Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

HTN Planning with Utilities

Ebaa Alnazer

Course of Study:	Computer Science M.Sc.
Examiner:	Prof. Dr. Marco Aiello
Supervisor:	Dr. Ilche Georgievski
Commenced:	May 2, 2019
Completed:	November 4, 2019

Abstract

Hierarchical Task Network (HTN) planning is a popular automated planning technique used in many real-world applications. HTN planners can be categorized depending on the search space in which they operate. Our work here analyzes and enhances an existing state-based HTN planner in order to make it deal with consumption-sensitive domains while taking the risk attitudes of the various tasks, as well as potential cycles into account. A typical planning algorithm proceeds by decomposing tasks into smaller ones. This is usually done non-determinstically by picking one of applicable decomposition methods. However, this choice may greatly affect the final planning outcome. In domains, which place a high importance on the consumption of resources, it is of utmost importance to make an informed choice of the decomposition method. We propose to choose the method that promises the best results in terms of resource consumption based on an estimation, known as the utility, that is calculated in a dedicated pre-processing phase that precedes the actual planning phase. Our main utility-aware planning algorithm takes these estimations into account. Furthermore, it adapts to the existence of cycles by trying to avoid the methods that may lead to them because entering a cycle by the planning algorithm means that it may repeatedly decompose tasks that were already decomposed. If this case is not handled, the planning algorithm may loop in the cycle infinitely. We prove the validity of our approach by presenting a prototypical implementation of the proposed algorithms. Furthermore, we evaluate the performance, usability, and the quality of the resulting plans of our approach and highlight its strengths and weaknesses compared to regular HTN planning.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

Hierarchical task network (HTN) planning is one of the most prominent automated planning techniques widely used to solve various real-world problems in a variety of domains [GNT04]. The basic idea behind this technique is to model the problem at hand as a *planning problem*, and solve it afterwards. A planning problem is represented via an initial *state of the world* and an initial *task network* consisting of a set of tasks that need to be achieved, in addition to rich *domain knowledge*. The state of the world gives a statement about the different objects of the problem and the relationships between them. Moreover, the domain knowledge consists of networks of *compound* and *primitive* tasks. A task is said to be primitive, if there is a *planning operator* that can execute it directly given a suitable state of the world, whereas a task is compound, if it needs to be refined into smaller sub-tasks using a refinement *method*. A task network represents a hierarchy of such tasks.

HTN planners are categorized depending on the search space in which they operate. Our work here is based on an HTN planner that is categorized as a *state-based planner* [GA15]. The planning process of such planners searches for a state that can accomplish the goal task network as follows: if the task network contains a compound task, then it is decomposed into simpler tasks using one of the methods described in the domain leading to a new decomposition level. However, the decomposition does not affect the state of the world. On the other hand, if the task is primitive, then it is executed directly using a planning operator, and as a result of this, the state is transferred into a new one, and the operator becomes part of the plan. The process continues until the task network is empty and the solution of the problem is represented as a sequence of operators [GA15]. A method, in the context of HTN planning, can be seen as a description of how to perform a task by decomposing it into simpler sub-tasks. This decomposition ensures that performing the sub-tasks leads to accomplishing the compound task. On the other hand, primitive tasks are accomplished by operators that can be performed directly in the current state of the world. This has the effects of changing the state by transforming it into another one.

Generally, when an HTN planner has to decompose a compound task, there might be different methods that are applicable at the same time in the current state of the world. However, the choice of an applicable method to use can dramatically impact the resulting plan. On the other hand, in real-world domains with large potential wins or losses of resources, such as money, fuel or even human beings, if the planning process ignores the risk associated with the application of each method, the outcome can be undesirable or of a low *quality*. In this work, the quality of a plan is defined in terms of the resource consumption incurred by applying it. This means that a plan that leads to a smaller resource consumption is preferable over a high resource-consuming plan.

The research study presented here tackles the problem of choosing a method among different alternatives to decompose a compound task. Utility theory defines the concept of a *utility* as a function of profit [RN16]. We adopt this concept and use it to deterministically choose the method that promises the least resource consumption. Hence, each method is assigned a utility value, which is an estimation of the resource consumption associated with choosing it. Therefore, the utility

function, in this context, expresses the loss or the consumption of resources. Moreover, the utility value depends on the risk attitude of the corresponding compound task being decomposed by the method. Our goal is to minimize the task utilities while taking the corresponding risk attitude into account.

Our approach requires that all domain operators are assigned a function that describes their resource consumption. Depending on these functions, the first step of our approach estimates the consumption value (or utility) for all tasks and methods. In the second step, the main algorithm searches for a plan considering the utilities computed in the previous step. Moreover, our algorithm adapts to the existence of *cycles* by trying to avoid methods that can lead to them and also backtracking to other alternatives if the consumption exceeds the available resource. A cycle occurs during the planning process when the planner has to decompose a task that was already decomposed in a previous step. If the planner is unable to handle such cases, it may loop in the cycle infinitely.

The remainder of this thesis is organized as follows. In Chapter 2, we give the necessary background knowledge needed to understand the work. We start by introducing automated planning, generally, and then broadly explaining the specific automated planning technique used in this work, i.e., HTN planning, by defining it formally. After that, we present an overview of utility theory and the main concepts including utilities, utility functions and the different risk attitudes. Chapter 3 represents the core of our work. We start by explaining the importance of a utility-aware planner in real-world applications. Afterwards, we explain the approach we followed and present our designed algorithms with a running example to clarify and highlight the main ideas of the solution. Chapter 4 provides details on how the prototypical implementation is developed to prove the validity of our conceptual framework and algorithms. In Chapter 5, we describe the framework used to evaluate the introduced utility-aware planner, and present the obtained results. Chapter 6 gives an overview of the related work. Finally, we list our findings and conclusions, in addition to our vision of the future work of this research in Chapter 7.

2 Background

In this chapter, we give an overview of the background information necessary to understand the presented research study. We start by defining *artificial intelligence planning*, then we introduce one particular planning technique, namely *Hierarchical planning*. In the last section, we define *utility theory* and present concepts related to it.

2.1 Automated Planning

Planning is a deliberation process that aims at finding and organizing a course of actions, by estimating their expected outcomes, in order to achieve some designated goals. The full potential of planning enfolds when having to achieve complex tasks and goals rather than simple and familiar everyday tasks where planning is done implicitly and unconsciously. Specifically, some systems, e.g., safety-critical systems require efficient handling of resources under defined constraints, e.g., time constraints, where alternative plans should be examined carefully. An example of such systems could be a system where a truck should move a pile of containers from one location to another. The containers should be transferred within a limited time to the destination and at each location there is a robot that can load/unload the containers to/from the truck. The resource here can be the fuel, for example. Such situations need careful planning that guarantees the delivery of the containers without exceeding the available resources; moreover, it is time constrained and has many alternatives that can be done to achieve the goal, i.e., following different routes leads to different solutions with varying resource consumption. One way of satisfying these requirements efficiently and effectively is the usage of *automated planning* [GNT04].

Automated planning is a sub-area of artificial intelligence that focuses on the computational study of the deliberation process [GNT04]. Such planning is used to automate the planning process in systems that show intelligent behavior. The goal of this process is to reach a set of pre-stated objectives by performing a set of actions that transform the state of the *planning domain* step by step until satisfying those objectives. The previously mentioned truck example could be an example of a planning domain that includes a truck, locations and robots. The domain state is represented by the position of the track as well as the locations of the containers. This state can be changed by performing an action, e.g., the truck movement from one location to another. As the goal of this example is to have the containers in a specific location, a sequence of actions that change the state of the system should be performed in order to achieve this defined goal. This is done by reasoning on the *conceptual model of the planning domain*, *effects*, *plans* and *goals*. The *conceptual model of the planning domain* is a representation of the different components of the domain, the states of those components and the actions that affect the system by changing its state. The result of the planning process is a *plan*, which consists of a set of actions to be performed to satisfy the requirements or the *goals* [VLP08]. However, automated planning is a resource-intensive technique that should only be used in complex applications, which are often characterized by uncertainty.

There exist multiple techniques to model a planning problem, and also a diversity of ways in which the planning process is performed. Examples of these techniques include *classical planning* and *hierarchical planning*. In this work, we focus on hierarchical planning.

2.2 Hierarchical Planning

Hierarchical Task Network (HTN) planning or *Hierarchical planning* is a popular domain-independent Artificial Intelligence (AI) planning technique, which is widely used in solving numerous real-world problems in different domains [GL14], e.g., Web services [SPW+04], and ubiquitous computing [AGPR04; GNA13]. Furthermore, various HTN planners exist, such as Simple Hierarchical Ordered Planner2 (SHOP2) (Nau et al., 2001), O-PLAN [TDD96], SIADEX (Castillo et al. 2006) [CFGP06], SH (Georgievski et al. 2017) [GNN+17]. Starting from a set of goals to be achieved and initial state, the goal of an HTN planning[1] is to find a plan that accomplishes the goal tasks.

In both of HTN planning and classical planning, a *state* is represented as a conjunction of ground predicates. A predicate expresses variable statements or the relation between those variables and it evaluates to true or false. The predicate is said to be ground if all its variables are bound to values. An example of a predicate is (at ?obj ?loc) predicate which specifies that an object is at a certain location. This predicate is ground when the location and object variables are bound to specific values.

Unlike classical planning where the planning system aims at achieving a set of goal *states*, HTN planning works towards performing a set of predefined *tasks*. An example of a task is moving an object from one position to another. In addition, the way HTN planning operates differs from classical planning [GNT04]: In classical planning, the planner decides on the applicable actions in each time step to transform the state of the world towards the goal state. Let us consider the previously explained planning problem in the subsection 2.1 and let us assume that we have two locations, a robot in each location, a truck, and a number of containers and the goal state is to have the truck in the first location and the containers in the second location. In this case, a plan that starts from an initial state in which the truck and the containers are in the first location, and then proceeds by picking up the containers, moving to the second location, dropping the containers there and then moving again to the first location is clearly a plan that consists of a sequence of actions each of which changes the state of the world and brings it closer to the goal state. However, in HTN planning, "decomposition methods" are used in order to find the plan which leads to achieving the designated tasks. In details, in order for an HTN planner to solve a planning problem, it is provided with (i) an initial *task network* with possible constraints on the ordering of those tasks, (ii) an initial state of the world and (iii) domain knowledge, which is provided by a domain expert [GA15]. The state of the world is expressed as a set of predicates, whereas the initial *task network* represents the set of tasks to be performed, i.e., the goal of the planning problem. Finally, the domain knowledge is represented as networks of tasks which can be either *primitive* or *compound*. A task is said to be

[1] the terms planning and AI planning, also HTN planning system and HTN planner are used interchangeably throughout the document

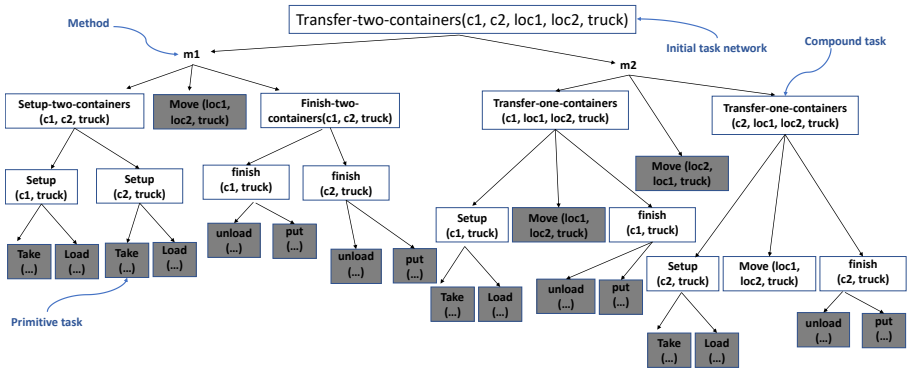


Figure 2.1: Example of an HTN planning domain model

primitive, if it can be performed by an action/operator directly when its precondition complies with the current state of the world, whereas the task is *compound* if it can be decomposed further into simpler sub-tasks using a set of methods provided as domain knowledge.

HTN planners are categorized depending on the search space in which they operate. Our work here is based on an HTN planner that is categorized as a *state-based planner* [GA15]. The planning process of such planners searches for a state that can accomplish the goal task network as follows: Starting from the initial state and a goal task network, HTN planners decompose *compound tasks* recursively until having a sequence of *primitive* executable tasks called *operators*, hence the solution of the problem is reached. Figure 2.1 shows an example of the same HTN planning domain model that was explained earlier where we have a truck to move containers between locations. As we see here, the goal task is to transfer two containers from the first location to the second one and the available methods to decompose the compound tasks are presented as well, e.g., "m1" method decomposes the "transfer-two-containers" task into partially ordered sub-tasks, namely: (i) set-up-two-containers, which is again a compound task that can be decomposed further, (ii) move, which is a primitive task that can be executed without any further decomposition , (iii) finish, which is a compound task. In this example, we notice that the "transfer-two-containers" task can be decomposed in two different ways by two different methods. A run of this example for a specific problem would bind all variables, i.e., c1, c2, loc1, loc2 and truck to values corresponding to this concrete problem. Furthermore, it will try to find the sequence of primitive tasks/operators to perform in order to achieve the goal of transferring the two containers between two locations.

The main reason which gives HTN planning an advantage over classical planning techniques, in terms of scalability and speed, is the rich and well-structured domain knowledge that is provided to the planner by a domain expert. Using this knowledge, the search space for a plan is reduced because of the useful control information that directs the planning process. Such knowledge is encoded in the domain description as illustrated in the previous example (see Figure 2.1), which consists of networks of primitive and compound tasks [GA15].

Many studies have formalized HTN planning with a set of definitions. Examples of these studies include [GA15; GNT04; NCLM99]. In this work, we adopt some of these definitions and later we will modify and adapt some of them to meet the concepts that are presented in our study.

Definition 2.2.1 (Predicate)
A predicate $p \in P$ *is defined as follows:* $p = \langle symbol(p), terms(p) \rangle$, *where*

- *$symbol(p)$ is the predicate symbol.*
- *$terms(p)$ are terms, such that $terms(p) = \langle \tau_1, \tau_2, \ldots, \tau_n \rangle$ is a set of terms. A term is either a constant $co \in CO$, where CO is a finite set of constants or is a variable $v \in V$, where V is an infinite set of variables.* □

A predicate gives a declaration about the different objects in the environment and the relations between them.

Definition 2.2.2 (Ground predicate)
A ground predicate is a predicate where all the terms are constants. □

Definition 2.2.3 (State)
A state s is defined as a set of ground predicates $s = \langle p_1, p_2, ..., p_n \rangle$. □

Definition 2.2.4 (Evaluation of the predicate)
An evaluation $Eval$ of a predicate p in state s is defined as follows:

$$Eval(p, s) = \begin{cases} true, & \text{if } p \in s; \\ false, & \text{otherwise.} \end{cases}$$

□

In the (Definition 2.2.3), a closed-world assumption is adopted. This assumption is defined by the previous $Eval$ function, such that the state of the world s should only include the predicates that evaluate to "True" while the ones that evaluate to "false" are excluded from s.

Definition 2.2.5 (Primitive task)
A primitive task $t_p \in T_p$ is a pair $t_p = \langle symbol(t_p), terms(t_p) \rangle$, where T_p is a finite set of primitive tasks and

- *$symbol(t_p)$ is a primitive task symbol.*
- *$terms(t_p)$ is a set of terms, such that $terms(t_p) = \{ \tau_1, \tau_2, \ldots, \tau_n \}$.* □

Definition 2.2.6 (Operator)
An operator $o \in O$ is a triple $o = \langle ptask(o), pre(o), eff(o) \rangle$, where O is a finite set of operators and

- *$ptask(o)$ is a primitive task.*
- *$pre(o)$ is the precondition of this operator and is defined as follows: a logical formula over predicates using logical connectives.*
- *$eff(o)$ is the effect that this operator makes when executed in the current state of the world.* □

An operator o is applicable in the current state of the world s if and only if $pre(o) \subseteq s$. Applying the operator transforms the current state of the world into a new state $s' = s[o]$ in which some predicates are deleted, i.e., are given the boolean value "false" while others are added. The sets of deleted and added predicates are denoted by $eff^-(o)$, $eff^+(o)$, respectively. Thus, the new state is defined as follows: $s' = s[o] = (s \setminus eff^-(o)) \cup eff^+(o)$. Operators are used to execute primitive tasks directly without decomposing them further into sub-tasks.

Definition 2.2.7 (Compound task)
The compound task $t_c \in T_c$ is defined in a similar way to the primitive task.
T_c is a finite set of compound tasks. □

Definition 2.2.8 (Task network)
A task network tn is a pair $\langle T_n, \prec \rangle$, where

- *$T_n = T_c \cup T_p$; it represents a finite set of primitive and compound tasks.*
- *$\prec$ is a partial order on T_n.*

$tn \subseteq TN$ such that TN is defined as the set of all task networks over T_n. □

The partial order $\prec$ of the tasks T_n specifies restrictions on the ordering of the tasks in the task network, such that this ordering must be maintained during the planning process and by the solution.

Definition 2.2.9 (Method)
A method $m \in M$ is a triple $m = \langle ctask(m), pre(m), tn(m) \rangle$, where M is a set of methods and

- *$ctask(m)$ is a compound task.*
- *$pre(m)$ is a precondition.*
- *$tn(m)$ is a task network.* □

Given a compound task t_c, a method m is applicable in the current state s if and only if $pre(m) \subseteq s$. Applying m in the state s, in order to decompose t_c, results into a task network $tn(m)$ that represents a set of sub-tasks to accomplish in order to accomplish t_c.

Definition 2.2.10 (HTN planning domain)
An HTN planning domain d is a pair $d = \langle O, M \rangle$, where

- *O is a set of operators.*
- *M is a set of methods.* □

Given the set of operators and the set of the methods in the domain description, we can infer the set of primitive tasks T_p and the compound tasks T_c that can appear during the planning process.

Definition 2.2.11 (HTN planning problem)
An HTN planning problem ρ is a tuple $\rho = \langle s_0, tn_0, d \rangle$, where

- *s_0 is the initial state of the system.*
- *tn_0 is the initial task network.*

- $d = (O, M)$ *is the planning domain.* □

Definition 2.2.12 (Solution)
Given a planning problem $\rho = \langle s_0, tn_0, d \rangle$*, where* $tn_0 = \langle T_0, \prec_0 \rangle$*, a sequence of operators* $o_1, o_2, \ldots, o_n$ *is a solution to* ρ *if and only if there exists a task* $t \in T_0$*, such that* $\langle t, t' \rangle \in \prec_0$ *for all* $t' \in T_0$*, i.e., task* t *has no predecessor in* T_0*. We distinguish two cases:*

- *case1: t is a primitive task and* $o_1 \in O$*, such that* $ptask(o_1) = t$ *and* o_1 *is applicable in* s_0*, such that* $o_2, o_3, \ldots, o_n$ *is a solution to* $\rho = \langle s_0[o_1], tn_0 \setminus \{t\}, d \rangle$*.*
- *case2: t is a compound task and there exists a method* $m \in M$*, such that* $ctask(m) = t$ *and* m *is applicable in* s_0 *and decomposes* t *into* $tn(m)$*, such that the new task network* tn' *is defined as follows:* $tn' = tn_0 \setminus \{t\} \cup tn(m)$ *and* $o_1, o_2, \ldots, o_n$ *is a solution to* $\rho = \langle s_0, tn', d \rangle$*.* □

2.3 Utility Theory

Solving real-world problems means facing alternative choices. *Utility theory* is a mathematical method that aims at quantifying *actors'* degree of preference to help them reason about these alternatives [SL08]. In addition, it studies the change in *actors'* behavior and preferences when they are uncertain about the result of their choices. *Actors*, in the utility theory context, are the decision makers.

An important concept in utility theory is the *rational agent*. An agent, in general, can be anything, e.g., a person, a robot, etc. that perceives its environment and acts accordingly. However, a *rational agent* is an agent that always tries to choose the action that maximizes its *performance measure*, given its built-in knowledge of the environment and the *percept sequence*, i.e., the full history of all what the agent has perceived [RN16]. The *performance measure* is a measure that captures the desirability and evaluation of a set of state sequences.

In order to model rational agent's desires, a *utility function* is used. This function maps the state of the world to real numbers. In case of uncertain outcomes in a particular state, the utility is computed as the expected value of the utility function considering an appropriate probability distribution across all states. In this case, the notion of "risk" is introduced in *utility theory* where different agents' attitudes towards risk are studied. Agents are categorized into three different categories based on their risk attitude: *risk averse/avoiding*, *risk seeking/loving* and *risk neutral*.

A risk averse/avoiding agent, as the name suggests, prefers reducing the risk of loss over profiting i.e., an agent with this risk attitude tends to accept a low gain with small investment rather than a high investment with probable high return but also high uncertainty. The utility function in this case has two characteristics: concavity and an upwards slope in which the utility increase rate decreases for every additional unit of gain Figure 2.2(b). For example, let us consider a risk averse agent that starts with zero gain and zero utility and let us assume that he earns one unit of gain and his utility increases by 1. At the gain of 1, if the agent earns an additional unit of gain, its utility would increase by an amount which is less than 1.
In contrast, a *risk seeking/loving* agent always prefers risky choices in case of uncertainty i.e., when the expected value is the same in all alternatives. In other words, an agent with this kind of risk attitude values even a small additional gain with a high risk compared to a lower guaranteed gain. The utility function in this case is convex Figure 2.2(c).

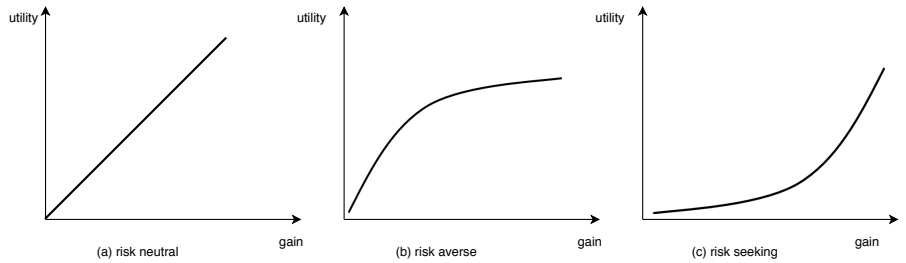


Figure 2.2: Risk attitudes: risk neutrality, risk aversion, risk seeking

Table 2.1: A comparison between different risk attitudes

Risk attitude	Agent's preference	Utility function	Increase rate of the utility function with respect to gain
Risk averse/avoiding	Prefers certain choices over gain	Concave	Decreasing
Risk seeking/loving	Prefers high risk with possible small gain	Convex	Increasing
Risk neutral	Indifferent to risk	Linear	Constant

The last risk attitude discussed by *utility theory* is *risk neutral* in which the agent is indifferent to risk when evaluating different alternatives. In that case, the agent focuses on the potential gain regardless of the risk associated with each choice. The utility function here is linear where the increment in the utility is proportional to the increment in the gain Figure 2.2(a).
Figure 2.2 illustrates the utility function of different risk attitudes and Table 2.1 presents a brief comparison between them.

3 Utility-based HTN Planning

In this chapter, we explain the core of our work. We start by showing the importance of this study and the motivation behind considering utilities when using HTN planning. After that, we describe the approach that we have followed including the assumptions and the processing stages. Lastly, we present the proposed algorithm that realizes each of these stages.

3.1 Motivation

Hierarchical Task Network (HTN) planning is a popular and widely used planning paradigm that solves problems in a variety of modern applications. However, most existent HTN planners work towards delivering satisfactory plans without considering the quality of such plans [SBM08]. A plan's *quality* is defined in terms of its total cost, which is the sum of the costs of all of the its operators, in addition to its length, which represents the total number of its constituent operators. Taking this quality into consideration is very important in most real-world problems since they are typically characterized by the unpredictability of alternative choices that arise during the planning process. Using the quality value of each alternative, we can choose more favorable plans. One way to estimate the quality of plans is by considering their *risk*. Thus, a planning technique that estimates the risk of each alternative is preferable over techniques that have an indifferent attitude towards it, especially in applications with large wins or losses of *resources*, such as money, time, power, equipment or even humans.

As discussed in Section 2.2, the primary reason behind HTN planning success is the rich domain knowledge that is given to a planner by a domain expert. Guiding the plan generation process using this knowledge makes HTN planners the closest approach to the way in which humans solve problems. Our study further strengthens this advantage by allowing different attitudes towards risk. This simulates the way in which humans, with different risk attitudes, think when solving a problem.

An example of applications, where accounting for resources loss is of a great importance, is the previously explained "Truck and containers" example (Figure 2.1). The resource here could be the available fuel or it could be the required time for transporting the containers. For example, if we consider the fuel consumption to be the risk factor that influences the planning process, we can come up with a solution that satisfies the goal of transferring the containers to the destination while keeping the fuel cost as little as possible.

Let's consider another example that shows the importance of taking risk into account during the planning process [GL14]: a fire breaks out in a smart building and all occupants should be rescued within a limited time. However, there are two alternatives to save them. The first is to guide people through a short path but with a high chance of encountering flames on the way out. The other way

is to let people follow a longer track, which is less dangerous. The planner has to find a plan that guarantees a successful escape for as much people as possible, but within a bounded time. In this case both the number of living people as well as the time are considered resources.

In the following section, we will explain in details how to enhance the planning domain with risk-related information. This information will then be used by a risk-aware HTN planner during the planning process to come up with the desired plan.

3.2 Approach

As discussed in Section 2.2, an HTN planner works as follows. It starts from an initial task network and decomposes the compound tasks repeatedly until obtaining a sequence of primitive tasks, i.e, the desired plan. However, when decomposing a compound task, and at a specific world state, the planner can possibly have different methods that are applicable at the same time and are able to decompose this task. However, a good HTN planner should select a promising method to guarantee the plan quality instead of randomly choosing an applicable method.

As defined earlier, in this work, the quality of plans refer to their resource consumption and length. Hence, we associate every method with a utility value and use it as an indicator that guides the planning process. This utility is an estimated value of the resource consumption and it depends on the risk attitude of the corresponding compound task being decomposed by the method. Notice that although we have shown in Section 2.3 that a utility typically expresses a function of the gain, in this case we will use it to express the loss or the consumption of resources.

An HTN planner is usually provided with two separate input files. The first one includes the domain model, which describes the available methods and operators, whereas the second file is a description of a concrete problem in this domain. A problem description consists of the initial state of the system and the list of tasks that have to be achieved by the planner. Obviously, different problems with different goal tasks can have the same domain. Consider the domain description that was explained before in Section 2.2 where we have locations, robots, a truck and containers. Figure 3.1 shows a concrete problem in this domain in which we have 5 different locations with a robot in each of them and we have the containers and the truck located initially in the first and the fifth locations respectively. The goal of this problem is to transfer the containers to location 3. However, in this domain we could have another problem description that has a different number of locations, as well as a different initial state and goal tasks. Moreover, the same task can appear multiple times during the planning process whether with the same binding of its terms list or with different ones.

Building on these two ideas, our solution is based on two steps, which are: the pre-processing step and the main algorithm step. In the pre-processing, we compute the utilities of all tasks and methods once without the need to repeat the same calculations for different problems or for each occurrence during the planning process. However, when the search for the desired plan is performed in the main algorithm, the utility information obtained in the first step are used. By following the two-step approach, we can have an efficient planning algorithm with a one-time utility computation process for all tasks and methods of the given domain.

In the following, we explain in details the two steps of our approach.

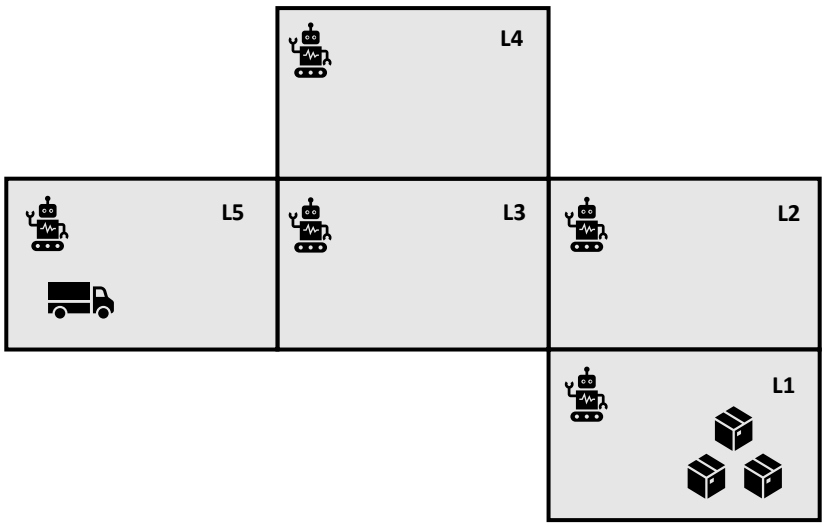


Figure 3.1: A description of a concrete problem in the "Truck example" domain. (Adopted from [GNT04])

3.2.1 Step One: Pre-processing

In order to compute the utilities in this step, we require the operators to be associated with a function of consumption as a non-negative value that expresses a single or combination of properties, such as an energy consumption and a failure rate. This cost expresses the resource consumption of applying this operator. Since we follow a two-step approach, we assume that the operator cost is determined when modeling the domain rather than at run time. The reason behind this assumption is that the bindings of the variables, which appear in the domain description, to concrete values are not known yet at this stage, but rather during the planning process itself, i.e., the second step. Thus, in order to be able to compute the utilities in the pre-processing step, the operator cost must be modeled as a constant function that is not related to a specific binding of the variables. Based on this requirement, we modify the definition of an operator (Definition 2.2.6) to include the function of consumption as follows:

Definition 3.2.1 (Operator)
An operator is a tuple $\langle ptask(o),\ pre(o),\ eff(o),\ c(o)\rangle$*, where* $ptask(o), pre(o)$ *and* $eff(o)$ *are defined as before but we add the cost of applying the operator* $c(o)$*, such that* $c(o) \in \mathbb{R}_{\geqslant 0}$. □

Starting from a compound task t_c, a *candidate decomposition path* is formed by adding this task to the path and at each decomposition level, one of its sub-tasks is also appended to this path. The same process is repeated until reaching either a primitive task or a task that is already in the path, i.e., forming a *candidate cycle*. Figure 3.2 serves as a demonstrative example of the candidate path and the candidate cycle in a domain description. As depicted here, $\langle t,\ t2,\ t8\rangle$ and $\langle t7,\ t18\rangle$ form two candidate paths. However, $\langle t6,\ t16,\ t6\rangle$ forms a candidate cycle. Since the $m8$ method decomposes $t6$ and has $t16$ in its task network, $m8$ is considered as a part of this candidate cycle.

The same applies to $m7$. The reason behind the naming "candidate" in these two concepts is given in the discussion after defining the estimated utilities of the compound tasks. In the following, we formally define the candidate decomposition path and the candidate cycle.

Definition 3.2.2 (Candidate decomposition path)
A candidate decomposition path $\pi \in \Pi$ is a sequence of tasks $\pi = \langle t_1, t_2, \ldots, t_n \rangle$, such that

- $n > 1$ *and*
- Π *is the set of all candidate decomposition paths and*
- $\forall\, t_i, t_{i+1} \in \pi, \exists m \in M$, *such that:*
  - $i \in \{1, 2, \ldots, n-1\}$ *and*
  - $t_i \in T_c$ *and* $ctask(m) = t_i$ *and*
  - $t_{i+1} \in tn(m)$ □

Definition 3.2.3 (Candidate cycle)
A candidate cycle $c \in C$ is a candidate decomposition path $\pi = \langle t_1, t_2, \ldots, t_n \rangle$, such that $t_1 = t_n$. □

Definition 3.2.4
A method m is said to be part of a candidate cycle c if and only if $\exists c = \langle \ldots, t_i, t_{i+1}, \ldots \rangle \in C$, such that:

- $ctask(m) = t_i$ *and*
- $t_{i+1} \in tn(m)$

Given a task t, the following function represents a template formula for estimating the utility function of t. This estimation is based on the risk or consumption attitude of t.

$$U(t) = \begin{cases} min_{o \in O:\, ptask(o) = t} c(\mathrm{o}), & \text{if } t \text{ is primitive;} \\ min_{m \in M_t} E(m) : E(m) \text{ is the estimation factor of the method } m, & \text{otherwise.} \end{cases}$$

A single primitive task can have multiple operators that differ in their preconditions and costs. This justifies our definitions of the primitive task (Definition 2.2.5) and the operator (Definition 2.2.6) where we have a one-to-many relationship between them. Following these definitions, the utility function of the primitive task is defined as the smallest cost of all operators that can execute this task. However, to concretely define the previous utility function in case of the compound task, we define the estimation factor $E(m)$ to be related to the risk attitude of the compound task. We distinguish four different attitudes towards risk, which are: (i) the risk-averse attitude (ii) the risk-seeking attitude (iii) the risk-neutral attitude (iv) the consumption-aware attitude. The first three risk attitudes are defined in the utility theory. However, since we focus in this study on the consumption-aware domain, the fourth attitude is added. In the following, we will define a formula to calculate $E(m)$ in each of these four cases.

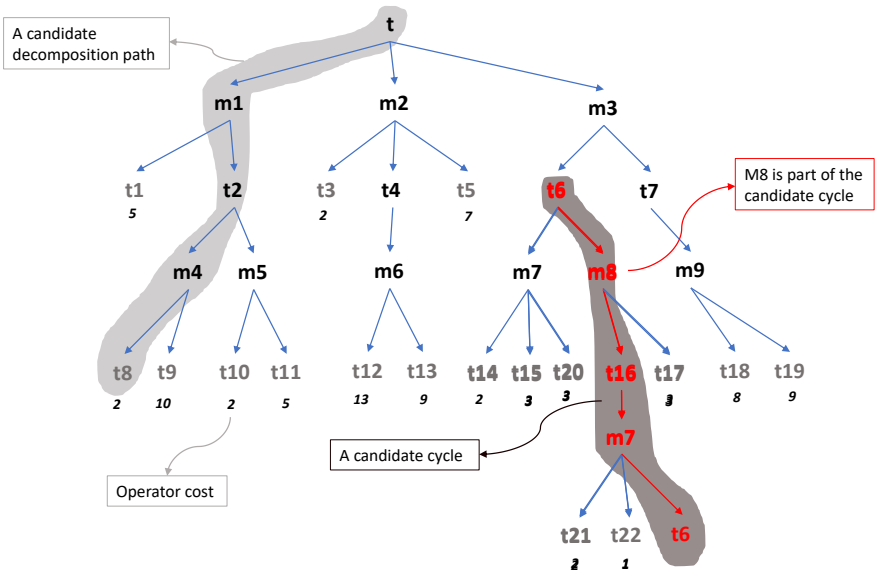


Figure 3.2: A demonstrative example of a candidate decomposition path and a candidate cycle.

Estimation Factor Definition Based on Different Risk-Attitudes:

Consumption-aware domains are characterized by their sensitivity towards losing *resources*. Examples of resources include money, fuel, equipment, time, etc. Thus, the resource value could be any value that expresses the amount of resource that the planning algorithm should try to preserve. Thus, in the following definitions of the method estimation factor, we distinguish two cases, the first one is when the method is not part of a candidate cycle. In this case the estimation factor is computed based on the risk attitude. On the other hand, if the method is part of a candidate cycle, its estimation factor is assigned the resource value. The reason behind this choice is explained in the Section 3.2.1 part after defining these estimation factors formally.

- **Risk-averse attitude:** A task is *risk-averse* if and only if it minimizes the maximum expected value of its methods' sub-tasks. A risk-averse task, as the name indicates, is a task that avoids risky choices. That is, it prefers the safest possible decision, even if it would eventually be associated with a bigger cost, over the decisions that are risky but may give a smaller cost. Thus, this type of tasks follows a pessimistic behavior by considering only the worst possible case, as follows:

$$E(m) = \begin{cases} max_{t' \in tn(m)} U(t'), & \text{if } m \text{ is not part of a candidate cycle;} \\ Resource, & \text{if } m \text{ is part of a candidate cycle.} \end{cases}$$

- **Risk-seeking attitude:** A task is *risk-seeking* if and only if it minimizes the minimum expected value of its methods' sub-tasks. A risk-seeking task, as the name suggests, is a task that prefers a risky choice over the safer ones if this choice may increase its gain, i.e.,

may lower the cost. Thus, this type of tasks shows an optimistic behavior towards risk by considering only the best possible case, as follows:

$$E(m) = \begin{cases} min_{t' \in tn(m)} U(t'), & \text{if } m \text{ is not part of a candidate cycle;} \\ Resource, & \text{if } m \text{ is part of a candidate cycle.} \end{cases}$$

- **Risk-neutral attitude:** A task is *risk-neutral* if and only if it minimizes the average of the expected utility values of all sub-tasks of a given task's method. A risk-neutral task is indifferent towards risk, i.e., it ignores the risk associated with each alternative when making a choice. The estimation factor of this type of tasks' methods is defined as follows:

$$E(m) = \begin{cases} \frac{\sum_{t' \in tn(m)} U(t')}{|tn(m)|}, & \text{if } m \text{ is not part of a candidate cycle;} \\ Resource, & \text{if } m \text{ is part of a candidate cycle.} \end{cases}$$

- **Consumption-aware task:** A task is *consumption-aware* if and only if it minimizes the sum of the expected utility values of its methods' sub-tasks. The estimation factor, in this case, is defined as follows:

$$E(m) = \begin{cases} \sum_{t' \in tn(m)} U(t'), & \text{if } m \text{ is not part of a candidate cycle;} \\ Resource, & \text{if } m \text{ is part of a candidate cycle.} \end{cases}$$

Discussion:

When decomposing a task t_c during the planning process, i.e., the second step, the planner has to choose one of the task's applicable methods and it performs the decomposition process repeatedly until finding a plan. However, if the algorithm backtracks at some point, i.e., cannot find a plan using this method, then it will try other methods. In case the planning algorithm is instructed to find the first possible plan, it stops when finding a solution and does not try all possible alternatives. However, in the pre-processing step the algorithm computes the utilities of all compound tasks in the domain description. Hence, it goes through all possible alternatives. This makes every decomposition path in this step a *candidate* for being a decomposition path during the planning process too. Moreover, in the planning process, all variables in the methods' and operators' preconditions and in the tasks' term lists are bound to some constant values of the concrete problem that is provided to the planner. However, in the pre-processing step, we work on the domain description rather than on a concrete problem. Thus, we consider the repetition of a task in a *candidate* decomposition path as a cycle regardless of a specific binding. This makes every cycle in the domain description a *candidate* for forming a real cycle in the planning process.

Figure 3.2 shows an example of a domain description. Let us assume that $t6$ has two variables x, y in its term list $terms(t6)$. In this case, ($t6$, $t16$, $t6$) forms a candidate cycle, which is not considered a cycle in the planning process if at least one of the terms takes a different value in each appearance of the task. However, if the two terms take the same value in the two occurrences of the task then this is a cycle. Figure 3.3 illustrates this idea. In the first case, we have a candidate cycle in the domain description while in the second case the same decomposition path does not lead to a cycle due to the different bindings. However, in the third case it does lead to a cycle due to the identical bindings.

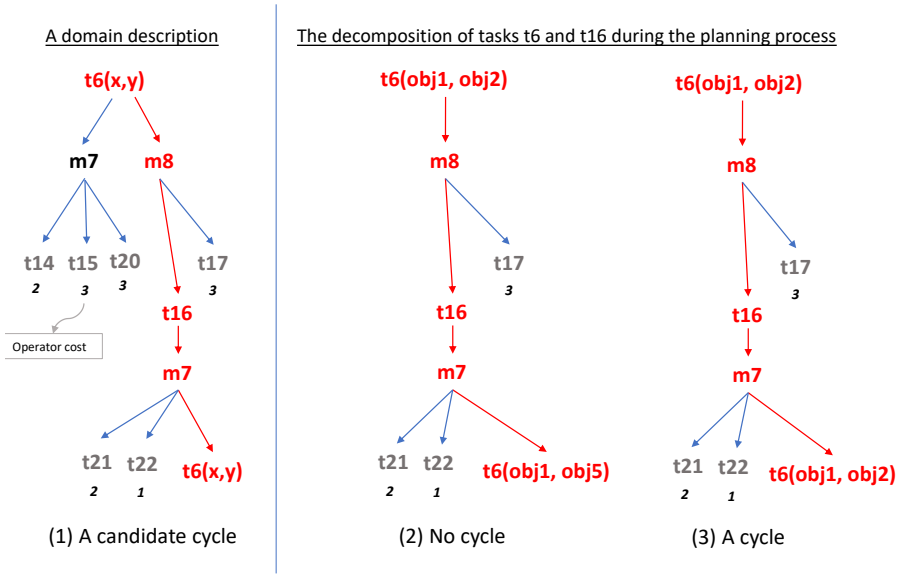


Figure 3.3: An example of the difference between candidate cycles in a domain description and cycles in a concrete problem.

Originally, an HTN planner chooses the first applicable method when decomposing a compound task. However, in this work, we make the planning algorithm choose the method that seems to be the most promising one considering its consumption (risk) value. Thus, the methods of a compound task are sorted in an ascending order based on their estimation factor so that the algorithm performs the decomposition starting from the method with the smallest estimation value. This explains our choice of the method's estimation factor to be equal to the resource value in the case where this method is part of a candidate cycle. Assigning this value to the estimation factor makes the planning algorithm avoid choosing such kind of methods as long as it has better choices. However, it should be noted that avoiding the methods that are part of a candidate cycle should hold true in all cases regardless of the risk attitude itself because choosing this kind of methods does not promise any better solution than avoiding it.

3.2.2 Step Two: The Main Planning Process

Generally, HTN planners aim at finding a sequence of operators that can achieve the goal task network without taking the resource consumption into account. However, as discussed earlier, in risk-sensitive domains, taking the resource value into consideration during the planning process is of a high importance. Hence, if the cost of the partial plan found during the planning algorithm exceeds the available resource, the algorithm should backtrack and try other alternatives, if any; consequently, the resource value serves as an *upper bound* to the planning algorithm's execution.

The main enhancement in our planning process is to consider the method estimation factors while searching for a plan. Such that, methods that have small estimation factor are preferable over others. To achieve that, the methods are sorted in an ascending order so that the planning algorithm firstly chooses the method with the best estimated value.

Discussion about the Resource and the Algorithm Execution Upper Bound:

Consumption-aware resources are characterized by their sensitivity towards losing resources. The main planning algorithm tries to preserve the available resource by taking the task utilities into account. Moreover, during the planning process, each application of an operator decreases the available resource by an amount equals to the cost of this operator. If the algorithm reaches a point where the resource value is fully consumed but the plan is still not found, the algorithm backtracks to try other alternatives.

If the planning algorithm enters a cycle, it may loop in this cycle infinitely if there is no bound that limits the algorithm execution. However, if there is a resource consumption through the cycle, the algorithm loops in it until the resource is exhausted, hence the algorithm backtracks from being in the cycle. Thus, we define the resource value to be an upper bound of the algorithm execution.

3.3 Algorithm

In this section, we explain in details the algorithm we use in each of the two steps.

3.3.1 The Pre-processing Algorithm

The pre-processing step starts from Algorithm 3.1. The "findUtilities" function takes the domain description as an input, i.e., the set of operators and the set of methods. Having these two sets, we infer the sets of primitive and compound tasks (lines 2 and 3). Since each primitive task can be executed by a number of operators, which differ in their preconditions and costs, we have to choose one of these costs to be assigned as this task's cost. Our goal is to minimize the resource consumption; therefore, we select the smallest operator's cost as the estimated cost of the primitive task. This is done in lines 4-6 in the algorithm. The algorithm initializes the task utilities and the method estimation factors and then it invokes the "findTaskUtility" function that computes the tasks' utilities. However, as we are iterating over all compound tasks in order to compute the utilities by using this function, it may happen that one of the tasks gets its utility computed while computing another task's utility due to its existence in the same candidate decomposition path with this task. Thus, before computing the task's utility, we check whether this value is already computed (line 12).

The "findTaskUtiltiy" function, as we can see in Algorithm 3.2, has the following inputs: (i) the $task$ for which the estimated utility will be computed, (ii) a $path$, (iii) the set of all methods from the domain description, (iv) The risk attitude, and (v) the available resource Re. The $path$ is a candidate decomposition path that includes all the previously visited tasks until reaching the current $task$.

The algorithm starts from a compound task and iterates over all the methods that can decompose it in order to compute their estimation factor (lines 11-21). However, if this estimation factor is already computed, it must not be computed again (see line 12). The algorithm then iterates over the sub-tasks in each method's task network and invokes the same function recursively to compute the utilities of these sub-tasks (line 13-15). The estimation factor of each method is calculated based on the task utilities of the method's task network. Moreover, its concrete function depends on the risk

attitude. The estimation factors are computed using the *computeEstimationFactor* method. Before assigning the value of the method's estimation factor, the algorithm examines again whether this value is already computed (line 17). The reason behind this repeated examination is that this task might be part of a candidate cycle and the algorithm may realize that during the recursive calls. Thus, one of its methods, which might be the current one could have been assigned an estimation factor value that equals to the resource value. Thus, we do the second examination to avoid erasing the already computed estimation factor.

The utility of the current task is computed after iterating over all the methods that can decompose it. Since our goal is to minimize the resource consumption, the algorithm returns the minimum of the method estimation factors (line 22).

Cycles in this algorithm are detected by using the $path$ sequence. A copy of this sequence is passed from a task to the tasks in its method task networks, such that a cycle can be found if the same task exists formerly in the $path$. When the algorithm detects a recursive task, i.e., a task that is part of a candidate cycle, it assigns all the estimation factors of the methods that form this candidate decomposition path to the resource value, starting from the recursive task in the $path$ (lines 5-9).

Next, we will explain the main planning algorithm that takes the information computed by the pre-processing algorithm into account.

3.3.2 The Main Planning Algorithm

Our main planning algorithm is a modification of the SHOP2 planning algorithm [NCLM99]. As shown in Algorithm 3.3, the input of our algorithm includes: (i) the state of the world, (ii) the remaining tasks to be decomposed, (iii) the domain description, (iv) the plan reached so far, and (v) the available resource.

The algorithm runs continuously until a plan is found, i.e., the set of the remaining tasks $tasks$ gets empty, or an empty set is returned, hence no solution was found. At each time, the algorithm takes the first task t from the remaining-tasks set $tasks$. If this task is a compound task, the algorithm searches for all applicable methods that can decompose it in the current state. If no such method is found, the algorithm fails to find a plan and returns the empty set (line 36). However, if one or more alternatives are found, the $Sort$ function (line 27) sorts these methods in an ascending order based on their estimated factor and assigns the result to the sequence $sortedM'$. Then, the algorithm invokes the same function, recursively, starting from the method with the smallest estimated factor. It should be noted that in this invocation, the chosen method's task network is appended to the remaining tasks (see line 29). The decomposition process continues until reaching a primitive task. In this case, the algorithm searches for an applicable operator in the current state. If no such operator is found, the algorithm backtracks and tries other alternatives. However, as in the methods case, if one or more operators are found, the algorithm sorts them in an ascending order and assigns them to the $sortedO'$ sequence and applies the one with the least cost only if the amount of the available resource is not exceeded. Otherwise, it returns an empty set which forces the algorithm to backtrack.

Algorithm 3.1 Initialize tasks and invoke the algorithm that computes the utilities

```
1:  function findUtilities(O, M, riskAttitude, resource) return ()
2:      T_p ← t_p : ∃o ∈ O : t_p = ptask(o)
3:      T_c ← t_c : ∃m ∈ M : t_c = ctask(m)
4:      for each t_p ∈ T_p do
5:          cost(t_p) ← min_{o∈O: ptask(o)= t_p} cost(o)
6:      end for
7:      for each m ∈ M do
8:          estimationFactor(m) ← -1
9:          utility(t_c: ctask(m) = t_c) ← -1
10:     end for
11:     for each t_c ∈ T_c do
12:         if utility(t_c) = -1 then
13:             path ← null
14:             utility(t_c) ← findTaskUtility(t_c, path, M, riskAttitude, resource)
15:         end if
16:     end for
17: end function
```

Algorithm 3.2 Compute utilities

```
1:  function findTaskUtility(task,path, M, Attitude, Re) return (taskUtility)
2:      if task is primitive then
3:          return cost(task)
4:      else
5:          if ∃t_i ∈ path = ⟨..., t_i, ..., t_n⟩ : task = t_i then
6:              M' = {m' : ∃t_j, t_{j+1} ∈ path ∧ t_j = ctask(m') ∧ t_{j+1} ∈ tn(m') ∧ j ≥ i}
7:              for each m' ∈ M' do
8:                  estimationFactor(m') ← Re
9:              end for
10:         end if
11:         for each m ∈ M : ctask(m) = task do
12:             if estimationFactor(m) = -1 then
13:                 for each subT ∈ tn(m) do
14:                     copy ← copy(path).append(task)
15:                     utility(subT) ← findTaskUtility(subT,copy,M,Attitude,Re)
16:                 end for
17:                 if estimationFactor(m) = -1 then
18:                     estimationFactor(m) ← computeEstimationFactor(m,Attitude)
19:                 end if
20:             end if
21:         end for
22:         Result ← Min_{m∈M: ctask(m)=task} (estimationFactor(m))
23:         return Result
24:     end if
25: end function
```

Algorithm 3.3 Find a sequence of operators/ a plan (adapted from SHOP2 algorithm [NCLM99])

```
1:  function findPlan(state, tasks, domain, plan, Resource)
        return (plan)
2:      if tasks = ∅ then
3:          return plan
4:      end if
5:      t ← the first task in tasks.
6:      R ← the remaining tasks.
7:
8:      if t is primitive then
9:          O′ = {o ∈ domain.O : ptask(o) = t ∧ pre(o) ⊆ state}
10:
11:         if O′ is not empty then
12:             sortAscendingly(O′, sortedO′)      // order the applicable operators ascendingly
13:             first ← POP(sortedO′)
14:             Resource ← Resource − c(first) // reduce the operator's cost from the resource
15:
16:             if Resource < 0 then           // check if the available resource is fully consumed
17:                 return ∅
18:             end if
19:             return findPlan(state[first], R, domain, append(plan, first), Resource)
20:         else
21:             return ∅
22:         end if
23:     else
24:         M′ = {m ∈ domain.M : ctask(m) = t ∧ pre(m) ⊆ state}
25:
26:         if M′ is not empty then
27:             sortAscendingly(M′, sortedM′)      // order the applicable methods ascndingly
28:             for i ← 1 to size(sortedM′) do
29:                 result ← findPlan(state, append(R, tn(sortedM′(i))), domain, plan, Resource))
30:
31:                 if result ≠ ∅ then
32:                     return result
33:                 end if
34:             end for
35:         end if
36:         return ∅
37:     end if
38: end function
```

3.4 Running Example

Consider the domain description illustrated in Figure 3.4. This domain models a problem of transporting a container from one location to another using a truck. To achieve that, we need to get the container to the same location as the truck. After that, the container is taken and loaded to the truck then the truck has to move the container to the target location where the container is unloaded and located. The domain is modeled such that the locations form a square grid and there can be shortcuts between different places.

The *meet-container-truck* compound task represents the first step of the whole process and it can be decomposed by two methods. These are: sending the container by post to the truck's location or moving the truck to the container's location. Sending the container by post includes writing the target address, doing online stamping, packaging and going to the post whereas moving the truck to a target location can be achieved by three different methods (m_4, m_5, m_6). If the truck is already in the final location, then nothing has to be done. Otherwise, if there is a shortcut from the current truck's location that can lead directly to the destination, then the truck can take this route. The third method is to go one step in one direction while considering not exceeding the grid size and then do the $move$ compound task, recursively.

Given the current domain with all operator costs, which are depicted in Figure 3.4 in red under the primitive tasks, the *findUtilities* algorithm computes the task utilities and the method estimation factors depending on the different risk attitudes. Since each method of m_7, m_8, m_9, m_{10}, m_6, m_4 decomposes the corresponding compound task into a single primitive task, then their estimation factors pertain their values regardless of the risk attitudes. These values are all ones except for method m_4 that has the value of 0 as an estimation factor. When the algorithm execution reaches the $move$ task in the m_5 method's task network, it finds out that this task is part of a candidate path. Thus, method m_5's estimation factor takes the resource value regardless of the risk attitude. However, method m_2' estimation factor takes the values 0, 50, 13.25, 53 in the risk-seeking, risk-averse, consumption-aware and risk-neutral risk attitudes, respectively. The same applies for methods m_{11}, m_{12}, m_1, which take different values for the different risk attitudes. After computing the estimation factors for all the methods that can decompose a task, the algorithm takes the minimum value as the utility of this task. For example, the $meet - container - truck$ task utility equals 0, which is the minimum value among m_2 and m_3 estimation factors.

After obtaining the task utilities and the method estimation factors, the main planning algorithm can take this information to find a plan. As discussed in Section 3.2.2, when JSHOP2 has to choose from multiple methods to decompose a task, it always chooses the first one. However, in UASHOP, the methods are ascendingly order based on their estimation factors. Hence, the two algorithms behaves differently even if they are given the same domain and problem. Consider the problem represented in Figure 3.5, which has the truck-container domain (see Figure 3.4). We assume that the algorithm has already finished the first two steps and the truck with the container are at the same location and the container has to be moved to the destination. Assuming that the methods that can decompose the $move$ task are ordered originally as follows m_4, m_5, m_6, JSHOP2 tries m_4 as the first option but since the container is not already in the target, this method is not applicable in this state of the world. Therefore, the algorithm considers m_5, which is applicable in this case, hence, it decomposes the $move$ task using this method into two tasks $go - step$ and a recursive $move$ task. Since $go - step$ is also a compound task, the algorithm decomposes it again using the first applicable method, which is $go - north$ in this case, assuming that the methods are ordered

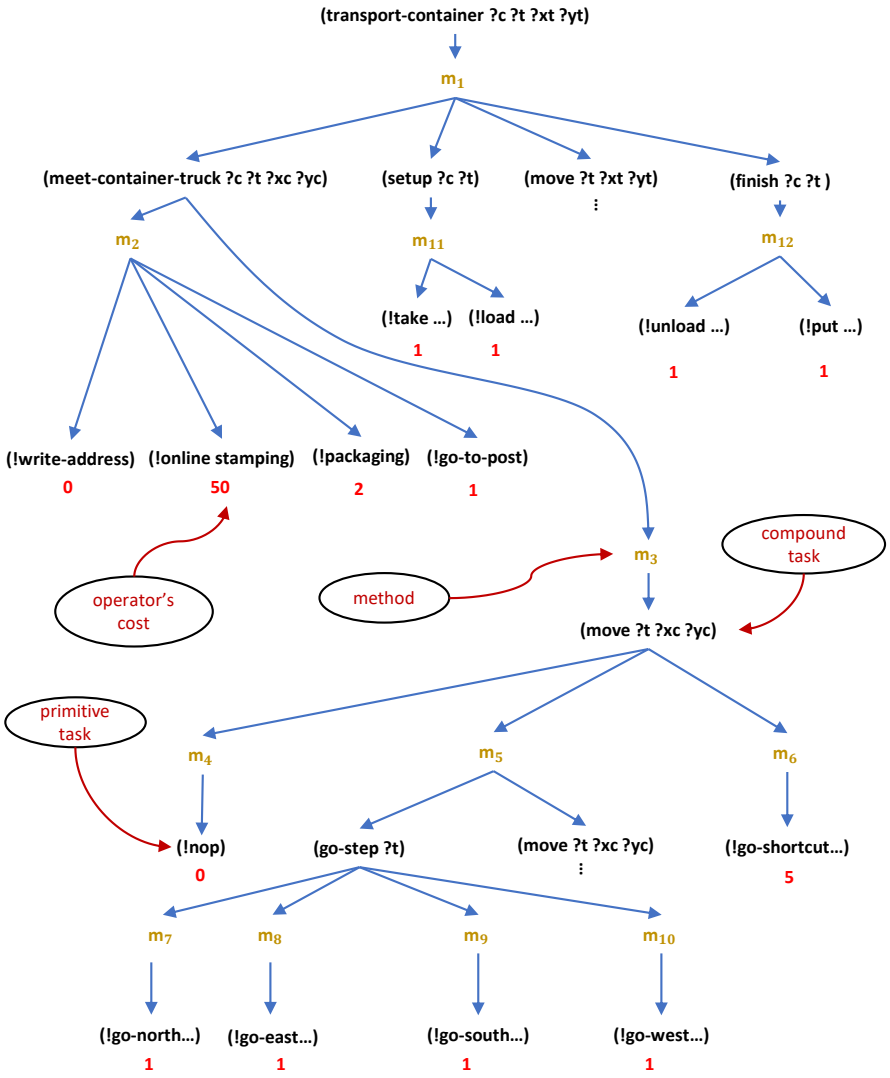
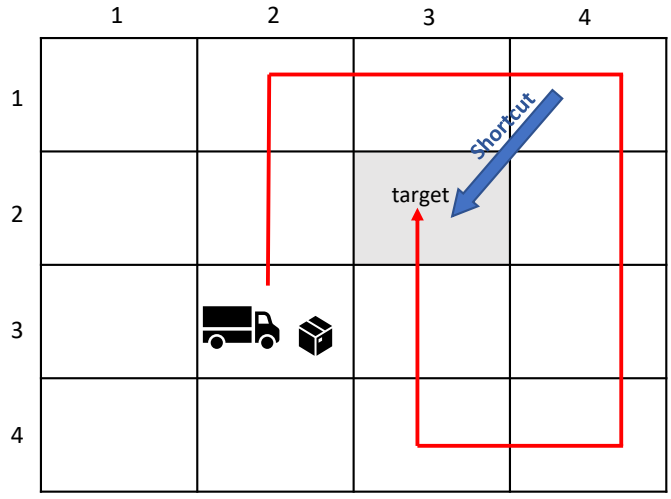
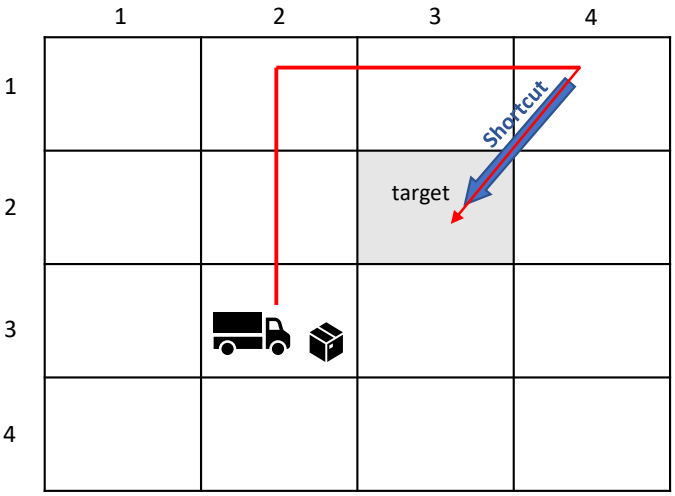


Figure 3.4: A domain description of the truck-container example. In this figure, `c` refers to the container, `t` refers to the truck, `xc` and `yc` refer to the coordinates of the container, and finally, `xt` and `yt` refer to the coordinates of the final target.

from left to right as depicted in Figure 3.4. The algorithm then decomposes the second task, which is again *move*. The same execution of going one step using one of the applicable direction methods and then decompose the *move* task again repeats until the truck reaches the target, hence the first method becomes applicable and leads to the no operation (!nop) primitive task. This execution produces a sequence of operators represented by the red line in Figure 3.5a. This plan has a cost of 10. On the other hand, when UASHOP has the same problem of choosing between these three methods to decompose the *move* task, it will order them as follows: m_4, m_6, m_5. The first two methods remain inapplicable until the truck reaches the (1,4) location in which there exists a direct shortcut to the target. Thus, m_6 is applicable and since it is preferable over m_5, the algorithm applies



(a) Regular JSHOP2 does not take the shortcut and searches for the target by moving one step and recursively calling the same method to move.

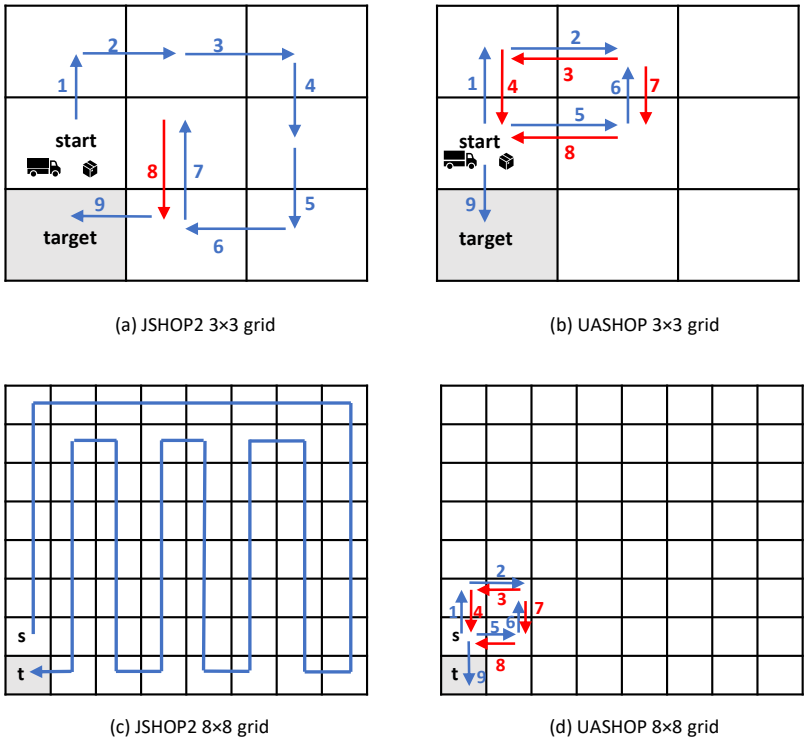


(b) UASHOP takes the shortcut and produces a better plan than JSHOP2.

Figure 3.5: An example that illustrates the difference in behavior between JSHOP2 and UASHOP when solving a problem in the truck-container domain.

it. The algorithm, in this particular problem where we assume that the resource >= 4, results in a plan that has a cost of 4. Therefore, choosing the methods based on their estimation factors can lead to plans with a smaller cost compared to the original JSHOP2.

UAHSOP tries to find solutions that minimize the resource consumption. Thus, a plan found in JSHOP2 is considered invalid by UASHOP, if it exceeds the available resource. Figure 3.6 shows a comparison example that illustrates the difference in behavior between the two algorithms when having a small resource given to UASHOP algorithm. In this example, we assume that the available resource equals to 2. Thus, since we do not have shortcuts here and the cost of each direction operator is 1, UASHOP backtracks after two steps if it could not reach the goal and continues doing



(a) JSHOP2 3×3 grid

(b) UASHOP 3×3 grid

(c) JSHOP2 8×8 grid

(d) UASHOP 8×8 grid

Figure 3.6: An example that illustrates modeling a problem with a resource in UASHOP compared to JSHOP2 where the available resource is not considered. Numbers show the ordering of the execution where red arrows represent the backtracking. The resource= 2.

this until reaching a goal in one step to the south. On the other hand, JSHOP2 keeps searching and ends up visiting the whole grid until reaching the goal. This difference becomes clearer when having a bigger grid; 8×8, for example. As we can see, the execution time and the cost of the resulting plan in JSHOP2 is much bigger than in UASHOP.

As discussed earlier, UAHSOP minimizes the estimated utility while taking the risk-attitudes of the tasks into account. Trying to solve the same problem in the same domain with different risk-attitudes may result in different solutions. **??** shows an example of a problem representing the first step of the truck-container domain where we want to deliver the container to the truck. In order to achieve that, there are two options, i.e., methods, that include either sending the container by post or moving the truck to the container's location; methods m_2 and m_3 in Figure 3.4, respectively. Since the risk-seeking behavior is being pessimistic in its choice of the method to use. Thus, in this example, it will be indifferent between the two choices M_2 and m_3, since it assigns a value of 0 to both method estimation factors. In this case, choosing the method to start with follows the original ordering of the methods, hence, it will choose to send the container by post and end up having a plan with 53 as a cost. On the other hand, in this particular example, all other risk-attitudes will behave the same because the estimation factor value of m_3 is always 0 while it is bigger than 0 for m_2. Therefore, all risk-attitudes except the risk-seeking one will choose to move the truck to the container, which leads to a plan of cost 8 but with a bigger execution time compared to the sending by post method.

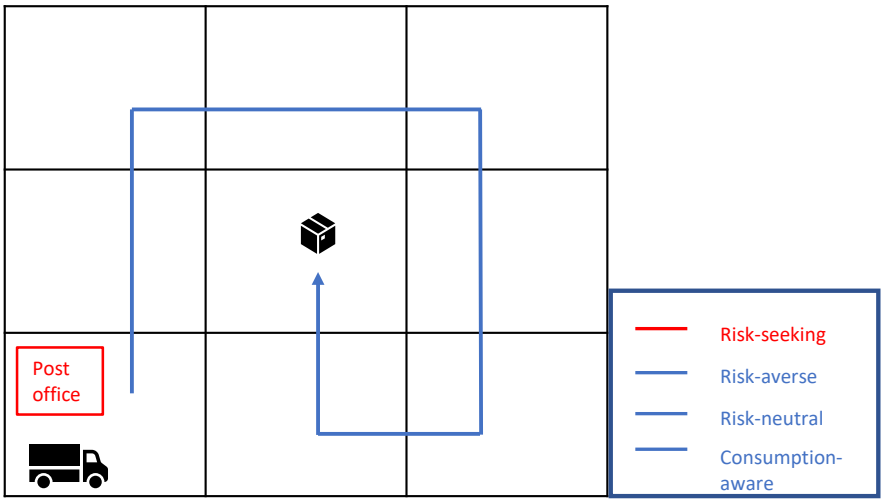


Figure 3.7: An example that illustrates the differences between the risk attitudes when solving a problem in UASHOP.

4 Implementation

In the previous chapters, we explained the theoretical framework of the utility-aware HTN planner through a set of definitions. Furthermore, we presented the approach and the algorithms that we use in order to achieve the goal of considering utilities during the planning process.

In this chapter, we validate our suggested approach by showing how we implemented it[1]. We start by giving an overview of the framework we used and introducing the HTN planner we used as a basis. After that, we explain the usage of the application and detail the different phases required to work with it. Then, we show the structure of the program in two class diagrams. Finally, we describe the input file structures, i.e., the domain and problem files.

4.1 Overview

As discussed in Section 2.2, due to HTN planning popularity in solving real-world problems, many HTN planners were implemented in different languages and frameworks. In this work, we extend an existing HTN planner called Java Simple Hierarchical Ordered Planner 2 (JSHOP2) in order to support solving problems in risk-sensitive domains. JSHOP2 is the java implementation of Simple Hierarchical Ordered Planner 2 (SHOP2) that is in turn the successor of Simple Hierarchical Ordered Planner (SHOP) [Ilg06; NAI+03; NCLM99].

JSHOP2 is a state-based HTN planner, hence it knows the current state of the world at each step during the planning process. This knowledge enables it to increase its domain description expressive power, e.g., using numeric computations in the method precondition, for example. Thus, it can be used in complex application domains [Ilg06].

Most domain-independent planners that accept a domain-specific knowledge to help guiding the planning process, are interpreters to their input, i.e., they interpret the provided domain description and the domain-specific information and act accordingly. However, what makes JSHOP2 stand out is that it compiles the domain description rather than just interpreting it. The compilation process uses some domain-independent templates that describe the structures that can appear in the domain description and as a result, it produces a domain-specific planner, which is tailored for this specific domain. One advantage of implementing the planner in this way is to improve the performance. For example, having a list in a general-purpose piece of code that implements the domain description means we need to have a list data structure of a varying size. However, compiling the domain description into a domain-specific planner allows us to know the size of this list in advance, hence we can use an array of a specific size instead, which in turn increases the efficiency of the code [IN03].

[1] The implementation is available on Github at: https://github.com/Ebaa-Alnazer/Utilities-JSHOP2.

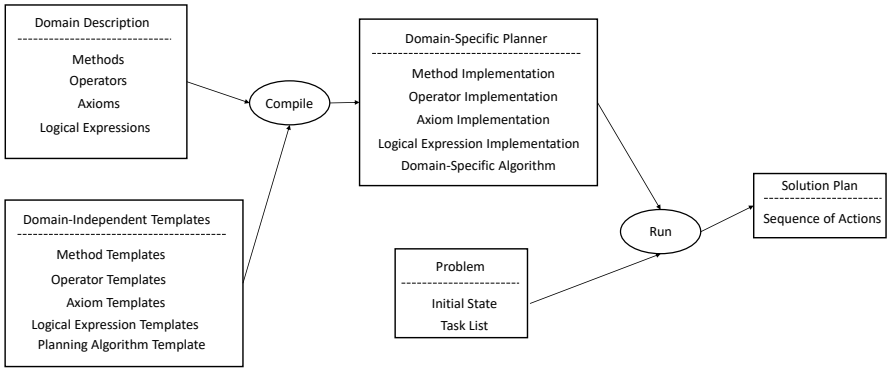


Figure 4.1: The compilation process of JSHOP2 (Adopted from [Ilg06])

Figure 4.1 illustrates this compilation process. As a rough idea, we can see that JSHOP2 takes the domain description as an input and since it has the templates of all structures in the domain in addition to the main planning algorithm, the compilation process produces a domain-specific planner. The details of this process will be explained later.

4.2 Usage

The steps required to run the algorithm against the two input files of the domain and the problem description are divided into two main consecutive phases, which are the preparation phase and the execution phase. The main reason behind these two phases is to produce a domain-specific planner that is tailored to work with the provided domain and this is done in the first phase. After that, in the second phase, the generated domain-specific planner is executed for a concrete problem in this domain to find solutions. In the following, we explain each of these phases in detail.

4.2.1 Phase One: Preparation Phase

Figure 4.2 presents the preparation steps that have to be done before running the planner for a concrete problem in order to produce the solution. This preparation phase consists mainly of three steps, which are explained in the following.

Step One: Generating a Lexer and a Parser

In order for the planner to work, the user has to provide it with two files; one has the domain description while the other has the problem description. These input files are written in a language used to describe the different structures that can appear in them. The syntax of these different structures is explained in Section 4.4, in detail. However, in order to translate the input files and convert them into programming-level objects that we can deal with in our program, we need to have a lexer and a parser tailored to handle the structured language of these files. This is done by using a tool called ANother Tool for Language Recognition (ANTLR tool) [Par14]. ANTLR is a powerful

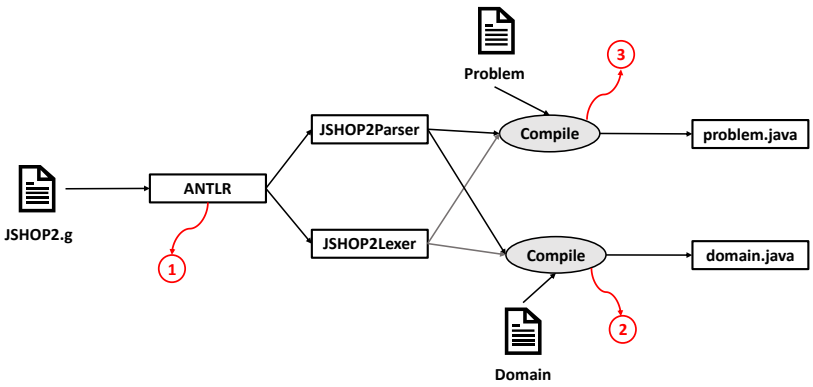


Figure 4.2: The three steps of the usage preparation phase.

tool for generating a lexer and a parser in a target language [Par14]. As we can see in Figure 4.2, ANTLR takes as an input a file (JSHOP2.g) that contains the language grammar used to write the domain and problem descriptions and then produces a parser and a lexer, which are named here JSHOP2Parser and JSHOP2Lexer, respectively.

Step Two: Compiling the Domain Description

Figure 4.2 shows that the next step after generating the parser and lexer is to compile the domain description input file in order to generate a Java class (domain.java). This class implements the functionality described in this input file and is necessary to handle this domain in our program.

After this step, the final preparation step follows.

Step Three: Compiling the Problem Description

The same compilation process applies for the problem description in order to obtain a Java class (problem.java). The only difference compared to the previously generated Java class is that the problem class is an executable class that uses an programming-level object of the generated domain class. An execution of this generated problem class represents the second phase of the whole process.

4.2.2 Phase Two: Execution Phase

This phase follows the domain and the problem Java classes generation steps. The purpose of this phase is to execute the problem Java class that has an object of the domain class, as illustrated in Figure 4.3. Running the problem generated class leads to an invocation of the pre-processing methods followed by the plan searching method to come up finally with the desired plan for the concrete problem in the domain.

A detailed explanation on how to perform the aforementioned phases using a command-line can be found in Appendix B.

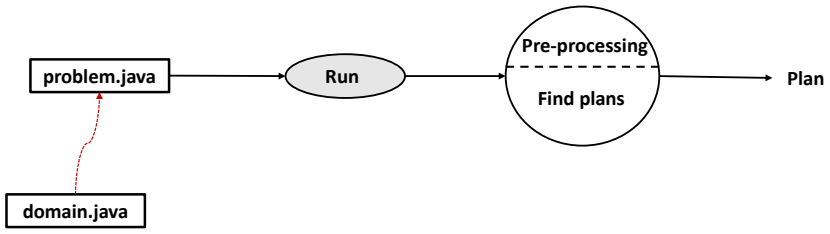


Figure 4.3: The second preparation phase.

4.3 Program Structure

In the following, we give an overview of the program structure using class diagrams where we present the main classes. As explained in Section 4.2, the first phase includes the compilation steps while the second phase is the execution phase. Thus, the classes fall into three categories, such that some of them are used in the compilation steps and some are used in the execution phase while others are used in both phases.

Figure 4.4 illustrates part of the classes used in the compilation phase. As we can see, the *JSHOP2Parser* has an object of the *InternalDomain* that it uses to assign the *InternalDomain* data structures, such as the *InternalOperator*, the *InternalMethod*, to values taken from the domain and problem input files. Furthermore, it invokes two methods in the *InternalDomain* to generate the Java classes for the domain and problem descriptions. Thus, the *InternalDomain* class represents any domain at the compilation time. Note that classes like *Term* and *Predicate* are used in both the compilation and the execution phase.

Another set of classes is presented in Figure 4.5 where we see the *problem* and *domain* classes that were generated by the *InternalDomain*. Since we want to be able to execute the classical algorithm, as well as our utility-aware algorithm for evaluation purposes, we use the *Factory design pattern*, such that the *Factory* decides the right class instance to be returned depending on the entered user input. These classes are *JSHOP2* and *UASHOP*. This is done in the *getAlgorithm* method of the *Factory* class. Moreover, since we want to avoid having many instances of this class, we used the *singleton design pattern* for it. The same design pattern is also used for both *JSHOP2* and *UASHOP* classes that implement the classical and the utility-aware algorithms, respectively. Classes such as *Domain*, *Method* are runtime classes.

4.4 Structure of Input Files

As discussed earlier in this chapter, the user provides the program with two input files, namely the domain description and the problem description. In this section, we explain the language syntax used in these files to describe the different structures that can appear in them.

- **Symbols:** Some structures have unique symbols, as follows:
  - the variable symbol is any symbol that starts with a question mark, e.g., ?x is a variable symbol.

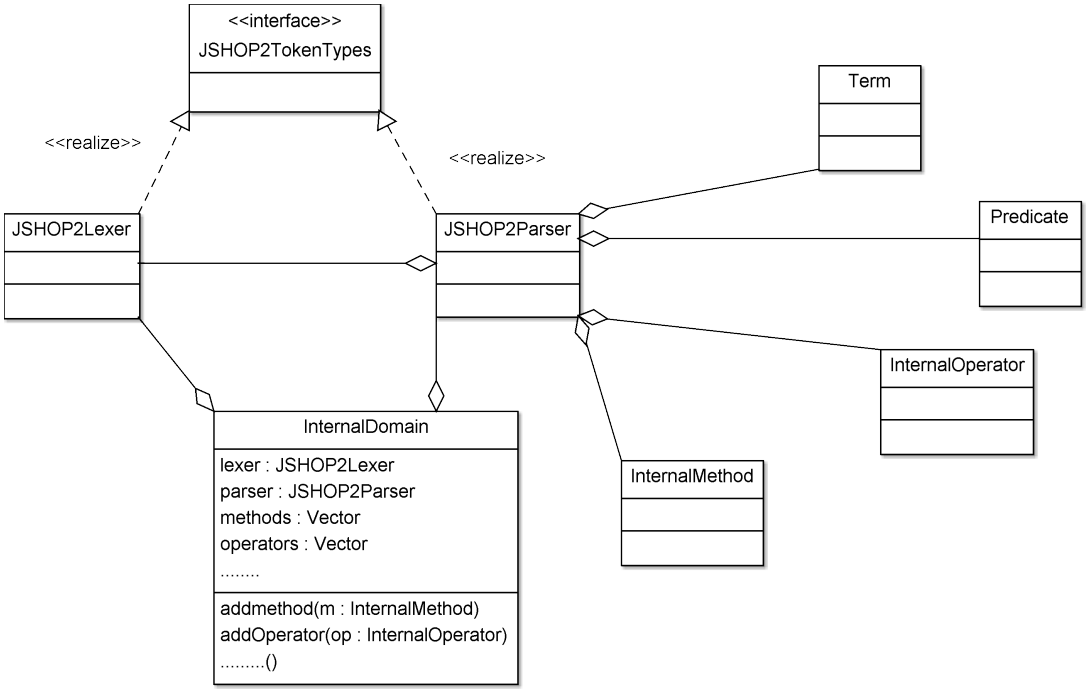


Figure 4.4: A class diagram that shows part of the classes used in the compilation phase of the program.

  - the primitive task symbol is any symbol that begins with an exclamation mark, e.g., !put
  - the compound task, predicate and constant symbols are defined as any symbol whose first character is either a letter or an underline.

- **A term** can be one of the following:
  - a variable
  - a constant, which is either a number or a specific object in the domain, e.g., the truck, location1
  - a call term that has the following form: $(call\ f\ t_1\ t_2\ \ldots\ t_n)$, where:
    - f is either a *function symbol* or a *built-in function*.
    - t_i is a term.

    Whenever a call term appears in the input file structures, it is replaced with its equivalent of applying the function f to the terms $t_1\ t_2\ \ldots\ t_n$. This happens during the planning process after binding the call term variables, i.e., after it becomes a ground term.

    There are a set of built-in functions, such as $+, -, /, *$. For example, using the built-in function $-$, the call term (call - ?x ?y) evaluates to 8 if the two variables ?x and ?y are bound to 10 and 2, respectively, during the planning process. In addition, functions can be defined externally and called in the domain description.

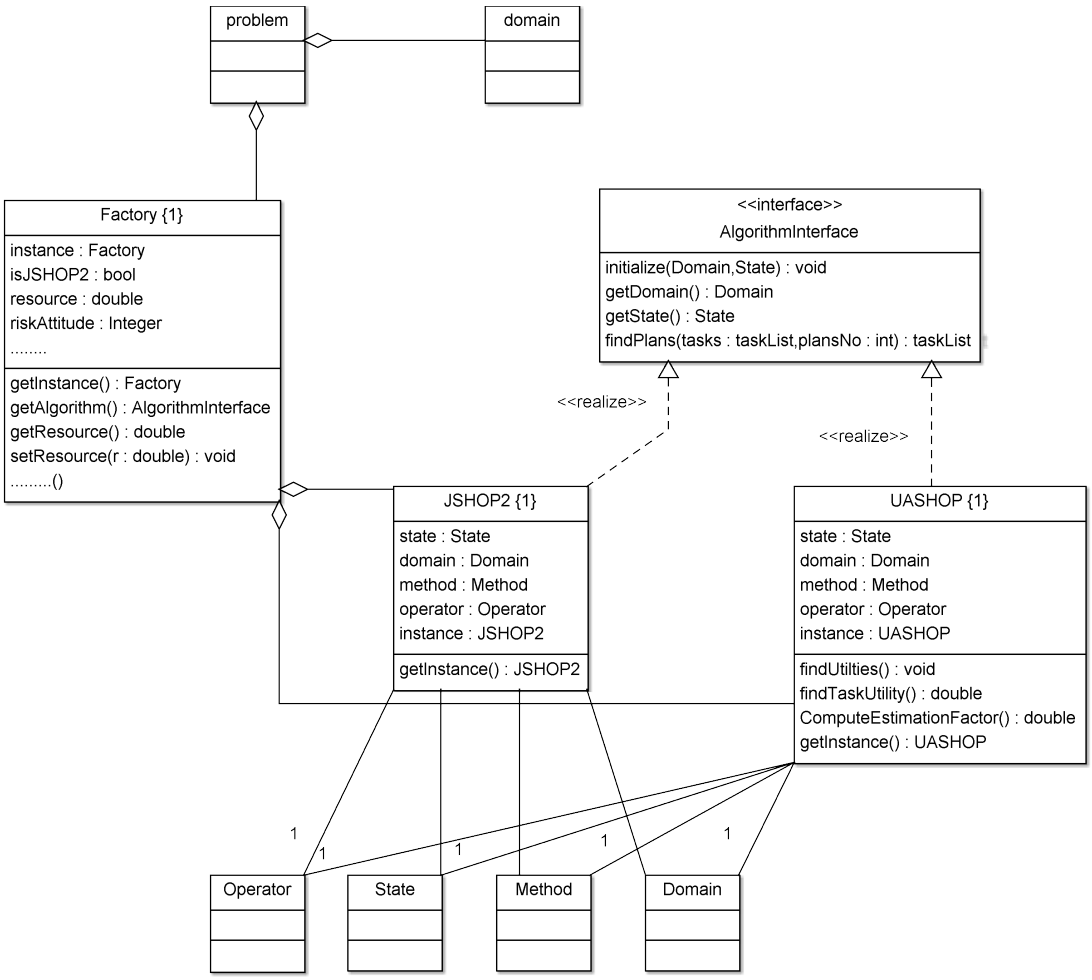


Figure 4.5: A class diagram that shows part of the classes used in the execution phase of the program.

- **A predicate** has the form:
  $(p\ t_1\ t_2\ \ldots\ t_n)$ where p is the predicate symbol and t_i is a term.
- **A logical expression** is either a predicate or one of the following: conjunction, disjunction, negation or a call expression.
  - A conjunction has the form: $([and]\ [L_1\ L_2\ \ldots\ L_n])$, where L_i is a logical expression. The square brackets indicate that their content is optional. Thus, this expression can be empty and in this case it always evaluates to true. Moreover, having $L_1\ L_2\ \ldots\ L_n$ without *and* indicates implicitly that all these logical expressions must evaluate to true in order to satisfy the conjunction logical expression.
  - A disjunction has the form: $(or\ L_1\ L_2\ \ldots\ L_n)$, where L_i is a logical expression.
  - A negation has the form: $(not\ L)$, where L is a logical expression.

  - A call expression has the same syntax as the call term. However, its semantic is different, such that it evaluates either to true or false.

- **A logical precondition** is a logical expression.
- **An axiom:** An axiom is an expression of the form $(:- a\ [name_1]\ L_1\ [name_2]\ L_2\ \ldots\ [name_n]\ L_n)$, where:
  - The head of the axiom a is a predicate and
  - the tail is a list $[name_1]\ L_1\ [name_2]\ L_2\ \ldots\ [name_n]\ L_n$ where L_i is a logical precondition and each $name_i$ is the name of the corresponding L_i.

  The meaning of the axiom is as follows: a is true if L_1 is true, or if L_1 is false but L_2 is true, or if $L_1,\ L_2,\ \ldots,\ L_{n-1}$ are false and L_n is true.
- **A task atom** represents the primitive and the compound tasks in our definitions in Section 2.2 and it has the form: $(s\ t_1\ t_2\ \ldots\ t_n)$, where:
  - the task atom is primitive if s is a primitive task symbol or
  - it is compound if s is a compound task symbol.
- **A task list** is either a task atom or an expression of the form: $([:\ unordered]\ [tasklist_1\ tasklist_2 \ldots\ tasklistn])$, where:
  - $tasklist_1,\ tasklist_2, \ldots,\ tasklistn$ are task lists and
  - the $:\ unordered$ keyword is optional, such that if it is used in the task list then JSHOP2 may interleave between the different tasks in the task lists. On the other hand, in the case where the $:\ unordered$ keyword is omitted, the tasks of different task lists cannot interleave. However, in both cases the ordering of the tasks in the same task list is preserved.
- **An operator** has the following form: $(:\ operator\ h\ P\ D\ A\ [c])$, where:
  - h is the head of the operator and it is a primitive task atom.
  - P is logical precondition.
  - D is a delete list, which is a list whose elements are predicates that can contain only variables appearing in h or P.
  - A is an add list defined similarly to D.
  - c is the operator's cost and it is optional. if the cost is not defined then it takes the default value of 1.
- **A method** has the form: $(:\ method\ h\ [name]\ L\ T)$, where:
  - h is the head of the method and is a compound task atom.
  - L is a logical precondition.
  - T is tail of the method, which is a task list.

  - *name* is the name of the following logical precondition and task list pair, and it is optional.

- **A planning domain** has the form: $(defdomain\ domainName\ (d_1\ d_2\ \ldots\ d_n))$, where each d_i can be either an operator, or a method, or an axiom.
- **A planning problem** has the form:
  $(defproblem\ problemName\ domainName\ ([a_1\ a_2\ \ldots a_n])\ T)$, where:
  - $problemName$ is the planning problem's name.
  - $domainName$ is the domain name in which this problem needs to be solved
  - $([a_1\ a_2\ \ldots a_n])$ is the initial state, where each a_i is a ground predicate.
  - T is the goal task list.
- **A plan** is a list of heads of ground operators. The cost of the plan is the sum of its operator costs.

Listing 4.1 shows an example of a domain input file that contains descriptions of operators, methods and axioms. In this domain, we have a truck and containers that can be located in different locations in a square grid of a specific size defined in the problem. In order for the truck to go to the container's location, we have the *move* compound task that can be decomposed by three different methods (see lines 18-28). For example, the second method (lines 22-24) is used in case there is a shortcut between the truck location and the container location but the truck must not be in the same location of the container. These method's preconditions are expressed in its logical precondition (line 23). The resulting task list of using this method consists of one task $!go-shortcut$ (line 24), which is a primitive task. The operator that executes this task is defined in this file with its preconditions and effects and it has a cost of 5 (lines 3-7). In addition, we define the *inside* axiom (lines 38-40) that we used in the $go-step$ method preconditions. This axiom examines the ?x and ?y values, such that they do not exceed the boundaries of the grid and it evaluates to true if each of the following logical preconditions evaluates to true $((size\ ?si)\ (call\ <\ ?x\ ?si)\ (call\ <\ ?y\ ?si)\ (call\ >\ ?x\ 0)\ (call\ >\ ?y\ 0))$. In this case, each of these logical preconditions is a call expression which either examines the ?x value or the ?y value.

In the same domain, we define a concrete problem, which is described in Listing 4.2. The problem input file specifies the problem name $move-problem$ and the domain name $move-container$. Moreover, it contains the initial state of the system, e.g., the objects that we have, such as the trucks and the containers, the initial positions of those objects, the size of the grid (lines 2-6). In addition, we specify the task to be achieved $move(t1\ 28\ 28)$ that tells the planner that we want to move the truck to the (28,28) position where the container is located.

Listing 4.1 An example domain input file.

```
(defdomain movecontainer (
;; ------------------------------------ operators
(:operator (!go-shortcut ?t ?x ?y ?x2 ?y2 ?s)
           ((not(dont-visit ?s ?x2 ?y2)))
           ((at ?t ?x ?y))
           ((at ?t ?x2 ?y2)(dont-visit ?s ?x ?y))
            5)
(:operator (!go-north ?t ?x ?y ?s)
           ((not(dont-visit ?s ?x (call + ?y 1))))
           ((at ?t ?x ?y))
           ((at ?t ?x (call + ?y 1))(dont-visit ?s ?x ?y)) )
(:operator (!go-east ?t ?x ?y ?s)
           ((not(dont-visit ?s (call + ?x 1) ?y)))
           ((at ?t ?x ?y))
           ((at ?t (call + ?x 1) ?y )(dont-visit ?s ?x ?y) ))
(:operator (!nop) () () () 0)
;; ------------------------------------ methods
(:method (move ?t ?xc ?yc)
((at ?t ?xc ?yc) )
((!nop) ))

(:method (move ?t ?xc ?yc)
((not(at ?t ?xc ?yc) )(at ?t ?x ?y)(shortcut ?x ?y ?xc ?yc))
((!go-shortcut ?t ?x ?y ?xc ?yc)  ))

(:method (move ?t ?xc ?yc)
((not(at ?t ?xc ?yc)) )
((go-step ?t) (move ?t ?xc ?yc)))

(:method (go-step ?t)
((at ?t ?x ?y) (inside ?x (call + ?y 1)) (stage ?s))
((!go-north ?t ?x ?y ?s) ))

(:method (go-step ?t)
((at ?t ?x ?y) (inside  (call + ?x 1) ?y) (stage ?s))
((!go-east ?t ?x ?y ?s) ))
;; ------------------------------------ axioms
(:- (inside ?x ?y)
((size ?si) (call < ?x ?si) (call < ?y ?si) (call > ?x 0)(call > ?y 0))
)
) )
```

Listing 4.2 An example problem input file.

```
(defproblem move-problem movecontainer
((size 30)
(stage 0)
(truck t1) (at t1 1 1)
(container c1) (at c1 28 28)
)
((move t1 28 28)
))
```

5 Evaluation

This chapter presents the evaluation study that we perform to assess the performance of the UASHOP planner. We start by describing the framework used for evaluation then we proceed by giving an overview on the different scenarios and problems used during the evaluation. After that, we present the results obtained from each of these scenarios and problems.

5.1 Framework

In this section, we give a brief description of the framework used for performing our evaluation study. The program that carries out the evaluation is implemented as a Java-based application that uses the main application of UASHOP as a library, i.e., a Jar file.

By default, this application uses the truck-container domain illustrated in Section 3.4 [1]. However, it can be programmed to use other domains. We modeled the different scenario categories, henceforth referred to as *settings*, as separate classes, namely *DiagonalScenario*, *ResourceBasedScenario*, *SmallResourceScenario* and *OrderingMethodsScenario*. On the other hand, the concrete scenarios we evaluate, are objects of the former classes instantiated within the following methods: *runDiagonalScenario*, *runDifferentRiskAttitudesScenario*, *runOrderingMethodsScenario*, *runResourceBasedScenario*, and *runSmallResourceScenario*. These methods are part of the class *AlgorithmEvaluationManagerTest*.

The set of considered concrete scenarios can be executed all at once by running the JUnit Test called *AlgorithmEvaluationManagerTest.runEvaluation*.

Finally, the evaluation application is hosted on Github and can be accessed via this link: `https://github.com/Ebaa-Alnazer/EvaluationOrchestrator`

5.2 Setup

In this section, we explain the scenarios used to evaluate UASHOP and compare it to JSHOP2 using the same domain illustrated in Section 3.4. Each of these evaluation scenarios studies a behavioral aspect of UASHOP including the execution time of the algorithm in addition to the cost and length of the resulting plans. These aspects are evaluated in different settings in terms of the size of the problem and the available resource value.

[1] details of the domain description can be found in Appendix A

In order to obtain a precise planning time, in each scenario, the execution is repeated ten times and the execution time is averaged out over all these executions, because the thread that runs the planning algorithm may have a different execution time each time it runs. Thus, averaging out the execution time over all these executions can compensate for these small differences. Moreover, the time of the pre-processing phase is not considered when calculating the planning time of UASHOP. Basically, this step can be performed once for each domain, but because in some scenarios the same problem is studied for different resources and every time the resource value changes, the pre-processing step has to be performed, it is more convenient to have the resource value as an input. However, due to the small time required by this step, which is so close to zero, not considering it as a part of the planning time never affects the results.

It has to be noted that, in the studied truck-container domain depicted in Figure 3.4, the original ordering of the methods that move the truck from one location to another is as follows (north → east → south → west).

In the following, we explain the setup of the five different scenarios used to evaluate the UASHOP and compare it to JSHOP2.

5.2.1 Varying Grid Size With a Fixed Resource Scenario

In order to test the impact of the problem size on the quality of the resulting plans and on the time needed by the planning algorithm to compute these plans, we create a scenario where the resource value is fixed and equals 350 but the grid size constantly increases starting from 4×4 and ending at 26×26 grid size. In this scenario, the container, truck and target location are located on the grid diagonal. The truck is located in the cell (1,1) of the grid, the container is located in the (grid-size -2, grid-size-2) cell and the target is in the middle of the grid diagonal. Moreover, in this scenario the only risk attitude studied in UASHOP is the risk-seeking one.

Comparing JSHOP2 results with the corresponding ones of UASHOP, we expect that the plan cost, the duration of the planning process and the length of the plan to increase with the increasing grid size in both cases. However, since UASHOP should take the resource into account, the resulting plan cost should not exceed the available resource, while it is acceptable for JSHOP2 to come up with solutions that exceed the resource value.

5.2.2 Comparing Different Risk Attitudes Scenario

This scenario is used to test the difference in behavior between the different risk attitudes. The truck, container and target location are set similarly to the previous scenario. Moreover, the resource here is also fixed and equals 350 and the grid size increases from 4×4 to 26×26. However, in this scenario we examine solving the planning problem with different risk attitudes.

We expect all risk attitudes except the risk-seeking one to act similarly, in this particular problem, because the risk-seeking attitude differs in its choice of the method that brings the container and truck to the same location, i.e., the first step of the whole transfer container process. When having a risk-seeking attitude, the chosen method in this step is to send the container by post to the truck location rather than moving the truck to the container location. This behavior is similar to the JSHOP2 behavior in the meet-container-truck step.

5.2.3 Ordering of Methods Scenario

Basically, UASHOP differs from JSHOP2 in that it takes the utilities into account while planning, such that when it has to choose between different methods, it starts with the method that has the smallest estimation factor. In order to compare the behavior of both algorithms, we have a fixed resource but a varying grid size similarly to the previous two scenarios. However, the truck and container are located at the same location in (grid-size -2, grid-size -2) cell and the target location is in the (1,1) cell. Moreover, the grid has a shortcut connecting the bottom right corner of the grid to the target location directly.

In this scenario, we expect the plan cost and length, as well as the planning time to be lower in UASHOP compared to JSHOP2.

5.2.4 Varying Resource Value With a Fixed Problem Size Scenario

As assumed, in UASHOP the available resource value is an upper bound for the algorithm execution, i.e., a plan that has a cost that exceeds the resource value is considered invalid. However, the amount of the available resource may impact the time needed to find a plan, since the algorithm may have to backtrack multiple times to find a solution within the resource bound.

In this scenario, we test the influence of the resource value on the results. We assume that the grid has a fixed size of 20×20 and the truck as well as the container are located at the same location in the bottom left corner of the grid, whereas the target is in the middle of the grid. The main reason of choosing to put the truck and container at the same location is because we want to focus on the resource impact solely rather than having the different risk attitudes act differently. Moreover, using this simple scenario makes the results more traceable and easier to explain than having a more complicated scenario where the truck and the container are at different locations.

The scenario works as follows. At first, we run the JSHOP2 algorithm on this problem and use the resulting plan cost as the initial resource value, which is used in the first run. After that, in each run we decrease the resource value by 2 and study how this decrement affects the different aspects of the results.

5.2.5 Small Resource with Varying Grid Size Scenario

One interesting behavior to see is having a problem that is solvable with a very small resource. Using this fixed resource value, we study the behavioral difference between JSHOP2 and UASHOP. As explained, JSHOP2 does not take this value into account and returns the first valid plan, whereas UASHOP tries to search for a cost-efficient plan.

This scenario is set as follows. The grid size is varying while the resource value is fixed and equals 5. The target location is assumed to be at the bottom left corner of the grid while the truck and container are put together at the same location in the cell directly above the target. The obvious solution to this problem is to go directly south to reach the target. However, the algorithms act differently based on the order of the direction methods, i.e., north → east → south → west. Furthermore, since UASHOP considers the resource while planning, we expect it to have a different outcome compared to JSHOP2. UASHOP backtracks after few steps and tries other alternatives due to the

small resource value while JSHOP2 continues searching for a plan in the grid. As a result, we expect UASHOP to find the obvious solution consisting of one step, i.e., to go south within a short time compared to JSHOP2 that will not be able to find such solution.

5.3 Results

In this section, we present the results obtained in each of the evaluation scenarios, which run in the truck-container domain explained in Section 3.4. Each of these scenarios represents a different problem that assesses UASHOP in various aspect and compares it to JSHOP2. The setup made for these experiments is explained in Section 5.2 and the acquired results are as follows.

5.3.1 Varying Grid Size With a Fixed Resource Scenario

This experiment studies how varying the grid size affects the resulting plans in both UASHOP and JSHOP2. The setup used in this experiment is explained in Section 5.3.1.

As depicted in Figure 5.1, JSHOP2 plan costs have a linear relation with the grid size, such that when the size increases the resulting plan cost increases too. The same applies to UASHOP until the grid size reaches 23×23. From this grid size on, UASHOP looks for plans that have costs under the resource bound, whereas JSHOP2 finds plans that have costs more than the available resource since it does not account for the resource consumption.

The plan length relation to the grid size is illustrated in Figure 5.2. The graph is linear and it is identical to the previous graph. The reason is that all the operators used to solve this problem have a cost value of 1 except for the operators that accomplish sending the container by post. Thus, the difference between these two graphs at each grid size value equals to the summation of the "sending by post" operators.

Moreover, searching for a plan with a limited resource means that the algorithm has to backtrack perhaps many times from being in the naive solution that exceeds the resource value until it finds a valid plan, hence, the duration of the planning process increases. As shown in Figure 5.3 and Figure 5.1, in order for UASHOP to obtain a plan cost, which is smaller by around 100 than the cost of the naive plan found by JSHOP2, UASHOP planning time increases by around 10 times.

5.3.2 Comparing Different Risk Attitudes Scenario

In this experiment, we evaluate the behavior of the different risk attitudes and its influence on the resulting plans. This experiment has the same setting as the previous one and a detailed explanation of these setting is illustrated in Section 5.2.2. Since all risk attitudes behave the same in this problem except for the risk-seeking attitude, we can see that the costs and lengths of the resulting plans are identical for all of them (see Figures 5.4 and 5.5). However, the planning time differs slightly between these three risk attitudes due to the small differences in the execution time of the thread that runs the algorithm multiple times. The planning time relation to the varying grid size for different risk attitude is presented in Figure 5.6.

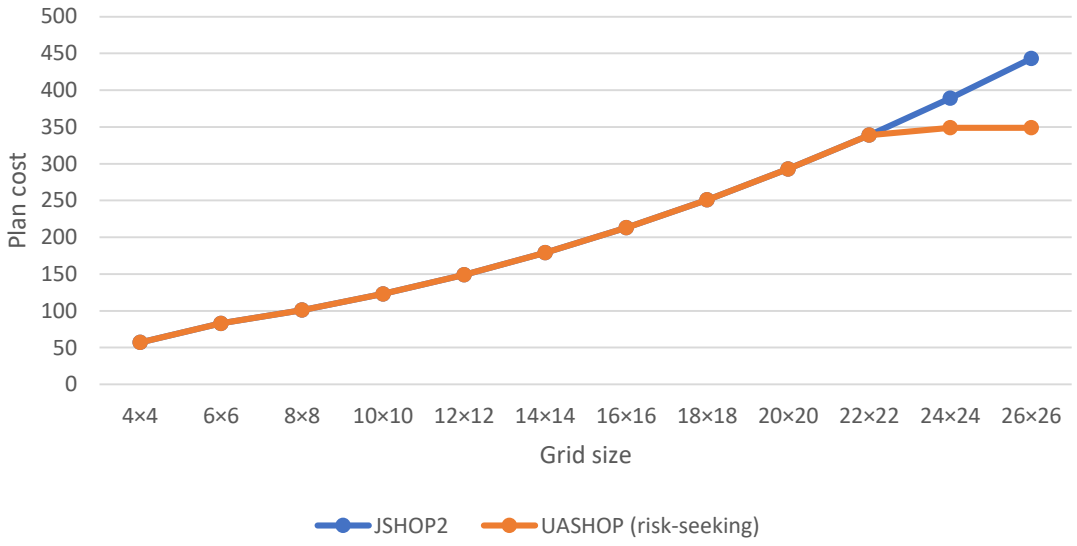


Figure 5.1: A comparison between the values of the resulting plan costs of JSHOP2 and UASHOP under a varying size of the grid.

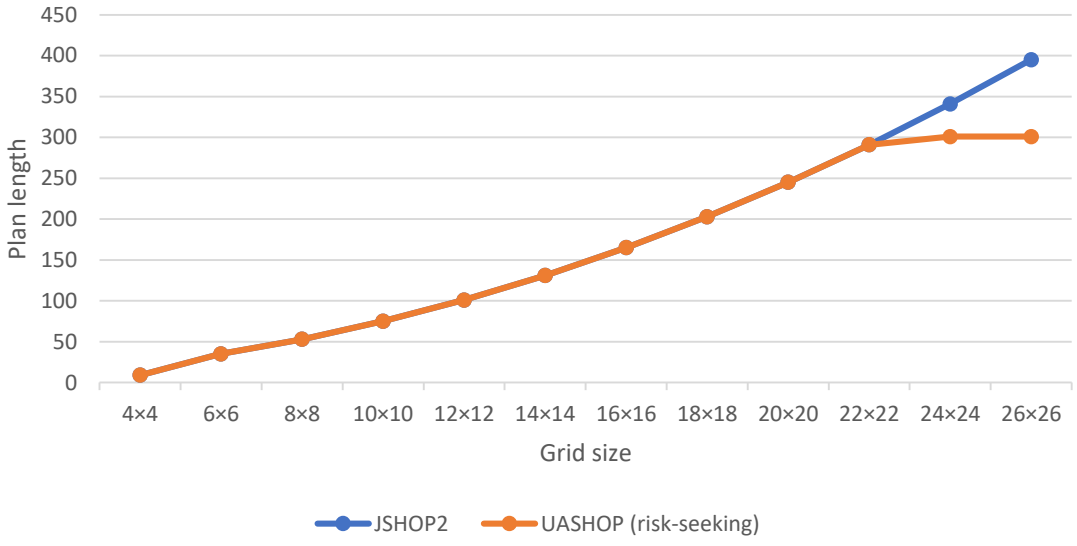


Figure 5.2: A comparison between the values of the resulting plan lengths of JSHOP2 and UASHOP under a varying size of the grid.

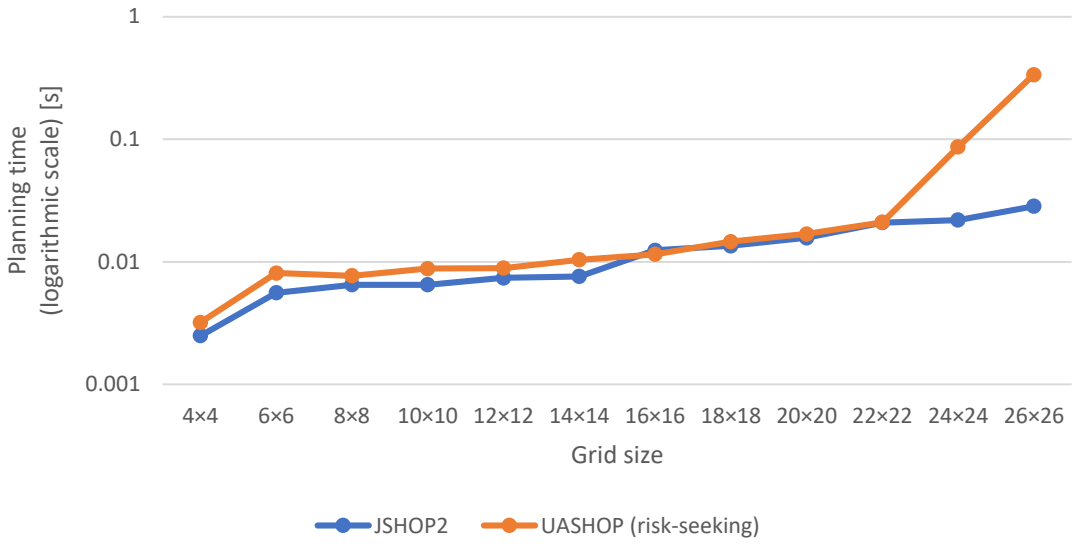


Figure 5.3: A comparison between the planning time of JSHOP2 and UASHOP under a varying size of the grid.

For the risk-seeking attitude, the cost of the resulting plan is higher than others, because in this risk type the algorithm chooses to send the container by post that is, in this particular case, is more costly than moving the truck to the container position. However, the plan length in this case is shorter compared to other risk attitudes.

5.3.3 Ordering of Methods Scenario

In this experiment, we study the importance of deterministically choosing a method among different alternatives by taking the resource consumption into account. Figure 5.10 shows an example of the settings used for this purpose, which include the container, truck and target positions as well as the shortcut that is added between two locations. In this figure, the behavior of the two algorithms is illustrated in a grid of size 6×6. It is obvious, in this particular example, that the time and cost needed to solve this problem in UASHOP should be smaller than the time and cost required by JSHOP2.

As UASHOP orders the methods depending on their estimation factor, it prefers taking the shortcut if it can lead directly to the destination rather than moving the truck to transport the container to the goal. In contrast, JSHOP2 takes the first applicable method based on the original ordering of these methods, hence, JSHOP2 chooses to move the truck. As depicted in Figures 5.7 to 5.9, we can see that the cost, length of the resulting plan, as well as the planning time differ dramatically between the two algorithms especially for large grid sizes where JSHOP2 may visit almost all the grid and backtrack multiple times before reaching the goal.

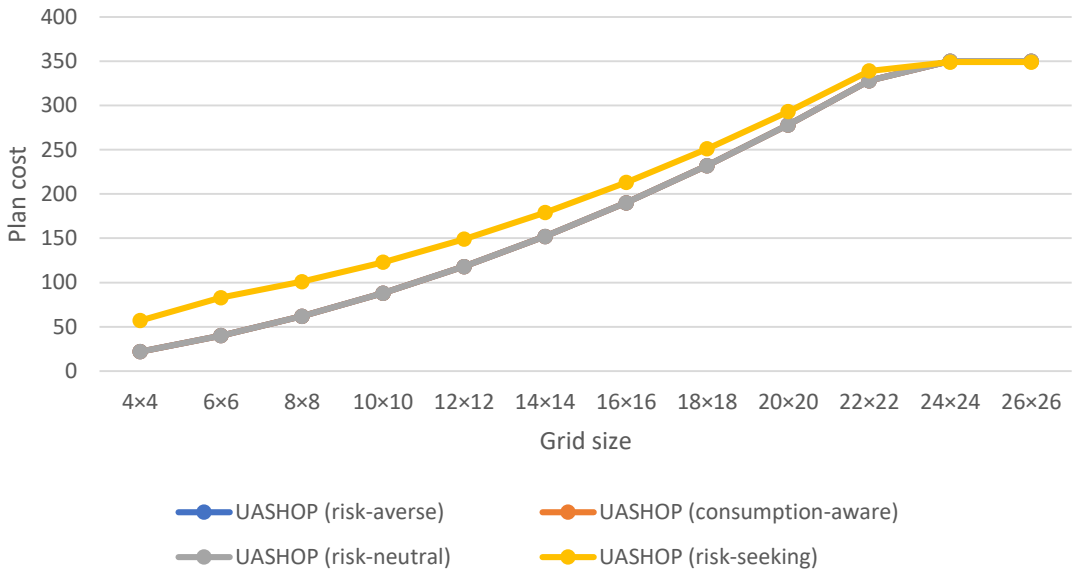


Figure 5.4: A comparison between the values of the resulting plan costs of the different risk attitudes in UASHOP under a varying size of the grid.

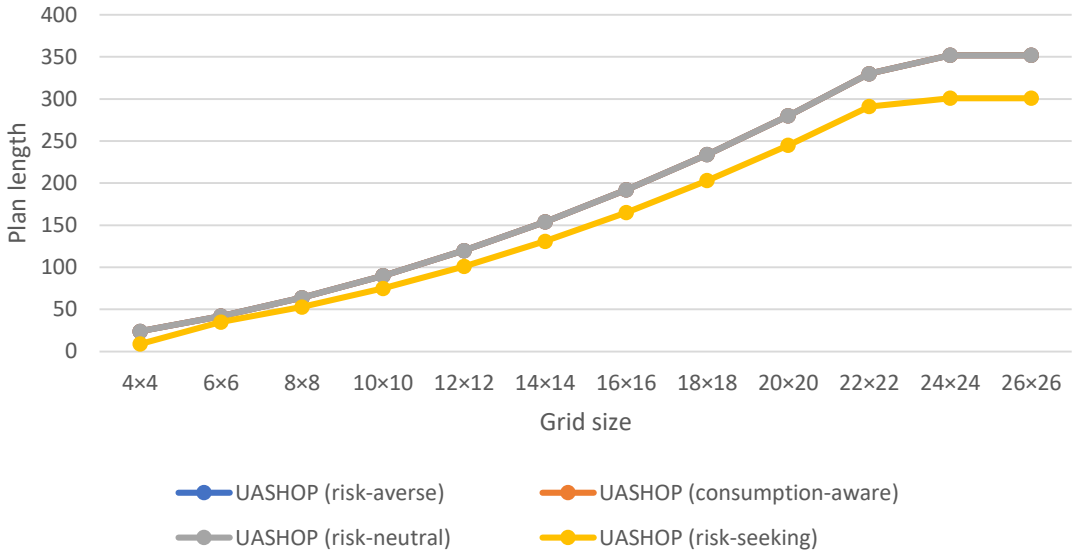


Figure 5.5: A comparison between the values of the resulting plan lengths of the different risk attitudes in UASHOP under a varying size of the grid.

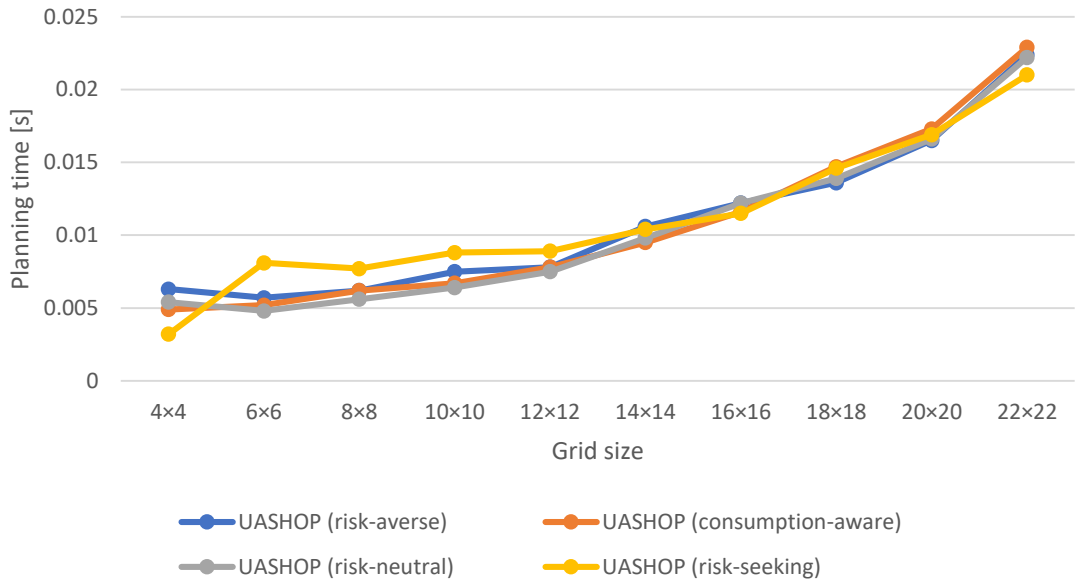


Figure 5.6: A comparison between the values of the planning time of the different risk attitudes in UASHOP under a varying size of the grid.

In this particular problem, we can see that in UASHOP, all risk-attitudes act similarly and produce the same results, since the ordering of the methods that move the truck is the same for all of them, because the estimation factor of taking the shortcut is smaller than the estimation factor of the other method, which is part of a cycle.

5.3.4 Varying Resource Value With a Fixed Problem Size Scenario

Since UASHOP works towards finding plans with costs under the available resource, it is interesting to study the impact of different resource values on the results.

This experiment has the settings explained in Section 5.2.4 and the starting point of the resource is the cost obtained by running JSHOP2 to get the naive solution that does not take the resource value into account. As shown in Figure 5.11 and Figure 5.12, the resulting plan cost and length have a linear relation to the resource value; having a smaller resource leads to obtaining a plan with smaller cost and length. The reason behind the identical slope for the linear graphs of both cost and length evaluations is that, all the operators used to solve this problem have a cost of 1.

Figure 5.13 illustrates the relation between the varying resource value and the required time for planning. We notice a repeated pattern with increasing values over the time. This pattern heavily depends on this particular domain and problem because the location at which the algorithm backtracks, due to exceeding the resource, affects the number of tries the algorithm makes to reach the goal within the resource boundary. Furthermore, the position at which the algorithm backtracks to try other alternatives has a big influence on the road that it will take to arrive at the goal location.

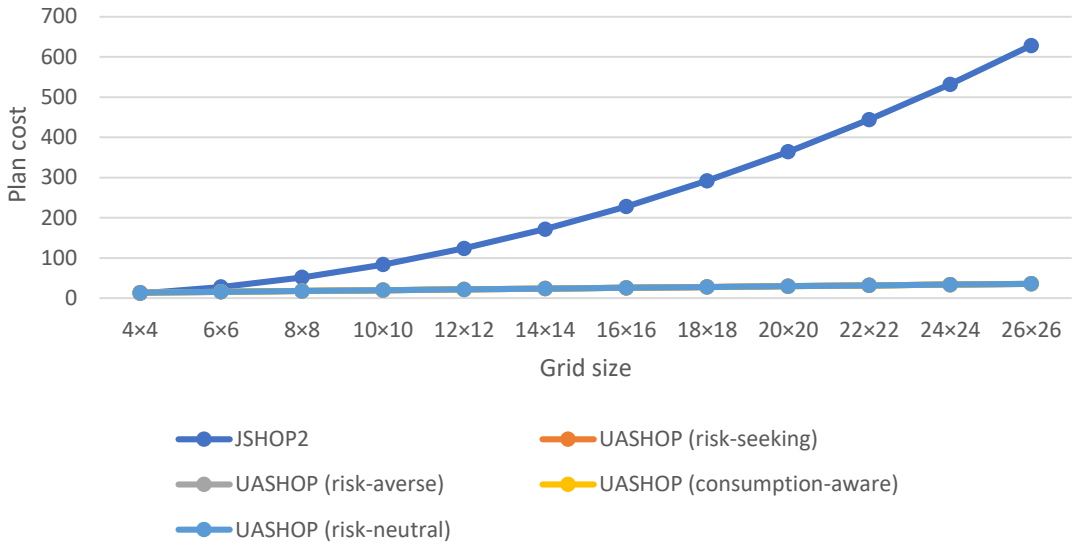


Figure 5.7: A comparison between JSHOP2 and UASHOP under a varying size of the grid to illustrate the impact of methods ordering on the resulting plan costs.

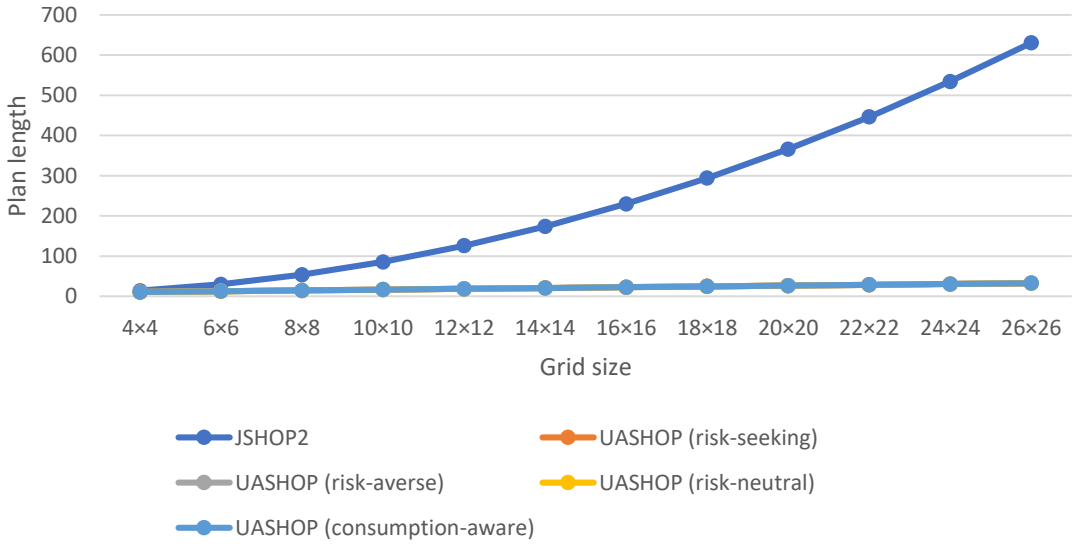


Figure 5.8: A comparison between JSHOP2 and UASHOP under a varying size of the grid to illustrate the impact of methods ordering on the resulting plan lengths.

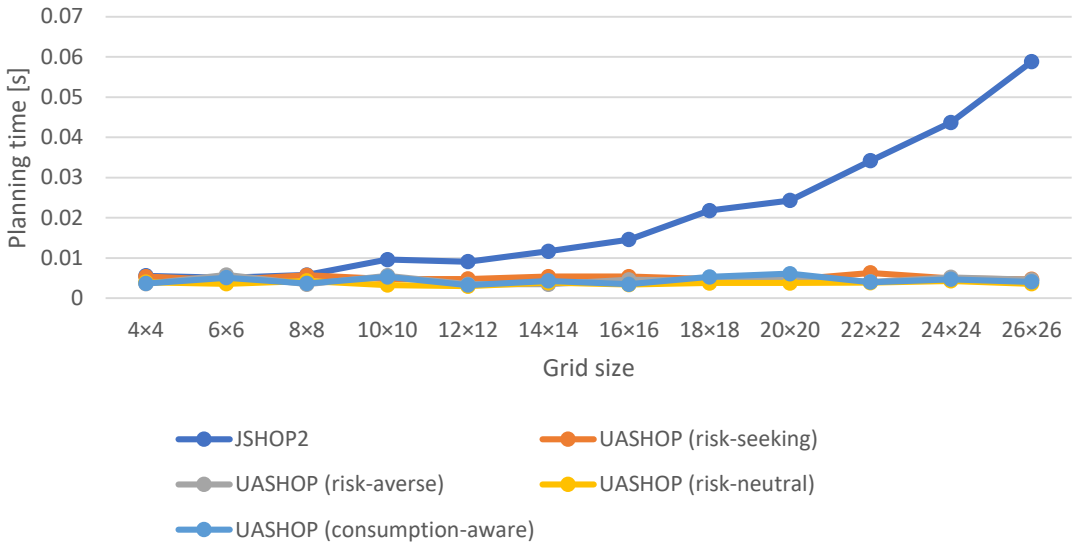


Figure 5.9: A comparison between JSHOP2 and UASHOP under a varying size of the grid to illustrate the impact of methods ordering on the planning time.

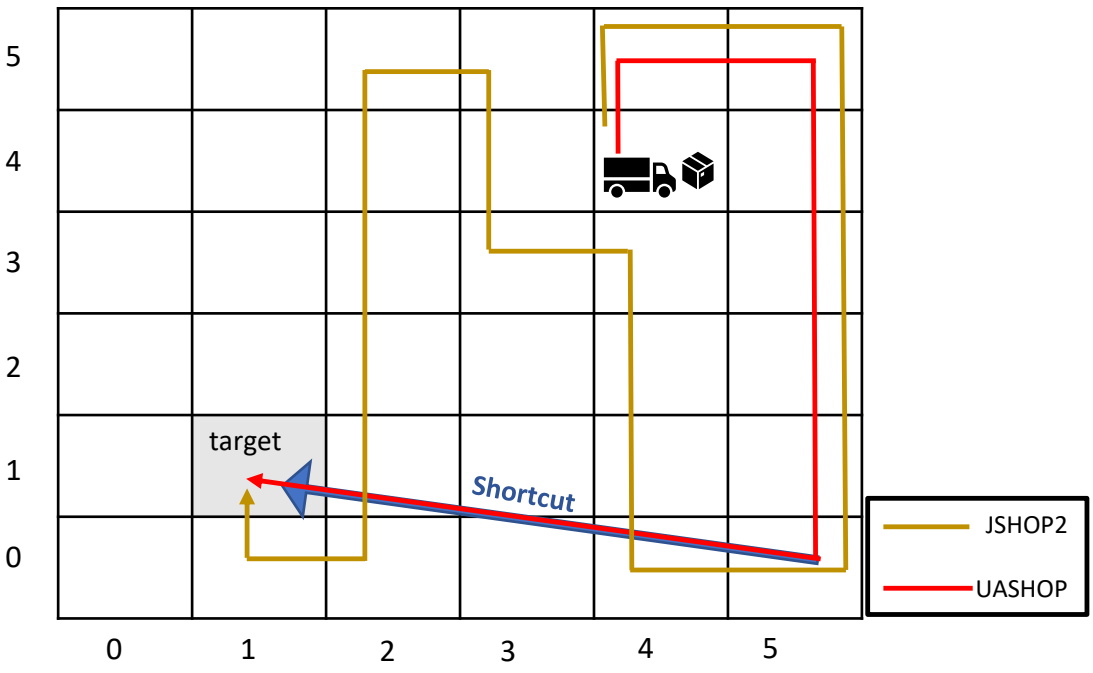


Figure 5.10: A comparison between the behavior of JSHOP2 and UASHOP when having multiple method alternatives.

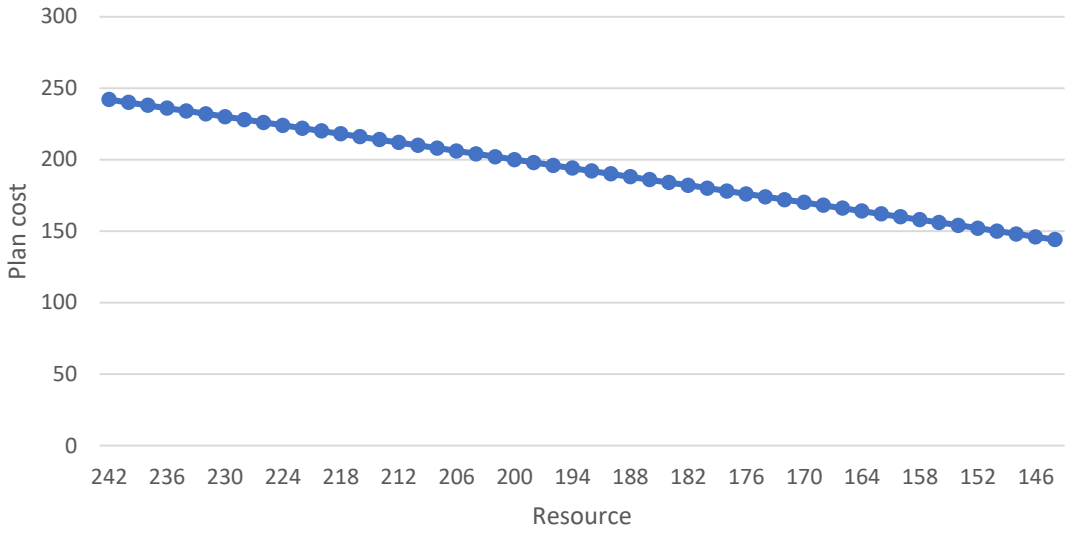


Figure 5.11: The resource value impact on the cost of the resulting plan in UASHOP. The starting point of this graph is the cost of the resulting plan in JSHOP2.

In general, in this particular problem, if we look at two resource values within a slightly long interval, we notice that the corresponding planning time is bigger when the resource is smaller, which is reasonable since UASHOP has to backtrack many times to come up with a valid plan.

In this domain, the only difference between the different risk attitudes is in the first step of the whole process when the container either is sent by post or the truck is moved to the container's location. In this problem, the container and the truck are already in the same location, hence the different risk attitudes, in this scenario, behave similarly and lead to identical plans.

However, in other problems, having a small resource value can lead to a very fast planning process and a small cost and length of the resulting plan. As an example, consider the following experiment.

5.3.5 Small Resource with Varying Grid Size Scenario

As discussed in Section 5.3.4, there are some problems in which having a small resource can improve the quality of the resulting plan. This experiment is made to prove this point. As explained in Section 5.2.5, the target location in this scenario is set to be right under the starting position in the bottom left corner of the grid. Furthermore, we have a resource value of 5. Looking at the quality of the resulting plan including the cost and length of it, depicted in Figures 5.14 and 5.15, we see that the quality of UASHOP plans have a constant cost and length for all risk attitudes regardless of the grid size, whereas JSHOP2 results are affected a lot by the grid size and their cost and length are increased dramatically compared to the resulting plans of UASHOP. Moreover, since UASHOP behaves similarly in all given grid sizes when searching for a valid plan, the planning time is not affected by the grid size, whereas JSHOP2 needs to visit almost the whole grid in order to reach the target location. The planning time relation with the grid size is illustrated in Figure 5.16.

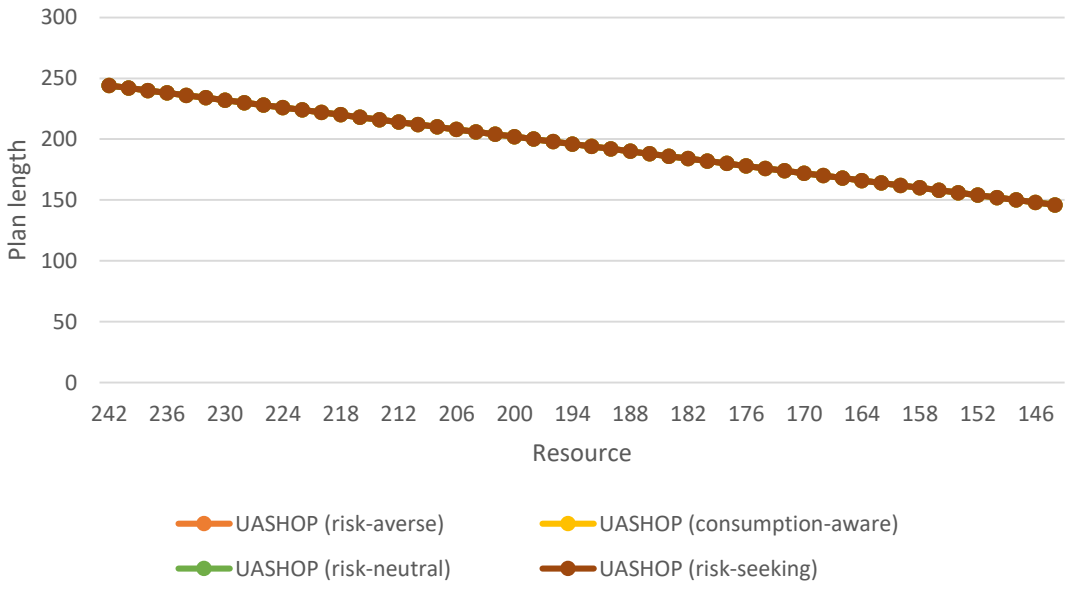


Figure 5.12: The resource value impact on the resulting plan length in UASHOP. The starting point of this graph is the cost of the resulting plan in JSHOP2.

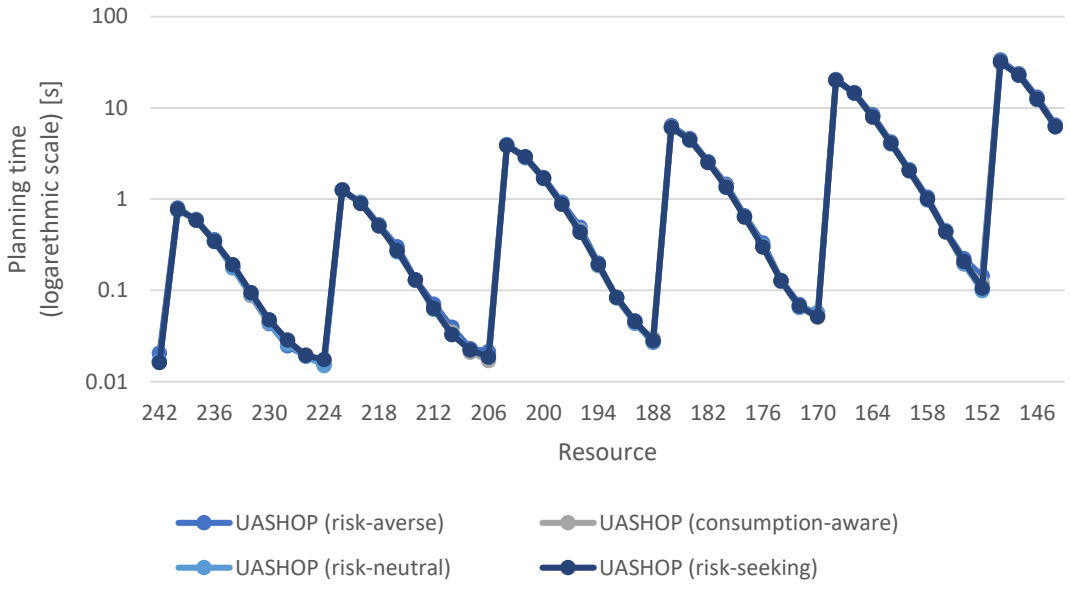


Figure 5.13: The resource value impact on the time planning in UASHOP. The starting point of this graph is the cost of the resulting plan in JSHOP2.

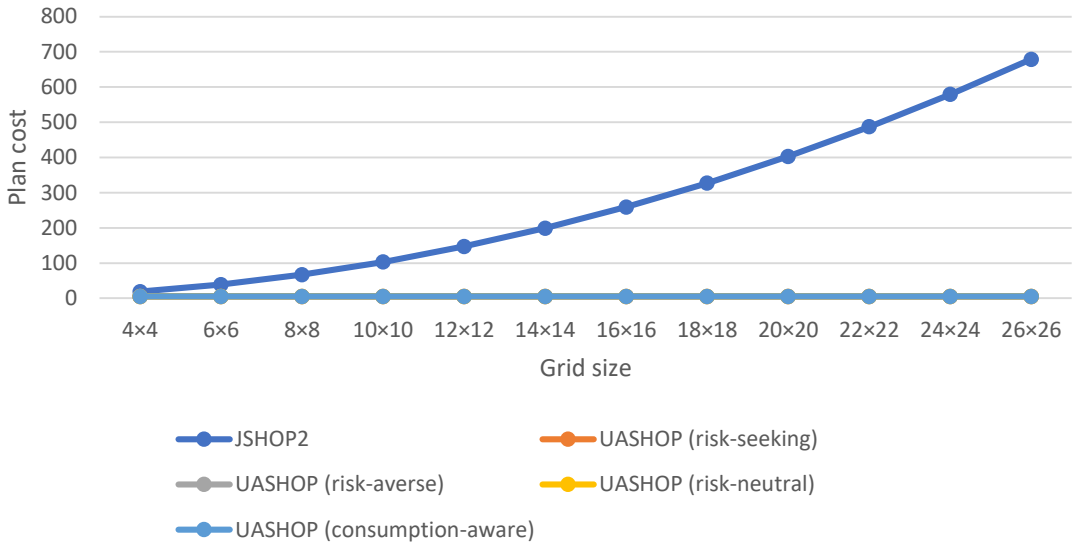


Figure 5.14: A comparison between JSHOP2 and UASHOP plan costs for a small resource value under a varying size of the grid.

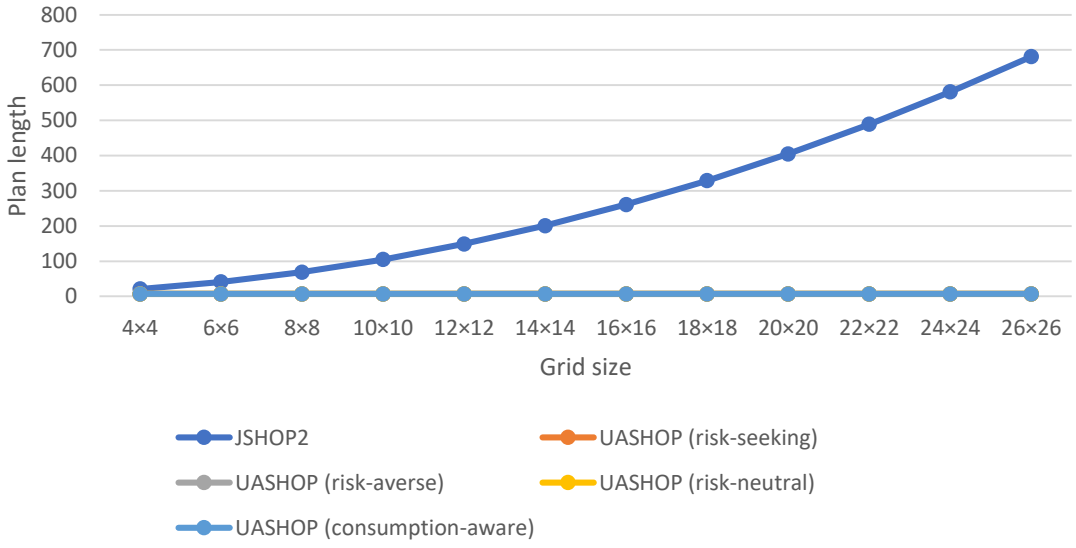


Figure 5.15: A comparison between JSHOP2 and UASHOP plan lengths for a small resource value under a varying size of the grid.

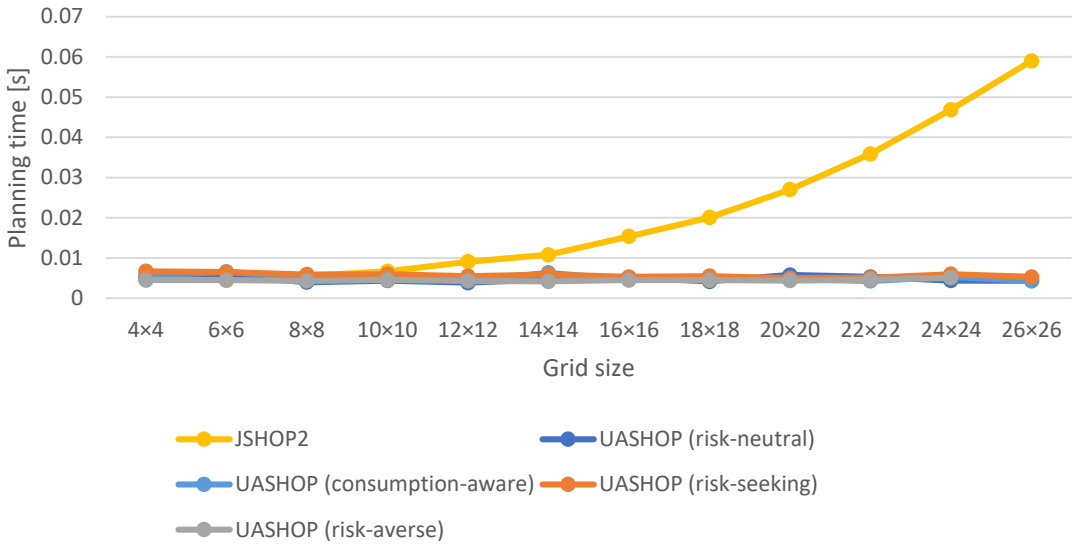


Figure 5.16: A comparison between JSHOP2 and UASHOP planning time for a small resource value under a varying size of the grid.

5.4 Summary of Results

UASHOP takes the task utilities into account while searching for a plan, such that it tries to reduce the resource consumption as much as possible depending on these estimated values, i.e., the utilities. Thus, it orders the methods depending on their estimation factors that is calculated based on the task attitude towards risk.

While planning, UASHOP always starts with the method that has the smallest estimation factor because such a method promises a better solution than others, which in turn can substantially improve the quality of the solution. We proved this point in the experiment discussed in Section 5.3.3 where the resulting plans of UASHOP had a better quality than those of JSHOP2. However, in the case where the methods ordering happens to be the same in both algorithms, the solutions of the two algorithms can still be different. The reason behind this is that during planning, UASHOP backtracks if it reaches a point where the available resource is exceeded. Since UASHOP does not model the resource in the planning process, a plan produced by it does not necessarily be a valid plan in UASHOP. Therefore, even if the two algorithms start with the same method, UASHOP may backtrack and choose another alternative if the resource is exceeded. In the experiments, we saw that, when increasing the size of the problem and when the cost of the plan exceeds the available resource, JSHOP2 continues producing plans with larger costs while UASHOP searches for alternatives that do not exceed the resource and this in turn can increase the required planning time of UASHOP.

Since UASHOP considers the different attitudes towards risk, the resulting plans can differ from one risk attitude to another because the method estimation factors are calculated based on the risk attitude type, hence, the ordering of the methods that can decompose a compound task can also differ. Evaluating these different risk attitudes, we saw that the risk-seeking attitude may produce

a solution faster but more costly compared to the other risk attitudes. However, in the particular domain that we do our evaluations in, the plans quality of the three remaining risk attitudes is indistinguishable.

When studying the resource value impact on the plans quality in UASHOP, we concluded that, generally, the smaller the available resource is compared to the naive solution cost of JSHOP2, the longer the time UASHOP takes to come up with a valid plan. However, the resulting plan cost becomes better for small available resource values. These results are heavily dependent on the concrete problem and domain because there are cases in UASHOP, where having a small resource can lead to much faster and shorter solutions compared to having bigger resource values.

6 Related Work

To the best of our knowledge, using utilities in HTN planning has not been covered in previous studies. However, there are few studies that addressed using utilities with planners that are basically different from HTN. For instance, the study of Koenig et al. [KS94] discussed how the planning approach changes when considering different risk attitudes with different utility functions. In this study, they used *probabilistic decision graphs* as a planning framework in which they transformed the decision graph in risk-sensitive problems into an equivalent graph for risk neutral attitude which then can be solved with known methods that solve planning problems with risk-neutral attitude.

In addition, preference-based planning was considered in many studies. For example, in [SBM09], an extended Planning Domain Definition Language (PDDL3) over HTN constructs is defined to encode preferences, then an algorithm that uses a metric function and heuristics searches for the most preferred plan. Heuristics are used to choose the method to use among many available options in order to speed up the planning process.

In Nau et al. [NAI+03] study, a limited branch-and-bound optimization algorithm in the SHOP2 planner was considered to guide the decomposition to the least-cost plan with a possibly chosen execution time limit. However, utilities and different risk attitudes were not discussed.

An example of other studies that address the problem of searching the space for the optimum valid plan in HTN planning is [LZZ11] in which a utility function is assigned to primitive tasks to evaluate the execution cost or effect of this task and a messy genetic algorithm is used to search for the best plan.

The closest study to ours is [KG09], in which the authors generalize the HTN planner SHOP2 [SPW+04]; a planner for web service composition, by taking into account user's ratings and social trust when having alternative possible compositions of the required web service. They suggest a planning algorithm called "Trusty" which takes the formal description of the problem as an input and generates a sequence of services that achieve the goal. In this study, they distinguish two types of services namely: atomic and composite services. Atomic services are services that can be executed directly in the current state of the system. For these services a user's trust value is assigned based on the social trust this user has in other users' ratings along with the rating of the service, which is influenced by its behavioral features. On the other hand, composite services have to be decomposed into atomic services whose trust values are propagated to their corresponding composite service. Three different strategies to compute trust values were discussed here, namely: overly-cautious, overly-optimistic and average. The first strategy aims at maximizing the minimum expected trust value across the sub-processes in contrast to the second strategy, which maximizes the maximum expected trust value. However, average strategy differs from the two previous ones, in that the average of the sub-processes' trust values is taken. However, in this study, recursive tasks are not taken into consideration. Moreover, the utilities here are defined in terms of profit instead of risk,

such a method with a higher estimated value is more preferable. In addition, this study is tailored for a specific domain of the semantic web service decomposition and is not generalized to work with other domains.

Another study that tackles the problem of choosing a method among many alternatives when decomposing a task is [CWY+18]. The approach followed by this study is to order the methods based on a heuristic function. This heuristic function defines the distance between the final goal state of the problem and the method's goal state. However, this approach may not work for the risk-sensitive domains, since it does not consider the risk associated with the application of each method neither does it take the different attitudes towards risk into account.

In their study, Alford et al. [ABH+16] work on translating tail-recursive problems automatically into a non-hierarchical representation that benefit from the hierarchical structures of HTN planning, in addition to the classical planning heuristics. In order to do that, they defined an upper and lower bounds for these tail-recursive tasks. However, in these bounds they did not consider the available resource value.

This work is based and builds on a previous study [GL14] that suggests the use of HTN planning for risk-sensitive domains, where the HTN planner takes method utilities into account.

7 Conclusion and Future Work

In this work, we studied the usage of HTN planning in domains that are characterized by their sensitivity towards risk. In order to achieve that, we defined a conceptual model of an HTN planner that is designed to work in such domains. After that, we designed a pre-processing algorithm that computes the task utilities as well as the method estimation factors. Moreover, we modified the existing JSHOP2 planning algorithm, such that the modified algorithm (UASHOP algorithm) is capable to choose the most promising method among other alternatives depending on their estimated factors computed by the pre-processing algorithm. Furthermore, this algorithm takes the resource consumption of operators into account to ensure that the resulting plan cost does not exceed the available resource. In addition to that, the algorithm adapts to the existence of cycles in the domain by trying to avoid methods that are part of a cycle via influencing their estimation factor. In order to validate our planning approach, we implemented the utility-aware planner by modifying the existing JSHOP2 planner.

Furthermore, we implemented a Java-based application to evaluate the planner by taking different measures including the time required by the planner to come up with the desired plan, the resulting plan cost and length. Those aspects are evaluated in a specific domain using different scenarios that describe various problems in this domain where each scenario is designed to explain an aspect of UASHOP planner and compare it to JSHOP2 planner. The results of the evaluation showed that ordering the methods depending on their estimation factors can dramatically decrease the planning time and lead to solutions with a small resource consumption compared to planning approaches that does not take the utilities into account. Moreover, the planner distinguishes the different risk attitudes, such that the resulting plan can differ depending on the followed attitude. In our evaluation, we saw that the risk-seeking attitude can lead to solutions that are less resource-efficient compared to other risk attitudes.

When changing the size of the problem, we observed that both UASHOP and JSHOP2 behave similarly until a point where JSHOP2 starts producing plans that exceed the available resource. At this point, UASHOP starts to backtrack and searches for plans that have costs within the resource limit. Thus, in this case, the planning duration of UASHOP becomes higher than JSHOP2. Furthermore, we concluded that the value of the available resource impacts the quality of the resulting plans. However, the influence of this value on the planning duration and the cost and length of the resulting plan is heavily dependent on the domain and the concrete problem in this domain.

As a future work, it is interesting to evaluate the different characteristics of the planner in different domains and compare the results. Furthermore, proving the soundness and completeness of the main planning algorithm is left to be done in the future.

In our approach, we adapted the planning algorithm to deal with cycles to prevent the planning algorithm from looping in this cycle infinitely. Our approach is based on the resource consumption, such that at each iteration in the cycle, an amount of the resource is reduced. This consumption equals to the summation of the involved operator costs. Hence, the algorithm backtracks when

reaching a point where the consumption exceeds the available resource. However, if the planning algorithm enters a cycle that has no consumption at all, the execution never halts. Hence, as a future work, we should find an execution upper bound that is not totally dependent on the resource but also it should take the resource into account, such that a plan that has a cost greater than the available resource must be considered invalid.

As we saw, our approach is based on two steps, namely the pre-processing step followed by the main algorithm step. It may be interesting to design the planner differently and compare the resulting planner with ours. For example, the pre-processing step may be skipped but the computations of task utilities and method estimation factor must be done during the planning process, such that the algorithm backtracks when reaching a point that leads to a worst solution compared to the solution found so far.

Finally, further investigation should be conducted in order to analyze the effect of using phantomization [GA19] in conjunction with our utility-aware approach and judge whether this combination is feasible or not.

A Sample Domain File

In the following, the input file of the truck-container domain is presented.

```
(defdomain movecontainer (
(:operator (!go-shortcut ?t ?x ?y ?x2 ?y2 ?s) ((not(dont-visit ?s ?x2 ?y2))) ((at ?t ?x ?y)) ((at ?t ?x2
?y2)(dont-visit ?s ?x ?y)) 5)
(:operator (!go-north ?t ?x ?y ?s) ((not(dont-visit ?s ?x (call + ?y 1)))) ((at ?t ?x ?y)) ((at ?t ?x (
call + ?y 1))(dont-visit ?s ?x ?y)) )
(:operator (!go-east ?t ?x ?y ?s) ((not(dont-visit ?s (call + ?x 1) ?y))) ((at ?t ?x ?y)) ((at ?t (call +
 ?x 1) ?y )(dont-visit ?s ?x ?y) ))
(:operator (!go-south ?t ?x ?y ?s) ((not(dont-visit ?s ?x (call - ?y 1)))) ((at ?t ?x ?y)) ((at ?t ?x (
call - ?y 1))(dont-visit  ?s ?x ?y) ))
(:operator (!go-west ?t ?x ?y ?s) ((not(dont-visit ?s (call - ?x 1) ?y))) ((at ?t ?x ?y)) ((at ?t (call -
 ?x 1) ?y )(dont-visit ?s ?x ?y) ))
(:operator (!nop) () () () 0)
(:operator (!take ?c ?t ) ((at ?c ?xc ?yc)(at ?t ?xc ?yc)) ((at ?c ?xc ?yc)) ())
(:operator (!load ) () ((stage 0)) ((stage 1)))
(:operator (!unload ) () () ())
(:operator (!put ?c ?t ) ((at ?t ?xt ?yt)) () ((at ?c ?xt ?yt)))
(:operator (!go-to-post ?t ?c ) ((at ?t ?xt ?yt)(at ?c ?xc ?yc)) ((at ?c ?xc ?yc)) ((at ?c ?xt ?yt)) 1)
(:operator (!online-stamping ) () () () 50)
(:operator (!packaging ) () () () 2)
(:operator (!write-address ) () () () 0)

;; ------------------------------------- methods

;; ----------------- transport-Container
(:method (transport-container ?c ?t ?xt ?yt)
transport-container
((at ?c ?xc ?yc) (at ?t ?x ?y))
((meet-container-truck ?t ?c ?xc ?yc) (setup ?c ?t)(move ?t ?xt ?yt)(finish ?c ?t)))

(:method (meet-container-truck ?t ?c ?xc ?yc)
send-by-post
((not(at ?t ?xc ?yc))(at ?t ?xt ?yt))
((!online-stamping)(!packaging)(!write-address)(!go-to-post ?t ?c))
)

(:method (meet-container-truck ?t ?c ?xc ?yc)
move-to-container
()
((move ?t ?xc ?yc))
)

;; ----------------- move
(:method (move ?t ?xc ?yc)
dont-do-anything
((at ?t ?xc ?yc) )
((!nop) ))
```

```
(:method (move ?t ?xc ?yc)
go-step-and-move
((not(at ?t ?xc ?yc)) )
((go-step ?t) (move ?t ?xc ?yc)))

(:method (move ?t ?xc ?yc)
go-shortcut
((not(at ?t ?xc ?yc) )(at ?t ?x ?y)(shortcut ?x ?y ?xc ?yc)(stage ?s))
((!go-shortcut ?t ?x ?y ?xc ?yc ?s)  ))

;; ----------------- go-step
(:method (go-step ?t)
go-north
((at ?t ?x ?y) (inside ?x (call + ?y 1)) (stage ?s))
((!go-north ?t ?x ?y ?s) ))

(:method (go-step ?t)
go-east
((at ?t ?x ?y) (inside  (call + ?x 1) ?y) (stage ?s))
((!go-east ?t ?x ?y ?s) ))

(:method (go-step ?t)
go-south
((at ?t ?x ?y) (inside ?x (call - ?y 1)) (stage ?s))
((!go-south ?t ?x ?y ?s) ))

(:method (go-step ?t)
go-west
((at ?t ?x ?y) (inside (call - ?x 1) ?y ) (stage ?s))
((!go-west ?t ?x ?y ?s) ))

;; ----------------- setup
(:method (setup ?c ?t)
setup
()
((!take ?c ?t )(!load ))
)

;; ----------------- finish
(:method (finish ?c ?t)
finish
()
((!unload )(!put ?c ?t ))
)

;; -------------------------------------- axioms
(:- (inside ?x ?y)
((size ?si) (call < ?x ?si) (call < ?y ?si) (call >= ?x 0)(call >= ?y 0))
)
) )
```

Listing A.1: The input file of the truck-container domain

B Execution of UASHOP Using the Command-Line

The UASHOP program can be packaged into an executable JAR file, which can then be used to trigger the generation of both the domain and problem Java source files based on given input files.

In order to package the program into an executable JAR file, run the following command from, e.g., `cmd`, while being in the root directory of the application:

```
mvn clean package
```

This command assumes the existence of the Maven tool[1].

This would create a folder named `target` that contains the desired JAR file, which is called, `Utilities-JSHOP2-1.0-SNAPSHOT-jar-with-dependencies.jar`. Using this JAR file, we can trigger the generation of a domain Java source file from a given domain input file with the following command:

```
java -jar .\Utilities-JSHOP2-1.0-SNAPSHOT-jar-with-dependencies.jar <domain-input-file>
```

Furthermore, we can generate the Java source file that corresponds to a specific problem instance from the previous domain using a given problem input file. The command that generates this file differs based on whether we want to execute the classical, or the utility-aware HTN planning algorithm.

For the first case, the command is:

```
java -jar .\Utilities-JSHOP2-1.0-SNAPSHOT-jar-with-dependencies.jar <num-of-plans> <problem-input-file> <
output-format>
```

Whereas, for the second case, the command is:

```
java -jar .\Utilities-JSHOP2-1.0-SNAPSHOT-jar-with-dependencies.jar <num-of-plans> <problem-input-file> <
resource-value> <risk-attitude> <output-format>
```

The semantics of the arguments of these two commands are as follows:

- The `num-of-plans` argument can take one of the following three values: `-r`, `-rSomeInteger`, `-ra`. `-r` indicates that only the first plan found by the planner should be returned, whereas, `-rSomeInteger` means that the planner should return at most `SomeInteger` plans. Finally, `-ra` indicates that all the valid plans should be returned. Hence, the user must use the last option only if the number of valid plans is traceable.

[1] Apache Maven Project: https://maven.apache.org/. Visited on: November 3, 2019

- The `output-format` arguments can take one of these two values: `csv` and `txt`. `csv` means that the final output of the planner will be in the form of a Comma-Separated File (CSV) that indicates things like the duration of the planning process, the length of the produced plans, as well as the cost incurred by their operators. `txt` on the other hand, would result in a verbose output listing the detailed steps of each resulting plan, as well as the planning duration and incurred cost.
- The `resource-value` argument takes any floating-point number that represents the total available resource.
- The `risk-attitude` argument can have one of the following four integer values, each of which specifies a different risk-attitude: `0`, `1`, `2` and `3`. These values refer to the `risk-averse`, `risk-seeking`, `risk-neutral` and `consumption-aware` attitudes, respectively.

After generating the source files, they should be compiled using the following command:

```
javac *.java
```

Finally, the planning algorithm can be executed using the following command:

```
java <problem-input-file>
```

Notice the usage of `problem-input-file`, since the generated Java class has the same name as the problem input file. The final output file, with the selected format, is placed in the same folder

Also notice that the program is tested under the following version of Java: `Java SE Development Kit 8u231`, which can be downloaded from: https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html (visited on November 3, 2019).

Bibliography

[ABH+16] R. Alford, G. Behnke, D. Höller, P. Bercher, S. Biundo, D. W. Aha. "Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems". In: *Twenty-Sixth International Conference on Automated Planning and Scheduling*. 2016 (cit. on p. 68).

[AGPR04] F. Amigoni, N. Gatti, C. Pinciroli, M. Roveri. "What planner for ambient intelligence applications?" In: *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 35.1 (2004), pp. 7–21 (cit. on p. 18).

[CFGP06] L. A. Castillo, J. Fernández-Olivares, O. Garcia-Perez, F. Palao. "Efficiently Handling Temporal Knowledge in an HTN Planner." In: *ICAPS*. 2006, pp. 63–72 (cit. on p. 18).

[CWY+18] K. Cheng, L. Wu, X. Yu, C. Yin, R. Kang. "Improving hierarchical task network planning performance by the use of domain-independent heuristic search". In: *Knowledge-Based Systems* 142 (2018), pp. 117–126 (cit. on p. 68).

[GA15] I. Georgievski, M. Aiello. "HTN planning: Overview, comparison, and beyond". In: *Artificial Intelligence* 222 (2015), pp. 124–156 (cit. on pp. 15, 18, 19).

[GA19] I. Georgievski, M. Aiello. "Phantomisation in state-based HTN planning". In: *International Conference on Advances in Signal Processing and Artificial Intelligence*. ASPAI'19. To appear. 2019 (cit. on p. 70).

[GL14] I. Georgievski, A. Lazovik. "Utility-Based HTN Planning." In: *ECAI*. 2014, pp. 1013–1014 (cit. on pp. 18, 25, 68).

[GNA13] I. Georgievski, T. A. Nguyen, M. Aiello. "Combining activity recognition and AI planning for energy-saving offices". In: *2013 IEEE 10th International Conference on Ubiquitous Intelligence and Computing and 2013 IEEE 10th International Conference on Autonomic and Trusted Computing*. IEEE. 2013, pp. 238–245 (cit. on p. 18).

[GNN+17] I. Georgievski, T. A. Nguyen, F. Nizamic, B. Setz, A. Lazovik, M. Aiello. "Planning meets activity recognition: Service coordination for intelligent buildings". In: *Pervasive and Mobile Computing* 38 (2017), pp. 110–139 (cit. on p. 18).

[GNT04] M. Ghallab, D. Nau, P. Traverso. *Automated Planning: theory and practice*. Elsevier, 2004 (cit. on pp. 15, 17–19, 27).

[Ilg06] O. Ilghami. "Documentation for JSHOP2". In: *Department of Computer Science, University of Maryland, Tech. Rep* (2006) (cit. on pp. 41, 42).

[IN03] O. Ilghami, D. S. Nau. *A general approach to synthesize problem-specific planners*. Tech. rep. MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE, 2003 (cit. on p. 41).

[KG09] U. Kuter, J. Golbeck. "Semantic web service composition in social environments". In: *International semantic web conference*. Springer. 2009, pp. 344–358 (cit. on p. 67).

[KS94] S. Koenig, R. G. Simmons. “Risk-sensitive planning with probabilistic decision graphs”. In: *Principles of Knowledge Representation and Reasoning*. Elsevier. 1994, pp. 363–373 (cit. on p. 67).

[LZZ11] J. Luo, C. Zhu, W. Zhang. “Messy Genetic Algorithm for the Optimum Solution Search of the HTN Planning”. In: *Foundations of Intelligent Systems*. Springer, 2011, pp. 93–98 (cit. on p. 67).

[NAI+03] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, F. Yaman. “SHOP2: An HTN planning system”. In: *Journal of artificial intelligence research* 20 (2003), pp. 379–404 (cit. on pp. 41, 67).

[NCLM99] D. Nau, Y. Cao, A. Lotem, H. Munoz-Avila. “SHOP: Simple hierarchical ordered planner”. In: *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*. Morgan Kaufmann Publishers Inc. 1999, pp. 968–973 (cit. on pp. 19, 33, 35, 41).

[Par14] T. Parr. *ANTLR*. 2014. URL: https://www.antlr.org/ (cit. on pp. 42, 43).

[RN16] S. J. Russell, P. Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016 (cit. on pp. 15, 22).

[SBM08] S. Sohrabi, J. A. Baier, S. A. McIlraith. “HTN Planning with Quantitative Preferences via Heuristic Search”. In: *Eighteeth International Conference on Automated Planning and Scheduling (ICAPS) Workshop on Oversubscribed Planning and Scheduling*. Sydney, Australia, Sept. 2008 (cit. on p. 25).

[SBM09] S. Sohrabi, J. A. Baier, S. A. McIlraith. “HTN planning with preferences”. In: *Twenty-First International Joint Conference on Artificial Intelligence*. 2009 (cit. on p. 67).

[SL08] Y. Shoham, K. Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008 (cit. on p. 22).

[SPW+04] E. Sirin, B. Parsia, D. Wu, J. Hendler, D. Nau. “HTN planning for web service composition using SHOP2”. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 1.4 (2004), pp. 377–396 (cit. on pp. 18, 67).

[TDD96] A. Tate, B. Drabble, J. Dalton. *O-Plan: a knowledge-based planner and its application to logistics*. University of Edinburgh, Artificial Intelligence Applications Institute, 1996 (cit. on p. 18).

[VLP08] F. Van Harmelen, V. Lifschitz, B. Porter. *Handbook of knowledge representation*. Vol. 1. Elsevier, 2008 (cit. on p. 17).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature