Abstraction-based Heuristics for Numeric Planning

Dongxu Li

Dr Enrico Scala, Dr Patrik Haslum, Dr Sergiy Bogomolov

A thesis submitted in partial fulfillment of the degree of
Bachelor of Advanced Computing (Honours) at
The Department of Computer Science
Australian National University

October 2017

Except where otherwise indicated, this thesis is my own original work.

Dongxu Li
October 26, 2017

Acknowledgements

I am grateful to my supervisors, Dr Enrico Scala, Dr Patrik Haslum and Dr Sergiy Bogomolov, for their support and encouragement throughout the whole year. Thank you for providing me with such an interesting topic to work on, for introducing me to heuristic search in numeric planning, for spending time giving me valuable feedback and advice. This project is one of the best experiences during my undergraduate studies.

I would also like to thank NWPU for providing me with a chance to study at ANU as an exchange student. My thanks also go to Dr Sylvie Thiebaux for giving me the chance to tutor the Artificial Intelligence course and Dr Lizhen Qu for providing me with the opportunity for the summer internship. These valuable experiences have helped me to prepare better for this project. I also thank other staff members in ANU CECS for offering me a great study experience during the past two years.

Finally, I would like to thank my parents Shuanghua Li and Jihong Zhang for supporting my decisions unconditionally all the time, even when sometimes I am wrong. Without their love, I would not be who I am.

Abstract

Numeric planning is a topic of research that has attracted attention in Artificial Intelligence in recent years. The motivation for research in numeric planning is to be able to better model and solve real-world planning problems requiring quantitative forms of reasoning over resources, time, space, etc.

A mainstream approach to automated planning is heuristic search, where the search is steered by heuristics that estimate the cost of achieving the goal. In this work, we present a novel approach to deriving a family of heuristics to help reasoning over planning tasks having linear numeric effects. We achieve this by constructing abstraction problems where linear numeric effects are approximated by piecewise constant numeric effects. We propose and theoretically prove conditions under which the abstraction problems constructed retain the solvability of the original problem. Using this approach, we manage to obtain two new heuristics that can be used in optimal and/or satisficing planning.

We evaluate the proposed heuristics by comparing them with other planning algorithms/systems. The result for satisficing planning is promising. On most of the benchmark problems, the proposed heuristic provides better search guidance: the planner ENHSP informed by our heuristic usually explores an one or more orders of magnitudes smaller search space. As a result, ENHSP informed by our heuristic solves 44% of the problem instances within the time restriction while other approaches solve at most 28.7% of the problem instances. The preliminary result for optimal planning shows that our proposed heuristic expands one or more orders of magnitudes fewer nodes than blind search on the instances used in the experiment, and there is a marginal improvement (within one order) concerning the node expansion when compared to the goal-sensitive heuristic.

The results indicate that the method we propose is promising enough to obtain informative heuristics and also shows the usefulness of our proposed heuristics.

Contents

Chapter 1

Introduction

Planning in real-world scenarios usually features numeric characteristics. To model and solve these planning problems compactly and efficiently, numeric planning has raised interests in the recent years. One mainstream approach to planning is heuristic search, where a search strategy is usually steered by a *heuristic function* (or in short, *heuristic*) estimating the cost of achieving the goal. A good heuristic function can help to avoid searching unpromising states, thus speeding up the search for a solution. Therefore, designing good heuristic functions has been a topic of interest since the very beginning of automated planning, including numeric planning.

A well-established approach to deriving heuristics for classical planning is by relaxing the constraints of achieving goal conditions simultaneously, the induced heuristics of which are known as *subgoaling heuristics*. Scala et al. [1] recently extended subgoaling heuristics to numeric planning. Their extensions are known as *hybrid subgoaling heuristics*. These extensions are, however, restricted to a certain spectrum of numeric planning problems, namely those involving only the conditions affected by constant numeric changes. A more expressive class of numeric planning problems, i.e. those involving conditions affected by linear numeric effects, are out of the scope of hybrid subgoaling heuristics. In this work, we aim at generalising hybrid subgoaling heuristics to this class of numeric planning problems.

1.1 Contributions

The thesis centres around the problem to generalise the subgoaling heuristics for numeric planning to cases where linear conditions can be affected by not only constant numeric effects but also linear numeric effects. This is achieved by constructing abstraction problems where linear numeric effects in the original problems are approximated by a set of piecewise constant numeric effects. After constructing abstraction problems, we apply hybrid subgoaling heuristics on the abstraction problems and take the heuristic value computed as the heuristic for the original problem.

The main results of this thesis include the development of a novel method to construct abstraction problems for numeric problems with linear numeric effects. Using this method, we introduce one inadmissible heuristic h^{add}_{abs} (Section 3.5) aiming at satisficing planning, and one admissible heuristic h^{max}_{abs} (Section 3.6) aiming at optimal

planning.

We give a formal proof (Section 3.4.1) to show that the heuristics derived from our method are safe-pruning, which means that the solvability of the planning problem is preserved using any complete search strategy steered by the proposed heuristics.

Experiments are mainly designed for understanding the effectiveness of the proposed heuristics (Section 4.1 to Section 4.3). The result in the satisficing setting shows that h^{add}_{abs} is more informative than other approaches: when informed by h^{add}_{abs}, ENHSP, which is the planner we use for the experiment, solves 15.7% more problem instances than when using AIBR [2] as the heuristic and 41.3% more problem instances than Metric-FF informed by the h^{FF} heuristic [3]. On experiment domains FO-COUNTERS-INV, FO-COUNTERS-RND, FARMLAND-LN and SAILING-LN, ENHSP informed by h^{add}_{abs} expands one or more orders of magnitude fewer search nodes and solves problems one or more orders of magnitude faster than the benchmark approaches.

h^{max}_{abs} is a preliminary attempt to the optimal setting. The evaluation result shows that ENHSP informed by h^{max}_{abs} expands one or more orders of magnitude fewer search nodes compared to blind search on instances solved while a marginal improvement (less than one order of magnitude) is observed when compared to the goal-sensitive heuristic. A similar result is found regarding running time.

1.2 Thesis Outline

The rest of the thesis is divided into the following five chapters:

- Chapter 2 introduces the reader to the necessary background knowledge our approach is based on, which includes fundamental concepts, such as classical planning, numeric planning, the modelling language and search algorithms. We also introduce the existing approaches to obtaining heuristics, including subgoaling relaxation and delete-relaxation and their numeric extensions. These approaches are the conceptual basis of our proposed heuristics or procedures used to obtain our heuristics.

- Chapter 3 introduces the abstraction-based approach we have adopted to derive heuristics in this work. First, we present a framework to construct abstraction problems and theoretically prove the properties of this approach. Then, under this framework, we propose two heuristics h^{add}_{abs} and h^{max}_{abs}. We end the chapter with discussions on restrictions of this framework.

- Chapter 4 evaluates the two proposed heuristics by comparing them with other heuristics and systems on various numeric planning domains. These domains are used in the planning competitions or extensions to classical planning domains with numeric constructs.

- Chapter 5 introduces some related work. More specifically, we examine abstraction techniques employed in the planning and model checking literature; besides, we also discuss other approaches used to obtain heuristics in the planning community other than the ones covered in the background section beforehand.

- Chapter 6 concludes the thesis by summarising the main contributions of this work. We also discuss possible further work in this chapter.

Appendix A provides the encoding for the planning domains we have used in our experiment. Appendix B provides the optimal lengths for the problem instances in FO-COUNTERS, which is one of the experiment domains we have used.

Chapter 2

Background

This chapter provides the necessary background material our work is based on. The discussion is organised as follows.

Section 2.1 introduces the conceptual model underlying sequential planning by formalising a planning domain as a state-transition system. **Section 2.2** covers STRIPS syntax and the semantics of classical planning, which is a set of archetypal planning problems based on propositional logic and also the basis for numeric planning. **Section 2.3** describes the numeric extension to the classical planning by introducing numeric variables and numeric expressions into the classical planning domains. The syntax and semantics of numeric planning are given formally in this section. **Section 2.4** introduces the Planning Domain Definition Language (PDDL), the modelling language of numeric planning we use for our work. We briefly introduce PDDL and its variations in general; and then the details of the syntax of PDDL 2.1 are illustrated by an example numeric domain.

Section 2.5 introduces the planning algorithms. Specifically, we focus on the approach to solving planning problems via heuristic search. The properties of heuristics are discussed, which are what we are concerned about when devising our heuristics. We also discuss the heuristics and procedures our proposed heuristics are based on, which include *delete relaxation*, an established approach to deriving heuristics in classical planning; *subgoaling heuristics* in classical planning; *additive interval-based relaxation*, which is an extension of delete relaxation to numeric planning and *hybrid subgoaling*, which is the extension of subgoaling heuristics to the numeric case.

2.1 Planning as State-transition System

Planning is the task of finding a sequence of actions, whose execution will take the world from an initial state to a goal state, while achieving as best as possible some certain prestated objectives [4], e.g. total cost.

The conceptual model for planning tasks is the transition system, which models behaviours of systems using a set of states and transitions. A state is a description of the properties of various objects in the environment [4]. Transitions specify how the state of the environment is changed by actions.

A *planning problem* is usually defined in the context of the *planning domain*, which

describes manipulable *objects* in the environment and their properties and *actions* that can be executed to change the state of the system. A planning domain can be formalised as a transition system by a 5-tuple $\Sigma = \langle V, S, A, \gamma, \delta \rangle$, where

- V is a set of variables;
- S is a finite or recursively enumerable set of *states* the world can be in. A state is an assignment over variables in the planning domain.
- A is a finite or recursively enumerable set of *actions*, each of which is a triple $a = \langle name(a), pre(a), eff(a) \rangle$, called an *action*, where
  - $name(a)$ is a parameterised expression $a(p_1, p_2, ..., p_k)$ in which a is a unique symbol and $(p_1, p_2, ..., p_k)$ is the signature of parameters.
  - $pre(a)$ is a set of conditions that needs to be fulfiled so that the action can be applied, known as *preconditions*.
  - $eff(a)$ is a set of assignments over variables to transform the world into a new state, called *effects*. The state transited to is denoted as $s' = eff(a)(s)$.
- γ is a *transition function*, which is a mapping $S \times A \to 2^S$. γ defines how world states are transited by actions. For $a \in A, s \in S$, if $s \models pre(a)$ and $\gamma(s, a) \neq \emptyset$; then a is *applicable* to the state s. The new state the world will be brought to by the execution of a on s is as follows

$$\gamma(s, a) = \begin{cases} eff(a)(s), & \text{if } s \models pre(a) \\ undefined, & \text{otherwise} \end{cases} \tag{2.1}$$

  Note that the definition of γ is not dependent on $pre(a)$. Instead, testing the satisfaction of $pre(a)$ is just a way to ensure that the world transitions to a non-null state.
- δ is a *cost function*, which is a mapping $S \times A \to \mathbb{Q}_{>0}$. $\delta(a)(s)$ assigns a positive cost to applying action a to state s.

A planning problem $\Pi = \langle \Sigma, s_0, G \rangle$ is an instantiated planning domain for which an *initial state* s_0, which states the world state before the execution of any action, and a set of *goal conditions* G, which characterises what constitutes a goal state, are specified. States satisfying all the conditions in G are *goal states*, denoted as s_g, i.e. $s_g \in \{s' \mid s' \models G\}$.

A *valid plan*, also referred to as a *plan*, is a sequence of actions, $\pi = \langle a_1, a_2, ..., a_n \rangle$ whose execution of π can bring the world from the initial state s_0 to a goal state s_g. The execution result of a plan $\pi = \langle a_1, a_2, ..., a_n \rangle$ on the state s, i.e. $exec(s, \pi)$, can be defined recursively as

$$exec(s, \pi) = \begin{cases} s, & \text{if } \parallel \pi \parallel = 0 \\ exec(\gamma(s, a_1), \pi'), & \text{otherwise} \end{cases} \tag{2.2}$$

The notation $\| \cdot \|$ denotes the number of actions in the action sequence; π' is the subsequence of π excluding the first action, i.e. $\pi' = \langle a_2, a_3, ..., a_n \rangle$.

The cost of π is defined as the sum of the cost of applying each action in the plan, i.e. $\delta(\pi) = \sum_{a_i \in \pi} \delta(a_i, s_{i-1})$. In this work, we focus on actions with the uniform action cost only, i.e. $\forall a_i \in \pi, \delta(a_i, s_{i-1}) = 1$, which is equal to $\| \pi \|$. A plan π is *optimal* if there does not exist any other plan π' such that $\delta(\pi') < \delta(\pi)$.

Given a planning problem, *satisficing planning* is meant for finding a valid plan of good quality; *optimal planning* is meant for finding an optimal plan.

Note that in this thesis, we focus on *deterministic* planning problems, where a transition system can only transform to *one* state by applying an action. The new state is fully determined by the effects of the action. Within this class we can find more or less expressive planning formalisms, e.g. STRIPS planning and numeric planning.

2.2 Classical Planning in STRIPS Formalism

There are various types of planning problems, characterised by different planning models and goals. One setting among them is *classical planning*, which is commonly expressed by STRIPS formalism [5] based on propositional logic. In STRIPS, a classical planning problem consists of a set of propositional facts (P) to express the world states, actions (A) and goal conditions (G).

A state s is a complete assignment over the atoms of the problems. Each atom must be grounded. Namely, each variable of the atom must be bounded by a specific object defined in the problem domain. STRIPS adopts the *closed-world* assumption, where an atom that does not appear in the world state is assumed to be false.

Preconditions of an action $pre(a)$ is a set of propositions. In order to make an action applicable in a state, the set of atoms in s must be a superset of the action precondition, i.e. a is applicable in s if and only if $pre(a) \subseteq s$.

The effect of an action $eff(a)$ is split into two sets of facts: $eff^+(a)$, called *add list* or *positive effects*, consisting of atoms that are true in the new state; $eff^-(a)$, called *delete list* or *negative effects*, consisting of atoms that are falsified in the new state. As a result, the transition function in STRIPS is formalised as

$$\gamma(s, a) = \begin{cases} (s \setminus eff^-(a)) \cup eff^+(a), & \text{if } pre(a) \subseteq s \\ undefined, & \text{otherwise} \end{cases} \tag{2.3}$$

Goal conditions G is a set of propositions to be satisfied. Each proposition is a goal condition, also known as a *subgoal*.

2.3 Numeric Planning

Though classical planning has wide applications [6, 7], its expressiveness is restricted due to the propositional representation of the world. For example, in real-world

problems involving quantitative resources, classical planning is usually not powerful enough for a compact encoding. *Numeric planning* aims at resolving this issue by extending classical planning with numbers.

2.3.1 Syntax

In addition to the set of propositional facts P, the numeric planning domain is extended with a set of numeric variables $V = \{v^1, v^2, ..., v^n\}$. Consequently, in numeric planning problems, a *state* is an assignment over $P \cup V$, notationally, a tuple $s = \langle p(s), v(s) \rangle$, where $p(s)$ is an assignment over P and $v(s) = (v^1(s), ..., v^n(s)) \in \mathbb{Q}^n$ is a vector of rational numbers representing the values of each variable in V [3]. With a little abuse of notations, we use $s.P$ to denote the propositional part of a state.

A *numeric expression* $expr$ is an arithmetic expression over V and rational numbers $\mathbb{Q}$, using operators $+, -, *, /$. $vars(expr)$ is denoted as the set of numeric variables appearing in the numeric expression $expr$.

A *numeric condition* is a comparison between a pair of numeric expressions, denoted as a triple $\langle expr, \trianglerighteq, expr' \rangle$, where $expr, expr'$ are numeric expressions and $\trianglerighteq$ is a comparator, $\trianglerighteq \in \{>, \geq, =, <, \leq\}$. The normalised form of a numeric condition is denoted as $\langle \xi, \trianglerighteq, 0 \rangle$, where ξ is acquired by moving $expr$ and $expr'$ to the same side of $\trianglerighteq$ and combining the like terms; $\trianglerighteq \in \{=, >, \geq\}$. We use this normalised form throughout our following discussions.

An *action precondition* $pre(a)$ is a pair $\langle pre_{prop}(a), pre(a)_{num} \rangle$, where $pre_{prop}(a)$ is a set of propositions as in the classical case and $pre(a)_{num}$ is a set of numeric conditions. A *numeric effect* is a triple $\langle v^i, \otimes, \xi \rangle$, where $v^i \in V$ is the variable affected by the numeric effect; $\otimes \in \{:=, +=, -=, *=, /=\}$, are known as the *assignment operators*; ξ is a normalised numeric expression representing the right-hand side of the effect. Action effect $eff(a)$ is a triple $\langle eff^+(a), eff^-(a), eff_{num}(a) \rangle$, where $eff^+(a)$ and $eff^-(a)$ are positive and negative propositional effects as defined in the classical case; $eff_{num}(a)$ is a set of numeric effects such that one numeric variable will be affected by at most one numeric effect [1]. A *numeric action* is a tuple $\langle pre(a), eff(a) \rangle$, where $pre(a)$ and $eff(a)$ are action preconditions and action effects, as defined above.

We use $op(e)$ to denote the assignment operator of the numeric effect e; $lhs(e)$ to denote the left-hand side of e, i.e. the numeric variable affected by the effect; $rhs(e)$ to denote the numeric expression on the right-hand side of e.

2.3.2 Semantics

The semantics of this formalism are described below.

The value a numeric variable v^i takes in the state s is denoted as $val(v^i, s)$. The value a numeric condition ξ evaluates to in a state s, denoted as $val(\xi, s)$, is acquired by first substituting all $v^i \in vars(\xi)$ with $val(v^i, s)$ and then simplifying it to a rational number. If the value of any v^i is undefined in s, or if a division by zero occurs, or if any

[1] This is to ensure that there are no *conflicting* effects that update a variable to different values

argument evaluates to a value outside the domain of the function, e.g. $\sqrt{x}$ is undefined whenever $x < 0$, then $val(\xi, s)$ renders an undefined value.

A numeric condition $\langle \xi, \trianglerighteq, 0 \rangle$ is satisfied in s, if $val(\xi, s)$ is defined and the comparison $val(\xi, s) \trianglerighteq 0$ stands. A numeric condition set C is satisfied if each condition of C is satisfied. C satisfied in s is written as $s \models C$. A numeric action is *applicable* if and only if $s \models pre(a)$. The execution of a numeric action a will assign new values to the numeric variables affected in the successor state $succ(s, a)$, while keeping the rest unchanged from the previous state s.

The execution result on the propositional variables remains the same as in the classical case:

$$succ(s, a).P = (s.P \backslash eff^{-}(a)) \cup eff^{+}(a) \tag{2.4}$$

For numeric variables, $\forall v^i \in V, e \in eff_{num}(a), val(v^i, succ(s, a)) =$

$$\begin{cases} val(rhs(e), s), & \text{if } \exists e\ lhs(e) = v^i \text{ and } \otimes = := \\ val(v^i, s) + val(rhs(e), s), & \textit{if } \exists e, lhs(e) = v^i \text{ and } \otimes = += \\ val(v^i, s) - val(rhs(e), s), & \textit{if } \exists e, lhs(e) = v^i \text{ and } \otimes = -= \\ val(v^i, s) * val(rhs(e), s), & \textit{if } \exists e, lhs(e) = v^i \text{ and } \otimes = *= \\ val(v^i, s) \div val(rhs(e), s), & \textit{if } \exists e, lhs(e) = v^i \text{ and } \otimes = /= \\ val(v^i, s), & \text{otherwise (Frame Axiom)} \end{cases} \tag{2.5}$$

The transition function can now be defined as

$$\gamma(s, a) = \begin{cases} succ(s, a), & \text{if } s \models pre(a), \\ undefined, & \text{otherwise} \end{cases} \tag{2.6}$$

2.4 Modelling Numeric Planning in PDDL

2.4.1 Introduction to PDDL

The Planning Domain Definition Language (PDDL) is a standard encoding language developed to model planning tasks in practice, and also to benchmark and compare performances of planning systems.

The first released PDDL version PDDL 1.2 [8], the language used for International Planning Competition 1 (IPC-1), supported encoding for classical STRIPS and ADL (an enhancement of STRIPS that is able to express conditional effects) planning. Afterwards, PDDL was further extended to incorporate richer constructs; these extensions are mostly motivated in the spirit of modelling practical planning tasks better. PDDL 2.1 [9] (for IPC-3) introduced numeric fluents (to model conditions involving non-binary resources), durative actions (actions whose execution is not instantaneous) and plan-metrics (that allow quantitative evaluation of plan quality involving numeric changes); PDDL 2.2 [10] (for IPC-4) introduced *derived predicates* (instantiated facts derivable from rules) and *timed initial literals* (facts that become True or False on predefined time points). PDDL 3 [11] (for IPC-5) emphasised the importance of

plan quality, therefore introduced *plan constraints* (constraints over possible actions or states in the plan) and *preference* (goals desired but not necessarily achieved). PDDL 3.1 [12] (for IPC-6, IPC-7) introduced *object fluents* (to allow functions ranging over not only integer or real but also any other object type). Apart from these official versions, there are also some other interesting variants. PDDL+ [13] aimed at better modelling changes initiated by the world and introduced *process* (that continuously changes values of numeric fluents over time) and *event* (similar to actions, but triggered by the world rather than agents); PPDDL [14] aimed at better modelling non-deterministic environment and introduced *probabilistic effects* (discrete probability distributions over possible effects of an action) and *rewards* (an encoding for Markovian rewards).

The numeric planning problems our work focuses on can be modelled using PDDL 2.1.

2.4.2 PDDL 2.1 – Support for Numeric Domains

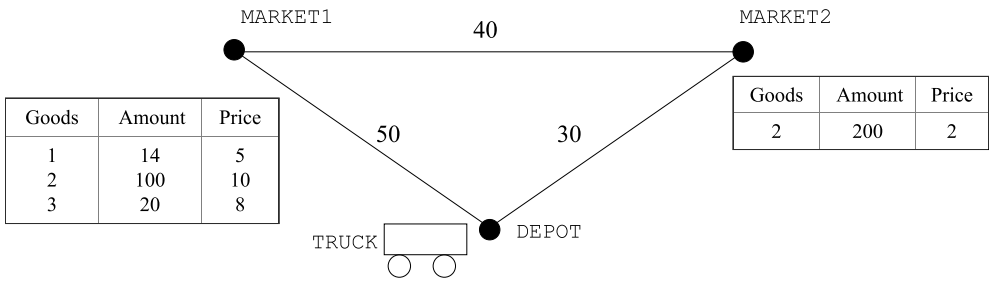
As mentioned, PDDL 2.1 was developed in the motivation of enhancing the expressiveness of temporal and numeric problems. Its features are usually identified into different *levels*: the fragment for supporting classical planning is referred to as level 1; the addition of numeric variables to model quantitative resources result in level 2; the extension of durative actions comprise level 3; continuous durative actions result in level 4; spontaneous events and processes feature level 5.

While solving planning problems features temporal and numeric components at multiple levels appears tempting yet challenging mainly due to scalability and efficiency issues, it is believed that simpler forms of planning problems do not receive enough attention [15], for example, planning problems focusing on extending the classical planning with numeric variables. Better handling of such types of problems can potentially help overcome the difficulties in more expressive planning tasks.

Our work concentrates on satisficing and optimal planning involving quantitative resources, which fits into the scope of PDDL 2.1 on level 2 [9]. Other extensions to PDDL are out of scope for this work. Since we use PDDL 2.1 throughout the thesis, we first describe the syntax of PDDL 2.1 briefly using an example planning domain, called *The Metric Travel Purchaser Problem* (TPP-metric).

2.4.3 An example domain – TPP

The TPP-metric domain is a generalisation of the Travelling Salesman Problem. The problem can be described as follows [16]: We have a set of goods and for each good, a known demand. There are a few markets, each of which provides a known limited amount of each good at a known price. The purchaser must select a subset of markets such that the given demand for each good can be purchased, and construct a tour that starts and finishes at a distinguished location (called a depot) and visits all the selected markets. The objective is to minimise a combination of the travel cost (sum of known costs for each leg of the tour) and purchases cost (sum of the quantity of the goods purchased at the market times the price at which it is offered).



Goods	Amount	Price
1	14	5
2	100	10
3	20	8

Goods	Amount	Price
2	200	2

Figure 2.1: Example of a simple TPP instance

There are two different purchasing actions available in the domain: `buy-all` and `buy-allneeded`: the first buys at a certain market the whole amount of a product sold at the market, while the second buys at a market the amount of the product that is needed to fulfil the demand[2].

Fig. 2.1 is an example showing a simple TPP instance [16]. There are three locations: two markets and one depot; and three types of goods. For each market, the amounts and prices of available goods are shown in the table next to the market. The goal of this particular small instance is to acquire 10 units of `goods1`, 100 units of `goods2` and 10 units of `goods3`, and return the truck to the depot. The optimal plan is to pass both markets, buying `goods1` and `goods3` at `market1` and `goods2` at `market2`. And then to drive the truck back to the `depot`.

2.4.4 Syntax of PDDL 2.1

To describe a planning problem in PDDL 2.1, two input files are required: a *domain* file and an *instance* file.

Listing 2.1 to Listing 2.4 display the domain file of the TPP-metric.

```
(define (domain TPP-Metric)
(:requirements :typing :fluents)
(:types
    place locatable - object
    depot market - place
    truck goods - locatable
)
```

Listing 2.1: TPP-metric domain name, requirement and objects in PDDL 2.1

First, as shown in the Listing 2.1, the domain file defines the *domain name* (`TPP-Metric`); *requirements* (specifying the features needed to be supported for the planning

[2]This encoding, as pointed out in the IPC-5 [16], introduces a constraint which is not present in the original problem, that the market from which a fraction of the available good is purchased must be the last market to be visited. Consequently, optimal plans cannot be represented with this encoding for some TPP-METRIC instances, as the objective of the optimisation couples the purchase and travel costs.

systems to solve the problem, e.g. `:typing :fluents`) and *types* (entity types used by each object). The domain defines two types: `place` and `locatable` (`place` is comprised of two subtypes: `depot` and `market`; `locatable` is comprised of two subtypes: `truck` and `goods`).

The predicate `loc` takes one argument of type `truck` and one of type `place` as shown in Listing 2.2. It represents the fact that an object is located at a particular place.

```
(:predicates (loc ?t - truck ?p - place))
```

Listing 2.2: Example of predicates in PDDL 2.1

The function `on-sale` maps an argument of type `goods` and one of type `market` to a real number as shown in Listing 2.3, representing the number of a particular type of goods available on a specific market. The function `drive-cost` maps two arguments of type `place` to a number representing the cost of the travel between the two places. The function `price` maps an argument of type `goods` and one of type `market` to a number representing the price of a particular type of goods available on a specific market. The function `on-sale` maps an argument of type `goods` to the number of such a particular type of goods already bought. The function `on-sale` maps an argument of type `goods` to the number of such a particular type of goods requested in total. The function `total-cost` takes no parameter and represents the sum of travel and purchase costs.

```
(:functions
    (on-sale ?g - goods ?m - market)
      (drive-cost ?p1 ?p2 - place)
      (price ?g - goods ?m - market)
      (bought ?g - goods)
      (request ?g - goods)
      (total-cost))
```

Listing 2.3: Example of functions in PDDL 2.1

Effects of the actions update the variables using assignment operators, namely **`assign`**, **`increase`** and **`decrease`**[3], and a numeric expression in prefix notation.

The action `drive`, which represents the operation of moving the truck from a place to another, takes one argument of type `truck`, and two of type `place` as shown Listing 2.4. The precondition for the action to be applicable is that the predicate `(loc ?t)` must be true, representing the fact that the truck is at the starting place. The effect of the action involves making the fact that the truck is at the ending place true, falsifying the predicate that the truck is at the starting place and increasing the function `total-cost` by the drive cost between the starting and ending places. The action `buy-allneeded`, which, as mentioned before, represents the operation of purchasing all the available goods in need from a market, takes one argument of type

[3]Numeric effects through operators $*=$ and $/=$ can be also expressed using `increase` and `decrease`. For example, $x\ /= y$ can be transformed into $x\ -= (y-1) \cdot x/y$, while $x\ *= y$ can be transformed into $x\ += (y-1) * x$.

`truck`, one of type `goods` and one of type `market`. The precondition for the action is that the predicate `(loc ?t ?m)` must be true, representing the fact that the truck needs to be at the market place when purchasing goods sold at the market; the numeric condition encoded in the prefix notation `(> (on-sale ?g ?m) 0)` holds true, showing that there must be some available goods on sale in the specific market and lastly, the numeric condition `(> (on-sale ?g ?m) (- (request ?g) (bought ?g))` holds true, representing that the number of available goods on sale in the market should be more than the amount needed. The effects involve decreasing the number of goods on sale by the number of goods in need, assigning the number of goods on request to the amount of goods already bought and increasing the `total-cost` by the purchasing cost. The action `buy-all` represents the operation of purchasing all the available goods in the market. `buy-all` has the same parameter signature as that of `buy-allneeded`. Its precondition differs from that of `buy-allneeded`, where the numeric condition `(<= (on-sale ?g ?m) (- (request ?g) (bought ?g)))` must hold true, showing that the number of goods on sale in the market should be no more than the number of goods in need. The effect of `buy-allneeded` involves assigning the number of goods on sale to 0, increasing the number of goods bought by the number of goods sold in the market and increasing the `total-cost` correspondingly.

```
(:action drive
 :parameters (?t - truck ?from ?to - place)
 :precondition (and (loc ?t ?from))
 :effect
       (and
         (not (loc ?t ?from)) (loc ?t ?to)
         (increase (total-cost) (drive-cost ?from ?to)))
        )

(:action buy-allneeded
 :parameters (?t - truck ?g - goods ?m - market)
 :precondition
       (and
         (loc ?t ?m) (> (on-sale ?g ?m) 0)
         (> (on-sale ?g ?m) (- (request ?g) (bought ?g)))
        )
 :effect
       (and
         (decrease (on-sale ?g ?m) (- (request ?g) (bought ?g)))
         (increase (total-cost) (* (- (request ?g) (bought ?g))
            (price ?g ?m)))
         (assign (bought ?g) (request ?g)))
        )

(:action buy-all
 :parameters (?t - truck ?g - goods ?m - market)
 :precondition
     (and
```

```
        (loc ?t ?m) (> (on-sale ?g ?m) 0)
        (<= (on-sale ?g ?m) (- (request ?g) (bought ?g)))
       )
 :effect (and
        (assign (on-sale ?g ?m) 0)
        (increase (total-cost) (* (on-sale ?g ?m) (price ?g ?m)))
        (increase (bought ?g) (on-sale ?g ?m)))
    )
)
```

Listing 2.4: Example of actions in PDDL 2.1

The second PDDL input file, instance file, specifies a problem instance in a domain. Listings 2.5 to 2.7 display the instance file of the TPP-metric example in Fig. 2.1.

First, as shown in Listing 2.5, the instance file needs to declare which domain the problem instance is from (`TPP-Metric`). Then it defines objects used in the particular instance using the **`:objects`** structure. For example, `market1`, `market2` are objects of the type `market`, representing two concrete markets in the problem instance; `depot0` is an object of the type `depot`, representing the depot. `truck0` is an object of the type `truck`, representing a truck. `goods1`, `goods2` and `goods3` are objects of the type `goods`, representing the three types of goods used in the problem instance.

```
(define (problem instance_example)
(:domain TPP-Metric)
(:objects
    market1 market2 - market
    depot0 - depot
    truck0 - truck
    goods1 goods2 goods3 - goods)
```

Listing 2.5: Example of objects in PDDL 2.1

The initial state, which is defined by the **`(:init`** structure, defines the initial values of the functions in the domain. Functions taking objects as parameters need to be initialised with specific objects, e.g. `(= (price goods1 market1) 5)` specifies that the price of `goods1` on the `market1` is 5; while others like `total-cost` are initialised independent of objects.

```
(:init
    (loc truck0 depot0)

    (= (price goods1 market1) 5)
    (= (on-sale goods1 market1) 14)
    (= (price goods1 market2) 2)
    (= (on-sale goods1 market2) 200)
    (= (price goods2 market1) 10)
    (= (on-sale goods2 market1) 100)
    (= (price goods3 market1) 8)
```

```
    (= (on-sale goods3 market1) 20)

    (= (drive-cost depot0 market1) 50)
    (= (drive-cost market1 depot0) 50)
    (= (drive-cost depot0 market2) 30)
    (= (drive-cost market2 depot0) 30)
    (= (drive-cost market1 market2) 40)
    (= (drive-cost market2 market1) 40)

    (= (bought goods1) 0)
    (= (bought goods2) 0)
    (= (bought goods3) 0)

    (= (request goods1) 10)
    (= (request goods2) 100)
    (= (request goods3) 10)

    (= (total-cost) 0))
```

Listing 2.6: Example of the initial state in PDDL 2.1

Goals, defined by the **(goals:** structure, encode the goals in the problem instance. As shown in the Listing 2.7, goals can be a conjunction of predicates and numeric conditions. For example, `(>= (bought goods1) (request goods1))` represents the goal that the number of `goods1` bought is supposed to be no less than the number of `goods1` requested; `(loc truck0 depot0)` represents the goal that the truck is supposed to return to the depot after the purchasing.

```
(:goal (and
    (>= (bought goods1) (request goods1))
    (>= (bought goods2) (request goods2))
    (>= (bought goods3) (request goods3))
    (loc truck0 depot0)))
)
```

Listing 2.7: Example of goals in PDDL 2.1

PDDL 2.1 also supports metrics used to allow planners to easily explore the effect of different metrics in the construction of solutions for problems in the same domain [9]. The metric is defined by the **(:metric** structure. The keyword `minimize` over `total-cost` specifies the optimisation objective.

```
(:metric minimize (total-cost)))
```

Listing 2.8: Example of metrics in PDDL 2.1

2.5 Planning Algorithms

2.5.1 Planning via Heuristic Search

The challenges of solving numeric planning problems in general can be summarised as follows:

- *Domain-independence*: planning systems are ideally supposed to be able to handle problems in different domains. This usually implies that few handcrafted strategies are allowed when designing a planner. As planning domains are usually characterised by various structures of the state space, devising such domain-independent strategies can be difficult.
- *Hardness of classical planning*: even with propositional variables only (no numerical constructs), the number of states grows *exponentially* in the number of variables: a planning problem of N propositional variables yields 2^N states. Even a toy planning problem can have hundreds of variables. As a result, the size of the state space easily becomes too large. This issue is known as the *state explosion*.

  In fact, the problem of deciding whether there is a valid plan for any arbitrary classical planning problem in STRIPS is in PSPACE-complete [17], which means there is no algorithm that efficiently solves planning problems in general. The explosion of the state space presents a challenge for the scalability of planning algorithms and systems.
- *Impact of numerical constructs*: the introduction of numerical variables and conditions makes the planning problems even harder. In one aspect, as the state variable can now take rational numbers that have an infinite range of values, the state space becomes infinite as well. On the other hand, the plan existence problem can be *undecidable* [18] even if it is restricted to some special cases, e.g. problems involving only two numerical variables, let alone the existence problem of optimal plans. This implies that an algorithm or system that can reliably recognise all the unsolvable planning tasks does not exist, which distinguishes numeric planning from classical planning fundamentally.

A widely-adopted technique to solve planning problems is formulating the planning problem as a search problem [19]: given a numeric planning domain $\Sigma = \langle V, S, A, \gamma, \delta \rangle$ and a planning problem $\Pi = \langle \Sigma, s_0, G \rangle$, the state space is formulated as a digraph $G = \langle V, E, w \rangle$, where V is a set of nodes, each corresponding to a state in S, and E is a set of directed edges represented by ordered pairs of nodes (n, n'), $n, n' \in V$, which shows that the state referred by n' can be transited to from the one referred by n by executing an applicable action a in A. w maps every edge (n, n') to a real number $w(n, n')$, called *weight*, representing the cost of executing a in s. The task of finding a plan through the plan state space is now expressed as finding a directed path from a given source node n_0 to a node n in the target set. Similarly, the optimal planning can be viewed as finding one such directed path while minimising the total weights along the path.

It is worth mentioning that a search node is a data structure encapsulating all the necessary information required to do the search, e.g. description of the state represented, information about the predecessor state, cost to reach the state. And for a state visited more than once during the search, there can be multiple nodes corresponding to it.

Due to the exponentially large size of the graph, search algorithms for planning are mostly incremental, in the sense that they do not require the whole search graph to be explicitly stored in the memory in the beginning of the search. Instead, the graph is constructed accumulatively on a per state basis as the search proceeds. Therefore, a good strategy for choosing the next state to expand can largely affect the effectiveness of planning algorithms.

For this reason, many modern planning systems adopt variations of the *heuristic search* [20] to find the path in the graph. With a good heuristic, *best-first search* algorithms can just expand states that are more *promising* to achieve the goal first. Whether a state is promising or not is gauged by an evaluation function $f(s)$ using $g(s)$ and/or $h(s)$, where $g(s)$ is the cost paid from the initial state to the current state s, $h(s)$ is a heuristic function that estimates the cost of the cheapest path from the state at the current node s to the goal state [21]. Best-first search algorithms usually maintain a priority queue, called a *frontier*, in which a state with lower $f(s)$ is considered to be more promising to the goal. Typical best-first search algorithms, including greedy best-first search, uniform-cost search, A* and weighted A*, are listed in Table 2.1.

The effectiveness of these algorithms is dependent on the quality of the heuristic function. Specifically, we are concerned about the following properties:

1. *Admissibility*: a heuristic is admissible if for every state $s \in S$, $h(s) \leq h^*(s)$, where $h^*(s)$ is the actual cost from the current state s to the goal state. A* algorithm informed by an admissible heuristic guarantees to find the optimal plan.

2. *Pruning-safeness*: a heuristic is safe-pruning if $h(s)$ gives ∞ only if a valid plan from s in the original planning problem does not exist. If a heuristic is safe-pruning, during the search, any state with an infinite heuristic value can be pruned, and all of the successor states will be ignored. Pruning such states without pruning-safeness guaranteed can miss plans even if the search strategy is complete.

3. *Polynomiality*: the time complexity of the procedure to compute a heuristic value should be polynomial. Any procedure in which plan existence cannot be decided in polynomial time is not practical to be applied on a per state basis.

2.5.2 Other Search Algorithms

There are also alternative search algorithms other than best-first search. Depth-first search (DFS) always expands the first successor state and goes as deep as possible and backtracks only when there is no successor state. DFS can be efficient when the depth of the goal is deep and has a low space requirement, however, it is complete

Algorithm	Description	Complete
Greedy Best-first Search	When a state is expanded, it evaluates each of its successor states n by $f(s) = h(s)$, then put all of the successor states into the frontier. The state with the lowest $f(s)$ in the frontier will be expanded next.	✓
Uniform-cost Search [23]	Similar to the greedy best-first search, however, it evaluates each state with $f(s) = g(s)$ instead. $f(s)$ estimates the cost of the action sequence from the initial state to s.	✓
A* [24]	Evaluates each state with $f(s) = g(s) + h(s)$ instead. $f(s)$ estimates the cost of the valid plan via the current state.	✓
Weighted A* [25]	Uses a weighted combination of $g(s)$ and $h(s)$, i.e. $f(s) = g(s) + w * h(s)$ to introduce a preference towards heuristics function. This usually leads to sub-optimal plan when $w > 1$.	✓
IDA* [26]	Iterative Deepening A* employs $f(s) = g(s) + h(s)$ to limit the maximal depth in the Depth-first Search. Since this algorithm does not maintain a frontier, it characterises linear space complexity. IDA* is preferable when the goals are dense and deep in the space.	✓
Enforced Hill-Climbing (EHC) [22]	Facing a new state, the algorithm performs a breadth-first search exhaustively until it finds a state n' with a lower heuristic value, then directly adds the path from n to n' into the plan and ignores all other siblings along the breadth-first search. While this can be fast, it is incomplete and usually needs to restart with a complete search engine if caught at a dead end.	✗

Table 2.1: Search Algorithms

only if there is no cycle in the search space. *Iterative-deepening A** (IDA*) is a variation of DFS, which limits the maximal search depth in DFS by the heuristics computed. Another alternative is *local search*, e.g. enforced hill-climbing that is widely used in the optimisation, planning and scheduling problems. The methodology behind local search is to look for better solutions from the neighbours. While local search appears to be effective in practice, it easily gets stuck in local minima or plateaus. Thus, no completeness or optimality can be guaranteed. *Enforced hill-climbing* is an example of local search used by the FF planning system [22].

Our work focuses on improving the performance of planning systems using heuristic search. In the following section, we introduce existing methods to derive heuristics. Our work can be viewed as an extension of them.

2.5.3 Delete Relaxation in Classical Planning

One well-established approach to deriving heuristics is by *relaxation* [22, 27–33]. Given a planning domain $\Sigma = \langle V, S, A, \gamma, \delta \rangle$ and a planning problem $\Pi = \langle \Sigma, s_0, G \rangle$, relaxing means weakening some of the constraints that restrict what the states, actions and plans are; restrict when an action or plan is applicable and the goals it achieves; and increase the costs of actions and plans [34]. The relaxation procedure induces a relaxed planning domain $\Sigma' = \langle V', S', A', \gamma', \delta' \rangle$ and a relaxed planning problem $\Pi' = \langle \Sigma', s'_0, G' \rangle$, such that a solution, not necessarily optimal though, to the relaxed problem is usually computable in polynomial time. Then by exploiting the information conceived in the solution to Π' on a per state basis, e.g. cost of the relaxed solution, a heuristic function for Π can be obtained.

One source of the complexity in classical planning comes from the delete lists. Since facts in the delete lists will not hold in the new state, the execution of an action with non-empty delete list can falsify the preconditions of other actions and lead to mutual exclusions between applicable actions, which makes planning computationally hard. From this observation, McDermott [33] first proposed a relaxation by simply ignoring the delete lists, called *delete-relaxation*. In the delete-relaxed problem Π', the state, which as described above is a set of true facts, will continually grow, and the relaxed problem is solved as soon as all the goals become true.

2.5.4 h^{max} and h^{add} - Subgoaling Heuristics in Classical Planning

In spite of the delete-relaxation, solving the delete-relaxed problem Π' optimally is NP-hard [17]. Bonet et al. [28] proposed further relaxation that the conjunctive subgoals are treated completely independently, i.e., the cost to achieve all the conditions is assumed to be the sum/maximum of the costs achieving each subgoal separately, known as *subgoaling*. Though this assumption is incorrect in general as achieving one condition will usually make other conditions easier or harder, it provides a good estimation of the difficulty to practically achieve a set of conditions.

Delete-relaxation and independent-subgoals together induce *subgoaling heuristics* h^{add} and h^{max} [28], which are computable in polynomial time. h^{max} is defined in a regression fashion as the solution to Equation 2.7, where C_p is a set of propositional subgoals (assigning G to C_p acquires estimation to the goal); the notation $|\cdot|$ denotes the size of a set; action a is an *achiever* of a singleton subgoal set C_p iff $C_p \subset eff^+(a)$ and $|C_p| = 1$, denoted as $ach(C_p)$.

$$h^{max}(s, C_p) = \begin{cases} 0, & \text{if } s \models C_p, \\ \min\limits_{a \in ach(C_p)} (h^{max}(s, pre(a)) + \gamma(a)), & \text{if } |C_p| = 1, \\ \max\limits_{C'_p \subset C_p, |C'_p|=1} h^{max}(s, C'_p), & \text{if } |C_p| > 1 \end{cases} \tag{2.7}$$

While h^{max} is admissible, it is not informative because it assumes that others will be achieved simultaneously while achieving the hardest subgoal in the set, thus underestimating the total costs over the subgoal set. In contrast, another inadmissible

but more informative heuristic h^{add} is obtained by replacing the maximum by the sum of estimated costs for subgoals in *non-singleton sets*, i.e. $\{C'_p \mid |C'_p| > 1\}$. Its inadmissibility is a result of ignoring the positive interactions between actions in achieving a set of goals.

2.5.5 Additive Interval-Based Relaxation

From a general perspective, delete relaxation c.f. Section 2.1.5 implies a possibilistic relaxed interpretation, in which variables accumulate sets of possible values monotonically [35]. Pioneered by Hoffmann [3], the delete-relaxation was extended to the numeric case using intervals representing accumulated possible values of variables. The definition of the *interval-based relaxation* (IBR) is as follows [4] [2].

The IBR problem of a numeric planning problem Π is denoted as Π'. In Π', the propositional part of the problem is handled by the standard delete-relaxation. The numeric part $v^+(s^+)$ of the relaxed state s^+ is a vector of intervals, each of which represents a set of values that variables can possibly attain. A closed interval $x = [\underline{x}, \overline{x}]$ denotes the lower bound $\underline{x}$ and upper bound $\overline{x}$ a variable x can attain. An open interval $x = (\underline{x}, \overline{x})$ is analogous but with lower bound $\underline{x}$ and upper bound $\overline{x}$ excluded. A mixed bounded interval mixes open and closed bounds. Closed interval binary operations between two intervals x and y are defined as follows [36]:

- $x + y = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$;
- $x - y = [\underline{x} - \overline{y}, \overline{x} - \underline{y}]$;
- $x \times y = [min(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}), max(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y})]$;
- $x \div y = [min(\underline{x} \div \underline{y}, \underline{x} \div \overline{y}, \overline{x} \div \underline{y}, \overline{x} \div \overline{y}), max(\underline{x} \div \underline{y}, \underline{x} \div \overline{y}, \overline{x} \div \underline{y}, \overline{x} \div \overline{y})]$ (if $0 \notin y$ otherwise one of the bounds diverges [36];

Binary operations between open or mixed bounded intervals follow the same rules; if an open and a closed bound contribute on the new interval bound, the result is open.

The syntax of numeric expression, condition and action remains unchanged while the semantics are now interpreted using interval representations. In particular, we use $val(v^i, s^+)$ to denote the interval v^i that takes in the relaxed state s^+. And the evaluation of an expression ξ in a relaxed state s^+, i.e. $val(\xi, s^+)$, is the interval derived by first substituting $v \in vars(\xi)$ with $val(v^+, s^+)$ and then computed using the interval operations defined above. A numeric condition $\langle \xi, \trianglerighteq, 0 \rangle$ is relaxed satisfied if one value in $val(\xi, s^+)$ that makes the comparison $\xi \trianglerighteq 0$ exists. A set of conditions

[4] The original definition of IBR is different from what is defined by Scala et.al [2]. Hoffmann first transformed numeric planning problems into *Linear Norm Form*, which is a monotonic form in the sense that all the numeric conditions in the domain prefer larger values. This is achieved by converting decreasing effects to the increasing effects on the so-called *"inverted variables"*, c.f. [3] Section 4 and Section 5. The underlying principle of this approach can however be interpreted as maintaining the lower bound and upper bound of possibly attainable values of a numeric variable.

C is relaxed satisfied in a relaxed s^+ if and only if each of them is relaxed satisfied, denoted as $s^+ \models C$. A numeric action is applicable if and only if $s^+ \models pre(a)$. To define the relaxed execution result of an action, we need the convex union:

Definition 2.1 (*Convex union*)**.** *The convex union between two closed intervals x,y is $x \sqcup y = [min\{\underline{x}, \underline{y}\}, max\{\overline{x}, \overline{y}\}]$. The extensions to open/mixed boundaries intervals are defined similarly.*

The execution of a numeric action a will update the numeric variables affected in the successor state $s_1^+ = succ^+(s^+, a)$ with the convex union of $val^+(v, s^+)$ and the intervals derived by action effects, while keeping the rest unchanged from the previous relaxed state s^+ [2], i.e. $\forall v \in V$, $val^+(v, s_1^+)$=

$$\begin{cases} val^+(v,s^+) \sqcup [rhs(e), rhs(e)], & \text{if } v = lhs(e), rhs(e) \text{ is a constant} \\ val^+(v,s^+) \sqcup val^+(rhs(e), s^+), & \text{if } v = lhs(e), op(e) \text{ is } := \\ val^+(v,s^+) \sqcup (val^+(v,s^+) + val^+(rhs(e), s^+)), & \text{if } v = lhs(e), op(e) \text{ is } += \\ val^+(v,s^+) \sqcup (val^+(v,s^+) - val^+(rhs(e), s^+)), & \text{if } v = lhs(e), op(e) \text{ is } -= \\ val^+(v,s^+), & \text{otherwise (Frame Axiom).} \end{cases} \tag{2.8}$$

As plan lengths in the numeric planning may be infinite, e.g. repeatedly applying $x \ /= \ 2$ on the initial state $x = 1$ will bring x only infinitely near to 0, but 0 remains unreachable in the non-asymptotic sense. Therefore, Aldinger et al. [37] consider *asymptotic reachability* in the interval relaxation instead [37], namely, a condition is asymptotically satisfiable if it can be satisfied after applying actions in the limit of unbounded number of times. Aldinger et al. [37] also showed that the reachability analysis in the interval-based relaxation is decidable for numeric planning problems, however without cyclic dependencies. The reason for this restriction lies in the presence of state-dependent non-additive effects.

Additive interval-based relaxation (AIBR) [2] resolves this issue by transforming non-additive effects to additive effects. The non-additive effect, under the formulation of PDDL 2.1, is the direct assignment operator :=, which can however be transformed into additive operators, e.g. $x := \xi$ can be rewritten as $x \mathrel{+}= \xi - x$. By this transformation, all the effects in the domain become additive, and the restriction on the acyclic restriction is effectively removed. Given a numeric planning problem Π, its additive interval-based relaxation is denoted as Π^{++}.

Deciding asymptotic reachability in domains with additive effects and cyclic dependencies is based on two observations:

1. state-dependent effects can be considered as a compact way to encode the conditional effects [2]. Each of these conditional effects depends on a specific value of the right-hand sides of effects. For example, effect $\langle x, +=, \xi \rangle$ can be written as an infinite set of conditional effects $\{when\ \langle val(\xi, s) = k \rangle\ \langle x, +=, k \rangle \mid k \in \mathbb{Q}\}$, where $\langle val(\xi, s) = k \rangle$ is a *guard* specifying the condition firing the effect.

Algorithm 1: AIBR reachability analysis

Input: Π^{++}
Output: Is G reachable?

$\Omega = supporters\ of\ A$;
$s^+ = s_0^+$;
$S = \{a \in \Omega : s^+ \models pre(a)\}$;
while $S \neq \emptyset$ *and* $s^+ \not\models G$ **do**
  $s^+ = succ^+(s^+, S)$;
  $\Omega = \Omega \setminus S$;
  $S = \{a \in \Omega : s^+ \models pre(a)\}$;
return $s^+ \models G$

2. asymptotic behaviour of additive effects can be easily modelled by looking at the sign of the right-hand side, e.g. given a state s and a numeric action a with the effect $e = \langle x, +=, \xi \rangle$, if $val(\xi, s) > 0$, even very small, applying a infinite number of times will asymptotically bring x to $+\infty$; if $val(\xi, s) < 0$, x will become $-\infty$. Similarly, for actions with decreasing effects, x will become $-\infty$ if $val(\xi, s) > 0$ and $+\infty$ if $val(\xi, s) < 0$.

Combining the two observations above, the reachability strengths of additive effects with cyclic dependencies can be modelled by a set of particular actions, called *"supporters"*. The intention of supporters is to model the state-dependency and asymptotic behaviour of additive effects explicitly, by using extra preconditions and assigning an interval as the effect to the variable.

Definition 2.2 (*Supporters of Action a*)**.** *[2] Each additive numeric effect $e \in eff_{num}(a)$ generates two supporters e^+ and e^-: if $op(e)$ is $+=$, then e^+ has the precondition $pre(a) \cup \langle rhs(e), >, 0 \rangle$ and the effect $lhs(e) := (\underline{x}, +\infty)$; the precondition of e^- is $pre(a) \cup \langle rhs(e), <, 0 \rangle$ and the effect $lhs(e) := (-\infty, \overline{x})$. The supporters generated by a decrease effect are defined analogously, but with the left-hand sides of the effects swapped. A constant assignment effect $x := k$ generates only one supporter, with precondition $pre(a)$ and the effect unchanged.*

With supporters, the reachability of AIBR can be decided by constructing the so-called *Asymptotic Relaxed Planning Graph (ARPG)*. The procedure exhaustively applies applicable supporters and tests whether goal conditions are satisfied or not. Algorithm 1 formalises such procedure, referred to as the *AIBR reachability analysis* in later discussions.

Scala et al. [2] also proposed a procedure to derive a heuristic by AIBR, the procedure of which is referred to as *AIBR counting*. AIBR counting exhaustively applies all the relaxed applicable actions till the goal is satisfied and then returns the number of actions executed, as shown in the Algorithm 2.

Algorithm 2: AIBR Counting

Input: Π^{++}
Output: Integer
1 A' is an empty multiset;
2 $s^+ = s_0^+$;
3 **while** *true* **do**
4 **foreach** $a \in A$ **do**
5 **if** *a is relaxed applicable in s* **then**
6 $s = succ^+(s, a)$;
7 $A' = A' \cup \{a\}$;
8 **if** $s \models G$ **then**
9 **return** $|A'|$

2.5.6 h^{max}_{hbd} nd h^{add}_{hbd} Subgoaling Heuristics in Numeric Planning

h^{max} and h^{add} were extended to numeric planning by redefining the concept of achiever, the definition of which relies on a parametric *regressor*. In the planning context, a regressor is a new condition yielded from an original condition through an action that ensures that the original condition holds in the states which are resulted from the execution of the action in any state satisfying the new condition [38]. In numeric planning, as effects are not *idempotent* [5], in the sense that applying an effect multiple times in a state leads to different successor states, Scala et al. [1] defined *m-times effect regressor*, by restricting actions in the domain to be *linear and self-interference free* (LSF), i.e. 1) *linear*, i.e. the right-hand sides of all the effects are linear, and 2) *self-interference free*, i.e. $\forall e, e' \in eff_{num}(a) : e \neq e', lhs(e) \cap vars(rhs(e')) = \emptyset$ to allow expressing the values of fluents affected by the repetitive executions of the action in closed form.

Theorem 2.1 (*m-times effect regressor*)**.** *Given a linear numeric condition* $c \equiv \sum_{v \in V}(w_{v,c}v) + w_c \trianglerighteq 0$*, where* $w_{v,c}, k_c \in \mathbb{Q}$*, an LSF action a, and a state s, it follows:*

$$s \models \overbrace{c^{r(a)\cdots r(a)}}^{m \text{ times}} \Leftrightarrow s \models c^{r(a,m)}$$

where

$$c^{r(a,m)} = \Big(\sum_{v \in V} w_{v,c}(f_{v,a}(m)) + w_c\Big) \trianglerighteq 0 \tag{2.9}$$

$$f_{v,a}(m) = \sum_{i=0}^{m-1} w^i_{v,a,v}\Big(\sum_{v' \neq v, v' \in V} w_{v',a,v}v' + k_{v,a}\Big) + w^m_{v,a,v}v \tag{2.10}$$

[5] Effects in classical planning, however, are idempotent.

$c^{r(a,m)}$ is the m-times regressor. Eq. 2.10 is the value of v after applying action a m times in the current state.

Definition 2.3 (*Possible Achiever*)**.** *[1] An LSF action a is said to be a (possible) achiever[6] of a numeric condition c from s if there exists an $m \in \mathbb{N}^+$ such that $s \models c^{r(a,m)}$.*

From Theorem 2.1 follows that a is a possible achiever of $c = (\sum_{v \in V} w_{v,c}(f_{v,a}(m)) + w_c) \trianglerighteq 0$ if there exists m such that

$$\sum_{v \in V} w_{v,c}(f_{v,a}(m)) + w_c \trianglerighteq 0 \tag{2.11}$$

We use $ach(c)$ to denote the set of *possible achievers* for a *numeric* condition c.

Intuitively, if a numeric condition c can be achieved by applying an action a for m times, then a is a possible achiever of c. However, it is not immediately obvious how to determine possible achievers as when $w_{x,a,x}$ is not 0 or 1, $w^m_{x,a,x}$ in Eq. 2.10 is an exponential term. As a result, determining the existence of m in Eq. 2.11 requires solving of a mixed integer non-linear optimisation problem, which is intractable.

To make the formulation tractable, Scala et al. define [1] a fragment of numeric planning problems, namely, the *simple numeric condition (SC)* case. In SCs, numeric conditions can only be affected by constant increase and decrease effects. With a little adaption on notations from their work, *SC* is defined formally as [1]:

Definition 2.4 (*Simple Numeric Condition*)**.** *Let a numeric condition c, $\langle \xi, \trianglerighteq, 0 \rangle$. c is a simple numeric condition (SC) if: (i) $\forall e \in \{e' | e' \in eff_{num}(a), a \in A, vars(\xi) \cap lhs(e) \neq \emptyset\}$, e is of the form $v \mathrel{+}= k_{v,a}$, $k_{v,a} \in \mathbb{Q}$; (ii) $\trianglerighteq \in \{>, \geq\}$, i.e. $\trianglerighteq$ is not an equality condition [7] (iii) ξ is linear, i.e. in the form of $\sum_{v \in V} w_{v,c} v + w_c$.*

This restriction allows the formulation of subgoaling heuristics in the numeric case. The admissible heuristic can now be defined as $h^{max}_{hbd}(s, C) =$

$$\begin{cases} 0, & \text{if } s \models C, \\ \min\limits_{a \in ach(C)} (h^{max}_{hbd}(s, pre(a)) + \gamma(a)), & \text{if } C \in C_{prop}, \\ \min\limits_{a \in ach(C)} (m\gamma(a)) + \min\limits_{a \in ach(C)} (h^{max}_{hbd}(s, pre(a))), & \text{if } C \in C_{num}, \\ \max\limits_{C' \subset C, |C'|=1} h^{max}_{hbd}(s, C'), & \text{if } |C| > 1 \end{cases} \tag{2.12}$$

where C_{prop}, C_{num} are the sets of propositional conditions and numeric conditions in the planning problem, respectively. The subscript *hbd* indicates the *HyBriD* subgoaling formulation.

[6] Even if there exists m such that $s \models c^{r(a,m)}$, $pre(a)$ may be falsified by repeatedly applying a for m' times, $m' < m$. This situation is however not considered in the definition of the possible achiever. That is to say, the definition of the possible achiever is a *necessary* but insufficient condition for an action to achieve a condition through repetitive executions.

[7] An equality $\langle \xi, =, 0 \rangle$ can however be expressed as the conjunction of $\{\langle \xi, \geq, 0 \rangle, \langle \xi, \leq, 0 \rangle\}$

$$h^{add}_{hbd}(s,C) = \begin{cases} 0, & \text{if } s \models C, \\ \min\limits_{a \in ach(C)} (h^{add}_{hbd}(s, pre(a)) + \gamma(a)), & \text{if } C \in C_{prop}, \\ \min\limits_{a \in ach(C)} (m\gamma(a) + h^{add}_{hbd}(s, pre(a))), & \text{if } C \in C_{num}, \\ \sum\limits_{C' \subset C, |C\prime|=1} h^{add}_{hbd}(s, C'), & \text{if } |C| > 1 \end{cases} \tag{2.13}$$

Similarly, an inadmissible setting $h^{add}_{hbd}(s,C)$ is acquired by substituting the maximum by sum for non-singleton sets in Eq. 2.12, and minimising $m\gamma(a) + h^{add}_{hbd}(s, pre(a))$ over possible achievers for singleton numeric condition sets, i.e. $\{C'_p \mid |C'_p| = 1\}$, as shown in Eq. 2.13.

2.6 Summary

This chapter first introduces the formalism of planning domains and problems, which we will use throughout this work. Classical planning, which involves propositional variables only, usually adopts STRIPS formalism. This formalism is usually not expressive enough for a compact representation for modelling many interesting real-world scenarios involving quantitative resources and was further extended with numeric constructs such as state variables that can take real numbers, numeric expressions and conditions. These numeric constructs characterise numeric planning, which is the focus of this work.

A brief overview of the Planning Domain Definition Language (PDDL) is also provided, which is a standard language used by most modern planning systems to encode planning domains and problem instances. While interesting variations of PDDL were revised, the syntax of PDDL 2.1, which is the particular PDDL version we will use in our work, was introduced with an example of TPP-METRIC domain.

One mainstream approach to solving a planning problem is by formulating it as a graph search problem. Due to the exponentially or even infinitely large size of the graph, search algorithms for planning usually construct planning graph incrementally. One type of such search algorithms are known as best-first search algorithms. We also described and compared key rationales behind some of the best-search algorithms.

By estimating the cheapest cost from a state to the goal, heuristic functions help best-search algorithms to explore the segment of search space that is more promising to contain the goal. An informative heuristic can be critical for the efficiency of the search while A* algorithm informed by an admissible heuristic can guarantee optimality of the plan. Since it is hard to achieve both of efficiency and optimality simultaneously, planning systems usually need to find a trade-off between informativeness and admissibility. Besides, in order to ensure the completeness of the search, heuristics should give infinite estimation only if the original problem is unsolvable, which is known as pruning-safeness. Lastly, the procedure to compute the heuristic needs to be in polynomial time in order to be applied on a per state basis.

A standard approach to deriving heuristics automatically is by relaxation. In classical planning, ignoring the delete lists of effects leads to the so-called "delete-relaxation". As solving a problem after delete-relaxation is still NP-hard, a further relaxation called "subgoaling" was proposed. Subgoaling relaxation takes the sum or maximum of costs to achieve each subgoal as the cost to achieve a set of goals, which leads to an informative but inadmissible heuristic h^{add} and an admissible but less informative heuristic h^{max}.

Both delete-relaxation and subgoaling heuristics were later extended to numeric planning. The extension of delete-relaxation led to the interval-based relaxation, where the evaluation of each variable is represented as an interval containing all the possibly attainable values. AIBR allows non-additive effects and removes the restriction of IBR on cyclic dependency; this effectively makes the interval-based relaxation decidable to numeric planning in general. Subgoaling heuristics were extended by migrating the concept of achievers in h^{add} and h^{max} for a condition to numeric planning. The definition of simple numeric cases allows the formulation of hybrid subgoaling heuristics, namely, h^{add}_{hbd} and h^{max}_{hbd}.

Chapter 3

Abstraction-based Heuristics

This chapter explains the approach we use to derive abstraction problems, based on which we proposed two new heuristics. The discussion is organised as follows.

Section 3.1 briefs the abstraction techniques in the context of planning. **Section 3.2** defines a framework of deriving such abstraction problems and how to acquire heuristics based on such abstractions. **Section 3.3** explains the problems that can make heuristics not permit pruning-safeness, which is the situation we have to avoid. **Section 3.4** proposes a theorem to overcome the problems mentioned, followed by the proof theoretically. **Section 3.5** details two particular heuristics we acquire under the framework, which also respect the theorem we propose.

3.1 Abstraction in Planning

In the context of planning, abstraction techniques are used to construct transition systems by aggregating states together [39]. Planning problems on such smaller systems derived are called *abstraction problems*. Abstraction problems are usually easier to investigate into than the original problem, and helpful information about the original problem can be acquired by analysing the abstraction problem.

Instead of having an explicit abstraction function, which maps a state in the original problem to a state in the abstraction problem [39–42], our approach maps every state-dependent transition into a set of state-independent ones. As a consequence, while the original state space remains unchanged, the space of the reachable states shrinks. The induced problem by abstracting state-dependence away in the transitions is the abstraction problem our heuristics are based on.

3.2 Abstraction by Decomposition and Subactioning

Our approach stems from two observations on the previous works of Scala et al. [1, 2]:

First, in the hybrid version of subgoaling, one reason they require simple numeric conditions is that with only constants involved as effects, whether an action is a possible achiever of a condition can be computed in a closed-form statically. For state-

dependent effects, online evaluations are required and it may involve solving non-linear optimisation problems as discussed in Section 2.5.6.

Second, in AIBR, however, there is no requirement on the state-independence to decide the reachability of a condition. This is because supporters capture the asymptotic behaviour of additive effects by encoding the state-dependence as extra preconditions explicitly and effects of supporters are state-independent.

Inspired by this alternative to express the state-dependence, we generalise subgoaling heuristics to a wider range of problems. The next subsection describes such fragment of problems where our approach fits in.

3.2.1 Linearly-affected Condition

Our approach is applicable to the spectrum of linear conditions that can be affected by additive linear effects [1], including state-independent effects, i.e. constant changes, and also state-dependent effects in *linear* form, which we refer to as the *non-constant linear effect*.

Definition 3.1. *(Non-constant Linear Effect) A numeric effect is a* non-constant linear effect *if it: 1) is linear, i.e. takes the form* $x \mathrel{+}= \sum_{v \in V} w_v v + w$*; 2) and has a non-constant right-hand side, i.e.* $vars(rhs(e)) \neq \emptyset$.

We denote the set of non-constant linear effects of an action a as $eff_{ncl}(a)$.

Definition 3.2. *(Linearly-affected Condition)* Let $c : \langle \xi, \trianglerighteq, 0 \rangle$, c is said to be a *linearly-affected condition* (LAC) if: (i) $\forall e \in \{e' \mid e' \in eff_{num}(a), a \in A, vars(\xi) \cap lhs(e) \neq \emptyset\}$, e is of the form $x \mathrel{+}= \sum_{v \in V} w_{v,c} v + w_c$, where $w_{v,c}, w_c \in \mathbb{Q}$; (ii) $\trianglerighteq \in \{>, \geq\}$; (iii) ξ is linear (c.f. Definition 2.4).

Intuitively, a linearly-affected condition is a condition whose numeric expression is linear; and this condition can only be affected by linear numeric effects (including constant numeric effects). Given a numeric problem Π, if all the numeric conditions are linearly-affected conditions, then Π is a *linearly-affected condition case*, which is the fragment of problems where our approach fits in.

The intuition for restricting ourselves to LAC lies in the fact that the concept of possible achievers in the hybrid version of subgoaling heuristics is defined on the linear condition assumption, with which our approach still complies. We will come back to elaborate the reason on this restriction in Section 3.7.1 after we introduce our approach.

Similarly, we require actions in the domain to be self-interference-free and linear (c.f. Section 2.5.6). As these are also restrictions imposed even in the simple condition case, we are not further narrowing down the range of problems, which can be handled by hybrid subgoaling heuristics.

[1] Non-additive effects via assignment can be transformed into additive ones through additive transformation mentioned in Section 2.5.5

3.2.2 Decomposition by Piecewise Action

Our approach relaxes the restriction on the simple numeric condition in the sense that state-dependent effects are approximated by conditional constant effects before they are handled by the hybrid subgoaling heuristics. This is achieved by approximating each action in the original problem with a set of *piecewise actions*.

The intuition underlying a piecewise action is very similar to how supporters are constructed in AIBR. The idea can be informally described as abstracting away the state-dependence of numeric effects by encoding them into extra preconditions. However, the difference is that instead of having two supporters, each assigning an interval and preconditioned on signs of the right-hand side of effects, we create a bunch of actions. Every action has a set of constant effects, each of which approximates one original effect on a certain range of possibly attainable values of the right-hand side.

For example, suppose we have an action with a non-constant linear effect, $a_1 = \langle \emptyset, \{\langle x, +=, \xi \rangle\} \rangle$[2], which increases the value of x depending on the value of ξ, and another action $a_2 = \langle \emptyset, \{\langle \xi, +=, 1 \rangle\} \rangle$, which increases the value of ξ by a constant, and the initially $val(x, s_0) = val(\xi, s_0) = 0$. The action a_1 can be *decomposed* into a set of actions with constant effects, i.e. $\{a_{1k} = \langle \{\langle \xi - k, >, 0 \rangle, \langle (k+1) - \xi, \geq, 0 \rangle\}, \{\langle x, +=, k+1 \rangle\} \rangle \mid k \in \mathbb{Z}_{\geq 0}\}$. Among them, for instance, $a_{10} = \langle \{\langle \xi, >, 0 \rangle, \langle 1 - \xi, \geq, 0 \rangle\}, \{\langle x, +=, 1 \rangle\} \rangle$ models the effect on x of executing a_1 when $0 < val(s, \xi) \leq 1$; $a_{11} = \langle \{\langle \xi, >, 1 \rangle, \langle 2 - \xi, \geq, 0 \rangle\}, \{\langle x, +=, 2 \rangle\} \rangle$ models the effect on x of executing a_1 when $1 < val(s, \xi) \leq 2$ and so on.

Before we can formally define piecewise actions, we introduce Definition 3.3 to Definition 3.8 as below. At first, we interpret the concept of *partition* in the set theory [43] in the interval context.

Definition 3.3 (*k-Partition of an Interval*)**.** *Given an interval $intv$, a finite set of k intervals $\{intv_i \mid i \leq k, i \in \mathbb{Z}_{>0}, k \in \mathbb{Z}_{>0}\}$, is a* k-partition *of $intv$, denoted as $Part(intv, k)$, iff:*

1. *$intv$ is the union of all the $intv_i$, $\bigcup_{i=1}^{k} intv_i = intv$;*
2. *$intv_i$ are disjointed with each other, $\forall m, n, m \neq n$, $intv_m$, $intv_n \in Part(intv, k)$: $intv_m \cap intv_n = \emptyset$.*

For example, suppose we have an interval $[0, 10]$, one possible *3-partition* can be $\{[0, 3), [3, 8), [8, 10]\}$. k is a parameter representing the size of the partition.

As observed before, state-dependent effect can be seen as a compact way to express conditional effects [2], and the "condition" is induced by the evaluation of the right-hand sides of the effects. Therefore, given a non-constant linear effect, our decomposition starts from partitioning on the possibly attainable values of its right-hand side, called the *domain*.

Definition 3.4 (*Reachable Closure of State s*)**.** *Given a planning problem $\Pi = \langle \Sigma, s_0, G \rangle$, and a state s', the* reachable closure *of s' is a set $S^R(s')$ containing all the reachable states by applying action sequences on s'. Formally, $S^R(s') = s' \cup (\bigcup_{a' \in A, s \in S^R(s'), s \models pre(a')} succ(s, a'))$.*

[2]For readability reason, we ignore the propositional part of the numeric action, i.e. $pre_{prop}(a)$, $eff^+(a)$ and $eff^-(a)$. Numeric actions are now denoted as tuples whose the first element is $pre_{num}(a)$ and the second is $eff_{num}(a)$.

Definition 3.5 (*Domain and Subdomain of Numeric Effect e*)**.** *Given a numeric planning problem* $\Pi = \langle \Sigma, s_0, G \rangle$*, a numeric effect* e*, the* domain *of* e *is an interval* $D(e)$ *containing a set of possibly attainable values of* $rhs(e)$ *in* Π*. Formally,* $D(e)$ *is an interval*[3] (a, b)*, such that* $\exists val \in \{eval(rhs(e), s) \mid s \in S^R(s_0)\} : val \in (a, b)$*. Each element in the k-partition of* $D(e)$*,* $Part(D(e), k)$*, is called a* subdomain*, denoted as* $d(e)_i$*,* $i \leq k$*.*

Given a subdomain of a numeric effect $d(e)_i$, we use $lb(d(e)_i)$ to denote the *lower bound* of $d(e)_i$ and $ub(d(e)_i)$ to denote the *upper bound* of $d(e)_i$.

After partitioning, we use a constant to approximate the behaviour of the effect in each subdomain. We call the constant the *representative sample*.

Definition 3.6 (*Representative Sample of a Subdomain*)**.** *Given a subdomain of a non-constant linear effect* $d(e)_i$*, its* representative sample *is a non-zero constant*[4] *in the subdomain, denoted as* $rep(d(e)_i)$*,* $rep(d(e)_i) \in d(e)_i$*,* $rep(d(e)_i) \neq 0$*.*

The definition of subdomains and representative sample allow us to decompose a numeric effect into a finite set of *conditional effects*, called *piecewise effects*. Each piecewise effect applies when the evaluation of the right-hand side falls into one particular subdomain, and has a constant numeric effect induced by the representative sample.

Definition 3.7 (*Numeric Conditional Effect*)**.** *A* numeric conditional effect e_{cond} *is a pair* $\langle C, E \rangle$*, where* C *is a set of numeric conditions, called* guards*, and* E *is a set of numeric effects. When and only when the action with* e_{cond} *is applicable and* C *is satisfied,* $lhs(e_{cond})$ *is updated by* E *in the numeric planning semantics; otherwise,* $lhs(e_{cond})$ *remains unchanged by* e_{cond}*.*

Definition 3.8 (*Piecewise Effects*)**.** *Given a numeric action* a*, one of its non-constant linear numeric effects* e *and subdomains* $Part(D(e), k) = \{d(e)_i \mid i \in \mathbb{Z}_{>0}, i \leq k\}$*,* piecewise effects of e*, denoted as* $Pwe(a, e, k)$*, is a finite set of* k *conditional effect* $\{pwe(a, e, k)_i \mid i \in \mathbb{Z}_{>0}, i \leq k\}$*, each of which 1) is guarded by the bounds of a* unique *subdomain* $d(e)_i$*; 2) has the operator* $op(e)$*; 3) changes* $rhs(e)$ *by* $rep(d(e)_i)$*. Formally,* $pwe(a, e, k)_i = \langle \{\langle rhs(e) - lb(d(e)_i), \trianglerighteq, 0\rangle, \langle ub(d(e)_i) - rhs(e)\ , s), \trianglerighteq, 0\rangle\}, \{\langle lhs(e), op(e), rep(d(e))_i\rangle\}\rangle$*,* $\trianglerighteq \in \{>, \geq\}$*.*

For example, given that an action is preconditioned on $x > 0$ with a non-constant linear effect $y \mathrel{+}= z$, one possible decomposition of the effect can be informally written as

$$e : y \mathrel{+}= z \Rightarrow \begin{cases} pwe(a, e, 3)_1 = \langle \{\langle z, >, -\infty\rangle, \langle -z, >, 0\rangle\}, \{\langle y, +=, -2\rangle\}\rangle, \\ pwe(a, e, 3)_2 = \langle \{\langle z, >, 0\rangle, \langle -z, \geq -2\rangle\}, \{\langle y, +=, 1\rangle\}\rangle, \\ pwe(a, e, 3)_3 = \langle \{\langle z, >, 2\rangle, \langle -z, >, -\infty\rangle\}, \{\langle y, +=, 2\rangle\}\rangle. \end{cases} \tag{3.1}$$

Note that by definition, the number of piecewise effects equals to the number of subdomains.

[3] Domain can be an open interval, a closed interval or a half-open interval.

[4] The marginal case where both the lower bound and upper bound of a subdomain are zero is not considered as any additive effect with a right-hand side evaluating to 0 will not change the value of the affected variable.

From our previous observation that state-dependence is encoded as extra preconditions of actions, the semantics of piecewise effects can be preserved in a similar way in *piecewise actions*, each of which approximates non-constant linear effects with a unique combination of piecewise effects and takes the union of guards as extra preconditions.

Definition 3.9 (*Piecewise Actions of Action a*)**.** *For a* linear action *(c.f. Section 2.5.6)* $a = \langle pre_{num}(a), eff_{num}(a)\rangle$*, where* $eff_{ncl}(a) = \{e_1, e_2, ..., e_m\}$*, a piecewise action* $pwa(a)$ *is acquired by replacing each* $e_i \in eff_{ncl}(a)$ *with one of its piecewise effect* $pwe(a, e_i, k_i)_j = \langle C_{ij}, E_{ij}\rangle, j \in \mathbb{Z}_{>0}, j \leq k_i$*, and taking the union of action preconditions and guards of each piecewise effect as the precondition, formally,* $pre(pwa(a)) = pre(a) \cup (\bigcup_{1\leq i\leq m}\{C_{ij}\})$*. Each piecewise action should be* unique*, that is to say,* $pwa(a) = pwa(a)'$ *implies that at least one original effect is replaced by different piecewise effects in* $pwa(a)$ *and* $pwa(a)'$*.*

So far, we approximate a numeric action with non-constant linear effects by a set of actions with only constant effects. However, in practice, the number of such piecewise actions of a numeric action can be exponential in the number of non-constant linear effects. For example, suppose we have an action a with 3 non-constant numeric effects, and for each of them, we have on average 4 subdomains, then, we end up with generating $4^3 = 64$ piecewise actions for one single action a. And for actions A in the domain, the number of piecewise actions is upper bounded by ${k_{max}}^{e_{max}} \cdot |A|$, where k_{max} is the maximal number of subdomains for non-constant linear effects and e_{max} is the maximal number of non-constant linear effects for each action. However, this drastically increases the problem size and makes it rather expensive for the approach to be applicable to even small problems and not promising to be scaled up to realistic scenarios.

3.2.3 Subactioning and Subactions

To handle the explosion of the problem size, we make further relaxation on piecewise actions. We call the proposed relaxation *subactioning*. A connection to subactioning can be found in the work by Coles et al. [44], where they formed additional actions from propositional conditional effects. However, their approach was not established in the context of numeric planning or, specifically, of the subgoaling heuristics.

If a conditional effect is dependent only on a static numeric condition, i.e. whether the condition holds or not does not change from state to state, then it is possible to decide in the grounding whether to apply it in the problem instance [44]. However, as piecewise effects are constrained by a set of conditions involving variables, whose values change on states, it is necessary to consider whether the effect applies in each state. This is achieved by forming each piecewise effect as a separate action called a *piecewise subaction*.

Definition 3.10 (*Piecewise Subaction*)**.** *Given a numeric action a, for each of its non-constant linear effect e, every piecewise effect* $pwe(a, e, k) = \langle C, E\rangle$ *generates a numeric action* $sub_{pw}(a, e, k)$*, called a* piecewise subaction*. The propositional preconditions of* $sub_{pw}(a, e, k)$ *is the same as that of a, i.e.* $pre_{prop}(sub_{pw}(a, e, k)) = pre_{prop}(a)$*. The numeric preconditions of*

$sub_{pw}(a,e,k)$ is the union of numeric preconditions of a and the guards of $pwe(a,e,k)$, i.e. $pre_{num}(sub_{pw}(a,e,k)) = pre_{num}(a) \cup C$. The propositional effect of $sub_{pw}(a,e,k)$ is empty, i.e. $eff_{prop}(sub_{pw}(a,e,k)) = \emptyset$. The numeric effect of $sub_{pw}(a,e,k)$ is the same as that of the piecewise effect, i.e. $eff_{num}(sub_{pw}(a,e,k)) = E$.

We denote all the piecewise subactions of action a as $sub_{pw}(a)$.

Following the example in the Section 3.2.3, after decomposing the non-constant linear effect $y \mathrel{+}= z$ into piecewise effects, we can generate a subaction for each piecewise effect, that is

$$\begin{cases} sub(a,e,3)_1 = \langle\{\langle x,>,0\rangle, \langle z,>,-\infty\rangle, \langle -z>0\rangle\}, \{\langle y,+=,-2\rangle\}\rangle, \\ sub(a,e,3)_2 = \langle\{\langle x,>,0\rangle, \langle z>0\rangle, \langle -z\geq -2\rangle\}, \{\langle y,+=,1\rangle\}\rangle, \\ sub(a,e,3)_3 = \langle\{\langle x,>,0\rangle, \langle -z,>,-\infty\rangle, \langle z,>,2\rangle\}, \{\langle y,+=,2\rangle\}\rangle, \end{cases}$$

For any action with propositional or constant numeric effects, we do not generate one subaction for each of these effects. Instead, we generate a *constant subaction* keeping all of its propositional and constant numeric effects aggregated to tighten our relaxation.

Definition 3.11 (*Constant Subaction*)**.** *Given a numeric action a, its* constant subaction*, denoted as $sub_{const}(a)$, has the same propositional and numeric preconditions as a, that is, $pre(sub_{const}(a)) = pre(a)$. The propositional effect is the same as that of a, and the numeric effects are the constant numeric effects of a, i.e. $eff_{prop}(sub_{const}(a)) = eff_{prop}(a)$; $eff_{num}(sub_{const}(a)) = \bigcup e_i, e_i \in eff_{num}(a), vars(rhs(e_i)) = \emptyset$.*

From the actions in A, we generate a set of piecewise subactions Ω_{pw} by the mechanism defined in Definition 3.10, and also a set of constant subactions Ω_{const} by Definition 3.11. The size of Ω_{pw} is *at most* $e_{avg} \cdot k_{avg} \cdot |A|$; the size of Ω_{const} is *at most* $|A|$. We use Ω to denote the union of Ω_{pw} and Ω_{const}, as the set of all the subactions generated for the actions in A.

3.2.4 Abstraction by Decomposition and Subactioning

Subactions allow us to define our abstraction problem.

Definition 3.12 (*Abstraction Problem by Decomposition and Subactioning*)**.** *Given a numeric planning domain $\Sigma = \langle V, S, A, \gamma, \delta\rangle$ and a numeric planning task $\Pi = \langle \Sigma, s_0, G\rangle$, an* abstraction problem by decomposition and subactioning *is acquired by replacing A with Ω. We denote the relaxed domain as Σ_{desub}[5]$= \langle V, S, \Omega, \gamma, \delta\rangle$ and the induced planning problem as $\Pi_{desub} = \langle \Sigma_{desub}, s_0, G\rangle$.*

The abstraction problem we obtain is a simple numeric condition problem, which can be handled by the hybrid subgoaling heuristics. Then, we apply h^{add}_{hbd} or h^{max}_{hbd} on Π_{desub} and take the heuristics computed as the heuristics for the original problem. We call such heuristics *abstraction-based heuristics*. Given a set of conditions C and a state

[5] Abbreviation for **de**composition and **sub**actioning.

s, the heuristic value to satisfy C from s computed by abstraction-based heuristics is denoted as $h_{abs}(s, C)$.

3.3 Problems with Abstraction

Although the rationale behind the abstraction appears to be intuitive, heuristics computed on particular abstractions problems may not be safe-pruning. We summarise reasons leading to such abstractions as *domain incompleteness* and *asymptotic behaviour distortion*.

3.3.1 Domain Incompleteness

Domain incompleteness is a consequence of decomposing a right-hand side of the effects on a domain containing only a subset of its reachable values. According to Definition 3.5, a domain may not contain all the reachable values of the right-hand side of the effect. Moreover, computing the exact set of reachable values of each right-hand side is at least as hard as determining the plan existence problem in the original problem. Therefore, it is quite likely that each of the domains, on which decomposition is based, contains only a subset of reachable values of the right-hand side of the effect. Consequently, for any state where the right-hand side evaluates to a value outside such an *incomplete* domain, there will not be any subaction whose indirect preconditions are satisfied in the abstraction problem. As a result, given the state, such effect cannot be approximated in the abstraction problem and conditions involving the affected variable would *possibly* be made unreachable.

For example, consider the example in Section 3.2.3, suppose the initial values are $x = 1$, $y = 0$ and $z = 3$, and the goal is to make y larger than 0. If the domain for the effect $e : y \mathrel{+}= z$ is $(-2, 2)$, then we generate piecewise subactions as below:

$$\begin{cases} sub(a, e, 2)_1 = \langle \{\langle x, >, 0\rangle, \langle z, >, -2\rangle, \langle -z, >, 0\rangle\}, \{\langle y, +=, -1\rangle\}\rangle, \\ sub(a, e, 2)_2 = \langle \{\langle x, >, 0\rangle, \langle z, \geq, 0\rangle, \langle -z, >, -2\rangle\}, \{\langle y, +=, 1\rangle\}\rangle, \end{cases}$$

where a is the action with e as the single effect.

Then, in the initial state, as $z = 3$ does not satisfy preconditions of either $sub(a, e, 2)_1$ or $sub(a, e, 2)_2$, there is no applicable action in the abstraction problem that can affect the value of y. Consequently, though the goal can be achieved by applying a once in the original problem, there is no plan to achieve it in the abstraction problem.

One trivial solution to ensure that each attainable value is captured in the domain is to extend the estimated domain to incorporate all values from $-\infty$ to ∞, as we mention later in Section 3.5.2.

3.3.2 Asymptotic Behaviour Distortion

Asymptotic behaviour distortion is the consequence of approximating effects using constants imprecisely. For a right-hand side of an effect that can attain both positive and

negative values in the original problem, after the decomposition, there may exist a subdomain containing both positive and negative reachable values. On such particular subdomains, if we generate a subaction with a positive representative sample, the subaction can only increase the value of the affected variable in the abstraction problem, while the original action can increase or decrease the affected variable. As a result, all the conditions which require the affected variable to be less than its initial value would be possibly made unreachable. Analogously, if we generate a subaction with a negative representative sample, in the abstraction problem such subaction can only decrease the affected variable, and all the conditions that require the affected variables to be larger than its initial value would be possibly made unreachable.

For example, consider the same example in Section 3.3.2 with the same initial values and the same goal. If the domain for the effect $e : y \mathrel{+}= z$ is $(-2, 4)$, then, we can generate piecewise subactions as below:

$$\begin{cases} sub(a, e, 2)_1 = \langle \{\langle x, >, 0\rangle, \langle z, >, -2\rangle, \langle -z, \geq, 1\rangle\}, \{\langle y, +=, -1\rangle\}\rangle, \\ sub(a, e, 2)_2 = \langle \{\langle x, >, 0\rangle, \langle z, >, -1\rangle, \langle -z, >, -4\rangle\}, \{\langle y, +=, -0.5\rangle\}\rangle, \end{cases}$$

Here, we use the constant -0.5 as the representative sample for the subdomain $[-1, 2)$, which cannot capture the behaviour of increasing y by a positive value when z evaluates to a positive value. As a result, though $sub(a, e, 2)_2$ is applicable initially, its execution(s) cannot make the condition true. Therefore, the goal becomes unreachable.

It is worth mentioning that it is possible that such conditions may remain reachable if there are other actions in the domain affecting the same variable. However, the existence of possible unsafe cases above is sufficient to show that the heuristics derived are not safe-pruning.

3.4 Principles of Abstraction

3.4.1 A Sufficient Condition on Pruning-safeness

Inspired by the examples above, we propose a sufficient condition in Theorem 3.1 to ensure that the abstraction-based heuristics derived from our approach are *safe-pruning*.

Theorem 3.1. *$h_{abs}(s, C)$ is safe-pruning if $\forall a \in A$, $\forall e \in eff_{ncl}(a)$, the following conditions hold:*

1. *$\forall s \in S^R(s_0) : eval(rhs(e), s) \in D(e)$, where $S^R(s_0)$ is the reachable closure of s_0.*
2. *$\forall d(e)_i \in D(e) : lb(d(e)_i) \cdot ub(d(e)_i) \geq 0$.*

Some explanation of this theorem is as follows. Condition (1) requires that each domain should contain valuations of the right-hand side in all the reachable states. This is to ensure that given any reachable state $s \in S^R(s_0)$, each variable that can be changed by an applicable action in the original problem can also be changed by

an applicable subaction in the abstraction problem; thus, avoiding the domain incompleteness issue. Condition (2) helps to avoid subdomains with both positive and negative values. The implication is that the subaction activated, given a state, has a representative sample sharing the same sign as that of the evaluation of the right-hand side of the effect. This is to ensure that the original action and the subaction can affect the variable towards the same direction, thus, avoiding the asymptotic behaviours distortion issue.

The proof of the Theorem 3.1 follows the idea that if possible achievers of any conditions can be preserved in the abstraction problem, then h_{abs} is safe-pruning. The "preservation" should be understood in the sense that given a state s, if a condition has an applicable achiever in Π, it also has an achiever in Π_{desub}, whose preconditions can be satisfied. Pruning-safeness can be guaranteed in this way because the set of conditions achievable in Π is now a subset of that in Π_{desub}, namely, any condition achievable in Π is also achievable in Π_{desub}. Before showing proof for the theorem, we introduce some definition and helpful lemmas.

For brevity, we consider effects whose operator is "increase" in the following proof. Note that this does not influence the correctness as any decrease effect can be obtained from an equivalent increase effect by adding a minus sign before its right-hand side.

First, we make it clear that whether applying an action can make a condition easier or harder to achieve by introducing the concept of *effect contribution*.

Definition 3.13 (*Effect Contribution On Condition*)**.** [6] *Given a state s, a numeric condition $c = \langle \xi, \trianglerighteq, 0 \rangle$, $s \not\models c$, and a numeric effect e, the contribution of e to c in s, denoted as $contri(e, c, s)$, is defined as:*

$$contri(e, c, s) = w_{lhs(e),\xi} \cdot eval(rhs(e), s) \cdot sgn(op(e)) \tag{3.2}$$

where $w_{lhs(e),\xi}$ is the coefficient of $lhs(e)$ in ξ; $eval(rhs(e), s)$ is the evaluation of $rhs(e)$ in s; $op(e)$ is the operator of the effect, $sgn(op(e))$ is a signum function of the effect operator:

$$sgn(op(e)) = \begin{cases} 1, & \text{if } op(e) \text{ is } +=, \\ -1, & \text{if } op(e) \text{ is } -= . \end{cases} \tag{3.3}$$

We say e has *positive contribution* to c in s if $contri(e, c, s) > 0$; e has *negative contribution* to c in s if $contri(e, c, s) < 0$. For brevity, we would focus on the effects whose operator is "increase" in the following discussions. Note that this does not influence the correctness of our proof as any decrease effect can be obtained from an equivalent increase effect by adding a minus sign before its right-hand side.

The contribution of an action a to the condition c in s is defined as the summation of contributions of all the numeric effects affecting the variables in ξ, as shown in Eq.

[6] The similar mechanism was used in the implementation of the the hybrid subgoaling heuristics [1]. However, it was originally implemented on the simple numeric cases only. We extend this concept to adapt to non-constant linear effects, where the actual influence of an effect needs to be evaluated on states.

3.4,

$$contri(a,c,s) = \sum_{\substack{e \in eff_{num}(a),\\ lhs(e) \in vars(\xi)}} contri(e,c,s) \tag{3.4}$$

After introducing the definition of effect contribution, we propose a lemma claiming that for an action with constant effects only, contributing positively to a condition implies that the action is a possible achiever of the condition.

Lemma 3.1. *Let s be a state in Π, c be a condition $\langle \xi, \trianglerighteq, 0 \rangle$, where ξ is $\sum_{v \in V} w_{v,c} v + w_c$, a be an action with constant effects only, i.e. $\forall e \in eff_{num}(a) : vars(rhs(e)) = \emptyset$. a is a possible achiever of c if and only if $contri(a,c,s) > 0$.*

Proof: (sufficiency) if $contri(a,c,s) > 0$, applying a infinitely many times makes ξ evaluates to $+\infty$. As $\trianglerighteq \in \{>, \geq\}$, c eventually holds.

(necessity) if $contri(a,c,s) < 0$, applying a can only decrease the evaluation of ξ. As $\trianglerighteq \in \{>, \geq\}$, c will never hold because of the execution of a. □

Then, we propose a lemma claiming that if in the original problem there exists an applicable action, which has a non-constant linear effect contributing positively to a condition, then in the abstraction problem, such a condition has an achiever whose preconditions can be satisfied.

Lemma 3.2. *Let s be a state in Π, c be a condition $\langle \xi, \trianglerighteq, 0 \rangle$, where ξ is $\sum_{v \in V} w_{v,c} v + w_c$, if*

1. *$\exists a \in A$, $\exists e \in eff_{ncl}(a) : contri(e,c,s) > 0$, $s \models pre(a)$;*
2. *condition (1), (2) of Theorem 3.1 hold,*

then the following statements hold, in Π_{desub}:

1. *$\exists a' \in \Omega : h_{abs}(s, pre(a')) < \infty$;*
2. *a' is a possible achiever of c.*

Proof: Each piecewise subaction has a *single constant* numeric effect. Therefore, all the subactions with effects, which have a positive contribution on c are possible achievers. First, we show that the preconditions of at least one of such subactions can be satisfied.

Due to condition (1) in Theorem 3.1, we know that $\exists d(e)_i \in D(e)$, such that $lb(d(e)_i) \leq eval(rhs(e), s) \leq ub(d(e)_i)$, namely, there exists such a piecewise effect whose indirect preconditions are already satisfied in s. We denote the piecewise effect induced by $d(e)_i$ as e', and the corresponding piecewise subaction as a'. Combined with $s \models pre(a)$, it can be shown that $s \models pre(a')$, which implies $h_{abs}(s, pre(a')) < \infty$.

Then we show such an applicable subaction can contribute positively to c.

Due to the condition (2) in Theorem 3.1, $rep(d(e)_i) = eval(rhs(e'), s)$ has the same sign as that of $eval(rhs(e), s)$, and considering that $w_{lhs(e),\xi}$ remains the same in Π_{desub}, therefore $contri(e', c, s) \cdot contri(e, c, s) > 0$. Combined with $contri(e, c, s) > 0$, we know $contri(e', c, s) > 0$.

Since each subaction takes a constant effect, a' contributes positively to c; from Lemma 3.1, we know a' is a possible achiever of c.

By now, we have proved Lemma 3.2. □

Different from the constant effect, a state-dependent effect can have a negative contribution to a condition, since the value of its right-hand side may be updated through the executions, the action with such effect can be a possible achiever of the condition. For example, consider the action a with effect e, which is $x \mathrel{+}= -3x$, and initially $x = 1$. The goal condition c is $x \geq 3$. Initially, the contribution of e on c is -3, which is negative. However, executing a twice in a row updates x to 4 and thus, satisfies the goal condition.

This example reveals that unlike in the simple numeric case, where an action has a positive contribution to a condition is both sufficient and necessary to say that it is a possible achiever of the condition; in the linearly-affected condition case, a positive contribution is only sufficient but not necessary for achieving a condition. As a result, if we want to prove that possible achievers can be preserved in the abstraction problem, we need to scrutinise not only actions that have a positive contribution to a condition, but also those that have negative contribution initially but eventually are able to achieve the condition.

Now, we proceed to show that achievers can be preserved in the abstraction problem in Lemma 3.3.

Lemma 3.3. *Let s be a state in Π, c be a condition $\langle \xi, \trianglerighteq, 0 \rangle$, where ξ is $\sum_{v \in V} w_{v,c} v + w_c$, if*

1. *$\exists a \in A$, a is a possible achiever of c; ;*
2. *condition (1), (2) of Theorem 3.1 hold,*

then, the following statements hold, in Π_{desub}:

1. *$\exists a' \in \Omega : a'$ is a possible achiever of c;*
2. *$h_{abs}(s, pre(a')) < \infty$.*

Proof Sketch: Given a condition c in Π, if there is an achiever a, $s \models pre(a)$. According to the definition of possible achiever (c.f. Section 2.5.6), we can denote the action sequence by applying a for multiple times as $\pi = \langle a, a, ..., a \rangle$, $|\pi| = m$, where m is the *minimal* number of executions of applying a from s to make c hold, and the sequence of states transits to by executing π is $tr(\pi) = \langle s_1, s_2, ..., s_{m-1}, s_m \rangle$. Since the last execution of a must have a positive contribution to c, we know that $contri(a, c, s_{m-1}) > 0$. We denote the aggregated contribution of all the constant effects of action a on s to c as $contri_{const}(a, c, s)$. $contri_{const}(a, c, s) = \sum_{e \in eff_{num}(a), e \notin eff_{ncl}(a), lhs(e) \in vars(\xi)} contri(e, c, s)$

Case 1: $contri_{const}(a, c, s_{m-1}) > 0$. Easy to know $contri_{const}(a, c, s) > 0$ as effects here are all constant effects. Because in Π_{desub}, all the constant numeric effects are preserved in the constant subaction $sub_{const}(a)$. As a result of Lemma 3.1, we can show $contri(sub_{const}(a), c, s) > 0$. Therefore, $sub_{const}(a)$ is a possible achiever of c. Since $pre(sub_{const}(a)) = pre(a)$, $s \models pre(a)$, therefore $s \models pre(sub_{const}(a))$, which implies that $h_{abs}(s, pre(sub_{const}(a))) < \infty$.

Case 2: $contri_{const}(a, c, s_{m-1}) \leq 0$. Here, the constant subaction is no longer a possible achiever to the condition, hence, we need to find a piecewise subaction instead. According to Eq. 3.4, we know $\exists e, e \in eff_{ncl}(a) : contri(e, c, s_{m-1}) > 0$. We denote $e = \langle lhs(e), +=, rhs(e) \rangle$, where $rhs(e) = \sum_{v \in V} w_{v,c} v + w_c$.

For convenience, we rewrite $lhs(e)$ as x, $eval(lhs(e), s)$ as x_0, $eval(lhs(e), s_i)$ as x_i. Since we are considering LSF actions only, executing action a for any times will not affect the values of variables in $rhs(e)$ except for x. Therefore, we can simply write $rhs(e) = w_{x,a,x} x + b$, where $b = eval(\sum_{v \in V, v \neq x} w_{v,a,x} v + w_{a,x}, s)$.

Now, we can compute the general forms of x_i and $eval(rhs(e), s_i)$.

$$x_{i+1} - x_i = w_{x,a,x} x_i + b$$
$$x_{i+1} + \frac{b}{w_{x,a,x}} = (w_{x,a,x} + 1)(x_i + \frac{b}{w_{x,a,x}})$$

Combined with x_0, we know $x_i + \frac{b}{w_{x,a,x}} = (x_0 + \frac{b}{w_{x,a,x}})(w_{x,a,x} + 1)^i$. Thus, we can derive $x_i = (x_0 + \frac{b}{w_{x,a,x}})(w_{x,a,x} + 1)^i - \frac{b}{w_{x,a,x}}$, and $eval(rhs(e), s_i) = (w_{x,a,x} x_0 + b)(w_{x,a,x} + 1)^i$.

From the general form of $eval(rhs(e), s_i)$, we can see that the sign of $eval(rhs(e), s_i)$ can change through executions of a depending on the value of $w_{x,a,x}$, which leads to changes on the effect contribution to conditions. Now, we exhaust the cases to show Lemma 3.3 holds for all $w_{x,a,x}$.

If $w_{x,a,x} \geq -1$, the sign of $eval(rhs(e), s_i)$ will not change[7] through executions of a. In this case, since we know $contri(e, c, s_{m-1}) > 0$, we can show $contri(e, c, s) > 0$. Combined with $s \models pre(a)$, from Lemma 3.2, Lemma 3.3 can be proved for this case.

If $w_{x,a,x} < -1$ and m is even. Same as the case where $w_{x,a,x} \geq -1$, $contri(e, c, s) > 0$ in this case. Lemma 3.3 can be also shown in the same way.

If $w_{x,a,x} < -1$, m is odd, and $contri(e, c, s) > 0$. Same as the case where $w_{x,a,x} \geq -1$.

If $w_{x,a,x} < -1$, m is odd, and $contri(e, c, s) < 0$.

From the general form of $eval(rhs(e), s_i)$, we know $eval(rhs(e), s) \cdot eval(rhs(e), s_1) < 0$ and therefore, $contri(e, c, s) \cdot contri(e, c, s_1) < 0$. Combined with $contri(e, c, s) < 0$, we know $contri(e, c, s_1) > 0$, which means in the domain of e, there exists at least one value that has an inverse sign as $eval(lhs(e), s)$. This is critical because if the subaction induced on the subdomain containing such value can be made applicable from s, from Lemma 3.2, Lemma 3.3 can be proved. We denote such subaction as β, and the subdomain as $d(e)_\beta$. In the following, we prove that β can be made applicable by proving that $pre(\beta)$ can be satisfied from s.

The first part of $pre(\beta)$ is $pre(a)$, which is known to be satisfied in s.

The second part of $pre(\beta)$ is the indirect precondition induced by $lb(d(e)_\beta)$ and $ub(d(e)_\beta)$. First, we show what the indirect precondition is.

First, consider the case where $eval(rhs(e), s) < 0$. Since $contri(e, c, s) \cdot contri(e, c, s_1) < 0$, we can show that $eval(rhs(e), s_1) > 0$. Due to the condition (2) in Theorem 3.1, we know $lb(d(e)_\beta) > 0$ and $ub(d(e)_\beta) > 0$. The indirect precondition of the subaction

[7] Though this is not conflicting with our following discussion, we point it out that if $w_{x,a,x} = -1$, for any $i \leq 1$, $contri(e, c, s_i) = 0$ and thus, cannot make c hold.

includes $rhs(e) > lb(d(e)_\beta)$ and $rhs(e) < ub(d(e)_\beta)$. We denote them as c_{lb} and c_{ub} in the following discussion.

Next, we show that the indirect precondition c_{lb} and c_{ub} can be satisfied in the abstraction problem from s.

c_{ub} is satisfied in s because $eval(rhs(e), s) < 0$ and $ub(d(e)_\beta) > 0$.

c_{lb} is not satisfied in s initially. In the following, we show it can, however, be satisfied by applying the subaction induced on the subdomain containing $eval(rhs(e), s)$. We denote such subaction as α and the subdomain as $d(e)_\alpha$.

Since $eval(rhs(e), s) < 0$, because of the condition (2) of Theorem 3.1, we know $rep(d(e)_\alpha) < 0$. Denote the piecewise effect induced by $d(e)_\alpha$ as pwe_α, then we know $eval(rhs(pwe_\alpha), s) = rep(d(e)_\alpha) < 0$. As $lhs(pwe_\alpha) = x$ and $w_{x,a,x} < 0$, according to 3.2, we know $contri(pwe_\alpha, c_{lb}, s) > 0$. Since α has only one effect, according to Eq. 3.4, we can show $contri(\alpha, c_{lb}, s) > 0$. From Lemma 3.1, we know α is a possible achiever of c_{lb}. That is to say, under the assumption of hybrid subgoaling, c_{lb} can be achieved by α.

The case where $eval(rhs(e), s) > 0$ can be proved analogously.

Now, we complete the proof for Lemma 3.3. Lemma 3.3 combined with the fact that h^{add}_{hbd} is safe-pruning, we are able to show the heuristics derived from abstraction problems are also safe-pruning. Thus, Theorem 3.1 is proved. □

3.5 h^{add}_{abs} - An Inadmissible Heuristic

The abstraction by decomposition and subactioning defines a framework for deriving abstraction problems, and the principles proposed in Theorem 3.1 guarantees the pruning-safeness of hybrid subgoaling heuristics acquired on the abstraction problems derived. However, there are strategies that need to be determined to obtain a specific abstraction from a planning problem, which we have not mentioned. In this section, we explain the strategies we are adopting to derive an inadmissible heuristics h^{add}_{abs}.

3.5.1 Estimation of Domains by AIBR

The domain of an effect is a set of values the right-hand side can evaluate to. As the effects in the linearly-affected case are state-dependent in general, determining domains for each effect precisely is equivalent to finding all the reachable states, which is at least as hard as solving the planning problem. Therefore, some procedures are required to provide an estimation on domains.

One observation is that the relaxed goal state in the AIBR counting procedure (c.f. Algorithm 2), assigns an interval to each variable. Such intervals contain values that each variable can obtain. We use these intervals as estimations for the possibly attainable values for each variable in the problem. Then, we substitute variables in each right-hand side of the effects using their interval estimations and finally, apply interval operations to compute the estimation of the effect domain.

It is worth mentioning that domain estimations obtained from the relaxed goal state in AIBR counting can contain values which are not attainable in the original problem, and on the other hand, some attainable values may not be captured in the domain. We use the following example to illustrate this.

First, recall that AIBR counting simply exhaustively applies all the applicable actions to each relaxed state until the goal is reached. Now, consider a problem, where we have an action a_{inc} that can increase the value of a variable by 1 and an action a_{dec} that can decrease the value of a variable by 1 on the precondition that after the decrement, the value of the variable cannot be less than 0. Initially, $x = 1$ and the goal is $x > 2$.

Initially, both a_{inc} and a_{dec} are applicable, thus, they are applied on the initial state, which results in s_1^+, where $val^+(x, s_1^+) = [0, 2]$. As the goal is not satisfied yet, the procedure continues. In s_1^+, a_{inc} is applicable. a_{dec} is also applicable as any value between 1 and 2 in the interval representing x can satisfy the precondition of a_{dec}. Hence, in s_1^+, we need to apply both actions again, which results in s_2^+, where $val^+(x, s_2^+) = [-1, 3]$. Since s_2^+ is a goal state, the procedure terminates.

In our approach, we just take $val^+(x, s_2^+)$ to estimate possibly attainable values of x, which is $[-1, 3]$. However, it is obvious that in the original problem, due to the precondition of a_{dec}, the value of x cannot go below 0, thus, -1 in $val^+(x, s_2^+)$ is not attainable. On the other hand, a_{inc} can be applied infinitely many times in the original problem and as a result, x can take any positive integer as its value, which is not captured in $val^+(x, s_2^+)$.

3.5.2 Uniform Bi-partition and Midpoint Representative Sample

It remains unclear to us what the best strategy is to partition each domain into subdomains in order to realise the best trade-off between the informedness of the heuristics and the computation overhead led to by the increased number of actions. In our approach, we simply uniformly partition each domain into two subdomains. If any subdomain contains both positive and negative values, we further partition it using 0 as a boundary to ensure that condition (2) of Theorem 3.1 holds. We take the midpoint of each subdomain as the representative sample.

To overcome the problem that some attainable values are not captured in the domain estimation, we add two extra subdomains: one is bounded by $-\infty$ and the lower bound of the domain estimation, on which we pick the lower bound of the domain estimation as the representative sample; another one is bounded by the upper bound of the domain estimation and $+\infty$, and then, we take the upper bound of the domain estimation as the representative sample for the subdomain. With these two additional subdomains, we can guarantee that condition (1) of Theorem 3.1 holds for abstraction problems obtained in our approach.

For subdomains whose boundaries are positive, we use non-strict inequality as the comparison operator for the indirect precondition induced by the upper bound and strict inequality for the one induced by the lower bound. In contrast, for subdomains whose boundaries are negative, we use strict inequality for the one induced by the

lower bound and non-strict inequality for the one induced by the upper bound. The rationale behind this is that as we are considering additive effects only, any effect whose right-hand side is evaluating to 0 will not change the value of the affected variable. Therefore, we keep the boundary on 0 always open.

3.6 h^{max}_{abs} - An Admissible Heuristic

Coming up with admissible heuristics for numeric planning is hard. To our best knowledge, there are only two existing works attempting to solve numeric planning problems optimally by using admissible heuristics, including h^{max}_{hbd} [1] as we mentioned before and h^{lma}_{hbd} [45]. However, they are both restricted to the simple numeric case due to the complications of state-dependent effects. In this work, we aim at making the first attempt at solving linearly-affected domains optimally, which results in an admissible heuristic h^{max}_{abs}.

3.6.1 Asymptotic Subaction

h^{max}_{abs} is derived by defining *asymptotic subaction*. Each asymptotic subaction models one possible asymptotic outcome of a non-constant linear effect of a. And the asymptotic outcome induces an over-estimation of the effect on variables naturally.

Definition 3.14 (Asymptotic Subaction of Action a)**.** *Each additive non-constant linear numeric effect $e \in eff_{ncl}(a)$ generates two asymptotic subaction, $sub^+(e)$ and $sub^-(e)$: the precondition of $sub^+(e)$ is $pre(a) \cup \{\langle rhs(e), >, 0\rangle\}$, and the effect is $\langle lhs(e), op(e), +\infty\rangle$; the precondition of $sub^-(e)$ is $pre(a) \cup \{\langle rhs(e), <, 0\rangle\}$, and the effect is $\langle lhs(e), op(e), +\infty\rangle$.*

From the set of actions A, we generate two asymptotic subactions for each non-constant linear effect, and generate constant subactions for constant effects. To distinguish with the notation used in h^{add}_{abs}, we denote the union of asymptotic subactions and constant subactions as Ψ. The size of Ψ is $(2|e_{ncl}|_{avg} + 1)|A|$, where $|e_{ncl}|_{avg}$ is the average number of non-constant linear effects per action.

By asymptotic subactions, we define the abstraction problem used to derive h^{max}_{abs}.

Definition 3.15 (*Abstraction Problem by Asymptotic Subaction*)**.** *Given a numeric planning domain $\Sigma = \langle V, S, A, \gamma, \delta\rangle$ and a numeric planning task $\Pi = \langle \Sigma, s_0, G\rangle$, an* abstraction problem by asymptotic subaction *is acquired by replacing A with Ψ. We denote the relaxed domain as $\Sigma_{asymp} = \langle V, S, \Psi, \gamma, \delta\rangle$ and the induced planning problem as $\Pi_{asymp} = \langle \Sigma_{asymp}, s_0, G\rangle$.*

Same as Π_{desub}, the abstraction problem by asymptotic subactions is a simple numeric condition problem, which can also be handled by the hybrid subgoaling heuristic. Given a set of conditions C and a state s, the heuristic value for satisfying C from s computed by $h^{max}_{hbd}(s, C)$ in Π_{asymp} is denoted as $h^{max}_{abs}(s, C)$.

Note that this configuration complies with Theorem 3.1 thus is pruning-safe.

3.6.2 Rationales for Admissibility

There are two intuitions backing up the admissibility of h^{max}_{abs}.

First, the implication of asymptotic subaction is that any condition which has a possible achiever in the original problem can now be achieved by one execution of a possible achiever in the abstraction problem, which is naturally an under-estimation as 1 is the minimal cost of applying any non-fractional cost action. In the recursive definition of h^{max}_{abs} (c.f. Eq. 2.12), this approximation replaces the minimisation term over action cost $\min_{a \in ach(C)} (m\gamma(a))$ by 1.

However, because the abstraction problem we acquired has more conditions than the original problem, we also need to ensure that maximisation step over subgoals is not a source of over-estimation. In the abstraction problem, any condition can be achieved by at most two executions. Consider a piecewise subaction induced by a non-constant linear effect e which is a possible achiever of a condition c. Given a state, if the applicable asymptotic subaction contributes positively to c, the cost to achieve the condition 1, which does not influence the value computed by the maximisation step. If the applicable asymptotic subaction contributes negatively to c, there are two steps to achieve c as shown in the proof of Lemma 3.3, where the executions of a subaction contribute negatively will activate the one with positive contribution. Applying this asymptotic subaction once can make the counterpart asymptotic subaction applicable, after which one execution of the asymptotic subaction with a positive contribution can make the condition hold. As the two steps are necessary in the original problem and for each step, we are using the minimal cost as the estimation. Thus, the maximisation step would not lead to over-estimation.

Our hypothesis on the admissibility of h^{max}_{abs} is backed up by the experiment results in Section 4.3.3, where our heuristics produces optimal plans on all the solved instances. A formal proof for the admissibility of h^{max}_{abs} and how to make it more informative might be good directions to follow in the future.

3.7 Discussions

3.7.1 Rationales behind Restriction on Linearly-affected Cases

Our work aims at generalising the applicability of hybrid subgoaling heuristics to linearly-affected cases. Namely, conditions in the domain and effects must be in linear form. We discuss rationales behind this restriction here.

Achievers for non-linear conditions are not well-defined. The definition for achiever is established by taking one action and one condition as inputs. However, this is because, for linear conditions, the change of the evaluation of a variable will not be diminished by other variables appearing in the right-hand side of t he effect. This is not the case for non-linear conditions though. For example, consider a non-linear condition $x^y > 2$, and initially $x = 1, y = 0$. There are two actions in the domain a_1 and a_2, which can increase the value of x and y by 1 respectively per execution. To achieve the goal, we need to apply both a_1 and a_2 at least once. While applying

both actions contributes "positively" to the goal, neither of them alone is a possible achiever under Definition 2.5.6. And determining whether applying a set of actions can achieve a condition is as hard as solving a plan existence problem without proper further relaxation.

The restriction on linear effects is due to the fact that the m-times regressor (c.f. Theorem 2.1) is defined on linear effects. Since we are using the hybrid subgoaling heuristics in a black-box manner, the regression of conditions by subactions is dependent on this definition. Since it is rather difficult to compute the closed-form of non-linear effects in general, determining non-linear achievers by Definition 2.5.6 is not possible.

3.7.2 Another Possible Pathway to Admissible Heuristics

While h^{max}_{abs} is admissible, it is not informative enough for solving even moderate problems efficiently. We are aware that there may be a possible pathway to derive a more informative yet potentially still admissible heuristics. We present this idea briefly here mainly for possible furture works.

The general idea is to be always "optimistic" about the effect. That is to say, when the right-hand side falls into a positive subdomain, we always pick the upper bound as the representative sample while picking the lower bound as the representative sample for negative subdomains. In this way, we may underestimate the number of repetitive executions but differs from h^{max}_{abs} where this underestimation is always 1. Abstraction problems are constructed by these subactions combined with asymptotic subactions. By applying h^{max}_{hbd} on this abstraction problem, we may be able to derive another admissible yet more informative heuristic for the original problem.

One difficulty of this approach arises when an action has more than one non-constant linear effects. In this situation, since each subaction has only a single effect, the positive interactions between effects in the original problems will not be captured. And ignoring such positive interactions can lead to over-estimations. One possible solution to this issue is to assign a fractional action cost to each subaction, which may be a good direction to follow in the future.

Chapter 4

Experiments

In this chapter, we evaluate our proposed abstraction-based heuristics empirically. We first describe the experiment environment including the hardware and software information in Section 4.1. Then we introduce the planning domains we use in the experiments in Section 4.2. We conduct three experiments in Section 4.3. The first one, in Section 4.3.1, compares our proposed inadmissible heuristics h^{add}_{abs} with other heuristics. In the second one, in Section 4.3.2, study in the performance of the planning systems we are using. Lastly, in Section 4.3.3, we study the efficiency of h^{max}_{abs} by comparing it with blind and goal-sensitive heuristic.

4.1 Experimental Environment

Hardware and Software Information

We give the hardware and software information of our experimental environment in Table 4.1.

CPU	Intel(R) Core(TM) i7-4790, 3.6 GHz, 8 cores
Memory	16 GB
Hard Disk	SAMSUNG PM851 256 GB SSD
Operating System	Ubuntu 16.04.3 LTS Xenial Xerus
Programming Language	Java 1.8

Table 4.1: Hardware and Software Information

Implementation

We implement the proposed abstraction-based heuristics in ENHSP (Expressive Numeric Heuristic Search Planner[1]), developed by Scala [1, 2, 15].

[1] https://bitbucket.org/enricode/expressive-numeric-heuristic-search-planner-enhsp-planner

4.2 Benchmark Planning Domains

We evaluate our proposed heuristics over four domains. All of these domains feature linear non-constant numeric effects. These domains include TPP-METRIC from IPC-5 [16]; first-order counters domain is an extended version of COUNTERS [1], including three variants FO-COUNTERS, FO-COUNTERS-INV and FO-COUNTERS-RND. We also include two domains adapted from simple numeric cases [1], which are FARMLAND-LN and SAILING-LN. For domains newly created, most of instances were automatically generated. In all, we have 331 problem instances as shown in Table 4.2. All the domains and instances are aiming at emphasising on the numeric reasoning on linear conditions. We expect our proposed heuristics provides better guidance on domains characterised by linear effects. The PDDL 2.1 description of these domains can be found in the Appendix A.

Table 4.2: Number of Instances of Domains

Problem Domain	# instances
TPP-METRIC	41
FO-COUNTERS	39
FO-COUNTERS-INV	39
FO-COUNTERS-RND	117
FARMLAND-LN	50
SAILING-LN	45
Total	**331**

TPP-METRIC

The TPP-METRIC domain is described in detail in Section 2.4.3. Usually the domain is used in satisficing tracks in the planning competitions while optimising the plan metric, which is the total cost of travelling and purchasing. However, because our abstraction-based heuristic is not cost-sensitive, which means it cannot deal with state-dependent planning metrics, we instead assign unit cost to each action to optimise the lengths of plans produced.

FO-COUNTERS

The FO-COUNTERS domain is an extended formulation of COUNTERS. There are N numeric variables $(X_1, X_2, ..., X_N)$, each of which is set to a value initially. The value of each variable can be increased or decreased by a rate $(\Delta X_1, \Delta X_2, ..., \Delta X_N)$, each rate can be increased or decreased by 1.

There are also constraints over the minimal and maximal values X_i and ΔX_i can take. While the minimal value is 0, the maximal value depends on the number of variables in the problem instance.

The goal is to set the values of the variables in ascending order ($X_1 < X_2 \wedge X_2 < X_3 \wedge ... \wedge X_{N-1} < X_N$).

There are three variations of this domain, all of which share the same domain while characterise different initial values for each variable. In FO-COUNTERS instances, all the variables are initially set to 0. In FO-COUNTERS-INV instances, variables are initially ordered inversely to the goal, that is ($X_1 > X_2 \wedge X_2 > X_3 \wedge ... \wedge X_{N-1} > X_N$). In FO-COUNTERS-RND instances, initial values for each variable are set randomly, however, not satisfying the goal. Instances scale on N from 2 to 40 while in FO-COUNTERS-RND, there are 3 random instances for each N.

FARMLAND-LN

The FARMLAND-LN domain models the problem of allocating manpower to farms. The are m farms in each domain, scaling from 2 to 10. Each farm initially has some workers and the total number of workers ranges from 100 to 1000.

Different farms have different productivity. Initially workers are usually allocated to farms with low productivity and the goal is to transfer workers between farms to satisfy a hard constraint on net benefit.

There are two options to transfer workers between adjacent farms. Firstly, workers can be transferred by a small amount per action without extra cost. Alternatively, cars can be hired to transfer more workers per transfer while leading to an overhead in cost as the consequence.

The trade-off introduced in FARMLAND-LN lies in the cost of transferring workers by cars. If too many cars are hired, such moving can be too expensive to achieve the goal while a smaller number of cars can be beneficial to find shorter plans.

SAILING-LN

The SAILING-LN domain models the problem of rescuing people in an unbounded area of ocean. The position of the boat and people to be rescued are described by their coordinates $(x, y) \in \mathbb{Q}^2$. To save a person, the boat must reach an area described by one or more linear inequalities [1].

The boat moves on a geometrical space defined by $\mathbb{Q}^2$. It can move in cardinal or ordinal directions except for the direction wind comes straight from. The speed of the boat can be increased or decreased. To save a person, the speed of the boat should be lower than a safety threshold.

Trade-offs on accelerating boat make SAILING-LN interesting. Firstly, if the speed is high enough that each move is longer than the distance between boundaries of the rescuing area, then such area becomes unreachable before any deceleration. Besides, accelerating too much implies taking more actions to decelerate before reaching the safty threshold to rescue people.

4.3 Experiment Result

4.3.1 Evaluation of h^{add}_{abs} with Other Heuristics

We compare our heuristics with the heuristic h^{FF}, which is used by Metric-FF, and heuristics produced by AIBR counting procedure (abbr. AIBR). Metric-FF is a forward-search planner developed by Hoffman [3]; made specifically for sequential numeric planning problems. The default configuration for Metric-FF uses enforced hill-climbing as the search strategies; heuristics are derived from the *relaxed planning graph*, which is a data structure encoding conditions and actions in layered representation; *helpful actions* are used to rule out actions of the search space. However, neither the local search technique nor the pruning by helpful action can ensure the completeness of the search. In order to focus on the quality of heuristics while making the comparison fair, we suppress the enforce hill-climbing and helpful actions in Metric-FF. In this experiment, we use greedy best-first search informed by different heuristics. Ties are broken preferring smaller g-values, which is the actual cost to reach each state. All heuristics are computed on a per state basis. Timeout for each instance is set to 1,800 seconds. The overall result is shown in Table 4.3.

From the result, it seems our approach is superior to h^{FF} and AIBR over all the domains evaluated. Indeed, our approach solves 137 more instances than Metric-FF informed by h^{FF} and 51 more than ENHSP informed by AIBR. On problems solved by both using AIBR and h^{add}_{abs} heuristics, using h^{add}_{abs} is usually one or sometimes two orders of magnitude faster than using AIBR and explores much smaller portion of search space.

At the same time there is a small but somewhat consistent degradation in the quality of the plans found. In cases where optimisation is required, the suboptimal plans found by h^{add}_{abs} can be either used as a candidate solution before attempting to find other optimal ones in neighbour, or as a fallback whenever optimal planning does not terminate in time. This degradation is possibly due that h^{add}_{hbd} is summing up cost for subgoals, which usually leads to over-estimations.

Table 4.3: Comparison among h^{FF}, AIBR and h^{add}_{abs} on satisficing planners. Entries for TPP-metric domain are calculated on instances solved by all systems (if appropriate). Other entries are calculated on instances solved by AIBR and h^{add}_{abs} due to limited number of instances solved by Metric-FF. Best performers are highlighted in bold.

	Coverage			CPU-Time (s)			Exp. Nodes			Eval. Nodes			Plan Lengths		
	h^{FF}	AIBR	h^{add}_{abs}	h^{FF}	AIBR	h^{add}_{abs}	h^{FF}	AIBR	h^{add}_{abs}	h^{FF}	AIBR	h^{add}_{abs}	h^{FF}	AIBR	h^{add}_{abs}
TPP-METRIC (41)	7	7	**31**	**0**	0.4	0.9	**8.8**	18	11.9	51.2	35.8	**35.2**	7.8	**7**	8.8
FO-COUNTERS (39)	1	**8**	7	NA	**2.2**	91.4	NA	**1.1k**	21k	NA	**13k**	277k	NA	**17**	19
FO-COUNTERS-INV (39)	0	6	**10**	NA	271.1	**1.25**	NA	71k	**632**	NA	730k	**7.4k**	NA	**24**	29
FO-COUNTERS-RND (117)	1	22	**30**	NA	43.9	**2.5**	NA	11k	**2.7k**	NA	170k	**20k**	NA	23.6	**17**
FARMLAND-LN (50)	0	**50**	**50**	NA	2.4	**0.7**	NA	638.8	**62.4**	NA	8.8k	**1k**	NA	**26.8**	60.3
SAILING-LN (45)	0	2	**18**	NA	369	**1.7**	NA	1.6M	**2.7k**	NA	3.1M	**11k**	NA	**61**	73

In Fig. 4.1, we compare heuristics in terms of number of solved problems within a CPU-time limit ranging from 10 milliseconds to 1,800 seconds. When the CPU-

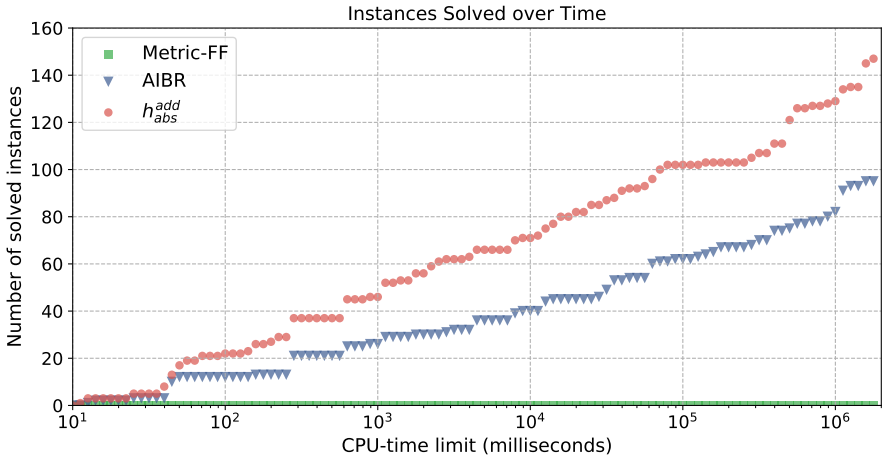
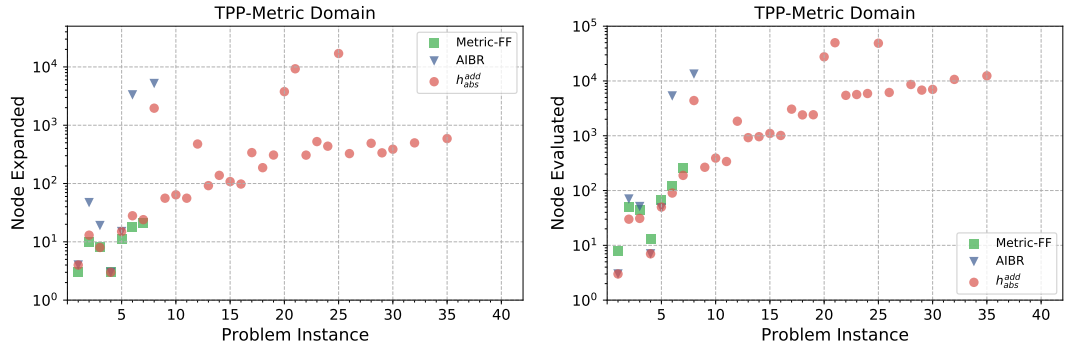


Figure 4.1: Number of instances solved by Metric-FF informed by h^{FF}, ENHSP informed by AIBR and h^{add}_{abs} with respect to an increasing CPU-time limit (logarithmic scale) across domains. The coverages for systems using h^{FF}, AIBR and h^{add}_{abs} are 2.7% (9/331), 28.7% (95/331), 44% (146/331).

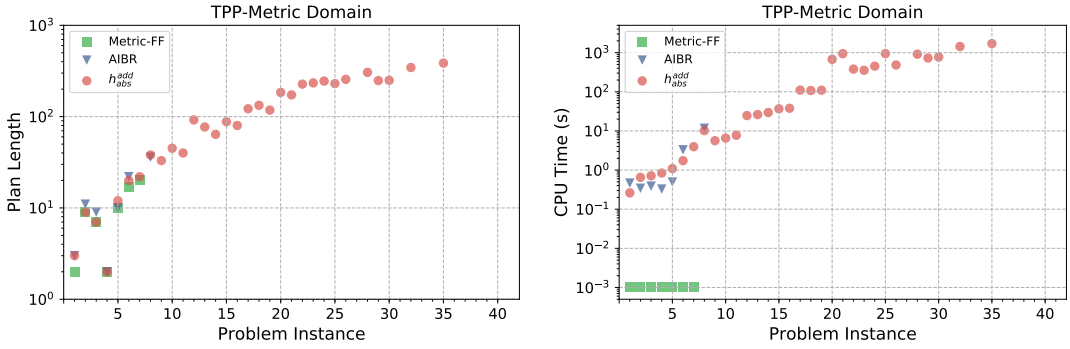
time limit is very low (less than 10 milliseconds), Metric-FF manages to solve most problems than any other heuristics; for CPU-time limits higher than 10 milliseconds h^{add}_{abs} solves more problems than the other heuristics. When the time limit is low, due to the efficient implementation of Metric-FF, it can solve small instances even without a good guidance by heuristics. However, with more time for search, ENHSP with h^{add}_{abs} or AIBR performs better.

Fig. 4.2 to Fig. 4.7 present the results on each domain respectively.



(a) Nodes expanded on TPP-METRIC instances

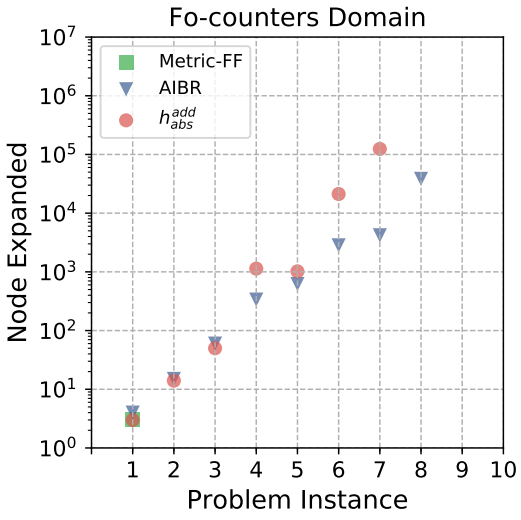
(b) Nodes evaluated on TPP-METRIC instances



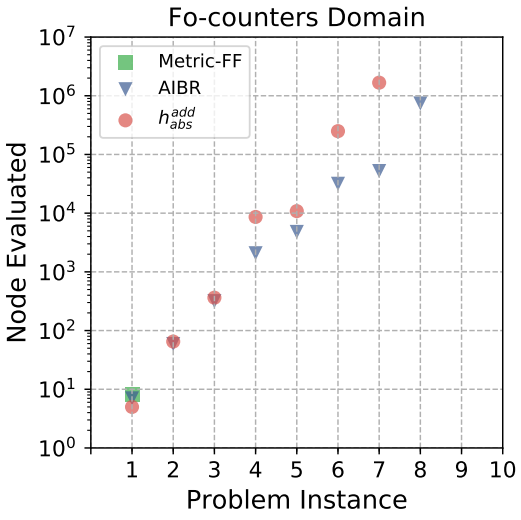
(c) Plan lengths of TPP-METRIC instances

(d) CPU-time on TPP-METRIC instances

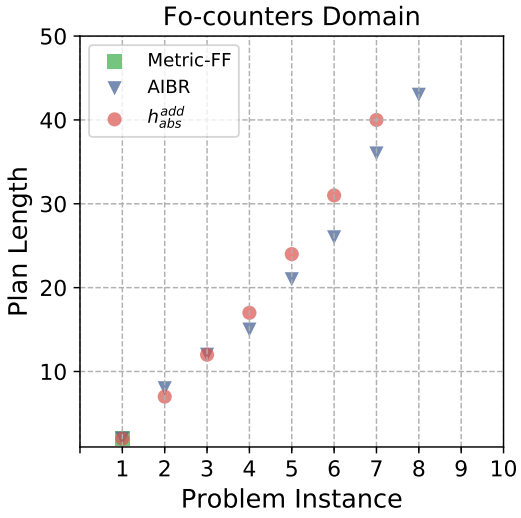
Figure 4.2: Comparison on the TPP-METRIC domain: The instances are organised in ascending order with respect to the total number of markets and goods. While systems using h^{FF} and AIBR can solve a limited number (7) of instances, h^{add}_{abs} manages to solve 31 out of 41 instances. Among instances solved by all three systems, h^{add}_{abs} expands and evaluates as many nodes as those by Metric-FF using h^{FF} and orders of magnitude fewer than those by AIBR. h^{add}_{abs} produces plans with reasonable lengths with respect to other heuristics. The running-time of AIBR and h^{add}_{abs} is not comparable with Metric-FF. This is partially due to the difference in performance of the underlying planners.



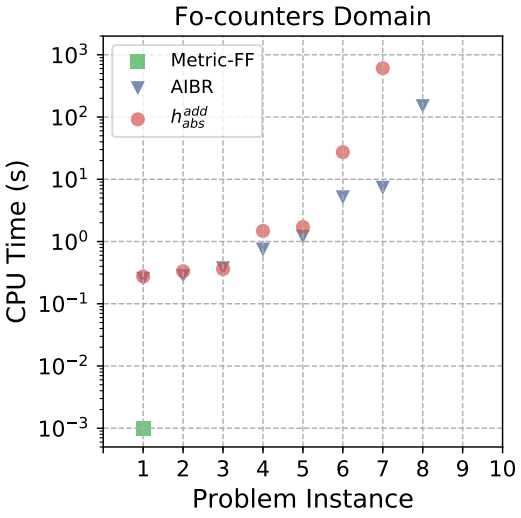
(a) Nodes expanded on FO-COUNTERS instances



(b) Nodes evaluated on FO-COUNTERS instances

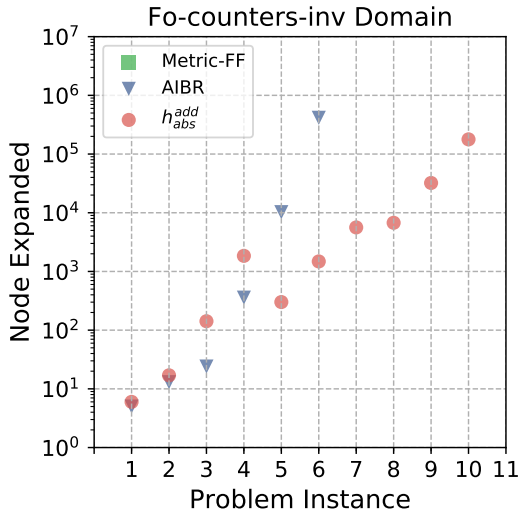


(c) Plan lengths of FO-COUNTERS instances

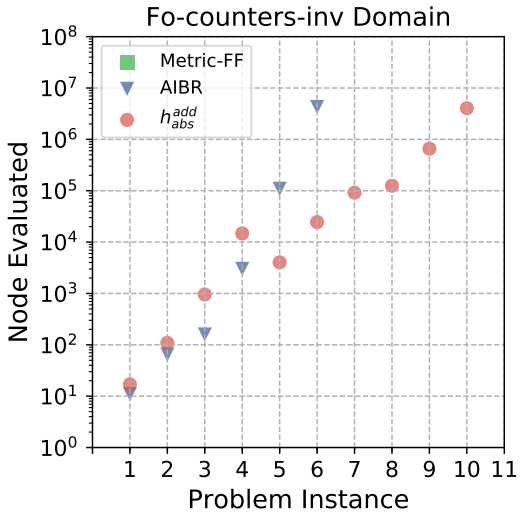


(d) CPU-time on FO-COUNTERS instances

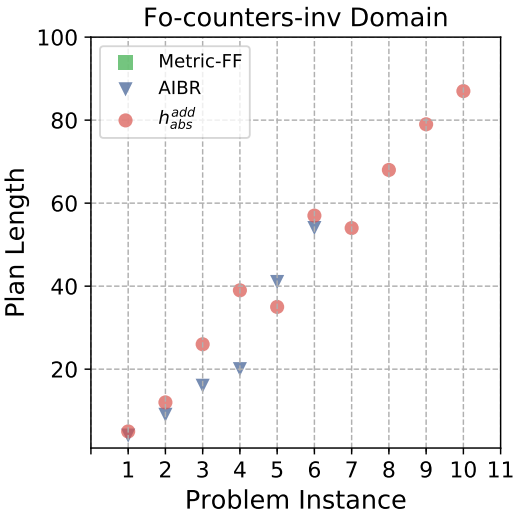
Figure 4.3: Comparison on the FO-COUNTERS domain: Instances are organised in ascending order with respect to the number of variables. Metric-FF using h^{FF} can solve only the easiest instance while AIBR solves most (8) and h^{add}_{abs} solves 7. Among instances solved by AIBR and h^{add}_{abs}, AIBR is consistently better than h^{add}_{abs} with respect to the node expansion, evaluation, plan lengths and CPU-time. Though on small-scale (1-3) instances, there is a small improvement observed by h^{add}_{abs} compared with AIBR. Metric-FF, though solves only one instance, performs orders faster than ENHSP.



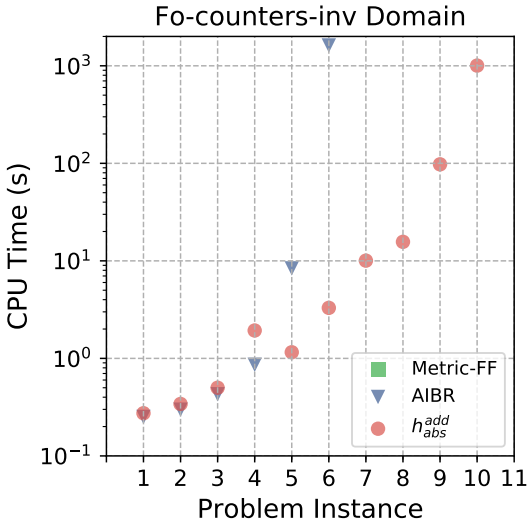
(a) Nodes expanded on FO-COUNTERS-INV instances



(b) Nodes evaluated on FO-COUNTERS-INV instances

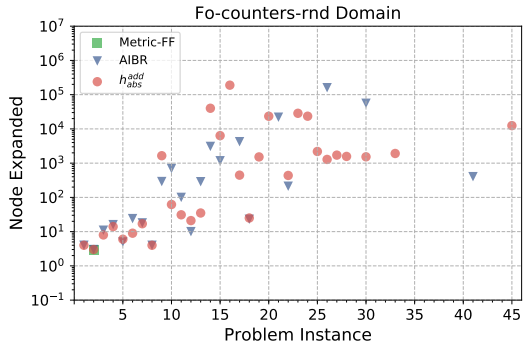


(c) Plan lengths of FO-COUNTERS-INV instances

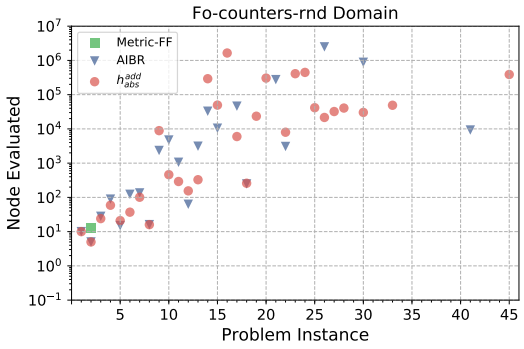


(d) CPU-time on FO-COUNTERS-INV instances

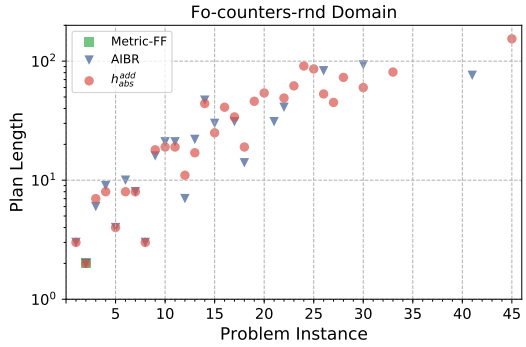
Figure 4.4: Comparison on FO-COUNTERS-INV domain: Instances are organised in ascending order with respect to the number of variables. Metric-FF using h^{FF} cannot solve any instance on the FO-COUNTERS-INV domain while AIBR solves 6 and h^{add}_{abs} solves 10. Both the size of the search space explored and the running time for AIBR and h^{add}_{abs} increases exponentially fast. However, AIBR scales worse than h^{add}_{abs} and soon times out for a substantial number of cases. On problems solved, h^{add}_{abs} provides a little bit degraded but reasonable plan quality compared with AIBR.



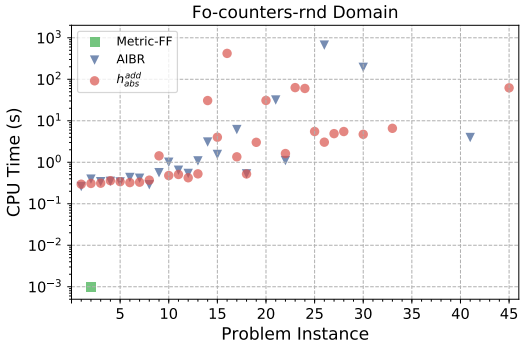
(a) Nodes expanded on FO-COUNTERS-RND instances



(b) Nodes evaluated on FO-COUNTERS-RND instances

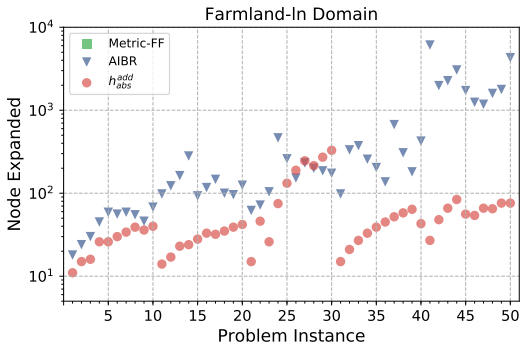


(c) Plan lengths of FO-COUNTERS-RND instances

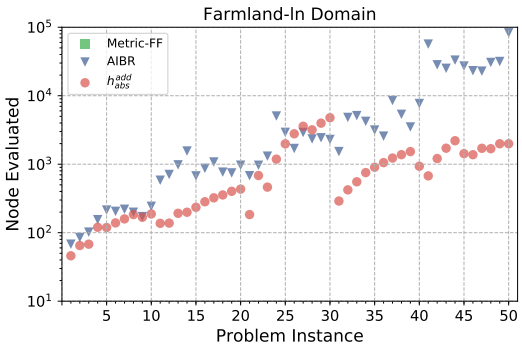


(d) CPU-time on FO-COUNTERS-RND instances

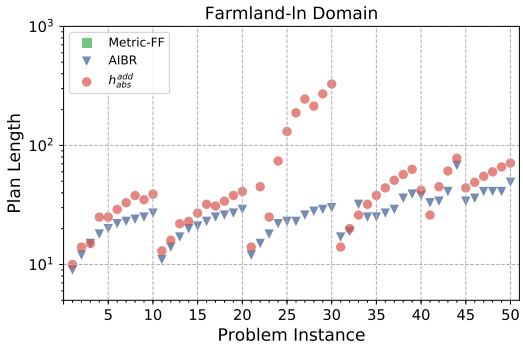
Figure 4.5: Comparison on the FO-COUNTERS-RND domain: Instances are organised in ascending order with respect to the number of variables N. For each N, there are three random instances differentiated by the initial values of each variable. Again, Metric-FF using h^{FF} solves one instance quickly while getting trapped for all of the others. h^{add}_{abs} manages to solve 30 instances while AIBR solves 22. On problems solved by both h^{add}_{abs} and AIBR, h^{add}_{abs} usually expands and evaluates fewer nodes. As the initial values are set randomly for each variable, the results on FO-COUNTERS-RND may be a better indicator to show that h^{add}_{abs} captures useful information in the first-order counter domain and is less susceptible by the initial settings. h^{add}_{abs} on average solves instances faster than AIBR with only a few exceptions observed.



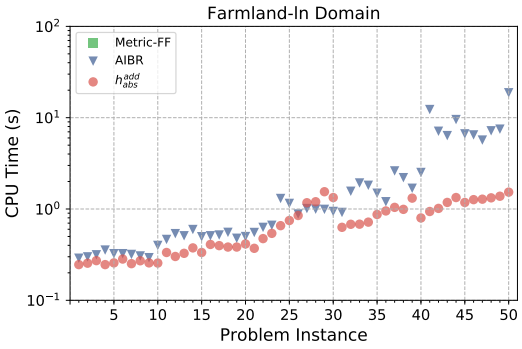
(a) Nodes expanded on FARMLAND-LN instances



(b) Nodes evaluated on FARMLAND-LN instances

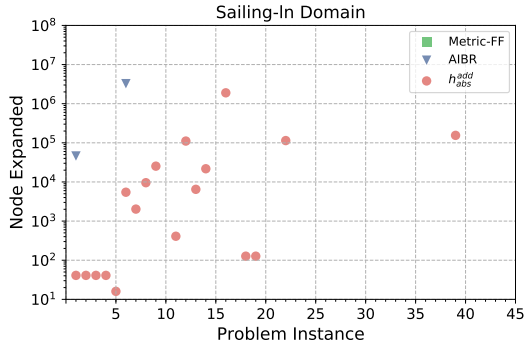


(c) Plan lengths of FARMLAND-LN instances

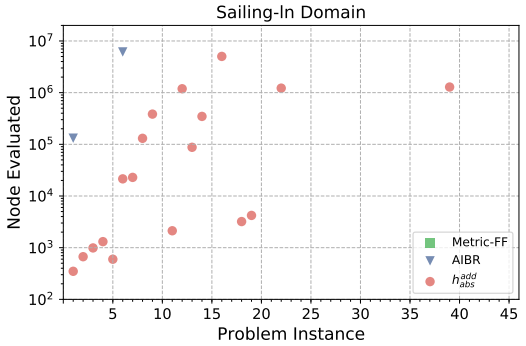


(d) CPU-time on FARMLAND-LN instances

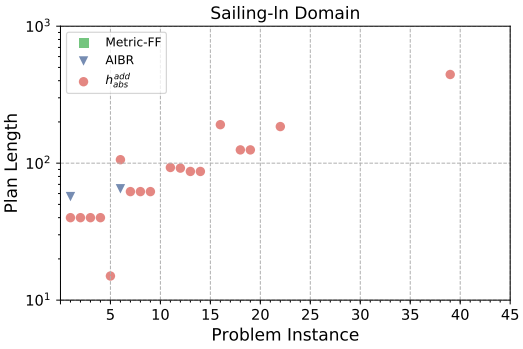
Figure 4.6: Comparison on the FARMLAND-LN Domain: There are two scaling factors in this domain: the number of farms and the total number of workers. The instances are sorted using the number of farms as the primary key and the total number of workers as the secondary key. Metric-FF using h^{FF} cannot solve any instance while both AIBR and h^{add}_{abs} solve all the instances. AIBR expands and evaluates much more nodes in general, especially when the number of farms becomes large. Again, h^{add}_{abs} produces plans with lower quality on nearly all instances with only a few exceptions observed though terminating much faster than AIBR.



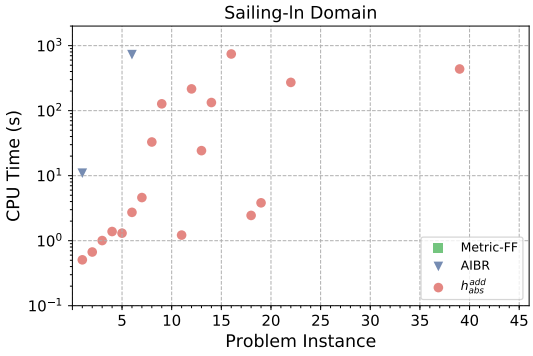
(a) Nodes expanded on SAILING-LN instances



(b) Nodes evaluated on SAILING-LN instances



(c) Plan lengths of SAILING-LN instances



(d) CPU-time on SAILING-LN instances

Figure 4.7: Comparison on the SAILING-LN domain: Metric-FF using h^{FF} cannot solve any instance on this domain. AIBR solves only 2 instances while h^{add}_{abs} solves 18 instances. On the only two problems solved by AIBR, h^{add}_{abs} expands and evaluates 2 or 3 orders of magnitude fewer nodes. AIBR is very susceptible to the problem size and times out for most of the instances in this domain.

In summary, Metric-FF informed by h^{FF} without the local search technique and helpful action is rather weak in providing search guidance and solves a very limited number of instances. h^{add}_{abs} solves most instances and seems to be more informative than AIBR in most cases. This allows h^{add}_{abs} to be scaled up to large instances, which can be reflected on the TPP-METRIC and SAILING-LN domains especially.

4.3.2 Evaluation of Planning Systems

Apart from the comparison between heuristics, it would be of our interest to evaluate performances of different planning systems. In our work, since as shown in Section 4.3.1, AIBR is in general not comparable with h^{add}_{abs}, we evaluate the performance of ENHSP informed by h^{add}_{abs} using greedy best-first search. For Metric-FF, we activate the enforced-hill climbing as the search strategy and allow the usage of helpful actions to prune the space, which is the standard configuration of Metric-FF in the planning competition. Timeout for solving each instance is 1,800 seconds. The result for are

Table 4.4: Performance of standard Metric-FF on the experiment domains. Entries are calculated on instances solved by both ENHSP with h^{add}_{abs} and standard Metric-FF. Best performers are highlighted in bold.

	Coverage		CPU-Time (s)		Eval. Nodes		Plan. Length	
	Metric-FF	ENHSP	Metric-FF	ENHSP	Metric-FF	ENHSP	Metric-FF	ENHSP
TPP-METRIC (41)	**41**	31	**26**	332.8	8.5k	**6.9k**	180.9	**132.8**
FO-COUNTERS (39)	**8**	7	**0**	91.4	**7.8k**	277k	21	**19**
FO-COUNTERS-INV (39)	7	**10**	**0.5**	2.5	**13k**	19k	**31.4**	32.6
FO-COUNTERS-RND (117)	23	**30**	23.9	**3.8**	63k	**34k**	**20.8**	21.5
FARMLAND-LN (50)	36	**50**	50.1	**0.57**	21.8k	**652**	**31.9**	41.5
SAILING-LN (45)	6	**18**	**0.05**	22	**2.1k**	64k	50.2	**39.5**

shown in Table 4.4.

On the experimented domains, ENHSP informed by h^{add}_{abs} solves more problem instances (146/331, 44.1%) than the standard Metric-FF (121/331, 36.6%) within the time restriction. Generally, ENHSP with h^{add}_{abs} is slower than Metric-FF. However, with the help of the better informative heuristics, ENHSP is nearly as good as the standard Metric-FF on node evaluation and produces reasonable, sometimes even better quality plans. This may be due to the standard Metric-FF is using the local search technique, which focuses more on the satisficing and does not consider much about optimising the plan quality.

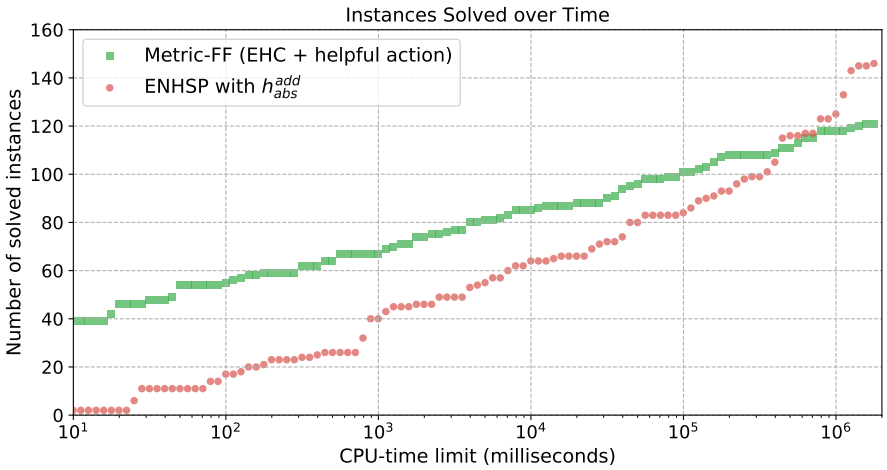


Figure 4.8: Number of instances solved by the standard Metric-FF and ENHSP informed by h^{add}_{abs} with respect to an increasing CPU-time limit (logarithmic scale) across domains. The coverages for the standard Metric-FF and ENHSP and are 36.6% (121/331) and 44.1% (146/331).

In Fig.4.8, we compare the two systems in terms of the number of solved problems within a CPU-time limit. When the time restriction is lower than ∼500 sec-

onds, Metric-FF solves more problem instances. This is probably because the standard Metric-FF is based on a more compact and efficient implementation and it also benefits from the local search technique. However, given more time, ENHSP, which is informed by h^{add}_{abs} heuristics, solves more problem instances. This shows the benefit of a better informative heuristic can bring to the planner.

From the experiment result, we may be able to conclude that ENHSP informed by h^{add}_{abs} is overall slower than the standard Metric-FF, but solves more instances. Furthermore, note that due to the adoption of the enforced hill-climbing search and the pruning technique by helpful action, the standard Metric-FF is *not* complete. That is to say, even given infinite running time on a solvable problem instance, the standard Metric-FF can still find no plan. In contrast, ENHSP informed by a pruning-safe heuristic is complete. Therefore, the comparison results in this experiment should be taken with carefulness.

4.3.3 Evaluation of h^{max}_{abs}

We evaluate h^{max}_{abs} against two benchmark approaches including uniform cost search and goal sensitive heuristics (1 to non-goal state, 0 to goal state). Ties are broken preferring larger g-values. Evaluation was conducted on mainly FO-COUNTERS and FARMLAND-LN domains as other benchmark domains seem to be hard to solve optimally for both with and without h^{max}_{abs}.

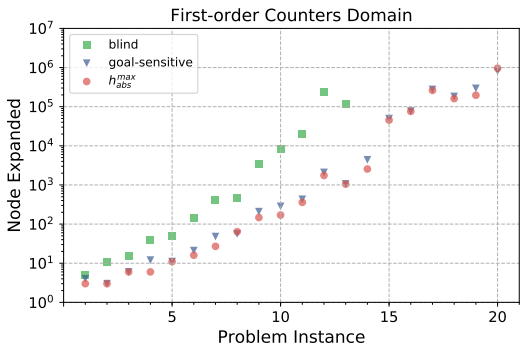
Table 4.5: Comparison between uniform cost search (blind) and A* informed by goal-sensitive (0-1) heuristics and h^{max}_{abs} on optimal planner. Entries are calculated on instances solved by all settings. Best performers are highlighted in bold.

	Coverage			CPU-Time (s)			Exp. Nodes			Eval. Nodes		
	blind	0-1	h^{max}_{abs}	blind	0-1	h^{max}_{abs}	blind	0-1	h^{max}_{abs}	blind	0-1	h^{max}_{abs}
FO-COUNTERS (39)	2	**4**	**4**	0.4	0.5	**0.2**	206	26	**15**	1k	72.5	**52.5**
FO-COUNTERS-INV (39)	2	**3**	**3**	3.5	0.5	**0.4**	57.7k	536.5	**531**	334k	1.5k	**1.2k**
FO-COUNTERS-RND (117)	9	**13**	**13**	2.5	0.6	**0.4**	30.5k	345.5	**280.1**	286.3k	1.2k	**1.1k**
FARMLAND-LN (50)	2	**13**	**13**	2.1	**0.2**	**0.2**	**195k**	**1.6k**	**1.6k**	588k	**2.9k**	**2.9k**

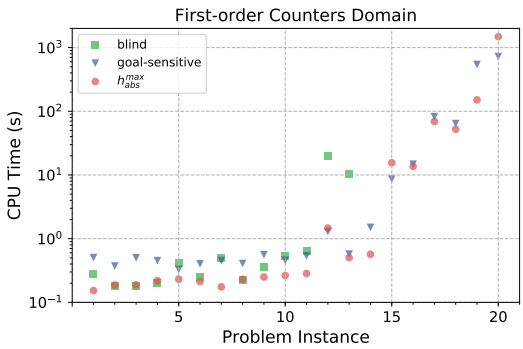
While uniform cost search performs in general much worse than the informed search, which can be anticipated, h^{max}_{abs} performs slightly better than the goal sensitive heuristics in all experimented domains. Fig. 4.9 and Fig. 4.10 compare the node expansion and CPU-time of the three approaches on each problem instance.

In-depth: The marginal improvement on first-order counters domain is due to h^{max}_{abs} captures the fact that increasing or decreasing a variable is dependent on the rate. Namely, if the rate is 0, an action increasing the rate must be executed before increasing or decreasing the value of the related variable. This ordered dependency is however not captured in the goal-sensitive heuristics.

On the FARMLAND-LN, however, we notice that h^{max}_{abs} falls back to the goal sensitive heuristics on most instances. This is a result of the minimisation step in h^{max}_{abs} is minimising the cost of achieving action preconditions and the cost of action repetitions separately (c.f. Eq. 2.12), which implies that the action with the cheapest preconditions might not be the one achieving the condition most efficiently. More specifically,

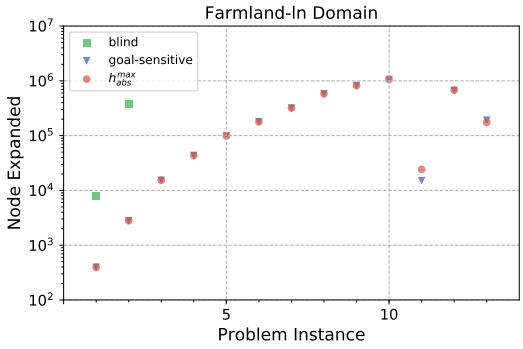


(a) Nodes expanded on First-order counters instances

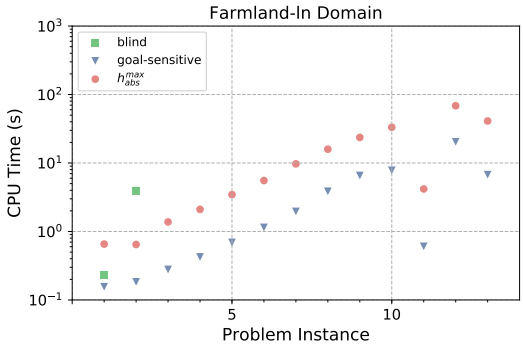


(b) CPU-time on First-order counters instances

Figure 4.9: Comparison on First-order counters domain: Instances are the mixtures of solvable instances from FO-COUNTERS, FO-COUNTERS-INV and FO-COUNTERS-RND, all sorted in ascending order with respect to the optimal plan lengths. Blind search solves least number of instances while expands most nodes. h^{max}_{abs} shows a slight improvement than the goal-sensitive heuristic concerning the node expansion and the CPU-time, which is due to the fact that h^{max}_{abs} captures the dependencies between actions a bit better.



(a) Nodes expanded on Farmland-ln instances



(b) CPU-time on FARMLAND-LN instances

Figure 4.10: Comparison on FARMLAND-LN domain: Solvable instances are sorted in ascending order with respect to the problem size: primarily by number of farms and secondly by total number of workers. Blind search solves least number of instances while expands most nodes. Because h^{max}_{abs} degrades to the goal-sensitive heuristic as discussed, the node expansions of the two approaches are mostly the same. In this circumstances, the informedness of h^{max}_{abs} does not justify its computation overhead and is usually one order of magnitude slower than the goal-sensitive heuristic.

on the FARMLAND-LN domain, there is an action `move-slow` (c.f. Appendix A) that can achieve the goal and its preconditions are always satisfied. Therefore, for the goal condition on the cost, the minimisation term over action precondition is always 0. The minimisation term over action cost is always 1 on non-goal state due to the existence

of asymptotic subaction and 0 to goal state. This is consistent with the experiment results, which show that h^{max}_{abs} and goal-sensitive heuristics give the same result with respect to the node expansion on most instances. Due to the overhead on computing the subgoaling heuristics, h^{max}_{abs} is unsurprisingly worse than goal-sensitive heuristics in the CPU-time.

Overall, as h^{max}_{abs} only attempts to capture the dependency relations between actions using the uniform cost, which sometimes can even be diminished due to the separate minimisation in h^{max}_{abs} as we show in the FARMLAND-LN domain, it seems h^{max}_{abs} is just a bit better than the goal-sensitive heuristics. However, the observation that all the plan lengths produced by h^{max}_{abs} are optimal, which may support our rationales on its admissibility.

Chapter 5

Related Work

This thesis is built mainly upon the work of hybrid subgoaling [1] and additive interval-based relaxation [2]. As we are aware of, there are other works in the literature exploring solutions to planning problems in hybrid domains. In this section, we examine these works in order.

Our discussions consist two main parts. In Section 5.1, we introduce abstraction techniques used in the planning and model checking community. Since both of these two communities are concerned about reachability analysis, we also attempt to understand the connections between approaches employed in these two fields. In Section 5.2, we examine other approaches adopted to obtain heuristics in the planning literature.

5.1 Abstraction Techniques in Hybrid Domains

In the planning literature, although abstraction techniques are widely adopted in classical planning [42, 46–49], it is only recently that people try to apply abstraction techniques to hybrid planning problems.

ASTER by León et al. [39] attempts to recompile numeric planning problems and construct abstraction problems which come down to solving classical planning problems. In their approach, propositional fluents that represent whether numeric conditions are achieved or not are explicitly encoded. Numeric effects in the domain are mapped into non-deterministic propositional effects, which assign values to propositional fluents. The non-determinism is the result of the information loss when abstracting away the numeric characteristics of effects. Different from our approach, they do not use abstraction problems to compute heuristics. Instead, they directly use the plan of the abstraction problem to extract a policy, which is a mapping between states to operators. Such policy is then used to guide the search in the original problem. When the policy is not consistent with the original state, the search employs a plan repairing and backtracks until finds a state consistent with the policy. Such policy-guided search aided by plan repairing procedure is repeated until a goal state is reached. By compiling numeric planning problems into non-deterministic classical planning, they may take advantage of existing advancement in the classical planning. Their approach is initially present on a very restrictive numeric case. Namely, nu-

meric conditions can only take the form $v > c$, where v is a numeric variable, $c \in \mathbb{R}$, and there are only constant numeric effects. Later on, León et al. complete ASTER to a wider range of numeric planning problems [41], the induced algorithm is called ASTER*. However, it is worth pointing out that the policy-guided search of ASTER* is not complete. Therefore, it cannot provide a guarantee on the solution even the problem is solvable, which highlights the advantage of our heuristics.

While abstraction techniques are employed in numeric planning only recently, there is a large body of research work in the hybrid model checking and verification community adopting this approach. A typical model checking problem concerns whether a system can reach some error states. A system is safe is no such an error state is reachable and unsafe otherwise. These problems are connected with planning problems as they both focus on the reachability analysis of systems. Among these works, we found that two of them which derive heuristics are closely connected to our work. Both works are based on the abstraction induced by the so-called "symbolic states" [50, 51], where a concrete state is over-approximated by a region containing it.

In the boxed-based heuristics [50], such regions are over-approximated by their Cartesian abstraction [52], which explains the name of the heuristic. The distance between two numeric states is then approximated by the distance between centres of the two Cartesian abstraction regions containing the concrete states. The connection of boxed-based heuristics to our heuristics is interesting to observe: symbolic states are usually implied by *invariants*, which are similar to the numeric conditions in our problem. An unsafe state, therefore, corresponds to a goal state. From this perspective, the construction of boxed regions can be considered as an abstraction from the original state space, which contains concrete numeric states, to an abstraction state space, where a state is an area induced by intervals of possibly attainable values of each variable. However, because this abstraction happens only on the numeric part of the problem, the box-based heuristic does not properly reflect the discrete part of the problem.

Another work [51] borrows the idea of pattern database (PDB) from classical planning [42, 46–49], which results in a PDB heuristics on hybrid domains. In the context of classical planning, a pattern database is a mapping from each abstraction state to the actual planning cost from that abstract state to the goal. These abstraction states are usually acquired by ignoring a portion of numeric variables in the domain. Instead of ignoring variables, the hybrid PDB heuristic uses support functions to over-approximate the symbolic state. As a consequence, reachable states of the hybrid model is also over-approximated, which forms the abstract state space. Given a concrete state, the heuristic is computed as the distance between the abstract state and the abstract error state. The corresponding abstract state must share the same concrete part as that of the original concrete state, which is different from box-based heuristics. It is worth mentioning that the distance between an abstract state to an abstract error state requires extra offline computation. Besides, because they concentrate on only abstract states which are considered useful, as a result, their method loses completeness.

These two techniques focus on the abstraction on symbolic states, while the underlying transition systems remain intact. On the other hands, there are also works

that concentrate on the abstraction on the transitions systems. Among them, Alur et al. [53] propose to use predicate abstraction for hybrid system analysis. Jha et al. [54] computes abstractions by ignoring some continuous variables. Bogomolov et al. [55] construct abstractions by merging locations. As these works are not directly related to heurisitc search, we do not present them in detail.

5.2 Heuristics for Numeric Planning

Apart from the planners we have seen in previous sections, there are certainly other planners that use heuristics for the search control. In this section, we examine their approaches. All of these approaches are either relevant to ours or interesting enough for a discussion.

Similar to Metric-FF (c.f. Section 4.3.1), many planning algorithms compute heuristics through constructing a relaxed planning graph. The planner Sapa [56] totally ignores the numeric part of effects and constructs a relaxed planning graph backwards from the goal. The number of actions in this relaxed planning graph is given as the raw estimation. Then, this estimation is calibrated by computing how many additional actions are required to produce resources consumed by actions in the relaxed planning graph. The number of these additional actions is then added to the raw estimation to acquire the adjusted heuristic. As the construction of relaxed planning graph phase does not consider numeric effects, heuristics derived by Sapa is supposed to be at most equally informative as that by Metric-FF.

LPRPG by Coles et al. [57] improves the idea of interval-based relaxation (IBR) used in Metric-FF [3] by linear programmes. In IBR, when a numeric resource is consumed, the interval representing the quantity of the resource does not shrink. As a result, such resource can always be reused without being produced, which suggests there can be a much shorter plan in the relaxation than in the original problem. Similarly, an action that transfers resources between two locations can increase the quantity of the resource in the destination without decreasing that in the starting location, which leads to misinterpretation of helpful actions. To solve these problems, they explicitly model the constraints for variables representing numeric resources. These constraints are meant for capturing the lower and upper bounds of the resource quantities after expanding each action layer. It is worth pointing out that the improvement made by LPRPG only handles numeric planning problems featuring numeric resource transferring.

There are also planners do not rely on the relaxed planning graph approach.

LPG [58] uses a layered data structure called "Numerical Action Graph" (NA-graph) to represent the actual plan. NA-graph contains the current values of the propositional and numeric fluents and also numeric conditions that hold in each layer. When applying an action, a corresponding action node is generated. Such node is then connected to its preconditions and conditions it contributes to. There are inconsistencies in NA-graph, for example, an action node with a precondition that is not supported and an NA-graph without inconsistencies induces a valid plan. Therefore,

find a valid plan is achieved by resolving inconsistencies in the graph and the heuristic computed in LPG is used for selecting the most promising action to resolve the inconsistencies. Such heuristic is a weighted sum of terms including the additional search cost introduced by the new action and also the cost required to achieve the preconditions of the action. In essence, LPG uses a local search technique in the space of NA-graph. The heuristic it computes can be considered as a measurement of the plan lengths in the partial plan space.

TFD planner [59] extends the idea of the context-enhanced additive heuristics [60] to numeric fluents. Given a planning problem, TFD first compiles a planning task into an planning problem in the SAS^+ formalism. Such a formalism explicitly captures the dependency relationships between finer-granularity terms of numeric conditions, which is stored in a causal graph. Then TFD tries to reach goals by recursively computing the costs to achieve all the terms in the causal graph and returns the accumulated costs as the estimation. Costs to achieve a numeric condition are estimated by the costs to change the value of comparison variables. However, TFD only reasons over numeric fluents in a qualitative way and their heuristic does not consider the number of actions required to to a goal.

5.3 Summary

Though the abstraction technique is an established approach adopted for the classical planning problems, its straightforward and successful numeric extensions to the numeric planning problems are not well explored in the planning literature. Therefore, the framework proposed in this thesis may be able to complement the current knowledge body of deriving heuristics for numeric planning problems using abstractions. On the other hands, similar techniques are studied for decades in the model checking and software verification community. And it might be helpful to understand the connection between the two fields such that the research works can be cross-fertilised.

Obtaining heuristics for numeric planning domains is a multi-aspect problem. While constructing relaxed planning graphs and extracting relevant information from them is the foundation of computing heuristics for many planning systems, there are also works trying to tackle the problem from other perspectives such as knowledge representation. Our proposed heuristics do not use relaxed planning graphs explicitly. However, it would be interesting to understand whether we can refine our heuristics by borrowing ideas from the relaxed planning graph approach, for example, the extraction of a relaxed plan may help to extract critical paths in the subgoaling heuristics.

Chapter 6

Conclusion

Finding effective solutions to numeric planning problems involves many aspects: the underlying model needs to be compact enough for expressing numeric constructs in the problem domain; different search strategies have distinguished characteristics which can be rather important depending on the search space structure and scenarios the problem fits in; apart from heuristic search, there are other methodologies handling such problem as discussed in Chapter 5. This thesis concentrates on one aspect among them, which aims at better search control by using domain-independent heuristics.

6.1 Summary of Contributions

We propose a framework that constructs abstraction problems by decomposition and subactioning. Such abstraction problems are characterised by linear conditions affected by constant effects, which can be directly handled by hybrid subgoaling heuristics. We then apply hybrid subgoaling heuristics on such abstraction problems and take the heuristic value computed to guide the search in the original problem.

Due to the observation that abstraction problem constructed may not preserve the solvability of the original problem, which implies that heuristics computed on the abstraction problem may not preserve the pruning-safeness, we propose a theorem which is sufficient to ensure any solvable task is also solvable in the abstraction problem giving the same initial state. The theorem proposed provides with guidance for constructions of further refined abstraction problems.

Using such framework, we managed to acquire two abstraction-based heuristics h^{add}_{abs} and h^{max}_{abs}. h^{add}_{abs} is an inadmissible heuristics designed for satisficing planning. We evaluate h^{add}_{abs} by comparison with Metric-FF heuristics and AIBR counting heuristics using forward search. The result shows that h^{add}_{abs} provides more informative heuristics than its competitors on most experiment domains and consequently the planner ENHSP informed by h^{add}_{abs} solves most problem instances within the time restriction. This is an indication of a good quality h^{add}_{abs}. Besides, to understand the performance of planning systems, we compare ENHSP informed by h^{add}_{abs} and Metric-FF with enforced hill-climbing and helpful action pruning turned on. The experiment result shows that Metric-FF is in general more efficient on small-scaled problems than ENHSP. How-

ever, due to better search control provided by h^{add}_{abs}, ENHSP solves more problem instances when the time restriction increases. Under the experiment setting, ENHSP eventually solves more instances than Metric-FF, which was not ever achieved using forward search by ENHSP on non-simple numeric cases.

On the other hand, h^{max}_{abs} is a preliminary attempt towards solving linearly affected cases optimally. As admissible heuristics for these particular problems remain non-existent, we compare h^{max}_{abs} with uniform cost search and A* with the goal-sensitive heuristic. While h^{max}_{abs} largely outperforms blind search on all domains, it just shows marginal improvements compared with goal-sensitive heuristics on some experiment domains.

6.2 Future Work

There are some directions to follow for further exploration.

- To understand what are the *necessary* conditions for heuristics derived to be safe-pruning.
- To understand what is the best strategy to determine the number of subdomains for each non-constant linear effects and what can be a principal way to pick the representative sample for each subaction.
- As mentioned in Section 3.7.2, a more informative admissible heuristic can be visioned.
- To support actions whose cost is state-dependent, and deal with planning tasks which aim at optimising plan metrics.
- To support more hybrid constructs such as events and processes.

In conclusion, although the two heuristics proposed already prove to be effective on the experiment domains, more importantly, we believe the framework, which compiles planning problems having linear numeric effects into abstraction problems where all the numeric effects are constant while preserving the pruning-safeness of the heuristics derived, is general enough to warrant further study. We hope on the foundations of the proposed framework and related abstraction-based heuristics, subgoaling heuristics can be generalised to numeric planning problems in general in the future and eventually benefits the real-world planning tasks.

Appendix A

Experiment Domains in PDDL 2.1

TPP-METRIC

```
(define (domain TPP-Metric)
(:requirements :typing :fluents)
(:types
  place locatable - object
    depot market -place
  truck goods -locatable)

(:predicates (loc ?t - truck ?p - place))

(:functions
    (on-sale ?g - goods ?m - market)
      (drive-cost ?p1 ?p2 - place)
      (price ?g - goods ?m - market)
      (bought ?g - goods)
      (request ?g - goods)
      (total-cost))

(:action drive
 :parameters (?t - truck ?from ?to - place)
 :precondition (and (loc ?t ?from))
 :effect
      (and
        (not (loc ?t ?from)) (loc ?t ?to)
        (increase (total-cost) (drive-cost ?from ?to)))
       )

(:action buy-allneeded
 :parameters (?t - truck ?g - goods ?m - market)
 :precondition
      (and
        (loc ?t ?m) (> (on-sale ?g ?m) 0)
        (> (on-sale ?g ?m) (- (request ?g) (bought ?g)))
       )
 :effect
```

```
      (and
        (decrease (on-sale ?g ?m) (- (request ?g) (bought ?g)))
        (increase (total-cost) (* (- (request ?g) (bought ?g))
           (price ?g ?m)))
        (assign (bought ?g) (request ?g)))
       )

(:action buy-all
 :parameters (?t - truck ?g - goods ?m - market)
 :precondition
     (and
        (loc ?t ?m) (> (on-sale ?g ?m) 0)
        (<= (on-sale ?g ?m) (- (request ?g) (bought ?g)))
       )
 :effect (and
        (assign (on-sale ?g ?m) 0)
        (increase (total-cost) (* (on-sale ?g ?m) (price ?g ?m)))
        (increase (bought ?g) (on-sale ?g ?m)))
     )
)
```

Listing A.1: TPP-Metric in PDDL 2.1

FO-COUNTERS

```
(define (domain fo-counters)
    (:requirements :strips :typing :equality :adl)
    (:types counter)

    (:functions
        (value ?c - counter)
        (rate_value ?c - counter)
        (max_int)
    )

    (:action increment
         :parameters (?c - counter)
         :precondition (and (<= (+ (value ?c) (rate_value ?c))
                            (max_int)))
         :effect (and (increase (value ?c) (rate_value ?c)))
    )

    (:action decrement
         :parameters (?c - counter)
         :precondition (and (>= (- (value ?c) (rate_value ?c)) 0))
         :effect (and (decrease (value ?c) (rate_value ?c)))
    )

    (:action increase_rate
```

```
        :parameters (?c - counter)
        :precondition (and (<= (+ (rate_value ?c) 1) 10))
        :effect (and (increase (rate_value ?c) 1))
    )

    (:action decrement_rate
        :parameters (?c - counter)
        :precondition (and (>= (rate_value ?c) 1))
        :effect (and (decrease (rate_value ?c) 1))
    )


)
```

Listing A.2: Fo-counters in PDDL 2.1

SAILING-LN

```
(define (domain sailing_ln)
    (:types boat -object person -object)
    (:predicates
        (saved ?t -person)
    )
    (:functions
        (x ?b -boat)
        (y ?b -boat)
        (v ?b -boat)
        (d ?t -person)
    )

    (:action go_north_east
        :parameters (?b -boat)
        :effect (and (increase (x ?b) (* (v ?b) 1.5))
                     (increase (y ?b) (* (v ?b) 1.5))
                )
    )
    (:action go_north_west
        :parameters (?b -boat)
        :effect (and (decrease (x ?b) (* (v ?b) 1.5))
                     (increase (y ?b) (* (v ?b) 1.5))
                )
    )

    (:action go_est
        :parameters (?b -boat)
        :effect (and (increase (x ?b) (* (v ?b) 3)))
    )

    (:action go_west
```

```
        :parameters (?b -boat)
        :effect (and (decrease (x ?b) (* (v ?b) 3)))
    )

    (:action go_south_west
        :parameters(?b -boat)
        :effect (and (increase (x ?b) (* (v ?b) 2))
         (decrease (y ?b) (* (v ?b) 2))
                )
    )

    (:action go_south_east
        :parameters(?b -boat)
        :effect (and (decrease (x ?b) (* (v ?b) 2))
                     (decrease (y ?b) (* (v ?b) 2))
                )
    )

    (:action go_south
        :parameters(?b -boat)
        :effect (and (decrease (y ?b) (* (v ?b) 2)))
    )

    (:action accelerate
        :parameters(?b -boat)
        :effect (and (increase (v ?b) 1))
    )

    (:action decelerate
        :parameters(?b -boat)
        :precondition (and (>= (- (v ?b) 1) 1))
        :effect (and (decrease (v ?b) 1))
    )

    (:action save_person
       :parameters(?b -boat ?t -person)
       :precondition ( and  (>= (+ (x ?b) (y ?b)) (d ?t))
                            (>= (- (y ?b) (x ?b)) (d ?t))
                            (<= (+ (x ?b) (y ?b)) (+ (d ?t) 25))
                            (<= (- (y ?b) (x ?b)) (+ (d ?t) 25))
                            (<= (v ?b) 1)
                     )
       :effect (and(saved ?t))
    )

)
```

Listing A.3: Sailing-ln in PDDL 2.1

FARMLAND-LN

```
(define (domain farmland_ln)
    (:types farm -object)

    (:predicates (adj ?f1 ?f2 -farm))
    (:functions
        (x ?b -farm)
        (cost)
        (num-of-cars)
    )

    (:action move-by-car
        :parameters (?f1 ?f2 -farm)
        :precondition (and (not (= ?f1 ?f2))
                           (>= (x ?f1) (* 4 (num-of-cars)))
                           (adj ?f1 ?f2)
                      )
        :effect (and  (decrease (x ?f1) (* 4 (num-of-cars)))
                      (increase (x ?f2) (* 4 (num-of-cars)))
                      (increase (cost) (* 0.1  (* 4 (num-of-cars))))
                )
    )

    (:action move-slow
         :parameters (?f1 ?f2 -farm)
         :precondition (and (not (= ?f1 ?f2))
                            (>= (x ?f1) 1)
                            (adj ?f1 ?f2)
                       )
         :effect (and (decrease (x ?f1) 1)
                      (increase (x ?f2) 1)
                 )
    )

    (:action hire-car
        :parameters ()
        :effect (and  (increase (num-of-cars) 1))
    )
)
```

Listing A.4: Farmland-ln in PDDL 2.1

Appendix B

Optimal Plan Lengths for FO-COUNTERS Instances

FO-COUNTERS was designed purposefully for measuring performances of planners in reasoning about domains involving non-constant linear dynamics. From the experiment, it is obvious that this domain poses to be challenging for even the state-of-art planners. Therefore, it would be of our interest to understand how good plans produced by planners are and how much can be improved. Here, we present the optimal plan lengths for instance 2 to 40 of FO-COUNTERS for convenience of reference. These results are derived analytically.

Table B.1: Optimal Plan Lengths of FO-COUNTERS Instances.

No.	Lengths	No.	Lengths	No.	Lengths	No.	Lengths
2	2	12	55	22	140	32	247
3	5	13	62	23	150	33	259
4	9	14	70	24	160	34	271
5	13	15	78	25	170	35	283
6	18	16	86	26	180	36	295
7	23	17	94	27	191	37	307
8	29	18	103	28	202	38	320
9	35	19	112	29	213	39	333
10	41	20	121	30	224	40	346
11	48	21	130	31	235		

Bibliography

[1] Enrico Scala, Patrik Haslum, and Sylvie Thiébaux. "Heuristics for Numeric Planning via Subgoaling." In: *IJCAI*. 2016, pp. 3228–3234.

[2] Enrico Scala et al. "Interval-Based Relaxation for General Numeric Planning." In: *ECAI*. 2016, pp. 655–663.

[3] Jörg Hoffmann. "Metric-FF Planning System: Translating" Ignoring Delete Lists" to Numeric State Variables". In: *Journal Of Artificial Intelligence Research* 20 (2003).

[4] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN: 1558608567.

[5] Richard E Fikes and Nils J Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving". In: *Artificial intelligence* 2.3-4 (1971), pp. 189–208.

[6] RS Aylett et al. "AI planning: solutions for real world problems". In: *Knowledge-Based Systems* 13.2 (2000), pp. 61–69.

[7] Michael Beetz. "Plan Representation for Robotic Agents". In: *In Proceedings of the Sixth International Conference on AI Planning and Scheduling*. Citeseer. 2002.

[8] Drew McDermott et al. "PDDL - The Planning Domain Definition Language". In: (Aug. 1998).

[9] Maria Fox and Derek Long. "PDDL2. 1: An extension to PDDL for expressing temporal planning domains". In: *Journal of artificial intelligence research* (2003).

[10] S Edelkamp and J Hoffmann. *PDDL 2.2: The Language for the Classical Part of the 4th International Planning Competition, Albert Ludwigs Universität Institüt fur Informatik, Freiburg*. Tech. rep. Germany, Technical Report, 2004.

[11] Alfonso Gerevini and Derek Long. "Plan Constraints and Preferences in PDDL3". In: *ICAPS* (2006), p. 7.

[12] Daniel L Kovacs. "BNF definition of PDDL 3.1". In: *Unpublished manuscript from the IPC-2011 website* (2011).

[13] Maria Fox and Derek Long. "Modelling Mixed Discrete-Continuous Domains for Planning". In: *Journal Of Artificial Intelligence Research* (2006).

[14] Håkan LS Younes and Michael L Littman. "PPDDL1. 0: An extension to PDDL for expressing planning domains with probabilistic effects". In: *Techn. Rep. CMU-CS-04-162* (2004).

[15] Enrico Scala et al. “Numeric Planning with Disjunctive Global Constraints via SMT.” In: *AAAI*. 2016.

[16] Alfonso E Gerevini et al. “Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners”. In: *Artificial Intelligence* 173.5 (2009), pp. 619–668.

[17] Tom Bylander. “The computational complexity of propositional STRIPS planning”. In: *Artificial Intelligence* 69.1-2 (1994), pp. 165–204.

[18] Malte Helmert. “Decidability and Undecidability Results for Planning with Numerical State Variables.” In: *PuK*. 2002.

[19] Hector Geffner and Blai Bonet. “A concise introduction to models and methods for automated planning”. In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 8.1 (2013), pp. 1–141.

[20] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co., Inc., Reading, MA, 1984.

[21] Stuart Russell, Peter Norvig, and Artificial Intelligence. “A modern approach”. In: *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* 25 (1995), p. 27.

[22] Jörg Hoffmann and Bernhard Nebel. “The FF planning system: Fast plan generation through heuristic search”. In: *Journal of Artificial Intelligence Research* 14 (2001), pp. 253–302.

[23] Edsger W Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[24] Peter E Hart, Nils J Nilsson, and Bertram Raphael. “A formal basis for the heuristic determination of minimum cost paths”. In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[25] Ira Pohl. “Heuristic search viewed as path finding in a graph”. In: *Artificial Intelligence* 1.3 (1970), pp. 193–204. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(70)90007-X. URL: http://www.sciencedirect.com/science/article/pii/000437027090007X.

[26] Richard E Korf. “Depth-first iterative-deepening: An optimal admissible tree search”. In: *Artificial intelligence* 27.1 (1985), pp. 97–109.

[27] Jorge A Baier, Adi Botea, et al. “Improving Planning Performance Using Low-Conflict Relaxed Plans.” In: *ICAPS*. 2009.

[28] Blai Bonet and Hector Geffner. “Planning As Heuristic Search: New Results”. In: *Proceedings of the 5th European Conference on Planning: Recent Advances in AI Planning*. ECP ’99. London, UK, UK: Springer-Verlag, 1999, pp. 360–372. ISBN: 3-540-67866-2. URL: http://dl.acm.org/citation.cfm?id=647868.737103.

[29] Carmel Domshlak, Jörg Hoffmann, and Michael Katz. “Red–black planning: A new systematic approach to partial delete relaxation”. In: *Artificial Intelligence* 221 (2015), pp. 73–114.

[30] Jörg Hoffmann. "Where'ignoring delete lists' works: Local search topology in planning benchmarks". In: *Journal of Artificial Intelligence Research* 24 (2005), pp. 685–758.

[31] Michael Katz, Jörg Hoffmann, and Carmel Domshlak. "Who said we need to relax all variables?" In: *ICAPS*. 2013.

[32] Emil Ragip Keyder, Jörg Hoffmann, Patrik Haslum, et al. "Semi-Relaxed Plan Heuristics." In: *ICAPS*. 2012.

[33] Drew McDermott. "A Heuristic Estimator for Means-Ends Analysis in Planning." In: *AIPS*. 1996.

[34] Malik Ghallab, Dana Nau, and Paolo Traverso. "The actor's view of automated planning and acting: A position paper". In: *Artificial Intelligence* 208 (2014), pp. 1–17.

[35] Peter Gregory et al. "Planning Modulo Theories: Extending the Planning Paradigm." In: *ICAPS*. 2012.

[36] Ramon E Moore, R Baker Kearfott, and Michael J Cloud. *Introduction to interval analysis*. SIAM, 2009.

[37] Johannes Aldinger, Robert Mattmüller, and Moritz Göbelbecker. "Complexity of interval relaxed numeric planning". In: *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*. Springer. 2015, pp. 19–31.

[38] Patrik Haslum. *Admissible heuristics for automated planning*. Diss. Linko ping : Linko pings universitet, 2006. Linko ping Department of Computer and Information Science, Linko pings universitet, 2006. ISBN: 9185497282. URL: http://openlibrary.org/books/OL25550886M.

[39] León Illanes and Sheila A McIlraith. "Numeric Planning via Search Space Abstraction". In: *Ninth Annual Symposium on Combinatorial Search*. 2016.

[40] Malte Helmert et al. "Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces". In: *Journal of the ACM (JACM)* 61.3 (2014), p. 16.

[41] León Illanes and Sheila A McIlraith. "Numeric Planning via Abstraction and Policy Guided Search". In: *IJCAI* (2017).

[42] Silvan Sievers, Manuela Ortlieb, and Malte Helmert. "Efficient Implementation of Pattern Database Heuristics for Classical Planning." In: *SOCS*. 2012.

[43] Susanna S Epp. *Discrete mathematics with applications*. Cengage Learning, 2010.

[44] Andrew I Coles and Amanda J Smith. "Marvin: A heuristic search planner with online macro-action learning". In: *Journal of Artificial Intelligence Research* 28 (2007), pp. 119–156.

[45] Enrico Scala et al. "Landmarks for Numeric Planning Problems". In: *IJCAI*. 2017.

[46] Stefan Edelkamp. "Planning with pattern databases". In: *Sixth European Conference on Planning*. 2014.

[47] Ariel Felner, Richard E Korf, and Sarit Hanan. "Additive pattern database heuristics". In: *Journal of Artificial Intelligence Research* 22 (2004), pp. 279–318.

[48] Patrik Haslum et al. "Domain-independent construction of pattern database heuristics for cost-optimal planning". In: *AAAI*. Vol. 7. 2007, pp. 1007–1012.

[49] Richard E Korf and Ariel Felner. "Disjoint pattern database heuristics". In: *Artificial intelligence* 134.1-2 (2002), pp. 9–22.

[50] Sergiy Bogomolov et al. "A box-based distance between regions for guiding the reachability analysis of SpaceEx". In: *Computer Aided Verification*. Springer. 2012, pp. 479–494.

[51] Sergiy Bogomolov et al. "Abstraction-based guided search for hybrid systems". In: *International SPIN Workshop on Model Checking of Software*. Springer. 2013, pp. 117–134.

[52] Thomas Ball, Andreas Podelski, and Sriram Rajamani. "Boolean and Cartesian abstraction for model checking C programs". In: *Tools and Algorithms for the Construction and Analysis of Systems* (2001), pp. 268–283.

[53] Rajeev Alur, Thao Dang, and Franjo Ivančić. "Predicate abstraction for reachability analysis of hybrid systems". In: *ACM transactions on embedded computing systems (TECS)* 5.1 (2006), pp. 152–199.

[54] Sumit Jha et al. "Reachability for linear hybrid automata using iterative relaxation abstraction". In: *Hybrid Systems: Computation and Control* (2007), pp. 287–300.

[55] Sergiy Bogomolov et al. "Assume-guarantee abstraction refinement meets hybrid systems". In: *Haifa verification conference*. Springer. 2014, pp. 116–131.

[56] Minh B Do and Subbarao Kambhampati. "Sapa: A multi-objective metric temporal planner". In: *Journal of Artificial Intelligence Research* 20 (2003), pp. 155–194.

[57] Andrew Coles et al. "A Hybrid Relaxed Planning Graph'LP Heuristic for Numeric Planning Domains." In: *ICAPS*. 2008, pp. 52–59.

[58] Alfonso Gerevini and Ivan Serina. "LPG: A Planner Based on Local Search for Planning Graphs with Action Costs." In: 2002.

[59] Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. "Using the Context-Enhanced Additive Heuristic for Temporal and Numeric Planning." In: *Towards Service Robots for Everyday Environments* 76 (2012), pp. 49–64.

[60] Malte Helmert. "A Planning Heuristic Based on Causal Graph Analysis." In: *ICAPS*. Vol. 4. 2004, pp. 161–170.