Adaptive Search Techniques in AI Planning and Heuristic Search

Maximilian Fickert

A dissertation submitted towards the degree
Doctor of Natural Sciences (Dr. rer. nat.)
of the Faculty of Mathematics and Computer Science
of Saarland University

Saarbrücken, 2022

UNIVERSITÄT
DES
SAARLANDES

Day of Colloquium	February 11, 2022
Dean of the Faculty	Prof. Dr. Thomas Schuster
Chair of the Committee	Prof. Dr. Jan Reineke
Reviewers	Prof. Dr. Jörg Hoffmann
	Prof. Dr. Antonio Krüger
	Prof. Wheeler Ruml, PhD
Academic Assistant	Daniel Fišer, PhD

Abstract

State-space search is a common approach to solve problems appearing in artificial intelligence and other subfields of computer science. In such problems, an agent must find a sequence of actions leading from an initial state to a goal state. However, the state spaces of practical applications are often too large to explore exhaustively. Hence, heuristic functions that estimate the distance to a goal state (such as straight-line distance for navigation tasks) are used to guide the search more effectively.

Heuristic search is typically viewed as a static process. The heuristic function is assumed to be unchanged throughout the search, and its resulting values are directly used for guidance without applying any further reasoning to them. Yet critical aspects of the task may only be discovered during the search, e.g., regions of the state space where the heuristic does not yield reliable values.

Our work here aims to make this process more dynamic, allowing the search to adapt to such observations. One form of adaptation that we consider is online refinement of the heuristic function. We design search algorithms that detect weaknesses in the heuristic, and address them with targeted refinement operations. If the heuristic converges to perfect estimates, this results in a secondary method of progress, causing search algorithms that are otherwise incomplete to eventually find a solution.

We also consider settings that inherently require adaptation: In online replanning, a plan that is being executed must be amended for changes in the environment. Similarly, in real-time search, an agent must act under strict time constraints with limited information.

The search algorithms we introduce in this work share a common pattern of online adaptation, allowing them to effectively react to challenges encountered during the search. We evaluate our contributions on a wide range of standard benchmarks. Our results show that the flexibility of these algorithms makes them more robust than traditional approaches, and they often yield substantial improvements over current state-of-the-art planners.

Zusammenfassung

Die Zustandsraumsuche ist ein oft verwendeter Ansatz um verschiedene Probleme zu lösen, die in der Künstlichen Intelligenz und anderen Bereichen der Informatik auftreten. Dabei muss ein Akteur eine Folge von Aktionen finden, die einen Pfad von einem Startzustand zu einem Zielzustand bilden. Die Zustandsräume von praktischen Anwendungen sind häufig zu groß um sie vollständig zu durchsuchen. Aus diesem Grund leitet man die Suche mit Heuristiken, die die Distanz zu einem Zielzustand abschätzen; zum Beispiel lässt sich die Luftliniendistanz als Heuristik für Navigationsprobleme einsetzen.

Heuristische Suche wird typischerweise als statischer Prozess angesehen. Man nimmt an, dass die Heuristik während der Suche eine unveränderte Funktion ist, und die resultierenden Werte werden direkt zur Leitung der Suche benutzt ohne weitere Logik darauf anzuwenden. Jedoch könnten kritische Aspekte des Problems erst im Laufe der Suche erkannt werden, wie zum Beispiel Bereiche des Zustandsraums in denen die Heuristik keine verlässlichen Abschätzungen liefert.

In dieser Arbeit wird der Suchprozess dynamischer gestaltet und der Suche ermöglicht sich solchen Beobachtungen anzupassen. Eine Art dieser Anpassung ist die Onlineverbesserung der Heuristik. Es werden Suchalgorithmen entwickelt, die Schwächen in der Heuristik erkennen und mit gezielten Verbesserungsoperationen beheben. Wenn die Heuristik zu perfekten Werten konvergiert ergibt sich daraus eine zusätzliche Form von Fortschritt, wodurch auch Suchalgorithmen, die sonst unvollständig sind, garantiert irgendwann eine Lösung finden werden.

Es werden auch Szenarien betrachtet, die schon von sich aus Anpassung erfordern: In der Onlineumplanung muss ein Plan, der gerade ausgeführt wird, auf Änderungen in der Umgebung angepasst werden. Ähnlich dazu muss sich ein Akteur in der Echtzeitsuche unter strengen Zeitauflagen und mit eingeschränkten Informationen bewegen.

Die Suchalgorithmen, die in dieser Arbeit eingeführt werden, folgen einem gemeinsamen Muster von Onlineanpassung, was ihnen ermöglicht effektiv auf Herausforderungen zu reagieren die im Verlauf der Suche aufkommen. Diese Ansätze werden auf einer breiten Reihe von Benchmarks ausgewertet. Die Ergebnisse zeigen, dass die Flexibilität dieser Algorithmen zu erhöhter Zuverlässigkeit im Vergleich zu traditionellen Ansätzen führt, und es werden oft deutliche Verbesserungen gegenüber modernen Planungssystemen erzielt.

Acknowledgements

First and foremost, I would like to express my gratitude towards my advisor Jörg Hoffmann for giving me the opportunity to write this thesis. This work would not have been possible without his support and mentorship throughout these years, and his feedback has been invaluable to guide me in the right direction.

I would like to thank Wheeler Ruml for his inspiration and advice in all these fruitful discussions. Working with him has been a great pleasure, and his unparalleled enthusiasm has always kept me going.

Thanks to my co-authors, most of all Tianyi and Ivan, for the constructive collaborations that helped make this thesis possible.

Thanks to Anna, Álvaro, Daniel, Ivan, and Tina for joining the proofreading effort for this thesis and greatly improving the quality of this work.

I would also like to thank everyone at the FAI group, I have had a great time working here. I will definitely miss the post-lunch table-tennis sessions!

Finally, thanks to Tina and my family for their continuous support and encouragement during this time.

Contents

1 Introduction 1
1.1 Contributions . . . 3
1.2 Experiments Setup . . . 5
1.3 Publications . . . 6

2 Background 9
2.1 Transition Systems . . . 9
2.2 Heuristic Search . . . 10
2.3 Classical AI Planning . . . 11
2.3.1 Induced Transition System . . . 12
2.3.2 Planning Heuristics . . . 13

I Adaptive Partial Delete Relaxation 15

3 Partial Delete Relaxation 17
3.1 Delete Relaxation . . . 17
3.2 Partial Delete Relaxation through Explicit Conjunctions . . . 19
3.2.1 h^{CFF} in Practice . . . 19
3.2.2 The Refinement Operation of h^{CFF} . . . 21
3.3 Red-Black Planning . . . 22
3.3.1 Tractable Fragment (ACI) . . . 24
3.3.2 Red-Black State-Space Search (RBS) . . . 25

4 Online Relaxation Refinement for Satisficing Planning 29
4.1 Background: Techniques We Build On . . . 32
4.1.1 Novelty Pruning . . . 33
4.1.2 Subgoal Counting . . . 33
4.2 Experiments Setup . . . 34
4.3 Converging Heuristic Functions . . . 35
4.4 Online-Refinement Hill-Climbing . . . 36
4.4.1 Episode-EHC . . . 37

4.4.2 Refinement-HC . . . 38
4.4.3 Completeness . . . 42
4.4.4 Experiments . . . 43
4.5 Refinement-HC with Novelty Pruning . . . 45
4.5.1 Replacing the Depth Bound with Novelty Pruning . . . 45
4.5.2 Novelty Pruning over Conjunctions . . . 46
4.5.3 Experiments . . . 46
4.6 Refinement-HC with Relaxed Subgoal Counting . . . 50
4.6.1 Method . . . 50
4.6.2 Experiments . . . 52
4.7 Greedy Best-First Search . . . 57
4.7.1 Online Refinement in GBFS . . . 57
4.7.2 GBFS with Subgoal-Counting Lookahead and Online Refinement . 58
4.7.3 Experiments . . . 60
4.8 Experiments . . . 63
4.8.1 Comparison to Baselines without Online Refinement . . . 63
4.8.2 Online vs. Offline Conjunctions Quality . . . 65
4.8.3 Comparison to the State of the Art . . . 66
4.9 Related Work . . . 70
4.10 Conclusion . . . 72

5 Ranking Conjunctions for Partial Delete Relaxation Heuristics 73
5.1 Candidate Ranking Strategies . . . 74
5.1.1 Ranking Strategies . . . 75
5.1.2 Motivation . . . 76
5.1.3 Practical Remarks . . . 76
5.2 Online Ranking Strategies . . . 77
5.2.1 Strategies . . . 77
5.2.2 Motivation . . . 78
5.3 Conflict Extraction Algorithm . . . 78
5.4 Experiments . . . 79
5.4.1 Conflict Extraction Algorithm . . . 80
5.4.2 Candidate Ranking Strategies . . . 82
5.4.3 Online Ranking Strategies . . . 86
5.5 Conclusion . . . 88

6 Finding Plans with Red-Black State-Space Search 89
6.1 Combining RBS with ACI . . . 90
6.1.1 The RBS+ACI Framework . . . 91

6.1.2 Overall Planning Process: Iterated RBS+ACI 94
6.2 Adaptive Refinement via Realizability 95
6.2.1 Realizability Refinement: X-RBS 96
6.2.2 Combination with ACI 97
6.3 Experiments 98
6.3.1 Coverage 98
6.3.2 Number of Black Variables until Finding a Solution in RBS 101
6.4 Conclusion 101

II Adaptive Heuristic Search Techniques 103

7 Choosing the Initial State for Online Replanning 105
7.1 Previous Work 107
7.2 Continual Online Planning 108
7.3 The Multiple Initial State Technique 109
7.4 Theoretical Analysis 112
7.5 MIST for Recoverable Tasks 113
7.6 Experiments 116
7.6.1 Benchmarks 116
7.6.2 Results 118
7.7 Conclusion 121

8 Exploiting Heuristic Uncertainty in Deterministic Real-Time Search 123
8.1 Background 124
8.1.1 Problem Definition 125
8.1.2 LSS-LRTA* 125
8.1.3 Real-Time Search as Decision Making Under Uncertainty 126
8.2 The Nancy Framework 126
8.2.1 Risk-Based Lookahead 126
8.2.2 Persistent Action Selection 127
8.2.3 Nancy Backups 128
8.3 Theoretical Analysis 128
8.4 Conclusion 133

9 Exploiting Heuristic Uncertainty in Suboptimal Search 135
9.1 Background 137
9.1.1 Problem Definition 137
9.1.2 Bounded-Cost Search 137
9.1.3 Bounded-Suboptimal Search 139

9.2 Exploiting Heuristic Uncertainty in Bounded-Cost Search 140
9.2.1 Expected Effort Search 141
9.2.2 BEES with Explicit Probability Estimates 144
9.2.3 Experimental Evaluation 144
9.2.4 Summary 148
9.3 Exploiting Heuristic Uncertainty in Bounded-Suboptimal Search 148
9.3.1 Exploiting Expected Effort 148
9.3.2 A Simple Round-Robin Scheme 150
9.3.3 Experimental Evaluation 151
9.4 Conclusion 154

III Conclusion **157**

10 Conclusion **159**

Bibliography **161**

1 Introduction

Many problems in the field of computer science—in particular in the area of artificial intelligence (AI)—can be cast as state-space search, where an agent must find a path from an initial state to a state satisfying some goal property. In this process, a state describes the current configuration of the model of the underlying problem, and transitions to other states represent the modifications that are available on the current configuration. For example, consider a navigation task where an agent must find a route from one location to another. This can be modeled as state-space search, where each state represents the current position of the agent, transitions between states correspond to the possible movement of the agent, and a state is a goal if the agent is at the desired target location. While this is a straightforward example, state-space search is a powerful tool that can be applied to various kinds of problems, e.g., solving combinatorial puzzles [Korf and Taylor, 1996; Korf, 1997], playing games like chess or Go [Campbell et al., 2002; Silver et al., 2016; 2017], intelligently controlling devices such as elevators [Koehler and Ottiger, 2002] or industrial printers [Ruml et al., 2011], or verifying network security through simulated penetration testing [Hoffmann, 2015].

Most practical applications have state spaces that are too large to be fully explored or even represented in memory. For example, chess is estimated to have between 10^{46} and 10^{52} different reachable positions [Allis, 1994; Chinchalkar, 1996]—roughly equivalent to the number of atoms on the planet Earth ($\sim 10^{50}$). This calls for methods that do not require constructing the full state space, and aim to keep the part of the generated state space as small as possible. One common technique is *heuristic search*, where the search through the state space is guided by heuristics [e.g., Doran and Michie, 1966; Hart et al., 1968]. Such heuristics estimate the remaining distance to the goal, allowing the search to focus on the most promising states. For example, straight-line distance can be used as a simple heuristic for navigation problems to prioritize states which are closer to the target location.

One area where heuristic search is particularly useful is AI planning (see Ghallab et al.'s [2016] textbook for an overview). This subarea of AI is concerned with general problem

solving; in other words, the aim is to build solvers that are given a high-level description of the domain and problem at hand, and are effective without any additional human input or domain-specific knowledge. Classical AI planning tasks consist of a set of finite-domain state variables, and a set of actions that have preconditions and effects on these variables. The objective is to find a sequence of actions leading from an initial state to a state that satisfies the goal conditions. Heuristic search is one of the most prominent approaches to solve planning tasks [e.g., Bonet and Geffner, 2001; Hoffmann and Nebel, 2001; Helmert, 2006; Richter and Westphal, 2010], in particular in recent years. Planning heuristics are domain-independent, and typically generate their estimates by solving a relaxation (a simplified version) of the input task; for example by ignoring negative effects of the actions or considering abstract states that group many original states into one.

Most heuristic search methods follow a static approach: The search algorithm and heuristic are selected in advance, and the state space is then systematically explored, expanding outward from the initial state until a solution is found. The heuristic is assumed to be unchanged throughout this process, and its raw estimates are used for guidance without applying any further reasoning to them.

Yet during the search additional information may become available, for example by discovering particularly challenging aspects of the task at hand or finding that the heuristic is not always as accurate as expected. If such observations were somehow taken into account by the search algorithm, it would enable the search to enhance its effectiveness online by *adapting to the problem at hand*. Such online adaptation could take different forms, e.g., improving the accuracy of the heuristic function through refinement, or adjusting the search strategy to consider the uncertainty of the heuristic. However, this raises nontrivial questions: When should the search adapt? How should the search adapt? These questions are central to this thesis; and they must be answered carefully to avoid computational overhead, which may cancel out the benefits of such an approach.

We introduce several novel heuristic search techniques that follow a common paradigm of online adaptation. Our contributions are set in different variations of heuristic search, ranging from classical AI planning to more complex settings where an agent is moving through the search space and has a limited time to choose the next action, or where an agent is currently executing a plan and must react to changes in its environment like additional goals. We show that our adaptive search algorithms can effectively react to challenges and difficulties encountered during search, and their superior flexibility can yield substantial improvements over other state-of-the-art approaches.

1.1 Contributions

In the first part of this thesis, we explore *online relaxation refinement* in classical AI planning as heuristic search. A popular method to derive planning heuristics is the delete relaxation [Bonet and Geffner, 2001; Hoffmann and Nebel, 2001], which ignores negative effects of the actions. Our techniques build on partial delete relaxation methods [Keyder et al., 2014; Domshlak et al., 2015; Fickert et al., 2016], which extend the delete relaxation by un-relaxing some aspects of the task (at the cost of computational complexity), enabling a full interpolation between the standard delete relaxation and the original task.

Chapter 4 We introduce a family of heuristic search algorithms that improve the heuristic during search by refining its underlying relaxation. These algorithms identify areas of the search space where the heuristic is inaccurate, and use targeted refinement operations to address these weaknesses. Compared to traditional heuristic search methods that instantiate the heuristic before search and keep it unchanged throughout, our online refinement methods can reduce overhead by refining the heuristic only if it is necessary to do so, and make the heuristic more precise on the part of the search space that is explored by the search. We evaluate our algorithms with the partial delete relaxation heuristic h^{CFF} [Keyder et al., 2014; Fickert et al., 2016], which treats a given set of combinations of facts (variable-value pairs) as atomic, and demonstrate superior performance over related methods and state-of-the-art planners.

Chapter 5 The refinement operation of h^{CFF} is based on counterexample-guided abstraction refinement—an iterative procedure that analyzes flaws in the current model, and applies fine-grained refinement steps to address the observed flaws. Specifically, the refinement operation of h^{CFF} generates a set of possible conflicts, out of which a single one is selected, and the heuristic is refined to avoid the exact conflict in future computations. In this chapter, we systematically investigate which conflict should be selected, and empirically evaluate a wide range of strategies.

Chapter 6 Red-black planning is another type of partial delete relaxation, where variables are partitioned into two sets (called *painting*), of which one is treated with delete-relaxed semantics while the others retain their normal semantics [Domshlak et al., 2015]. This technique has been used to derive red-black heuristics, yet the painting must satisfy some conditions to keep the computation of the heuristic efficient (the *tractable fragment*). Another direction, and the one we extend in this chapter, is red-black search,

where the relaxation is used for the search instead of the heuristic. Specifically, the search is performed directly on the relaxed task, and the relaxation is refined until the computed plans are plans for the original task. We show how this search strategy can be combined with the tractable fragment to reduce the search effort, and design an adaptive variant that employs such refinement locally where needed.

In the second part of this thesis, we turn to different variations of heuristic search. We first consider online replanning, where an agent is currently executing a plan and must react to a change in its environment such as additional goals that must be achieved. In real-time search, an agent is moving through the state space and only has a limited time to act before each step. Finally, we consider bounded-suboptimal search, where the generated solutions must satisfy a given bound (either an absolute cost bound or a factor of the minimal solution cost).

Chapter 7 When the need for online replanning arises, one important question is how long the agent should proceed on the original plan, i.e., at which state the agent should deviate to a new plan. If the agent's execution reaches the chosen state too early, the planning process might not have finished. On the other hand—if the selected state is too far along the original plan, an opportunity to proceed more efficiently may have been missed. The most common solution of prior approaches is to always select a deviation state at some fixed time in the future, but this requires making an offline estimate of the planning time which may be inaccurate. We introduce an algorithm that incorporates this choice into the search process itself, and is able to select the most suitable state by reasoning about its own planning time. Our evaluation shows that this approach yields robust behavior and consistently outperforms methods using offline predictions.

Chapter 8 In real-time search, the agent must commit to the next action within a fixed time bound. In this time frame, only a small portion of the search space ahead can be explored during a short lookahead such that the agent may move towards the most promising state according to the heuristic. However, even in deterministic search, the heuristic estimates contain an inherent uncertainty. Recent work has shown how this uncertainty can be modeled explicitly [Mitchell et al., 2019], replacing heuristic values with belief distributions in the search. We provide a theoretical analysis of this framework here, and prove that this approach is complete, i.e., if the task is solvable then the search will eventually find a solution. Our proof applies to a general class of real-time search algorithms, in particular, we provide a more general completeness proof for LSS-LRTA* [Koenig and Sun, 2009]—one of the most popular state-of-the-art real-time search algorithms—that imposes fewer restrictions on the heuristic than the original proof.

Chapter 9 In many applications, computing optimal solutions is too expensive, yet high-quality solutions are desired. In that case, one may employ bounded-cost search, which aims to find a plan within a given absolute cost bound as quickly as possible. Building on the aforementioned models of the heuristic uncertainty, we show how to estimate the probability of finding a solution within the bound under a given state, and introduce a search algorithm that aims to minimize the expected search time. We prove that this search strategy is optimal in a simplified model, and show that it is highly effective in practice.

1.2 Experiments Setup

Throughout this thesis, we interleave theoretical contributions with corresponding empirical evaluations. We give an overview over the shared basic setup here, which holds for all experiments unless noted otherwise.

All algorithms are implemented in Fast Downward [Helmert, 2006]; the source code is publicly available and linked in the respective chapters. The experiments were run using the Downward Lab framework [Seipp et al., 2017] on a cluster of Intel Xeon E5-2660 processors with a clock rate of 2.2 GHz. The time and memory limits were set to 30 minutes respectively 4 GB.

The algorithms discussed in this work are evaluated on the benchmarks of the International Planning Competitions (IPC).[1] We give more details on the specific instances used in each chapter as the benchmark sets are slightly different and sometimes require additional inputs (for example, a cost bound for our bounded-cost search experiments in Chapter 9). We typically consider the *coverage* (the number of solved instances of the benchmark set) as the main indication for an algorithm's performance. In addition to the overall coverage, we sometimes discuss the *normalized coverage* to account for different numbers of instances across the domains of the benchmark set (computed as the average fraction of solved instances per domain). Another important measure is the *search time*, in which we exclude the preprocessing time of Fast Downward as it is the same among all algorithms.

[1] See `https://www.icaps-conference.org/competitions/`.

1.3 Publications

This thesis is mainly based on the following publications:[2]

- Maximilian Fickert, Tianyi Gu, and Wheeler Ruml: "New Results in Bounded-Suboptimal Search". In *Proceedings of the 36th AAAI Conference on Artificial Intelligence, AAAI 2022*.
- Maximilian Fickert and Jörg Hoffmann: "Online Relaxation Refinement for Satisficing Planning: On Partial Delete Relaxation, Complete Hill-Climbing, and Novelty Pruning". In *Journal of Artificial Intelligence Research, Volume 73*.
- Maximilian Fickert, Tianyi Gu, and Wheeler Ruml: "Bounded-cost Search Using Estimates of Uncertainty". In *Proceedings of the 30th International Joint Conference on Artificial Intelligence, IJCAI 2021*.
- Maximilian Fickert, Ivan Gavran, Ivan Fedotov, Jörg Hoffmann, Rupak Majumdar, and Wheeler Ruml: "Choosing the Initial State for Online Replanning". In *Proceedings of the 35th AAAI Conference on Artificial Intelligence, AAAI 2021*.
- Maximilian Fickert: "A Novel Lookahead Strategy for Delete Relaxation Heuristics in Greedy Best-First Search". In *Proceedings of the 30th International Conference on Automated Planning and Scheduling, ICAPS 2020*.
- Maximilian Fickert, Tianyi Gu, Leonhard Staut, Wheeler Ruml, Jörg Hoffmann, and Marek Petrik: "Beliefs We Can Believe in: Replacing Assumptions with Data in Real-Time Search". In *Proceedings of the 34th AAAI Conference on Artificial Intelligence, AAAI 2020*.
- Maximilian Fickert: "Making Hill-Climbing Great Again through Online Relaxation Refinement and Novelty Pruning". In *Proceedings of the 11th International Symposium on Combinatorial Search, SOCS 2018*.
- Maximilian Fickert, Daniel Gnad, and Jörg Hoffmann: "Unchaining the Power of Partial Delete Relaxation, Part II: Finding Plans with Red-Black State Space Search". In *Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI 2018*.
- Maximilian Fickert and Jörg Hoffmann: "Complete Local Search: Boosting Hill-Climbing through Online Relaxation Refinement". In *Proceedings of the 27th International Conference on Automated Planning and Scheduling, ICAPS 2017*.

[2] We give more details on how these works relate to this thesis and the individual contributions at the beginning of the chapters that build on them.

- Maximilian Fickert and Jörg Hoffmann: "Ranking Conjunctions for Partial Delete Relaxation Heuristics in Planning". In *Proceedings of the 10th International Symposium on Combinatorial Search, SOCS 2017*.

Additionally, the following papers have been published during the author's doctoral studies but are not part of this thesis:

- Joschka Groß, Álvaro Torralba, and Maximilian Fickert: "Novel Is Not Always Better: On the Relation between Novelty and Dominance Pruning". In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*.
- Rebecca Eifler, Maximilian Fickert, Jörg Hoffmann, and Wheeler Ruml: "Refining Abstraction Heuristics during Real-Time Planning". In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*.
- Rebecca Eifler and Maximilian Fickert: "Online Refinement of Cartesian Abstraction Heuristics". In *Proceedings of the Eleventh International Symposium on Combinatorial Search, SOCS 2018*.

2 Background

We next introduce the common background on heuristic state-space search and AI planning that provide the theoretical framework for our contributions.

2.1 Transition Systems

State spaces are formally defined as *transition systems*:

Definition 2.1 (Transition System)**.** A labeled, weighted *transition system* is a tuple $\Theta = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, c, \mathcal{I}, \mathcal{G} \rangle$, where

- $\mathcal{S}$ is the (finite) set of *states*,
- $\mathcal{A}$ is the (finite) set of *actions*,
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the *transition relation*,
- $c : \mathcal{A} \mapsto \mathbb{R}_0^+$ is the *cost function*,
- $\mathcal{I} \in \mathcal{S}$ is the *initial state*, and
- $\mathcal{G} \subseteq \mathcal{S}$ is the set of *goal states*.

We usually denote a transition $(s, a, s') \in \mathcal{T}$ by $s \xrightarrow{a} s'$ (or just $s \rightarrow s'$ when a is not relevant or clear from the context). In this work, assume the transition relation to be *deterministic*, i.e., for each pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ there is at most one $s' \in \mathcal{S}$ such that $s \xrightarrow{a} s' \in \mathcal{T}$. We say an action a is *applicable* in a state s if there exists a state s' such that $s \xrightarrow{a} s' \in \mathcal{T}$, and denote its *successor* through a by $s[\![a]\!] := s'$ (conversely, s is a *predecessor* of s'). The set of applicable actions in a state s is denoted by $\mathcal{A}(s)$.

If the cost function returns one for all actions, we say that Θ has *unit costs*. In some contexts, we consider transition systems $\Theta = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{I}, \mathcal{G} \rangle$ where the cost function is omitted; we assume unit costs in this case.

Given a sequence of transitions of the form $s_0 \xrightarrow{a_1} s_1, s_1 \xrightarrow{a_2} s_2, \ldots, s_{n-1} \xrightarrow{a_n} s_n$, the action sequence $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ is called a *path* from s_0 to s_n, and its cost $C(\pi)$ is the sum of the cost of its actions $\sum_{i=1}^{n} c(a_i)$. A *solution* for a state $s \in \mathcal{S}$ is a path from s to a state $s_G \in \mathcal{G}$ (also called a *plan*), and it is *optimal* if its cost is minimal among all solutions for s. A solution for a transition system Θ is a solution for its initial state $\mathcal{I}$.

We say that a state s' is *reachable* from another state s if there exists a (possibly empty) path from s to s'. If at least one goal state is reachable from a state s then s is *solvable*, otherwise it is called a *dead end*.

A *search node* n is a state in the search space with an associated path to it from the initial state. The cost of this path is denoted by $g(n)$ and is usually referred to as the node's *g value*.

2.2 Heuristic Search

A *heuristic function* (or short *heuristic*), is a function $h : \mathcal{S} \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ that estimates the cost of a solution for a given state s, or returns ∞ to indicate that s is a dead end. The *perfect heuristic* h^* returns the cost of an optimal solution for solvable states, and ∞ otherwise.

A heuristic is *admissible* if it never overestimates the optimal solution cost, i.e., $h(s) \leq h^*(s)$ for all states $s \in \mathcal{S}$. This is an important property if optimal solutions are desired since it is a requirement for most optimal search algorithms such as A* [Hart et al., 1968]. A stronger property is *consistency*, which requires that $h(s) \leq h(s[\![a]\!]) + c(a)$ for all states s and all applicable actions a. A heuristic is *goal-aware* if $h(s_G) = 0$ for all goal states $s_G \in \mathcal{G}$. Throughout this work, we assume that heuristics are *safe*, that is, if $h(s) = \infty$ then s is indeed a dead end (this is the case for the vast majority of heuristics that are used in practice, especially in domain-independent planning).

Sometimes it can be useful to estimate the *goal distance* $d(s)$ for some state s, i.e., the number of transitions from s to reach a goal state. Distance estimates are typically derived from a heuristic by considering unit action costs.

Another important measure of a search node n is its f value, $f(n) := g(n) + h(n)$. If h is admissible, then $f(n)$ is a lower bound on the solution cost below n.

The most common type of search algorithms are those that systematically explore the state space, such as A* [Hart et al., 1968] or Greedy Best-First Search (GBFS). Starting from the initial state, these algorithms iteratively expand states (generating their successors) until a goal state is reached. The states that have been generated but not yet

expanded are kept in an *open list*. The search strategy is defined by the open list ordering; for example, A* orders the search nodes in the open list by increasing f value, GBFS orders them by increasing h. A search algorithm is called *complete* if it terminates in finite time, returning a plan in case the task is solvable, or proving unsolvability otherwise (A* and GBFS are examples of complete search algorithms).

2.3 Classical AI Planning

We formalize planning tasks using the finite-domain representation (FDR) [Bäckström and Nebel, 1995; Helmert, 2009], where states are encoded by variables with finite domains, and actions have preconditions and effects on these variables.

Definition 2.2 (FDR Planning Task). An FDR planning task is a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, c, \mathcal{I}, \mathcal{G} \rangle$, where

- $\mathcal{V}$ is the (finite) set of *variables*, where each $v \in \mathcal{V}$ has a finite *domain* $\mathcal{D}_v$,
- $\mathcal{A}$ is the (finite) set of *actions*, where each $a \in \mathcal{A}$ is a pair $(\mathsf{pre}(a), \mathsf{eff}(a))$ of its *preconditions* and *effects* (each is a partial assignment of $\mathcal{V}$),
- $c : \mathcal{A} \mapsto \mathbb{R}_0^+$ is the *cost function* (we sometimes omit this, and assume unit action costs in this case),
- $\mathcal{I}$ is the *initial state* (complete assignment of $\mathcal{V}$), and
- $\mathcal{G}$ is the *goal* (partial assignment of $\mathcal{V}$).

A variable-value pair $var = val$ is called a *fact*, and we denote the set of all facts by $\mathcal{F}$. A *state* is a complete assignment of $\mathcal{V}$, and we usually treat them as sets of facts. The set of all states is denoted by $\mathcal{S}$. For a partial variable assignment p, we denote the set of variables defined in p by $\mathcal{V}(p)$, and we denote the projection of p to a subset of its variables $V \subseteq \mathcal{V}(p)$ by $p|_V$. If a variable v is defined in a partial assignment p (i.e., $v \in \mathcal{V}(p)$), we denote its value in p by $p(v)$. An action a is *applicable* in a state s if its preconditions are satisfied in s, i.e., $\mathsf{pre}(a) \subseteq s$ (equivalent to $s|_{\mathcal{V}(\mathsf{pre}(a))} = \mathsf{pre}(a)$), and the successor $s[\![a]\!]$ retains the variable assignments from s in variables where $\mathsf{eff}(a)$ is not defined, and has the assignments $\mathsf{eff}(a)(v)$ in the variables $v \in \mathcal{V}(\mathsf{eff}(a))$.

We will require the concept of *regression*, which describes the minimal set of facts required to achieve a subgoal through a given action. A set of facts $g \subseteq \mathcal{F}$ is *regressable* over an action $a \in \mathcal{A}$ if:

- $\mathsf{eff}(a) \cap g \neq \emptyset$ (a achieves some part of g),
- there is no variable $v \in \mathcal{V}$ which is defined in both $\mathsf{eff}(a)$ and g but with different values (a does not invalidate g), and
- there is no variable $v \in \mathcal{V}$ which is defined in both $\mathsf{pre}(a)$ and g but with different values, and which is not defined in $\mathsf{eff}(a)$ (the preconditions of a are compatible with g).

If g is regressable over a, then the *regression of g over a* is defined as $R(g,a) = (g \setminus \mathsf{eff}(a)) \cup \mathsf{pre}(a)$, otherwise we write $R(g,a) = \bot$.

Some planning heuristics make use of the *causal graph*, which describes dependencies between variables [Knoblock, 1994; Brafman and Domshlak, 2003]. The causal graph is a directed graph $\langle V, E\rangle$ with vertices $V = \mathcal{V}$, and there is an arc $\langle v, v'\rangle \in E$ if $v \neq v'$ and there exists an action $a \in \mathcal{A}$ such that v is defined in $\mathsf{pre}(a)$ and v' is defined in $\mathsf{eff}(a)$, or both v and v' are defined in $\mathsf{eff}(a)$.

2.3.1 Induced Transition System

A planning task $\Pi = \langle \mathcal{V}, \mathcal{A}, c, \mathcal{I}, \mathcal{G}\rangle$ *induces the transition system* $\Theta_\Pi = \langle \mathcal{S}', \mathcal{A}', \mathcal{T}, c', \mathcal{I}', \mathcal{G}'\rangle$, where

- $\mathcal{S}'$ is the set of all states $\mathcal{S}$ of Π,
- $\mathcal{A}' = \mathcal{A}$,
- $\mathcal{T}$ is defined by the actions of Π: for each state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, iff a is applicable in s then $s \xrightarrow{a} s[\![a]\!]$ is a transition in $\mathcal{T}$,
- $c' = c$,
- $\mathcal{I}' = \mathcal{I}$, and
- $\mathcal{G}' = \{s \in \mathcal{S} \mid s \supseteq \mathcal{G}\}$ is the set of states consistent with $\mathcal{G}$.

A solution for Θ_Π is also a solution for Π and vice versa (retaining optimality). When the terms defined for transition systems are used for a planning task Π they refer to its transition system Θ_Π; e.g., a path from a state s to a state s' is a sequence of actions that can be consecutively applied to s and result in the state s'.

2.3.2 Planning Heuristics

Most planning heuristics are based on a *relaxation* Π^+ of the original task Π, and base their estimate for a state s on a plan $\pi[h](s)$ computed in that relaxation. In addition to a heuristic value $h(s)$, some heuristics also yield a set of *helpful actions* (in some contexts also called *preferred operators*) $H(s)$ [Hoffmann and Nebel, 2001; Richter et al., 2008; Richter and Helmert, 2009]. For heuristics based on a relaxation, $H(s)$ is typically the subset of actions in $\pi[h](s)$ that is applicable in s. A search that uses *helpful actions pruning* only considers the actions $H(s)$ when expanding a state s, discarding all other successors of s [Hoffmann and Nebel, 2001]. Helpful actions pruning can be a source of incompleteness, as $H(s)$ may exclude the only actions that start a plan for s. An alternative and completeness-preserving method to exploit helpful actions is the use of a *dual queue* [Helmert, 2006]. The search then maintains two separate open lists—one containing all open search nodes, and one containing only nodes generated by helpful actions—and nodes are expanded from each list alternately.

The first part of this thesis is concerned with *satisficing* AI planning, where solutions are not required to be optimal. In the second part, we will consider more general heuristic search problems (not restricted to AI planning). We explore various settings with additional challenges such as parallel planning and execution or suboptimality bounds, and introduce the relevant definitions in the respective chapters.

Part I

Adaptive Partial Delete Relaxation

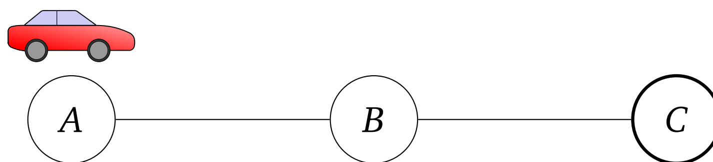
3 Partial Delete Relaxation

The *delete relaxation* [Bonet and Geffner, 2001; Hoffmann and Nebel, 2001], under which variables accumulate their values instead of switching between them, is one of the most popular relaxations for heuristics used in satisficing planning. However, delete relaxation heuristics sometimes ignore critical aspects of a planning task such as fuel consumption. By "un-relaxing" critical parts of the planning task, *partial delete relaxation* attempts to address these issues and make the relaxation more precise. Our work describes contributions to different partial delete relaxation approaches; we introduce the relevant background in this chapter.

3.1 Delete Relaxation

In the delete relaxation, any facts that are made true are assumed to remain true forever. In FDR planning, that means that state variables can accumulate multiple values instead of switching between them.

Example 3.1. *Consider the following task:*



The car must drive from A to C, consuming fuel at each step. Initially, the car holds one unit of fuel, so a plan for this task must refuel at location B before it can proceed to C.

Formally, the task has a variable "fuel" with domain $\{0, 1\}$ *indicating the amount of available fuel, and a variable "at" with domain* $\{A, B, C\}$ *indicating the position of the car. The "refuel" action has no preconditions, and its only effect is setting fuel to* 1*. The "drive(x,y)" actions take the car from location x to location y; they have preconditions* $\{at = x, fuel = 1\}$ *and effects* $\{at = y, fuel = 0\}$.

One plan for this task is ⟨drive(A, B), refuel, drive(B, C)⟩. A delete-relaxed plan—for example ⟨drive(A, B), drive(B, C)⟩—does not need to include the refuel action, since the fuel = 1 fact is still available after applying a drive action in the relaxation.

The *perfect delete relaxation heuristic* h^+ maps each state to the length of an optimal delete-free plan, or to ∞ if no such plan exists. Computing an optimal delete-relaxed plan is **NP**-complete [Bylander, 1994]. Hence, in practice the approximative h^{FF} heuristic [Hoffmann and Nebel, 2001] is used instead, which bases its estimates on (not necessarily optimal) delete-relaxed plans $\pi[h^{\text{FF}}]$ (or returns ∞ in the same case as h^+). These plans for h^{FF} are usually generated using a *best-supporter function* $bs : \mathcal{F} \mapsto \mathcal{A}$, which, given a fact $f \in \mathcal{F}$, returns an action that is deemed to be the cheapest achiever for f. The plan extraction procedure then starts by adding the best supporter of each goal fact to the relaxed plan, and preconditions of the selected actions are then recursively propagated until an achiever for each open precondition (that is not already true in the current state) has been added.

Most implementations of h^{FF} use the additive heuristic h^{add} [Bonet and Geffner, 2001] as the best-supporter function. Given a state s, the heuristic assigns a value to each fact f according to the following equation:

$$h^{\text{add}}(s, f) = \begin{cases} 0 & \text{if } f \in s, \\ \min\limits_{a \in \mathcal{A}, f \in \mathsf{eff}(a)} [c(a) + \sum\limits_{p \in \mathsf{pre}(a)} h^{\text{add}}(s, p)] & \text{otherwise.} \end{cases}$$

Intuitively, the heuristic yields a pessimistic estimate of the cost to achieve a given fact f under the delete relaxation—it simply sums up the costs of all preconditions on a delete-relaxed path to f, disregarding the fact that making progress to one of the preconditions may help with achieving another one. The best supporter for each fact f is the $\arg\min$ of the minimum in the second case in the equation, i.e., the action for which the sum of h^{add} values of its preconditions added to its cost is lowest.

Delete relaxation is a simple approach that yields effective heuristics in practice [e.g., Hoffmann and Nebel, 2001; Richter and Westphal, 2010]. However, it ignores some aspects of the task—like the fuel consumption in the example above—which may render the heuristic inaccurate in some domains. The concept of *partial delete relaxation* aims to improve the accuracy of delete relaxation heuristics by taking *some* delete information into account. There are two main lines of research on partial delete relaxation: treating *conjunctions of facts* as atomic [Keyder et al., 2014; Fickert et al., 2016], and *red-black planning*, which relaxes only *some* variables instead of all of them [Domshlak et al., 2015].

3.2 Partial Delete Relaxation through Explicit Conjunctions

One technique for partial delete relaxation is based on *explicit conjunctions*, where a given set of conjunctions (fact sets) C are treated as atomic, and the facts contained in a conjunction $c \in C$ must be achieved *simultaneously* [Haslum, 2012; Keyder et al., 2012; 2014; Hoffmann and Fickert, 2015; Fickert et al., 2016]. The $h^{C\text{FF}}$ heuristic and its idealized counterpart h^{C+} compute such C*-relaxed* plans: Whenever a conjunction $c \in C$ is a subset of the preconditions of an action, the partially relaxed plan $\pi[h^{C\text{FF}}]$ must satisfy c instead of the individual facts contained in c. A conjunction c can only be achieved by an action a if c is regressable over a, i.e., a makes some part of the conjunction true and its other preconditions and effects are not incompatible with c, and the remaining facts of c that are not achieved by a are treated as additional preconditions.

Example 3.2. *Consider again the task shown in Example 3.1. The critical issue of the relaxed plan in Example 3.1 is that the preconditions of the drive*(B,C) *action are not satisfied under normal semantics, as the fact fuel* $=1$ *is assumed to still be available after driving to* B*. Consider the conjunction* $c = \{fuel = 1, at = B\}$*. If* c *is contained in the set of conjunctions* C *used by the heuristic, then a* C*-relaxed plan must consider* c *as a required precondition for drive*(B,C)*. Observe that* c *can only be achieved through the refuel action: The only actions that achieve some part of* c *are refueling and the drive*(x,B) *actions, but the latter also sets the fuel variable to* 0*, which conflicts with* c *(which contains fuel* $=1$*). Furthermore, achieving* c *through refueling requires at* $=B$ *to be true beforehand, so the* C*-relaxed plan that* h^{C+} *would compute for this example is* $\langle drive(A,B), refuel, drive(B,C)\rangle$*, which is also a real plan.*

The set of conjunctions C controls the degree of the relaxation for the corresponding heuristic. Throughout this work, we assume that C always consists of at least all singleton facts, i.e., $C \supseteq C_0$ where $C_0 := \{\{f\} \mid f \in \mathcal{F}\}$. If C does not contain any non-singleton facts ($C = C_0$), then a C-relaxed plan is just a relaxed plan and $h^{C+} = h^+$. On the other hand, if it contains all combinations of facts, i.e., $C = \mathcal{P}(\mathcal{F})$, then every C-relaxed plan is also a plan under non-relaxed semantics, and $h^{C+} = h^*$. Hence, this allows for a smooth interpolation between fully relaxed and non-relaxed semantics. In practice, the set of conjunctions for $h^{C\text{FF}}$ must be chosen carefully, as the heuristic becomes more expensive to compute with each added conjunction. Standard methods generate C using counterexample-guided abstraction refinement (see Section 3.2.2).

3.2.1 $h^{C\text{FF}}$ in Practice

Like h^{FF}, the partially relaxed plans for $h^{C\text{FF}}$ are extracted using a best-supporter function. The standard choice (and the one we use in this work) is $h^{C\text{add}}$, which generalizes h^{add}

to an arbitrary set of conjunctions C:

$$h^{Cadd}(s,c) = \begin{cases} 0 & \text{if } c \subseteq s \\ \min_{a \in \mathcal{A}, R(c,a) \neq \bot} [c(a) + \sum_{c_{\mathsf{pre}} \in R(c,a)^C} h^{Cadd}(s, c_{\mathsf{pre}})] & \text{otherwise} \end{cases}$$

The definition makes use of the shorthand "X^C" to indicate all conjunctions that are contained in X, i.e., $\{c \in C \mid c \subseteq X\}$. In practice, we only consider the *non-dominated* conjunctions, i.e., those that are not a subset of a different conjunction also contained in the set.

We can compute the h^{Cadd} values through a forward exploration until a fixed point is reached. First, $h^{Cadd}(s,c)$ is set to 0 for all conjunctions $c \subseteq s$. Now we can iteratively set the h^{Cadd} value for conjunctions where h^{Cadd} is yet undefined, and for which there is an action a such that h^{Cadd} is defined for all conjunctions $c_{\mathsf{pre}} \in R(c,a)^C$. In that case, $h^{Cadd}(s,c)$ is set to the sum of the h^{Cadd} values of these conjunctions c_{pre} plus the cost of the action.

This algorithm can be efficiently implemented using counters that keep track of the number of precondition conjunctions $c_{\mathsf{pre}} \in R(c,a)^C$ that need to be made true before a conjunction c can be reached through an action a [Hoffmann and Fickert, 2015; Fickert et al., 2016].[1] Specifically, such counters are attached to each conjunction-action pair (c,a) where a can be used to achieve c, i.e., all such pairs where $R(c,a) \neq \bot$. The counters are initially set to $|R(c,a)^C|$, and are decremented whenever the h^{Cadd} value for a conjunction in $R(c,a)^C$ is set. When a counter attached to a pair (c,a) reaches zero, all precondition conjunctions that are necessary to achieve c through a have been reached, and c is reached in the next iteration.

In the experiments, we will sometimes discuss the increase in computational complexity of h^{CFF} as conjunctions are added to C. Since, computing h^{Cadd} is typically the main bottleneck to generate a C-relaxed plan for h^{CFF}, we approximate the complexity of h^{CFF} by the number of counters that are maintained in h^{Cadd}. We measure this as a factor of the number of counters that are being tracked by h^{CFF} when using only singleton conjunctions; for example, if the heuristic has a *growth factor* of 2, that means that the implementation must keep track of twice as many counters as with C_0.

[1] This is similar to the h^{FF} implementation [Hoffmann and Nebel, 2001], which tracks the number of unsatisfied preconditions for each action.

3.2.2 The Refinement Operation of h^{CFF}

While h^{CFF} can use an arbitrary set of conjunctions according to the desired degree of relaxation, C must be chosen carefully in practice since each added conjunction makes the heuristic more expensive to compute. The known methods iteratively generate C via counterexample-guided abstraction refinement: Let s be a state where $h^{\mathrm{CFF}}(s) \neq \infty$, and let $\pi[h^{\mathrm{CFF}}](s)$ be the corresponding partially relaxed plan. Either $\pi[h^{\mathrm{CFF}}](s)$ is a real plan for s, or there must be a conflict in the form of an invalidated precondition or goal. In the latter case, a conjunction c can be generated to address that conflict, and c is added to C. In the example discussed above, the deletion of the $fuel = 1$ fact by the $drive(A, B)$ action forms a conflict, and the conjunction $\{fuel = 1, at = B\}$ can be added to address it.

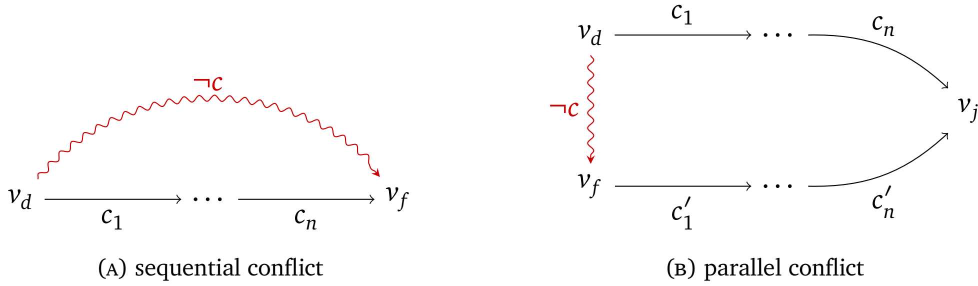
The original refinement algorithm by Haslum [2012] guarantees that $\pi[h^{\mathrm{CFF}}](s)$ is no longer a valid C-relaxed plan after adding the generated conjunctions to C. Keyder et al.'s [2014] method generates only a single conjunction, with the weaker guarantee that this conjunction was not contained in C before. However, this method is more efficient in practice, since adding a single conjunction is typically sufficient for h^{CFF} to compute a different plan, and it induces significantly less computational overhead. Hence, we use Keyder et al.'s [2014] method in this work; we summarize the algorithm in the following.

Internally, the partially relaxed plans returned by h^{CFF} consist of pairs of an action and a set of supported conjunctions, called *action occurrences*. For an action occurrence (a, G), the set of supported conjunctions $G \subseteq C$ indicates the conjunctions that are achieved by a. The preconditions of (a, G) consist of the preconditions of a and the parts of the conjunctions that are not achieved by a, i.e., $\mathsf{pre}((a, G)) = (\bigcup_{c \in G} R(c, a))^C$. Keyder et al.'s [2014] refinement method first constructs a data structure called *best-supporter graph (BSG)* from the partially relaxed plan.

Definition 3.1 (Best-Supporter Graph)**.** Given a state s for a planning task Π, a set of action occurrences O corresponding to a partially relaxed plan $\pi[h^{\mathrm{CFF}}](s)$, and the corresponding best supporter function bs, the *best-supporter graph* is a directed acyclic graph $\phi = \langle V, E \rangle$, where $V = O \cup \{(a_{\mathcal{G}}, \emptyset)\}$, and $E = \{\langle v, v' \rangle \mid \exists c \in \mathsf{pre}(v'), v = bs(c)\}$. Each vertex is labelled with the action occurrence it represents, and each arc $\langle v, v' \rangle$ is labelled with the set of (non-dominated) precondition conjunctions $\{c_{pre} \mid c_{pre} \in \mathsf{pre}(v'), v = bs(c_{pre})\}$.

The BSG models the dependencies between action occurrences: if there is an arc from a vertex v to another vertex v', the action occurrence represented by v achieves a conjunction that is required as a precondition for the action occurrence represented by v'. There is an additional vertex for the goal which is represented by an artificial action $a_{\mathcal{G}}$ with $\mathsf{pre}(a_{\mathcal{G}}) = \mathcal{G}$, and incoming arcs from each action occurrence that achieves a goal conjunction.

If the partially relaxed plan fails to execute in the original planning task, there must be an action (the *deleter*) that invalidates a precondition of another action (or the goal) occurring later in the plan (the *failed action*). The deleter and failed action can either be ordered sequentially or have a common descendant in the BSG, as shown in Figure 3.1.



(A) sequential conflict (B) parallel conflict

FIGURE 3.1: Conflict types in relaxed plans.

In the sequential case, there is a path from the deleter d to the failed action f. Let L be the set of labels on the path from d to f. Possible new conjunctions addressing this conflict are $\{c_L \cup c \mid c_L \in L\}$. In the parallel case, there is no path from d to f, but they have a common descendant j. Let L_d be the set of edge labels on the path from d to j and L_f be the edge labels on the path from f to j. Possible new conjunctions are $\{c_{L_d} \cup c_{L_f} \mid c_{L_d} \in L_d, c_{L_f} \in L_f \cup \{c\}\}$. As only one conjunction will be selected to be added to C, it suffices to consider one candidate for a single conflict to avoid redundancy. The considered candidate conjunction is $c_n \cup c$ in the sequential case and $c_n \cup c'_n$ in the parallel case, which is guaranteed to not be contained in C yet [Keyder et al., 2014, Lemma 3].

In Chapter 4, we will introduce a family of search algorithms that refine the heuristic online, and use h^{CFF} to evaluate them. In Chapter 5, we evaluate different strategies to rank candidate conjunctions in the h^{CFF} refinement procedure, as well as ranking strategies for conjunctions that have already been added to C.

3.3 Red-Black Planning

Red-black planning [Katz et al., 2013a; b; Domshlak et al., 2015; Gnad et al., 2016], where only some variables are treated with delete-relaxed semantics, is another approach to partial delete relaxation.

Definition 3.2 (Red-Black Planning Task). A red-black planning task is a tuple $\Pi^{\text{RB}} = \langle \mathcal{V}^{\text{R}}, \mathcal{V}^{\text{B}}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ with $\mathcal{V}^{\text{B}} \cap \mathcal{V}^{\text{R}} = \emptyset$, where

- $\mathcal{V}^{\text{R}}$ is the set of *red variables*, which have delete-relaxed semantics,
- $\mathcal{V}^{\text{B}}$ is the set of *black variables*, which have standard FDR semantics, and

- $\Pi = \langle \mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ with $\mathcal{V} = \mathcal{V}^{\mathrm{R}} \cup \mathcal{V}^{\mathrm{B}}$ is an FDR planning task.

In a red-black state s^{RB}, the red variables are mapped to a subset of their domain, while black variables use the standard semantics. In other words, a red-black state may contain multiple facts for each red variable (like a delete-relaxed state), and has exactly one fact for each black variable (like a standard FDR state). The red-black initial state $\mathcal{I}^{\mathrm{RB}}$ consists of the same facts as $\mathcal{I}$, and a red-black state s^{RB} is a goal state if $\mathcal{G} \subseteq s^{\mathrm{RB}}$. An action a is applicable to a red-black state s^{RB} if $\mathsf{pre}(a) \subseteq s^{\mathrm{RB}}$. The state $s^{\mathrm{RB}}[\![a]\!]$ resulting from applying a in s^{RB} is the same state as s for all variables where $\mathsf{eff}(a)$ is not defined, the black effects switch (i.e., for variables $v^{\mathrm{B}} \in \mathcal{V}(\mathsf{eff}(a)) \cap \mathcal{V}^{\mathrm{B}}$, we get $s^{\mathrm{RB}}[\![a]\!](v^{\mathrm{B}}) = \mathsf{eff}(a)(v^{\mathrm{B}})$), and the red effects accumulate (i.e., for variables $v^{\mathrm{R}} \in \mathcal{V}(\mathsf{eff}(a)) \cap \mathcal{V}^{\mathrm{R}}$, we get $s^{\mathrm{RB}}[\![a]\!](v^{\mathrm{R}}) = s^{\mathrm{RB}}(v^{\mathrm{R}}) \cup \{\mathsf{eff}(a)(v^{\mathrm{R}})\}$). A plan for the red-black task Π^{RB} is called a red-black plan, and we also refer to it as a red-black plan for the underlying task Π.

The specific red-black relaxation is defined by its *painting*, i.e., the partitioning of the variables $\mathcal{V}$ into red variables $\mathcal{V}^{\mathrm{R}}$ and black variables $\mathcal{V}^{\mathrm{B}}$. The degree of the relaxation depends on the black variables in the painting: If all variables are painted black ($\mathcal{V}^{\mathrm{B}} = \mathcal{V}$), then the red-black task uses standard semantics; if all variables are painted red ($\mathcal{V}^{\mathrm{B}} = \emptyset$), then the red-black task is fully delete-relaxed. The paintings thus form a partial order, where a painting with black variables $\mathcal{V}^{\mathrm{B}}_1$ is more refined than a painting with black variables $\mathcal{V}^{\mathrm{B}}_2 \subsetneq \mathcal{V}^{\mathrm{B}}_1$.

To illustrate techniques related to red-black planning, we use a slightly more complex running example:

Example 3.3. *Consider the following task:*

The task is to drive to the store at location B, buy two products P_1 and P_2, and return to A. This time, there is no fuel consumption, so the drive actions only have preconditions and effects on the position of the car. The two products have associated variables "have–P_1" and "have–P_2" with domain $\{0,1\}$; they are set to 0 in the initial state, and buying the corresponding product in the store sets them to 1. We have two units of money available initially (money $= 2$), and buying each product costs one. For each product P_x and available amount of money $m > 0$, there is a buy action buy(P_x, m) with preconditions $\{at = B, money = m\}$ and effects $\{money = m-1, have\text{–}P_x = 1\}$. The goal is $\{at = A, have\text{–}P_1 = 1, have\text{–}P_2 = 1\}$.

A plan for this task is $\langle drive(A,B), buy(P_1,2), buy(P_2,1), drive(B,A)\rangle$. *Under the delete relaxation, a possible plan is* $\langle drive(A,B), buy(P_1,2), buy(P_2,2)\rangle$. *This relaxed plan has two flaws: (1) the car does not return to location A after buying the products, and (2) it does not account for the spent money, so it may use an incompatible buy action for the second product.*

To address both issues, we must paint both the "at" and "money" variables black: un-relaxing "at" forces the car to return to A, and un-relaxing "money" ensures that the money is accounted for. The have—P_x *variables can be treated with relaxed semantics, since their values only change once and do not need to change back (as there are no conditions on have—*$P_x = 0$*). Any red-black plan using the painting* $\mathcal{V}^{\mathrm{B}} = \{at, money\}$ *is also a plan for the original task.*

3.3.1 Tractable Fragment (ACI)

The initial line of research on red-black planning focused on its use as a heuristic [Domshlak et al., 2015], culminating in the success of the Mercury planner in the 2014 International Planning Competition [Katz and Hoffmann, 2014]. The red-black heuristic is based on the *tractable fragment ACI*, which imposes some constraints on the painting to make the generation of red-black plans possible in polynomial time. We summarize this approach in the following, simplifying some details for easier exposition.

ACI requires that (a) the causal graph over the black variables is *acyclic*, and that (b) every black variable is *invertible*. In this context, a variable v is invertible if, for every action a that changes the value of v, there is an action a' that can revert the value of v under the same or easier conditions on other variables.

Example 3.4. *Consider the task shown in Example 3.3. Painting the location of the car black is possible. The black causal graph is acyclic (since it only contains one vertex and does not have any arcs), and all actions with effects on the black variable (i.e., the drive actions) are invertible. Each* $drive(x,y)$ *action can be inverted by* $drive(y,x)$*: The only precondition of* $drive(y,x)$ *is* $at = y$*, which must be available if* $drive(x,y)$ *was applied before since it is one of its effects.*

On the other hand, painting the money variable black is not possible. While the causal graph is again acyclic, the money variable is not invertible because its values can only decrease.

In the tractable fragment, a red-black plan can be generated by computing a fully delete-relaxed plan π^+, and running *ACI plan repair* on π^+ to obtain a red-black plan π^{RB}. The repair process executes π^+ step-by-step under the red-black semantics; whenever a

condition (precondition or goal) $v^B = g$ on a black variable $v^B \in \mathcal{V}^B$ is not satisfied, a subsequence π achieving $v^B = g$ is inserted into the plan. Generating such a repair sequence is always possible, in time polynomial in the length of π [Domshlak et al., 2015, Theorem 11]: Since the black causal graph is acyclic, black variables can be moved individually without affecting other black variables; and the invertibility criterion guarantees that the required value $g \in \mathcal{D}_v$ can be reached from the current value of v^B. For our experiments, we adapted the more advanced *red facts following* algorithm for ACI plan repair [Katz and Hoffmann, 2013; Domshlak et al., 2015, Section 5.2], which yields red-black plans with less redundancy than those returned by the basic repair algorithm.

As discussed in Example 3.4, an ACI painting can only make the location of the car a black variable. Yet in order to obtain conflict-free plans, the money variable would also need to be black, but it cannot be handled through ACI. Intuitively, ACI is useful for variables that have to move back and forth (like the location of car in the example), but it typically can not handle situations such as resource consumption or other non-invertible variables.

3.3.2 Red-Black State-Space Search (RBS)

In order to enable convergence to real planning in the limit, we require red-black planning methods that can handle arbitrary paintings. As general red-black planning is **PSPACE**-complete [Domshlak et al., 2015, Theorem 3], this necessitates search. To this end, Gnad et al. [2016] have introduced *red-black state-space search (RBS)*—a hybrid between forward search and delete-relaxed planning: If $\mathcal{V}^B = \mathcal{V}$, then RBS performs search with standard semantics; if $\mathcal{V}^B = \emptyset$, then RBS simplifies to fully delete-relaxed planning. Essentially, RBS performs forward search with a relaxed fixed point over the red variables at each state. When RBS reaches a goal state, it augments the extraction of the solution path with a relaxed plan extraction step at each state transition. We introduce this framework in detail here as we introduce extensions for it in Chapter 6.

We first introduce some notations that we will use to define the underlying transition system of red-black state-space search.

At each red-black state s^{RB}, the *red actions* can be used to compute the relaxed fixed point. The red actions of s^{RB}, denoted $\mathcal{A}^R(s^{RB})$, are actions that comply with the black variable values of s^{RB}, i.e., $\mathcal{A}^R(s^{RB}) := \{a^R \mid a \in \mathcal{A}, \mathsf{pre}(a)|_{\mathcal{V}^B} \subseteq s^{RB}, \mathsf{eff}(a)|_{\mathcal{V}^B} \subseteq s^{RB}\}$, where the preconditions and effects of a^R are those of a projected onto the red variables.

The relaxed fixed point at s^{RB} is now formalized in terms of a local red-black planning task with only red variables, namely the task $\Pi^+(s^{RB}) := \langle \emptyset, \mathcal{V}^R, \mathcal{A}^R(s^{RB}), s^{RB}|_{\mathcal{V}^R}, \emptyset \rangle$. The *red*

completion of s^{RB} is the red-black state $\mathcal{F}^+(s^{RB})$ where $\mathcal{F}^+(s^{RB})|_{\mathcal{V}^B} = s^{RB}|_{\mathcal{V}^B}$, and $\mathcal{F}^+(s^{RB})|_{\mathcal{V}^R}$ is the set of all facts reachable in $\Pi^+(s^{RB})$.

Definition 3.3 (Red-Black State Space)**.** Let $\Pi^{RB} = \langle \mathcal{V}^R, \mathcal{V}^B, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ be a red-black planning task. The *red-black state space* induced by Π^{RB} is the labeled transition system $\Theta^{RB} = \langle \mathcal{S}^{RB}, \mathcal{A}, \mathcal{T}^{RB}, \mathcal{I}, \mathcal{G}^{RB} \rangle$, where

- $\mathcal{S}^{RB}$ is the set of all states of Π^{RB};
- $\mathcal{A}$ are the actions of Π^{RB};
- $\mathcal{T}^{RB}$ is defined by the actions of Π^{RB}: for each red-black state $s^{RB} \in \mathcal{S}^{RB}$ and action $a \in \mathcal{A}$, iff a is applicable to $\mathcal{F}^+(s^{RB})$ and $\mathsf{eff}(a)|_{\mathcal{V}^B} \nsubseteq s^{RB}$, then $s^{RB} \xrightarrow{a} \mathcal{F}^+(s^{RB})[\![a]\!]$ is a transition in $\mathcal{T}^{RB}$;
- $\mathcal{I}$ is the initial state of Π^{RB}; and
- $\mathcal{G}^{RB}$ is the set of all goal states of Π^{RB}.

In the red-black state space, state transitions induced by actions which affect black variables are interleaved with red-variable fixed points.

Example 3.5. *Consider again the task shown in Example 3.3. Assume that we paint the money variable black, i.e., $\mathcal{V}^B = \{money\}$. In the initial state, $at = B$ is reachable by red actions, so it is added to the red completion of $\mathcal{I}$. The applicable actions with effects on black variables are buy$(P_1, 2)$ and buy$(P_2, 2)$. If we apply the action buy$(P_1, 2)$, we reach the red-black state $\{at = \{A, B\}, money = 1, have\text{–}P_1 = 1, have\text{–}P_2 = 0\}$. The red completion of that state does not add any new facts. We can again apply two different buy actions, namely buy$(P_1, 1)$ and buy$(P_2, 1)$: applying buy$(P_2, 1)$ leads to the goal state $\{at = \{A, B\}, money = 0, have\text{–}P_1 = 1, have\text{–}P_2 = 1\}$; applying buy$(P_1, 1)$ leads to a dead end where we are out of money but have failed to obtain both products.*

The red-black state space preserves red-black plans π^{RB} in the sense that the subsequence of actions in π^{RB} that affect black variables labels a solution in Θ^{RB} [Gnad et al., 2016, Theorem 1].

In order to construct the solution, *red-black plan extraction* augments the sequence of transitions taken in the state-space search by the necessary red actions through relaxed plan extraction steps. The red-black plan extraction makes use of a variant of regression projected to red variables: Given a set of red facts g and an action a, the *red regression of g over a* is defined as $R^R(g, a) := (g \setminus \mathsf{eff}(a)|_{\mathcal{V}^R}) \cup \mathsf{pre}(a)|_{\mathcal{V}^R}$. The red regression is overloaded on action sequences by applying it recursively, i.e., the red regression of g over an action sequence $\langle a_1, \dots, a_n \rangle$ is defined as $R^R(g, \langle a_1, \dots, a_n \rangle) := R^R(R^R(g, a_n), \langle a_1, \dots, a_{n-1} \rangle)$.

Let $\pi = \langle a_0, \dots, a_{n-1} \rangle$ be a plan for Θ^{RB}. If π is empty, then a red-black plan is just a relaxed plan using the red actions to achieve the red goals $\mathcal{G}|_{\mathcal{V}^{\mathrm{R}}}$. Otherwise, assume that backward red-black plan extraction has already extracted a red-black plan for the postfix $\pi_k := \langle a_k, \dots, a_{n-1} \rangle$, and assume that the transition taken by a_{k-1} in π is $s^{\mathrm{RB}}_{k-1} \xrightarrow{a_{k-1}} s^{\mathrm{RB}}_k$. Then the *red goal* for relaxed plan extraction at this transition is $G(s^{\mathrm{RB}}_{k-1}) := R^{\mathrm{R}}(\mathcal{G}|_{\mathcal{V}^{\mathrm{R}}}, a_{k-1} \circ \pi_k)$. Intuitively, $G(s^{\mathrm{RB}}_{k-1})$ is the set of red facts that must be achieved in s^{RB}_{k-1} before a_{k-1} can be applied, and that cannot be achieved further along the plan. Any relaxed plan extraction mechanism can now be used to find a relaxed plan $\pi^+(s^{\mathrm{RB}}_{k-1})$ achieving $G(s^{\mathrm{RB}}_{k-1})$. Then π_k is replaced by $\pi^+(s^{\mathrm{RB}}_{k-1}) \circ a_{k-1} \circ \pi_k$, and the red-black plan extraction process continues recursively.

Example 3.6. *Consider again Example 3.5, and consider the sequence of (black) actions $\pi = \langle buy(P_1, 2), buy(P_2, 1) \rangle$ that reaches a goal state in Θ^{RB}. Let $s^{\mathrm{RB}}_0, s^{\mathrm{RB}}_1, s^{\mathrm{RB}}_2$ be the red-black states traversed by π, i.e.,*

- $s^{\mathrm{RB}}_0 = \{at = A, money = 2, have\text{–}P_1 = 0, have\text{–}P_2 = 0\}$,
- $s^{\mathrm{RB}}_1 = \{at = \{A, B\}, money = 1, have\text{–}P_1 = 1, have\text{–}P_2 = 0\}$, *and*
- $s^{\mathrm{RB}}_2 = \{at = \{A, B\}, money = 0, have\text{–}P_1 = 1, have\text{–}P_2 = 1\}$.

The red-black plan extraction starts by processing the empty postfix of the plan, where it must find a relaxed plan to achieve $\mathcal{G}|_{\mathcal{V}^{\mathrm{R}}} = \{at = A, have\text{–}P_1 = 1, have\text{–}P_2 = 1\}$ from the red-black state s^{RB}_2. Since s^{RB}_2 already satisfies all goal facts, the resulting relaxed plan is empty.

Next, the transition $s^{\mathrm{RB}}_1 \xrightarrow{buy(P_2,1)} s^{\mathrm{RB}}_2$ is considered. The red goal at this step is $G(s^{\mathrm{RB}}_1) = R^{\mathrm{R}}(\mathcal{G}|_{\mathcal{V}^{\mathrm{R}}}, buy(P_2, 1)) = \{at = A, at = B, have\text{–}P_2 = 1\}$. These facts are again already true in s^{RB}_1, so no additional actions are inserted into the plan.

Finally, the plan extraction procedure processes the transition $s^{\mathrm{RB}}_0 \xrightarrow{buy(P_1,2)} s^{\mathrm{RB}}_1$, with the red goal $G(s^{\mathrm{RB}}_0) = R^{\mathrm{R}}(\mathcal{G}|_{\mathcal{V}^{\mathrm{R}}}, \langle buy(P_1, 2), buy(P_2, 1) \rangle) = \{at = A, at = B\}$. The fact $at = B$ is missing in s^{RB}_0, so the red action $drive(A, B)$ must be inserted into the plan to achieve it.

Thus, the overall red-black plan returned by the plan extraction procedure is $\langle drive(A, B)$, $buy(P_1, 2), buy(P_2, 1) \rangle$.

The red-black plan extraction procedure guarantees that the resulting plan π^{RB} is indeed a red-black plan for the original task [Gnad et al., 2016, Theorem 2].

Note that the red-black plan constructed in Example 3.6 is correct about the black variable *money*, but has a flaw in the location of the car as it does not return to A. This is

complementary to the tractable fragment ACI in Example 3.4, which can un-relax the "*at*" variable but cannot address flaws in the money variable.

In Chapter 6, we propose a method motivated by this observation, combing red-black state-space search with the tractable fragment ACI to handle each kind of flaw with the most appropriate method. Additionally, we introduce a more flexible variant of red-black state-space search that can use different paintings for different areas of the search space, refining them adaptively when needed.

4 Online Relaxation Refinement for Satisficing Planning

Most planning heuristics can compute their estimates at different levels of precision depending on how the underlying relaxation is instantiated: *Abstraction heuristics* [e.g., Clarke et al., 1994; Culberson and Schaeffer, 1998; Edelkamp, 2001; Helmert et al., 2007; 2014; Seipp and Helmert, 2018] construct an abstract state space, which can range from just a single state (where all heuristic estimates would be zero) to the full state space of the input task (computing the perfect heuristic h^*). *Critical-path heuristics* [Haslum and Geffner, 2000; Haslum, 2006; Fickert et al., 2016] compute their estimates based on the most costly subgoals towards the goal, where considering larger subgoals results in a more accurate heuristic. *Partial delete relaxation heuristics* [Keyder et al., 2014; Domshlak et al., 2015; Fickert et al., 2016] ignore some of the delete effects of the input task, interpolating between the full delete relaxation and non-relaxed semantics.

All of these techniques offer a trade-off between heuristic accuracy and computational complexity. A practical approach to make this decision is based on iterative *refinement* operations: starting from a base abstraction, the abstraction is repeatedly refined until the desired level of precision is reached. Most such heuristics eventually converge to h^* with sufficient refinement operations if it is not made infeasible through technical limitations (e.g., time or memory constraints).

One commonly used strategy to refine a heuristic is *counterexample-guided abstraction refinement* [Clarke et al., 2003], short CEGAR. Starting from a simple abstraction, CEGAR identifies flaws in the current model that are then resolved by making the abstraction more precise. This can be applied iteratively, and typically leads to convergence as eventually the abstract model becomes perfect. The main advantage of the CEGAR approach is that it focuses the refinement on areas where the heuristic is currently flawed, making the refinement procedure more effective. In planning, CEGAR is used as the refinement method for Cartesian abstractions [Seipp and Helmert, 2013; 2018], pattern database heuristics [Rovner et al., 2019], and the partial delete relaxation heuristic $h^{C\text{FF}}$ based on atomic conjunctions [Keyder et al., 2014; Fickert et al., 2016].

Traditionally, the heuristic is refined offline, before starting the search, until some criterion is met, such as hitting a time or memory bound. Yet the most challenging aspects of the planning task at hand may only be discovered during the search, and these difficulties may not be considered when constructing the heuristic offline (this is particularly true for CEGAR approaches which may identify flaws in the heuristic on the current region of the search space). *Online relaxation refinement*, during search, thus is a promising approach. However, it has seen limited success in the literature so far.

In optimal planning, online refinement has been tried using Cartesian abstraction heuristics [Eifler and Fickert, 2018], as their fine-grained CEGAR method is well suited to be applied during search. However, in practice, state-of-the-art variants using offline refinement are still superior due to the added overhead and other practical limitations of these online approaches. A different form of online refinement are per-state heuristic value updates in real-time search [e.g., Korf, 1990; Koenig and Sun, 2009], though this method of refining the heuristic does not generalize to the part of the state space that has not been explored yet as the relaxation underlying the heuristic is not refined. If the search uses an ensemble of heuristics, their combination can be improved via online refinement [Felner et al., 2004; Fink, 2007; Katz and Domshlak, 2010; Karpas et al., 2011; Domshlak et al., 2012; Seipp, 2021], but not with a guarantee of convergence.

Here, we explore online heuristic refinement for satisficing planning. A key issue for online refinement is the question of *when* to refine the relaxation. Intuitively, refinement should be triggered when the heuristic is inaccurate, such as in local minima or plateaus. In satisficing planning, the state of the art is currently dominated by planners using systematic search algorithms such as GBFS or weighted A*. Detecting local minima or plateaus online is difficult in these search algorithms, as the search does not focus on limited areas of the search space at a time. Local search algorithms like hill-climbing seem more suitable for this task as their exploration is constrained to small areas of the search space. Yet such algorithms have fallen out of favor, since they are incomplete (the search can get stuck in dead ends), and have been outperformed by complete search algorithms very broadly and consistently for more than a decade.

In this chapter, we introduce changes fundamentally altering the properties and competitiveness of local search in this context. We introduce multiple search algorithms that are designed for online refinement, in particular a family of hill-climbing algorithms that we call Refinement-HC. Similar to FF's enforced hill-climbing [Hoffmann and Nebel, 2001], Refinement-HC explores the local search space around the current state, but with the addition of a bound on the local search. If the local search space does not contain a state with lower heuristic value than that of the root state s of the local exploration, then s must be a local minimum. Instead of trying to escape s through brute-force search (as enforced

hill-climbing would do), Refinement-HC aims to *remove the local minimum from the search space surface* by refining the heuristic. If the refinement operation of the heuristic satisfies a suitable convergence criterion, Refinement-HC is a *complete* search algorithm, thus fixing the major theoretical weakness of local search in satisficing planning.

The simplest variant of Refinement-HC uses a depth bound to limit the local search. This bound controls the trade-off between search and refinement: Smaller bounds shift the focus towards refinement, while larger bounds give the search more time to find a better state. We devise a more effective approach leveraging novelty pruning [Lipovetzky and Geffner, 2012; 2014] instead of a simple depth bound. This form of local search discards states that do not contain facts that have not yet been seen in the current lookahead. We further combine this technique with subgoal counting [Lipovetzky and Geffner, 2017], which can be used as a simple and computationally efficient approximation for delete relaxation heuristics, reducing the overhead of the local explorations.

In addition to our hill-climbing algorithms, we introduce an extension of GBFS with online refinement. This variant of GBFS repeatedly performs bounded lookahead searches based on Refinement-HC, trying to find a state with strictly better heuristic value. If the lookahead search succeeds, it allows the GBFS search to quickly jump towards the goal; otherwise refinement is triggered to improve the heuristic. This variant of GBFS is not just useful for online refinement, but can also be used in combination with other heuristics to boost search progress.

We instantiate our online-refinement search algorithms with the h^{CFF} heuristic and evaluate them on the International Planning Competition (IPC) benchmarks as well as on the Autoscale benchmarks [Torralba et al., 2021], which are designed to make performance differences of recent planners more visible compared to the IPC instances of the same domains. Our online-refinement methods yield substantial improvements over comparable baselines and state-of-the-art planners such as LAMA [Richter and Westphal, 2010], MERWIN [Katz et al., 2018], and Dual-BFWS [Francès et al., 2018; Lipovetzky and Geffner, 2017], and are competitive even with complex state-of-the-art portfolios. On the Autoscale benchmarks the advantage increases further, e.g., beating the portfolio planner and winner of the IPC'18 satisficing track Fast Downward Stone Soup [Seipp and Röger, 2018; Helmert et al., 2011] by more than 90 (out of 780) solved instances.

To summarize, our contributions are:

- A family of hill-climbing search algorithms called Refinement-HC that resolve local minima through online refinement of the heuristic function instead of brute-force search. We prove that Refinement-HC is complete if the refinement operation of the heuristic meets a suitable convergence criterion.

- A variant of greedy best-first search augmented with local lookahead searches that can be used both with and without online refinement.
- Extensive experiments on both the IPC and the Autoscale benchmarks, evaluating different configurations of our algorithms, and demonstrating their advantages over related baselines as well as state-of-the-art planners.

This chapter is structured as follows. First, we give a brief summary of the existing techniques that we use in our algorithms (Section 4.1), and discuss the common setup of the experiments in this chapter (Section 4.2). In Section 4.3, we give a formal description on heuristic refinement operations, and describe the convergence property that is required to make our hill-climbing search algorithms complete. We introduce our online-refinement hill-climbing search algorithm Refinement-HC in Section 4.4, and show how to extend it with novelty pruning (Section 4.5) and subgoal counting (Section 4.6). In Section 4.7, we show how the ideas behind our hill-climbing algorithms can be transferred to GBFS. We empirically compare our online-refinement search algorithms to related baselines and to the state of the art in Section 4.8. Finally, we give a more detailed discussion of related work (Section 4.9) before concluding this chapter in Section 4.10.

Papers and Contributions This chapter is based on the paper "Online Relaxation Refinement for Satisficing Planning: On Partial Delete Relaxation, Complete Hill-Climbing, and Novelty Pruning" [Fickert and Hoffmann, 2022], and transitively on "Complete Local Search: Boosting Hill-Climbing through Online Relaxation Refinement" [Fickert and Hoffmann, 2017a], "Making Hill-Climbing Great Again through Online Relaxation Refinement and Novelty Pruning" [Fickert, 2018], and "A Novel Lookahead Strategy for Delete Relaxation Heuristics in Greedy Best-First Search" [Fickert, 2020]. All papers were principally developed by the author. Episode-EHC and the depth-bounded Refinement-HC variant (Section 4.4) were already included in the author's Master's thesis [Fickert, 2016].

4.1 Background: Techniques We Build On

Our online-refinement search algorithms leverage novelty pruning and subgoal counting for the local exploration component. We summarize these techniques in the following.

4.1.1 Novelty Pruning

Novelty is a concept to capture the similarity of a given state compared to a set of states that have been seen before. Formally, given a set of states seen so far $\mathcal{T}$, the *novelty* of a state s is the size of the smallest tuple of facts t such that $t \subseteq s$ and $t \nsubseteq s'$ for all $s' \in \mathcal{T}$. In its simplest form, novelty can be used as a pruning function in a search, discarding all states that are not sufficiently novel: A search with *k-novelty pruning*, which we denote by $\mathcal{N}_k$, prunes all states with novelty greater than k. This kind of pruning is used in *Iterated Width Search* (IW) [Lipovetzky and Geffner, 2012], where each iteration IW(k) is a simple breadth-first search with k-novelty pruning. IW(k) expands at most $|\mathcal{F}|^k$ states, and is incomplete unless $k = |\mathcal{F}|$, where $\mathcal{N}_k$ only prunes duplicate states. Novelty relates to the theoretical notion of *width* in that IW(w) is guaranteed to find a solution for tasks of width at most w.

More recently, novelty measures have been introduced that take the heuristic into consideration, comparing the novelty of a given state only to previously seen states with equal [Lipovetzky and Geffner, 2017] or lower [Katz et al., 2017] heuristic value. Best-First Width Search (BFWS) [Lipovetzky and Geffner, 2017] is a best-first search which uses a novelty measure as the main evaluation function to guide the search.

4.1.2 Subgoal Counting

The best-performing BFWS configuration uses the novelty measure $w_{\#g,\#r}$ as the main search guidance (breaking ties by $\#g$), where, for a state s,

- $\#g(s)$ is the number of unsatisfied goal facts in s, and
- $\#r(s)$ is the number of achieved subgoals of the last relaxed plan π^+ along the path to s from the state in which π^+ was computed [Lipovetzky and Geffner, 2014].

Relaxed plans are only computed in states s where $\#g(s)$ is different from its parent (and in the initial state). The path-dependent counter $\#r$ then keeps track of the relaxed plan's subgoals that are achieved below such a state s. Combined with the fact that the main evaluation function is based on simple counters, the infrequent computation of the main heuristic makes BFWS extremely lightweight, which is one of the main reasons for its success at the 2018 IPC [Francès et al., 2018].

In this chapter, we introduce several search algorithms that make use of the idea to use subgoal counting (similar to the $\#r$ counter) as an approximation of a relaxation heuristic. Given a plan $\pi[h](s)$, we denote the *subgoal-counting heuristic* for that plan

by $h^{SC}[\pi[h](s)]$ (or short h^{SC} where the underlying plan is not relevant or is clear from context). In the context of h^{SC}, a *subgoal* is a fact that appears as an effect in one of the actions of the plan underlying h^{SC}, and is either a goal or a precondition for another action in the plan. For a state s', the heuristic value $h^{SC}[\pi[h](s)](s')$ is the number of subgoals that are not true in at least one state along the path from s to s'.

The main difference of h^{SC} compared to the $\#r$ counter is that h^{SC} counts in the opposite direction, making it consistent with other heuristics where lower values indicate that the state is closer to the goal (this was not a concern in BFWS where $\#r$ was only used in a novelty measure, not as a heuristic). Additionally, we also make a slight technical adjustment in our definition of subgoals for h^{SC}: While $\#r$ considers *all* effects of the actions in the underlying plan as subgoals, we only consider the *necessary* ones. This change more accurately captures the "intention" of the underlying plan, and in preliminary experiments we found that it improves the performance of our search algorithms introduced here as well as that of BFWS (though to a lesser degree).

4.2 Experiments Setup

Our online-refinement search algorithms are generally independent of the used heuristic, given that it offers a refinement operation with the convergence properties discussed in the following section. However, in our experiments, we use the $h^{C\mathrm{FF}}$ heuristic, and the required convergence properties have been derived from those of $h^{C\mathrm{FF}}$. We use Keyder et al.'s refinement method for $h^{C\mathrm{FF}}$, with the minor change of extracting the conflicts from the sequentialized plan as returned by $h^{C\mathrm{FF}}$ instead of any valid ordering (cf. Section 5.3).

We evaluate our algorithms on all unique STRIPS instances from the satisficing tracks of the International Planning Competition (IPC) domains up to 2018, which yields a total of 1695 instances from 48 domains. All experiments with hill-climbing search algorithms use helpful actions pruning, and all GBFS configurations (including our GBFS extension introduced in Section 4.7) use a dual queue for preferred operators. When comparing our methods to the state of the art, we additionally present results on the Autoscale benchmarks (Section 4.8.3).

Several of the algorithms we experiment with ($h^{C\mathrm{FF}}$, hill-climbing) use randomness to break ties. In these cases, we average the results over 5 runs with different random seeds.

The source code of our implementations of $h^{C\mathrm{FF}}$ and the online-refinement algorithms is available at `https://github.com/fickert/fast-downward-conjunctions`.

4.3 Converging Heuristic Functions

Given a heuristic h and a refinement operation ρ, the strongest possible convergence property would be to have $h = h^*$ after a finite number of applications of ρ. However, this property is not always practical, and the h^{CFF} heuristic in particular does not satisfy it. We therefore identify a slightly weaker convergence property that suffices to make our online-refinement hill-climbing algorithms complete. We start our discussion based on h^{CFF}, and then give a more general definition.

As pointed out in Section 3.2, the h^{C+} heuristic satisfies the strong convergence to h^*. The h^{CFF} heuristic on the other hand does not, because the C-relaxed plans are not optimal, so the resulting heuristic value may be an overestimation of the actual goal distance. However, there exists a set of conjunctions C such that h^{CFF} agrees with h^* on states s where $h^*(s) = \infty$, and its partially relaxed plans become real plans on solvable states.

More precisely, let $C_* := \mathcal{P}(\mathcal{F})$ be the maximal set of conjunctions, considering *all* combinations of facts of a given task with facts $\mathcal{F}$. With $C = C_*$, the h^{CFF} heuristic is converged, and we can prove the aforementioned property:

Proposition 4.1. *Let $\Pi = \langle \mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ be a planning task, and let s be a state of Π. Then there exists a set of conjunctions C such that (a) in case s is unsolvable, we have $h^{CFF}(s) = \infty$; and (b) in case s is solvable, $\pi[h^{CFF}](s)$ is a plan for s. In particular, both (a) and (b) hold for $C = C_*$.*

Proof. We prove (a) and (b) for $C = C_* = \mathcal{P}(\mathcal{F})$.

For (a): By Fickert et al.'s [2016] Corollary 1, $h^{CFF}(s) = \infty$ iff $h^{C+}(s) = \infty$, and there exists C such that $h^{C+}(s) = h^*(s)$. As $h^{C_*+} \geq h^{C+}$ for any C, this shows the claim.

For (b): As s is solvable, $h^*(s) \neq \infty$, so $h^{C_*+}(s) \neq \infty$ and $h^{C_*\mathrm{FF}}(s) \neq \infty$. Thus, we can run C-refinement on s. Assume that $\pi[h^{CFF}](s)$ is not a plan for s. Then, by Keyder et al.'s [2014] Lemma 3, C-refinement on s generates an atomic conjunction $c \notin C_*$, in contradiction. □

Since each refinement operation on h^{CFF} adds a conjunction to C, eventually the heuristic converges as C grows towards C_*. Proposition 4.1 shows that, with repeated refinement, h^{CFF} eventually detects all dead ends, and computes real plans for all other states. This is exactly the convergence property that is required for completeness in our online-refinement hill-climbing search algorithms.

Formally, we define a *refinement operation* as a function ρ that maps a heuristic h to a modified heuristic $\rho[h]$, where ρ is again applicable to its output. This function may

possibly require a state s as a secondary input where $h(s) \neq \infty$ and the relaxed solution $\pi[h](s)$ is not a real plan (this is the case for the refinement operation of h^{CFF}). This allows us to define convergence as follows:

Definition 4.2 (Converging Heuristic). Let $\mathcal{S}$ be the set of states of a planning task Π, and let h be a heuristic function with relaxed solutions $\pi[h]$, and let ρ be a refinement operation for h. The heuristic h *converges with* ρ if there exists $N \in \mathbb{N}_0$ such that, for all states $s \in \mathcal{S}$,

(C1) if $h^*(s) = \infty$, then $\rho^N[h](s) = \infty$, and

(C2) otherwise $\pi[\rho^N[h]](s)$ is a plan for s.

As discussed, $h^{\mathcal{C}\text{FF}}$ converges with Keyder et al.'s [2014] refinement method. Another example for converging heuristics are abstraction heuristics (as long as their abstract state space can be refined to the real state space, allowing them to converge to h^*). This is particularly true for Cartesian abstractions, which converge with a CEGAR-based refinement operation similar to $h^{\mathcal{C}\text{FF}}$. However, in practice, combining multiple smaller Cartesian abstractions (that are constrained to a subproblem of the input task) via cost partitionings is the most effective approach [Seipp and Helmert, 2014; 2018; Seipp et al., 2020], yet convergence is only guaranteed if abstractions are merged (which is expensive) or only a single abstraction is used [Eifler and Fickert, 2018]. In principle though, any such heuristic that converges according to Definition 4.2 is sufficient to guarantee completeness of our hill-climbing algorithms introduced in the following.

4.4 Online-Refinement Hill-Climbing

We introduce a family of hill-climbing-style local search algorithms with the underlying idea of escaping local minima by *refining the heuristic* instead of brute-force search. *Standard hill-climbing (HC)* performs a simple gradient descent, selecting the action that leads to the immediate successor with lowest h value at each step until it reaches a state s with $h(s) = 0$. The FF planner [Hoffmann and Nebel, 2001] introduced *enforced hill-climbing (EHC)*, which replaces this strategy with a complete lookahead at each step: From the current state s, it runs breadth-first search (BrFS) until it finds a state s' with $h(s') < h(s)$. The lookahead strategy of our algorithms lies between those extremes: We consider more than just the immediate successors, but add a bound to the lookahead search. This bound can be defined by a lookahead horizon k (i.e., maximum search depth). If the lookahead from s does not yield a state s' with $h(s') < h(s)$, then s is a local minimum of depth k

under h. In that case, the search algorithm will attempt to raise $h(s)$ through heuristic refinement until the local minimum is removed from the search space surface.

We first describe an intermediate algorithm called *Episode-EHC*, which augments EHC with restarts and a global dead-end cache. Based on Episode-EHC, we introduce our local search algorithm with online heuristic refinement which we call *Refinement-HC*.

4.4.1 Episode-EHC

The FF planner uses EHC with helpful actions pruning as an incomplete first search phase, switching to GBFS in case of failure. Without helpful actions pruning, EHC can still fail to find a solution by walking into a dead end that is not recognized by the heuristic. In Episode-EHC, we handle this situation by globally marking the state as a dead end, and then restarting from the initial state. While Episode-EHC is merely an intermediate step towards our online-refinement search algorithms, we include it to show that the addition of a global dead end cache is already sufficient to achieve completeness (without helpful actions pruning).

Algorithm 1: Episode-EHC

```
1  C_de := ∅ // cross-episode dead-end cache
2  s := I
3  while I ∉ C_de do
4      Run BrFS (pruning states in C_de) from s for a state s′ with h(s′) < h(s) or s′ ⊇ G
5      if no such s′ exists then
           // mark s as a dead end and start a new episode
6          C_de := C_de ∪ {s}
7          s := I
8      else
9          s := s′
10         if s ⊇ G then
11             return SOLVED
12 return UNSOLVABLE
```

Algorithm 1 shows the pseudocode for Episode-EHC. Essentially, it adds a restart mechanism to the standard EHC procedure. While EHC gives up in case it cannot find a better state in the BrFS phase, Episode-EHC instead marks the root state of the lookahead as a dead end and starts a new EHC *episode* by resetting the search to the initial state (Lines 5–7). The global dead end cache ensures that the search continually makes progress: Either an EHC episode succeeds by finding a solution, or it adds a new state to the dead-end cache, pruning it in subsequent iterations.

Proposition 4.3. *Episode-EHC is a complete search algorithm.*

Proof. Every EHC episode adds at least one new state into $\mathcal{C}_{de}$. After at most N episodes, where N is the number of dead-end states, $\mathcal{C}_{de}$ contains all dead-ends. Hence, the EHC episode $N+1$ will either fail directly as $\mathcal{I} \in \mathcal{C}_{de}$, or will find a plan. □

Note that using helpful actions pruning in Episode-EHC will break completeness: Since the lookahead search space is not guaranteed to be fully explored (because successors via non-helpful actions are pruned), the root state of the lookahead is not necessarily a dead end and can not be safely added to $\mathcal{C}_{de}$.

4.4.2 Refinement-HC

Based on Episode-EHC, we can now introduce Refinement-HC (short RHC). The key extensions of Refinement-HC over (Episode-)EHC are *bounding the lookahead search*, and handling the lookahead failure by *refining the heuristic*. We assume that the heuristic h is (1) based on abstract plans $\pi[h]$, and (2) has a refinement operation ρ, which is applicable in a state s if $h(s) \neq \infty$ and $\pi[h](s)$ is not a real plan for s. We will show that Refinement-HC is complete even if the lookahead itself is not (Section 4.4.3). Hence, the discussion below assumes that the lookahead may use helpful actions pruning.

The pseudocode for Refinement-HC is shown in Algorithm 2. The lookahead search depth is bounded by a parameter k (denoted by BrFS[k]; line 3). Like in (Episode-)EHC, if the lookahead from the current state s succeeds in finding a state s' with $h(s') < h(s)$ within that horizon, the search proceeds to that state for the next lookahead iteration (line 15). However, if the lookahead does not yield such a state, then the heuristic is refined in s (Lines 9 to 14). The refinement proceeds until $h(s)$ is raised above the minimal heuristic value seen in the lookahead, which aims to ensure that the next lookahead search succeeds in finding a better state. In contrast to Episode-EHC, Refinement-HC does not need the cross-iteration dead-end cache. Instead, progress is guaranteed through converging refinement operations, leading to completeness even with helpful actions pruning (see the next subsection).

Note that the lookahead horizon k defines a trade-off between search and refinement. For smaller values of k, most progress is made by refining the heuristic (with the extreme at $k=1$ which is similar to standard HC with the addition of heuristic refinement). On the other hand, a larger horizon relaxes the requirement for refinement, thus giving the search more opportunity to find a better state without frequent refinement operations; until at the extreme end of $k=\infty$, the search is similar to EHC, triggering heuristic refinement only if the entire search space below s is exhausted unsuccessfully. Intermediate values of k allow the refinement to focus on regions of the search space where the

Algorithm 2: Refinement-HC (RHC)

```
1  s := I
2  while h(I) ≠ ∞ do
3      Run BrFS[k] from s for a state s' with h(s') < h(s) or s' ⊇ G
4      if no such s' exists then // lookahead failed
5          if the lookahead search space was exhausted before reaching the depth bound then
               // s is likely a dead end
6              HANDLE_EXHAUSTION
7          if the previous lookahead iteration also originated at s and s ≠ I then
               // previous refinement was unsuccessful
8              HANDLE_STAGNATION
           // raise h(s) to resolve the local minimum
9          Let h_min be the minimal h value observed in the current lookahead
10         while h(s) ≤ h_min do
11             REFINE_HEURISTIC
12             if h(s) = ∞ then
13                 HANDLE_DEAD_END
14                 break
15     else
16         s := s'
17         if s ⊇ G then
18             return SOLVED
19 return UNSOLVABLE
20 macro REFINE_HEURISTIC
21     if π[h](s) is a plan for s then
22         return SOLVED
23     refine h on s, i.e., replace h with ρ[h]
```

heuristic is poor (deep local minima), leaving more shallow local minima to be escaped via search.

The Refinement-HC pseudocode contains several *macros* (typeset in UPPERCASE)—in contrast to subprocedures, any contained control-flow statements (like **break**, **continue**, or **return**) refer to the position where the macro is inserted instead of the macro itself. The REFINE_HEURISTIC macro applies one refinement step to the heuristic (line 20). We first check whether the underlying relaxed plan is a real plan. If that is the case, we can terminate the search, and return the relaxed plan appended to the path to s as the solution. Otherwise, the preconditions for the refinement operation are satisfied, and we can update the heuristic.

The HANDLE_* macros are called in specific situations that offer some freedom in our search algorithm design. Before discussing the specific macros and the possible options

Algorithm 3: Backjump

input : a state s, a function $\lambda : \mathcal{S} \mapsto bool$ describing the break condition
output: the first state s' when chaining back towards $\mathcal{I}$ where $\lambda(s') = true$, or $\mathcal{I}$

```
1 while s ≠ I do
2     s := the predecessor of s (along the path from I to s)
3     if λ(s) then
4         break
5 return s
```

for each scenario, we introduce the *Backjump* function (see Algorithm 3). The function is given a state s and a Boolean function λ as inputs, and chains backwards until it reaches a state s' where $\lambda(s') = true$, which it returns (or $\mathcal{I}$ if no state along the path satisfies λ).

Algorithm 4: Handle Lookahead Search Space Exhaustion

```
1  macro HANDLE_EXHAUSTION
2      switch Exhaustion do
3          case ExhaustionContinue do
4              pass // do nothing
5          case ExhaustionRestart do
6              REFINE_HEURISTIC
7              s := I
8              continue
9          case ExhaustionBackjump do
10             REFINE_HEURISTIC
11             s := Backjump(s, λ(s) ↦
                BrFS[k] from s does not exhaust its search space before reaching the bound)
12             continue
```

First, the HANDLE_EXHAUSTION macro is called in case the search space is exhausted without reaching the depth bound (Algorithm 2, line 6). Since we assume that the search uses helpful actions pruning, this does not guarantee that the root state of the lookahead is a dead end. However, it can still be an indication that the state is likely a dead end and should be avoided. Algorithm 4 shows the different options that we consider: We can simply ignore this case and proceed as normal (*ExhaustionContinue*), restart from the initial state (*ExhaustionRestart*), or jump back to a state where the lookahead search space does not exhaust (*ExhaustionBackjump*). The backjump option performs a full lookahead search as in a normal iteration of Refinement-HC, but prunes the states which the backjump procedure has chained back from. This ensures that the search does not immediately move back into the state from which we want to escape. Note that we need to do one iteration of refinement when using either *ExhaustionRestart* or *ExhaustionBackjump*:

Otherwise, the search could eventually end up in the same state in which the exhaustion case was triggered, causing an infinite loop.

Algorithm 5: Handle Refinement Stagnation

```
1  macro HANDLE_STAGNATION
2    switch Stagnation do
3      case StagnationContinue do
4        pass // do nothing
5      case StagnationRestart do
6        s := I
7        continue
8      case StagnationBackjump do
9        s := Backjump(s, λ(s) ↦
          BrFS[k] from s yields a state s′ with h(s′) < h(s) or s′ ⊇ G)
10       continue
```

If the lookahead from s fails to find a better state s', the heuristic is refined until $h(s)$ increases over h_{min} (the lowest h value observed in the lookahead). Note that h_{min} is not recomputed during the iterative refinement, i.e., in the next lookahead after the refinement phase h_{min} might have increased as well, which would trigger another refinement phase with the same root state. It might be useful to treat this case separately (Algorithm 2, line 8), and we consider several options via the HANDLE_STAGNATION macro (see Algorithm 5). Similar to the lookahead search space exhaustion case, we can opt to not do any special handling (*StagnationContinue*), restart from the initial state (*StagnationRestart*), or go back to the last state where the lookahead would succeed. Like with *ExhaustionBackjump*, the lookahead search in a backjump phase prunes states from which the backjump procedure is chaining back.

Algorithm 6: Handle Dead End

```
1  macro HANDLE_DEAD_END
2    switch DeadEnd do
3      case DeadEndRestart do
4        s := I
5      case DeadEndBackjump do
6        s := Backjump(s, λ(s) ↦ h(s) ≠ ∞)
```

Finally, it might happen that the heuristic recognizes s as a dead end after a refinement step (Algorithm 2, line 13). In that case, it does not make sense to start the next lookahead iteration from s, and instead we consider either restarting or going back along the current path to the most recent state that is not a dead end for the HANDLE_DEAD_END macro (Algorithm 6).

4.4.3 Completeness

Like Episode-EHC, Refinement-HC can be understood as a series of EHC episodes where new episodes are started by changing the root state of the next lookahead iteration via the Backjump or Restart options of the HANDLE_* macros. Observe that in each such episode, the search will either eventually reach a goal state or refine the heuristic at least once: HANDLE_EXHAUSTION explicitly invokes the refinement procedure before starting a new episode, stagnation can only happen if the heuristic was refined in the previous lookahead iteration (in that same episode), and HANDLE_DEAD_END is only invoked after the refinement step. If the heuristic converges with the refinement procedure according to Definition 4.2, then the search must eventually reach a goal or refine the heuristic to convergence, making Refinement-HC complete.

Theorem 4.4. *Given a heuristic h converging with ρ, Refinement-HC is a complete search algorithm.*

Proof. Observe that every (unsuccessful) episode refines h at least once, and that this sequence of refinements stops only if either (a) a plan is found, or (b) $h(\mathcal{I}) = \infty$. Say the input task Π is unsolvable. Then (a) never happens, and termination on (b) is an unsolvability proof as desired. Unless termination on (b) happens earlier, h will eventually converge. At this point, by Definition 4.2 (C1) we have $h(s) = \infty$ for all unsolvable s (including $\mathcal{I}$), leading to termination on (b) (Algorithm 2, line 19).

Say now that Π is solvable. Then (b) never happens, and (a) is the desired termination. Unless that termination happens earlier, h will eventually converge, at which point $h(s) = \infty$ for all unsolvable s, and $\pi[h](s)$ is a plan for all solvable s by Definition 4.2 (C2). If, at that point, s is a dead end ($h(s) = \infty$), the search will start a new episode from a solvable state through HANDLE_DEAD_END (line 13). In that episode, the search will eventually reach a goal state (line 18) or call REFINE_HEURISTIC, where it terminates on (a) since $\pi[h](s)$ is a plan for s (line 22). □

Note that the completeness proof holds for *all 18 possible instantiations* of the HANDLE_* macros, i.e., an entire family of search algorithms. In fact, Refinement-HC does not even depend on using depth-bounded breadth-first search as the lookahead search algorithm—*any* incomplete lookahead search algorithm will do (even one that would always fail immediately), as completeness is guaranteed through the converging heuristic.

For unsolvable tasks, completeness relies only on convergence property (C1) ($h(\mathcal{I}) = \infty$). For solvable tasks, if the heuristic were to converge to h^*, then each BrFS[k] lookahead iteration from s would always succeed in finding a state s' with $h(s') < h(s)$ until a goal

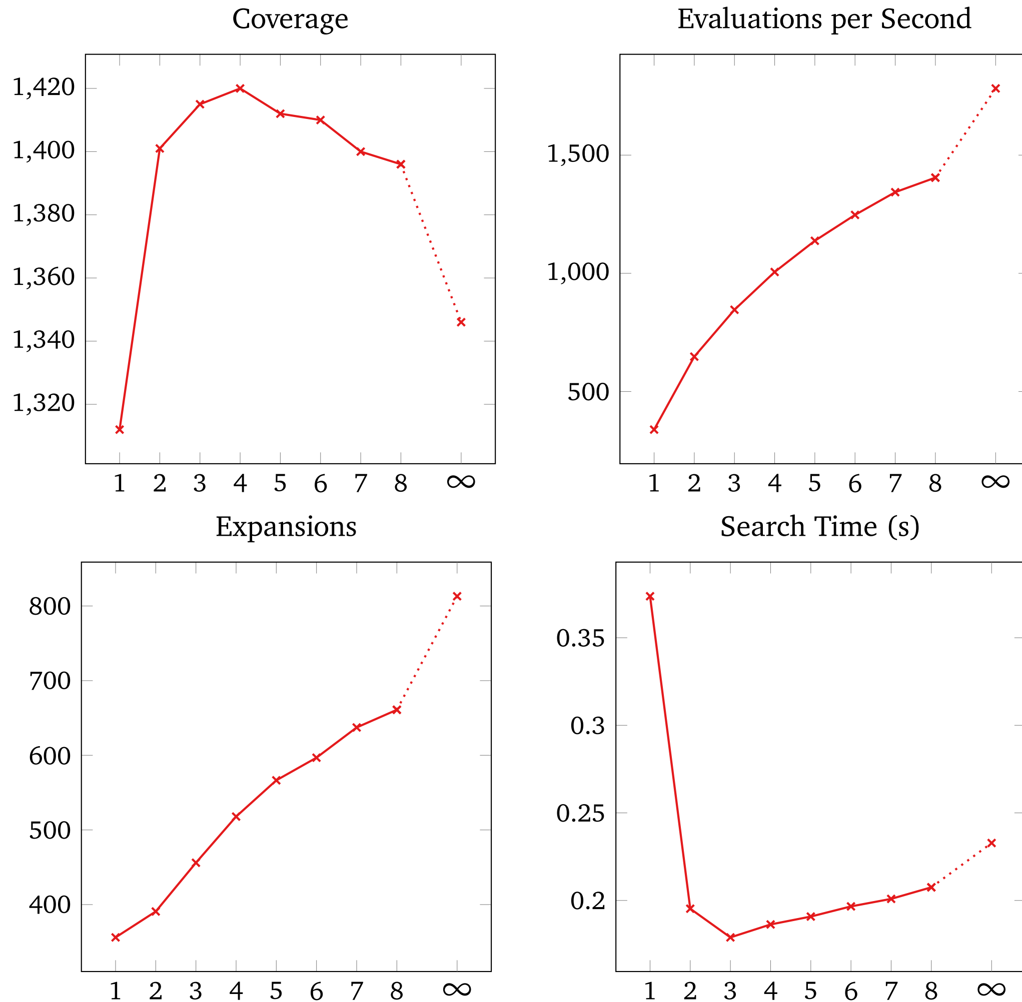


FIGURE 4.1: Results for Refinement-HC with varying depth bounds (x-axis): total coverage, geometric mean of the heuristic evaluations per second, and geometric means across commonly solved instances of the number of expansions and search time.

state is reached. Our weaker convergence property (C2) suffices since $\pi[h](s)$ will be a plan for s in case no better state is found during the lookahead and the refinement is triggered.

4.4.4 Experiments

Our experiments in this subsection focus on the heuristic refinement vs. search trade-off from varying the depth bound parameter in Refinement-HC. The HANDLE_* macros are instantiated with *DeadEndRestart*, *StagnationBackjump*, and *ExhaustionRestart* (we evaluate all possible instantiations in Section 4.5.3.2).

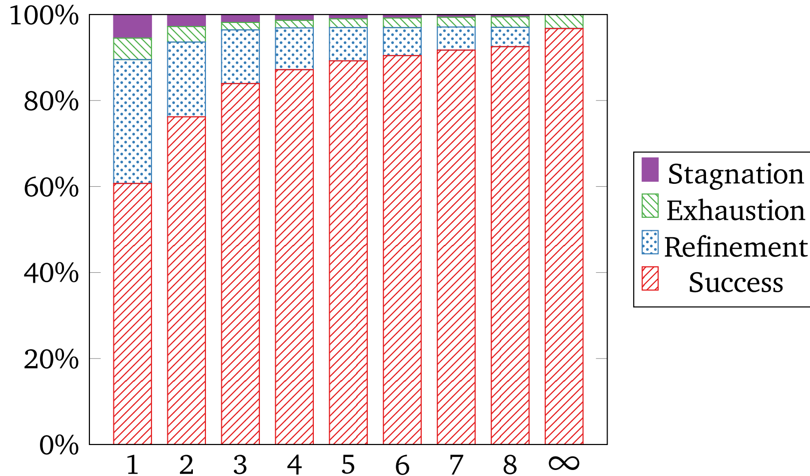


FIGURE 4.2: Lookahead result for Refinement-HC with varying depth bounds (x-axis).

Figure 4.1 highlights key statistics for Refinement-HC on the IPC benchmarks with depth bounds ranging from 1 to 8 and ∞. As expected, with lower depth bounds, the heuristic is more accurate as refinement is triggered frequently, and the search needs fewer expansions overall. On the other hand, frequent refinement makes the heuristic more expensive to compute, and choosing larger depth bounds allows the heuristic to remain computationally efficient. This trade-off has its sweet spot at a depth bound of 4, where Refinement-HC reaches its peak coverage of 1413.8 (with a standard error of 5.89).

This pattern is consistent across most domains (though the exact sweet spot may vary slightly). One notable exception is Sokoban, where the average coverage increases from 2.2 with a bound of 1 to 11.2 when setting the bound to infinity, and to a lesser degree on Freecell (where coverage increases from 71.6 to 79.6). Conversely, smaller bounds work best on Transport, where coverage decreases with growing bounds (from 56.2 down to 26.8); similarly, on TPP and Woodworking the coverage is mostly unaffected, but search time consistently increases with larger depth bounds. These domains correspond to cases where the conjunctions are generally useful for h^{CFF} (Transport, TPP, Woodworking) vs. cases where h^{FF} works better (Sokoban, Freecell); Section 4.8.1 discusses this in more detail.

Figure 4.2 shows the distribution of the lookahead results to give further insight into the search behavior of Refinement-HC for different depth bounds. Recall that each lookahead in Refinement-HC can have four distinct results: (1) it succeeds in finding a state with lower heuristic value, (2) the heuristic is refined, (3) the HANDLE_EXHAUSTION case is triggered, or (4) the HANDLE_STAGNATION case is triggered. With small depth bounds, the heuristic refined much more frequently (after 28.8% of lookaheads for a depth bound of 1), while larger bounds let the lookahead search run longer, enabling it to find a better state more often.

4.5 Refinement-HC with Novelty Pruning

We next show that the hill-climbing methods just introduced can be synergistically combined with novelty pruning. We first discuss how to replace the depth bound in the Refinement-HC lookahead with novelty pruning and why this is a good idea; then we introduce a generalization of novelty pruning over arbitrary conjunction sets C and point out the possible synergy with online refinement of h^{CFF}; finally, we evaluate these techniques experimentally as before.

4.5.1 Replacing the Depth Bound with Novelty Pruning

As pointed out in Section 4.4.3, Refinement-HC is complete irrespective of the specific lookahead search algorithms used. Depth-bounded BrFS seems like an obvious choice since it is a minimal change from the traditional EHC lookahead, and it yields a good intuition for heuristic refinement: the search is stuck in a local minimum or plateau of the given depth, so refining the heuristic can make that search region easier to navigate. Varying the depth bound allows some trade-off between prioritizing progress via search vs. heuristic refinement. However, a simple depth bound on the lookahead ignores the structure of the local space: It might be useful to explore certain regions in more depth, while others could be abandoned early. Additionally, BrFS does not use a heuristic for more effective guidance (though the expansion order becomes less important if the lookahead uses a simple depth bound).

A more practical choice for bounding the lookahead is to use *incomplete novelty pruning*. Using novelty pruning instead of restricting search depth still effectively bounds the lookahead search, as there is only a finite number of novel states (e.g., at most $|\mathcal{F}|$ states for simple 1-novelty pruning). With novelty pruning, regions of the local search space that do not contain novel facts are avoided, whereas branches with states that do pass the novelty test can be explored in more depth.

Essentially, our Refinement-HC variants with novelty pruning replace BrFS$[k]$ by a search algorithm with incomplete novelty pruning like IW(k). The only other notable change is that `HANDLE_EXHAUSTION` is invoked if the lookahead search space was exhausted without pruning a state due to novelty instead of search depth. Note that when using novelty pruning instead of a depth bound, the expansion order of the lookahead becomes important; not just to potentially find a state with lower heuristic value more quickly, but it also changes which states are novel. In our experiments, we consider best-first search with the open list orderings g, $g+h$, and h, i.e., BrFS, A*, and GBFS.

The novelty in the lookahead search is only evaluated locally (i.e., only within a single lookahead iteration, not across the overall search), since its main goal is to bound each lookahead search, and not to apply aggressive cross-iteration pruning. Thus, we only exchange the local search algorithm of the lookahead, retaining completeness of Refinement-HC as discussed in Section 4.4.3.[1]

4.5.2 Novelty Pruning over Conjunctions

IW(k) applies k-novelty pruning ($\mathcal{N}_k$), pruning all states that do not contain a novel fact tuple of size at most k. We can generalize the definition of $\mathcal{N}_k$ to consider arbitrary conjunctions instead of fixed-size tuples:

Definition 4.5 (C-Novelty Pruning)**.** Given a set of conjunctions C and a set of states seen so far $\mathcal{T}$, a search with *C-novelty pruning* (denoted $\mathcal{N}_C$) prunes a state s if there does not exist a conjunction $c \in C$ such that $c \subseteq s$ and $c \nsubseteq s'$ for all $s' \in \mathcal{T}$.

This idea of generalizing novelty pruning to arbitrary conjunctions was previously mentioned (e.g., in the conclusion of Katz et al.'s [2017] work on novelty heuristics), but has not been explored before. A key problem is how to effectively generate a set C of conjunctions specifically suited for novelty pruning. We do not provide an answer to that question here, however, we realize the obvious synergy with partial delete relaxation methods selecting such a set of conjunctions. Running Refinement-HC with $h^{C\text{FF}}$, we can simply re-use the set of conjunctions from $h^{C\text{FF}}$ for novelty pruning as well.

Using $\mathcal{N}_C$ in the lookahead search for Refinement-HC with $h^{C\text{FF}}$ has an interesting synergistic side effect. The $h^{C\text{FF}}$ heuristic becomes more costly to compute with each added conjunction, so refinement should be applied carefully. On the other hand, the pruning provided by $\mathcal{N}_C$ becomes less aggressive as C grows. Thus, as more conjunctions are added to C, the lookahead can expand more states before refinement is triggered, reducing the overhead for $h^{C\text{FF}}$, and gradually shifting the trade-off between search and refinement towards the former.

4.5.3 Experiments

We next compare Refinement-HC with different types of novelty pruning to the depth-bounded variant, and then compare different expansion orders in the lookahead with

[1] Note that even with global novelty pruning Refinement-HC would still be complete due to the refinement of the heuristic, though extremely ineffective in practice as it would usually require convergence on the initial state as the search eventually runs out of novel states.

Lookahead Search	BrFS[4]	BrFS[$\mathcal{N}_1$]	BrFS[$\mathcal{N}_C$]	BrFS[$\mathcal{N}_2$]	A*[$\mathcal{N}_C$]	GBFS[$\mathcal{N}_C$]
Agricola (20)	7.8	10.2	10.0	**11.8**	11.0	11.6
Airport (50)	45.0	46.6	46.6	33.2	**47.4**	46.4
Barman (40)	30.8	30.4	28.8	3.6	**40.0**	**40.0**
Childsnack (20)	2.8	5.2	6.0	5.2	8.8	**9.6**
DataNetwork (20)	15.0	15.0	16.4	10.2	**17.2**	16.6
Freecell (80)	73.0	76.2	76.8	**78.2**	77.2	78.0
GED (20)	14.6	**20.0**	**20.0**	19.0	**20.0**	**20.0**
Logistics (63)	59.4	**63.0**	**63.0**	59.0	**63.0**	**63.0**
Parking (40)	23.4	**40.0**	39.4	33.2	**40.0**	**40.0**
Pipes-notank (50)	45.8	**46.0**	45.8	42.8	45.6	45.6
Snake (20)	6.8	10.4	10.2	11.8	**12.6**	12.4
Sokoban (30)	6.6	10.4	11.2	**11.6**	**11.6**	11.0
Spider (20)	1.0	9.6	11.6	12.0	**12.4**	12.2
Storage (30)	26.8	28.6	28.4	24.2	28.4	**28.8**
Tetris (20)	10.0	14.0	14.2	1.0	**16.4**	15.8
Transport (60)	39.2	43.4	42.0	33.8	51.6	**54.2**
VisitAll (37)	15.4	16.6	16.2	5.2	18.0	**19.2**
Others (1075)	990.4	984.6	991.0	982.8	**994.0**	993.0
Sum (1695)	1413.8	1470.2	1477.6	1378.6	1515.2	**1517.4**
Std. Error	5.9	5.3	5.2	4.9	4.2	4.6
Exp. per Lookahead	27.9	15.3	16.6	43.6	15.2	14.9
Lookahead Success	88.5%	87.8%	88.7%	95.4%	89.5%	89.6%

TABLE 4.1: Coverage results for Refinement-HC with BrFS lookahead and a depth bound of 4 compared to BrFS with different types of novelty pruning (left part of the table), and A* and GBFS lookahead with C-novelty pruning (right part of the table). Domains where the difference in coverage between the best and worst configuration is at most 3 are grouped into "Others". The last two rows additionally show the number of expansions per fully explored lookahead (geometric mean across all commonly solved instances where at least one lookahead was fully explored by all configurations) and the average percentage of lookaheads that result in a state with lower heuristic value to be found.

novelty pruning. Finally, we evaluate all instantiations of the `HANDLE_*` macros with the best-performing lookahead search.

4.5.3.1 Comparison of Lookahead Search Algorithms

We first compare different lookahead search algorithms for Refinement-HC, again using the macro instantiations *DeadEndRestart*, *StagnationBackjump*, and *ExhaustionRestart*.

The left side of Table 4.1 shows the coverage for Refinement-HC with breadth-first search lookahead using different bounding methods. The Refinement-HC variants with $\mathcal{N}_1$ and $\mathcal{N}_C$ heavily outperform the best depth-bounded variant by +56.4 respectively +63.8 overall coverage. Comparing the IW(C) and BrFS[4] variants directly, the former is better in 23 domains and worse in only 6. The most significant gains come from Parking (+16

solved instances), Spider (+10.6), and GED (+5.4). However, in domains where the depth-bounded lookahead works better, the difference is comparatively small: The largest per-domain losses in coverage are just two fewer solved instances in each of Barman and Nomystery.

The novelty variant using $\mathcal{N}_2$ performs considerably worse compared to the ones with $\mathcal{N}_1$ and $\mathcal{N}_C$ because the pruning is much less aggressive, so each lookahead may expand a large number of states before exhausting the novel ones. Most domains where the IW(2) lookahead works well are those that also benefit from large depth bounds, examples are Freecell, Sokoban, and Spider.

The last two rows of Table 4.1 give further insight into the difference between using a depth bound compared to novelty pruning in the lookahead. The "Exp. per Lookahead" statistic indicates the average lookahead search space size, giving some insight into the search vs. refinement trade-off for the different bounding methods. Specifically, it shows how many states are expanded in lookahead iterations where the search space is fully exhausted until the depth bound is reached, or all novel states are expanded (i.e., those resulting in refinement or stagnation). While BrFS[4] considers more states on average compared to IW(1) and IW(C), the percentage of lookahead iterations that result in a state with lower heuristic value to be found is similar (88.5% compared to 87.8% respectively 88.7%), which shows that the novelty-based lookahead is similarly effective with less search effort. Overall, Refinement-HC with IW(C) needs on average 38% fewer expansions to find a solution compared to Refinement-HC with BrFS[4] lookahead on commonly solved instances. For our remaining experiments that use a novelty-based lookahead we stick to $\mathcal{N}_C$ as the best-performing novelty variant overall.

Consider now the right part of Table 4.1, which shows the results when the lookahead uses the heuristic for guidance instead of a pure breadth-first search. Using either A* or GBFS instead of BrFS for the lookahead (i.e., replacing BrFS[k] by A* or GBFS with novelty pruning in Algorithm 2 and the HANDLE_* macros) further boosts the performance of Refinement-HC: The lookahead success rate improves slightly, and these variants achieve 37.6 respectively 39.8 more solved instances in total. The biggest difference can be seen in Transport (+12.2 coverage for GBFS[$\mathcal{N}_C$] compared to BrFS[$\mathcal{N}_C$]) and Barman (+11.2), though the advantage is consistent across most domains, dropping only slightly in 5 domains (yet at most by −0.6).

4.5.3.2 Evaluating the Macro Choices

Table 4.2 shows the coverage for all available combinations of options in the HANDLE_* macros. Depending on the chosen settings, the coverage can vary significantly. For the

	DEBackjump			*DERestart*		
	SContinue	*SBackjump*	*SRestart*	*SContinue*	*SBackjump*	*SRestart*
EContinue	1412.0	1422.0	1397.2	1451.8	1457.0	1415.8
EBackjump	1436.2	1440.6	1457.4	1476.8	1479.2	1465.4
ERestart	1510.6	1514.6	1467.8	1508.2	**1517.4**	1465.8

TABLE 4.2: Coverage of Refinement-HC using GBFS with C-novelty pruning as the lookahead search algorithm for different instantiations of the `HANDLE_*` macros (abbreviating *DeadEnd (DE)*, *Stagnation (S)*, and *Exhaustion (E)*; results are averaged over 5 runs, the standard error ranges from 4.0 to 6.5).

DeadEnd and *Stagnation* options, there are no clear winners, but we can make some general observations:

(a) *DeadEndBackjump* is inferior to *DeadEndRestart* in most cases.

(b) For *Stagnation*, *Backjump* is usually the best configuration, and *Restart* is generally the worst.

(c) For *Exhaustion*, *Restart* is always the best configuration, and *Continue* is always the worst.

Observation (a) shows that recognized dead ends are most effectively escaped through a full restart. However, when combined with *ExhaustionRestart*, the results for the different choices of the *DeadEnd* options are very close, as restarting too frequently diminishes its benefit.

Regarding observation (b); restarting the search in case of stagnation seems to be an overreaction, while backjumping can avoid excessive refinement in a single state. Both *Restart* and *Backjump* reduce the number of conjunctions added during the search compared to *Continue* as expected, but restarting from the initial state adds significant search effort (+26% expansions compared to *Continue*, +25% over *Backjump*) as the search must redo some of the effort to move away from the initial state.

For *Exhaustion*, restarting from the initial state is clearly the best choice. The results indicate that, if the search space is exhausted with helpful actions pruning, the root state of the lookahead is indeed likely a dead end, and a restart from the initial state effectively escapes that region of the search space (similar to the observations for the *DeadEnd* options). The advantage of *ExhaustionRestart* is most apparent in domains with many dead ends. For example, in Sokoban, the *ExhaustionRestart* configurations have a coverage of 11.2 on average compared to 3.3 for *ExhaustionContinue* and 6.7 for *ExhaustionBackjump*, and in Pegsol the same comparison yields average coverage values of 34.7, 20.9, and 24.1.

The overall best-performing configuration is also the combination of the individually strongest options in *DeadEndRestart* with *StagnationBackjump* and *ExhaustionRestart*, with an overall coverage of 1517.4 ± 4.6. This result is robust across most domains; the only two domains where other configurations solve at least three more instances on average are Snake (where the mentioned configuration has an average coverage of 12.4, but configurations using *StagnationBackjump* and *ExhaustionBackjump* have 15.6) and Tetris (15.8 vs. 19.2 for configurations with *StagnationContinue* and either *ExhaustionContinue* or *ExhaustionBackjump*).

4.6 Refinement-HC with Relaxed Subgoal Counting

We now show that relaxed subgoal counting can be leveraged to speed up the costly computation arising from refined heuristic functions during Refinement-HC. We first describe our idea and method, then evaluate the method experimentally.

4.6.1 Method

While the heuristic becomes more accurate through iterative refinement, it typically also becomes more computationally expensive to compute, which is particularly true for $h^{C\mathrm{FF}}$. In depth-bounded Refinement-HC, one can attempt to compensate for this effect by choosing the depth bound sufficiently large, thereby avoiding too many refinement operations and shifting more responsibility to the search (yet this is detrimental to overall performance). Another strategy to counteract the computational complexity of the heuristic is to attempt to reduce the frequency of heuristic evaluations. One way is to cache heuristic values, thereby avoiding reevaluations on states that have been seen before, but that approach contradicts the purpose of online refinement.

The solution we propose here is to use an approximative heuristic instead: We compute a relaxed plan only in the root state of the current lookahead, and then use a subgoal-counting heuristic h^{SC} based on that to guide the lookahead search. In Refinement-HC, we check all states s' that are explored in the lookahead from s for $h(s') < h(s)$, considering them as potential root states for the next lookahead iteration. With our subgoal-counting variant, we only compute h twice for each lookahead iteration: once in the root state s, and then on the state from the lookahead with minimal h^{SC} value s', which is the only state for which we test $h(s') < h(s)$. If the test passes, the next lookahead iteration continues from s' as before. Otherwise, we refine the heuristic once and continue again from s.

Algorithm 7: Refinement-HC with Subgoal Counting (RHC-SC)

```
1  s := I
2  while h(I) ≠ ∞ do
3      Let h^SC be the subgoal counting heuristic derived from π[h](s)
4      Run a bounded lookahead search from s
5      Let s' be the state with minimal h^SC value seen in the lookahead
6      if h(s') ≥ h(s) then // lookahead failed
7          if lookahead search space was exhausted without pruning any state then
               // s is a dead end (no helpful actions pruning)
8              Mark s as a dead end, pruning it in future lookaheads
9              HANDLE_DEAD_END
10         if the previous lookahead iteration also originated at s and s ≠ I then
               // previous refinement was unsuccessful
11             HANDLE_STAGNATION
           // refine h once
12         REFINE_HEURISTIC
13         if h(s) = ∞ then
14             HANDLE_DEAD_END
15             break
16     else
17         s := s'
18         if s ⊇ G then
19             return SOLVED
20 return UNSOLVABLE
```

The pseudocode for Refinement-HC with subgoal counting (short RHC-SC) is shown in Algorithm 7. The macros are defined as before (only in the Backjump procedure of the HANDLE_STAGNATION macro, we stop chaining back if the state with minimal h^{SC} value seen in the lookahead has a lower h value). One major change is that we do not use helpful actions pruning in the lookahead since h^{SC} does not define helpful actions. This means that we can collapse the "search space exhaustion" case with the "dead end" case, as not reaching the lookahead horizon means that s is in fact a dead end (line 9). The main motivations behind only doing a single iteration of refinement instead of refining h until $h(s) > h(s')$ are (a) since we only sample one state in the lookahead, the difference between $h(s)$ and $h(s')$ might be large, leading to many iterations of refinement, and (b) the lookahead is very cheap, so we can afford to start another one immediately after each refinement step.

When using a subgoal counting heuristic h^{SC}, the two key challenges are (1) selecting the underlying plan from which the subgoals are derived, and (2) ensuring that h^{SC} values are not compared across different underlying plans. In BFWS [Lipovetzky and Geffner, 2017], relaxed plans for subgoal counting are computed in states where the

	DEBackjump			*DERestart*		
	SContinue	*SBackjump*	*SRestart*	*SContinue*	*SBackjump*	*SRestart*
g	1419.0	1482.4	1277.2	1439.4	1491.2	1278.8
$g+h$	1479.8	1526.8	1437.6	1504.0	**1534.0**	1435.2
h	1450.2	1507.6	1428.0	1467.0	1510.4	1427.2

TABLE 4.3: Coverage of Refinement-HC with subgoal counting and C-novelty pruning for different instantiations of the HANDLE_* macros and different expansion orders in the lookahead (averaged over 5 runs, the standard error ranges from 4.5 to 5.8).

number of satisfied top-level goals increases over that of its parent. However, BFWS does compare subgoal-counting values between states with potentially different underlying relaxed plans, though the subgoal counting value is not used as a heuristic directly but instead as part of a novelty measure, making challenge (2) less important. The lookahead of Refinement-HC is perfectly suitable for a subgoal counting heuristic, since the root state of the lookahead is a straightforward choice for computing a relaxed plan with which to instantiate h^{SC}, and only one instance of the heuristic is used in each lookahead (ensuring comparability between the h^{SC} values).

Refinement-HC with subgoal counting inherits the completeness property from standard Refinement-HC:

Proposition 4.6. *Given a heuristic h converging with ρ, Refinement-HC with subgoal counting is a complete search algorithm.*

Proof. Observe again that every (unsuccessful) episode refines h at least once, and that this sequence of refinements stops only if either (a) a plan is found, or (b) $h(\mathcal{I}) = \infty$. Then by the same chain of reasoning as applied in the proof of Theorem 4.4, Refinement-HC with subgoal counting is complete. □

4.6.2 Experiments

First, we again evaluate the different choices for the HANDLE_* macros and different expansion orders for the lookahead search. We highlight an important implementation detail, and finally compare RHC-SC to Refinement-HC without subgoal counting. In all experiments for RHC-SC we use $\mathcal{N}_C$ to bound the lookahead search.

4.6.2.1 Evaluating the Macro Choices

Table 4.3 shows an overview of the different configurations for RHC-SC. The main takeaways are:

(a) *DeadEndRestart* is better than *DeadEndBackjump* when using *Continue* or *Backjump* for the `HANDLE_STAGNATION` case, and they are close to equal when using *StagnationRestart*.

(b) *StagnationBackjump* is generally the best option followed by *StagnationContinue*, while the performance of *StagnationRestart* trails far behind.

(c) Using the expansion order $g + h$ works best, followed by h and then g.

Consistent to our results in the previous section, restarts can help performance, but not if done too frequently (i.e., combining *DeadEndRestart* with *StagnationRestart* is not a good idea). Like before, *DeadEndRestart* is consistently the best option. The only configurations where *DeadEndBackjump* and *DeadEndRestart* have similar performance are those that are combined with *StagnationRestart*, but both *StagnationBackjump* and *StagnationContinue* yield much better results overall.

Using h^{SC} to guide the lookahead positively impacts the results compared to pure breadth-first search. In Refinement-HC without subgoal counting, expanding nodes greedily by $h^{C\mathrm{FF}}$ in the lookahead works best, whereas for the less accurate h^{SC} the more conservative choice of an open list ordered by $g + h$ is superior.

Overall, the combination of *DeadEndRestart* and *StagnationBackjump* with an A* lookahead works best, yielding a coverage of 1534 ± 5.1. There are domains where other combinations are better though: On Parking and Airport, configurations using BrFS for the lookahead have a coverage of up to 39.6 (+19.8 over the overall best combination) respectively 47.8 (+8); on Thoughtful, configurations using *StagnationRestart* have a coverage of up to 20 (+8).

4.6.2.2 Memory Overhead

In satisficing planning with limits similar to the settings used by the International Planning Competitions (30 minute timeout and memory limits between 2 and 8 GB), time is usually the most constraining factor. While there exist some domains where already the (grounded) problem representation poses an issue for standard memory constraints (e.g., Organic Synthesis), the memory usage is mostly dominated by the state data of

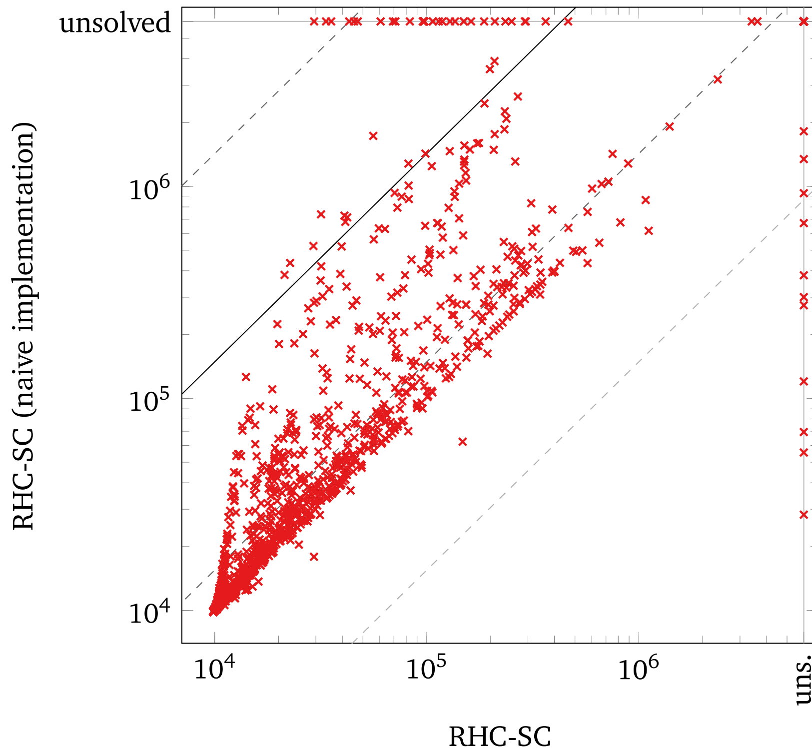


FIGURE 4.3: Comparison of the peak search memory usage (in kilobytes) for Refinement-HC with subgoal counting.

generated states, which is kept for duplicate detection. Since many satisficing planners use heuristics that are non-trivial to compute, they tend to hit the time limit before the number of generated states becomes an issue.

The subgoal-based lookahead in RHC-SC may generate a large number of states very quickly since the heuristic is a simple counter. To avoid memory issues, we release the states that were generated in each lookahead from memory, keeping only the states along the current path from the initial state and those that were evaluated by h^{CFF} (to be able to prune states with $h^{\mathrm{CFF}} = \infty$).

Figure 4.3 shows a comparison of the memory usage of our implementation to a naive version that keeps all generated states in memory. Except for few random outliers, the naive implementation uses significantly more memory, often by more than one order of magnitude. The naive implementation runs out of memory on 55–57 instances (depending on the random seed) compared to just 5 instances when the lookahead states are discarded. In Childsnack and Parking, this implementation detail has the biggest impact, reducing the number of instances where the search runs out of memory from 11 (out of 20) respectively 27 (out of 40) to zero, and increasing coverage by 3.6 respectively 5.4.

Coverage	RHC	RHC-SC	Diff.
Agricola (20)	11.6	**11.8**	+0.2
Airport (50)	**46.4**	39.8	−6.6
Childsnack (20)	9.6	**13.4**	+3.8
DataNetwork (20)	16.6	**19.4**	+2.8
Freecell (80)	**78.0**	77.4	−0.6
Nomystery (20)	**10.4**	10.0	−0.4
Openstacks (90)	89.6	**90.0**	+0.4
OrgSynth-split (20)	**2.6**	1.6	−1
Parcprinter (40)	**40.0**	39.8	−0.2
Parking (40)	**40.0**	19.8	−20.2
Pegsol (35)	**35.0**	33.8	−1.2
Pipes-notank (50)	45.6	**48.8**	+3.2
Pipes-tank (50)	44.2	**47.0**	+2.8
Satellite (36)	**36.0**	31.0	−5
Snake (20)	12.4	**18.0**	+5.6
Sokoban (30)	11.0	**11.4**	+0.4
Spider (20)	12.2	**14.2**	+2
Storage (30)	28.8	**30.0**	+1.2
Termes (20)	4.0	**9.6**	+5.6
Tetris (20)	15.8	**20.0**	+4.2
Thoughtful (20)	**20.0**	12.0	−8
Tidybot (20)	18.0	**19.4**	+1.4
Transport (60)	54.2	**60.0**	+5.8
Trucks (30)	16.2	**18.8**	+2.6
VisitAll (37)	19.2	**37.0**	+17.8
Others (817)	**800.0**	**800.0**	± 0
Sum (1695)	1517.4	**1534.0**	+16.6
Std. Error	4.6	5.1	

TABLE 4.4: Coverage of Refinement-HC with vs. without subgoal counting. Domains with equal coverage are grouped into "Others".

4.6.2.3 Comparison to Refinement-HC without Subgoal Counting

Table 4.4 shows a direct comparison of the coverage for the best-performing variants of Refinement-HC with vs. without subgoal counting. Both RHC and RHC-SC fully solve all domains that are grouped into Others, except for Organic Synthesis, where both solve 3 instances and the Fast Downward translator runs out of memory on the other 17. None of the two variants consistently outperforms the other, however, there are 16 domains where RHC-SC has the upper hand compared to 9 where it is worse. The domains with the biggest change in coverage are Parking (−20.2) and VisitAll (+17.8). In Parking, the subgoal-counting heuristic is unreliable: Only 3.6% of lookahead searches result in a state with lower h^{CFF} value to be found for RHC-SC (compared to 93.7% for RHC). On

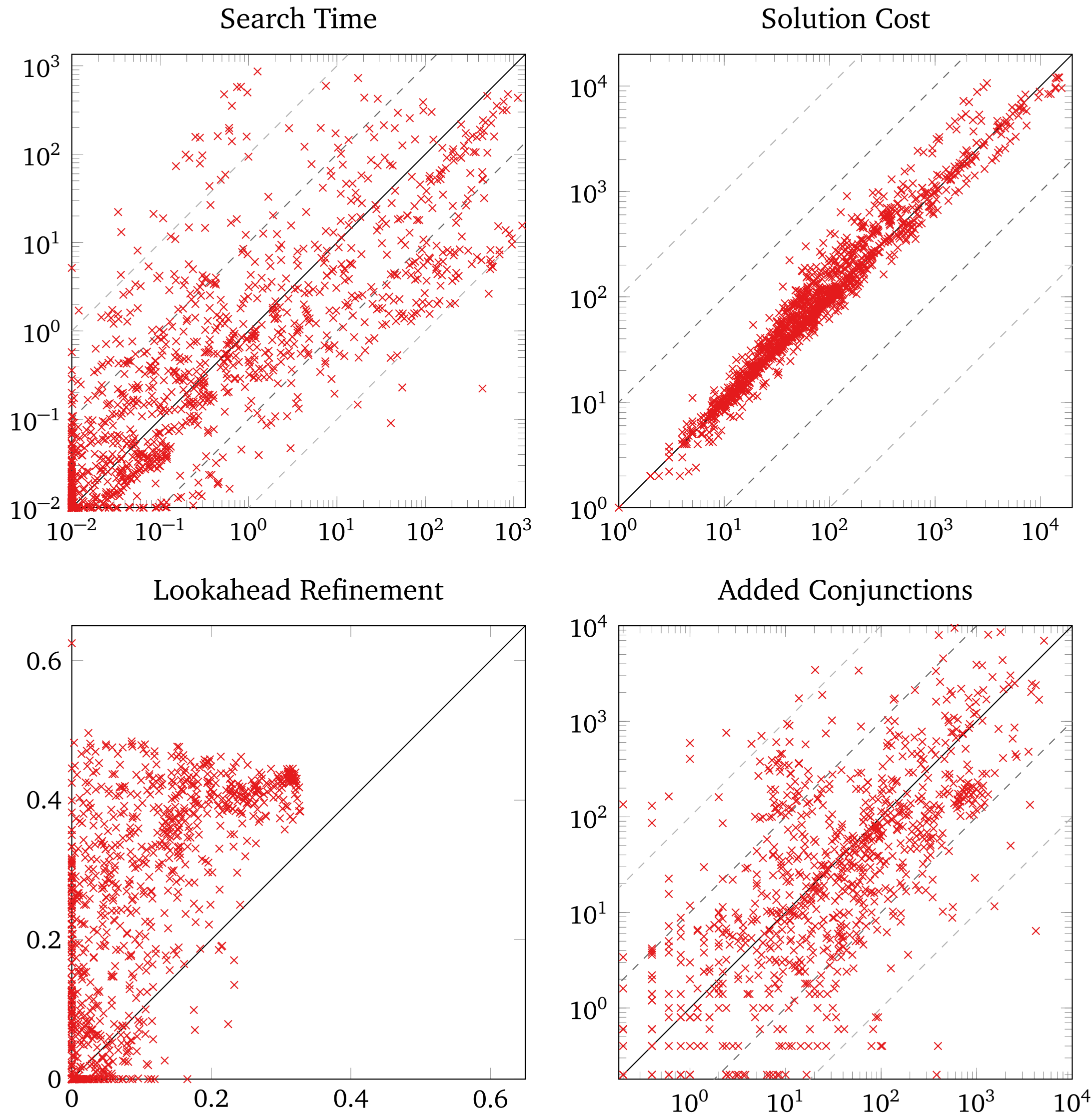


FIGURE 4.4: Search time, solution cost, percentage of lookaheads that result in refinement, and number of added conjunctions for commonly solved instances of RHC (x-axis) and RHC-SC (y-axis).

the other hand, h^{SC} yields good guidance in VisitAll, enabling RHC-SC to make significant jumps towards the goal after each lookahead iteration.

Figure 4.4 compares additional statistics between Refinement-HC with and without subgoal counting. When aggregating across the full benchmark set, the overall search times for Refinement-HC and RHC-SC are very similar. However, the scatter plot reveals that there are many instances where they differ by multiple orders of magnitude. As mentioned above, there are extreme cases where h^{SC} is consistently accurate (VisitAll) or inaccurate (Parking), but also many domains where either approach has an advantage on some instances. Using the weaker h^{SC} heuristic in the lookahead also comes with a slight penalty in solution cost (on average, plans computed by RHC are 12.8% cheaper

than those of RHC-SC).

Many more lookahead iterations of Refinement-HC with subgoal counting result in refinement compared to standard Refinement-HC. This is expected, since RHC-SC already triggers refinement if *one* lookahead state (i.e., the state with minimal h^{SC} value) does not have a better $h^{C\text{FF}}$ value than the root state of the lookahead, whereas RHC may consider all states that have been explored in the lookahead. However, this does not necessarily result in more added conjunctions overall, mainly because the refinement procedure in RHC-SC is more conservative, adding just one conjunction to $h^{C\text{FF}}$ instead of refining until the $h^{C\text{FF}}$ value of the lookahead root state increases sufficiently.

4.7 Greedy Best-First Search

We finally consider greedy best-first search as a systematic-search alternative to hill climbing. We first discuss general challenges with online refinement in GBFS, then introduce our online-refinement GBFS variant, and evaluate it empirically.

4.7.1 Online Refinement in GBFS

Intuitively, hill-climbing algorithms are well suited for online refinement because of the local exploration phases, giving a clear indication that the heuristic is weak on the local search space. However, most state-of-the-art satisficing planners are based on greedy best-first search, which maintains a global open list, and expands nodes by lowest h values. Since GBFS does not have a similar focus on smaller areas of the search space, it is much harder to identify local minima or plateaus in the search procedure (and thereby more difficult to define suitable refinement criteria).

Our initial ideas for online refinement in GBFS were focused on attempting to identify local minima, following the central paradigm of Refinement-HC. Specifically, we wanted to consider the set S_0 of states with the lowest h value seen so far, and then track the evaluation of their successors (this is not straightforward even just for the immediate successors, as GBFS is typically used with lazy evaluation, where the heuristic is only evaluated on expansion, not generation, of a state). If h has been evaluated on all the successors of at least one state $s \in S_0$ and none of these successors have a better heuristic value, then s is a local minimum under h and we have identified an opportunity for refinement. We implemented this idea, including different variants where we require that all states in S_0 are proven local minima and/or consider successors up to a given depth, as well as refining only a single conjunction instead of refining until the local minimum is

removed from the search space surface. However, in preliminary experiments, we found this variant of GBFS to be vastly inferior to our hill-climbing algorithms, with the overall coverage ranging between 1368 and 1434.

Instead, we augment GBFS with a lookahead function similar to the one we use in Refinement-HC. Note that the lookahead search based on subgoal counting as introduced in Section 4.6 is particularly suitable as a lookahead function in GBFS due to its low overhead. This lookahead strategy achieves two purposes in GBFS: (1) it can identify states where refining the heuristic may be effective (if the lookahead does not yield a better state according to the heuristic), and (2) it allows the search to skip ahead to a state closer to the goal (if the lookahead does find a better state).[2]

4.7.2 GBFS with Subgoal-Counting Lookahead and Online Refinement

As discussed in the context of Refinement-HC, the lookahead search based on subgoals is extremely lightweight, which opens up its use as a lookahead method in GBFS. Our GBFS variant, that we call GBFS-SCL (GBFS with *Subgoal-Counting Lookahead*)[3], invokes this lookahead procedure after each expansion, and either inserts the resulting state at the front of the open list, or refines the heuristic.

Algorithm 8 shows the pseudocode of GBFS-SCL, with the changes to standard GBFS highlighted in red. After each expansion, GBFS-SCL invokes the lookahead search that is also used by RHC-SC, i.e., it performs a bounded lookahead search using h^{SC} for guidance. The lookahead search maintains its own closed list starting from an empty one at the beginning of each lookahead, and does not consider the closed list of the overall search (neither for lookup nor updating). Like in RHC-SC, the lookahead search returns the best lookahead state s' according to h^{SC}. If s' is already closed, the search proceeds like standard GBFS.[4] Otherwise, s' is evaluated with the main heuristic h. If the heuristic value decreases from the root state of the lookahead, then s' is inserted at the front of the open list, and otherwise the heuristic is refined. When GBFS-SCL finds a goal state, it reconstructs the plan from parent pointers like standard GBFS. Hence, we store the parent information for the path to s' from the lookahead, which can be updated if a better path (with lower g value) is found.

[2] This method relates to previous works on lookahead strategies in GBFS [Vidal, 2004; 2011; Nakhost and Müller, 2009; Lipovetzky and Geffner, 2017]; we will discuss these below.

[3] In the original conference paper [Fickert, 2020] this algorithm was introduced as GBFS-RSL (for *Relaxed Subgoals Lookahead*). We prefer the name GBFS-SCL because it seems slightly more apt and for consistency with RHC-SC.

[4] We tested other options on what to do if s' is closed (namely, (a) invoking `REFINE_HEURISTIC`, or (b) invoking `REFINE_HEURISTIC` if $h(s') < h(s)$), but found no statistically significant effect on the results. We thus stick to what we consider the most straightforward option of just continuing without refinement as shown in the pseudocode.

Algorithm 8: GBFS-SCL

```
1  Open := [I]
2  Closed := ∅
3  while Open ≠ [] do
4      s := Open.pop()
5      if s ∈ Closed then
6          continue
7      if s ⊇ G then
8          return SOLVED
9      Closed := Closed ∪ {s}
10     if h(s) ≠ ∞ then
11         Insert the successors of s into Open
12         Let h^SC be the subgoal counting heuristic derived from π[h](s)
13         Run a bounded lookahead search from s
14         Let s' be the state with minimal h^SC value seen in the lookahead
15         if s' ∈ Closed then
16             continue
17         else if h(s') < h(s) then
18             Insert s' at the front of Open
19         else
20             REFINE_HEURISTIC
21 return UNSOLVABLE
```

Note that the online refinement of the heuristic causes the open list to be ordered according to mixed versions of the heuristic (with different degree of refinement). This is less significant with lazy evaluation, but it can create a small bias towards states closer to the initial state as the heuristic values increase with refinement. On the other hand, this can also help escape local minima as refinement should quickly cause the heuristic value to increase in such regions, while open states that were evaluated before reaching the local minimum retain their original value.

GBFS-SCL can not only be used as an online-refinement algorithm, but also as a standard search algorithm by simply continuing the search normally if the lookahead does not return a state with lower h value (i.e., dropping the **else** case in line 19 that invokes `REFINE_HEURISTIC`). The only remaining requirement is that h^{SC} must somehow be instantiated, either by using heuristics that have an underlying relaxed plan (e.g., heuristics based on (partial) delete relaxation or abstractions), or by using an alternative method to derive subgoals such as landmarks. The addition of the lookahead is generally a non-intrusive change to GBFS, and does not affect compatibility with most search-enhancing techniques like a dual queue for preferred operators (which we use in our experiments).

GBFS-SCL is related to the YAHSP planner [Vidal, 2004; 2011]. YAHSP is based on greedy best-first search with h^{FF}. After each expansion of a state s, YAHSP attempts to

repair the current relaxed plan $\pi[h^{\text{FF}}](s)$, and inserts the state resulting from following the applicable (under non-relaxed semantics) prefix of the repaired plan into the open list. Our lookahead based on subgoal counting uses the same underlying idea of extracting information from the relaxed plan to generate a lookahead state. While YAHSP uses the actions of the relaxed plan, GBFS-SCL uses its subgoals, following the relaxed plan more loosely compared to YAHSP.

Extending GBFS with local exploration methods has been considered before, either via random walks [Nakhost and Müller, 2009], or bounded local search [Xie et al., 2014; Lipovetzky and Geffner, 2017]. Lipovetzky and Geffner's method is similar to ours in that they also exploit novelty to enhance the exploration aspect, and they observed significant gains over previous methods. The main difference to our work is that these methods aim to use local exploration as a tool to escape local minima or plateaus, and it is only triggered if the search is considered to be stuck (by tracking the number of expansions since the minimal heuristic value has decreased). GBFS-SCL instead uses low-overhead local searches after each GBFS expansion with the main goal of accelerating the search, and using it as a trigger to improve the heuristic in the online-refinement variant.

4.7.3 Experiments

GBFS-SCL does not have options like the `HANDLE_*` macros of Refinement-HC that control certain aspects of the algorithm, but there is still one choice to make: the instantiation of the lookahead search algorithm. Like with RHC-SC, we evaluate BrFS, A*, and GBFS with novelty pruning. While we are mainly interested in GBFS-SCL with online refinement of h^{CFF}, we also evaluate the GBFS-SCL variant without online refinement using various (partial) delete relaxation heuristics. Specifically, we consider standard h^{FF}, as well as the partial delete relaxation heuristics h^{RB}, h^{gray}, and h^{CFF} with offline refinement. The red-black heuristic h^{RB} [Domshlak et al., 2015] is based on the tractable fragment as introduced in Section 3.3.1. The gray-planning heuristic h^{gray} extends h^{RB} by additionally considering *limited-memory* variables that remember a limited number of their most recent assignments [Speicher et al., 2017]. For h^{CFF} with offline refinement, the heuristic is iteratively refined in the initial state until its internal complexity has increased to a growth factor of 1.5 (or a timeout of 15 minutes is reached), following the best performing settings of previous work on h^{CFF} [Fickert et al., 2016]. We compare GBFS-SCL to standard GBFS, as well as to a GBFS variant similar to GBFS-SCL but using YAHSP's lookahead instead, i.e., after each expansion, we consider the state returned by YAHSP's lookahead method and insert it at the front of the open list if it has a lower heuristic value.

h	GBFS-SCL BrFS	GBFS-SCL A*	GBFS-SCL GBFS	GBFS	YAHSP
h^{FF}	1390.6	<u>1462.4</u>	1457.6	1397.6	1477.0
h^{RB}	1388	1425	<u>1450</u>	1397	1427
h^{gray}	1437	1463	<u>1469</u>	1441	1467
$h^{C\text{FF}}_{off}$	1406.8 (1400.6)	<u>1490.2</u> (1491.8)	1463.2 (1465.4)	1372.6	1498.2
$h^{C\text{FF}}_{on}$	1499.4 (1445.2)	**<u>1558.8</u>** (1530.8)	1519.4 (1494.0)	–	1464.6

TABLE 4.5: Coverage of GBFS-SCL with different configurations, compared to standard GBFS and GBFS using YAHSP's lookahead. The $h^{C\text{FF}}$ heuristic is included with offline ($h^{C\text{FF}}_{off}$) and online ($h^{C\text{FF}}_{on}$) refinement variants. The GBFS-SCL configurations for h^{FF}, h^{RB}, and h^{gray} use 1-novelty pruning, for $h^{C\text{FF}}$ results with both C-novelty and 1-novelty pruning are shown (with the latter in parentheses). For each heuristic, the best-performing lookahead search algorithm in GBFS-SCL is <u>underlined</u>. The overall best configuration is **boldfaced**.

Table 4.5 shows an overview of the results. A* is the best choice for the lookahead search algorithm when using $h^{C\text{FF}}$ (like with RHC-SC) and h^{FF}, while GBFS works better for h^{RB} and h^{gray}.[5] Regarding $\mathcal{N}_1$ vs. $\mathcal{N}_C$ for $h^{C\text{FF}}$; while there is no clear winner for the offline variant of $h^{C\text{FF}}$, $\mathcal{N}_C$ is superior when using online refinement (as before). This observation again highlights the synergistic effect of sharing the set of conjunctions with novelty pruning for online refinement discussed in Section 4.5.2, i.e., by sharing the set of conjunctions, further refinement is reduced over time.

For all considered heuristics, GBFS-SCL (without online refinement) improves overall coverage compared to standard GBFS: +64.8 for h^{FF}, +53 for h^{RB}, +28 for h^{gray}, and +117.6 for (offline) $h^{C\text{FF}}$. For h^{FF}, most of the advantage in coverage is gained in Transport (+21.2), VisitAll (+17), and Barman (+15.6). In some domains, GBFS-SCL performs worse because the lookahead only rarely yields a better state, in particular in domains with many dead ends. For example, in Floortile, on average only 1% of lookaheads are successful resulting in −4.4 coverage, and we get similar results in Sokoban (4% successful lookaheads, −3.4 solved instances). Another source of ineffectiveness for GBFS-SCL is the added overhead of the lookahead, e.g., in Parking the lookahead uses 96% of the overall time (mostly for successor generation and evaluating novelty), which, again combined with a low lookahead success rate of 3%, leads to leading to 11 fewer solved instances compared to standard GBFS.

Partial delete relaxation methods are designed to make delete relaxation heuristics more accurate, which should intuitively mean that their subgoals provide better guidance, making them better suited for GBFS-SCL. However, while the increase in coverage over standard GBFS is greater for $h^{C\text{FF}}$ than it is for h^{FF}, this is not the case for h^{RB} and h^{gray}.

[5] For h^{FF} here, we use the $h^{C\text{FF}}$ implementation with only singleton conjunctions, to avoid implementational differences when comparing h^{FF} and $h^{C\text{FF}}$ throughout this chapter (in particular in Section 4.8.1).

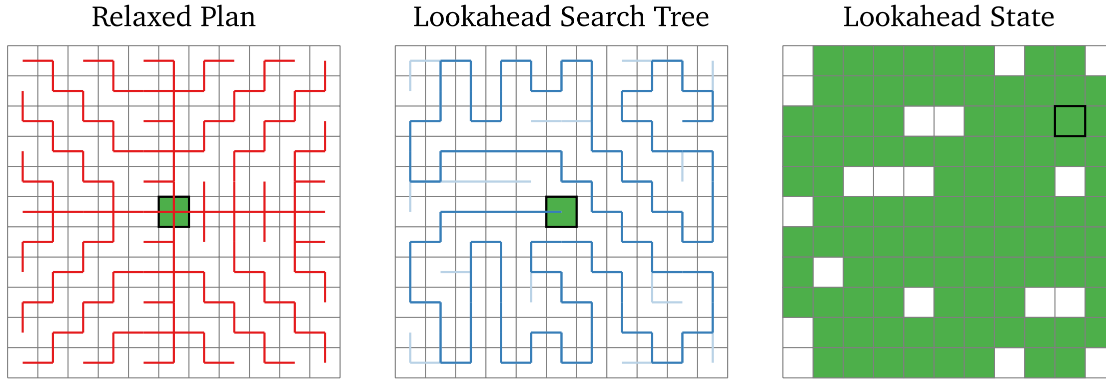


FIGURE 4.5: Illustration of GBFS-SCL's lookahead on an 11x11 instance of VisitAll.

We believe this can be explained by the structure of the partially relaxed plans: For red-black and gray planning, repair sequences are inserted into the relaxed plan to resolve conflicts (unsatisfied preconditions) on the un-relaxed variables. In the context of GBFS-SCL though, these sequences may lead the lookahead search away from the "intended" path of the relaxed plan, as the subgoal-based lookahead likely does not follow these sequences exactly.

Even in its variant without online refinement, GBFS-SCL can be an effective method to boost the search, and sometimes makes big leaps to states closer to a goal in a single lookahead (we illustrate such an example below). However, GBFS with YAHSP's lookahead leads to similar results, but this picture changes when considering the online-refinement variants. In combination with h^{CFF} and online refinement, GBFS-SCL clearly outclasses all other configurations with an overall coverage of 1558.8 ± 4.7, beating e.g. its corresponding offline-refinement configuration by +68.6, and configurations with other heuristics by an even larger margin. Interestingly though, this is not the case if we add online refinement to GBFS with YAHSP's lookahead in the same way as in GBFS-SCL (i.e., invoking REFINE_HEURISTIC if the lookahead does not yield a state with lower heuristic value), suggesting that a failure of YAHSP's lookahead does not necessarily indicate that the heuristic needs improvement. In some domains, YAHSP's lookahead frequently fails to return a better state, leading to large computational overhead due to excessive refinement, e.g., dropping down to just 2 out of 40 solved instances in Barman, 1.4 out of 40 in Parking, and 2.4 out of 20 on Termes (whereas online-refinement GBFS-SCL has a coverage of 39.4, 35.6, and 6.8 in these domains).

VisitAll, where an agent must visit all cells in a given grid, is a simple domain that is particularly well suited for GBFS-SCL. Figure 4.5 illustrates the lookahead on a small instance of that domain. The agent is located in the center of an 11x11 grid (illustrated by the highlighted border), and must visit all other cells. For the experiment illustrated

here, we use h^{FF} with arbitrary tie breaking and GBFS as the lookahead search algorithm in GBFS-SCL. The first panel shows the relaxed plan computed by h^{FF}, branching out in all directions to reach each location of the grid. The center panel shows the search tree resulting from the lookahead search in GBFS-SCL. Since the lookahead uses 1-novelty pruning, each location is visited exactly once (each fact of the form $at = x$ is novel once). The highlighted path leads to the state with lowest h^{SC} value, which is shown in the third panel. In that state, the agent has already visited 102 of the 121 locations in the grid, bringing it much closer to the goal than the original state. After two more lookaheads, GBFS-SCL already returns a solution, after having computed h^{FF} only four times in total (on the three root states of the lookaheads and in the goal state). For comparison, standard GBFS with h^{FF} expands 12227 states on this instance.

4.8 Experiments

In this section, we compare our algorithms to similar algorithms without online refinement to highlight its benefits. Furthermore, we compare our algorithms to state-of-the-art planners, both on the IPC benchmarks and the recently published Autoscale benchmarks [Torralba et al., 2021].

4.8.1 Comparison to Baselines without Online Refinement

Table 4.6 compares the best-performing configurations of our online-refinement algorithms to related baselines. The baselines we consider here are incomplete enforced hill-climbing (EHC), the FF [Hoffmann and Nebel, 2001] strategy of running EHC first and then switching to GBFS in case of failure, and standard GBFS, each with h^{FF} and (offline-refined) $h^{C\text{FF}}$.

Comparing just h^{FF} with offline $h^{C\text{FF}}$; while the added conjunctions help in some domains (in particular Floortile and Woodworking), the opposite is true in many others (e.g., Openstacks). In total coverage, both the FF and GBFS search algorithms perform better with h^{FF} than with $h^{C\text{FF}}$. This has two major reasons. First, adding conjunctions introduces overhead in domains where standard h^{FF} is already a sufficiently strong heuristic, both through the time spent for the refinement at the start of the search, but also by slowing down the heuristic computation. Second, while the heuristic becomes more informed and its heuristic values increase, this added information may be confined to a small area of the search space and harm the search performance overall [see e.g., Wilt and Ruml, 2016], as the search may be guided into areas where the heuristic is less informed and its values are closer to h^{FF}.

Coverage	EHC		FF		GBFS		RHC	RHC-SC	GBFS-SCL
	h^{FF}	h^{CFF}	h^{FF}	h^{CFF}	h^{FF}	h^{CFF}			
Agricola (20)	4.6	3.8	**13.4**	13.0	12.8	12.8	11.6	11.8	12.4
Airport (50)	12.6	22.2	34.4	34.8	34.4	34.2	**46.4**	39.8	41.2
Barman (40)	31.6	19.2	31.2	18.6	24.4	5.4	**40.0**	**40.0**	39.4
Childsnack (20)	0.0	0.0	0.8	0.4	0.4	1.4	9.6	**13.4**	8.4
DataNetwork (20)	1.4	1.6	13.0	8.4	15.0	13.2	16.6	**19.4**	18.6
Depot (22)	11.2	16.4	19.8	21.8	19.0	21.4	**22.0**	**22.0**	**22.0**
DriverLog (20)	7.8	8.0	**20.0**	**20.0**	**20.0**	**20.0**	**20.0**	**20.0**	**20.0**
Elevators (40)	**40.0**	37.0	**40.0**	37.4	**40.0**	39.4	**40.0**	**40.0**	**40.0**
Floortile (40)	0.0	37.2	8.4	39.6	8.8	39.8	**40.0**	**40.0**	**40.0**
Freecell (80)	59.6	62.6	**80.0**	79.8	79.4	78.8	78.0	77.4	**80.0**
GED (20)	18.2	16.2	18.6	16.4	**20.0**	19.2	**20.0**	**20.0**	**20.0**
Gripper (20)	**20.0**	19.8	**20.0**	**20.0**	**20.0**	**20.0**	**20.0**	**20.0**	**20.0**
Hiking (20)	0.0	2.6	18.4	19.2	**20.0**	19.8	**20.0**	**20.0**	**20.0**
Logistics (63)	59.4	62.6	59.2	62.4	62.8	**63.0**	**63.0**	**63.0**	62.4
Miconic (150)	**150.0**	136.2	**150.0**	**150.0**	**150.0**	**150.0**	**150.0**	**150.0**	**150.0**
Mprime (35)	**35.0**	34.8	**35.0**	**35.0**	**35.0**	**35.0**	**35.0**	**35.0**	**35.0**
Mystery (19)	15.0	17.8	18.6	**19.0**	18.6	18.8	**19.0**	**19.0**	**19.0**
Nomystery (20)	2.2	0.4	8.4	6.2	8.6	6.0	**10.4**	10.0	9.6
Openstacks (90)	**90.0**	52.0	**90.0**	56.2	**90.0**	66.0	89.6	**90.0**	**90.0**
OrgSynth (20)	2.8	2.8	**3.0**	**3.0**	**3.0**	**3.0**	**3.0**	**3.0**	**3.0**
OrgSynth-split (20)	0.4	0.0	**11.2**	10.4	10.6	10.4	2.6	1.6	8.0
Parcprinter (40)	19.0	26.2	33.6	34.0	28.8	32.8	**40.0**	39.8	**40.0**
Parking (40)	11.2	18.2	18.2	21.2	33.2	19.4	**40.0**	19.8	35.6
Pathways (30)	22.2	22.2	25.4	24.8	21.8	21.0	**30.0**	**30.0**	24.8
Pegsol (35)	3.6	5.6	**35.0**	**35.0**	**35.0**	**35.0**	**35.0**	33.8	34.6
Pipes-notank (50)	23.8	27.2	41.8	42.6	41.4	41.2	45.6	48.8	**49.6**
Pipes-tank (50)	28.4	27.6	38.8	38.0	38.4	39.0	44.2	47.0	**48.8**
PSR (50)	0.0	8.0	**50.0**	**50.0**	**50.0**	**50.0**	**50.0**	**50.0**	**50.0**
Rovers (40)	39.2	38.4	39.0	39.0	**40.0**	**40.0**	**40.0**	**40.0**	**40.0**
Satellite (36)	35.8	35.0	35.8	**36.0**	**36.0**	35.8	**36.0**	31.0	30.8
Scanalyzer (28)	27.0	26.0	27.6	26.2	26.0	**28.0**	**28.0**	**28.0**	**28.0**
Snake (20)	3.6	5.0	4.6	6.8	6.8	6.2	12.4	**18.0**	17.2
Sokoban (30)	0.0	0.0	**28.8**	26.0	28.2	25.4	11.0	11.4	17.8
Spider (20)	3.2	4.0	13.6	12.6	13.6	12.8	12.2	14.2	**15.8**
Storage (30)	6.4	5.4	20.0	20.6	20.8	20.0	28.8	**30.0**	**30.0**
Termes (20)	0.0	0.6	2.4	3.6	**13.4**	11.8	4.0	9.6	6.8
Tetris (20)	0.0	0.2	12.2	8.6	14.4	13.8	15.8	**20.0**	19.8
Thoughtful (20)	12.0	12.2	14.0	15.4	10.8	12.8	**20.0**	12.0	15.0
Tidybot (20)	14.0	9.0	17.0	16.2	16.4	16.2	18.0	19.4	**19.8**
TPP (30)	27.6	27.6	27.6	28.0	**30.0**	**30.0**	**30.0**	**30.0**	**30.0**
Transport (60)	26.0	25.4	26.8	27.2	38.8	38.8	54.2	**60.0**	**60.0**
Trucks (30)	4.0	2.8	18.2	16.6	18.0	16.8	16.2	**18.8**	18.4
VisitAll (37)	5.8	5.4	5.6	5.4	20.0	18.2	19.2	**37.0**	**37.0**
Woodworking (40)	4.0	39.6	32.2	**40.0**	33.0	**40.0**	**40.0**	**40.0**	**40.0**
Others (90)	**90.0**	**90.0**	**90.0**	**90.0**	**90.0**	**90.0**	**90.0**	**90.0**	**90.0**
Sum (1695)	968.6	1014.8	1351.6	1335.4	1397.6	1372.6	1517.4	1534.0	**1558.8**
Std. Error	8.0	10.4	6.2	7.1	5.5	7.0	4.6	5.1	4.7
Normalized (%)	50.2	54.0	75.6	75.1	78.5	77.6	85.0	86.7	**88.0**

TABLE 4.6: Coverage on the IPC benchmarks for traditional search algorithms using h^{FF} and (offline-refined) h^{CFF} compared to our online-refinement search algorithms.

Our online-refinement algorithms on the other hand are much more flexible, and ensure that there is no unnecessary overhead by avoiding refinement if the heuristic is already sufficiently accurate. Furthermore, the refinement can focus on areas of the search space where the heuristic is inaccurate, resulting in very targeted refinement spread out over the explored search space. RHC, RHC-SC, and GBFS-SCL are clearly superior to these baselines, dominating coverage over the baselines in almost all domains. The biggest gains are in Barman, Childsnack, Parcprinter, Snake, Storage, and Transport, where even the worst of our online-refinement algorithms beats the best baseline by a substantial margin. One example where online refinement performs poorly is Sokoban. This domain contains many dead ends, so the local lookahead searches of our algorithms frequently fail to yield a better state, and our hill-climbing algorithms need to restart often from getting stuck in those dead ends. Overall, our online-refinement algorithms consistently yield vast improvements over traditional search algorithms with the same heuristic.

4.8.2 Online vs. Offline Conjunctions Quality

If the set of conjunctions C for $h^{C\text{FF}}$ is generated online, then the set of conjunctions at the end of the search is composed of information gained from many different states observed in an actual search. In contrast, if C is generated offline, it only contains information learned from (partially) relaxed plans generated in the initial state. This observation should conclude that the “quality” of an online-generated set of conjunctions C_{on} should be superior to a similar set of conjunctions C_{off} generated offline, i.e., they should result in a heuristic better suited to guide the search.

In order to assess this hypothesis, we compare runs of greedy best-first search with $h^{C\text{FF}}$ using either a set of conjunctions C_{on} generated by one of our online-refinement search algorithms or a set of conjunctions C_{off} of the same size generated via repeated refinement only in the initial state. Figure 4.6 compares the number of expansions of such GBFS searches. In this experiment, C_{on} is generated by RHC, i.e., we consider the final set of conjunctions when RHC finds a solution or reaches the time limit of 30 minutes, and then start GBFS with $h^{C\text{FF}}$ using these conjunctions. We only consider tasks where such C_{on} contains at least one added conjunction for all five random seeds (917 instances). While there is some variance, C_{on} leads to fewer expansions in 333 instances, compared to 207 instances where GBFS with C_{off} is better. Furthermore, the search with C_{on} has a better coverage (680.8 vs. 648.2), and averages 32% fewer expansions on the 544 commonly solved instances. We repeated the experiment with conjunctions generated by RHC-SC and GBFS-SCL, with similar results: 14% fewer expansions and +36.4 coverage (on 885 tasks) for C_{on} resulting from RHC-SC, and 19.5% fewer expansions and +37.8 coverage (on 953 tasks) for conjunctions generated by GBFS-SCL.

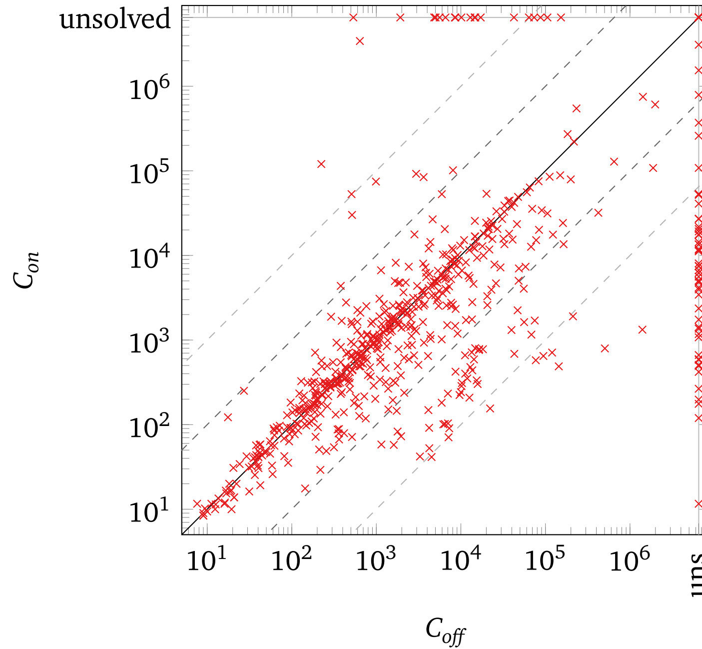


FIGURE 4.6: Expansions for GBFS with h^{CFF} using conjunctions generated online by RHC (C_{on}) vs. conjunctions generated offline in the initial state (C_{off}).

4.8.3 Comparison to the State of the Art

We compare our best-performing search algorithms to the following state-of-the-art satisficing planners:

- *LAMA* [Richter and Westphal, 2010], which runs a greedy best-first search using h^{FF} and a landmark-counting heuristic [Richter et al., 2008] in an alternating queue,
- *Mercury* [Katz and Hoffmann, 2014], which is based on LAMA, replacing h^{FF} by a partial delete relaxation heuristic h^{RB} based on red-black planning [Domshlak et al., 2015],
- *MERWIN* [Katz et al., 2018], which is similar to Mercury but replaces h^{RB} with a novelty heuristic that uses h^{RB} for the underlying estimates [Katz et al., 2017],
- *Dual-BFWS* [Francès et al., 2018; Lipovetzky and Geffner, 2017], a best-first search using a tie-breaking sequence of multiple novelty heuristics based on estimates using delete relaxation, landmarks, and (sub-)goal counting,
- the 2018 version of *Fast Downward Stone Soup (FDSS)* [Seipp and Röger, 2018; Helmert et al., 2011], an anytime portfolio planner running 41 different configurations in an automatically tuned sequence with varying time limits, and
- *Saarplan* [Fickert et al., 2018a], another portfolio planner using several state-of-the-art techniques, including decoupled search [Gnad and Hoffmann, 2018], gray

planning [Speicher et al., 2017], and landmarks, and using an earlier version of Refinement-HC with h^{CFF} as one of its core components.

FDSS and Saarplan additionally use Alcázar and Torralba's [2015] h^2 preprocessor, which can reduce the size of the translated task by pruning operators and facts that are detected to be unreachable.

We include Saarplan not only to provide a comparison point to a state-of-the-art portfolio planner, but also to demonstrate that Refinement-HC can be an effective component inside a portfolio. Saarplan is set up as a sequential portfolio, starting with two different configurations of decoupled search, then switching to gray planning (but only to evaluate the heuristic on the initial state, returning the partially relaxed plan as solution if it is also a plan under non-relaxed semantics) and finally to search with h^{CFF}. This last component first runs Refinement-HC (using BrFS and C-novelty pruning in the lookahead search) until a time bound or a maximum growth factor of 8 is reached for h^{CFF}, using the remaining time for a LAMA-like configuration of GBFS with an alternating queue of h^{CFF} and a landmark heuristic.

Table 4.7 shows the coverage on the IPC benchmarks. Domains that are fully solved by all shown planners are grouped into "Others". While the complex portfolio planners FDSS and Saarplan have the highest overall coverage, all three of our search algorithms with online refinement of the h^{CFF} heuristic solve more instances than LAMA, Mercury, MERWIN, and Dual-BFWS. Our online-refinement planners are particularly effective in the Data Network and Pipesworld (with tankage) domains, where each of them solves more instances than any other planner except Saarplan (though in Data Network, 11 of the 19 solved instances by Saarplan are only solved by its h^{CFF} components). RHC-SC and GBFS-SCL beat all other planners in the Snake domain by (except for Dual-BFWS) significant margins. On the other hand, our planners have comparatively weak performance in Nomystery, where fuel consumption causes issues for delete relaxation heuristics and is hard to capture with conjunctions in h^{CFF}, and Sokoban, which has a large number of dead ends where the lookahead is not effective and our hill-climbing algorithms are frequently trapped.

The last two rows of Table 4.7 additionally show the search time and solution cost as the geometric means across all commonly solved instances (excluding FDSS and Saarplan). We omit the portfolio planners as their search times and solution costs are not comparable since they use additional preprocessing, potentially run many configurations before reaching one that finds a solution, and continue search to improve plans after finding the first solution. Our online-refinement algorithms beat LAMA, Mercury, MERWIN, and Dual-BFWS not only with regard to coverage, but also in search time. GBFS-SCL needs

Coverage	RHC	RHC-SC	GBFS-SCL	LAMA	Mercury	MERWIN	Dual-BFWS	FDSS'18	Saarplan
Agricola (20)	11.6	11.8	12.4	12	9	9	11	**13**	8
Airport (50)	46.4	39.8	41.2	34	35	36	44	**47**	45
Barman (40)	**40.0**	**40.0**	39.4	**40**	36	**40**	**40**	**40**	**40**
Childsnack (20)	9.6	13.4	8.4	6	5	0	8	18	**20**
DataNetwork (20)	16.6	**19.4**	18.6	11	13	16	9	11	19
Depot (22)	**22.0**	**22.0**	**22.0**	19	18	21	**22**	21	**22**
Elevators (40)	**40.0**	**40.0**	**40.0**	**40**	**40**	**40**	**40**	39	**40**
Floortile (40)	**40.0**	**40.0**	**40.0**	8	9	8	5	**40**	**40**
Freecell (80)	78.0	77.4	**80.0**	77	79	**80**	**80**	**80**	79
GED (20)	**20.0**	**20.0**	**20.0**	13	**20**	**20**	**20**	**20**	**20**
Hiking (20)	**20.0**	**20.0**	**20.0**	**20**	11	19	11	**20**	**20**
Logistics (63)	**63.0**	**63.0**	62.4	**63**	**63**	**63**	62	**63**	**63**
Mystery (19)	**19.0**	**19.0**	**19.0**	**19**	16	**19**	**19**	**19**	**19**
Nomystery (20)	10.4	10.0	9.6	11	13	**19**	18	**19**	**19**
Openstacks (90)	89.6	**90.0**	**90.0**	86	88	**90**	89	89	**90**
OrgSynth (20)	**3.0**	**3.0**	**3.0**	**3**	**3**	**3**	**3**	**3**	**3**
OrgSynth-split (20)	2.6	1.6	8.0	11	8	11	**12**	10	9
Parcprinter (40)	**40.0**	39.8	**40.0**	**40**	**40**	**40**	39	**40**	**40**
Parking (40)	**40.0**	19.8	35.6	**40**	34	**40**	**40**	**40**	**40**
Pathways (30)	**30.0**	**30.0**	24.8	23	29	**30**	**30**	**30**	29
Pegsol (35)	**35.0**	33.8	34.6	**35**	**35**	**35**	**35**	**35**	**35**
Pipes-notank (50)	45.6	48.8	49.6	43	43	44	**50**	44	48
Pipes-tank (50)	44.2	47.0	**48.8**	41	40	42	41	43	45
Satellite (36)	**36.0**	31.0	30.8	**36**	**36**	**36**	31	**36**	**36**
Snake (20)	12.4	**18.0**	17.2	4	5	6	15	8	11
Sokoban (30)	11.0	11.4	17.8	**29**	27	26	25	**29**	**29**
Spider (20)	12.2	14.2	15.8	**19**	12	14	16	11	15
Storage (30)	28.8	**30.0**	**30.0**	19	20	24	**30**	25	28
Termes (20)	4.0	9.6	6.8	**14**	13	13	9	12	**14**
Tetris (20)	15.8	**20.0**	19.8	6	14	18	15	19	**20**
Thoughtful (20)	**20.0**	12.0	15.0	15	12	17	19	**20**	**20**
Tidybot (20)	18.0	19.4	**19.8**	17	13	17	18	19	19
TPP (30)	**30.0**	**30.0**	**30.0**	**30**	**30**	**30**	29	**30**	**30**
Transport (60)	54.2	**60.0**	**60.0**	57	**60**	**60**	**60**	57	**60**
Trucks (30)	16.2	18.8	18.4	15	17	21	17	**23**	19
VisitAll (37)	19.2	**37.0**	**37.0**	**37**	**37**	**37**	**37**	**37**	**37**
Woodworking (40)	**40.0**	**40.0**	**40.0**	**40**	**40**	31	24	**40**	**40**
Others (433)	**433.0**	**433.0**	**433.0**	**433**	**433**	**433**	**433**	**433**	**433**
Sum (1695)	1517.4	1534.0	1558.8	1466	1456	1508	1506	1583	**1604**
Normalized (%)	85.0	86.7	88.0	81.8	80.4	85.1	85.0	90.1	**91.9**
Search Time (s)	0.32	0.36	**0.26**	0.34	0.43	0.51	0.33	–	–
Solution Cost	87.4	100.3	94.8	79.7	78.2	76.9	**62.9**	–	–

TABLE 4.7: Coverage on the IPC benchmarks.

Coverage	RHC	RHC-SC	GBFS-SCL	LAMA	Mercury	MERWIN	Dual-BFWS	FDSS'18	Saarplan
Barman (30)	4.4	8.0	6.0	**22**	18	15	4	12	17
Blocksworld (30)	**29.0**	22.8	16.8	22	18	22	9	15	24
Childsnack (30)	7.4	12.0	14.8	11	7	2	8	24	**30**
DataNetwork (30)	26.8	**29.8**	**29.8**	19	15	21	16	19	29
Depot (30)	25.8	**26.0**	**26.0**	18	15	15	20	16	22
DriverLog (30)	**23.8**	12.0	11.6	14	15	15	11	12	20
Elevators (30)	25.0	**30.0**	**30.0**	18	**30**	**30**	28	14	**30**
Floortile (30)	6.8	8.2	8.8	2	2	2	2	7	**9**
Grid (30)	12.0	**21.0**	19.6	15	5	12	14	12	14
Gripper (30)	**30.0**	**30.0**	**30.0**	**30**	**30**	**30**	**30**	**30**	**30**
Hiking (30)	6.0	**28.8**	15.2	15	4	6	6	9	23
Logistics (30)	20.0	21.0	17.6	15	**26**	**26**	12	15	15
Miconic (30)	**30.0**	**30.0**	**30.0**	**30**	**30**	**30**	**30**	**30**	**30**
Nomystery (30)	3.8	3.2	2.6	7	25	**29**	12	**29**	20
Openstacks (30)	13.8	15.0	17.0	13	20	**21**	15	14	19
Parking (30)	17.2	**25.0**	**25.0**	17	17	17	19	13	13
Rovers (30)	**30.0**	27.0	26.8	**30**	25	24	23	**30**	**30**
Satellite (30)	18.0	9.0	9.0	14	**18**	**18**	9	**18**	16
Scanalyzer (30)	**15.0**	**15.0**	**15.0**	**15**	13	13	12	13	12
Snake (30)	25.2	**30.0**	**30.0**	5	4	6	18	8	14
Storage (30)	9.4	14.0	14.0	5	7	9	12	8	**17**
TPP (30)	23.2	**24.0**	**24.0**	20	19	14	9	14	14
Transport (30)	**18.0**	**18.0**	**18.0**	12	16	16	13	13	15
VisitAll (30)	13.6	25.8	27.8	29	23	23	**30**	21	28
Woodworking (30)	**30.0**	13.0	19.8	10	17	6	3	12	29
Zenotravel (30)	**16.0**	**16.0**	**16.0**	**16**	14	14	12	14	13
Sum (780)	480.2	514.6	501.2	424	433	436	377	422	**533**
Search Time (s)	1.07	1.07	**0.95**	1.47	1.01	1.21	1.99	–	–
Solution Cost	140	168	164	127	**112**	123	127	–	–

TABLE 4.8: Coverage on the Autoscale benchmarks.

less time to find solutions than any of these planners in 15 domains (of the 47 domains where these planners have at least one commonly solved instance); averaged across all domains GBFS-SCL is 23% faster than LAMA, 39% faster than Mercury, 49% faster than MERWIN, and 21% faster than Dual-BFWS. On the other hand, our methods result in more expensive plans. However, this disadvantage can potentially be compensated for by running them in an anytime configuration where solutions are continually improved.

Since many domains are fully solved by all state-of-the-art planners (e.g., 11 domains are fully solved by all planners we consider here), we also compare the performance on the Autoscale benchmarks [Torralba et al., 2021]. This benchmark set is designed to be challenging for recent planners through automatic tuning of the parameters of the instance generators, and typically yields a much larger range of coverage values on most domains

when comparing state-of-the-art planners. Table 4.8 shows the results on these benchmarks. Like on the IPC benchmarks, all of our online-refinement planners beat LAMA, Mercury, MERWIN, and Dual-BFWS. FDSS also falls behind on these domains, which can be attributed to two major factors. First, FDSS has been optimized for the IPC domains (before 2018) and thus may not be tuned effectively for this benchmark set. Second, the instances here tend to be larger in size than the ones from the IPC benchmarks, so running many different configurations with very small time limits may not be as effective as running fewer configurations with larger bounds. Saarplan again has the highest coverage overall with 533 solved instances, followed by RHC-SC (514.6 ± 3.2), GBFS-SCL (501.2 ± 3.6), and RHC (480.2 ± 3.9). However, on half of the domains, RHC-SC is the (at least shared) best configuration (in comparison, Saarplan is the best on only 7 out of the 26 domains). All our online-refinement search algorithms are particularly effective in Depot, Snake, TPP, and Transport, where they beat all other considered planners, and partially also in Blocksworld (RHC), DriverLog (RHC), Grid (RHC-SC and GBFS-SCL), Hiking (RHC-SC), Parking (RHC-SC and GBFS-SCL), and Woodworking (RHC). Conversely, Barman and again Nomystery are domains where our planners are generally inferior to the others.

Overall, all of our online-refinement algorithms, in particular RHC-SC and GBFS-SCL, show very competitive performance on both the IPC and Autoscale benchmarks, even compared to state-of-the-art portfolios.

4.9 Related Work

As pointed out before, the heuristic of our choice, h^{CFF}, uses a refinement operation based on counterexample-guided abstraction refinement (CEGAR) to instantiate its set of conjunctions. CEGAR was originally established in model checking [Clarke et al., 2003], and has recently been used in planning to great success. In optimal planning, it most prominently serves as the refinement method for Cartesian abstraction heuristics [Seipp and Helmert, 2013; 2018]. More recently, it has also been shown to be an effective method to instantiate pattern database heuristics [Rovner et al., 2019].

The most closely related approach to ours using online heuristic refinement addresses Cartesian abstraction heuristics in optimal classical planning [Eifler and Fickert, 2018], which has been inspired by our earlier work on Refinement-HC [Fickert and Hoffmann, 2017a]. Like h^{CFF}, their fine-grained CEGAR-based refinement operation makes them a suitable candidate for online refinement. Abstraction heuristics, including those based

on Cartesian abstractions, are most effective when multiple smaller abstractions are combined via cost partitionings [Seipp and Helmert, 2014; 2018]. However, in order to guarantee convergence of the heuristic, abstractions must be merged, which is expensive and diminishes the advantage gained by effective cost partitionings. In practice, the results of this approach have been promising, but have not yet reached the performance of state-of-the-art planners that are based on Cartesian abstractions with offline refinement.

When multiple heuristics are used, their combination can be refined instead of the heuristics themselves. For example, Fink [2007] refines a weighted sum of multiple admissible heuristics for optimal heuristic search. Domshlak et al. [2012] use online learning to obtain a classifier that selects the best heuristic to use in each state. Some approaches for cost partitionings allow selecting the best one from a set of partitionings generated before search [Felner et al., 2004; Karpas et al., 2011; Seipp et al., 2020], or may generate a new partitioning optimized for each state [Katz and Domshlak, 2010; Seipp et al., 2020]. More recently, Seipp [2021] introduced a method to generate additional diverse partitionings online, allowing the search to select the best one in each state for the remainder of the search. However, none of these approaches refine the underlying relaxations, and they do not yield a convergence guarantee.

Apart from refining the heuristic function, other forms of online relaxation refinement exist: Steinmetz and Hoffmann [2017a,b, 2018] use online-refinement of conjunctions to learn a dead-end detector. Another example is Wilt and Ruml's [2013] bidirectional search algorithm, which uses the frontier of the backwards part to improve the heuristic in the forward search component.

Arfaee et al. [2011] learn a heuristic function for large state spaces through a bootstrapping approach that trains a heuristic on increasingly difficult instances. While this is still a form of offline refinement, the authors point out that it can be adapted to solve single instances by interleaving refinement and search by periodically starting a search algorithm in a new thread using the current version of the heuristic. In principle, this strategy could be used with any offline refinement algorithm, but does not constitute online refinement in the sense that the heuristic is static during each search.

In real-time search, per-state updates are a common approach to improve the heuristic and ensure completeness [Korf, 1990; Barto et al., 1995; Bonet and Geffner, 2003]. In particular, the search strategy of LSS-LRTA* [Koenig and Sun, 2009] bears resemblance to Refinement-HC: Each search step first performs a bounded lookahead, after which the heuristic values of the local search space are updated from observations of the frontier. Such per-state updates only correct the heuristic values on states that have been explored, but lack generalization to those that have not been encountered yet as the relaxation

underlying the heuristic remains unchanged. Following the initial work on online refinement of Cartesian abstractions [Eifler and Fickert, 2018], Eifler et al. [2019] have shown that the idea can be effectively transferred to real-time planning, often resulting in better performance due to the added generalization.

Finally, Thayer et al. [2011] provide a simple method to compute a linear correction factor for the heuristic based on observed errors on the search space surface, but this method does not yield convergence guarantees.

4.10 Conclusion

Typically, heuristics are instantiated before starting the search, yet many heuristics offer a refinement operation that can in principle also be applied online. Online refinement has obvious benefits: Computational overhead can be reduced by only refining the heuristic if it is actually necessary, and online refinement can use information gained during the search, allowing the heuristic to adapt to the state space explored by the search. Despite these advantages, online relaxation refinement has barely been addressed before, as critical questions of when and how to refine have no clear answer. The search algorithms we introduced in this chapter use local exploration to evaluate whether the heuristic is sufficiently accurate on the local search space, and use online refinement to escape local minima and plateaus. Converging refinement makes our hill-climbing algorithms complete, and, instantiated with the $h^{C\mathrm{FF}}$ heuristic, makes them competitive with state-of-the-art systematic search approaches. On the IPC and Autoscale benchmarks, our algorithms with $h^{C\mathrm{FF}}$ online refinement substantially beat related state-of-the-art planners, and are highly competitive with complex portfolios where they can also be used as a strong component.

One avenue for future work is to combine online relaxation refinement for the heuristic with similar refinement operations for other purposes. A straightforward example is nogood learning, choosing new conjunctions for $h^{C\mathrm{FF}}$ so as to be able to prune more dead-end states [Steinmetz and Hoffmann, 2017b; 2018]. Another example, in the specific arrangement of our techniques relying crucially on novelty pruning, is conjunction learning for novelty pruning. While, here, we already use C-novelty pruning with conjunctions added during the search, we use the conjunctions that were selected to be useful for $h^{C\mathrm{FF}}$, without any information flow specific to novelty pruning. It remains an open question how to identify conjunctions that are effective for novelty, learning conjunctions specifically for that purpose. This could be a promising way to improve planners such as MERWIN and Dual-BFWS, that rely on novelty measures as the main function to guide the search.

5 Ranking Conjunctions for Partial Delete Relaxation Heuristics

Except for Haslum's [2012] initial work, all existing approaches for partial delete relaxation with explicit conjunctions [e.g., Keyder et al., 2014; Fickert et al., 2016; Fickert and Hoffmann, 2017a; see also Chapter 4] use Keyder et al.'s [2014] method based on counterexample-guided abstraction refinement to generate the set of conjunctions C. At each refinement step, the procedure generates a set of candidate conjunctions C_{cand} addressing specific flaws in the current partially relaxed plan, and one of the candidates is selected to be added to C. The set of candidates can be huge, and in some domains millions of candidates may be generated in one iteration. As observed early on [Haslum, 2012], the choice of conjunctions is important for the overall performance of the heuristic, and small changes can have a large impact on the results. The previous literature merely suggests one simple ranking strategy, preferring conjunctions incurring a low computational overhead [Keyder et al., 2014]. Furthermore, many candidate conjunctions may incur the same overhead, so even when following that ranking a large selection of "best" candidates may remain.

Here, we address the candidate selection in more detail. We explore a vast range of ranking strategies, aiming not only to minimize overhead, but also to capture the relative "importance" of conjunctions. We systematically evaluate the performance of these strategies, both individually and as part of lexicographic tie-breaking sequences. Furthermore, we devise ranking strategies for conjunctions that have already been added to C, which allows taking into account observations and statistics during search. This can be used to forget conjunctions, thereby reducing computational overhead, and allowing new conjunctions to be added to address flaws in more recently computed partially relaxed plans. Finally, we devise a new variant of the conflict extraction preceding the candidate ranking, which resolves a major bottleneck in many standard planning domains. On the IPC benchmarks, we find that the different ranking strategies lead to a large variance in

performance, and that our new strategies can be useful. Furthermore, we devise a simple periodic conjunction-forgetting strategy for greedy best-first search, which significantly improves results compared to leaving the heuristic unchanged.

Papers and Contributions This chapter is based on the paper "Ranking Conjunctions for Partial Delete Relaxation Heuristics in Planning" [Fickert and Hoffmann, 2017b]. The evaluation has been expanded and updated for recent developments in partial delete relaxation with explicit conjunctions (cf. Chapter 4).

5.1 Candidate Ranking Strategies

In each iteration, Keyder et al.'s [2014] refinement algorithm yields a set of candidate conjunctions C_{cand} that are not yet contained in the set of conjunctions C used by the heuristic. While it is possible to add all the candidate conjunctions to C at once, in practice it is better to only add a single conjunction to minimize the computational overhead and avoid redundant conjunctions addressing the same conflicts [Keyder et al., 2014]. However, in order to select a suitable candidate, the conjunctions must be ranked according to some criteria. We now discuss a range of ranking strategies. These strategies can be combined using lexicographic tie-breaking, which we denote as $\langle s_1, s_2, \ldots \rangle$ (the ranking strategy s_1 is applied first, then s_2 is used to break the remaining ties, and so on).

Keyder et al. [2014] introduced the following ranking strategies:

min-distance Rank the candidates based on the number of vertices between the deleter and the failed action in the BSG for sequential conflicts (e.g., if the deleter and failed action are ordered immediately after each other, this value is 0). For parallel conflicts this is set to 1.

min-counters Rank the candidates based on the number of additional counters that need to be tracked by the heuristic. For a candidate conjunction c, this is computed as the number of actions over which c is regressable, i.e., $|\{a \in \mathcal{A} \mid R(c,a) \neq \bot\}|$.

The intuition of the *min-distance* strategy is that preferring conflicts with minimal distance between the deleter and the failed action tends to yield more direct and hence more relevant conflicts. Preferring conflicts with minimal distance in the best supporter graph has another practical side effect: Conflicts with lower distance are computationally easier to find—so if those are preferred anyway the algorithm can terminate earlier, speeding up the conflict extraction process.

Keyder et al. [2014] combine these two strategies as ⟨*min-distance*, *min-counters*⟩: From the conflicts with minimal distance, the one that introduces the minimal number of new counters is chosen in order to minimize the computational overhead.

The overhead is not the only important metric when considering which conjunctions to add to C—some conjunctions might yield a more informative heuristic than others, which can offset some overhead. We introduce novel strategies with very different approaches in the following, and then explain their underlying motivation.

5.1.1 Ranking Strategies

We consider the following candidate ranking strategies:

random Rank the candidates in a random order.

arbitrary Rank the candidates in an arbitrary order (in our implementation, we leave them in the order they are generated).

min-size/max-size Rank the candidates according to their size (number of facts).

min-counters-estimate Rank the candidates by their *estimated* number of additional counters. Given a candidate conjunction c, we compute this estimate as $\sum_{f \in c} |\{a \in \mathcal{A} \mid R(\{f\}, a) \neq \bot\}|$.

max-occurrences Prefer candidates that have been added to C_{cand} multiple times for different conflicts.

min-influence/max-influence Rank the candidates based on the number of counters where the conjunction will appear as a precondition when added to C.

min-deleter-alternatives/max-deleter-alternatives Rank the candidates based on how many of the deleter's supported conjunctions could be achieved equally well by different actions. Specifically, we calculate this as the percentage of supported conjunctions of the deleter that have other best supporters that do not delete the missing precondition of the failed action.

max-cost-increase Rank the candidates based on how much the $h^{C\text{add}}$ value of the resulting conjunction would increase over its dominated conjunctions in the current state.

These strategies range from trivial methods (*random* and *arbitrary*), over strategies taking account simple features of the candidate conjunctions (such as *min-size*), to more complex strategies analyzing certain aspects of the considered candidate conjunctions. We next discuss the ideas behind the non-trivial strategies.

5.1.2 Motivation

The *min-counters-estimate* strategy is an attempt to reduce the computational overhead of *min-counters*: The latter requires computing $R(c, a)$ for all candidates $c \in C_{cand}$ and actions $a \in \mathcal{A}$. On the other hand, we only need the regressions for all unit conjunctions for *min-counters-estimate*—which are already available since the unit conjunctions are always contained in C.

The *max-occurrences* strategy arises from the observation that different conflicts may yield the same candidate conjunctions. These candidates can fix multiple conflicts at once, so intuitively, adding them to C should more effectively reduce conflicts in partially relaxed plans computed afterwards.

The *influence*-based strategies consider how many existing counters will be affected by a new conjunction, as an attempt to estimate their influence on the overall heuristic.

The strategies considering the *deleter alternatives* attempt to predict the changes to the partially relaxed plans when a conjunction is added. If there are no alternative actions to achieve the conjunctions supported by the deleter, the structure of the next relaxed plan is likely to change more compared to the deleter just being exchanged for a different action.

Finally, the *max-cost-increase* prioritizes conjunctions that are more difficult to reach than its subconjunctions. Such conjunctions can directly lead to an increase in the overall heuristic value if they must be included in a partially relaxed plan.

5.1.3 Practical Remarks

Some ranking strategies require additional computation or memory. For example, the *min-counters* strategy requires computing $R(c, a)$ for all candidate conjunctions $c \in C_{cand}$ and actions $a \in \mathcal{A}$, and for the *max-occurrences* strategy we have to keep track of the number of times each conjunction is added to the set of candidates. Our implementation avoids this overhead whenever it is not needed. For example, if the *min-counters* strategy only appears as the second tie-breaker, it is only evaluated for the best candidate conjunctions according to the first ranking strategy. Similarly, if *max-occurrences* is not used as a ranking strategy, we do not need to count the number of times the same candidate is added to C_{cand}.

5.2 Online Ranking Strategies

In addition to ranking candidates that are not yet in C, we can also devise rankings for conjunctions that are already used by the heuristic. For these *online ranking strategies*, we can take features and statistics into consideration that can only be measured during search. This way, we may get more detailed information about the conjunctions. Online ranking of conjunctions in C can be useful to detect conjunctions that turned out not to be useful in the search, so removing them from C may reduce the computational overhead without significantly reducing the informativeness of the heuristic. We will evaluate these strategies by periodically replacing existing conjunctions in C (those that deemed worst according to the online ranking strategies) by new ones.

5.2.1 Strategies

We consider the following strategies:

random Rank the conjunctions randomly.

oldest Rank the conjunctions by how long they have been contained in C.

max-counters Rank the conjunctions by the number of attached counters.

min-rp-frequency Rank the conjunctions by how frequently they appear as a supported conjunction of any action occurrence in a partially relaxed plan. Specifically, we consider the percentage of generated relaxed plans containing this conjunction since it was added to C.

min-$h^{C\mathrm{add}}$-increase Rank the conjunctions according to how frequently their $h^{C\mathrm{add}}$ value is greater than that of its dominated conjunctions (as a percentage of the number of evaluations since this conjunction was added to C).

Additionally, we introduce a more complex strategy based on the *effectiveness* of a conjunction, defined as follows:

Definition 5.1 (Conjunction Effectiveness)**.** A conjunction c is called *effective* in a state s if all its subconjunctions $c' \subset c, c' \in C$ have $h^{C\mathrm{add}}(s,c') < h^{C\mathrm{add}}(s,c)$, and either

1. $c \subseteq \mathcal{G}$ and if $h^{C\mathrm{add}}(s,c) = \infty$, then all other goal conjunctions $c' \subseteq \mathcal{G}, c' \in C, c' \neq c$ have $h^{C\mathrm{add}}(s,c') \neq \infty$, or
2. there exists a counter attached to a conjunction c' and action a with $c \subseteq R(c',a)$, such that either

(a) $h^{\mathrm{Cadd}}(s,c) < \infty$ and a is a best supporter of c', or

(b) $h^{\mathrm{Cadd}}(s,c) = \infty, h^{\mathrm{Cadd}}(s,c') = \infty$, and for all conjunctions $c'' \in R(c',a)^C, c'' \neq c$ we have $h^{\mathrm{Cadd}}(s,c'') \neq \infty$.

A conjunction c is considered effective in a state s if its h^{Cadd} value increases over that of its subconjunctions $c' \subset c$, and the conjunction contributes in some way to the overall h^{Cadd} value. This contribution means that c should either be part of the goal, or be required as a precondition for another conjunction. If c is unreachable ($h^{\mathrm{Cadd}}(s,c) = \infty$), it must make either the goal or some other conjunction unreachable.

This yields the following ranking strategy:

min-effective Rank the conjunctions according to their effectiveness (as a percentage of the number of evaluations where a given conjunction was effective according to Definition 5.1 since it was added to C).

5.2.2 Motivation

Since *min-counters* is already a successful ranking strategy for refinement, it appears logical that during search, replacing conjunctions with many attached counters by new ones with fewer counters can be beneficial.

The *min-rp-frequency* strategy considers how often conjunctions appear in the partially relaxed plans, assuming that conjunctions that are used in the relaxed plans often are more useful than others.

The *min-h^{Cadd}-increase* and *min-effective* strategies represent the attempt to capture the importance of a conjunction. The former is rather simple, and just considers if the conjunction tends to be more difficult to achieve than its subconjunctions. The latter additionally requires the conjunction to be used for something: either as part of the goal; or in a precondition of a counter attached to another conjunction, and the other conjunction is reached through that counter.

5.3 Conflict Extraction Algorithm

We finally designed a variant of the conflict extraction step, which precedes the selection of candidate conjunctions.

The refinement algorithm introduced by Keyder et al. [2014] searches conflicts in the best-supporter graph. Since the BSG models any valid ordering of the relaxed plan, this conflict extraction method can lead to a very high number of conflicts (thus taking a lot of time), with many of them not actually appearing in the ordering returned by the heuristic. We suggest a slightly different approach and extract the conflicts directly from the sequenced relaxed plan instead, only using the BSG to identify the conflict type and generate the candidate conjunction accordingly. In the experiments section we will show that this is significantly reduces the worst-case runtime of the refinement procedure on the IPC benchmarks, while retaining similar informativeness of the resulting conjunctions.

Keyder et al.'s [2014] implementation of the conflict extraction is optimized for their lexicographic tie breaking: First, only direct sequential conflicts (with a distance of zero between the deleter and failed action in the BSG) are collected. Only if no such conflicts are found, the algorithm proceeds with the computationally much more difficult procedures to collect all other sequential conflicts and parallel conflicts. We also apply this optimization in our conflict extraction method whenever *min-distance* is used as the primary ranking strategy.

5.4 Experiments

We follow the setup described in Section 4.2: The benchmark set consists of all unique STRIPS instances of the satisficing tracks up to and including the 2018 IPC, and average results over 5 random seeds. The source code of our ranking strategies is included in the same repository at `https://github.com/fickert/fast-downward-conjunctions`.

In our evaluation of the candidate ranking strategies, we focus on the online-refinement search algorithms introduced in the previous chapter. To evaluate the online ranking strategies, we use a simple replacement strategy in GBFS. We enable helpful actions pruning in all search algorithms based on hill climbing, and use a dual queue for preferred operators in GBFS-based algorithms. Unless noted otherwise, the default lexicographical tie-breaking for candidate conjunctions is ⟨*min-distance*, *min-counters*⟩, i.e., the one introduced by Keyder et al. [2014].

The online-refinement search algorithms add conjunctions whenever the heuristic is inaccurate, failing to find a state with lower heuristic value in a bounded lookahead. When the search terminates, the number of added conjunctions during the search is a useful metric to judge the informativeness gained by the added conjunctions: if fewer conjunctions were added with ranking strategy A than with ranking strategy B, then strategy A is more effective in improving the accuracy of the heuristic. Another useful statistic is

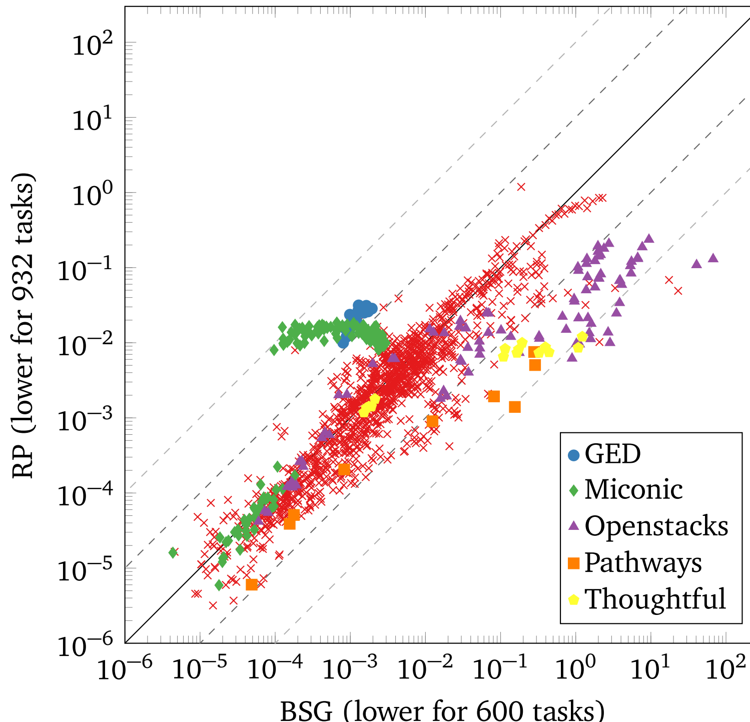


FIGURE 5.1: Average time of the conflict extraction step (in seconds) of the refinement procedure when extracting conflicts from the BSG (x-axis) vs. relaxed plan (y-axis).

the growth factor (the increase in the number of counters that the h^{CFF} implementation needs to keep track of, see Section 3.2.1) as a metric for the increase in computational complexity of the heuristic.

We start out by assessing the impact of our changes to the conflict extraction algorithm. Then we evaluate the different ranking strategies; first considering the candidate ranking strategies and combinations of them before we turn to the online ranking strategies.

5.4.1 Conflict Extraction Algorithm

The main motivation to extract conflicts from the sequentialized relaxed plan instead of the best supporter graph is to reduce the computational effort by focusing on the subset of possible conflicts that appear in the concrete relaxed plan returned by h^{CFF}. In addition to analyzing the computational aspect, we will investigate whether this change affects the informativeness of the resulting conjunctions.

In order to evaluate the computational effort of the conflict extraction phase, we ran offline refinement for h^{CFF} with either conflict extraction algorithm for 15 minutes (or until the current partially relaxed plan is a real plan). Figure 5.1 compares the average time of each conflict extraction operation between the two variants, highlighting domains where the difference is particularly pronounced (discussed below). Extracting conflicts

from the relaxed plan is often faster, but not always: With the default ranking strategy, the conflict extraction procedure can stop early if zero-distance conflicts are found. If, due to the specific ordering of the relaxed plan, zero-distance conflicts only appear when considering all possible orderings in the BSG, then extracting conflicts from the BSG is more efficient. This happens particularly frequently in GED and (to a lesser degree) in Miconic, where extracting conflicts from the BSG is on average 17.7 respectively 6.8 times faster than extracting them from the relaxed plan. On most domains though, the conflict extraction from the sequentialized relaxed plan is faster, most prominently in Pathways (16x faster), Thoughtful (11.7x), and Openstacks (10.4x). Averaged across all domains, our method yields a speedup of 32% over Keyder et al.'s [2014] approach. Furthermore, it brings down the worst measured conflict extraction time (instance average) from 67.4 to 1.2 seconds.

Figure 5.2 shows the number of added conjunctions during a run of the state-of-the-art online-refinement search algorithms from Chapter 4 (RHC, RHC-SC, and GBFS-SCL). For all three algorithms, there is no clear advantage for either conflict-extraction method, and the difference is generally small: on average, conflict extraction from the relaxed plan results in 9.5% more refined conjunctions in RHC, 4.0% fewer in RHC-SC, and 2.8% fewer in GBFS-SCL. The highlighted domains in Figure 5.2 are cases where we can see a slight advantage for either approach: Spider and Parcprinter are examples where our method works slightly better, conversely, the BSG-based approach is better in Freecell and Trucks. In terms of coverage, we found no significant difference between the different conflict extraction strategies with any of these search algorithms: considering only instances where both configurations generate at least one conjunction, we get 936.2 ± 4.2 (relaxed plan) vs. 940.8 ± 4.1 solved instances with RHC, 1018.2 ± 4.6 vs. 1023.4 ± 4.0 with RHC-SC, and 1000.4 ± 4.3 vs. 999.8 ± 4.1 with GBFS-SCL.

In conclusion, extracting conflicts from the relaxed plan is typically faster than considering all conflicts in the best supporter graph (apart from cases where the BSG consistently yields zero-distance conflicts which are not contained in the sequentialized relaxed plan), and significantly improves the worst-case runtime. Our evaluation of the difference in informativeness of the resulting conjunctions shows no significant advantage for either approach (though there are some domains where one or the other has a small advantage). For the remaining experiments in this chapter, we use our method of extracting conflicts from the relaxed plan.

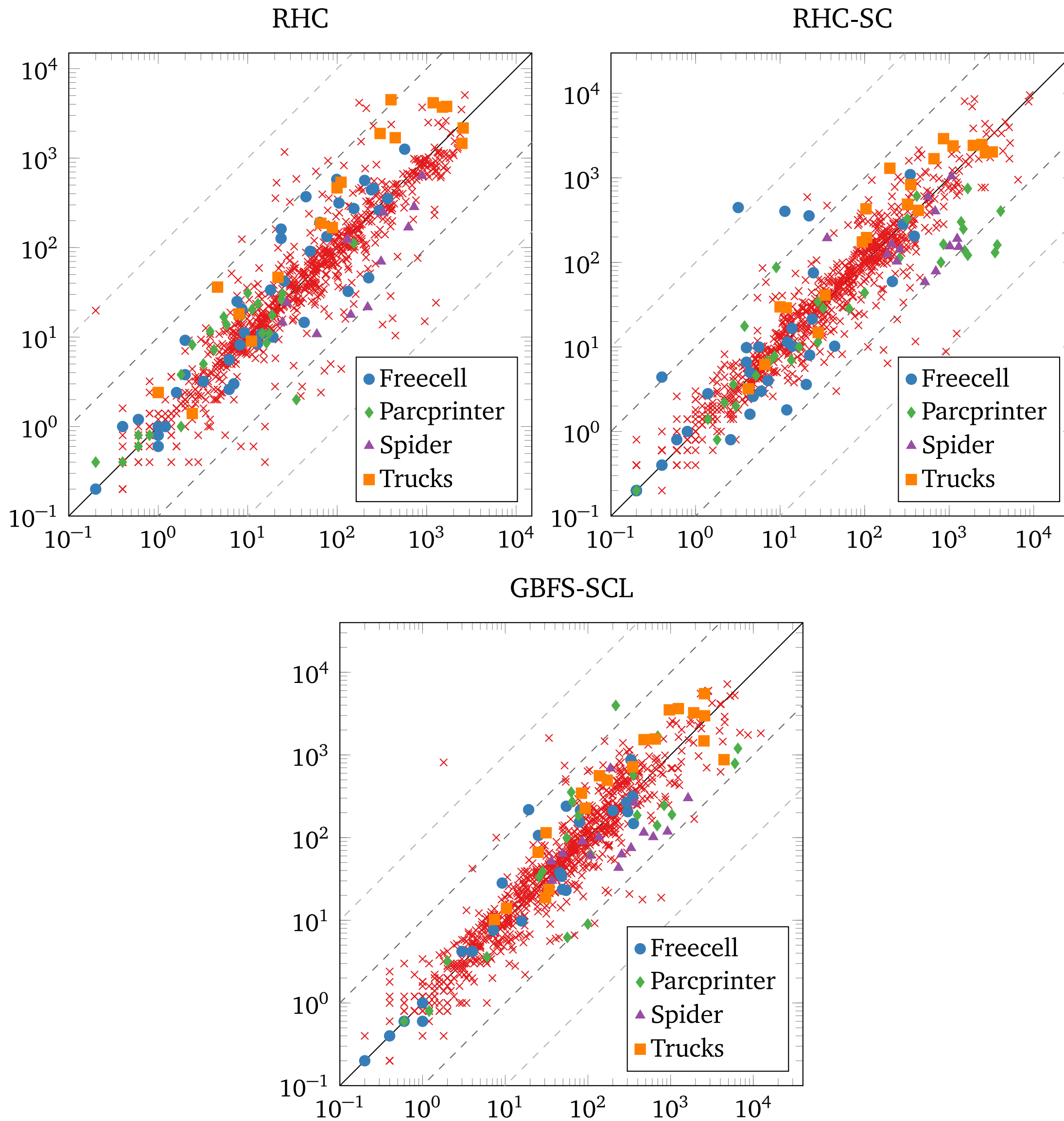


FIGURE 5.2: Number of conjunctions added during the search when extracting conflicts from the BSG (x-axis) vs. relaxed plan (y-axis).

5.4.2 Candidate Ranking Strategies

We next evaluate the different candidate ranking methods, starting with the basic strategies before considering their combinations. Our evaluation uses Refinement-HC with the most restrictive depth bound of 1 (cf. Section 4.4.2). We choose this search algorithm instead of the stronger variants (RHC with novelty pruning, RHC-SC, or GBFS-SCL) because it triggers refinement more often, resulting in (a) more instances where at least one conjunction is added, yielding a larger benchmark set where the ranking strategy can make a difference, and (b) more conjunctions being added in those instances, increasing the difference between the strategies. For example, the difference in overall coverage between the best and worst strategies in the following experiment is 192, but would have been

Ranking Strategy	Coverage	Conjunctions	Growth Factor	Ties
⟨*min-distance*⟩	1287.2 ± 6.6	30.4	2.01	9.2
⟨*min-size*⟩	1234.8 ± 7.2	42.0	2.31	12.6
⟨*min-counters*⟩	1206.8 ± 6.5	46.9	1.99	3.1
⟨*random*⟩	1206.8 ± 6.5	43.7	2.41	–
⟨*min-influence*⟩	1188.0 ± 6.9	36.1	2.29	3.1
⟨*arbitrary*⟩	1185.4 ± 6.1	51.7	2.74	–
⟨*min-deleter-alternatives*⟩	1180.4 ± 6.9	63.3	3.33	20.7
⟨*max-cost-increase*⟩	1148.8 ± 6.0	60.3	3.02	3.0
⟨*max-deleter-alternatives*⟩	1144.4 ± 6.7	72.1	3.73	6.4
⟨*min-counters-estimate*⟩	1135.6 ± 7.1	55.0	2.34	3.2
⟨*max-influence*⟩	1114.6 ± 6.9	63.0	3.12	5.3
⟨*max-occurrences*⟩	1105.6 ± 7.1	89.2	4.43	3.8
⟨*max-size*⟩	1095.0 ± 7.3	81.6	4.31	7.6

TABLE 5.1: Overview of the results with the basic candidate ranking strategies in Refinement-HC with a depth bound of one (sorted by coverage).

just 97 with GBFS-SCL. We found that the relative effectiveness of the ranking strategies is mostly consistent across the different search algorithms, so the observations made here also transfer to the state-of-the-art online-refinement search algorithms.

Table 5.1 shows an overview of the results for the candidate ranking strategies; remaining ties are broken arbitrarily. The table shows the coverage, and geometric means of the number of added conjunctions on commonly solved instances with non-empty C, growth factor of the heuristic, and number of tied candidate conjunctions after applying the given ranking strategy. The choice of the strategy has a huge effect on the overall coverage, ranging from 1095.0 ± 7.3 to 1287.2 ± 6.6. Overall, *min-distance* clearly outperforms the other ranking strategies, and requires the fewest refinement steps in Refinement-HC.

Interestingly, the *max-size* strategy results in the least informative conjunctions while inducing the greatest computational overhead. One reason for this behavior is that this strategy favors a set of conjunctions that is focused on few facts rather than a universally useful one, as new conjunctions that dominate already existing ones are prioritized over conjunctions that contain facts for which no (non-singleton) conjunction exists yet. Conversely, *min-size* yields much better results, and is only second to *min-distance* in terms of absolute coverage.

Like the *min-counters* strategy, the approximate version *min-counters-estimate* also reduces the overhead of the added conjunctions (e.g., *random* and *arbitrary* add fewer conjunctions yet yield a larger growth factor in the heuristic). However, it performs significantly worse, and yields less informative conjunctions than *min-counters*. We found

Coverage	⟨*min-distance*⟩	⟨*min-size*⟩	⟨*min-counters*⟩	⟨*random*⟩	⟨*min-influence*⟩	⟨*arbitrary*⟩	⟨*min-deleter-alt.*⟩
Airport (50)	**41.8**	36.4	34.4	39.8	34.4	25.2	33.8
Barman (40)	**6.8**	0.4	2.4	2.2	1.2	0.0	0.8
Blocksworld (35)	34.2	28.4	34.8	34.8	33.0	**35.0**	32.8
Childsnack (20)	3.4	3.0	**6.0**	**6.0**	3.4	1.0	4.0
DataNetwork (20)	16.2	**17.0**	10.8	14.6	**17.0**	9.6	13.4
Floortile (40)	**40.0**	5.6	**40.0**	13.8	7.4	36.0	8.6
Freecell (80)	**72.4**	70.0	65.0	70.8	67.2	63.4	69.0
Hiking (20)	19.8	19.6	19.6	**20.0**	**20.0**	14.2	**20.0**
Miconic (150)	**150.0**	**150.0**	70.4	**150.0**	**150.0**	**150.0**	**150.0**
Nomystery (20)	8.0	7.8	5.2	6.8	8.0	**17.0**	5.8
Openstacks (90)	**73.4**	69.4	54.6	44.6	53.0	40.4	68.8
Parking (40)	6.4	9.8	**15.8**	3.8	4.8	0.4	1.0
Pegsol (35)	26.6	19.2	**34.2**	26.2	15.6	33.0	19.0
Pipes-tank (50)	**42.0**	40.8	41.6	39.8	40.4	34.0	36.6
Storage (30)	27.4	29.2	**29.6**	27.2	26.8	24.4	28.0
Tetris (20)	**18.0**	14.6	15.8	15.4	10.4	10.8	12.0
Tidybot (20)	7.2	9.2	**10.2**	7.0	7.6	3.8	4.4
Transport (60)	38.8	53.6	**59.2**	35.6	39.6	36.6	37.6
Trucks (30)	10.0	10.0	**13.8**	9.8	9.4	12.0	8.4
VisitAll (37)	8.8	10.4	11.0	4.0	10.6	**11.6**	3.4
Others (808)	**636.0**	630.4	632.4	634.6	628.2	627.0	623.0
Sum (1695)	**1287.2**	1234.8	1206.8	1206.8	1188.0	1185.4	1180.4
Normalized (%)	**69.3**	66.1	68.5	66.0	64.3	64.4	63.1

TABLE 5.2: Coverage results with selected candidate ranking strategies. Domains where the coverage difference between the best and worst strategy is less than 5 are grouped to "Others".

that the intended advantage of saving some computation time does not really have an effect as the evaluation of the ranking strategies takes only a very small share of the overall search time, in particular with our adapted conflict extraction method.

Table 5.2 shows the coverage of the best-performing strategies in more detail. In some domains results are very consistent between the different ranking strategies, in others they can fluctuate wildly. For example, in Floortile, coverage ranges between 5.6 all the way to 40.0. Nomystery is another interesting example: The *arbitrary* strategy works much better than the others in that domain. In our implementation, arbitrary simply chooses the first conflict, which tends to occur early in the relaxed plan. Since the main difficulty in Nomystery is fuel consumption, this results in conjunctions that are immediately useful. Conflicts further along the relaxed plan might result in conjunctions that are never reachable, such as the current value of the fuel combined with a truck location that

Coverage	⟨*md*, *min-counters*⟩	⟨*md*, *arbitrary*⟩	⟨*md*, *min-deleter-alt.*⟩	⟨*md*, *min-size*⟩	⟨*md*, *min-counters-est.*⟩	⟨*md*, *min-influence*⟩	⟨*md*, *random*⟩
Airport (50)	**43.4**	41.8	42.2	41.2	40.2	41.8	41.0
Barman (40)	2.2	**6.8**	5.4	0.8	1.2	3.4	3.4
DataNetwork (20)	14.4	16.2	16.8	**17.6**	13.8	**17.6**	16.0
Freecell (80)	68.8	**72.4**	71.4	71.0	70.8	68.4	69.2
Openstacks (90)	56.4	**73.4**	71.2	60.8	54.2	56.0	45.4
Parking (40)	**10.2**	6.4	4.0	7.6	8.2	4.4	0.8
Pegsol (35)	29.8	26.6	28.4	27.4	**30.6**	23.2	27.0
Transport (60)	**56.8**	38.8	38.6	42.0	37.8	38.8	36.6
Trucks (30)	**14.4**	10.0	10.0	11.0	11.8	10.0	11.6
VisitAll (37)	**11.4**	8.8	8.2	10.8	**11.4**	10.4	8.4
Others (1213)	**992.8**	986.0	989.2	986.0	991.2	987.2	984.6
Sum (1695)	**1300.6**	1287.2	1285.4	1276.2	1271.2	1261.2	1244.0
Normalized (%)	**70.5**	69.3	69.6	69.3	69.2	68.7	67.9
Conjunctions	64.6	59.0	58.5	59.0	65.0	58.5	60.6
Growth Factor	2.34	2.60	2.59	2.52	2.43	2.54	2.66

TABLE 5.3: Coverage results with selected candidate ranking strategies as lexicographical tie breaking after *min-distance*. Domains where the coverage difference between the best and worst strategy is less than 3 are grouped to "Others".

requires the truck to move (and thereby consume some fuel) first. While *min-distance* is outperformed in a few domains (most significantly in Transport and Parking), it is otherwise very robust and yields the best results overall.

As the overall best ranking strategy (*min-distance*) leaves some room for additional tie breaking (see Table 5.1), we next look at combinations of strategies.

We ran all remaining candidate ranking strategies as lexicographical tie breaking after *min-distance*, with results ranging from a total coverage of 1228.2 ± 5.7 (with the tie-breaking sequence ⟨*min-distance*, *max-influence*⟩) to 1300.6 ± 6.3 (with ⟨*min-distance*, *min-counters*⟩). Table 5.3 shows the results of the best-performing configurations. In most domains, the variance across the ranking strategies is now small, but the difference is still quite large in Openstacks and Transport. The latter is the domain where the overall best configuration stands out the most: the second-best tie breaking sequence (⟨*min-distance*, *min-size*⟩) solves 14.8 fewer instances than ⟨*min-distance*, *min-counters*⟩.

While Keyder et al.'s [2014] original strategy is the best overall, there are some domains where our novel strategies have merit. In DataNetwork and Pegsol, other tie breaking

sequences yield higher coverage (see Table 5.3). However, there are also domains where coverage remains similar, but the number of expansions reduces significantly with different strategies: in Nomystery, ⟨*min-distance*, *arbitrary*⟩ has 50% fewer expansions than ⟨*min-distance*, *min-counters*⟩; in Miconic, ⟨*min-distance*, *min-influence*⟩ has 44% fewer expansions.

In most domains, no further tie breaking can be applied. We tested using a third ranking strategy after ⟨*min-distance*, *min-counters*⟩ in selected domains that had remaining ties, but found no significant impact on the results.

5.4.3 Online Ranking Strategies

We now consider the strategies to rank conjunctions that are already contained in C and used by the heuristic. We evaluate the online ranking strategies using GBFS with offline refinement until a growth factor of 1.5 or a timeout of 900 seconds is reached, and periodically (after every 25 evaluations) replace a single conjunction by a new one. The conjunction to be replaced is selected according to each specific online ranking strategy, while the new conjunction is selected using the default candidate selection strategy, ⟨*min-distance*, *min-counters*⟩. We require that the removed conjunction must have been part of C for at least 250 evaluations (except in the beginning before the threshold of 250 evaluations is reached for any conjunction) to avoid removing conjunctions that have recently been added. After replacing a conjunction, search is continued with the open and closed lists unchanged.

Table 5.4 shows an overview of the results. The best strategy overall is *oldest* with a total coverage of 1450.2 ± 5.7. The *max-counters* succeeds in reducing the computational effort of the heuristic, but yields a less informative heuristic than the other replacement strategies (higher number of expansions), which outweighs the computational advantage. As shown by the search time, the more complex *min-*$h^{C\text{add}}$*-increase* and *min-effective* strategies incur additional overhead, which does not pay off in increased informativeness over the simpler strategies.

All of our replacement strategies yield better results than keeping the set of conjunctions fixed (rightmost column), boosting not only overall coverage, but also significantly reducing the number of expansions on commonly solved instances. Since the heuristic steadily includes new conjunctions, it adapts itself to the region of the search space that is currently being explored. Conjunctions are often only relevant in some areas of the search space, but fail to improve the informativeness of the heuristic in others while inducing computational overhead. The *oldest* strategy is the most effective one in addressing this

Coverage	*oldest*	*random*	*min-rp-frequency*	*max-counters*	*min-$h^{C^{add}}$-increase*	*min-effective*	fixed *C*
Airport (50)	**37.4**	37.2	35.4	35.2	33.4	32.8	34.2
Barman (40)	**34.6**	33.8	25.4	28.8	19.4	19.6	5.4
DataNetwork (20)	13.0	13.6	14.8	14.0	**16.4**	14.6	13.2
Openstacks (90)	**68.2**	61.8	60.0	56.6	51.4	52.8	66.0
OrgSynth-split (20)	10.6	10.6	11.0	**11.2**	7.0	7.4	10.4
Parcprinter (40)	**37.0**	34.2	35.4	34.8	34.6	34.8	32.8
Parking (40)	**26.6**	23.8	24.6	21.8	22.0	16.4	19.4
Pipes-tank (50)	**43.0**	42.8	**43.0**	42.4	42.6	42.8	39.0
Snake (20)	**11.4**	10.2	**11.4**	8.2	9.8	10.8	6.2
Storage (30)	**27.2**	**27.2**	25.6	24.2	26.8	26.0	20.0
Tetris (20)	19.0	18.8	**19.8**	13.4	9.2	9.0	13.8
Transport (60)	40.0	40.4	39.8	**47.4**	41.0	42.6	38.8
Trucks (30)	18.2	17.0	**19.0**	16.2	15.6	16.6	16.8
VisitAll (37)	17.0	17.0	16.8	17.0	12.2	12.8	**18.2**
Others (1148)	1047.0	1048.6	**1049.8**	1047.6	1046.4	1046.2	1038.4
Sum (1695)	**1450.2**	1437.0	1431.8	1418.8	1387.8	1385.2	1372.6
Std. Error	5.7	6.0	6.0	6.4	6.3	6.6	7.0
Normalized (%)	**82.4**	81.9	82.0	80.5	79.0	78.8	77.6
Growth Factor	1.55	1.53	1.59	**1.31**	1.50	1.54	1.45
Expansions	323.16	321.63	**320.12**	361.70	349.12	330.25	444.89
Search Time (s)	**0.50**	0.51	**0.50**	0.52	0.81	0.79	0.57

TABLE 5.4: Coverage results for the online ranking strategies in GBFS with offline refinement and periodic replacement, and GBFS with a fixed set of conjunctions. Domains where the coverage difference between the best and worst strategy is less than 3 are grouped to "Others". The last three rows show the average growth factor, and geometric means of the number of expansions and search time on commonly solved instances.

problem, as the oldest conjunctions were likely added in now distant parts of the search space and should thus be less relevant in recently generated relaxed plans.

In domains where GBFS with a fixed set of conjunctions is already better than the online-refinement algorithms (cf. Table 4.6 in Section 4.8.1), the replacement variant increases the advantage slightly. For example, it improves coverage in Sokoban from 25.4 to 27.2 (where the best-performing online-refinement algorithm solves 17.8 instances), and in Termes from 11.8 to 12.6 (compared to at most 9.6 with online refinement). Overall, this simple replacement strategy is already competitive with planners like LAMA [Richter and Westphal, 2010] and Mercury [Katz and Hoffmann, 2014], which achieve a coverage of 1466 respectively 1456 on this benchmark set (cf. Table 4.7 in Section 4.8.3), though it falls behind more recent planners and the state-of-the-art online refinement methods.

5.5 Conclusion

Previous work on partial delete relaxation with explicit conjunctions has glossed over the detail of how to rank conjunctions in the refinement procedure. We filled that gap with an extensive evaluation of complex strategies and their combinations, both for the selection of candidate conjunctions, and to rank conjunctions that are already used by the heuristic. As it turns out, the initial strategies suggested by Keyder et al. [2014] are already very effective—not just because they reduce the computational overhead as intended, but also because they result in very informative conjunctions. However, there are domains where our novel strategies can be superior. We evaluated the ranking of already added conjunctions with a simple online-replacement mechanism in GBFS, and achieved surprisingly good results by simply replacing the oldest conjunction repeatedly as it dynamically adapts the heuristic to the region of the state space currently being explored.

Given our extensive set of strategies, we conjecture that there is little room for further improvement through other strategies. However, different approaches could be tried. For example, a machine-learning approach could combine several of the underlying statistics gathered for our ranking strategies to create a more sophisticated one. As the best-performing strategies are not always the same across all domains, a per-domain (or per-instance) selection mechanism similar to those used by portfolio planners [e.g., Cenamor et al., 2016; Sievers et al., 2019] could potentially boost performance on domains where the default ranking strategy is not ideal.

Following the success of the simple replacement strategy in GBFS, incorporating conjunction forgetting into an online-refinement search algorithm might be another interesting direction for future work. Forgetting conjunctions could significantly reduce the computational overhead of the heuristic in cases where many conjunctions are added over the duration of the search. This would be difficult to implement in our hill-climbing algorithms as they rely on converging refinement for completeness, which seems impossible to retain with conjunction forgetting. However, it could potentially yield improvements in GBFS-SCL, though it would require a reliable mechanism to detect conjunctions that are no longer useful and can be removed without sacrificing informativeness.

6 Finding Plans with Red-Black State-Space Search

There are two main lines of research in red-black planning. The tractable fragment ACI [Domshlak et al., 2015] enables its use as a heuristic, but imposes restrictions in the possible paintings. On the other hand, red-black state-space search (RBS) [Gnad et al., 2016] allows for full interpolation between fully delete-relaxed and real planning, but comes with the complexity of state-space search.

ACI requires that the causal graph of black variables is acyclic, and that all back variables are invertible. These restrictions typically only allow a small fragment of variables to be painted black, and the resulting red-black plans are very different from real plans.

RBS addresses these restrictions, and can generate red-black plans that are real plans in the limit. Gnad et al. [2016] explored RBS as a method to generated seed plans for plan repair with LPG [Gerevini et al., 2003; Fox et al., 2006], and to prove unsolvability, exploiting the fact that if the task is unsolvable in the relaxation, the original task must also be unsolvable. For the latter, they use an incremental search strategy: Starting out with a fully relaxed painting (painting all variables red); whenever a solution is found, the painting is iteratively refined by painting more and more variables black until the relaxation becomes unsolvable.

Here, we explore RBS as a method to generate plans directly. We follow a similar approach to the incremental search strategy outlined above, using incremental refinement of the painting until we obtain a red-black plan that is also a real plan. The challenge is to make RBS produce real plans early on, with few black variables. We design two enhancements to this end:

1. We create synergy between RBS and ACI, by replacing delete-relaxed planning with ACI planning in RBS. This uses ACI where possible (e.g., driving back and forth on an invertible road map), and uses RBS where not (e.g., resource consumption and

other non-invertible variables). We identify a maximally permissive condition on the black-variable dependencies under which this combination is possible.

2. We design a variant of red-black state-space search with adaptive refinement, allowing the use of different paintings in different parts of the search space. Every transition $s \xrightarrow{a} s'$ is checked for *realizability* of the red parts, i.e., whether the delete-relaxed plan here works in reality. Non-realizable transitions are pruned, and spawn *refinement options*: red-black planning tasks starting at s, with additional black variables addressing the non-realizability of $s \xrightarrow{a} s'$. The refinement options become search nodes in an overall heuristic search.

We evaluate our techniques on the IPC benchmarks. In overall performance, 1 is competitive, while 2 often suffers from the added overhead of spawning too many refinement options. Compared to Gnad et al.'s [2016] original approach of using red-black plans as seed plans for plan repair, generating plans directly through the combination of RBS with ACI is better overall, and both of our contributions are highly complementary to the plan repair approach per domain. In five domains, our best configurations outperform the state-of-the-art planners LAMA [Richter and Westphal, 2010] and Mercury [Katz and Hoffmann, 2014] by large margins.

Papers and Contributions This chapter is based on the paper "Unchaining the Power of Partial Delete Relaxation, Part II: Finding Plans with Red-Black State Space Search" [Fickert et al., 2018b]. The paper was principally developed by the author, in joint work with Daniel Gnad and Jörg Hoffmann. The plot showing the coverage depending on the number of black variables (Figure 6.1) and the corresponding discussion (Section 6.3.2) are Daniel Gnad's work.

6.1 Combining RBS with ACI

Any flaw in a red-black plan π^{RB} can in principle be fixed by painting the respective variable v black and re-running RBS. However, the red-black state space grows exponentially in $|\mathcal{V}^{B}|$, raising the question whether we can somehow avoid the computational cost incurred by painting v black.

As we show in the following, the answer is yes—if we can address flaws in v by ACI instead (like for the variable "*at*" in Example 3.4). We can use ACI to effectively handle a tractable part of the task at hand (e.g., invertible variables moving back and forth), combined with RBS to handle the remainder (e.g., variables modeling resource consumption).

6.1.1 The RBS+ACI Framework

Our combined framework, that we baptize *RBS+ACI*, distinguishes black variables of two different kinds: those handled by RBS vs. those handled by ACI. So a painting now partitions $\mathcal{V}$ into three subsets $\mathcal{V}^{\text{RBS}}, \mathcal{V}^{\text{ACI}}, \mathcal{V}^{\text{R}}$, where $\mathcal{V}^{\text{B}} = \mathcal{V}^{\text{RBS}} \cup \mathcal{V}^{\text{ACI}}$.

Assume that such a partition is given. We need a red-black plan relative to the entire set $\mathcal{V}^{\text{B}}$ of black variables, i.e., for the red-black planning task $\Pi^{\text{RB}} = \langle \mathcal{V}^{\text{RBS}} \cup \mathcal{V}^{\text{ACI}}, \mathcal{V}^{\text{R}}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$. The basic idea is to apply ACI plan repair on the outcome of running red-black state-space search on the coarser (more relaxed) task $\Pi^{\text{RB}}_{+} := \langle \mathcal{V}^{\text{RBS}}, \mathcal{V}^{\text{R}} \cup \mathcal{V}^{\text{ACI}}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$.

ACI plan repair is defined for fully delete-relaxed plans, not red-black plans, so we must adapt the repair process. We must make sure that the repair (a) is always possible given the black part $\mathcal{V}^{\text{RBS}}$ already fixed, and (b) never affects that fixed part.

Let π be the plan found by RBS for Π^{RB}_{+}. Our adapted repair process, *RBS+ACI plan repair*, computes a plan without conflicts on the entire set of black variables $\mathcal{V}^{\text{RBS}} \cup \mathcal{V}^{\text{ACI}}$, fixing unsatisfied conditions only on $\mathcal{V}^{\text{ACI}}$ without modifying the conflict-free $\mathcal{V}^{\text{RBS}}$.

To ensure (b), an obvious and natural requirement is that there is no action $a \in \mathcal{A}$ with $\mathcal{V}(\mathsf{eff}(a)) \cap \mathcal{V}^{\text{ACI}} \neq \emptyset$ and $\mathcal{V}(\mathsf{eff}(a)) \cap \mathcal{V}^{\text{RBS}} \neq \emptyset$. In other words, the repair actions will never affect any variables in $\mathcal{V}^{\text{RBS}}$.

Ensuring (a) is more tricky. In the red-black state-space search on Π^{RB}_{+}, the red completion $\mathcal{F}^{+}(s^{\text{RB}})$ of any state s^{RB} uses only actions whose preconditions are satisfied given the black variable assignment $s^{\text{RB}}|_{\mathcal{V}^{\text{RBS}}}$. So one may think (and we did think at first) that no further restrictions are needed. However, across transitions $s^{\text{RB}} \xrightarrow{a} t^{\text{RB}}$, the fixed repair context changes from $s^{\text{RB}}|_{\mathcal{V}^{\text{RBS}}}$ to $t^{\text{RB}}|_{\mathcal{V}^{\text{RBS}}}$. This causes problems because, during RBS, the values reached for $\mathcal{V}^{\text{ACI}}$ in $\mathcal{F}^{+}(s^{\text{RB}})$ are propagated to t^{RB}. But due to the different context $t^{\text{RB}}|_{\mathcal{V}^{\text{RBS}}}$, the repair process at t^{RB} is not necessarily able to reach these values.

Similar to Gnad and Hoffmann [2015], we impose that there is no $a \in \mathcal{A}$ with $\mathcal{V}(\mathsf{eff}(a)) \cap \mathcal{V}^{\text{ACI}} \neq \emptyset$ and $\mathcal{V}(\mathsf{pre}(a)) \cap \mathcal{V}^{\text{RBS}} \neq \emptyset$, i.e., the actions used in the repair process do not have preconditions on $\mathcal{V}^{\text{RBS}}$. We next show that this restriction is sufficient to ensure (a), and the repair will always work. We then show that the restriction is necessary for computational reasons.

The conjunction of our two restrictions is equivalent to the absence of a causal graph arc from $\mathcal{V}^{\text{RBS}}$ to $\mathcal{V}^{\text{ACI}}$. In this case, we say that $\mathcal{V}^{\text{ACI}}$ *does not depend on* $\mathcal{V}^{\text{RBS}}$.

Proposition 6.1 (Soundness). *Given a red-black planning task* $\Pi^{\text{RB}} = \langle \mathcal{V}^{\text{B}}, \mathcal{V}^{\text{R}}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$*, and a partition of* $\mathcal{V}^{\text{B}}$ *into* $\mathcal{V}^{\text{RBS}}$ *and* $\mathcal{V}^{\text{ACI}}$ *such that the task* $\langle \mathcal{V}^{\text{ACI}}, \mathcal{V}^{\text{R}} \cup \mathcal{V}^{\text{RBS}}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ *is in ACI, and*

$\mathcal{V}^{\mathrm{ACI}}$ does not depend on $\mathcal{V}^{\mathrm{RBS}}$. Let π be a red-black plan for $\Pi^{\mathrm{RB}}_{+} = \langle \mathcal{V}^{\mathrm{RBS}}, \mathcal{V}^{\mathrm{R}} \cup \mathcal{V}^{\mathrm{ACI}}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$. Then RBS+ACI plan repair on π succeeds, and its output π^{RB} is a red-black plan for Π^{RB}.

Proof. Any action a that may be inserted by ACI plan repair, and hence by RBS+ACI plan repair, affects a variable in $\mathcal{V}^{\mathrm{ACI}}$. Since we require that $\mathcal{V}^{\mathrm{ACI}}$ does not depend on $\mathcal{V}^{\mathrm{RBS}}$, we know that a has no preconditions or effects on $\mathcal{V}^{\mathrm{RBS}}$. Hence, the arguments given by Domshlak et al. [2015, Theorem 11] remain applicable, i.e., ACI plan repair can be run on π and generates a valid red-black plan for Π^{RB} in polynomial time. □

Example 6.1. *Consider again the task from Example 3.3, where we want to buy two products in the store at location B and return to A (without fuel consumption):*

Assume we set $\mathcal{V}^{\mathrm{RBS}} = \{money\}$ and $\mathcal{V}^{\mathrm{ACI}} = \{at\}$. Note that money depends on at: this dependency direction is allowed (but not the other way around).

RBS is run on the task $\Pi^{\mathrm{RB}}_{+} = \langle \{money\}, \{at, have{-}P_1, have{-}P_2\}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ (the same task on which we ran RBS in Example 3.5). The plan returned by RBS is again $\pi = \langle drive(A,B)$, $buy(P_1, 2)$, $buy(P_2, 1)\rangle$. Running ACI plan repair on π finds the unsatisfied goal condition $g = \{at = A\}$. This is repaired by appending the action $drive(B,A)$ to the end of π, yielding the overall plan $\langle drive(A,B)$, $buy(P_1, 2)$, $buy(P_2, 1)$, $drive(B,A)\rangle$, which is a valid plan for the original task.

Proposition 6.1 shows that our RBS+ACI framework is sound for red-black planning in Π^{RB}. This framework is also complete:

Proposition 6.2 (Completeness)**.** *Under the prerequisites of Proposition 6.1, a red-black plan for $\Pi^{\mathrm{RB}} = \langle \mathcal{V}^{\mathrm{RBS}} \cup \mathcal{V}^{\mathrm{ACI}}, \mathcal{V}^{\mathrm{R}}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ exists iff a red-black plan for $\Pi^{\mathrm{RB}}_{+} = \langle \mathcal{V}^{\mathrm{RBS}}, \mathcal{V}^{\mathrm{R}} \cup \mathcal{V}^{\mathrm{ACI}}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ exists.*

Proof. The “if” direction holds by Proposition 6.1. The “only if” direction holds because Π^{RB} is a refinement of Π^{RB}_{+}. □

So our approach works provided there is no causal graph arc from $\mathcal{V}^{\mathrm{RBS}}$ to $\mathcal{V}^{\mathrm{ACI}}$. Let us show that this restriction is necessary. Consider the decision problem *RBS-dependent ACI PlanGen*, defined as follows. Given the red-black planning task $\Pi^{\mathrm{RB}} = \langle \mathcal{V}^{\mathrm{B}}, \mathcal{V}^{\mathrm{R}}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$,

and a partition of $\mathcal{V}^{\text{B}}$ into $\mathcal{V}^{\text{RBS}}$ and $\mathcal{V}^{\text{ACI}}$ such that $\langle \mathcal{V}^{\text{ACI}}, \mathcal{V}^{\text{R}} \cup \mathcal{V}^{\text{RBS}}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ is in ACI, and all causal graph arcs between $\mathcal{V}^{\text{RBS}}$ and $\mathcal{V}^{\text{ACI}}$, if any, are directed from $\mathcal{V}^{\text{RBS}}$ to $\mathcal{V}^{\text{ACI}}$. Given a red-black plan π for $\Pi^{\text{RB}}_{+} = \langle \mathcal{V}^{\text{RBS}}, \mathcal{V}^{\text{R}} \cup \mathcal{V}^{\text{ACI}}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$. Let $\pi|_{\mathcal{V}^{\text{RBS}}}$ the subsequence of actions in π that have effects on $\mathcal{V}^{\text{RBS}}$. Decide whether $\pi|_{\mathcal{V}^{\text{RBS}}}$ is a subsequence of a red-black plan for Π^{RB}.

Theorem 6.3 (**NP**-Hardness)**.** *RBS-dependent ACI PlanGen is **NP**-hard.*

Proof. We prove the claim by reduction from SAT. Let ϕ be a CNF formula with propositions $p_1, \dots, p_n$ and clauses $c_1, \dots, c_m$. Our planning encoding first chooses values for p_i, then satisfies the clauses c_j. The construction sets $\mathcal{V}^{\text{RBS}}$ to contain a single "indicator" variable, determining whether we can right now set p_i to 0 or to 1; $\mathcal{V}^{\text{ACI}}$ represents this choice of values; and $\mathcal{V}^{\text{R}}$ represents whether a clause has been satisfied yet.

In detail, we set $\mathcal{V}^{\text{RBS}} = \{v\}$ with domain $\{0, 1\}$, initial value 0, and a single action $a[v01]$ going from 0 to 1. We set $\mathcal{V}^{\text{ACI}} = \{v_{p_1}, \dots, v_{p_n}\}$ with domain $\{\bot, 0, 1\}$, initial value $\bot$, actions going from $\bot$ to 0 with precondition $v = 0$, actions going from $\bot$ to 1 with precondition $v = 1$, and actions going from either 0 or 1 back to $\bot$ with no additional precondition on v. We set $\mathcal{V}^{\text{R}} = \{v_{c_1}, \dots, v_{c_m}\}$ with domain $\{0, 1\}$, initial value 0, goal value 1, and an action $a[v_{c_j}01]$ setting v_{c_j} from 0 to 1 with precondition $\{v = 1, v_{p_i} = x\}$ for each fact $(p_i = x) \in c_j$.

Observe first that this red-black planning task Π^{RB} does satisfy the prerequisites: all $v_{p_i} \in \mathcal{V}^{\text{ACI}}$ are invertible, and there are no dependencies across these variables; the dependencies between $\mathcal{V}^{\text{RBS}}$ and $\mathcal{V}^{\text{ACI}}$ consist of the causal graph arcs $\langle v, v_{p_i} \rangle$.

Consider now $\pi|_{\mathcal{V}^{\text{RBS}}} := \langle a[v01] \rangle$. This is a subsequence of a red-black plan π for Π^{RB}_{+}: We can move each variable v_{p_i} to $v_{p_i} = 0$ before the application of $a[v01]$, and to $v_{p_i} = 1$ after that application. Any formula ϕ can be satisfied that way.

But is $\pi|_{\mathcal{V}^{\text{RBS}}}$ a subsequence of a red-black plan for Π^{RB}? The answer is "yes" iff ϕ is satisfiable. This is because $\pi|_{\mathcal{V}^{\text{RBS}}}$ is (trivially) a subsequence of any red-black plan for Π^{RB}, and a red-black plan for Π^{RB} exists iff ϕ is satisfiable. The latter is true because, in Π^{RB}, each v_{p_i} can support the clause-satisfying actions $a[v_{c_j}01]$ with only a single truth value. First, $v_{p_i} = 1$ can only be reached after $a[v01]$, at which point $v_{p_i} = 0$ is no longer reachable. Second, we can set $v_{p_i} = 0$ before the application of $a[v01]$. But at that point, $a[v_{c_j}01]$ is not yet applicable due to its precondition $v = 1$. So we must apply $a[v01]$, and afterwards we can no longer reach $v_{p_i} = 1$. □

By Theorem 6.3, given the fixed solution path $\pi|_{\mathcal{V}^{\text{RBS}}}$ found by red-black state-space search for Π^{RB}_{+}, augmenting $\pi|_{\mathcal{V}^{\text{RBS}}}$ to a red-black plan for Π^{RB} is hard. In our framework, such

augmentation is done by red (delete-relaxed) planning in Π^{RB}_{+} alongside $\pi|_{\mathcal{V}^{\mathrm{RBS}}}$, followed by RBS+ACI plan repair. Hence, one of these steps would need to have worst-case exponential runtime (unless $\mathbf{P} = \mathbf{NP}$). In other words, efficient RBS+ACI plan repair is not possible when allowing causal graph arcs from $\mathcal{V}^{\mathrm{RBS}}$ to $\mathcal{V}^{\mathrm{ACI}}$.

In practice, i.e., in our overall planning algorithm introduced next, one can ameliorate the situation by *attempting* RBS+ACI plan repair even if $\mathcal{V}^{\mathrm{ACI}}$ does depend on $\mathcal{V}^{\mathrm{RBS}}$. If the repair succeeds, all is fine—we only need to act (by removing the problematic variable(s) from $\mathcal{V}^{\mathrm{ACI}}$) in case the repair fails.

6.1.2 Overall Planning Process: Iterated RBS+ACI

We now know how to solve any red-black task Π^{RB} with an RBS+ACI painting $\mathcal{V}^{\mathrm{RBS}}$, $\mathcal{V}^{\mathrm{ACI}}$, $\mathcal{V}^{\mathrm{R}}$ that qualifies for Proposition 6.1. But our aim here is to find real plans, for the original FDR input task Π. Thus, RBS+ACI becomes a tool within our overall planning process.

That process is a loop around RBS+ACI searches with increasingly refined paintings, similar to Gnad et al.'s [2016] iterated approach for unsolvable tasks. In a pre-process, we compute an ACI painting $\mathcal{V}^{\mathrm{B}}_0, \mathcal{V}^{\mathrm{R}}_0$ using the default painting strategy of Mercury, which orders the variables by causal graph level and iteratively paints variables red until the black causal graph is a DAG [Katz and Hoffmann, 2014]. We then initialize our painting as $\mathcal{V}^{\mathrm{RBS}} := \emptyset$, $\mathcal{V}^{\mathrm{ACI}} := \mathcal{V}^{\mathrm{B}}_0$, $\mathcal{V}^{\mathrm{R}} := \mathcal{V}^{\mathrm{R}}_0$, and start our iterative planning process. First, we run RBS+ACI using the current painting. If a red-black plan does not exist, we know that the original task Π is unsolvable and we stop. Otherwise, we now have a red-black plan π^{RB}. We check whether π^{RB} is a real plan for Π. If yes, we stop; otherwise, we refine our painting. In order to refine the painting, we simulate the execution of π^{RB} under the real planning semantics in Π, and we count the number of flaws (unsatisfied preconditions or goals) associated with each variable $v \in \mathcal{V}^{\mathrm{R}}$. We select $v \in \mathcal{V}^{\mathrm{R}}$ with a maximal number of flaws (a criterion adapted from Mercury) and set $\mathcal{V}^{\mathrm{RBS}} := \mathcal{V}^{\mathrm{RBS}} \cup \{v\}$ and $\mathcal{V}^{\mathrm{R}} := \mathcal{V}^{\mathrm{R}} \setminus \{v\}$, and proceed to the next iteration with the updated painting.

Adding v to $\mathcal{V}^{\mathrm{RBS}}$ may introduce dependencies of $\mathcal{V}^{\mathrm{ACI}}$ on $\mathcal{V}^{\mathrm{RBS}}$. Therefore, as discussed above, at some point RBS+ACI plan repair may fail. In that case, we move the culprit variable(s) from $\mathcal{V}^{\mathrm{ACI}}$ to $\mathcal{V}^{\mathrm{R}}$, re-establishing the Proposition 6.1 guarantee that repair will succeed. The red-black relaxation considered is, then, no longer a refinement of the previous one. However, convergence to $\mathcal{V}^{\mathrm{B}} = \mathcal{V}$ remains intact (in the worst case, $\mathcal{V}^{\mathrm{ACI}}$ may become empty, but then $\mathcal{V}^{\mathrm{RBS}}$ eventually converges to $\mathcal{V}$), so the completeness of the overall planning process is preserved.

This overall planning process can be seen as a form of counterexample-guided abstraction refinement—in each iteration, we either find a real plan and are done, or the current abstraction (painting) is refined based on the flaws encountered when attempting to execute it.

Whenever checking whether an intermediate red-black plan π^{RB} works under the real planning semantics in Π, a variant is to *commit* to the conflict-free prefix of π^{RB}, i.e., start the next iteration from the resulting state instead of the initial state; we will refer to this variant as *prefix execution (PE)*. This saves some search effort if the new start state is closer to the goal than the original initial state, however, it loses completeness if the task contains dead ends.

6.2 Adaptive Refinement via Realizability

The iterated refinement loop around RBS, as in our overall RBS+ACI planning framework, is wasteful in that every iteration starts from the initial state, and has to rebuild the red-black search space from scratch. Prefix execution fixes this, but in a limited way and at the cost of completeness. Ideally, like other abstraction refinement processes, we ought to refine in an adaptive manner, *only where needed*, and do so incrementally within a single, iteratively refined, relaxed search space.

There is no obvious answer on how to do this effectively in RBS for the purpose of finding real plans. The straightforward approach would be to search until a red-black plan π^{RB} is found, execute π^{RB} under the real semantics until the first flaw occurs at the red-black state s^{RB}, then generate a refined painting which is used to rebuild the search space below s^{RB} with RBS. Similar to prefix execution, this would save the search effort to reach s^{RB} again, but we want to be able to continue search into other parts of the state space, and with different paintings refined to the specific flaws found in those areas.

However, there are some issues with this approach. With many black variables—as needed to find real plans—finding π^{RB} becomes very expensive so there will be long time intervals between the local refinement steps. Furthermore, the red-black plan π^{RB} may already have a flaw close to the root state, so a lot of the search effort that was used to find the flawed plan has been wasted. To illustrate this point, say that the only action applicable at the root of an RBS subtree s^{RB} has red preconditions p and q, each of which is reached in $\mathcal{F}^+(s^{\text{RB}})$, but which are in conflict so their conjunction is not reachable under the real semantics. In that case, all search below s^{RB} is obsolete.

Given these observations, here we design an eager approach, refining the painting locally whenever a transition in Θ^{RB} will not work out under non-relaxed semantics. We first

show how this can be incorporated into RBS, then we again discuss the combination with ACI.

6.2.1 Realizability Refinement: X-RBS

Let s^{RB} be any red-black state in Θ^{RB}, and let $s^{\mathrm{RB}} \xrightarrow{a} t^{\mathrm{RB}}$ be any outgoing transition of s^{RB}. By construction, we know that $\mathsf{pre}(a)|_{\mathcal{V}^{\mathrm{R}}} \subseteq \mathcal{F}^{+}(s^{\mathrm{RB}})$, i.e., the red preconditions of a are reachable from s^{RB} in the delete relaxation. Let now π_X^{+} be a relaxed plan from s^{RB} for the goal $\mathsf{pre}(a)|_{\mathcal{V}^{\mathrm{R}}}$, extracted by some relaxed-plan extraction method X. If π_X^{+} achieves $\mathsf{pre}(a)|_{\mathcal{V}^{\mathrm{R}}}$ under the *real* semantics, we say that $s^{\mathrm{RB}} \xrightarrow{a} t^{\mathrm{RB}}$ is *realized by* π_X^{+} and is *realizable given* X.

Definition 6.4 (X-RB State Space). Let Π^{RB} be a red-black planning task, and let X be a relaxed-plan extraction method. The *X-RB state space* is the transition system Θ_X^{RB}, which is defined like Θ^{RB}, except that:

(a) transitions $s^{\mathrm{RB}} \xrightarrow{a} t^{\mathrm{RB}}$ not realizable given X are pruned;

(b) if $s^{\mathrm{RB}} \xrightarrow{a} t^{\mathrm{RB}}$ is realized by π_X^{+}, then t^{RB} is the outcome state of executing $\pi_X^{+} \circ a$ in s^{RB} under the real semantics.

The X-RB state space models a modified version of the original red-black state space, taking into account the specific relaxed plan generation method X. At each transition $s^{\mathrm{RB}} \xrightarrow{a} t^{\mathrm{RB}}$, the red preconditions are achieved by inserting a relaxed plan for $\mathsf{pre}(a)$—but if the specific relaxed plan π_X^{+} generated by X to achieve $\mathsf{pre}(a)$ does not work under real semantics, we already know that the final red-black plan will be flawed at that transition. Hence, the X-RB state space prunes such a transition. It is of course a restriction here to commit to the plan generation method X. But there is no systematic alternative: short of a full-scale planning process for $\mathsf{pre}(a)$—giving up on the relaxation altogether—if X does not find a real plan, then the best one could do is try another relaxed plan extraction method X' (which again might not find a real plan, and a real plan for $\mathsf{pre}(a)$ may in fact not exist at all).

That said, Definition 6.4 is only one half of the story. Whenever a transition $s^{\mathrm{RB}} \xrightarrow{a} t^{\mathrm{RB}}$ is pruned by (a), we spawn a *refinement option*, discussed in detail below. A refinement option is a refined red-black planning task at s^{RB}, addressing the reason for the non-realizability of $s^{\mathrm{RB}} \xrightarrow{a} t^{\mathrm{RB}}$.

Point (b) of the X-RB state space definition has the immediate effect that every reachable state s^{RB} in Θ_X^{RB} is in fact a real state. This turns the red part of the search (the relaxed plan

generation method X) into a fast macro generator to the next applicable black-variable-affecting action.

Observe that this is a natural match with our realizability check. The realizability check ensures that we are able to reach $\mathsf{pre}(a)$ at s^{RB} under the real semantics of the task by verifying the relaxed plan π_X^+, so it only makes sense to consider the state resulting from that specific plan. In contrast, the over-approximated state transition, without (b), would pretend that we can reach *the entire set* $\mathcal{F}^+(s^{\mathrm{RB}})$. Intuitively, we can check the validity of $s^{\mathrm{RB}} \xrightarrow{a} t^{\mathrm{RB}}$ only in a limited way, because a priori we do not know what the full red goal might be here at plan extraction time, so we restrict to the known preconditions of the transition at hand.

We now give the details on the *refinement options* in the X-RB state space:

Definition 6.5. Let $\Pi^{\mathrm{RB}} = \langle \mathcal{V}^{\mathrm{B}}, \mathcal{V}^{\mathrm{R}}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ be a red-black planning task; let $s^{\mathrm{RB}} \xrightarrow{a} t^{\mathrm{RB}}$ be a non-realizable transition pruned in Θ_X^{RB}, with the corresponding relaxed plan π_X^+; and let $v \in \mathcal{V}^{\mathrm{R}}$ be a red variable such that π_X^+ contains a maximal number of flaws on v. Then $\Pi_{+v}^{\mathrm{RB}}(s^{\mathrm{RB}}) := \langle \mathcal{V}^{\mathrm{B}} \cup \{v\}, \mathcal{V}^{\mathrm{R}} \setminus \{v\}, \mathcal{A}, s^{\mathrm{RB}}, \mathcal{G} \rangle$ is a *refinement option* for $s^{\mathrm{RB}} \xrightarrow{a} t^{\mathrm{RB}}$.

Whenever a transition $s^{\mathrm{RB}} \xrightarrow{a} t^{\mathrm{RB}}$ is pruned in our exploration of Θ_X^{RB}, we generate a refinement option $\Pi_{+v}^{\mathrm{RB}}(s^{\mathrm{RB}})$, which locally refines the painting based on the variable with the most conflicts in the relaxed plan for $\mathsf{pre}(a)$. The refinement option is inserted as a search node into the overall (heuristic) search. Thus, the X-RB search process decides not only *which states* to explore, but also *which refinement* is used to explore that state. We will refer to this overall search framework as *X-RBS*.

Observe that point (b) in Definition 6.4 is an under-approximation that loses completeness: By committing to π_X^+, we may exclude solutions, and the resulting X-RB search space may not contain a plan. As an optional fix, we can also spawn refinement options at nodes s^{RB} all of whose descendants have been unsuccessfully explored. In such a case, we do not have a concrete flaw to fix, so we pick a variable $v \in \mathcal{V}^{\mathrm{R}}$ to paint black arbitrarily. We refer to this variant as *refinement explored (RE)*.

6.2.2 Combination with ACI

The number of refinement options can be a major source of computational overhead in X-RBS. One way to ameliorate this is to combine X-RBS with ACI (*X-RBS+ACI*): replacing delete-relaxed planning with tractable red-black planning will result in fewer flaws, and in more realizable transitions.

The combination is simple in X-RBS as relaxed planning occurs only at individual transitions $s^{RB} \xrightarrow{a} t^{RB}$. Relaxed planning is used for two purposes: (1) to generate $\mathcal{F}^+(s^{RB})$ in order to test whether pre(a) is relaxed-reachable, and (2) to extract a specific relaxed plan using the method X, in order to check realizability and generate the successor.

Using ACI instead, (1) remains unchanged. For (2), we use ACI plan repair on top of the relaxed plan extraction method X. As before, this again uses separate sets of black variables $\mathcal{V}^{RBS}$ and $\mathcal{V}^{ACI}$, but with no constraint on their dependencies: in a realizability check—against the real semantics—a success guarantee cannot be given anyway.

6.3 Experiments

Our implementation is based on Gnad et al.'s [2016] original implementation of red-black state-space search, which modifies Fast Downward [Helmert, 2006] in a minimally intrusive way, exchanging the state and state transition data structures while preserving all search algorithms. The source code is available at `https://github.com/fickert/fast-downward-redblack-search`.

We use the satisficing STRIPS benchmarks from the International Planning Competitions up to 2014 for the evaluation. All our configurations run greedy best-first search using a dual queue for preferred operators with Gnad et al.'s [2016] extended h^{FF} heuristic.

We run each of RBS and X-RBS with vs. without ACI. We run RBS with vs. without prefix execution (PE), and X-RBS with vs. without refinement explored (RE), yielding eight different configurations. Among these, RBS with neither ACI nor prefix execution is a baseline easily derived from (though not evaluated by) Gnad et al.'s [2016] original work in red-black state-space search. We compare against LAMA [Richter and Westphal, 2010] and Mercury [Katz and Hoffmann, 2014] as representatives of the state of the art. We also run the best-performing LPG-plan-repair configuration by Gnad et al. [2016]. This paints 90% of the variables black, uses RBS to find a red-black plan π^{RB}, and then invokes LPG to repair π^{RB} into a real plan.

6.3.1 Coverage

Consider Table 6.1, and the variants of RBS (leftmost part of the table). Compared to the baseline, both our techniques (+ACI and +PE) improve performance substantially. This is clearly visible in overall coverage. Per domain, +PE yields better coverage in 14 domains, +ACI in 12, and the two together in 15. Both techniques also have their drawbacks, as +PE does not work well if the prefix often leads into dead ends (e.g., in

	RBS				X-RBS						
		+PE	+ACI	+ACI +PE		+RE	+ACI	+ACI +RE	RBS +LPG	LAMA	Mer-cury
Airport (50)	27	28	27	28	41	43	41	**44**	42	32	32
Barman (40)	0	3	0	3	0	7	0	0	24	39	**40**
Blocksworld (35)	**35**	**35**	**35**	**35**	**35**	**35**	24	33		**35**	**35**
Childsnack (20)	5	**20**	9	10	0	0	0	0	4	5	0
Depots (22)	15	17	16	18	1	9	14	15	**21**	20	**21**
DriverLog (20)	19	18	**20**	19	2	7	3	9	18	**20**	**20**
Elevators (50)	45	47	**50**	**50**	0	12	**50**	**50**	**50**	**50**	**50**
Floortile (40)	3	3	6	7	0	4	0	0	**9**	8	8
Freecell (80)	71	69	71	69	69	61	69	60	35	79	**80**
GED (20)	10	9	10	10	**20**	**20**	14	0	4	**20**	**20**
Grid (5)	4	4	**5**	4	0	2	4	**5**	4	**5**	**5**
Hiking (20)	**20**	**20**	15	17	18	15	18	**20**	19	18	**20**
Logistics (63)	62	62	**63**	**63**	0	12	**63**	**63**	35	**63**	**63**
Maintenance (20)	**11**	7	**11**	7	0	0	0	0		0	7
Mprime (35)	**35**	34	**35**	**35**	3	18	**35**	34	**35**	**35**	**35**
Mystery (19)	16	13	17	13	1	8	**19**	18	16	**19**	**19**
Nomystery (20)	**19**	**19**	**19**	17	0	4	1	4	**19**	11	14
Parcprinter (50)	49	49	49	49	39	48	36	37	35	49	**50**
Parking (40)	12	13	11	13	0	0	0	0	0	**40**	**40**
Pathways (30)	21	28	21	28	27	26	27	26	21	23	**30**
Pegsol (50)	**50**	**50**	**50**	**50**	**50**	**50**	**50**	37	16	**50**	**50**
Pipes-notank (50)	35	38	36	38	34	25	25	17	39	43	**44**
Pipes-tank (50)	31	26	28	30	26	20	34	18	24	**42**	**42**
PSR (50)	**50**	**50**	**50**	**50**	0	49	0	49	**50**	**50**	**50**
Rovers (40)	**40**	**40**	**40**	**40**	2	16	18	20		**40**	**40**
Satellite (36)	**36**	**36**	**36**	**36**	0	5	**36**	**36**		**36**	**36**
Scanalyzer (50)	42	46	42	**50**	43	42	44	44	46	**50**	**50**
Sokoban (50)	20	15	22	13	44	44	29	9	5	**48**	42
Storage (30)	18	20	18	18	16	17	**28**	**28**	25	19	19
Tetris (20)	0	3	0	2	1	0	3	2	0	13	**19**
Thoughtful (20)	6	11	6	10	15	13	9	5		**16**	13
Tidybot (20)	8	6	7	8	0	2	0	0	13	**17**	15
TPP (30)	**30**	**30**	**30**	**30**	0	10	**30**	27	**30**	**30**	**30**
Transport (70)	31	33	**70**	**70**	0	20	61	57	45	61	**70**
Trucks (30)	12	12	12	12	4	10	0	8	**20**	15	19
VisitAll (40)	3	4	**40**	**40**	3	3	**40**	**40**	4	**40**	**40**
Woodworking (50)	**50**	49	**50**	49	17	16	10	13	47	**50**	**50**
Zenotravel (20)	**20**	**20**	**20**	**20**	1	7	**20**	**20**		**20**	**20**
Sum (1385)	961	987	1047	1061	512	680	855	848	755	1211	**1238**
Normalized (%)	67.5	70.3	72.5	73.2	32.8	45.7	57.7	57.7	63.4	84.5	**86.9**

TABLE 6.1: Coverage results of different configurations of red-black state-space search. RBS+LPG is RBS followed by LPG plan repair (empty entries could not be run, see text). Domains where all tested planners have full coverage are omitted.

Sokoban). Furthermore, +ACI can sometimes introduce more conflicts into the partially relaxed plan. This happens for example in Childsnack, where otherwise the RBS+PE configuration only needs to paint the sandwich objects and tray locations black (22–25% of the total variables) to make the red-black plan a real plan, solving all instances in less than 5 seconds.

For the X-RBS method (in the middle part of Table 6.1), results are much worse, in many domains individually and hence overall. A key reason is the overhead from too many refinement options: On average, 74% of the generated transitions are realizable, in some domains much fewer (15% in Parking, 18% in Tetris). As expected, the combination with ACI ameliorates this significantly, boosting overall coverage by 343 without +RE, and by 168 with +RE. However, X-RBS+ACI is still not competitive with the other methods overall, and it remains a question for future work how it can be brought to that level of performance. While the +RE option helps in domains where X-RBS fails frequently, it also increases the overhead of too many refinement options.

Consider now the RBS+LPG baseline, using LPG plan repair on seed plans obtained from RBS. The empty entries in Table 6.1 are domains where that architecture did not run properly, for implementation reasons (these domains are also omitted in Gnad et al.'s [2016] original RBS paper). Filling in the gaps optimistically—assuming that RBS+LPG can solve *all* instances in those missing domains—overall coverage becomes 934. This still lags behind our RBS methods, even the baseline. On a per-domain level though, the methods are highly complementary: Of the 32 domains, RBS beats RBS+LPG in 12 and is inferior in 12; RBS+ACI+RE beats RBS+LPG in 16 and is inferior in 11.

For our X-RBS configurations, the comparison to RBS+LPG is, naturally, less favorable. Complementarity at per-domain level persists though: X-RBS+ACI beats RBS+LPG in 13 domains, and is inferior in 14.

Consider finally LAMA and Mercury. All our configurations are far from their performance overall. Our best configuration, RBS+ACI+PE, beats LAMA in 5 domains and is inferior in 20; for Mercury, these numbers are 2 vs. 22.

That said, there are five domains in which at least one of our configurations works exceptionally well. In Airport, our best method gains +12 coverage over the best of LAMA and Mercury; in Childsnack, +15; in Maintenance, +4; in Nomystery, +5; in Storage, +9. Thus, our new methods can potentially contribute in portfolios or per-domain autoconfiguration.

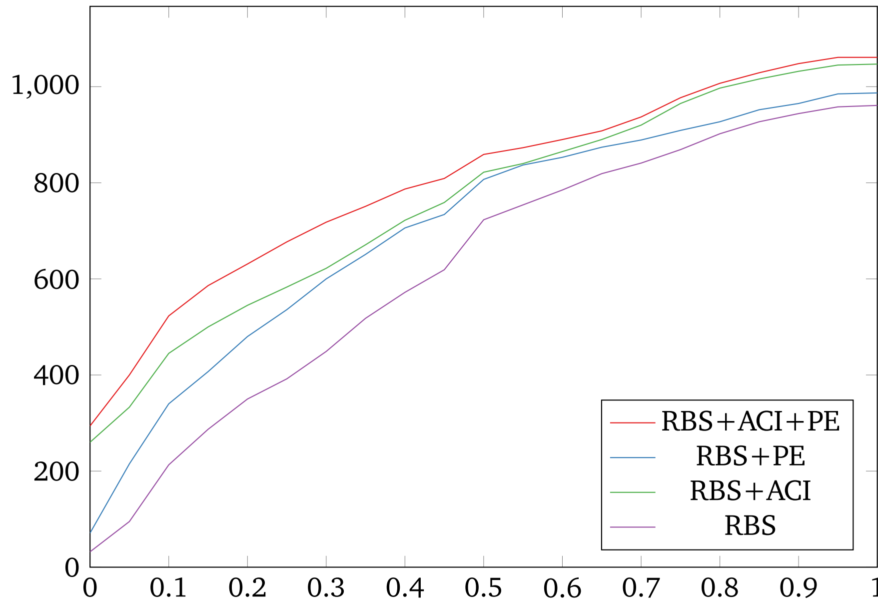


FIGURE 6.1: Coverage as a function of the fraction of RBS variables, $|\mathcal{V}^{\text{RBS}}|/|\mathcal{V}|$, in the first iteration of RBS that finds a real plan.

6.3.2 Number of Black Variables until Finding a Solution in RBS

The major motivation behind our +ACI and +PE extensions to RBS is to reduce the size of $\mathcal{V}^{\text{RBS}}$ required to find a real plan. Figure 6.1 measures this impact directly.

Both extensions clearly help as intended. Without +ACI, few instances can be solved without search ($|\mathcal{V}^{\text{RBS}}| = 0$) as, there, the delete-relaxed plan for the initial state has to be a real plan. The advantage of our extensions remains strong when allowing larger $\mathcal{V}^{\text{RBS}}$, until about $|\mathcal{V}^{\text{RBS}}|/|\mathcal{V}| = 50\%$, where the gap narrows. After that, the difference is mainly due to benchmarks that ACI solves in the initial state (like Transport), but that are beyond reach of RBS search alone.

6.4 Conclusion

We have shown that red-black state-space search can be synergetically combined with ACI tractable red-black planning, obtaining an overall red-black planning approach with convergence to real planning and low overhead for tractable variables. The overall RBS+ACI planning framework with iterative refinement significantly improves performance over the RBS baseline, but is still outperformed by state-of-the-art planners like LAMA and Mercury on most domains.

With X-RBS, we also introduced a more flexible RBS approach based on adaptive local refinement, though its performance is not yet competitive. Alternate choices for the transitions in the X-RBS might be an interesting topic for future work. For example, instead of always refining the painting further, it might be beneficial to relax the painting again to reduce search effort; or when refining the painting, one could attempt to re-use previously generated paintings to reduce the overhead.

Overall, our work contributes another piece in the puzzle how to tap into the power of partial delete relaxation without incurring a prohibitive overhead, and how the partial relaxation can be adaptively refined as needed. This fits into the larger puzzle of how to use informative but costly approximations. We believe that such research is valuable to complement the more prominent focus on fast-but-inaccurate approximations, and we hope that our ideas and insights may be useful for approaches other than red-black planning as well.

Part II

Adaptive Heuristic Search Techniques

7 Choosing the Initial State for Online Replanning

Planning and execution are typically viewed separately—a plan is computed by the planner, which is then executed by an agent. However, it may happen that the execution does not succeed, and the previously computed plan must be adapted on the fly. This can happen for various reasons. For example, the plan might have been computed using an abstract model of the world, and inaccuracy in the model causes the plan to fail in the real world [Cashmore et al., 2019]. Perhaps the environment or the agent's goals have changed in the meantime, or the agent has made new observations which prompt a reaction [e.g., Stentz and Hebert, 1995; Knight et al., 2001; McGann et al., 2007]. Even if the original plan could still work, it might be possible to come up with a more efficient plan to switch to [Likhachev et al., 2005; Ferguson et al., 2008].

If the ensuing replanning process uses forward search, it faces a critical decision: At which state along the execution should the agent deviate, i.e., which state should be the initial state for the replanning process? If the current state of the agent is selected, then the agent must sit idle until the replanning process finishes. Choosing a state further along the original plan allows the replanning process to proceed in parallel to the execution, but proceeding too far might also be detrimental and result in less efficient plans. For example, if an updated goal requires the agent to go back to its original position, any progress along the original plan must eventually be undone.

One common approach is to select a (hard-coded) fixed time point in advance such that the replanning process can finish in time before the agent reaches the corresponding state, and the agent can then continue along the new plan [e.g., Muscettola et al., 2002; McGann et al., 2007; Ruml et al., 2011]. However, this approach requires hand tuning, and a constant can be insufficient if the planning time fluctuates, leading to inefficiency. In this chapter, we introduce the *Multiple Initial State Technique (MIST)*; a planning algorithm that takes a principled approach to make this choice automatically based on estimations of its own planning time.

As the name suggests, MIST considers multiple initial states simultaneously in its search, representing the possible choices at which the agent can deviate from its original plan to a new one. MIST allows the execution to proceed during the planning process, and uses estimates of its own planning time in order to judge for which search nodes it will be able to find a solution before the corresponding initial state is passed. The search nodes are ranked by an estimate of the goal achievement time, taking into account both the planning time and the execution time of the overall plan. This allows MIST to adaptively select the currently most promising search node (and corresponding initial state) at each expansion, and allows it to remain flexible with regard to the deviation state until the new plan is finalized.

The basic version of MIST discards search nodes corresponding to initial states that are already passed by the execution, pessimistically assuming that these states are no longer reachable (and hence cannot be used as an initial state of the new plan). We additionally present a variant of MIST that can use a concept of recoverability, which allows it to assume that passed states can still be reached at some additional cost, avoiding this loss of progress.

Under suitable conditions on both the heuristic function and the planning time estimate, we prove that the expansion order of (both variants of) MIST is correct in the sense that the first returned plan cannot be improved by further search. In order to evaluate MIST on a set of different domains, we implemented MIST in Fast Downward [Helmert, 2006], and extended benchmarks from the International Planning Competitions to the online replanning setting. In our evaluation, we found that MIST reduces the goal achievement time compared to various baselines, and yields significantly more robust behavior across different domains and planning-to-execution time ratios.

Papers and Contributions This chapter is based on the paper “Choosing the Initial State for Online Replanning” [Fickert et al., 2021b]. The paper was principally developed by the author and Ivan Gavran, in joint work with Ivan Fedotov, Jörg Hoffmann, Rupak Majumdar, and Wheeler Ruml. The algorithms have been co-developed by the author and Ivan Gavran; the theoretical analysis is due to Ivan Gavran (Section 7.4 and Lemma 7.4); the implementation and experimental evaluation are the author’s work. Ivan Fedotov made minor contributions to the translator component and the reference-state-selection part of the implementation.

7.1 Previous Work

Many practical applications naturally require replanning. One typical example are autonomous vehicles used in space missions, where plan execution systems are designed to be reactive. Whenever the plan execution fails (e.g., by detecting a previously unseen obstacle), the system reacts and computes a new plan, usually fixing the initial state of the new plan by imposing a predefined constant as the worst-case replanning bound [Muscettola et al., 2002; McGann et al., 2007]. Industrial printing systems are another application for replanning as printing jobs may be scheduled asynchronously. Ruml et al. [2011] again use a hand-tuned constant as maximum planning time, and use it to decide when the system can switch to the new plan. If the planning process takes longer than expected, it is interrupted and the replanning process is restarted at a later time. In the context of navigating robots through partially-known environments, Likhachev et al. [2003, 2005] also use a constant as prediction for the planning time, and use an anytime search to dynamically improve robots' trajectories during their execution of previously computed (suboptimal) plans. Our work aims to make such hand-tuned constants obsolete by integrating the selection of the deviation state into the replanning process itself.

If the planning time is negligible compared to the execution time, a straightforward approach (and one that we consider as a baseline in our experiments) is to halt the execution and switch to the new plan as soon as possible [Knight et al., 2001; Ma et al., 2017; 2019]. Conversely, if the planning process were comparatively slow, then in some applications finishing the execution can be the best strategy. Our approach is designed to be applicable to either setting by making explicit estimations on its own planning time.

Continual online planning (COP) is a particular type of online planning where new goals appear periodically [Benton et al., 2007; Lemons et al., 2010; Burns et al., 2012]. COP tasks are modeled as Markov Decision Processes, where goals may arrive at each time step according to some probability distribution. If this distribution is known, it can be taken into account by the planner to "prepare" for their arrival (we do not make such assumptions here).

Cashmore et al. [2019] consider similar online replanning scenarios to ours for temporal planning. Their approach involves a black-box 'bail-out action generator,' which has access to a more detailed model of the agent's local state to yield possible ways for how the agent can interrupt the action that is currently being executed. The overall replanning process can take different initial states into account in the form of timed initial literals (a feature of PDDL2.2 [Haslum et al., 2019]), which represent the changes of the world along the part of the execution that the agent has already committed to. This replanning mechanism is based on situated temporal planning with deadlines [Cashmore et al.,

2018; Shperberg et al., 2019] which attempts to maximize the probability of finding a feasible plan that meets a set of temporal constraints. Our work instead aims to minimize the overall planning and execution time, and is placed in a more generic state-space search setting.

In addition to the aforementioned works on situated temporal planning [Cashmore et al., 2018; Shperberg et al., 2019], there are heuristic search algorithms that take the passing of time during planning into account. Deadline-aware search [Dionne et al., 2011] attempts to estimate the planning time for each currently open search node in order to prune those that will lead to failure in meeting a given time bound. Bugsy [Burns et al., 2013] also reasons about its own planning time in order to maximize a given utility function (a linear combination of planning time and solution cost). These approaches do not consider the additional complexity of plan execution during the planning process, and can thus use a static initial state.

7.2 Continual Online Planning

We formalize our online replanning setting as an extension of FDR tasks where a new goal appears during the execution of a plan. This also captures the case where an agent periodically receives additional jobs (by chaining such tasks). However, MIST is applicable to other types of replanning as well, as long as it is possible to proceed with the execution during the replanning process.

A *continual online planning task* extends an FDR task by modeling an ongoing execution along a given plan $\pi_{s_0,\mathcal{G}_{old}}$, when a new goal $\mathcal{G}_{new}$ appears, and action costs are interpreted as their durations:

Definition 7.1 (COP Task)**.** A *continual online planning (COP) task* is defined by the tuple $\langle \mathcal{V}, \mathcal{A}, c, s_0, \mathcal{G}_{old}, \mathcal{G}_{new}, \pi_{s_0,\mathcal{G}_{old}} \rangle$, where

- $\mathcal{V}$, $\mathcal{A}$, and c are the FDR variables, actions, and the cost function (here interpreted as execution durations of the actions),
- s_0 is the current state of the agent (when the new goal $\mathcal{G}_{new}$ appears),
- $\mathcal{G}_{old}$ is the *old goal* (partial assignment of $\mathcal{V}$),
- $\mathcal{G}_{new}$ is the *new goal* (partial assignment of $\mathcal{V}$), and
- $\pi_{s_0,\mathcal{G}_{old}} = \langle a_1, a_2, \ldots, a_n \rangle$ is a path from s_0 to a state $s_{\mathcal{G}_{old}} \supseteq \mathcal{G}_{old}$ (the *current plan* of the agent).

The plan $\pi_{s_0,\mathcal{G}_{old}}$ is a suffix of some original plan that the agent is now executing, and s_0 is the current state of the agent along it—all actions that the agent may have executed before are already in the past and hence are no longer of interest. We assume the actions to be non-interruptible: if $\mathcal{G}_{new}$ appeared during the execution of an action a, then s_0 is the state resulting from the execution of a.

For simplicity, we assume that there is no direct conflict between $\mathcal{G}_{old}$ and $\mathcal{G}_{new}$, i.e., if a variable is defined in both then its assigned value must be identical. A solution for a COP task is a plan π that leads from the state s_0 to a goal state $s_G \supseteq (\mathcal{G}_{old} \cup \mathcal{G}_{new})$. Such a plan consists of two parts: a (possibly empty) prefix $\pi_1 = \langle a_1, a_2, \ldots, a_j \rangle$ of $\pi_{s_0,\mathcal{G}_{old}}$ followed by a (possibly empty) newly planned extension $\pi_2 = \langle b_1, b_2, \ldots, b_m \rangle$. If the π_2 is not empty, then the state in which its first action b_1 is applied is called the *deviation state*. A solution is said to be optimal if it minimizes the total planning and execution time, i.e., the overall time from the arrival of the new goal $\mathcal{G}_{new}$ until the execution of π has terminated and taken the agent to a state $s_G \supseteq (\mathcal{G}_{old} \cup \mathcal{G}_{new})$.

In some scenarios it can be useful to deviate from the original plan $\pi_{s_0,\mathcal{G}_{old}}$ early. For example, consider a domain where an agent must deliver packages, and new jobs (additional packages) may appear at any time. Assume that the agent is currently delivering a package to some location l. If a new goal appears to deliver an additional package from a location close to the agent's current position to a location close to l, ideally the agent should deviate from the original plan as quickly as possible to avoid redundant movement. On the other hand, if the new package should be picked up at l and be delivered to a location where the agent came from, then first completing the execution of $\pi_{s_0,\mathcal{G}_{old}}$ would not affect the overall plan quality. This gives the agent more time for the replanning process in parallel to the execution of $\pi_{s_0,\mathcal{G}_{old}}$, potentially leading to a cheaper plan for the second delivery.

Generally, it is not known upfront how much of the original plan $\pi_{s_0,\mathcal{G}_{old}}$ should be executed before deviating to the new plan—the approach we present here aims to make this trade-off automatically, and in full generality.

7.3 The Multiple Initial State Technique

The pseudocode of the Multiple Initial State Technique (MIST) is shown in Algorithm 9. MIST is based on A* [Hart et al., 1968]; the important differences in the pseudocode are highlighted in red.

In contrast to A*'s fixed initial state, MIST considers a set of *reference states* R. These reference states are potential deviation states along the original plan $\pi_{s_0,\mathcal{G}_{old}}$ where the

Algorithm 9: MIST

```
Input: s_0, G_old, h, G_new, π_{s_0,G_old}, R
1  γ := 0
2  Closed := ∅
3  Open := {(r, r) | r ∈ R}
4  while Open ≠ ∅ do
5      (s, ref_s) := argmin_{(t,ref_t)∈Open} f(t, ref_t, γ)
6      if (s, ref_s) is not consistent with the state of the execution then
7          continue
8      if (G_old ∪ G_new) ⊆ s then
9          return path to s
10     Closed := Closed ∪ {(s, ref_s)}
11     γ := γ + 1
12     for each successor t of s do
13         ref_t := ref_s
14         if (t, ref_t) ∉ (Open ∪ Closed) or g_{ref_t}(t) < g^old_{ref_t}(t) then
15             Open := Open ∪ {(t, ref_t)}
16 return unsolvable
```

agent can switch from the old plan to the new one. The specific set R is sampled from $\pi_{s_0,\mathcal{G}_{old}}$ and given to the algorithm as a parameter.

In MIST, the open list contains *pairs* of states and corresponding reference states (representing the deviation state for this search node), initialized to the set of all reference states (paired with themselves, see line 3). When a search node is expanded, the children inherit the reference state of their parent (line 13).

The core of the algorithm is similar to A*: MIST maintains an open list based on an ordering function f, and expands the nodes in best-first order until a goal state is found (in this case, a state that is compliant with the combined goal $\mathcal{G}_{old} \cup \mathcal{G}_{new}$, see line 8). If a state s is already closed, it is only re-inserted into the open list if a cheaper path to s was found (line 14).

During the planning process in MIST, the execution of $\pi_{s_0,\mathcal{G}_{old}}$ is ongoing. Hence, some search nodes become invalid as their path is inconsistent with the state of the execution; such nodes are pruned lazily in MIST (line 6).

The most important difference to A* is that MIST aims to optimize the overall planning and execution time, not just the plan quality. This is reflected by the design of the open list ordering function f, which depends not only on the current state s, but also its reference node ref_s, and the number of expansions so far γ:

$$f(s, ref_s, \gamma) = C(\pi_{s_0,ref_s}) + g_{ref_s}(s) + h(s) + os(s, ref_s, \gamma).$$

The first component of this sum, $C(\pi_{s_0,ref_s})$, represents the time that it takes the agent to reach the reference state ref_s along the ongoing execution of $\pi_{s_0,\mathcal{G}_{old}}$. The next part, $g_{ref_s}(s)+h(s)$, corresponds to the f function in A*, combining the cost—in our interpretation, the execution time—to reach s from an initial state (here, the reference state ref_s) with an estimate of the cost-to-go h. Finally, the last part is the overshot function os, which models the case where the planning process takes too long, causing the state s to become irrelevant once the execution passes the corresponding reference state ref_s:

$$os(s, ref_s, \gamma) = \begin{cases} 0 & \text{if } (\gamma + \eta(s)) \cdot t_{exp} \leq C(\pi_{s_0,ref_s}) \\ \infty & \text{otherwise} \end{cases}$$

We model os as a binary function here, evaluating either to 0 if the planning process is estimated to finish in time, or ∞ otherwise. The estimate of the planning time is given by $(\gamma + \eta(s)) \cdot t_{exp}$, where $\eta(s)$ is an estimate of the remaining number of expansions to reach a goal, and t_{exp} is the time per expansion—a (constant) factor translating expansions to time such that it can be compared to the execution time until ref_s is reached along $\pi_{s_0,\mathcal{G}_{old}}$. The estimate for the remaining planning time can be obtained by adapting prior work [Thayer and Ruml, 2009; Burns et al., 2013], we explain the details of our implementation in the evaluation (Section 7.6).

Observe that the f function depends on the number of expansions γ, in other words, the relative ordering of search nodes in the open list may change at each step. This would effectively require a full reevaluation of the open list after each expansion, which is clearly not feasible in practice. Instead, we maintain a separate queue for each reference state (containing only the corresponding search nodes), and sort them merely by $g + h$. Whenever a node is to be selected for expansion, we evaluate the full f function only for the nodes at the front of each open list, and select the best one among them. Observe that the term $C(\pi_{s_0,ref_s})$ is the same among all search nodes in each separate open list and thus does not affect the ordering. However, this is not the case for the overshot function: Our approximation pretends that all states in each such open list share the same value for the planning time estimation η, and thus disregards changes in the relative ordering caused by different values of the overshot component. In preliminary experiments, we also tried a reevaluation strategy in exponentially increasing intervals similar to that of Bugsy [Burns et al., 2013], but found no significant difference in solution quality compared to our approximation method.

In practice, we can make another small optimization: If the state s of a search node (s, ref_s) is a reference state itself that is further along $\pi_{s_0,\mathcal{G}_{old}}$ than ref_s, it can be safely pruned if the path to s found by the search is not cheaper than that along $\pi_{s_0,\mathcal{G}_{old}}$, i.e., if $C(\pi_{s_0,ref_s}) + g_{ref_s}(s) \geq C(\pi_{s_0,s})$.

7.4 Theoretical Analysis

A* is guaranteed to find an optimal solution, provided that the heuristic function is admissible (and nodes can be reopened). A similar guarantee can not be given for MIST: The essential difference between the two settings (and thus necessarily between the two algorithms) is that in an offline setting, the exploration of the state space during the planning phase comes at no cost. On the other hand, in an online setting, exploring a part of the search space that is not going to be used in the solution can decrease the quality of the final plan, since that time was not used effectively.

Consider a situation where the only optimal plan deviates at the reference state r, and expanding all the nodes on that path takes exactly the time that the agent needs to reach r—so expanding any other node will make this solution unreachable for MIST. Hence, unless the heuristic functions h and η were perfect, there is no guarantee that MIST will find an optimal solution.

With optimality out of reach, we can prove a simpler property: the stopping criterion of MIST is correct. MIST stops when the first goal state (satisfying the combined goal $\mathcal{G}_{old} \cup \mathcal{G}_{new}$) is found, returning the path to that state as the solution. We show that continuing the search afterwards can not result in a better plan, assuming the heuristic functions h and η are admissible.

Similar to h^*, we use η^* to denote the perfect version of η, i.e., the true remaining number of expansions until the end of the planning process; and use f^* to denote the f function computed with h^* and η^*. We are using the notation γ_s to indicate the point in time (i.e., the value of γ) when the state s was expanded by the search.

Theorem 7.2. *Let h be admissible with respect to planned execution time and η admissible with respect to the number of expansions. Let $\sigma_1 = s_0, s_1, \ldots, s_i, p_1, p_2, \ldots, p_m$ be the sequence of states corresponding to the first solution π_1 found by MIST (with the deviation state s_i). Assume the algorithm continued the search and found another solution with the state sequence $\sigma_2 = s_0, s_1, \ldots, s_j, q_1, q_2, \ldots, q_n$ (with the deviation state s_j). Then $f^*(p_m, s_i, \gamma_{p_m}) \leq f^*(q_n, s_j, \gamma_{q_n})$.*

Proof. Our proof follows that of a similar property of Bugsy [Burns et al., 2013].

$$
\begin{aligned}
f^*(p_m, s_i, \gamma_{p_m}) &= C(\pi_{s_0,s_i}) + g_{s_i}(p_m) + os^*(p_m, s_i, \gamma_{p_m}) \\
&\le f(p_m, s_i, \gamma_{p_m}) && (7.1) \\
&\le f(q_l, s_j, \gamma_{p_m}) && (7.2) \\
&= C(\pi_{s_0,s_j}) + g_{s_j}(q_l) + h(q_l) + os(q_l, s_j, \gamma_{p_m}) \\
&\le C(\pi_{s_0,s_j}) + g_{s_j}(q_l) + h^*(q_l) + os^*(q_l, s_j, \gamma_{p_m}) && (7.3) \\
&\le C(\pi_{s_0,s_j}) + g_{s_j}(q_l) + h^*(q_l) + os^*(q_l, s_j, \gamma_{q_l}) && (7.4) \\
&\le C(\pi_{s_0,s_j}) + g_{s_j}(q_n) + os^*(q_n, s_j, \gamma_{q_n}) && (7.5) \\
&= f^*(q_n, s_j, \gamma_{q_n})
\end{aligned}
$$

The true cost of the solution π_1 is $f^*(p_m, s_i, \gamma_{p_m}) = C(\pi_{s_0,s_i}) + g_{s_i}(p_m) + os^*(p_m, s_i, \gamma_{p_m})$. Following the search structure of MIST, at some point we chose to expand p_m. Since p_m is the last state on the path, h^* must be zero (and hence also the admissible h). Furthermore, since os is admissible due to the admissibility of η, f must be admissible with respect to f^* (inequality 7.1).[1] Inequality 7.2 comes from our choice of the state p_m over some state q_l from σ_2.

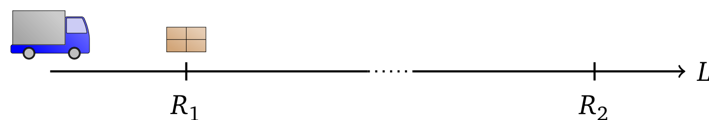
Inequality 7.3 again follows from the admissibility of h and os. For the state q_l and its reference state s_j, the value $os^*(q_l, s_j, \gamma)$ increases monotonically with the time γ. Since $\gamma_{p_m} < \gamma_{q_n}$, we get inequality 7.4.

Finally, consider inequality 7.5. First, note that $\gamma_{q_n} + \eta^*(q_n) = \gamma_{q_l} + \eta^*(q_l)$, as both sides of the equation are equal to the overall planning time. Second, since $h^*(q_l)$ is the minimal cost to reach a goal from q_l, we have $g_{s_j}(q_l) + h^*(q_l) \le g_{s_j}(q_n)$. The inequality comes from the fact that a different goal state (other than q_n) may be cheaper to reach from q_l. □

7.5 MIST for Recoverable Tasks

In some situations, the replanning approach taken by MIST can incur a large cost compared to an optimal solution.

Example 7.1. *Consider the following example of a logistics task:*



[1] Here, the two terms are in fact equal since os and os^* are both zero. We still denote this step as an inequality such that the proof also applies to the alternative os function introduced in Section 7.5.

The truck is currently driving to the location L. During the execution of the current plan, a new goal appears: a delivery request of a new package, also to the location L. Along the execution, MIST considers the two reference states R_1 and R_2 as possible deviation states. The state R_1 is closest to the package, however, it might happen that the replanning process does not finish before the truck passes R_1. In that case, MIST requires the truck to continue along the original execution to R_2, and only there the truck may turn back to pick up the package. Depending on the distance between R_1 and R_2, this can incur an arbitrarily large overhead compared to turning around at some intermediate state between R_1 and R_2.

Similar scenarios may happen even if the reference states along the original plan are sampled in smaller intervals if MIST repeatedly misses its predicted planning time.

A simple solution addressing such issues would be to halt the execution at the next reference state if the planning process is expected to finish computing a plan deviating from that state soon. However, that would require coming up with a strategy for when to commit the planning process to a given reference state, and would not be a robust solution if the planning time estimate is unreliable. Instead, we suggest a more principled approach in the form of MIST_{rec}, a variant of MIST that is adapted for COP tasks that satisfy the following *recoverability* property.

Definition 7.3 (Recoverability)**.** For a COP task $\langle \mathcal{V}, \mathcal{A}, c, s_0, \mathcal{G}_{old}, \mathcal{G}_{new}, \pi_{s_0,\mathcal{G}_{old}} \rangle$, let $s_0, \ldots, s_n$ be the state sequence induced by $\pi_{s_0,\mathcal{G}_{old}}$. We denote the action subsequence taking the agent from s_i to s_j, with $i < j$, by $\vec{\alpha}_{i,j}$. We call the task *recoverable* if, for every such $\vec{\alpha}_{i,j}$, there exists an action sequence $\overleftarrow{\alpha}_{i,j}$ such that every fact in s_i that appears as a precondition or goal also holds in the state resulting from applying $\overleftarrow{\alpha}_{i,j}$ in s_j.

This recoverability property gives the planning process some room for error in the planning time estimation: If the execution passes a reference state r, then search nodes under r do not need to be discarded immediately. Instead, the planner may finish computing a plan that uses r as a deviation state because there exists a recovery sequence bringing the agent from the current state of the execution back to r.

Many applications relating to COP tasks are naturally recoverable, for example, logistics domains where agents can freely move back and forth. Furthermore, recovery sequences are often known upfront or can be generated quickly. Recoverability relates to the well-studied notions of invertibility and undoability [Hoffmann, 2005; Daum et al., 2016]. In our experiments, we focus on domains where actions have a direct inverse of the same cost; in that case, the recovery sequence is just the sequence of the inverse actions.

In order to adapt MIST to recoverable tasks, we make two changes. First, search nodes corresponding to reference states that were already passed by the execution are no longer

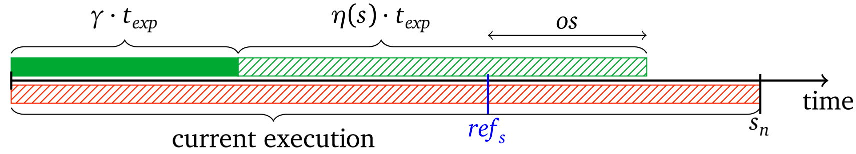
discarded. Second, we change *os* to no longer be a binary function. Instead, *os* should estimate the additional cost incurred by inserting the recovery sequence to move the agent back to a passed reference state.

The adapted overshot function for MIST_{rec} is defined as

$$os(s, ref_s, \gamma) = C(\vec{\alpha}) + C(\bar{\alpha}) + \max((\gamma + \eta(s)) \cdot t_{exp} - C(\pi_{s_0,\mathcal{G}_{old}}), 0),$$

where $\vec{\alpha}$ is the expected execution of $\pi_{s_0,\mathcal{G}_{old}}$ past the reference state ref_s, and $\bar{\alpha}$ is its recovery sequence. The overshot function now returns the added cost when planning is estimated to finish after reaching the reference state ref_s: The added execution time is accounted for by $C(\vec{\alpha}) + C(\bar{\alpha})$, and the third term of the sum describes the potential waiting time in the final state of the execution of $\pi_{s_0,\mathcal{G}_{old}}$ in case the planning process takes longer than the full execution of $\pi_{s_0,\mathcal{G}_{old}}$. As before, *os* evaluates to zero if the planning process is estimated to finish before the execution reaches ref_s (in that case $\vec{\alpha}$ and $\bar{\alpha}$ are empty, and the third term of the sum evaluates to zero).

Consider the following illustration of the overshot function:



The illustration shows a timeline of the replanning process. The red bar below the axis shows the progress of the execution of $\pi_{s_0,\mathcal{G}_{old}}$, leading to the state $s_n \supseteq \mathcal{G}_{old}$. The green bar shows the progress of the planning process; the solid part labeled by $\gamma \cdot t_{\text{exp}}$ shows the time that has been spent so far, the dashed part labeled by $\eta(s) \cdot t_{\text{exp}}$ shows the estimated time until planning finishes. In this example, planning is estimated to finish after the execution has already passed the desired deviation state ref_s. Thus, the overshot function yields the additional execution time along $\pi_{s_0,\mathcal{G}_{old}}$, plus the time it takes to bring the agent back to ref_s.

Lemma 7.4. *Assume that η is admissible, and that for a path $\vec{\alpha}$ that is a prefix of a path $\vec{\alpha}'$ it holds that $C(\vec{\alpha}) + C(\bar{\alpha}) \leq C(\vec{\alpha}') + C(\bar{\alpha}')$ (well-behaved recovery paths). Then os is admissible, i.e., $os(s, ref_s, \gamma) \leq os^*(s, ref_s, \gamma)$.*

Proof. Let $\vec{\alpha}$ be the subsequence of actions on $\pi_{s_0,\mathcal{G}_{old}}$ taking the agent from the reference state ref_s to the state in which it would be at time $\gamma + \eta(s) \cdot t_{exp}$, and let $\vec{\alpha}^*$ be the subsequence of actions to the state at time $\gamma + \eta^*(s) \cdot t_{exp}$. Since $\eta \leq \eta^*$, $\vec{\alpha}$ must be a subsequence of $\vec{\alpha}^*$. With the assumption of well-behaved recovery paths, we have $C(\vec{\alpha}) + C(\bar{\alpha}) \leq C(\vec{\alpha}^*) + C(\bar{\alpha}^*)$, and thus $os \leq os^*$. □

Since *os* is again admissible, the proof of Theorem 7.2 also applies to MIST_{rec}.

Consider again Example 7.1. While MIST must continue along the plan until reaching R_2, MIST_{rec} will be able to finish computing a new plan from the (already passed) reference state R_1, and can turn around without having to go to R_2 first.

7.6 Experiments

As explained in Section 7.3, in our implementation of MIST, we use a standard A* open list for each reference state, using the MIST extensions to the f-function only to select the open list to be used for the next expansion to avoid having to re-sort the open list. For MIST_{rec}, our implementation assumes that each action has an inverse action with the same cost. The source code and benchmarks are available at `https://github.com/fickert/fast-downward-mist`.

Like Bugsy, we estimate the remaining number of expansions as $\eta = \mathit{delay} \cdot d$ [Burns et al., 2013; Dionne et al., 2011], where *delay* is the (moving) average number of expansions between inserting a node into the open list and expanding it, and d is an estimation of the remaining distance to the goal (i.e., ignoring action costs). The expansion delay is important to counteract *search vacillation* [Dionne et al., 2011], referring to the search fluctuating between different solution paths and, in our case, potentially of different reference states. For d, we don't use the distance estimate of the current state, but instead the minimal distance of any evaluated state that corresponds to the considered reference state to make the planning time estimations more stable.

Our key performance metric is the *goal achievement time* (GAT), i.e., overall time for online planning and execution, measured from the moment when the new goals appear. We measure this time as a number of expansions to make the experiments more robust. Action costs are translated into execution time using an instance-specific factor from cost to expansions (we give more details in the next subsection).

In all experiments, the search is guided by h^{FF}. We use a moving average over the last 100 expansions for the expansion delay.

7.6.1 Benchmarks

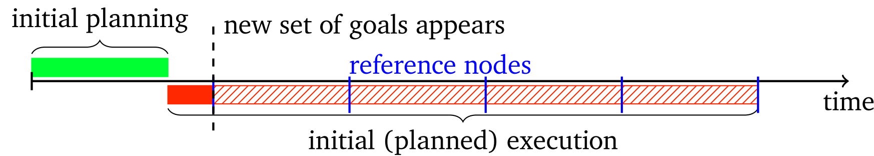
We adapted the IPC domains Elevators, Logistics, Rovers, Transport, and VisitAll to our setting, as representatives of applications where (a) goals are of an additive nature and

there are no conflicts between them, and (b) all action sequences $\vec{\alpha}$ have a recovery sequence $\tilde{\alpha}$ with the same cost. Criterion (b) is required for our implementation of MIST$_{rec}$. We furthermore experiment with Tidybot, which we adapted to satisfy (b). In Tidybot, there are cases where objects are placed behind each other, and the robot cannot reach behind the object in the front. We added an un-finish action to ensure recoverability. However, previously finished objects must be picked up again in these cases, necessitating the planner to falsify and re-achieve previously achieved goals. We assume actions to be non-interruptible.

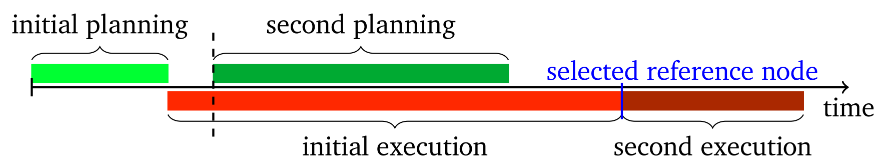
In some of our benchmark domains, recovery sequences with lower cost can exist. For example, there could be shortcuts to inverting the agent's movements, and in Rovers photos would not need to be "un-taken". In such cases, our implementation of MIST$_{rec}$ is pessimistic and more practical implementations may achieve lower plan costs.

The instances were adapted by splitting the set of goals in two: the first half is available in the beginning, and the other one becomes available later. The second set of goals is scheduled to appear during the execution of the first computed plan to obtain interesting instances.

A run of MIST on one such instance will look as follows:



The initially computed plan is being executed as a new job arrives. Here, the planner considers 5 reference states as potential initial states for the new plan.



The planner has computed an updated plan that starts from the second-to-last reference state. The initial plan is executed until that point before switching to the new plan. The goal achievement time is the time from the start of the second planning phase to the end of the overall execution.

In order to obtain interesting benchmark instances, we tried to ensure that the second planning phase starts and ends during the first planned execution. Thus, we generated the instances such that the second set of goals appears after 10% of the initial plan is

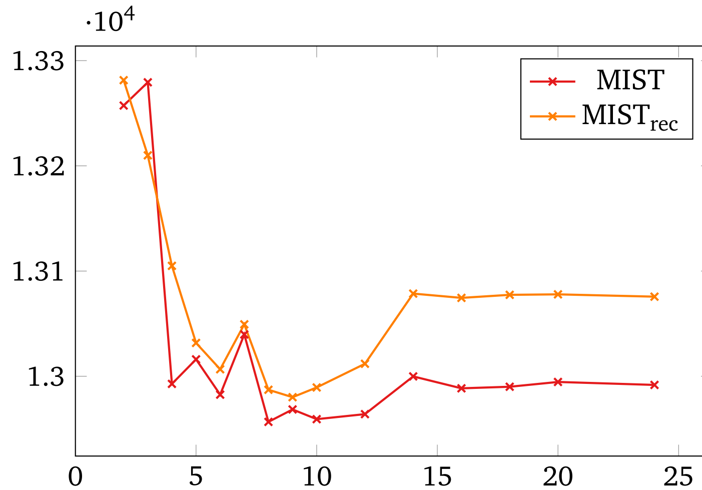


Figure 7.1: GAT as geometric mean over all instances (y-axis) for MIST with different numbers of reference states (x-axis).

executed. Furthermore, we estimated the length of the second planning phase by running the planner offline for both goals from the original start state, and used that to generate different experimental setups where the second planning phase is estimated to end at $E = 0.2, 0.3, \ldots, 0.9$ of the initially planned execution. This is achieved by adjusting the factor for the translation of the action cost to execution time, thereby changing the duration of the initial execution.

7.6.2 Results

We compare MIST to the following baselines:

finish Finish execution and plan only for the new goals.

stop Stop execution and re-plan from the current state.

approximate Approximate the duration of the re-planning phase, and use the state where the agent is expected to be at that time as the deviation state. We use the same estimation for the number of expansions as MIST, i.e., $\eta(m) = delay \cdot d(m)$, using the average expansion delay from the initial planning phase and the estimated distance of the current state.

fixed latency Stop execution at a fixed point in time (we test values of $10^1, 10^2, \ldots, 10^7$ expansions for this time point). We also consider a theoretical oracle configuration that chooses the best-performing time point to stop the execution (out of the tested values) per instance.

MIST has one important parameter: the selection of the reference states. In our implementation, we set a number of reference states n_R, which are then selected in uniform intervals from the current plan. Figure 7.1 shows the goal achievement time (in number of expansions) for different values of n_R across our full benchmark set. If there are too few reference states, the algorithm does not have the best starting point for the next plan available. On the other hand, the performance also decreases if too many reference states are used, as it becomes more difficult to settle on the most promising one quickly (especially if the planning time estimation is not very accurate). Across the tested numbers of reference states, MIST chooses nodes for expansion corresponding to the reference state which is used for the solution 38% of the time on average, more for fewer reference states (55% for $n_R = 3$), and less the more reference states are used (30% for $n_R = 24$). The overall best results are obtained with $n_R = 8$ for MIST and $n_R = 9$ for MIST_{rec}, and we use these settings for the remaining experiments.

Figure 7.2 shows the relative goal achievement time compared to MIST for the considered algorithms for different expected end points of the second planning phase. If the planning time is very short compared to the execution time (small values of E), stopping the execution as soon as possible works well, but loses out compared to MIST if planning is non-trivial ($E > 0.2$). As expected, finishing the execution becomes better with increasing expected planning times, though MIST always performs better. The fixed latency configurations offer some interpolation between the two extremes of stopping or finishing the execution. Given the diversity of our benchmark set, a fixed latency can not accurately predict the planning time, and these configurations are outperformed by MIST. The approximation baseline also works well for short planning times, but is prone to overestimate. On average, MIST reduces the goal achievement time by 8.6% compared to stopping and re-planning immediately, by 6.8% compared to finishing the planned execution, and by 5.1% compared to approximating the re-planning time.

Figure 7.3 gives more insight into the individual domains. The observations from the overall results hold across most domains, with minor exceptions. On VisitAll, the *approximate* baseline comes very close to MIST on average, beating it for some values of E. This can can be attributed to the planning time estimation being more accurate. While that also helps MIST to select the correct reference state to expand towards more frequently (46% of the time compared to 35% on other domains), MIST can still suffer from the added overhead. In the Rovers domain, MIST and MIST_{rec} outperform all competitors for all values of E, and may even beat the oracle (which can be inaccurate if the best deviation state is between two of the considered time points). Both *stop* and *approximate* perform particularly poorly in that domain, with up to 35% respectively 23% worse goal achievement time compared to MIST when considering large expected planning times.

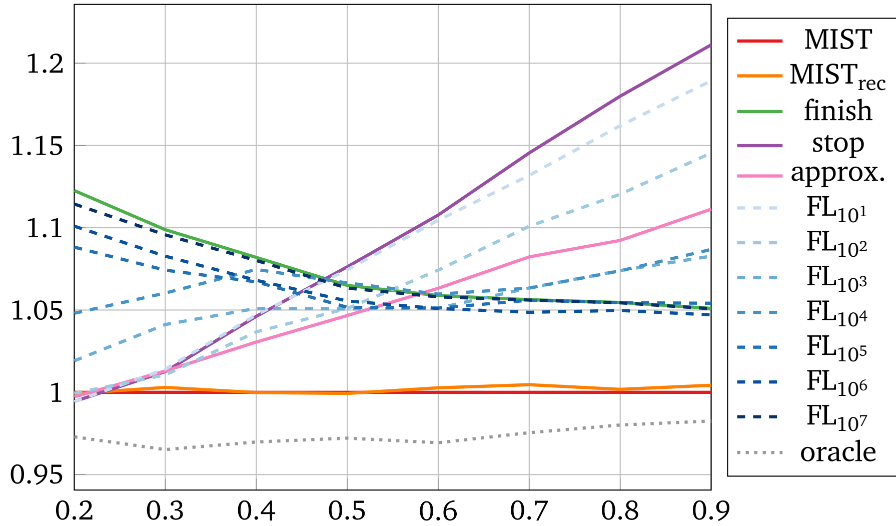


FIGURE 7.2: GAT as geometric mean over all instances relative to MIST (y-axis) for $E = 0.2, 0.3, \dots, 0.9$ (x-axis).

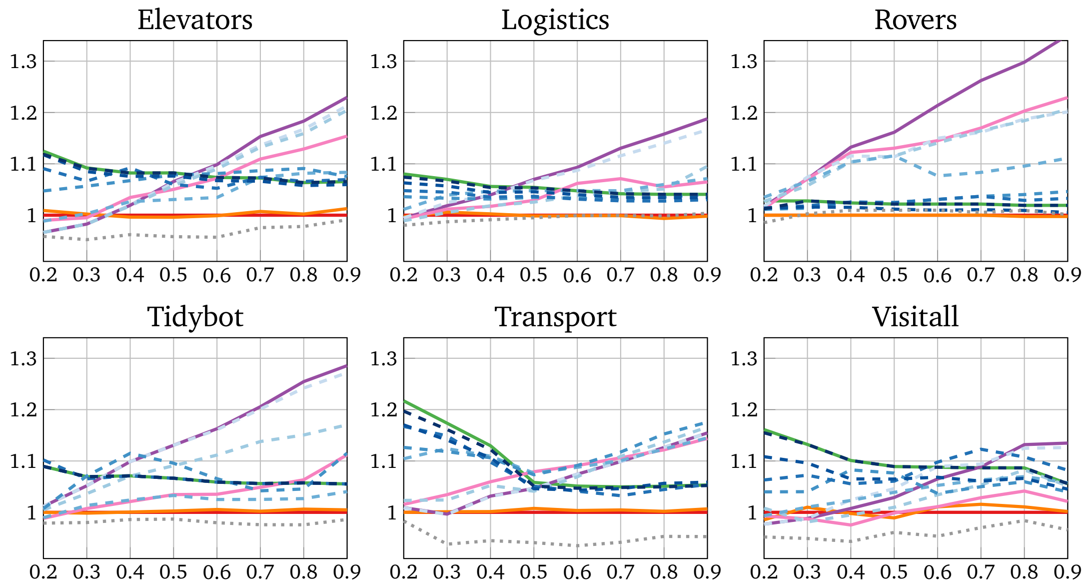


FIGURE 7.3: GAT relative to MIST (y-axis) for $E = 0.2, 0.3, \dots, 0.9$ (x-axis).

Conversely, *finish* comes close to MIST, which indicates that interrupting the execution while re-planning is particularly costly in that domain.

In these experiments, MIST$_{rec}$ and MIST exhibit similar performance. This suggests two conclusions. First, the way we generate testing instances averages out the edge cases in which MIST$_{rec}$ significantly outperforms MIST. Second, even though MIST$_{rec}$ does not prune reference nodes, it is able to effectively focus its search effort just as well as MIST.

7.7 Conclusion

Selecting the initial state for online replanning is a problem that has so far been largely glossed over and tackled by ad-hoc solutions such has hand-tuned replanning time estimates. With MIST, we introduced a principled technique by considering multiple states along the ongoing execution as potential initial states for the new plan. This enables MIST to adaptively select the state that minimizes the overall planning and execution time. We showed that this approach leads to strong and robust performance in scenarios with online goal arrival, outperforming several baselines on planning domains adapted to this setting.

One practical improvement for MIST is to follow an anytime approach [e.g., Likhachev et al., 2005], i.e., continuing the planning process even after the first plan has been found in order to find a better plan. While this idea seemingly contradicts Theorem 7.2, the assumption of an admissible planning time estimate is typically impossible in practice. Generally, allowing more interaction between the search process and the ongoing execution can be useful, for example by signaling execution to stop at a reference state that the search find promising to avoid potentially having to move back.

Another interesting direction for future work would be modeling the uncertainty of the estimates used by MIST, similar to the work presented in Chapters 8 and 9. In particular, the basic version of MIST attempts to estimate whether planning will finish before the execution reaches a certain state, but only uses that information to generate a binary value (yes or no). Instead, it might be worth attempting to estimate the probability of meeting that deadline, allowing MIST to take some risk in order to reduce the expected search time.

8 Exploiting Heuristic Uncertainty in Deterministic Real-Time Search

Some applications of heuristic search impose real-time constraints, forcing the agent to select its next action within a fixed time frame, typically before a full plan can be generated. For example, in video games, players expect characters to move immediately after giving a corresponding command [e.g., Bulitko et al., 2010; Lawrence and Bulitko, 2013; Kiesel et al., 2015]. Another application is autonomous aircraft control [Bulitko et al., 2008] or similar situations where an agent must quickly commit to actions while navigating through unknown terrain or reacting to local observations.

Most real-time search algorithms follow the iterative three-phase paradigm initially described in Korf's [1990] seminal work:

1. Perform a lookahead search with a limited number of expansions from the agent's current state.
2. Consider the heuristic values of the frontier nodes and the costs of the transitions towards them to find the cheapest estimated path to a goal, and commit to the first action along that path.
3. Update the heuristic values on the local lookahead search space to avoid moving in circles and ensure progress.

The perhaps most popular instantiation of this concept is LSS-LRTA* [Koenig and Sun, 2009], which uses A* as the lookahead search algorithm, commits to the action towards the frontier node with minimal f value, and updates heuristic values using a Dijkstra-like backup procedure.

While this approach is elegant and often yields good results, it neglects the inherent uncertainty of being forced to make a commitment without full knowledge about the

search space outside the locally explored area. As observed early on [e.g., Mutchler, 1986; Pemberton and Korf, 1994; Pemberton, 1995], the best lookahead search strategy is thus not necessarily to expand towards the most promising search node. Instead, it can be useful to collect more information about other areas of the search space in order to gain confidence that those parts do not lead to potentially better states.

Nancy is a recently developed real-time search algorithm that explicitly reasons about this kind of uncertainty [Mitchell et al., 2019]. Instead of heuristic values, Nancy considers beliefs (probability distributions) about the true cost to a goal state, and uses a lookahead search strategy based on minimizing the risk that the most promising node is not the correct one. In the update phase, Nancy modifies the beliefs, maintaining its modeling of the heuristic uncertainty.

While Nancy has shown to be effective in practice, a theoretical analysis has been missing so far. We close that gap, and prove that Nancy is indeed a complete real-time search algorithm, giving more insight into its search behavior. Our proof applies to a general class of real-time search algorithms, including LSS-LRTA*. Furthermore, we impose fewer constraints on the heuristic: in contrast to the original completeness proof for LSS-LRTA* [Koenig and Sun, 2009], we do not require the heuristic to be consistent or admissible. We also introduce the notion of persistence, which aims to avoid erratic behavior of the agent without affecting completeness.

Papers and Contributions This chapter is based on the paper "Beliefs We Can Believe in: Replacing Assumptions with Data in Real-Time Search" [Fickert et al., 2020]. The paper was principally developed by the author, Tianyi Gu, and Leonhard Staut, in joint work with Wheeler Ruml, Jörg Hoffmann, and Marek Petrik. This chapter focuses on the theoretical analysis, which is the author's contribution. The generalized Nancy algorithm (Section 8.2) has been co-developed by the author, Tianyi Gu, and Leonhard Staut, based on earlier work by Mitchell et al. [2019]; Nancy's persistent action selection is the author's work.

8.1 Background

We next give a formal definition of the real-time search framework that we consider in this work, and briefly summarize the background on LSS-LRTA* and Nancy.

8.1.1 Problem Definition

Real-time search is a state-space-search problem where the agent is subject to real-time constraints. The search may explore a region of the search space starting from the current state of the agent, and must provide the next action to apply within a predefined and fixed time frame. We model this time bound as a number of expansions, and assume the bound to be greater than zero (i.e., we can always expand at least the agent's current state). The goal is to lead the agent to a goal state as quickly as possible.

The real-time search framework naturally leads to algorithms that follow the iterative three-phase design of a local lookahead search (the *lookahead phase*), committing to an action that leads to the most promising state on the search frontier (the *action selection*), and updating heuristic values on the lookahead search space to avoid cycles (the *learning phase*) [Korf, 1990]. We next give a recap on LSS-LRTA* [Koenig and Sun, 2009] as the most canonical instantiation of this paradigm, and then discuss the Nancy algorithm as introduced by Mitchell et al. [2019].

8.1.2 LSS-LRTA*

LSS-LRTA* [Koenig and Sun, 2009] instantiates the lookahead phase with an A* search. However, expanding the best nodes according to their f value may not be the best use of the limited time to explore the local search space [Mutchler, 1986]. A better approach can be to take the error of the heuristic into account, and use the expected value $\hat{f}$ (an estimate of the true cost to a goal f^*) instead of the lower bound f to guide the lookahead [Kiesel et al., 2015].

In the action-selection step, the algorithm must decide which action the agent should execute next, based on the knowledge gained in the lookahead phase. In LSS-LRTA*, the action leading to the frontier node with minimal f value is chosen.

Finally, the heuristic values of the local search space are updated in the learning phase. This is done by backing up the information of the frontier nodes towards the root state of the lookahead search. LSS-LRTA* uses a Dijkstra-like update procedure of the heuristic values h, improving the estimates of the states in the local search space for future lookahead phases. This process propagates the heuristic values from the best successor to their parent:

$$h(s) := \min_{a \in \mathcal{A}(s)} (h(s[\![a]\!]) + c(a)).$$

8.1.3 Real-Time Search as Decision Making Under Uncertainty

Nancy [Mitchell et al., 2019] is a recent real-time search algorithm based on the idea of casting real-time search as planning under uncertainty. The algorithm replaces the scalar heuristic estimates by probability distributions over the true cost to the goal. Such beliefs can be constructed as Gaussian distributions around the expected value $\hat{f} := g + \hat{h}$, where $\hat{h}$ is a potentially inadmissible estimate that corrects the heuristic for online observations of the observed error [Thayer et al., 2011]. The variance of the distributions is also derived from the heuristic error. Since the error can be expected to decrease when getting closer to a goal, so does the variance of the beliefs. As pointed out by Fickert et al. [2020], the belief distributions obtained this way rely on assumptions on the behavior and error model of the heuristic, and do not necessarily resemble actual cost-to-goal distributions. However, they show that Nancy can also be instantiated with distributions generated by per-domain offline learning, which alleviates some of the assumptions made by Nancy.

We explain the generalized Nancy framework in more detail in the following section, and then prove its completeness (Section 8.3).

8.2 The Nancy Framework

Like LSS-LRTA*, Nancy follows the three-phase paradigm of lookahead, action selection, and heuristic updates. The core of the Nancy algorithm is shown in Algorithm 10; we explain the details of the three phases in the following.

8.2.1 Risk-Based Lookahead

In the lookahead phase (line 4), Nancy aims to minimize the regret that would be incurred in case the selected action for the agent later turns out not to be the best choice. This is calculated by pairwise considering the best frontier nodes under each top-level action, and simulating each expansion to estimate which one will most effectively increase the confidence in the node that the agent is going to move towards.[1] The lookahead returns the frontier node with minimal expected value $\hat{f}$, breaking ties by $\hat{h}$.

[1] More details on the risk-based lookahead can be found in Fickert et al.'s [2020] work. The specifics of this part are not relevant to our proofs, so we omit them here.

Algorithm 10: Nancy

```
1  s := s_start
2  π_curr := ⟨⟩
3  while s⟦π_curr⟧ is not a goal state do
4      Let t be the best frontier node resulting from a risk-based lookahead from s, and let
         π be the path from s to t.
5      π_curr := update_path(s, t, π_curr, π)
6      apply_next(s, π_curr)
7      Perform Nancy backups on the local search space.
8  while s is not a goal state do
9      apply_next(s, π_curr)
10 fn update_path(s, t, π_curr, π)
11     if (s⟦π_curr⟧ was expanded in the lookahead or
12         t is a goal state or
13         f̂(t) < f̂(s⟦π_curr⟧) or
14         f̂(t) = f̂(s⟦π_curr⟧) and ĥ(t) < ĥ(s⟦π_curr⟧)) then
15         return π
16     return π_curr
17 fn apply_next(s, π_curr)
18     let a_0, ..., a_n be the action sequence of π_curr
19     s := s⟦a_0⟧
20     π_curr := ⟨a_1, ..., a_n⟩
```

8.2.2 Persistent Action Selection

In the original description of Nancy [Mitchell et al., 2019], the search would commit to the first action on the plan towards the best state t found in the lookahead. However, the disparity in guidance of the lookahead search (which uses risk) and the action selection (which uses $\hat{f}$) can cause the agent to move back and forth between states until the backups sufficiently update the beliefs to allow the agent move on. A slight change from the original Nancy algorithm can improve this situation—instead of selecting a new course of action after each lookahead, we follow along the plan towards the best known state so far (π_{curr}). Nancy keeps moving along π_{curr} (line 6) until it reaches the target state, and only deviates from that plan if a strictly better state (either in terms of being a goal state or having strictly lower $\hat{f}$ value) is discovered in the lookahead (see the `update_path()` function, line 10). We call this technique *persistent* action selection. Persistence aims to improve search performance by preventing Nancy from meandering around; Fickert et al. [2020] found that it improves performance in classical planning, and is sometimes slightly better and sometimes slightly worse in other search benchmarks.[2]

[2] In the conference paper where we originally introduced persistence [Fickert et al., 2020], we further assumed that it is necessary for completeness. However, as we show in our updated proofs in Section 8.3,

If the lookahead returns a goal state, π_{curr} will be updated to a path towards it, and the agent executes that path without doing further iterations of lookahead (lines 8 and 9). Thus, whenever a goal state is seen the lookahead, the agent will eventually arrive at that state. We call this property *goal-awareness*, and it will be important to prove completeness later on.

8.2.3 Nancy Backups

Nancy's update procedure (invoked in line 7) is similar to that of LSS-LRTA*, but adapted for belief distributions. The belief of a state s is updated to the belief of the best successor, shifting the expected value of the distribution by the action cost:

$$\mathcal{B}(s) := \mathcal{B}(s[\![a^*]\!]) + c(a^*), \text{ where } a^* := \arg\min_{a \in \mathcal{A}(s)} (\hat{h}(s[\![a]\!]) + c(a)).$$

The effect on the expected cost $\hat{h}$ is equivalent to the heuristic update used by LSS-LRTA*, i.e., it satisfies the equation

$$\hat{h}(s) = \min_{a \in \mathcal{A}(s)} (\hat{h}(s[\![a]\!]) + c(a)).$$

This will become important for our proofs in the following section.

8.3 Theoretical Analysis

In this section, we show that Nancy is complete. We give a completeness proof for a more general class of real-time search algorithms, and then show that Nancy is a member of this class. Our proofs use $\hat{h}$ to denote the updated heuristic, which, for real-time algorithms based on belief distributions such as Nancy, is the expected value of the (updated) belief distributions.

Our analysis is based on the following assumptions:

(A1) all action costs are strictly positive;

(A2) for every state, there is a goal reachable from it;

(A3) all initial beliefs have a finite expected value; and

(A4) the state space is finite.

requiring actions to be selected by minimal $\hat{f}$ value is sufficient. Nancy satisfies this both with and without using persistence.

Our proof follows the style of Korf's [1990] proof for RTA* and Bulitko and Sampley's [2016] proof for Weighted Lateral LRTA*: We first prove that incompleteness implies that there must exist a subset of states within which the agent circulates forever. Then we prove that there cannot exist such a set due to the updates made by the real-time search algorithm's learning rule.

Definition 8.1. In a real-time search algorithm, a subset of states $S_\circ$ is called a *circulating set* if there exists a time $t_\circ$ after which the agent will visit only states $s \in S_\circ$ and visit each one an infinite number of times.

Lemma 8.2. *Under assumptions (A2) and (A4), if a real-time search algorithm is incomplete, it must have a circulating set $S_\circ$.*

Proof. Since a goal is reachable from all states (A2), a real-time search only terminates when it reaches a goal state, so incompleteness means that the search never terminates. Because the state space is finite (A4), there must exist a subset of non-goal states $S_\circ$ such that the agent will re-visit each of the states in $S_\circ$ an infinite number of times after some initial time t. Let S be the set of non-goal states. If there exist states $s \in S$ that are not visited an infinite number of times, let the last time such a state s is visited be t_s. Then $t_\circ := \max(t, t_s)$ as per Definition 8.1 satisfies the claim. □

Definition 8.3. A real-time search is called *goal aware* if, upon generating a goal state in its lookahead, it commits to the path towards it.

Nancy is goal aware (see Section 8.2.2) and so is LSS-LRTA* [Koenig and Sun, 2009, Figure 5, line 32].

Lemma 8.4. *Under assumptions (A2) and (A4), if a goal-aware real-time search algorithm has a circulating set $S_\circ$, then (1) there exists a finite set of non-goal states $S_\infty \supseteq S_\circ$ that are expanded infinitely often, (2) every successor state $s' \in S_F := \{s[\![a]\!] \mid s \in S_\infty, a \in \mathcal{A}(s)\} \setminus S_\infty$ appears infinitely often in the frontier of lookaheads from states in $S_\circ$, (3) there is a time t_1 after which no $s' \in S_F$ is expanded, and (4) S_F is non-empty.*

Proof. The lookaheads from all states $s \in S_\circ$ are performed infinitely often with a fixed number of expansions, so for each $s \in S_\circ$ there must be a set of states $S^s_\infty \supseteq \{s\}$ that are expanded infinitely often, and so is their union S_∞. The set S_∞ cannot contain a goal as a goal-aware search would head towards it, breaking the circulation. The neighbor states S_F are obviously generated infinitely often from S_∞; as they are not in S_∞, they are expanded only a finite number of times so the claimed time t_1 exists. This set S_F is non-empty because S_∞ does not contain a goal state, but the goal is reachable from all states (A2), so S_F must contain at least one state on a path to the goal. All of these sets must be finite since the state space is finite (A4). □

Definition 8.5. A learning algorithm is called *dynamic-programming-like* if it updates the heuristic values of the states S expanded in the local search space (and no others) such that afterwards the heuristic values of all $s \in S$ are *locally consistent*, that is, satisfy

$$\hat{h}(s) = \min_{a \in \mathcal{A}(s)} (c(a) + \hat{h}(s[\![a]\!])).$$

A standard result is that, if updates of the form $\hat{h}(s) := \min_{a \in \mathcal{A}(s)}(c(a) + \hat{h}(s[\![a]\!]))$ are performed infinitely often on a finite state-space graph, then the state values will eventually converge from arbitrary finite initial values to locally consistent ones (assuming positive action costs and reachable goals) [Bertsekas and Tsitsiklis, 1996, Proposition 2.3].

Lemma 8.6. *Under assumptions (A1)-(A4) with S_∞ as in Lemma 8.4, if a goal-aware real-time search algorithm that performs dynamic-programming-like learning has a circulating set $S_\circ$, then there exists a time t_2 after which, for every $s \in S_\infty$, $\hat{h}(s)$ will be locally consistent.*

Proof. Consider the state space sub-graph S' induced by $S_\infty \cup S_F$ (Lemma 8.4). After time t_1 as per Lemma 8.4, the update operation is performed infinitely often on all states $s \in S_\infty$, and only on those states. All update operations are well defined, i.e., consider successors contained in S'. Due to the convergence of the dynamic-programming-like learning procedure with positive action costs (A1) and bounded initial $\hat{h}$ values (A3), the $\hat{h}$ values of all states $s \in S_\infty$ will eventually converge to a solution of the state-update equation as claimed. □

In the following, we will use f_s and g_s to denote the f value and g value, respectively, of a state in a lookahead search space with respect to the current root state s of the lookahead.

Lemma 8.7. *Under assumptions (A1), (A2), and (A4), if $\hat{h}$ is locally consistent on all states S_E^s expanded in a lookahead search space rooted at a state s that does not contain a goal, then all states s' on a cheapest path within the lookahead from s to a frontier node with minimal $\hat{f}_s$ have $\hat{f}_s(s') = \hat{f}_s(s)$.*

Proof. Let S_F^s be the frontier nodes of the lookahead from s. Due to assumption (A2), S_F^s must be non-empty since S_E^s does not contain a goal state. Let π be a path from s to the frontier such that each step satisfies the state-update equation (i.e., taking the arg min according to Definition 8.5). Such a path must exist because the lookahead search space is finite due to assumption (A4) and the heuristic is locally consistent on each state within the lookahead. Furthermore, due to positive action costs (A1), the path can not contain cycles and hence must eventually reach the frontier. Each state s' on π has $\hat{f}_s(s') = \hat{f}_s(s)$ by construction. This includes the state $s[\![\pi]\!] \in S_F^s$, thus $\hat{f}_s(s)$ is an upper bound on the

minimal $\hat{f}_s$ value among the frontier nodes. Observe that, due to local consistency, $\hat{f}_s$ can only increase on any path from s to the frontier. Therefore, there can not be a state $s_f \in S_F^s$ with $\hat{f}_s(s_f) < \hat{f}_s(s)$. Furthermore, for all states $s_f \in S_F^s$ with $\hat{f}_s(s_f) = \hat{f}_s(s)$, there must be a path from s to s_f such that all states s' on that path have $\hat{f}_s(s') = \hat{f}_s(s)$ as claimed. □

In particular, Lemma 8.7 holds in each lookahead search space that does not contain a goal after the corresponding learning phase of a real-time search algorithm that performs dynamic-programming-like learning. Furthermore, it holds for every lookahead search space that only expands states $s \in S_\infty$ after time t_2 as per Lemma 8.6.

Definition 8.8. A real-time search algorithm is called *reasonable*, if it (1) is goal aware, (2) uses dynamic-programming-like learning, and (3) performs action selection by $\hat{f}$, i.e., unless the search already discovered a goal which it is moving towards, after the lookahead it applies the first action of the cheapest path (within the lookahead) towards a frontier node with minimal $\hat{f}$ value.

Lemma 8.9. *Under assumptions (A1)-(A4), a reasonable real-time search algorithm cannot have a circulating set.*

Proof. We proceed by contradiction. Assume that the search algorithm does have a circulating set $S_\circ$. Then there must be sets S_∞ and S_F as per Lemma 8.4. After some point in time t_2 as per Lemma 8.6, $\hat{h}$ converged to locally consistent values on all states $s \in S_\infty$. Let $s = \arg\min_{s \in S_\circ} \hat{h}(s)$ be a state from the circulating set with minimal $\hat{h}$ value after convergence. Since s is visited infinitely often, there must be a time $t_3 > t_2$ when a lookahead originates from s.

Let $s_f = \arg\min_{s_f \in S_F^s} \hat{f}_s(s_f)$ be a frontier node with minimal $\hat{f}_s$, and let $s, s_1, \ldots, s_f$ be the states along the cheapest path in the lookahead to s_f. According to Lemma 8.7, the $\hat{f}_s$ values on this path must all be equal, i.e., $\hat{f}_s(s) = \hat{f}_s(s_1) = \cdots = \hat{f}_s(s_f)$. Due to the assumption of positive action costs (A1), this implies $\hat{h}(s) > \hat{h}(s_1) > \cdots > \hat{h}(s_f)$. Following action selection by $\hat{f}$, the agent will make a step towards s_f by moving to s_1 after the lookahead, implying that $s_1 \in S_\circ$. Since s was selected to have minimal $\hat{h}$ value among all states in $S_\circ$, we have a contradiction with $\hat{h}(s_1) < \hat{h}(s)$. □

Theorem 8.10. *Under assumptions (A1)-(A4), a reasonable real-time search algorithm will eventually reach a goal.*

Proof. If the search never reaches a goal it has a circulating set according to Lemma 8.2, contradicting Lemma 8.9. □

We next show that Nancy is a reasonable search algorithm according to Definition 8.8, which allows us to prove its completeness. However, in order to prove that Nancy does action selection by $\hat{f}$ even when using persistence, we first need the following lemma.

Lemma 8.11. *Under assumptions (A1), (A2), and (A4), if Nancy does not switch to a new path after the lookahead from a state s (i.e., does not go into the* **then** *branch in the* `update_path()` *function, see Algorithm 10, line 10), then (1) π_{curr} must lead through the frontier, (2) the state of π_{curr} that lies on the frontier has minimal $\hat{f}_s$ value among all frontier nodes, (3) $\hat{f}_s(s') = \hat{f}_s(s)$ for all states s' on π_{curr}, and (4) the $\hat{h}$ values of these states remain unchanged in the learning phase.*

Proof. Let s_l be the root state of the lookahead when π_{curr} was set. In that lookahead, $s_l[\![\pi_{curr}]\!]$ was a frontier node with minimal $\hat{f}_{s_l}$ value, and, after the learning phase, all states s' on π_{curr} have equal $\hat{f}_{s_l}$ according to Lemma 8.7. When Nancy moves along π_{curr} in the action selection phase, the $\hat{f}$ values relative to the current root state are equally reduced by the cost of the applied action for all remaining states along π_{curr}. Thus, only the heuristic update in the learning phase could invalidate the claim that all states along the path have equal $\hat{f}_s$ value.

If Nancy does not switch to a new path after a lookahead rooted at s, then we know that (a) $s[\![\pi_{curr}]\!]$ was not expanded, so π_{curr} must lead through the frontier, and (b) there is no state s_f on the frontier of the lookahead with $\hat{f}_s(s_f) < \hat{f}_s(s[\![\pi_{curr}]\!])$. Before the learning phase, all states s' on π_{curr} have $\hat{f}_s(s') = \hat{f}_s(s)$. Since one such state s' is on the frontier (a) and has minimal $\hat{f}_s$ value (b), this still holds after the learning phase according to Lemma 8.7, and the $\hat{h}$ values can not have changed. □

Now we can prove that Nancy is reasonable as it performs action selection by $\hat{f}$ even when following the path for persistence.

Lemma 8.12. *Nancy is a reasonable real-time search algorithm.*

Proof. Nancy is goal aware and does dynamic-programming-like learning (Section 8.2.3). For action selection, Nancy distinguishes two cases: If Nancy does not continue along the cached path, the new path is selected towards a frontier node with minimal $\hat{f}$ value. Otherwise, Nancy follows the path cached for persistence, which, according to Lemma 8.11, also leads to a frontier node with minimal $\hat{f}$ value. Thus, Nancy does indeed perform action selection by $\hat{f}$ in both cases, and satisfies the definition of a reasonable real-time search algorithm. □

Corollary 8.13. *Under assumptions (A1)-(A4), Nancy will eventually reach a goal.*

Proof. Nancy is a reasonable real-time search algorithm (Lemma 8.12), and is thereby complete under assumptions (A1)-(A4) via Theorem 8.10. □

Theorem 8.10 also implies that LSS-LRTA* is complete even for heuristics that are not consistent or admissible (a consistent heuristic is the only case proven in the original paper [Koenig and Sun, 2009]):

Lemma 8.14. *LSS-LRTA* is a reasonable real-time search algorithm.*

Proof. LSS-LRTA* is goal aware [Koenig and Sun, 2009, Figure 5, line 32], does dynamic-programming-like learning [Koenig and Sun, 2009, Figure 5, lines 17–26], and does action selection by $\hat{f}$ [Koenig and Sun, 2009, Figure 5, line 34].[3] □

Corollary 8.15. *Under assumptions (A1)-(A4), LSS-LRTA* will eventually reach a goal.*

Proof. LSS-LRTA* is a reasonable real-time search algorithm as per Lemma 8.14, and is thereby complete under assumptions (A1)-(A4) via Theorem 8.10. □

8.4 Conclusion

We proved that Nancy is complete under minimal assumptions (e.g., that the initial beliefs have a finite expected value), complementing its success in practice with theoretical justification. Our theoretical analysis not only yields completeness for Nancy, but for an entire class of real-time search algorithms. We were able to show that LSS-LRTA* is a member of this class. In contrast to the original completeness proof of LSS-LRTA*, our proof does not require the heuristic to be admissible or consistent, opening up LSS-LRTA* for a wider range of heuristics. Our theoretical framework enables future algorithms to prove completeness by simply arguing that they are reasonable—which is much easier to do than proving completeness from scratch.

[3] Koenig and Sun [2009] use a slightly different real-time search setting, allowing the agent to execute the full path to the frontier node in a single step instead of just the first action. However, our results are not affected by this difference, as the arguments made in the proof of Lemma 8.9 would still hold if the entire path were executed.

9 Exploiting Heuristic Uncertainty in Suboptimal Search

Many practical applications are too complex to solve optimally under realistic time and memory constraints: Proving optimality necessitates exploring all states that could potentially lead to a better solution, which is extremely costly in large state spaces even if the heuristic is almost perfect [Helmert and Röger, 2008]. If control of the solution quality is desired, this leaves two options: bounded-cost search, where solutions must satisfy an absolute cost bound C, and bounded-suboptimal search, where solutions must be within a factor w of an optimal solution.

In both of these settings, the aim is to find a solution satisfying the given bound as quickly as possible. In bounded-cost search, a simple approach is to run any suboptimal search algorithm and prune search nodes that violate the cost bound. However, this does not account for the likelihood of meeting the bound, and can cause some overhead if the heuristic is inaccurate and the search explores some branches deeper than necessary. Hence, specialized bounded-cost search algorithms have been devised that use the bound in the evaluation function guiding the search, such as potential search (PTS) [Stern et al., 2011; Stern et al., 2014] and Bounded-Cost Explicit Estimation Search (BEES) [Thayer et al., 2012].

PTS is based on the idea of prioritizing nodes with a higher probability of leading to a solution within the bound. This is achieved by sorting the open list according to a function that is shown to yield the same ordering as the desired probability, assuming the heuristic satisfies a linear error model. However, this search strategy ignores the "as fast as possible" aspect of bounded-cost search. BEES takes a different approach, using a distance estimate in addition to the heuristic in order to guide the search to solutions faster.

In this chapter, we introduce Expected Effort Search (XES), which combines the probability of finding a feasible solution as well as the search effort required to find such a solution into a single estimation function. Specifically, XES uses two estimates: $T(n)$,

the estimated time until a solution is found under the current node n, and $p(n)$, the probability that such a solution will be within the bound. XES then performs a best-first search on the expected effort $T(n)/p(n)$. This idea has previously been proposed by Dobson and Haslum [2017] and implemented as an internal objective function for specific planning heuristics. We generalize that idea to heuristic search with arbitrary underlying heuristics, and prove its correctness in a simplified formal model.

In contrast to PTS, XES requires a concrete estimate of the probability (where PTS simply relies on a function that yields the same ordering). We obtain such an estimate by modeling the heuristic uncertainty with Nancy-style belief distributions [Mitchell et al., 2019; Fickert et al., 2020; see also Chapter 8]. We use online observations of the heuristic error [Thayer et al., 2011] to adaptively obtain more accurate estimates of the expected solution cost. The probability that a search node leads to a solution within the cost bound is then derived from the distribution of the expected total cost corrected for the heuristic error.

Compared to bounded-cost search, bounded-suboptimal search poses the additional challenge that the exact bound is not known upfront. Hence, typical search algorithms use an admissible heuristic and keep track of the f values of open nodes, from which information about the suboptimality bound can be derived [Pearl and Kim, 1982; Gilon et al., 2016]. We show how XES can be adapted to this setting: We first introduce a relatively straightforward adaptation called Dynamic Expected Effort Search (DXES), prioritizing nodes based on their expected effort like XES. While DXES aims to find feasible solutions quickly, it neglects the implicit secondary objective of bounded-suboptimal search—gathering information about the suboptimality bound. We therefore design a more complex variant of DXES called Considerate Dynamic Expected Effort Search (CDXES) that addresses this issue by explicitly weighing the effort of raising the best known suboptimality bound against the search effort to find a solution under the current one. Finally, we introduce a simple round-robin scheme that can be instantiated with different node ordering strategies, including expected search effort.

We evaluate our novel contributions on the IPC benchmarks, in the most comprehensive comparison of bounded-cost and bounded-suboptimal search algorithms on the planning benchmarks to date. Our experiments show that XES is vastly superior to the current state of the art, exhibiting robust behavior across different domains and cost bounds. We find that our adaptations of XES to bounded-suboptimal search do not yield the same level of performance. However, our simple round-robin algorithms beat state-of-the-art bounded-suboptimal search algorithms, showing that there is still room for improvement for more principled approaches.

Papers and Contributions This chapter is based on the papers "Bounded-cost Search Using Estimates of Uncertainty" [Fickert et al., 2021a] and "New Results in Bounded-Suboptimal Search" [Fickert et al., 2022]. Both works are principally developed by the author and Tianyi Gu, in joint work with Wheeler Ruml. This chapter focuses on the author's contributions: The XES algorithm and its soundness proof are the author's work; the novel BEES variants (Section 9.2.2) and the DXES algorithm (Section 9.3.1.1) have been co-developed with Tianyi Gu; the CDXES algorithm is the author's work; the round-robin algorithms have been co-developed with Tianyi Gu (Section 9.3.2); the implementation in Fast Downward and evaluation on the IPC domains are the author's work.

9.1 Background

We first formally describe the bounded-cost and bounded-suboptimal search settings, and then give a brief overview over the relevant existing work in both areas.

9.1.1 Problem Definition

A *bounded-cost planning task* extends an FDR task with an absolute cost bound $C \in \mathbb{R}_0^+$. The cost of a plan π for such a task may not exceed C; if no such plan exists then the task is unsolvable. Similarly, a *bounded-suboptimal planning task* extends an FDR task with a suboptimality factor w, and the cost of a plan π may not exceed $w \cdot C^*$, where C^* is the cost of an optimal plan.

In addition to a heuristic h and distance estimate d, some search algorithms introduced in the following make use of potentially inadmissible estimates $\hat{h}$ and $\hat{d}$. Similar to $f := g + h$, we use $\hat{f}$ to denote $g + \hat{h}$.

9.1.2 Bounded-Cost Search

A straightforward strategy for bounded-cost search is to run an arbitrary search algorithm and prune any node n with $g(n) > C$ (or $f(n) > C$ when using an admissible heuristic) [e.g., Haslum, 2013]. However, this leaves the underlying search insensitive to the cost bound, and nodes that are close to the bound are treated the same way as ones that leave some room for error in the heuristic estimates. Hence, specialized bounded-cost search algorithms have been developed, taking C into account to guide the search.

9.1.2.1 Potential Search

Potential search (PTS) [Stern et al., 2011] aims to prioritize nodes that have a higher probability of leading to a solution within the cost bound C (the node's *potential*), which can be expressed as $P^C(n) = Pr(h^*(n) \leq C - g(n))$ for a search node n. This probability value is hard to estimate, and PTS instead constructs a function $u(n) := \frac{C-g(n)}{h(n)}$ that is used to guide the search. Stern et al. [2011] show that this function results in the same ordering of search nodes as P^C under the assumption that the heuristic follows a linear error model.

One drawback of the potential-based approach is that it neglects the search effort required to find a solution—a node close to the goal with similar potential as a node close to the initial state may not be prioritized. However, PTS has shown to be effective in practice on a wide range of search domains, even though it does not explicitly follow the main objective of bounded-cost search.

9.1.2.2 Bounded-Cost Explicit Estimation Search

Bounded-Cost Explicit Estimation Search [Thayer et al., 2012], short BEES, is a bounded-cost variant of Explicit Estimation Search (summarized in Section 9.1.3.2). BEES is a search algorithm using two queues: an open list ordered by f, and a focal list sorted by a distance estimate $\hat{d}$, containing only nodes with $\hat{f} \leq C$. As such, BEES makes use of three estimates to guide its search—the heuristics h and $\hat{h}$, and the distance estimate $\hat{d}$. The focal list contains only nodes that are believed to lead to solutions within the bound according to the inadmissible estimate $\hat{f}$, and is prioritized for expansion over the open list (falling back on the open list only if the focal list is empty). Sorting the focal list by the estimated goal distance aims to guide the search to a goal quickly. Thayer et al. [2012] also introduce a potential-based variant of BEES called BEEPS, where the open list is ordered by u instead of f.

9.1.2.3 The Expected Work Heuristic

Dobson and Haslum [2017] have introduced the idea of combining a probability estimate with an estimate of the search effort required to reach a goal from the current state. Given a probability $P^C(n)$ to reach a goal within the cost bound from a search node n, they observe that, on average, one can expect having to explore $1/P^C(n)$ many n-like nodes until a solution within the bound is found. In combination with an estimate of the required search effort $T^C(n)$ to explore the C-bounded subtree under n, the *expected work*

to find a feasible solution under n is given by $H^C := T^C(n)/P^C(n)$. Dobson and Haslum [2017] adapt pattern database heuristics [Culberson and Schaeffer, 1996] to use the heuristic H^C as an internal objective function for plans in the abstraction, and show that the approach works well on selected planning domains.

To instantiate P^C Dobson and Haslum [2017] use the potential u; the search time is estimated using a distance estimate to a goal state within the bound d^C raised to the power b, where b is the average branching factor of the search tree. They note that the potential u can be a poor approximation of the probability to reach a goal within the bound, and speculatively propose that P^C could instead be derived from online observations of the heuristic error. We further study this idea in this chapter, show how it can be implemented for arbitrary underlying heuristics, and compare it to the state-of-the-art algorithms for bounded-cost search.

9.1.3 Bounded-Suboptimal Search

The most well-known algorithm for bounded-suboptimal search is weighted A* [Pohl, 1970], which performs best-first search on $f_w = g + w \cdot h$. If h is admissible, then the solution found by weighted A* is guaranteed to be within a factor w of optimal.

9.1.3.1 Focal Search

The main difficulty of bounded-suboptimal search in comparison to bounded-cost search is that the bound is not known in advance (since the cost of an optimal solution C^* is not known). However, using an admissible heuristic, the minimal f value among open nodes, f_{min}, is an underestimation of C^*. *Focal search* is a two-queue approach based on this observation [Pearl and Kim, 1982]. It uses a standard open list sorted by f, and a *focal list* containing only nodes n with $f(n) \leq w \cdot f_{min}$, sorted by an arbitrary ordering function. This way of filtering which nodes qualify for the focal list ensures that any goal node found in the focal list satisfies the suboptimality bound: for a goal node n, we have $f(n) = g(n) \leq w \cdot f_{min} \leq w \cdot C^*$, and hence the path represented by n satisfies the optimality bound.

Most bounded-suboptimal search algorithms are instantiations or extensions of focal search, with different ordering functions or queue selection mechanisms. Pearl and Kim [1982] introduced A^*_ϵ (the first focal search algorithm for bounded-suboptimal search), which sorts the focal list by a distance estimate d in order to reach a goal more quickly. However, as observed by Thayer and Ruml [2009], the node with the minimal distance

estimate often has a high f value, so its successors frequently fail to qualify for the focal list. A^*_ϵ thus tends to be outperformed by more recent algorithms.

9.1.3.2 Explicit Estimation Search

Explicit Estimation Search (EES) [Thayer et al., 2012] uses three kinds of estimates to guide its search: an admissible heuristic h, an inadmissible heuristic $\hat{h}$, and a (potentially inadmissible) distance estimate $\hat{d}$. EES is an extension to focal search using three queues: (1) a standard open list sorted by f, giving access to the node $best_f$ with the minimal f value f_{min}, (2) an open list sorted by $\hat{f}$, giving access to the node $best_{\hat{f}}$ with the minimal $\hat{f}$ value $\hat{f}_{min}$, and (3) a focal list sorted by $\hat{d}$ which contains only nodes n with $\hat{f}(n) \leq w \cdot \hat{f}_{min}$, giving access to the node $best_{\hat{d}}$ with minimal $\hat{d}$ value among the nodes in the focal list.

At each expansion, one of the nodes at the front of these queues is selected according to the following rules: If $\hat{f}(best_{\hat{d}}) \leq w \cdot f_{min}$, then select $best_{\hat{d}}$ (this node is deemed to lead to a solution with in the bound and is close to the goal according to $\hat{d}$). Otherwise, if $\hat{f}(best_{\hat{f}}) \leq f_{min}$, then select $best_{\hat{f}}$ (this node is expected to lead to the best solution according to $\hat{f}$). Finally, if both cases fail, EES expands $best_f$, aiming to increase f_{min} in order to allow expanding $best_{\hat{d}}$ or $best_{\hat{f}}$ later.

9.1.3.3 Dynamic Potential Search

Dynamic Potential Search (DPS) [Gilon et al., 2016] is a bounded-suboptimal variant of Potential Search. The adapted function to sort the open list uses the dynamic bound $w \cdot f_{min}$ in place of the fixed cost bound C as in PTS: $ud(n) := \frac{w \cdot f_{min} - g(n)}{h(n)}$. In order to maintain the correct ordering, DPS uses a queue based on buckets of nodes with the same (g, h) values (and hence the same ud values), and reorders the buckets whenever f_{min} changes. DPS does not need to use a focal search approach as the node with minimal ud value is guaranteed to be within the suboptimality bound [Gilon et al., 2016, Section 4.1]. DPSU [Gilon et al., 2017] is a unit-cost variant of DPS that can sometimes improve over DPS, but is generally outperformed by DPS and EES.

9.2 Exploiting Heuristic Uncertainty in Bounded-Cost Search

In this section, we introduce our bounded-cost search algorithm based on Dobson and Haslum's [2017] idea of estimating the expected effort required to find a solution within

the bound, and prove that this form of search guidance is optimal under certain assumptions. We also present a new variant of BEES that uses explicit probability estimations, and evaluate our contributions on the IPC benchmarks.

9.2.1 Expected Effort Search

Extending the line of work initiated by Dobson and Haslum [2017], we propose a new bounded-cost search algorithm called Expected Effort Search (XES). We design a combined expected effort heuristic that accounts for both the probability of finding a solution within the cost bound and the effort required to do so. We first provide a simplified formal model to give a theoretical justification for this heuristic and then show how the probability estimate of finding a solution within the bound can be instantiated based on the belief distributions used by the Nancy algorithm [Mitchell et al., 2019; see also Chapter 8].

9.2.1.1 A Simple Formal Model

Let $p(n)$ be an estimate of a node n's potential, i.e., the probability that there is a solution under n within the cost bound. We will model expected search time using three assumptions:

(A1) We assume that the search procedure works exclusively in one subtree at a time. Let $T(n)$ be an estimate of the search effort to find a solution under n. Performing search below a node n can have two possible outcomes: either (a) a solution will be found under n with expected effort $T(n)$, or (b) the search does not find a solution under n.

(A2) For case (b), we assume that the search abandons that subtree after having spent $T(n)$ time, the same as in case (a).

(A3) Finally, we assume that the subtrees are independent: not finding a solution in one subtree does not affect the probability of finding a solution in any other subtree.

Now, let $\sigma = \langle n_0, n_1, \ldots, n_m \rangle$ be the ordering of the search nodes that are currently in the open list. We first have to consider the case where a solution is found below n_0, spending $T(n_0)$ time overall with a probability of $p(n_0)$. Otherwise, we need to next explore the search tree below n_1, where we would find a solution with overall probability $(1-p(n_0))p(n_1)$ (considering the remaining probability of not finding a solution below n_0) and expected overall search time $(T(n_0)+T(n_1))$ since we explore both the search

trees below n_0 and n_1, and so on. Abusing notation, let $T(n+m) = T(n) + T(m)$. Then, the expected overall search time for this ordering is

$$\begin{aligned} E(\sigma) = {} & p(n_0)T(n_0) + \\ & (1 - p(n_0))p(n_1)(T(n_0 + n_1)) + \\ & (1 - p(n_0))(1 - p(n_1))p(n_2)(T(n_0 + n_1 + n_2)) + \\ & \dots \end{aligned}$$

We note that there will be a final term in this expression representing the time in the case in which there is no solution; this quantity might be finite or infinite depending on the search tree and search procedure.

If we want to know whether we should prefer a node n over another node m, we can, without loss of generality, compare the open list orderings $\sigma_n = \langle n, m, \dots \rangle$ and $\sigma_m = \langle m, n, \dots \rangle$. Note that all but the first two terms of the corresponding expected search times are identical. We will use this to show that for ordering the open list, it is sufficient to use this shortened expression for expected search effort:

$$xe(n) = T(n)/p(n)$$

Theorem 9.1. *Given two open list orderings $\sigma_n = \langle n, m, \dots \rangle$ and $\sigma_m = \langle m, n, \dots \rangle$, we have $E(\sigma_n) < E(\sigma_m) \Leftrightarrow xe(n) < xe(m)$.*

Proof. We start with $E(\sigma_n) < E(\sigma_m)$. Removing equal terms from both sides yields

$$\begin{aligned} & p(n)T(n) + (1 - p(n))p(m)(T(n + m)) < \\ & p(m)T(m) + (1 - p(m))p(n)(T(m + n)). \end{aligned}$$

Multiplying out, we get

$$\begin{aligned} & p(n)T(n) + p(m)(T(n + m)) - p(n)p(m)(T(n + m)) < \\ & p(m)T(m) + p(n)(T(m + n)) - p(m)p(n)(T(m + n)). \end{aligned}$$

Simplifying by removing equal terms yields

$$\begin{aligned} & p(n)T(n) + p(m)(T(n + m)) < \\ & p(m)T(m) + p(n)(T(m + n)). \end{aligned}$$

Then by multiplying out again we get

$$p(n)T(n) + p(m)T(n) + p(m)T(m) <$$
$$p(m)T(m) + p(n)T(m) + p(n)T(n)$$

and after removing equal terms again:

$$p(m)T(n) < p(n)T(m).$$

Finally, dividing by $p(n)$ and $p(m)$ yields

$$T(n)/p(n) < T(m)/p(m). \qquad \square$$

This result is related to the optimal strategy for decision-theoretic troubleshooting, where a faulty component must be identified through sequential testing, and the best approach is based on a similar notion of expected effort [Kalagnanam and Henrion, 1988; Heckerman et al., 1995].

9.2.1.2 Estimating Expected Search Effort

XES is a best-first search on the expected search effort $xe(n) = T(n)/p(n)$. The key question is how to obtain the estimates $T(n)$ and $p(n)$. Here, we instantiate $T(n)$ with the debiased distance estimate $\hat{d}(n) = \frac{d(n)}{1-\epsilon_d}$, where ϵ_d is the mean one-step error in d [Thayer et al., 2011], and we derive $p(n)$ from Nancy-style probability distributions.

Nancy's [Mitchell et al., 2019] belief distributions model the true cost to a goal based on heuristic estimates. For a search node n, it creates a Gaussian distribution around $\hat{f}(n)$ with variance $(\frac{\hat{f}(n)-f(n)}{2})^2$. The distribution is truncated at a lower bound of $g(n)$ (or $f(n)$ if h is admissible). The probability of finding a solution under n within the cost bound C is then just the area below the distribution up to C, as illustrated in Figure 9.1. This can be calculated directly using the cumulative density function for truncated Gaussian distributions [Hald, 1952].

XES weighs the potential of a node against its estimated search effort. If the cost bound is very generous, then $p(n)$ should be close to 1 for most nodes and XES converges to a best-first search on $\hat{d}$, effectively guiding the search to a goal as quickly as possible. We prune nodes with $h(n) = \infty$ (dead ends) and nodes with $g(n) > C$, but not nodes with $T(n)/p(n) = \infty$ as this case can occur when the heuristic overestimates significantly (so

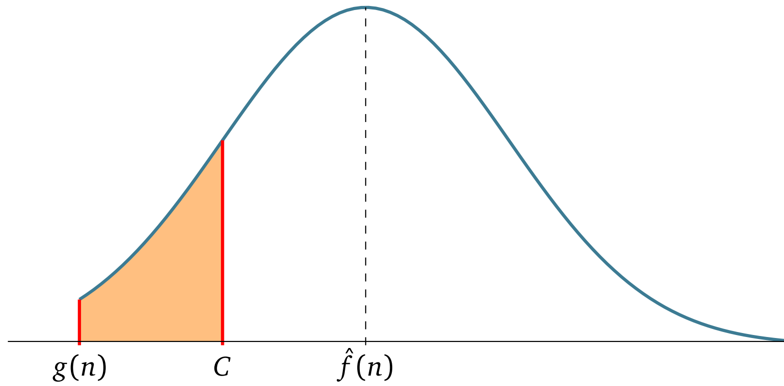


FIGURE 9.1: Estimating $p(n)$.

$p(n)$ is very low and may round to zero). XES satisfies the usual completeness guarantee from a systematic best-first search, as all nodes that may satisfy the cost bound are eventually expanded until a solution is found.

9.2.2 BEES with Explicit Probability Estimates

BEES and BEEPS maintain a focal list containing search nodes that are deemed likely to be within the bound. This is tested by comparing the inadmissible estimate $\hat{f}$ with the cost bound, i.e., checking whether a node n satisfies $\hat{f}(n) \leq C$. Instead, we can also use the explicit probability estimate derived from Nancy-style belief distributions as introduced in the previous subsection. This allows us to introduce new variants of these algorithms, called BEES95 and BEEPS95, that replace these checks with $p(n) \geq 0.95$.

9.2.3 Experimental Evaluation

While Theorem 9.1 suggests that the search strategy employed by XES is optimal, it relies on the strong simplifying assumptions (A1)–(A3), which may not be realistic in practice.

To evaluate XES and our new BEES variants, we use the benchmark set from the bounded-cost track of the 2018 International Planning Competition, which contains 180 instances from 9 domains. Furthermore, we consider the instances of the satisficing tracks of all previous IPCs, where we use the upper bounds from Planning.Domains [Muise, 2016] as the cost bounds; omitting instances where no bound is available, this leaves 2147 instances from 48 domains. These bounds correspond to best found solutions for these instances, and 1218 of them are known to be optimal.

We use h^{FF} as the heuristic h and use the length of the relaxed plans as the distance estimator d. All algorithms use a dual queue for preferred operators.

Coverage	GBFS	PTS	$\widehat{\text{PTS}}$	BEES	BEEPS	BEES95	BEEPS95	XES
Agricola (20)	**1**	0	0	0	0	0	0	0
Caldera (20)	8	10	11	10	10	12	11	**13**
Caldera-split (20)	**4**	2	2	2	2	2	2	2
DataNetwork (20)	2	0	0	3	3	3	3	**4**
Nurikabe (20)	4	10	7	10	10	**11**	9	9
Settlers (20)	4	5	7	10	10	**11**	**11**	**11**
Snake (20)	4	**5**	**5**	4	**5**	4	**5**	**5**
Spider (20)	7	**11**	9	10	10	10	9	9
Termes (20)	11	9	6	11	10	11	10	**13**
Sum (180)	45	52	47	60	60	64	60	**66**
Expansions ($*10^3$)	1.93	3.93	6.75	2.10	2.62	2.25	2.24	**1.77**
Search time (s)	**1.69**	4.11	7.20	2.14	2.79	2.32	2.39	1.91

TABLE 9.1: Coverage on the instances of the IPC'18 bounded-cost track. The last two rows show the and geometric means of the number of expansions (multiply by 10^3) and search time on commonly solved instances.

To compute the debiased heuristic $\hat{h}(n)$ for $\widehat{\text{PTS}}$, BEES, and XES, we use $\hat{h}(n) = h(n) + \epsilon_h \cdot \hat{d}(n)$ where ϵ_h is the mean one-step error in h [Thayer et al., 2011]. We initialize ϵ_h with 100 virtual samples to avoid a large fluctuation of $\hat{h}$ values at the beginning of the search. The initial value is set to -0.5, making initial $\hat{h}$ estimates optimistic.

The source code is available at `https://github.com/fickert/fast-downward-xes`. This repository also includes the implementations of the bounded-suboptimal search algorithms introduced in Section 9.3.

9.2.3.1 IPC'18 Results

Table 9.1 compares XES, BEES95, and BEEPS95 to previous bounded-cost search algorithms. We also include GBFS (with pruning on g), which was used by most planners in the 2018 IPC.

XES performs best overall with a coverage of 66, followed by BEES95 (64) and the other algorithms of the BEES family (60). The potential-based algorithms PTS and $\widehat{\text{PTS}}$ have significantly lower coverage, and the statistics on the number of expansions show that they are not as effective in guiding the search to a goal. GBFS is not competitive with the specialized bounded-cost search algorithms overall, though it can sometimes beat them if it quickly finds a cost-effective path to the goal (e.g., in Agricola and Caldera-split). On five of the nine domains, XES has the highest coverage of the considered algorithms. Across the commonly solved instances, XES also needs the fewest expansions on average,

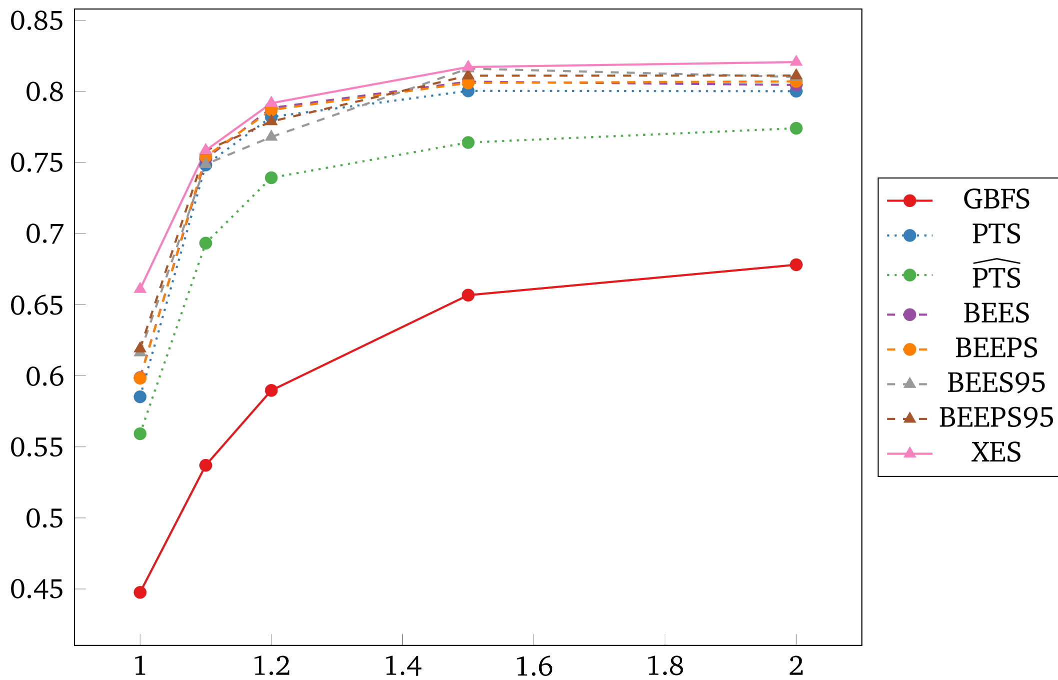


FIGURE 9.2: Normalized coverage as the cost bound is increased.

and its search time is only beaten by GBFS (which solves significantly fewer instances, but those that it can solve are solved quickly).

9.2.3.2 Satisficing Results

Table 9.2 shows the results on the instances of the satisficing tracks using the cost bounds from Planning.Domains. Consistent with the results on the IPC'18 bounded-cost instances, XES clearly outperforms the other considered algorithms: on 14 of the 48 domains it solves strictly more instances than any other algorithm, and on further 14 domains XES has the shared best coverage. Our new BEES variants BEES95 and BEEPS95 show small but relatively consistent improvements over their respective base algorithms. While PTS and $\widehat{\mathrm{PTS}}$ show good performance in some individual domains (e.g., Floortile and Freecell), they again lag behind the other bounded-cost search algorithms overall.

Figure 9.2 shows the normalized coverage as the cost bounds are multiplied by increasing factors. The relative strength of the algorithms remains mostly consistent across the different bounds, with XES being the best performing algorithm for all considered values, and the specialized bounded-cost search algorithms still have a significant advantage over GBFS with pruning even as the cost bound is relaxed.

Coverage	GBFS	PTS	$\widehat{\text{PTS}}$	BEES	BEEPS	BEES95	BEEPS95	XES
Airport (49)	26	24	24	31	31	**32**	**32**	31
Assembly (30)	17	18	18	25	24	**30**	29	**30**
Barman (40)	5	0	0	6	6	9	9	**10**
Blocksworld (35)	19	26	26	30	26	30	26	**31**
Cavediving (8)	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**
Childsnack (6)	1	0	0	0	0	0	0	**3**
CityCar (13)	1	4	4	4	4	**5**	**5**	4
Depot (21)	6	13	13	**14**	13	12	13	13
DriverLog (20)	10	**19**	18	18	18	18	**19**	15
Elevators (39)	7	12	12	22	22	**24**	**24**	**24**
Floortile (27)	6	**20**	**20**	8	8	9	8	9
Freecell (80)	20	**65**	57	37	47	39	58	41
GED (20)	0	**1**	0	0	0	**1**	**1**	**1**
Grid (5)	1	3	3	3	3	3	3	**4**
Gripper (20)	7	8	8	7	8	7	8	**12**
Hiking (18)	8	**18**	16	15	17	14	17	16
Logistics (63)	33	46	38	51	50	52	**53**	52
Maintenance (17)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Miconic (439)	342	362	427	403	426	397	426	**437**
Movie (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
Mprime (35)	29	**34**	33	33	33	**34**	33	33
Mystery (19)	17	**19**	**19**	**19**	**19**	**19**	**19**	**19**
Nomystery (18)	4	16	16	**18**	14	**18**	14	17
Openstacks (98)	33	31	41	53	54	56	56	**58**
Opt. Telegr. (4)	**4**	2	2	2	2	**4**	**4**	**4**
Parcprinter (40)	26	25	20	**29**	27	24	20	24
Parking (40)	4	**21**	10	11	12	16	13	15
Pathways (57)	13	20	**27**	22	22	25	22	24
Pegsol (35)	32	**35**	**35**	34	**35**	34	**35**	**35**
Philosophers (45)	**45**	11	5	11	11	11	11	14
Pipes-notank (46)	20	41	42	**43**	**43**	42	42	**43**
Pipes-tank (45)	15	29	28	28	32	26	31	**33**
PSR (116)	107	111	110	111	110	111	110	**113**
Rovers (40)	12	20	18	25	25	27	25	**30**
Satellite (36)	20	20	20	**23**	21	21	**23**	**23**
Scanalyzer (28)	20	16	17	20	18	**22**	17	21
Schedule (150)	36	51	47	62	73	60	75	**89**
Sokoban (30)	24	**29**	**29**	**29**	**29**	**29**	**29**	**29**
Storage (28)	14	19	20	20	20	16	18	**22**
Tetris (18)	**7**	4	3	4	4	2	2	3
Thoughtful (20)	6	10	10	8	10	9	9	**12**
Tidybot (17)	3	11	11	**12**	**12**	11	11	**12**
TPP (30)	10	16	13	20	19	15	18	**24**
Transport (53)	7	9	10	15	15	**19**	**19**	17
Trucks (31)	18	30	30	30	29	**31**	**31**	30
VisitAll (37)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Woodworking (31)	18	**24**	11	19	19	21	19	22
Zenotravel (20)	8	14	13	13	13	**16**	**16**	14
Sum (2147)	1098	1344	1361	1425	1461	1438	1490	**1550**
Normalized (%)	44.8	58.5	55.9	59.9	59.8	61.6	61.9	**66.1**
Expansions	1961	1135	1787	401	491	**365**	485	369
Search time (s)	0.53	0.40	0.57	0.19	0.23	**0.18**	0.24	**0.18**

TABLE 9.2: Coverage on the IPC satisficing instances. The last two rows show the geometric means of the expansions and search time on commonly solved instances.

9.2.4 Summary

XES performs consistently well across a wide range of domains, and is clearly superior to previous state-of-the-art algorithms overall. The distributionally-enhanced BEES variants BEES95 and BEEPS95 also tend to yield small improvements over BEES and BEEPS in most domains. One advantage of XES over these BEES variants is that it is less reliant on the probability estimate being accurate: If the probability estimate consistently over- or underestimates, then the expected effort values are affected equally and the ordering of the open list does not change much, while the focal list is either over- or underutilized in the BEES algorithms. PTS and $\widehat{\text{PTS}}$ perform poorly on most domains; furthermore, they have worse scaling for larger bounds as they do not use a distance estimate, and thus have less effective guidance towards a goal state than the other algorithms.

9.3 Exploiting Heuristic Uncertainty in Bounded-Suboptimal Search

In this section, we explore how the idea behind XES can be transferred to the bounded-suboptimal search setting. We first introduce a straightforward adaptation, and then discuss a more sophisticated variant that explicitly considers increasing the bound in favor of goal-oriented exploration. Additionally, we introduce a simple round-robin scheme that can be instantiated with different priority functions. We finally evaluate our novel algorithms on the IPC domains.

9.3.1 Exploiting Expected Effort

Moving from a static cost bound C to the dynamic suboptimality bound $w \cdot f_{min}$ necessitates a focal search approach, where focal contains only search nodes with $f(n) \leq w \cdot f_{min}$, ordered by the expected effort, and open is ordered by f to track f_{min}. Following XES's paradigm of explicitly taking uncertainty into account, we also model the uncertainty of the suboptimality bound. We propose two different approaches to adapt XES: a relatively simple focal-search-like variant called Dynamic Expected Effort Search (DXES), and a more complex adaptation called Considerate Dynamic Expected Effort Search (CDXES).

9.3.1.1 Dynamic Expected Effort Search

As described in Section 9.2.1.2, XES uses a single normal distribution $\mathcal{B}_{bound}$ centered on $\hat{f}$ to estimate the solution cost under a search node n, and thus the probability that it is

within the cost bound:

$$\mathcal{B}_{cost}(n) \sim \mathcal{N}(\hat{f}(n), (\frac{\hat{f}(n)-f(n)}{2})^2).$$

In DXES, we also use a second distribution $\mathcal{B}_{bound}$ to describe our current belief about the cost bound given by the suboptimality factor w, of which our best guess is $w \cdot \hat{f}_{min}$. The value $\hat{f}_{min}$ may change after each expansion—either due to changes in the average heuristic error (when deriving $\hat{h}$ from h with online error correction) or due to nodes being added to or removed from the open list. We record the $\hat{f}_{min}$ value after each expansion in a collection δ, and use its variance to obtain a belief distribution of the bound:

$$\mathcal{B}_{bound} \sim \mathcal{N}(w \cdot \hat{f}_{min}, var(\delta)).$$

The probability that a node n leads to a solution within the suboptimality bound can now be expressed as the probability that a sample from $\mathcal{B}_{cost}(n)$ is not greater than a sample from $\mathcal{B}_{bound}$. This probability $P(\mathcal{B}_{cost}(n) \leq \mathcal{B}_{bound})$ is equivalent to $P(\mathcal{B}_{bound} - \mathcal{B}_{cost}(n) \geq 0)$, and we can construct the distribution $\mathcal{B}(n) = \mathcal{B}_{bound} - \mathcal{B}_{cost}(n)$ by subtracting the means and adding the variances:

$$\mathcal{B}(n) \sim \mathcal{N}(w \cdot \hat{f}_{min} - \hat{f}(n), (\frac{\hat{f}(n)-f(n)}{2})^2 + var(\delta)).$$

From this distribution, we can compute the probability mass that is greater or equal to zero to obtain the probability that n leads to a solution within the bound (similar to XES).

Note that this approach requires fast access to three search nodes at each expansion, namely the ones with minimal expected effort, f value, and $\hat{f}$ value respectively. Accordingly, our implementation uses three queues even though DXES always expands the node $best_{xe}$ with minimal expected effort xe_{min}, which will always exist as the queue must at least contain $best_f$.

9.3.1.2 Considerate Dynamic Expected Effort Search

DXES focuses on finding a solution within the current known lower bound, however, it may be useful to raise the bound as well: expand $best_f$ until f_{min} raises sufficiently such that more promising nodes (that are currently outside $w \cdot f_{min}$) become available in the focal list. We now introduce Considerate Dynamic Expected Effort Search (CDXES), which carefully considers the expected effort required to raise f_{min} through successive expansions of $best_f$ in order to make a node more promising than $best_{xe}$ available to focal. For example, consider a node n that is not in focal with $xe(n) = 10$ expansions when

$xe(best_{xe}) = 20$. If raising f_{min} sufficiently to include n in focal takes fewer than 10 expansions, then it would be worth doing that so we can expand n afterwards and still expect less total search effort.

We estimate the effort to raise f_{min} to a desired f value $f_\circ$ (i.e., the number of required $best_f$ expansions) by

$$T_{f_\circ} = \sum_{f=f_{min}}^{f_\circ - 1} \#open_f \cdot \frac{f_\circ - f}{\epsilon_h},$$

where $\#open_f$ is the number of open nodes with the given f value, and ϵ_h is the mean one-step error in h [Thayer et al., 2011]. The one-step heuristic error gives us an idea of how much the f value increases on average for each expansion. For example, if $\epsilon_h = 0.2$, we can expect f to increase by one after five expansions. $T_{f_\circ}$ takes into account both the amount by which f_{min} must increase to $f_\circ$ and the number of open nodes with each intermediate f value.

In order to express the expected effort in expansions as well, we change the remaining time estimator in $xe(n)$ from $T(n) = \hat{d}(n)$ to $T(n) = \hat{d}(n) \cdot delay$, where $delay$ is the average expansion delay [Dionne et al., 2011]. Now we can consider whether it would be beneficial to expand $best_f$ in order to raise f_{min}: If there is a node n that is currently not in the focal list that has $xe(n) + T_{f(n)} < xe_{min}$, then CDXES expands $best_f$ instead of the usual $best_{xe}$.

9.3.2 A Simple Round-Robin Scheme

The main motivation behind CDXES as an improvement over DXES is the observation that progress in bounded-suboptimal search can mean different things—finding nodes closer to the goal, but also gathering more information about the cost bound. The cost bound can be estimated by $w \cdot \hat{f}_{min}$, and a lower bound is given by $w \cdot f_{min}$. Most algorithms discussed before use at least one of these values in their search strategy, but not all of them aim to gain more knowledge about them (by expanding the nodes $best_{\hat{f}}$ or $best_f$ respectively). EES and CDXES do attempt to estimate when raising the bound is useful through careful metareasoning. However, even if the inference rules are well founded, this metareasoning is based on online heuristic information which itself may be quite unreliable. Hence, we also investigate a simpler alternative.

We study a basic round-robin scheme using three queues: a focal list that can be instantiated with any evaluation function, an open list sorted by $\hat{f}$, and a cleanup list sorted by f. In order to guarantee that solutions are within the given suboptimality bound, the first two queues only contain nodes with $f(n) \leq w \cdot f_{min}$, while the cleanup list contains

all open nodes. The search then simply alternates between these queues, expanding the node at the front of the current queue at each expansion.

We explore three instantiations for the focal list ordering: d, ud, and xe, resembling the main search strategies of EES, DPS, and DXES respectively. In particular, note that EES also considers expanding the node with either minimal distance, f, or $\hat{f}$, but follows a more sophisticated selection strategy between those queues.

One drawback of the round-robin scheme is that it does not converge to speedy search (best-first search on d) with increasing suboptimality bounds. This can be a useful property to have since it should lead to a solution the fastest in case the solution quality does not matter [Wilt and Ruml, 2014]. Both EES and DXES satisfy that property: in both search algorithms, eventually all nodes are in the focal list, and in DXES all probability estimates become one with sufficiently large weights. DPS on the other hand does not converge to speedy search (and does not use a distance estimate at all in its open list ordering function).

9.3.3 Experimental Evaluation

We compare the search algorithms using all unique STRIPS instances from the optimal tracks of the International Planning Competitions, for a total of 1652 instances from 48 domains. We use the admissible landmark-cut heuristic [Helmert and Domshlak, 2009] as both heuristic and distance estimator (using unit action costs for the latter). As before, the estimates $\hat{h}$ and $\hat{d}$ are obtained by correcting for online observations of the one-step error [Thayer et al., 2011]; we initialize the error with 100 virtual samples of zero.

9.3.3.1 Exploiting Expected Effort

Figure 9.3 shows an overview of the results for various suboptimality bounds ranging from $w = 1$ to $w = 2$. Somewhat surprisingly, the success of XES in bounded-cost search does not directly transfer to the bounded-suboptimal setting as its adaptation DXES performs poorly here. A major reason is that DXES neglects to raise the suboptimality bound: when looking at its search behavior in comparison to algorithms such as EES, we found that f_{min} increases much more slowly throughout the search in DXES. This was the main motivation behind CDXES and the round-robin strategies, which explicitly aim to raise f_{min} by expanding $best_f$. Yet, as Figure 9.3 shows, CDXES performs even worse than DXES. By periodically expanding $best_f$, the probability estimates may significantly change due to fluctuations in f_{min} (and $\hat{f}_{min}$), leading to many nodes with outdated values in the open list. As an attempt to ameliorate this, we tested a variant of CDXES that lazily reevaluates

Coverage	WA*	EES	DPS	DXES	CDXES↻	RR-*d*	RR-DPS	RR-DXES
Airport (50)	31	30	**33**	29	32	**33**	32	**33**
Blocksworld (35)	33	31	34	28	31	34	32	**35**
DataNetwork (20)	14	14	14	14	14	**16**	13	**16**
Depot (22)	9	10	10	8	9	**11**	8	**11**
DriverLog (20)	**15**	**15**	**15**	14	**15**	**15**	**15**	**15**
Elevators (30)	23	21	23	21	22	**28**	23	**28**
Floortile (40)	22	17	**23**	12	16	21	21	22
Freecell (80)	15	16	15	15	17	17	15	**19**
GED (20)	15	15	**19**	15	15	15	**19**	15
Grid (5)	2	2	2	2	2	**3**	2	**3**
Gripper (20)	**20**	15	**20**	8	9	9	16	18
Hiking (20)	10	10	10	9	10	**12**	10	11
Logistics (63)	**48**	42	**48**	43	42	41	45	44
Mprime (35)	22	22	**23**	22	21	22	**23**	**23**
Mystery (19)	**17**	**17**	**17**	15	15	**17**	**17**	**17**
Nomystery (20)	**20**	**20**	**20**	14	17	**20**	**20**	**20**
Openstacks (80)	36	34	36	33	33	37	31	**38**
OrgSynth-split (20)	**16**	**16**	**16**	15	**16**	**16**	**16**	**16**
Parcprinter (30)	**30**	**30**	**30**	**30**	**30**	**30**	28	**30**
Parking (40)	18	18	20	7	13	22	17	**23**
Pathways (30)	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**
Pegsol (36)	35	33	35	33	34	**36**	35	**36**
Pipes-notank (50)	23	24	23	17	24	26	24	**27**
Pipes-tank (50)	13	13	13	14	14	17	13	**18**
PNetAlignment (20)	11	11	11	8	9	**13**	10	**13**
PSR (50)	49	49	**50**	49	**50**	**50**	**50**	**50**
Rovers (40)	14	12	13	11	15	14	14	**18**
Satellite (36)	**14**	13	**14**	11	13	13	**14**	12
Scanalyzer (28)	17	12	**20**	12	15	16	17	18
Snake (20)	7	7	7	6	**8**	7	7	7
Sokoban (30)	**29**	28	**29**	28	28	**29**	**29**	**29**
Spider (20)	**12**	11	**12**	11	11	**12**	11	**12**
Storage (30)	16	17	16	15	17	**18**	16	17
Termes (20)	7	**11**	9	6	6	10	7	10
Tetris (17)	7	7	**8**	6	6	**8**	7	**8**
Tidybot (30)	21	21	21	19	22	23	20	**26**
TPP (30)	7	8	8	8	**9**	**9**	7	8
Transport (59)	18	19	18	18	20	22	18	**23**
Trucks (30)	**21**	18	18	14	18	20	**21**	19
VisitAll (33)	21	20	21	18	21	**23**	21	**23**
Woodworking (30)	26	27	26	26	28	29	27	**30**
Zenotravel (20)	14	14	**15**	13	13	14	14	14
Others (274)	**191**	**191**	**191**	**191**	**191**	**191**	**191**	**191**
Sum (1652)	995	967	1012	894	957	1025	982	**1052**
Normalized (%)	58.7	57.0	60.0	51.5	55.6	60.7	57.9	**62.5**
Expansions	569	558	472	734	511	383	665	**371**
Search time (s)	0.65	0.91	**0.55**	1.09	0.83	0.65	1.05	0.65

TABLE 9.3: Coverage over the IPC instances for $w = 1.5$.

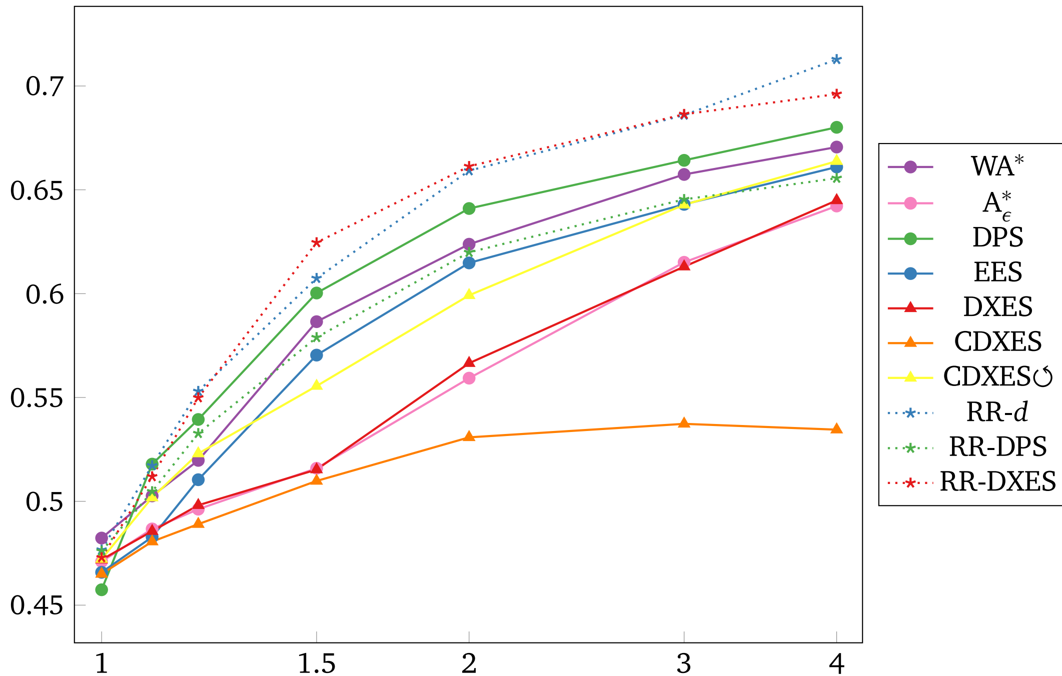


FIGURE 9.3: Normalized coverage over increasing suboptimality bounds.

nodes whose *xe* values change between insertion and expansion: the CDXES↺ configuration requeues nodes if their updated *xe* estimate differs by more than 5%. This significantly improves results, though it still does not reach the overall performance of WA* and DPS. We tested this approach also for DXES and RR-DXES, but found that the results of the former remain almost unchanged, and performance degrades for the latter. Similarly, a bucket-based reordering strategy akin to DPS also did not improve results for DXES.

Table 9.3 shows the per-domain coverage for selected algorithms with a suboptimality bound of $w = 1.5$. CDXES↺ has its merit in a few domains: it has strictly higher coverage than the other algorithms in Spider, and has equal (or better) coverage with fewer expansions in Grid, Miconic, Parcprinter, and TPP on commonly solved instances. Overall though, the XES variants for bounded-suboptimal do not carry over the success from bounded-cost search and are outperformed by the baselines in most domains. It appears that, even though it is possible to port bounded-cost algorithms to the bounded-suboptimal setting [Gilon et al., 2016], it is not trivial to achieve high performance.

9.3.3.2 Round-Robin Algorithms

Consider again Figure 9.3 and Table 9.3. The round-robin variants RR-*d* and RR-DXES perform best overall, significantly outperforming the other algorithms across most tested suboptimality bounds. Interestingly, the potential variant of the round-robin algorithms,

RR-DPS, does not share the success of RR-d and RR-DXES, and is generally outperformed by its base version DPS. For example, for $w = 1.5$ (see Table 9.3), RR-DPS improves coverage over DPS in 4 domains, but lowers it in 18. In contrast, RR-DXES demonstrates extremely robust performance. In terms of coverage, it is a strict improvement over DXES, boosting coverage in 38 (!) domains while never falling below DXES for $w = 1.5$; and almost halving the number of expansions on average. Excluding domains where all algorithms have the same coverage, RR-DXES is the (at least shared) best algorithm on 28 out of 42 domains. As the suboptimality bound increases, RR-d catches up and surpasses RR-DXES at $w = 3$.

In order to test whether periodically expanding both $best_f$ and $best_{\hat{f}}$ is important, we also tested configurations that alternate the focal queue only with either $best_f$ or $best_{\hat{f}}$, and found them to be inferior to the variants presented here. We conclude that both $best_f$ and $best_{\hat{f}}$ are useful components for bounded-suboptimal search.

9.4 Conclusion

In bounded-cost search, the goal is to find a solution within a user-provided cost bound as fast as possible. We have introduced a relatively simple algorithm called XES that explicitly follows that objective. Its evaluation function guides the search based on the expected search effort to find a feasible solution. This is computed using two estimates: the effort required to find a solution, and the probability that the cost of such a solution is within the bound. We have shown that this search strategy is correct in a simplified formal model. While our proof relies on assumptions that are unrealistic in practice, XES turns out to be extremely effective: On the IPC benchmarks, XES consistently yields strong results across a wide range of domains, substantially outperforming other state-of-the-art bounded-cost search algorithms.

Our experiments with DXES and CDXES show that an adaptation to bounded-suboptimal search is possible, but achieving the same kind of success as in the bounded-cost setting remains difficult. In contrast, a simple round-robin scheme can yield state-of-the-art performance with either d or xe for guidance. Its advantage over sophisticated metareasoning approaches shows that there is still some room for improvement for more principled methods, with potentially better trade-offs between aiming to reach a goal quickly and gaining knowledge about the suboptimality bound.

Overall, our work advances the recent trend of leveraging distributional information to boost performance in a deterministic search setting [e.g., O'Ceallaigh and Ruml, 2015;

Mitchell et al., 2019], highlighting the benefit of explicitly modeling the uncertainty of the heuristic estimates.

Part III

Conclusion

10 Conclusion

This work challenges the traditional view of heuristic search as a static process in which the heuristic is treated as a black box. The algorithms introduced in this work follow a more dynamic approach, allowing them to make adjustments to the search strategy online based on observations such as the behavior of the heuristic.

In the first part of this thesis, we explored adaptive partial delete relaxation methods for AI planning. We introduced search algorithms that identify weaknesses in the heuristic, and apply targeted refinement operations on the underlying relaxation. Our algorithms based on local search use a converging heuristic function as an additional form of making progress and guaranteeing completeness. Combined with the partial delete relaxation heuristic h^{CFF}, these methods are exceptionally effective in practice. Our best-performing variants beat even recent portfolio planners across a wide range of benchmarks, and bring local search back to the state of the art in satisficing planning.

We explored tie breaking strategies in the refinement procedure of h^{CFF}. Our extensive evaluation closes a gap of prior research, and results in a better understanding of the performance of existing strategies. We achieved surprisingly good results with a simple periodic replacement strategy in greedy best-first search, which can potentially boost performance of online-refinement search algorithms by reducing their overhead.

For partial delete relaxation with red-black planning, we have shown how to combine the converging (but computationally expensive) red-black state-space search with the tractable fragment. We further devised a flexible approach that refines the relaxation locally where needed. While our methods are not competitive to the state of the art overall, they yield improvements in some domains, and our ideas can potentially inspire other adaptive refinement approaches.

In the second part, we investigated more general heuristic search techniques that are not focused on AI planning. We introduced a principled approach to select the initial state for online replanning by reasoning about the planning time. As shown by our experiments,

this results in superior generality and robustness compared to previous methods such as deriving the initial state of the new plan from an offline estimate of the planning time.

We proved that the recently introduced real-time search algorithm Nancy is complete. Our completeness proof applies not only to Nancy, but to a general class of real-time search algorithms that can be used to simplify completeness proofs for similar algorithms. As a side effect, our analysis shows that the popular LSS-LRTA* is complete under fewer assumptions than its original proof.

Finally, we proposed novel algorithms for suboptimal search based on explicit estimations of the probability of finding a solution within a given cost bound. Our bounded-cost search algorithm XES consistently outperforms the current state of the art on the IPC benchmarks. While its success does not straightforwardly carry over to bounded-suboptimal search, we have shown how these ideas can be used in a simple round-robin scheme to achieve state-of-the-art performance.

Given the success of online relaxation refinement in satisficing planning, a natural avenue for future work would be to try similar approaches in other settings, such as optimal planning. Initial results of online refinement of Cartesian abstractions [Eifler et al., 2019] have been promising, but not yet competitive with state-of-the-art planners. However, more could be tried, e.g., using other types of abstractions [Helmert et al., 2014; Rovner et al., 2019], testing different approaches for when and how much to refine, or combining heuristic refinement with orthogonal techniques such as online refinement of cost partitionings [Seipp, 2021].

The search algorithms discussed in Chapters 8 and 9 make use of belief distributions instead of heuristic functions that provide scalar values. However, the distributions are still derived from a standard heuristic. One interesting direction for future work would be the combination with methods that naturally yield distributions. One example are heuristics based on neural networks [Ferber et al., 2020], whose output can be interpreted as a distribution over possible heuristic values.

Bibliography

[Alcázar and Torralba, 2015] Vidal Alcázar and Álvaro Torralba. "A Reminder about the Importance of Computing and Exploiting Invariants in Planning". In: *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015*. Ed. by Ronen I. Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein. AAAI Press, 2015, pp. 2–6.

[Allis, 1994] Louis Victor Allis. "Searching for Solutions in Games and Artificial Intelligence". PhD thesis. Maastricht University, 1994.

[Arfaee et al., 2011] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C. Holte. "Learning heuristic functions for large state spaces". In: *Artif. Intell.* 175.16-17 (2011), pp. 2075–2098.

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

[Barto et al., 1995] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. "Learning to Act Using Real-Time Dynamic Programming". In: *Artif. Intell.* 72.1-2 (1995), pp. 81–138.

[Benton et al., 2007] J. Benton, Minh B. Do, and Wheeler Ruml. "A Simple Testbed for On-line Planning". In: *Proceedings of the ICAPS-07 Workshop on Planning and Plan Execution for Real-World Systems*. 2007.

[Bertsekas and Tsitsiklis, 1996] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-dynamic programming*. Vol. 3. Optimization and neural computation series. Athena Scientific, 1996.

[Bonet and Geffner, 2001] Blai Bonet and Hector Geffner. "Planning as heuristic search". In: *Artif. Intell.* 129.1-2 (2001), pp. 5–33.

[Bonet and Geffner, 2003] Blai Bonet and Hector Geffner. "Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming". In: *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003), June 9-13, 2003, Trento, Italy*. Ed. by Enrico Giunchiglia, Nicola Muscettola, and Dana S. Nau. AAAI, 2003, pp. 12–21.

[Brafman and Domshlak, 2003] Ronen I. Brafman and Carmel Domshlak. "Structure and Complexity in Planning with Unary Operators". In: *J. Artif. Intell. Res.* 18 (2003), pp. 315–349.

[Bulitko and Sampley, 2016] Vadim Bulitko and Alexander Sampley. "Weighted Lateral Learning in Real-Time Heuristic Search". In: *Proceedings of the Ninth Annual Symposium on Combinatorial Search, SOCS 2016, Tarrytown, NY, USA, July 6-8, 2016*. Ed. by Jorge A. Baier and Adi Botea. AAAI Press, 2016, pp. 10–18.

[Bulitko et al., 2008] Vadim Bulitko, Mitja Lustrek, Jonathan Schaeffer, Yngvi Björnsson, and Sverrir Sigmundarson. "Dynamic Control in Real-Time Heuristic Search". In: *J. Artif. Intell. Res.* 32 (2008), pp. 419–452.

[Bulitko et al., 2010] Vadim Bulitko, Yngvi Björnsson, and Ramon Lawrence. "Case-Based Subgoaling in Real-Time Heuristic Search for Video Game Pathfinding". In: *J. Artif. Intell. Res.* 39 (2010), pp. 269–300.

[Burns et al., 2012] Ethan Burns, J. Benton, Wheeler Ruml, Sung Wook Yoon, and Minh Binh Do. "Anticipatory On-Line Planning". In: *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*. Ed. by Lee McCluskey, Brian Charles Williams, José Reinaldo Silva, and Blai Bonet. AAAI, 2012.

[Burns et al., 2013] Ethan Burns, Wheeler Ruml, and Minh Binh Do. "Heuristic Search When Time Matters". In: *J. Artif. Intell. Res.* 47 (2013), pp. 697–740.

[Bylander, 1994] Tom Bylander. "The Computational Complexity of Propositional STRIPS Planning". In: *Artif. Intell.* 69.1-2 (1994), pp. 165–204.

[Campbell et al., 2002] Murray Campbell, A. Joseph Hoane Jr., and Feng-Hsiung Hsu. "Deep Blue". In: *Artif. Intell.* 134.1-2 (2002), pp. 57–83.

[Cashmore et al., 2018] Michael Cashmore, Andrew Coles, Bence Cserna, Erez Karpas, Daniele Magazzeni, and Wheeler Ruml. "Temporal Planning while the Clock Ticks". In: *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*. Ed. by Mathijs de Weerdt, Sven Koenig, Gabriele Röger, and Matthijs T. J. Spaan. AAAI Press, 2018, pp. 39–46.

[Cashmore et al., 2019] Michael Cashmore, Andrew Coles, Bence Cserna, Erez Karpas, Daniele Magazzeni, and Wheeler Ruml. "Replanning for Situated Robots". In: *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2019, Berkeley, CA, USA, July 11-15, 2019*. Ed. by J. Benton, Nir Lipovetzky, Eva Onaindia, David E. Smith, and Siddharth Srivastava. AAAI Press, 2019, pp. 665–673.

[Cenamor et al., 2016] Isabel Cenamor, Tomás de la Rosa, and Fernando Fernández. "The IBaCoP Planning System: Instance-Based Configured Portfolios". In: *J. Artif. Intell. Res.* 56 (2016), pp. 657–691.

[Chinchalkar, 1996] Shirish Chinchalkar. "An Upper Bound for the Number of Reachable Positions". In: *J. Int. Comput. Games Assoc.* 19.3 (1996), pp. 181–183.

[Clarke et al., 1994] Edmund M. Clarke, Orna Grumberg, and David E. Long. "Model Checking and Abstraction". In: *ACM Trans. Program. Lang. Syst.* 16.5 (1994), pp. 1512–1542.

[Clarke et al., 2003] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement for symbolic model checking". In: *J. ACM* 50.5 (2003), pp. 752–794.

[Culberson and Schaeffer, 1996] Joseph C. Culberson and Jonathan Schaeffer. "Searching with Pattern Databases". In: *Advances in Artificial Intelligence, 11th Biennial Conference of the Canadian Society for Computational Studies of Intelligence, AI '96, Toronto, Ontario, Canada, May 21-24, 1996, Proceedings*. Ed. by Gordon I. McCalla. Vol. 1081. Lecture Notes in Computer Science. Springer, 1996, pp. 402–416.

[Culberson and Schaeffer, 1998] Joseph C. Culberson and Jonathan Schaeffer. "Pattern Databases". In: *Comput. Intell.* 14.3 (1998), pp. 318–334.

[Daum et al., 2016] Jeanette Daum, Álvaro Torralba, Jörg Hoffmann, Patrik Haslum, and Ingo Weber. "Practical Undoability Checking via Contingent Planning". In: *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016*. Ed. by Amanda Jane Coles, Andrew Coles, Stefan Edelkamp, Daniele Magazzeni, and Scott Sanner. AAAI Press, 2016, pp. 106–114.

[Dionne et al., 2011] Austin J. Dionne, Jordan Tyler Thayer, and Wheeler Ruml. "Deadline-Aware Search Using On-Line Measures of Behavior". In: *Proceedings of the Fourth Annual Symposium on Combinatorial Search, SOCS 2011, Castell de Cardona, Barcelona, Spain, July 15.16, 2011*. Ed. by Daniel Borrajo, Maxim Likhachev, and Carlos Linares López. AAAI Press, 2011.

[Dobson and Haslum, 2017] Sean Dobson and Patrik Haslum. "Cost-Length Tradeoff Heuristics for Bounded-Cost Search". In: *Proceedings of the ICAPS-17 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP 2017)*. 2017.

[Domshlak et al., 2012] Carmel Domshlak, Erez Karpas, and Shaul Markovitch. "Online Speedup Learning for Optimal Planning". In: *J. Artif. Intell. Res.* 44 (2012), pp. 709–755.

[Domshlak et al., 2015] Carmel Domshlak, Jörg Hoffmann, and Michael Katz. "Red-black planning: A new systematic approach to partial delete relaxation". In: *Artif. Intell.* 221 (2015), pp. 73–114.

[Doran and Michie, 1966] James E. Doran and Donald Michie. "Experiments with the graph traverser program". In: *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 294.1437 (1966), pp. 235–259.

[Edelkamp, 2001] Stefan Edelkamp. "Planning with Pattern Databases". In: *Proceedings of the Sixth European Conference on Planning, September 12-14, 2001, Toledo, Spain*. Ed. by Amedeo Cesta and Daniel Borrajo. AAAI Press, 2001, pp. 13–24.

[Eifler and Fickert, 2018] Rebecca Eifler and Maximilian Fickert. “Online Refinement of Cartesian Abstraction Heuristics”. In: *Proceedings of the Eleventh International Symposium on Combinatorial Search, SOCS 2018, Stockholm, Sweden - 14-15 July 2018*. Ed. by Vadim Bulitko and Sabine Storandt. AAAI Press, 2018, pp. 46–54.

[Eifler et al., 2019] Rebecca Eifler, Maximilian Fickert, Jörg Hoffmann, and Wheeler Ruml. “Refining Abstraction Heuristics during Real-Time Planning”. In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 7578–7585.

[Felner et al., 2004] Ariel Felner, Richard E. Korf, and Sarit Hanan. “Additive Pattern Database Heuristics”. In: *J. Artif. Intell. Res.* 22 (2004), pp. 279–318.

[Ferber et al., 2020] Patrick Ferber, Malte Helmert, and Jörg Hoffmann. “Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space”. In: *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*. Ed. by Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang. Vol. 325. Frontiers in Artificial Intelligence and Applications. IOS Press, 2020, pp. 2346–2353.

[Ferguson et al., 2008] Dave Ferguson, Thomas M. Howard, and Maxim Likhachev. “Motion planning in urban environments”. In: *J. Field Robotics* 25.11-12 (2008), pp. 939–960.

[Fickert, 2016] Maximilian Fickert. “A Study on Online Generation of Explicit Conjunctions for Partial Delete Relaxation Heuristics”. Master’s Thesis. Saarland University, 2016.

[Fickert, 2018] Maximilian Fickert. “Making Hill-Climbing Great Again through Online Relaxation Refinement and Novelty Pruning”. In: *Proceedings of the Eleventh International Symposium on Combinatorial Search, SOCS 2018, Stockholm, Sweden - 14-15 July 2018*. Ed. by Vadim Bulitko and Sabine Storandt. AAAI Press, 2018, pp. 158–162.

[Fickert, 2020] Maximilian Fickert. “A Novel Lookahead Strategy for Delete Relaxation Heuristics in Greedy Best-First Search”. In: *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*. Ed. by J. Christopher Beck, Olivier Buffet, Jörg Hoffmann, Erez Karpas, and Shirin Sohrabi. AAAI Press, 2020, pp. 119–123.

[Fickert and Hoffmann, 2017a] Maximilian Fickert and Jörg Hoffmann. “Complete Local Search: Boosting Hill-Climbing through Online Relaxation Refinement”. In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*. Ed. by Laura

Barbulescu, Jeremy Frank, Mausam, and Stephen F. Smith. AAAI Press, 2017, pp. 107–115.

[Fickert and Hoffmann, 2017b] Maximilian Fickert and Jörg Hoffmann. "Ranking Conjunctions for Partial Delete Relaxation Heuristics in Planning". In: *Proceedings of the Tenth International Symposium on Combinatorial Search, SOCS 2017, 16-17 June 2017, Pittsburgh, Pennsylvania, USA*. Ed. by Alex Fukunaga and Akihiro Kishimoto. AAAI Press, 2017, pp. 38–46.

[Fickert and Hoffmann, 2022] Maximilian Fickert and Jörg Hoffmann. "Online Relaxation Refinement for Satisficing Planning: On Partial Delete Relaxation, Complete Hill-Climbing, and Novelty Pruning". In: *J. Artif. Intell. Res.* 73 (2022), pp. 67–115.

[Fickert et al., 2016] Maximilian Fickert, Jörg Hoffmann, and Marcel Steinmetz. "Combining the Delete Relaxation with Critical-Path Heuristics: A Direct Characterization". In: *J. Artif. Intell. Res.* 56 (2016), pp. 269–327.

[Fickert et al., 2018a] Maximilian Fickert, Daniel Gnad, Patrick Speicher, and Jörg Hoffmann. "SaarPlan: Combining Saarland's Greatest Planning Techniques". In: *IPC 2018 planner abstracts*. 2018, pp. 11–16.

[Fickert et al., 2018b] Maximilian Fickert, Daniel Gnad, and Jörg Hoffmann. "Unchaining the Power of Partial Delete Relaxation, Part II: Finding Plans with Red-Black State Space Search". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. Ed. by Jérôme Lang. ijcai.org, 2018, pp. 4750–4756.

[Fickert et al., 2020] Maximilian Fickert, Tianyi Gu, Leonhard Staut, Wheeler Ruml, Jörg Hoffmann, and Marek Petrik. "Beliefs We Can Believe in: Replacing Assumptions with Data in Real-Time Search". In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 9827–9834.

[Fickert et al., 2021a] Maximilian Fickert, Tianyi Gu, and Wheeler Ruml. "Bounded-cost Search Using Estimates of Uncertainty". In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*. Ed. by Zhi-Hua Zhou. ijcai.org, 2021, pp. 1675–1681.

[Fickert et al., 2021b] Maximilian Fickert, Ivan Gavran, Ivan Fedotov, Jörg Hoffmann, Rupak Majumdar, and Wheeler Ruml. "Choosing the Initial State for Online Replanning". In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 2021, pp. 12311–12319.

[Fickert et al., 2022] Maximilian Fickert, Tianyi Gu, and Wheeler Ruml. “New Results in Bounded-Suboptimal Search”. In: *The Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Vancouver, BC, Canada, February 22 - March 1, 2022*. AAAI Press, 2022. Accepted.

[Fink, 2007] Michael Fink. “Online Learning of Search Heuristics”. In: *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics, AISTATS 2007, San Juan, Puerto Rico, March 21-24, 2007*. Ed. by Marina Meila and Xiaotong Shen. Vol. 2. JMLR Proceedings. JMLR.org, 2007, pp. 114–122.

[Fox et al., 2006] Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina. “Plan Stability: Replanning versus Plan Repair”. In: *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*. Ed. by Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey. AAAI, 2006, pp. 212–221.

[Francès et al., 2018] Guillem Francès, Nir Lipovetzky, Hector Geffner, and Miquel Ramírez. “Best-First Width Search in the IPC 2018: Complete, Simulated, and Polynomial Variants”. In: *IPC 2018 planner abstracts*. 2018, pp. 23–27.

[Gerevini et al., 2003] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. “Planning Through Stochastic Local Search and Temporal Action Graphs in LPG”. In: *J. Artif. Intell. Res.* 20 (2003), pp. 239–290.

[Ghallab et al., 2016] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

[Gilon et al., 2016] Daniel Gilon, Ariel Felner, and Roni Stern. “Dynamic Potential Search - A New Bounded Suboptimal Search”. In: *Proceedings of the Ninth Annual Symposium on Combinatorial Search, SOCS 2016, Tarrytown, NY, USA, July 6-8, 2016*. Ed. by Jorge A. Baier and Adi Botea. AAAI Press, 2016, pp. 36–44.

[Gilon et al., 2017] Daniel Gilon, Ariel Felner, and Roni Stern. “Dynamic Potential Search on Weighted Graphs”. In: *Proceedings of the Tenth International Symposium on Combinatorial Search, SOCS 2017, 16-17 June 2017, Pittsburgh, Pennsylvania, USA*. Ed. by Alex Fukunaga and Akihiro Kishimoto. AAAI Press, 2017, pp. 119–123.

[Gnad and Hoffmann, 2015] Daniel Gnad and Jörg Hoffmann. “Red-Black Planning: A New Tractability Analysis and Heuristic Function”. In: *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11-13 June 2015, Ein Gedi, the Dead Sea, Israel*. Ed. by Levi Lelis and Roni Stern. AAAI Press, 2015, pp. 44–52.

[Gnad and Hoffmann, 2018] Daniel Gnad and Jörg Hoffmann. “Star-topology decoupled state space search”. In: *Artif. Intell.* 257 (2018), pp. 24–60.

[Gnad et al., 2016] Daniel Gnad, Marcel Steinmetz, Mathäus Jany, Jörg Hoffmann, Ivan Serina, and Alfonso Gerevini. “Partial Delete Relaxation, Unchained: On Intractable

Red-Black Planning and Its Applications". In: *Proceedings of the Ninth Annual Symposium on Combinatorial Search, SOCS 2016, Tarrytown, NY, USA, July 6-8, 2016*. Ed. by Jorge A. Baier and Adi Botea. AAAI Press, 2016, pp. 45–53.

[Gross et al., 2020] Joschka Gross, Álvaro Torralba, and Maximilian Fickert. "Novel Is Not Always Better: On the Relation between Novelty and Dominance Pruning". In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 9875–9882.

[Hald, 1952] Anders Hald. *Statistical Theory with Engineering Applications*. Probability and Statistics Series. Wiley, 1952.

[Hart et al., 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Trans. Syst. Sci. Cybern.* 4.2 (1968), pp. 100–107.

[Haslum, 2006] Patrik Haslum. "Improving Heuristics Through Relaxed Search - An Analysis of TP4 and HSP*a in the 2004 Planning Competition". In: *J. Artif. Intell. Res.* 25 (2006), pp. 233–267.

[Haslum, 2012] Patrik Haslum. "Incremental Lower Bounds for Additive Cost Planning Problems". In: *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*. Ed. by Lee McCluskey, Brian Charles Williams, José Reinaldo Silva, and Blai Bonet. AAAI, 2012.

[Haslum, 2013] Patrik Haslum. "Heuristics for Bounded-Cost Search". In: *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*. Ed. by Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini. AAAI, 2013.

[Haslum and Geffner, 2000] Patrik Haslum and Hector Geffner. "Admissible Heuristics for Optimal Planning". In: *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO, USA, April 14-17, 2000*. Ed. by Steve A. Chien, Subbarao Kambhampati, and Craig A. Knoblock. AAAI, 2000, pp. 140–149.

[Haslum et al., 2019] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019.

[Heckerman et al., 1995] David Heckerman, John S. Breese, and Koos Rommelse. "Decision-Theoretic Troubleshooting". In: *Commun. ACM* 38.3 (1995), pp. 49–57.

[Helmert, 2006] Malte Helmert. "The Fast Downward Planning System". In: *J. Artif. Intell. Res.* 26 (2006), pp. 191–246.

[Helmert, 2009] Malte Helmert. "Concise finite-domain representations for PDDL planning tasks". In: *Artif. Intell.* 173.5-6 (2009), pp. 503–535.

[Helmert and Domshlak, 2009] 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, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*. Ed. by Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis. AAAI, 2009.

[Helmert and Röger, 2008] Malte Helmert and Gabriele Röger. "How Good is Almost Perfect?" In: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*. Ed. by Dieter Fox and Carla P. Gomes. AAAI Press, 2008, pp. 944–949.

[Helmert et al., 2007] Malte Helmert, Patrik Haslum, and Jörg Hoffmann. "Flexible Abstraction Heuristics for Optimal Sequential Planning". In: *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*. Ed. by Mark S. Boddy, Maria Fox, and Sylvie Thiébaux. AAAI, 2007, pp. 176–183.

[Helmert et al., 2011] Malte Helmert, Gabriele Röger, and Erez Karpas. "Fast Downward Stone Soup: A baseline for building planner portfolios". In: *Proceedings of the ICAPS-11 Workshop on Planning and Learning (PAL 2011)*. 2011.

[Helmert et al., 2014] Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. "Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces". In: *J. ACM* 61.3 (2014), 16:1–16:63.

[Hoffmann, 2005] Jörg Hoffmann. "Where 'Ignoring Delete Lists' Works: Local Search Topology in Planning Benchmarks". In: *J. Artif. Intell. Res.* 24 (2005), pp. 685–758.

[Hoffmann, 2015] Jörg Hoffmann. "Simulated Penetration Testing: From "Dijkstra" to "Turing Test++"". In: *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015*. Ed. by Ronen I. Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein. AAAI Press, 2015, pp. 364–372.

[Hoffmann and Fickert, 2015] Jörg Hoffmann and Maximilian Fickert. "Explicit Conjunctions without Compilation: Computing $h^{\text{FF}}(\Pi^C)$ in Polynomial Time". In: *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015*. Ed. by Ronen I. Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein. AAAI Press, 2015, pp. 115–119.

[Hoffmann and Nebel, 2001] Jörg Hoffmann and Bernhard Nebel. "The FF Planning System: Fast Plan Generation Through Heuristic Search". In: *J. Artif. Intell. Res.* 14 (2001), pp. 253–302.

[Kalagnanam and Henrion, 1988] Jayant Kalagnanam and Max Henrion. “A Comparison of Decision Analysis and Expert Rules for Sequential Diagnosis”. In: *UAI ’88: Proceedings of the Fourth Annual Conference on Uncertainty in Artificial Intelligence, Minneapolis, MN, USA, July 10-12, 1988*. Ed. by Ross D. Shachter, Tod S. Levitt, Laveen N. Kanal, and John F. Lemmer. North-Holland, 1988, pp. 271–282.

[Karpas et al., 2011] Erez Karpas, Michael Katz, and Shaul Markovitch. “When Optimal Is Just Not Good Enough: Learning Fast Informative Action Cost Partitionings”. In: *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011*. Ed. by Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert. AAAI, 2011.

[Katz and Domshlak, 2010] Michael Katz and Carmel Domshlak. “Optimal admissible composition of abstraction heuristics”. In: *Artif. Intell.* 174.12-13 (2010), pp. 767–798.

[Katz and Hoffmann, 2013] Michael Katz and Jörg Hoffmann. “Red-Black Relaxed Plan Heuristics Reloaded”. In: *Proceedings of the Sixth Annual Symposium on Combinatorial Search, SOCS 2013, Leavenworth, Washington, USA, July 11-13, 2013*. Ed. by Malte Helmert and Gabriele Röger. AAAI Press, 2013.

[Katz and Hoffmann, 2014] Michael Katz and Jörg Hoffmann. “Mercury Planner: Pushing the Limits of Partial Delete Relaxation”. In: *IPC 2014 planner abstracts*. 2014, pp. 43–47.

[Katz et al., 2013a] Michael Katz, Jörg Hoffmann, and Carmel Domshlak. “Red-Black Relaxed Plan Heuristics”. In: *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. Ed. by Marie desJardins and Michael L. Littman. AAAI Press, 2013.

[Katz et al., 2013b] Michael Katz, Jörg Hoffmann, and Carmel Domshlak. “Who Said We Need to Relax All Variables?” In: *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*. Ed. by Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini. AAAI, 2013.

[Katz et al., 2017] Michael Katz, Nir Lipovetzky, Dany Moshkovich, and Alexander Tuisov. “Adapting Novelty to Classical Planning as Heuristic Search”. In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*. Ed. by Laura Barbulescu, Jeremy Frank, Mausam, and Stephen F. Smith. AAAI Press, 2017, pp. 172–180.

[Katz et al., 2018] Michael Katz, Nir Lipovetzky, Dany Moshkovich, and Alexander Tuisov. “MERWIN Planner: Mercury Enchanced With Novelty Heuristic”. In: *IPC 2018 planner abstracts*. 2018, pp. 53–56.

[Keyder et al., 2012] Emil Ragip Keyder, Jörg Hoffmann, and Patrik Haslum. “Semi-Relaxed Plan Heuristics”. In: *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June*

25-19, 2012. Ed. by Lee McCluskey, Brian Charles Williams, José Reinaldo Silva, and Blai Bonet. AAAI, 2012.

[Keyder et al., 2014] Emil Ragip Keyder, Jörg Hoffmann, and Patrik Haslum. "Improving Delete Relaxation Heuristics Through Explicitly Represented Conjunctions". In: *J. Artif. Intell. Res.* 50 (2014), pp. 487–533.

[Kiesel et al., 2015] Scott Kiesel, Ethan Burns, and Wheeler Ruml. "Achieving Goals Quickly Using Real-time Search: Experimental Results in Video Games". In: *J. Artif. Intell. Res.* 54 (2015), pp. 123–158.

[Knight et al., 2001] Russell Knight, Gregg Rabideau, Steve A. Chien, Barbara Engelhardt, and Rob Sherwood. "Casper: Space Exploration through Continuous Planning". In: *IEEE Intell. Syst.* 16.5 (2001), pp. 70–75.

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

[Koehler and Ottiger, 2002] Jana Koehler and Daniel Ottiger. "An AI-Based Approach to Destination Control in Elevators". In: *AI Mag.* 23.3 (2002), pp. 59–78.

[Koenig and Sun, 2009] Sven Koenig and Xiaoxun Sun. "Comparing real-time and incremental heuristic search for real-time situated agents". In: *Auton. Agents Multi Agent Syst.* 18.3 (2009), pp. 313–341.

[Korf, 1990] Richard E. Korf. "Real-Time Heuristic Search". In: *Artif. Intell.* 42.2-3 (1990), pp. 189–211.

[Korf, 1997] Richard E. Korf. "Finding Optimal Solutions to Rubik's Cube Using Pattern Databases". In: *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA*. Ed. by Benjamin Kuipers and Bonnie L. Webber. AAAI Press / The MIT Press, 1997, pp. 700–705.

[Korf and Taylor, 1996] Richard E. Korf and Larry A. Taylor. "Finding Optimal Solutions to the Twenty-Four Puzzle". In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 2*. Ed. by William J. Clancey and Daniel S. Weld. AAAI Press / The MIT Press, 1996, pp. 1202–1207.

[Lawrence and Bulitko, 2013] Ramon Lawrence and Vadim Bulitko. "Database-Driven Real-Time Heuristic Search in Video-Game Pathfinding". In: *IEEE Trans. Comput. Intell. AI Games* 5.3 (2013), pp. 227–241.

[Lemons et al., 2010] Seth Lemons, J. Benton, Wheeler Ruml, Minh Binh Do, and Sung Wook Yoon. "Continual On-line Planning as Decision-Theoretic Incremental Heuristic Search". In: *Embedded Reasoning, Papers from the 2010 AAAI Spring Symposium, Technical Report SS-10-04, Stanford, California, USA, March 22-24, 2010*. AAAI, 2010.

[Likhachev et al., 2003] Maxim Likhachev, Geoffrey J. Gordon, and Sebastian Thrun. "ARA*: Anytime A* with Provable Bounds on Sub-Optimality". In: *Advances in Neural*

Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]. Ed. by Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf. MIT Press, 2003, pp. 767–774.

[Likhachev et al., 2005] Maxim Likhachev, David I. Ferguson, Geoffrey J. Gordon, Anthony Stentz, and Sebastian Thrun. "Anytime Dynamic A*: An Anytime, Replanning Algorithm". In: *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5-10 2005, Monterey, California, USA*. Ed. by Susanne Biundo, Karen L. Myers, and Kanna Rajan. AAAI, 2005, pp. 262–271.

[Lipovetzky and Geffner, 2012] Nir Lipovetzky and Hector Geffner. "Width and Serialization of Classical Planning Problems". In: *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31, 2012*. Ed. by Luc De Raedt, Christian Bessiere, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter J. F. Lucas. Vol. 242. Frontiers in Artificial Intelligence and Applications. IOS Press, 2012, pp. 540–545.

[Lipovetzky and Geffner, 2014] Nir Lipovetzky and Hector Geffner. "Width-based Algorithms for Classical Planning: New Results". In: *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*. Ed. by Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan. Vol. 263. Frontiers in Artificial Intelligence and Applications. IOS Press, 2014, pp. 1059–1060.

[Lipovetzky and Geffner, 2017] Nir Lipovetzky and Hector Geffner. "Best-First Width Search: Exploration and Exploitation in Classical Planning". In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. Ed. by Satinder P. Singh and Shaul Markovitch. AAAI Press, 2017, pp. 3590–3596.

[Ma et al., 2017] Hang Ma, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. "Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks". In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*. Ed. by Kate Larson, Michael Winikoff, Sanmay Das, and Edmund H. Durfee. ACM, 2017, pp. 837–845.

[Ma et al., 2019] Hang Ma, Wolfgang Hönig, T. K. Satish Kumar, Nora Ayanian, and Sven Koenig. "Lifelong Path Planning with Kinematic Constraints for Multi-Agent Pickup and Delivery". In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 7651–7658.

[McGann et al., 2007] Conor McGann, Frederic Py, Kanna Rajan, Hans Thomas, Rrichard Henthorn, and Rob McEwen. “T-REX: A Model-Based Architecture for AUV Control”. In: *Proceedings of the ICAPS-07 Workshop on Planning and Plan Execution for Real-World Systems*. 2007.

[Mitchell et al., 2019] Andrew Mitchell, Wheeler Ruml, Fabian Spaniol, Jörg Hoffmann, and Marek Petrik. “Real-Time Planning as Decision-Making under Uncertainty”. In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 2338–2345.

[Muise, 2016] Christian Muise. “Planning.Domains”. In: *The 26th International Conference on Automated Planning and Scheduling - Demonstrations*. 2016.

[Muscettola et al., 2002] Nicola Muscettola, Gregory A. Dorais, Chuck Fry, Richard Levinson, and Christian Plaunt. “IDEA: Planning at the Core of Autonomous Reactive Agents”. In: *Proceedings of the AIPS-02 Workshop on On-line Planning and Scheduling*. 2002, pp. 49–55.

[Mutchler, 1986] David Mutchler. “Optimal Allocation of Very Limited Search Resources”. In: *Proceedings of the 5th National Conference on Artificial Intelligence. Philadelphia, PA, USA, August 11-15, 1986. Volume 1: Science*. Ed. by Tom Kehler. Morgan Kaufmann, 1986, pp. 467–471.

[Nakhost and Müller, 2009] Hootan Nakhost and Martin Müller. “Monte-Carlo Exploration for Deterministic Planning”. In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*. Ed. by Craig Boutilier. 2009, pp. 1766–1771.

[O’Ceallaigh and Ruml, 2015] Dylan O’Ceallaigh and Wheeler Ruml. “Metareasoning in Real-Time Heuristic Search”. In: *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11-13 June 2015, Ein Gedi, the Dead Sea, Israel*. Ed. by Levi Lelis and Roni Stern. AAAI Press, 2015, pp. 87–95.

[Pearl and Kim, 1982] Judea Pearl and Jin H. Kim. “Studies in Semi-Admissible Heuristics”. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 4.4 (1982), pp. 392–399.

[Pemberton, 1995] Joseph C. Pemberton. “k-Best: A New Method for Real-Time Decision Making”. In: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*. Morgan Kaufmann, 1995, pp. 227–235.

[Pemberton and Korf, 1994] Joseph C. Pemberton and Richard E. Korf. “Incremental Search Algorithms for Real-time Decision Making”. In: *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems, University of Chicago, Chicago, Illinois, USA, June 13-15, 1994*. Ed. by Kristian J. Hammond. AAAI, 1994, pp. 140–145.

[Pohl, 1970] Ira Pohl. "Heuristic Search Viewed as Path Finding in a Graph". In: *Artif. Intell.* 1.3 (1970), pp. 193–204.

[Richter and Helmert, 2009] Silvia Richter and Malte Helmert. "Preferred Operators and Deferred Evaluation in Satisficing Planning". In: *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*. Ed. by Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis. AAAI, 2009.

[Richter and Westphal, 2010] Silvia Richter and Matthias Westphal. "The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks". In: *J. Artif. Intell. Res.* 39 (2010), pp. 127–177.

[Richter et al., 2008] Silvia Richter, Malte Helmert, and Matthias Westphal. "Landmarks Revisited". In: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*. Ed. by Dieter Fox and Carla P. Gomes. AAAI Press, 2008, pp. 975–982.

[Rovner et al., 2019] Alexander Rovner, Silvan Sievers, and Malte Helmert. "Counterexample-Guided Abstraction Refinement for Pattern Selection in Optimal Classical Planning". In: *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019*. Ed. by J. Benton, Nir Lipovetzky, Eva Onaindia, David E. Smith, and Siddharth Srivastava. AAAI Press, 2019, pp. 362–367.

[Ruml et al., 2011] Wheeler Ruml, Minh Binh Do, Rong Zhou, and Markus P. J. Fromherz. "On-line Planning and Scheduling: An Application to Controlling Modular Printers". In: *J. Artif. Intell. Res.* 40 (2011), pp. 415–468.

[Seipp, 2021] Jendrik Seipp. "Online Saturated Cost Partitioning for Classical Planning". In: *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021*. Ed. by Susanne Biundo, Minh Do, Robert Goldman, Michael Katz, Qiang Yang, and Hankz Hankui Zhuo. AAAI Press, 2021, pp. 317–321.

[Seipp and Helmert, 2013] Jendrik Seipp and Malte Helmert. "Counterexample-Guided Cartesian Abstraction Refinement". In: *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*. Ed. by Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini. AAAI, 2013.

[Seipp and Helmert, 2014] Jendrik Seipp and Malte Helmert. "Diverse and Additive Cartesian Abstraction Heuristics". In: *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*. Ed. by Steve A. Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml. AAAI, 2014.

[Seipp and Helmert, 2018] Jendrik Seipp and Malte Helmert. “Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning”. In: *J. Artif. Intell. Res.* 62 (2018), pp. 535–577.

[Seipp and Röger, 2018] Jendrik Seipp and Gabriele Röger. “Fast Downward Stone Soup 2018”. In: *IPC 2018 planner abstracts*. 2018, pp. 80–82.

[Seipp et al., 2017] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. *Downward Lab*. https://doi.org/10.5281/zenodo.790461. 2017.

[Seipp et al., 2020] Jendrik Seipp, Thomas Keller, and Malte Helmert. “Saturated Cost Partitioning for Optimal Classical Planning”. In: *J. Artif. Intell. Res.* 67 (2020), pp. 129–167.

[Shperberg et al., 2019] Shahaf S. Shperberg, Andrew Coles, Bence Cserna, Erez Karpas, Wheeler Ruml, and Solomon Eyal Shimony. “Allocating Planning Effort When Actions Expire”. In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 2371–2378.

[Sievers et al., 2019] Silvan Sievers, Michael Katz, Shirin Sohrabi, Horst Samulowitz, and Patrick Ferber. “Deep Learning for Cost-Optimal Planning: Task-Dependent Planner Selection”. In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 7715–7723.

[Silver et al., 2016] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nat.* 529.7587 (2016), pp. 484–489.

[Silver et al., 2017] David Silver et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. In: *CoRR* abs/1712.01815 (2017).

[Speicher et al., 2017] Patrick Speicher, Marcel Steinmetz, Daniel Gnad, Jörg Hoffmann, and Alfonso Gerevini. “Beyond Red-Black Planning: Limited-Memory State Variables”. In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*. Ed. by Laura Barbulescu, Jeremy Frank, Mausam, and Stephen F. Smith. AAAI Press, 2017, pp. 269–273.

[Steinmetz and Hoffmann, 2017a] Marcel Steinmetz and Jörg Hoffmann. “Critical-Path Dead-End Detection versus NoGoods: Offline Equivalence and Online Learning”. In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*. Ed. by Laura

Barbulescu, Jeremy Frank, Mausam, and Stephen F. Smith. AAAI Press, 2017, pp. 283–287.

[Steinmetz and Hoffmann, 2017b] Marcel Steinmetz and Jörg Hoffmann. "State space search nogood learning: Online refinement of critical-path dead-end detectors in planning". In: *Artif. Intell.* 245 (2017), pp. 1–37.

[Steinmetz and Hoffmann, 2018] Marcel Steinmetz and Jörg Hoffmann. "LP Heuristics over Conjunctions: Compilation, Convergence, Nogood Learning". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. Ed. by Jérôme Lang. ijcai.org, 2018, pp. 4837–4843.

[Stentz and Hebert, 1995] Anthony Stentz and Martial Hebert. "A complete navigation system for goal acquisition in unknown environments". In: *Auton. Robots* 2.2 (1995), pp. 127–145.

[Stern et al., 2014] Roni Stern, Ariel Felner, Jur van den Berg, Rami Puzis, Rajat Shah, and Ken Goldberg. "Potential-based bounded-cost search and Anytime Non-Parametric A^*". In: *Artif. Intell.* 214 (2014), pp. 1–25.

[Stern et al., 2011] Roni Tzvi Stern, Rami Puzis, and Ariel Felner. "Potential Search: A Bounded-Cost Search Algorithm". In: *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011*. Ed. by Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert. AAAI, 2011.

[Thayer and Ruml, 2009] Jordan Tyler Thayer and Wheeler Ruml. "Using Distance Estimates in Heuristic Search". In: *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*. Ed. by Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis. AAAI, 2009.

[Thayer et al., 2011] Jordan Tyler Thayer, Austin J. Dionne, and Wheeler Ruml. "Learning Inadmissible Heuristics During Search". In: *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011*. Ed. by Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert. AAAI, 2011.

[Thayer et al., 2012] Jordan Tyler Thayer, Roni Stern, Ariel Felner, and Wheeler Ruml. "Faster Bounded-Cost Search Using Inadmissible Estimates". In: *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*. Ed. by Lee McCluskey, Brian Charles Williams, José Reinaldo Silva, and Blai Bonet. AAAI, 2012.

[Torralba et al., 2021] Álvaro Torralba, Jendrik Seipp, and Silvan Sievers. "Automatic Instance Generation for Classical Planning". In: *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou,*

China (virtual), August 2-13, 2021. Ed. by Susanne Biundo, Minh Do, Robert Goldman, Michael Katz, Qiang Yang, and Hankz Hankui Zhuo. AAAI Press, 2021, pp. 376–384.

[Vidal, 2004] Vincent Vidal. “A Lookahead Strategy for Heuristic Search Planning”. In: *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*. Ed. by Shlomo Zilberstein, Jana Koehler, and Sven Koenig. AAAI, 2004, pp. 150–160.

[Vidal, 2011] Vincent Vidal. “YAHSP2: Keep It Simple, Stupid”. In: *IPC 2011 planner abstracts*. 2011, pp. 83–90.

[Wilt and Ruml, 2013] Christopher Makoto Wilt and Wheeler Ruml. “Robust Bidirectional Search via Heuristic Improvement”. In: *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. Ed. by Marie desJardins and Michael L. Littman. AAAI Press, 2013.

[Wilt and Ruml, 2014] Christopher Makoto Wilt and Wheeler Ruml. “Speedy Versus Greedy Search”. In: *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014*. Ed. by Stefan Edelkamp and Roman Barták. AAAI Press, 2014.

[Wilt and Ruml, 2016] Christopher Makoto Wilt and Wheeler Ruml. “Effective Heuristics for Suboptimal Best-First Search”. In: *J. Artif. Intell. Res.* 57 (2016), pp. 273–306.

[Xie et al., 2014] Fan Xie, Martin Müller, and Robert Holte. “Adding Local Exploration to Greedy Best-First Search in Satisficing Planning”. In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*. Ed. by Carla E. Brodley and Peter Stone. AAAI Press, 2014, pp. 2388–2394.