Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Replanning in Risk Aware HTN Planning

Lukas Nabakowski

Course of Study:	Informatik
Examiner:	Prof. Dr. Marco Aiello
Supervisor:	Ebaa Alnazer, M.Sc.
Commenced:	November 4, 2024
Completed:	May 5, 2025

Abstract

Hierarchical Task Network (HTN) planning is a powerful paradigm for modeling complex decision-making problems through task decomposition. However, traditional HTN planning assumes deterministic task costs, limiting its applicability in real-world scenarios where execution outcomes are uncertain. Risk-aware HTN planning introduces the use of utility theory to rank plans in domains with probabilistic costs, according to an agents risk attitude. However, the utility of a plan is only evaluated during initial planning. Agents with dynamic risk attitudes lack a mechanism to reevaluate the plan during execution, if unexpected high or low costs are incurred. We address this gap, by developing an approach for replanning in risk-aware HTN planning. We introduce a framework to model risk-aware replanning problems. Additionally, we present an algorithm to solve such problems, allowing agents to modify the plan during execution to represent their change in risk-attitude. We implement this solution in Risk Aware Replanning in JSHOP2 (RARJSHOP2), an extension of the Java Simple Hierarchical Ordered Planner 2 (JSHOP2), supporting probabilistic costs and automatic replanning, based on changes in plan utility due to incurred costs. Evaluation in an electric vehicle routing domain demonstrates that our approach successfully adapts plans according to changing risk preferences, though at increased computational cost.

Kurzfassung

HTN-Planning ist ein mächtiges Paradigma zur Modellierung komplexer Entscheidungsprobleme durch Task-Decomposition. Traditionelles HTN-Planning geht jedoch von deterministischen Task-Kosten aus, was ihre Anwendbarkeit in realen Szenarien mit unsicheren Ausführungsergebnissen einschränkt. Risk-aware HTN-Planning erweitert dieses Modell mit der Anwendung von Utility-Theorie, um Pläne in Domains mit probabilistischen Kosten entsprechend der Risk-Attitude eines Agenten zu bewerten. Der Utility-Wert eines Plans wird jedoch ausschließlich während der initialen Planung berechnet. Agenten mit dynamischer Risk-Attitude fehlt ein Mechanismus, um den Plan während der Ausführung neu zu bewerten, wenn unerwartet hohe oder niedrige Kosten auftreten. Wir schließen diese Lücke, indem wir einen Ansatz zur Replanung im Kontext von risk-aware HTN-Planning entwickeln. Wir stellen ein Framework vor, das risk-aware Replanning-Probleme formal modelliert. Darüber hinaus präsentieren wir einen Algorithmus zur Lösung dieser Probleme, der es Agenten ermöglicht, den Plan während der Ausführung anzupassen, um Änderungen in ihrer Risk-Attitude zu berücksichtigen. Wir implementieren unsere Lösung in RARJSHOP2, einer Erweiterung des JSHOP2-Planners, die probabilistische Kosten sowie automatisches Replanning, basierend auf Änderungen in der Plan-Utility aufgrund angefallener Kosten, unterstützt. Die Evaluation in einer Domain zur Routenplanung für Elektrofahrzeuge zeigt, dass unser Ansatz Pläne erfolgreich an veränderte Risk-Attitudes anpasst, allerdings mit erhöhtem Rechenaufwand.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

AI Artificial Intelligence. 21

ANTLR Another Tool for Language Recognition. 49

HTN Hierarchical Task Network. 3

JSHOP2 Java Simple Hierarchical Ordered Planner 2. 3

PANDA Planning and Acting in a Network Decomposition Architecture. 23

RARJSHOP2 Risk Aware Replanning in JSHOP2. 3

SHOP Simple Hierarchical Ordered Planner. 38

SHOP2 Simple Hierarchical Ordered Planner 2. 23

1 Introduction

Automated Planning aims to calculate sequences of actions, or plans, that solve a given problem when executed step by step. Hierarchical Task Network (HTN) planning is a widely used technique in this field, leveraging a hierarchical structure where high-level tasks are decomposed into sequences of lower-level tasks until primitive, executable actions are reached [GNT04].

Traditional HTN planning assigns fixed costs to primitive tasks to represent resource expenditures, such as time or energy. However, real-world scenarios often involve risk, for example, driving between locations may incur variable time or energy costs due to traffic, weather, or terrain. To address this, Alnazer et al. [AGA22] introduced a risk-aware HTN planning framework that models probabilistic costs. Their approach employs utility theory to evaluate plans based on an agent's risk attitude. Risk attitudes are represented using utility functions, which define how risk seeking or risk averse an agent is.

While this framework enables risk-sensitive decision-making before execution, it lacks mechanisms to adapt plans during execution if actual costs deviate from expectations. For instance, unexpectedly low costs might incentivize an agent to adopt a more risk seeking attitude for remaining tasks, while high incurred costs could make an agent more risk averse. This work is concerned with, how agents can change their risk attitude during execution of a plan and modify the remaining plan to reflect this change in risk attitude.

We present four key contributions:

- a formal model for replanning in risk-aware HTN planning that captures dynamic risk attitude changes during execution,
- an algorithm for solving risk-aware replanning problems,
- Risk Aware Replanning in JSHOP2 (RARJSHOP2), an implementation extending Java Simple Hierarchical Ordered Planner 2 (JSHOP2) with our risk-aware replanning capabilities, and
- a comprehensive evaluation demonstrating our approach's effectiveness using a risk-aware domain.

We developed our solution through a systematic process, beginning with comprehensive study of HTN planning and utility theory fundamentals. After analyzing the risk-aware HTN planning framework, we surveyed state-of-the-art HTN planners to identify a suitable implementation platform. Selecting JSHOP2 as our base system, we first enhanced it with core risk-awareness capabilities for handling probabilistic costs. Building on the risk-aware HTN planning framework, we formalized a model for risk-aware replanning that allows dynamically adjusting plans during execution based on evolving risk preferences. This model was then implemented as an extension to JSHOP2, creating our RARJSHOP2 system. To validate the approach, we designed a risk-involving electric vehicle routing domain with variable energy costs and generated representative test problems.

Finally, we conducted an extensive evaluation of RARJSHOP2, testing its performance, using different utility functions and parameter configurations, to assess both plan quality according to risk attitudes, and computational efficiency.

The remainder of this work is structured as follows. Chapter 2 establishes the fundamentals of HTN planning and utility theory, as well as the risk-aware HTN planning framework, on which our approach is based on. Chapter 3 then introduces our approach to modeling risk-aware HTN replanning problems and presents the algorithms, used to solve such problems. In Chapter 4, we detail our implementation of these algorithms, explaining how we extended the JSHOP2 planner. Chapter 5 evaluates our implementation on a risk-aware planning domain, analyzing results across different parameter configurations for multiple utility functions. Chapter 6 positions our work within the research landscape, comparing them with similar works, concerned with risk-awareness or replanning in HTN planning. Finally, Chapter 7 summarizes our findings and outlines promising directions for future work.

2 Background

This chapter provides the foundational knowledge required to understand the contributions of our work. We begin with the concept of automated planning. Next, we cover a specific planning technique, called HTN planning. We then present utility theory, a mathematical framework for modeling risk-sensitive decision-making. Finally, we explain how these concepts have been integrated in risk-aware HTN planning.

2.1 Automated Planning

Automated Planning is an area within Artificial Intelligence (AI) that focuses on the deliberation process of selecting and organizing actions to achieve a specific goal [GNT04]. This deliberation is required, when an activity deals with tasks that are complex, unfamiliar, or require careful adaptation to constraints. Such constraints might include operating in high-risk or high-cost environments, coordinating actions with other agents, or synchronizing with dynamic systems that change over time. There are various techniques in Automated Planning for modeling a planning problem, with HTN and Classical Planning being two notable examples [GNT04].

While this work primarily focuses on HTN planning, we will use Classical Planning as an example to explain the concept of Automated Planning. To better understand this concept, consider a practical example: a robot tasked with delivering packages between homes. In this scenario, packages are initially located at specific homes and need to be transported to other designated homes. The robot can perform actions like picking up and dropping packages, as well as driving between adjacent homes. The challenge lies in figuring out the best sequence of actions for the robot to achieve its goal efficiently.

Classical Planning is inherently domain-independent, meaning it can be applied to any problem that can be formally described using its abstract representation. This generality eliminates the need for domain-specific knowledge when designing the planning algorithm itself, making it a highly versatile and reusable approach applicable to a wide range of planning problems. Domain-independent planning is particularly valuable because it reduces the cost and effort of developing specialized algorithms for each new problem.

In Classical Planning, the process begins with two key inputs: a *domain* and a *problem*. The domain describes the general framework of the environment, including the types of actions that can be taken and the rules governing how those actions affect the world. The problem, on the other hand, provides the specific details about the environment and the task at hand [GNT04]. To describe these, the concept of the *state* is used. A state is a complete snapshot of the environment at a given moment, represented as the set of all currently true propositions about the environment.

For the delivery robot example, the domain would define the basic concepts, such as what a home or a package is, and specify the actions available to the robot, such as picking up a package, dropping it, or driving from one home to another. Whereas the problem defines the *initial state*. The initial state is the state of the environment before any action has been taken. It would include which homes and packages exist, their relationships, such as which homes are adjacent to each other, and where the robot and packages are located initially. For instance, the problem might describe a setup with three homes $Home_1$,$Home_2$, and $Home_3$ and two packages $Package_1$ and $Package_2$, where home $Home_1$ is next to home $Home_2$, and home $Home_2$ is next to home $Home_3$ and package $Package_1$ is at home $Home_1$, package $Package_2$ is at home $Home_2$, and the robot starts at home $Home_3$. The problem also specifies the *goal state*, which is the desired state to be achieved, For example, the goal state could require that package $Package_1$ be at home $Home_2$ and package $Package_2$ at home $Home_3$. The actions defined in the domain have the capability to alter the state of the environment. For example, if the robot drives from home $Home_3$ to home $Home_2$, the state changes to reflect that the robot is now located at home $Home_2$. The objective of automated planning is to identify a plan, which is a sequence of actions that, when applied to the initial state, results in the achievement of the goal state. One possible plan in this scenario could be to let the robot drive to $Home_2$, pickup $Package_2$, drive to $Home_3$, drop $Package_2$, drive to $Home_2$, drive to $Home_1$, pickup $Package_1$, drive to $Home_2$ and drop $Package_1$.

Automated Planning is fundamentally concerned with computing a sequence of actions that achieves a given goal. While finding any valid plan may suffice in some cases, many applications require additional considerations [GNT04]. These include minimizing computation time or improving the quality of the resulting plan. Plan quality can be measured in various ways, such as the number of actions, the total execution costs, or other domain-specific criteria.

When total execution cost is considered as a quality metric, each action has an associated cost, which represent the resources required to execute them, such as time, energy, or fuel [GNT04]. For example, if driving between homes might have a cost of 1 unit, the plan above would have a total cost of 5, due to five drive actions. However, alternative plans with lower costs might exist. For instance, the robot could drive to $Home_2$, drive to $Home_1$, pickup $P1$, drive to $Home_2$, drop $Package_1$, pickup $Package_2$, drive to $Home_3$ and drop $Package_2$. This plan would have a total cost of 4, as it involves only four drive actions.

We refer to an algorithm or software system, that solves planning problems, as a *planner* or a *planning system*.

2.2 HTN Planning

HTN is an approach to domain-independent automated planning. Unlike classical planning, which aims to achieve specific goals, in HTN planning, the objective is to perform a set of tasks. A task can be either a *primitive task*, which represents an action that can be directly executed, or a *compound task*, which represents a complex objective. Compound tasks are resolved through *decomposition*, breaking them down into smaller subtasks that define how the higher-level task can be accomplished. Since subtasks can themselves be primitive or compound, HTN planning forms a hierarchical structure of tasks.

Like Classical Planning, HTN planning requires a domain and a problem as input. The domain includes the primitive and complex tasks, as well as *operators*, that describe when a primitive task can be executed and how it changes the resulting state, and *methods* that describe when and how compound task can be decomposed into subtasks. The problem includes the initial state and a list of goal tasks that need to be achieved. The HTN planning process works by recursively decomposing compound tasks until a sequence of primitive tasks has been reached, that can be executed from the initial state. Due to its ability to encode problem-solving 'recipes' that align with human domain expertise, HTN has been widely used to solve problems in different domains [GNT04]. Such domains include web service composition [KGS05; SPW+04] and adversarial video game AI [LABX20].

Several HTN planners and planning systems with HTN capabilities have been developed, including Simple Hierarchical Ordered Planner 2 (SHOP2) [NAI+03], Planning and Acting in a Network Decomposition Architecture (PANDA) [HBBB21], and SH [GPAA24]. HTN planners can be categorized into state-based and plan-based planners, according to the strategy they use to solve HTN planning problems. Since this work uses a state-based HTN planner, we outline the definitions and approach for state-based HTN planning.

Multiple works use slightly different definitions of HTN concepts. In this document, we adapt definitions from [GA14; GA15; GNT04].

Tasks are the representation of any action that can be taken and are separated into compound and primitive tasks. Primitive tasks represent actions that can be taken directly, whereas compound tasks represent complex actions, which need to be decomposed into one or more tasks.

Definition 2.2.1 (Task)
A task t is the representation of an action that can be performed. Tasks can be either primitive, denoted as t_p, or compound, denoted as t_c.

Consider again the example of the delivery robot. Picking up a package would be represented by a primitive task, as it is one of the fundamental actions the robot can take. Delivering a package however would be represented as a compound task because it involves multiple sequential steps. Specifically, the robot must first drive to the package's location, pick up the package, then drive to the destination, and finally drop the package. Each of these steps constitutes a subtask within the overall compound task.

A *predicate* is a logical statement, which evaluates to true or false and gives information about the state of objects in the environment.

Definition 2.2.2 (Predicate)
A predicate p is a tuple $(symbol(p), terms(p))$ where $symbol(p)$ is a predicate symbol and $terms(p)$ is a list of terms. A term is either a constant symbol or a variable symbol.

In the delivery robot domain, there is need to keep track of where the robot is located, or which packages exist. A predicate $(robotAt, (Home_1))$, for example, would indicate, that the robot is at home $Home_1$. Similarly, a predicate $(at, (Package_2, Home_2))$ would specify, that the package $Package_2$ is at home $Home_2$.

The state is the set of all predicates, which evaluate to true. Conversely, all predicates, which are not included in the state evaluate to false.

Definition 2.2.3 (Ground Predicate)
A predicate p_g is ground, if all terms in $terms(p_g)$ are constant symbols.

Definition 2.2.4 (State)
A state s is a set of ground predicates $s = \{p_1, ...p_n\}$.

For each primitive task there exists exactly one operator. An operator defines the conditions under which a primitive task can be executed and the changes it causes in the state. The *preconditions* outline what must be true or false in the current state for the operator to be applicable, with true conditions represented by positive preconditions and false conditions by negative preconditions. The effects describe how the state changes by adding new predicates to the state through positive effects or removing existing ones through negative effects after the operator is applied.

Definition 2.2.5 (Operator)
An Operator o is a triple $(t_p(o), pre(o), eff(o))$, where $t_p(o)$ is a primitive task, $pre(o)$ are preconditions and $eff(o)$ are effects. $pre^+(o)$ and $eff^+(o)$ denote the positive preconditions and effects, while $pre^-(o)$ and $eff^-(o)$ denote the negative preconditions and effects.

An operator o_{pickup} with $t_p(o_{pickup}) = t_{pickup}(pkg)$ would specify the preconditions and effects for the primitive task t_{pickup}, which represents the robot picking up a package pkg. The preconditions would consist of following positive preconditions $pre^+(o_{pickup}) = \{(robotAt, (loc)), (at, (pkg, loc))\}$. These ensure, that the robot is at some location loc, at which the package pkg is also located, since the robot can only pick up a package, if they are at the same location. The effects would consist of one positive effect $eff^+(o_{pickup}) = \{(have, (pkg))\}$ and one negative effect $eff^-(o_{pickup}) = \{(at, (pkg, loc))\}$. The positive effect indicates, that the robot now holds the package, while the negative effect would remove the predicate stating that the package is at location loc, making this statement false in the resulting state.

Methods define how a compound task decomposes into a task network, as well as the preconditions for this decomposition. The *task network* represents the subtasks that must be executed to accomplish the compound task, along with constraints on their ordering. The strict partial order $\prec$ specifies which tasks must precede others when the order matters, while leaving unrelated tasks unordered for flexible execution. For a single complex task there may exist multiple different methods, defining different ways to accomplish the task.

Definition 2.2.6 (Task Network)
A task network tn is a pair $(T, \prec)$, with T a finite set of tasks and $\prec$ a strict partial order on T.

Definition 2.2.7 (Method)
A method m is a triple $(t_c(m), pre(m), tn(m))$, where $t_c(m)$ is a compound task, $pre(m)$ are preconditions and $tn(m)$ is a task network.

For the delivery robot domain, a suitable complex task $t_{deliver}(pkg, loc)$ would represent the delivery of a package pkg to a location loc. To achieve this task, the robot would need to travel from his current location to the pickup location, pickup the package, travel to the delivery destination and drop the package. The method $m_{deliver}$ with $t_c(m_{deliver}) = t_{deliver}$ defines this decomposition. The preconditions $pre(m_{deliver})$ for this method would include the two

predicates $(at, (pkg, pickupLoc))$ and $(robotAt, (currentLoc))$, which let us capture the locations $pickupLoc$ and $currentLoc$, at which the package pkg and the robot are situated. The task network $t_c(m_{deliver})$ would contain following tasks: $t_1 = t_{travel}(currentLoc, pickupLoc)$, $t_2 = t_{pickup}(pkg)$, $t_3 = t_{travel}(pickupLoc, loc)$ and $t_4 = t_{drop}(pkg)$. The partial order $\prec$ for this task network would require:

- $t_1 \prec t_2$ (must travel to pickup location before picking up)
- $t_2 \prec t_3$ (must pick up before traveling to destination)
- $t_3 \prec t_4$ (must arrive before dropping)

This creates a total order $t_1 \prec t_2 \prec t_3 \prec t_4$ where each task must be completed before the next can begin. The total order reflects the strict sequential dependencies in package delivery: the robot cannot pick up what it hasn't traveled to, nor deliver what it hasn't picked up. Notably, there are no unordered task pairs in this network as each action logically depends on the previous one.

A planning problem formalizes a problem to be solved within domain D, where s_0 represents the initial world state, the task network tn_0 specifies both the tasks to be accomplished and their ordering constraints, while D provides the available operators O for primitive tasks and methods M for decomposing compound tasks.

Definition 2.2.8 (Planning Domain)
A planning domain is a pair $D = (O, M)$, where O is a set of operators and M is a set of methods.

Definition 2.2.9 (Planning Problem)
A planning problem is a tuple $P = (s_0, tn_0, D)$, where s_0 is the initial state, tn_0 is the initial task network and D is a domain.

For example the delivery robot domain $D_{robot} = (O_{robot}, M_{robot})$ would consist of the operators $O_{robot} = \{o_{pickup}, o_{drop}, o_{drive}\}$ for the primitive tasks $t_{pickpup}$, t_{drop} and t_{drive}, respectively. The methods $M_{robot} = \{m_{deliver}, m_{travel}\}$ contains methods for the compound tasks $t_{deliver}$ and t_{travel}. The task $t_{deliver}$ and the method $m_{deliver}$ have been described above. The task t_{travel} represents the robot traveling from any location to any other and the method m_{travel} allows decomposition into multiple t_{drive} tasks, that would take the robot to the location via intermediate locations. [1]

$P_{robot} = (s_{0-robot}, tn_{0-robot}, D_{robot})$ is a HTN planning problem for the delivery robot problem described in section 2.1. The initial state

$$\begin{aligned} s_{0-robot} = \{&(nextTo, (Home_1, Home_2)),\\ &(nextTo, (Home_2, Home_3)),\\ &(at, (Package_1, Home_1)),\\ &(at, (Package_2, Home_2)),\\ &(robotAt, (Home_3))\} \end{aligned}$$

[1] Multiple methods for edge cases have been omitted for simplicity. For example a method for decomposing $t_{deliver}$, when the package is already at its destination, or a method for decomposing t_{travel} when it is already at a location next to its destination.

describes an environment where three homes are connected linearly, with packages at Home_1 and Home_2, and the robot starting at Home_3. The task network

$$tn_{0-robot} = (\{t_{deliver}(Package_1, Home_2), t_{deliver}(Package_2, Home_3)\}, \emptyset)$$

specifies two delivery goals with no ordering constraints, allowing the robot to choose which package to deliver first.

The *solution* to a HTN planning problem is a sequence of operators, called a *plan*. This plan must be the result of multiple decompositions from the initial task network and be applicable to the initial state, with each operator being applicable in the state created by its predecessor.

Definition 2.2.10 (Plan)
A plan $\pi = (o_1, o_2, ...o_n)$ is a sequence of operators.

Definition 2.2.11 (Solution of a HTN Planning Problem)
A plan π is a solution for a planning problem $P = (s_0, tn_0, D)$ with $tn_0 = (T_{tn_0}, \prec_{tn_0})$ if one of the following cases apply.

- *tn_0 is empty and π is empty.*
- *There is a primitive task $t_p \in tn_0$ with no predecessors in tn_0, o_1 is applicable to t_p in s_0 and s_1 is the resulting state. The plan $\pi' = (o_2, o_3, ...o_n)$ is a solution for the planning problem $P' = (s_1, tn_0 \setminus t_p, D)$.*
- *There is a compound task $t_c \in tn_0$ with no predecessors in tn_0 and a method $m \in O$ that decomposes t_c into $tn(m) = (T_{tn(m)}, \prec_{tn(m)})$ and is applicable in s_0. The plan π is a solution to the planning problem $P'' = (s_0, tn'_0, D)$ with $tn'_0 = (T_{tn_0} \cup T_{tn(m)} \setminus t_c, \prec_{tn_0} \cup \prec_{tn(m)})$.*

State-Based HTN Planning

State-based HTN planners solve planning problems using an early commitment strategy, meaning they make decisions based on the current state at each step [GA14]. These planners begin from the initial state and progressively apply the first applicable method or operator, transforming the state with each action. They maintain and update the current state throughout the planning process, ensuring that operators are executed in a totally ordered sequence to avoid conflicts in preconditions or effects. If a chosen method leads to a dead-end with no further are applicable operators or methods, the planner backtracks to explore alternative paths, either by trying different variable bindings or selecting other decomposition methods.

2.3 Utility Theory

Utility theory allows modeling the preferences of a decision-maker, called an agent, when confronted with alternative choices [SL08]. Each choice comes with a different amount of gain. While a rational agent will always prefer a choice with higher gain over a choice with lower gain, the degree of preference can vary for different agents. This degree of preference is represented by the subjective value of utility the agent assigns to each choice, where choices with higher assigned utility are preferred to choices with lower assigned utility. The utility for a choice is derived from the utility function for that agent, that maps the gain of the choice to its utility.

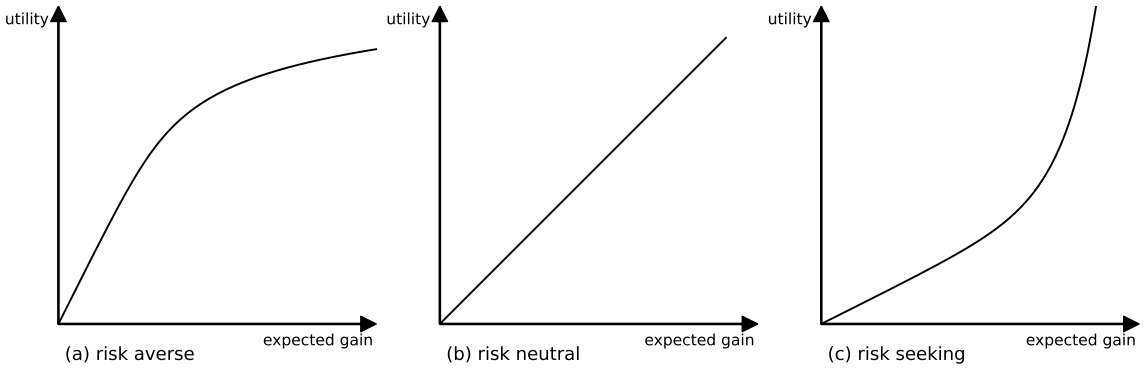


Figure 2.1: Utility functions for different risk attitudes

2.3.1 Risk Attitudes

In environments, where risk is involved, i.e. choices having multiple possible outcomes with associated gains and probabilities, the expected utility for a choice can be calculated by the sum of all gains, weighted by their respective probabilities. Agents are categorized into three risk attitudes: risk averse, risk neutral and risk seeking [SL08]. A risk averse agent, as the name implies, will try to avoid risky choices and assign a higher expected utility to choices with guaranteed gain. This preference pattern is captured by concave utility functions with diminishing marginal utility, where the rate of utility increase declines as gains grow. An example of a risk averse utility function is shown in Figure 2.1(a). A risk neutral agent evaluates choices purely based on their expected gain, showing no preference between certain and uncertain outcomes with equal expected gain. This is represented by linear utility functions where utility increases at a constant rate with gain, as seen in Figure 2.1(b). A risk seeking agent prefers uncertain choices that offer the possibility of higher gains, even when the expected gain is equal to or less than safer alternatives. This behavior is characterized by convex utility functions where equal increments in gain lead to progressively larger increases in utility. Figure 2.1(c) illustrates such a utility function.

Consider a simple decision scenario, where an agent must choose between receiving a guaranteed payment of 5€ or taking a gamble with equal chance of winning 10€ or receiving nothing. While both options have the same expected value of 5€, agents with different risk attitudes will evaluate these choices differently based on their utility functions. A risk averse agent with a utility function $u(x) = ln(x + 1)$, would calculate the expected utility of the safe option as $1 \cdot ln(5 + 1) \approx 1.792$. When evaluating the risky option, they would calculate an expected utility of $0.5 \cdot ln(0 + 1) + 0.5 \cdot ln(10 + 1) \approx 1.199$, making the risky option less appealing. In contrast, a risk seeking agent with a utility function $u(x) = e^x - 1$ would calculate the expected utility for the safe option as $1 \cdot e^5 - 1 \approx 147.4$ and expected utility of the risky option as $0.5 \cdot (e^0 - 1) + 0.5 \cdot (e^1 0 - 1) \approx 11012.7$, thus preferring the risky option.

Dynamic Risk Attitudes

The risk attitude of an agent must not remain static, but can vary depending on external factors, such as their wealth level. These changing risk attitudes are referred to as dynamic risk attitudes, whereas risk attitudes that do not change are considered static risk attitudes [AGA22]. Imagine an individual

having the choice between a guaranteed payout of 10000€ or a gamble to gain 20000€ with 50% probability or gain nothing. You might expect that the decision depends on the individual's current wealth [Bel88]. Someone with little money would probably take the safe €10,000, avoiding the risk of ending up empty-handed. But a wealthier person might prefer the gamble, since the chance to double the money outweighs the risk of losing. The risk attitude of that individual would be dynamic and depend on his wealth level.

2.3.2 One-Switch Utility Functions

To model dynamic risk attitudes, Bell [Bel88] introduces one-switch utility functions based on the one-switch rule.

A utility function is called a one-switch utility if it satisfies the one-switch property. These utility functions do not depend on the gain of a choice, but on the resulting wealth level. A utility function must be either a *quadratics*, *sumex*, *linear plus exponential*, or *linear times exponential* function to qualify as a one-switch utility function. One-switch utility functions allow a single switch of the risk attitude, depending on wealth level, such as being risk neutral for high wealth levels, but being risk averse for low wealth levels.

Definition 2.3.1 (One-Switch Rule)
An agent obeys the one-switch rule if, for every pair of alternatives whose ranking is not independent of wealth level, there exists a wealth level above which one alternative is preferred, below which the other is preferred.

Definition 2.3.2 (One-Switch Utility Function)
A utility function $u(w)$ is a one-switch utility function if and only if it belongs to one of the following families:
(i) the quadratics: $u(w) = aw^2 + bw + c$
(ii) the sumex functions: $u(w) = ae^{bw} + ce^{dw}$
(iii) the linear plus exponential: $u(w) = aw + be^{cw}$
(iv) the linear times exponential: $u(w) = (aw + b)e^{cw}$

Consider again the individual with a choice for a guaranteed 20000€ or the 50% chance to gain 40000€. When modeling their risk attitude with the one-switch utility function

$$u(w) = -e^{-0.00002w} + 0.003e^{0.00003w}$$

we can observe the change in risk attitude in relation to their wealth. For an initial wealth of 10000€, the certain option yields $u(10000 + 20000) \approx -0.475$ utility, whereas the gamble would only have a utility of $0.5 \cdot u(10000) + 0.5 \cdot u(10000 + 40000) \approx -0.505$, showing a risk averse attitude. With a higher wealth level of 50000€ however, their attitude becomes risk seeking, as the utility of $u(50000 + 20000) \approx -0.002$ for the certain option is lower than the utility of $0.5 \cdot u(50000) + 0.5 \cdot u(50000 + 40000) \approx 0.024$ for the gamble.

2.4 Risk-Aware HTN Planning

In real-world domains where actions incur uncertain costs, traditional HTN planning fails to account for risk-sensitive decision-making. Alnazer et al. [AGA22] address this limitation by introducing a risk-aware HTN planning framework grounded in expected utility theory, enabling agents to explicitly optimize plans according to probabilistic cost distributions and their risk attitude.

To allow modeling the risk in taking an action, a cost-variable operator is defined. This operator is similar to the operator in Definition 2.2.5, but contains multiple possible effects and their corresponding costs. These outcomes each have an associated probability of occurring. Additionally, since Alnazer et al. [AGA22] deal with a cost-based domain, only negative value costs are considered.

Definition 2.4.1 (Cost-Variable Operator)
A cost-variable operator is a tuple $o = (t_p(o), pre(o), eff(o), c(o))$*, where* $t_p(o)$ *and* $pre(o)$ *are defined as in Definition 2.2.5 and* $eff(o)$ *and* $c(o)$ *are tuples that represent the effects and costs of the operator and are defined as follows.*
$eff(o) = ((p_1(o), eff_1(o)), (p_2(o), eff_2(o)), \ldots, (p_n(o), eff_n(o)))$ *and*
$c(o) = ((p_1(o), c_1(o)), (p_2(o), c_2(o), \ldots, (p_n(o), c_n(o)))$
where $eff_i(o)$ *is the i-th effect with its corresponding cost* $c_i(o)$ *and probability* $p_i(o)$*. With* $\forall i \in [1, \ldots, n] : c_i(o) < 0$ *and* $0 < p_i(o) < 1$ *and* $\sum_{x=1}^{n} p_x(o) = 1$.

A risk-aware planning problem is similar to Definition 2.2.9 a tuple containing an initial state and task network, but uses a domain with cost-variable operators and includes a utility function, that expresses the risk attitude of the agent. The solution to a risk-aware is not simply a plan that achieves the initial task network from the initial state, but the plan with the maximum expected utility according to the chosen utility function.

Definition 2.4.2 (Risk-Aware Planning Problem)
A risk-aware planning problem is a tuple $P = (s_0, tn_0, D, U)$*, where* s_0 *is the initial state,* tn_0 *is the initial task network,* $D = (O, M)$ *is a risk-involving planning domain with cost-variable operators* O *and methods* M*, and* U *is a utility function that expresses a certain attitude ATT by evaluating the operator costs.*

Definition 2.4.3 (Solution of a Risk-Aware Planning Problem)
A plan π *is a solution to a risk-aware planning problem* P *if and only if it is a solution to* P *as described in Definition 2.2.11 and* π *has a maximum expected utility* $EU(\pi)$*, that reflects the chosen attitude ATT.*

A plan is divided into multiple segments, where each segment represents an action or action sequence with multiple possible outcomes, corresponding costs, and associated probabilities. The expected utility of a plan is then computed by aggregating the utilities of all possible trajectories over these segments weighted by their joint probabilities, where the utilities are calculated by using the sum of the trajectories costs as the input to the utility function.

Definition 2.4.4 (Expected Utility of a Plan)
The expected utility of a plan $\pi = (o_1, o_2, \ldots, o_f)$ is defined as:

$$E(u(\pi)) = \sum_{i=1}^{x_i} \sum_{j=1}^{x_j} \ldots \sum_{m=1}^{x_m} \left[p_{1i} p_{2j} \ldots p_{sm} u \left(c_{1i}(o_1) + c_{2j}(o_2) + \ldots + c_{sm}(o_f) \right) \right]$$

where:

- *s is the number of segments in the plan.*
- *$k, l, \ldots, n$ are the numbers of possible effects or costs for the first, second, ..., and last operator.*
- *$c_{1i}, c_{2j}, \ldots, c_{sm}$ are the costs of operators in the first, second, ..., and s-th segments.*
- *$p_{1i}, p_{2j}, \ldots, p_{sm}$ are the corresponding probabilities of operators in the first, second, ..., and s-th segments.*

2.4.1 Planning with Dynamic Risk Attitudes

For agents with dynamic risk attitude a disposable resource is used instead of the traditional wealth level. Unlike wealth, using a resource is more general, because it can represent anything that is finite and consumed over time. For instance, the resource could be energy, fuel, or time. This resource is a positive real value $R > 0$, that can be used a limited number of times, until it is exhausted. Each time an operator is executed, the resource is decreased by an amount equivalent to the cost of the operator. In the delivery robot example, the robot could contain a battery from which it uses energy to execute actions. The resource could represent the energy available to the robot.

Alnazer et al. [AGA22] define a family of one-switch utility functions, referred to as U_d, as shown in Equation (2.1), where:

- $\mathcal{D} > 0$ determines the tradeoff between risk neutrality and risk aversion,
- α determines the degree of risk aversion,
- R represents the resource.

This utility function allows for a single switch in risk attitude, from risk neutral at a high resource value to risk averse at a low resource value.

$$U_d(R + c_j(o_i)) = R + \mathcal{D} \frac{1 - e^{-\alpha(R + c_j(o_i))}}{\alpha} \tag{2.1}$$

3 Replanning in Risk-Aware HTN Planning

This chapter presents the core of our work. We begin by motivating the need for risk-aware HTN replanning. Following this, we introduce the formal definitions for our approach. We then describe the algorithms used, to solve the resulting replanning problems. Finally, we introduce the utility functions that are later used to model dynamic risk attitudes within our framework.

3.1 Motivation

As discussed in Section 2.2, HTN planning is a widely used approach to solving planning problems. Current HTN planners like SHOP2, JSHOP2, and PANDA allow plan quality to be measured through the summation of operator costs, where each cost typically represents a fixed resource expenditure such as time, energy, or monetary cost. Real-world problems, however, often involve situations where an action can produce multiple possible outcomes, each requiring a different amount of resources. Consider the delivery robot example, where the robot navigates residential neighborhoods to deliver packages. With traditional HTN planning, a planner can only assign a fixed time estimate for traveling between two homes, but the actual duration can vary significantly due to factors like obstacles in the pathway, or varying weather conditions. This illustrates a limitation of conventional HTN planning, which is its inability to accurately model and plan for actions with probabilistic outcomes and variable costs.

Utility theory, as discussed in Section 2.3, provides a framework for decision-making under risk by quantifying preferences over alternatives with probabilistic outcomes. Through utility functions that encode specific risk attitudes, individuals can evaluate choices between such alternatives and make decisions, that align with their preferences. The risk-aware HTN planning approach, shown in Section 2.4, builds on this foundation. It leverages utility theory to rank potential plans based on their expected utility, considering the probabilistic cost distributions of operators. Risk-aware HTN planning allows agents to identify plans with cost-variable operators that align with their preferred risk profile. A risk averse agent would opt for a plan with less risky operators, even if the expected cost is not the most favorable. On the other hand, a risk seeking agent would be inclined to choose a plan with operators that present a chance for low costs, despite the potential for higher costs.

However, one limitation of risk-aware HTN planning is, that it cannot modify the plan during execution. While an agent may select a plan that aligns with their chosen risk attitude, the actual outcomes during execution might differ from what was expected. These outcomes, though not anticipated, are still possible due to the probabilistic nature of the operator costs. In such cases, the agent might experience outcomes that reflect good or bad luck. As a result, the agent may want to adjust their risk attitude, shifting to a more risk averse or risk seeking stance, based on the outcomes, even though the initial plan was chosen according to a fixed preference. Alnazer et al. [AGA22] have explored the use of a one-switch utility function in conjunction with a declining resource

to model dynamic risk attitudes. This approach, though, only addresses the situation before the execution of a plan and does not consider changes to the plan that would reflect a dynamic shift in the agent's risk attitude during execution.

For example, the delivery robot might use a risk averse attitude when planning its route with risk-aware HTN planning. This would result in a plan that avoids routes with high risks, even if it means the delivery time is not the shortest expected. The chosen route would minimize the chance of very high delivery times. However, if the robot is lucky during execution and the travel times between homes turn out to be lower than expected, it might be beneficial for the robot to adjust the remaining route to take on more risk. Even if bad outcomes occur along the way, they will matter less since the earlier part of the plan was so successful.

In the following, we will present our approach for replanning in risk-aware HTN planning. This approach enables changing the plan during execution, to model dynamic risk attitudes.

3.2 Approach

As shown in Section 2.4, risk-aware HTN planning already accounts for the probabilistic costs and effects of cost-variable operators, using utility theory to select the plan that best reflects the agent's risk attitude. In general, we want a method that, upon meeting certain conditions during execution of a plan, can adjust the risk attitude and recompute the not yet executed part of the plan, that most reflects this new risk attitude. Utility theory supports such dynamically changing risk attitudes through one-switch utility functions, as detailed in Section 2.3.2. The application of these one-switch utility functions to risk-aware HTN planning has already been proposed, as discussed in Section 2.4.1. This is why we will base our approach on this proposition, adapting the definitions for risk-aware HTN planning from Section 2.4 and using the general HTN planning definitions from Section 2.2. The workflow of this general idea can be seen in Figure 3.1.

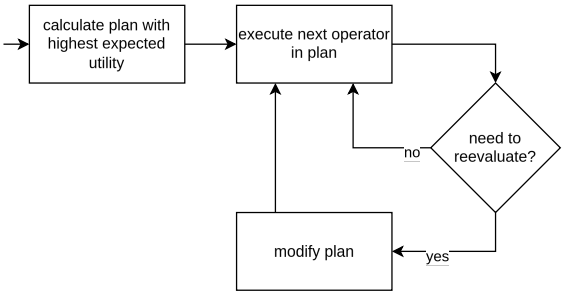


Figure 3.1: Workflow of the general approach.

3.2.1 Cost-Variable Operators

Risk-aware HTN planning uses cost-variable operators to encapsulate probabilistic effects and costs for executed actions. The definition for these cost-variable operators in Definition 2.4.1 allows for multiple effects and associated costs. We narrow this definition to only allow one single effect that

alters the state, as multiple effects would mean that a plan composed of cost-variable operators might not be applicable in every possible resulting state. This restriction is necessary because such multi-effect functionality is not supported in current HTN planners.

Additionally, the original definition assumed that every cost-variable operator has more than one possible cost and that each cost must have a value less than zero. Risk-aware HTN planning uses negative cost values due to its cost-based domain formulation. In our definition, costs are represented as positive values while maintaining their negative interpretation. This adjustment aligns with the notation of domain definitions for HTN planners, where positive cost values are typically used. Furthermore, we extend the definition to allow for cases where an operator may have only one possible cost with probability 1, including zero-cost operators. This relaxation is necessary because not every operator must be influenced by the modeled risk factors. For instance, in the delivery robot domain, while the drive operator might have probabilistic costs due to weather or traffic conditions, pickup and drop operators would have a single deterministic cost, as they are not affected by these conditions. Similarly, zero-cost operators are valuable for modeling actions with negligible resource requirements or those that primarily provide execution feedback without measurable cost implications.

The definition for our cost-variable operator can be seen in Definition 3.2.1. Like traditional HTN planning operators, it consists of an associated primitive task, preconditions and effects, but also contains one or more possible costs and their respective probabilities.

Definition 3.2.1 (Cost-Variable Operator)
A cost-variable operator is a tuple $o = (t_p(o), pre(o), eff(o), c(o))$*, where*

- $t_p(o)$ *is a primitive task,*
- $pre(o)$ *are preconditions,*
- $eff(o)$ *are effects, and*
- $c(o)$ *is a tuple that represents the cost distribution of the operator and is defined as follows:* $c(o) = ((p_1(o), c_1(o)), (p_2(o), c_1(o)), \ldots, (p_n(o), c_n(o)))$*, where* $c_i(o)$ *is a possible cost with its associated probability* $p_i(o)$*.*
  With $\forall i \in [1, \ldots, n] : 0 < p_i \leq 1$ *and* $\sum_{x=1}^{n} p_x(o) = 1$*.*

For example in the delivery robot domain, we can use a cost-variable operator o_{drive} to model risk in the time in minutes it takes the robot, to drive from one location to another. The operator could like this

$$o_{drive} = (t_p(o_{drive}), pre(o_{drive}), eff(o_{drive}), c(o_{drive}))$$

$$t_p(o_{drive}) = t_{drive}$$
$$pre^{+}(o_{drive}) = at(loc_1)$$
$$pre^{-}(o_{drive}) = \{\}$$
$$eff^{+}(o_{drive} = \{at(loc_2)\}$$
$$eff^{-}(o_{drive} = \{at(loc_1)\}$$

$c(o_{drive}) = ((0.4, 1), (0.3, 1.5), (0.3, 2))$

This operator is associated with the primitive task t_{drive} and represents the robot driving from location loc_1 to location loc_2. Therefore, it contains a positive precondition, stating the robot must be located at loc_1. The positive and negative effects state that after executing this operator, the robot is no longer at loc_1, but at loc_2. Finally, the cost distribution shows three different possible costs. The cost distribution offers three potential outcomes for this action. There is a 40% chance the robot takes 1 minute, while the probabilities are 30% each for durations of 1.5 or 2 minutes.

3.2.2 Partially Executed Plans

Before we can define risk-aware replanning problems, we must formalize the notion of partially executed plans. When an agent executes a plan composed of cost-variable operators, it incurs probabilistic costs. While these costs were uncertain during planning, once execution occurs, each operator's cost resolves to one specific value. These concrete costs must be saved to maintain an accurate representation of the current situation. A partially executed plan, as seen in Definition 3.2.2, represents a plan, that has been executed from operator o_1 up to some operator o_k. It captures both, which operators have already been executed, as well as the concrete costs c_Π, that have been incurred in their execution.

Definition 3.2.2 (Partially Executed Plan)
A partially executed plan, where the first k operators have been executed, is a tuple $\Pi = (\pi, k, c_\Pi)$*, where*

- $\pi = (o_1, o_2, \ldots, o_n)$ *is a plan with cost-variable operators,*
- k *is the amount of operators that have already been executed, and*
- $c_\Pi = (c_\Pi(o_1), c_\Pi(o_2), \ldots, c_\Pi(o_k))$ *are the experienced costs of already executed operators, with every* $c_\Pi(o_i)$ *being one of the possible costs in* $c(o_i)$*.*

When an agent executes a plan, the partially executed plan is continuously updated. Each time an operator is executed, the value k is increased by one, and the corresponding execution cost is appended to the cost list c_Π. For example, consider the plan $\pi = (o_1, o_2, o_3, o_4)$, where o_1, o_2, o_3, and o_4 are all drive operators o_{drive}. The initial partially executed plan $\Pi = (\pi, 0, ())$ starts with no executed operators and no incurred costs. Suppose the experienced cost for operator o_1 is 1.5. The partially executed plan is then updated by increasing k to 1, and the experienced cost of 1.5 added to the cost list. The resulting partially executed plan would look like this $\Pi = (\pi, 1, (1.5))$. Likewise, if the second operator o_2 is executed and incurs a cost of 1, k is increased to 2 and the cost of 1 is added to the cost list, resulting in a partially executed plan $\Pi = (\pi, 2, (1.5, 2))$. This process continues, until all operators in the plan have been executed.

The objective of risk-aware HTN planning problems is finding the plan with the highest possible expected utility. The approach for the expected utility of a plan in Definition 2.4.4 is, to traverse over every possible combination of outcomes from operators. For each combination, the total cost is summed up and used in the utility function to calculate the utility of that combination. This utility is then multiplied by the probability of the outcome. The weighted utilities for every combination are then summed up to get the expected utility of the plan.

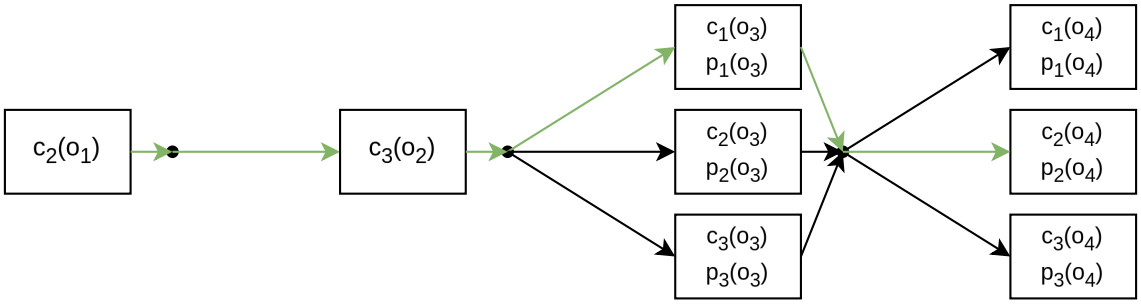


Figure 3.2: A partially executed plan with the four operators o_1, o_2, o_3, and o_4. The first two operators have been executed, resulting in costs $c_2(o_1)$ and $c_3(o_2)$, respectively. The other two operators each have three possible costs $c_i(o_j)$ with respective probabilities $p_i(o_j)$. A possible combination of costs is marked in green.

Because we deal with partially executed plans, we need to adapt this definition to consider, that some operators have already been executed. For the first k operators, the costs are already set. Thus, the other possible costs for these operators must not be considered. For the k+1-th to the last operator, we must consider all possible costs with their respective probabilities. We examine every possible combination of these set and potential costs. For every combination we calculate the utility and multiply it by the combination's probability. These weighted utilities are then summed up.

Since we want to use one-switch utility functions to encode dynamic risk attitudes, we also introduce a value R_0, which is similarly to the value R in Section 2.4.1 and defines an initial amount of resources. To calculate the utility of a single combination, we sum up the costs of that combination and subtract this from the initial resources. This results in a value, that represents the resources we would have left, if that specific combination of costs occurred. The utility function is then applied to this amount to get the utility of the combination.

The probability of a specific combination is the product of the probabilities associated with the costs from the k+1-th to the last operator, considered in the combination. Because the first k operators have already been executed, their outcome is set and the probability for that outcome must not be considered.

Figure 3.2 shows a partially executed plan with four operators, where the first two have already been executed. Traversing every path in the figure, yields every possible combination of costs. For example, the combination $c_2(o_1)$,$c_3(o_2)$,$c_1(o_3)$,$c_2(o_4)$ is marked in green. The utility from a utility function U of this combination would be $U(R_0 - c_2(o_1) - c_3(o_2) - c_1(o_3) - c_2(o_4)$, and would have to be weighted by $p_1(o_3) \cdot p_2(o_4)$.

The definition of expected utility for partially executed plans is presented in Definition 3.2.3. It closely mirrors the formulation in Definition 2.4.4, with a few key adjustments. First, the summation and probability calculations begin at operator $k + 1$, as the costs of the first k operators are already known. These fixed costs are represented by C, the sum of the already incurred costs. In addition, because we use one-switch utility functions to model changing risk attitudes, the utility is calculated based on the remaining resources. This is done by subtracting all costs of the combination from the initial resources R_0.

Definition 3.2.3 (Expected Utility of a Partially Executed Plan)
The expect utility of a partially executed plan $\Pi = (\pi, k, c_\Pi)$*, with* $\pi = (o_1, o_2, \ldots, o_n)$*, that has been executed until operator* o_k *is defined as*

$$E(u(\Pi)) = \sum_{i_{k+1}=1}^{x_{k+1}} \sum_{i_{k+2}=1}^{x_{k+2}} \ldots \sum_{i_n=1}^{x_n} [p_{i_{k+1}}(o_{k+1}) \ldots p_{i_n}(o_n) U(R_0 - C - c_{i_{k+1}}(o_{k+1}) - c_{i_{k+2}}(o_{k+2}) - \ldots - c_{i_n}(o_n))]$$

where

- C *is the cost of all already executed operators* $C = \sum_{j=1}^{k} c_\Pi(o_j)$*, where every* $c_\Pi(o_j)$*, is the experienced cost of executing operator* o_j*,*
- $x_{k+1}, x_{k+2}, \ldots, x_n$ *are the numbers of possible costs for the k-th, k+1-th, . . . , last operator,*
- $c_1(o_j), c_2(o_j), \ldots, c_{x_j}$ *are the possible costs for an operator* o_j*,*
- $p_1(o_j), p_2(o_j), \ldots, p_{x_j}$ *are their associated probabilities,*
- U *is the chosen utility function, and*
- R_0 *are the initial resources.*

3.2.3 Risk-Aware Replanning Problems

As stated before, the idea in replanning for risk-aware HTN planning is, to calculate a plan, according to a risk attitude, and start executing it. During execution, the risk attitude can change due to the experienced costs, and the plan is modified.

We can use a partially executed plan to track the state of the execution, updating it with every executed operator. At any point during execution, we can try to find a modified plan. These modified plans must start identically to the original plan, containing every operator, that has already been executed. The alternative plans all share the same sequence of executed operators, but differ in their remaining steps. Every alternative can be expressed as partially executed plan, with the incurred costs from the execution of the original plan. Then we can compare the alternatives by calculating their expected utilities. Now we choose the partially executed plan with the highest expected utility and continue the execution.

Risk-aware HTN replanning problems are formalized in Definition 3.2.4. Similar to the traditional HTN planning problem in Definition 2.2.9, it consists of an initial state s_0, an initial task network t_0, and a domain D. The operators in the domain must be cost-variable, as defined in Definition 3.2.1. Like the risk-aware planning problem in Definition 2.4.2, a utility function U defines the attitude of the agent. This utility function evaluates the initial resources R_0 minus the experienced and possible costs of operators in a partially executed plan. Finally, the current state of the execution is described by the partially executed plan Π_0.

Definition 3.2.4 (Risk-Aware Replanning Problem)
A risk-aware replanning problem is a tuple $P = (s_0, tn_0, D, U, R_0, \Pi_0)$*, where*

- s_0 *is the initial state,*
- tn_0 *is the initial task network,*

- $D = (O, M)$ *is a risk-involving planning domain with cost-variable operators* O *and methods* M*,*
- U *is a utility function, that expresses a certain attitude by evaluating the amount of resources left,*
- R_0 *is the initial amount of resources, and*
- Π_0 *is a partially executed plan,*

The solution to a risk-aware replanning problem is a new partially executed plan Π_s, as shown in Definition 3.2.5. This partially executed plan must start with the same operators $o_1, \ldots, o_k$, that have already been executed, and must have the incurred costs for these operators in its cost list. Only the rear part of the plan can differ from the original one. The plan must still be the result of decompositions from the initial task network and be applicable in the initial state, with each operator being applicable in the state created by its predecessor, like in Definition 2.2.11.

Definition 3.2.5 (Solution of a Risk-Aware Replanning Problem)
A partially executed plan $\Pi_s = (\pi_s, k_s, c_{\Pi_s})$ *is a solution to a risk-aware replanning problem* $P = (s_0, tn_0, D, U, R_0, \Pi_0, \delta)$*, with* $\Pi_0 = (\pi_0, k_0, c_{\Pi_0})$*, if and only if*

- π_s *is a solution to the HTN planning problem* $P' = (s_0, tn_0, D)$*, as per Definition 2.2.11,*
- $k_s = k_0$*,*
- *the first* k_0 *operators in* π_s *are the same as the first* k_0 *operators in* π_0*,*
- $c_{\Pi_s} = c_{\Pi_0}$*, and*
- Π_s *has the maximum expected utility* $E(u(\Pi_s))$*, that reflects the chosen attitude represented by* U*.*

This new partially executed plan can then be used to continue the execution. The costs incurred from the next operators can be added to the cost list of Π_s, then at any point, a new risk-aware replanning problem can be formulated, using Π_s as Π_0, and the plan can be altered again.

3.3 Replan Threshold

While we have explained how risk-aware replanning problems are formulated and what their solution looks like, we do not have a way to define when replanning should be initiated. One option is to always replan when the next operator is executed, but this can be inefficient, as planning might take time and the modified plan might be very similar, or even identical, to the original one when the incurred costs of operators are close to the average expected costs. To address this, we introduce a value $\delta >= 0$, which represents a replan threshold. When we calculate a plan, whether it is the first plan or the result of replanning, we calculate the utility u_0 of this partially executed plan. Each time an operator is executed, the incurred cost is added to the cost list of the current partially executed plan, and we calculate the new utility u_n. If the incurred cost is close to the average expected cost, the difference between u_0 and u_n will be small. However, if the cost incurred is much higher or lower than the average expected cost, the difference between u_0 and u_n will be large. We then look

at the ratio $r = \frac{u_n}{u_0}$. If the difference between u_0 and u_n is small, the ratio will be close to 1, while a larger difference will cause the ratio to deviate more from 1. The δ value defines a threshold for this ratio. If the ratio r deviates by more than δ from 1 (i.e., $r \geq 1 + \delta$ or $r \leq 1 - \delta$), we initiate replanning.

3.4 Algorithm

With the risk-aware replanning problem and the replan threshold formalized, we can now describe the algorithms used, to solve such problems. We begin by presenting the main algorithm, which provides a high-level overview of how execution of a plan and replanning interact. Following this, we explain the algorithm used to compute a plan with the highest expected utility, based on the chosen utility function and the agent's initial resource configuration. Finally, we describe the replanning procedure, used to calculate the best modified plan during execution.

3.4.1 Main Algorithm

The core algorithm is outlined in Algorithm 3.1. It is invoked by the function RUN, that takes the initial task network as an input (line 2). Initially, a utility function is selected that reflects the agent's preferences and risk attitude (line 3). The amount of initial resources, R_0, which is used in the utility function, is then set, along with the replan threshold, δ (lines 4–5). Once the planning parameters are defined, a plan p with the highest expected utility is computed using the function FINDPLAN (line 6), which will be explained in Section 3.4.2. The expected utility of this plan is stored in u_0 (line 7). The plan's execution then begins. For each operator in the plan, the incurred cost is recorded and added to the list of *costList* (lines 9–10). The new expected utility, u_n, is calculated using the function EXPECTEDUTILITY (line 11). The expected utility u_n differs from u_0, because EXPECTEDUTILITY considers the already incurred costs in *costList*, as per Definition 3.2.3. Then, the ratio $r = \frac{u_n}{u_0}$ is evaluated (line 12). If the ratio exceeds $1 + \delta$ or falls below $1 - \delta$, replanning is triggered (line 13). A modified plan is generated and stored in p using the REPLAN function (line 14), which will be explained in Section 3.4.2. The expected utility of this modified plan is then updated in u_0, and execution continues with the next operator (line 15). This process concludes once all operators in the current plan have been executed. Figure 3.3 illustrates the individual steps of the main algorithm.

3.4.2 Finding the best Plan

To find the plan with the highest possible expected utility, we use an adapted version of the Simple Hierarchical Ordered Planner (SHOP) algorithm [NCLM99]. This algorithm can be seen in Algorithm 3.2. We showcase changes to the original algorithm with red lines.

The function FINDPLAN begins by taking a task network as input and initializing several global variables (line 3). The variables *currentPlan* and *bestPlan* store the currently explored plan and the currently best found plan, respectively, and are initialized to empty plans. The lists *currentIndices* and *bestIndices* store the indices of for loops that lead to the current and best plan, and are initialized to empty lists. They will later be needed in the replanning algorithm. The algorithm then calls the

Algorithm 3.1 Main Algorithm

```
1:  Global Variables: costList, R_0

2:  function RUN(taskNetwork)
3:      choose utility function for EXPECTEDUTILITY
4:      initialize R_0 for EXPECTEDUTILITY
5:      initialize δ
6:      p ← FINDPLAN(taskNetwork)
7:      u_0 ← EXPECTEDUTILITY(p)
8:      for all operators o in p do
9:          input cost of o
10:         append cost to costList
11:         u_n ← EXPECTEDUTILITY(p)
12:         r ← u_n / u_0
13:         if r ≥ 1 + δ or r ≤ 1 − δ then
14:             p ← REPLAN(taskNetwork)
15:             u_0 ← EXPECTEDUTILITY(p)
16:         end if
17:     end for
18: end function
```

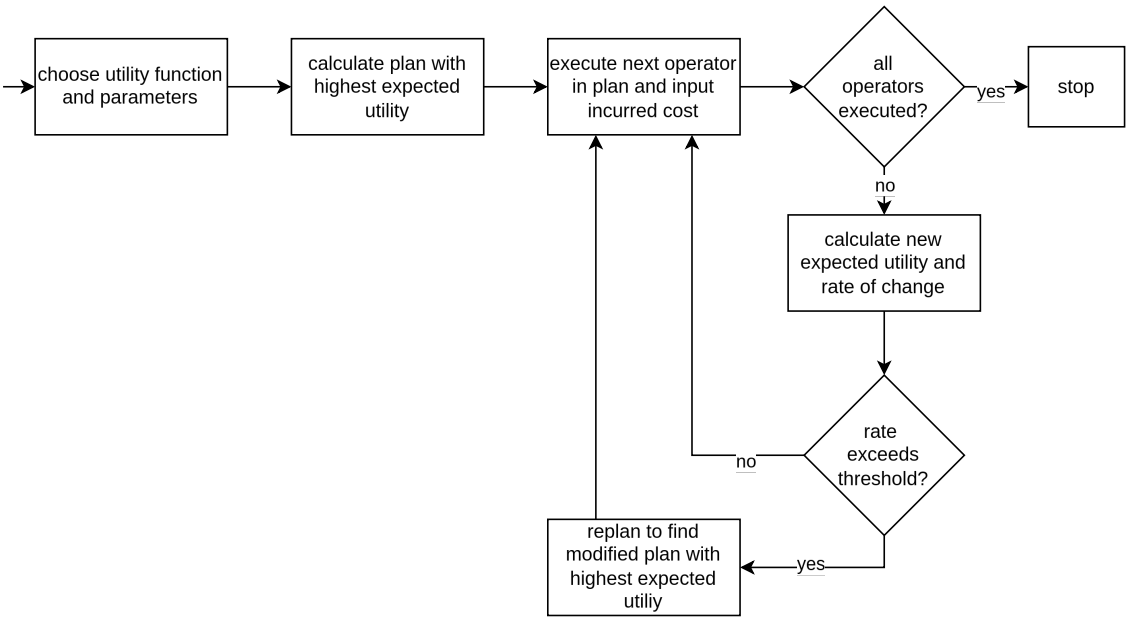


Figure 3.3: Workflow of the main algorithm.

recursive helper function FINDPLANHELPER to explore all possible plans, that solve the task network (line 4). Once the search is complete, the plan with the highest expected utility, stored in *bestPlan*, is returned (line 5).

The recursive function FINDPLANHELPER performs a depth-first traversal of the task network. If the network is empty, a complete plan has been constructed (line 8). At this point, the expected utility of the current plan is compared to that of the best plan found so far (line 9). If the new plan has higher expected utility, it and its corresponding indices in *currentIndices* are saved into *bestPlan* and *bestIndices*, respectively (lines 10–11). This allows the exact sequence of choices that led to the optimal plan to be reproduced later. The function then returns, backtracking to continue exploring alternative plans (line 13).

If the task network is not empty, its first task is stored in t (line 15). If this task is primitive (line 16), the task is removed from the task network (line 17), and all applicable operators and their valid bindings for variables are considered (lines 18 and 20). If an operator is applicable in the current state with the selected binding (line 22), it is applied to the current state, appended to the current plan, and the search continued recursively (lines 23–25). After recursion, the modifications to the state are undone for further exploration (line 26).

If the task is compound, the algorithm explores all matching decomposition methods and valid bindings (lines 35 and 37). For each possibility, the current task is replaced in the task network with its subtasks (line 39), and the recursive search continues (line 40). After recursion, the task network is restored to consider other decompositions (line 41).

The search concludes when all alternatives have been evaluated. At this point, *bestPlan* contains the plan with the highest expected utility.

Whenever the algorithm loops over multiple possible operators, bindings, or methods, the index of the current iteration is appended to the list *currentIndices* (lines 19, 21, 36, and 38). When the iteration ends, the index is removed from the list (lines 29, 31, 42, 44). If a plan is found, that has higher expected utility than the currently best known plan, these indices are saved in *bestIndices* (line 11). These stored indices allow the replanning algorithm later, to reconstruct the exact selections of operators, variable bindings, and method decompositions that led to the optimal plan.

3.4.3 Replanning

The replanning process is outlined in Algorithm 3.3. The function REPLAN has to be invoked after FINDPLAN has found the best plan, and needs to get the same initial task network as input (line 2). The list *costList* needs to contain the experienced costs for all already executed operators. First, like in FINDPLAN, *currentPlan* and *currentIndices* need to be initialized to an empty plan and an empty list (line 3). Then the variables *operatorCount* and *indexPointer* are set to 0 (lines 4–5). *operatorCount* will be used to track how many operators were added to the current plan and *indexPointer* will be used to traverse the indices in *bestIndices*. Then the function REPLANHELPER is called, which will find the new best plan (line 6). This new best plan is then returned (line 7).

Algorithm 3.2 Finding the Plan with the highest Utility. (Adapted from [NCLM99])

```
1: Global Variables: bestPlan, bestIndices, currentPlan, currentIndices

2: function findPlan(taskNetwork)
3:     initialize currentPlan, bestPlan, currentIndices
4:     findPlanHelper(taskNetwork)
5:     return bestPlan
6: end function

7: function findPlanHelper(taskNetwork)
8:     if taskNetwork is empty then
9:         if expectedUtility(currentPlan) > expectedUtility(bestPlan) then
10:            bestPlan ← clone of currentPlan
11:            bestIndices ← clone of currentIndices
12:        end if
13:        return
14:    end if
15:    t ← first task from taskNetwork
16:    if t is primitive then
17:        remove t from taskNetwork
18:        for all operators o that match t do
19:            add index to currentIndices
20:            for all valid bindings b do
21:                add index to currentIndices
22:                if o is applicable with b then
23:                    apply o to state
24:                    add o to currentPlan
25:                    findPlanHelper(taskNetwork)
26:                    undo o from state
27:                    remove o from currentPlan
28:                end if
29:                remove index from currentIndices
30:            end for
31:            remove index from currentIndices
32:        end for
33:        restore t to taskNetwork
34:    else
35:        for all methods m that match t do
36:            add index to currentIndices
37:            for all valid bindings b do
38:                add index to currentIndices
39:                replace t with decomposition from m using b
40:                findPlanHelper(taskNetwork)
41:                undo task replacement
42:                remove index from currentIndices
43:            end for
44:            remove index from currentIndices
45:        end for
46:    end if
47: end function
```

The function REPLANHELPER works similar to FINDPLANHELPER. But instead of iterating over all possible operators, decompositions, and bindings, it directly chooses them, like they were chosen for the best plan found by FINDPLAN. It saves the first task from the task network in t (line 14). Then it checks, if this task is primitive or compound (line 15).

If the current task is primitive, it is removed from the task list (line 16), and the algorithm retrieves the index of the recorded operator from *bestIndices* (line 17), finds the corresponding operator (line 18), and adds the index to *currentIndices* (line 19). It then increments *indexPointer* to retrieve the index of the corresponding binding (lines20–22), and adds this index to *currentIndices* as well (line 23). Then, *indexPointer* is increased again for the next choice of an operator or decompositions. The operator is applied to the current state (line 25), appended to *currentPlan*, and the count of executed operators is updated (lines 19–21). The function then recurses to continue traversing the task network.

If the task is compound, the algorithm retrieves the index of the recorded method and its corresponding binding in the same way (lines 30–37). It then replaces the task with the appropriate decomposition and proceeds recursively (lines 38–39).

By following the stored indices in *bestIndices*, this algorithm retraces the steps of the original best plan. When the current plan contains the same amount of operators, as the amount of recorded costs in *costList*, the algorithm has to reconstruct the plan to the point where execution was interrupted to replan (line 10). Then the function REPLANHELPER is used to find the plan with the highest expected utility, that starts with the already executed operators (line 11).

3.5 Utility Functions

As formalized in Definition 3.2.4, the risk attitude of an agent is defined by a utility function that evaluates an amount of resources. Any such utility function can be used to model the agent's preferences, but we focus on utility functions that encode dynamic risk attitudes since the motivation for risk-aware replanning is that agents want to adjust their risk attitude and then replan. One-switch utility functions are particularly useful in this context, as they allow encoding a single switch in the risk attitude. We present two families of utility functions that follow the one-switch rule, that allow to switch the risk attitude from risk neutral to risk averse, and from risk seeking to risk neutral, respectively.

The first family is denoted as U_a and is defined in Equation (3.1). Utility functions of that family model a change in the agent's risk attitude, where it switches from being risk neutral at high resource levels to becoming risk averse at low resource levels. This behavior can be seen in Figure 3.4. The function shown in yellow, grows linear for high resource R, meaning it encodes risk neutrality. Around $R = 0$ it changes to a concave shape, encoding risk aversion. It obeys the one-switch rule, because it is a linear plus exponential function of type $u(w) = a'w + b'e^{c'w}$, with $w = R$, $a' = 1$, $b' = D \cdot (-a)^{-1}$, and $c' = -a$.

$$U_a(R) = R + D \cdot (-a)^{-1} \cdot e^{-(a \cdot R)} \tag{3.1}$$

Algorithm 3.3 Retracing the executed Operators and initiating Replanning.

```
1: Global Variables: bestPlan, bestIndices, currentPlan, currentIndices, operatorCount, indexPointer, costList

2: function REPLAN(taskNetwork)
3:     initialize currentPlan, currentIndices
4:     operatorCount ← 0
5:     indexPointer ← 0
6:     REPLANHELPER(taskNetwork)
7:     return bestPlan
8: end function

9: function REPLANHELPER(taskNetwork)
10:    if operatorCount ≥ size of costList then
11:        FINDPLANHELPER(taskNetwork)
12:        return
13:    end if
14:    t ← first task from taskNetwork
15:    if t is primitive then
16:        remove t from taskNetwork
17:        i ← bestIndices[indexPointer]
18:        o ← operator matching t at index i
19:        add i to currentIndices
20:        indexPointer ← indexPointer + 1
21:        j ← bestIndices[indexPointer]
22:        b ← binding for o at index j
23:        add j to currentIndices
24:        indexPointer ← indexPointer + 1
25:        apply o to state
26:        add o to currentPlan
27:        operatorCount ← operatorCount + 1
28:        REPLANHELPER(taskNetwork)
29:    else
30:        i ← bestIndices[indexPointer]
31:        m ← method matching t at index i
32:        add i to currentIndices
33:        indexPointer ← indexPointer + 1
34:        j ← bestIndices[indexPointer]
35:        b ← binding for m at index j
36:        add j to currentIndices
37:        indexPointer ← indexPointer + 1
38:        replace t with decomposition from m using b
39:        REPLANHELPER(taskNetwork)
40:    end if
41: end function
```

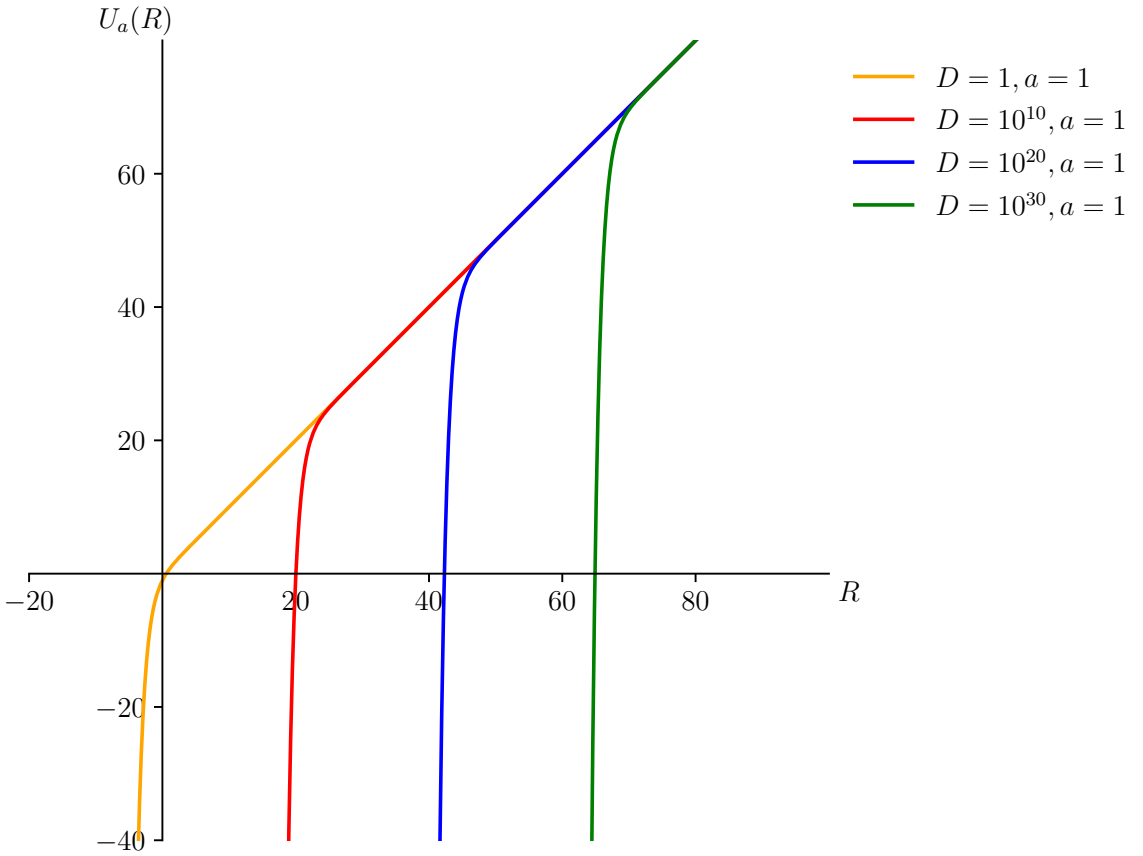


Figure 3.4: $U_a(R)$ utility functions with varying constant value D, which shifts the point, where the risk attitude changes from risk neutral to risk averse.

The parameter $D > 0$ shifts the point, where the risk attitude changes. In Figure 3.4, multiple functions of family U_a with different values for D can be seen. All functions show the same linear growth for high R and the same concave shape for low R. But the point at which the change from linear to concave occurs is shifted towards higher R for higher D.

The parameter $a > 0$ also shifts the point of change, but additionally controls, how fast that change occurs. In Figure 3.5, multiple functions of family U_a with different values for a are shown. Functions with higher a show a gradual change from risk neutrality to risk aversion, while also shifting the point of change towards higher R. Functions with lower a show an abrupt change from risk neutrality to risk aversion.

The second family of functions is denoted as U_s and is defined in Equation (3.2). Similarly to U_a, functions of family U_s encode a change of an agent's risk attitude depending on R. The function, shown in yellow in Figure 3.6, is convex for high R, encoding a risk seeking attitude. Around $R = 0$ it changes to a linear growth, encoding risk neutrality. Like U_a, it obeys the one-switch rule, because it is a linear plus exponential function of type $u(w) = a'w + b'e^{c'w}$, with $w = R$, $a' = 1$, $b' = F^{-1} \cdot b$, and $c' = b$.

(3.2) $U_s(R) = R + F^{-1} \cdot b \cdot e^{b \cdot R}$

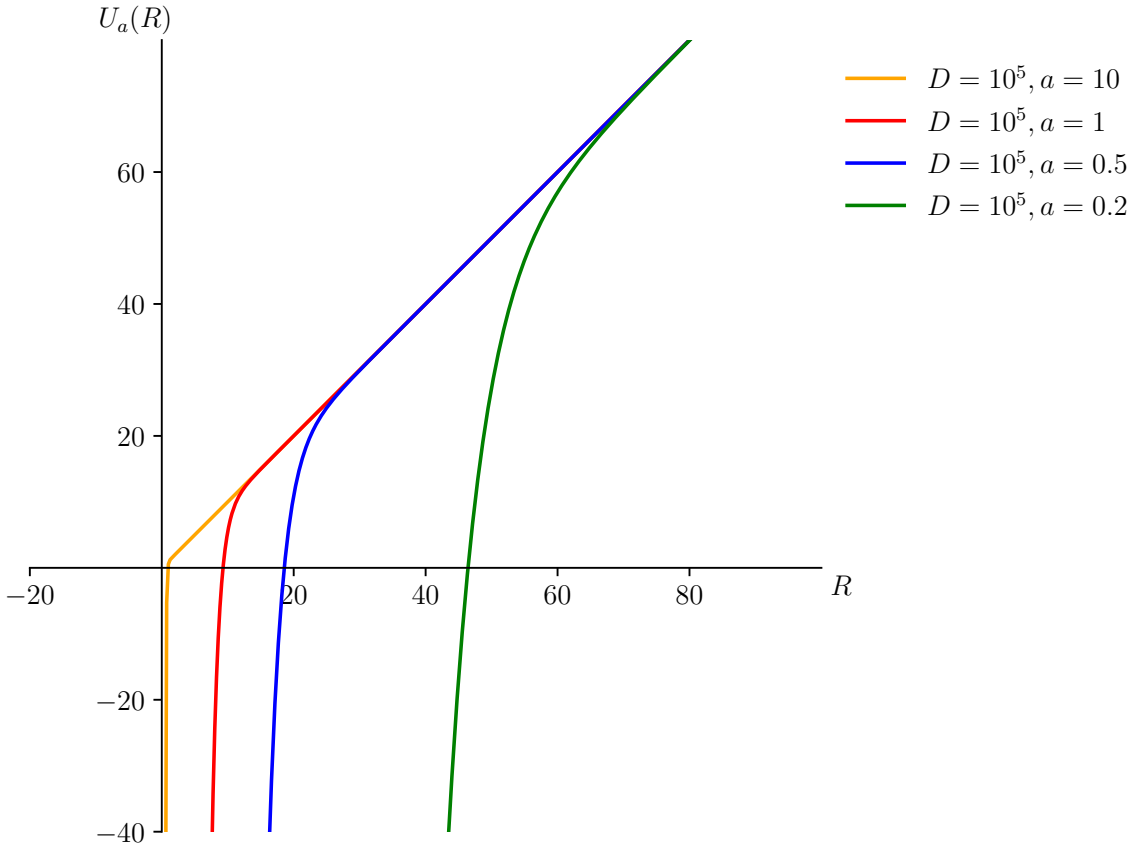


Figure 3.5: $U_a(R)$ utility functions with varying constant value a, which shifts the point, where the risk attitude changes from risk neutral to risk averse, and changes how fast that change occurs.

The parameter $F > 0$ works similar to the parameter D from U_a, and shifts the point, where the risk attitude changes. In Figure 3.6, multiple U_s functions with varying values for F are shown. All functions exhibit the same convex shape for high R and linear growth for low R. But functions with higher F shift the point, where the change from linear growth to convex shape occurs, towards higher R.

Similarly to a, the parameter $b > 0$ shifts the point of change, but also controls, how fast that change occurs. In Figure 3.7, utility functions of family U_s with different values for b are shown. Functions with higher b, shift their point of change towards higher R, and have a smoother transition from linear growth to convex shape, while functions with lower b show a more abrupt change from linear growth to convex shape.

These utility functions are similar to the family U_d defined by [AGA22], shown in Section 2.4.1, but follow the definition of one-switch utility functions, which have been explored by multiple works [Bel88; Li22]. With them, we can encode a multitude of dynamic risk attitudes, which we will later use in our evaluation.

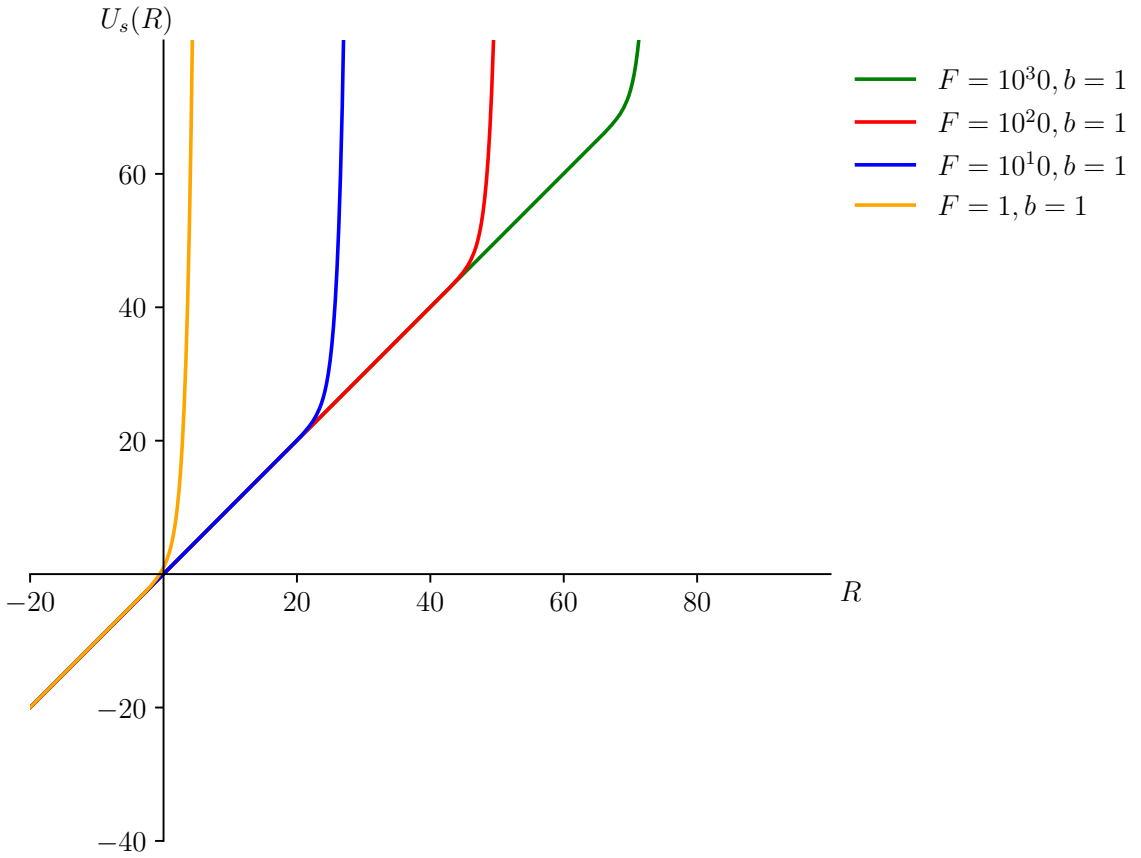


Figure 3.6: $U_s(R)$ utility functions with varying constant value F, which shifts the point, where the risk attitude changes from risk seeking to risk neutral.

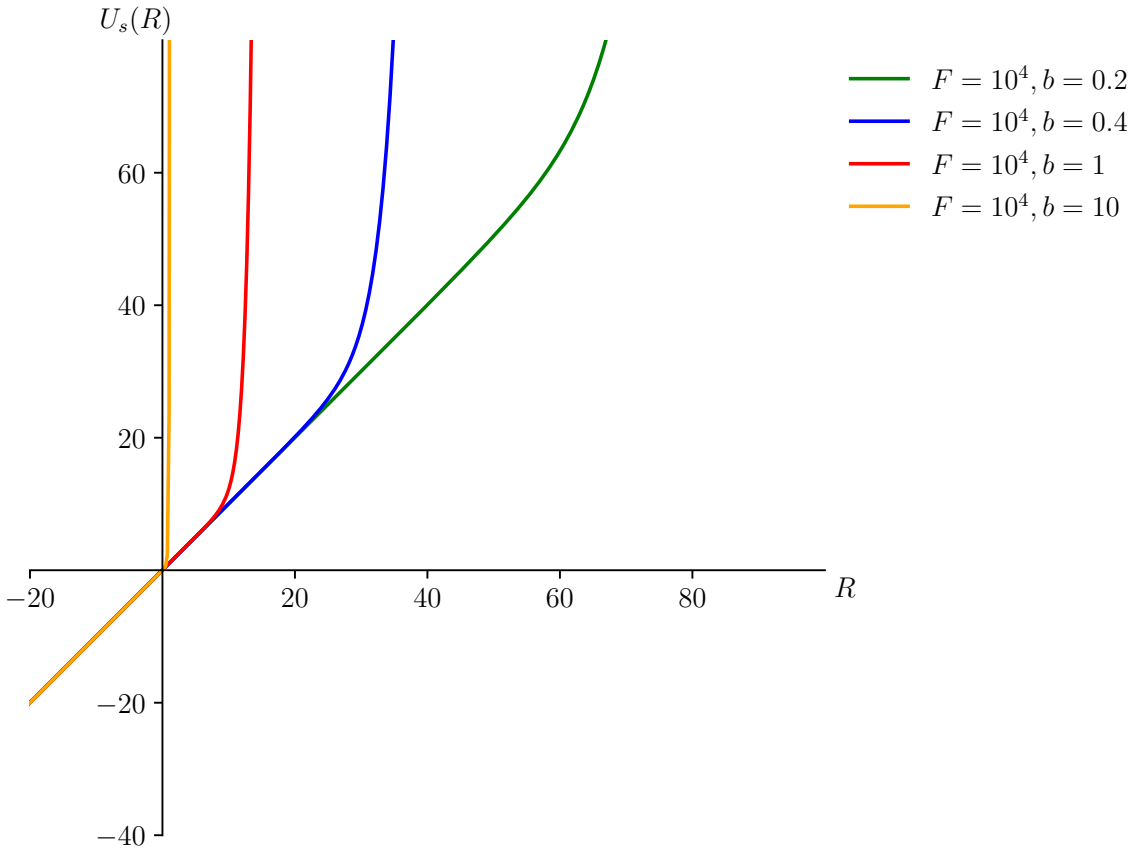


Figure 3.7: $U_s(R)$ utility functions with varying constant value b, which shifts the point, where the risk attitude changes from risk seeking to risk neutral, and changes how fast that change occurs.

4 Implementation

This work adapts the HTN planner JSHOP2 into RARJSHOP2 by introducing modifications that enable replanning for risk-aware planning problems. To provide context, we first present an overview of JSHOP2's core functionality and typical usage. Then we will introduce the changes made to JSHOP2 as part of our risk-aware replanning adaption.

4.1 JSHOP2

Unlike most domain-independent planners that operate by interpreting descriptions for HTN planning domains and planning problems at runtime, JSHOP2 employs a distinctive compilation-based approach. It transforms the descriptions a given HTN planning domain its associated planning problem into a domain-specific planner. This transformation is achieved through compilation of the input descriptions directly into executable Java code. The compilation process takes advantage of JSHOP2's built-in repository of domain-independent templates, such as templates for methods and operators, and the Another Tool for Language Recognition (ANTLR) parser generator to produce highly efficient planning code tailored to the specific domain structure. The advantage of this compilation strategy lies in its capacity to implement static optimizations that would not be possible in an interpreted execution model. A concrete example of such optimization includes the replacement of dynamic data structures with more efficient fixed-size array implementations in cases where the domain description enables such optimizations. Furthermore, the compilation to Java enables rigorous type checking and efficient integration with external code, addressing limitations of the original LISP-based SHOP2 implementation, where such integrations were more flexible but less performant [Ilg06].

This compilation process is illustrated in Figure 4.1 from JSHOP2's documentation. We can see that the description for the planning domain is combined with JSHOP2's templates to compile the domain-specific planner. This domain-specific planner then calculates a plan for the planning problem description.

4.1.1 Usage

The process of solving planning problems using JSHOP2 involves two distinct phases - the initial setup of language processing tools and the problem-solving cycle. The first phase is a one-time setup that generates the lexer and parser, with which JSHOP2 can process the descriptions for planning problems and planning domains. The second phase generates plans for planning problems and must be performed for each individual planning problem.

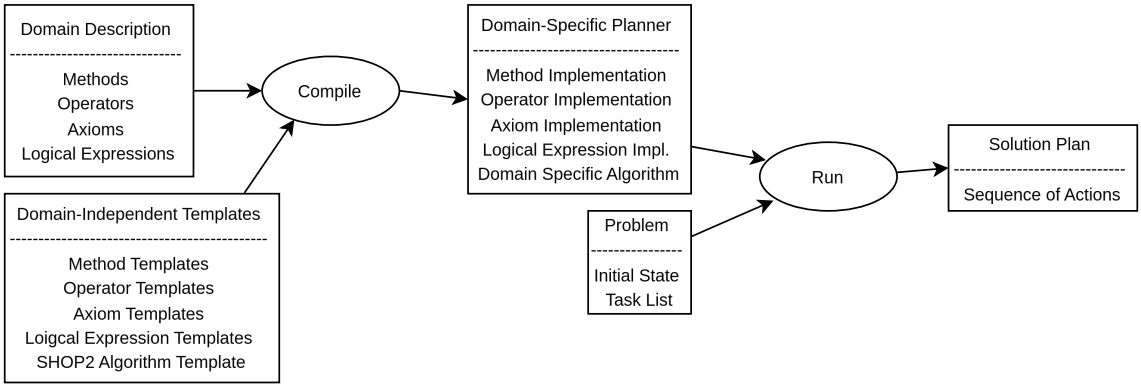


Figure 4.1: JSHOP2 compilation process [Ilg06]

Generating the Lexer and Parser

Before any planning problems can be solved, JSHOP2 requires language processing components to interpret the input format. In this step ANTLR is used to process the grammar file JSHOP2.g. ANTLR is a powerful parser generator that can process language grammars and generate corresponding lexers and parsers [Par14]. From the grammar file ANTLR generates the JSHOP2Lexer, which is used to tokenize input files, and the JSHOP2Parser, which is used to construct abstract syntax trees from the parsed input. These generated components allow JSHOP2 to understand and process subsequent planning domain and planning problem descriptions written in JSHOP2's input syntax. This lexer and parser generation only needs to be performed once during the initial setup of the planning environment, as the resulting components can be reused for all subsequent planning problems.

Solving Planning Problems

The process of solving planning problems in JSHOP2 involves compiling a domain-specific planner from the domain and problem description files. First, the domain description file is processed to generate a corresponding Java class domain.java. This domain class encapsulates all the domain-specific planning logic described in the domain description, including operators, methods, and predicates. Next, the problem description undergoes a similar compilation process, producing a Java class problem.java. The problem class contains the specific problem instance, including the initial state and goal task network. Depending on the options used in this step, the final planner will either return only the first found plan or all valid plans. These generated classes are then compiled into Java byte code, producing a domain-specific planner that can be executed. Finally, this planner is executed. During the execution, the planner first performs necessary preprocessing operations to initialize the planning state. It then invokes the core plan search algorithm, which systematically explores the task decomposition space defined by the domain class while respecting the constraints and initial conditions specified in the problem class. This search process continues until it either finds a valid plan that solves the specified problem or exhausts all possible decomposition paths.

4.1.2 Input Syntax

To submit a planning domain and a planning problem to JSHOP2, they have to be described in input files using the JSHOP2 input syntax. This section describes the basic input syntax for JSHOP2 from [Ilg06]. We will omit syntax for features that are not relevant for this work, such as extending JSHOP2 with Java code in the domain input file. To denote optional parameters or keywords we use square brackets.

- **Comment**
  A comment is anything in a line that follows a semicolon and will be ignored.
- **Symbol**
  Some structures come with their own symbols, which are one of the following:
  - A variable symbol is a symbol whose first character is a question mark, such as *?x*.
  - A primitive task symbol is a symbol that begins with an exclamation point, such as *!drop*.
  - A constant symbol, predicate symbol or compound task symbol is a symbol that starts with a letter or an underline, such as *deliver*.
- **Term**
  A term can be any of the following:
  - A variable symbol.
  - A constant symbol.
  - A number.
  - A list term. List terms are of the form *(*t_1 t_2 $\ldots$ t_n*)* where each t_i is a term.
  - A call term. Call terms are of the form *(call* f t_1 t_2 $\ldots$ t_n*)*, where f is a built-in function such as +, -, *, or /. A call term tells JSHOP2 that it should replace the call term with the result of applying function f on the arguments t_1 t_2 $\ldots$ t_n.
- **Logical Atom**
  A logical atom has the form *(*p t_1 t_2 $\ldots$ t_n*)* where p is a predicate symbol and each t_i is a term.
- **Logical Expression**
  A logical expression is a logical atom or one of the following complex expressions:
  - A conjunction of the form *([and] [*L_1 L_2 $\ldots$ L_n*])* with each L_i being a logical expression. If there are no logical expressions, meaning the expression is *()*, it evaluates to true.
  - A disjunction of the form *(or* L_1 L_2 $\ldots$ L_n*)* where each L_i is a logical expression.
  - A negation of the form *(not L)* where L is a logical expression.
  - A call expression, which has the same form as a call term, but its value interpreted as false if it is an empty list, or true if it is not.
- **Logical Precondition**
  A logical precondition is a logical expression.

- **Axiom**
  An axiom is an expression of the form *(:- a [name$_1$] L_1 [name$_2$] L_2 ... [name$_n$] L_n)*. Where a is a logical atom and all L_i are logical preconditions with optional names $name_i$.
  An axiom is evaluated to true if L_1 is true, or, if L_1 is false but L_2 is true, ..., or if all of L_1, L_2, ... L_{n-1} are false but L_n is true.
- **Task Atom**
  A task atom is a representation of a primitive or compound task of the form *(s t_1 t_2 ... t_n)*, where s is either a primitive or a compound task symbol and the arguments $t_1, t_2, \ldots, t_n$ are terms.
- **Task List**
  A task list is either a task atom or an expression of the form *([:unordered] [$tasklist_1$ $tasklist_2$ ... $tasklist_n$])*, where every $tasklist_i$ is a task list. Notably, a task list can be empty. The optional : $unordered$ keyword defines a task list to be unordered, whereas omitting it defines a task list to be ordered. JSHOP2 must perform the task lists of an ordered task list in the order they are given. For an unordered task list, JSHOP2 may interleave tasks between different task lists.
- **Operator**
  An operator has the form *(:operator h P D A [c])*.
  - The operator's head h is a primitive task atom.
  - The operator's precondition P is a logic precondition. P can contain any variable symbols that can be either in h or not, provided that there can be at most one satisfier for P in the current state at any given point in the planning process.
  - The operator's delete list D and add list A are each either
    * a list of predicates that can only contain variables appearing in h or P,
    * or an expression of the form *(forall V Y Z)*, where V is a list of variables in Y, Y is a logical expression, and Z is a list of logical atoms that only contain variable symbols from h, P, or V.
  - The operator's cost c is an optional term. If c is omitted, its default value is 1.
- **Method**
  A method is a list of the form *(:method h [name$_1$] L_1 T_1 [name$_2$] L_2 T_2 ... [name$_n$] L_n T_n)*
  - The method's head h is a compound task atom.
  - Each $name_i$ is an optional name for the succeeding *(L_i T_i)* pair.
  - Each precondition L_i is a logical precondition.
  - Each tail T_i is a task list.
- **Planning Domain**
  A planning domain has the form *(defdomain domain-name (d_1 d_2 ... d_n))*, where *domain-name* is a symbol and each d_i is either an operator, a method, or an axiom.

- **Planning Problem**
  A planning problem has the form *(defproblem problem-name domain-name ([a_1 a_2 ... a_n]) T)*, where *problem-name* and *domain-name* are symbols, each a_i is a ground logical atom, and T is a task list.
- **Plan**
  A plan is a list *(h_1 h_2 ... h_n)*, where each h_i is a head of a ground instance of an operator o_i. The total cost of a plan is the sum of each o_i's cost.

An example domain file for the delivery robot example is shown in Listing 4.1. The domain begins with the *defdomain* keyword and its name *robot* and then defines each operator, method and axiom.

Each operator consists of its head, precondition, delete list, add list and cost. The operator *!pickup* in line 4 uses one variable *?p*. It represents the robot picking up the package *?p*. The preconditions for an operator define what must be true in the state so that the operator can be executed. Line 5 contains a precondition that specifies, that the *!pickup* operator requires *?p* to be a package and both the robot and the package *?p* have to be located at the same location *?l*. The delete and add lists of an operator define the negative and positive effects this operator has on the state. Each predicate in the delete list will be removed from the state, while each predicate in the add list will be added to the state. The delete list for the *!pickup* operator in line 6 specifies a negative effect *(at ?p ?l)*. This predicate will be removed from the state, meaning that the package *?p* is no longer at the location *?l*. The add list for that operator in line 7 defines a positive effect *(have ?p)*. This predicate will be added to the state, meaning that the robot now has the package *?p*. Finally, the operator's cost is defined, which as seen in line 8, is 0.

Methods consist of a head and multiple possible decompositions, each with an optional name, a precondition and a task list. Line 35 to 43 show the decompositions for the compound task *deliver*. It takes two variables *?p* and *?destination* and represents delivering the package *?p* to a location *?destination*. The method contains two possible decompositions, called *m4* and *m5*. The decomposition *m4* is an edge case, where the package is already at its destination, as can be seen in the precondition in line 37. The resulting task list in line 38 is empty, since no action has to be taken. *m5* defines the standard case and captures the current location of the robot and the package in the variables *?current* and *?pickup* in its precondition. Line 41 and 42 show, that the task *deliver* can be decomposed into traveling to the pickup location, picking up the package, driving to the destination, and dropping the package.

Axioms are inference rules, used to derive additional facts about the state, without explicitly modifying it. They consist of a head and a list of preconditions. For example the axiom *adjacent* from line 45 to 47 defines a new predicate, that allows checking if two locations *?l1* and *?l2* are adjacent. Its precondition in line 46 shows, that *adjacent* evaluates to true, if either *?l1* is next to *?l2*, or *?l2* is next to *?l1*. This shortened form can then be used in preconditions for operators and methods, like in lines 17 and 31.

Problem files describe a planning problem that has to be solved. They consist of the names of the problem and its associated domain, an initial state and a task list. An example problem for the delivery robot domain is shown in Listing 4.2. It specifies *problem* as its name and *robot* as the name of its associated domain in line 1. Lines 4 to 15 define the initial state using a list of predicates. Similar to the problem in Section 2.1, three homes and two packages are defined, with

Listing 4.1 The domain input file for the delivery robot example.

```
(defdomain robot
  (
;-----------------------------------------------operators
    (:operator (!pickup ?p)
      ((package ?p) (location ?l) (robot-at ?l) (at ?p ?l))
      ((at ?p ?l))
      ((have ?p))
      0
    )
    (:operator (!drop ?p)
      ((package ?p) (location ?l) (robot-at ?l) (have ?p))
      ((have ?p))
      ((at ?p ?l))
      0
    )
    (:operator (!drive ?l1 ?l2)
      ((location ?l1) (location ?l2) (robot-at ?l1) (adjacent ?l1 ?l2))
      ((robot-at ?l1))
      ((robot-at ?l2))
      1
    )
;-----------------------------------------------methods
    (:method (travel ?from ?to)
      m1
      ((robot-at ?from) (robot-at ?to))
      ()
      m2
      ((robot-at ?from) (not (robot-at ?to)) (adjacent ?from ?to))
      ((!drive ?from ?to))
      m3
      ((robot-at ?from) (not (robot-at ?to)) (not (adjacent ?from ?to))
       (adjacent ?from ?intermediate))
      ((!drive ?from ?intermediate) (travel ?intermediate ?to))
    )
    (:method (deliver ?p ?destination)
      m4
      ((at ?p ?destination))
      ()
      m5
      ((not (at ?p ?destination)) (at ?p ?pickup) (robot-at ?current))
      ((travel ?current ?pickup) (!pickup ?p) (travel ?pickup ?destination)
       (!drop ?p))
    )
;-----------------------------------------------axioms
    (:- (adjacent ?l1 ?l2)
      (or (next ?l1 ?l2) (next ?l2 ?l1))
    )
  )
)
```

Listing 4.2 An example problem input file for a delivery robot.

```
1  (defproblem problem robot
2
3  ;---------------------------------------initial state
4      (
5          (location Home1)
6          (location Home2)
7          (location Home3)
8          (next Home1 Home2)
9          (next Home2 Home3)
10         (package Package1)
11         (package Package2)
12         (at Package1 Home1)
13         (at Package2 Home2)
14         (robotAt Home3)
15     )
16
17 ;---------------------------------------initial task list
18     (
19         (:unordered
20           (deliver Package1 Home2)
21           (deliver Package2 Home3)
22         )
23     )
24 )
```

Package1 being at *Home1* and *Package2* being at *Home2*. The connections between the homes and the location of the robot are defined as well. Lines 18 to 23 define the initial task list, that should be achieved. The robot has to deliver *Package1* to *Home2* and *Package2* to *Home3* in any order.

4.2 Modifications

This section describes the changes made to the source code of JSHOP2, to enable risk-aware replanning in RARJSHOP2

4.2.1 Input Syntax

To allow risk-aware planning, the operators in the domain must be cost-variable. Thus, we change the definition of an operator in the domain description file.

An operator has the form *(:operator h P D A c p)*.

- The operator's head *h* is a primitive task atom.
- The operator's precondition *P* is a logic precondition. *P* can contain any variable symbols that can be either in h or not, provided that there can be at most one satisfier for P in the current state at any given point in the planning process.

Listing 4.3 An example cost-sensitive operator.

```
(:operator (!drive ?l1 ?l2)
      ((location ?l1) (location ?l2) (robot-at ?l1) (adjacent ?l1 ?l2))
      ((robot-at ?l1))
      ((robot-at ?l2))
      (1 1.5 2)
      (0.4 0.3 0.3)
)
```

- The operator's delete list D and add list A are each either
  - a list of predicates that can only contain variables appearing in h or P,
  - or an expression of the form *(forall V Y Z)*, where V is a list of variables in Y, Y is a logical expression, and Z is a list of logical atoms that only contain variable symbols from h, P, or V.
- The operator's costs c is a list of numbers.
- The operator's probabilities p is a list of positive numbers. The sum of these probabilities must be 1 and the length of p must be equal to the length of c.

Instead of a single optional cost, an operator now has a list of possible costs. Additionally, each operator contains a list of probabilities. The costs and probabilities are directly associated with each other such that the first cost corresponds to the first probability, the second cost to the second probability, and this pairing continues for all entries. The probabilities and cost lists must always have the same length, such that every cost has a corresponding probability value.

An example for a cost-variable operator can be seen in Listing 4.3. It is an adaption of the *!drive* operator from line 16 to 21 in Listing 4.1. Instead of a single cost, the operator now has the possible costs 1 with probability 0.4, 1.5 with probability 0.3, and 2 with probability 0.3.

4.2.2 Search Algorithm

We introduced several modifications to the JSHOP2 search function, analogous to the changes made to the SHOP algorithm illustrated in Algorithm 3.2. Unlike the original implementation, which returns either the first valid plan, a fixed number of plans, or all possible plans, that solve a given planning problem, RARJSHOP2 explores all possible plans, evaluates each plan based on a utility function, and returns the one with the highest utility.

In addition to finding the best plan, the algorithm now records the steps taken to find this plan. Specifically, it stores a list of indices representing the choices made during search, such as the selected operators, methods, and variable bindings. Two such index lists are maintained: one for the plan currently being explored, and another for the best plan found so far. If a new plan is found, that is better than the currently best found plan, the index list for the best plan is overwritten with a copy of the index list from the new plan.

4.2.3 Replanning

To support replanning, we implemented a function analogous to Algorithm 3.3. This function re-executes the search algorithm, but instead of exploring freely, it deterministically follows the sequence of choices for operators, methods, and variable bindings, stored in the index list of the best plan. The replanning process continues until the last operator executed during the interrupted plan is added to the currently explored plan. Once this point is reached, control is passed to the modified search algorithm. From there, the search continues, focusing on exploring possible plans from that point onward. The function will return the plan, that starts with the sequence of already executed operators and has the highest expected utility.

4.2.4 Calculating Expected Utility

We added a function, that calculates the expected utility of a plan. This calculation is analogous to Definition 3.2.3, and traverses every possible combination for costs of every operator in the plan. For every combination, the utility is calculated, by subtracting the sum of the costs from the initial resources and using a chosen utility function on that result. The utilities for every combination are weighted by their probabilities and summed up. This sum will then be returned as the expected utility of the plan.

If some operators have already been executed, a list of indices for the respective costs of these operators can be passed to the function. Then, instead of traversing every possible combination of costs, only those combinations are considered, that contain the incurred costs for the already executed operators.

We implemented the utility functions U_a and U_s, that have been introduced in Section 3.5, as well as a risk neutral utility function.

4.2.5 Main Function

The main function of RARJSHOP2 works analogous to the functionality described in Algorithm 3.1. First, the user has to provide his choice of utility functions, and its parameters, as well as the amount of initial resources and the replan threshold δ. Then, the search function is used to find the plan with the highest utility. This plan is then shown to the user. After which, execution of that plan begins. The user is prompted to input the costs for every operator. The expected utility of the plan is recalculated after every input cost. If the ratio between this new expected utility and the original expected utility is off by δ or more, the replanning function is called. The resulting plan is again output and the user is prompted for the next operator cost. This process continues, until the whole plan has been executed.

Additionally, whenever a plan is searched for, whether using the standard search function or the replanning function, the computation time for the search is recorded and output. For debugging purposes, the user can also manually trigger the replanning function at any time.

4.2.6 Usage

To use RARJSHOP2, first, the lexer and parser need to be generated. Then the domain and problem files need to be compiled. The resulting Java byte code can be executed, at which point the interactive (re-)planning process begins. Before any of these steps, the user needs to use following command, to correctly set the *CLASSPATH* variable:

```
$ source path.sh
```

Generating the Lexer and Parser

Before any replanning problem can be solved, the lexer and parser need to be generated with following command:

```
$ make c
```

This needs to be only done once, as the lexer and parser can be reused for every replanning problem.

Solving a Replanning Problem

To solve a specific replanning problem, the domain must be compiled using the following command:

```
$ java JSHOP2.InternalDomain DOMAIN_FILE
```

Afterward, the problem file should be converted into Java code by executing:

```
$ java JSHOP2.InternalDomain -r PROBLEM_FILE
```

This command produces a Java file named according to the problem name defined within the problem file. The resulting Java file must then be compiled using:

```
$ javac PROBLEM_NAME.java
```

Once compiled, the problem can be executed with:

```
$ java PROBLEM_NAME
```

During execution, the user is prompted to select a utility function and provide its parameters, as well as the initial resources and the replan threshold. In the following example, the user's inputs are highlighted in red. The utility function U_s is chosen, with parameters $b = 3$, $F = 7$, $R_0 = 100$, and $\delta = 0.1$:

```
Use utility function Ua(R) [1], Us(R) [2], or risk neutral [3]? 2
b: 3
F: 7
Initial Resources: 100
Replan Threshold: 0.1
```

Once the inputs are provided, a search for the best plan is carried out. The resulting plan's utility, the sequence of operators, and the total computing time are then displayed, as illustrated below:

```
--------------------
Plan utility: 12.234

(!drive Home2)
(!drive Home1)
(!pickup Package1)
(!drive Home2)
(!drop Package1)
--------------------

Time Used = 0.012
```

Following this, the user is asked to specify the costs incurred for each operator in the plan. Each operator is presented with a list of potential costs, from which one must be selected. In the example below, the user's selections are again shown in red. A cost of 2.0 is chosen for the first operator, and a cost of 1.5 for the second:

```
Please input effects for operators. Enter 'r' to replan

Operator 1: (!drive Home2)
[0]: cost = 1.0   [1]: cost = 1.5   [2]: cost = 2.0
2
Operator 2: (!drive Home1)
[0]: cost = 1.0   [1]: cost = 1.5   [2]: cost = 2.0
1
```

If replanning is triggered, a new plan is generated and displayed. Cost selection then resumes with the next operator in the updated plan. This process concludes once costs have been specified for all operators in the final plan.

5 Evaluation

This chapter presents the evaluation of the RARJSHOP2 implementation. We begin by describing the planning domain and the problem instances used in the experiments. Next, we outline the benchmarking methodology, detailing how the evaluation was conducted. We then present the benchmark results, examining each configuration to analyze the influence of various parameters on the planning process. This is followed by an overview of the implementation's performance. Finally, we summarize the key findings of the evaluation.

5.1 Planning Domain

For our evaluation, we define the problem domain of an electric vehicle. The vehicle can drive between locations, that are connected via a one-directional road. The cost metric in this scenario is energy consumption, with the initial battery charge serving as the initial resources.

5.1.1 Risk Factors

Electric vehicle energy efficiency is significantly affected by various risk factors, including environmental conditions and driving patterns. Low temperatures substantially increase energy demand due to the additional power required for cabin heating and battery temperature regulation. Similarly, high temperatures negatively impact efficiency as the climate control system works harder to maintain comfortable cabin conditions [ASS22; PKL+24]. Driving behavior represents another critical factor - the frequent acceleration and deceleration characteristic of urban driving leads to notably higher energy consumption compared to maintaining steady speeds [Gal17; MDSN22; PKL+24]. While highway driving generally offers better efficiency through more consistent speeds, the aerodynamic drag and increased power requirements at higher velocities can still result in elevated energy usage [Gal17; MDSN22; PKL+24].

Real-world consumption data reveals that minimum energy consumption typically measures approximately half of maximum consumption values. Such as consumption of 93-189 Wh/km for the Tesla Model 3 and 109-217 Wh/km for the BMW i4 eDrive40 [EV 25].

Because the probability distributions of the specific energy consumption in different scenarios requires empirical measurements from such scenarios, we assume probabilities in our domain.

5.1.2 Operators

To capture these effects in our model, we implement three distinct driving operators with different cost distributions, detailed in Listing 5.1. These operators serve as abstract representations that collectively account for all risk factors. All operators share the same basic structure, requiring, that the vehicle is currently at location *?from* and that a road connection exists between *?from* and *?to* (lines 4, 12, and 20). When executed, each operator removes the predicate *(at ?from)* from the state and adds *(at ?to)*, moving the vehicle to the destination location (lines 5–6, 13–14, and 21–22).

The operators differ only in their road type requirements. The operator *!drive1* is exclusively applicable to *road1* connections (line 4), while *!drive2* and *!drive3* require *road2* (line 12) and *road3* (line 20) connections, respectively. This association between operators and road types allows us to model different energy consumption patterns for different roads, while maintaining identical state transition semantics across all driving actions.

Each operator features cost values ranging from 10 to 20 units while maintaining comparable average costs across all three variants. Operator *!drive1* offers the most favorable average cost of 16.0, with a 50% probability of 14 units, 20% probability of 15 units, and 30% probability of 20 units (line 7). This distribution provides the best average performance while allowing for occasional high-consumption scenarios. Operator *!drive2* (line 15) demonstrates more consistent behavior with an average cost of 16.5 units. Its distribution shows a 10% probability of 15 units, 30% probability of 16 units, and 60% probability of 17 units, creating a stable pattern that avoids extreme outcomes while maintaining slightly higher average costs than *!drive1*. The most variable distribution belongs to operator *!drive3* (line 23), which maintains the same 16.5 unit average as *!drive2* but with significantly different probabilities: 30% chance of achieving the optimal 10 units, 10% probability of 15 units, and 60% probability of the maximum 20 units.

Additionally, we define an operator *!arriveOp*, seen in Listing 5.2, that signifies the car arriving at the final destination. It requires the car being at the destination *?loc* (line 3), but has no effects (lines 4-5) and zero cost (line 6).

5.1.3 Methods

The goal in our domain is, to plan a route from a start location to a destination via intermediate locations. We implement this through a compound task *arrive*, which represents the complete process of driving to and arriving at the destination. As shown in Listing 5.3, the method decomposes into two sequential steps: first, a compound task *drive*, which navigates to the target location, followed by an *!arriveOp* operator that formally marks the successful completion of the journey.

The compound task *drive* is used to represent the navigation from the current location to a destination *?to*. It has three decompositions, shown in Listing 5.4. The first decomposition *m1* (lines 5–7) handles the case, where the car already is at the location, as seen in the precondition in line 6. Since no action is required to get the car to the destination, the resulting task list is empty (line 7). The decomposition *m2* (lines 10–12) handles the case, where the car is one step from the destination, i.e. the location, where the car is currently has a road, connecting it to the destination location. Its precondition in line 11 captures the current location of the car with *(at ?from)* and then checks, if there is a road connection from the current location *?from* to the destination *?to*, using the axiom *connected*. This axiom, presented in Listing 5.5, evaluates to true, if there is at least one

Listing 5.1 *drive* Operators

```
;----------------------------------------road1
(:operator
    (!drive1 ?to)                   ; head
    ((at ?from) (road1 ?from ?to))  ; precondition
    ((at ?from))                    ; delete list
    ((at ?to))                      ; add list
    (14 15 20) (0.5 0.2 0.3)        ; cost --- expected 16.0
)
;----------------------------------------road2
(:operator
    (!drive2 ?to) ; head
    ((at ?from) (road2 ?from ?to))  ; precondition
    ((at ?from))                    ; delete list
    ((at ?to))                      ; add list
    (15 16 17) (0.1 0.3 0.6)        ; cost --- expected 16.5
)
;----------------------------------------road3
(:operator
    (!drive3 ?to)                   ; head
    ((at ?from) (road3 ?from ?to))  ; precondition
    ((at ?from))                    ; delete list
    ((at ?to))                      ; add list
    (10 15 20) (0.3 0.1 0.6)        ; cost --- expected 16.5
)
```

Listing 5.2 *arriveOp* Operator

```
(:operator
    (!arriveOp ?loc) ; head
    ((at ?loc))      ; precondition
    ()               ; delete list
    ()               ; add list
    (0) (1)          ; cost
)
```

Listing 5.3 *arrive* Method

```
(:method
    (arrive ?loc)                    ; head
    ()                               ; precondition
    ((drive ?loc) (!arriveOp ?loc))  ; task list
)
```

Listing 5.4 *drive* Methods

```
1  (:method
2      (drive ?to) ; head
3
4      ;---------------------------------------- method 1: already at destination
5      m1
6          ((at ?to))                            ; precondition
7          ()                                    ; task list
8
9      ;---------------------------------------- method 2: one step from destination
10     m2
11         ((at ?from) (connected ?from ?to))    ; precondition
12         ((driveStep ?to))                     ; task list
13
14     ;---------------------------------------- method 3: recursively go towards destination
15     m3
16         (                                     ; precondition
17             (at ?from)
18             (not (connected ?from ?to))
19             (connected ?from ?intermediary)
20         )
21         ((driveStep ?intermediary) (drive ?to)) ; task list
22 )
```

Listing 5.5 *connected* Axiom

```
1  (:- (connected ?l1 ?l2)
2      ((or (road1 ?l1 ?l2) (road2 ?l1 ?l2) (road3 ?l1 ?l2)))
3  )
```

of the predicates *road1*, *road2*, or *road3* connecting two locations. Continuing in Listing 5.4, the method *m2* decomposes the *drive* task into a single compound task *driveStep*, which handles driving between two connected locations. The method *m3* (lines 15–21) handles any case, where there is no direct connection between the current location and the destination. In its precondition, it captures the current location of the car *?from* (line 17), checks, that there is no direct connection to the destination (line 18), and captures a location *?intermediary* to which a connection from *?from* exists (line 19). The resulting task list consist of a compound task *driveStep*, which moves the car to the location *?intermediary*, and a compound task *drive*, which continues toward the destination from the new location (line 21). The recursion works by progressively reducing the problem size - each recursive call to *drive* operates on a strictly shorter path as the vehicle moves closer to the destination through intermediary locations.

The *driveStep* compound task is used by decompositions from *m2* and *m3*, Listing 5.4 lines 12 and 21, and represents the car using any road to drive from the current location the next location. A method to decompose this task exists for every road type, which can be seen in Listing 5.6. The three methods have similar preconditions and task list, but use predicates and operators specific to their road type. The method for *road1* can be seen in lines 1–5, the method for *road2* in lines 7–11,

Listing 5.6 *driveStep* Methods

```
1  (:method
2      (driveStep ?to)                ; head
3      ((at ?from) (road1 ?from ?to)) ; precondition
4      ((!drive1 ?to))                ; task list
5  )
6
7  (:method
8      (driveStep ?to)                ; head
9      ((at ?from) (road2 ?from ?to)) ; precondition
10     ((!drive2 ?to))                ; task list
11 )
12
13 (:method
14     (driveStep ?to)                ; head
15     ((at ?from) (road3 ?from ?to)) ; precondition
16     ((!drive3 ?to))                ; task list
17 )
```

and the one for *road3* can be seen in lines 13–17. Each method's precondition checks, if there is a road of its respective type from the current location to the next location (lines 3, 9, and 15). They all decompose into the respective operator *drive1*, *drive2*, or *drive3* (lines 4, 10, and 16).

5.2 Planning Problems

For our evaluation, we generate problems with a regular structure of different sizes. A problem of size 3 can be seen in Listing 5.7.

The initial state of a problem of size N contains a start location, a finish location and N layers of intermediary locations. The layers consist of three locations each, which are named lXa, lXb, and lXc for the layer X. The start location connects to locations $l1a$, $l1b$, and $l1c$ with *road1*, *road2*, and *road3*, respectively. Each location connects to every location of the next layer with a specific road. Every lXa location connects to every lYZ with *road1*, every lXb to every lYZ with *road2* and every lXc to every lYZ with *road3*, where Y is the layer after X, and $Z \in a, b, c$. For example, in Listing 5.7 lines 20–23, we can see, that $l2b$ connects to $l3a$, $l3b$, and $l3c$ with *road2*. The locations of the final layer connect to the *finish* location with their respective roads (lines 26–28). Finally, the car is located on the *start* location (line 30). The initial task list of a problem consist of a single *(arrive finish)* task (line 35), meaning, the electric vehicle has to drive from the *start* to the *finish* location. Any plan, that is a solution to such a problem of length N, will have $N + 2$ operators. One drive operator to go from the *start* location to a location of first level, $N - 1$ drive operators to go through all levels, one drive operator to go from the last level to the *finish* location, and one final *arriveOp* operator.

This problem structure allows the planner to choose a road of different type at every point, enabling us to evaluate the risk-aware replanning features of RARJSHOP2.

Listing 5.7 A Problem of Size 3

```
(defproblem problem ev

;---------------------------------------initial state
    (
        (road1 start l1a)
        (road2 start l1b)
        (road3 start l1c)
        (road1 l1a l2a)
        (road1 l1a l2b)
        (road1 l1a l2c)
        (road2 l1b l2a)
        (road2 l1b l2b)
        (road2 l1b l2c)
        (road3 l1c l2a)
        (road3 l1c l2b)
        (road3 l1c l2c)
        (road1 l2a l3a)
        (road1 l2a l3b)
        (road1 l2a l3c)
        (road2 l2b l3a)
        (road2 l2b l3b)
        (road2 l2b l3c)
        (road3 l2c l3a)
        (road3 l2c l3b)
        (road3 l2c l3c)
        (road1 l3a finish)
        (road2 l3b finish)
        (road3 l3c finish)

        (at start)
    )

;---------------------------------------initial task list
    (
        (arrive finish)
    )
)
```

5.3 Setup

For our evaluation, we developed a benchmarking tool, that automates the process of inputting operator costs, eliminating the need for manual cost entry. The tool selects costs according to one of three predefined strategies: *best*, *middle*, or *worst*. The *best* strategy chooses the lowest available cost for each operator, the *worst* selects the highest, and the *middle* strategy picks the cost closest to the average. During execution, the tool records utility values for partially executed plans at each step, the points where replanning is triggered, the frequency of each operator in the plans, and the

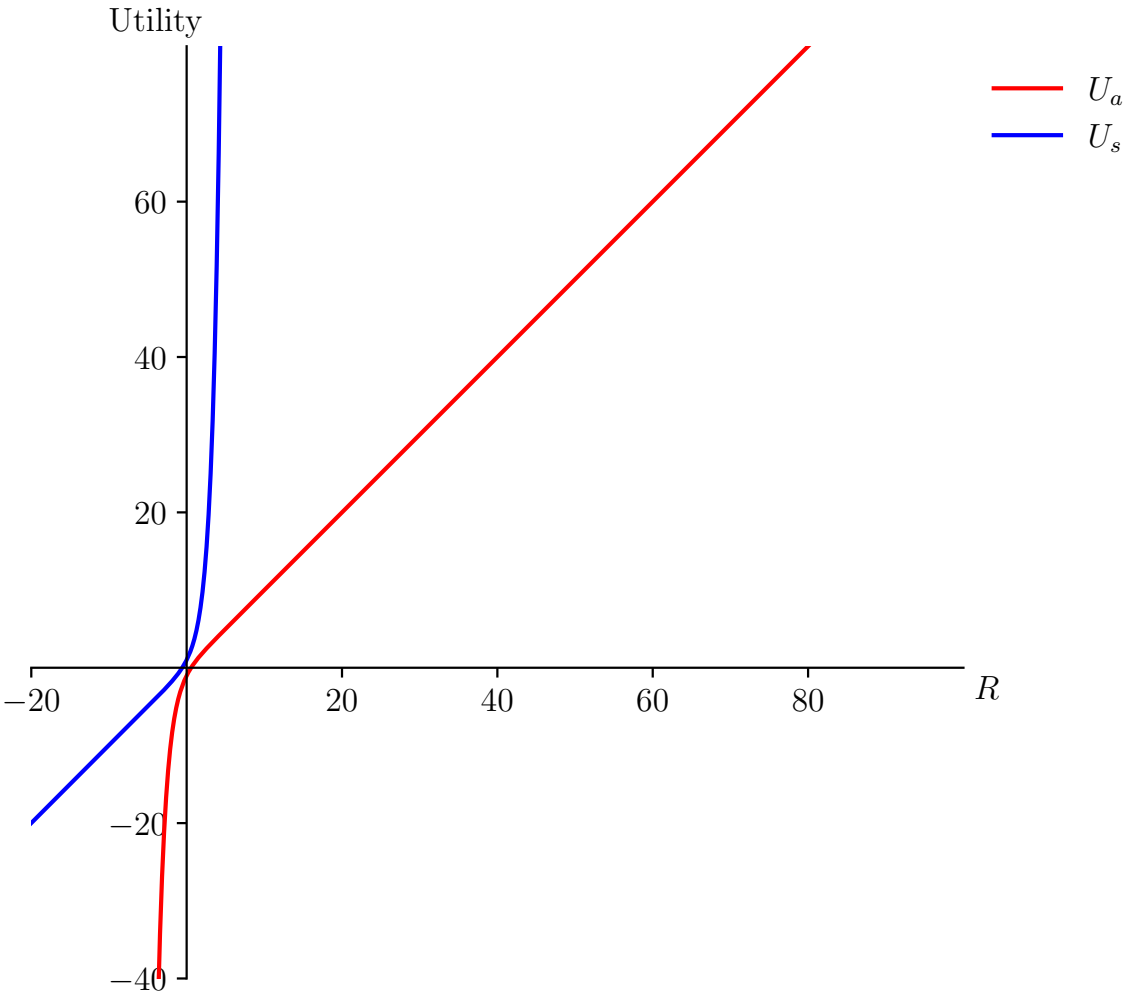


Figure 5.1: U_a and U_s with the base parameters for the benchmarks.

computing times for both initial plans and replans. This setup simplifies the benchmarking process, allowing us to consistently and easily evaluate the impact of different cost selection strategies on the replanning process.

We conduct benchmarks on problem instances of sizes 8, 9, and 10. Each problem is evaluated using both utility functions U_a and U_s. For both utility functions, a set of base parameters is defined. The base values are $R_0 = 16.5 \cdot (N + 2)$, where N is the problem size, and $\delta = 0.05$, with U_a using $a = 1$ and $D = 1$, and U_s using $b = 1$ and $F = 1$. The initial resource value is chosen as $16.5 \cdot (N + 2)$ because, as said before, a problem of size N yields a plan of length $N + 2$, and the *drive2* and *drive3* operators have an average cost of 16.5.

Under these base parameters, both U_a and U_s change their risk attitudes when the remaining resources R approach zero, as can be seen in Figure 5.1. We aim to observe this switching behavior and how it is influenced by different cost selection strategies and parameter settings.

To assess the effect of individual parameters, we vary one parameter at a time while keeping all others fixed at their base values. The variations include a 10% increase and decrease in the initial resources R_0, a higher setting for a or b (set to 10), a larger value for D or F (set to 10^5), and a

higher and lower replanning threshold δ (set to 0.2 and 0.01, respectively). Each of these parameter configurations is evaluated using all three cost selection strategies: *best*, *middle*, and *worst*. This allows for a comprehensive comparison of the resulting plan utilities, replanning behavior, and the operators chosen during execution. A complete list of all benchmark configurations can be seen in Table 5.1.

Table 5.1: Benchmark configurations for all problem sizes. Every configuration is run, using both U_a and U_s for all three cost selection strategies *best*, *middle*, and *worst*.

Problem Size	a **or** b	D **or** F	R_0	δ	**Description**
8	1	1	165.0	0.05	Base configuration
	1	1	181.5	0.05	10% more resources
	1	1	148.5	0.05	10% fewer resources
	10.0	1	165.0	0.05	Higher a or b
	1	10^5	165.0	0.05	Higher D or F
	1	1	165.0	0.01	Lower δ
	1	1	165.0	0.2	Higher δ
9	1	1	181.5	0.05	Base configuration
	1	1	199.7	0.05	10% more resources
	1	1	163.4	0.05	10% fewer resources
	10.0	1	181.5	0.05	Higher a or b
	1	10^5	181.5	0.05	Higher D or F
	1	1	181.5	0.01	Lower δ
	1	1	181.5	0.2	Higher δ
10	1	1	198.0	0.05	Base configuration
	1	1	217.8	0.05	10% more resources
	1	1	178.2	0.05	10% fewer resources
	10.0	1	198.0	0.05	Higher a or b
	1	10^5	198.0	0.05	Higher D or F
	1	1	198.0	0.01	Lower δ
	1	1	198.0	0.2	Higher δ

The benchmarks were conducted on a system with a Ryzen 5 2600 processor, running the latest version of Arch Linux as of April 24, 2025, with 1 GB of memory allocated to the Java Virtual Machine.

5.4 Results

In this section, we present the results of our benchmarks. We begin with an overview of the measurements obtained using the base parameter configuration. This includes a comparison of expected utility values, replanning frequencies, and operator usage across the three cost selection strategies. Following this, we examine the impact of varying parameters. For each variation, we

analyze, how the changes affect the planning and replanning behavior, as well as the frequency and types of operators used during execution. Finally, we analyze the computing times required for initial planning and replanning across all configurations.

5.4.1 Base Parameters

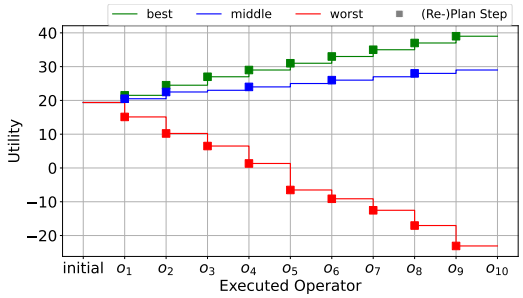
We first present the results for the base configuration. In Figure 5.2 and Figure 5.3 we can see the change in expected utility for U_a and U_s, respectively. These figures show, the expected utility for different cost selection strategies for the initial plan, and after execution of each operator, as the utility for the partially executed plan is recalculated at these points. Additionally, the steps, at which a replan was initiated, are shown with square markers.

For U_a, we observe that the utility increases when the best costs are incurred, but the growth is more moderate for middle costs and declines when worst costs are selected across all problem sizes. This behavior is consistent with the fact that U_a starts as risk neutral, where its utility increases steadily as the cost decreases, but without a significant preference for either low or high costs.

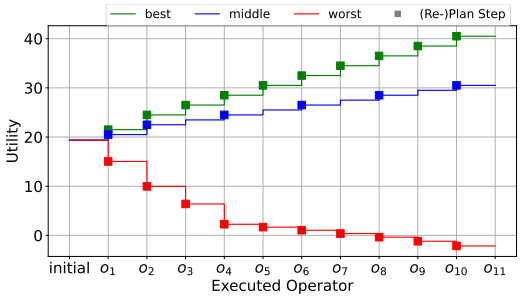
In contrast, for U_s, the utility shows a much larger increase when best costs are incurred, and it stays relatively constant when middle or worst costs are selected, across all problem sizes. This aligns with the fact that U_s starts as risk seeking, placing a higher emphasis on the possibility of low costs, which leads to significant gains when the best costs are chosen but little change when the costs are higher.

The algorithm initiates a replan at almost every step for both utility functions, when using the cost selection strategies *best* and *worst*. This is because both of these strategies involve selecting more extreme costs, resulting in larger changes to the expected utility of the partially executed plan, which often triggers replans.

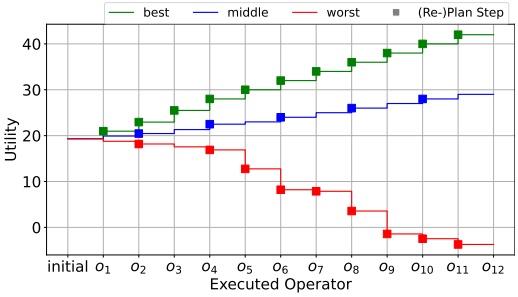
In contrast, for the *middle* strategy, the number of replans is noticeably lower. This is expected, as the *middle* strategy selects the cost closest to the average cost. Consequently, the expected utility for the partially executed plan does not change as drastically, and the ratio of the expected utility between the calculated plan and the current partially executed plan remains closer to 1.



(a) Problem size 8



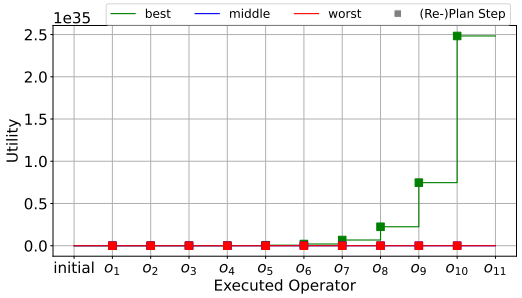
(b) Problem size 9



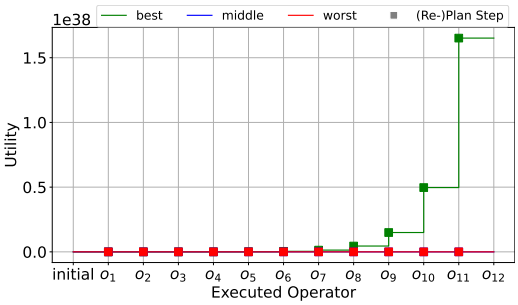
(c) Problem size 10

Figure 5.2: Expected utility for the initial plan and after executing each operator, for problem sizes 8, 9, and 10, using utility function U_a with base parameters.

(a) Problem size 8



(b) Problem size 9



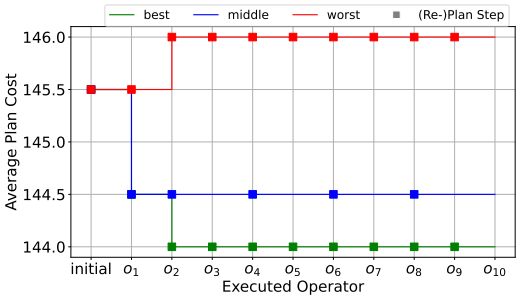
(c) Problem size 10

Figure 5.3: Expected utility for the initial plan and after executing each operator, for problem sizes 8, 9, and 10, using utility function U_s with base parameters.

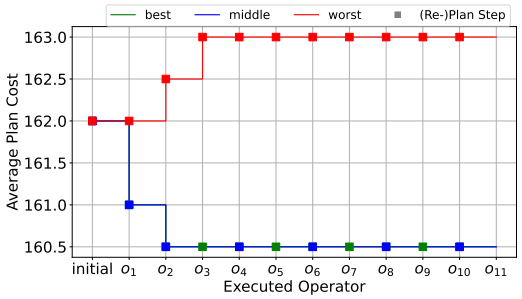
Utility Function U_a

In Figure 5.4, the average cost of each computed plan is shown for problem sizes 8, 9, and 10, with the utility function U_a. All problem sizes exhibit the same behavior. For the *worst* strategy, the plan gets altered, and its average cost increases. For the *best* and *middle* strategies, the plan gets altered, and its average cost decreases.

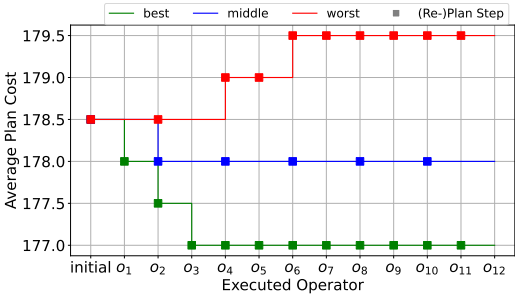
Interestingly, for problem size 9, the average costs for the *middle* and *best* strategies change in the same way. However, for problem sizes 8 and 10, the average cost for the *best* strategy is lower than for the *middle* strategy.



(a) Problem size 8



(b) Problem size 9



(c) Problem size 10

Figure 5.4: Average costs for the initial plan and the plans after each replan step, for problem sizes 8, 9, and 10, using utility function U_a with base parameters.

Figure 5.5 shows why the change in average costs occurs. It shows the number of each type of operator in the computed plans for problem size 10.

All three cost selection strategies start with 4 *drive2* operators, 5 *drive1* operators, and 1 *arriveOp* operator. This makes sense because the difference between the strategies lies in the incurred costs, which have no impact on the initial plan. Additionally, no *drive3* operators are selected, which is expected since *drive3* has a high risk of high costs, and U_a does not exhibit a risk seeking attitude at any point.

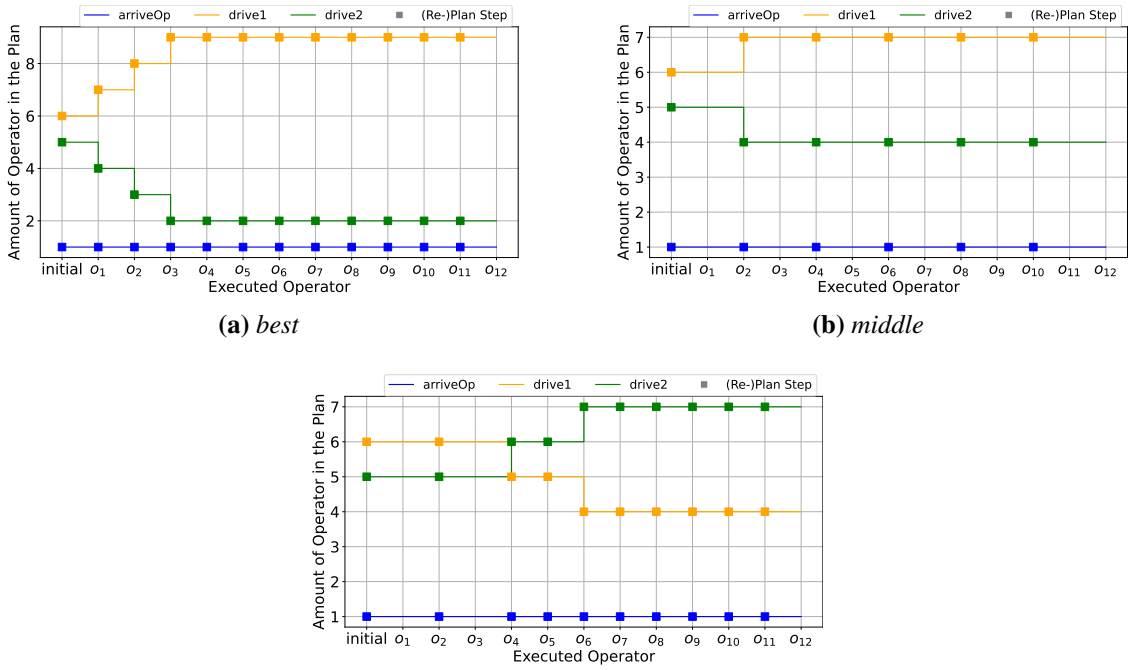
The *arriveOp* operator is always included once in the plan, as the plan must always end with *arriveOp*, and there is no alternative to it like there is for the three different drive operators.

For the *best* and *middle* strategies, fewer *drive2* operators and more *drive1* operators are selected. This makes sense because U_a becomes more risk averse as the remaining resources decrease. The *best* and *middle* costs result in higher remaining resources than expected, which pushes U_a closer to

a risk neutral attitude. The *best* strategy chooses more *drive1* operators than the *middle* strategy, as it results in more remaining resources and thus pushes U_a further towards risk neutrality. Since *drive1* has a lower average cost, it is preferred by a risk neutral attitude.

For the *worst* cost selection strategy, over time, more *drive2* operators are chosen instead of *drive1*. This aligns with expectations, as *worst* costs imply fewer remaining resources than expected, which pushes U_a towards a more risk averse behavior. The operator *drive2* has less risk for very high costs, compared to *drive1*, making it the preferred choice under a risk averse attitude.

The difference in average costs for *drive1* and *drive2* explain the change in average costs in Figure 5.4. Since the *best* and *middle* strategies select more *drive1* operators, the average cost decreases. Conversely, since the *worst* strategy selects more *drive2* operators, the average cost increases.



(a) *best*

(b) *middle*

(c) *worst*

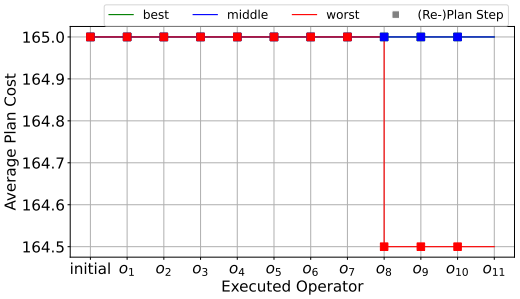
Figure 5.5: Operator usage for different cost selection strategies for problem size 10, using utility function U_a with base parameters.

We are not certain, why the effects for the cost strategies *middle* and *best* result in the exact same behavior for problem size 9, when this does not occur for problem sizes 8 and 10. It is possible, that this is just an effect of this specific combination of problem size and the way the base parameters were calculated.

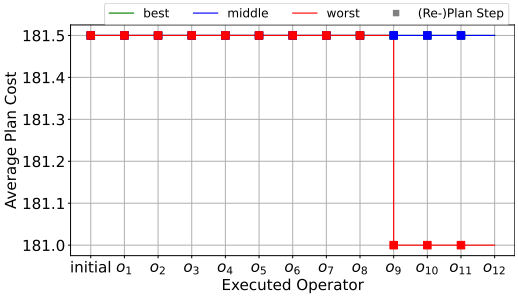
Utility Function U_s

Figure 5.6 shows the average costs for the three problem sizes when the utility function U_s is used. For all problem sizes, the *middle* and *best* strategies show no change in the average plan costs. For problem size 8, the *worst* strategy also shows no change. However, for problem sizes 9 and 10, the average cost only drops once, near the end of the execution.

(a) Problem size 8



(b) Problem size 9



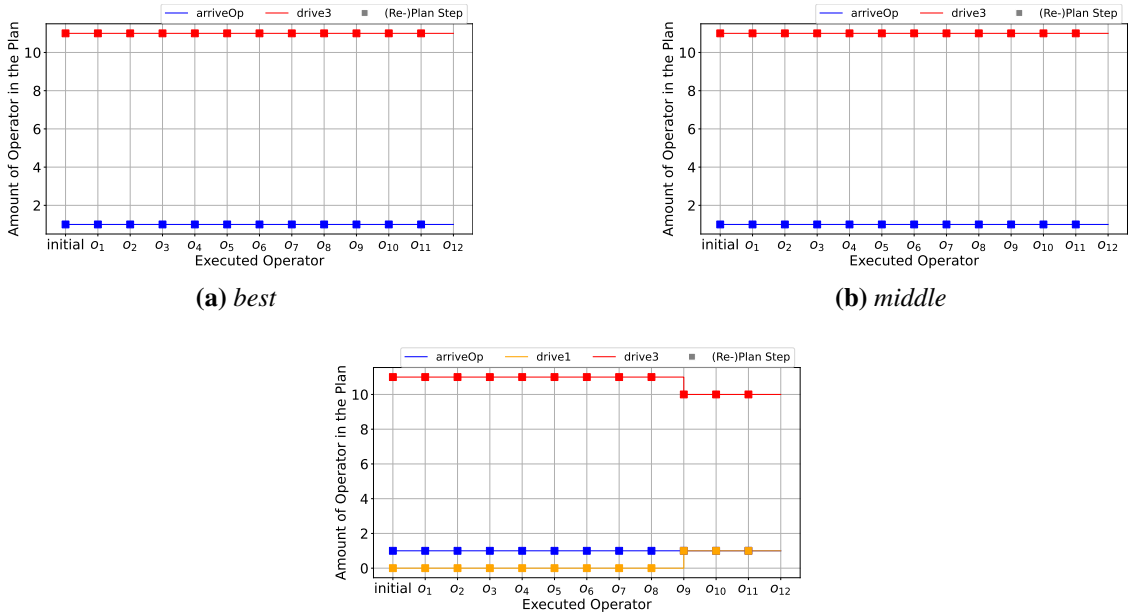
(c) Problem size 10

Figure 5.6: Average costs for the initial plan and the plans after each replan step, for problem sizes 8, 9, and 10, using utility function U_s with base parameters.

Figure 5.7 shows the operators used in the plans, similar to the previous analysis. As in the previous case, there is always one *arriveOp* in the plan, since the plan must end with this operator. The plans start with 11 *drive3* operators and no *drive1* or *drive2* operators. The operator *drive3* carries a high risk for very high costs but also offers the possibility of very low costs, making it appealing to a risk seeking attitude. Since U_s exhibits a risk seeking attitude for high remaining resources, this behavior is expected.

This pattern persists throughout the execution for both the *best* and *middle* strategies. However, for the *worst* strategy, after executing o_9, one *drive3* operator is replaced by a *drive1* operator. This is due to the *worst* cost selection strategy reducing remaining resources more significantly than the other strategies, which pushes U_s toward a more risk neutral attitude as fewer resources remain.

The small changes to the plan, or the lack thereof, suggest, that the base parameters place U_s firmly in the region where it exhibits a risk seeking attitude. Even with high incurred costs, U_s only shifts toward a more risk neutral attitude towards the end of the execution.



(a) *best*

(b) *middle*

(c) *worst*

Figure 5.7: Operator usage for different cost selection strategies for problem size 10, using utility function U_s with base parameters.

Replan Positions

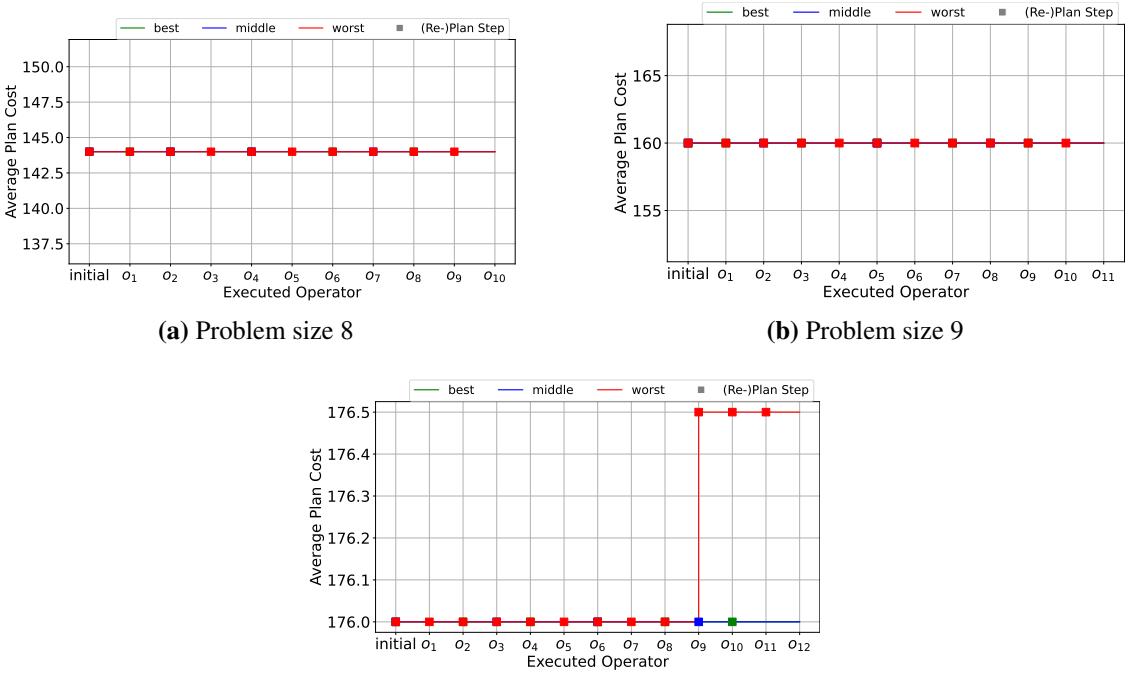
As has been seen in Figure 5.5 and Figure 5.7, the algorithm initiates replanning at many positions. However after many replans, the resulting plan is the same as the plan before replanning. This shows an interesting characteristic of the algorithm. When a replan is triggered to find a plan with higher utility in a new situation, the current plan might already be the best plan for this new situation.

5.4.2 Higher R_0

In the following, we will explore the effects of increasing the initial resources R_0 by 10%. This constitutes a value of $R_0 = 181.5$, $R_0 = 199.7$, and $R_0 = 217.8$ for problem sizes 8, 9, and 10, respectively.

Utility Function U_a

When using U_a, with a 10% increase for R_0, the average costs, shown in Figure 5.8, remain unchanged for problem sizes 8 and 9 across all cost selection strategies. For problem size 10, however, a deviation from the initial plan is observed only under the *worst* cost selection strategy. Towards the end of the plan, the average cost for the *worst* strategy begins to increase.



(a) Problem size 8

(b) Problem size 9

(c) Problem size 10

Figure 5.8: Average costs for the initial plan and the plans after each replan step, for problem sizes 8, 9, and 10, using utility function U_a with 10% higher initial resources.

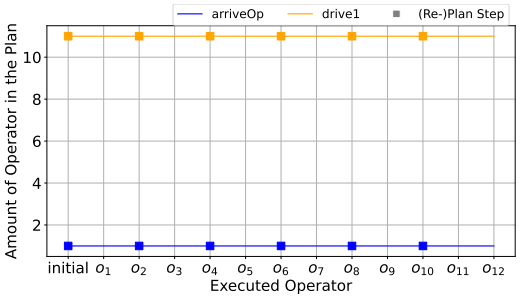
The change in used operators for the problem size 10 is shown in Figure 5.9. We observe that for both the *best* and *middle* cost selection strategies, the plans consistently consist of one *arriveOp* and eleven *drive1* operators. Only for the *worst* strategy does the plan deviate slightly, with one *drive1* operator being replaced by a *drive2* operator towards the end of execution.

This behavior is expected, as U_a displays a risk neutral attitude with high resources. Since the initial resources were increased by 10%, the function is pushed further towards risk neutrality and only reaches the point where it shifts towards a risk averse attitude, when many high costs were incurred.

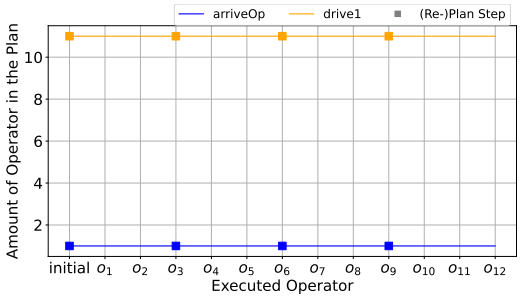
Utility Function U_s

A similar behavior can be observed when using U_s with 10% higher initial resources. Figure 5.10 shows the average plan costs for all three problem sizes. As with the base configuration, the average costs for plans in problem size 8 do not change. However, unlike in the base configuration, the average costs also do not change for problem sizes 9 and 10, when using a 10% increase for R_0.

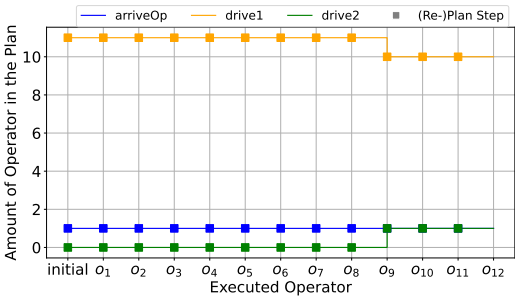
Figure 5.9 shows, that every plan in the problem of size 10 only uses *drive3* operators, with the single *arriveOp* operator. Since the base parameters already pushed U_s into its risk seeking attitude, with only high costs resulting in a single change in the plan for problem size 10, it makes sense, that higher resources result in no changes to the plan. These higher resources shift the utility function even further towards the risk seeking attitude and even high costs can not reach the point, where the risk attitude changes.



(a) *best*

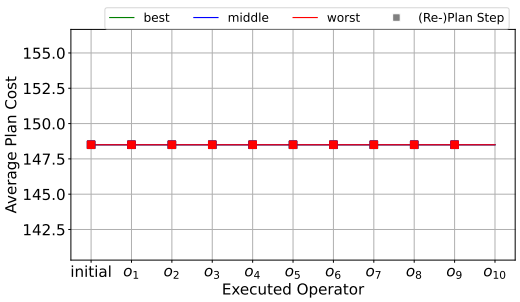


(b) *middle*

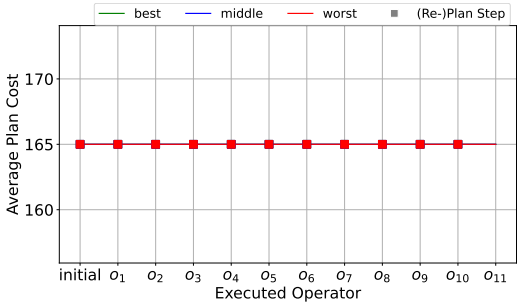


(c) *worst*

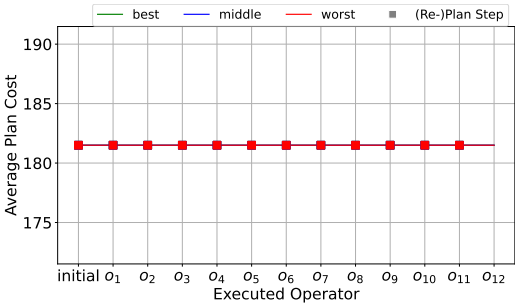
Figure 5.9: Operator usage for different cost selection strategies for problem size 10, using utility function U_a with 10% higher initial resources.



(a) Problem size 8

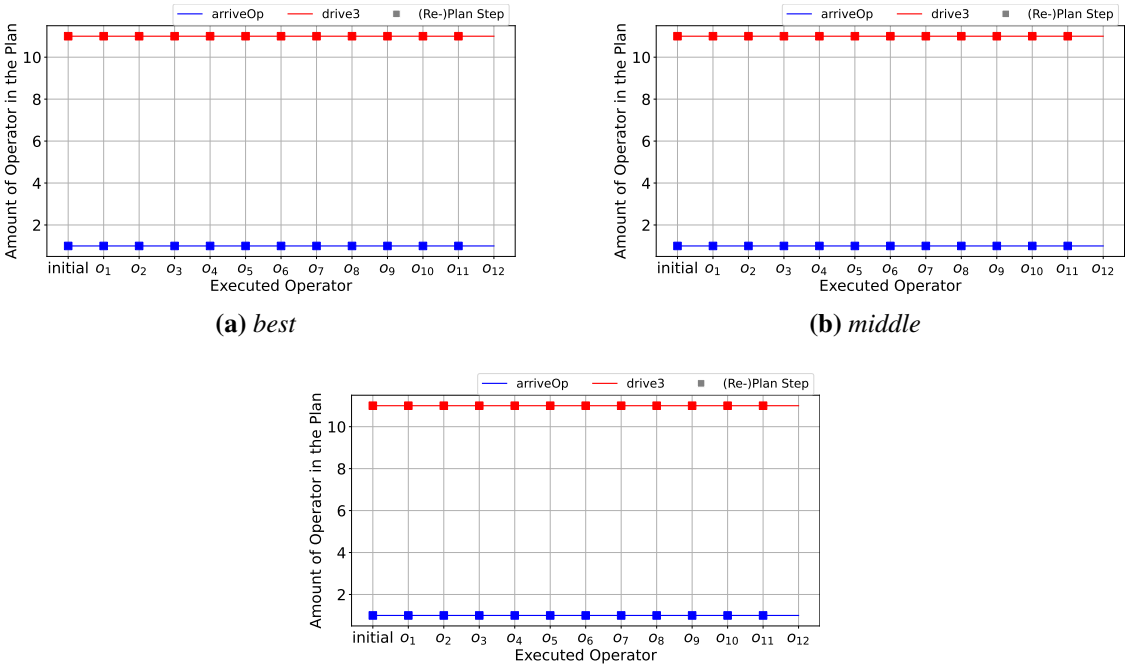


(b) Problem size 9



(c) Problem size 10

Figure 5.10: Average costs for the initial plan and the plans after each replan step, for problem sizes 8, 9, and 10, using utility function U_s with 10% higher initial resources.



(a) *best*

(b) *middle*

(c) *worst*

Figure 5.11: Operator usage for different cost selection strategies for problem size 10, using utility function U_s with 10% higher initial resources.

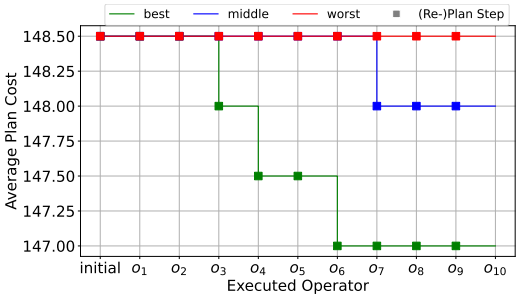
5.4.3 Lower R_0

After examining the impact of higher initial resources, we now explore the effects of decreasing the initial resources R_0 by 10%. This results in new values of $R_0 = 148.5$, $R_0 = 163.4$, and $R_0 = 178.2$ for problem sizes 8, 9, and 10, respectively.

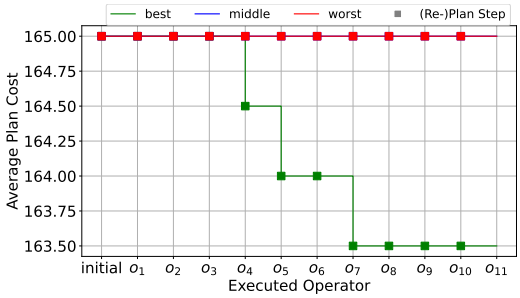
Utility Function U_a

For U_a, the average costs, shown in Figure 5.12, exhibit a similar pattern to the base configuration. For the *best* cost selection strategy, the average costs of the plans decrease over time, though this change occurs later in the execution. In contrast, both the *middle* and *worst* strategies show no change in average costs throughout the execution. The only exception is for problem size 8, where the *middle* strategy results in a slight decrease in average costs towards the end of the plan.

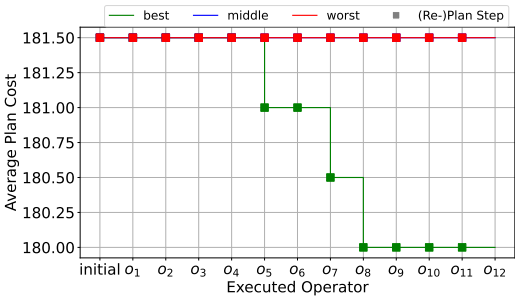
For problem size 10, the operators used in the plans are shown in Figure 5.13. The plans start with only *drive2* operators and the necessary *arriveOp*. Both the *middle* and *worst* cost selection strategies show no change in the operator selection throughout the execution. However, under the *best* strategy, the planner begins to incorporate more *drive2* operators to the plans, after the middle of the execution. This behavior is expected, as the lower initial resources push U_a towards a more risk averse attitude and, as said earlier, *drive2* is more attractive to a risk averse agent, than *drive1*. As low costs are incurred in the *best* cost selection strategy, the utility function shifts towards risk neutral behavior, resulting in more *drive1* operators being chosen, as they offer a lower average cost. This behavior is similar to the behavior observed with the base parameters. However, since the plan



(a) Problem size 8



(b) Problem size 9



(c) Problem size 10

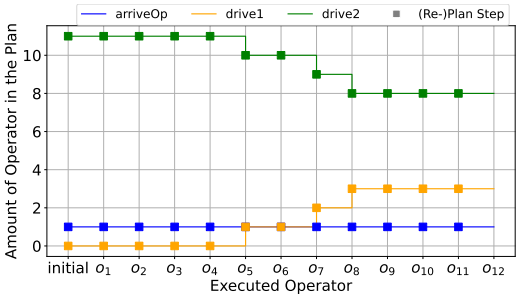
Figure 5.12: Average costs for the initial plan and the plans after each replan step, for problem sizes 8, 9, and 10, using utility function U_a with 10% less initial resources.

begins with only *drive2* operators, in the *worst* strategy, the planner is unable to introduce more *drive2* operators as it does in the base case. Additionally, since the *middle* strategy remains in the risk averse region of the utility function, it does not introduce *drive1* operators.

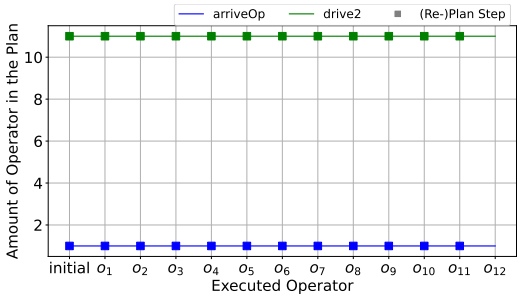
Utility Function U_s

The average plan costs when using U_s, as shown in Figure 5.14, are similar to those observed in the base configuration. However, the drop in average costs when incurring the high costs of the *worst* cost selection strategy, exists also for the problem of size 8, and occurs earlier for problem sizes 9 and 10. For *middle* and *best* cost selection strategies, the average plan costs remain unchanged, similar to the average costs of the base configuration.

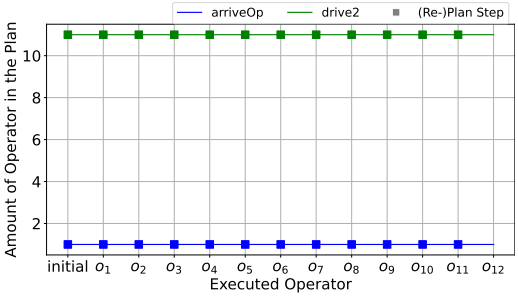
The operator choices for problem size 10, illustrated in Figure 5.15, show, that for the *best* and *middle* costs, the resulting plans are the same as in the base configuration, with only *drive3* operators and the *arriveOp* operator being used. The plans for *worst* costs is similar to the base configuration, but the shift towards *drive1* operators occurs earlier, with more *drive1* operators being selected. This makes sense, as the lower R_0 pushes the utility function further into its risk neutral region. While it still starts out risk seeking, it more quickly transitions to a more risk neutral attitude, when incurring high costs.



(a) *best*

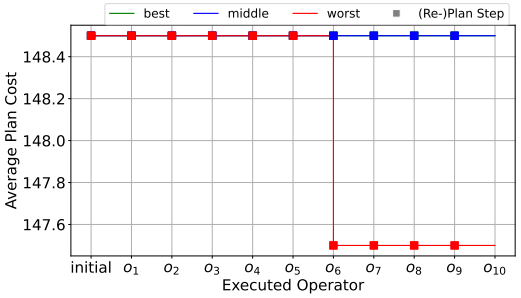


(b) *middle*

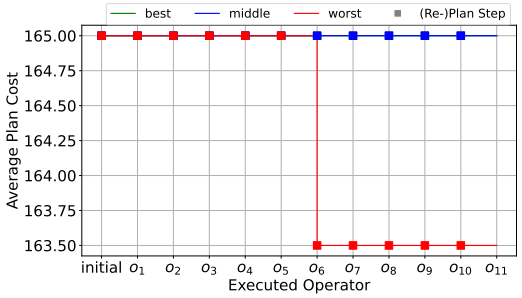


(c) *worst*

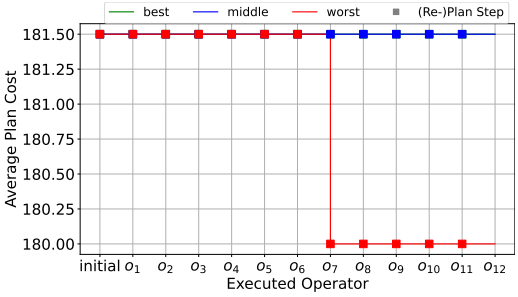
Figure 5.13: Operator usage for different cost selection strategies for problem size 10, using utility function U_a with 10% less initial resources.



(a) Problem size 8

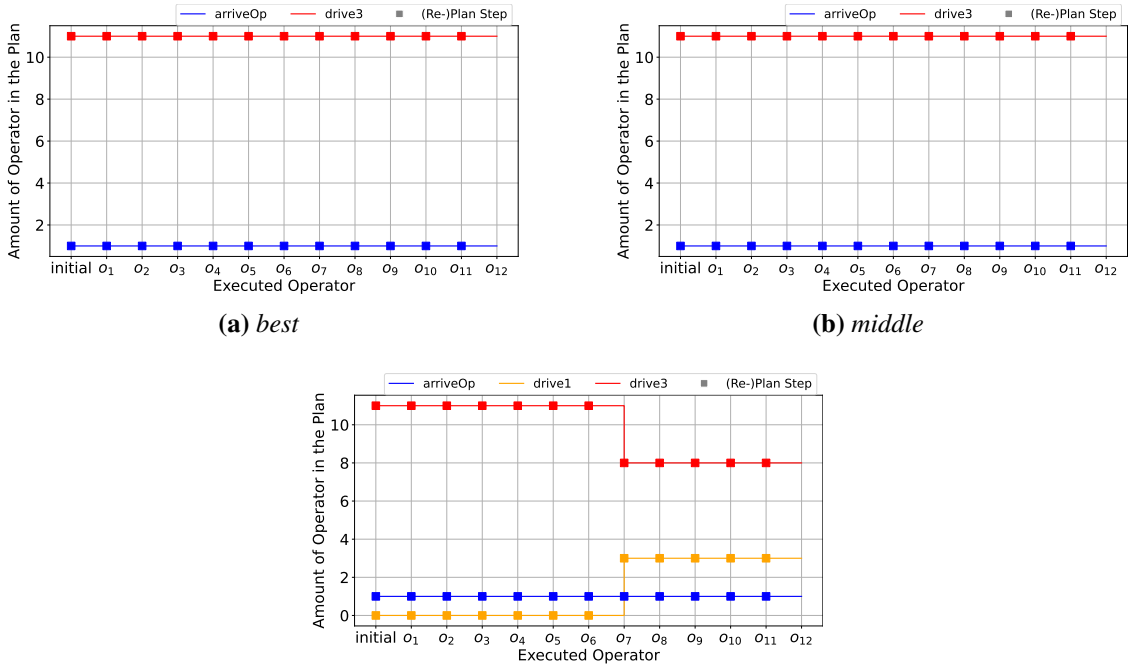


(b) Problem size 9



(c) Problem size 10

Figure 5.14: Average costs for the initial plan and the plans after each replan step, for problem sizes 8, 9, and 10, using utility function U_s with 10% less initial resources.



(a) *best*

(b) *middle*

(c) *worst*

Figure 5.15: Operator usage for different cost selection strategies for problem size 10, using utility function U_s with 10% less initial resources.

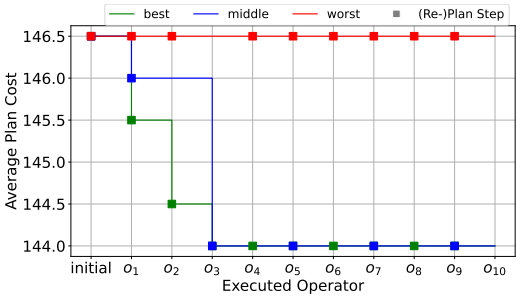
5.4.4 Higher a or b

In the following we analyze the effects of increasing the values for the parameters a and b for the utility function U_a and U_s, respectively. Specifically, we consider $a = 10$ and $b = 10$. As seen in Section 3.5, in Figure 3.5 and Figure 3.7, increasing these parameters makes the transition between the risk attitudes more pronounced, with a sharper and more drastic shift in behavior.

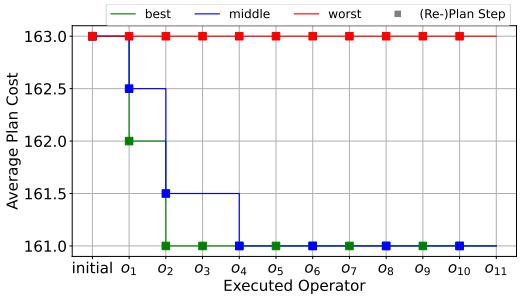
Utility Function U_a

The average plan cost, when using utility function U_a with $a = 10$, are shown in Figure 5.16. These average costs start higher than in the base configuration. For the *worst* cost selection strategy, the average cost remains the same, with only problem size 10 experiencing a slight increase early in the execution. For both the *best* and *middle* strategies, similar drops in average costs are observed, as in the base configuration. However, unlike the base configuration, the *best* and *middle* costs result in the same average costs.

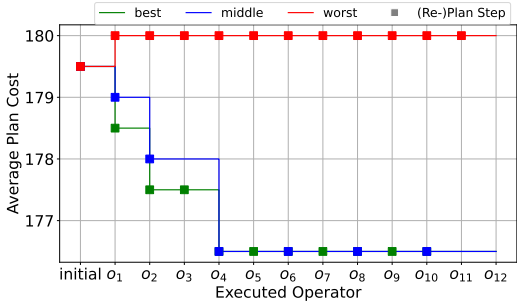
The operator selection for the plans in the problem of size 10, which are shown in Figure 5.17, explain these average costs. Like the base configuration, using U_a with $a = 10$, results in an initial plan with multiple *drive1* and *drive2* operators. These initial plans however, use one more *drive2* for one less drive1 *operator*, than in the base configuration. This is because, in the base configuration, U_a has a more gradual change from risk neutral to risk averse. But with a higher a and a more acute change, the risk attitude is more distinctly risk averse and thus one more drive2 *operator* is chosen.



(a) Problem size 8



(b) Problem size 9



(c) Problem size 10

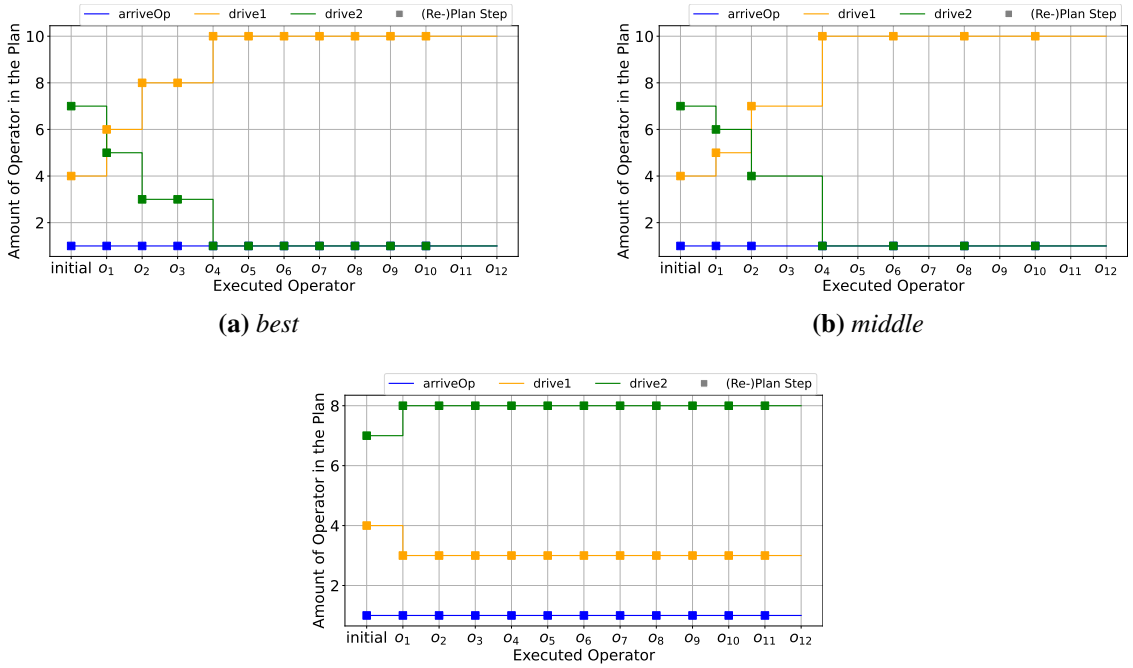
Figure 5.16: Average costs for the initial plan and the plans after each replan step, for problem sizes 8, 9, and 10, using utility function U_a with $a = 10$.

For *best* and *middle* costs, a more drastic change in the later plans can be observed, with the last plans having no *drive2* operators. Similarly, for the *worst* costs, the plan changes earlier and more drastically, than in the base configuration, with 8 *drive2* operators in contrast to the 7 in the base configuration.

These effects also result from the less gradual change in risk attitude, as for any given remaining resource, the attitude is either more pronounced risk neutral or risk averse.

Utility Function U_s

When using U_s with $b = 10$, the implementation recorded an expected utility of infinity for the resulting plans. This is due to the exponential function generating excessively large values in the Java implementation. Consequently, it became impossible to reliably select the optimal plan, as multiple plans could have reached this infinite utility and therefore could not be effectively compared. This highlights the importance of avoiding excessively high parameter values, as they can lead to numerical overflow in the Java implementation.



(a) *best*

(b) *middle*

(c) *worst*

Figure 5.17: Operator usage for different cost selection strategies for problem size 10, using utility function U_a with $a = 10$.

5.4.5 Higher D or F

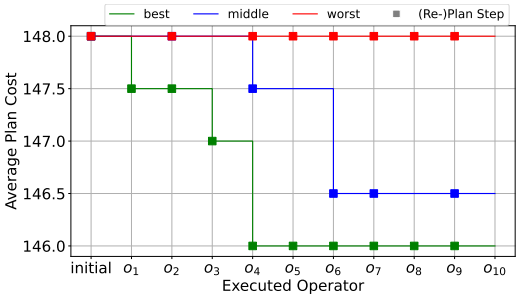
Next, we analyze the effect of setting $D = 10^5$ and $F = 10^5$ for the utility functions U_a and U_s respectively. As seen in Section 3.5, in Figure 3.4 and Figure 3.6, increasing these values shifts the point where the risk attitude changes towards higher R.

Utility Function U_a

When using U_a with $D = 10^5$, the average costs, shown in Figure 5.18, generally start at higher values compared to the base configuration. For the *worst* cost selection strategy, the average plan costs remain unchanged throughout the execution. However, for the *middle* and *best* strategies, the average costs decrease, albeit at a slower rate than observed with the base parameters.

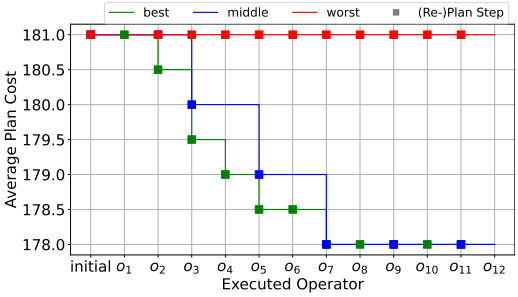
Figure 5.19 shows the operators used in the plans for problem size 10. Initially, the plans consist solely of *drive2* operators, with no *drive1* operators selected. Over time, for both the *middle* and *best* cost selection strategies, the use of *drive1* operators increases, although the rate of change is slower and the total number of *drive1* operators is lower compared to the base configuration. For the *worst* cost selection strategy, the plan consistently includes only *drive2* operators.

This behavior is similar to that observed with 10% fewer initial resources, but less pronounced. The similarity in behavior is expected, as increasing D shifts the utility function towards higher R values, effectively subtracting from the remaining resources before, as illustrated in Figure 3.4.



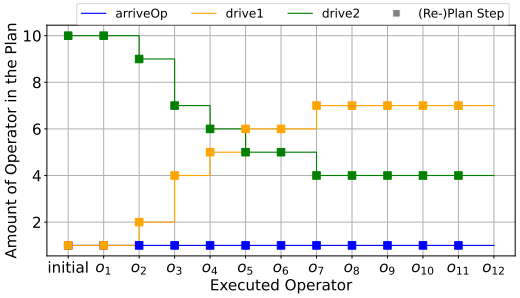
(a) Problem size 8

(b) Problem size 9

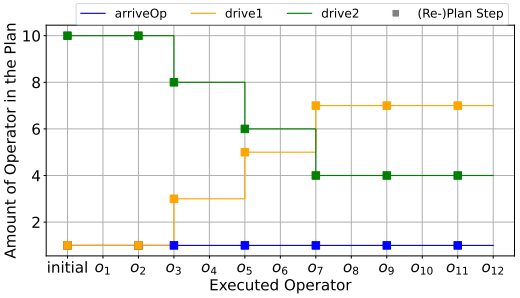


(c) Problem size 10

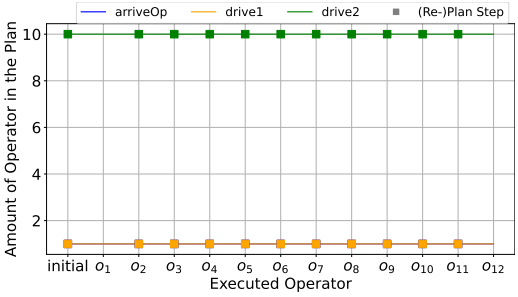
Figure 5.18: Average costs for the initial plan and the plans after each replan step, for problem sizes 8, 9, and 10, using utility function U_a with $D = 10^5$.



(a) *best*

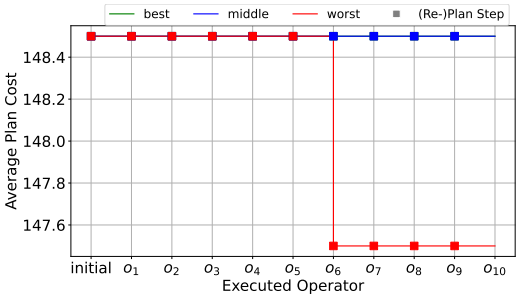


(b) *middle*



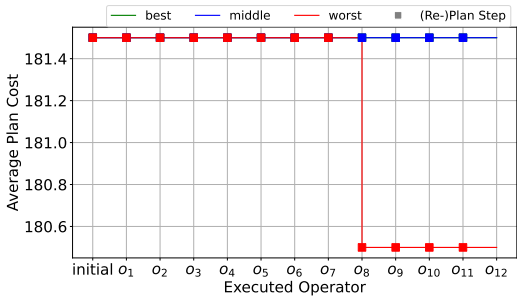
(c) *worst*

Figure 5.19: Operator usage for different cost selection strategies for problem size 10, using utility function U_a with $D = 10^5$.



(a) Problem size 8

(b) Problem size 9



(c) Problem size 10

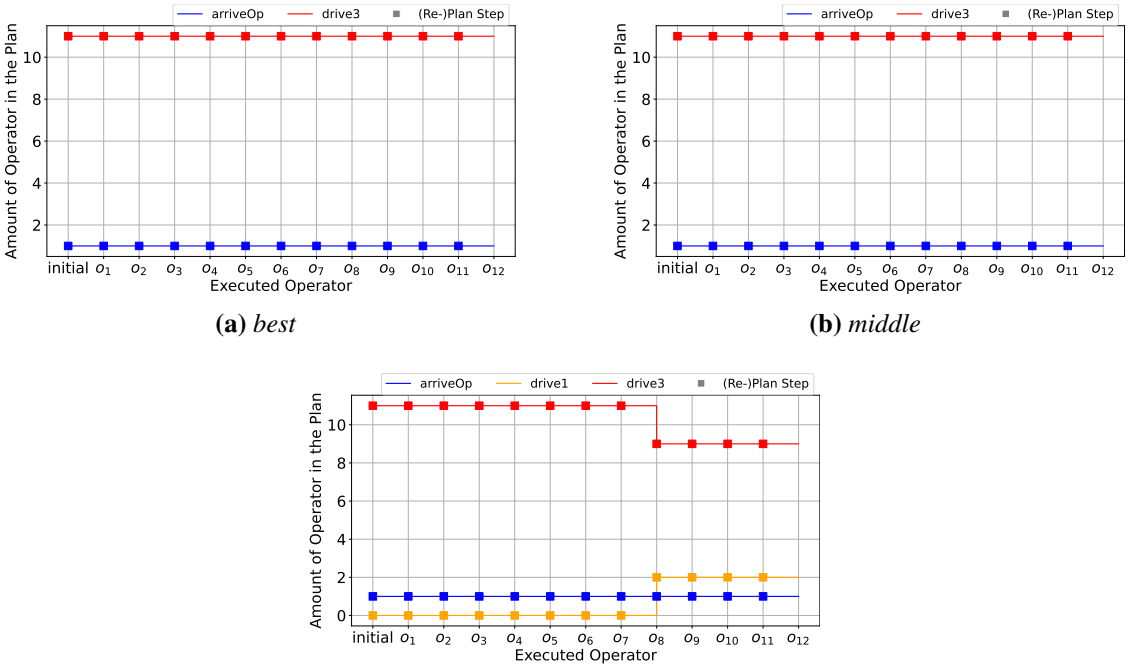
Figure 5.20: Average costs for the initial plan and the plans after each replan step, for problem sizes 8, 9, and 10, using utility function U_s with $F = 10^5$.

Utility Function U_s

When using U_s with $F = 10^5$, as shown in Figure 5.20, the initial average plan costs are higher than those observed with the base configuration. For the *best* and *middle* cost selection strategies, the average costs remain constant throughout the execution. In contrast, for the *worst* strategy, the average costs experience a drop after reaching the midpoint of the plan.

The operator choices for problem size 10 are shown in Figure 5.21. Initially, the plans consist solely of *drive3* and one *arriveOp* operator. For both the *best* and *middle* strategies, the selection of operators remains unchanged throughout the plan's execution. However, in the case of the *worst* strategy, the use of *drive1* operators gradually increases after the midpoint.

Similar to the effect of increasing D for U_a, the behavior here mirrors that observed with fewer initial resources. This similarity is expected, as increasing F, like D, shifts the utility function towards higher R values, effectively reducing the remaining resources before the risk attitude changes, as illustrated in Figure 3.6.



(a) *best* **(b)** *middle*

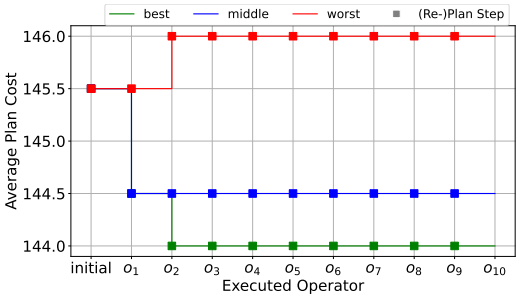
(c) *worst*

Figure 5.21: Operator usage for different cost selection strategies for problem size 10, using utility function U_s with $F = 10^5$.

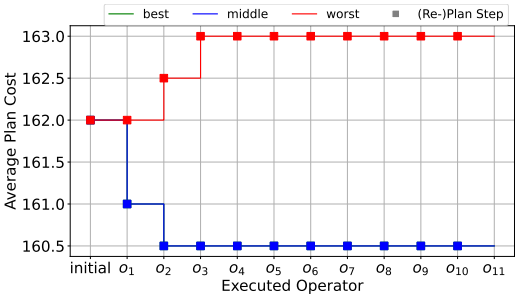
5.4.6 Lower δ

Using a lower value for the replan threshold of $\delta = 0.01$ increases the amount of replans during execution, as can be seen Figure 5.22, where a replan is initiated at every step, in contrast to the base configuration, where for the *middle* costs, a replan was not initiated after every executed operator. The increase in replans can affect the plan, as can be seen in Figure 5.22 (c), where the average cost of the plan for the *middle* costs differs throughout the execution from the base configuration, seen in Figure 5.4 (c). While δ has no direct impact on the choice for the plan with the highest utility, an earlier replan might introduce an earlier change in operators in the plan, for which different costs are incurred, than for the original operators. These different costs then shape the risk attitude differently.

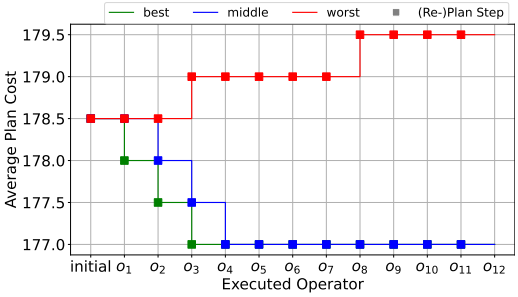
Since the base configuration with utility function U_s already did a replan at every step, decreasing δ has no effect, as can be seen in Figure 5.23.



(a) Problem size 8



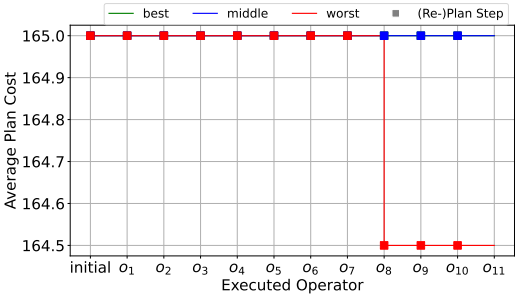
(b) Problem size 9



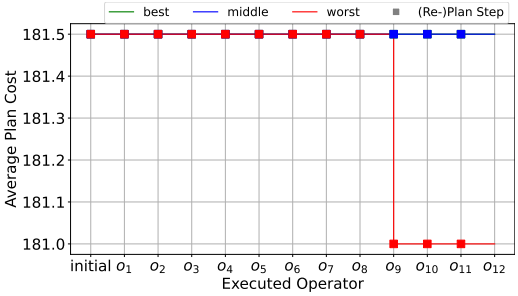
(c) Problem size 10

Figure 5.22: Average costs for the initial plan and the plans after each replan step, for problem sizes 8, 9, and 10, using utility function U_a with $\delta = 0.01$.

(a) Problem size 8



(b) Problem size 9



(c) Problem size 10

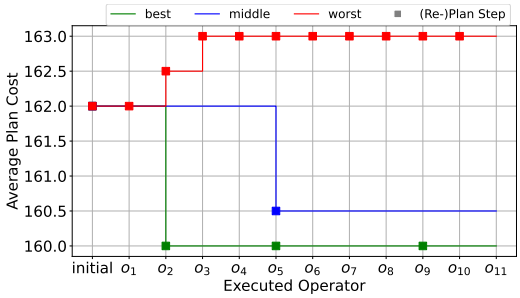
Figure 5.23: Average costs for the initial plan and the plans after each replan step, for problem sizes 8, 9, and 10, using utility function U_s with $\delta = 0.01$.

5.4.7 Higher δ

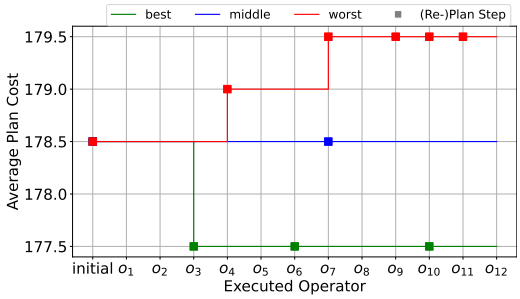
Increasing the value of the replan threshold to $\delta = 0.2$ reduces the amount of replan steps. Figure 5.24 shows that less replans were initiated, than in the base configuration, when using U_a. When using U_s, the amount of replans remains the same, as can be seen in Figure 5.25. This is due to the U_s utility function being exponential for the relevant R values in the problem, leading to exponential changes in utility even for linear changes in R.

As with using a lower value for δ, a higher value can have an impact on the resulting plan. In Figure 5.24 (c) the plan for the *middle* costs never changes, whereas in the base configuration it does, as seen in Figure 5.4 (c).

(a) Problem size 8



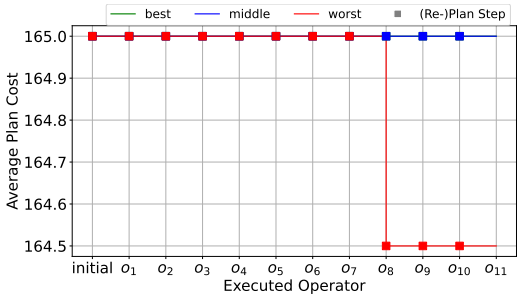
(b) Problem size 9



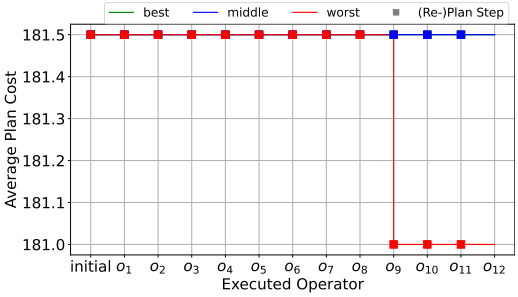
(c) Problem size 10

Figure 5.24: Average costs for the initial plan and the plans after each replan step, for problem sizes 8, 9, and 10, using utility function U_a with $\delta = 0.2$.

(a) Problem size 8



(b) Problem size 9



(c) Problem size 10

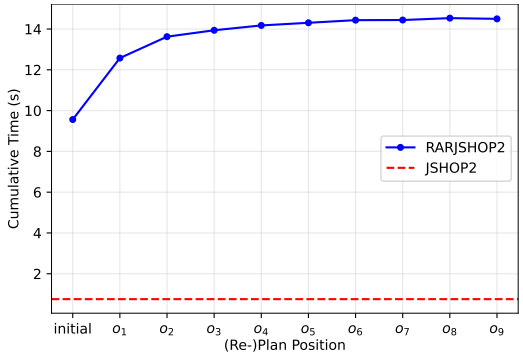
Figure 5.25: Average costs for the initial plan and the plans after each replan step, for problem sizes 8, 9, and 10, using utility function U_s with $\delta = 0.2$.

5.4.8 Performance

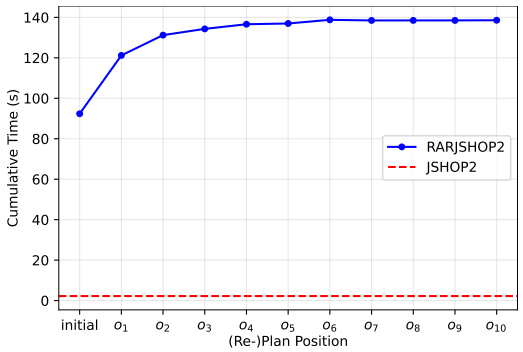
Finally, we analyze the performance of our implementation. In Figure 5.26, we can see the average cumulative calculation times required for both the initial plan and each replan, across all benchmark configurations for problem sizes 8, 9, and 10. We also include a reference to the calculation time of JSHOP2 searching for all plans in the same problem instances, using a modified domain where operators are not cost-variable.

From the data, we can observe a clear trend across all problem sizes where the cumulative calculation time grows logarithmically. This indicates, that the time required for a single (re-)plan increases exponentially with the remaining plan length. Furthermore, the calculation time for RARJSHOP2 is orders of magnitude slower than JSHOP2, which solves the same size problems without the risk-aware functionality, emphasizing the significant overhead introduced by our risk-aware planning approach.

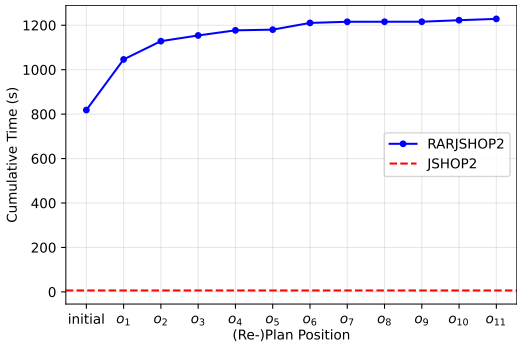
The RARJSHOP2 implementation builds directly on JSHOP2, using its native planning mechanism to find all valid plans. The additional computational effort comes from saving index positions and calculating the expected utility of each plan. Storing indices in a list has negligible impact on performance, but calculating the expected utility is considerably more demanding. In our domain, each drive operator has three possible cost values, and a plan for a problem of size N includes $N+1$ such operators. This results in 3^{N+1} possible cost combinations per plan, each of which requires an expected utility calculation. Therefore, the total number of expected utility calculations is $3^{N+1} \cdot$ *(number of found plans)*.



(a) Problem size 8



(b) Problem size 9



(c) Problem size 10

Figure 5.26: Average cumulative RARJSHOP2 calculation times for problem sizes 8, 9, and 10 across all benchmark configurations, compared to JSHOP2 calculation times using the same problems without cost-sensitive operators.

For partially executed plans, however, the costs of already executed operators are fixed. This shrinks the number of cost combinations to 3^{N+1-k}, where k is the number of executed operators. Therefore, the computational effort decreases exponentially with the number of executed steps, and the cumulative calculation time exhibits logarithmic growth relative to the remaining plan length. This relationship explains the observed trend, where planning becomes progressively faster as more of the plan is executed.

5.5 Summary of Results

RARJSHOP2 calculates plans with the highest expected utility based on a given utility function that evaluates the remaining resources. After each operator execution, calculates the ratio of change in expected utility, caused by the specific incurred costs. If this change is large enough a replan is triggered. Each replan then recalculates the plan with highest expected utility in the new situation, where some operators were already executed and incurred their specific costs.

Changes in problem parameters behave as expected. Modifying the initial resource level R_0, or the parameters D for U_a and F for U_s, shifts the points at which the respective utility functions change their risk attitudes. Adjusting the parameters a for U_a and b for U_s controls how gradually or sharply the respective utility functions transition between different risk attitudes. As a result, the generated plans vary in their use of operators to reflect the different risk attitudes.

Replanning occurs frequently under the default threshold value of $\delta = 0.05$, at nearly every step. Lowering δ further increases replanning frequency, while raising it reduces the number of replans. These changes can affect the final plans, as different sequences of operator choices and their associated costs influence the cumulative expected utility in distinct ways.

In terms of performance, RARJSHOP2 is several orders of magnitude slower than JSHOP2. This overhead stems primarily from the expected utility computation, which dominates runtime. The number of required evaluations scales exponentially with the number of operators in the plan that have uncertain costs and have not yet been executed, making this the key bottleneck in the planning process.

6 Related Work

Our approach builds directly on the foundations presented by Alnazer et al. [AGA22], who introduce a general framework for risk-aware HTN planning. Their work applies utility theory to compute the expected utility of plans, enabling planners to make decisions according to a specific risk attitude. They propose methods for modeling both static and dynamic risk attitudes, with static attitudes captured using general utility functions and dynamic attitudes represented through one-switch utility functions, which we adopt in our implementation. Their framework supports operators with both multiple possible costs and multiple possible effects, while our approach simplifies this model to only allow operators with multiple possible costs and a single effect. They also explore approaches for solving risk-aware planning problems using both plan-based and state-based HTN planners. To address the computational challenges of finding the plan with highest expected utility, Alnazer et al. present a heuristic method, applicable to segmentable utility functions.

Hirzel [Hir23] presents RAPANDA3, a plan-based risk-aware HTN planner built on top of PANDA. Like our work, their approach builds on the framework introduced by [AGA22], applying utility theory to incorporate risk sensitivity into HTN planning. They use utility functions to heuristically guide the search toward plans that align with a predefined, static risk attitude. Similar to our implementation, RAPANDA3 supports operators with multiple possible costs but restricts each operator to a single effect. However, in contrast to our focus on dynamic risk attitudes using one-switch utility functions, their work addresses static risk attitudes. Furthermore, while our framework uses a state-based planning approach, Hirzel [Hir23] utilizes plan-based planning.

Another work, building on the framework by [AGA22], is presented in [Ado23], which uses risk-aware planning in an agricultural domain, specifically for irrigation under uncertain weather conditions. Like our approach, Adomat [Ado23] uses the state-based planner JSHOP2. However, unlike our implementation, their system does not provide a general mechanism for representing risk within the planner itself. Instead, they rely on JSHOP2's capability to invoke external functions, which are used to precalculate expected utilities of operators. Risk considerations are embedded directly in the specific domain model, and a plan computed according to a static risk attitude.

A further work that builds on the framework by [AGA22] is presented in [Smi23], which applies risk-aware HTN planning to the domain of No Limit Texas Hold'em Poker. Their implementation, like ours, is based on JSHOP2 and models risk using cost and probability pairs within operator definitions, similar to our approach. In contrast to our support for dynamic risk attitudes and in-planner evaluation of expected utility, their planner adopts a static risk attitude. Expected utilities are precalculated for methods and used to guide planning decisions, by selecting the method decomposition with the highest expected utility.

Bansod et al. [BPNR22] present IPyHOP, a HTN planner designed to handle execution failures by enabling planning to resume from the point where a failure occurred within an ongoing task network. IPyHOP preserves the hierarchical structure of the solution by returning a task network, rather than a flat action sequence. This enables re-entrant planning, where the planner can continue

from the interrupted point instead of starting from scratch. While our work also involves replanning after partial plan execution, the underlying motivation and mechanism differ. IPyHOP focuses on structural recovery from failed execution by preserving the task network, whereas we support replanning by saving the indices of operator, method, and variable binding selections made during initial planning.

7 Conclusion and Future Work

In this work, we introduced a model for risk-aware HTN replanning problems. This model allows formulating planning problems, that contain tasks with probabilistic costs. The plans that solve these problems are ranked by their respective expected utility. This expected utility results from the risk attitude of the agent, and is expressed by a one-switch utility function. The risk attitude of the agent can change during execution of the plan, depending on the incurred costs. A replan of the plan is triggered, when the change in expected utility of the plan, due to the change in risk attitude, exceeds a predefined ratio.

We presented an algorithm, that solves such risk-aware replanning problems, returning the plan with the highest expected utility, and retracing its steps during a replan. Extending the JSHOP2 HTN planner and its input syntax, we implemented this algorithm in RARJSHOP2.

To evaluate the implementation, we applied it to problems of different size in a risk-sensitive domain and tested it under various parameter configurations and utility functions. The results demonstrate that the planner successfully produces plans that reflect the dynamic changes in risk attitude, represented in the utility functions.

We found, that the replan threshold δ not only changes the frequency of replans, but also indirectly impacts the resulting plans, as different replanning positions can lead to alternative operator choices and cost realizations, ultimately affecting the expected utility and structure of the final plan.

In terms of performance, we observed that our implementation is several orders of magnitude slower than the original JSHOP2 planner. This slowdown is primarily caused by the exponential number of expected utility evaluations needed when handling uncertain costs across multiple operators. At each replanning step, we must evaluate large numbers of possible cost combinations, which quickly becomes computationally expensive.

One promising direction for future work is to reduce the number of expected utility calculations. Currently, our planner must evaluate every plan in full because the one-switch utility functions we use are not segmentable, meaning the expected utility of a partial plan cannot be reliably estimated without knowing the entire plan. If it were possible to prune or terminate plan evaluation early, for example based on partial utility bounds, this could significantly reduce the number of full plan evaluations required, thereby improving planning efficiency.

Another area worth exploring is optimizing the expected utility computation itself. Since this step involves evaluating the utility for every possible combination of costs across a plan, the same total cost can arise from many different combinations, especially in longer plans with repeated operators. Introducing a caching mechanism for utility values associated with total costs could avoid redundant computations, by storing and reusing previously calculated results. This approach could greatly reduce the overhead of utility evaluation, particularly in domains with high cost overlap and long operator chains. Together, these directions offer promising opportunities to make risk-aware HTN planning more scalable and practical for complex real-world applications.

Bibliography

[Ado23] J. Adomat. “Risk-aware HTN planning for agricultural tasks”. B.S. thesis. 2023 (cit. on p. 91).

[AGA22] E. Alnazer, I. Georgievski, M. Aiello. “Risk Awareness in HTN Planning”. In: *arXiv preprint arXiv:2204.10669* (2022) (cit. on pp. 19, 27, 29–31, 45, 91).

[ASS22] Y. Al-Wreikat, C. Serrano, J. R. Sodré. “Effects of ambient temperature and trip characteristics on the energy consumption of an electric vehicle”. In: *Energy* 238 (2022) (cit. on p. 61).

[Bel88] D. E. Bell. “One-switch utility functions and a measure of risk”. In: *Management science* 34.12 (1988) (cit. on pp. 28, 45).

[BPNR22] Y. Bansod, S. Patra, D. Nau, M. Roberts. “HTN Replanning from the Middle”. In: *The International FLAIRS Conference Proceedings*. Vol. 35. 2022 (cit. on p. 91).

[EV 25] EV Database. *EV Database*. 2025. URL: https://ev-database.org/ (cit. on p. 61).

[GA14] I. Georgievski, M. Aiello. “An overview of hierarchical task network planning”. In: *arXiv preprint arXiv:1403.7426* (2014) (cit. on pp. 23, 26).

[GA15] I. Georgievski, M. Aiello. “HTN planning: Overview, comparison, and beyond”. In: *Artificial Intelligence* 222 (2015) (cit. on p. 23).

[Gal17] R. Galvin. “Energy consumption effects of speed and acceleration in electric vehicles: Laboratory case studies and implications for drivers and policymakers”. In: *Transportation Research Part D: Transport and Environment* 53 (2017) (cit. on p. 61).

[GNT04] M. Ghallab, D. Nau, P. Traverso. *Automated Planning: theory and practice*. Elsevier, 2004 (cit. on pp. 19, 21–23).

[GPAA24] I. Georgievski, A. V. Palghadmal, E. Alnazer, M. Aiello. “SH: Service-oriented system for HTN planning in real-world domains”. In: *SoftwareX* 27 (2024) (cit. on p. 23).

[HBBB21] D. Höller, G. Behnke, P. Bercher, S. Biundo. “The PANDA framework for hierarchical planning”. In: *KI-Künstliche Intelligenz* 35.3 (2021) (cit. on p. 23).

[Hir23] T. Hirzel. “A proof of concept for risk-aware plan-based HTN planning”. MA thesis. 2023 (cit. on p. 91).

[Ilg06] O. Ilghami. “Documentation for JSHOP2”. In: *Department of Computer Science, University of Maryland, Tech. Rep* (2006) (cit. on pp. 49–51).

[KGS05] M. Klusch, A. Gerber, M. Schmidt. “Semantic Web Service Composition Planning with OWLS-Xplan.” In: *AAAI fall symposium: agents and the semantic web*. 2005 (cit. on p. 23).

[LABX20] S. Lin, Z. Anshi, L. Bo, F. Xiaoshi. “HTN guided adversarial planning for RTS games”. In: *2020 IEEE International Conference on Mechatronics and Automation (ICMA)*. IEEE. 2020 (cit. on p. 23).

[Li22] Y. Li. “One-switch Utility Functions and Applications: a Review Article”. In: (2022) (cit. on p. 45).

[MDSN22] S. Mamarikas, S. Doulgeris, Z. Samaras, L. Ntziachristos. “Traffic impacts on energy consumption of electric and conventional vehicles”. In: *Transportation Research Part D: Transport and Environment* 105 (2022) (cit. on p. 61).

[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) (cit. on p. 23).

[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*. 1999 (cit. on pp. 38, 41).

[Par14] T. Parr. *ANTLR*. 2014. URL: https://antlr.org (cit. on p. 50).

[PKL+24] I. Park, C. Kim, H. Lee, C.-L. Myung, K. Min. “Comprehensive Analysis of Battery Thermal Management and Energy Consumption in an Electric Vehicle: Impact of Driving Modes and Ambient Temperatures”. In: *International Journal of Automotive Technology* (2024) (cit. on p. 61).

[SL08] Y. Shoham, K. Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008 (cit. on pp. 26, 27).

[Smi23] R. Smith. “Risk awareness in poker planning agents”. B.S. thesis. 2023 (cit. on p. 91).

[SPW+04] E. Sirin, B. Parsia, D. Wu, J. Hendler, D. Nau. “HTN planning for web service composition using SHOP2”. In: *Journal of Web Semantics* 1.4 (2004) (cit. on p. 23).

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