UNIVERSITÄT BASEL

Action Pruning Through Under-approximation Refinement

Master's Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Martin Wehrle

Manuel Heusner

August 1$^{\text{th}}$, 2013

Abstract

Planning as heuristic search is the prevalent technique to solve planning problems of any kind of domains. Heuristics estimate distances to goal states in order to guide a search through large state spaces. However, this guidance is often moderate, since still a lot of states lie on plateaus of equally prioritized states in the search topology. Additional techniques that ignore or prefer some actions for solving a problem are successful to support a search in such situations. Nevertheless, some action pruning techniques lead to incomplete searches.

We propose an *under-approximation refinement* framework for adding actions to under-approximations of planning tasks during a search in order to find a plan. For this framework, we develop a refinement strategy. Starting a search on an initial under-approximation of a planning task, the strategy adds actions determined at states close to a goal, whenever the search does not progress towards a goal, until a plan is found. Key elements of this strategy consider helpful actions and relaxed plans for refinements.

We have implemented the under-approximation refinement framework into the greedy best first search algorithm. Our results show considerable speedups for many classical planning problems. Moreover, we are able to plan with far less actions than standard greedy best first search.

Contents

Chapter 1

Introduction

Classical planning is the problem of finding a plan from an initial state to a goal state with some given actions. For human beings, solving problems can be a challenging and time consuming task. To save time, domain-specific solvers that can be run on a computer were developed for specific kinds of problems. These solvers work quite good as they take advantage of domain specific knowledge provided by the user. This knowledge allows for developing e.g. accurate heuristics and action pruning techniques that guide searches through large state spaces. However, implementing a solver for each domain is costly, which has called for planning systems that are able to solve problems of arbitrary domains. The challenge of developing a domain-independent planner based on heuristic search is to find general heuristics and action pruning techniques that work well for a considerable amount of domains.

A lot of search heuristics have been developed for heuristic-forward search planners over the past years. Although they calculate almost perfect heuristic estimates, a search is still confronted to a lot of states having the same heuristic value. We can avoid many of these states by pruning actions. In fact, most planning problems contain a vast amount of actions that are not used for an optimal plan. Table 1.1 shows the relation between available actions and actions included in a plan of some planning tasks that were used in last IPC challenges. We observe that some tasks require less than 1% of available actions to build an optimal plan.

Some action pruning techniques for satisficing planning eliminate actions previously to a search [10, 5] (static action pruning). Helpful actions, a method implemented into the FF planner [9], prune actions locally (i.e. transitions) during a search (dynamic action pruning). While the technique introduced by Haslum and Peter Jonsson [5] is computational costly and keeps a lot of actions, the method of Hoffmann and Nebel [9] and the method of Nebel, Dimopoulos and Koehler[10] are more efficient but do not preserve the completeness of an underlying search algorithm. Planners applying these incomplete techniques simply restart a search if the first search did not find a solution. This second search runs without applying a pruning technique.

However, it is possible to refine the search by adding new actions to the search instead of doing a restart. This thesis provides an approach for refining under-approximations of a planning task in order to find a plan. An under-approximated planning task is defined by being a planning task that has a

Task	# actions in task	# actions in plan
sokoban-sat11-strips-p09.pddl	464	429
sokoban-sat11-strips-p16.pddl	1496	47
schedule-probschedule-2-0.pddl	49	2
schedule-probschedule-12-0.pddl	289	13
logistics98-prob01.pddl	360	26
logistics98-prob35.pddl	676	30
scanalyzer-sat11-strips-p01.pddl	32736	10
scanalyzer-sat11-strips-p02.pddl	9504	12
mystery-prob01.pddl	151	5
mystery-prob10.pddl	36738	8

Table 1.1: Contrast between the number of all actions and the number of actions used in an optimal plan.

reduced set of actions. The idea of our approach is similar to *counterexample-guided abstraction refinement* (CEGAR) [3], which was adapted to classical planning by Seipp and Helmert [13] for calculating heuristic estimates. They approximate plans by iteratively refining an over-approximation of a planning task. Each refinement intends to improve the approximated plan in order to provide accurate heuristic estimates. Similarly, our approach refines under-approximations of a planning task. We search in an under-approximated planning task and iteratively add actions to the under-approximation if there is no progress of the search towards a goal. Table 1.1 shows that some tasks require a fraction of available actions. Therefore, if we are able to add suitable actions, we could save a lot of search effort.

We provide a general framework for under-approximation refinement (UAR) which is built upon the greedy best-first search algorithm. This framework allows employing static action pruning techniques, as well as strategies for refining under-approximated planning tasks. For this framework, we provide a refinement strategy that considers helpful actions and relaxed plans of the FF heuristic [9]. The strategy is called *best states under-approximation refinement* (BSUAR) strategy.

This thesis is organized as explained next. After showing the notation and presenting the search algorithm, on which our approach is based, we will define under-approximated planning tasks and provide the general framework of under-approximation refinement. Afterwards, BSUAR, a refinement strategy for the framework, will be introduced including some variations. We will continue by comparing main settings of BSUAR to state-of-the-art planning configurations. Furthermore, we will evaluate variations of BSUAR to one another. In the end of this thesis, we will discuss some pruning techniques in relation to under-approximation refinement. Finally, we will summarize the results of this thesis and suggest further possibilities in connection to our under-approximation refinement framework.

Chapter 2

Preliminaries

This chapter defines the notation that is used throughout the paper. Furthermore, it presents planning as heuristic search by showing a specific heuristic as well as a specific search algorithm. In the last section, we will introduce a refinement approach used in another context but entailing the idea for our approach.

2.1 Notation

This work considers planning tasks formalized in a *SAS*$^+$-like [1] finite-domain representation.

Definition 2.1 (planning task)**.** *A planning task is a 4-tuple* $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}\rangle$*:*

- $\mathcal{V}$ *is a finite set of* ***state variables*** v *which have finite domains* $D(v)$*. An* ***atom*** *is a pair* $\langle v, d\rangle$ *with* $v \in \mathcal{V}$ *and* $d \in D(v)$*. A* ***state*** $\{\langle v, d\rangle \mid v \in \mathcal{V}\}$ *is a set that contains exactly one atom for each* $v \in \mathcal{V}$*. A* ***partial state*** $\{\langle v, d\rangle \mid v \in \mathcal{V}_s \subset \mathcal{V}\}$ *is a set that contains exactly one atom for a subset of* $\mathcal{V}$*.* $\mathcal{S}(\Pi)$ *is the set of all states of* Π.
- s_0 *is the* ***initial state****.*
- $S_\star$ *is a partial state defining the* ***goal atoms****.*
- $\mathcal{A}$ *is a finite set of* ***actions****. An action is a triple* $a = \langle \mathit{pre}(a), \mathit{eff}(a), \mathit{cost}(a)\rangle$*:*
  - $\mathit{pre}(a)$ *is a partial state defining the* ***preconditions****.*
  - $\mathit{eff}(a)$ *is a partial state defining the* ***effects****.*
  - $\mathit{cost}(a) \in \mathbb{R}$ *is the* ***cost*** *of applying a*

  Actions are also called operators*.*

An action a is *applicable* in state s if s contains the preconditions of a. If a is applicable in s, the *transition function* $\mathit{result}(a, s)$ replaces those atoms of s with atoms of $\mathit{eff}(a)$ which include the same variable. In this case, the function produces a new state s' which is called the successor state of s. If a is not applicable in s, then $\mathit{result}(a, s)$ is undefined.

A planning task induces a *state space.* This thesis considers a state space similar to Bonet and Geffner's definition of state spaces [2].

Definition 2.2 (state space). *A* state space *is a tuple* $\mathrm{S} = \langle S, s_0, S_\star, A, T, c \rangle$*:*

- *S is a finite set of states s.*
- *$s_0 \in S$ is the initial state.*
- *$S_\star \subseteq S$ is a set of goal states.*
- *$A(s) \subseteq A$ denotes actions applicable in a state s of S.*
- *$T(a, s)$ denotes a transition function defined for all states s of S and actions a of $A(s)$.*
- *$c(a)$ is the cost function defined for each action a.*

A planning task $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A} \rangle$ defines a state space S_Π with states $\mathcal{S}(\Pi)$, initial state s_0, goal states $\{s \in \mathcal{S}(\Pi) \mid s_\star \subseteq s\}$, actions $\mathcal{A}$, transition function $\mathit{result}(a, s)$ and cost function $\mathit{cost}(a)$.

A problem is formalized in a planning task in order to be solved. To solve a planning task means to find a *plan* in its state space.

Definition 2.3 (plan). *Given a state space of a planning task, a* plan *is a sequence $\pi = a_0 \dots a_n$ of actions in A. Moreover, to be a valid plan for the planning task, π must generate a path of transitions in the state space that starts in s_0 and ends in a goal state $s_\star$ of $S_\star$.*

Searching for a plan in a state space describes what is called planning. A plan is optimal, if the cost of plan π is minimal. The cost of a plan π is the sum of each action cost in π. While *optimal planning* intends to find optimal plans, *satisficing planning* prefers to find cheap plans but also allows to find any plan.

Usually, planners start exploring the state space from s_0. The part of the state space that can be reached by any transition path starting in s_0 is called *reachable state space.*

2.2 Planning as Heuristic Search

A search-based planner systematically explores the state space of a planning task in order to find a path to a goal state. In addition, planning as heuristic search considers heuristics to guide the search towards a goal. In this section, we will first consider a heuristic based search algorithm. Afterwards, we will present an established search heuristic.

2.2.1 Greedy Best-first Search

Greedy best-first search (GBFS) [12] is an informed search algorithm. It searches for a goal state by repeatedly expanding states with the smallest heuristic value. In contrast to $A^\star$ [4], GBFS ignores the cost from the initial state to the current state. Therefore, it mostly returns suboptimal plans instead of optimal plans.

A *heuristic* $h(s)$ is a function that evaluates a state s and returns an estimated distance to a goal state. Where it is clear from the context, we use the term *heuristic* to denote the returning value. Moreover, the minimal heuristic value among evaluated states is called best heuristic value. In the next section, we will consider a specific heuristic.

Algorithm 2.1 shows the pseudocode of GBFS as it is used in this thesis. In contrast to literature, this pseudocode only shows states instead of nodes because GBFS is presented as a graph search algorithm and state related data does never change.

Algorithm 2.1: Greedy Best-first Search

```
Data: $Open \leftarrow$ set of open states (open list), initially empty
$Closed \leftarrow$ set of closed states (closed list), initially empty
1  $Open \leftarrow s_{init}$;
2  while $true$ do
3      if $Open = \emptyset$ then
4          return $no\ plan\ exists$;
5      $s \leftarrow$ get $s$ from $Open$ with minimal $h(s)$;
6      $Open \leftarrow Open \setminus \{s\}$;
7      if $isGoal(s)$ then
8          return $plan$;
9      $Closed \leftarrow Closed \cup \{s\}$;
10     $S \leftarrow expand(s)$;
11     $S_{new} \leftarrow S \setminus (Closed \cup Open)$;
12     $Open \leftarrow Open \cup S_{new}$;
```

The algorithm starts by inserting the initial state into the empty open list (line 1). The open list, also called frontier, contains unexpanded states. In each iteration of the algorithm, a state having the best heuristic is removed from the open list (line 5–6). This state is assumed to be nearest to a goal state. If this state itself is a goal state, the algorithm stops and returns a plan (line 7–8). Otherwise, the state is inserted into the closed list and is expanded (line 9–10). An expansion generates the successors of the current state by applying actions that are applicable in this state. Only new states are inserted into the open list (line 11–12). The algorithm proceeds the search until the open list is empty, being a signal of the explored search space. In this case, no solution can be found and the algorithm stops (line 3–4).

We now know an informed search algorithm for satisficing planning. The next section shows an heuristic that can be used for GBFS.

2.2.2 The Fast Forward Heuristic

The Fast Forward (FF) heuristic, denoted with h^{FF}, was introduced by Hoffmann and Nebel [9]. It explores a *relaxed planning task* in order to effectively find a *relaxed plan*. This plan does not map directly to a plan of the original planning task. Nevertheless, the cost of this plan is a good estimation of the distance to a goal state.

A relaxed planning task, derived from a planning task, ignores *negative interactions* of actions, i.e. actions applicable in a current state remain applicable in all succeeding states. In our notation, it means that the transition function $result(s, a)$ leaves all atoms in a state while atoms of $eff(a)$ are added. Therefore, state variables can be assigned to multiple values. As a consequence, succeeding states contain an increasing amount of atoms while more actions become applicable. Thus, the task can be solved more efficiently.

The FF heuristic applied to a state generates a relaxed planning task with the current state being the initial state. After the FF heuristic has found a relaxed plan, the cost of actions in the plan are cumulated resulting in a heuristic estimate for this state.

In addition, the relaxed plan contains actions being considered as helpful. *Helpful actions* are those actions of the relaxed plan that are applicable at the current state. A search algorithm that preferably applies these actions can find a plan without considering a lot of other actions.

2.3 Counterexample-guided Abstraction Refinement

The approach presented in this thesis is based on the idea of counterexample-guided abstraction refinement (CEGAR) [3] in the context of classical planning introduced by Seipp and Helmert [13]

In the context of classical planning, CEGAR iteratively refines Cartesian abstractions of transition systems. An abstraction ignores some values of state variables. In contrast to other abstractions, the abstract transition system of CEGAR allows different levels of granularity.

In more detail, CEGAR starts with a first abstraction of a planning task. It searches for an optimal plan on the abstraction and tests the plan on the concrete planning task. If the plan does not reach a goal state (this plan is a counterexample), an algorithm determines why it failed and refines the abstraction in order to avoid the same failure in future iterations. For refinining the abstraction, the algorithm splits the abstract state that corresponds to the concrete state at which the plan failed into two abstract states. Afterwards, a new search is performed on the refined transition system. This procedure continues until an abstract plan is a valid concrete solution. As Seipp and Helmert intend to use CEGAR for producing heuristics, the algorithm can be interrupted after a time or memory limit.

Chapter 3

Under-approximation Refinement: The general Framework

To reduce the complexity of planning tasks, one possibility is to remove actions from planning tasks. Depending on a pruning method, "elementary" actions for building a plan could also be removed from the task. However, as a consequence, a corresponding under-approximation of the original task might not be solvable any more. Therefore, the idea of this work is to refine an incomplete task by actions from the original task in order to find a plan. We call this approach *under-approximation refinement* (UAR). This approach was inspired by CEGAR [3, 13] but using under-approximations instead of over-approximations is the central difference. As a consequence, UAR has the property that if a plan is found, this plan corresponds to a real plan because any action sequence applicable in the under-approximation is applicable in the original task as well. Our algorithm essentially consists of following steps:

1. Select a first under-approximation of a planning task Π^i; $i := 0$.
2. Search for a plan on Π^i
3. If a plan is found on Π^i, return the *plan*
4. If the *refinement guard* triggers:
    (a) Apply the *refinement method* to determine Π^{i+1}, i.e. add some missing actions back to the task.
    (b) Go to point two.

In the above description, the refinement guard describes *when* to refine while the refinement method describes *which* actions to introduce to the UA. Obviously, the guard should trigger the refinement method if the reachable state space is completely explored. In this thesis, we will investigate more sophisticated ways.

After introducing definitions related to under-approximated planning tasks, a framework for under-approximation refinements will be embedded into the greedy search algorithm.

3.1 Under-approximated Planning Task

This thesis considers under-approximations (UA) of a planning task. An UA planning task is defined as follows.

Definition 3.1 (under-approximated planning task)**.** *Given a planning task* $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A} \rangle$*, an under-approximated planning task* $\Pi' = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}' \rangle$ *is derived from the planning task* Π*, where* $\mathcal{A}' \subset \mathcal{A}$*.*

In other words, an UA planning task excludes actions of the original planning task. The exclusion of actions can reduce the state space drastically.

Note that in the context of UA planning task, optimal plans can be understood differently. An optimal plan of an UA planning task can be a satisfying plan of the original planning task. For the rest of this thesis, optimal plans refer to those ones of original planning tasks.

The part of the pruned state space including the initial state is called reachable UA state space. States beyond the reachable UA state space cannot be considered for a plan. If all goal states lie beyond this space, it is impossible to find a plan from the initial state to a goal state. In this case, refinements of UA planning tasks are required.

Definition 3.2 (refined task)**.** *Given an UA planning task* $\Pi' = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}' \rangle$ *and its originating task* $\Pi_{orig} = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}_{orig} \rangle$, $\Pi'' = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}'' \rangle$ *is a refined task of* Π' *with* $A' \subset \mathcal{A}''$ *and* $\mathcal{A}'' \subseteq \mathcal{A}_{orig}$*.*

A refinement of an UA planning task can yield to the original planning task, when all actions of the original planning task have been inserted to the UA planning task.

The definitions related to UA planning tasks allow to integrate UAR into the GBFS.

3.2 GBFS with Under-approximation Refinement

We extend the GBFS algorithm, as it was presented in Algorithm 2.1, to under-approximate an original planning task and to iteratively refine it in order to find a plan.

GBFS explores the state space of a planning task in order to find a satisficing plan. We use GBFS, as in the context of UAR, optimality cannot be achieved. We show it by following example: We assume an UA planning task with actions that correspond to a satisfying plan but not to an optimal plan. An optimal search strategy needs the task to be refined while a satisfying search is able to find the plan. Moreover, determining the missing actions of an optimal plan would be hard. Even if an UA planning task lacks a plan, completing a satisfying plan is much easier than completing an optimal one.

The framework of UAR interacts with the search algorithm at two different locations. The creation of the initial UA planning task precedes the search, whereas the search itself includes a *refinement strategy.* The initial task can be created by a static action pruning method. Those methods usually prune actions form a planning task previously to a search.

Definition 3.3 (refinement strategy)**.** *A* refinement strategy *consists of a* refinement guard *and a* refinement method*.*

The refinement guard *returns true iff the UA is refined based on the refinement strategy. Otherwise, the guard returns false.*

Given an UA planning task $\Pi' = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}' \rangle$*, the* refinement method *RA determines actions* $\mathcal{A}'' := RA(\mathcal{A}')$ *of the (next) refined task* $\Pi'' = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}'' \rangle$*.*

Π'' *is a* proper refinement *if* $\mathcal{A}' \subset \mathcal{A}''$*.*

Algorithm 3.1 shows the pseudocode of the UAR framework implemented into the GBFS algorithm. The depicted algorithm can be implemented in a more sophisticated and efficient way than shown in the pseudocode. For a better comprehension, the pseudocode is kept as simple as possible. The implementation details will be presented in Section 5.1.

Algorithm 3.1: GBFS with Under-approximation Refinement

```
Data: $Open \leftarrow$ set of open states (open list), initially empty
$Closed \leftarrow$ set of closed states (open list), initially empty
$A_{all} \leftarrow$ set of actions of the original planning task
$UA \leftarrow$ set of actions of the under-approximated planning task, initially empty

1  $UA \leftarrow createInitialUA(A_{all})$;
   $Open \leftarrow s_{init}$;
   while $true$ do
4      if ($RefinementStrategy.guard() = true \lor Open = \emptyset$) then
5          $A_{new} \leftarrow RefinementStrategy.refine(UA)$;
6          $UA \leftarrow UA \cup A_{new}$;
7          foreach $a \in A_{new}$ do
8              foreach $s \in Closed$ do
9                  if $isApplicable(a, s)$ then
10                     $Open \leftarrow Open \cup \{s\}$;          /* re-opening */
       if $Open = \emptyset$ then
           return $no\ plan\ exists$;
       $s \leftarrow$ get $s$ from $Open$ with minimal $h(s)$;
       $Open \leftarrow Open \setminus \{s\}$;
       if $isGoal(s)$ then
           return $plan$;
       $Closed \leftarrow Closed \cup \{s\}$;
18     $S \leftarrow expand(s, UA)$;          /* applies actions from UA */
       $S_{new} \leftarrow S \setminus (Closed \cup Open)$;
       $Open \leftarrow Open \cup S_{new}$;
```

Initially, a first UA planning task is created (line 1). This task builds the basis for the search, which can be adapted by the refinement strategy. The refinement strategy comes into action before each node expansion. Under certain refinement conditions, the refinement guard decides if the UA needs to be changed (line 4).

Afterwards, the GBFS continues the search as usual, by expanding the next state from the open list. This expansion only applies actions of UA (line 18). When the search is not able to generate all successor states of an expanding state due to the UA task, then we call this state a *partly expanded state*.

Otherwise, if the guard admits a refinement, the strategy starts its refinement method, possibly determining supplementary actions (line 5). If the refinement is proper, i.e. the refinement method determined additional actions, the new actions are added to the UA (line 6). Also, the state space needs to be updated by reopening all closed states where the new actions are applicable (line 7–10). Afterwards, GBFS continues searching on the refined task.

To complete the discussion of the algorithm, one fundamental refinement condition has to be considered in more detail. It demands to refine an explored UA state space. This condition is independent from the refinement strategy. Therefore, it was directly implemented into the framework (line 4). If the refinement method does not determine a proper refinement, i.e. if no additional actions are introduced to UA, no re-openings will happen.

However, a complete search algorithms must always return a solution if one exists (otherwise the algorithm is not complete). Given the GBFS algorithm with UAR, it can be shown that its completeness strongly depends on the refinement strategy. Therefore, we define a completeness-preserving property for refinement strategies.

Definition 3.4 (completeness-preserving refinement strategy)**.** *Given a completely explored reachable UA state space, the refinement method of a* completeness-preserving refinement strategy *must provide at least one new action which is applicable in a partly expanded state.*

In the situation of a completely explored reachable UA state space all partly expanded states are in the closed list.

Theorem 3.1. *GBFS with UAR is complete, given a completeness-preserving refinement strategy.*

Proof. To be complete, the GBFS with UAR must find a solution if one exists. Otherwise it must stop under the guarantee that no solution is possible.

The completeness of GBFS, implemented as a graph search, is a given for this proof [12].

To show the completeness of GBFS with UAR, it is sufficient to show that GBFS is able to visit all reachable states. To generate these states, GBFS needs to receive all actions from the refinement method of the strategy which successively generate the reachable states.

Specifically, in the situation of an explored reachable UA state space, GBFS requires a new action that is applicable in one of its closed states. A completeness-preserving refinement strategy is guaranteed to provide this action in this situation.

Moreover, the reopening of states, where this new action is applicable, allows the GBFS to proceed the search by using the recently added actions. □

Now, we can use this framework to develop and to evaluate the refinement strategy, which will be presented in the next section.

Chapter 4

Refinement Strategy

In the general context of the framework, the emphasis is now put on a specific refinement strategy. A refinement strategy, being the main building block of UAR, intends to occasionally refine UA planning tasks during a search. A strategy consists of a *refinement guard* and a *refinement method*; the guard describes *when* to refine whereas the method describes *which* actions to add. The strategy developed in this work, we call it *best states under-approximation refinement strategy* (BSUAR), essentially includes following ideas:

- Refine an UA planning task whenever the search does not progress towards a goal state. (refinement guard).
- Add actions, which are determined in states that are as close as possible to a goal. We call those states *best states.* (refinement method).

Note that both ideas can be formalized by using search heuristics.

The refinement method itself consists of two components: a state selection strategy and an action selection strategy. While the former strategy chooses a subset of best states, the latter strategy is the most important component of BSUAR, as it determines actions to be added to the current UA.

We will, first, define the refinement guard and the refinement method. Afterwards, we will introduce some state and action selection strategies. As some of the action selection strategies are not completeness-preserving, we propose a setting in the last section, which preserves the completeness of GBFS.

4.1 Refinement Guard

A refinement guard determines stages of a search where an adaption of a UA planning task seems to be useful. The condition to refine an explored UA planning task is already implemented into the UAR framework. Yet, a refinement guard can provide more sophisticated conditions.

We intend to formulate a guard that triggers the refinement method whenever the search does not proceed towards a goal. We can detect such a situation by observing the search progress. The search progress can be derived from the evolution of heuristic values given by states that are fetched from the open list.

The refinement guard of BSUAR compares two heuristic values: the value of the last expanded state and the best heuristic value among states in the open list.

The heuristic value of the state is denoted by h_{last}, the latter value is denoted by h_{open}. Given these two heuristic values, the guard observes a decreasing search progress if $h_{last} < h_{open}$, i.e. the heuristic value increases, which means that the search is assumed get further away from a goal. Although $h_{last} < h_{open}$ seems to be a reasonable condition, it does not take heuristic plateaus of the open list into account. Heuristic plateaus are layers of states having the same heuristic value. In other words, these states are assumed to have all the same distance to a goal state. Defining $h_{last} = h_{open}$ as an additional refinement condition enables the search to escape earlier from a plateau. Now, both conditions can be merged into one expression: $h_{last} \leq h_{open}$. The refinement guard of BSUAR uses the conditions encoded in this expression to trigger the refinement method.

4.2 Refinement Method

After a search passes the refinement guard, a refinement method determines new actions in order to refine a task. We intend to define a refinement method for BSUAR, which determines new actions at states that are assumed to be close to a goal state. Actions, determined at those states might be relevant for the proceeding search.

In more detail, we determine new actions by using partly expanded states. These states are stored on layers; each layer containing states having the same heuristic value. A layer, therefore, has the value of their including states. Furthermore, the layers are sorted by heuristic values. Starting with the layer of the best heuristic value, the refinement method searches for the first layer that provides new actions.

The method can follow a strategy for selecting new actions. The strategy consists of two single strategies: a state selection strategy and an action selection strategy.

Definition 4.1 (state selection strategy)**.** *A* state selection strategy *determines a subset of states among given states, at which an action selection strategy has selected actions for the refined task.*

Definition 4.2 (action selection strategy)**.** *An* action selection strategy *determines new actions for the refined task by using a given state.*

Specific state and action selection strategies will be introduced in Section 4.3.

With Definitions 4.1 and 4.2, it is now possible to understand the pseudocode of the refinement algorithm of BSUAR depicted in Algorithm 4.1. For the ease of understanding, possible optimizations are omitted in the pseudocode and will be shown in Section 5.1. In the pseudocode, function *selectActions* consist of a state selection strategy and an action selection strategy. This function is responsible to only return actions that are not jet included into the current UA planning task.

Algorithm 4.1: Best States Refinement Method

```
Data: $P \leftarrow$ set of $S_h$, initialized with partly expanded states
$S_h \leftarrow$ set of states with the same heuristic value of $h$
$UA \leftarrow$ set of actions of the under-approximated planning task
$explored \leftarrow true$ if $Open = \emptyset$, otherwise $false$
1  while $P \neq \emptyset$ do
2      $S_h \leftarrow$ get $S_h$ from $P$ with minimal $h$;
3      $P \leftarrow P \setminus \{S_h\}$;
4      $A_{new} \leftarrow selectActions(S_h)$;
5      if $A_{new} \neq \emptyset$ then
6          if $explored = false$ then
7              return $A_{new}$;
8          else /* explored UA state space; consider applicable
           actions in order to preserve completeness */
9              foreach $a \in A_{new}$ do
10                 foreach $s \in S_h$ do
11                     if $isApplicable(a, s)$ then
12                         return $A_{new}$;
13 return $\emptyset$
```

The refinement method of BSUAR operates on all partly expanded states, which are stored in P. Set P consists of layers S_h of states having the same heuristic value h. The method iteratively removes S_h from P whereas h has the smallest value in P (line 2–3). The function *selectActions* applied to the removed layer S_h selects actions according to a state selection strategy and action selection strategy (line 4). If it determines new actions, the method stops by returning them (line 5–7). In this case, the refinement is proper. Otherwise, the method proceeds by removing the next S_h. If it is not able to eventually determine new actions at the given layers by following the strategies, the method does not return any actions (line 13). This results in a non-proper refinement. However, as long as the GBFS continues, the refinement method has a supply of partly expanded states. These states give further possibilities to find new actions in future iterations.

According to Definition 3.4, to be completeness-preserving, the refinement method of BSUAR needs to return at least one new action that is applicable among partly expanded states when GBFS has explored the reachable UA state space entirely. Therefore, the refinement method handles this situation separately (line 8–12). At this point, the method needs to catch actions that are applicable in a state of the current layer. It can only find them by ignoring other non-applicable actions.

To maintain the completeness-preserving property for BSUAR, we must define completeness preserving state selection and action selection strategies. These definitions are similar to the definition of a completeness-preserving refinement strategy shown in Definition 3.4.

Definition 4.3 (completeness-preserving state selection strategy)**.** *Given a completely explored reachable UA state space, a* completeness-preserving state selection strategy *applied to each heuristic layer of expanded states must* eventually *select at least one state where an action selection method provides new actions that are applicable among partly expanded states.*

Definition 4.4 (completeness-preserving action selection strategy)**.** *Given a completely explored reachable UA state space, a* completeness-preserving action selection strategy *applied to all expanded states must* eventually *select at least one new action which is applicable among partly expanded states.*

Theorem 4.1. *BSUAR is completeness-preserving, given a completeness-preserving state selection strategy and action selection strategy.*

Proof. Given an unsolvable UA planning task, this proof deals with the situation of a completely explored reachable UA state space.

At this stage of the search, the refinement method of BSUAR has gathered all expanded states of the reachable UA state space. A completeness-preserving action selection strategy is is guaranteed to provide at least one new action which is applicable among these states. Furthermore, a completeness-preserving state selection strategy is guaranteed to select a state, at which the action selection strategy provides new actions. Consequently, the function *selectActions* will, finally, return at least one new and applicable action.

To be completeness preserving, BSUAR must return applicable actions to the calling UAR framework in the situation of a completely explored reachable UA state space. To be able to return these actions, the refinement method of BSUAR checks each action returned by *selectActions* for its applicability among states from the layer. Ignoring the actions that are not applicable allows to catch the applicable ones, which will extend the UA state space. □

Up to now, this proof guarantees the completeness of GBFS in combination with the BSUAR refinement strategy. To complete BSUAR, we define specific state and action selection strategies in the next section.

4.3 State and Action Selection Strategies

We can vary the refinement method of BSUAR in two dimensions. One dimension considers states having the same heuristic value, i.e. states that share an heuristic layer. We intend to select a subset of those states according to a strategy in order to only slightly refine UA planning tasks. The other dimension considers actions. Adding actions to an UA planning task is the main subject of UAR. The following steps show how both strategies are used together in the refinement method of BSUAR:

1. Apply an action selection strategy to each state from a heuristic layer.
2. Filter out states where the action selection strategy did not determine new actions. We call the remaining states candidates.
3. Select states from the candidates of step two by following a state selection strategy.
4. Return the actions which where determined by the action selection strategy at the selected states.

In the following sections, will define some state selection strategies as well as action selection strategies.

4.3.1 State Selection Strategies

A state selection strategy selects states among candidates, i.e. states at which an action selection strategy provides new actions. With this approach, we intend to only slightly refine UA planning tasks.

This thesis evaluates following strategies:

first Select the first state from a list of candidates.

all Select all candidates.

least Select states from candidates, where the applied action selection strategy has determined the least number of new actions.

most Select states from candidates, where the applied action selection strategy has determined the most number of new actions.

heur Select states from candidates which have the smallest heuristic value. The heuristic used by this strategy must not be the heuristic used by the search.

The first strategy intends to only slightly refine the UA planning task. The result of this strategy depends on the state ordering. In contrast to the fist strategy, the second strategy is independent of the state ordering, as all states on a layer are considered in the same iteration. This strategy might add too many actions to the UA planning task. Therefore, the last three methods systematically select a subset of states. The last method consults a heuristic to determine states that are assumed to lie closer to a goal than the other states from the same layer.

Being in the situation of having an explored UA state space, all these strategies must ignore candidates where the new actions are not applicable in order to be completeness-preserving.

Although state selection strategies are able to select a convenient subset of new actions, they need an action selection strategy which provides these actions.

4.3.2 Action Selection Strategies

The action selection strategy is the most important building block of BSUAR. This strategy is responsible for determining new actions at partly expanded states and to provide them to the state selection strategy.

In this thesis, we evaluate the three strategies listed below:

appl Select all actions which are applicable in the given state.

ha Selects the *helpful actions* determined by evaluating h^{FF} at the given state.

rp Selects actions of the relaxed plan, which is found by evaluating h^{FF} at the given state.

The first two methods select actions that can be applied at the given state. While the first method simply takes all applicable ones, the second method is more selective. In contrast to the second method, the third method additionally determines actions that cannot be applied locally, but might be suitable for future state expansions.

While using applicable operators for refinements is an intuitive approach, considering helpful actions or the relaxed plan are built upon more concrete ideas.

Helpful actions are known to be used for reducing the state space [9]. In the context of BSUAR, helpful actions are used to increase a state space instead of reducing it. Nevertheless, for both purposes, helpful actions are assumed to be more relevant for the search than other actions. Furthermore, actions of a relaxed plan generated by the FF heuristic are assumed to also be relevant for a plan of the concrete task.

The strategy of refining an UA planning task with all actions that are applicable in given states is without doubt solution-preserving. In contrast to this strategy, the other two strategies based on FF do not preserve the completeness of GBFS. Helpful actions are known to be incomplete when all other actions are ignored in a search [9]. Although relaxed plans provide supplementary actions, these actions are not applicable in the evaluated state. Therefore, using relaxed plans is also not completeness-preserving.

In the next section we provide a technique to guarantee completeness of GBFS if a non-completeness-preserving setting of BSUAR is used for UAR.

4.4 Completeness-preserving Setting

In the last section, we presented suitable but non-completeness-preserving action selection strategies for BSUAR. This section introduces a technique allowing the use of these strategies without affecting the completeness of GBFS.

The idea is to combine a non-completeness-preserving configuration of BSUAR with a configuration of BSUAR that preserves completeness. The latter configuration comes into action whenever the former one is not able to provide applicable actions in the situation of having an entirely explored reachable UA state space.

Now, all the action selection strategies of the previous section can be used while maintaining the completeness of GBFS. More specificially, the strategies which use helpful actions or relax plans can be combined with the strategy of refining an UA planning task with all applicable actions.

Chapter 5

Experimental Results

Experiments were set up in order to show, how a state-of-the-art planner configuration combined with different refinement strategies behaves. This chapter starts by presenting details about the implementation of the UAR framework and the refinement strategy BSUAR. Knowing the implementation, we take the environment on which experiments were run as well as common configurations into account. In the remaining sections of this chapter, results of experiments, which have run different configurations will be compared to one another. We start with a comparison of our approach to baseline configurations in order to get an impression of the effect of UAR. Afterwards, single strategies will be compared to one another.

5.1 Implementation Details

This section presents the evaluation system as well as optimizations of the UAR framework and the refinement strategy developed in this thesis. The optimizations do not change the basis algorithms. They rather focus on minimizing redundant work in order to save computation time.

5.1.1 Integration into Fast Downward

The UAR framework was implemented into Fast Downward [6]. Fast Downward is a state-of-the-art planning system often used as an evaluation system. Its modular framework allows implementing new search techniques. Furthermore, it includes a GBFS implementation and the FF heuristic, which are required for the UAR framework and BSRM.

The GBFS was extended with the UAR framework. The UAR framework gives the possibility to easily implement and evaluate refinement strategies. It also allows adding action pruning techniques which operate prior to a search. We have followed the modular approach of Fast Downward when implementing BSUAR. Therefore, further state and action selection strategies can be implemented and evaluated as well.

Search options can be directly specified on the command line. Next to specifying a search technique and a heuristic, it is now possible to apply pruning techniques (which operate prior to a search) as well as refinement strategies. Furthermore, we can define state and action selection strategies for BSUAR.

5.1.2 Code Optimizations

As we presented simplified algorithms in previous sections, we now give more details about code optimizations applied to the UAR framework and BSUAR refinement strategy. We present three optimizations: one is applied to UAR and two are applied BSUAR.

The UAR framework updates the search space by reopening those states where new actions can be applied. For this updates, in line 7–10 of Algorithm 3.1, we iterate through the set of new actions and the closed list in order to check if a new action can be applied to a closed state. If it is the case for an action and a state, the state is reopened by pushing it back to the open list. For the implementation, we intend to save time and to use memory instead. We maintain a list of buckets that contain states. States in a bucket can be advanced with the same action whereas the action is not yet in the UA planning task. At the first expansion of a state, the algorithm throws the state in each bucket related to actions which are applicable in this state, but which are not in the current UA planning task. When updating the UA search space, the algorithm directly takes the buckets related to the new actions and reopens the closed states from the buckets.

Further optimization can be accomplished in the context of BSUAR. The refinement method of BSUAR, as we saw it in Algorithm 4.1, requires a list of all partly expanded states. In the implementation, this method collects these expanded states during the search and throw them into buckets related to the heuristics of the states. These buckets represent heuristic layers. According to Section 4.3, the refinement method applies an action selection method to heuristic layers determining candidates for the state selection strategy. It is obvious that we do not need to evaluate all states again in a next iteration, as some states will not be candidates anymore. Therefore, we can delete states, which will not provide new actions saving a lot of processing time.

The second optimization in the context of BSUAR considers those action selection strategies which evaluate states using h^{FF} in order to get helpful actions or the relaxed plans. As these strategies are used in combination with GBFS using h^{FF}, we consider the fact that evaluating states multiple times, in order to get the heuristic estimate or in order to get the actions, is not necessary. Depending on the strategy, a relaxed plan or just the helpful actions can be cached together with the state in a node. Later, during a refinement an action selection method can retrieve the cached actions instead of evaluating the state with h^{FF} again. Hence, the evaluations are costly; the algorithm can save a lot of processing time.

5.2 Experimental Setup

To evaluate the BSRM, experiments were run on Fast Downward extended with UAR. Fast Downward was executed on machines equipped with 2 x Nehalem Quad-core 2.6Ghz CPUs and 24 GB RAM. When talking about an experiment, we mean to run several configurations of Fast Downward on a set of planning tasks in order to compare them. Planning tasks are formalized for different domains. The experiments in this thesis considered planning tasks of all domains from the benchmark suite which is provided with Fast Downward. This suite consists of 58 benchmark domains including in total 2252 tasks from previous IPC challenges, which were used for the evaluation of satisficing planners. For the experiments, Fast Downward was restricted to spend at most 30 min CPU-time and 2 GB RAM in order to find a solution for a planning task. The configuration of all experiments includes GBFS as a search method and h^{FF} as the search heuristic. The basis configuration of BSUAR uses the state selection strategy which selects all states. Configurations that differ from these settings will be mentioned.

5.3 Results

In this section, we report and discuss the results of some experiments. We start with the big picture by investigating UAR in general and comparing it to a baseline configuration.

After that, we compare our approach to a search enhancement that is implemented in Fast Downward and uses helpful actions in a different way than our approach.

The refinement strategy developed in this thesis includes two components: the state and the action selection strategy. First, we evaluate different action selection strategies in Section 5.3.3. Afterwards, different state selection strategies will be considered in Section 5.3.4.

UAR intends to refine incomplete planning tasks. As the configurations of the first experiments start on under-approximated planning tasks containing no actions, we investigate the effect of having another initial under-approximation of planning tasks in Section 5.3.5.

Our refinement strategies store states in order to preserve completeness and cache actions intending to save evaluations. Therefore, we assume that a lot of instances cannot be solved due to the memory limit. Section 5.3.6 investigates this assumption.

5.3.1 The Big Picture

This section presents the effect of GBFS with under-approximation refinement. An experiment was conducted to compare two configurations. The first configuration, the baseline, had run GBFS using h^{FF}. The second configuration additionally uses UAR.

The refinement strategy for UAR is BSUAR which uses relaxed plans for its action selection strategy. The action selection strategy is is not completeness-preserving. Therefore, it was combined with BSUAR using applicable operators as introduced in Section 4.4. Table 5.1 shows which domains could not be solved correctly by only adding relaxed plan actions. As state selection strategy, we applied the strategy of selecting all states from a heuristic layer. The searches have started on empty UA planning tasks, i.e. on tasks without actions.

In this section, this second configuration is called *UAR*. During the remaining sections, we call this second configuration of the experiment *bs-rp* which stands for BSUAR (bs) used with the relaxed plan action selection strategy (rp).

Domains (# affected tasks)		
airport (5)	parcprinter-sat11-strips (1)	sokoban-sat08-strips (12)
depot (4)	pegsol-08-strips (1)	sokoban-sat11-strips (7)
driverlog (5)	psr-large (13)	storage (12)
miconic-fulladl (7)	psr-middle (39)	trucks (1)
mystery (4)	psr-small (50)	woodworking-sat08-strips (4)
Sum (165)		

Table 5.1: Domains which cannot be solved with the action selection strategy of using relaxed plan actions.

	baseline	**bs-rp**
coverage	1576	1753
search time	1.00	0.63
expansions	1.00	0.82
generations	1.00	0.29
evaluations	1.00	0.34
plan cost	1.00	1.02
applied actions	1.00	0.40

Table 5.2: Total results of UAR relative to the baseline.

Table 5.2 compares the total result of the baseline and UAR. The *coverage* is the number of planning tasks solved by a configuration. In our experiment, searches on planning tasks failed because they exceeded the time or the memory limit. UAR covered 177 more planning tasks than the baseline. All other values in this table are relative to the baseline configuration. Commonly covered tasks are solved on average substantially faster with UAR. In context of UAR, the number of *expansions* also includes *re-expansions* of states. The number of *generations* shows how many states were generated. When a state is re-expanded, only successor states generated by newly added actions are counted. In a graph search, the number of *evaluations* is the amount of distinct states that were generated during the search. As expected, UAR saves a lot of generations and evaluations due to the restricted set of actions in UA planning tasks. The decrease of expansions compared to the baseline indicates that UAR is able to guide the search through the state space. The plan cost only increases slightly. This increase of plan cost could be a consequence of having fewer actions in the UA planning task. The number of applied actions confirms this assumption. For that number, each distinct action that were applied during a search is only counted once. We find it more interesting to compare this number relative to the number of applied actions of standard GBFS than to the number of all actions in a planning task because the GBFS algorithm never touches all actions during a search.

We now investigate the number of solved tasks in more detail. Figure 5.1 shows that UAR solves about 150 domains more than the baseline during the first 0.10 seconds. After 10 seconds UAR increases the coverage relative to the baseline. A more detailed view on the coverage and the search times of single domains will reveal further information about the effect of using UAR.

Table 5.3 shows the coverage of 34 domains resulting the experiment. Domains identically covered by both configurations are filtered out. The two right-most columns show the difference of UAR relative to the baseline in terms of number of solved tasks. The difference is represented in two numbers: the first one shows the number of tasks solved by the baseline but not by UAR,

	Coverage		Difference	
Domain (# Tasks)	baseline	UAR (bs-ha)	−	+
airport (50)	34	34	3	3
barman-sat11-strips (20)	2	20	0	18
depot (22)	15	14	2	1
driverlog (20)	18	19	1	2
elevators-sat08-strips (30)	11	12	0	1
freecell (80)	79	78	2	1
grid (5)	4	5	0	1
logistics98 (35)	30	34	0	4
mprime (35)	31	34	1	4
mystery (30)	17	18	0	1
nomystery-sat11-strips (20)	10	15	0	5
optical-telegraphs (48)	4	35	1	32
parcprinter-08-strips (30)	22	23	0	1
parcprinter-sat11-strips (20)	5	7	0	2
pathways (30)	10	13	0	3
pathways-noneg (30)	11	14	0	3
philosophers (48)	48	40	8	0
pipesworld-notankage (50)	33	41	2	10
pipesworld-tankage (50)	21	34	1	14
psr-middle (50)	38	39	1	2
rovers (40)	23	32	0	9
satellite (36)	27	34	0	7
scanalizer-08-strips (30)	28	30	0	2
scanalizer-sat11-strips (20)	18	20	0	2
schedule (150)	37	116	0	79
sokoban-sat08-strips (30)	28	27	1	0
tpp (30)	23	23	2	2
storage (30)	18	17	1	0
transport-sat08-strips (30)	11	23	0	12
transport-sat11-strips (20)	0	4	0	4
trucks (30)	17	18	0	1
trucks-strips (30)	17	16	3	2
woodworking-sat08-strips (30)	27	14	13	0
woodworking-sat11-strips (20)	12	3	10	1
total (2252)	1576	1753	52	229

Table 5.3: Coverage of planning domains and the differences between the baseline and UAR. Column "−" shows the number of tasks solved by the baseline but not by the UAR. Column "+" shows the number of tasks solved by the UAR but not by the baseline.

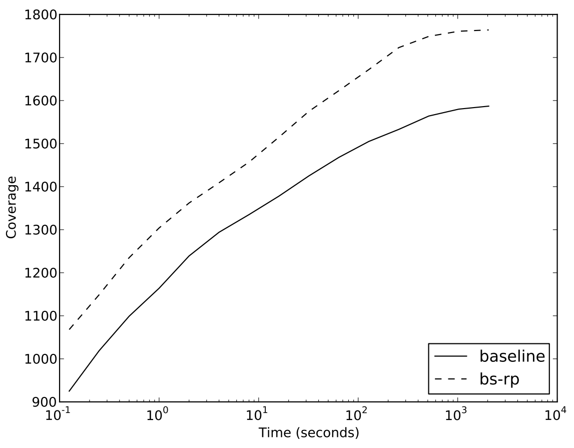
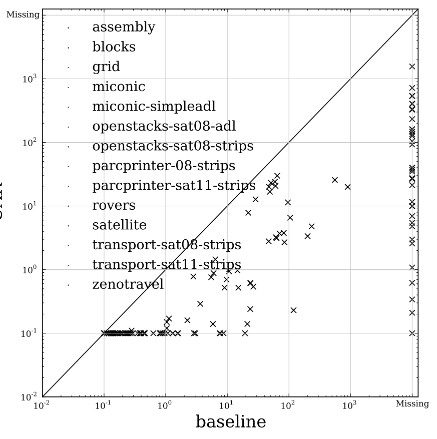


Figure 5.1: Cumulated number of covered tasks in function of the search time for the baseline and UAR configurations.

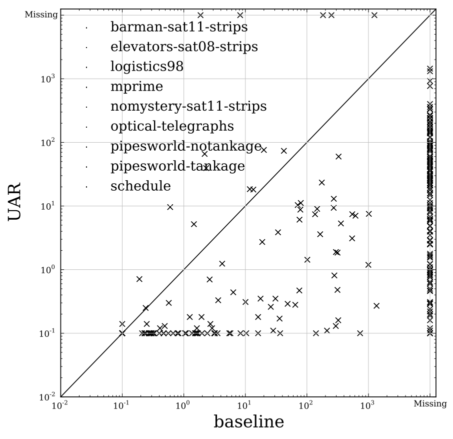
whereas the second one shows the number of tasks solved by UAR but not by the baseline. UAR is able to solve a lot of new tasks in many domains. For a few domains, the coverage decreases significantly. For more than a half of the domains listed in the table, UAR is able to cover new tasks without loosing the coverage of other tasks. For some of the other domains, it seems ambiguous which configuration is stronger. UAR seems to work well for *barman*, *optical-telegraphs*, both of the *pipesworld* domains, *rovers*, *satellite*, *schedule* and both of the *transport* domains. The *philosophers* and the two *woodworkers* domains seem to suffer from UAR. We will investigate some of these domains in more detail after a further general comparisons.

The search times of planning tasks are presented in scatter plots. These plots show planning tasks depicted as points. Tasks which are solved within the same time by both experiments are close to the diagonal line. Tasks lying in the lower right area of the plots are solved faster by UAR than by the baseline. Tasks lying at the upper and left border are not covered by one of the configurations. Tasks that are solved faster or equal than 0.10 seconds lie on a line close to the bottom or left border.

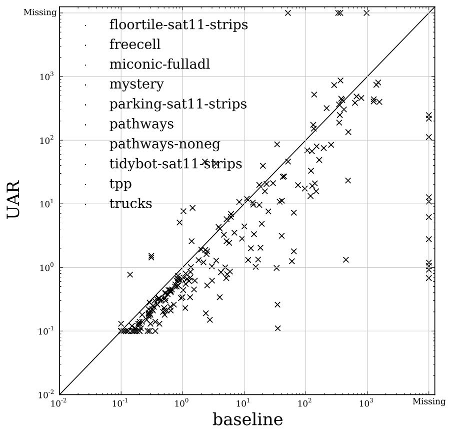
The domains were visually classified into groups depending on how their tasks respond to UAR. Figure 5.2 includes domains where UAR solves clearly more task than the baseline. Likewise, Figure 5.3 contains domains where the baseline solves clearly more task. The remaining domains can be found in Figure 5.4. Their task are either solved in nearly the same time by both configurations or the domain has more or less the same amount of tasks on both sides of the diagonal line. We further classified domains into strictly, considerably or slightly; depending on the orientation of the tasks in the plots. The adjectives describe how much faster or slower the tasks of a domain are solved by UAR than by the baseline. In addition, Table 5.4 lists search time related information. The numbers in this table are relative to the baseline. By considering the relative numbers of expanded, generated and evaluated states of domains, we are able to give possible explanations for the behaviour of the domains.



(a) strictly faster

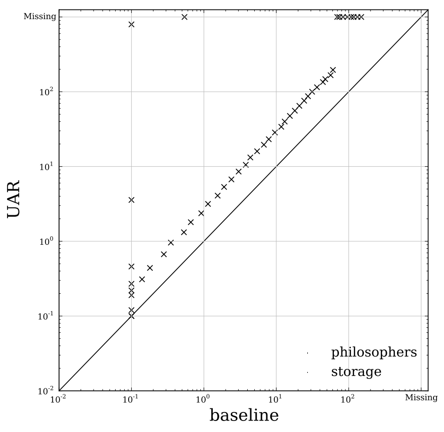


(b) considerably faster

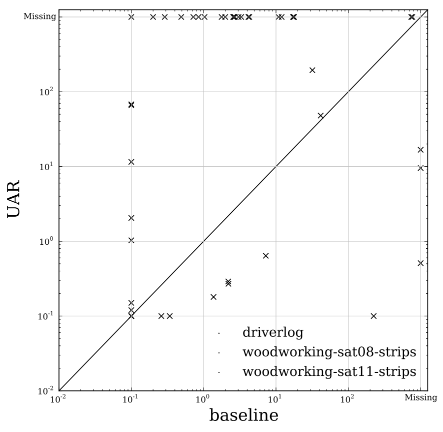


(c) slightly faster

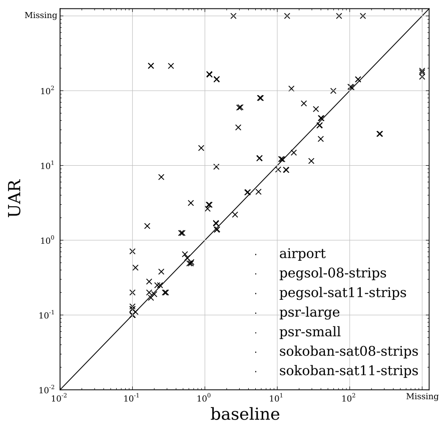
Figure 5.2: Domains whose tasks need clearly less time to be solved by UAR than by the baseline.



(a) strictly slower



(b) considerably slower

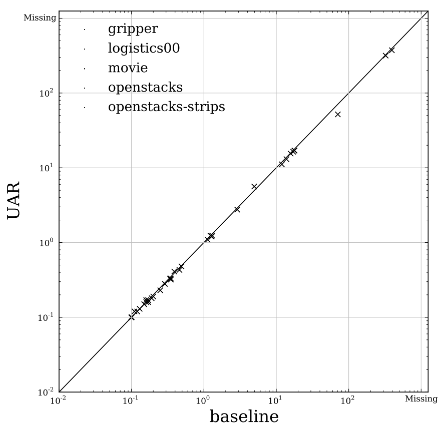


(c) slightly slower

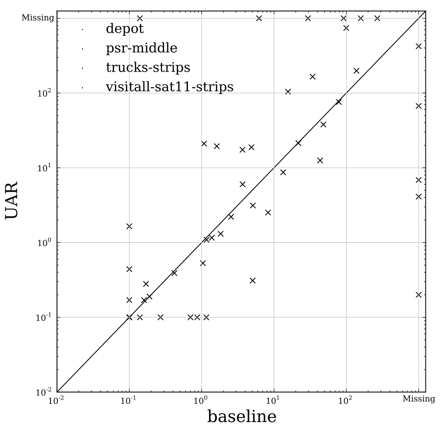
Figure 5.3: Domains whose tasks need more time to be solved by UAR than by the baseline.

Domain	**Search Time**	**Expanded**	**Generated**	**Evaluated**
airport	1.86	2.54	2.14	1.92
assembly	0.68	0.89	0.14	0.14
barman-sat11-strips	0.20	0.43	0.27	0.19
blocks	0.56	0.19	0.11	0.12
depot	0.83	1.46	0.60	0.47
driverlog	1.56	9.34	2.71	1.58
elevators-sat08-strips	0.05	0.09	0.04	0.04
floortile-sat11-strips	0.83	0.92	0.82	0.53
freecell	0.45	0.72	0.25	0.28
grid	0.27	0.31	0.14	0.13
gripper	0.99	0.98	0.77	0.88
logistics00	1.00	0.93	0.35	0.38
logistics98	0.08	0.32	0.03	0.03
miconic	0.95	1.19	0.40	0.51
miconic-fulladl	0.85	1.09	0.47	0.68
miconic-simpleadl	0.99	1.17	0.43	0.62
movie	1.00	1.11	0.08	1.00
mprime	0.27	0.75	0.02	0.02
mystery	0.73	1.15	0.13	0.18
nomystery-sat11-strips	0.50	0.73	0.20	0.24
openstacks	1.00	1.19	0.96	0.96
openstacks-sat08-adl	0.63	0.45	0.34	0.41
openstacks-sat08-strips	0.64	0.46	0.34	0.41
openstacks-strips	0.97	1.19	0.96	0.96
optical-telegraphs	0.08	0.05	0.05	0.06
parcprinter-08-strips	0.32	0.34	0.20	0.23
parcprinter-sat11-strips	0.04	0.06	0.02	0.04
parking-sat11-strips	0.53	1.41	0.69	0.66
pathways	0.57	0.55	0.25	0.20
pathways-noneg	0.31	0.34	0.15	0.15
pegsol-08-strips	1.06	1.28	1.00	1.01
pegsol-sat11-strips	1.12	1.33	1.16	1.15
philosophers	2.31	2.40	2.46	2.40
pipesworld-notankage	0.15	0.21	0.06	0.07
pipesworld-tankage	0.52	1.10	0.24	0.24
psr-large	1.36	2.20	1.60	1.60
psr-middle	1.09	1.54	1.23	1.25
psr-small	1.01	1.24	0.93	0.90
rovers	0.35	0.27	0.10	0.10
satellite	0.19	0.68	0.06	0.06
scanalyzer-08-strips	0.34	0.81	0.19	0.21
scanalyzer-sat11-strips	0.19	0.57	0.13	0.14
schedule	0.26	0.27	0.02	0.02
sokoban-sat08-strips	2.56	2.74	2.52	2.49
sokoban-sat11-strips	3.15	3.00	2.83	2.82
storage	2.54	8.15	3.10	2.47
tidybot-sat11-strips	0.86	1.17	0.79	0.82
tpp	0.59	0.97	0.49	0.41
transport-sat08-strips	0.27	0.26	0.08	0.11
trucks	0.88	1.35	0.16	0.52
trucks-strips	1.14	1.28	0.13	0.45
visitall-sat11-strips	1.65	1.33	1.35	1.43
woodworking-sat08-strips	1.12	5.01	0.82	0.69
woodworking-sat11-strips	0.36	1.14	0.14	0.17
zenotravel	0.49	0.82	0.08	0.10
Geometric mean	0.58	0.82	0.29	0.34

Table 5.4: Comparison of UAR (bs-rp) relative the baseline in context of time related properties.



(a) equal



(b) unclear

Figure 5.4: Domains that either are solved within the same time by both configurations or domains which do not fit into another class.

UAR often expands more states than the baseline for solving tasks which lie close to the diagonal line. Nevertheless, UAR compensates additional expansions by generating fewer states. Domains of these tasks can be found in Figure 5.2c. Although, tasks in Figure 5.4a are solved in a similar time by both configurations, UAR also compensates additional expansions with fewer generations when solving these tasks. UAR performs best on task from Figure 5.2b. In comparison to the baseline, it reaches considerable speedups due to fewer state expansions.

Task that are solved considerably slower by UAR are shown in Figure 5.3b. We assume that, UAR respective the relaxed plans distort the search in a wrong direction in the beginning of the search. With the given refinement guard and refinement method the search cannot add the required actions for a long time.

Figure 5.3c contains slightly slower solved tasks. Plans for those tasks often require a large percentage of actions of the original tasks. Furthermore, UAR needs a lot of refinements and re-expansions for solving these tasks. This means that required actions are often provided later than when they are needed.

We conclude the presentation of search times by showing more detailed results of the *barman* and *optical-telegraphs* domains in Figure 5.5. We especially present those two domains because they are not covered as well by another technique, which will be evaluated in Section 5.3.2.

UAR has minor effect on plan related properties. Nevertheless, we can detect some trends. A table listing the cost, the length and the number of actions of a plan relative to the baseline configuration can be found in Figure 5.5.

The cost and the length of plans strongly correlate because most of the actions have a cost of one. For domains with larger action costs, we can state that the plan cost does not rise as much as the plan length. Moreover, single actions often occur multiple times in plans generated by UAR, while plans found by the baseline include a larger variety of actions. For tasks of domains having action cost larger than one , we can now assume that UAR applies cheaper actions more often than costly actions. UAR was able to lower the plan

Domain	**Cost**	**Length**	**# Actions**
airport	0.98	0.98	0.99
assembly	1.03	1.03	1.00
barman-sat11-strips	1.07	1.16	0.98
blocks	0.85	0.85	0.78
depot	0.96	0.96	0.97
driverlog	1.07	1.07	1.05
elevators-sat08-strips	0.80	0.97	0.95
floortile-sat11-strips	1.00	1.00	1.00
freecell	1.03	1.03	1.03
grid	0.98	0.98	0.97
gripper	1.02	1.02	1.00
logistics00	1.01	1.01	1.00
logistics98	1.08	1.08	1.02
miconic	1.00	1.00	1.00
miconic-fulladl	1.00	1.00	0.99
miconic-simpleadl	1.00	1.00	1.00
movie	1.00	1.00	1.00
mprime	1.15	1.15	1.15
mystery	1.07	1.07	1.07
nomystery-sat11-strips	0.99	0.99	0.99
openstacks	1.00	1.00	1.00
openstacks-sat08-adl	1.00	1.00	0.95
openstacks-sat08-strips	1.00	1.00	0.95
openstacks-strips	1.00	1.00	1.00
optical-telegraphs	1.00	1.00	1.00
parcprinter-08-strips	1.00	0.99	0.99
parcprinter-sat11-strips	1.00	1.00	1.00
parking-sat11-strips	1.05	1.05	1.05
pathways	1.03	1.03	1.00
pathways-noneg	1.04	1.04	1.00
pegsol-08-strips	1.02	1.01	1.01
pegsol-sat11-strips	1.01	1.01	1.01
philosophers	1.00	1.00	1.00
pipesworld-notankage	0.98	0.98	0.95
pipesworld-tankage	1.02	1.02	1.01
psr-large	1.13	1.13	1.13
psr-middle	1.04	1.04	1.04
psr-small	1.01	1.01	1.01
rovers	1.05	1.05	0.98
satellite	1.13	1.13	1.09
scanalyzer-08-strips	1.11	1.26	1.25
scanalyzer-sat11-strips	1.10	1.22	1.21
schedule	1.01	1.01	1.00
sokoban-sat08-strips	1.00	1.03	1.01
sokoban-sat11-strips	1.02	1.02	1.01
storage	1.06	1.06	1.05
tidybot-sat11-strips	1.00	1.00	1.00
tpp	1.00	1.00	1.06
transport-sat08-strips	1.03	0.96	0.95
trucks	1.03	1.03	1.03
trucks-strips	1.01	1.01	1.01
visitall-sat11-strips	1.17	1.17	1.08
woodworking-sat08-strips	1.10	1.02	1.00
woodworking-sat11-strips	1.03	1.05	1.02
zenotravel	1.01	1.01	1.01
Geometric mean	1.02	1.03	1.01

Table 5.5: Comparison of UAR (bs-rp) relative the baseline in context of plan related properties.

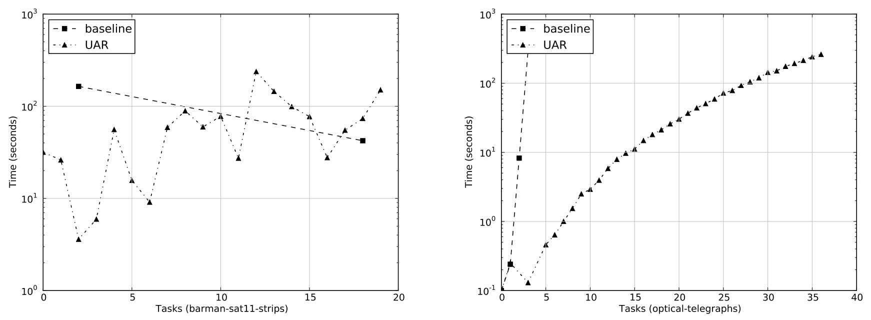


Figure 5.5: Search times of *optical-telegraphs* and *barman* tasks.

costs of *elevators* much more than decreasing the length of the plan. The most interesting result shows the *blocks* domain, where the plan length as well as the number of actions considerably decreases. The plan costs of these two domains are depicted in Figure 5.6.

Coming to the end of this section, we take a view on refinements. Principal questions are: How many times are UA planning tasks refined and how many actions are added refined tasks? Figure 5.7 gives a representing collection of examples of searches on planning tasks. It shows the development of heuristic values and the changing number of actions of a planning task. Raising values for refinements mean that new actions are added; the size of the step tells us about the amount of actions. Despite of changing heuristic values, we see a lot of large plateaus in the refinement graphs. Respecting the refinement condition, we would expect refinements, whenever the heuristic increases or stays the same. These refinements do happen a lot of times. However, they are not able to add new actions, because all expanded states have already been checked for new actions in previous iterations, which means that we have a lot of non-proper refinements. New states have to be expanded in oder to find new actions in relaxed plans. As we see from the size of the plateaus, it often takes a long time until the refinement method discovers new actions.

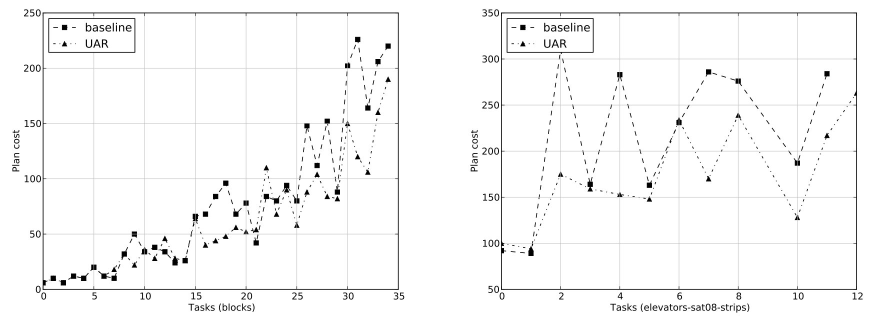


Figure 5.6: Plan costs of *blocks* and *elevators* tasks.

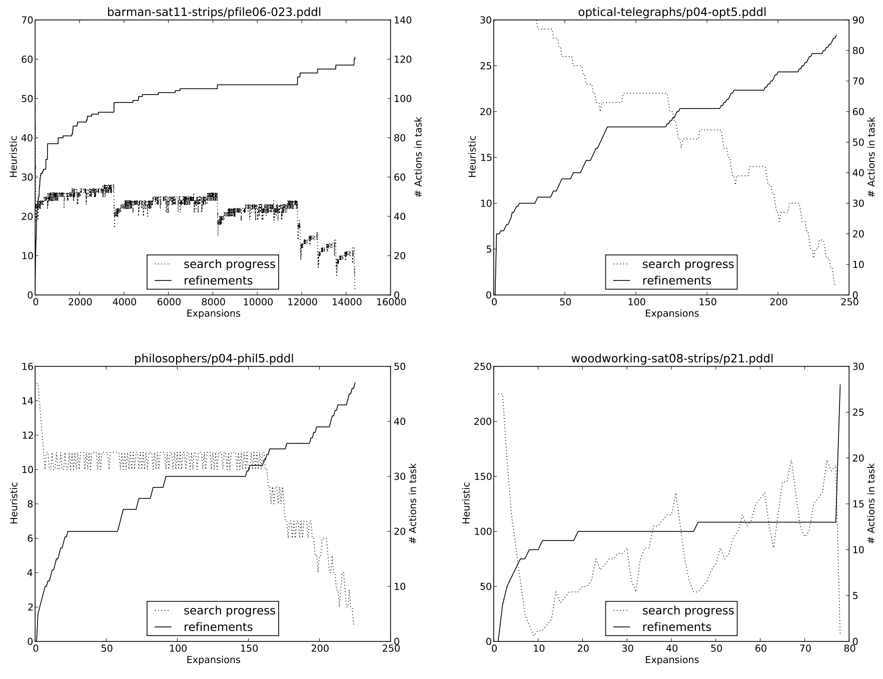


Figure 5.7: Search progress and refinements of some tasks.

5.3.2 Helpful Actions

One of the action selection strategies of BSUAR refines UA planning tasks by adding helpful actions. Fast Downward can also use helpful actions. These actions are preferred to other actions in a search and are, therefore, called preferred operators [7]. States generated by preferred operators are stored in a separate list, also called preferred list, while the open list of GBFS takes all states generated in a search as usual. During a search, Fast Downward alternates between expanding states from the preferred list and expanding states from the ordinary open list.

Considering the relationship between these two approaches, there was the question of which approach results in a better planner performance. Furthermore, these approaches are orthogonal and can, therefore, easily be combined. An additional question was: How does this combination perform?

To answer these questions, an experiment was conducted. A first configuration run searches using helpful actions as preferred operators. This configuration of the experiment is called the preferred operator approach or *pref* in tables and figures. The second configuration used BSUAR with helpful actions, denoted by *bs-ha* , starting from empty UA planning tasks. To preserve the completeness of a search, the refinement strategy was combined with BSUAR using applicable operators as presented in Section 4.4. In the following, bs-ha implies this completeness preserving combination. The third configuration applies the preferred operators approach and bs-ha in a same search. We call it pref+bs-ha.

	baseline	**pref**	**bs-ha**	**pref+bs-ha**
coverage	1576	1806	1707	1792
search time	1	0.91	0.60	0.80

Table 5.6: Total results of the preferred operator approach, bs-ha, and the combination of both.

Table 5.6 lists the total results of the run experiment relative to the baseline configuration. The preferred operator approach reaches the highest coverage among these experiments. Although the configuration bs-ha solves about 100 fewer instances than the preferred operators approach, it needs less time to solve the instances. Configuration pref+bs-ha gets close to the coverage of preferred operators and still spends less time than the preferred operator approach with solving the domains.

Figure 5.8 shows the cumulated number of solved problems depending on the search time for the conducted experiments. The preferred operator approach solves about 200 fewer instances than bs-ha during the first 0.1 seconds, but increases the coverage faster. At around 4 seconds the preferred operator approach covers more instances than bs-ha. Configuration pref+bs-ha dominates both other configurations in the time lapse between 0.4 and 12 seconds. Afterwards, it is slightly dominated by the preferred operator approach.

Table 5.7 shows the results reached by the preferred operator approach and the bs-ha configuration relative to the baseline. The first column shows the absolute coverage of the baseline configuration while the numbers in the following columns are the differences to the baseline. For each domain, the highest coverage being reached by a configuration is highlighted.

The preferred operator approach slightly decreases the coverage of four domains while it increases the coverage of 25 domains. Configuration bs-ha decreases the coverage of 14 domains. It covers new instances in 22 domains.

On the one hand, the domains *barman*, *optical-telegraphs* and *satellite* get more instances solved by bs-ha than by the preferred operator approach. On

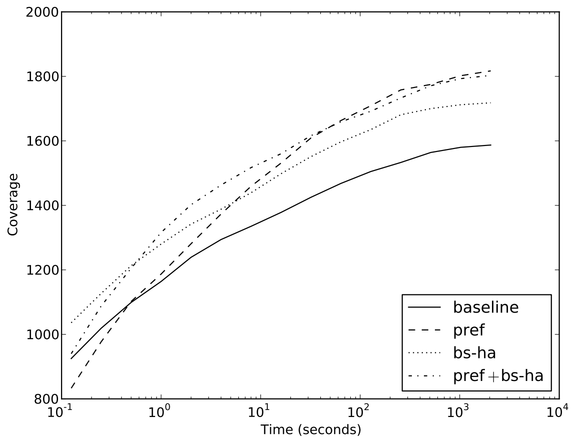


Figure 5.8: Cumulated number of solved task in function of the search time for the preferred operator approach, bs-ha and pref+bs-ha.

domain	**baseline**	**pref**	**bs-ha**
airport (50)	34	+**2**	−6
assembly (30)	**30**	**0**	−2
barman-sat11-strips (20)	2	+5	+**17**
depot (22)	15	+**3**	−1
driverlog (20)	18	+**2**	+1
elevators-sat08-strips (30)	11	0	+**2**
floortile-sat11-strips (20)	7	0	+**1**
freecell (80)	79	+**1**	0
grid (5)	4	0	+**1**
logistics98 (35)	30	+**4**	+2
miconic-fulladl (150)	136	+**1**	+**1**
mprime (35)	31	+**4**	+3
mystery (30)	17	0	+**1**
nomystery-sat11-strips (20)	10	+3	+**6**
openstacks (30)	**30**	**0**	−4
openstacks-strips (30)	**30**	**0**	−4
optical-telegraphs (48)	4	0	+**19**
parcprinter-08-strips (30)	**22**	−2	**0**
parcprinter-sat11-strips (20)	**5**	−2	**0**
parking-sat11-strips (20)	**20**	−1	−5
pathways (30)	10	+**12**	+3
pathways-noneg (30)	11	+**11**	+4
philosophers (48)	**48**	**0**	−11
pipesworld-notankage (50)	33	+**10**	+9
pipesworld-tankage (50)	21	+12	+**14**
rovers (40)	23	+**16**	+10
satellite (36)	27	+1	+**7**
scanalyzer-08-strips (30)	28	+**2**	+**2**
scanalyzer-sat11-strips (20)	18	+**2**	+**2**
schedule (150)	37	+**113**	+74
sokoban-sat08-strips (30)	**28**	**0**	−1
sokoban-sat11-strips (20)	**18**	**0**	−1
storage (30)	18	+**1**	−1
tidybot-sat11-strips (20)	**15**	−1	−1
tpp (30)	23	+**7**	−3
transport-sat08-strips (30)	11	+9	+**11**
transport-sat11-strips (20)	0	+3	+**5**
trucks (30)	17	+**1**	0
woodworking-sat08-strips (30)	27	+**3**	−14
woodworking-sat11-strips (20)	12	+**8**	−10

Table 5.7: Difference of coverage of bs-ha and the preferred operator approach in comparison to the baseline.

the other hand, preferred operators cover many more instances on the *pathways* domains, *philosophers*, *rovers*, *schedule*, *tpp* and *woodworking* domains.

The combination pref+bs-ha performs differently on the domains. Although the preferred operators approach is used orthogonal to bs-ha, the combination is only a success for a few domains. The results grouped by the effect of pref+bs-ha can be found in Tables 5.8–5.13 . Both configurations improve one another in about the same amount of domains. In general, the results show that bs-ha is not able to support the preferred operator approach as much as the preferred operator approach supports bs-ha.

We will investigate a few of these domains in more detail after a short discussion of how preferred operators and bs-ha work together. Configuration bs-ha allows fewer actions to be applied in a state. These actions are helpful and applicable at some of the expanded states. They can also be applied in states occurring later in a search. For these states, the actions are not necessarily helpful. In contrast to bs-ha, the preferred operator approach considers *all* actions that are applicable in a state. Its strenght lies in frequently expanding states which are generated by helpful actions. Interactions between these two techniques are possible. The under-approximation refinement approach can in-

domain	baseline	pref	bs-ha	pref+bs-ha
mystery (30)	17	0	+1	**+2**
optical-telegraphs (48)	4	0	+19	**+21**
pathways (30)	10	+12	+3	**+13**
pathways-noneg (30)	11	+11	+4	**+12**
pipesworld-notankage (50)	33	+10	+9	**+11**
pipesworld-tankage (50)	21	+12	+14	**+18**
psr-large (50)	13	0	0	**+1**
transport-sat08-strips (30)	11	+9	+11	**+13**
transport-sat11-strips (20)	0	+3	+5	**+6**
trucks (30)	17	+1	0	**+2**

Table 5.8: Domains in which the preferred operator approach and bs-ha improved each other.

domain	baseline	pref	bs-ha	pref+bs-ha
freecell (80)	79	**+1**	0	−1
parking-sat11-strips (20)	**20**	−1	−5	−7
storage (30)	18	**+1**	−1	−2
tidybot-sat11-strips (20)	**15**	−1	−1	−2

Table 5.9: Domains in which the preferred operator approach and bs-ha afflicted each other.

domain	baseline	pref	bs-ha	pref+bs-ha
barman-sat11-strips (20)	2	+5	**+17**	+13
elevators-sat08-strips (30)	11	0	**+2**	+1
floortile-sat11-strips (20)	7	0	**+1**	**+1**
grid (5)	4	0	**+1**	**+1**
nomystery-sat11-strips (20)	10	+3	**+6**	+4
parcprinter-08-strips (30)	**22**	−2	**0**	**0**
parcprinter-sat11-strips (20)	**5**	−2	**0**	**0**
satellite (36)	27	+1	**+7**	**+7**

Table 5.10: Domains in which the bs-ha improved the preferred operator approach.

domain	baseline	pref	bs-ha	pref+bs-ha
airport (50)	34	**+2**	−6	−5
assembly (30)	**30**	**0**	−2	−2
depot (22)	15	**+3**	−1	−1
driverlog (20)	18	**+2**	+1	+1
logistics98 (35)	30	**+4**	+2	+3
mprime (35)	31	**+4**	+3	+3
philosophers (48)	**48**	**0**	−11	−11
schedule (150)	37	**+113**	+74	+112
sokoban-sat08-strips (30)	**28**	**0**	−1	−1
sokoban-sat11-strips (20)	**18**	**0**	−1	−1
woodworking-sat08-strips (30)	27	**+3**	−14	−14
woodworking-sat11-strips (20)	12	**+8**	−10	−10

Table 5.11: Domains in which the bs-ha afflicted the preferred operator approach.

domain	baseline	pref	bs-ha	pref+bs-ha
airport (50)	34	**+2**	−6	−5
logistics98 (35)	30	**+4**	+2	+3
openstacks (30)	**30**	**0**	−4	**0**
openstacks-strips (30)	**30**	**0**	−4	**0**
rovers (40)	23	**+16**	+10	**+16**
schedule (150)	37	**+113**	+74	+112
tpp (30)	23	**+7**	−3	**+7**

Table 5.12: Domains in which the preferred operator approach improved bs-ha.

domain	**baseline**	**pref**	**bs-ha**	**pref+bs-ha**
barman-sat11-strips (20)	2	+5	+**17**	+13
elevators-sat08-strips (30)	11	0	+**2**	+1
nomystery-sat11-strips (20)	10	+3	+**6**	+4

Table 5.13: Domains in which the preferred operator approach afflicted bs-ha.

fluence the combination pref+bs-ha by restricting the set of applicable actions. The preferred operator approach can influence the combination by guiding the search through the state space being created by this restricted set of actions. As the refinement condition depends on the search progress, the search controlled by helpful actions also influences the selection of new actions for the UA planning task.

The short discussion of how the two techniques work, allows now to investigate the domains listed above. Table 5.14 presents the effect of pref+bs-ha relative to the preferred operator approach and relative to bs-ha.

Preferred operators show no effect in the *philosophers* domain when used together with bs-ha. The preferred operator approach works also worse in comparison to the baseline with respect to the search times. Therefore, we assume that helpful actions, independently of how they are used, are not suitable for *philosophers*.

The preferred operator approach is able to slightly guide searches on tasks from the *woodworking* domains, which results in fewer refinements. As tasks of these domains are well solved by the baseline and the preferred operator approach, we assume that the refinement strategy lacks in adding required actions early in the search. Furthermore, it seems that required actions cannot be added to an UA planning task for a long time.

In the *barman* domain, preferred operators trouble the search, while bs-ha is able to slightly control the search.

When solving instances from *optical-telegraphs* and *satellite*, pref+bs-ha takes most advantage from under-approximation refinement. Both techniques, the preferred operator approach and bs-ha, positively influences the coverage of pref+bs-ha on *optical-telegraph*.

The preferred operator approach has the largest effect in the *pathways* domains, *rovers*, *schedule* and *tpp*. For these domains, preferred operators are able to considerably decrease the number of expansions relative to bs-ha. We now

	pref+bs-ha					
	rel. to pref		rel. to bs-ha			
domain	**cov.**	**exp.**	**cov.**	**exp.**	**refin.**	**# actions**
barman-sat11-strips	+8	1.05	−4	2.79	1.63	1.58
optical-telegraphs	+21	0.09	+2	0.82	0.98	0.99
pathways	+1	1.57	+10	0.08	1.01	1.00
pathways-noneg	+1	1.22	+8	0.09	1.01	1.00
philosophers	−11	3.43	0	1.00	1.00	1.00
rovers	0	1.28	+6	0.18	0.98	1.00
satellite	+6	1.41	0	0.74	0.95	0.98
schedule	−1	0.73	+38	0.03	1.00	1.00
tpp	0	1.64	+10	0.09	0.80	0.88
woodworking-sat08-strips	−17	10.67	0	1.03	0.98	1.01
woodworking-sat11-strips	−18	1.20	0	1.00	0.89	0.99

Table 5.14: Effect of the combination pref+bs-ha relative to preferred operator approach (left hand side) and relative to bs-ha (right hand side).

give a possible explanation for this behavior. We assume that this behavior occurs because actions added to an UA planning task can be applied everywhere during a search. As stated in the short discussion at the beginning of this section, actions that are helpful in an expanded state will also be applicable but not helpful for other states. Consequently, a search can apply more and more action as bs-ha adds more and more actions during a search. The pruning power is, therefore, stronger in the beginning of a search than later in a search. Later in the search, preferred operators come into action and clearly support the search.

Investigating the number of refinements and actions, we see that only for *tpp* the combination needs fewer refinements. Note, despite of preferred operators guiding searches on tasks, they do not lead to fewer refinements and fewer actions in other domains. We assume that after some time, it is difficult to find new helpful actions for the UA planning task. Our assumption can be explained as follows: Using bs-ha, an UA planning task can be refined with actions that are helpful for already expanded states. At some point of the search, helpful actions of all expanded states are included in the UA planning task. Furthermore, helpful actions of many upcoming states could already be included in the UA planning task. Therefore, the UA planning task cannot be refined by bs-ha for a long time.

In this section we learned about interactions of the preferred operator approach and the UAR approach, both using helpful actions. The preferred operator approach works well on most of the domains and play an important role in a combination with bs-ha. UAR with bs-ha performs much worse on a lot of domains. Nevertheless, it needs less time to solve many tasks.

5.3.3 Action Selection Strategies

While in previous sections, we have compared configurations of BSUAR to existing methods, this section compares them among themselves. More specifically, we investigated different action selection strategies being one of two components of BSUAR. This component was introduced with BSUAR in Section 4.3.2. We were interested in the difference between adding relaxed plans and adding helpful actions to UA planning tasks. Furthermore, we identified the effect of keeping back actions during a search. This is the case when we follow the strategy of adding all applicable actions of a state to an UA planning task.

We conducted an experiment with three configurations, each one using another of the three action selection strategies presented in Section 4.3.2. The strategy of selecting relaxed plans and the strategy of selecting helpful actions were already applied in experiments of sections 5.3.1 and 5.3.2. These configurations are denoted by bs-rp respective bs-ha. In addition to these configurations, the third configuration includes BSUAR with the action selection strategy of adding applicable operators to UA planning tasks. We call it bs-appl.

In advance to the results, we shortly shed light on the difference of bs-appl to the baseline. The difference of bs-ha to the baseline lies in the availability of actions during a search. The configuration bs-appl does not add applicable actions of each state which occurs during the search. Therefore the baseline can apply more actions during a search than bs-appl. This difference in the number of actions is expected to be larger in the beginning of a search than later in a search.

	baseline	**bs-rp**	**bs-ha**	**bs-appl**
coverage	1576	1753	1707	1560
search time	1	0.63	0.62	1.15
refinements	-	1	1.57	2.17
#actions	-	1	0.90	2.15

Table 5.15: Total results of using BSUAR with different action selection strategies compared to the baseline.

The total results of the experiments relative to the baseline are shown in Table 5.15. The configuration bs-appl performs slightly worse than the baseline. The configurations bs-rp and bs-ha has a similar total search time. In contrast to bs-rp, bs-ha needs more refinements which add fewer actions in total. On the other hand, the configuration bs-rp covers considerably more instances than bs-ha.

Figure 5.9 shows the number of covered tasks depending on the search time of bs-rp, bs-ha and bs-appl. Every moment, the coverage of the baseline is slightly larger than the coverage of bs-appl. Configurations bs-rp and bs-ha behaves similarly. We note that bs-rp gains more coverage than bs-ha after 50 seconds.

Table 5.16 gives more information about the coverage of single domains. The configuration bs-appl decreases the coverage in 18 domains relative to the baseline. The differences lie mostly between one and four instances, except for *philosophers*, where bs-appl solved nine instances less than the baseline. The bs-appl configuration increases the coverage by one in three domains. It is also able to increase *airport* by four and *optical-telegraphs* by 19 instances.

By comparing these results to the already known results of bs-rp and bs-ha, we can assume some reasons for the resulting coverage. We will investigate *airport*, *optical-telegraphs*, *philosophers* and the *woodworking* domains.

The *airport* domain is best solved by bs-appl. The tasks are solved differently by each configuration. Therefore, no trend can be determined. Nevertheless,

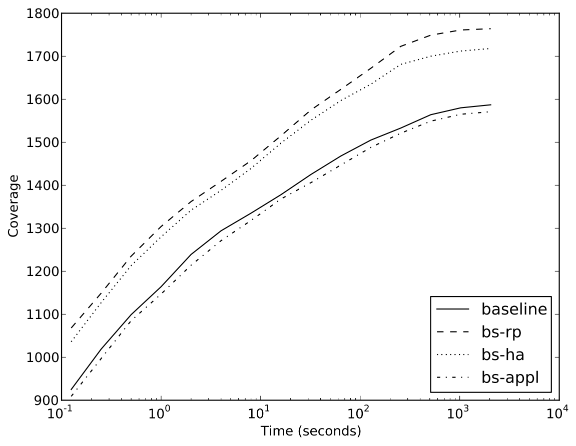


Figure 5.9: Cumulated number of solved task in function of the search time for the different action selection method.

domain	**baseline**	**bs-rp**	**bs-ha**	**bs-appl**
airport (50)	34	0	−6	+**4**
assembly (30)	**30**	**0**	−2	**0**
barman-sat11-strips (20)	2	+**18**	+17	+1
depot (22)	**15**	−1	−1	−1
driverlog (20)	18	+**1**	+**1**	−1
elevators-sat08-strips (30)	11	+1	+**2**	0
floortile-sat11-strips (20)	7	0	+**1**	0
freecell (80)	**79**	−1	**0**	**0**
grid (5)	4	+**1**	+**1**	0
logistics98 (35)	30	+**4**	+2	0
miconic-fulladl (150)	136	0	+**1**	0
mprime (35)	31	+**3**	+**3**	−1
mystery (30)	17	+**1**	+**1**	0
nomystery-sat11-strips (20)	10	+5	+**6**	−1
openstacks (30)	**30**	**0**	−4	−4
openstacks-strips (30)	**30**	**0**	−4	−4
optical-telegraphs (48)	4	+**31**	+19	+19
parcprinter-08-strips (30)	22	+**1**	0	−2
parcprinter-sat11-strips (20)	5	+**2**	0	−3
parking-sat11-strips (20)	**20**	**0**	−5	**0**
pathways (30)	10	+**3**	+**3**	−1
pathways-noneg (30)	11	+3	+**4**	−1
philosophers (48)	**48**	−8	−11	−9
pipesworld-notankage (50)	33	+8	+**9**	−2
pipesworld-tankage (50)	21	+13	+**14**	+1
psr-middle (50)	38	+**1**	0	0
rovers (40)	23	+9	+**10**	0
satellite (36)	27	+**7**	+**7**	−1
scanalyzer-08-strips (30)	28	+**2**	+**2**	0
scanalyzer-sat11-strips (20)	18	+**2**	+**2**	0
schedule (150)	37	+**79**	+74	0
sokoban-sat08-strips (30)	**28**	−1	−1	**0**
sokoban-sat11-strips (20)	**18**	**0**	−1	**0**
storage (30)	**18**	−1	−1	**0**
tidybot-sat11-strips (20)	**15**	**0**	−1	−1
tpp (30)	23	0	−3	+**1**
transport-sat08-strips (30)	11	+**12**	+11	0
transport-sat11-strips (20)	0	+4	+**5**	0
trucks (30)	17	+**1**	0	−3
trucks-strips (30)	**17**	−1	**0**	−4
woodworking-sat08-strips (30)	**27**	−13	−14	−1
woodworking-sat11-strips (20)	**12**	−9	−10	−2

Table 5.16: Difference of coverage between configurations of bs using different action selection strategies relative to the baseline.

bs-ha seems to need more actions in addition helpful actions or relaxed plans.

The *optical-telegraphs* domain is best solved by bs-rp. The configurations bs-ha and bs-appl solve the same amount of instances. They solve this domain exactly in the same manner. Just the number of actions in the last UA planning task and the number of generated states are different. The number of evaluations is identical. In this context, it implies that using helpful actions saves only the generation of a few states. Furthermore, each third instance, starting at instance 4, remains unsolved. Configuration bs-rp is able to solve these remaining instances. Additionally, bs-rp expands fewer states. While the BSUAR refinement method using helpful actions or applicable actions reveal no difference to each other when solving instances of *optical-telegraphs*, the actions of a relaxed plan clearly support searches on these instances.

The results show that bs-appl performs worse on *philosophers*. Investigating the number of evaluations, we can see that all configurations of BSUAR evaluate many more states than the baseline, meaning that many actions are not available when they are required. The number of evaluations corresponds with the number of states stored in the open or closed list. Configurations bs-appl and bs-ha stores a similar amount of states. The latter has a lower coverage due to requiring more memory for maintaining a separate list of expanded states in order to preserve completeness. Thus, it solves fewer instances. The configuration bs-rp has the highest coverage due to fewer evaluated nodes than the other two configurations.

All configurations of BSUAR perform worse on the *woodworking* domains than the baseline. The best coverage among these configurations has bs-appl. This configuration solves the instances much faster than the baseline due to fewer state generations and evaluations. We assume that the relaxed plans respectively the helpful actions mislead the search. Furthermore, we assume that other actions than helpful actions or relaxed plan actions are added to the UA planning task for a long time.

In this section, we compared the different action selection strategies. The experiment running bs-appl helped us to understand the effects of under-approximation refinement using BSUAR in more detail.

5.3.4 State Selection Strategies

This section evaluates strategies for selecting a subset of states among states having the same heuristic value. A state selection strategy forms one of two components of the BSUAR refinement strategy, which was introduced in Section 4.3.1. The configurations that have been discussed in the previous sections use the strategy of taking all states from a heuristic layer. In this section, we test other strategies next to the already investigated strategy:

first It takes the first state among provided states where an action selection method provides new actions.

least First, it applies an action selection strategy to all of the given states. Afterwards, it takes the state that provides the least amount of new actions.

most It differs from the previous strategy in choosing the states where an action selection strategy provides the most amount of new actions.

heur First, it applies an action selection strategy to all of the given states. Afterwards, it applies the context enhanced additive heuristic [8], among the states where the action selection strategy provides new actions. Now, one of the states having the smallest heuristic value is selected.

We created an experiment including a configuration for each of these strategies. Configuration bs-rp, which is the configuration UAR from Section 5.3.1, was taken as a basis. We only considered the action selection strategy which uses relaxed plans, as we expect similar results for the other configurations.

The results are listed in Table 5.17. As we see from this table, the different strategies perform nearly identically. This is not surprising, as a refinement strategy has often only one state to consider, as all other expanded states were already checked for new actions.

This fact raises the question of how the state selection strategies behave when another refinement condition is used. Thus, a supplementary experiment was run which refined only if the heuristic values increased, i.e. expanding states get

	baseline	bs-rp (all)	first	least	most	heuristic
coverage	1576	1752	1749	1750	1751	1751
search time	1.00	0.59	0.61	0.60	0.60	0.62
refinements	-	1.00	1.01	1.00	1.00	1.01
# actions	-	1.00	0.99	0.99	1.00	0.99

Table 5.17: Total results of the different state selection strategies relative to the baseline configuration.

away from the goal. In other words, BSUAR ignores heuristic plateaus. Using this condition, a search builds large heuristic plateaus. Then, a refinement strategy can choose among a lot of states.

	baseline	bs-rp (all,<=)	all	first	least	most	heuristic
coverage	1576	1752	1699	1695	1687	1701	1689
search time	1.00	0.60	0.62	0.63	0.63	0.63	0.69
refinements	-	1.00	0.60	0.75	0.68	0.63	0.75
# actions	-	1.00	0.94	0.89	0.91	0.92	0.89

Table 5.18: Total results of the different state selection strategies used in combination with the refinement condition that ignores plateaus.

Table 5.18 shows the result of using the modified refinement condition relative to the baseline. Refinements and number of actions are relative to the usual refinement condition. The modified condition performs in average worse than the usual condition. Nevertheless, we see a decrease in the amount of refinements and number of actions in the last task. Having a lower coverage when using the modified condition confirms that that the refinement guard as it was defined in Section 4.1 was a good choice.

5.3.5 Initial UA Planning Tasks

The under-approximation framework allows to start a search from an incomplete planning task. Experiments of previous sections started the searches on empty UA planning tasks. In this section, we investigate consequences of beginning a search with an initial set of actions. For the initial set, we chose all actions of the relaxed plan generated by h^{FF}evaluated at the initial state. We assumed that an UA planning task having these actions at the beginning of the search needs fewer refinements.

To check our assumption, we conducted an experiment with two configurations; both starting searches with a first relaxed plan. One configuration is based on bs-ha from Section 5.3.2, the other extends configuration bs-appl, which was defined in Section 5.3.3. To differ the configurations, we call the configurations from previous sections *empty* or $\emptyset$ while *rp* denotes configurations defined in this section.

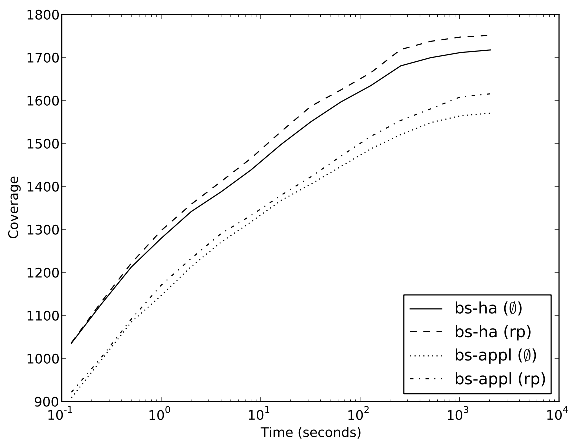


Figure 5.10: Cumulated number of solved tasks in function of the search time for configurations bs-ha and bs-appl that either starts searching on an empty UA planning task (∅) or starts searches being supplied with actions of a first relaxed plan (rp).

	baseline	**bs-rp**	**bs-ha (∅)**	**bs-ha (rp)**	**bs-appl (∅)**	**bs-appl (rp)**
coverage	1576	1753	1707	1741	1560	1605
search time	1.00	0.63	0.63	0.62	1.16	1.06
refinements	-	1.00	1.56	1.04	2.16	1.72
# actions	-	1.00	0.90	0.89	2.15	1.96

Table 5.19: Total results of taking different initial UA planning tasks relative to the baseline.

Table 5.19 shows the effect relative to the baseline configuration of Section 5.3.1. Configurations rp are able to increase the coverage of bs-ha and bs-appl. As expected, rp needs fewer refinements. Furthermore, bs-appl (rp) decreases the search time, the refinements and the number of actions in the last UA task.

Figure 5.10 shows the result of both versions of the configurations. Providing a starting search with a single relaxed plan clearly supports the search in solving tasks of different complexities.

Looking at the coverage reached by bs-ha, Table 5.20 shows that bs-ha (rp) covered slightly more instances in some domains, except for *optical-telegraphs*, where it solved considerably more tasks. Table 5.21 showing the results for bs-appl contains even more domains, where bs-appl (rp) increases the coverage. Likewise to configuration bs-ha (rp), configuration bs-appl (rp) also increases the coverage of *optical-telegraphs* by many tasks. Remembering the results of Section 5.3.3, where we found each third task of *optical-telegraphs* being unsolved by bs-ha and bs-appl, it seems now that actions of a first relaxed plan give important impulses to searches on tasks from this domain.

domain	**baseline**	**bs-ha ($\emptyset$)**	**bs-ha (rp)**
airport (50)	**34**	−6	−1
assembly (30)	**30**	−2	**0**
barman-sat11-strips (20)	2	+**17**	+13
freecell (80)	**79**	**0**	−1
logistics98 (35)	30	+2	+**4**
miconic-fulladl (150)	136	+1	+**2**
openstacks (30)	**30**	−4	**0**
openstacks-strips (30)	**30**	−4	**0**
optical-telegraphs (48)	4	+19	+**31**
parcprinter-08-strips (30)	22	0	+**1**
parcprinter-sat11-strips (20)	5	0	+**2**
parking-sat11-strips (20)	**20**	−5	−4
pathways (30)	10	+3	+**5**
pathways-noneg (30)	11	+4	+**5**
rovers (40)	23	+10	+**11**
tidybot-sat11-strips (20)	**15**	−1	−2
transport-sat11-strips (20)	0	+**5**	+4
trucks (30)	17	0	+**1**
woodworking-sat08-strips (30)	**27**	−14	−13
woodworking-sat11-strips (20)	**12**	−10	−9

Table 5.20: Domains where the non-empty initial UA has an effect on the coverage when using bs-ha. The two columns on the right hand side show the differences to the baseline for both kinds of initial planning tasks.

domain	**baseline**	**bs-appl ($\emptyset$)**	**bs-appl (rp)**
airport (50)	34	+**4**	+2
barman-sat11-strips (20)	2	+1	+**2**
depot (22)	15	−1	+**1**
floortile-sat11-strips (20)	**7**	**0**	−2
freecell (80)	79	0	+**1**
logistics98 (35)	**30**	**0**	−2
miconic-fulladl (150)	136	0	+**2**
mprime (35)	31	−1	+**2**
mystery (30)	17	0	+**1**
nomystery-sat11-strips (20)	**10**	−1	**0**
openstacks (30)	**30**	−4	**0**
openstacks-strips (30)	**30**	−4	**0**
optical-telegraphs (48)	4	+19	+**33**
parcprinter-08-strips (30)	**22**	−2	−1
parcprinter-sat11-strips (20)	**5**	−3	**0**
pathways (30)	**10**	−1	**0**
pipesworld-notankage (50)	**33**	−2	**0**
satellite (36)	27	−1	+**2**
schedule (150)	**37**	**0**	−2
storage (30)	18	0	+**1**
tpp (30)	23	+**1**	0
transport-sat08-strips (30)	11	0	+**1**
trucks (30)	**17**	−3	−1
trucks-strips (30)	**17**	−4	−3
woodworking-sat08-strips (30)	27	−1	+**1**
woodworking-sat11-strips (20)	12	−2	+**2**

Table 5.21: Domains where the non-empty initial UA has an effect on the coverage when using bs-appl. The two columns on the right hand side show the differences to the baseline for both kinds of initial planning tasks.

5.3.6 Providing more Memory

Configuration bs-rp of the experiments additionally uses bs-appl that stores its own list of expanded states in order to preserve completeness. This additional list growing large for some domains requires a lot of memory. Moreover, further memory is used by bs-rp due to caching relaxed plans in order to save computation time.

We conducted an experiment to investigate the consequence of the memory limit. The configurations of these experiments were the same as for the baseline and UAR/bs-rp of Section 5.3.1. In addition to these configurations, the experiments were allowed to use up to 6 GB RAM for each task instead of 2 GB.

Table 5.22 shows domains, which respond to the larger supply of memory.

	Coverage	
Domain	without UAR	with UAR
airport (50)	$34 + 1$	$34 + 1$
barman-sat11-strips (20)	$2 + 4$	20
floortile-sat11-strips (20)	7	$7 + 1$
miconic-fulladl (150)	136	$136 + 1$
optical-telegraphs (48)	4	$35 + 8$
parcprinter-08-strips (30)	22	$23 + 1$
parcprinter-sat11-strips (20)	5	$7 + 1$
pathways (30)	10	$13 + 1$
pathways-noneg (30)	11	$14 + 1$
philosophers (48)	48	$40 + 6$
psr-large (50)	$13 + 2$	$13 + 3$
psr-middle (50)	$38 + 2$	$39 + 4$
rovers (40)	23	$32 + 1$
schedule (150)	37	$116 + 6$
sokoban-sat08-strips (30)	28	$27 + 1$
trucks-strips (30)	17	$16 + 2$
visitall-sat11-strips (20)	3	$3 + 1$
total (2252)	$1576 + 9$	$1753 + 39$

Table 5.22: Effect on coverage when providing 6 GB RAM instead of 2 GB.

It did not surprise, that the experiment using UAR was capable to further improve the coverage. The domains *optical-telegraphs*, *philosophers* and *schedule* benefit most from supplementary memory. It seems likely that the remaining tasks of these domains can also be solved within 30 minutes, when memory is unlimited.

Chapter 6

Related Work

This section discusses related work restricted to action pruning techniques in the area of satisficing planning. In general, action pruning means ignoring some actions during a search. For this discussion of related work, we differentiate between two kinds of action pruning techniques: static action pruning and dynamic action pruning. Next to action pruning techniques, we discuss search enhancements that are not classified as dynamic action pruning techniques, but guide searches dynamically by preferring actions to other actions.

Static action pruning techniques exclude actions from a planning task, previously to a search. As a consequence, transitions are pruned globally from the state space at once. Dynamic action pruning techniques determine actions during a search. Rather than pruning whole actions, the dynamic techniques cut transitions locally from expanding states.

Haslum and Jonsson [5] as well as Nebel, Dimopoulos and Koehler [10] introduced specific static pruning techniques.

Haslum and Jonsson described a method to identify redundant operators. The method finds and eliminates single operators, which are equivalent to operator sequences. During the preprocess of a search, a greedy algorithm compares each operator in combination with operator sequences by following determined conditions. The conditions guarantee that this approach is completeness-preserving. Although their approach is suitable to prune actions while preserving the completeness of a search, it is costly to check all actions of a complex planning task. Our UAR approach can ignore all actions of the planning task in the beginning of the search.

The work of Nebel, Dimopoulos and Koehler presents a method to determine relevant facts and operators previously to a search. The method generates an AND-OR tree of a relaxed planning task in order to find a minimal set of relevant facts while relevant operators can be extracted from this tree. Restricting the search to the sole use of operators determined by the AND-OR tree does not preserve solution existence in a search. Therefore, Nebel et al. restart a search on the original problem after no solution was found using the determined operators. Instead of restarting the search, their approach could use the UAR framework applying an appropriate refinement strategy to proceed on the given search.

Hoffmann and Nebel introduced one of the first dynamic action pruning techniques together with the Fast Forward planner [9]. The FF planner implements the FF heuristic h^{FF} which determines helpful actions in a state evaluation. It applies these actions in a state while other actions applicable in this state are ignored. Using helpful actions in this way is not solution preserving. If a search fails, FF starts a new search, this time allowing all actions to be applied. In contrast to the FF planner, UAR avoids those restarts by allowing other actions than just the helpful actions to be applied. The main difference is that the FF planner considers only state transitions which are induced by helpful actions, while UAR considers all transitions induced by actions that were helpful in at least one of the preceding state expansions. Consequently, a search using BSUAR with the helpful actions action selection strategy can follow more transitions than the FF planner. They have in common, that states are only advanced with helpful actions, respective with actions that were helpful at one of the past states. In addition to helpful actions, a configuration of BSUAR extracts the relaxed plan of h^{FF} for refining under-approximation planning tasks. This allows the search to follow even more transitions.

Following works describe techniques that guide the search by preferring successor states to states generated by non-preferred actions. These techniques still apply all actions which are applicable in expanding states. Therefore, they do not really prune actions respective transitions like the FF planner.

YAHSP [14], yet another heuristic search planner, introduced by Vidal uses relaxed plans of h^{FF} in a lookahead strategy. The strategy takes the longest sequence consisting of relaxed plan actions that is applicable in the concrete search. Such a sequence, called lookahead plan, is directly applied in a expanding state generating so called lookahead states. These states are preferred to other successor states having the same heuristic value. Completeness of a GBFS is preserved by advancing states with all their applicable actions as usual. In contrast to YAHSP, BSUAR is able to add all the actions of a relaxed plan independently of their applicability. YAHSP maintains completeness by generating all successor states. UAR saves this generations of states by reexpanding states in order to apply the remaining actions and to preserve completeness. In YAHSP, a search is guided by transitions that are induced by helpful actions or the lookahead plans. Searches using UAR are rather restricted by a small set of available actions than guided by transitions.

Helmert introduced the causal graph heuristic which extracts helpful transitions [7] similarly to helpful actions of FF. He calls this kind of actions/transitions preferred operators. In contrast to the FF planner and likewise to YAHSP, Fast Downward, the planner developed by Helmert, prefers these actions to other actions instead of eliminating them from the search space. Fast Downward has implemented two open queues into the GBFS: a first queue for successor states generated by preferred operators, a second queue maintains all successors as usual. GBFS alternates between fetching states of the first queue and fetching states of the second queue. The availability of all actions and all generated states preserves solution existence. This approach and slight variants of this approach have been investigated in detail by Helmert and Richter [11]. We compared Helmert's preferred operator approach to UAR in Section 5.3.2. The preferred operator approach performs much better than UAR using helpful actions. Although UAR often needs less time to solve planning tasks, it is not able to solve more tasks because of actions that are missed in the under-approximations of the planning tasks. Search time reductions are caused by less generations of successor states.

Wehrle, Kupferschmid and Podelsky defined useless actions [15]. An action is said to be useless if it is not missing in an optimal plan to the goal. Using an arbitrary heuristic as a distance estimator, this method classifies an action as relatively useless if $h(\Pi_a, s) \leq h(\Pi, s')$ i.e. the heuristic value of state s is smaller or equal in the planning task without action a (Π_a) than the heuristic value of the successor state s' generated by action a in the original task Π. Cutting useless actions from a state does not preserve solution existence. To overcome this problem and to combine useless actions with preferred operators, a search maintains three open queues, each for a different kind of successor states: useless, preferred and unknown. In difference to the dual queue approach, these queues contain distinct sets of states. The search alternates between expanding preferred states and expanding unknown states. In the range of a small probability, it also expands useless states. Likewise to preferred operators, useless actions are determined in a current state. In difference to preferred operators, a search needs to evaluate successor states in order to determine a relatively useless action. Despite of requiring successor states, the useless actions approach could also be used in an action selection strategy for BSUAR for filtering useless actions out of helpful actions. Furthermore the approach of determining useless actions is orthogonal to UAR. Therefore, BSUAR and the useless actions approach could be used together in a search.

Chapter 7

Conclusion

We conclude with a classical conclusion in the next section. The last section suggests future work that also considers some of the related works.

7.1 Overall Results

We introduced a pruning technique that follows a similar approach like GEGAR as it was used by Seipp and Helmert [13]. The subjects of their approach as well as of our approach are approximations of planning tasks, because approximations can be solved more efficiently than the original tasks. However, either the plans that are found in approximated tasks are not valid for concrete tasks or no plan can be found at all. The sharing idea was to refine approximations of planning tasks until a valid plan for the concrete task is found in the approximation. UA planning tasks are refined by inserting actions from the original planning task to the UA.

We have asked in the introduction for a method that adds a small but sufficient amount of actions to under-approximations of planning tasks. In order to provide a solution for this demand, we have created a general framework for refining under-approximations of planning tasks. For this framework, we developed and implemented a specific refinement strategy. This strategy, called BSUAR, is able to select a sufficient set of actions in order to find plans. Moreover, BSUAR is able to apply in average less than a half of the actions touched during standard GBFS. This reduced set of actions often results in less generations and evaluations of states, which saves a lot of search effort. Moreover, the average plan cost increases only slightly. The algorithm of BSUAR consists of two further components allowing to specify different state and action selection strategies. While different state selection strategies compared to each other show minor differences in planner performance, the action selection strategies behaves differently. Especially, the action selection strategy which refines under-approximations by adding actions of relaxed plans produced by $h^{FF}$was able to solve many more task than other action selection strategies and standard GBFS. The best results come form experiments using BSUAR in combination with preferred operators [7] because preferred operators are also able to guide the search through UA state spaces. Some planning tasks do not benefit from BSUAR. However, developing action selection strategies that are more dynamic seems to be the key to also cover these tasks in future.

7.2 Future Work

The UAR framework allows to implement additional refinement strategies. BSUAR uses a search heuristic in its refinement process. Nevertheless, other refinement strategies could be developed that consider the preconditions and effects of actions in UA planning task.

Moreover, BSUAR allows implementing an action selection strategy. Next to the strategies developed in this thesis, we could also apply strategies using preferred operators or relaxed plans of other heuristics [11].

We considered some search enhancements for guiding searches in Section 6. We already evaluated the preferred operators approach in Section 5.3.2. Moreover, we could also evaluate YAHSP [14] and useless actions [15] for searching on under-approximations of planning tasks.

With a critical review on the plateaus, it can reasonably be expected that the developed refinement condition of BSUAR can burden the search; namely, when states on a relatively small plateau can be advanced to suitable successor states. In this case, additional actions might not be useful and would blow up the state space. Therefore, we suggest to trigger refinements depending on the size of plateaus.

Bibliography

[1] Christer Bäckström and Bernhard Nebel. Complexity results for SAS$^+$ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[2] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1):5–33, 2001.

[3] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169, 2000.

[4] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions Systems Science and Cybernetics*, 4(2):100–107, 1968.

[5] Patrik Haslum and Peter Jonsson. Planning with reduced operator sets. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling*, pages 150–158, 2000.

[6] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[7] Malte Helmert. *Understanding Planning Tasks Domain Complexity and Heuristic Decomposition*, volume 4929 of *LNAI*. Springer-Verlag, 2008.

[8] Malte Helmert and Héctor Geffner. Unifying the causal graph and additive heuristics. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling*, pages 140–147, 2008.

[9] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[10] Bernhard Nebel, Yannis Dimopoulos, and Jana Koehler. Ignoring irrelevant facts and operators in plan generation. In *In Proceedings of the 4th European Conference on Planning*, pages 338–350, 1997.

[11] Silvia Richter and Malte Helmert. Preferred operators and deferred evaluation in satisficing planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, pages 273–280, 2009.

[12] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3th international edition)*. Pearson Education, 2010.

[13] Jendrik Seipp and Malte Helmert. Counterexample-guided cartesian abstraction refinement. In *Proceedings of the 23th International Conference on Automated Planning and Scheduling*, 2013.

[14] Vincent Vidal. A lookahead strategy for solving large planning problems. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1524–1525, 2003.

[15] Martin Wehrle, Sebastian Kupferschmid, and Andreas Podelski. Useless actions are useful. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling*, pages 388–395, 2008.