Understanding and Improving Merge-and-Shrink Abstraction for Cost Optimal Planning

by

Gaojian Fan

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

In this thesis, we study merge-and-shrink (M&S), a flexible abstraction technique for generating heuristics for cost optimal planning. We first propose three novel *merging strategies* for M&S, namely, *Undirected Min-Cut* (`UMC`), *Maximum Intermediate Abstraction Size Minimizing* (`MIASM`), and *Dynamic MIASM with Heuristic Quality* (`DM-HQ`) that improve the quality of M&S heuristics for problems in International Planning Competition (IPC) domains. We also introduce a lightweight M&S method *MS-lite* for constructing M&S heuristics in an extremely efficient way, which can complement other, relatively expensive M&S methods. We show that the combination of MS-lite and `DM-HQ` substantially outperforms the previous state of the art M&S method. We then focus on how to improve search with M&S heuristics when there are diverse action costs. We first study how diverse action cost affects search *without* heuristics. We prove a *No Free Lunch* theorem showing that, under mild assumptions, when no heuristic is used the positive and negative effects of diverse costs on search are perfectly balanced. We then show that cost diversity has negative effects on M&S heuristics for IPC problems. Finally, we propose a M&S method called *Merge-and-Shrink with Delta Cost Partitioning* (DCP-MS) that largely reduces the negative effects of cost diversity on the impacted IPC domains.

Acknowledgments

First of all, I would like to express my deepest gratitude to my supervisors, Robert Holte and Martin Müller for their support and mentorship. I feel so lucky to have both of them as my supervisors. They always gave me the freedom to pursue my research interests, while were also available to help me out whenever I stumbled. Their enthusiasm, patience and guidance were of paramount importance to the success of this research. I will always be very grateful for how much I have learned from them during my years as a doctoral student.

I would also like to thank my committee members Ryan Hayward, Zachary Friggstad and Christopher Beck for their insightful comments and encouragements. During my PhD studies, I have visited a few research groups and met many great researchers. They made the visits valuable and memorable experiences for me. Among many others, I would like to thank Silvan Sievers, Martin Wehrle, Álvaro Torralba, Peter Kissmann and Roni Stern for the helpful discussions related to my work during my visits to their groups.

I am also very grateful to my friends in the AI lab of the Department of Computing Science at the University of Alberta. In particular, I wish to thank Fan Xie, Levi Lelis and Rick Valenzano. I would also like to acknowledge the financial support of NSERC and the Department of Computing Science at the University of Alberta.

Many other friends have also been there for me over the years of my PhD studies. They include Jundong Li, Ping Jin, Wanxin Gao, Yuchen Wang, Jiuyu Sun, Yeqin Zhang, Zhaoxing Bu, Baoliang Wang, Lei Yao, Jingwei Chen, Chao Gao, Xuebin Qin, and many others. I will always appreciate the friendship and the great time we shared together over the years. I would like to especially

thank Yang Gao and Qingna Jin whose friendship help me endure my loneliest times in Edmonton. I am also very grateful to all my rock climbing friends, specially Yongxu Chen, with whom I spent hours bouldering and lead climbing.

Last, but not least, I want to thank my parents for their unconditional love, understanding and support during my whole life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Planning

Planning is an area of Artificial Intelligence (AI) that concerns the task of finding a sequence of actions that leads from an initial state to a goal state. Planning problems include finding a path between two places on a map, controlling multiple elevators to move passengers, changing a multi-joint robotic arm from one configuration to another without causing collisions, or logistics problems such as delivering packages using vehicles.

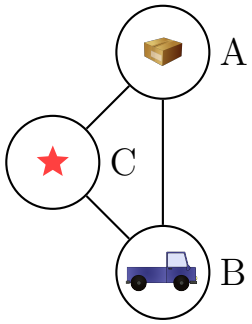


Figure 1.1: A simple logistics task.

Figure 1.1 illustrates a simple planning problem, a logistics task where a truck needs to deliver a package. The package can be loaded into the truck if they are at the same location. If the package is on the truck, it can be unloaded at the current location of the truck. There are three locations A, B and C, and the truck can move among them. Initially, the truck is at location B and the package is at location A. The goal is to deliver the package to its destination location C (marked by a star).

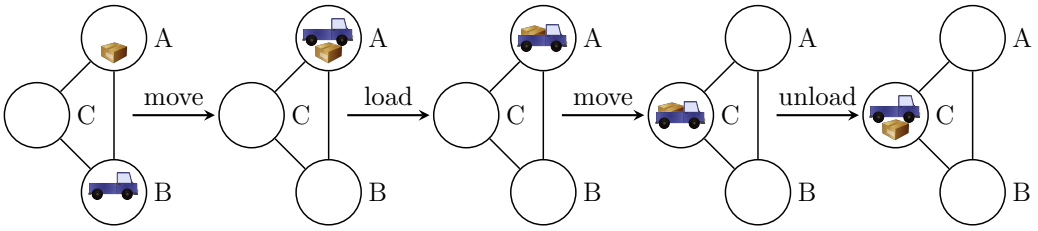


Figure 1.2: A plan for the logistics task shown in Figure 1.1

To achieve the goal, we can first move the truck to A—the location of the package. Next, we load the package into the truck. We then move them together to location C. Finally, we unload the package off the truck. This multi-step process is shown Figure 1.2. Each step between two states is called an *action*. The whole process forms a 4-action plan for this logistics task.

Each action can also have a cost. The cost of a plan is the sum of the costs of its actions. In this thesis, we study *optimal planning* that finds a plan of minimum cost. Such a plan is called an *optimal plan*. If costs of all actions are equal, then optimal planning is to find a plan of the minimum number of actions. In our logistics example, the 4-action plan in Figure 1.2 is an optimal plan if all action costs are the same. If moving the truck between A and C costs 3 but moving the truck between A and B and between B and C both cost 1, an optimal plan would use a detour through B to move the truck from A to C after picking up the package, instead of the direct move from A to C.

For k locations, n trucks and m packages, there are $k^n(k+1)^m$ states, i.e., combinations of locations of the truck and the package. Figuring out a plan for our example task is not difficult, because there are only 12 states. However, if a few more trucks/packages/locations are considered, there will be more states. For example, for a task of 10 locations, 10 packages and 2 trucks, there are 2,593,742,460,100 states. This *exponential* difference between the number of states of a planning task and the complexity for describing the task is known as the *state explosion* problem. For many planning problems, this is a main challenge a planning algorithm needs to address in order to solve the problem.

1.2 Heuristic Search

Heuristic search is a powerful method for solving difficult planning tasks. A search algorithm provides a systematic way to find a desired plan through explorations in the *state space*, i.e., the collection of all possible states. Because of the state explosion problem such explorations may need to check prohibitively many states before finding a plan. *Heuristic functions* are used to reduce the amount of exploration that needs to be done in a search. For any state, a heuristic function provides an estimation of the future cost for reaching a goal state from the state. For example, for our logistics tasks we can estimate the future cost of a state based on the location of the package. If the package is in the truck, we use the cost of unloading as the estimate. If the package is at a location different from the goal location, we use the sum of costs of loading and unloading as the estimate. If the package is in its goal location, the estimate is zero. Such estimates can improve the efficiency of search algorithm by avoiding the exploration of some "unpromising" states—states that are less likely to appear in desired plans. In general, the more accurate heuristic functions are, the more unnecessary exploration can be avoided.

Although the word "heuristic" often appears in the context of algorithms that obtain sub-optimal solutions to a problem, in AI planning and heuristic search, the use of heuristic functions does not necessarily sacrifice the optimality. If the heuristic estimations are *underestimations* of the true future costs, then we can guarantee the optimality of the plan found. Such heuristics are called *admissible* heuristics. The package-location-based heuristic mentioned above is an admissible heuristic function for logistics tasks because for any state the loading and/or unloading actions considered in its estimations must appear at least once in any valid plan for the state.

1.3 Merge-and-Shrink

Abstraction is an important technique for creating admissible heuristics for planning. It creates mappings that combine multiple states into one abstract

state. The costs of the optimal plans that change abstract states to an abstract goal states are then used as the heuristic estimations.

Traditional abstraction methods have limited flexibility in how to combine states in an abstract mapping, which make it hard to create heuristic functions of high quality when computational resources are constrained. *Merge-and-shrink* (M&S) is an abstraction method that converts a set of small abstractions to one abstraction by an iterative process of multiple M&S operations, namely merging, shrinking, pruning and label reduction, that are performed on the explicit representations of the state spaces. These operations and the M&S framework are defined in detail in Chapter 2.

1.4 Contributions of this Thesis

M&S has greater flexibility of what abstraction mappings can be computed than traditional abstraction methods. How to construct and use M&S heuristics to improve the efficiency of optimal planning is an interesting research topic and is also the focus of this research. In this thesis, we propose new methods for building M&S heuristics and explore new ways to use multiple M&S heuristics to improve the efficiency of planning. In addition, we also present theoretical and practical analyses of the effects of action costs on search and M&S heuristics.

Chapter 3 introduces three new *merging strategies* for M&S: `UMC`, `MIASM` and `DM-HQ`. The merging strategy of M&S defines in what order the merging operations are performed in the iterative process of M&S. Each of these merging strategies explores a unique idea for constructing better M&S abstractions. `UMC` stands for "Undirected Minimum Cut" because it uses the minimum cuts in a modified causal graph $\mathcal{G}^{\texttt{UMC}}$ to develop a merging order. The edge weights in $\mathcal{G}^{\texttt{UMC}}$ can be viewed as a measurement of how strongly variables, or equivalently the intermediate abstractions processed by M&S, interact with each other. The idea behind `UMC` is to merge abstractions with stronger interaction earlier to avoid information loss, and thus to produce a more informative M&S heuristic. `MIASM` stands for "Maximal Intermediate Abstraction Size Minimiz-

ing" whose goal is to find merging orders that exploit *free pruning*—an lossless M&S operation—to avoid harmful shrinking as much as possible. `MIASM` was a strong merging strategy that outperformed all existing merging strategies at the time it was published. Building on `MIASM`'s principle, a simplified version of `MIASM` called dynamic `MIASM` was later developed. Our third merging strategy `DM-HQ` was developed on top of dynamic `MIASM` with an additional merge scoring function f_{ms}^{HQ} which utilizes information about heuristic quality to help make merging decisions. This additional function improves the performance of M&S on several domains, and makes `DM-HQ` the currently best performing merging strategy when used alone.

Many M&S methods, like the ones proposed in Chapter 3, are trying to build a heuristic that is as accurate as possible. This often requires an expensive construction process in terms of time and memory consumption. In Chapter 4, we present a lossy but extremely fast M&S method called *MS-lite* that can be used to enhance other M&S methods to achieve superior performance. It can produce decent heuristics on tasks for which other M&S methods cannot even finish the heuristic construction within the resource limits. We also observe that MS-lite produces better heuristics on some benchmark domains despite its lossy construction process. We exploit the complementary strength of MS-lite to enhance other M&S methods by a method called *lite-enhancement*. We show that lite-enhancement works best for `DM-HQ` and it dramatically outperforms the state-of-the-art M&S on the benchmark set.

Many benchmark planning tasks use the same *unit costs* for all of their actions. However, in realistic planning problems diverse *non-unit action costs*, i.e., different costs for different actions, are often used. In addition to proposing new general M&S methods, this thesis also studies how action cost diversity affect the two components of M&S heuristic search planner, namely, the search algorithms and the domain-independent M&S heuristic construction methods.

In Chapter 5, we provide experimental evidence and theoretical analysis showing that search can also benefit from action cost diversity, despite previous studies demonstrating that cost diversity makes search more difficult. The main contribution of this chapter is a "No Free Lunch" (NFL) theorem for the

effects of cost diversity on Dijkstra's algorithm. The NFL theorem shows that, when ignoring the states that are the same distance from the start state as the goal state, the negative effects of cost diversity are perfectly counterbalanced by the positive effects. The states ignored by the NFL theorem are called the *TIE* states. We further show that when the influence of TIE states is considered it is advantageous to have a strongly concentrated distribution of solution costs. In many domains, unit costs give rise to a more concentrated distribution than diverse costs, but we give an example typifying domains in which the opposite is the case.

Chapter 6 shifts the focus to the effects of non-unit action costs on M&S heuristics. In this chapter, we first experimentally demonstrate that there are negative impacts of action cost diversity on M&S heuristics. We then propose a new *cost partitioning* scheme called *delta cost partitioning* (DCP) to address the negative effects of diverse costs on M&S. Our experiments show that M&S using DCP produces a set of additive M&S heuristics whose combination is much more informative than a single M&S heuristic that directly encodes the original diverse costs.

In Chapter 7, we summarize the contributions of this thesis and discuss some limitations and future research directions of this study.

Chapter 2

Basic Concepts

In this chapter, we first define the basic concepts of planning in Section 2.1 and heuristic search in Section 2.2. We then recap *abstraction* and *abstraction heuristics* in Section 2.3, followed by a detailed introduction of *merge-and-shrink*—a flexible abstraction method for planning—in Section 2.4. In Section 2.5, we briefly talk about the benchmark problems and the basic evaluation measurements used in this thesis. Most concepts introduced in this chapter are standard, but some are not formally defined in the literature but derived from the implementation details of related techniques.

2.1 Planning

In this thesis, we study so-called *classical planning* problems. Each classical planning task consists of a fully specified *initial state*, a desired *goal* condition, and a set of deterministic *actions* that define the transitions from one state to another.

2.1.1 Planning Task

There are a few formalisms for representing classical planning tasks. We use the SAS$^+$ formalism [BN95] with action costs to represent classical planning tasks:

Definition 1 (SAS+ Formalism)**.** A *SAS+ planning task* with action costs, or simply *planning task* is a 5-tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, \mathcal{C}, s_{\text{init}}, s_* \rangle$.

- $\mathcal{V}$ is a set of *state variables*. Each variable $v \in \mathcal{V}$ is associated with a finite domain $\mathcal{D}_v$.

- A function s is a *partial variable assignment* over $\mathcal{V}$ if s is defined on $V_s \subseteq \mathcal{V}$ such that $s(v) \in \mathcal{D}_v$ for $v \in V_s$. If $V_s = \mathcal{V}$, s is called a *state*. s_{init} is a state called the *initial state*, and s_* is a partial variable assignment called the *goal*. The variables in V_{s_*} are called the *goal variables*.

- $\mathcal{A}$ is a set of *actions* in which each action is a pair of partial variable assignments $\langle \text{pre}, \text{eff} \rangle$, the *precondition* and the *effect*.

- $\mathcal{C}$ is an *action cost* function that assigns a non-negative cost $\mathcal{C}(a) \in \mathbb{R}_0^+$ to each action $a \in \mathcal{A}$.

A state s *satisfies* a partial variable assignment p if $s(v) = p(v)$ for all $v \in V_p$. We say an action $a = \langle \text{pre}, \text{eff} \rangle$ is *applicable* to a state s if s satisfies pre. If a is applicable to s, applying a to s will result in a state s' such that $s'(v) = \text{eff}(v)$ for all $v \in V_{\text{eff}}$ and $s'(v) = s(v)$ for all $v \in \mathcal{V} \setminus V_{\text{eff}}$.

For a planning task $\Pi = \langle \mathcal{V}, \mathcal{A}, \mathcal{C}, s_{\text{init}}, s_* \rangle$, we call a sequence of $n \in \mathbb{N}_0$ actions $(a_1, a_2, ..., a_n)$ a *plan* of length n for Π if $a_i \in \mathcal{A}$ and there is a sequence of states $(s_0, s_1, ..., s_n)$ such that a_i is applicable to state s_{i-1} and the application results in state s_i for $i \in \{1, 2, ..., n\}$, and s_0 is the initial state s_{init} and s_n satisfies the goal s_*.

The cost of the plan $(a_1, a_2, ..., a_n)$ is the sum of the costs of its actions $\sum_{i=1}^{n} \mathcal{C}(a_i)$. A plan is *cost-optimal* or *optimal* for short if its cost is minimal among all plans for the task. *Planning* is to find a plan for a given planning task, and *optimal* planning is to find an optimal plan for a task.

2.1.2 Transition System

A *transition system* is a directed graph with *labelled transitions*, an *initial state* and a set of *goal states*. Formally,

Definition 2 (Transition System)**.** A *transition system* is a 6-tuple $\Theta = \langle S, L, \mathcal{C}, T, s_{\text{init}}, S_* \rangle$ where

- S is a finite set of *states* called the *state space*.
- L is a finite set of *transition labels*.
- $\mathcal{C}$ is the cost function that maps each label $l \in L$ to a non-negative cost $\mathcal{C}(l) \in \mathbb{R}_0^+$.
- $T \subseteq S \times L \times S$ is a set of *labelled transitions* or *transitions* for short. The cost of the transition $\langle s, l, s' \rangle \in T$ is $\mathcal{C}(l)$. We use $s \xrightarrow{l} s'$ to denote the transition $\langle s, l, s' \rangle$.
- s_{init} is the *initial state*.
- S_* is a set of *goal states*.

The number of transitions of a transition system is not considered by most techniques in this thesis, while the number of states is an important quantity for many of these techniques. We define the size of a transition system Θ as the number of states in its state space, denoted as $|\Theta|$.

Each planning task $\Pi = \langle \mathcal{V}, \mathcal{A}, \mathcal{C}, s_{\text{init}}, s_* \rangle$ is associated with a transition system $\Theta(\Pi) = \langle S, L, \mathcal{C}, T, s_{\text{init}}, S_* \rangle$ where

- S is the set of states over $\mathcal{V}$.
- $L = \mathcal{A}$ and $s \xrightarrow{a} s' \in T$ if and only if a is applicable to s and applying a to s results in s'. The cost of the transition $s \xrightarrow{a} s'$ is $\mathcal{C}(a)$.
- the initial state of $\Theta(\Pi)$ is the initial state of Π.
- $s \in S_*$ if and only if s satisfies s_*.

For any s and t such that $s \xrightarrow{l} t \in T$, we say t is a successor of s and s is a predecessor of t. A *path* of length $n \in \mathbb{N}_0$ from s_0 to s_n is a sequence $(s_0, l_1, s_1, l_2, s_2, l_3, ..., l_n, s_n)$ such that $s_{i-1} \xrightarrow{l_i} s_i \in T$ for $i \in \{1, 2, ..., n\}$. We use $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \xrightarrow{l_3} \cdots \xrightarrow{l_n} s_n$ to denote the path $(s_0, l_1, s_1, l_2, s_2, l_3, ..., l_n, s_n)$. We say a state t is *reachable* from a state s if there exists a path from s to t.

Definition 3 (Live and Dead State [Sie17])**.** A state in a transition system is *live* if it is reachable from the *initial state* and there is a goal state reachable from it. A state is *dead* if it is not live.

We call the percentage of live states in a transition system the **R**-*value* of the transition system. **R**-value is used in the definition of one of our new methods called `MIASM` in Chapter 3.

Definition 4 (**R**-value)**.** Let Θ be a transition system with state space S. We call the ratio of the number of live states of Θ to the total number of states, the **R**-*value*, denoted as $\mathbf{R}(\Theta)$. Formally,

$$\mathbf{R}(\Theta) = \frac{|\{s \mid s \text{ is a live state in } \Theta\}|}{|S|}$$

There is a one-to-one correspondence between paths in $\Theta(\Pi)$ and action sequences in Π. We call a path from the initial state to a goal state in $\Theta(\Pi)$ a *solution path*. Each plan for Π corresponds to a solution path of the same cost in $\Theta(\Pi)$, and each optimal plan for Π is a least-cost solution path in $\Theta(\Pi)$. We say a transition system is *solvable* if it contains a solution path.

2.2 Heuristic Search

The representation size of a planning task Π can be exponentially smaller than that of its associated transition system $\Theta(\Pi)$. This exponential difference in the representation sizes of a planning task and its associated transition system is known as the "state explosion" problem, which is the central challenge of classical planning: how to find a plan for a planning task that has an exponentially large state space.

Heuristic search is a method for finding paths in large transition systems. Because planning for Π can be cast as a path finding problem in $\Theta(\Pi)$, heuristic search can be used for planning. *Planning as heuristic search* [BG01] is one of the most powerful planning methods.

We denote the cost-from-start of a state, i.e., how far away a state is from the initial state, as the **g**-*value* of the state, and the cost-to-go of a state, i.e.,

how close the current state is to the nearest goal state, as the **h**-*value* of the state:

Definition 5 (**g**-value and $\mathbf{g}_\Theta$)**.** Let Θ be a transition system and s be a state in Θ. If s is reachable from the initial state, then the **g**-value of s in Θ is the cost of a least-cost path from s_{init} to s. If s is not reachable from s_{init}, the **g**-value of s in Θ is ∞. $\mathbf{g}_\Theta(s) : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ is the function that maps a state to its **g**-value.

Definition 6 (**h**-value and $\mathbf{h}_\Theta$)**.** Let Θ be a transition system and s be a state in Θ. If there are any goal states reachable from s, the **h**-value of s in Θ is the cost of a least-cost path from s to the nearest goal state. If no goal state is reachable from s, then the **h**-value of s in Θ is ∞. $\mathbf{h}_\Theta(s) : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ is the function of Θ that maps a state to its **h**-value.

A *heuristic function* or a *heuristic* for short estimates **h**-values of states.

Definition 7 (Heuristic Function)**.** For a state space S, a *heuristic function* $h : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ estimates the **h**-value of each state.

For a planning task Π, we wish to search for a solution path of $\Theta(\Pi)$ without generating the whole transition system. We only generate a set of states that are necessary to find a solution path. A heuristic search algorithm keeps such a set as small as possible.

A* [HNR68] is a heuristic search algorithm that can be used for planning. It maintains two mutually exclusive sets of states, OPEN and CLOSED, and an *evaluation* function $f(s) = g(s) + h(s)$ for $s \in \text{OPEN} \cup \text{CLOSED}$ where $g(s)$ is the cost of the *currently known* least-cost path from the initial state to s and h is the heuristic function used by A*. Functions g and f are only defined for states in OPEN$\cup$CLOSED, and they are updated during A* search when a new state is added to OPEN $\cup$ CLOSED or a cheaper path to a state in OPEN $\cup$ CLOSED is found. In our study, the heuristic function h stays unchanged throughout the search.

For a planning task $\Pi = \langle \mathcal{V}, \mathcal{A}, \mathcal{C}, s_{\text{init}}, s_* \rangle$ and a heuristic h for the state space of $\Theta(\Pi)$, the pseudocode of A* on Π is shown in Algorithm 1. The goal

checking (Line 6) and successor generation (Line 10 - 11) can be performed in polynomial time in the representation size of the planning task. $pred(s)$ records the previous state and action on the currently best known path to s, so that a solution path can be reconstructed (Line 7). The computation carried out in the else-branch between Line 9 and Line 21 is called *node expansion*. The *number of node expansions* is an important measurement of the efficiency of A*.

Definition 8 (Number of Node Expansions)**.** The *number of node expansions* performed by A* is the number of times the else-branch between Line 9 and Line 21 in Algorithm 1 is executed.

Algorithm 1 A* Search for Planning

Input: $\Pi = \langle \mathcal{V}, \mathcal{A}, \mathcal{C}, s_{\text{init}}, s_* \rangle$ and a heuristic h for Π
Output: a solution for Π or a signal that Π has no solution

```
1: OPEN ← {s_init}, CLOSED ← ∅
2: g(s_init) ← 0
3: pred(s_init) ← ⟨⟩
4: while OPEN ≠ ∅ do
5:     s ← a state from OPEN with the minimal f(s)
6:     if s satisfies s_* then
7:         return ReconstructSolution(s, pred)
8:     else
9:         OPEN ← OPEN \ {s} and CLOSED ← CLOSED ∪ {s}
10:        for all a = ⟨pre, eff⟩ ∈ A such that s satisfies pre do
11:            t ← apply a to s
12:            if t ∉ OPEN ∪ CLOSED or g(s) + C(a) < g(t) then
13:                if t ∉ OPEN ∪ CLOSED then
14:                    OPEN ← OPEN ∪ {t}
15:                else if t ∈ CLOSED then
16:                    CLOSED ← CLOSED \ {t} and OPEN ← OPEN ∪ {t}
17:                end if
18:                g(t) ← g(s) + C(a)
19:                pred(t) ← ⟨s, a⟩
20:            end if
21:        end for
22:    end if
23: end while
24: return Π has no solution
```

A state may be expanded multiple times if it is "re-opened" in Line 16,

i.e., moved back to OPEN after being moved to CLOSED. The number of expansions by A* is highly related to the quality of the heuristic it uses.

Definition 9 (Admissible Heuristic)**.** A heuristic h for a transition system Θ with state space S is *admissible* if $h(s) \leq \mathbf{h}_\Theta(s)$ for all $s \in S$.

When A* uses an admissible heuristic, the solution path found is guaranteed to be optimal, i.e., a least-cost path from the initial state to the nearest goal state.

Definition 10 (Consistent Heuristic)**.** A heuristic h for a transition system $\Theta = \langle S, L, \mathcal{C}, T, s_{\text{init}}, S_* \rangle$ is *consistent* if $h(s) \leq \mathcal{C}(l) + h(t)$ for all $s \xrightarrow{l} t \in T$ and $h(s) = 0$ for $s \in S_*$.

Consistency implies admissibility, i.e., a consistent heuristic is also an admissible heuristic, but not vice versa. When A* uses a consistent heuristic, states never move back from CLOSED to OPEN (Line 16 in Algorithm 1), and thus there are no re-expansions of states.

In general, the closer a heuristic approximates $\mathbf{h}_\Theta$, the fewer A* node expansions are needed to find a solution [DP85] (but note [Hol10]). We call a heuristic h *perfect* if $h(s) = \mathbf{h}_\Theta(s)$ for all $s \in S$. Throughout this thesis, we use A* as our search algorithm. All heuristics studied are consistent and some of them are perfect.

2.3 Abstraction

There is a large body of research for heuristic search planning on how to automatically generate heuristic functions. A variety of heuristic generation methods have been proposed for planning, e.g., [BG01; Ede01; Has+07; HG00; HD09; HHH07; HN01; SH13]. These methods are *domain-independent*, meaning they are not designed to generate heuristics for a specific planning domain, e.g., logistics problems, by exploiting some domain knowledge, but can create heuristic functions for any planning tasks that can be described in a planning formalism such as SAS+.

Abstraction is a fundamental domain-independent method for generating consistent and admissible heuristics. In this section we recap the basic concepts of abstraction.

2.3.1 Abstraction Mapping

The central concept of abstraction is *abstraction mapping*:

Definition 11 (Abstraction Mapping)**.** An *abstraction mapping* α for a transition system Θ with state space S is a mapping from S to a set S^α of *abstract states* called the *abstract state space*.

An abstraction mapping for Θ induces an *abstract transition system* of Θ:

Definition 12 (Abstract Transition System)**.** Let $\Theta = \langle S, L, \mathcal{C}, T, s_{\text{init}}, S_* \rangle$, and let $\alpha : S \mapsto S^\alpha$ be an abstraction mapping for Θ. The *abstract transition system*, or *abstraction* for short, induced by α is a transition system $\alpha(\Theta) = \langle S^\alpha, L, \mathcal{C}, T^\alpha, s^\alpha_{\text{init}}, S^\alpha_* \rangle$ where

- S^α is the abstract state space, i.e., the set of *abstract states*.
- $T^\alpha = \{\alpha(s) \xrightarrow{l} \alpha(s') \mid s \xrightarrow{l} s' \in T\}$ is the set of *abstract transitions*.
- $s^\alpha_{\text{init}} = \alpha(s_{\text{init}})$ is the *abstract initial state*.
- $S^\alpha_* = \{\alpha(s) \mid s \in S_*\}$ is the set of *abstract goal states*.

We call paths in $\alpha(\Theta)$ *abstract paths*. We call Θ the *concrete transition system*, and states, transitions and paths in Θ the *concrete states, transitions and paths*.

For any transition system Θ and any abstraction mapping α for Θ, each concrete path $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \xrightarrow{l_3} \cdots \xrightarrow{l_n} s_n$ in Θ corresponds to an abstract path $\alpha(s_0) \xrightarrow{l_1} \alpha(s_1) \xrightarrow{l_2} \alpha(s_2) \xrightarrow{l_3} \cdots \xrightarrow{l_n} \alpha(s_n)$. Thus, if $\alpha(t)$ is not reachable from $\alpha(s)$ in $\alpha(\Theta)$ then t is not reachable from s in Θ. This has two useful implications:

Proposition 1. Let Θ be a transition system, α be an abstraction mapping for Θ, and $\alpha(\Theta)$ be the abstract transition system induced by α.

(a) If an abstract state s^α is dead in $\alpha(\Theta)$, then all s such that $\alpha(s) = s^\alpha$ are dead in Θ.

(b) If $\alpha(\Theta)$ is not solvable, then Θ is not solvable.

2.3.2 Abstraction Heuristic

The **h**-values in the abstract transition system can be used as a consistent and admissible heuristic.

Definition 13 (Abstraction Heuristic)**.** Let α be an abstraction mapping for Θ with state space S, and let $\alpha(\Theta)$ be the induced abstract transition system. The abstraction heuristic $h^\alpha : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ defined by α is a function such that $h^\alpha(s) = \mathbf{h}_{\alpha(\Theta)}(\alpha(s))$ for $s \in S$.

While every concrete path is mapped to an abstract one, there may be abstract paths that are spurious, i.e., there are no concrete paths mapped to the abstract paths [FH15]. This is the reason that abstraction heuristics are underestimates of the perfect heuristic values. The underestimations can be even lower when there are a wide range of costs, because when there are multiple labelled transitions between the same pair of states, only the smallest label cost is used and all higher costs are ignored.

2.3.3 Projection

A *projection* is an abstraction mapping that produces an abstract transition system by ignoring some variables in the planning task. The *pattern database* (PDB) approach in planning is based on projections [Ede01].

Definition 14 (Projection)**.** Let Π be a planning task with variable set $\mathcal{V}$, and $\Theta(\Pi)$ be the transition system of Π with state set S. For any variable subset $V \subseteq \mathcal{V}$, a *projection on* V, denoted by π_V, is an abstraction of $\Theta(\Pi)$ with abstraction mapping α such that $\alpha(s) = \alpha(s')$ if and only if $s(v) = s'(v)$ for all $v \in V$. The projection on a singleton set $\{v\}$ is called an *atomic projection*, denoted by π_v.

Projection preserves all of the information about the variables in V but has no information about other variables. An abstraction mapping in such a restricted form may not provide an informative heuristic. Some high-quality heuristics based on projection use sophisticated methods for generating a collection of complementary projections [Ede06; Fra+17; Has+07].

2.4 Merge-and-Shrink

Merge-and-shrink [HHH07; Hel+14] is an algorithm that transforms the set of atomic projections of a planning task into a single transition system through four transformation operations, namely, *merging*, *shrinking*, *pruning* and *label reduction*.

2.4.1 Transformation Operations

We introduce the four transformation operations used in merge-and-shrink.

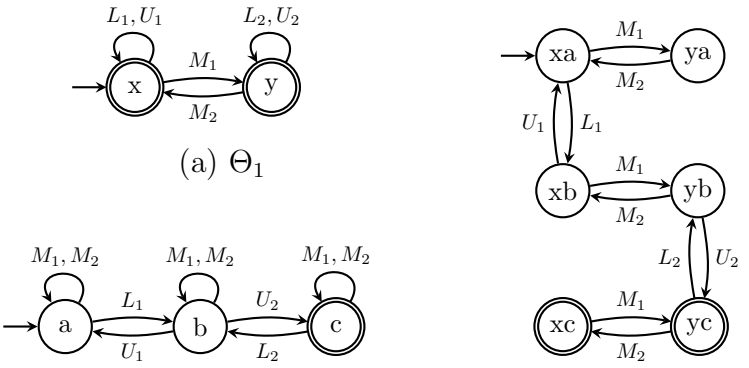
Merging Operation

The *merging* operation combines information from two transition systems by computing their *synchronized product*.

Definition 15 (Merging and Synchronized Product)**.** Let $\Theta_1 = \langle S^1, L, \mathcal{C}, T^1, s^1_{\text{init}}, S^1_* \rangle$ and $\Theta_2 = \langle S^2, L, \mathcal{C}, T^2, s^2_{\text{init}}, S^2_* \rangle$ be two transition systems. *Merging* Θ_1 and Θ_2, denoted by $\Theta_1 \otimes \Theta_2$, produces a transition system $\Theta_{1,2} = \langle S^{1,2}, L, \mathcal{C}, T^{1,2}, s^{1,2}_{\text{init}}, S^{1,2}_* \rangle$ where

- $S^{1,2} = S^1 \times S^2$.
- $T^{1,2} = \{(s^1, s^2) \xrightarrow{l} (t^1, t^2) \mid s^1 \xrightarrow{l} t^1 \in T^1 \text{ and } s^2 \xrightarrow{l} t^2 \in T^2\}$.
- $s^{1,2}_{\text{init}} = (s^1_{\text{init}}, s^2_{\text{init}})$.
- $S^{1,2}_* = S^1_* \times S^2_*$.

We call $\Theta_{1,2}$ the *synchronized product* of Θ_1 and Θ_2, and Θ_1 and Θ_2 the *factor* transition systems of $\Theta_{1,2}$. An example of merging is shown in Figure 2.1.



(a) Θ_1

(b) Θ_2

(c) $\Theta_{1,2} = \Theta_1 \otimes \Theta_2$

Figure 2.1: Two transition systems Θ_1 and Θ_2 and the synchronizted product $\Theta_{1,2} = \Theta_1 \otimes \Theta_2$. In each tranistion system, the inital state is marked with an unlabeled incoming edge not from any other states and the goal states are marked as double circles. The pair (s^1, s^2) of two states s^1 and s^2 is shown as $s^1 s^2$ for simplicity.

We consider transition systems equal if there is an isomorphism between their graphs that preserves the initial state, the goal states and the transition labels. Under this definition of equality of transition systems, the merging operation is associative and commutative. Different ways of merging a set of transition systems in any order yield equal synchronized products [Hel+14]. For a set $\mathcal{P}$ of transition systems, let $\bigotimes \mathcal{P}$ denote the synchronized product of merging all transition systems in $\mathcal{P}$.

Shrinking Operation

A *shrinking operation* is used to reduce the size of a transition system by applying abstraction to it.

Definition 16 (Shrinking)**.** Let Θ be a transition system. A *shrinking* α of Θ is an abstraction mapping for Θ that induces the abstract transition system $\alpha(\Theta)$.

Pruning Operation

Pruning of a transition system removes some states and their incident transitions from the transition system.

Definition 17 (Pruning)**.** Let $\Theta = \langle S, L, \mathcal{C}, T, s_{\text{init}}, S_* \rangle$ be a transition system. A *pruning* β for Θ induces its *pruned transition system* $\beta(\Theta) = \langle S^\beta, L, \{s \xrightarrow{l} t \in T \mid s, t \in S^\beta\}, s_{\text{init}}, S^\beta \cap S_* \rangle$ where $S^\beta \subseteq S$ is the state space associated with β such that S^β must contain s_{init}.

Label Reduction

Both merging and shrinking depend on how the transitions are labeled. *Label reduction* reduces the number of labels of a transition system.

Definition 18 (Label Reduction)**.** Let $\Theta = \langle S, L, \mathcal{C}, T, s_{\text{init}}, S_* \rangle$ be a transition system with cost function $\mathcal{C}$. A *label reduction* for L is a non-injective mapping $\tau : L \mapsto L^\tau$ such that $\mathcal{C}^\tau(\tau(l)) \leq \mathcal{C}(l)$ for all $l \in L$ where $\mathcal{C}^\tau : L^\tau \mapsto \mathbb{R}_0^+$ is the cost function associated with τ. Applying τ to Θ induces a transition system $\tau(\Theta) = \langle S, \tau(L), \{s \xrightarrow{\tau(l)} t \mid s \xrightarrow{l} t \in T\}, s_{\text{init}}, S_* \rangle$ with cost function $\mathcal{C}^\tau$.

2.4.2 Merge-and-Shrink Abstraction

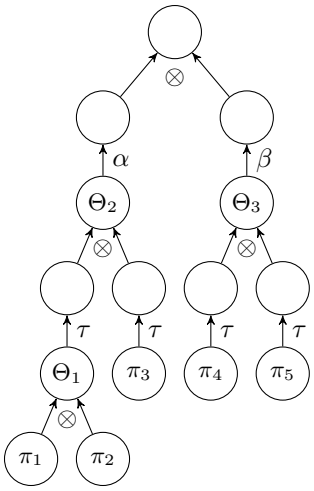
The transition systems produced by applying the above-mentioned transformation operations on atomic projections of a planning task are called *merge-and-shrink abstractions*.

Definition 19 (Merge-and-Shrink Abstraction)**.** Let Θ be a transition system with variables $\mathcal{V}$. A *merge-and-shrink abstraction* over $V \subseteq \mathcal{V}$ of Θ is defined inductively with the operations (A), (S), (P), (L) and (M) as follows.

- (A). For each $v \in \mathcal{V}$, the atomic projection π_v is a merge-and-shrink abstraction over $\{v\}$ of Θ.
- (S), (P) and (L). For any merge-and-shrink abstraction Θ' over V of Θ, and for any shrinking (S), any pruning (P), or any label reduction (L) χ for Θ', $\chi(\Theta')$ is also a merge-and-shrink abstraction over V of Θ.
- (M). For two merge-and-shrink abstractions Θ_1 and Θ_2 over V_1 and V_2 of Θ respectively such that $V_1 \cap V_2 = \emptyset$, and both abstractions share the same label set and the same cost function, $\Theta_1 \otimes \Theta_2$ is a merge-and-shrink abstraction over $V_1 \cup V_2$ for Θ.

1. Merging operation that produces $\Theta_1 = \pi_1 \otimes \pi_2$
2. Apply label reduction τ to Θ_1, π_3, π_4 and π_5
3. Merging operation that produces $\Theta_2 = \tau(\Theta_1) \otimes \tau(\pi_3)$
4. Merging operation that produces $\Theta_3 = \tau(\pi_4) \otimes \tau(\pi_5)$
5. Apply shrinking α to Θ_2
6. Apply pruning β to Θ_3
7. Merge $\alpha(\Theta_2)$ and $\beta(\Theta_3)$

(a)



(b)

Figure 2.2: (a) The M&S transformation process that converts 5 atomic projections π_1, π_2, π_3, π_4 and π_5 to one M&S abstraction; (b) The M&S tree representing the transformation process.

The transformation process producing merge-and-shrink abstractions can be represented as a binary tree called a *merge-and-shrink tree* in which the nodes are M&S abstractions and edges correspond to M&S transformation operations. In particular, the leaf nodes are the atomic projections, the root is the final M&S abstraction produced by the transformation process, and all internal nodes are the intermediate results of the transformation process. For nodes with only one child, the edge between them corresponds to an operation of shrinking, pruning, or label reduction, which is applied to the abstraction of the child node and produces the abstraction of the parent node. Each node with two children is the synchronized product of the abstractions of the child nodes. The two edges connecting the children to the parent node correspond to the merging operation producing the synchronized product. Figure 2.2 shows an example of a M&S transformation process and its corresponding M&S tree.

We call the size of the largest M&S abstraction in a M&S tree the *maximum intermediate abstraction size* of the M&S tree.

Definition 20 (Maximum Intermediate Abstraction Size)**.** For any M&S tree $\mathcal{T}$, the *maximum intermediate abstraction size* of $\mathcal{T}$, denoted as $|\mathcal{T}|_{\max}$, is the size of the largest abstraction in $\mathcal{T}$.

Algorithm 2 Merge-and-Shrink Initialization: **InitMAS**

Input: task Π with variables $\mathcal{V}$
Output: set $\mathcal{P}$ of all atomic projections of Π

```
1: P ← ∅
2: for v ∈ V do
3:     π_v ← CheckSolvabilityAndPruning(π_v)
4:     P ← P ∪ {π_v}
5: end for
6: return P
```

2.4.3 The Merge-and-Shrink Algorithm

The general merge-and-shrink framework maintains a set $\mathcal{P}$ of M&S abstractions which is initialized as the set of all atomic projections. The general merge-and-shrink algorithm is an iterative process in which each iteration replaces two M&S abstractions with their synchronized product. This is interleaved with possible shrinking, label reduction and/or pruning operations.

The initialization process **InitMAS** is shown in Algorithm 2. For any M&S abstraction $\Theta' = \langle S', L, \mathcal{C}, T', s'_{\text{init}}, S'_* \rangle$, the function CheckSolvabilityAndPruning(Θ') first computes the $\mathbf{g}$-values and $\mathbf{h}$-values of Θ' using Dijkstra's algorithm. If $\mathbf{h}_{\Theta'}(s'_{\text{init}}) = \infty$, we know Θ' is not solvable which implies that the given planning task has no solution because Θ' is an abstraction of the transition system of the task (Proposition 1(b)). In this case, we terminate the merge-and-shrink algorithm and output a signal that the given planning task has no solution. Otherwise, the function prunes the dead states, i.e., the states $s' \in S'$ with $\mathbf{h}_{\Theta'}(s') = \infty$ or $\mathbf{g}_{\Theta'}(s') = \infty$, and returns the pruned transition system. By Proposition 1(a), this pruning only removes dead states in the transition system of the task.

The general framework does not specify in each iteration what transformations are to be applied or in what order they are applied. Algorithm 3 shows the merge-and-shrink algorithm used in FastDownward [Hel06; Sie18]. In this implementation, in each iteration two transition systems $\Theta_1, \Theta_2 \in \mathcal{P}$ are first selected as the systems to be merged next. Then, a possible label reduction is computed based on Θ_1, Θ_2 and other transition systems in $\mathcal{P}$ and applied to the label set associated with $\mathcal{P}$. Possible shrinking may then be applied

Algorithm 3 The Merge-and-Shrink Algorithm in FastDownard [Hel06]

Input: task Π
Output: M&S abstraction $\Theta_{\text{M\&S}}$

1: $\mathcal{P} \leftarrow$ **InitMAS**(Π)
2: **while** $|\mathcal{P}| > 1$ **do**
3: $\quad \Theta_1, \Theta_2 \leftarrow \text{ChooseNextMerge}(\mathcal{P})$
4: $\quad \mathcal{P} \leftarrow \text{LabelReduction}(\Theta_1, \Theta_2, \mathcal{P})$
5: $\quad (\Theta_1, \Theta_2) \leftarrow \text{Shrinking}(\Theta_1, \Theta_2)$
6: $\quad \Theta_{1,2} \leftarrow \Theta_1 \otimes \Theta_2$
7: $\quad \Theta_{1,2} \leftarrow \text{CheckSolvabilityAndPruning}(\Theta_{1,2})$
8: $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{\Theta_{1,2}\} \setminus \{\Theta_1, \Theta_2\}$
9: **end while**
10: **return** $\Theta_{\text{M\&S}} \in \mathcal{P}$

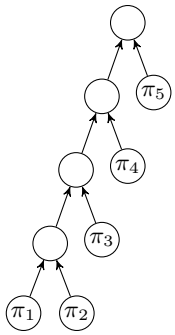
to Θ_1 and/or Θ_2. The synchronized product $\Theta_{1,2}$ of the two factor transition systems is produced after the possible label reduction and shrinking. Then, CheckSolvabilityAndPruning is called to check the solvability of the new transition system $\Theta_{1,2}$ and prune dead states in $\Theta_{1,2}$ if it is solvable. Finally, $\mathcal{P}$ is updated to $\mathcal{P} \setminus \{\Theta_1, \Theta_2\} \cup \{\Theta_{1,2}\}$. The iterative process ends when $\mathcal{P}$ becomes a singleton set containing one final transition system $\Theta_{\text{M\&S}}$.

For a task Π with variables $\mathcal{V}$, Algorithm 3 either produces a M&S abstraction $\Theta_{\text{M\&S}}$ over $\mathcal{V}$ of $\Theta(\Pi)$ or terminates if it detects that $\Theta(\Pi)$ is unsolvable (by the CheckSolvabilityAndPruning function). On each iteration, $\mathcal{P}$ is called the current *abstraction pool* of M&S.

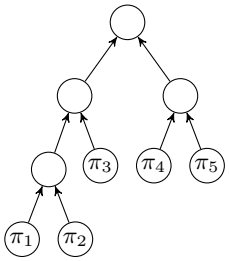
In the following, we give an overview of *merging strategies*, i.e., how to select the two transition systems to merge next, *shrinking strategies*, i.e., when and how to shrink transition systems, the *free pruning* that is used to remove dead states, and *exact label reduction*, which is a way to reduce labels without information loss.

2.4.4 Merging Strategy

Without shrinking and label reduction, merging a set of transition systems in any order yields the equal synchronized product. However, when merging is interleaved with shrinking, pruning and label reduction, as in the M&S process, different merging orders can produce different results. A *merging*



(a) A linear merge tree



(b) A non-linear merge tree

Figure 2.3: Linear and non-linear merge trees for 5 variables.

strategy determines, at each M&S iteration, which two transition systems in the current abstraction pool should be merged next. The sequence of merge choices form a merging order that can be represented by a *merge tree*.

Definition 21 (Merge Tree)**.** Let Θ be a transition system with variables $\mathcal{V}$, and let $V \subseteq \mathcal{V}$. A *merge tree* $\mathcal{T}$ over V is a merge-and-shrink tree using only operations (A) and (M) defined in Definition 19.

A merge tree is *linear* if it corresponds to a merging order in which at most one non-atomic abstraction exists in $\mathcal{P}$ at any time during the M&S process. This happens if we only merge an atomic projection with the non-atomic abstraction, after merging two atomic projections in the first iterative step of M&S. A *non-linear* merge tree is a tree that is not linear. Figure 2.3 shows both a linear merge true and a non-linear merge tree. A merging strategy is *linear* if it only produces linear merge trees, and is *non-linear* otherwise.

Linear Merging Strategies

All published linear merging strategies [HHH07; NHH11] and some non-linear merging strategies use information from the *causal graph* [Hel04; Kno94] of a planning task to help derive a merging order. The causal graph reveals dependencies between variables.

Definition 22 (Variable Dependency)**.** Let Π be a planning task with variable set $\mathcal{V}$ and action set $\mathcal{A}$. Let $u, v \in \mathcal{V}$. The *causal dependency* of v on u, denoted

as $w_{\text{causal}}(u, v)$, is the number of actions $\langle \text{pre}, \text{eff} \rangle \in \mathcal{A}$ such that $u \in V_{\text{pre}}$ and $v \in V_{\text{eff}}$. The *effect dependency* of v on u, denoted as $w_{\text{effect}}(u, v)$, is the number of actions $\langle \text{pre}, \text{eff} \rangle \in \mathcal{A}$ such that $u \in V_{\text{eff}}$ and $v \in V_{\text{eff}}$.

Definition 23 (Causal Graph)**.** Let Π be a planning task with variable set $\mathcal{V}$ and action set $\mathcal{A}$. Let $w(u, v) = w_{\text{causal}}(u, v) + w_{\text{effect}}(u, v)$ for $u, v \in \mathcal{V}$. The *causal graph* of Π is a directed graph $\langle \mathcal{V}, \mathcal{E} \rangle$ such that $(u, v) \in \mathcal{E}$ if and only if $u \neq v$ and $w(u, v) > 0$. The weight of a causal graph edge $(u, v) \in \mathcal{E}$ is $w(u, v)$.

A total order on n variables determines a unique sequence $(v_1, v_2, v_3, ..., v_n)$ of the variables, which in turn gives a linear merging order $\pi_{v_1} \otimes \pi_{v_2} \otimes \pi_{v_3} \otimes \cdots \otimes \pi_{v_n}$. The *level order* of variables [Hel06] is a total order on the variables based on the causal graph.

Definition 24 (Level Order of Variables)**.** Let Π be a planning task with variable set $\mathcal{V}$. Let $m \in \mathbb{N}$ and $(\langle V_1, E_1 \rangle, \langle V_2, E_2 \rangle, ..., \langle V_m, E_m \rangle)$ be the *topologically sorted* strongly connected components of the causal graph of Π. We compute a total order $\prec_{\texttt{LVL}}$ on $\mathcal{V}$ such that

- For $0 \leq i < j \leq m$, $u \prec_{\texttt{LVL}} v$ if $u \in V_i$ and $v \in V_j$.
- For each $0 \leq i \leq m$, we specify $\prec_{\texttt{LVL}}$ on variables in V_i. Set $V' := V_i$ and $E' := E_i$. Then, iteratively apply the following steps until V' and E' are empty:
  1. Pick a variable v with the minimal total weight of *incoming* edges in $\langle V', E' \rangle$, i.e., $v \in \arg\min_{v \in V'} \sum_{(u,v) \in E'} w(u, v)$. Set $v \prec_{\texttt{LVL}} u$ for all $u \in V' \setminus \{v\}$.
  2. Remove v from V' and all incident edges of v from E', i.e., $V' := V' \setminus \{v\}$ and $E' := E' \setminus \{(u, v) \in E'\} \setminus \{(v, u) \in E'\}$.

For $v \in \mathcal{V}$, $level(v) = |\{u \mid v \succ_{\texttt{LVL}} u\}|$ is called the level of v. The level order of variables in Π is the ordering of variables from the highest level to the lowest level.

Variable level is used in all existing linear merging strategies:

- The strategy `LVL` [HHH07] derives the linear merging order from the level order of variables.
- The strategy `ReverseLevel` or `RL` for short [NHH11] derives the linear merging order from the *reversed* level order of variables.
- The strategy `CausalGraph/Goal-Level` or `CGGL` for short [HHH07] iteratively selects variables using information about goal variables and the causal graph, and variable level for tie-breaking. For each iteration, let U be the set of unselected variables and P be the predecessors of selected variables in the causal graph.
  1. If $U \cap P \neq \emptyset$, select the variable in $U \cap P$ with the highest level (the `CausalGraph` rule).
  2. Otherwise, select the goal variable in U with the highest level (the `Goal` rule).

Non-Linear Merging Strategies

A natural way to derive a non-linear merging order is to select a pair of abstractions from the abstraction pool at each M&S iteration. Such approaches often use a sequence of *merge scoring functions* to select the pair of abstractions with the minimal score. Each scoring function is used to break ties among the best candidates selected by the previous scoring function. All scoring functions assume that smaller scores are better. We call such a merging strategy *scoring-based*.

Definition 25 (Merge Scoring Function)**.** Let $\mathcal{TS}$ denote the set of transition systems that share a label set and a cost function. A *merge scoring function* is a mapping $f_{ms} : \mathcal{TS} \times \mathcal{TS} \mapsto \mathbb{R}$.

All existing scoring-based merging strategies follow a sequence in which the first scoring function is a function called the *goal-relevance* function, and the last scoring function is a function that defines a total order of merge candidates

for tie-breaking, and between them are the *main* scoring functions that make the scoring-based merging strategies different from each other.

The *goal-relevance* function considers a transition system better if the system contains both goal and non-goal states.

Definition 26 (Goal-Relevance Merge Score)**.** Let Θ be a transition system with states S and goal states S_*. We say Θ is *goal-relevant* if $S \setminus S_* \neq \emptyset$. Let Θ_1 and Θ_2 be two transition systems. The *goal-relevance merge score* of Θ_1 and Θ_2 is

$$f_{ms}^{\mathrm{GR}}(\Theta_1, \Theta_2) = \begin{cases} 0, & \text{if either } \Theta_1 \text{ or } \Theta_2 \text{ is goal-relevant} \\ 1, & \text{otherwise} \end{cases}$$

The last scoring function is based on a total order `CNRL` to eliminate any ties of previous scores. `CNRL` is short for "Composite", "New", and "Reverse Level", the three preference rules shown to perform best for tie-breaking in practice [SWH16].

Definition 27 (Total Order `CNRL`)**.** `CNRL` is a total order $\prec_{\texttt{CNRL}}$ on a set of M&S abstractions derived using the following three preference rules:

- "Composite" Preference: for any non-atomic (composite) abstraction Θ and any atomic projections π_v, $\Theta \prec_{\texttt{CNRL}} \pi_v$.
- "New" Preference: for two non-atomic abstractions Θ and Θ', $\Theta \prec_{\texttt{CNRL}} \Theta'$ if Θ is produced later than Θ' in M&S.
- "Reverse Level" Preference: for two atomic projections π_u and π_v, $\pi_u \prec_{\texttt{CNRL}} \pi_v$ if the level of u is lower than the level of v.

As merging is commutative, we only need to consider the ordered pairs of abstractions (Θ, Θ') such that $\Theta \prec_{\texttt{CNRL}} \Theta'$. The total order $\prec_{\texttt{CNRL}}$ on M&S abstractions induces a total order $\prec$ on the ordered pairs: $(\Theta_1, \Theta_2) \prec (\Theta_3, \Theta_4)$ if $\Theta_1 \prec_{\texttt{CNRL}} \Theta_3$, or $\Theta_1 = \Theta_3$ and $\Theta_2 \prec_{\texttt{CNRL}} \Theta_4$. The pairs of abstractions are sorted by $\prec$ and their indexes after sorting are used as scores for them.

Other than the goal-relevance function and the total-order function, we call the merge scoring functions the main merge scoring functions. `DFP` [DFP06;

DFP09] is a scoring-based non-linear merging strategy used for M&S. Its main merge scoring function f_{ms}^{DFP} is based on a *ranking* of the labels of a transition system.

Definition 28 (DFP Label Ranking [DFP06])**.** Let Θ be a transition system with transitions T and labels L. The *ranking* of a label $l \in L$ is

$$r(l, \Theta) = \min\{\mathbf{h}_\Theta(s) \mid \exists\, t \text{ such that } t \xrightarrow{l} s \in T\} \tag{2.1}$$

Lower label ranking indicates there is a transition with this label which appears closer to a goal state.

Definition 29 (DFP Merge Score [DFP06])**.** Let Θ_1 and Θ_2 be two transition systems with label set L. The *DFP merge score* of Θ_1 and Θ_2 is

$$f_{ms}^{\text{DFP}}(\Theta_1, \Theta_2) = \min\{\max\{r(l, \Theta_1), r(l, \Theta_2)\} \mid l \in L\}$$

2.4.5 Shrinking Strategy

A shrinking strategy determines when and how to shrink a transition system. Definition 30-32 introduce some desired properties of an abstraction produced by shrinking.

Definition 30 (**h**-preserving Abstraction [HHH07])**.** Let α be an abstraction mapping for Θ with state space S. We say α is **h**-preserving if $\mathbf{h}_\Theta(s) = \mathbf{h}_\Theta(t)$ for any $s, t \in S$ such that $\alpha(s) = \alpha(t)$. The transition system induced by α is an **h**-preserving abstraction of Θ.

An **f**-preserving abstraction [HHH07] is an **h**-preserving abstraction that additionally preserves **g**-values.

Definition 31 (**f**-preserving Abstraction [HHH07])**.** Let α be an abstraction mapping for Θ with state space S. We say α is **f**-preserving if $\mathbf{h}_\Theta(s) = \mathbf{h}_\Theta(t)$ and $\mathbf{g}_\Theta(s) = \mathbf{g}_\Theta(t)$ for any $s, t \in S$ such that $\alpha(s) = \alpha(t)$. The transition system induced by α is an **f**-preserving abstraction of Θ.

f-preserving shrinking is targeted at an **f**-preserving abstraction of the transition system. Note that **h**-preserving shrinking and **f**-preserving shrinking

do not necessarily produce the perfect heuristic because they only preserve the **h**-values of the transition system to be shrunk (not the original transition system of the planning task). *Bisimulation abstraction*, however, can produce the perfect heuristic.

Definition 32 (Bisimulation Abstraction [NHH11])**.** Let α be an abstraction mapping for $\Theta = \langle S, L, \mathcal{C}, T, s_{\text{init}}, S_* \rangle$. We say α is a bisimulation abstraction if for $s, t \in S$ such that $\alpha(s) = \alpha(t)$,

- either $s, t \in S_*$ or $s, t \notin S_*$, and
- for any $l \in L$, if $s \xrightarrow{l} s' \in T$ then there exists t' such that $t \xrightarrow{l} t' \in T$ and $\alpha(s') = \alpha(t')$.

There exists a unique *coarsest* bisimulation abstraction [Mil90], i.e., the bisimulation abstraction with the fewest states. The coarsest bisimulation abstraction can be computed by an iterative process of *refinements* of the abstraction with a single abstract state[1] [NHH11]. Each refinement "splits" abstract states that violate the bisimulation criterion in Definition 32 so that the violation is resolved, i.e., if α is the current abstraction mapping and state s and t such that $\alpha(s) = \alpha(t)$ do not satisfy the bisimulation criterion, then we refine α to α' such that $\alpha'(s) \neq \alpha'(t)$.

Shrinking strategies can be *passive* or *active* regarding whether the shrinking is triggered by a *size limit* μ. Passive shrinking strategies shrink one or both of the two factor transition systems Θ_1, Θ_2 until $|\Theta_1| \cdot |\Theta_2| \leq \mu$. **f**-preserving shrinking [HHH07] and the *non-greedy* bisimulation shrinking [NHH11] are passive shrinking strategies. If $|\Theta_1| \cdot |\Theta_2| > \mu$, passive shrinking strategies first shrink Θ_1 and/or Θ_2 to their desired abstractions $\alpha(\Theta_1)$ and/or $\alpha(\Theta_2)$. If $|\alpha(\Theta_1)| \cdot |\alpha(\Theta_2)| > \mu$, additional more aggressive shrinking will be applied to ensure μ is not exceeded. **h**-preserving shrinking is often a choice at this time.

Active shrinking strategies shrink a transition system to an abstraction with desired properties without considering the size of the transition system.

[1] In FastDownward's implementation, the bisimulation refinement process actually starts with the minimal **h**-preserving abstraction.

Greedy bisimulation shrinking [NHH11] is an active shrinking strategy that shrinks a transition system to its coarsest bisimulation abstraction. No further shrinking is applied even if the coarsest bisimulation abstraction is too large. The greedy bisiumulation shrinking continues as long as there is enough memory and time.

2.4.6 Free Pruning

In M&S, the pruning method called *free pruning* is applied whenever the algorithm is about to add a new (solvable) M&S abstraction to the pool. Free pruning removes dead states of a transition system and thus does not affect the accuracy of the M&S heuristic.

Definition 33 (Free Pruning [Hel+14])**.** Let Θ be a solvable M&S abstraction with state space S. The *free pruning* β for Θ induces a transition system $\beta(\Theta) = \langle S^\beta, L, \{s \xrightarrow{l} t \in T \mid s, t \in S^\beta\}, s_{\text{init}}, S^\beta \cap S_* \rangle$ where $S^\beta = \{s \mid s \in S$ and s is live in $\Theta\}$.

2.4.7 Exact Label Reduction

Label reduction maps a label set to another (smaller) label set. Such reductions can make a transition system more compact and are extremely important for bisimulation shrinking. In particular, label reduction is critical for obtaining polynomial size bisimulation abstractions in some planning domains [NHH11; SWH14]. All existing M&S methods use "exact" label reduction, which means that the label reduction does not compromise heuristic quality.

Definition 34 (Cost-Exact Label Reduction)**.** Let L be a label set and $\mathcal{C}$ be a cost function on L. Let τ be a label reduction for L and $\mathcal{C}$, and $\mathcal{C}^\tau$ be the cost function associated with τ. We say τ is *cost-exact* if $\mathcal{C}^\tau(\tau(l)) = \mathcal{C}(l)$ for all $l \in L$.

Definition 35 (Transition-Exact Label Reduction)**.** Let $\mathcal{P}$ be a set of transition systems which all share a label set L. Let $\tau : L \mapsto L^\tau$ be a label reduction for L, and $\mathcal{P}^\tau = \{\tau(\Theta) \mid \Theta \in \mathcal{P}\}$. Let T be the transitions of $\bigotimes \mathcal{P}$ and T^τ

be transitions of $\bigotimes \mathcal{P}^\tau$. We say τ is *transition-exact* if $s \xrightarrow{l'} t \in T^\tau$ implies $s \xrightarrow{l} t \in T$ for some l such that $\tau(l) = l'$.

A label reduction is *exact* if it is cost-exact and transition-exact.

2.5 Benchmark and Evaluation

In this thesis, we use International Planning Competition (IPC) domains for our experimental evaluations. IPC is a series of competitions that are held to empirically evaluate state of the art planning systems on a number of benchmark domains. The domains used in IPC form a large variety of planning tasks that can be used to test planning techniques.

We use A* as the search algorithm for the optimal planners in this thesis. There are two important measurements for evaluating the performance of an optimal planner. The first one is the *coverage* which is the number of planning tasks solved by a planner within certain time and memory limits. For optimal planning, it is hard to achieve even a small improvement in coverage. The second performance measurement is the number of nodes expanded by A*. Improvements of planning techniques can result in reduction of the number of node expansions by orders of magnitude. For this reason, we compare the numbers of node expansions of two planners in a log scale.

Chapter 3

Non-Linear Merging Strategies

In this chapter, we present three novel non-linear merging strategies: `UMC`, `MIASM` and `DM-HQ`. Each strategy has a unique motivation behind its design.

`UMC` is a simple non-linear merging strategy that uses causal graph information to derive a merging order. `UMC` stands for Undirected Minimum Cut. It uses minimum cuts in a modified causal graph to generate a merging order.

`MIASM` stands for Maximal Intermediate Abstraction Size Minimizing. The goal of `MIASM` is to find merging orders that exploit free pruning to avoid harmful shrinking as much as possible. Based on the pricinple of `MIASM`, a scoring-based merging strategy called *dynamic* `MIASM` (`DYN-MIASM`) was later developed by Sievers, Wehrle, and Helmert [SWH16].

Our third non-linear merging strategy `DM-HQ` is a scoring-based merging method `DM-HQ` that uses both `DYN-MIASM`'s scoring function and a new function f_{ms}^{HQ} as the main scoring functions. This function utilizes information about heuristic quality to help make merging decisions.

The sections on `UMC` and `MIASM` are based on [FMH14], and the section on f_{ms}^{HQ} and `DM-HQ` is based on [FHM18].

3.1 Introduction

In early work on M&S [HHH07; KHH12; NHH11], merging strategies are restricted to be linear because of their simplicity and the focus on shrinking strategies. In this chapter, we explore the potential of non-linear merging strategies.

We first introduce two novel non-linear merging strategies, UMC and MIASM. The only non-linear merging strategy that existed before UMC and MIASM was the DFP merging strategy, originally proposed by Dräger, Finkbeiner, and Podelski in the model checking community [DFP06; DFP09]. DFP is a scoring-based non-linear merging strategy. An advantage of scoring-based strategies is that they can be computed in $\mathcal{O}(n^3)$ time if there are n variables, but it comes at a price: such strategies can only compare *local* merge candidates, i.e., the pairs of abstractions in the current abstraction pool of M&S. Our new merging strategies UMC and MIASM break this limitation to seek good merge candidates in a larger set of non-local candidates.

UMC derives a merging order using a top-down hierarchical clustering of variables, which has a better global view than the scoring-based merging strategies. The hierarchical clustering is developed by recursively using a minimum cut in the causal graph to split variable sets. Such cuts can be computed in polynomial time in n.

MIASM performs a guided global search before the start of M&S to find subsets of variables which, if merged, yield a large number of *dead* states—those not on any solution path of a transition system. Such states can be pruned freely, without compromising the quality of the heuristic. The global search in MIASM takes many more merging candidates into consideration than the scoring-based methods.

The exploitation of free pruning makes MIASM a strong merging strategy. It lays the foundation for a competitive scoring-based merging strategy called DYN-MIASM [SWH16] and our third new merging strategy DM-HQ developed on top of DYN-MIASM. The novel idea behind DM-HQ is a new merge scoring function f_{ms}^{HQ} that evaluates the heuristic quality improvement of current merge candidates. Because the ultimate goal of M&S is to build a heuristic of high quality, the addition of f_{ms}^{HQ} to DYN-MIASM improves the number of tasks that can be solved on several standard test domains, and makes it the currently best performing merging strategy when used alone [FHM18].

We introduce UMC, MIASM and DM-HQ and show their experimental results in Sections 3.2, 3.3, and 3.4 respectively.

3.2 UMC: Merging Using the Minimum Cuts of Causal Graphs

The motivation of UMC is to develop a merge order that prioritizes the merge of strongly dependent variables. This dependency information can be obtained from the causal graphs. Unlike some previous usage of causal graphs, for UMC we do not care about the direction of the dependency, so we use *undirected* causal graphs:

Definition 36 (Undirected Causal Graph)**.** Let Π be a planning task with variable set $\mathcal{V}$. Let $w(u,v) = w_{\text{causal}}(u,v) + w_{\text{causal}}(v,u) + w_{\text{effect}}(u,v)$ for $u, v \in \mathcal{V}$. The *undirected causal graph* of Π is an undirected graph $\langle \mathcal{V}, \mathcal{E} \rangle$ such that $(u,v) \in \mathcal{E}$ if and only if $u \neq v$ and $w(u,v) > 0$. The weight of the edge $(u,v) \in \mathcal{E}$ is $w(u,v)$.

Since an undirected causal graph edge weight only measures the dependency between two variables connected by the edge, we do not have a direct measurement on the dependency among more than two variables or between two variables that are not connected by an edge. Thus, instead of looking for collections of most intra-dependent variables, we look for *partitioning* of variables into *least inter-dependent* subsets. As the merging operation is binary, a two-way partitioning fits our purpose of developing a merging order, which relates naturally to *cuts* of a graph.

Definition 37 (Graph Cut)**.** Let $G = \langle V, E \rangle$ be an undirected graph. A *cut* of G is a partition $\langle V_1, V_2 \rangle$ of V such that $V_1 \cap V_2 = \emptyset$, $V_1 \neq \emptyset$, $V_2 \neq \emptyset$ and $V_1 \cup V_2 = V$. The *cut edges* are those $(v_1, v_2) \in E$ such that $v_1 \in V_1$ and $v_2 \in V_2$.

Since we want the least inter-dependent subsets and smaller edge weights in the undirected causal graph mean weaker dependency, we are interested in the minimum cut of a graph.

Definition 38 (Minimum Cut)**.** The *weight* of a cut is the sum of the weight of its edges. A *minimum cut*, or *min-cut* for short, is a cut with the minimum weight among all cuts.

Algorithm 4 Recursive M&S Procedure of UMC: **UMC**

Input: variable set $V \subseteq \mathcal{V}$, induced subgraph G of $\mathcal{G}^{\texttt{UMC}}$ on V, and $\mathcal{P}$
Output: transition system $\Theta_{1,2}$ over V

1: **if** $|V| = 1$ **then**
2: **return** $\pi_v \in \mathcal{P}$ for the $v \in V$
3: **end if**
4: $\langle V_1, V_2 \rangle \leftarrow$ **ComputeMinCut**(G)
5: $\Theta_1 \leftarrow$ **UMC**$(V_1, \mathcal{G}^{\texttt{UMC}}_{\langle V_1 \rangle}, \mathcal{P})$
6: $\Theta_2 \leftarrow$ **UMC**$(V_2, \mathcal{G}^{\texttt{UMC}}_{\langle V_2 \rangle}, \mathcal{P})$
7: $\mathcal{P} \leftarrow$LabelReduction$(\Theta_1, \Theta_2, \mathcal{P})$
8: $(\Theta_1, \Theta_2) \leftarrow$Shrinking$(\Theta_1, \Theta_2)$
9: $\Theta_{1,2} \leftarrow \Theta_1 \otimes \Theta_2$
10: $\Theta_{1,2} \leftarrow$CheckSolvabilityAndPruning$(\Theta_{1,2})$
11: $\mathcal{P} \leftarrow \mathcal{P} \cup \{\Theta_{1,2}\} \setminus \{\Theta_1, \Theta_2\}$
12: **return** $\Theta_{1,2}$

With a min-cut $\langle V_1, V_2 \rangle$, a reasonable merging decision is to construct M&S abstractions Θ_{V_1} and Θ_{V_2} separately first and then merge Θ_{V_1} and Θ_{V_2}, because the dependency between variables in V_1 and variables in V_1 is the weakest among all possible ways to separate $V_1 \cup V_2$. The min-cut for a variable set corresponds to the *last* merging operation for producing the M&S abstraction over the variable set. We can further split V_1 (and V_2) using a min-cut of the subgraph of the undirected causal graph induced by V_1 (and V_2). UMC obtains a complete merging order by recursively using min-cut partitioning until a variable set contains only one variable.

UMC considers goal variables more important and prefers a cut whose edges connect non-goal variables. In UMC, we add an artificial large constant c to (v_1, v_2) if v_1 or v_2 is a goal variable. Here, c is equal to the total weight of all edges in the undirected causal graph. This penalty ensures that a cut whose edges connect only non-goal variables always has a smaller weight than a cut whose edges are incident to goal variables. We denote this modified undirected causal graph used by UMC as $\mathcal{G}^{\texttt{UMC}}$, and the induced subgraph of $\mathcal{G}^{\texttt{UMC}}$ on variable set V as $\mathcal{G}^{\texttt{UMC}}_{\langle V \rangle}$.

Because UMC uses only the causal graph information, the complete merge tree can be computed without performing any M&S operations. The merge

Algorithm 5 Merge-and-Shrink Using Merging Strategy UMC

Input: task Π with variables $\mathcal{V}$
Output: M&S abstraction $\Theta^{\texttt{UMC}}_{\text{M\&S}}$
1: $\mathcal{G}^{\texttt{UMC}} \leftarrow$ BuildCausalGraphUMC(Π)
2: $\mathcal{P} \leftarrow$**InitMAS**(Π)
3: $\Theta^{\texttt{UMC}}_{\text{M\&S}} \leftarrow$**UMC**($\mathcal{V}$, $\mathcal{G}^{\texttt{UMC}}$, $\mathcal{P}$)
4: **return** $\Theta^{\texttt{UMC}}_{\text{M\&S}}$

tree then controls the merge decisions of M&S. However, a natural and simple way to implement a top-down merging strategy is to use a recursive function. Such a function **UMC** is shown in Algorithm 4. The complete M&S algorithm for UMC is shown in Algorithm 5. For a task with variable set $\mathcal{V}$, $\mathcal{G}^{\texttt{UMC}}$ is constructed and $\mathcal{P}$ is initialized first, then the call to **UMC** with $\mathcal{V}$, $\mathcal{G}^{\texttt{UMC}}$ and $\mathcal{P}$ returns the M&S abstraction produced by using the UMC merging strategy. Note that in Algorithm 4, the steps of label reduction, shrinking, pruning and updating $\mathcal{P}$ (Lines 7-11) are exactly the same as in the bottom-up merge-and-shrink Algorithm 3.

The **ComputeMinCut** function implements a simple undirected min-cut algorithm developed by Stoer and Wagner (1997). The worst-case run-time complexity of the algorithm on a graph with n vertices and m edges is $\mathcal{O}(nm + n^2 \log n)$. If there exist multiple min-cuts, **ComputeMinCut** returns the first min-cut found.

3.2.1 Example of UMC in Action

Now we show how UMC produces a merge tree in the example of Figure 3.1. In the example, the variable set $\mathcal{V}$ consists of 6 variables $v_1, v_2, v_3, v_4, v_5, v_6$. Variables v_1 and v_2 are goal variables, shown in double circles in the graphs in Figure 3.1. Figure 3.1(a) shows the original UMC causal graph $\mathcal{G}^{\texttt{UMC}}$ in which $c = 16$. Starting with the whole set $\mathcal{V}$ (Figure 3.1(b)), we find the min-cut of $\mathcal{G}^{\texttt{UMC}}$ (highlighted in gray in Figure 3.1(c)). UMC uses this min-cut to separate $\mathcal{V}$ into two subsets $U = \{v_5, v_6\}$ and $V = \{v_1, v_2, v_3, v_4\}$. This split corresponds to the last merge step of the merge tree that UMC is constructing (Figure 3.1(d)). Next, UMC needs to construct the merge trees for U and V

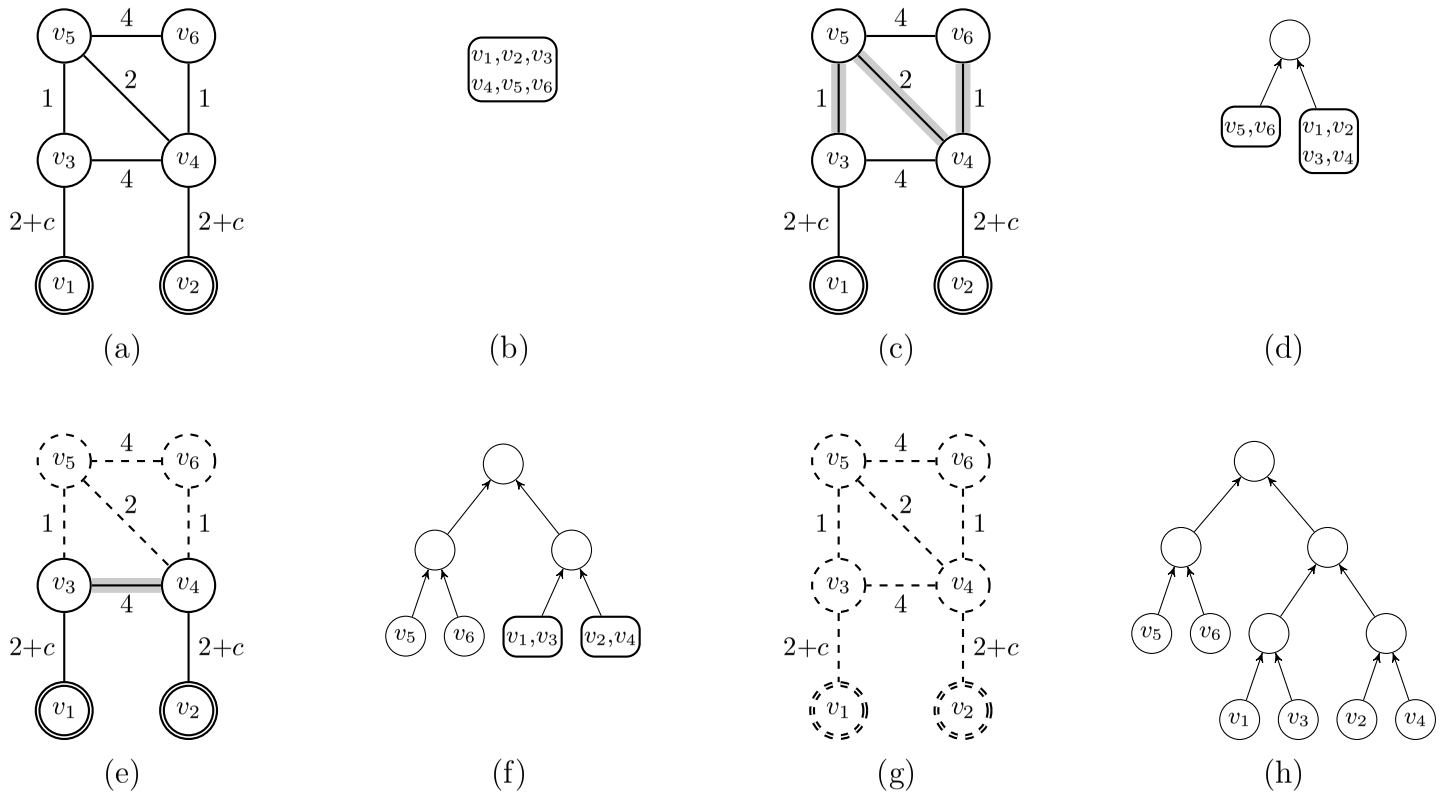


Figure 3.1: Example of UMC merge. (a) The causal graph $\mathcal{G}^{\texttt{UMC}}$; (b) The variable set $\mathcal{V}$; (c) The min-cut of $\mathcal{G}^{\texttt{UMC}}$; (d) UMC merge splits $\mathcal{V}$ into two subsets $U = \{v_5, v_6\}$ and $V = \{v_1, v_2, v_3, v_4\}$ according to the min-cut of $\mathcal{G}^{\texttt{UMC}}$; (e) The min-cut of $\mathcal{G}^{\texttt{UMC}}_{\langle V \rangle}$; (f) UMC merge splits V to subsets $\{v_1, v_3\}$ and $\{v_2, v_4\}$; (g) Merge decisions for all variables are defined; (h) The complete merge tree constructed by UMC.

Shrinking Strategy	Bisimulation				**f**-preserving			
Merging Strategy	DFP	RL	CGGL	UMC	DFP	RL	CGGL	UMC
Total Coverage	**449**	**449**	419	438	**346**	318	330	336
Minimum Expansions	182	183	179	**206**	167	166	195	**196**

Table 3.1: Coverage for DFP, RL, CGGL and UMC with non-greedy bisimulation shrinking and **f**-preserving shrinking.

respectively. U contains only two variables, so there is only one possible cut $\langle\{v_5\}, \{v_6\}\rangle$. The min-cut of $\mathcal{G}^{\texttt{UMC}}_{\langle V\rangle}$, highlighted in grey in Figure 3.1(e), splits V into $\{v_1, v_3\}$ and $\{v_2, v_4\}$, shown in Figure 3.1(f). Both $\{v_1, v_3\}$ and $\{v_2, v_4\}$ contain only two variables, so there is only one cut for each set. Combining these steps, we obtain the complete merge tree in Figure 3.1(h).

3.2.2 Experiments

We compare UMC with previous merging strategies RL, CGGL and DFP. We evaluate the merging methods in combination with non-greedy bisimulation shrinking [NHH11] and with **f**-preserving shrinking [HHH07]. The M&S size limit μ is 50,000 states. Generalized label reduction [SWH14] is used. All algorithms related to UMC are implemented on the top of Fast Downward [Hel06].

We ran experiments on all domains from IPC 1998 to IPC 2011 except four domains: Movie, MPrime, Storage and Trucks. For domains that appear in both IPC 2008 and IPC 2011, we only use the IPC 2011 problems. This benchmark set has 33 domains and 1051 tasks. The experiments were performed on an AMD Opteron 6134 CPU with a clock speed of 2.3 GHz. The memory and the total CPU time for a planner to solve a task are limited to 2 GB and 15 minutes respectively.

Coverage

The coverage for UMC and previous merging strategies are shown in Table 3.1. Row "Total Coverage" shows the total coverage. UMC solves 18 more tasks than RL when using **f**-preserving shrinking, but solves 11 fewer tasks than RL when non-greedy bisimulation shrinking is used. For both shrinking strategies, UMC outperforms CGGL but solves fewer tasks than DFP.

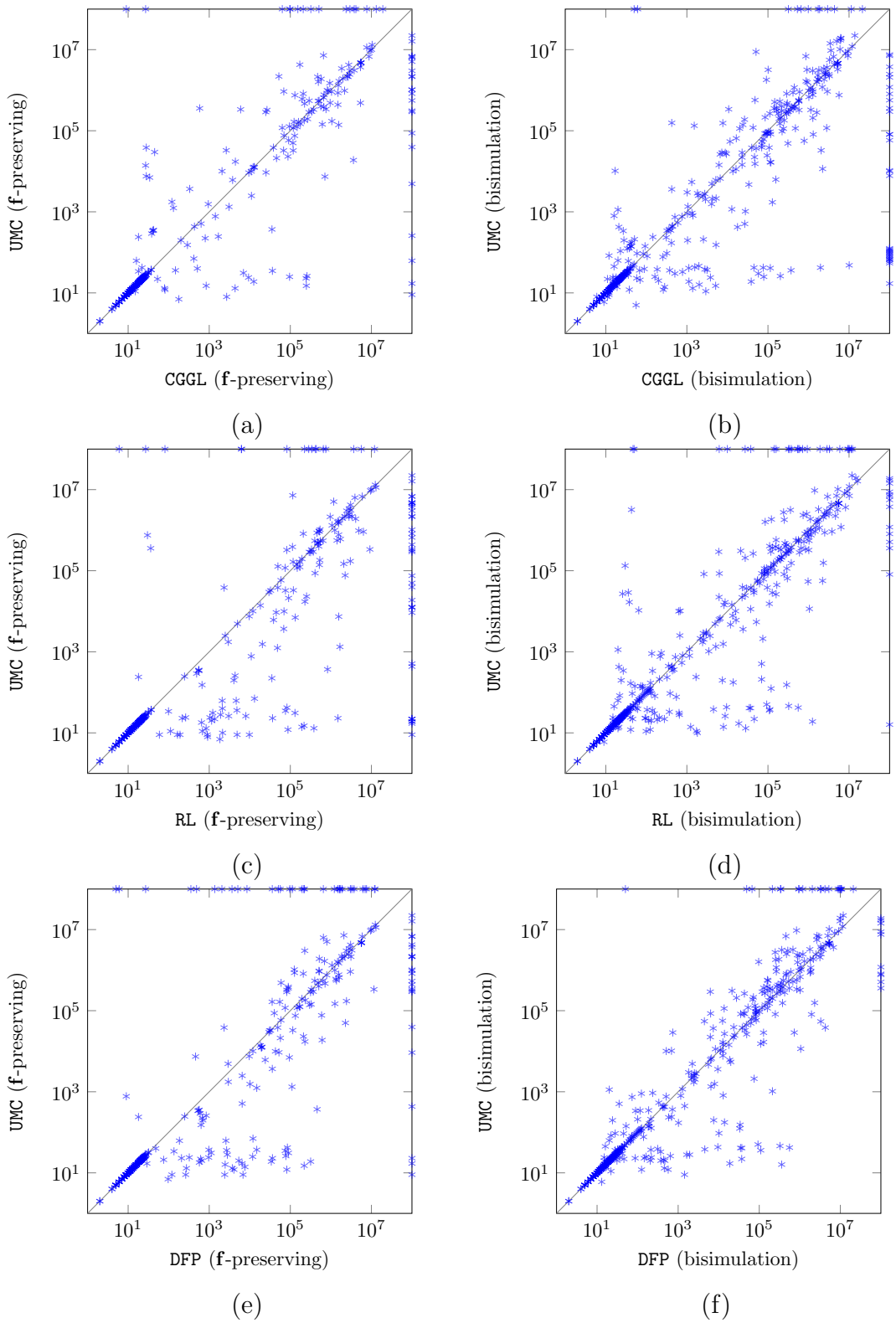


Figure 3.2: Comparison of UMC with CGGL, RL and DFP in terms of number of A* node expansions. The left three plots use **f**-preserving shrinking and the right three plots use bisimulation shrinking.

Row "Minimum Expansions" in Table 3.1 shows the total number of tasks that are solved with the minimum number of expansions, i.e., the number of expansions is the same as the length of the optimal solution path. This quantity indicates the number of tasks for which the heuristic constructed has a high quality in terms of reducing the search effort to the minimum. For both shrinking strategies, UMC solves the most tasks with the minimum number of expansions.

Number of Node Expansions

The plots in Figure 3.2 compare UMC with CGGL, RL and DFP in the number of node expansions. Each plot uses two *log-scale* axes to compare the numbers of node expansions of two methods. We call the method labeled below the x-axis, the "x method", and the method labeled to the left of the y-axis, the "y method". Each point in the plot corresponds to one planning task. The x value of the point is the number of nodes expanded to solve the corresponding task using "x method", and the y value of the point is the number of nodes expanded to solve the corresponding task using "y method". In Figure 3.2, UMC is the "y method". Points below the $y = x$ line represent tasks that required more nodes to be expanded using "x method" than using "y method", and points above the $y = x$ line represent tasks for which "x method" does a fewer node expansions than "y method". Points on the right-most (respectively, top-most) edge of the plot are tasks that were not solved using method label with x-axis (respectively, y-axis) within the resource limits.

In these plots, we see that no system dominates in general. In Figure 3.2(a) and Figure 3.2(d), the point distributions are almost symmetric about the diagonal $y = x$. In other plots, we see more points below the diagonal than above it, indicating that, overall, UMC expands fewer nodes than previous systems. For all the merging methods, the number of expansions is reduced when bisimulation is used instead of f-preserving shrinking, but some methods benefit more than others, with RL benefiting most.

3.3 Minimizing the Maximum Intermediate Abstraction Size

In this section, we introduce another new non-linear merging strategy: the *maximum intermediate abstraction size minimizing*, or `MIASM` merging strategy. `MIASM` is based on the motivation that it is more important to do free pruning, i.e., reveal and prune the dead states, in smaller abstractions than in larger ones. In particular, the merging orders developed by `MIASM` prioritize the merge of the smallest sets of variables that induce the most free pruning, so that smaller but equally informative M&S abstractions can be constructed early on. This results in less shrinking later, and thus can produce better heuristics.

In Section 3.3.1, we illustrate this motivation behind `MIASM` with a simplified example from IPC domain TPP. Then, we present the `MIASM` merging algorithm in Section 3.3.2, followed by experimental results in Section 3.3.3.

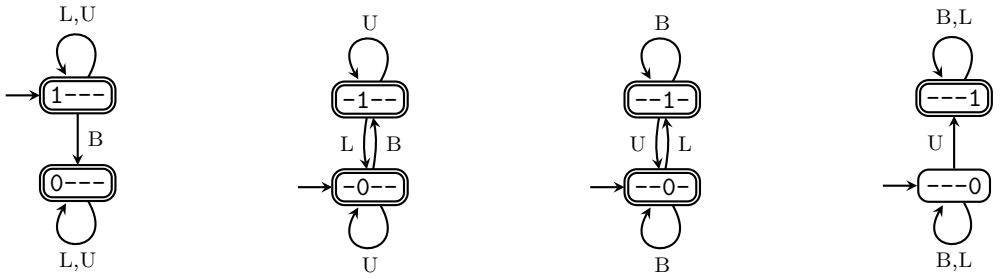
3.3.1 Motivation

Shrinking is used to control abstraction sizes and larger abstractions can invoke more shrinking. If we produce smaller but equally informative M&S abstractions during a M&S process, we may avoid more lossy shrinking and produce a better heuristic. Free pruning is a way to produce "smaller but equally informative" M&S abstractions. In this section, we use an example to demonstrate this motivation behind `MIASM`, namely, using merging to trigger more free pruning and less shrinking, and finally produce better heuristics.

A TPP Example

The example is from the IPC domain TPP, short for Travelling Purchase Problem. This domain models purchasing different commodities from markets, transporting them to depots and storing them in the depots. Each commodity is associated with four binary variables O, P, L and S, indicating the four conditions of the commodity:

- whether it is on-sale at a market ($O = 1$) or not ($O = 0$),



(a) π_O (b) π_P (c) π_L (d) π_S

Figure 3.3: The atomic projections of variables O, P, L and S of a commodity for the simplified TPP domain. The abstract inital state is marked with an incoming edge not from any other states and abstract goal states are marked as double ovals.

- whether it is purchased at a market ($P = 1$) or not ($P = 0$),
- whether it is loaded in a truck ($L = 1$) or not ($L = 0$), and
- whether it is stored in the depot ($S = 1$) or not ($S = 0$).

There are three actions BUY, LOAD and UNLOAD that change the conditions of a commodity. For illustration purposes, we simplify the task so that there is only one market, one depot and one truck, which are all at the same location. With this simplification, the actions that change the conditions of a commodity only depend on the current condition of the commodity.

- If a commodity is on-sale but not purchased ($O = 1, P = 0$), BUY changes it to purchased but not on-sale ($O = 0, P = 1$).
- If a commodity is purchased but not loaded ($P = 1, L = 0$), LOAD changes it to loaded but not purchased ($P = 0, L = 1$).
- If a commodity is loaded but not stored ($L = 1, S = 0$), UNLOAD changes it to stored but not loaded ($L = 0, S = 1$).

In a TPP task, each commodity is initially on-sale ($O = 1, P = 0, L = 0, S = 0$) and the goal is to store it ($S = 1$). We use a bit vector of length 4 to represent the values of O, P, L and S. For example, `0100` means $O = 0$, $P = 1$, $L = 0$ and $S = 0$. For each commodity, we represent abstract states

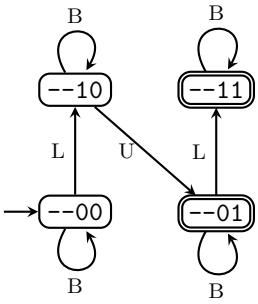
in a projection over a subset of $\{O, P, L, S\}$ in the same format but with "-" replacing the variables projected out. For example, `-1-0` is an abstract state in the projection over $\{P, S\}$ with $P = 1$, $S = 0$, and O and L projected out. Figure 3.3 shows the atomic projections on O, P, L and S of one commodity.

Merge and Prune

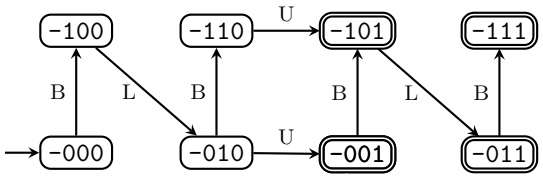
We now illustrate in this simplified TPP example that merging can affect free pruning and sizes of intermediate abstractions. We ignore shrinking and label reduction, and call a merge-and-shrink tree constructed using only merging and free pruning a *merge-and-prune* (M&P) tree. Since there is no shrinking, different merge-and-prune trees over the same variables have the same final abstractions. However, as we will demonstrate, different merge-and-prune trees can produce different *intermediate* abstractions and can have different maximum intermediate abstraction sizes.

Figure 3.3 shows that the atomic projections have no dead states. Consider merging the atomic projections in the linear order $(((\pi_S \otimes \pi_L) \otimes \pi_P) \otimes \pi_O)$. Let $\Theta_1 = \pi_S$, $\Theta_2 = \pi_S \otimes \pi_L$, $\Theta_3 = \pi_S \otimes \pi_L \otimes \pi_P$, and $\Theta_4 = \pi_S \otimes \pi_L \otimes \pi_P \otimes \pi_O$. The synchronized products Θ_2, Θ_3 and Θ_4 are shown in Figure 3.4. Because a commodity always is in exactly one of the four conditions, any state in Θ_4 whose bit vector does not have exactly one `1` is a dead state. However, all states in Θ_2 and Θ_3 are live due to the abstractions. Let β denote the function that maps a transition system to its pruned transition system induced by free pruning.

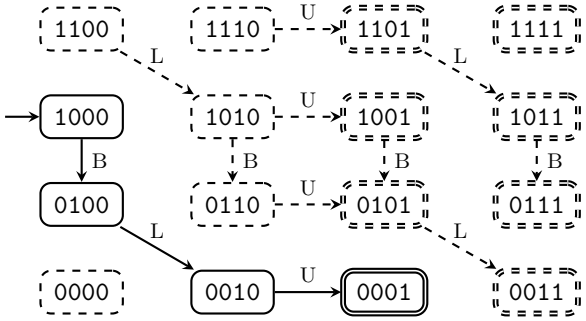
Let V_{others} be a set of variables that are independent of O, P, L and S, e.g., a set of variables representing another commodity. Let Θ' be a M&S abstraction over V_{others} with n states and no dead states. We consider two orders for merging Θ' with the atomic projections over O, P, L and S : $\pi_S \otimes \pi_L \otimes \pi_P \otimes \pi_O \otimes \Theta'$ and $\Theta' \otimes \pi_S \otimes \pi_L \otimes \pi_P \otimes \pi_O$. In both orders, free pruning is applied whenever possible. In the first order, atomic projections over O, P, L and S are merged first, then their product after free pruning $\beta(\Theta_4)$ is merged with Θ'. The merge-and-prune tree $\mathcal{T}_1$ of this merging order is shown in Figure 3.5(a). The number next to a node indicates the size of the associated M&S abstraction.



(a) $\Theta_2 = \pi_S \otimes \pi_L$

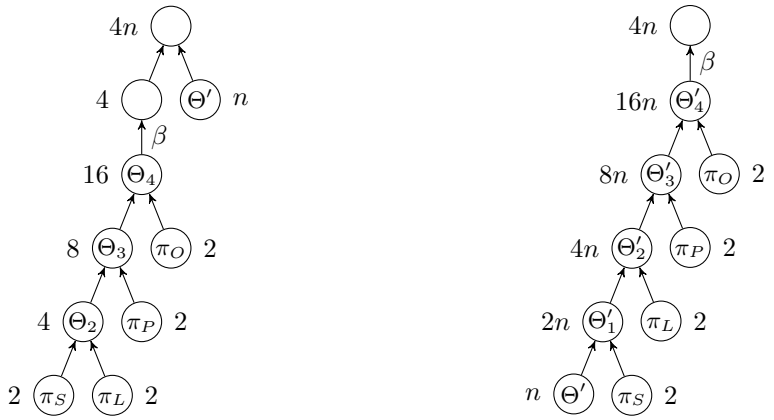


(b) $\Theta_3 = \pi_S \otimes \pi_L \otimes \pi_P$



(c) $\Theta_4 = \pi_S \otimes \pi_L \otimes \pi_P \otimes \pi_O$

Figure 3.4: Synchronized products (a) $\Theta_2 = \pi_S \otimes \pi_L$, (b) $\Theta_3 = \pi_S \otimes \pi_L \otimes \pi_P$ and (c) $\Theta_4 = \pi_S \otimes \pi_L \otimes \pi_P \otimes \pi_O$. The dead states are drawn in dashed ovals.



(a) $\mathcal{T}_1 : \beta(\pi_S \otimes \pi_L \otimes \pi_P \otimes \pi_O) \otimes \Theta'$ (b) $\mathcal{T}_2 : \beta(\Theta' \otimes \pi_S \otimes \pi_L \otimes \pi_P \otimes \pi_O)$

Figure 3.5: The merge-and-prune trees of the merging orders: (a) $\pi_S \otimes \pi_L \otimes \pi_P \otimes \pi_O \otimes \Theta'$; (b) $\Theta' \otimes \pi_S \otimes \pi_L \otimes \pi_P \otimes \pi_O$. The number next to a node indicates the size of the associated M&S abstraction.

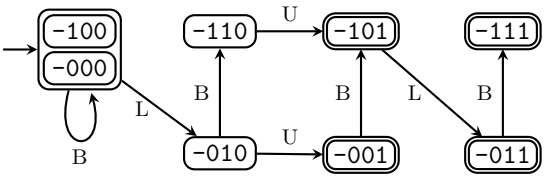
We see $|\mathcal{T}_1|_{\max} = \max(16, 4n)$.

The second order differs from the first one in that Θ' is moved to the front to merge with π_S first, then π_L, π_P and π_O in the same order as in the first order. Let $\Theta'_1 = \Theta' \otimes \pi_S$, $\Theta'_2 = \Theta' \otimes \pi_S \otimes \pi_L$, $\Theta'_3 = \Theta' \otimes \pi_S \otimes \pi_L \otimes \pi_P$ and $\Theta'_4 = \Theta' \otimes \pi_S \otimes \pi_L \otimes \pi_P \otimes \pi_O$. Since variables in V_{others} are independent of O, P, L and S, abstractions Θ'_1, Θ'_2 and Θ'_3 do not contain dead states. Because of the associative property of merging, Θ'_4 is equal to $\Theta_4 \otimes \Theta'$ which contains $12n$ dead states that are "copies" of the 12 dead states in Θ_4. Those dead states can only be detected after Θ'_4 is produced. The merge-and-prune tree $\mathcal{T}_2$ of this merging order is shown in Figure 3.5(b). Both orders generate the same final M&P abstraction with the $4n$ states, but the second order must generate Θ'_4 with its $16n$ states. As a result, $|\mathcal{T}_2|_{\max} = 16n$.

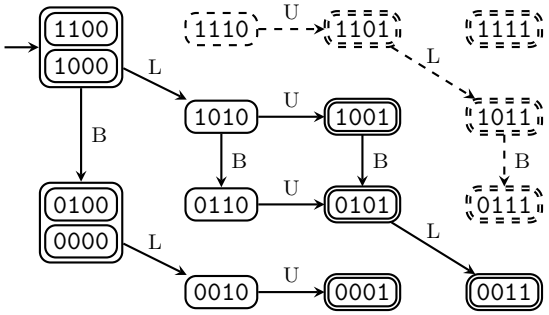
For any $n > 3$, $|\mathcal{T}_2|_{\max} - |\mathcal{T}_1|_{\max} = 12n$. A typical size limit of M&S abstractions is $50,000$ states, so there could be a large difference in the abstraction sizes of the merging orders if n reaches this size limit.

Lossy Shrinking

So far, we have restricted merge-and-shrink to use only merging and free pruning. In practice, there are memory and time limits and/or passive shrinking.



(a) $\alpha(\Theta_3)$



(b) $\alpha(\Theta_3) \otimes \pi_O$

Figure 3.6: Illustration of how shrinking reduces free pruning. (a) Shrinking α that combines `-100` and `-000` in Θ_3; (b) $\alpha(\Theta_3) \otimes \pi_O$ has only five dead states.

If no shrinking or only an active shrinking strategy is used, a merging order producing larger abstractions may run out of computational resources quickly while a merging order that produces smaller abstractions may succeed.

When passive shrinking is used, larger abstraction sizes can trigger more shrinking to keep the size under control. There are two detrimental effects of shrinking. First, unlike free pruning, shrinking can be lossy and often result in lower accuracy of heuristics[1]. A huge difference in abstraction sizes, as in our example, can induce a significant difference in M&S heuristic quality. Second, shrinking can result in combinations of dead states with live states. For example, consider the shrinking operation α that combines `-100` and `-000` in Θ_3 (Figure 3.4(b)), that induces the abstraction $\alpha(\Theta_3)$, shown in Figure 3.6(a). The combination of `-100` and `-000` in Θ_3 induces the combinations of `1100` with `1000` and of `0100` with `0000` in $\alpha(\Theta_3) \otimes \pi_O$, shown in Figure 3.6(b). These combinations cause many dead states in $\Theta_3 \otimes \pi_O$ to become live in $\alpha(\Theta_3) \otimes \pi_O$. Only 5 states can be pruned freely in $\alpha(\Theta_3) \otimes \pi_O$ after Θ_3 is shrunk by α. What is worse, the lossy shrinking and pruning forms a negative

[1]Sometimes smaller abstractions can produce more accurate heuristics. See Chapter 4.

Algorithm 6 Merge-and-Shrink Using Merging Strategy `MIASM`

Input: task Π with variables $\mathcal{V}$
Output: M&S abstraction
1: $\mathcal{F} \leftarrow$ **SubsetSearch**(Π)
2: $Singletons \leftarrow \{\{v\} \mid v \in \mathcal{V}\}$
3: $\mathcal{F} \leftarrow \mathcal{F} \cup Singletons$
4: $\mathcal{F}_{\text{partition}} \leftarrow$ **MaxSetPacking**($\mathcal{F}$)
5: **return MS-Partition**(Π, $\mathcal{F}_{\text{partition}}$, $Internal$, $External$)

feedback loop in which lossy shrinking reduces the amount of free pruning that can be done, which in turn produces larger abstractions and thus triggers more lossy shrinking.

3.3.2 Merging Strategy `MIASM`

The motivating example shows that if there is a subset of variables over which the M&S abstraction contains many dead states, it can be beneficial to merge their atomic projections first before merging them with other M&S abstractions.

The `MIASM` merging strategy is designed to find such variable subsets and use them for developing merging orders. The algorithm of `MIASM` is shown in Algorithm 6. Unlike other merging strategies, `MIASM` has a preprocessing step before starting a M&S construction. This preprocessing first finds a family of variable subsets over which the M&P abstractions *reveal* dead states (Line 1) and then computes a partition of the variables that contains subsets that enable most free pruning overall (Line 4). At last, a merging order is developed based on this partition (Line 5). In the following, we describe each of these three steps of `MIASM`.

Subset Search

If the projection over a variable subset V contains dead states, it does not necessarily mean that it is the smallest abstraction in which the dead states can be pruned. In the TPP example shown in Section 3.3.1, the projection Θ'_4 over $V_{\text{others}} \cup \{O, P, L, S\}$ does contain dead states, but these dead states

are just refinements of dead states in the projection Θ_4 over $\{O, P, L, S\}$, i.e., dead states in Θ_4 are abstract states of the dead states in Θ'_4.

We denote the $\mathbf{R}$-value of the projection over V as $\mathbf{R}_V$. We want to find a variable subset V such that not only is $\mathbf{R}_V < 1$ but also $\mathbf{R}_V < \mathbf{R}_U \cdot \mathbf{R}_{V \setminus U}$ for all non-empty proper subsets U of V. We call a variable subset such as V one that *reveals* dead states.

Definition 39 (Revealing Dead States)**.** Let $\mathcal{V}$ be the variables of a planning task and let $V \subseteq \mathcal{V}$ and $V \neq \emptyset$.

$$\mathbf{P}^*_V = \begin{cases} \mathbf{R}_V & \text{if } |V| = 1 \\ \min\limits_{U \subset V,\ U \neq \emptyset} \mathbf{R}_U \cdot \mathbf{R}_{V \setminus U} - \mathbf{R}_V & \text{otherwise} \end{cases}$$

We say V *reveals* dead states if $\mathbf{P}^*_V > 0$, and we call $\mathbf{P}^*_V$ the $\mathbf{P}^*$-value of V.

While $(1 - \mathbf{R}_V)$ measures the amount of free pruning that can be done on the projection over V, $\mathbf{P}^*_V$ measures the *minimum* amount of free pruning that can be done on a M&P abstraction over V considering all possible ways for building the M&P abstraction. $\mathbf{P}^*$-values take the pruning before building a M&P abstraction over V into consideration.

If $\mathbf{P}^*_V = 0$, it means that all dead states in π_V can be pruned in the abstractions over some subsets of V. To see this, let $|V| > 1$ and $\mathbf{P}^*_V = 0$, and let U be a non-empty proper subset of V such that $\mathbf{R}_U \cdot \mathbf{R}_{V \setminus U} = \mathbf{R}_V$ (by definition, there must exist such a subset U if $\mathbf{P}^*_V = 0$). Let $\Theta_V = \beta(\pi_V)$, $\Theta_U = \beta(\pi_U)$ and $\Theta_{V \setminus U} = \beta(\pi_{V \setminus U})$ where β is the free pruning operator. The synchronized product $\Theta_U \otimes \Theta_{V \setminus U}$ is an abstraction over V. Since $|\Theta_U \otimes \Theta_{V \setminus U}| = |\pi_U| \cdot \mathbf{R}_U \cdot |\pi_{V \setminus U}| \cdot \mathbf{R}_{V \setminus U} = |\pi_U| \cdot |\pi_{V \setminus U}| \cdot \mathbf{R}_V = |\pi_V| \cdot \mathbf{R}_V = |\Theta_V|$, $\Theta_U \otimes \Theta_{V \setminus U}$ does not contain any dead states. If we first build Θ_U and $\Theta_{V \setminus U}$ and then merge them, we produce a M&P abstraction over V that is isomorphic to Θ_V and all dead states in π_V have been pruned as soon as we have built Θ_U and $\Theta_{V \setminus U}$.

There are exponentially many variable subsets, and checking whether a subset reveals dead states requires construction of M&P abstractions. Therefore, we use a best-first search to look for subsets that reveal dead states. We

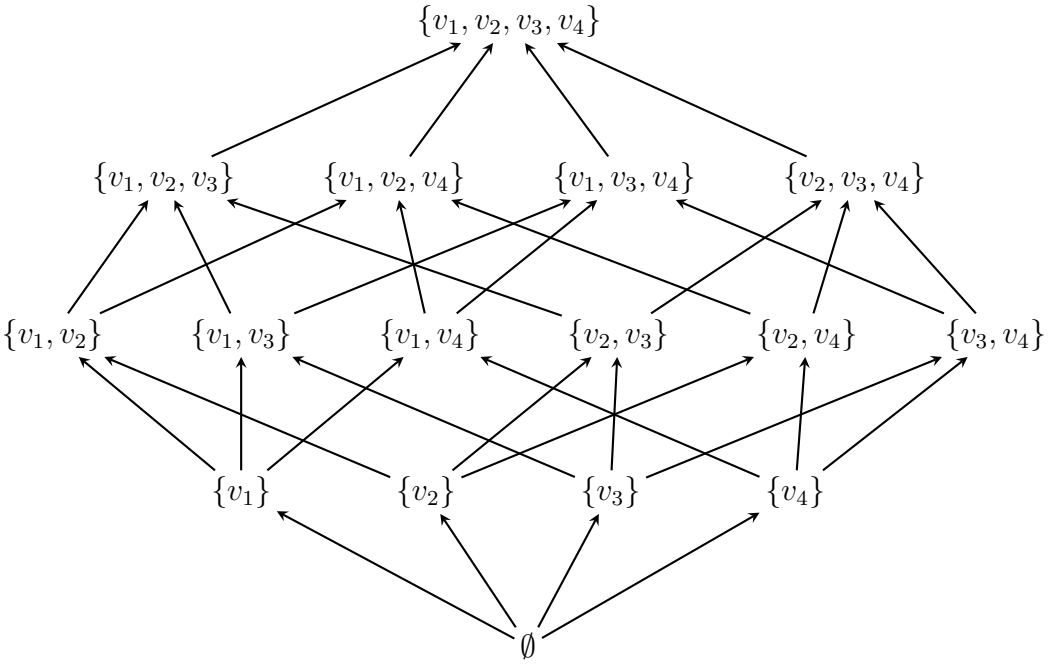


Figure 3.7: The lattice space of four variables v_1, v_2, v_3, v_4.

carry out the search in the lattice space of subsets—a state space in which states are subsets and the successors of a set are the supersets of the set with exactly one more variable. The lattice space of four variables is shown in Figure 3.7.

Algorithm 7 shows the **SubsetSearch** algorithm. In the following, we explain some important design choices of this search algorithm.

Quick Start (Line 1). The search starts with singleton sets of variables of the task. Since we expand subsets by adding only one variable at a time, it may take a long time to reach subsets that reveal dead states. We add two groups of "promising" subsets to the initial priority queue:

- The sets of variables that form a strongly connected component in the causal graph.
- The sets of variables whose atoms form mutex groups detected in the translation process from PDDL to SAS$^+$ [Hel06].

Expansion Priority (Line 5). The best-first search uses a priority queue PQ in which subsets with larger $\mathbf{P}^*$-value or equal $\mathbf{P}^*$-value but smaller size

Algorithm 7 SubsetSearch

Input: task Π with variables $\mathcal{V}$
Output: a family of subsets that reveal dead states

```
1: PQ ← PromisingSubset(Π) ∪ {{v} | v ∈ V}
2: Checked ← ∅
3: F ← ∅
4: while PQ ≠ ∅ do
5:     U ← GetBestSubset(PQ)
6:     PQ ← PQ \ U
7:     for v ∈ V \ U do
8:         V ← U ∪ {v}
9:         if V ∉ Checked then
10:            Checked ← Checked ∪ {V}
11:            ComputeAndCacheValues(V)
12:            if SizeLimitsNotViolated then
13:                PQ ← PQ ∪ {V}
14:            end if
15:            if P*_V > 0 then
16:                F ← F ∪ {V}
17:            end if
18:        end if
19:    end for
20: end while
21: return F
```

(number of variables) have higher priority. We use the $\mathbf{P}^*$-value, instead of the $\mathbf{R}$-value, to guide the search because the $\mathbf{R}$-value can mislead the search to subsets that do not reveal dead states. For example, let $\mathbf{R}_{\{v_1\}} = \mathbf{R}_{\{v_2\}} = 2/3$, $\mathbf{R}_{\{v_3\}} = 5/9$ and $\mathbf{R}_{\{v_1,v_2\}} = 4/9$. When both $\{v_1, v_2\}$ and $\{v_3\}$ are in the priority queue, if we use the $\mathbf{R}$-value to prioritize expansions, $\{v_1, v_2\}$ will be expanded before $\{v_3\}$ since $\mathbf{R}_{\{v_1,v_2\}} < \mathbf{R}_{\{v_3\}}$. However, $\{v_1, v_2\}$ does not reveal dead states because $\mathbf{R}_{\{v_1,v_2\}} = \mathbf{R}_{\{v_1\}} \cdot \mathbf{R}_{\{v_2\}}$ and $\mathbf{P}^*_{\{v_1,v_2\}} = 0$, but $\{v_3\}$ reveals dead states because $\mathbf{P}^*_{\{v_3\}} = \mathbf{R}_{\{v_3\}} = 5/9$.

Subset Caching (Line 11). In order to compute $\mathbf{P}^*_V$, we need to compute their $\mathbf{R}_V$ and $\mathbf{R}_U$ for all non-empty sets $U \subset V$. We using merging strategy `LVL` to build the M&P abstraction on a new variable subset. To avoid re-building of M&P abstractions, after we get the $\mathbf{R}$-value and $\mathbf{P}^*$-value of a subset, we cache the values for later use. We do not cache the actual abstractions built

because there is a large number of them, so we would run out of memory quickly during the subset search.

Bounding and Termination (Line 12). We want to find as many subsets V with $\mathbf{P}^*_V > 0$ as possible, so the search does not terminate with some goal condition. To avoid exhaustively searching the lattice space, we do not add a subset V to the queue if

1. the M&P abstraction over V is larger than an abstraction size limit μ, or

2. the total number of states in abstractions constructed during the subset search has exceeded a bound parameter μ_{total}.

We use condition 1 because for abstractions larger than μ it is uncertain whether it is beneficial to prioritize their construction. In the actual merge-and-shrink process, there could be shrinking in constructing these abstractions, so the dead states are not guaranteed to be pruned. We use Condition 2 to indirectly control the amount of exploration of the subset search. The search terminates when the priority queue is empty, and outputs a family $\mathcal{F}$ of variable subsets that reveal dead states.

Max Set Packing

Because merge-and-shrink needs to merge all variables and can merge a variable only once, we want to find a partition of the variable set $\mathcal{V}$ of the task, i.e., disjoint subsets whose union is $\mathcal{V}$. $\mathcal{F}$ consists only of subsets that reveal dead states, so it is possible that some variable does not appear in any subset in $\mathcal{F}$. We add all singleton sets to $\mathcal{F}$ to ensure that every variable is in at least one subset in $\mathcal{F}$ (Line 3 in Algorithm 6).

Since a variable can appear in multiple subsets in $\mathcal{F}$, we need to choose which subset to use for it. In particular, we want to find a partition $\mathcal{F}_{\text{partition}} \subseteq \mathcal{F}$ of $\mathcal{V}$ that suggests the most amount of free pruning overall, i.e., minimizing $\prod_{V \in \mathcal{F}_{\text{partition}}} \mathbf{R}_V$. Note that the product only indicates an upper bound of the percentage of live states (i.e., a lower bound of free pruning) because "new"

dead states may appear after merging two abstractions that are free of dead states.

Since $\mathbf{R}_V \leq 1$ for any subset V, $-\log(\mathbf{R}_V)$ is a non-negative real number. Minimizing $\prod_{V \in \mathcal{F}_{\text{partition}}} \mathbf{R}_V$ is equal to maximizing $\sum_{V \in \mathcal{F}_{\text{partition}}} (-\log(\mathbf{R}_V))$. If we use $-\log(\mathbf{R}_V)$ as the weight of V, this is exactly the *maximum weighted set packing problem*: given a family of sets, each of which is a subset of a universe and has an associated real weight, find a subfamily of disjoint sets of maximum total weight. The problem is NP-hard [GJ79]. The standard approach is a simple greedy algorithm:

1. Choose a $V \in \mathcal{F}$ with maximum weight and add it to $\mathcal{F}_{\text{partition}}$.
2. Remove all subsets that intersect with V from $\mathcal{F}$.
3. Repeat 1 and 2 until $\mathcal{F}$ is empty.

The greedy algorithm approximates the optimal solution within a factor of k for arbitary weights, where k is the maximum size of subsets in the family [CH01]. **MaxSetPacking** at Line 4 of Algorithm 6 implements this algorithm, which takes $\mathcal{F}$ as input and outputs a partition $\mathcal{F}_{\text{partition}} \subseteq \mathcal{F}$ of $\mathcal{V}$ that minimizes $\prod_{V \in \mathcal{F}_{\text{partition}}} \mathbf{R}_V$. Other approximate algorithms for this problem can be found in [AH97; CH01].

Partition Based Merging Strategy

With the partition $\mathcal{F}_{\text{partition}}$ of $\mathcal{V}$, we can develop a *partition based* merging strategy. This merging strategy first merges atomic projections of variables within V for each $V \in \mathcal{F}_{\text{partition}}$, which produces a merge-and-shrink abstraction Θ_V over V for each $V \in \mathcal{F}_{\text{partition}}$. Then, Θ_V for $V \in \mathcal{F}_{\text{partition}}$ are merged. We call the merging strategy for merging the atomic projections for each variable subset the *Internal* strategy, and the merging strategy for merging M&S abstractions over subsets in the partition the *External* strategy. Merge-and-shrink using this partition based merging strategy is implemented as **MS-Partition** (Line 5 in Algorithm 6).

Shrinking Strategy	Bisimulation				**f**-preserving			
Merging Strategy	DFP	RL	CGGL	MIASM	DFP	RL	CGGL	MIASM
Total Coverage	**449**	**449**	419	446	346	318	330	**357**
Min Expansion	182	183	179	**217**	167	166	195	**225**

Table 3.2: Coverage for DFP, RL, CGGL and MIASM with non-greedy bisimulation shrinking and **f**-preserving shrinking.

In our study, we use LVL for the *Internal* strategy, the same simple merging strategy used in constructing abstractions during subset search. For the *External* merging strategy, we use CGGL+, a variant of the CGGL merging strategy generalized to work on variable subsets. We define the level of a variable set V as $\max_{v \in V} level(v)$. For each selection iteration of CGGL+, let $\mathcal{F}_U$ be the family of unselected variable subsets and $\mathcal{F}_P$ be the family of subsets that contains causal graph predecessors of variables in selected variable sets.

1. If $\mathcal{F}_U \cap \mathcal{F}_P \neq \emptyset$, select the smallest variable set in $\mathcal{F}_U \cap \mathcal{F}_P$ with the highest level.

2. Otherwise, select the smallest variable set in $\mathcal{F}_U$ that contains a goal variable and has the highest level.

3.3.3 Experiments

The experimental setting for MIASM is the same as for UMC in the previous section. The bound μ_{total} on the total number of states of the constructed abstractions in the subset search is set to 1,000,000. The total time of MIASM includes the time for the subset search, maximum set packing and for building a M&S abstraction heuristic based on the partition $\mathcal{F}_{\text{partition}}$ computed, and running A* search on the task.

A special case for $\mathcal{F}_{\text{partition}}$ is that it contains only the singleton sets. This happens when the subset search did not find any subsets revealing dead states. In this case, the merging strategy is the same as the *External* merging strategy, which is CGGL+ in our study and is equivalent to CGGL because all sets are singleton. We define "singleton" tasks as tasks for which MIASM finishes subset search and maximum set packing but produces such a singleton set $\mathcal{F}_{\text{partition}}$.

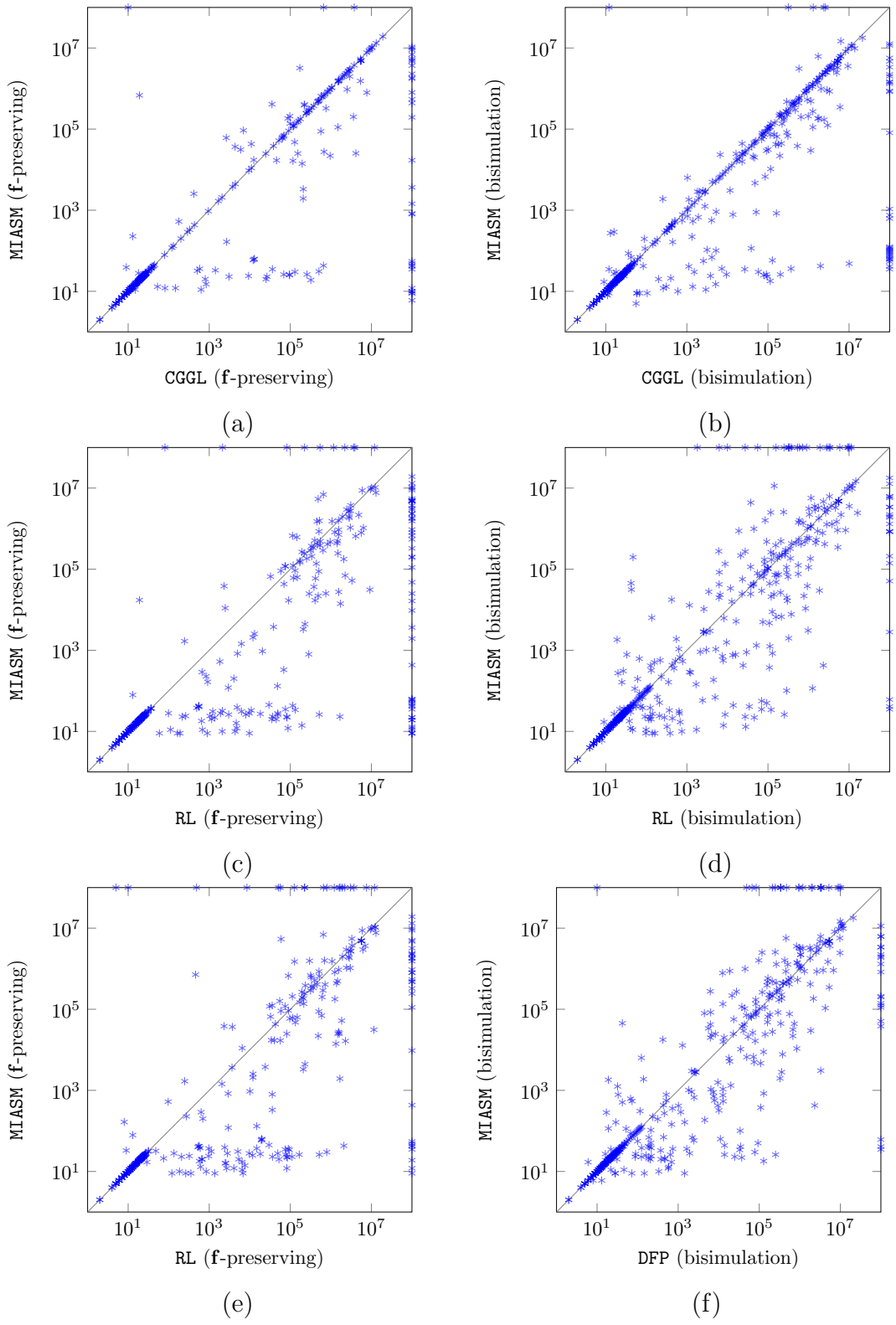


Figure 3.8: Comparison of MIASM with CGGL, RL and DFP in terms of number of node expansions. The left three plots use **f**-preserving shrinking and the right three plots use bisimulation shrinking.

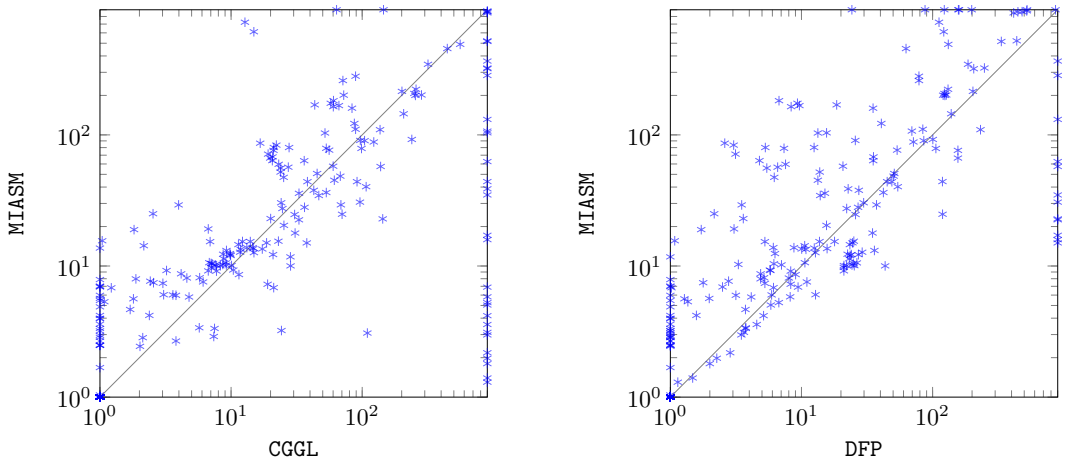


Figure 3.9: Comparison of MIASM with CGGL and DFP in total running time, when bisimulation shrinking is used.

Coverage

The coverage data are shown in Table 3.2. For **f**-preserving shrinking, MIASM outperforms all other methods in total coverage. For bisimulation shrinking, MIASM solves 446 tasks in total, only 3 fewer than DFP and RL. There are 184 "singleton" tasks in our experiments. For these tasks, our MIASM specification falls back to CGGL. Alternatively, we can switch from MIASM to another merging strategy for these tasks. If we switch from MIASM to either DFP or RL for "singleton" tasks, the coverage increases to 451.

Row "Min Expansions" in Table 3.2 shows that in terms of the number of tasks that are solved with the minimum number of expansions, MIASM outperforms other methods by a large margin of more than 30 more tasks.

Number of Expansions

Figure 3.8 is the same as Figure 3.2 except with MIASM as the "y method". In Figure 3.8(b)-(c) and Figure 3.8(e), the clusters of points with y values between 10^1 and 10^2 but x values range from 10^3 to the right edge represent tasks that are solved by MIASM expanding fewer than 100 nodes but require orders of magnitude more node expansions by the previous methods. In Figure 3.8(d) and Figure 3.8(f), we see that when bisimulation shrinking is used, MIASM is

complementary to DFP and RL in terms of which tasks each solves, as there are a great number of tasks one method solves that the other does not (19 and 16 points on top and right edges of Figure 3.8(d) respectively and 18 and 15 points on top and right edges of Figure 3.8(f) respectively).

Total Time

MIASM's subset search can be slow as it involves building a large number of M&S abstractions, computing **h**-values and **g**-values, and applying free pruning. The plots in Figure 3.9 compare MIASM with CGGL and DFP in total running time when bisimulation shrinking is used. In these plots, we see that for the most commonly solved tasks, MIASM's total running time is higher than the previous methods, even though the number of node expansions for MIASM is lower as shown in Figure 3.8.

UMC and MIASM

We have presented two novel non-linear merging strategy UMC and MIASM. UMC builds a merge tree using a top-down clustering based causal graph information, and MIASM extracts the information about the amount of free pruning that can be done when merging a set of variables and uses that information to make merging decisions. Both algorithms have superior strength over previous methods. MIASM outperforms UMC in terms of total coverage, making it the state-of-the-art (at the time of its publication). In the following section, we will introduce another new non-linear merging strategy developed based on the central idea of MIASM.

3.4 Heuristic Quality Guided Merging

The goal of M&S is to construct a final heuristic of high quality. Merging strategies in the current literature work towards this goal indirectly using information derived from transition systems and task descriptions. In this section, we present a merge scoring function f_{ms}^{HQ} that is aimed at achieving this goal directly.

In Section 3.4.1, we give the background of two non-linear merging strategies SCC-DFP and DYN-MIASM published after UMC and MIASM. In Section 3.4.2, we present the generic form of our new scoring function f_{ms}^{DM} and three instantiations. Experiments show that these instantiations alone do not outperform DYN-MIASM or SCC-DFP. In Section 3.4.3, we integrate f_{ms}^{HQ} with DYN-MIASM, and show that in this combination, one of the variants of DM-HQ achieves better performance than the start of the art merging strategy SCC-DFP.

3.4.1 Related Work

SCC-DFP [SWH16] is a variant of DFP which uses strongly connected components (SCC's) of the causal graph as a general guideline before applying the DFP strategy. It can be viewed as a partition based merging strategy in which the variable partition is the set of SCC's of the causal graph and both the *Internal* and *External* merging strategies use DFP.

Another competitive non-linear merging strategy is DYN-MIASM which is a scoring-based variant of the MIASM merging strategy [SWH16]. It uses the **R**-value of the product of a merge candidate as the score of the candidate:

Definition 40 (Dynamic MIASM Merge Score)**.** Let Θ_1 and Θ_2 be two transition systems, and let $\Theta_{1,2}$ be the synchronized product of possibly shrunk abstractions of Θ_1 and Θ_2. The *Dynamic MIASM merge score* of (Θ_1, Θ_2) is

$$f_{ms}^{\mathrm{DM}}(\Theta_1, \Theta_2) = \mathbf{R}(\Theta_{1,2})$$

DYN-MIASM and SCC-DFP as well as previous merging strategies CGGL, LVL, RL, DFP and MIASM were evaluated on a benchmark set including all domains for optimal planning from IPC-1998 to IPC-2014 [SWH16]. The time limit was 30 minutes. Other experiment settings are the same as in Section 3.3. In this experimental setting, MIASM was shown to have better performance than all other merging strategies except SCC-DFP. There are tasks which appear in more than one IPC set. For example, all IPC-2011 WoodWorking tasks are included in IPC-2008, and 10 TidyBot tasks appear in both the IPC 2011 and 2014. There are also 7 unsolvable tasks from domain mystery. In this

section, we use the same benchmark domains as in [SWH16], but exclude duplicate tasks and unsolvable tasks. Our benchmark contains a total of 1499 tasks from 39 domains.

We use the same limits as in [SWH16]: 30 minutes time and 2 GB memory in total for heuristic construction and search, and 50,000 states limit for abstraction size. Experiments are performed on Intel Xeon X5670 CPUs at 2.93GHz. In this section, we compare our methods with `SCC-DFP`, the current state of the art merging strategy at the time of publication.

3.4.2 Scoring Heuristic Quality Improvement

The generic form of our new merge score function is:

$$f_{ms}^{\mathrm{HQ}}(\Theta_1, \Theta_2) = -\mathbf{I_Q}(\Theta_{1,2}, \Theta_1, \Theta_2)$$

where Θ_1 and Θ_2 are the two transition systems of a merge candidate and $\Theta_{1,2}$ is the synchronized product of Θ_1 and Θ_2, after possibly shrinking them first. $\mathbf{Q}$ is a heuristic quality evaluator and $\mathbf{I}$ is an improvement evaluator. The negative sign before the improvement evaluator is to ensure that larger improvements get smaller scores, just to follow the convention that smaller scores mean better candidates. We now discuss why we design our scoring function in this form and what evaluation functions we can choose for $\mathbf{Q}$ and $\mathbf{I}$.

How to Evaluate Heuristics?

There are many ways to evaluate a heuristic. For example, one could use the average heuristic values of a set of sampled states, or an estimation of the search effort when using the heuristic. Like `DYN-MIASM`, our scoring function depends on the product transition system $\Theta_{1,2}$ produced by merging and possibly shrinking Θ_1 and Θ_2 for each candidate. This makes the whole M&S process very time-consuming. To avoid additional computational overhead, we simply use the heuristic value of the initial state as the heuristic quality evaluator. The initial state's heuristic value is often a reasonable indicator of the number of A* node expansions, although it may not be as accurate

as other evaluators. For a transition system Θ, the initial heuristic evaluator $\mathbf{Q}_0$ uses the $\mathbf{h}$-value of the initial state to evaluate the heuristic quality, i.e., $\mathbf{Q}_0(\Theta) = \mathbf{h}_\Theta(s_{\text{init}})$ where s_{init} is the initial state in Θ.

Why Evaluate Improvements?

Our evaluator $f_{ms}^{\text{HQ}}(\Theta_1, \Theta_2)$ aims to measure an "improvement of heuristic quality", rather than measuring heuristic quality of the synchronized product alone, i.e., $f_{ms}^{\text{HQ}}(\Theta_1, \Theta_2)$ is not defined to be just $\mathbf{Q}(\Theta_{1,2})$. If we use an evaluation of heuristic quality of the synchronized product only, we may end up with a merge strategy that always prefers to merge large transition systems, whose product gives a high-quality heuristic simply due to its large size. This tendency to merge large transition systems may result in a linear merge strategy where a dominant transition system keeps drawing other transition systems in. It seems an unfair bias to directly compare small and large transition systems produced in an M&S process. Instead of evaluating the heuristic quality of synchronized products directly, we evaluate the improvements of heuristic quality that result from merging two transition systems.

How to Evaluate Improvement?

Since we evaluate heuristic quality by heuristic $\mathbf{Q}_0$ scores, we can evaluate the heuristic quality improvement by how much the heuristic values increase after merging. Note that before computing $\Theta_{1,2}$, shrinking Θ_1 and Θ_2 may be needed. Since this shrinking is always $\mathbf{h}$-preserving, $\mathbf{Q}_0(\Theta_{1,2}) \geq \max(\mathbf{Q}_0(\Theta_1), \mathbf{Q}_0(\Theta_2))$. There are several ways to define how much of an increase $\mathbf{Q}_0(\Theta_{1,2})$ represents over $\mathbf{Q}_0(\Theta_1)$ and $\mathbf{Q}_0(\Theta_2)$.

We considered three evaluators:

1. $\mathbf{I}^{+}_{\mathbf{Q}_0}(\Theta_{1,2}, \Theta_1, \Theta_2) = \mathbf{Q}_0(\Theta_{1,2}) - (\mathbf{Q}_0(\Theta_1) + \mathbf{Q}_0(\Theta_2))$.
2. $\mathbf{I}^{\max}_{\mathbf{Q}_0}(\Theta_{1,2}, \Theta_1, \Theta_2) = \mathbf{Q}_0(\Theta_{1,2}) - \max(\mathbf{Q}_0(\Theta_1), \mathbf{Q}_0(\Theta_2))$.
3. $\mathbf{I}^{\min}_{\mathbf{Q}_0}(\Theta_{1,2}, \Theta_1, \Theta_2) = \mathbf{Q}_0(\Theta_{1,2}) - \min(\mathbf{Q}_0(\Theta_1), \mathbf{Q}_0(\Theta_2))$.

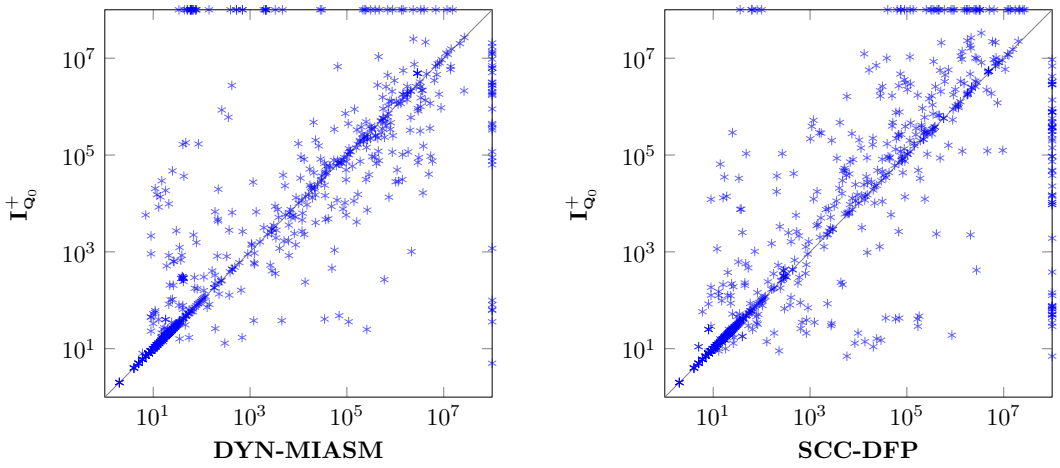


Figure 3.10: Comparing numbers of expansions by A* using different M&S heuristic: (a) $\mathbf{I}^+_{\mathbf{Q}_0}$ (y-axis) vs. `DYN-MIASM` (x-axis); (b) $\mathbf{I}^+_{\mathbf{Q}_0}$ (y-axis) vs. `SCC-DFP` (x-axis);

We run experiments of M&S methods with scoring-based merging strategies that use the three evaluators $\mathbf{I}^+_{\mathbf{Q}_0}$, $\mathbf{I}^{\max}_{\mathbf{Q}_0}$ and $\mathbf{I}^{\min}_{\mathbf{Q}_0}$ respectively (as their main scoring functions). The total coverages are 661, 618 and 648 for $\mathbf{I}^+_{\mathbf{Q}_0}$, $\mathbf{I}^{\max}_{\mathbf{Q}_0}$ and $\mathbf{I}^{\min}_{\mathbf{Q}_0}$ respectively. The best performing evaluator $\mathbf{I}^+_{\mathbf{Q}_0}$ is slightly worse than `DYN-MIASM` and `SCC-DFP`. The plots in Figure 3.10 compare the numbers of A* node expansions of $\mathbf{I}^+_{\mathbf{Q}_0}$ with that of `DYN-MIASM` and `SCC-DFP`. Despite the smaller coverage, $\mathbf{I}^+_{\mathbf{Q}_0}$ still shows some advantage over `DYN-MIASM` and `SCC-DFP`. There are 34 and 48 points on the top edges in Figure 3.10(a) and Figure 3.10(b) respectively, but there are also 29 and 48 points on the rightmost edges in Figure 3.10(a) and Figure 3.10(b) respectively. In both plots, we see many points on both sides of the diagonal line.

3.4.3 Integration with `DYN-MIASM`

We now consider the integration of f^{HQ}_{ms} and f^{DM}_{ms}. There are two motivations for combining the two. First, the two functions measure complementary information about a merging candidate. f^{HQ}_{ms} focuses on information about heuristic quality, whereas f^{DM}_{ms} only evaluates the $\mathbf{R}$-value of a synchronized product, which measures only the percentage of live states, but says nothing about the quality of the heuristic. Second, both scoring functions are product-dependent,

which means they require computing synchronized products of merge candidates before making merging decisions. This process can be very expensive. However, since we only need to compute synchronized products once to obtain both f_{ms}^{HQ} and f_{ms}^{DM} scores, combining the two functions introduces little computational overhead.

A natural way to integrate two merge scoring functions is to use one function as tie-breaker for the other. We use f_{ms}^{HQ} as tie-breaker for f_{ms}^{DM} in `DYN-MIASM`. The reason we use f_{ms}^{HQ} as tie-breaker for f_{ms}^{DM} rather than the other way around is that f_{ms}^{DM} seems superior than f_{ms}^{HQ} in the greedy process of scoring-based merging. Although f_{ms}^{HQ} is aimed at improving heuristic quality, the merging strategy is still a hill-climbing process and may end up with local optima. It turns out that f_{ms}^{HQ} can be more myopic than f_{ms}^{DM} (i.e., `DYN-MIASM`) on a number of tasks, even though the $\mathbf{R}$-value used by f_{ms}^{DM} contains no information about heuristic quality. The advantage of f_{ms}^{DM} over f_{ms}^{HQ} when used in the local greedy process of scoring-based merging is that the $\mathbf{R}$-values seem to have long-term effects while the heuristic improvement measured by f_{ms}^{HQ} is purely local. This is due to the negative feedback between shrinking and pruning (discussed in "Lossy Shrinking" of Section 3.3.1), which implies that less pruning can trigger more harmful shrinking in subsequent iterations, so even if a merge choice is only locally optimal it has benefits for future iterations. The heuristic improvement measured by f_{ms}^{HQ} does not imply good or bad effects in the long term.

With three specifications of f_{ms}^{HQ}, we now have three variants of `DYN-MIASM`: `DYN-MIASM` with tiebreaker $\mathbf{I}_{\mathbf{Q}_0}^{+}$, $\mathbf{I}_{\mathbf{Q}_0}^{\max}$ and $\mathbf{I}_{\mathbf{Q}_0}^{\min}$. The total coverage for the three are 681, 636 and 674 for `DYN-MIASM` with $\mathbf{I}_{\mathbf{Q}_0}^{+}$, $\mathbf{I}_{\mathbf{Q}_0}^{\max}$ and $\mathbf{I}_{\mathbf{Q}_0}^{\min}$ as tie-breakers respectively. `DYN-MIASM` with tie-breakers $\mathbf{I}_{\mathbf{Q}_0}^{+}$ and $\mathbf{I}_{\mathbf{Q}_0}^{\min}$ improve the coverage of `DYN-MIASM`, and the $\mathbf{I}_{\mathbf{Q}_0}^{+}$ tie-breaker has the highest coverage among the three variants. We call `DYN-MIASM` with $\mathbf{I}_{\mathbf{Q}_0}^{+}$ tiebreaker `DM-HQ` (short for Dynamic MIASM and Heuristic Quality). Figure 3.11(a) compares the number of expansions by A* using `DM-HQ` and `DYN-MIASM`. The grey zone indicates the difference is within a factor of 10. We see there are many tasks where `DM-HQ` expands one or more magnitudes fewer nodes than `SCC-DFP` (points below the

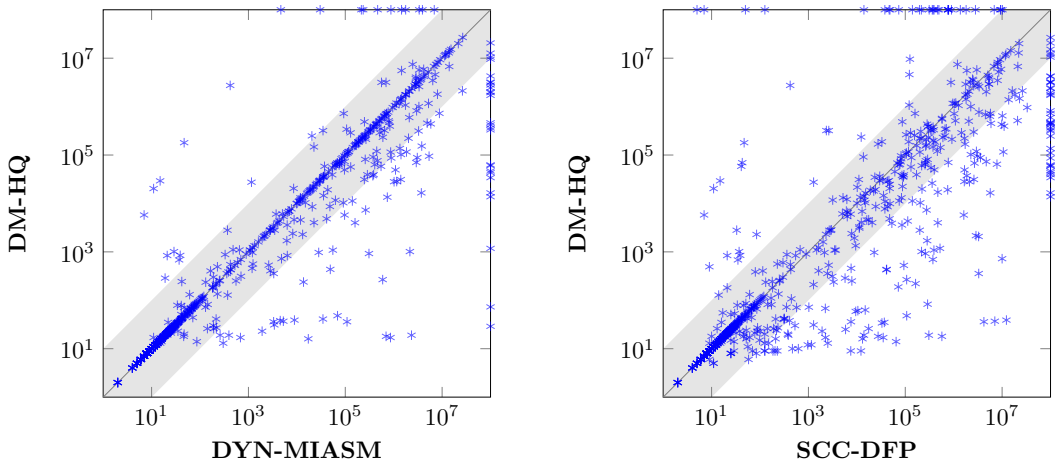


Figure 3.11: Comparison of numbers of expansions using different M&S heuristics: (a) `DM-HQ` (y-axis) vs. `DYN-MIASM` (x-axis); (b) `DM-HQ` (y-axis) vs. `SCC-DFP` (x-axis);

grey zone). On only a few tasks, `DM-HQ` expands at least 10 times more nodes than `SCC-DFP` (points above the grey zone). For the coverage difference, there are 9 points on the top edge and 24 points on the rightmost edge of the plots.

Figure 3.11(b) is same as Figure 3.11(a) but compares `DM-HQ` with `SCC-DFP`. On a large number of tasks `DM-HQ` expands one or more magnitudes fewer nodes than `SCC-DFP`, while on many fewer tasks, `SCC-DFP` expands one or more magnitudes fewer nodes than `DM-HQ`. `DM-HQ` solves 36 tasks on which `SCC-DFP` fails, but it also fails on 26 tasks that `SCC-DFP` solves. In balance, `DM-HQ` solves 10 more tasks than `SCC-DFP`.

3.5 Conclusions

In this chapter, we have presented three novel non-linear merging strategies, `UMC`, `MIASM` and `DM-HQ` based on different motivations. Experiments show that `UMC` and `MIASM` are superior to previous methods in terms of the number of expansions and the number of tasks solved with minimum numbers of expansions. The idea of `MIASM` of exploiting free pruning is shown to be effective in improving M&S heuristics. The new merging strategy `DM-HQ` adds a new merge scoring function to the scoring-based variant of `MIASM`, making it the

new state of the art standalone M&S method.

Chapter 4

MS-lite: A Lightweight, Complementary Merge-and-Shrink Method

In this chapter, we present *MS-lite*, an extremely fast merge-and-shrink method that always shrinks transition systems to their smallest **h**-preserving abstractions. MS-lite has complementary strength compared to other merge-and-shrink methods due to its construction efficiency. In addition, we observe that its simple shrinking strategy can lead to better heuristics for some planning tasks.

We exploit the complementary strength of MS-lite to enhance other merge-and-shrink methods. Our experiments show that MS-lite complements `DM-HQ` very well: their combination dramatically outperforms the current (2018) state-of-the-art merge-and-shrink method by solving 75 more tasks on an IPC benchmark set of 1499 tasks. This chapter is based on [FHM18].

4.1 Introduction

Transformations in merge-and-shrink process transition systems in their explicit representations. It can be very expensive to build an M&S heuristic for a large planning task. For example, Table 4.1 shows the M&S construction time of the state-of-the-art M&S method `SCC-DFP`[1] [SWH16] and our new

[1] `DM-HQ` was published in the same paper of MS-lite [FHM18]. `SCC-DFP` was the best M&S method in the literature before their publication. In this chapter, we refer to `SCC-DFP` as the current state of the art.

No. Var.	126	169	212	255	298
SCC-DFP	157	304	666	1059	(timeout)
MS-lite	0.1	0.2	0.5	0.6	0.9

Table 4.1: The M&S construction time (in seconds) for SCC-DFP and MS-lite on a series of tasks with increasing numbers of variables.

M&S method called MS-lite on a series of planning tasks from IPC domain Airport with increasing numbers of variables. In the row for SCC-DFP, we see that with each constant increase in the number of variables, M&S construction time roughly doubles, until the last task fails due to a 30 minute limit timeout for construction.

Previous improvements to M&S such as ours (Chapter 3) and others [HHH07; NHH11; SWH14; SWH16] focus on how to create more informative M&S heuristics. In the first part of this chapter, we shift our focus onto how to efficiently construct an M&S heuristic of reasonable quality. We propose MS-lite, a fast M&S method that shrinks every transition system to its smallest **h**-preserving abstraction. Such extreme shrinking gives MS-lite super efficiency. The row "MS-lite" in Table 4.1 shows the M&S construction time for this method. For all the Airport tasks, MS-lite takes less than 1 second for this construction. In addition, MS-lite can even create better heuristics on some tasks. Table 4.2 shows the numbers of nodes expanded by A* to solve a small task from the IPC domain Blocks using heuristics created by SCC-DFP under different abstraction size limits. The right half of Table 4.2, from column 10^4 to column 10^6, shows the expected behavior. With a larger size limit, M&S can store more information in an abstraction and produce a better heuristic. As the size limit increases from 10^4 to 10^6, the number of node expansions decreases as expected. However, for size limits less than or equal to 10^4, we see an unexpected trend where the number of A* node expansions *increases* as the size limit increases. The extreme case is column "MS-lite", where only the minimum **h**-preserving abstractions are used. In this case, the maximum abstraction size is the number of distinct **h**-values in each abstract transition system, which is at most 15 for this Blocks task.

Size Limit	MS-lite	10^2	10^3	10^4	10^5	10^6
No. Expan.	396	9,670	21,058	44,643	14,065	9,230

Table 4.2: Numbers of nodes expanded by A* using M&S heuristics constructed with different size limits.

This heuristic, which is created by our new algorithm MS-lite, is far better here than heuristics created with much larger abstractions.

Our experiments on IPC domains show that MS-lite solves many planning tasks that other M&S methods fail to solve. This strength is due to both factors: faster construction in many cases, and better heuristics in some cases. More importantly, as MS-lite is very quick to compute, it can be combined with other M&S heuristics with little computational overhead. We show that such combinations greatly enhance the coverage over previous M&S methods.

4.2 A Lightweight Merge-and-Shrink Method

MS-lite is a M&S method that *actively* shrinks every transition system to the *minimal* **h**-preserving abstraction. All states with the same **h**-value are combined into a single state, and the size of the abstraction is equal to the number of distinct **h**-values. MS-lite uses no label reduction and chooses transition systems for merging at random. The algorithm of MS-lite is shown in Algorithm 8. What distinguishes MS-lite from other M&S methods is its "counter-intuitive" shrinking strategy, which shrinks transition systems as early and as much as possible, as long as the **h**-values are preserved. In contrast, existing M&S methods with non-greedy bisimulation shrinking tend to shrink as late and as little as possible. They retain as much information as possible for later and larger transition systems.

Since MS-lite does not construct a bisimulation abstraction, there is little advantage to using label reduction.[2] The observation that MS-lite's performance is largely independent of its merging strategy emerged from a small

[2]In our experiments, label reduction has a large run time overhead that is only worthwhile when bisimulations are used. For MS-lite, turning on label reduction does not help it solve any more tasks, but reduces the total coverage by 33 tasks.

Algorithm 8 Merge-and-Shrink Algorithm of MS-Lite

1: $\mathcal{P} \leftarrow$ **InitMAS**
2: **while** $|\mathcal{P}| \geq 1$ **do**
3: $\quad \Theta_1, \Theta_2 \leftarrow \text{ChooseRandom}(\mathcal{P})$
4: $\quad \Theta_1 \leftarrow \text{Minimal-}\mathbf{h}\text{-preserving-Abstraction}(\Theta_1)$
5: $\quad \Theta_2 \leftarrow \text{Minimal-}\mathbf{h}\text{-preserving-Abstraction}(\Theta_2)$
6: $\quad \Theta_{1,2} \leftarrow \Theta_1 \otimes \Theta_2$
7: $\quad \text{CheckSolvabilityAndPruning}(\Theta_{1,2})$
8: $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{\Theta_{1,2}\} \setminus \{\Theta_1, \Theta_2\}$
9: **end while**
10: **return** the transition system in $\mathcal{P}$

experiment we undertook during its development. We ran MS-lite 10 times per task, with a different randomly chosen merging order on each run. On 35 of 39 test domains, MS-lite solved exactly the same number of tasks in all 10 random runs, and among these 35 domains there are 28 domains on which the number of A* node expansions and heuristic values on the initial states are exactly the same for all 10 random runs for all the tasks solved by MS-lite. In the remaining 4 domains, the coverage difference between the best and the worst of 10 MS-lite runs is only 1. In the coverage and node expansion results reported for MS-lite in this section, we use the average over 10 runs of MS-lite with the merging order chosen at random. For the coverage data shown in this chapter, an integer indicates that the coverage of all random runs are the same, and a fractional number indicates that there is variance.

MS-lite is extremely efficient by design: it maintains only minimal **h**-preserving abstractions and does not spend time exploring merge choices or reducing labels. In exchange for efficiency, it gives up a lot of information during shrinking. The question is: can MS-lite, with such aggressive shrinking, possibly compete with M&S methods equipped with shrinking, merging and label reduction techniques? The short answer is no. Table 4.3 shows the coverages of MS-lite and other M&S heuristics as well as the blind heuristic, which sets the heuristic value to the cost of the cheapest action for non-goal states and to zero for goal states. As expected, MS-lite has a smaller total coverage than most existing M&S methods. It solves 625.5 tasks in total, 45.5 fewer than `SCC-DFP` and 55.5 fewer than `DM-HQ`.

	MS-lite	DH	SD	DM	CGGL	LVL	RL	blind
Total (1499)	625.5	681	671	666	622	605	636	537
airport (50)	23	18	18	18	15	15	18	**21**
parking (40)	13	1	**6**	1	**6**	0	**6**	0
tidybot (30)	18	0	1	0	1	1	1	**16**
blocks (35)	28	21	26	25	24	**28**	25	18
tetris (17)	8	1	2	1	0	0	2	**8**
mystery (23)	17	15	16	15	**17**	**17**	16	15
pipesworld (100)	31	25	**31**	28	30	30	**31**	25
visitall (33)	21	13	12	12	12	**21**	12	11

Table 4.3: Coverages of MS-lite, `DM-HQ` (DH), `SCC-DFP` (SD), `DYN-MIASM` (DM), `CGGL`, `LVL`, `RL` and the blind heuristic (blind). The domains shown are those on which MS-lite's coverage is at least as good as the best (bold numbers) of all others. Numbers in brackets after each domain name indicate the total number of tasks in the domain.

However, considering the tiny abstractions that MS-lite constructs, it is surprising that MS-lite can solve this many tasks, which is actually more than methods `CGGL` and `LVL`—the M&S methods that can use abstractions with up to 50,000 states, bisimulation shrinking and label reduction. Our per-domain coverage investigation reveals where the strength of MS-lite is. In Table 4.3, we list all 8 domains (the rows below row "Total") where the coverage of MS-lite is larger than or equal to the *best* of the other heuristics.

In the following, we first discuss two reasons why MS-lite excels on domains such as those shown in Table 4.3, namely, (1) its efficient construction of simple heuristics for easy-but-complex tasks in Section 4.3 and (2) its ability to create better heuristics in some situations in Section 4.4. Then, in Section 4.5, we present *lite-enhancement*—a method that exploits MS-lite's complementary strength to enhance another M&S heuristic by taking the maximum of both heuristic values. We then evaluate lite-enhancement and discuss the results in Section 4.6.

4.3 Efficient Construction

In this section, we demonstrate that MS-lite can construct M&S heuristics very efficiently, and that there exist IPC tasks that are easy enough to be solved

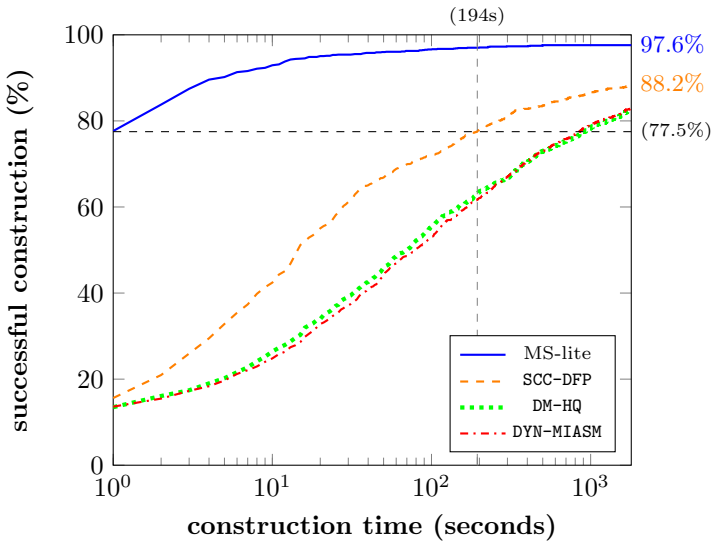


Figure 4.1: The percentages of tasks (y-axis) for which M&S construction of MS-lite, `SCC-DFP`, `DM-HQ` and `DYN-MIASM` are finished respectively within a certain amount of time (x-axis, in seconds) and within the 2GB memory limit.

with simple heuristics such as those created by MS-lite but are too complex to allow a normal M&S heuristic to be constructed within standard time and memory limits.

4.3.1 Construction Efficiency

Figure 4.1 shows the percentages of tasks for which MS-lite, `SCC-DFP`, `DM-HQ` and `DYN-MIASM` finish constructing their heuristic (within the 2GB memory limit) as a function of time. For any given time limit between 1 and 1800 seconds, MS-lite can construct many more M&S heuristics than `SCC-DFP`, `DM-HQ` and `DYN-MIASM`. The latter two are slower than `SCC-DFP` due to their product-dependent merge scoring function.

We now focus on comparing the construction efficiency of MS-lite and `SCC-DFP` when the time limit is 1800 seconds. MS-lite constructs M&S heuristics successfully for 97.6% (1463/1499) of all tasks. It fails on the remaining 36 tasks because it exceeds the 2 GB memory limit during M&S construction. There are no timeout failures. By comparison, `SCC-DFP` finishes constructing a heuristic for 88.2% (1322/1499) of tasks. It runs out of memory during construction for 42 tasks, including all 36 tasks where MS-lite fails. It runs out of

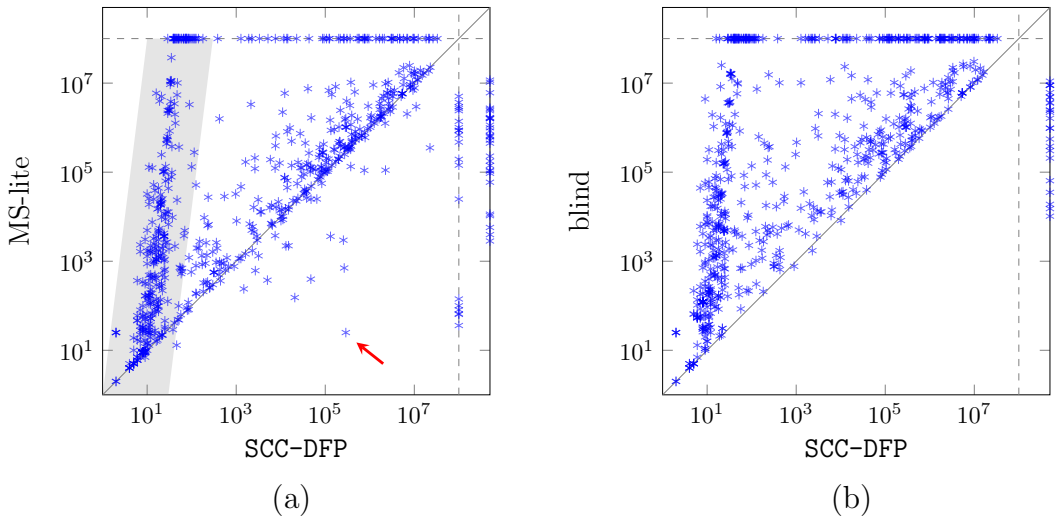


Figure 4.2: Comparing numbers of expansions by A* using different heuristics: (a) MS-lite (y-axis) vs. SCC-DFP (x-axis); (b) The blind heuristic (y-axis) vs. SCC-DFP (x-axis).

time during construction on an additional 135 tasks. With a 2GB limit, the main limitation of SCC-DFP construction is the running time. MS-lite finishes within 1 second for 1162 tasks (77.5% of all tasks), indicated by the horizontal dashed line in the figure. It requires a 194 second time limit, indicated by the vertical dashed line in the figure, for SCC-DFP to complete construction on the same number of tasks. On the 88.2% of tasks for which both MS-lite and SCC-DFP finish the heuristic construction, MS-lite uses at most 26 seconds per task while SCC-DFP can use up to 1716 seconds.

4.3.2 Complex but Easy Tasks

The advantage of MS-lite's efficient heuristic construction is that there is more time and memory left for A* to solve the task. We now show that this advantage does contribute to the coverage excellence of MS-lite on some domains in which tasks are easy enough to be solved with simple heuristics but are too complex for normal M&S construction.

To see this, we show the performance difference of MS-lite and SCC-DFP in an expansion comparison plot that separates the failures during M&S construction and the failures during A* search for cases where M&S construction succeeds. The plot is shown in Figure 4.2(a) in which each planning task is

shown as an asterisk, whose x and y values represent the numbers of A* node expansions using the SCC-DFP and MS-lite heuristics respectively. Failures in the M&S construction of SCC-DFP are shown on the rightmost edge of the plot, and those of MS-lite are shown on the top edge of the plot. Cases where the heuristic construction finishes successfully but A* fails to solve the task with the constructed heuristic are shown on the inset dashed lines, vertical for SCC-DFP and horizontal for MS-lite. Tasks on which both methods fail, for either reason, are not shown in the plot.

First of all, the fact that the majority of points are above the diagonal confirms that MS-lite's heuristic is less informative than SCC-DFP's overall. The concentrated distribution of points close to the leftmost edge, indicated by the shaded area, shows an exponentially growing number of A* node expansions with the MS-lite heuristic on a series of tasks[3] that are solved easily by SCC-DFP.

However, Figure 4.2(a) also demonstrates the complementary strength of MS-lite over SCC-DFP, on the rightmost edge and the vertical dashed line. For now, we focus on the 30 tasks that appear on the rightmost edge of the plot, which MS-lite solves while SCC-DFP fails during the heuristic construction phase. These tasks are from the domains TidyBot, Tetris, Airport and PipesWorld where complex tasks have hundreds or even thousands of variables. It is expensive to build an M&S heuristic with so many variables as the number of abstractions to construct is linear in the number of variables. Most of these abstractions contain tens of thousands of states. By keeping all abstractions small, MS-lite greatly reduces the computational burden. Constructing M&S heuristics for these complex tasks becomes feasible, and some of them are easy enough to be solved with MS-lite heuristics.

Many of these complex tasks are so easy that they can be solved even with the blind heuristic. Figure 4.2(b) is in the same format as Figure 4.2(a) but shows the blind heuristic on the y-axis. In this plot, there are 24 tasks on

[3] Tasks following the whole trend are mainly from Gripper, Miconic, NoMystery, PARCPrinter, Satellite and WoodWorking. Tasks from DriverLog, Hiking, Logistics, PSR-Small, Rovers, Transport and Trucks follow the trend but only up to the lower half (end at around $y = 10^4$).

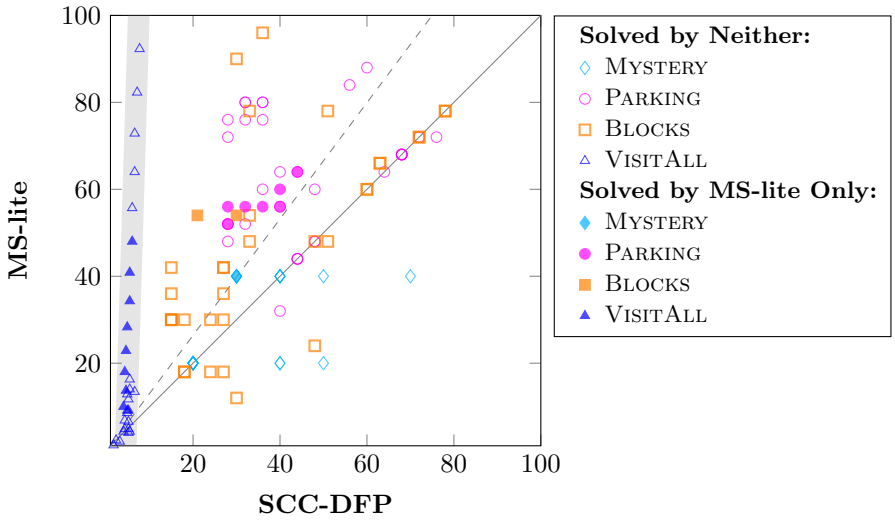


Figure 4.3: Comparing the heuristic value of the initial state of MS-lite heuristic (y-axis) vs. `SCC-DFP` (x-axis).

the rightmost edge from domains TidyBot, Tetris and Airport which are exactly the three domains where the blind heuristic has the highest coverage among the M&S heuristics excluding MS-lite, as shown in Table 4.3.

4.4 Better Heuristics on Some Domains

MS-lite's efficient construction and the existence of easy-but-complex tasks explain the points on the rightmost edge of Figure 4.2(a). We now look at points below the diagonal and on the inset vertical dashed line, which are an unexpected bonus for MS-lite. For points below the diagonal, in one case from domain VisitAll, indicated by a red arrow in Figure 4.2(a), A* expands only 25 nodes with MS-lite's heuristic, but almost 300,000 with `SCC-DFP`'s. The majority of tasks for which MS-lite has fewer expansions than `SCC-DFP` come from three domains: Depot, Mprime and Mystery.

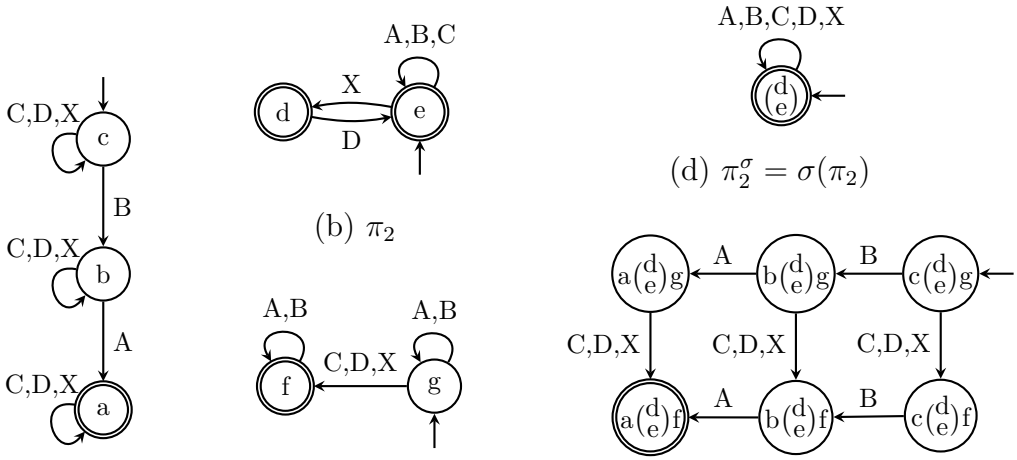
There are also 19 points on the vertical dashed line of Figure 4.2(a) representing tasks for which both methods construct their heuristics but only MS-lite solves the problem. Did `SCC-DFP` spend too many resources for constructing its heuristic, leaving too few resources for A* to find a solution? A look at the heuristic values of the initial states shows that it is more likely that MS-lite creates better heuristics than `SCC-DFP` here. The points on the

vertical dashed line in Figure 4.2(a) consist of tasks from 4 domains: VisitAll, Parking, Blocks and Mystery. In Figure 4.3, we compare the initial heuristic values on all tasks from these 4 domains. We highlight the 19 tasks that are only solved by MS-lite as filled markers while the tasks unsolved by both are shown as hollow markers. For clarity, we scale the heuristic values to fit in the range [0, 100]. Heuristic values from the same domain are scaled by the same factor. In Figure 4.3, many more points lie above the diagonal line, indicating that MS-lite creates better initial heuristics for most of the tasks from these 4 domains. On the 19 tasks solved only by MS-lite, its initial heuristic values are at least 33% larger, indicated by being above the dashed line, than those of `SCC-DFP`. On tasks from VisitAll (points in the shaded gray strip), the initial heuristic values of MS-lite grow about *30 times* faster than those of `SCC-DFP` as the tasks scale up.

4.4.1 An Example of Beneficial Active Shrinking

MS-lite actively shrinks an abstraction whenever possible, while the passive shrinking of other M&S systems is applied only when abstractions become too large to merge. How can MS-lite with its active shrinking produce better heuristics than `SCC-DFP`, which uses passive shrinking? In the following, we show an example where active **h**-preserving shrinking produces a better heuristic than passive **h**-preserving shrinking. The example is abstracted from a task in IPC domain Blocks.

In this example, we need to merge three atomic transition systems π_1, π_2, and π_3, shown in Figure 4.4(a), (b) and (c) respectively. Let σ be the function that maps a transition system to its minimal **h**-preserving abstraction. In other words, $\sigma(\Theta)$ is the transition system produced by applying minimal **h**-preserving shrinking on transition system Θ. Because all the states in π_1 and π_3 have different **h**-values, $\sigma(\pi_1) = \pi_1$ and $\sigma(\pi_3) = \pi_3$. Figure 4.4(d) shows $\sigma(\pi_2)$, the minimal **h**-preserving abstraction of π_2, which contains a single state with a self-loop labelled by all actions. We use the notation $\binom{\mathrm{s}}{\mathrm{t}}$ to label the abstract state that is the combination of two states s and t. For example, the single state $\binom{\mathrm{d}}{\mathrm{e}}$ in $\sigma(\pi_2)$ is the abstract state produced by the **h**-preserving



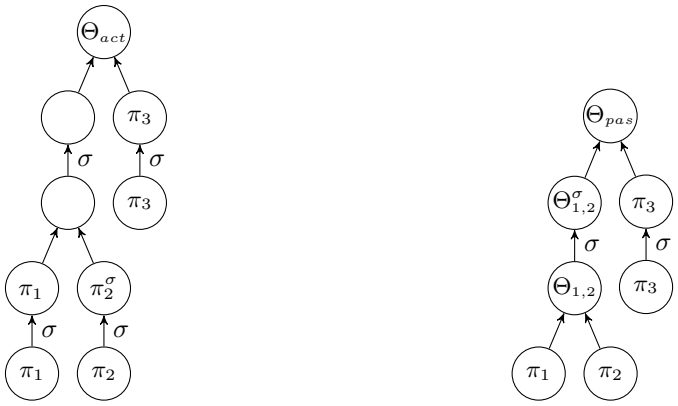
(a) $\pi_1 = \sigma(\pi_1)$ (b) π_2 (c) $\pi_3 = \sigma(\pi_3)$ (d) $\pi_2^\sigma = \sigma(\pi_2)$ (e) $\Theta_{act} = (\pi_1 \otimes \sigma(\pi_2)) \otimes \pi_3$

Figure 4.4: (a) π_1 and its minimal **h**-preserving abstraction; (b) π_2; (c) π_3 and its minimal **h**-preserving abstraction; (d) The minimal **h**-preserving abstraction of π_2; (e) The synchronized product Θ_{act} of merging the atomic projections in order $(\pi_1 \otimes \pi_2) \otimes \pi_3$ with active shrinking.

shrinking that combines states d and e in π_2.

We consider a fixed merging order $(\pi_1 \otimes \pi_2) \otimes \pi_3$ but two alternatives for the shrinking strategy. The first alternative is MS-lite's active **h**-preserving shrinking. The second alternative is the passive and late **h**-preserving shrinking that occurs when using non-greedy bisimulation shrinking, and the size limit is smaller than the bisimulation size, as in the best M&S methods `SCC-DFP`, `DYN-MIASM` and `DM-HQ`. In this small example, we set the size limit μ to 8 to force a late **h**-preserving shrinking. We will see that a larger heuristic value for the initial state is produced by the first alternative than by the second alternative.

Alternative 1: Figure 4.5(a) shows the M&S tree with MS-lite's active shrinking, which applies the minimal **h**-preserving shrinking before every merge. Because $\pi_3 = \sigma(\pi_3)$ and $\pi_1 = \sigma(\pi_1)$, the final product is $\sigma(\pi_1 \otimes \sigma(\pi_2)) \otimes \pi_3$. Since $\sigma(\pi_2)$ is a single state with a self-loop labelled by all the actions, the synchronized product $\pi_1 \otimes \sigma(\pi_2)$ and its abstraction $\sigma(\pi_1 \otimes \sigma(\pi_2))$ are isomorphic to π_1 itself. Figure 4.4(e) shows the synchronized product $\Theta_{act} = (\pi_1 \otimes \sigma(\pi_2)) \otimes \pi_3$ which is equal to $\sigma(\sigma(\pi_1) \otimes \sigma(\pi_2)) \otimes \sigma(\pi_3)$.

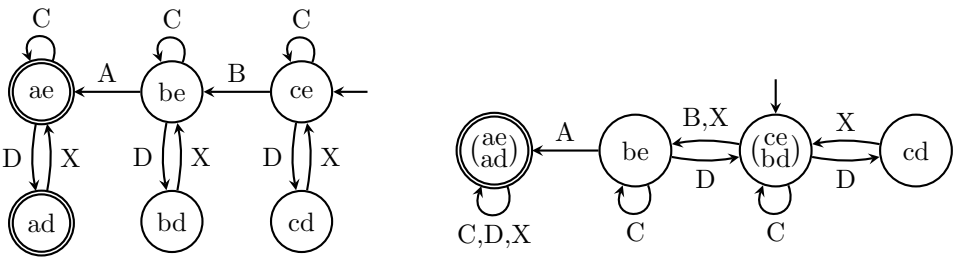


(a) M&S tree of active shrinking (b) M&S tree of passive shrinking

Figure 4.5: Active shrinking and passive shrinking with size limit $\mu = 8$ for the same merging order $(\pi_1 \otimes \pi_2) \otimes \pi_3$.

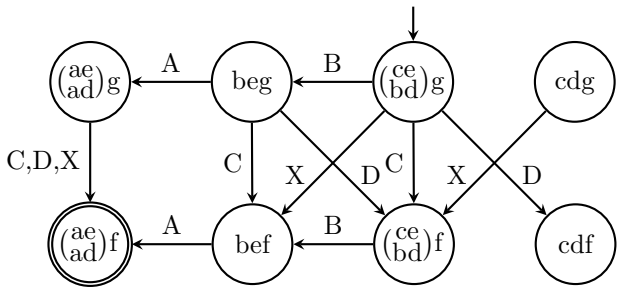
The shortest path from the initial state to a goal state in Θ_{act} is of length 3.

Alternative 2: Figure 4.5(b) shows the M&S tree using the same merging order as in Alternative 1 but with a passive shrinking strategy with size limit $\mu = 8$. Since the size of $\Theta_{1,2} = \pi_1 \otimes \pi_2$ is less than $\mu = 8$, no shrinking is done. Their synchronized product $\Theta_{1,2}$ is shown in Figure 4.6(a). Since $\Theta_{1,2}$ has 6 states and π_3 has 2 states, the size of their synchronized product is 12 which is larger than 8. **h**-preserving shrinking has to be applied to either $\Theta_{1,2}$ or π_3 or both. Because $\sigma(\pi_3) = \pi_3$, we can only shrink $\Theta_{1,2}$ and the transition system after shrinking can have at most 4 states. The **h**-preserving shrinking that reduces $\Theta_{1,2}$ to 4 states is the minimal **h**-preserving shrinking. The resulting transition system, denoted as $\Theta_{1,2}^\sigma$, is shown in Figure 4.6(b). The key difference between this and Figure 4.6(a) is the transition from the initial state to state "be" using action X. This transition was not possible before the shrinking occurred and was introduced into the transition system by combining state "bd" with the initial state "ce". This does no immediate harm, since the mapping is **h**-preserving, but it has ramifications when this transition system is merged with π_3, where action X plays a crucial role. The final result, the synchronized product $\Theta_{pas} = \sigma(\pi_1 \otimes \pi_2) \otimes \pi_3$, is shown in Figure 4.6(c). In this transition system, the shortest path from the initial state to the goal is only length 2.



(a) $\Theta_{1,2} = \pi_1 \otimes \pi_2$

(b) $\Theta^{\sigma}_{1,2} = \sigma(\Theta_{1,2})$



(c) $\Theta_{pas} = \sigma(\Theta_{1,2}) \otimes \pi_3$

Figure 4.6: (a) The product $\Theta_{1,2}$ of merging π_1 and π_2; (b) The **h**-preserving abstraction $\Theta^{\sigma}_{1,2}$ of $\Theta_{1,2}$ with 4 states; (c) The product Θ_{pas} of merging the atomic projections in order $(\pi_1 \otimes \pi_2) \otimes \pi_3$ using passive shrinking with size limit $\mu = 8$.

The key observation here is that the late **h**-preserving shrinking $\sigma(\pi_1 \otimes \pi_2)$, which combines "bd" with "ce", can be more harmful than the early **h**-preserving shrinking $\sigma(\pi_2)$, which essentially combines state "be" with state "bd" and state "ce" with state "cd" but keeps states "bd" and "ce" separate in $\pi_1 \otimes \sigma(\pi_2)$. Of course, if size limits are large enough to allow a complete bisimulation refinement, the problem would be avoided. However, in practice partial bisimulation refinements are common and they are not guaranteed to reverse every harmful combination induced by shrinking. For this example, even with size limit of 10, the bisimulation refinement process will refine the combined goal state in $\Theta^{\sigma}_{1,2}$ but still fail to split the combined initial state, since states closer to the goal typically have higher priority.

4.5 MS-Lite Enhancement

Although the MS-lite heuristic alone is not competitive with other M&S heuristics, its complementary strength can be used to enhance another M&S heuristic by taking the maximum of both. We call the other M&S method enhanced by MS-lite the *base* M&S method. By taking the maximum of the MS-lite and base heuristics, we exploit the superiority of the MS-lite heuristic on some IPC domains.

We primarily want to exploit MS-lite's fast construction. We build the MS-lite heuristic first and then limit the time and memory given to the base M&S heuristic as follows: if MS-lite finishes building its heuristic within the standard 30min/2GB limits, then we attempt to build the base heuristic within 15min/1.5GB[4] limits. If this attempt fails, we simply use the MS-lite heuristic by itself for the A* search. Setting tighter limits for the base M&S reduces the coverage if there are hard tasks that are only solvable with a high quality heuristic produced by the base M&S method, but improves the coverage if there are tasks that are too complex to build a base M&S heuristic but are solvable with MS-lite. We have seen such tasks in Section 4.3. We call this method *lite-enhanced* M&S, and denote the lite-enhanced base M&S method X as lite-X, e.g., lite-`SCC-DFP` for lite-enhanced `SCC-DFP`.

4.6 Experiments

We test lite-enhanced M&S with the two best-performing M&S methods, `SCC-DFP` and `DM-HQ`, as the base methods. We run each lite-enhanced M&S 5 times per task on the 1499 tasks in our benchmark using the standard 30min/2GB limits on the computing resources.

In the following, we first show that lite-enhancement for both `SCC-DFP` and `DM-HQ` have low variance (Section 4.6.1) and infrequent, small performance degeneration (Section 4.6.2). Then we show that the improvement of lite-

[4]In theory, we can use any memory limit smaller than 2GB, but in our implementation, we can only check the peak memory usage periodically and need to keep a margin of reserve memory to avoid termination of the planner during the base M&S construction.

Coverage	SCC-DFP	DM-HQ
Original	671	681
Lite-Enhanced	718.8	746.2
Changes	+47.8	+65.2

Table 4.4: Total coverage of the base M&S heuristics (row "Original"), and their lite-enhanced variants (row "Lite-Enhanced") and the coverage difference between lite-enhanced and base M&S.

DM-HQ over lite-SCC-DFP is much better than the improvement of DM-HQ over SCC-DFP in Section 4.6.3. This indicates the stronger complementarity between MS-lite and DM-HQ compared to MS-lite and SCC-DFP. Finally, in Section 4.6.4, we give a detailed per-domain analysis of the improvement of lite-DM-HQ over SCC-DFP.

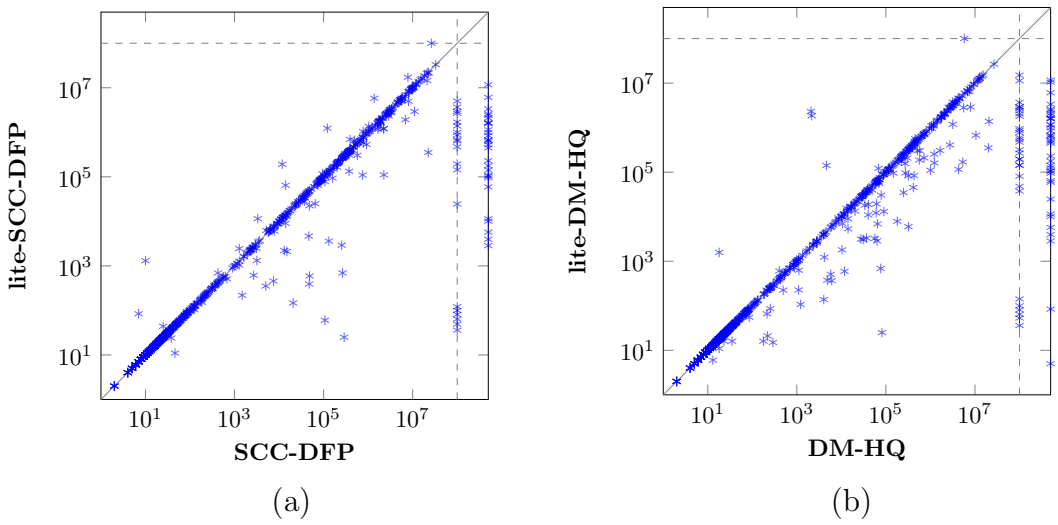
4.6.1 Low Variance of Lite-Enhancement

As with standalone MS-lite, the performance of lite-enhanced M&S varies little for random merging orders. Over 39 domains, there is only 1 domain (Airport) where the coverages of random runs of lite-enhanced SCC-DFP vary. For lite-enhanced DM-HQ, there are 4 such domains (Airport, Mystery, Scanalyzer and TidyBot). The coverage difference between the best and worst random runs for these enhanced M&S is only 1.

In the results shown, the coverage of lite-enhanced M&S is the average of 5 random runs. In plots comparing numbers of expansions, if all runs solve the task then we use the average number of expansions of the runs. If there is any run that fails to solve the task we consider it as a failure for lite-enhanced M&S on that task. The total coverages of lite-enhanced SCC-DFP and DM-HQ are shown in row "Lite-Enhanced" in Table 4.4.

4.6.2 Small Performance Degeneration

Lite-enhancement may cause performance degeneration due to the overhead of building and using the MS-lite heuristic, as well as the earlier termination of the base M&S method. However, the results suggest that neither situation occurs frequently. For both base M&S methods, we see only 1 task that is



(a) (b)

Figure 4.7: Comparing numbers of expansions by A* using different M&S heuristics: (a) lite-enhanced SCC-DFP heuristic (y-axis) vs. SCC-DFP (x-axis); (b) lite-enhanced DM-HQ heuristic (y-axis) vs. DM-HQ (x-axis).

solved by the base M&S method alone but not by their lite-enhanced variant. For SCC-DFP, it is a task from HIKING, and for DM-HQ, it is a task from NOMYSTERY. These tasks are indicated by the single points on the horizontal dashed lines in the plots of Figure 4.7(a) and of Figure 4.7(b) respectively.

The plots in Figure 4.7 show points on both the vertical dashed line and the rightmost edge. There are more points below the diagonal than above the diagonal in the plots. This suggests that lite-enhancement improves the coverage not only due to the faster heuristic construction, but also due to the better heuristics created on some tasks.

4.6.3 Stronger Complementarity with DM-HQ

The most important observation in our experiments is that MS-lite complements DM-HQ much better than SCC-DFP. The row "Changes" in Table 4.4 shows that the total coverage improvement of lite-enhancement is much greater for DM-HQ than for SCC-DFP. In the previous chapter, we have seen that DM-HQ solves 36 tasks on which SCC-DFP fails but it also fails on 26 tasks that SCC-DFP solves. The following analysis shows that lite-enhancement is more helpful for DM-HQ in solving the 26 tasks on which it fails but SCC-DFP succeeds than for SCC-DFP in solving the 36 tasks on which it fails but DM-HQ succeeds.

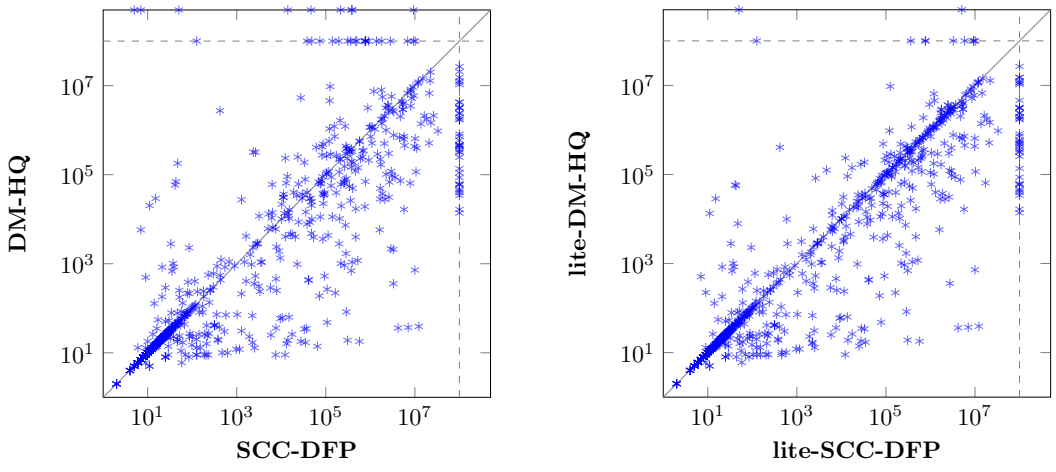


Figure 4.8: Comparing numbers of expansions by A* using different M&S heuristics: (a) `DM-HQ` (y-axis) vs. `SCC-DFP` (x-axis); (b) lite-enhanced `DM-HQ` (y-axis) vs. lite-enhanced `SCC-DFP` (x-axis);.

To show this, we compare `DM-HQ` against `SCC-DFP` with and without lite-enhancements. Figure 4.8(a) compares the number of A* node expansions using M&S heuristics produced by `DM-HQ` and `SCC-DFP`. This plot is the same as Figure 3.11(b) in Chapter 3, except that Figure 4.8(a) additionally distinguishes search failures and M&S construction failures. Figure 4.8(b) compares the number of A* node expansions when lite-enhancement is used for these two M&S methods. While both methods benefit a lot from the lite-enhancement, there are fewer points on the top edge and horizontal dashed line in Figure 4.8(b) than in Figure 4.8(a) whereas the numbers of points on the right edges and vertical dashed lines in the two plots are very similar. This observation, combined with the previous observation in Section 4.6.2 that almost all tasks solved with `DM-HQ` and `SCC-DFP` alone are solved with their lite-enhancements, implies that many of the 36 tasks on the horizontal dashed line in Figure 4.8(a) become solved when `DM-HQ` is enhanced with MS-lite, but many of the 26 tasks on which `SCC-DFP` fails but `DM-HQ` succeeds remain unsolved with the lite-enhancement for `SCC-DFP`. All these results show that MS-lite has a much stronger complementarity with `DM-HQ` than with `SCC-DFP`.

4.6.4 Detailed Per-Domain Analysis

In this section, we compare the performance of all four methods SCC-DFP, DM-HQ, lite-enhanced SCC-DFP and lite-enhanced DM-HQ on each IPC domain from our test set.

Table 4.5 shows the coverage of SCC-DFP, and increases/decreases of coverage of DM-HQ, lite-enhanced SCC-DFP and lite-enhanced DM-HQ with respect to SCC-DFP. The table lists all domains where DM-HQ, lite-SCC-DFP or lite-DM-HQ solve a different number of tasks than SCC-DFP. To illustrate the strengths of lite-DM-HQ over SCC-DFP, we divided these domains into 3 groups (a), (b) and (c):

(a) Domains where lite-enhanced DM-HQ has better coverage than DM-HQ.

(b) Domains where DM-HQ gains no coverage improvement from lite-enhancement but outperforms SCC-DFP.

(c) Domains where SCC-DFP solves more tasks than DM-HQ, and lite-enhancement does not improve DM-HQ.

Within each group, domains are sorted in decreasing order of the respective coverage improvement, i.e., lite-DM-HQ coverage minus DM-HQ coverage for (a), and DM-HQ coverage minus SCC-DFP coverage, which are the numbers in column "DM-HQ", for (b) and (c). For example, on Parking in (a), DM-HQ solves 5 fewer tasks than SCC-DFP, but enhanced DM-HQ solves 7 more than SCC-DFP, so the total improvement is +12 which is larger than the corresponding value +10 for VisitAll.

Group (a) contains most of the domains where DM-HQ performs worse than SCC-DFP. 19 tasks from those domains are solved by SCC-DFP but not DM-HQ. However, lite-DM-HQ solves many more tasks than SCC-DFP. Group (b) contains most domains where DM-HQ outperforms SCC-DFP by itself, and we see no improvement from lite-enhancement on DM-HQ. The total coverage improvement of DM-HQ over SCC-DFP from those domains is 35 tasks. Group (c) contains 5 of the 7 domains that lite-DM-HQ solves fewer tasks than SCC-DFP (the other 2

	Domains	SCC-DFP	DM-HQ	Lite-SD	Lite-DH
(a)	tidybot (30)	1	-1	+16	+16.4
	parking (40)	6	-5	+7	+7
	visitall (33)	12	+1	+9	+11
	tetris (17)	2	-1	+6	+6
	blocks (35)	26	-5	+2	+2
	pipesworld (100)	31	-6	+2	+1
	airport (50)	18	0	+4.8	+4.8
	mystery (23)	16	-1	+1	-0.2
	scanalyzer (30)	13	-1	0	-0.8
	Subtotal (358)	125	-19	+47.8	+47.2
(b)	floortile (40)	6	+9	0	+9
	elevators (30)	13	+7	0	+7
	sokoban (30)	26	+4	0	+4
	woodworking (30)	19	+3	0	+3
	hiking (20)	13	+2	-1	+2
	rovers (40)	6	+2	0	+2
	transport (60)	17	+2	0	+2
	nomystery (20)	18	+2	0	+1
	logistics (63)	25	+1	0	+1
	openstacks (80)	30	+1	0	+1
	trucks (30)	7	+1	0	+1
	depot (22)	6	+1	+1	+1
	Subtotal (465)	186	+35	0	+34
(c)	gripper (20)	20	-1	0	-1
	miconic (150)	78	-1	0	-1
	parcprinter (30)	26	-1	0	-1
	tpp (30)	8	-1	0	-1
	pegsol (35)	35	-2	0	-2
	Subtotal (245)	147	-6	0	-6
	Changes (1089)	478	+10	+47.8	+75.2
	Others (410)	193	0	0	0
	Total (1499)	671	681	718.8	746.2

Table 4.5: Coverage of SCC-DFP and the increases/decreases of DM-HQ over SCC-DFP (column "DM-HQ"), lite-enhanced SCC-DFP over SCC-DFP (column "Lite-SD") and of lite-enhanced DM-HQ over SCC-DFP (column "Lite-DH"). "Others" summarizes the 13 domains for which all four systems have the same coverage.

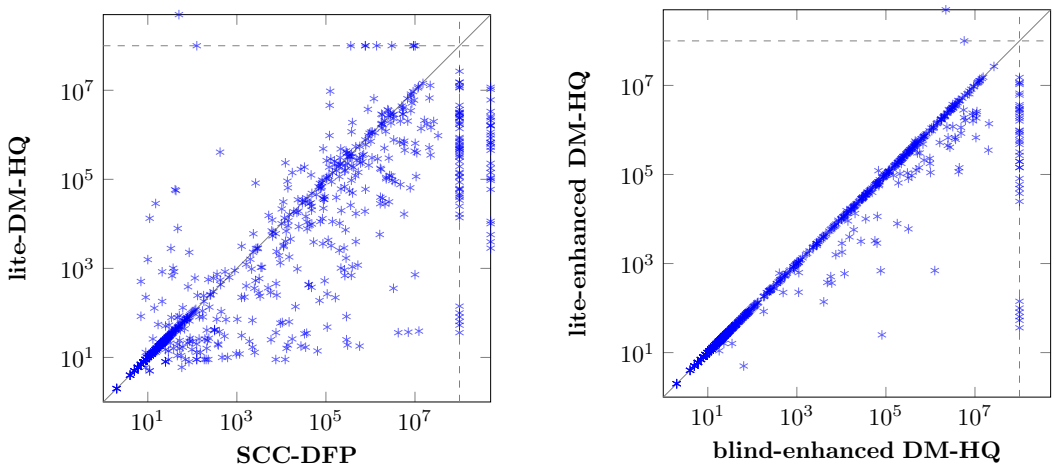


Figure 4.9: Comparing numbers of expansions by A* using different M&S heuristics: (a) lite-DM-HQ (y-axis) vs. SCC-DFP (x-axis); (b) lite-enhanced DM-HQ (y-axis) vs. blind-enhanced DM-HQ (x-axis).

domains are included in group (a)). The coverage decrease is only 6 tasks for group (c) and 7.0 tasks in total.

Figure 4.9(a) compares the number of A* node expansions of lite-DM-HQ and SCC-DFP. The distribution of points for tasks solved by both methods are similar to that in Figure 4.8(a) showing the superior heuristics created by DM-HQ. The weaknesses of DM-HQ largely disappear with the help of lite-enhancement, as shown by fewer points on the top edge and the horizontal dashed line in Figure 4.9(a) compared to Figure 4.8(a). These points correspond to tasks in domains of group (a) in Table 4.5. Many points on the right edge and vertical dashed line correspond to tasks that are not solved by SCC-DFP but solved by lite-DM-HQ because of DM-HQ's own strength and the additional power from lite-enhancement.

4.7 Other Fallback Heuristics

Before we conclude this chapter, we look at two other fallback heuristics that can be used to enhance a merge-and-shrink heuristic.

4.7.1 Blind Heuristic

Table 4.3 shows that the blind heuristic also has some complementary strength over normal M&S heuristics. We now consider *blind-enhancement*, i.e., enhancing a base M&S heuristic with the blind heuristic in the same way as MS-lite is used in the lite-enhancement.

We use `SCC-DFP` and `DM-HQ` as the base methods for blind-enhancement. The total coverage of blind-enhanced `SCC-DFP` and `DM-HQ` is 693 and 711 respectively. Both numbers are larger than the base M&S methods (by 22 for `SCC-DFP` and 30 for `DM-HQ`), but smaller than the lite-enhanced variants. Figure 4.9(b) compares the numbers of A* node expansions of lite-enhanced and blind-enhanced `DM-HQ`. Since both MS-lite and blind heuristics are relatively weaker than the base M&S heuristic `DM-HQ`, most points appear very close to the diagonal line. The great number of points below the diagonal line and on the vertical dashed line are due to the better heuristic created by MS-lite. Points on the top edge and the horizontal dashed line indicate there are the tasks that can be solved by blind-enhanced `DM-HQ` but lite-enhancement `DM-HQ` spend too much memory or time in M&S construction and thus either fails the M&S construction or leaves insufficient resource for the A* search.

4.7.2 Partial Merge-and-Shrink Heuristic

In lite-enhancement and blind-enhancement, when M&S is terminated all the intermediate abstractions are abandoned. However, these abstractions can still be used to generate a heuristic. Sievers [Sie18] proposed a *partial merge-and-shrink* heuristic: a generalization of the standard M&S that extracts a heuristic from multiple M&S abstractions by either taking the maximum of all abstraction heuristics or chooses one abstraction heuristic according to some criterion.

Similar to MS-lite in lite-enhancement, a partial M&S heuristic is used as a fallback heuristic if it takes too long or too many resources to construct the final M&S abstraction. Two types of termination conditions were tested: limiting time and limiting the number of transitions. Sievers' experiments

show that termination by limiting the number of transitions do not improve the performance, but terminating M&S construction when time exceeds 15 minutes and then using the max of the abstraction heuristics of all intermediate abstractions at the time of termination can improve the performance of SCC-DFP and DYN-MIASM: on their benchmark of 1667 tasks[5] and with a total time limit of 30 minutes and a higher memory limit of 3.5 GB, the coverage improvements are 32 for SCC-DFP and 34 for DYN-MIASM.

4.8 Conclusions

In this chapter, we have presented MS-lite, a M&S method that maintains only the smallest **h**-preserving abstractions. The minimalism of MS-lite avoids expensive shrinking, merging and label reduction operations, allowing very efficient construction of heuristics even for complex tasks. MS-lite's strengths are complementary to other M&S methods: not only its superior construction efficiency, but also its better heuristics on some tasks. We demonstrate in an example that the active shrinking of MS-lite can result in better heuristics than normal passive shrinking. More importantly, the efficiency of MS-lite makes it perfect for enhancing other M&S heuristics by using the maximum of both heuristics for search because there is little overhead for constructing the MS-lite heuristic. Such MS-lite enhancement improves the coverage of M&S methods SCC-DFP and DM-HQ by a large number of tasks. Our experiments on IPC domains show that MS-lite complements DM-HQ much better than SCC-DFP, and lite-enhanced DM-HQ dramatically outperforms the previous state of the art method SCC-DFP.

[5]Their benchmark does not remove duplicate tasks and unsolvable tasks like ours.

Chapter 5

The Two-edged Nature of Diverse Action Costs

In the previous chapters, we have presented methods for creating high quality merge-and-shrink heuristics in general. Most of the planning tasks used in the studies use *unit costs*, i.e., all actions have the same cost. However, many real-world planning applications use *non-unit costs* where different actions may have different costs. In the following two chapters, we study how action cost affects the two components of a heuristic search planner, namely, the search algorithm and the domain-independent heuristic construction method. In this chapter, our study focuses on the general effects of action cost on search *without* using heuristics, and in the next chapter (Chapter 6), we shift our focus to the effects of non-unit action costs on merge-and-shrink heuristics. This chapter is based on the publication [FMH17b].

5.1 Introduction

Diverse action costs occur naturally in many planning problems. For example, in the IPC domain Transport, loading and unloading a package is much cheaper than moving a vehicle, and the cost of moving a vehicle varies widely with the distance between locations.

In recognition of its importance, planning with action costs has been studied in both the optimal and satisficing settings in recent years. One notable trend is a focus on the negative impact of non-unit costs on planning. Sev-

eral studies demonstrate cases where planning with non-unit action costs is more difficult than planning in the same problems with unit costs, where every action has the same cost [Ben+10; CBK11; WR11; WR14]. Other studies showed that some search algorithms perform better on domains with non-unit action costs when they use heuristics based on unit costs instead of relying entirely on heuristics based on the given non-unit costs [RW10; TBH12; TR09; TR11].

One clear disadvantage of non-unit costs is a kind of "horizon effect". If a search space has huge regions reachable with low-cost actions, but the solution requires a high-cost action a, the low-cost regions will be exhaustively searched before the state s reached by a is expanded. In the unit cost model, state s would be expanded much earlier, allowing the solution to be found much more quickly.

Thus there is considerable evidence that action cost diversity is harmful for search. Is it true that this is generally the case? This is the question we address in this chapter. We provide both theoretical and experimental investigations of the effects of changing action costs on the number of nodes expanded. Our experiments give a variety of examples, including problems from the IPC domains, where diversity of action costs is beneficial, i.e., the number of nodes expanded when using non-unit action costs is substantially lower than the number using unit costs.

Our main theoretical result is a "No Free Lunch" (NFL) theorem about the impact, on the number of nodes expanded by Dijkstra's algorithm [Dij59], of changing from any cost function $\mathcal{C}$ to any other $\mathcal{C}'$. We prove that, when the *TIE* states (the states that are the same distance from the start state as the goal state) are ignored, the advantage of $\mathcal{C}$ over $\mathcal{C}'$ are exactly counterbalanced by the advantage of $\mathcal{C}'$ over $\mathcal{C}$ when all problem instances in the state space are taken into consideration.

The NFL theorem applies to all state spaces, including the "ε-cost traps" that Cushing et al. [CBK10; CBK11] designed to illustrate that search with non-unit action costs can expand exponentially more states than search with unit costs. For a type of ε-cost trap called the *cycle trap*, we show experimen-

tally that the total number of nodes expanded over all possible goal states is *almost* identical whether one uses unit costs or the alternative non-unit cost function defined by Cushing et al.

The reason the totals in the cycle trap experiment are not exactly equal is that the NFL theorem does not take the TIE states into account. Our second theoretical contribution is to analyze the impact of TIE states. We show that unit costs will often have an advantage over non-unit costs because of TIE states (as is the case for cycle traps) but we also describe a new planning domain—Hazardous Logistics—in which TIE states work to the advantage of non-unit costs.

The remainder of the chapter is organized as follows. In Section 5.2 we review previous work on the impact of diverse action costs on search. Section 5.3 gives three motivating examples in which diverse action costs are beneficial. Section 5.4 experimentally shows that many of the IPC problems from domains with non-unit costs are more easily solved with non-unit costs than with unit costs. Section 5.5 describes the setting for our theoretical analysis and presents the NFL theorem and the cycle trap experiments. Section 5.6 contains our theoretical study related to tie-breaking and introduces the Hazardous Logistics domain.

5.2 Related Work

Some domains with non-unit action cost functions have been used in the IPC since 2008. Many recent planners contain a search or evaluation component where the true action cost function $\mathcal{C}$ is replaced by the unit cost function $\mathcal{U}$. We will call such a component either $\mathcal{C}$-based or $\mathcal{U}$-based, depending on which cost function it uses. For example, a $\mathcal{U}$-based heuristic, $h(s)$, estimates the number of actions that must be applied to state s to reach a goal state.

The use of $\mathcal{U}$-based heuristics was empirically shown to improve the performance of many planners on IPC problems. For example, the first stage of LAMA [RW10; RWH11], the winner of the IPC 2008 sequential satisficing track, uses $\mathcal{U}$-based FF [HN01] and landmark heuristics [RW10]. Later stages

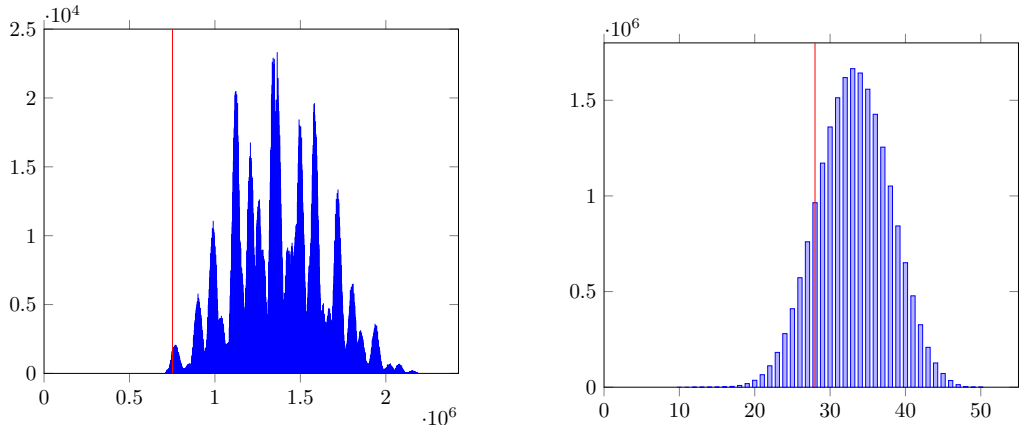
use $\mathcal{C}$-based heuristics. Explicit Estimation Search [TR11] uses both $\mathcal{C}$-based and $\mathcal{U}$-based heuristics to determine the next node to expand in bounded suboptimal search. Wilt and Ruml [WR11] give examples where $\mathcal{U}$-based search algorithms outperform $\mathcal{C}$-based ones. They also demonstrate cases where $\mathcal{U}$-based Greedy Best First Search (GBFS) expands fewer nodes than $\mathcal{C}$-based GBFS [WR14]. The A* algorithm sometimes performs better with $\mathcal{U}$ than with $\mathcal{C}$ for the state spaces studied in [Wil14; WR11].

Along with these empirical observations, the following explanations for why $\mathcal{U}$-based search is easier than $\mathcal{C}$-based search were proposed. Wilt and Ruml [WR11; WR14] argue that the difficulty of $\mathcal{C}$-based search is due to large heuristic errors and large local minima caused by action costs with high variance. Cushing, Benton and Kambhampati [CBK11] give a theoretical analysis of Dijkstra's algorithm and provide examples where $\mathcal{C}$-based search expands exponentially many more nodes than $\mathcal{U}$-based search .

All these studies seem to indicate that *diverse action costs are harmful for best-first search* in planning. While there is evidence to support this belief, it is not the full story. The parameterized complexity analysis by Aghighi and Bäckström [AB15; AB16] shows that optimal planning with positive integer costs is no harder than planning with unit costs, and the difference between maximal and minimal costs is irrelevant to the inherent computational complexity of planning. The literature also contains some evidence that diverse action costs can speed up search. In Wilt's study of the effect of costs on A* [Wil14], in two out of the five domains, A* expands fewer nodes for higher variance action costs. In two of four domains used by Wilt and Ruml [WR14] GBFS expands fewer nodes with non-unit action costs. In the IPC domains Floortile and Woodworking, GBFS using a $\mathcal{C}$-based FF heuristic solves more problems than GBFS using a $\mathcal{U}$-based FF [Nak13].

5.3 Motivating Examples

The following examples give evidence that diverse action costs can be beneficial for search. Example 1 also foreshadows the use of surely expanded nodes in our



(a) Non-Unit Cost $\mathcal{C}$ (b) Unit Cost $\mathcal{U}$

Figure 5.1: Histograms for f-value (x-axis) and optimal solution cost (indicated by the red vertical line) for problem p08 of IPC domain PARCPRINTER with (a) the original non-unit cost function $\mathcal{C}$ and (b) the unit cost function $\mathcal{U}$.

analysis in Section 5.5 and Example 2 provides the gist of the NFL theorem showing that the existence of a previously published example in which diverse costs are harmful implies the existence of examples showing the opposite.

5.3.1 Example 1: IPC PARCPrinter Problem

To illustrate that diverse costs can result in faster search than unit cost, consider the behavior of A* with FastDownward's pattern database heuristic[1] [Ede01] on problem p08 from the IPC domain PARCPRINTER. The original IPC cost function $\mathcal{C}$ assigns a variety of costs to actions, ranging from 0 to more than 200,000. Using $\mathcal{C}$, A* expands 61,458 states in solving this problem, while A* with unit cost function $\mathcal{U}$ requires 2,453,418 node expansions, more than 39 times as many.

The additional state expansions are not due to unlucky tie-breaking with $\mathcal{U}$. There is a more fundamental reason. With a consistent heuristic, A* must expand every state s for which $f(s) < C^*$, the optimal solution cost (Theorem 12 [Pea84]). Such states are called "surely expanded" by A*, or SE for short. In this problem task, there are 61,456 SE states under cost function $\mathcal{C}$ (to the

[1]In the default configuration of the pattern database heuristic (`www.fast-downward.org/Doc/Heuristic#Pattern_database_heuristic`)

	1	2	3
4	5	6	7
8	9	10	11
13	**12**	**15**	**14**

(a)

	15	14	13
12	11	10	9
8	7	6	5
3	**4**	**1**	**2**

(b)

	15	14	13
12	11	10	9
8	7	6	5
4	3	2	1

(c)

Figure 5.2: (a) tiles 14 and 15 in each other's goal positions as are tiles 12 and 13; (b) state (a) "reversed"; (c) standard goal state "reversed".

	Standard	Reversed
$\mathcal{U}$	33,798	33,798
$\mathcal{L}$	1,841,122	3,660
$\mathcal{L}^2$	not solved	1,274

Table 5.1: Number of nodes expanded by A* for each cost function on two different problems.

left of the red vertical line in Figure 5.1(a)), but 2,453,415 under $\mathcal{U}$ (to the left of the red vertical line in Figure 5.1(b)), so A* using $\mathcal{U}$ *must* expand millions of states more than it does using $\mathcal{C}$.

5.3.2 Example 2: 15-Puzzle

Consider the following successively more diverse cost functions for the 15-puzzle: unit cost $\mathcal{U}$, "linear" cost $\mathcal{L}$, and "square" cost $\mathcal{L}^2$, where an action a that moves tile T has $\mathcal{L}(a) = T$ and $\mathcal{L}^2(a) = T^2$. Wilt and Ruml [WR11] gave a 15-puzzle task for which the number of nodes expanded by A*, with a suitably adjusted definition of Manhattan Distance (MD), increased dramatically when changing from $\mathcal{U}$ to $\mathcal{L}$ and from $\mathcal{L}$ to $\mathcal{L}^2$. This is illustrated in the column "Standard" of Table 5.1, which shows the number of nodes expanded using each of the cost functions when the initial state is the state shown in Figure 5.2(a) and the goal state is the standard 15-puzzle goal. A* ran out of memory (2GB limit) before it solved this problem using $\mathcal{L}^2$.

In their example, tiles 14 and 15 occupy each other's goal locations in the initial state so each of these tiles must be moved at least once, which requires using the most expensive actions under $\mathcal{L}$ and $\mathcal{L}^2$. For problems whose solution path does not necessarily use the expensive actions, we observe the opposite trend. For example, "reversing" the tile numbers (i.e., each tile T is replaced

with $16 - T$) maps the state in Figure 5.2(a) to the state in Figure 5.2(b) and maps the standard goal state to the state in Figure 5.2(c). The number of nodes expanded by A* to solve this problem is shown in the "Reversed" column of Table 5.1 and here we see the opposite trend: the number decreases as the cost diversity increases.

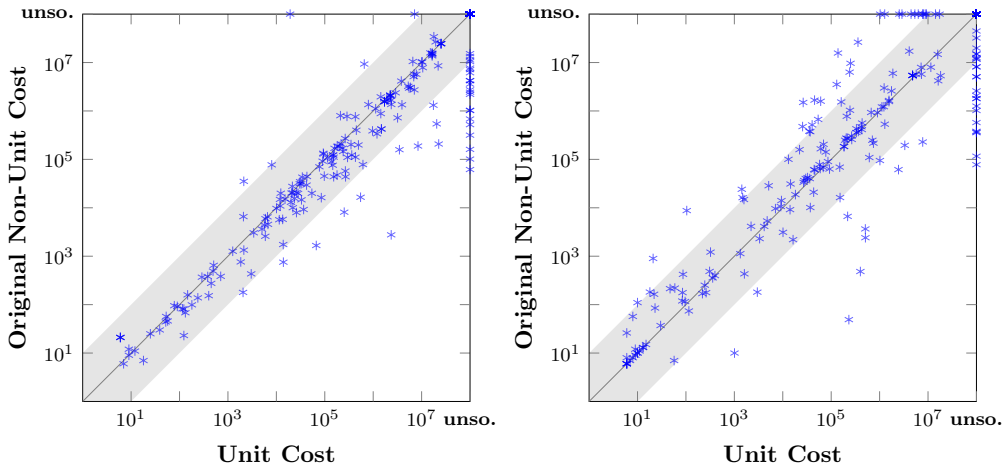
5.3.3 Example 3: Heuristics as Diverse Action Costs

As our final motivating example, the very idea of using a heuristic to guide search can be viewed as taking advantage of action cost diversity, in the following sense. It is well known [ES12; Ike+94; Mar77] that A* using a consistent heuristic h in combination with the original action costs behaves the same as Dijkstra's algorithm using no heuristic but with action costs adjusted as follows. If $\mathcal{C}(a)$ is the original cost of the transition $s \xrightarrow{a} t$ then the adjusted cost $c(s \xrightarrow{a} t)$ of the transition is $\mathcal{C}(a) - h(s) + h(t)$. In this view, the guidance a heuristic provides towards the goal is converted to *guidance towards the goal by diversified cost*: the cost of actions moving towards the goal (as estimated by the heuristic) is reduced, while the cost of actions leading away from the goal is increased.

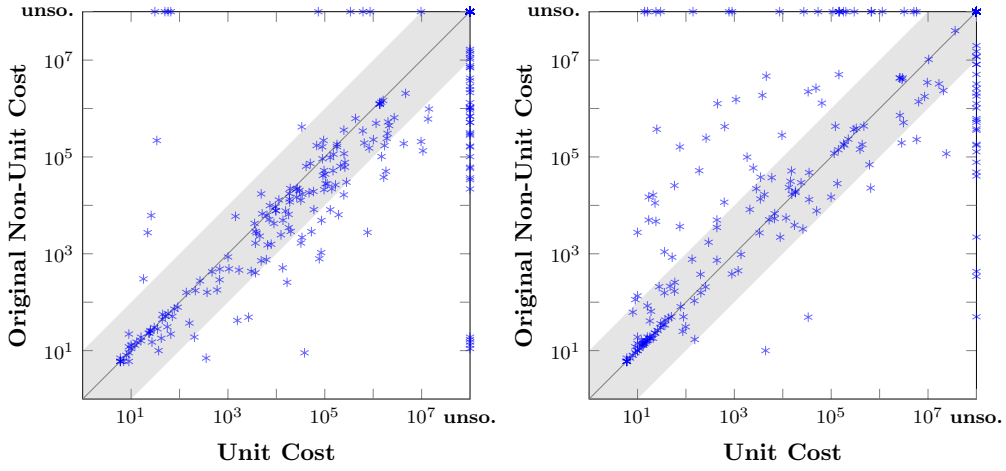
As an illustration, consider the sliding tile puzzle with unit cost and the Manhattan Distance (MD) heuristic. Any action that moves a tile towards its goal position decreases MD by 1, so the adjusted cost of this action is 0. In contrast, any action that moves a tile away from its goal position increases MD by 1 and has an adjusted cost of 2. A* solves problems much faster with MD than with no heuristic, and Dijkstra's algorithm with the corresponding adjusted action costs (but no heuristic) will see the same speedup compared to searching with unit costs. The adjusted cost function has costs of 0 and 2 and is therefore more diverse than the unit cost.

5.4 Diverse Costs in IPC Domains

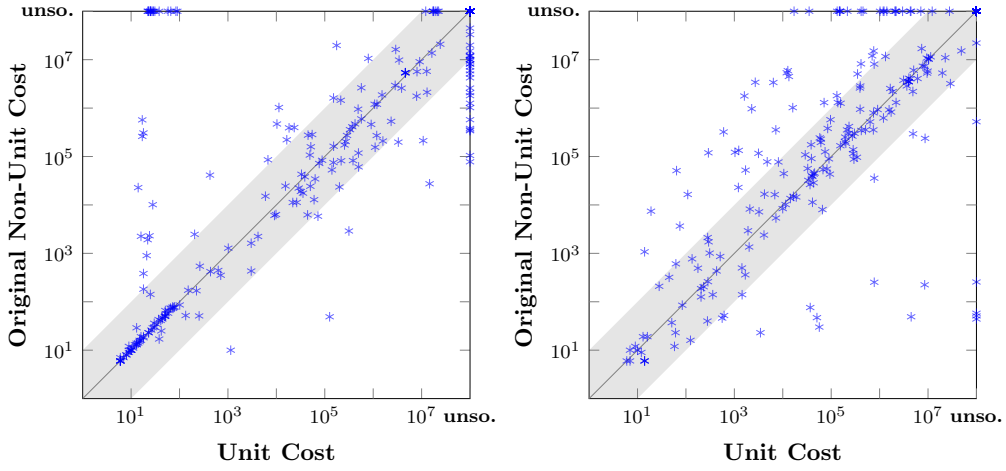
The motivating examples are not accidental. In this section, we experimentally demonstrate that action costs can have positive effects on several IPC



(a) $h^{\max}$ (b) PDB



(c) LM-cut (d) iPDB



(e) M&S (f) CEGAR

Figure 5.3: Comparisons of numbers of A* node expansions for solving non-unit cost tasks with their original IPC costs and the unit cost using heuristics: (a) $h^{\max}$, (b) PDB, (c) LM-cut, (d) iPDB, (e) M&S, and (f) CEGAR.

domains. Our experiments use all 13 IPC domains that have non-unit action costs: Barman, CaveDiving, CityCar, Elevators, FloorTile, OpenStacks, PARCPrinter, Pegsol, Scanalyzer, Sokoban, Tetris, Transport and WoodWorking. We call these domains NUC (stands for "non-unit cost") domains and the tasks in these domains the NUC tasks. The action cost distinctiveness in these domains varies. For example, Tetris tasks use only action costs 1, 2 and 3 while in PARCPrinter tasks the action costs can range from 0 to more than 200,000. An action cost of 0 appears in 6 domains: CityCar, Elevators, OpenStacks, PARCPrinter, Pegsol and Sokoban.

To study the effects of action costs, we create a unit cost version of each task in these IPC domains. We then run A* on the tasks using both unit and non-unit costs and on the same tasks using the unit cost.

5.4.1 Effects on A* with Heuristics

We use admissible heuristics $h^{\max}$ [BG01], PDB [Ede01], LM-cut [HD09], iPDB [Has+07], M&S [Hel+14] and Additive CEGAR [SH14]. For a NUC task and a heuristic, we denote the number of nodes A* expanded to solve the instance using the original IPC action cost function $\mathcal{C}$ by $N_{\mathcal{C}}$ and the number of nodes expanded to solve it using the unit cost function $\mathcal{U}$ by $N_{\mathcal{U}}$. Each plot in Figure 5.3 corresponds to one heuristic. Each point in a plot represents a task whose x value is $N_{\mathcal{U}}$ and y value is $N_{\mathcal{C}}$. Both the x-axis and y-axis use a log scale. A task not solved within 30 minutes and 2 GB memory is plotted on the top border if an original non-unit cost is used and on the right border if the unit cost is used. Tasks below the $y = x$ line represent tasks that required more A* node expansions using $\mathcal{U}$ than using $\mathcal{C}$, i.e., $N_{\mathcal{U}} > N_{\mathcal{C}}$. Tasks in the shaded zone have $\frac{1}{10}N_{\mathcal{C}} \leq N_{\mathcal{U}} \leq 10N_{\mathcal{C}}$, i.e., solving them with $\mathcal{U}$ requires expanding at most 10 times as many nodes as solving them with $\mathcal{C}$ and at least one tenth as many nodes as solving them with $\mathcal{C}$.

For each heuristic used in our experiments, there are tasks on both sides of the diagonal line and of the shaded zone. The numbers of tasks in different regions of the plots in Figure 5.3 are shown in Table 5.2. We first look at the tasks solved with both cost functions, ignoring the tasks on "**unso.**" lines.

Regions	$h^{\max}$	PDB	LM-cut	iPDB	M&S	CEGAR	no heuristic
$N_{\mathcal{C}} =$ **unso.** $N_{\mathcal{U}} \neq$ **unso.**	2	13	9	23	22	22	2
$N_{\mathcal{C}} > 10N_{\mathcal{U}}$	2	19	5	31	22	29	0
$N_{\mathcal{C}} > N_{\mathcal{U}}$	41	84	33	85	68	84	44
$N_{\mathcal{U}} > N_{\mathcal{C}}$	123	40	115	45	57	82	99
$N_{\mathcal{U}} > 10N_{\mathcal{C}}$	17	11	26	6	7	13	16
$N_{\mathcal{U}} =$ **unso.** $N_{\mathcal{C}} \neq$ **unso.**	32	24	38	28	27	6	22
$N_{\mathcal{U}} =$ **unso.** $N_{\mathcal{C}} =$ **unso.**	205	193	166	169	165	177	245

Table 5.2: The numbers of tasks in different regions in plots in Figure 5.3 and Figure 5.4. "**unso.**" indicates the task is unsolved within the time and memory limits.

There are more tasks with $N_{\mathcal{U}} > N_{\mathcal{C}}$ than with $N_{\mathcal{C}} > N_{\mathcal{U}}$ when A* uses the $h^{\max}$ (Figure 5.3(a)) and LM-cut heuristics (Figure 5.3(c)). For these two heuristics, there are also more tasks with $N_{\mathcal{U}} > 10N_{\mathcal{C}}$ than with $N_{\mathcal{C}} > 10N_{\mathcal{U}}$. When using the PDB (Figure 5.3(b)) and iPDB (Figure 5.3(d)) heuristics, however, there are more tasks solved by fewer expansions with the unit cost than with the non-unit costs. For CEGAR (Figure 5.3(f)) and M&S (Figure 5.3(e)), tasks distribute mostly evenly on both sides of the diagonal, but with more above the shaded zone than below.

In terms of the number of tasks solved with one cost function but not the other, all heuristics except additive CEGAR favor non-unit costs over unit costs. In Figure 5.3(a)-(e), a large number of tasks are solved with the IPC costs but not with the unit costs.

5.4.2 Effects on A* Without Heuristics

Since A* uses $g(s) + h(s)$ to prioritize the expansion of states and both $g(s)$ and heuristic generation methods are dependent of action costs, the observed effect of action costs on A* in Figure 5.3 is the combination of effects of action

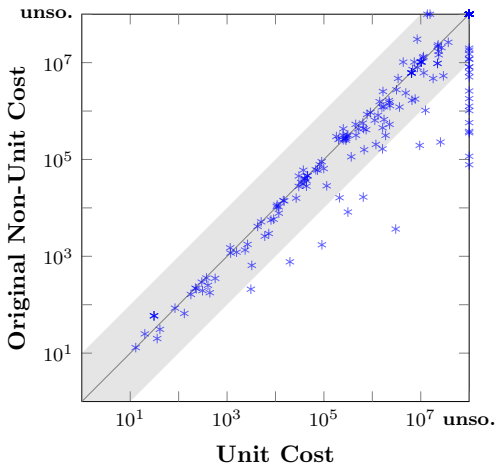


Figure 5.4: Comparing the number of nodes expanded by A* with no heuristic for solving non-unit cost tasks with their original IPC costs and the unit cost.

costs on $g(s)$ and $h(s)$. We now analyze how action costs affect A* through $g(s)$ alone by running A* without using any heuristics.

Figure 5.4 compares $N_{\mathcal{U}}$ and $N_{\mathcal{C}}$ for A* using no heuristic. Overall, in solving these tasks more nodes need to be expanded using $\mathcal{U}$ than using $\mathcal{C}$. The last column of Table 5.2 shows many more tasks with $N_{\mathcal{U}} > N_{\mathcal{C}}$ (99 tasks) than $N_{\mathcal{C}} > N_{\mathcal{U}}$ (44 tasks). There are 22 tasks solved with the non-unit costs but not with unit costs. In contrast, only 2 tasks are solved only with unit cost. In general, Figure 5.4 shows that NUC tasks become harder to solve with unit cost when no heuristics are used. This may be one reason why there are many tasks below the diagonal lines in Figure 5.3.

5.5 No Free Lunch Theorem

The previous experiments on the NUC domains have shown that cost diversity can have both positive and negative impacts on planning. In this section, we provide some theoretical foundation for this observation. Our main result is a *No Free Lunch* (NFL) theorem for A* without heuristics showing that, when excluding the states that have the same cost-from-start (**g**-value) as the goal, the advantage one cost function enjoys over another cost function on some problem instances is *exactly* counterbalanced by the disadvantage it suffers on other problem instances.

In the following, we first give the formal theoretical setting for our theorem. We then present the proof of the NFL theorem. Finally, we use concrete examples to illustrate the theorem.

5.5.1 Theoretical Setting

Our theoretical analysis does not take the effects of a heuristic function into account. Its application is restricted to Dijkstra's shortest path algorithm, which is equivalent to A* (Algorithm 1) with no heuristic. In addition, our analysis considers only transition systems of the form $\langle S, L, \mathcal{C}, T, s_{\text{init}}, \{s_{\text{goal}}\}\rangle$, which have only one goal state s_{goal}. For a fixed state space S, a *problem instance* in S is a pair $(s_{\text{init}}, s_{\text{goal}})$ where $s_{\text{init}} \in S$ is the initial state and $s_{\text{goal}} \in S$ is the goal state.

We analyze the *overall* effects of cost functions on the set of all problem instances with a *fixed* initial state in a state space. In the definition of transition system (Definition 2), the cost function and the goal states are fixed and given. Therefore, for our analysis, we use a slightly different formulation in which we fix everything in a transition system except the cost function $\mathcal{C}$ and the goal state s_{goal}, which are considered as variables.

Let $\mathbf{g}(\mathcal{C}, s)$ denote the cost of a least-cost path from the initial state to state s using the cost function $\mathcal{C}$. The optimal solution cost is $\mathbf{g}(\mathcal{C}, s_{\text{goal}})$. Let $\mathbf{SE}(\mathcal{C}, s_{\text{goal}})$ denote the set of states that are "surely expanded" by Dijkstra's algorithm, which are the states that are strictly closer to s_{init} than s_{goal}, i.e., $\{s \mid \mathbf{g}(\mathcal{C}, s) < \mathbf{g}(\mathcal{C}, s_{\text{goal}})\}$. $\mathbf{NE}(\mathcal{C}, s_{\text{goal}})$ (stands for "never expanded") denotes the set of states strictly further from s_{init} than s_{goal}, i.e., $\{s \mid \mathbf{g}(\mathcal{C}, s) > \mathbf{g}(\mathcal{C}, s_{\text{goal}})\}$. Under no conditions will Dijkstra's algorithm expand states in $\mathbf{NE}(\mathcal{C}, s_{\text{goal}})$. We call the states with $\mathbf{g}(\mathcal{C}, s) = \mathbf{g}(\mathcal{C}, s_{\text{goal}})$ the *TIE* states. Dijkstra's algorithm might expand some (or all) of the TIE states or it might expand none. We begin our analysis ignoring the TIE states in this section. Section 5.6 studies the effects of TIE states specifically.

5.5.2 The NFL Theorem

We compare the sets $\mathbf{SE}(\mathcal{C}, s_{\text{goal}})$ and $\mathbf{NE}(\mathcal{C}, s_{\text{goal}})$ for two different cost functions, $\mathcal{C}$ and $\mathcal{C}'$, considering all possible goal states s_{goal} in the state space. We say that the pair of states (s, t) *favors* $\mathcal{C}$ *over* $\mathcal{C}'$ if, when the goal state is t, s is surely expanded when the cost function is $\mathcal{C}'$ but never expanded when it is $\mathcal{C}$ (Definition 41). We then show that the number of pairs that favor $\mathcal{C}$ over $\mathcal{C}'$ is exactly the same as the number of pairs that favor $\mathcal{C}'$ over $\mathcal{C}$ (Theorem 1). This immediately implies a No Free Lunch theorem (Theorem 2), which can be paraphrased thus: for a given start state, no cost function is any "easier" for Dijkstra's algorithm than any other cost function when all possible goal states are taken into consideration and TIE states are ignored.

Note that we consider all possible goal states but fix the initial state because the key to our NFL theorem is to take an average of the performance difference between $\mathcal{C}$ and $\mathcal{C}'$ over a set of problem instances, and the minimal set we could find that had an average of 0 was the set defined by any one fixed initial state paired with all possible goal states. This set of instances having an average of 0 immediately implies that an average of 0 will also be obtained by taking any set of initial states (for example, all possible initial states) as long as each is paired with all possible goal states.

The key fact underpinning our analysis is the following simple observation:

Lemma 1. For any cost function $\mathcal{C}$ and any states s and t, $s \in \mathbf{SE}(\mathcal{C}, t)$ if and only if $t \in \mathbf{NE}(\mathcal{C}, s)$.

Proof.

$$\begin{aligned} s \in \mathbf{SE}(\mathcal{C}, t) &\iff \mathbf{g}(\mathcal{C}, s) < \mathbf{g}(\mathcal{C}, t) \\ &\iff \mathbf{g}(\mathcal{C}, t) > \mathbf{g}(\mathcal{C}, s) \\ &\iff t \in \mathbf{NE}(\mathcal{C}, s)\,. \end{aligned}$$

□

Definition 41. A pair of states (s, t) *favors* $\mathcal{C}$ *over* $\mathcal{C}'$ if $s \in \mathbf{NE}(\mathcal{C}, t)$ and $s \in \mathbf{SE}(\mathcal{C}', t)$. $favor(\mathcal{C}, \mathcal{C}')$ is the set of state pairs that favor $\mathcal{C}$ over $\mathcal{C}'$.

Applying Lemma 1 to the definitions of $favor(\mathcal{C}, \mathcal{C}')$ and $favor(\mathcal{C}', \mathcal{C})$ gives the following:

Theorem 1. For any two cost functions $\mathcal{C}$ and $\mathcal{C}'$ there is a one-to-one correspondence between the elements of $favor(\mathcal{C}, \mathcal{C}')$ and $favor(\mathcal{C}', \mathcal{C})$.

Proof. For any $(s,t) \in favor(\mathcal{C}, \mathcal{C}')$, $s \in \mathbf{NE}(\mathcal{C}, t)$ and $s \in \mathbf{SE}(\mathcal{C}', t)$. This implies, by Lemma 1, that $t \in \mathbf{SE}(\mathcal{C}, s)$ and $t \in \mathbf{NE}(\mathcal{C}', s)$, i.e., that $(t,s) \in favor(\mathcal{C}', \mathcal{C})$. Therefore the mapping $\phi(s,t) = (t,s)$ is a one-to-one correspondence between $favor(\mathcal{C}, \mathcal{C}')$ and $favor(\mathcal{C}', \mathcal{C})$. □

Definition 42. For a goal state t, and two cost functions $\mathcal{C}$ and $\mathcal{C}'$, $\delta_t(\mathcal{C}, \mathcal{C}')$ is the number of states s such that (s,t) favors $\mathcal{C}$ over $\mathcal{C}'$.

By definition, we have

$$\begin{aligned} |favor(\mathcal{C}, \mathcal{C}')| &= \left| \bigcup_{t \in S} \{(s,t) \mid (s,t) \in favor(\mathcal{C}, \mathcal{C}')\} \right| \\ &= \sum_{t \in S} \delta_t(\mathcal{C}, \mathcal{C}') \,. \end{aligned} \tag{5.1}$$

Let $\Delta_t(\mathcal{C}, \mathcal{C}') = \delta_t(\mathcal{C}, \mathcal{C}') - \delta_t(\mathcal{C}', \mathcal{C})$ whose absolute value $|\Delta_t(\mathcal{C}, \mathcal{C}')|$ indicates how many *more* (if $\Delta_t(\mathcal{C}, \mathcal{C}') > 0$) or *fewer* (if $\Delta_t(\mathcal{C}, \mathcal{C}') < 0$) states are expanded when solving the problem with $\mathcal{C}'$ than with $\mathcal{C}$ when the goal is t. By definition, $\Delta_t(\mathcal{C}, \mathcal{C}') = -\Delta_t(\mathcal{C}', \mathcal{C})$.

Definition 43. For cost functions $\mathcal{C}$ and $\mathcal{C}'$ we say state t is a goal state that favors $\mathcal{C}$ over $\mathcal{C}'$ if $\Delta_t(\mathcal{C}, \mathcal{C}') > 0$. *goals-favor*$(\mathcal{C}, \mathcal{C}')$ is the set of goal states that favor $\mathcal{C}$ over $\mathcal{C}'$.

The following theorem shows that the *advantage* that $\mathcal{C}$ enjoys over $\mathcal{C}'$ on the goal states that favor $\mathcal{C}$ over $\mathcal{C}'$ is *exactly* counterbalanced by the *disadvantage* it suffers on the goal states that favors $\mathcal{C}'$ over $\mathcal{C}$.

Theorem 2 (No Free Lunch Theorem)**.** For any two cost functions $\mathcal{C}$ and $\mathcal{C}'$,

$$\sum_{t \in \textit{goals-favor}(\mathcal{C}, \mathcal{C}')} \Delta_t(\mathcal{C}, \mathcal{C}') \;=\; \sum_{t \in \textit{goals-favor}(\mathcal{C}', \mathcal{C})} \Delta_t(\mathcal{C}', \mathcal{C}) \,.$$

Proof.

$$
\begin{aligned}
& |\mathit{favor}(\mathcal{C}, \mathcal{C}')| = |\mathit{favor}(\mathcal{C}', \mathcal{C})| && \text{(by Theorem 1)} \\
\iff & \sum_{t \in S} \delta_t(\mathcal{C}, \mathcal{C}') = \sum_{t \in S} \delta_t(\mathcal{C}', \mathcal{C}) && \text{(by Equation (5.1))} \\
\iff & 0 = \sum_{t \in S} \delta_t(\mathcal{C}, \mathcal{C}') - \delta_t(\mathcal{C}', \mathcal{C}) = \sum_{t \in S} \Delta_t(\mathcal{C}, \mathcal{C}') \,. && (5.2)
\end{aligned}
$$

Let $G^+ = \mathit{goals\text{-}favor}(\mathcal{C}, \mathcal{C}') = \{t \mid \Delta_t(\mathcal{C}, \mathcal{C}') > 0\}$, $G^- = \mathit{goals\text{-}favor}(\mathcal{C}', \mathcal{C}) = \{t \mid \Delta_t(\mathcal{C}, \mathcal{C}') < 0\}$ and $G^0 = \{t \mid \Delta_t(\mathcal{C}, \mathcal{C}') = 0\}$. Since $S = G^+ \cup G^- \cup G^0$, we have

$$
\begin{aligned}
& \text{Equation (5.2)} \\
\iff 0 \; &= \sum_{t \in G^+ \cup G^- \cup G^0} \Delta_t(\mathcal{C}, \mathcal{C}') \; = \sum_{t \in G^+ \cup G^-} \Delta_t(\mathcal{C}, \mathcal{C}') \\
\iff 0 \; &= \sum_{t \in G^+} \Delta_t(\mathcal{C}, \mathcal{C}') + \sum_{t \in G^-} \Delta_t(\mathcal{C}, \mathcal{C}') \\
\iff 0 \; &= \sum_{t \in \mathit{goals\text{-}favor}(\mathcal{C}, \mathcal{C}')} \Delta_t(\mathcal{C}, \mathcal{C}') \; + \sum_{t \in \mathit{goals\text{-}favor}(\mathcal{C}', \mathcal{C})} \Delta_t(\mathcal{C}, \mathcal{C}') \\
\iff 0 \; &= \sum_{t \in \mathit{goals\text{-}favor}(\mathcal{C}, \mathcal{C}')} \Delta_t(\mathcal{C}, \mathcal{C}') \; - \sum_{t \in \mathit{goals\text{-}favor}(\mathcal{C}', \mathcal{C})} \Delta_t(\mathcal{C}', \mathcal{C}) \\
\iff & \sum_{t \in \mathit{goals\text{-}favor}(\mathcal{C}, \mathcal{C}')} \Delta_t(\mathcal{C}, \mathcal{C}') \; = \sum_{t \in \mathit{goals\text{-}favor}(\mathcal{C}', \mathcal{C})} \Delta_t(\mathcal{C}', \mathcal{C}) \,.
\end{aligned}
$$

□

From a probabilistic point of view, Theorem 1 also implies that the expected performance difference between two cost functions is zero when goal states are drawn uniformly at random and TIE states are ignored.

Theorem 3. Let $\boldsymbol{D}$ be the random variable for $\Delta_t(\mathcal{C}, \mathcal{C}')$ when the goal state t is drawn uniformly at random from the state space S. Let $G_d = \{t \mid \Delta_t(\mathcal{C}, \mathcal{C}') = d\}$ and let $P(\boldsymbol{D} = d) = |G_d|/|S|$ be the probability that $\boldsymbol{D} = d$. The expected value of $\boldsymbol{D}$ is zero, i.e.,

$$
\mathbf{E}[\boldsymbol{D}] = \sum_d \left(d \cdot P(\boldsymbol{D} = d)\right) = 0 \,.
$$

Proof. For any d,

$$d \cdot P(\boldsymbol{D} = d) = \frac{d \cdot |G_d|}{|S|} = \frac{\sum_{t \in G_d} \Delta_t(\mathcal{C}, \mathcal{C}')}{|S|} .$$

Because $\sum_{t \in S} \Delta_t(\mathcal{C}, \mathcal{C}') = 0$ (Equation (5.2)) and $S = \bigcup_d G_d$,

$$\sum_d d \cdot P(\boldsymbol{D} = d) = \frac{\sum_{t \in S} \Delta_t(\mathcal{C}, \mathcal{C}')}{|S|} = 0 .$$

□

5.5.3 Example: ε-Cost Cycle Traps

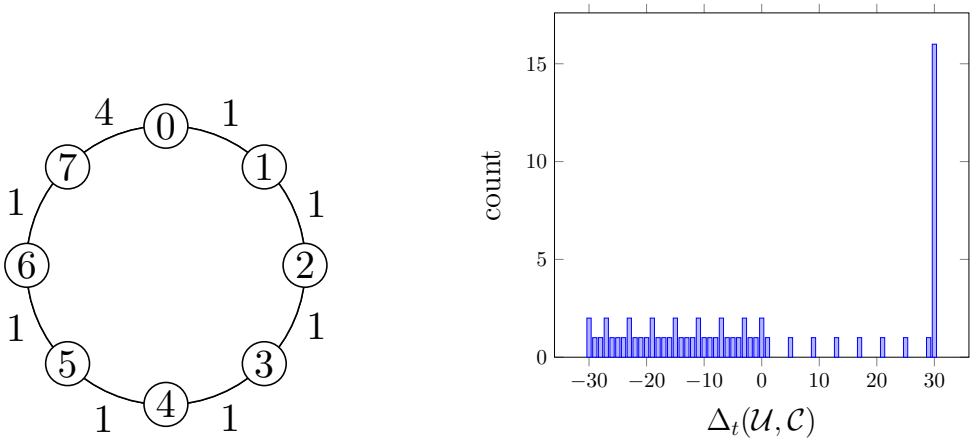
The No Free Lunch theorem applies to all state spaces, including the "ε-cost traps" that Cushing et al. [CBK10; CBK11] designed to illustrate that search with a non-unit cost function $\mathcal{C}$ can expand exponentially more states than search with the unit cost function $\mathcal{U}$. In their discussion of these traps, Cushing et al. focused on the specific goal states that revealed this exponential difference. Our NFL theorem says that these differences in favor of $\mathcal{U}$ will be exactly counterbalanced by performance on other goal states in favor of $\mathcal{C}$.

We will illustrate this with one of the traps Cushing et al. defined, the cycle trap. The states in a cycle trap are the integers from 0 to $2^k - 1$ for some k. Actions increment or decrement an integer modulo 2^k, including the overflow increment (increase $2^k - 1$ to 0), and the overflow decrement (decrease 0 to $2^k - 1$). In the non-unit cost function $\mathcal{C}$ defined by Cushing et al. the normal increment/decrement actions cost 1 and the overflow actions cost 2^{k-1}. Figure 5.5(a) shows the cycle trap for $k = 3$, where the numbers in the circles identify the states and the numbers next to the edges are the costs of actions.

For the initial state $s_{\text{init}} = 0$, if the goal state t is close to $2^k - 1$ many fewer nodes will be expanded using $\mathcal{U}$ than using $\mathcal{C}$. For example, when $t = 2^k - 2$, Dijkstra's algorithm using $\mathcal{U}$ expands 3 states while the search using $\mathcal{C}$ has to expand $2^{k-1} + 2$ states. The NFL theorem says that the advantage for $\mathcal{U}$ on goal states such as $2^k - 2$ is exactly counterbalanced by the disadvantage for $\mathcal{U}$ on other goal states. Figure 5.5(b) illustrates this for the cycle trap for $k = 6$. It shows how many states t have a given value of $\Delta_t(\mathcal{U}, \mathcal{C})$ (x-axis).

k	$N_{\mathcal{U}}(k)$	$N_{\mathcal{C}}(k)$	$N_{\mathcal{C}}(k) - N_{\mathcal{U}}(k)$
2	5	6	1
3	25	26	1
4	113	116	3
5	481	488	7
6	1985	2000	15
7	8065	8096	31
8	32513	32576	63
9	130561	130688	127
10	523265	523520	255
11	2095105	2095616	511
12	8384513	8385536	1023
13	33546241	33548288	2047
14	134201345	134205440	4095
15	536838145	536846336	8191
16	2147418113	2147434496	16383

Table 5.3: Total number of nodes expanded with cost functions $\mathcal{U}$ and $\mathcal{C}$ and their difference for cycle traps of various sizes.



(a) (b)

Figure 5.5: (a) The cycle trap for $k = 3$; (b) Histogram for $\Delta_t(\mathcal{U},\mathcal{C})$, for the cycle trap with $k = 6$.

There are 64 states in this space, and a quarter of them have a large positive $\Delta_t(\mathcal{U}, \mathcal{C})$ value (the spike at the right edge of the figure). However, the benefit of $\mathcal{U}$ over $\mathcal{C}$ on these and the other states having $\Delta_t(\mathcal{U}, \mathcal{C}) > 0$ are exactly counterbalanced by the benefit of $\mathcal{C}$ over $\mathcal{U}$ on the states having $\Delta_t(\mathcal{U}, \mathcal{C}) < 0$.

Table 5.3 shows $N_{\mathcal{C}}$ and $N_{\mathcal{U}}$, the total number of nodes Dijkstra's algorithm expanded with all possible goal states on cycle traps of sizes $k = 2$ to $k = 16$ using $\mathcal{C}$ and $\mathcal{U}$ respectively.

As can be seen, $N_{\mathcal{C}}$ is not exactly equal to $N_{\mathcal{U}}$, it is always slightly larger. This can also be seen in the following formulas, which give the exact number of nodes expanded for any k:

$$N_{\mathcal{U}}(k) = 2^{2k-1} - 2^k + 1\,,$$

$$N_{\mathcal{C}}(k) = 2^{2k-1} - 3 \cdot 2^{k-2}\,.$$

The difference, $N_{\mathcal{C}}(k) - N_{\mathcal{U}}(k)$, is $2^{k-2} - 1$.

This difference is tiny relative to the total number of states expanded, but not zero, because all cycle trap state spaces contain slightly more TIE states with $\mathcal{U}$ than they do with $\mathcal{C}$. The NFL theorem assures us that $N_{\mathcal{C}}$ and $N_{\mathcal{U}}$ would be exactly equal if the effect of the TIE states was removed. In the next section we show that often (but not always) the TIE states work in favor of the unit cost distribution, as they did in this example.

5.6 Goal-Preference Tie-Breaking

The NFL theorem holds as long as there is no bias[2] to a particular problem taken into consideration. Such problem-specific biases include the tie-breaking strategy of best-first search, which breaks ties in favor of a goal state of a particular problem (i.e., goal states are expanded before non-goal states in tie-breaking). When this tie-breaking strategy is used, the expansion order of TIE states would be different for different goals, and thus the one-to-one correspondence in Theorem 1 is not guaranteed to exist between those states.

[2] Excluding problem bias is also required for NFL theorems for general search and optimization [Dav95; Dav97].

For any cost function, let $n \in \mathbb{N}$ be the number of distances from the initial state to all states in the state space and A_i for $i \in \{1, 2, ..., n\}$ be the set of states that have the i-th smallest distance, and let $a_i = |A_i|$ for $i \in \{1, 2, ..., n\}$. If the goal state is in A_i, the states in A_j for all $j < i$ are surely expanded regardless of the tie-breaking strategy. The total number of surely expanded states for *all* goal states in A_i is $SE(A_i) = a_i \sum_{j=1}^{i-1} a_j$. States in A_i are the TIE states whose expansion depends on the tie-breaking strategy. Let a *bias-free* search be Dijkstra's search algorithm that does not break ties in favor of a goal state and expands TIE states in a goal-independent fixed order, such as a lexicographic order of states. For a bias-free search, the total number of states expanded when the goal state is from A_i is $SE(A_i) + 0 + 1 + 2 + \cdots + a_i - 1 = SE(A_i) + \frac{a_i \cdot (a_i - 1)}{2}$, where $\frac{a_i \cdot (a_i - 1)}{2}$ is the cumulative number of extra expansions in addition to the sure expansions for all goal states in A_i by the bias-free search.

By contrast, if Dijkstra's algorithm breaks ties in favor of a goal state and there are no zero cost actions, none of these extra expansions by the bias-free search are needed before the goal from A_i is found, which means the first state to be checked after all sure expansions is the goal state. In this best case, the total number of expansions *saved* by tie-breaking over all goal states from all A_i is thus

$$\sum_{i=1}^{n} \frac{a_i \cdot (a_i - 1)}{2} = \frac{1}{2} \sum_{i=1}^{n} a_i^2 - \frac{|S|}{2}\,, \tag{5.3}$$

and the total number of expansions by Dijkstra's algorithm is equal to

$$\begin{aligned}
&\sum_{i=1}^{n} SE(A_i) = \sum_{i=1}^{n} \Big(a_i \cdot \sum_{j=1}^{i-1} a_j \Big) \\
&= (a_1 a_2) + (a_1 a_3 + a_2 a_3) + \cdots + (a_1 a_n + a_2 a_n + a_3 a_n + \cdots + a_{n-1} a_n) \\
&= \frac{(a_1 + a_2 + \cdots + a_n)^2 - (a_1^2 + a_2^2 + \cdots + a_n^2)}{2} \\
&= \frac{1}{2}(|S|^2 - \sum_{i=1}^{n} a_i^2)
\end{aligned} \tag{5.4}$$

The savings due to the best-case tie-breaking depend on the sum-of-squares $\sum_{i=1}^{n} a_i^2$. The larger this sum is, the more expansions are saved by tie-breaking (Equation (5.3)) and the fewer number of states are expanded by Dijkstra's

	unit	linear	square	inverse	sqrt
Predict	82,676	88,966	90,426	90,501	90,663
Actual	82,185	88,318	89,714	90,026	90,021

Table 5.4: The actual average performance of the search and the prediction based on Equation (5.4) on the 8-puzzle.

algorithm with tie-breaking (Equation (5.4)). This sum-of-squares is smaller when the distribution of a_i is more spread out and the minimum value is achieved when $a_i = 1$ for all i. It is larger when the distribution is more concentrated, and the maximum value of $|S|^2$ is achieved when $n = 1$ and $a_1 = |S|$. However, this extreme case requires all action costs to be 0 which means the best case tie-breaking is not guaranteed, i.e., additional expansions may be needed to reach the goal state after all surely expanded states are expanded. In practice, the sum-of-squares is much smaller than $|S|^2$.

Our experiments on the 8-puzzle show that the actual average performance of the search matches the prediction of Equation (5.4). We tested the five cost functions for the 8-puzzle used previously [Wil14; WR11]. In Table 5.4, the entry "Predict" contains the average of $(|S|^2 - \sum_{i=1}^{n} a_i^2)/2$ of 1,500 random initial states, from which the exhaustive search runs and computes a_i values. The row "Actual" contains the average performance of search over $10,000$ random pairs of initial states and goal states.

The benefit of the tie-breaking is linked to how concentrated the distance distribution is. On one hand, cost functions that provide diverse action costs but no 0 cost seem to naturally induce a more spread out distribution than the unit cost function, which means that unit action cost can benefit more from tie-breaking. On the other hand, cost functions that contain 0 cost may induce a distance distribution that is even more concentrated than that of the unit cost function, suggesting actions with 0 cost may bring some advantages through tie-breaking. Nevertheless, 0 cost actions means additional TIE states may be expanded before reaching the goal state, and they may cause "g-value plateaus" which increase the number of nodes expanded [Ben+10].

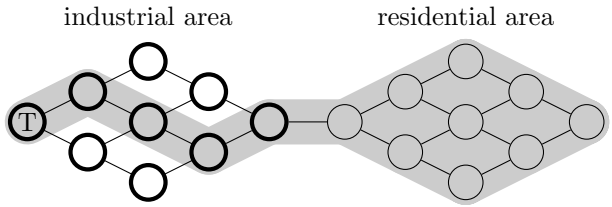


Figure 5.6: An illustration for hazardous logistics. Thicker circles represent industrial locations and thinner circles represent residential locations. The gray area indicates locations that a residential mode truck can visit. A truck starts at location T.

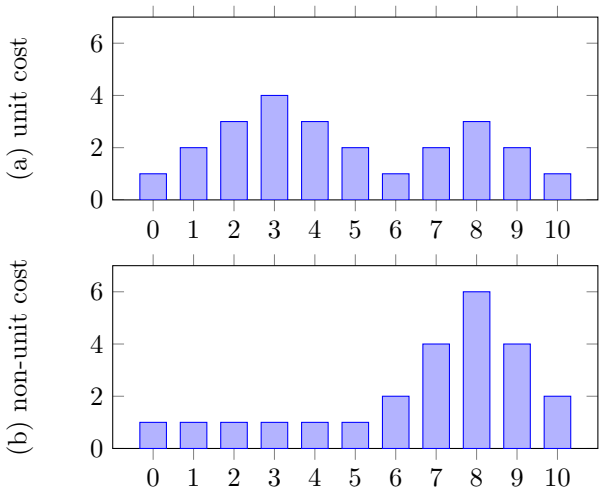


Figure 5.7: For the example of Figure 5.6, distance distribution induced by unit cost function (a) is less concentrated than that of non-unit cost function (b).

5.6.1 Hazardous Logistics

While unit action costs *often* produce a more concentrated distance distribution than non-unit action cost, this is not always the case. In this section, we construct a transition system where the non-unit cost function induces the more concentrated distribution.

Consider a logistics-like problem where the locations are grouped into residential and industrial regions. In industrial regions, hazardous products may be transported. Each truck can be in one of three modes: unassigned, residential and industrial. An unassigned truck cannot move, but can be assigned to residential or industrial mode at most once (to avoid the risk of mixing hazardous and safe products). An industrial mode truck is restricted to move

within the industrial region. A residential mode truck can only visit a corridor subset of industrial locations. For simplicity, we consider only moving a truck, without transporting any product. See Figure 5.6 for an illustration. In the initial state, the truck is at the leftmost location T on the map, in the unassigned mode. The first step for using this truck is to make a choice between industrial and residential mode. If the truck is changed to industrial mode, then it can move only in the industrial area. If the truck is changed to residential mode, then it first moves across the industrial region and then in the residential area. In different modes, the truck generates different sets of states. With unit cost, the reachable states correspond to two peaks in a bimodal distribution since the residential area is further away (Figure 5.7 (a)). However, if we assign a cost of 6 for changing the truck into industrial mode, then we get a more concentrated distribution as the two peaks in unit cost become a single peak in a unimodal distribution (Figure 5.7 (b)). The sum-of-squares is 62 for the unit cost function, and 82 for non-unit cost.

5.7 Conclusions

In this chapter, we have studied how action costs affect search. We started with examples and experimental results showing that diverse cost functions *can* be beneficial for search and planning. Increasing the cost of an action can have either a positive or a negative effect on a given problem instance: it can lead to additional search by delaying the application of an expensive but necessary action, but it can also accelerate the search by blocking irrelevant actions and making a path to a goal more attractive to search.

We then focused on a theoretical analysis of effects of action costs on search without heuristics. While analyzing single instances can be useful in practice, we have shown that the advantage of one cost function over another largely disappears when averaging over problem instances within the same state space. Our No Free Lunch theorem makes this claim precise. Furthermore, we have analyzed the effect of tie-breaking, and shown that its effect on search efficiency is controlled by the concentration of the distribution of path costs. Unit costs

often have better concentration than non-unit costs but we have introduced a new planning domain, Hazardous Logistics, in which the opposite is true.

Chapter 6

Additive Merge-and-Shrink Heuristics for Diverse Action Costs

In this chapter, we study the effects of diverse action costs on merge-and-shrink algorithms. We first demonstrate that action cost diversity has negative impacts on M&S heuristics on IPC domains. We then propose a new *cost partitioning scheme* called *delta cost partitioning* (DCP) to address these negative effects. Our experiments show that M&S using DCP produces a set of *additive* M&S heuristics whose combination is much more informative than a single M&S heuristic that directly encodes the original diverse costs. This is chapter is based on publication [FMH17a].

6.1 Introduction

There have been several studies on the effects of action cost diversity on heuristic search [AB15; AB16; CBK10], Chapter 5, and on improved search algorithms designed to address diverse action costs [TR11; WR11; WR14]. However, to the best of our knowledge, there is no study on the influence of diverse action costs on the construction and performance of domain-independent heuristics in the planning literature.

While merge-and-shrink heuristics can be computed for general non-unit action costs, little is known about how action cost diversity affects this abstraction method. In this chapter, we first experimentally analyze the impact

of action cost diversity on merge-and-shrink, and then develop an improved algorithm, called *DCP-MS*, which computes additive M&S heuristics for diverse action costs.

After a brief discussion of background in Section 6.2, we show in Section 6.3 that the M&S heuristic suffers from diverse action costs on several domains from IPC, compared to their unit cost counterparts. The section includes an in-depth analysis of the experimental results. Motivated by these results, in Section 6.4 we propose a new cost partitioning method, *delta cost partitioning*, which limits the cost diversity of partitioned cost functions to only two distinct costs so that M&S can produce better heuristics.

The cost partitioning exploits the power of M&S for unit costs to help improve the quality of the heuristic for the non-unit cost case. Our experiments in Section 6.5 show that an additive set of M&S abstractions using delta cost partitioning can produce much more informed heuristics than a single M&S heuristic that uses the diverse action costs directly. The benefits are most pronounced in domains where action cost diversity has extremely negative effects on M&S. Our results show a significant improvement to M&S's handling of diverse action costs. At last, we prove that for IPC domain GRIPPER with non-unit costs, DCP-MS produces perfect heuristics with polynomial size M&S abstractions.

6.2 Background

Let Θ be a transition system with state space S. A set of heuristic functions h_1, h_2, ..., h_n is *additive* for Θ if $h_{\text{sum}}(s) = \sum_{i=1}^{n} h_i(s)$ for any $s \in S$ is an admissible heuristic for Θ. *Cost partitioning* is a way to distributes the cost of an action to a set of costs.

Definition 44 (Cost Partitioning)**.** Let L be a transition label set and $\mathcal{C}$ a cost function on L. A cost partitioning of $\mathcal{C}$ is a set of cost functions $\mathcal{C}_1$, $\mathcal{C}_2$, ..., $\mathcal{C}_n$ on L, such that $\sum_{i=1}^{n} \mathcal{C}_i(l) = \mathcal{C}(l)$ for any $l \in L$.

For any cost function $\mathcal{C}$ for Θ and any abstraction mapping α for Θ, we use $h_\alpha^{\mathcal{C}}$ to denote the abstraction heuristic induced by α with cost function $\mathcal{C}$.

A cost partitioning can be used to produce additive heuristics. Let $\mathcal{C}$ be the cost function for transition system Θ. For any $n \in \mathbb{N}$, any cost partitioning $\mathcal{C}_1$, $\mathcal{C}_2$, ..., $\mathcal{C}_n$ of $\mathcal{C}$ and any abstraction mapping α_1, α_2, ..., α_n for Θ, the set of heuristics $h^{\mathcal{C}_1}_{\alpha_1}$, $h^{\mathcal{C}_2}_{\alpha_2}$, ..., $h^{\mathcal{C}_n}_{\alpha_n}$ is additive for Θ [Yan+08].

For the M&S configuration in this chapter, we use the `DFP` merging strategy, bisimulation shrinking with a maximum of 50,000 states per abstraction and label reduction before shrinking unless stated otherwise. Each run, which consists of the M&S construction and the A* search, has a 30 minute time limit and 2 GB memory limit.

6.3 Action Cost Diversity and M&S

In this section, we first inspect the effects of diverse action costs on M&S heuristics and show that cost diversity has a negative impact. We then discuss how diverse action costs can affect the M&S construction process.

6.3.1 Experimental Inspection

An accurate way to inspect the effects of diverse costs on a heuristic is to check how A* using the heuristic performs differently with unit and non-unit cost. In order to do that, one has to run A* with these heuristics. We have such results for M&S in Chapter 5. However, as we have also seen that cost diversity affects A* even without heuristics. To analyze the influence of costs on heuristics, we need to separate the influence on search without heuristics and on the heuristic function itself. Thus, in the following, we use the experimental results both with and without M&S heuristics for our analysis.

Our test set contains the same 12 IPC non-unit action cost domains used in Chapter 5 with an additional domain of Gripper whose IPC cost is unit cost. In a Gripper task, the objects are a robot, a set of balls and two rooms. The robot needs to transport the balls from one room to the other. The robot has two "gripper" hands. Each hand can carry one ball at a time. We include Gripper because it is a domain for which M&S can produce bisimulations with polynomial size which result in perfect heuristics [NHH11] when unit costs are

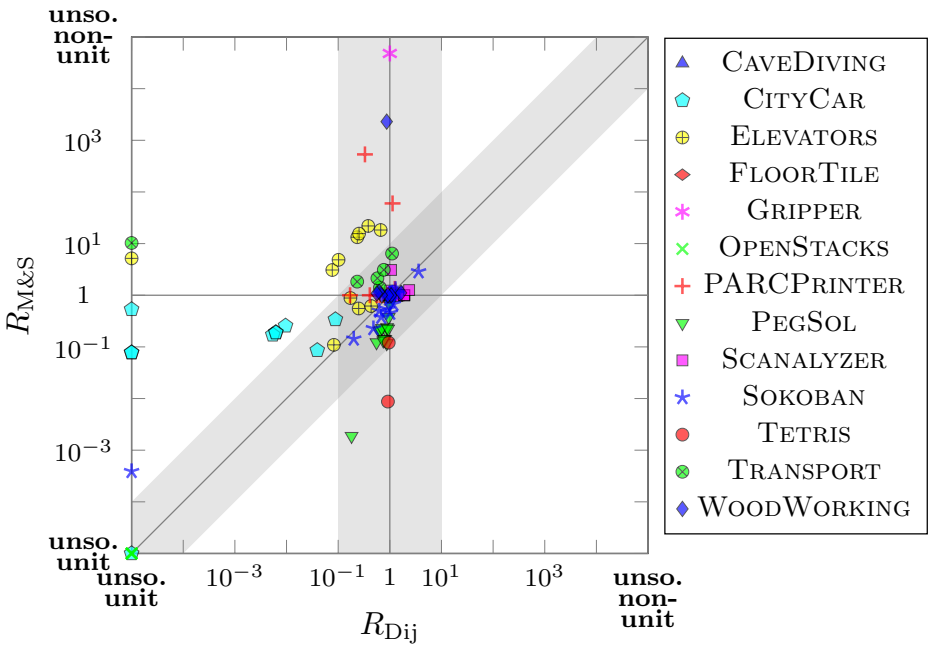


Figure 6.1: R_{Dij} vs. $R_{\text{M\&S}}$ on tasks solved by both A* with M&S and Dijkstra's algorithm

used. We build a non-unit action cost version of Gripper tasks as follows: the `move` action still costs 1, but `pick-up` and `drop` actions for ball b now have cost $(b \bmod n_c) + 1$, where $n_c = 4$ in this experiment.

Commonly Solved Tasks

For any planning task, let $\mathcal{U}$ and $\mathcal{C}$ denote the unit cost function and the non-unit cost function respectively. We compute the ratio $R_{\text{M\&S}} = N^{\mathcal{C}}_{\text{M\&S}}/N^{\mathcal{U}}_{\text{M\&S}}$ for each task, where $N^{\mathcal{C}}_{\text{M\&S}}$ and $N^{\mathcal{U}}_{\text{M\&S}}$ are the numbers of node expansions by A* with M&S heuristics using $\mathcal{C}$ and $\mathcal{U}$ respectively. $R_{\text{M\&S}}$ is the y-axis of Figure 6.1 (the x-axis is defined below, for now we only consider $R_{\text{M\&S}}$, the height of each point in Figure 6.1). The plot uses a log scale and $R_{\text{M\&S}} = 1$ is the horizontal line in the middle of the plot. Points above this line represent tasks for which $R_{\text{M\&S}} > 1$, i.e. A* with M&S expands more nodes (performs worse) with $\mathcal{C}$ than it does with $\mathcal{U}$. Points below the line are tasks on which A* with M&S performs better with $\mathcal{C}$ than with $\mathcal{U}$. If a task is solved using $\mathcal{C}$ but not $\mathcal{U}$ it is placed at the bottom of the plot (on the line labelled **unso. unit**). There are no tasks solved using $\mathcal{U}$ but not $\mathcal{C}$ in this plot. Tasks that

Ranges of R_{Dij}	No. Tasks
$[0, 1)$	61
$[0, 0.1)$	17
$(1, \infty)$	36
$(10, \infty)$	0

(a)

Ranges of $\frac{R_{\text{M\&S}}}{R_{\text{Dij}}}$	No. Tasks
$[0, 1)$	41
$[0, 0.1)$	3
$(1, \infty)$	53
$(10, \infty)$	19

(b)

Table 6.1: The numbers of tasks in specific regions in Figure 6.1

are unsolved with both $\mathcal{U}$ and $\mathcal{C}$ are not shown in the plot.

Because changing the action costs can affect search performance even without using any heuristic, points with $R_{\text{M\&S}} > 1$ may have nothing to do with M&S's performance with $\mathcal{C}$. It may be that these tasks are simply harder to solve with $\mathcal{C}$ than with $\mathcal{U}$. In order to separate the effect of diverse costs on M&S from their effect on overall search performance we solve all the tasks again, with both cost functions, but without a heuristic. Analogous to $R_{\text{M\&S}}$ we define R_{Dij} to be the ratio of the number of nodes Dijkstra's algorithm expands using $\mathcal{C}$ to the number of nodes it expands using $\mathcal{U}$. This is the x-axis in Figure 6.1. Like the y-axis it is a log-scale. $R_{\text{Dij}} = 1$ is the vertical line in the middle of the plot, and tasks solved using $\mathcal{C}$ but not $\mathcal{U}$ are placed on the left edge of the plot on the line labelled **unso. unit**. The vertical gray zone indicates $10^{-1} \leq R_{\text{Dij}} \leq 10$. Note that Dijkstra's algorithm performs better with $\mathcal{C}$: there are more tasks with $R_{\text{Dij}} < 1$ than $R_{\text{Dij}} > 1$ and more with $R_{\text{Dij}} < 10^{-1}$ than $R_{\text{Dij}} > 10$. This has been observed in Chapter 5.

The points above the diagonal line $y = x$ are tasks for which $R_{\text{M\&S}} > R_{\text{Dij}}$, meaning that *compared to* Dijkstra's algorithm, A* with M&S heuristic performs worse with $\mathcal{C}$ than with $\mathcal{U}$. For such tasks, the M&S heuristic may reduce the number of node expansions for both $\mathcal{C}$ and $\mathcal{U}$, but the reduction is proportionally less for $\mathcal{C}$ than for $\mathcal{U}$. For tasks below the line $y = x$, M&S performs worse using $\mathcal{U}$ than $\mathcal{C}$ compared to Dijkstra's algorithm. For a point close to the diagonal line, the M&S heuristic shows no obvious advantage for either $\mathcal{C}$ or $\mathcal{U}$ relative to Dijkstra's algorithm. Within the diagonal gray zone,

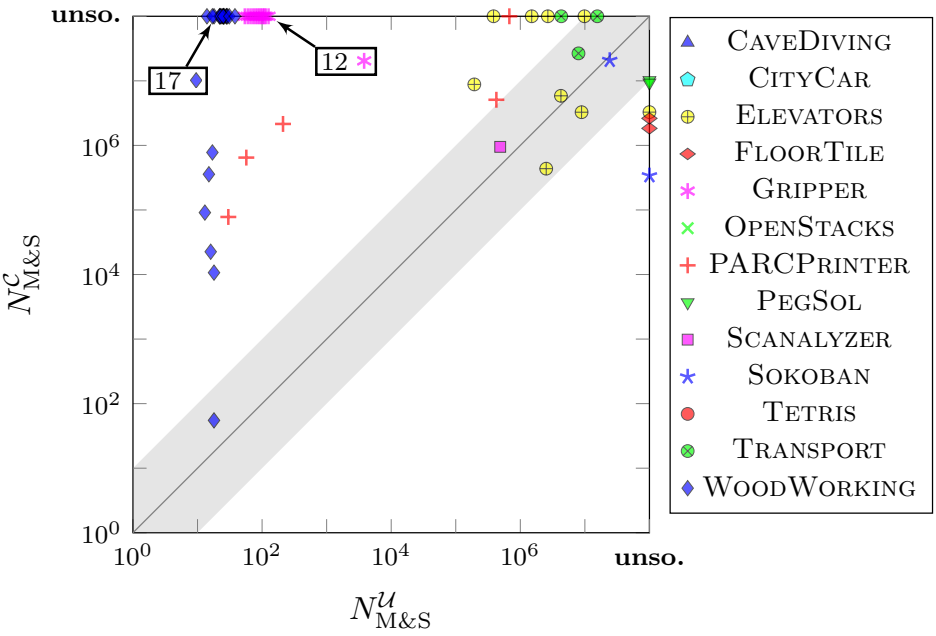


Figure 6.2: $N_{\mathcal{U}}$ vs. $N_{\mathcal{C}}$ on instances that can only be solved by A* with M&S within the time and memory limit.

the difference is within a factor of 10.

Table 6.1 gives the number of tasks in specific regions of Figure 6.1. Dijkstra's algorithm expands more nodes using $\mathcal{U}$ than $\mathcal{C}$ on more tasks (61 tasks for $R_{\text{Dij}} < 1$ vs. 36 for $R_{\text{Dij}} > 1$ in Table 6.1(a). In contrast, the M&S heuristic shows inferior performance when using $\mathcal{C}$ instead of $\mathcal{U}$ compared to Dijkstra's algorithm (41 tasks for $R_{\text{M\&S}} < R_{\text{Dij}}$ but 53 for $R_{\text{M\&S}} > R_{\text{Dij}}$ in Table 6.1(b). 19 tasks have a more than 10-fold increase in the ratio of node expansions from using $\mathcal{C}$ compared to using $\mathcal{U}$, while only 3 tasks have a more than 10-fold decrease.

Tasks Solved Only by M&S

A handful of tasks that cannot be solved with either $\mathcal{U}$ or $\mathcal{C}$ by Dijkstra's algorithm can be solved by A* with M&S heuristics. These tasks are omitted from Figure 6.1 as they do not have a meaningful value of R_{Dij}. We show this in Figure 6.2, which compares their $N^{\mathcal{U}}_{\text{M\&S}}$ (x-axis) and $N^{\mathcal{C}}_{\text{M\&S}}$ (y-axis). Thus, Figure 6.2 gives information which is complementary to Figure 6.1 about the cost effects on M&S on these tasks.

The figure shows many more tasks above the diagonal line than below. The tasks above the gray zone ($N^{\mathcal{C}}_{\text{M\&S}} > 10N^{\mathcal{U}}_{\text{M\&S}}$) or on the top **unso.** line are those solved by A* with M&S using $\mathcal{U}$ but requiring orders of magnitude more node expansions or being unsolvable within memory/time limits using $\mathcal{C}$. There are 46 such tasks in total, from 5 domains: WoodWorking, Gripper, Elevators, PARCprinter and Transport. There are only 6 tasks for which A* with M&S performs much better using $\mathcal{C}$ than using $\mathcal{U}$. These are shown on the **unso.** line on the right edge of Figure 6.2.

Overall Effects

The results in Figure 6.1, Figure 6.2 and Table 6.1 show that negative effects outweigh positive ones for action cost diversity in A* with M&S heuristics. Diverse action costs have an extreme negative effect on domains WoodWorking and Gripper, while Elevators, PARCprinter and Transport are affected negatively to a more moderate degree. We do not see $\mathcal{U}$ or $\mathcal{C}$ has a clear advantage in domains CaveDiving, Scanalyzer, Tetris[1] and OpenStacks. Non-unit cost $\mathcal{C}$ beats unit cost $\mathcal{U}$ in domains PegSol, FloorTile and Sokoban by a small margin. CityCar is a special domain because its tasks are easier to solve with $\mathcal{C}$ than with $\mathcal{U}$, both with and without M&S, but the M&S heuristics reduce the search effort more for $\mathcal{U}$ than for $\mathcal{C}$.

6.3.2 Action Cost in M&S Construction

We now discuss how action cost can affect the M&S construction process. In particular, we analyze the effects of action cost on transformation operations of M&S.

Free Pruning

The reachability in a state space is independent of action cost, so free pruning—the operation that removes dead states—is not affected by action

[1]There are 5 Tetris tasks not solved by the M&S method with either $\mathcal{C}$ or $\mathcal{U}$, but solved by Dijkstra's algorithm. This is because M&S cannot build the abstraction within the time limit for these tasks. Since this is not a cost-related issue, we omit these tasks.

cost.

Merging Strategy

MIASM's subset search is designed to find variable sets that maximize free pruning, so these variable sets are not influenced by action cost. All linear merging strategies and UMC are not affected by action costs either, because their merging decisions are determined by the causal graph and by whether a variable is a goal variable. Neither the goal relevance scoring function f_{ms}^{GR} nor the total order function f_{ms}^{TO} use information related to action costs. Thus, whether a scoring-based merging strategy is affected by action costs depends on the main scoring functions it uses. Therefore, DYN-MIASM, the scoring-based variant of MIASM, is also independent of action costs.

Changes of action costs can result in changes of the costs of paths in the transition system, so **g**-values and **h**-values of M&S abstractions can be affected. DFP depends on action costs because the label rank (Equation (2.1)) in the DFP's scoring function f_{ms}^{DFP} (Definition 29) depends on **h**-values. Similarly, SCC-DFP depends on action costs but less so because the general merging guideline is determined by strongly connected components of the causal graph. DM-HQ, the variant of DYN-MIASM with an additional tie-breaking merge scoring function f_{ms}^{HQ}, can be affected by action costs as f_{ms}^{HQ} depends on **h**-values.

Non-unit cost and unit cost induce different distributions of states regarding **h**-values (or **g**-values) in M&S abstractions. Unit cost is more likely to induce more concentrated distributions while non-unit cost with wider cost ranges is more likely to induce less concentrated distributions. In our experiments, we use the DFP merging strategy. Its main scoring function f_{ms}^{DFP} that is based on label ranking get more ties for unit cost and fewer ties for non-unit cost. As a result, it is more likely the last tie-breaking function f_{ms}^{TO} of DFP is used with unit cost than with a non-unit cost. This implies that f_{ms}^{DFP} is more likely to be more effective with a non-unit cost function than a unit cost function.

Shrinking Strategy

Greedy bisimulation shrinking only relies on action labels but not on the cost of the labels, so it will not be affected by action costs. Both **h**-preserving and **f**-preserving shrinking can produce different abstractions for different cost functions because both **g**-values and **h**-values depend on action cost. Non-greedy bisimulation shrinking becomes a **h**-preserving shrinking when the size limit is smaller than the largest bisimulation size and thus can be affected by action costs. The **h**-preserving shrinking in bisimulation shrinking tends to prioritize refinement of regions close to the goal. Less concentrated **h**-value distribution, which is more likely induced by non-unit action cost, implies a smaller refinement difference between regions close to the goal and elsewhere, than the more concentrated **h**-value distribution usually induced by unit action cost.

Exact Label Reduction

Exact label reduction facilitates bisimulation shrinking by reducing the bisimulation abstraction sizes without information loss. It is a critical, often necessary (e.g., for GRIPPER), technique for M&S to create compact bisimulation abstractions. Because exact label reduction only maps labels with the same cost to a single label to avoid information loss, action costs have direct and clear impact on exact label reduction (unlike merging and shrinking strategies): with unit costs, cost-exactness is guaranteed trivially, and with non-unit costs, exact label reductions are much more limited. As a result, with more distinct labels in a transition system, bisimulation abstractions become larger and more harmful shrinking operations are required to reduce the abstraction size.

6.4 Cost Partitioning for Diverse Action Costs

Cost partitioning is a technique often used to improve the quality of admissible heuristics [KD08; Pom+15; SH14; Yan+08]. In this section, we propose a cost

$\mathcal{C}$	c_1	c_2	c_3	$\cdots$	c_{n-1}	c_n
$\mathcal{C}_1$	Δ_1	Δ_1	Δ_1	$\cdots$	Δ_1	Δ_1
$\mathcal{C}_2$	0	Δ_2	Δ_2	$\cdots$	Δ_2	Δ_2
$\mathcal{C}_3$	0	0	Δ_3	$\cdots$	Δ_3	Δ_3
$\vdots$				$\vdots$		
$\mathcal{C}_n$	0	0	0	$\cdots$	0	Δ_n

(a)

$\mathcal{C}$	1	3	10
$\mathcal{C}_1$	1	1	1
$\mathcal{C}_2$	0	2	2
$\mathcal{C}_3$	0	0	7

(b)

Figure 6.3: (a) The cost mapping of delta cost partitioning; (b) DCP of a cost function $\mathcal{C}$ that has three different costs 1, 3 and 10.

partitioning method, called *delta cost partitioning (DCP)*, that is aimed at reducing cost diversity. As M&S benefits from unit cost in many cases, we partition action costs so that each cost function in the partitioning is simplified and resembles the unit cost function in a certain way.

Let L be the label set of a planning task, and $\mathcal{C}$ be any cost function on L. Let $c_0 = 0$ and $0 < c_1 < c_2 < \cdots < c_n$ be the $n \in \mathbb{N}$ different positive cost values to which the labels in L are mapped by $\mathcal{C}$, and let $\Delta_i = c_i - c_{i-1}$ for $i \in \{1, 2, ..., n\}$. Let $L_i = \{l \mid \mathcal{C}(l) = c_i\}$ be the set of labels that have cost c_i for $i \in \{0, 1, ..., n\}$. DCP divides the costs $c_1, c_2, \cdots, c_n$ among n *delta cost functions* $\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_n$ as follows. For $i \in \{1, 2, ..., n\}$,

$$\mathcal{C}_i(l) = \begin{cases} 0, & l \in \bigcup_{j=0}^{i-1} L_j \,, \\ \Delta_i, & l \in \bigcup_{j=i}^{n} L_j \,. \end{cases}$$

DCP maps a label to a new cost in a delta cost function depending on the label's original cost and the delta cost function. This mapping is illustrated in Figure 6.3(a), and an example is shown in Figure 6.3(b).

Since each delta cost function $\mathcal{C}_i$ maps a label to either cost 0 or Δ_i, it has at most two different costs. With the limited number of distinct costs in a delta cost function, label reduction becomes much less restricted. This could be extremely beneficial if label reduction is essential for constructing a high quality M&S heuristic for the planning task.

After the cost partitioning, we then run M&S on each delta cost function and obtain an additive set of M&S heuristics. In the rest of the chapter, we call this method DCP-MS and use the term "single M&S" to refer to the classical use of a single M&S heuristic.

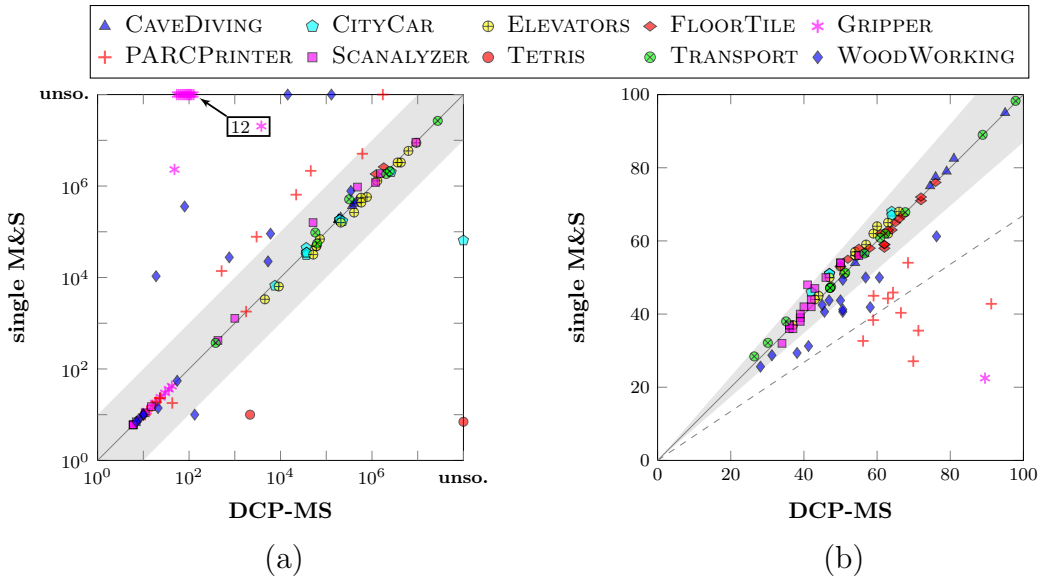


Figure 6.4: Comparing DCP-MS (x-axes) and single M&S (y-axes) on: (a) Numbers of node expansions; (b) The final f-value before time/memory limit is reached for unsolved instances by both methods, but with M&S abstractions built successfully.

6.5 Experiments

There could be complex interactions between action costs and M&S's behaviour. While cost functions with limited cost diversity seem beneficial for label reduction, we do not know how shrinking and merging strategies are affected, so the effects of delta cost partitioning on M&S need to be tested experimentally.

In addition, DCP-MS has obvious computational overhead due to multiple M&S construction, and multiple heuristic look-ups for the additive heuristic. We need to evaluate the overhead to see whether DCP-MS becomes inferior to the single M&S even if the new system produces better heuristics.

Each run of DCP-MS, which includes cost partitioning, M&S constructions and A* search, has the same time and memory limit as a single M&S run regardless of the number of delta cost functions it produces. Each M&S construction of DCP-MS also uses the same M&S configuration as the single M&S. Domains PegSol, Sokoban and OpenStacks are excluded here, since these domains use only action costs 0 and 1. Therefore, delta cost partition-

ing only reproduces the single original cost function. In addition, from the results in Section 6.3.1, these three domains are not affected negatively by the action cost diversity in their non-unit cost function, so they are not the target domains for our method.

6.5.1 Performance of Delta Cost Partitioning

Figure 6.4(a) compares the numbers of nodes expanded by single M&S (y-axis) and DCP-MS (x-axis). In the gray zone, the difference is within a factor of 10. Points above that zone are strongly favourable for DCP-MS, while for points below it is much worse.

Overall, DCP-MS outperforms single M&S on many more instances, with 24 instances above the gray zone, of which 15 are solved only by DCP-MS. These instances are from the three domains Gripper, WoodWorking and PARCprinter, which are the domains that are most affected by action cost diversity, according to the results in Section 6.3.1. In particular, the 12 Gripper instances solved only by DCP-MS are solved with minimal search effort, i.e., only states along the optimal solution are expanded. The performance of DCP-MS on the non-unit cost version of Gripper matches that of single M&S on the unit cost version of Gripper. Single M&S outperforms DCP-MS on only 4 instances. Among them, two are unsolved by DCP-MS because it times out while building its multiple abstractions.

Final f-value on Unsolved Instances

A* expands an open node with smallest f-value. This f-value, which increases as A* search progresses for consistent heuristics, indicates how much progress A* has made towards finding an optimal path to goal. For instances that are unsolved by A* with either DCP-MS or single M&S, but with M&S abstractions built successfully and A* search started, we compare their final f-values when the time/memory limit is reached.

Let f_{single} and f_{DCP} denote the largest f-value of states A* has expanded when the time/memory limit is reached, using single M&S and DCP-MS respectively. We scale the f-values for each domain so that their values are

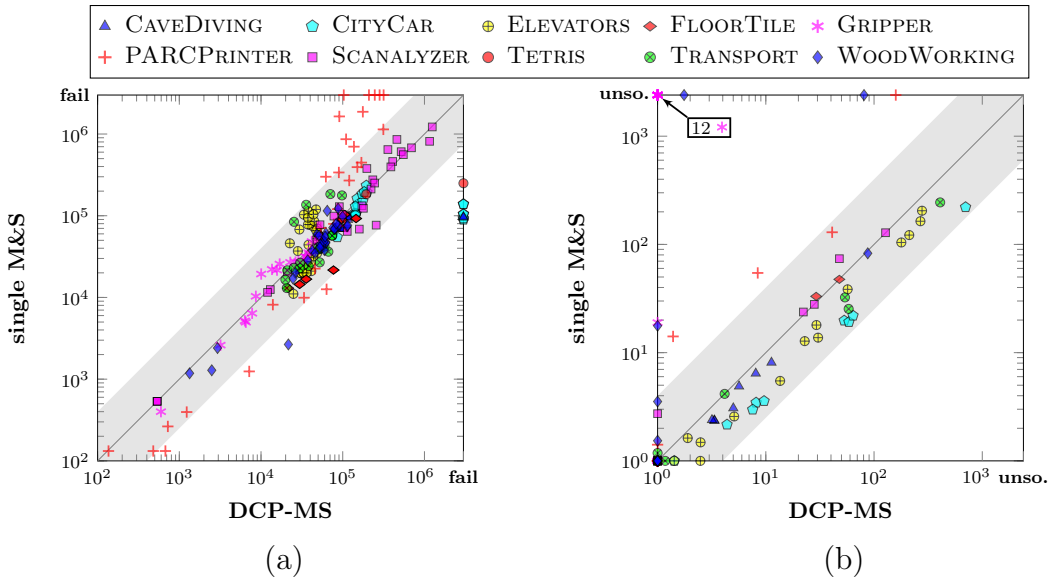


Figure 6.5: Comparing DCP-MS (x-axes) and single M&S (y-axes) on: (a) Memory (in KB) used for M&S construction; (b) Search time (in seconds) of instances with successful M&S abstraction construction. Search time less than 1 second is plotted as 1 second.

within [0,100]. A small difference between f_{single} and f_{DCP} could mean an exponential difference in numbers of node expansions, so Figure 6.4(b) compares f_{DCP} (x-axis) and f_{single} (y-axis) in a linear scale plot. The gray zone lies between the lines $f_{\text{single}} = \frac{11}{10} f_{\text{DCP}}$ and $f_{\text{single}} = \frac{10}{11} f_{\text{DCP}}$. The dashed line is defined by $f_{\text{DCP}} = \frac{3}{2} f_{\text{single}}$. In Figure 6.4(b), there is only one instance above the gray zone where f_{single} is at least 10% larger than f_{DCP}. In contrast, there are 21 instances below the gray zone where f_{DCP} is at least 10% larger than f_{single}, and 7 of them are even below the dashed line indicating $f_{\text{DCP}} > 3/2 \times f_{\text{single}}$.

6.5.2 Computational Overhead

For the IPC domains in our experiments, a delta cost partitioning produces between 2 and 27 delta cost functions depending on the planning task. DCP-MS may have heavy computational overhead due to multiple M&S construction and multiple heuristic look-ups.

M&S Construction in DCP-MS

Building a M&S abstraction can be an expensive process. However, our results show that both DCP-MS and single M&S finish the M&S construction on most of the tasks. Among the 237 planning tasks tested in our experiments, there are 9 tasks where DCP-MS fails during the M&S construction phase due to timeout but single M&S succeeds, and 5 (PARCPRINTER) tasks where single M&S fails to build the M&S abstraction due to running out of memory but DCP-MS builds multiple M&S abstractions successfully. These 5 tasks have highly diverse action costs which induces a large number of different **h**-values. The current implementation of M&S in FastDownward must keep at least one state for each **h**-value, thus single M&S runs out of memory when there are too many different **h**-values, while cost partitioning of DCP-MS reduces the cost diversity and thus the number of different **h**-values of these tasks, making it feasible to build multiple M&S abstractions.

Figure 6.5(a) compares the maximum memory allocated during M&S construction phases of DCP-MS (x-axis) vs. single M&S (y-axis). The memory usage of M&S construction for DCP-MS does not increase linearly with the number of delta cost functions. Most points are located within the gray zone, with a difference within a factor of 4, and clustered approximately evenly on both sides of the diagonal line, meaning the neither single M&S nor DCP-MS has a clear advantage in peak memory usage. On 5 PARCPRINTER tasks, single M&S runs out of memory during the M&S construction, while DCP-MS successfully builds multiple abstractions.

The key to the lower memory consumption of DCP-MS in such cases, and its good memory performance overall, is that M&S requires much less memory for the heuristic lookup tables than for abstract transition systems. Because only the heuristic lookup tables are needed in search and DCP-MS constructs one M&S abstraction at a time for each cost function, the memory for the M&S abstract transition graphs can be released and reused after each construction.

DCP-MS fails to build abstractions while single M&S succeeds for 9 tasks (right "**fail**" line) due to timeout. These 9 tasks are from the domains CAVE-

Domain	N	ratio	DCP-MS	single
CaveDiving	4.44	4.08	597.66	146.31
CityCar	3.00	3.40	502.47	147.71
Elevators	8.38	3.21	15.15	4.72
FloorTile	4.00	4.50	70.27	15.61
Gripper	4.00	2.46	15.97	6.50
PARCPrinter	11.92	1.23	125.80	102.35
Scanalyzer	2.00	1.59	147.53	92.62
Tetris	3.00	1.60	1,281.99	802.71
Transport	19.55	13.48	44.65	3.31
WoodWorking	5.80	4.04	202.39	50.12

Table 6.2: Comparing the M&S construction time (in seconds) for DCP-MS and single M&S.

diving, Citycar and Tetris where building a M&S abstraction takes a relatively long time. Table 6.2 compares the M&S construction time used by DCP-MS and single M&S for tasks where both methods build M&S abstractions successfully. In Table 6.2, column "N" shows the average number of M&S abstractions that DCP-MS has to build for each domain. The M&S construction time of DCP-MS is expected to be N times larger than that of single M&S. Column "ratio" gives the ratios of the average construction time of DCP-MS (shown in columns "DCP-MS") to that of single M&S (shown in columns "single"). The ratio of actual time of M&S construction for DCP-MS to that of single M&S is often less than N (the ratio is larger than N in only two domains). Note that in domain PARCprinter, the ratio of construction time for DCP-MS to single M&S is much less than N, which means building a M&S abstraction with the original cost function takes much more time than building one with a delta cost function of reduced cost diversity.

Search Time

To evaluate the overhead of multiple heuristic look-ups, we compare the search time of DCP-MS and single M&S on tasks with successful M&S construction. Figure 6.5(b) shows the results. DCP-MS reduces the search time for several tasks from Gripper, Woodworking and PARCprinter, because the reduction in node expansions outweighs the overhead of multiple heuristic look-ups

for the three domains, whose average numbers of heuristic look-ups per state are 4.00, 3.30 and 12.13 respectively (tasks that fail during M&S construction are excluded). There are also tasks where search with a single M&S heuristic takes less time than with the additive heuristics of DCP-MS. However, for all these tasks, the increase of search time of DCP-MS compared to single M&S is within a factor of 4 (the gray zone in Figure 6.5(b)). Among these tasks, Transport and Elevators tasks have the two highest average numbers of heuristic look-ups for each state evaluation, 13.86 and 8.25 respectively. The search time of DCP-MS on tasks from these two domains certainly does not grow linearly in the number of heuristic look-ups per state, as the number of node expansions of DCP-MS on these tasks is very close to that of single M&S (see Figure 6.4(a)).

There are 105 tasks where both methods finish building M&S abstractions but fail during search. On only 8 of these tasks DCP-MS fails due to timeout during search while single M&S fails due to the memory limit. On the other 97 tasks, both methods fail due to the memory limit.

Overall, there is only a small computational overhead in the M&S construction and search of DCP-MS.

6.6 DCP-MS for Gripper

Gripper is a domain in which label reduction is essential but the tasks suffer from cost diversity. In this section, we first prove that DCP-MS can produce perfect heuristics for Gripper with diverse costs for `pick-up` and `drop` actions through exponentially smaller M&S abstractions than single M&S. Then, we perform experiments on Gripper with diverse costs to see how DCP-MS ameliorates the negative effects of cost diversity on this domain in practice.

6.6.1 Perfect Heuristic with Polynomial Size M&S Abstractions

Without label reduction, bisimulations for Gripper are exponentially large [NHH11]. Thus, single M&S cannot produce polynomial bisimulation for Grip-

PER with diverse action costs because the different costs prohibit exact label reduction. We show that DCP-MS produces perfect heuristics with polynomial size bisimulation. The proof has two parts. We first show that the additive heuristics of the bisimulation abstractions based on any cost partitioning produces a perfect heuristic for GRIPPER for any original cost function that has no zero cost. We then show that DCP can ensure the bisimulation size is only polynomial in the representation size of a GRIPPER task, based on how bisimulation abstraction aggregates states for these tasks [NHH11].

Perfect Heuristics with DCP

A GRIPPER task contains two rooms A and B. All balls are initially in one room A and the goal is to move all balls to room B. For any GRIPPER task with m balls, there are the following variables.

- R indicates the location of the robot. Its domain contains two values `A` and `B` meaning the robot is in room A and in room B respectively.
- G_l and G_r are variables for the left and right gripper of the robot. The domain of each gripper variable has $m + 1$ values `E` and $\texttt{B}_i$ for $i \in \{1, 2, ..., m\}$. `E` indicates that the gripper is empty and available for picking up a ball, and $\texttt{B}_i$ indicates that the gripper holds ball i.
- B_i is the variable for ball i and can have 3 values `H`, `A` and `B` which means the ball is being held by a gripper, is in room A and is in room B respectively.

Let P be a path and let $s \xrightarrow{a} P$ denote the concatenation of s and action a at the beginning of P. $P(s)$ denotes a path starting with s. For a state s, $s(v)$ denotes the value of variable v of s.

Definition 45 (Necessary Solution Path for GRIPPER)**.** The *necessary* solution path $P(s)$ for a GRIPPER state s are defined as follows.

$$P(s) = \begin{cases} (s) & \text{if } s \text{ is a goal state} \\ s \xrightarrow{a} P(s') & \text{otherwise} \end{cases}$$

where $s \xrightarrow{a} s'$ is a transition and a is an action defined as follows.

- if $s(R) = \texttt{B}$
  - if $s(G_l) = \texttt{E}$ and $s(G_r) = \texttt{E}$, then a is the `move` action that changes the location of the robot from `B` to `A`.
  - if $s(G_l) = \texttt{B}_i$ (or $s(G_r) = \texttt{B}_i$) then a is the `drop` action that drops ball i from the left (or right) gripper
- if $s(R) = \texttt{A}$
  - if $s(G_l) \neq \texttt{E}$ and $s(G_r) \neq \texttt{E}$, or $s(B_i) \neq \texttt{A}$ for all $i \in \{1, 2, ..., n\}$, then a is the `move` action that changes the location of the robot from `A` to `B`.
  - if $s(B_i) = \texttt{A}$ for some $i \in \{1, 2, ..., n\}$ and $s(G_l) = \texttt{E}$ (or $s(G_r) = \texttt{E}$), then a is the `pick-up` action that pick up the ball i by the left (or right) gripper.

In the necessary solution path, each ball in room A is picked up and dropped exactly once, and the robot carries two balls whenever possible and moves from room A to room B only when carrying balls and moves from room B to room A only with empty grippers and only when there are still balls in room A. We have the following observation.

Proposition 2. Let Θ be the transition system for a GRIPPER task and let s be a state in Θ. The necessary solution path $P(s)$ for s is optimal for any cost function $\mathcal{C}$, i.e., $\mathcal{C}(P(s)) = \mathbf{h}^{\mathcal{C}}_{\Theta}(s)$.

Proof. Let $m_{\texttt{A}}$ be the number of balls in room A in s. In any path from s to a goal state, each ball in room A is picked up and dropped at least once, so there are at least $m_{\texttt{A}}$ `pick-up` actions and at least $m_{\texttt{A}}$ `drop` actions. The robot has to move from room A to room B at least $\lceil m_{\texttt{A}}/2 \rceil$ times. If the robot is currently in room B, it has to move from room B to room A $\lceil m_{\texttt{A}}/2 \rceil$ times. If the robot is currently in room A, it has to move from room B to room A $\lceil m_{\texttt{A}}/2 \rceil - 1$ times. The number of actions in the necessary path of s reaches these lower-bounds, so it is optimal for any action cost. □

Proposition 3. Let Θ be the transition system for a GRIPPER task and let s be a state in Θ. Let $\mathcal{C}$ be a cost function for Θ. Let $n \in \mathbb{N}$ and $\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_n$ be any cost partitioning for $\mathcal{C}$ Then $\mathbf{h}^{\mathcal{C}}_{\Theta}(s) = \sum_{i=1}^{n} \mathbf{h}^{\mathcal{C}_i}_{\Theta}(s)$.

Proof. By Proposition 2, $\mathbf{h}^{\mathcal{C}}_{\Theta}(s) = \mathcal{C}(P(s))$ and $\mathbf{h}^{\mathcal{C}_i}_{\Theta}(s) = \mathcal{C}_i(P(s))$ for any $i \in \{1, 2, ..., n\}$. Thus $\mathbf{h}^{\mathcal{C}}_{\Theta}(s) = \mathcal{C}(P(s)) = \sum_{i=1}^{n} \mathcal{C}_i(P(s)) = \sum_{i=1}^{n} \mathbf{h}^{\mathcal{C}_i}_{\Theta}(s)$. □

Since M&S with bisimulation shrinking can produce heuristic $h_i(s) = \mathbf{h}^{\mathcal{C}_i}_{\Theta}(s)$ for all $\mathcal{C}_i$ [NHH11], we can get the perfect heuristic for the original cost function.

Bisimulation with Polynomial Size

Now we show that when greedy bisimulation shrinking and exact label reduction are used, there exists a merging order such that the size of the maximum intermediate abstraction is polynomial in the number of balls and exponential in the number of different costs for `pick-up` and `drop` actions. The following known fact is used for our proof.

Proposition 4 ([NHH11])**.** For any GRIPPER task, the bisimulation shrinking aggregates the states that have the same number of balls in each room and the same condition for the robot (values for R, G_l and G_r) which results in a bisimulation abstraction of polynomial size in the number of balls.

By Proposition 4, for any variable set V of a GRIPPER task, the bisimulation of π_V can be produced through a merge-and-shrink process in which the maximum intermediate abstraction size is polynomial in the number of balls considered in V.

Proposition 5. Let Π be a GRIPPER task of m balls and n different costs for `pick-up` and `drop` actions. We can obtain a bisimulation of $\Theta(\Pi)$ through a merge-and-shrink process in which the maximum intermediate abstraction size is polynomial in m and exponential in n.

Proof. Let $V_0, V_1, V_2, ..., V_n$ be a partitioning of $\mathcal{V}$ in which $V_0 = \{R, G_l, G_r\}$ and V_i for $i \in \{1, 2, ..., n\}$ is a set of m_i ball variables for which their `pickup`

costs are the same and `drop` costs are also the same. We use greedy bisimulation shrinking and exact label reduction. We merge variables in each V_i first, then merge the $n+1$ produced M&S abstractions of V_i for $i \in \{0, 1, ..., n\}$ which are bisimulations of π_{V_i}. The size of the projection π_{V_0} is $2 * (m+1) * (m+1)$, a polynomial in m. For any $i \in \{1, 2, ..., n\}$, the bisimulation of π_{V_i} can be produced through a merge-and-shrink process in which the maximum intermediate abstraction size is polynomial in m_i and thus polynomial in m. Since we have $n + 1$ bisimulations, all with sizes polynomial in m, the size of their product is also polynomial in m but exponential in n. □

n can be as large as m, so the maximum intermediate abstraction size can be exponential in m if we use the above strategy. However, DCP produces n delta cost functions with at most 2 different costs each, so for each delta cost function, we can obtain the bisimulation of the transition system of a Gripper task with size polynomial in m (in particular, $\mathcal{O}(m^6)$) and thus the sum of the sizes of these bisimulations is also polynomial in m.

6.6.2 Experiment Results

The non-unit cost function in the modified Gripper domain from Section 6.3.1 has $n_c = 4$ different costs. Our earlier experiments showed that single M&S has trouble dealing with such diversity, while DCP-MS works very well. We now further test how well DCP-MS performs as the cost diversity increases.

The `pick-up/drop` action costs for ball b are (b mod n_c)+1. The parameter n_c gives an upper bound on the number of different costs for `pick-up/drop` actions. The number of balls in IPC Gripper tasks ranges from 4 to 42. We compare the number of tasks solved by single M&S and DCP-MS as n_c increases. In addition to the default M&S configuration, we also test the variant that does not limit abstraction size, which has the best performance on the unit cost Gripper.

Table 6.3 shows the coverage of DCP-MS and single M&S for different n_c. For the last row of n_c = "#balls", each ball has a different `pick-up/drop` cost. There are 20 Gripper tasks. The columns under "50K" and "no limit"

n_c	50K		no limit	
	DCP-MS	single	DCP-MS	single
1	19	19 (19)	20	20
2	19	12 (12)	20	20
4	19	7 (6)	20	20
8	16	7 (4)	20	11
16	14	7 (4)	20	7
#balls	13	7 (4)	18	7

Table 6.3: Coverages of GRIPPER with increasing cost diversity for DCP-MS and single M&S using two different M&S configurations.

show the coverage of DCP-MS and single M&S using M&S configurations with a size limit of 50,000 states (non-greedy bisimulation shrinking) and without any size limit (greedy bisimulation shrinking) respectively. For single M&S with 50K size limit, the number of tasks solved with the minimal search effort is shown in brackets. For all other methods, all solved tasks are solved with minimal search effort. With either M&S configuration, the coverage of single M&S decreases faster than DCP-MS as n_c increases. When every ball has a different cost, DCP-MS with no size limit can solve 18 tasks, with minimal search effort, while single M&S solves only the 7 smallest tasks.

6.7 Conclusions

We have studied the effects of diverse action costs on M&S. We have shown that action cost diversity can affect M&S negatively and have proposed a new method, DCP-MS, additive M&S with delta cost partitioning, to address this issue. The experiments show that DCP-MS produces much more informative heuristics than the standard M&S on several IPC domains, especially on those affected negatively by action cost diversity. Our case study on GRIPPER demonstrates the power of DCP-MS for this IPC domain in theory and in practice.

Chapter 7

Conclusions

In this chapter, we summarize the contributions of this thesis. We also discuss the limitations and some future directions of this thesis research.

7.1 Contributions

In this thesis, we studied how to effectively create and use merge-and-shrink heuristics for cost-optimal planning. In the following, we describe the contributions of this thesis.

In Chapter 3, we proposed three non-linear merging strategies `UMC`, `MIASM` and `DM-HQ`. Each of the proposed merging strategies explores a unique idea for constructing more informative M&S heuristics. `UMC` is the first non-linear merging strategies proposed in planning community, which is based on the idea that more closely related variable should be merged earlier. `MIASM` is a sophisticated merging strategies that improves the M&S heuristic by trying to maximize the amount of free-pruning to reduce harmful shrinking during the construction. This idea has been proved to be an important criterion for a successful merging strategy. Unlike `MIASM` and `UMC` which use an indirect criteria for constructing better M&S heuristics, `DM-HQ` uses a scoring function that seeks high-quality M&S heuristics directly through an estimation of the heuristic quality. Experiments show that `DM-HQ` currently outperforms other M&S method when used alone.

In Chapter 4, we proposed an extreme efficient M&S method called MS-lite, and explored the potential of combining two complementary M&S heuristics.

Our experiments show that MS-lite complements other M&S methods very well, and the complementarity between MS-lite and `DM-HQ` is the strongest. The combination of the two outperforms the state-of-the-art M&S method dramatically by solving 75 more tasks on an IPC benchmark set of 1499 tasks, making it the current best performing M&S method.

In Chapter 5, we showed that non-unit action costs can have both positive and negative effects on heuristic search planning and provide the foundation for our studies on how action costs affect M&S heuristic in Chapter 6. Prior to our study, planning with non-unit action costs was considered more difficult than with the unit cost. Our research provided experimental evidence that diverse action costs can make planning easier. The main contribution of this study is a No Free Lunch theorem showing that, when heuristics and TIE states are not considered, the overall negative effect of cost diversity are counterbalanced by the overall positive effects. We also showed that it is advantageous to have a strongly concentrated distribution of solution costs when taking the goal-preference tie-breaking into consideration. Unit costs often give rise to a more concentrated distribution than non-unit costs, but we also provided an example domain in which the opposite is the case.

In Chapter 6, we first showed that there are the negative effects of diverse action costs on M&S heuristics on IPC domains with diverse action costs. We then proposed an additive M&S method DCP-MS to handle such negative effects. The study in this chapter is the first to analyze effects of action cost diversity on a domain-independent heuristic. Our experiments demonstrate that DCP-MS produces more informative heuristics than a single M&S for the impacted IPC domains. We also proved that for IPC domain Gripper with non-unit costs, DCP-MS produces perfect heuristics with polynomial size M&S abstractions.

7.2 Limitations

We now discuss some of the limitations of the current research.

Free Pruning Dependency of `MIASM`. The main limitation of `MIASM` is that it works only when there are states that can be removed by free pruning. If a state space has no dead states, `MIASM` cannot provide any suggestion for merging.

Construction of Merging Products for Evaluations. `MIASM` and many scoring functions require construction of synchronized products of potential merges. These synchronized products only exist for evaluating the merging decisions and may require a lot of time and memory for their construction.

Heuristic Quality Guided Greedy Search. A scoring-based merging strategy is a greedy process. Heuristic quality scoring functions f_{ms}^{HQ} seem to suffer from the greediness of this process. This is the reason why using f_{ms}^{HQ} alone does not provide an outstanding performance even though its score measures the heuristic quality directly. In `DM-HQ`, f_{ms}^{HQ} is used as a secondary function to avoid this problem.

Heuristic Evaluation on Initial States. In f_{ms}^{HQ} functions, we use only the initial states to evaluate heuristics. This choice may make a merging strategy concentrate the refinement on the region around the solution paths but leave other regions under refined. This could result in poor heuristics with smaller heuristic values on states further away from the solution paths and mislead the search into the regions consist of these states.

The Limitation of the NFL Theorem. The theoretical setting of the NFL theorem is limited. It does not extend to the cases where heuristics are used or multiple goal states are considered, which is standard for heuristic search planning. It is unclear whether there exist similar theorems for a more relaxed setting.

Complexity of DCP. The number of cost functions of DCP is the number of different action costs of the original cost function. Although experiments

show that DCP has small overhead for most non-unit cost IPC domains, there can be domains with too many different costs for DCP to handle.

7.3 Future Work

In this section, we discuss interesting directions for future research. Some of the directions are directly inspired by the limitations mentioned above.

Generalizing `MIASM`. The subset search of `MIASM` is a global search for good merging decisions. It is more expensive than the local search of scoring-based merging strategy, but reveals long-term effects of merging decisions. One potential direction for improving its merging strategy is to generalize the subset search of `MIASM` by using other optimization criteria such as heuristic quality estimation in f_{ms}^{HQ}.

Synchronized Search on Factor Systems. Many successful merging strategies need to temporarily construct the product transition systems of potential merges to evaluate the merging decisions, e.g., finding the dead states for `MIASM`, or computing the heuristic value for f_{ms}^{HQ}. They construct a temporary product so that they can perform searches (Dijkstra's algorithm that does not terminate with goal states) on the transition system to get information for the merging decision evaluations. However, in theory, search can still be performed without doing the actual merging operation. This is possible with a *synchronized* search that can verify conditions in multiple factor systems for applicability of actions and goal checking, and apply actions in these factor systems simultaneously. Such a synchronized search is not limited to only two factor systems of a merging operation but can be applied to multiple factor systems, i.e., for evaluating merging more than two systems. Using this kind of search can reduce the time and memory for merging decision evaluations.

Deeper Search of Scoring-Based Merging. The scoring-based merging strategies only evaluate one merging step ahead. We could also do the evaluations on more than one merging step. This increases the complexity dramat-

ically, so some pruning is probably needed. One way to reduce the evaluation complexity is to do the lookahead of the next step only with the few optimal or near-optimal candidates (according to a scoring function) of the current step. Again, this is a research direction that would explore improvements of merging strategies by looking more into the long-term effects of merging decisions.

Fundamental Causes for Smaller M&S Abstractions Producing Better Heuristics. Our experiments show that there are problems where smaller **h**-preserving abstraction can product better heuristics. When it comes to M&S abstraction it is not simply that "the larger the better". Although we constructed an example to demonstrate how this counter-intuitive situation can happen, we do not know if there is a fundamental cause for this. Investigating this phenomenon would be an interesting research direction for improving our understanding of M&S and may lead to better shrinking strategies.

DCP for Other Heuristics. DCP improves M&S heuristic by reducing the number of *distinct* action costs, which is helpful for the exact label reduction of M&S. It is possible other domain-independent heuristics are also negatively affected by the distinctiveness of action costs. It is worth further research to study whether DCP can help other planning heuristics to handle diverse action costs better.

References

[AB15] Meysam Aghighi and Christer Bäckström. "Cost-Optimal and Net-Benefit Planning - A Parameterised Complexity View." In: *Proceedings of the 24th International Joint Conference on Artificial Intelligence*. 2015, pp. 1487–1493.

[AB16] Meysam Aghighi and Christer Bäckström. "A Multi-Parameter Complexity Analysis of Cost-Optimal and Net-Benefit Planning." In: *Proceedings of the 26th International Conference on Automated Planning and Scheduling*. 2016, pp. 2–10.

[AH97] Esther M. Arkin and Refael Hassin. "On Local Search for Weighted k-Set Packing." In: *Algorithms - ESA '97, 5th European Symposium on Algorithms*. 1997, pp. 13–22.

[BN95] Christer Bäckström and Bernhard Nebel. "Complexity Results for SAS+ Planning." In: *Computational Intelligence* 11 (1995), pp. 625–656.

[Ben+10] J. Benton, Kartik Talamadupula, Patrick Eyerich, Robert Mattmüller, and Subbarao Kambhampati. "G-Value Plateaus: A Challenge for Planning." In: *Proceedings of the 20th International Conference on Automated Planning and Scheduling*. 2010, pp. 259–262.

[BG01] Blai Bonet and Hector Geffner. "Planning as Heuristic Search." In: *Artificial Intelligence* 129.1-2 (2001), pp. 5–33.

[CH01] Barun Chandra and Magnús Halldórsson. "Greedy Local Improvement and Weighted Set Packing Approximation." In: *Journal of Algorithms* 39.2 (2001), pp. 223–240. ISSN: 0196-6774.

[CBK10] William Cushing, J. Benton, and Subbarao Kambhampati. "Cost Based Search Considered Harmful." In: *Proceedings of the 3rd Annual Symposium on Combinatorial Search*. 2010, pp. 140–141.

[CBK11] William Cushing, J. Benton, and Subbarao Kambhampati. "Cost Based Satisficing Search Considered Harmful." In: *CoRR* abs/1103.3687 (2011).

[Dav95] William G. Macready David H. Wolpert. *No Free Lunch Theorems for Search*. Tech. rep. Santa Fe Institute, 1995.

[Dav97] William G. Macready David H. Wolpert. "No Free Lunch Theorems for Optimization." In: *IEEE Transactions on Evolutionary Computation* 1.1 (1997), pp. 67–82.

[DP85] Rina Dechter and Judea Pearl. "Generalized Best-First Search Strategies and the Optimality of A*." In: *Journal of the ACM* 32.3 (1985), pp. 505–536.

[Dij59] Edsger W. Dijkstra. "A Note on Two Problems in Connexion with Graphs." In: *Numerische Mathematik* 1.1 (1959), pp. 269–271.

[DFP06] Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. "Directed Model Checking with Distance-Preserving Abstractions." In: *Proceedings of Model Checking Software, 13th International SPIN Workshop*. Vol. 3925. 2006, pp. 19–34.

[DFP09] Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. "Directed Model Checking with Distance-Preserving Abstractions." In: *STTT, International Journal on Software Tools for Technology Transfer* (2009), pp. 27–37.

[Ede01] Stefan Edelkamp. "Planning with Pattern Databases." In: *Proceedings of the 6th European Conference on Planning*. 2001, pp. 84–90.

[Ede06] Stefan Edelkamp. "Automated Creation of Pattern Database Search Heuristics." In: *Model Checking and Artificial Intelligence*. 2006, pp. 35–50.

[ES12] Stefan Edelkamp and Stefan Schrödl. *Heuristic Search - Theory and Applications*. Academic Press, 2012.

[FH15] Gaojian Fan and Robert C. Holte. "The Spurious Path Problem in Abstraction." In: *Proceedings of the 8th Annual Symposium on Combinatorial Search*. 2015, pp. 18–27.

[FHM18] Gaojian Fan, Robert Holte, and Martin Müller. "MS-Lite: A Lightweight, Complementary Merge-and-Shrink Method." In: *Proceedings of the 28th International Conference on Automated Planning and Scheduling*. 2018, pp. 74–82.

[FMH14] Gaojian Fan, Martin Müller, and Robert Holte. "Non-Linear Merging Strategies for Merge-and-Shrink Based on Variable Interactions." In: *Proceedings of the 7th Annual Symposium on Combinatorial Search*. 2014, pp. 53–61.

[FMH17a] Gaojian Fan, Martin Müller, and Robert Holte. "Additive Merge-and-Shrink Heuristics for Diverse Action Costs." In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. 2017, pp. 4287–4293.

[FMH17b] Gaojian Fan, Martin Müller, and Robert Holte. "The Two-Edged Nature of Diverse Action Costs." In: *Proceedings of the 27th International Conference on Automated Planning and Scheduling*. To appear. 2017.

[Fra+17] Santiago Franco, Álvaro Torralba, Levi H. S. Lelis, and Mike Barley. "On Creating Complementary Pattern Databases." In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. 2017, pp. 4302–4309.

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[HNR68] 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* (1968), pp. 100–107.

[Has+07] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. "Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning." In: *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*. 2007, pp. 1007–1012.

[HG00] Patrik Haslum and Hector Geffner. "Admissible Heuristics for Optimal Planning." In: *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems*. 2000, pp. 140–149.

[Hel04] Malte Helmert. "A Planning Heuristic Based on Causal Graph Analysis." In: *Proceedings of the 14th International Conference on Automated Planning and Scheduling*. 2004, pp. 161–170.

[Hel06] Malte Helmert. "The Fast Downward Planning System." In: *Journal of Artificial Intelligence Research* 26 (2006), pp. 191–246.

[HD09] Malte Helmert and Carmel Domshlak. "Landmarks, Critical Paths and Abstractions: What's the Difference Anyway?" In: *Proceedings of the 19th International Conference on Automated Planning and Scheduling*. 2009.

[HHH07] Malte Helmert, Patrik Haslum, and Jörg Hoffmann. "Flexible Abstraction Heuristics for Optimal Sequential Planning." In: *Proceedings of the 17th International Conference on Automated Planning and Scheduling*. 2007, pp. 176–183.

[Hel+14] Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. "Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces." In: *Journal of the ACM* (2014), 16:1–16:63.

[HN01] Jörg Hoffmann and Bernhard Nebel. "The FF Planning System: Fast Plan Generation Through Heuristic Search." In: *Journal of Artificial Intelligence Research* (2001), pp. 253–302.

[Hol10] Robert C. Holte. "Common Misconceptions Concerning Heuristic Search." In: *Proceedings of the 3rd Annual Symposium on Combinatorial Search*. 2010, pp. 46–51.

[Ike+94] T. Ikeda, Min-Yao Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. "A Fast Algorithm for Finding Better Routes by AI Search Techniques." In: *Proceedings of Vehicle Navigation and Information Systems Conference*. 1994, pp. 291–296.

[KD08] Michael Katz and Carmel Domshlak. "Optimal Additive Composition of Abstraction-based Admissible Heuristics." In: *Proceedings of the 18th International Conference on Automated Planning and Scheduling*. 2008, pp. 174–181.

[KHH12] Michael Katz, Jörg Hoffmann, and Malte Helmert. "How to Relax a Bisimulation?" In: *Proceedings of the 22nd International Conference on Automated Planning and Scheduling*. 2012.

[Kno94] Craig A. Knoblock. "Automatically Generating Abstractions for Planning." In: *Artificial Intelligence* 68.2 (1994), pp. 243–302.

[Mar77] Alberto Martelli. "On the Complexity of Admissible Search Algorithms." In: *Artificial Intelligence* 8.1 (1977), pp. 1–13.

[Mil90] Robin Milner. "Operational and Algebraic Semantics of Concurrent Processes." In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*. 1990, pp. 1201–1242.

[Nak13] Hootan Nakhost. "Random Walk Planning: Theory, Practice, and Application." PhD thesis. University of Alberta, 2013.

[NHH11] Raz Nissim, Jörg Hoffmann, and Malte Helmert. "Computing Perfect Heuristics in Polynomial Time: On Bisimulation and Merge-and-Shrink Abstraction in Optimal Planning." In: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*. 2011, pp. 1983–1990.

[Pea84] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984. ISBN: 978-0-201-05594-8.

[Pom+15] Florian Pommerening, Malte Helmert, Gabriele Röger, and Jendrik Seipp. "From Non-negative to General Operator Cost Partitioning." In: *Proceedings of the 29th AAAI Conference on Artificial Intelligence*. AAAI'15. 2015.

[RW10] Silvia Richter and Matthias Westphal. "The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks." In: *Journal of Artificial Intelligence Research* 39 (2010), pp. 127–177.

[RWH11] Silvia Richter, Matthias Westphal, and Malte Helmert. "LAMA 2008 and 2011." In: *International Planning Competition*. 2011, pp. 117–124.

[SH13] Jendrik Seipp and Malte Helmert. "Counterexample-Guided Cartesian Abstraction Refinement." In: *Proceedings of the 23rd International Conference on Automated Planning and Scheduling*. 2013.

[SH14] Jendrik Seipp and Malte Helmert. "Diverse and Additive Cartesian Abstraction Heuristics." In: *Proceedings of the 24th International Conference on Automated Planning and Scheduling*. 2014, pp. 289–297.

[Sie17] Silvan Sievers. "Merge-and-shrink Abstractions for Classical Planning: Theory, Strategies, and Implementation." PhD thesis. University of Basel, 2017.

[Sie18] Silvan Sievers. "Merge-and-Shrink Heuristics for Classical Planning: Efficient Implementation and Partial Abstractions." In: *Proceedings of the 7th International Symposium on Combinatorial Search*. 2018, p. 99.

[SWH14] Silvan Sievers, Martin Wehrle, and Malte Helmert. "Generalized Label Reduction for Merge-and-Shrink Heuristics." In: *Proceedings of the 28th AAAI Conference on Artificial Intelligence*. 2014, pp. 2358–2366.

[SWH16] Silvan Sievers, Martin Wehrle, and Malte Helmert. "An Analysis of Merge Strategies for Merge-and-Shrink Heuristics." In: *Proceedings of the 26th International Conference on Automated Planning and Scheduling*. 2016, pp. 294–298.

[SW97] Mechthild Stoer and Frank Wagner. "A simple min-cut algorithm." In: *Jouranal of the ACM* 44.4 (1997), pp. 585–591.

[TBH12] Jordan T. Thayer, J. Benton, and Malte Helmert. "Better Parameter-Free Anytime Search by Minimizing Time Between Solutions." In: *Proceedings of the 5th Annual Symposium on Combinatorial Search*. 2012, pp. 120–128.

[TR09] Jordan T. Thayer and Wheeler Ruml. "Using Distance Estimates in Heuristic Search." In: *Proceedings of the 19th International Conference on Automated Planning and Scheduling*. 2009, pp. 382–385.

[TR11] Jordan T. Thayer and Wheeler Ruml. "Bounded Suboptimal Search: A Direct Approach Using Inadmissible Estimates." In: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*. 2011, pp. 674–679.

[Wil14] Christopher Wilt. "Steps Towards a Science of Heuristic Search." PhD thesis. University of New Hampshire, 2014.

[WR11] Christopher Wilt and Wheeler Ruml. "Cost-Based Heuristic Search Is Sensitive to the Ratio of Operator Costs." In: *Proceedings of the 4th Annual Symposium on Combinatorial Search*. 2011, pp. 172–179.

[WR14] Christopher Wilt and Wheeler Ruml. "Speedy Versus Greedy Search." In: *Proceedings of the 7th Annual Symposium on Combinatorial Search*. 2014, pp. 184–192.

[Yan+08] Fan Yang, Joseph C. Culberson, Robert Holte, Uzi Zahavi, and Ariel Felner. "A General Theory of Additive State Space Abstractions." In: *Journal of Artificial Intelligence Research* 32 (2008), pp. 631–662.