Refining Abstraction Heuristics During Real-Time Planning

Rebecca Eifler and **Maximilian Fickert** and **Jörg Hoffmann**
Saarland University, Saarland Informatics Campus
Saarbrücken, Germany
{eifler,fickert,hoffmann}@cs.uni-saarland.de

Wheeler Ruml
University of New Hampshire
Durham, NH USA
ruml at cs.unh.edu

Abstract

In real-time planning, the planner must select the next action within a fixed time bound. Because a complete plan may not have been found, the selected action might not lead to a goal and the agent may need to return to its current state. To preserve completeness, real-time search methods incorporate learning, in which heuristic values are updated. Previous work in real-time search has used table-based heuristics, in which the values of states are updated individually. In this paper, we explore the use of abstraction-based heuristics. By refining the abstraction on-line, we can update the values of multiple states, including ones the agent has not yet generated. We test this idea empirically using Cartesian abstractions in the Fast Downward planner. Results on various benchmarks, including the sliding tile puzzle and several IPC domains, indicate that the approach can improve performance compared to traditional heuristic updating. This work brings abstraction refinement, a powerful technique from offline planning, into the real-time setting.

Introduction

In some AI applications, it is unacceptable for the system to exhibit unbounded pauses while planning. In real-time planning, the planner must return the next action for the system to take within a strict time limit. Real-time search is a well-established area addressing this through dedicated heuristic search algorithms (e.g. Korf 1990; Bulitko and Lee 2006; Koenig and Sun 2009; Bulitko et al. 2011; Hernández and Baier 2012; Kiesel, Burns, and Ruml 2015). One important issue in this context is completeness: guaranteeing that the agent will eventually reach a goal state. In domains without dead-ends (unsolvable states from which a goal is not reachable), this can be achieved through the refinement of state-value estimates: optimistic estimates of the true goal distance of a state, continuously refined throughout the search, possibly until convergence to optimal values. Traditionally, real-time search algorithms do this by storing distance estimates for individual states, and refining these with Bellman state-value updates (Bellman 1957).

However, more effective methods may exist. In particular, *abstraction* is a well-explored means to approximate transition systems (e.g. Clarke, Grumberg, and Long 1994), including the approximation of goal distance (e.g. Edelkamp 2001; Haslum et al. 2007; Seipp and Helmert 2013; Helmert et al. 2014). In this paper, we explore abstract state spaces that are quotient graphs of the state space, where abstract states are blocks in a state partition, and the goal distance estimate for any state is the distance of its block to the nearest goal block. Refinement operations split blocks into smaller ones, converging at the latest when all blocks are singletons.

Compared to per-state updates, abstraction offers several potential advantages: i) *generalization*, as a single refinement operation may improve the estimates of many states; ii) *compactness*, as an abstraction may use exponentially less space to store the same value approximation; and iii) *flexibility*, as the abstraction can be refined anywhere in the state space, not just on the states explored by search. Of course, it is a priori not clear how to realize this potential. Abstraction refinement i) is computationally more expensive than a per-state update. Compactness ii) may be outweighed by the overhead of maintaining a global heuristic function rather than just a function on the state set seen thus far. Flexibility iii) is a mixed blessing as it introduces the need to define how and where to refine the abstraction.

Here we begin to explore these research questions. We design several strategies for refining abstractions given a real-time lookahead search space. These strategies are, per se, agnostic of how the abstraction is represented and how refinement operations are realized. Where relevant, we discuss these questions for *Cartesian abstraction*, which assumes a representation of the state space in terms of state variables, and restricts blocks to be cross-products of state-variable domain subsets (Ball, Podelski, and Rajamani 2001; Seipp and Helmert 2013). Cartesian abstraction allows fine-grained state partitions yet supports effective refinement operations, and it underlies the current state of the art in using abstractions to design heuristic functions in planning (Seipp and Helmert 2014; Seipp 2017; Seipp and Helmert 2018).

Based on the *Fast Downward (FD)* planning system (Helmert 2006) and its implementation of Cartesian abstraction, we run experiments on standard planning benchmarks from the *International Planning Competition (IPC)*, as well as on a planning encoding of the sliding tiles puzzle. Our results exhibit complementary strengths and weaknesses, of traditional Bellman updates vs. Cartesian abstraction refinement. On a variety of benchmarks our methods result in significant coverage improvements, in particular on Logistics, Mprime, Parcprinter, Termes, Transport, and Woodworking.

Background

We briefly provide the necessary background in real-time search, classical planning, and Cartesian abstractions.

Real-Time Search

Following the seminal work of Korf (1990), the conventional paradigm in real-time heuristic search has two phases: lookahead, in which the planner expands nodes starting from the agent's current state to form a *local search space*, and then learning, in which the heuristic cost-to-go values of nodes on the lookahead frontier are used to derive possibly-improved heuristic values for states in the local search space. If the current heuristic value of a state is lower than the minimum, over all of its successors, of the cost to the successor plus the successor's heuristic value, then the state's heuristic value can be raised to that value. This update is a deterministic version of Bellman's backup (Bellman 1957) as used in value iteration methods from Markov decision processes and reinforcement learning (Barto, Bradtke, and Singh 1995). Although Korf's algorithms updated only the value of the agent's current state, Koenig and Sun (2009) noted that the entire local search space can be updated. If the lookahead is bounded, then the learning is bounded, and the agent can select an action within a real-time bound.

Most heuristic search research considers domains with discrete states, and updated heuristic values are usually stored in a hash table. In continuous domains, as often considered in reinforcement learning, or in very large domains, generalization across states is crucial and heuristic state values are often approximated, e.g. by linear functions whose parameters are adjusted during learning (Boyan and Moore 1995; Lagoudakis and Parr 2003). Here, we consider discrete domains but aim at generalization through abstraction.

Heuristics based on abstraction have been extensively explored in off-line search. Often these heuristics are constructed before search begins (e.g. Edelkamp 2001), although in hierarchical search methods the heuristic is created on-line by instantiating only the portion of the abstract state space that is relevant for the states reached during search (e.g. Larsen et al. 2010). However, in these methods the abstraction itself is defined before search begins. In this paper, we aim to revise the abstraction online using experience from the on-going search.

Classical Planning

While the methods we explore in this paper are in principle applicable to any domain addressed by real-time search, our current implementation and experiments are done in the context of *classical planning*. Planning as a sub-area of AI is concerned with sequential decision making problems where the state space is compactly represented through factored models of states and actions (Ghallab, Nau, and Traverso (2004) give an overview). In classical planning, the initial state is completely known, actions are deterministic, and the agent's objective is to reach one of a set of goal states using a plan of minimum cost. Specifically, we consider the *finite-domain representation (FDR)* (Bäckström and Nebel 1995; Helmert 2009), where a *planning task* is a tuple $\Pi = (V, A, c, I, G)$, with V being the set of finite-domain *state variables*, each $v \in V$ with a finite *domain* D_v; A the set of *actions*; $c : A \mapsto \mathbb{R}_0^+$ the *cost function*; I the *initial state*, and G the *goal*. States s are value-assignments to V. An action a is modeled by its *precondition* pre_a, a partial assignment to V that must hold for a to be *applicable*; and its *effect* eff_a, a partial assignemnt to V that is set by a when applied. If a is applicable in a state s, the result of applying a to s is the state $s[[a]]$, whose variable values are given by $s[[a]](v) := \mathit{eff}_a(v)$ if $\mathit{eff}_a(v)$ is defined, and $s[[a]](v) := s(v)$ otherwise. G is a partial assignment to V that must hold at the end of a solution (a *plan*): an action sequence that, applied in I, leads to a state that contains G.

Cartesian Abstraction Heuristics

Many kinds of admissible heuristics have been developed for FDR problems; one prominent type is based on abstraction. Given that the number of states in a planning task is exponential in its size, the question arises how an abstraction – a partition of the state space – should be represented. Available answers are *pattern databases* (Culberson and Schaeffer 1998; Edelkamp 2001; Haslum et al. 2007), where an abstraction is defined in terms of a state-variable subset projected onto; *merge-and-shrink abstraction* (Dräger, Finkbeiner, and Podelski 2006; Helmert, Haslum, and Hoffmann 2007; Helmert et al. 2014), which computes arbitrary abstractions by iteratively merging state variables and abstracting (shrinking) the product; and *Cartesian abstraction* (Ball, Podelski, and Rajamani 2001; Seipp and Helmert 2013), where abstract states are restricted to be cross-products of state-variable domain subsets.

We use Cartesian abstraction here as it allows fine-grained state partitions (in difference to pattern databases) yet supports effective refinement operations (in difference to merge-and-shrink abstraction). Furthermore, Cartesian abstractions can provide state-of-the-art performance when using not one but an *ensemble* of abstractions, made *additive* through *cost partitioning* (Katz and Domshlak 2008; Seipp and Helmert 2014; Seipp 2017) where the cost of each action is distributed across abstractions.

To obtain a Cartesian abstraction in practice, i.e., to find and compute a concrete abstraction, counter-example guided abstraction refinement (CEGAR) is used (Seipp and Helmert 2018). This procedure iteratively splits abstract states, starting from a trivial abstraction (single abstract state containing the entire state space). In each iteration, an optimal solution is computed in the abstraction as a trace (an alternating sequence of abstract states and actions) $\tau = \langle [s'_0], a_1, \ldots, [s'_{n-1}], a_n, [s'_n] \rangle$, where $[s]$ denotes the abstract state containing the state s. If no solution can be found, the task is unsolvable and we are done. Otherwise, we test if τ can be converted into a *concrete* trace τ'. By applying the actions in τ' we obtain a sequence of states $s_0, s_1, \ldots, s_n$. One of the following flaws may occur in the concrete trace:

1. The action a_{i+1} is not applicable in the concrete state s_i.

2. The concrete state s_i is not contained in the corresponding abstract state $[s'_i]$ in τ, i.e. $[s_i] \neq [s'_i]$.
3. The concrete trace is completed, but s_n is not a goal state.

If there is no flaw, we already have a solution to the task. Otherwise, we can split an abstract state such that the flaw cannot occur in future iterations as follows, corresponding to the cases above:

1. Split $[s_i]$ into $[t']$ and $[u']$ such that $s_i \in [t']$ and a_{i+1} is not applicable in any of the states contained in $[t']$.
2. Split $[s_{i-1}]$ into $[t']$ and $[u']$ such that $s_{i-1} \in [t']$ and a_i does not lead from a state in $[t']$ to a state in $[s'_i]$.
3. Split $[s_n]$ into $[t']$ and $[u']$ such that $s_n \in [t']$ and $[t']$ does not contain a goal state.

The abstraction is then updated by replacing the state that was split with the two resulting states, and the transitions are updated accordingly.

The CEGAR process is traditionally run offline, before the search for a plan begins. Left running indefinitely, it will eventually either find a plan for the task, or prove the task unsolvable (i.e., the CEGAR process *converges*). In practice, this is infeasible, so instead some termination criterion (typically a time or memory bound) is used to abort the process early on. This termination criterion controls the trade-off between heuristic accuracy vs. computational overhead.

Eifler and Fickert (2018) have recently proposed a setup where, in addition to the offline CEGAR process, the abstraction is continuously refined online, during an A* search, to improve weaknesses encountered. We adopt this setting to the real-time context here, where the initial heuristic function is obtained through offline CEGAR, while further refinements are made in each real-time search iteration. In particular, we employ the framework of Eifler and Fickert (2018) to realize refinement operations for additive ensembles of Cartesian abstractions. To guarantee convergence in this setting, this framework combines standard refinement steps (splitting abstract states) with abstraction-merging operations as well as operations adding more saturated cost partitionings based on different orders.

Eifler and Fickert (2018) have explored the benefit of online heuristic function refinement over offline such refinement, in a complete search setting. Here we instead focus on using online refinement in real-time search.

Real-Time Search with Abstraction Refinement

In keeping with the real-time setting, we consider time intervals of a fixed, predetermined size. Because we are comparing methods that use very different primitive operations, we use actual CPU time instead of node expansions to demarcate each time slot. This requires some care, as we explain below, as it is not obvious how much time before the end of the slot to reserve for the learning phase.

Bellman Updates

As a baseline, we implement the heuristic update scheme of the popular LSS-LRTA* algorithm (Koenig and Sun 2009), which uses a Dijkstra-like method to organize backups from the lookahead frontier back through the local search space. To avoid having to predict how long this will take, we perform the learning step of one lookahead at the start of the next iteration (see Figure 1). While sensible, our implementation is not strictly real-time: the Bellman updates are always completely executed even if the time slot is exceeded, and in the lookahead phase, at least one state is expanded so that the agent can commit to an action.

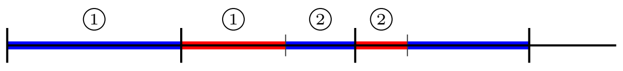

Figure 1: Time distribution for Bellman updates. The individual time intervals are marked with the black bars. The blue part in each time interval represents the time spent on lookahead, the read part is time spent on performing Bellman backups. The numbers indicate the lookahead search spaces to which the phases belong.

To identify the most promising action in constant time, we label each successor of the agent's current state with the action taken to reach it, and then subsequent states inherit during lookahead the label of their best known parent.

Abstraction Refinement

We can replace the Bellman-style per-state updates by refinement of the heuristic function. In each time interval, we employ a similar scheme as with the Bellman updates. We perform the same lookahead search procedure, but reserve a fraction of the time slot for the refinement of the abstraction underlying the heuristic function. This fraction is controlled by a lookahead ratio parameter l, which allows for a trade-off between lookahead and refinement (see Figure 2).

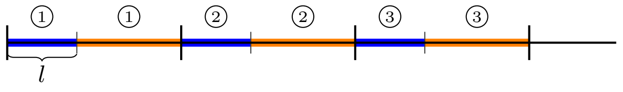

Figure 2: Time distribution for abstraction refinement. The orange part is time spent on refinement.

After the lookahead, the abstraction is iteratively refined on the current root state until the end of the time interval.

While Bellman updates only improve the heuristic on the states seen during the lookahead, abstraction refinement improves it for the remainder of the search, including states not yet visited. However, the refinement process refines the heuristic based on conflicts in the abstract solution, starting backwards from the goal. Thus, many refinement operations may be necessary until the heuristic value changes across states in the local search space around the current state.

Abstraction Refinement + Bellman Updates

Abstraction refinement can be combined with Bellman updates to combine the strengths of both: Bellman backups help quickly distinguishing between states in the local search space, while abstraction refinement improves the heuristic estimates also on future states.

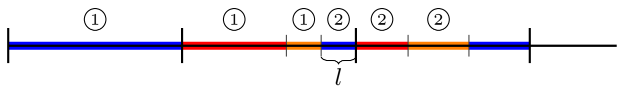

Figure 3: Time distribution for Abstraction Refinement + Bellman Updates. Note that l now describes a fraction of the remaining time frame after the Bellman updates.

In each time interval, we first apply the Bellman updates for the previous interval, then the remaining time of the current time frame is used for abstraction refinement and lookahead, again controlled by a parameter l (see Figure 3). Additionally, the refinement is stopped early if the Bellman equation is satisfied in the state on which refinement is called, leaving more time for the next lookahead which allows more states to be considered for the Bellman update (this improved results on preliminary experiments).

In this setting, we can slightly simplify the refinement algorithm by not merging abstractions. This modification avoids the most costly suboperation which may take several full time intervals to resolve. While it breaks convergence of the heuristic, we still retain completeness due to the convergence of the Bellman updates.

The heuristic function is now computed as the maximum of the value in the Bellman table and the heuristic value of the abstraction heuristic.

Simulating Bellman Updates by Refinement

Bellman updates can be simulated by abstraction refinement, i.e. performing the refinement in a way such that afterwards, every state in the lookahead satisfies the Bellman equation. To achieve this, we use a modified refinement algorithm described in the following. We restrict this to a heuristic using only a single abstraction (this is explained below).

Initially, the abstraction is refined such that each abstract state either only contains goal states, or none of the contained states is a goal. In each time frame, we perform refinement on each state s expanded in the lookahead as follows (in the same order as the Bellman updates).

In order to satisfy the Bellman equation, $h(s)$ must be at least $B(s) := \min_{a \in \mathcal{A}} h(s[[a]]) + c(a)$. By *Bellman refinement*, we refer to the following process. While $h(s) < B(s)$, let A' be an abstract successor state of $A = [s]$ that is reached by applying the action a', with $h(A') + c(a') < B(s)$. Then the action a' is a *shortcut* in the abstraction, leading to the estimate being too low in s. As applying a' in s can by construction of $B(s)$ not lead to a shortcut, this must be due to a state $t \in A$, $t \neq s$, in which a' is applicable and leads to a state contained in A'. Thus we can split A to address this issue (illustrated in Figure 4):

- If a' is not applicable in s, then we split A into states A_0 and A_1 such that $s \in A_0$ and a' is not applicable in any of the states contained in A_0 (this is similar to the first case in the CEGAR algorithm, c.f. Background).
- Otherwise, we split A into states A_0 and A_1 such that $s \in A_0$ and a' does not lead from a state in A_0 to a state in A' (similar to the second case in the CEGAR algorithm).

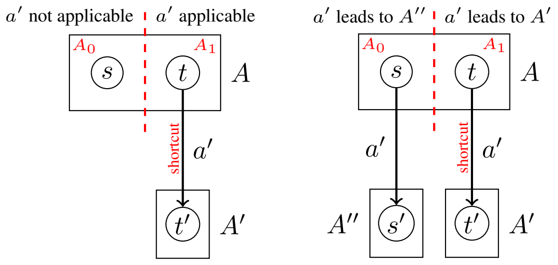

Figure 4: Illustration of the splits performed by the Bellman refinement procedure when there is a shortcut, i.e. there is an action a' with its abstract successor A' such that $h(A') + c(a') < B(s)$.

Proposition 1. *Let $\Pi = (V, A, c, I, G)$ be a planning task, and h a heuristic function for Π represented by a Cartesian abstraction. Let s be a state that does not satisfy the Bellman equation, and let h_b be the outcome of updating s in h. Then Bellman refinement terminates after at most $\sum_{v \in V}(|D_v| - 1)$ iterations, and the outcome heuristic h_{br} dominates h_b.*

Proof. The second part of the claim holds because 1) heuristic values can only increase through abstraction refinement, and 2) after Bellman refinement, by construction we have $h(s) = \min_{a \in \mathcal{A}} h(s[[a]]) + c(a)$, i.e., s satisfies the Bellman equation.

To see the first part of the claim, observe that the abstract state A that contains s becomes smaller in each iteration: for at least one state variable v, at least one value $d \in D_v$, $d \neq s(v)$, is removed from the subset of D_v underlying A. Hence, denoting that latter subset by $A[v]$, after at most $\sum_{v \in V}(|D_v| - 1)$ iterations, for all v we have $A[v] = \{s(v)\}$. Then s is the only state left in A, so there are no shortcuts anymore, and the process stops. □

This variant is mainly of theoretical interest, because typically, the number of refinement operations required to satisfy the Bellman equation for all states in the lookahead is prohibitively high in practice.

In principle, it is possible to adapt this algorithm to use an additive set of abstractions, but several issues arise. First, it may happen that the sum of the abstractions does not satisfy the Bellman equation, but all the individual ones do. Furthermore, it is not guaranteed that a shortcut as described in the algorithm exists in a specific abstraction. These issues could be fixed by merging abstractions whenever such a case occurs, but this is a very costly operation, and presumably, many abstractions would need to be merged very quickly. Thus, we only consider using a single abstraction here, and leave the question how this can be done efficiently with an additive set of abstractions for future work.

Experiments

Our techniques are implemented in Fast Downward (FD) (Helmert 2006), building upon the implementation of Cartesian abstraction heuristics (Seipp and Helmert 2018) and its online-refinement framework (Eifler and Fickert 2018).

We use the notation h_b for the configurations using Bellman updates, h_r for abstraction refinement, and h_{r+b} for the combination of refinement with Bellman updates.

We evaluate our techniques on IPC benchmarks and on the 15-puzzle domain. The experiments are run on a cluster of Intel Xeon E5-2650 v3 machines, with a time limit of 10 minutes and memory limit of 4 GB. Since we use CPU time to bound the time intervals, the results can differ across several runs. Hence, we average all results over three runs, counting an instance as solved if it was solved in at least two.

One metric we will discuss is *goal achievement time* (GAT), which is the overall time spent on planning and execution of the plan (Kiesel, Burns, and Ruml 2015). We assume the execution of an action takes exactly one time step.

Planning Benchmarks

We ran experiments on all STRIPS benchmarks from the optimal tracks of all IPCs up to IPC'18, yielding 1797 instances from 47 domains in total. Note that some of the domains have dead ends, where the real-time search approaches considered here are incomplete since decisions leading into a dead end cannot be reversed.

Ratio between Lookahead and Refinement First, we try to find the best ratio between lookahead and refinement in h_r by scaling the lookahead parameter l from 0.1 to 0.9. The results for time slots of 0.1 seconds are shown in Figure 5.

With more lookahead (increasing l), more expansions are necessary because the heuristic is less informed (see Figure 5b). However, with a lot of refinement, there is only little time left to do lookahead, so the decision which action should be applied has less information as fewer states were observed. The peak in both overall coverage (Figure 5a) and goal achievement time (Figure 5d) is at $l = 0.8$, yielding the best trade-off between lookahead and refinement.

The number of iterations until a goal is found can be less than the plan length (see Figure 5e), because whenever we find a goal in the lookahead, we immediately commit to all actions leading to this state.

For the remaining experiments, we use the overall best performing value of $l = 0.8$ for h_r. We use $l = 0.7$ for h_{r+b}, which we determined from similar experiments.

Coverage Results Table 1 shows the coverage results on the IPC domains for h_b, h_r, and h_{r+b}, with time steps of 0.01 and 0.1 seconds. All heuristics are initialized with an additive Cartesian abstraction heuristic of at most 1000 abstract states. The table also shows data for goal achievement time and generalization, which is discussed later.

With time intervals of 0.01 seconds, h_b performs best overall with 904 solved instances in total. The h_r configuration is best in only three domains (Logistics, Parcprinter, and Woodworking), but with significant margins to the next best heuristic in Parcprinter (+13) and Woodworking (+12). The combination of Bellman updates with abstraction refinement works better than only refinement, but remains slightly worse than h_b in most domains. Overall, h_b performs best in 14 domains, h_r in 3 domains, and h_{r+b} in 10 domains.

At larger time intervals of 0.1 seconds, the results for our abstraction refinement techniques improve as the reduced lookahead is compensated better by an improved heuristic through abstraction refinement. Here, h_{r+b} is the best performing configuration overall with a total coverage of 1067. It is best in 14 domains, compared to 9 and 9 domains for h_b and h_r respectively. The biggest advantages over h_b are

Coverage	h_b	h_r	h_{r+b}	h_b	h_r	h_{r+b}	**Gen.**
Time Steps	0.01s			0.1s			
Agricola (20)	**1**	0	0	**9**	4	4	0.00
Airport (50)	**21**	19	**21**	**27**	24	**27**	0.31
Barman (34)	**0**	**0**	**0**	0	**2**	0	–
Blocks (35)	16	11	**21**	17	14	**19**	0.45
Childsnack (20)	**0**	**0**	**0**	**2**	0	1	–
DataNetwork (20)	**10**	9	**10**	**12**	**12**	**12**	0.40
Depots (22)	8	5	**11**	10	9	**18**	0.52
Driverlog (20)	**16**	13	**16**	17	15	**20**	0.82
Elevators (50)	**29**	10	20	**34**	27	33	0.72
Floortile (40)	**0**	**0**	**0**	**0**	**0**	**0**	–
Freecell (80)	**69**	19	52	**76**	36	73	0.31
GED (20)	7	5	**9**	10	11	**12**	0.16
Grid (5)	2	2	**5**	**3**	**3**	**3**	0.63
Gripper (20)	**20**	19	**20**	**20**	19	**20**	1.00
Hiking (20)	**13**	7	**13**	14	**20**	**20**	0.54
Logistics (63)	36	**41**	37	42	**52**	**52**	0.89
Miconic (150)	**150**	115	**150**	**150**	**150**	**150**	0.96
Mprime (35)	17	**25**	20	21	**30**	27	0.48
Mystery (30)	**11**	**11**	**11**	14	**16**	15	0.14
Nomystery (20)	6	**10**	6	**10**	**10**	9	0.49
Openstacks (100)	**76**	**76**	72	75	**80**	76	0.01
OrgSynth (20)	**7**	**7**	**7**	**7**	**7**	**7**	–
OrgSynth-Split (20)	**1**	**1**	**1**	**1**	**1**	**1**	–
Parcprinter (50)	12	**25**	11	17	**27**	13	0.50
Parking (40)	**0**	**0**	**0**	**0**	**0**	**0**	–
Pathways (30)	**4**	**4**	2	**4**	**4**	**4**	–
Pegsol (50)	**10**	8	8	21	**23**	13	0.25
PetriNetAlign (20)	0	0	**1**	**0**	**0**	**0**	–
Pipes-NT (50)	**33**	9	25	36	22	**42**	0.45
Pipes-T (50)	11	6	**12**	15	12	**20**	0.34
PSR (50)	48	47	**49**	47	**48**	47	0.84
Rovers (40)	**20**	19	19	**33**	21	25	0.85
Satellite (36)	**7**	6	**7**	8	9	**12**	0.58
Scanalyzer (50)	**21**	16	19	22	29	**43**	0.64
Snake (20)	**18**	15	9	**20**	18	15	0.69
Sokoban (50)	11	7	**13**	15	11	**21**	–
Spider (20)	9	5	6	15	10	12	0.11
Storage (30)	**19**	14	**19**	20	17	**23**	0.47
Termes (20)	0	2	**12**	1	1	**4**	0.86
Tetris (17)	**14**	7	12	14	9	**16**	0.61
Tidybot (40)	**39**	12	32	**40**	28	37	0.11
TPP (30)	**26**	13	20	**23**	21	12	0.49
Transport (70)	16	11	**24**	22	17	**32**	0.45
Trucks (30)	**2**	**2**	1	3	**5**	2	0.99
Visitall (40)	**40**	35	**40**	**40**	39	**40**	0.90
Woodworking (50)	11	**24**	12	15	**28**	15	0.73
Zenotravel (20)	**17**	13	14	**20**	**20**	**20**	0.71
Sum (1797)	**904**	705	869	1022	961	**1067**	0.54
#best	14	3	10	9	9	14	
GAT (s)	0.21	0.29	0.26	2.59	2.85	3.13	

Table 1: Coverage on the IPC domains for time steps of 0.01 and 0.1 seconds. For each time frame size, the best value is highlighted. The lookahead ratio is set to 0.8 and 0.7 for h_r and h_{r+b} respectively. The second to last row shows the number of domains in which each configuration was the best performing one (no ties). The last row shows the geometric mean of the goal achievement time across all domains. The rightmost column shows the generalization for h_r, with the average over all domains in the last row.

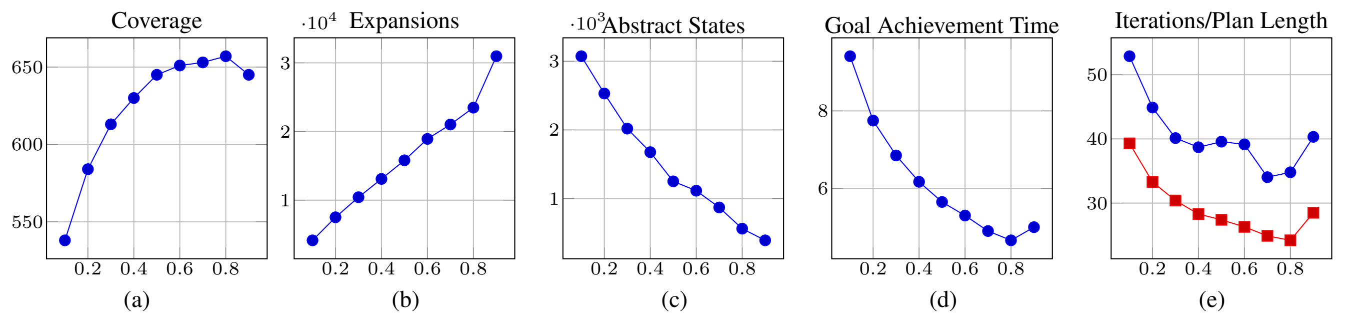

Figure 5: Results for h_r with a lookahead ratio of $l = 0.1, 0.2, \dots, 0.9$ and time intervals of 0.1s. Instances solved in the first lookahead by all configurations are omitted (328). From left to right: total coverage, expansions, total number of abstract states added during search, time to goal, number of iterations to goal (red) and plan length (blue). All values are given as the geometric mean over all commonly solved instances (except total coverage).

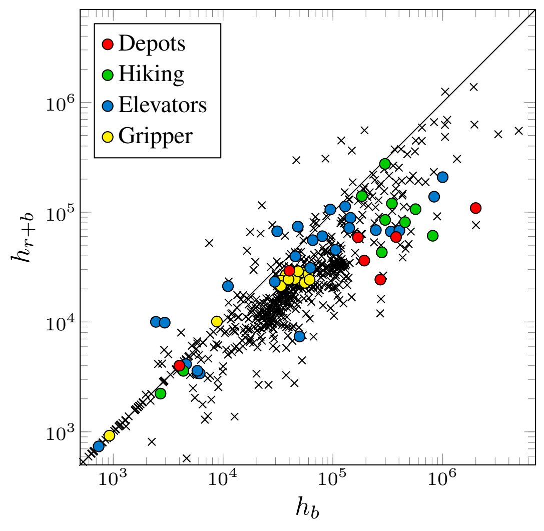

Figure 6: Number of expansions for h_b and h_{r+b} on commonly solved instances with time steps of 0.1s.

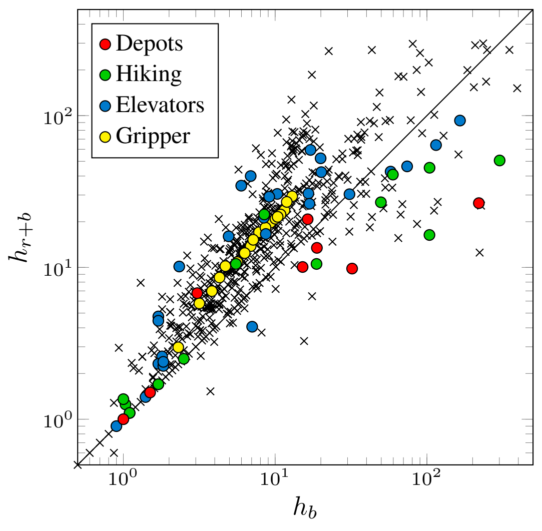

Figure 7: Goal achievement time for h_b and h_{r+b} on commonly solved instances with time steps of 0.1s.

in Scanalyzer (+14), Transport (+10), and Depots (+8); bad domains are e.g. TPP (−10), Rovers (−8), and Agricola (−5). The h_r configuration works best in Parcprinter and Woodworking, where online refinement of Cartesian abstractions also works exceptionally well in the offline planning setting (Eifler and Fickert 2018).

Expansions and Goal Achievement Time Figures 6 and 7 show the number of expansions respectively the goal achievement time on commonly solved instances of h_{r+b} compared to h_b. Selected domains are highlighted to exemplify observations, where Depots and Hiking are examples where h_{r+b} works well, and Elevators and Gripper as examples for domains where h_b is better.

Almost universally, we can observe a significant reduction in the number of expansions (Figure 6), often by more than one order of magnitude (e.g. on the larger instances of Depots and Hiking). In the Elevators domain, the abstraction refinement is not as effective, and on some instances h_{r+b} has more expansions than h_b.

The last row of Table 1 shows the mean goal achievement time over all domains. On average, the abstraction refinement approaches perform worse than Bellman updates in that regard. The biggest reason is that at the end of each time interval, we have to wait for the last refinement operation to finish, which can slightly extend the time frame (which means spending time on planning without having an action execution in parallel). For time slots of 0.1 seconds, the time frames are extended by 14% for h_r on average. This effect is amplified in domains where many merges are necessary, e.g. Gripper or Visitall. The most extreme case is Termes, where time frames are extended by almost a factor of 4. With smaller time steps of 0.01 seconds this effect is also more pronounced, extending time frames by 23% on average.

More detailed statistics for the goal achievement time are shown in Figure 7. In most domains, h_b has a better goal achievement time than h_{r+b}, though e.g. in the Depots domain, h_{r+b} performs better. This is more pronounced on larger instances, and this trend can be observed in other domains, too (e.g. Hiking, and Elevators to some degree).

Generalization The biggest advantage of abstraction refinement over Bellman updates is that the learned information in one lookahead phase results in a more informed heuristic even on states not seen so far. We measure the generalization by the fraction of states that, when first encountered in the search, have a heuristic value different to the original (un-refined) estimate. We only collect this statistic starting at the second iteration, i.e. after the first refinement.

The rightmost column in Table 1 shows this for each domain (average over all solved instances). Across all domains, on average 54% of newly visited states have a different heuristic value compared to the orignal heuristic. The best generalization can be observed in Gripper, Trucks, and Miconic, with a generalization very close to 1. On the other hand, on Agricola and Openstacks, the abstraction refinement does not translate to a more informed heuristic on states not seen before.

Compactness Finally, we compare the compactness of the abstraction heuristic to that of the lookup table used for Bellman updates. In order to make a fair comparison, we consider the abstraction refinement variant that simulates Bellman updates, and use a heuristic with only one abstraction for both configurations. We compare the number of abstract states to the number of entries in the lookup table on commonly solved instances (see Figure 8).

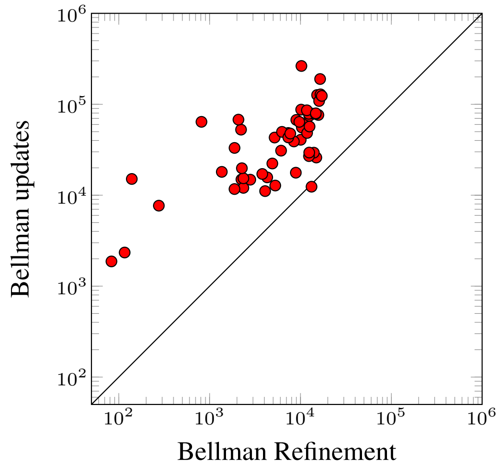

Figure 8: Compactness: number of entries in the lookup table for Bellman updates compared to the number of abstract states added during search when simulating Bellman updates through abstraction refinement.

The number of abstract states is almost always lower than the number of entries in the lookup table. In the most extreme cases, the advantage is more than three orders of magnitude. While the memory required to store an abstract state is slightly more than it takes to store an entry in the lookup table, this difference is only constant.

On the other hand, each refinement operation is very costly, and many refinement operations are required to simulate Bellman updates, leading to a detrimental trade-off in runtime and non-competitive performance on the IPC benchmarks (333 and 328 overall coverage with time steps of 0.01 and 0.1 seconds respectively).

15-Tile Puzzle

We generated a benchmark set for the 15-Puzzle by performing random walks of length up to 100 (in increments of 10) backwards from the goal, and then grouping the instances by optimal plan length.

The results for time steps of 0.1 and 1 seconds are shown in Figures 9 and 10 respectively. While the results for all three configurations are close, h_r tends to have a higher coverage than the other two, especially on larger instances. The biggest advantage over Bellman updates is at an optimal solution length of 18 for time steps of 0.1 seconds (0.57 vs. 0.46), and 24 and 26 for time steps of 1 second (0.45 vs. 0.30 and 0.28 vs. 0.15 respectively).

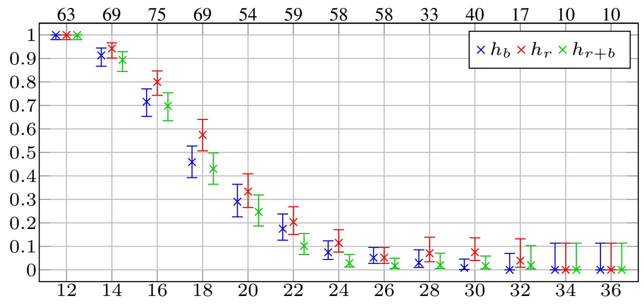

Figure 9: Coverage results as fraction of solved instances on the 15-Puzzle with time steps of 0.1 seconds. The instances are grouped by optimal plan length on the x-axis, the number of instances for each plan length is depicted at the top of the plot. The y-axis shows the fraction of solved instances with asymmetric confidence intervals of 95%.

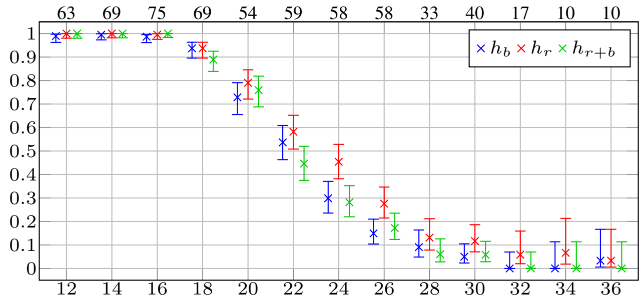

Figure 10: Coverage results as fraction of solved instances on the 15-Puzzle with time steps of 1 second.

Conclusion

Completeness in real-time search requires learning techniques to avoid infinite looping behavior. While the traditional learning method on discrete domains are per-state updates, we observe that one can leverage known abstraction refinement methods instead. Our initial exploration of this idea reveals that it has potential. Much remains to be done to fully understand the technique and its empirical implications. Apart from better understanding its strengths and weaknesses relative to traditional methods, an interesting line of research are more radical adaptations of abstraction refinement, re-designing them to interact more deeply with the information provided by lookahead searches and Bellman state updates.

Acknowledgments. This work is supported by the Air Force Office of Scientific Research under award number FA9550-18-1-0245, and by the the German Research Foundation (DFG) under grant HO 2169/5-1, "Critically Constrained Planning via Partial Delete Relaxation".

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS$^+$ planning. *Computational Intelligence* 11(4):625–655.

Ball, T.; Podelski, A.; and Rajamani, S. K. 2001. Boolean and cartesian abstraction for model checking C programs. In Margaria, T., and Yi, W., eds., *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, 268–283. Springer.

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

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Boyan, J. A., and Moore, A. W. 1995. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in neural information processing systems*, 369–376.

Bulitko, V., and Lee, G. 2006. Learning in real-time search: A unifying framework. *Journal of Artificial Intelligence Research* 25:119–157.

Bulitko, V.; Björnsson, Y.; Sturtevant, N. R.; and Lawrence, R. 2011. Real-time heuristic search for pathfinding in video games. In *Artificial Intelligence for Computer Games*. Springer. 1–30.

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

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

Dräger, K.; Finkbeiner, B.; and Podelski, A. 2006. Directed model checking with distance-preserving abstractions. In Valmari, A., ed., *Proceedings of the 13th International SPIN Workshop (SPIN 2006)*, volume 3925 of *Lecture Notes in Computer Science*, 19–34. Springer-Verlag.

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

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

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

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In Howe, A., and Holte, R. C., eds., *Proceedings of the 22nd National Conference of the American Association for Artificial Intelligence (AAAI'07)*, 1007–1012. Vancouver, BC, Canada: AAAI Press.

Helmert, M.; Haslum, P.; Hoffmann, J.; and 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).

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

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

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173:503–535.

Hernández, C., and Baier, J. A. 2012. Avoiding and escaping depressions in real-time heuristic search. *Journal of Artificial Intelligence Research* 43:523–570.

Katz, M., and Domshlak, C. 2008. Optimal additive composition of abstraction-based admissible heuristics. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E., eds., *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS'08)*, 174–181. AAAI Press.

Kiesel, S.; Burns, E.; and Ruml, W. 2015. Achieving goals quickly using real-time search: experimental results in video games. *Journal of Artificial Intelligence Research* 54:123–158.

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

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

Lagoudakis, M. G., and Parr, R. 2003. Least-squares policy iteration. *Journal of Machine Learning Research* 4:1107–1149.

Larsen, B.; Burns, E.; Ruml, W.; and Holte, R. C. 2010. Searching without a heuristic: Efficient use of abstraction. In *Proceedings of AAAI-10*.

Seipp, J., and Helmert, M. 2013. Counterexample-guided Cartesian abstraction refinement. In Borrajo, D.; Fratini, S.; Kambhampati, S.; and Oddi, A., eds., *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS'13)*, 347–351. Rome, Italy: AAAI Press.

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

Seipp, J., and Helmert, M. 2018. Counterexample-guided cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research* 62:535–577.

Seipp, J. 2017. Better orders for saturated cost partitioning in optimal classical planning. In Fukunaga, A., and Kishimoto, A., eds., *Proceedings of the 10th Annual Symposium on Combinatorial Search (SOCS'17)*. AAAI Press.