-hand	turn-off-faucet-1: 0.0614 add-water-kettle-1: 0.9622 dry-hand: 0.0614
8	add-water-kettle-1	turn-off-faucet-1: 1.0 add-water-kettle-1: 0.3769 dry-hand: 0.0199
9	turn-off-faucet-1	turn-off-faucet-1: 0.039 switch-on-kettle-1: 1.0 add-water-kettle-1: 0.3696 dry-hand: 0.039
10	switch-on-kettle-1	turn-off-faucet-1: 0.0382 add-water-kettle-1: 0.384 dry-hand: 0.0382 switch-on-kettle-1: 0.0229 switch-off-kettle-1: 0.9771 use-soap: 0.0229
11	switch-off-kettle-1	turn-off-faucet-1: 0.0373 add-water-kettle-1: 0.3975 dry-hand: 0.0373 get-cup-1: 0.9557 switch-off-kettle-1: 0.0443 use-soap: 0.0443

Continued on next page

Table C.5 – continued from previous page

Step Num.	Step Name	PS_{step} for This Step	
12	get-cup-1	open-tea-box-1:	0.4681
		add-water-cup-1:	0.9362
		turn-off-faucet-1:	0.0366
		add-water-kettle-1:	0.4098
		dry-hand:	0.0366
		get-cup-1:	0.0638
		open-coffee-box-1:	0.4681
		use-soap:	0.0638
13	open-coffee-box-1	add-coffee-cup-1:	0.8396
		turn-off-faucet-1:	0.0328
		add-tea-cup-1:	0.0425
		add-water-kettle-1:	0.4446
		dry-hand:	0.0328
		use-soap:	0.1179
		open-coffee-box-1:	0.1179
		add-water-cup-1:	1.0
14	add-water-cup-1	add-coffee-cup-1:	0.832
		turn-off-faucet-1:	0.0281
		close-coffee-box-1:	0.0305
		add-water-kettle-1:	0.5051
		add-tea-cup-1:	0.0364
		add-water-cup-1:	0.1428
		dry-hand:	0.0281
		open-coffee-box-1:	0.1011
		use-soap:	0.2134
15	close-coffee-box-1	add-coffee-cup-1:	0.832
		turn-off-faucet-1:	0.0281
		close-coffee-box-1:	0.0305
		add-water-kettle-1:	0.5051
		add-tea-cup-1:	0.0364
		add-water-cup-1:	0.1428
		dry-hand:	0.0281
		open-coffee-box-1:	0.1011
		use-soap:	0.2134

Continued on next page

Table C.5 – continued from previous page

Step Num.	Step Name	PS_{step} for This Step	
16	open-coffee-box-1	add-coffee-cup-1:	0.832
		turn-off-faucet-1:	0.0281
		close-coffee-box-1:	0.0305
		add-water-kettle-1:	0.5051
		add-tea-cup-1:	0.0364
		add-water-cup-1:	0.1428
		dry-hand:	0.0281
		open-coffee-box-1:	0.1011
		use-soap:	0.2134
17	add-coffee-cup-1	add-coffee-cup-1:	0.2539
		turn-off-faucet-1:	0.0295
		add-water-cup-1:	0.1007
		close-coffee-box-1:	0.7461
		add-water-kettle-1:	0.5887
		dry-hand:	0.0252
		use-soap:	0.3502
18	close-coffee-box-1	turn-off-faucet-1:	0.0286
		drink:	0.6032
		add-water-cup-1:	0.0941
		close-coffee-box-1:	0.3027
		add-water-kettle-1:	0.6147
		dry-hand:	0.0236
		use-soap:	0.3918
19	drink	turn-off-faucet-1:	0.0297
		dry-hand:	0.0239
		add-water-kettle-1:	0.6084
		drink:	0.388
		use-soap:	0.3823