Journal of Artificial Intelligence Research 73 (2022) 67–115 Submitted 06/2021; published 01/2022

Online Relaxation Refinement for Satisficing Planning: On Partial Delete Relaxation, Complete Hill-Climbing, and Novelty Pruning

Maximilian Fickert FICKERT@CS.UNI-SAARLAND.DE
Jörg Hoffmann HOFFMANN@CS.UNI-SAARLAND.DE
Saarland University, Saarland Informatics Campus
Saarbrücken, Germany

Abstract

In classical AI planning, heuristic functions typically base their estimates on a relaxation of the input task. Such relaxations can be more or less precise, and many heuristic functions have a refinement procedure that can be iteratively applied until the desired degree of precision is reached. Traditionally, such refinement is performed offline to instantiate the heuristic for the search. However, a natural idea is to perform such refinement *online* instead, in situations where the heuristic is not sufficiently accurate. We introduce several online-refinement search algorithms, based on hill-climbing and greedy best-first search. Our hill-climbing algorithms perform a bounded lookahead, proceeding to a state with lower heuristic value than the root state of the lookahead if such a state exists, or refining the heuristic otherwise to remove such a local minimum from the search space surface. These algorithms are complete if the refinement procedure satisfies a suitable convergence property. We transfer the idea of bounded lookaheads to greedy best-first search with a lightweight lookahead after each expansion, serving both as a method to boost search progress and to detect when the heuristic is inaccurate, identifying an opportunity for online refinement. We evaluate our algorithms with the partial delete relaxation heuristic h^{CFF}, which can be refined by treating additional conjunctions of facts as atomic, and whose refinement operation satisfies the convergence property required for completeness. On both the IPC domains as well as on the recently published Autoscale benchmarks, our online-refinement search algorithms significantly beat state-of-the-art satisficing planners, and are competitive even with complex portfolios.

1. Introduction

AI planning is a long-standing subfield of AI, concerned with general problem solving mechanisms that decide about the actions taken by an agent (for an overview, see Ghallab, Nau, & Traverso, 2004). This work is set in so-called classical planning where the planning problem can be cast as finding a path from an initial state to a goal state (a plan) in a large state space. The state space is compactly described in terms of a set of Boolean state variables, with actions that have preconditions and effects over these variables. We focus on satisficing planning, where the objective is to find a good plan as quickly as possible (as opposed to scenarios where a guarantee on plan quality must be given).

One of the most prominent approaches to satisficing classical planning is heuristic state-space search (e.g., Bonet & Geffner, 2001; Hoffmann & Nebel, 2001; Helmert, 2006; Richter & Westphal, 2010), where a heuristic guides the search with estimates of the remaining distance to a goal state. Such heuristics are domain-independent, that is, they use only the

input task description as a basis to compute their heuristic values, without any human input other than that description. Such heuristics are typically based on solving a relaxation (a simplified version) of the task.

Most planning heuristics can compute their estimates at different levels of precision: *Abstraction heuristics* (e.g., Clarke, Grumberg, & Long, 1994; Culberson & Schaeffer, 1998; Edelkamp, 2001; Helmert, Haslum, & Hoffmann, 2007; Helmert, Haslum, Hoffmann, & Nissim, 2014; Seipp & 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 & Geffner, 2000; Haslum, 2006; Fickert, Hoffmann, & Steinmetz, 2016) compute their estimates based on the most costly subgoals toward the goal, where considering larger subgoals results in a more accurate heuristic. *Partial delete relaxation heuristics* (Keyder, Hoffmann, & Haslum, 2014; Domshlak, Hoffmann, & Katz, 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, Grumberg, Jha, Lu, & Veith, 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 & Helmert, 2013, 2018), pattern database heuristics (Rovner, Sievers, & Helmert, 2019), and the partial delete relaxation heuristic h^{CFF} 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 & 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 & 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, Korf, & Hanan, 2004; Fink, 2007; Katz & Domshlak, 2010; Karpas, Katz, & Markovitch, 2011; Domshlak, Karpas, & Markovitch, 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 greedy best-first search (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 work, 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 & 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 toward refinement, while larger bounds give the search more time to find a better state. We devise a more effective approach leveraging novelty pruning (Lipovetzky & 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 & 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 toward 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, Seipp, & Sievers, 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 & Westphal, 2010), MERWIN (Katz, Lipovetzky, Moshkovich, & Tuisov, 2018), and Dual-BFWS (Francès, Lipovetzky, Geffner, & Ramírez, 2018; Lipovetzky & Geffner, 2017), and are competitive even with complex state-of-the-art portfolios. On the Autoscale benchmarks the advantage increases further, for example, beating the portfolio planner and winner of the IPC'18 satisficing track Fast Downward Stone Soup (Seipp & Röger, 2018; Helmert, Röger, & Karpas, 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.

The paper is structured as follows. First, we introduce the general planning formalisms and the common setup for the experiments throughout this paper (Section 2). We next give a brief summary on the existing techniques that we use in our algorithms, including h^{CFF} and novelty pruning (Section 3). In Section 4, 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 5, and show how to extend it with novelty pruning (Section 6) and subgoal counting (Section 7). In Section 8, 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 9. Finally, we give a more detailed discussion of related work (Section 10) before concluding the paper in Section 11.

2. Preliminaries

We first introduce the basic planning formalisms and notation. Since we interleave our contributions with experimental evaluations focusing on the algorithms introduced in the respective sections, we also discuss the common setup for the experiments here.

2.1 Planning Framework

We use the STRIPS framework for classical AI planning (Fikes & Nilsson, 1971), where each state is a set of Boolean facts. Formally, a STRIPS task is a 4-tuple $\Pi = \langle \mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, where

- $\mathcal{F}$ is a set of propositional *facts*,
- $\mathcal{A}$ is a set of *actions*, where each action $a \in \mathcal{A}$ is a triple of the fact sets $pre(a)$, $add(a)$, $del(a)$ (*preconditions*, *add effects*, and *delete effects*),
- $\mathcal{I} \subseteq \mathcal{F}$ is the *initial state*, and
- $\mathcal{G} \subseteq \mathcal{F}$ is the *goal*.

A *state* $s \subseteq \mathcal{F}$ is a set of facts. An action a is *applicable* in s if $pre(a) \subseteq s$, and applying it in s results in the state $s[\![a]\!] := (s \setminus del(a)) \cup add(a)$. A *plan* for s is a sequence of successively applicable actions leading from s to a goal state s_G with $\mathcal{G} \subseteq s_G$, and a plan for $\mathcal{I}$ is a plan for Π. A plan for s is called *optimal* if it is the shortest (we consider unit action costs in this work) among all plans for s.

The set of all states is denoted by $\mathcal{S}$. A *heuristic function*, short *heuristic*, is a function $h : \mathcal{S} \mapsto \mathbb{N}_0 \cup \{\infty\}$ estimating the length of a plan for the given state, or evaluating to ∞ to indicate that the state is a *dead end* (a state for which no plan exists). We assume that heuristics are *safe*, that is, if $h(s) = \infty$ then s is indeed a dead end, which is the case for most if not all planning heuristics. The *perfect heuristic* h^* maps each state s to the length of an optimal plan for s, or to ∞ if s is a dead end.

Most 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*) $H(s)$. For heuristics based on a relaxation, $H(s)$ is typically the subset of actions in $\pi[h](s)$ that are 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.

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. One source of incompleteness is helpful actions pruning, as $H(s)$ may exclude the only actions that start a plan for s.

2.2 Experiments Setup

Our implementation is based on Fast Downward (Helmert, 2006). The code and raw experiment data is available on GitHub.[1] 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 8) 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 9.3).

The experiments were run on a cluster of machines with Intel E5-2660 processors at 2.2GHz using the Downward Lab framework (Seipp, Pommerening, Sievers, & Helmert, 2017). All experiments use a timeout of 30 minutes and a memory limit of 4 GB.

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

1. https://github.com/fickert/fast-downward-conjunctions

3. Background: Techniques We Build On

Our online-refinement search algorithms leverage novelty pruning and subgoal counting for the local exploration component, and we use h^{CFF} as the heuristic of our choice in the experiments. We summarize these techniques in the following.

3.1 Novelty Pruning and Subgoal Counting

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 $t \not\subseteq s'$ 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 & 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 & Geffner, 2017) or lower (Katz, Lipovetzky, Moshkovich, & Tuisov, 2017) heuristic value. Best-First Width Search (BFWS) (Lipovetzky & Geffner, 2017) is a best-first search which uses a novelty measure as the main evaluation function to guide the search. 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 & 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 work, 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 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).

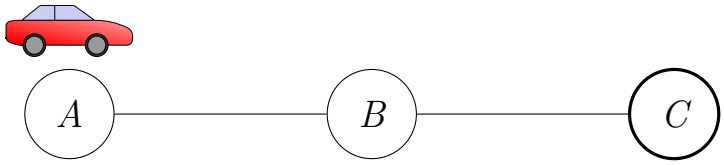
3.2 The $h^{C\mathrm{FF}}$ Heuristic and its Refinement Operation

Our online-refinement search algorithms are generally independent of the used heuristic, given that it offers a refinement operation with certain convergence properties. 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 give a brief summary of $h^{C\mathrm{FF}}$ and its CEGAR-based refinement operation here; further details can be found in Fickert et al.'s (2016) and Keyder et al.'s (2014) work respectively.

3.2.1 The $h^{C\mathrm{FF}}$ Heuristic

The *delete relaxation*, which assumes the delete lists of all actions to be empty, is one of the most popular relaxations for heuristics used in satisficing planning. 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**-hard, so in practice the approximative h^{FF} heuristic (Hoffmann & Nebel, 2001) is used instead, which bases its estimate on (not necessarily optimal) delete-relaxed plans $\pi[h^{\mathrm{FF}}]$ (or returns ∞ in the same case as h^+).

Example 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 fact "fuel" indicating whether the car currently has fuel, and a fact "at(x)" indicating the position of the car for each location $x \in \{A, B, C\}$. The "refuel" action has no preconditions or delete effects, and its only add effect is the fuel fact. The "drive(x, y)" actions take the car from location x to location y, and have preconditions $\{at(x), fuel\}$, add effects $\{at(y)\}$, and delete effects $\{at(x), fuel\}$.

One plan for this task is $\langle drive(A, B), refuel, drive(B, C)\rangle$. A delete-relaxed plan, for example $\langle drive(A, B), drive(B, C)\rangle$, does not need to include the refuel action, since the drive actions do not delete the fuel fact under the relaxation.

The concept of *partial delete relaxation* aims to improve the accuracy of delete relaxation heuristics by taking *some* delete information into account. One such technique 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, Hoffmann, & Haslum, 2012; Keyder et al., 2014; Hoffmann & 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 a achieves some part of the conjunction ($add(a) \cap c \neq \emptyset$) and does not delete another ($del(a) \cap c = \emptyset$), and the remaining facts of c that are not added by a ($c \setminus add(a)$) are treated as additional preconditions.

Example 2 *Consider again the task shown in Example 1. The critical issue of the relaxed plan in Example 1 is that the preconditions of the drive(B, C) action are not satisfied under normal semantics, as the fuel fact is assumed to still be available after driving to B. Consider the conjunction $c = \{fuel, at(B)\}$. 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 remove part of c (the fuel fact). 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.*

Throughout this paper, we assume that C always consists of at least all singleton facts, that is, $C \supseteq C_0$ where $C_0 := \{\{f\} \mid f \in \mathcal{F}\}$. In the experiments, we will sometimes discuss the increase in computational complexity of $h^{C\text{FF}}$ as conjunctions are added to C. In practice, $h^{C\text{FF}}$ is implemented through counters that keep track of the number of unsatisfied preconditions that are required to reach a conjunction through a given action (Hoffmann & Fickert, 2015; Fickert et al., 2016), similar to the h^{FF} implementation, which tracks the number of unsatisfied preconditions for each action (Hoffmann & Nebel, 2001). We approximate the increase in computational complexity of $h^{C\text{FF}}$ by the number of counters that were added by (non-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 (the number of counters that would need to be tracked by h^{FF}).

3.2.2 The Refinement Operation of $h^{C\text{FF}}$

The set of conjunctions C controls the degree of the relaxation for the corresponding heuristic. 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, that is, $C = \mathcal{P}(\mathcal{F})$, then every C-relaxed plan is also a plan under non-relaxed semantics, and $h^{C+} = h^*$.

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. The known methods iteratively generate C via counterexample-guided abstraction refinement. Each refinement step works as follows: Let s be a state where $h^{C\text{FF}}(s) \neq \infty$, and let $\pi[h^{C\text{FF}}](s)$ be the corresponding partially relaxed plan. Either $\pi[h^{C\text{FF}}](s)$ is a real plan for s, or there must be a conflict in the form of a deleted 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 in the previous subsection, the deletion of the fuel fact by the *drive(A, B)* action formed a conflict, and the conjunction $\{fuel, at(B)\}$ could be added to address it.

The original refinement algorithm by Haslum (2012) guarantees that $\pi[h^{C\text{FF}}](s)$ is no longer a valid C-relaxed plans 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 $h^{C\text{FF}}$ to compute a different plan, and it induces significantly less computational overhead. Hence, we use Keyder et al.'s method here, with one minor change as suggested by Fickert and Hoffmann (2017b): Instead of considering conflicts occurring in *any* of the valid orderings of the partially relaxed plan in the refinement process, we only consider those that actually appear in the sequentialized plan returned by the heuristic.

4. 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 $h^{C\text{FF}}$ 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 $h^{C\text{FF}}$, and then give a more general definition.

As pointed out in Section 3.2.2, the h^{C+} heuristic satisfies the strong convergence to h^*. The $h^{C\text{FF}}$ 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 $h^{C\text{FF}}$ 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 $h^{C\text{FF}}$ heuristic is converged and we can prove the aforementioned property:

Proposition 1 *Let $\Pi = (\mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G})$ be a planning task, and let s be a state. Then there exists a set of conjunctions C such that (i) in case s is unsolvable, we have $h^{C\text{FF}}(s) = \infty$; and (ii) in case s is solvable, $\pi[h^{C\text{FF}}](s)$ is a plan for s. In particular, both (i) and (ii) hold for $C = C_*$.*

Proof: We prove (i) and (ii) for $C = C_* = \mathcal{P}(\mathcal{F})$.

For (i): By Fickert et al.'s (2016) Corollary 1, $h^{C\text{FF}}(s) = \infty$ iff $h^{C+}(s) = \infty$, and there exists C s.t. $h^{C+}(s) = h^*(s)$. As $h^{C_*+} \geq h^{C+}$ for any C, this shows the claim.

For (ii): As s is solvable, $h^*(s) \neq \infty$, so $h^{C_*+}(s) \neq \infty$ and $h^{C_*\text{FF}}(s) \neq \infty$. Thus we can run C-refinement on s. Assume that $\pi[h^{C\text{FF}}](s)$ 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 $h^{C\text{FF}}$ adds a conjunction to C, eventually the heuristic converges as C grows toward C_*. Proposition 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 1 (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 $s \in \mathcal{S}$, (i) if $h^*(s) = \infty$, then $\rho^N[h](s) = \infty$, and (ii) otherwise $\pi[\rho^N[h]](s)$ is a plan for s.*

As discussed, h^{CFF} 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^{CFF}. 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 & Helmert, 2014, 2018; Seipp, Keller, & Helmert, 2020), yet convergence is only guaranteed if abstractions are merged (which is expensive) or only a single abstraction is used (Eifler & Fickert, 2018). In principle though, any such heuristic that converges according to Definition 1 is sufficient to guarantee completeness of our hill-climbing algorithms introduced in the following.

5. 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 & 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*.

5.1 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 toward 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
6          // mark s as a dead end and start a new episode
7          C_de := C_de ∪ {s}
8          s := I
9      else
10         s := s'
11         if s ⊇ G then
12             return SOLVED
13 return UNSOLVABLE
```

Algorithm 1 shows the pseudo-code 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 5–8). 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 2 *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, 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 $\mathcal{C}_{de}$.

5.2 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 5.3). Hence, the discussion below assumes that the lookahead may use helpful actions pruning.

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 bound then
6              // s is likely a dead end
7              HANDLE_EXHAUSTION
8          if the previous lookahead iteration also originated at s and s ≠ I then
9              // previous refinement was unsuccessful
10             HANDLE_STAGNATION
11         // raise h(s) to resolve the local minimum
12         Let h_min be the minimal h value observed in the current lookahead
13         while h(s) ≤ h_min do
14             REFINE_HEURISTIC
15             if h(s) = ∞ then
16                 HANDLE_DEAD_END
17                 break
18     else
19         s := s'
20         if s ⊇ G then
21             return SOLVED
22 return UNSOLVABLE
23 macro REFINE_HEURISTIC
24     if π[h](s) is a plan for s then
25         return SOLVED
26     refine h on s, replacing h by ρ[h]
```

The pseudo-code 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 18). However, if the lookahead does not yield such a state, then the heuristic is refined in s (Lines 11 to 17). 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 heuristic is poor (deep local minima), leaving more shallow local minima to be escaped via search.

The Refinement-HC pseudo-code contains several *macros* (typeset in UPPERCASE), where any control-flow statements like **break**, **continue**, or **return** inside a macro refer to the position where the macro is inserted (in contrast to subprocedures). The REFINE_HEURISTIC macro applies one refinement step to the heuristic (Line 23). 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.

Algorithm 3: Backjump

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

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 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 $\lambda(s') = \mathit{true}$, which it returns (or $\mathcal{I}$ if no state along the path satisfies λ).

First, the HANDLE_EXHAUSTION macro is called in case the search space is exhausted without reaching the depth bound (Algorithm 2, Line 7). 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

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
```

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; hence, 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 10), 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 backchaining.

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 16). 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).

5.3 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 1, then the search must eventually reach a goal or refine the heuristic to convergence, making Refinement-HC complete.

Theorem 1 *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 1 (i) we have $h(s) = \infty$ for all unsolvable s (including $\mathcal{I}$), leading to termination on (b) (Algorithm 2, Line 22).

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 (by Definition 1 (ii)) $\pi[h](s)$ is a plan for all solvable s. 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 16). In that episode, the search will eventually reach a goal state (Line 21) or call REFINE_HEURISTIC, where it terminates on (a) since $\pi[h](s)$ is a plan for s (Line 25). ■

Note that the completeness proof holds for *all 18 possible instantiations* of the HANDLE_* macros, and thus for 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 (i) ($h(\mathcal{I}) = \infty$). 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 state is reached. Our weaker convergence property (ii) 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.

5.4 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 6.3.2).

Figure 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 9.1 discusses this in more detail.

Figure 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

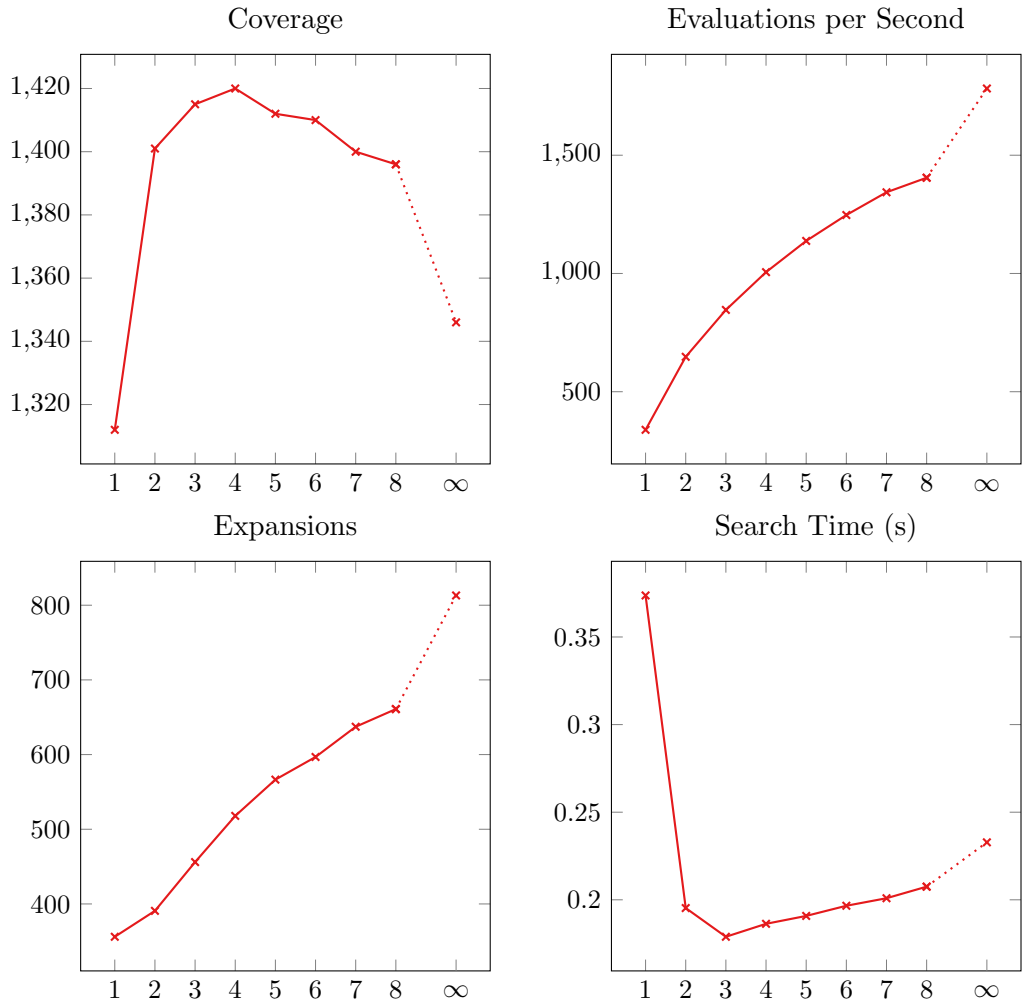


Figure 1: Results for Refinement-HC with varying depth bounds (x-axis): total coverage, geometric mean of the heuristic evaluations per second, and geometric means across commonly solved instances of the number of expansions and search time.

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.

6. 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

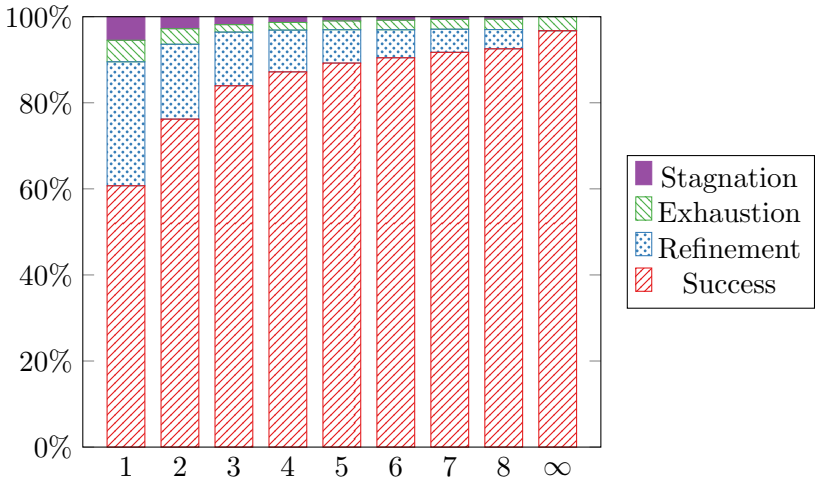


Figure 2: Lookahead result for Refinement-HC with varying depth bounds (x-axis).

the possible synergy with online refinement of h^{CFF}; finally, we evaluate these techniques experimentally as before.

6.1 Replacing the Depth Bound with Novelty Pruning

As pointed out in Section 5.3, 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 5.3.[2]

6.2 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 2 (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 $c \not\subseteq s'$ 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 toward the former.

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

6.3.1 Comparison of Lookahead Search Algorithms

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

2. 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 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.

The left side of Table 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 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 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).

6.3.2 Evaluating the Macro Choices

	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 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).

Table 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 *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*).

7. 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.

7.1 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^{CFF}. 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 that 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 lowest 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.

The pseudo-code 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 backchaining 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 10). The main motivations behind only doing a single iteration of refinement instead of refining h until $h(s) > h(s')$ are (1) 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 (2) 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}, key challenges are (a) selecting the underlying plan from which the subgoals are derived, and (b) ensuring that h^{SC} values are not compared across different underlying plans. In BFWS (Lipovetzky & Geffner, 2017), relaxed plans for subgoal counting are computed in states where the 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 (b) 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).

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
8              // s is a dead end (no helpful actions pruning)
9              Mark s as a dead end, pruning it in future lookaheads
10             HANDLE_DEAD_END
11         if the previous lookahead iteration also originated at s and s ≠ I then
12             // previous refinement was unsuccessful
13             HANDLE_STAGNATION
14         // refine h once
15         REFINE_HEURISTIC
16         if h(s) = ∞ then
17             HANDLE_DEAD_END
18             break
19     else
20         s := s'
21         if s ⊇ G then
22             return SOLVED
23 return UNSOLVABLE
```

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

Proposition 3 *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 1, Refinement-HC with subgoal counting is complete. ■

7.2 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.

	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 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).

7.2.1 Evaluating the Macro Choices

Table 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^{CFF} 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).

7.2.2 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

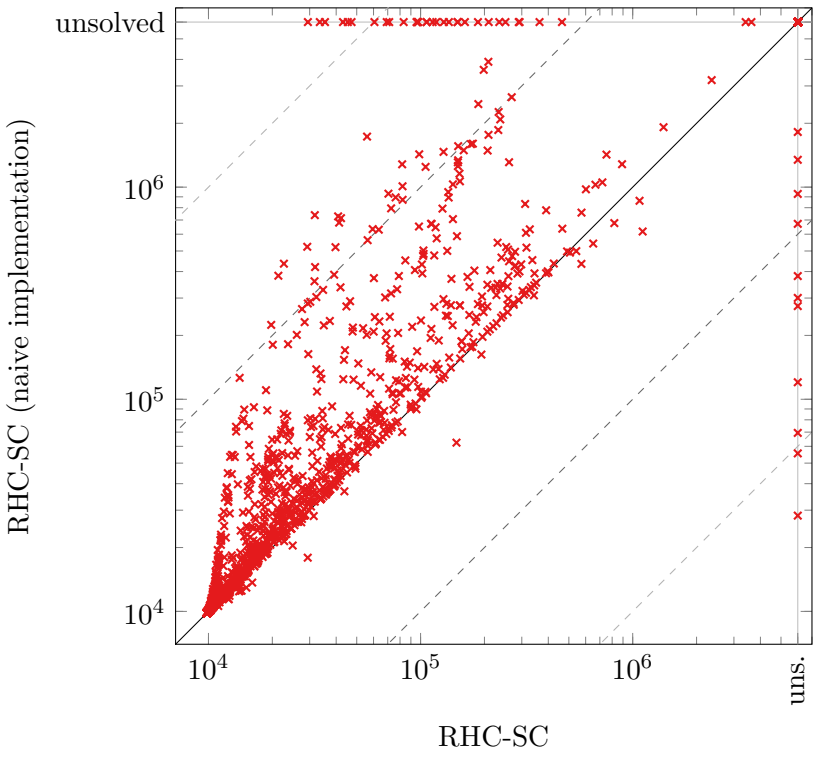


Figure 3: Comparison of the peak search memory usage (in kilobytes) for Refinement-HC with subgoal counting.

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 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 $h^{CFF} = \infty$).

Figure 3 shows a comparison of the memory usage of our implementation to a naive version that keeps all generated states in memory. With the exception of 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: Coverage of Refinement-HC with vs. without subgoal counting. Domains with equal coverage are grouped into "Others".

7.2.3 Comparison to Refinement-HC without Subgoal Counting

Table 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

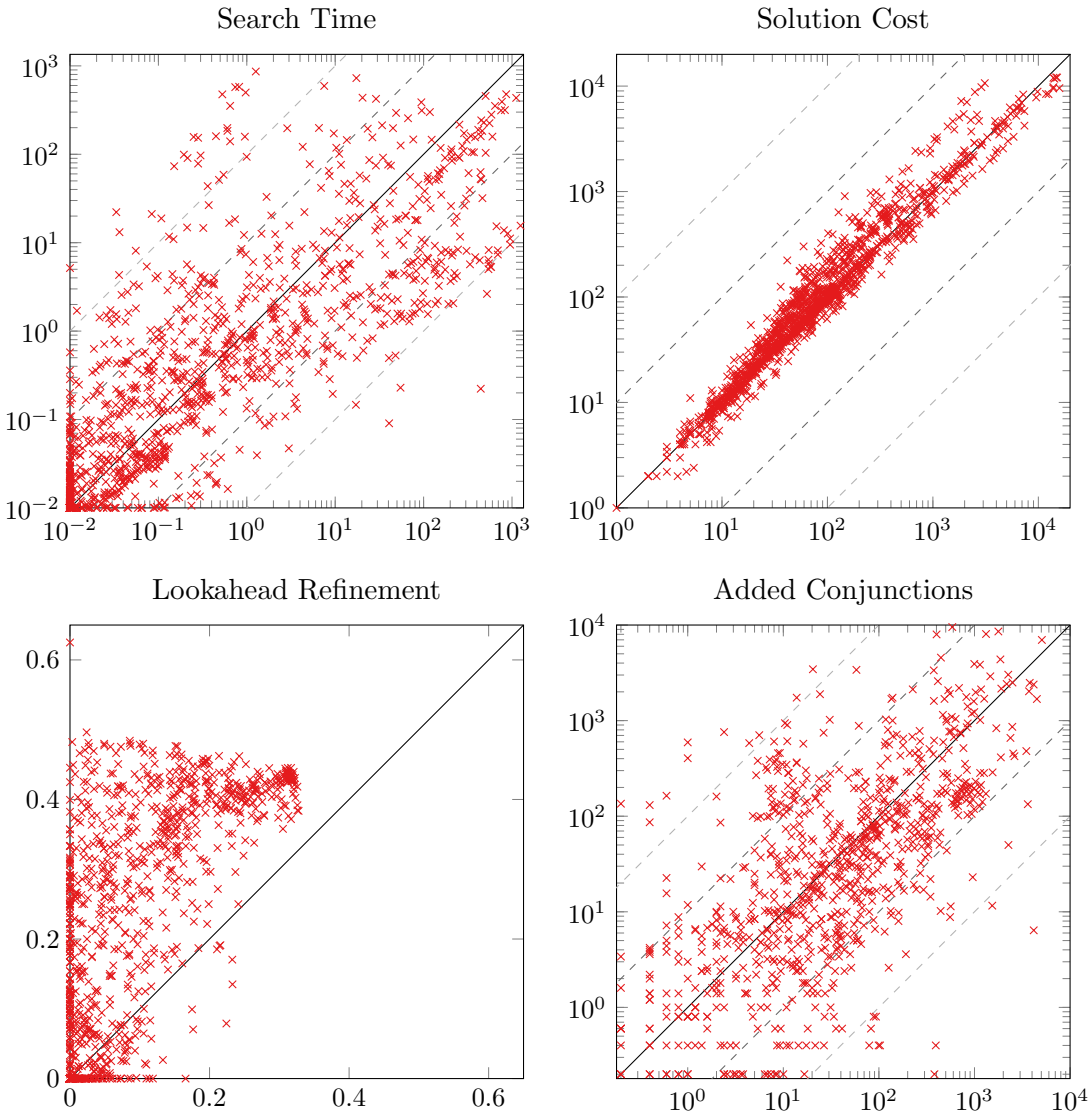


Figure 4: Search time, solution cost, percentage of lookaheads that result in refinement, and number of added conjunctions for commonly solved instances of RHC (X-axis) and RHC-SC (Y-axis).

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

Figure 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^{CFF} 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^{CFF} instead of refining until the h^{CFF} value of the lookahead root state increases sufficiently.

8. 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.

8.1 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 7 is particularly suitable as a lookahead function in GBFS due to its low over-

head. This lookahead strategy achieves two purposes in GBFS: (i) 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 (ii) it allows the search to skip ahead to a state closer to the goal (if the lookahead does find a better state).[3]

8.2 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*)[4], 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: 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
```

Algorithm 8 shows the pseudo-code 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, in other words, it performs a bounded lookahead search using h^{SC}

3. This method relates to previous works on lookahead strategies in GBFS (Vidal, 2004, 2011; Nakhost & Müller, 2009; Lipovetzky & Geffner, 2017); we will discuss these below.
4. 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.

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.[5] 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 (irrespective of its heuristic value), 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.

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 (Helmert, 2006) (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 $\pi[h^{\mathrm{FF}}](s)$, 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 & Müller, 2009), or bounded local search (Xie, Müller, & Holte, 2014; Lipovetzky & 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

5. 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 pseudo-code.

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 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 for both C-novelty and 1-novelty pruning are shown (with the latter in parentheses). The best-performing lookahead search algorithm in GBFS-SCL for each heuristic is <u>underlined</u>, and the overall best configuration is **boldfaced**.

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.

8.3 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 $h^{C\text{FF}}$, 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 $h^{C\text{FF}}$ with offline refinement. The red-black heuristic h^{RB} "un-relaxes" a subset of the finite-domain state variables (these variables correspond to mutex groups in the STRIPS framework) by inserting repair sequences for these variables into the relaxed plans (Domshlak et al., 2015). 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, Steinmetz, Gnad, Hoffmann, & Gerevini, 2017). For $h^{C\text{FF}}$ 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 $h^{C\text{FF}}$ (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, that is, 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.

Table 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}.[6] 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 6.2, namely, 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, for example, 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}. 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 $h^{C\text{FF}}$ and online refinement, GBFS-SCL clearly outclasses all other configurations with an overall coverage of 1558.8 ± 4.7, beating for example 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 (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, for example, 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).

Table 6 directly compares GBFS-SCL with $h^{C\text{FF}}$ using online vs. offline refinement. The online-refinement variant is almost universally superior, with better coverage on 23 domains

6. 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 paper (in particular in Section 9.1).

Coverage	$h^{\text{CFF}}_{\text{off}}$	$h^{\text{CFF}}_{\text{on}}$	Diff.
Airport (50)	37.6	**41.2**	+3.6
Barman (40)	37.8	**39.4**	+1.6
Childsnack (20)	6.8	**8.4**	+1.6
DataNetwork (20)	12.0	**18.6**	+6.6
Freecell (80)	78.4	**80.0**	+1.6
Nomystery (20)	6.2	**9.6**	+3.4
Openstacks (90)	63.8	**90.0**	+26.2
Parcprinter (40)	32.6	**40.0**	+7.4
Parking (40)	**39.6**	35.6	-4.0
Pathways (30)	23.2	**24.8**	+1.6
Pipes-notank (50)	44.2	**49.6**	+5.4
Pipes-tank (50)	43.6	**48.8**	+5.2
Snake (20)	14.4	**17.2**	+2.8
Sokoban (30)	**23.2**	17.8	-5.4
Spider (20)	13.8	**15.8**	+2.0
Storage (30)	25.8	**30.0**	+4.2
Termes (20)	**12.8**	6.8	-6.0
Tetris (20)	14.8	**19.8**	+5.0
Trucks (30)	14.8	**18.4**	+3.6
Others (995)	944.8	**947.0**	+2.2
Sum (1695)	1490.2	**1558.8**	+68.6
Std. Err.	5.8	4.7	
Normalized (%)	84.5	**88.0**	
Search Time (s)	0.86	**0.36**	

Table 6: Coverage of GBFS-SCL (using A* in the lookahead) with $h^{\text{CFF}}_{\text{off}}$ and $h^{\text{CFF}}_{\text{on}}$. Domains where the difference in coverage is less than one are grouped into "Others".

(worse on 5). The most significant exceptions are Termes (−6.0 coverage), Sokoban (−5.4), and Parking (−4.0). In Termes and Sokoban, online refinement is triggered frequently, making the heuristic much more costly to compute (the growth factor becomes more than 50 on some instances). On the other hand, in many domains few conjunctions suffice (or even none at all), and $h^{\text{CFF}}_{\text{off}}$ incurs more overhead than $h^{\text{CFF}}_{\text{on}}$. One such example is Openstacks, where online refinement never triggered on 87 of the 90 tasks. Furthermore, offline refinement results in a less informative heuristic than h^{FF}. This combination of unnecessary overhead and a weaker heuristic results in the large difference of 26.2 solved instances between $h^{\text{CFF}}_{\text{on}}$ and $h^{\text{CFF}}_{\text{off}}$. Overall, $h^{\text{CFF}}_{\text{on}}$ not only vastly improves coverage, but also reduces search time on commonly solved instances by 58% on average.

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 5 illustrates the lookahead on a small

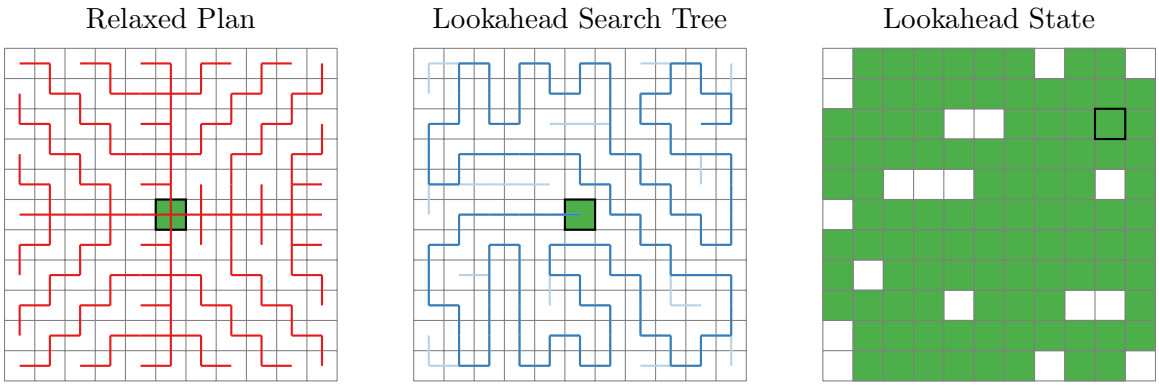


Figure 5: Illustration of GBFS-SCL's lookahead on an 11x11 instance of VisitAll.

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.

9. 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 as well as the recently published Autoscale benchmarks (Torralba et al., 2021).

9.1 Comparison to Baselines without Online Refinement

Table 7 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 & 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) $h^{C\mathrm{FF}}$.

Comparing just h^{FF} with offline $h^{C\mathrm{FF}}$; 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 $h^{C\mathrm{FF}}$. This has two major reasons. First, adding conjunctions intro-

Coverage	EHC		FF		GBFS		RHC	RHC-SC	GBFS-SCL
	h^{FF}	$h^{C\mathrm{FF}}$	h^{FF}	$h^{C\mathrm{FF}}$	h^{FF}	$h^{C\mathrm{FF}}$			
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 7: Coverage on the IPC benchmarks for traditional search algorithms using h^{FF} and (offline-refined) $h^{C\mathrm{FF}}$ compared to our online-refinement search algorithms.

duces 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 & Ruml, 2016), as the search may be guided into areas where the heuristic is less informed and its values are closer to h^{FF}.

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.

9.2 Online vs. Offline Conjunctions Quality

If the set of conjunctions C for $h^{C\mathrm{FF}}$ 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, that is, 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 $h^{C\mathrm{FF}}$ 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 6 compares the number of expansions of such GBFS searches. In this experiment, C_{on} is generated by RHC – 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 $h^{C\mathrm{FF}}$ 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.

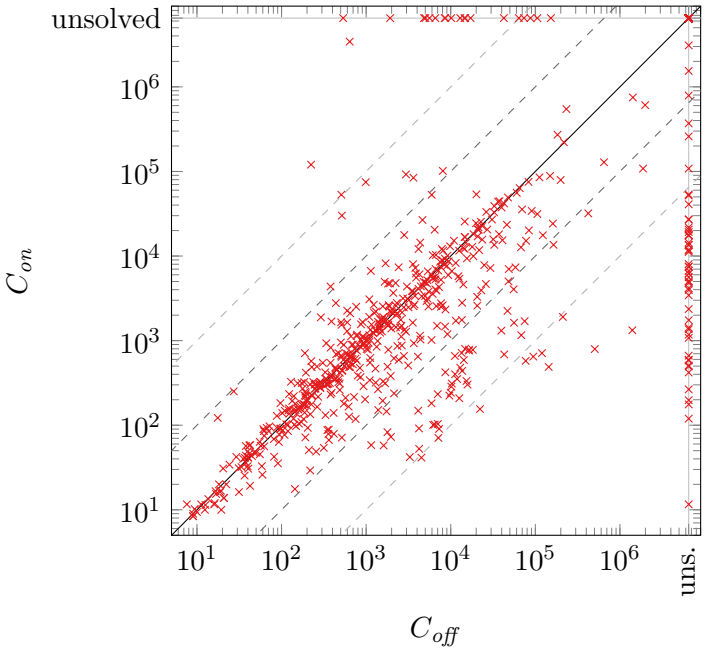


Figure 6: Expansions for GBFS with h^{CFF} using conjunctions generated online by RHC (C_{on}) vs. conjunctions generated offline in the initial state (C_{off}).

9.3 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 & Westphal, 2010), which runs a greedy best-first search using h^{FF} and a landmark-counting heuristic (Richter, Helmert, & Westphal, 2008) in an alternating queue,
- *Mercury* (Katz & 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 & 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 & 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, Gnad, Speicher, & Hoffmann, 2018), another portfolio planner using several different state-of-the-art techniques, including decoupled search (Gnad & 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 Alcazar 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 a strong 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 8 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 rows of Table 8 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 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

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 8: 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
Blocks (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 9: Coverage on the Autoscale benchmarks.

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 9 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 to e.g. Saarplan, which 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.

10. Related Work

As pointed out above, 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 & 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 & Fickert, 2018), which has been inspired by our earlier work on Refinement-HC (Fickert & 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 & 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 & 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 abstractions, 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) 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, Zilles, and Holte (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, Bradtke, & Singh, 1995; Bonet & Geffner, 2003). In particular, the search strategy of LSS-LRTA* (Koenig & 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 & Fickert, 2018), Eifler, Fickert, Hoffmann, and Ruml (2019) have shown that the idea can be transferred to real-time planning, often resulting in better performance due to the added generalization.

Finally, Thayer, Dionne, and Ruml (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.

11. 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 work 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 $h^{C\text{FF}}$ heuristic, makes them competitive with state-of-the-art systematic search approaches. On the IPC and Autoscale benchmarks, our algorithms with $h^{C\text{FF}}$ 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 $h^{C\text{FF}}$ so as to be able to prune more dead-end states (Steinmetz & 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^{CFF}, without any information flow specific to novelty pruning. It remains an open question how to identify conjunctions that are useful 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.

Acknowledgements

Maximilian Fickert was funded by DFG grant 389792660 as part of TRR 248 – CPEC, see https://perspicuous-computing.science. Part of this work has previously been published in three conference papers (Fickert & Hoffmann, 2017a; Fickert, 2018, 2020). This paper adds several extensions, including a novel online-refinement hill-climbing algorithm leveraging subgoal counting in its local lookahead search, and analyzes the algorithms and results in much more depth.

References

Alcázar, V., & Torralba, Á. (2015). A reminder about the importance of computing and exploiting invariants in planning. In Brafman, R., Domshlak, C., Haslum, P., & Zilberstein, S. (Eds.), *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS'15)*, pp. 2–6. AAAI Press.

Arfaee, S. J., Zilles, S., & Holte, R. C. (2011). Learning heuristic functions for large state spaces. *Artificial Intelligence*, *175*(16-17), 2075–2098.

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, *72*(1–2), 81–138.

Bonet, B., & Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, *129*(1–2), 5–33.

Bonet, B., & Geffner, H. (2003). Labeled RTDP: Improving the convergence of real-time dynamic programming. In Giunchiglia, E., Muscettola, N., & Nau, D. (Eds.), *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS'03)*, pp. 12–21, Trento, Italy. AAAI Press.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., & Veith, H. (2003). Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the Association for Computing Machinery*, *50*(5), 752–794.

Clarke, E. M., Grumberg, O., & Long, D. E. (1994). Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, *16*(5), 1512–1542.

Culberson, J. C., & Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, *14*(3), 318–334.

Domshlak, C., Hoffmann, J., & Katz, M. (2015). Red-black planning: A new systematic approach to partial delete relaxation. *Artificial Intelligence*, *221*, 73–114.

Domshlak, C., Karpas, E., & Markovitch, S. (2012). Online speedup learning for optimal planning. *Journal of Artificial Intelligence Research*, *44*, 709–755.

Edelkamp, S. (2001). Planning with pattern databases. In Cesta, A., & Borrajo, D. (Eds.), *Proceedings of the 6th European Conference on Planning (ECP'01)*, pp. 13–24. Springer-Verlag.

Eifler, R., & Fickert, M. (2018). Online refinement of Cartesian abstraction heuristics. In Bulitko, V., & Storandt, S. (Eds.), *Proceedings of the 11th Annual Symposium on Combinatorial Search (SOCS'18)*. AAAI Press.

Eifler, R., Fickert, M., Hoffmann, J., & Ruml, W. (2019). Refining abstraction heuristics during real-time planning. In Hentenryck, P. V., & Zhou, Z.-H. (Eds.), *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI'19)*. AAAI Press.

Felner, A., Korf, R., & Hanan, S. (2004). Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, *22*, 279–318.

Fickert, M. (2018). Making hill-climbing great again through online relaxation refinement and novelty pruning. In Bulitko, V., & Storandt, S. (Eds.), *Proceedings of the 11th Annual Symposium on Combinatorial Search (SOCS'18)*, pp. 158–162. AAAI Press.

Fickert, M. (2020). 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'20)*, pp. 119–123. AAAI Press.

Fickert, M., Gnad, D., Speicher, P., & Hoffmann, J. (2018). Saarplan: Combining Saarland's greatest planning techniques. In *IPC 2018 planner abstracts*.

Fickert, M., & Hoffmann, J. (2017a). Complete local search: Boosting hill-climbing through online heuristic-function refinement. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS'17)*, pp. 107–115. AAAI Press.

Fickert, M., & Hoffmann, J. (2017b). Ranking conjunctions for partial delete relaxation heuristics in planning. In Fukunaga, A., & Kishimoto, A. (Eds.), *Proceedings of the 10th Annual Symposium on Combinatorial Search (SOCS'17)*, pp. 38–46. AAAI Press.

Fickert, M., Hoffmann, J., & Steinmetz, M. (2016). Combining the delete relaxation with critical-path heuristics: A direct characterization. *Journal of Artificial Intelligence Research*, *56*(1), 269–327.

Fikes, R. E., & Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*, 189–208.

Fink, M. (2007). Online learning of search heuristics. In *Proceedings of the 11th International Conference on Artificial Intelligence and Statistics (AISTATS'07)*, pp. 114–122.

Francès, G., Lipovetzky, N., Geffner, H., & Ramírez, M. (2018). Best-first width search in the IPC 2018: Complete, simulated, and polynomial variants. In *IPC 2018 planner abstracts*.

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Gnad, D., & Hoffmann, J. (2018). Star-topology decoupled state space search. *Artificial Intelligence*, *257*, 24 – 60.

Haslum, P. (2006). Improving heuristics through relaxed search - an analysis of TP4 and HSP$_a^*$ in the 2004 planning competition. *Journal of Artificial Intelligence Research*, *25*, 233–267.

Haslum, P. (2012). Incremental lower bounds for additive cost planning problems. In Bonet, B., McCluskey, L., Silva, J. R., & Williams, B. (Eds.), *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, pp. 74–82. AAAI Press.

Haslum, P., & Geffner, H. (2000). Admissible heuristics for optimal planning. In Chien, S., Kambhampati, R., & Knoblock, C. (Eds.), *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS'00)*, pp. 140–149, Breckenridge, CO. AAAI Press, Menlo Park.

Helmert, M. (2006). The Fast Downward planning system. *Journal of Artificial Intelligence Research*, *26*, 191–246.

Helmert, M., Haslum, P., & Hoffmann, J. (2007). Flexible abstraction heuristics for optimal sequential planning. In Boddy, M., Fox, M., & Thiebaux, S. (Eds.), *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS'07)*, pp. 176–183, Providence, Rhode Island, USA. Morgan Kaufmann.

Helmert, M., Haslum, P., Hoffmann, J., & Nissim, R. (2014). Merge & shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the Association for Computing Machinery*, *61*(3), 16:1–16:63.

Helmert, M., Röger, G., & Karpas, E. (2011). Fast Downward Stone Soup: A baseline for building planner portfolios. In *Proceedings of the ICAPS Workshop on Planning and Learning (PAL'11)*.

Hoffmann, J., & Fickert, M. (2015). Explicit conjunctions w/o compilation: Computing $h^{\mathrm{FF}}(\Pi^C)$ in polynomial time. In Brafman, R., Domshlak, C., Haslum, P., & Zilberstein, S. (Eds.), *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS'15)*. AAAI Press.

Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, *14*, 253–302.

Karpas, E., Katz, M., & Markovitch, S. (2011). When optimal is just not good enough: Learning fast informative action cost-partitionings. In Bacchus, F., Domshlak, C., Edelkamp, S., & Helmert, M. (Eds.), *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS'11)*, pp. 122–129. AAAI Press.

Katz, M., & Domshlak, C. (2010). Optimal admissible composition of abstraction heuristics. *Artificial Intelligence*, *174*(12–13), 767–798.

Katz, M., & Hoffmann, J. (2014). Mercury planner: Pushing the limits of partial delete relaxation. In *IPC 2014 planner abstracts*, pp. 43–47.

Katz, M., Lipovetzky, N., Moshkovich, D., & Tuisov, A. (2017). Adapting novelty to classical planning as heuristic search. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS'17)*, pp. 172–180. AAAI Press.

Katz, M., Lipovetzky, N., Moshkovich, D., & Tuisov, A. (2018). MERWIN planner: Mercury enchanced with novelty heuristic. In *IPC 2018 planner abstracts*, pp. 53–56.

Keyder, E., Hoffmann, J., & Haslum, P. (2012). Semi-relaxed plan heuristics. In Bonet, B., McCluskey, L., Silva, J. R., & Williams, B. (Eds.), *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, pp. 128–136. AAAI Press.

Keyder, E., Hoffmann, J., & Haslum, P. (2014). Improving delete relaxation heuristics through explicitly represented conjunctions. *Journal of Artificial Intelligence Research*, *50*, 487–533.

Koenig, S., & Sun, X. (2009). Comparing real-time and incremental heuristic search for real-time situated agents. *Auton. Agents Multi Agent Syst.*, *18*(3), 313–341.

Korf, R. E. (1990). Real-time heuristic search. *Artificial Intelligence*, *42*(2-3), 189–211.

Lipovetzky, N., & Geffner, H. (2012). Width and serialization of classical planning problems. In Raedt, L. D. (Ed.), *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI'12)*, pp. 540–545, Montpellier, France. IOS Press.

Lipovetzky, N., & Geffner, H. (2014). Width-based algorithms for classical planning: New results. In Schaub, T. (Ed.), *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI'14)*, pp. 1059–1060, Prague, Czech Republic. IOS Press.

Lipovetzky, N., & Geffner, H. (2017). Best-first width search: Exploration and exploitation in classical planning. In Singh, S., & Markovitch, S. (Eds.), *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI'17)*, pp. 3590–3596. AAAI Press.

Nakhost, H., & Müller, M. (2009). Monte-carlo exploration for deterministic planning. In Boutilier, C. (Ed.), *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pp. 1766–1771, Pasadena, California, USA. Morgan Kaufmann.

Richter, S., Helmert, M., & Westphal, M. (2008). Landmarks revisited. In Fox, D., & Gomes, C. (Eds.), *Proceedings of the 23rd National Conference of the American Association for Artificial Intelligence (AAAI'08)*, pp. 975–982, Chicago, Illinois, USA. AAAI Press.

Richter, S., & Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, *39*, 127–177.

Rovner, A., Sievers, S., & Helmert, M. (2019). Counterexample-guided abstraction refinement for pattern selection in optimal classical planning. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS'19)*, pp. 362–367. AAAI Press.

Seipp, J. (2021). Online saturated cost partitioning for classical planning. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS'21)*, pp. 317–321. AAAI Press.

Seipp, J., & Helmert, M. (2013). Counterexample-guided Cartesian abstraction refinement. In Borrajo, D., Fratini, S., Kambhampati, S., & Oddi, A. (Eds.), *Proceedings of the*

23rd International Conference on Automated Planning and Scheduling (ICAPS'13), pp. 347–351, Rome, Italy. AAAI Press.

Seipp, J., & Helmert, M. (2014). Diverse and additive Cartesian abstraction heuristics. In Chien, S., Do, M., Fern, A., & Ruml, W. (Eds.), *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS'14)*. AAAI Press.

Seipp, J., & Helmert, M. (2018). Counterexample-guided Cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research*, *62*, 535–577.

Seipp, J., Keller, T., & Helmert, M. (2020). Saturated cost partitioning for optimal classical planning. *Journal of Artificial Intelligence Research*, *67*, 129–167.

Seipp, J., Pommerening, F., Sievers, S., & Helmert, M. (2017). Downward Lab. https://doi.org/10.5281/zenodo.790461.

Seipp, J., & Röger, G. (2018). Fast Downward Stone Soup 2018. In *IPC 2018 planner abstracts*.

Speicher, P., Steinmetz, M., Gnad, D., Hoffmann, J., & Gerevini, A. (2017). Beyond red-black planning: Limited-memory state variables. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS'17)*, pp. 269–273. AAAI Press.

Steinmetz, M., & Hoffmann, J. (2017a). Critical-path dead-end detection vs. nogoods: Offline equivalence and online learning. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS'17)*, pp. 283–287. AAAI Press.

Steinmetz, M., & Hoffmann, J. (2017b). State space search nogood learning: Online refinement of critical-path dead-end detectors in planning. *Artificial Intelligence*, *245*, 1–37.

Steinmetz, M., & Hoffmann, J. (2018). LP heuristics over conjunctions: Compilation, convergence, nogood learning. In Lang, J. (Ed.), *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*, pp. 4837–4843.

Thayer, J. T., Dionne, A. J., & Ruml, W. (2011). Learning inadmissible heuristics during search. In Bacchus, F., Domshlak, C., Edelkamp, S., & Helmert, M. (Eds.), *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS'11)*. AAAI Press.

Torralba, Á., Seipp, J., & Sievers, S. (2021). Automatic instance generation for classical planning. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS'21)*, pp. 376–384. AAAI Press.

Vidal, V. (2004). A lookahead strategy for heuristic search planning. In Koenig, S., Zilberstein, S., & Koehler, J. (Eds.), *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS'04)*, pp. 150–160, Whistler, Canada. AAAI Press.

Vidal, V. (2011). YAHSP2: Keep it simple, stupid. In *IPC 2011 planner abstracts*, pp. 83–90.

Wilt, C. M., & Ruml, W. (2013). Robust bidirectional search via heuristic improvement. In desJardins, M., & Littman, M. (Eds.), *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI'13)*, Bellevue, WA, USA. AAAI Press.

Wilt, C. M., & Ruml, W. (2016). Effective heuristics for suboptimal best-first search. *Journal of Artificial Intelligence Research*, *57*, 273–306.

Xie, F., Müller, M., & Holte, R. (2014). Adding local exploration to greedy best-first search in satisficing planning. In Brodley, C. E., & Stone, P. (Eds.), *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14)*, pp. 2388–2394, Austin, Texas, USA. AAAI Press.