

Refinement Strategies for Counterexample-Guided Cartesian Abstraction Refinement

Bachelor's thesis

Faculty of Science at the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
<https://ai.dmi.unibas.ch/>

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Jendrik Seipp

Martin Zumsteg
mar.zumsteg@stud.unibas.ch
2016-056-111

July 20, 2019

Acknowledgments

I would like to thank Prof. Dr. Malte Helmert for giving me the opportunity to work on an interesting, practical thesis, as well as access to the AI group's Fast Downward planner[2] and evaluation suite.

Additionally, my thanks go to Dr. Jendrik Seipp for helping me get started with the Fast Downward suite and frequent feedback on my results.

Calculations were performed at sciCORE (<http://scicore.unibas.ch/>) scientific computing center at University of Basel.

Abstract

Abstractions are a simple yet powerful method of creating a heuristic to solve classical planning problems. In this thesis we make use of cartesian abstractions generated with CEGAR [4]. This method refines abstractions incrementally by finding flaws and then resolving them until the abstraction is sufficiently evolved.

The goal of this thesis is to implement and evaluate algorithms which select solutions of such flaws, in a way which results in the best abstraction (that is, the abstraction which causes the problem to then be solved most efficiently by the planner).

We measure the performance of a refinement strategy by running the Fast Downward planner [2] on a problem and measuring how long it takes to generate the abstraction, as well as how many expansions the planner requires to find a goal using the abstraction as a heuristic. We use a suite of various benchmark problems [6] for evaluation, and we perform this experiment on the original problem, then by adding landmarks as a subtask to the abstraction, and finally by computing a saturated cost partitioning.

Finally we attempt to predict which refinement strategy should be used based on parameters of the SAS+ problem, potentially allowing the planner to automatically select the best strategy at runtime.

Table of Contents

Acknowledgments	ii
Abstract	iii
1 Introduction	1
2 Background	2
2.1 Counterexample-guided abstraction refinement (CEGAR)	3
2.1.1 Building the abstraction on subtasks	4
2.1.2 Saturated cost partitioning	4
2.2 Refinement strategies of the Fast Downward planner	5
2.2.1 The UNWANTED-strategies	5
2.2.2 The REFINED-strategies	6
2.2.3 The HADD-strategies	6
3 New refinement strategies	7
3.1 The CG-strategies	7
3.2 The GOAL_DIST-strategies	8
3.3 The HIGHER_DIST-strategies	8
3.4 The ACTIVE_OPS-strategies	8
3.5 Predicting the best refinement strategy	9
4 Results	11
4.1 Running the planner	11
4.2 Original task	12
4.2.1 Number of distinct splits	13
4.2.2 Progress of average goal distance	17
4.3 Including subtasks	18
4.3.1 Number of distinct splits	21
4.4 Applying SCP	22
4.5 Predicting the best refinement strategy	24
5 Conclusions	25
5.1 Future work	25
5.1.1 Tie-breaking strategies	25

5.1.2	A strategy with the most freedom	25
5.1.3	Varying split selection strategy during SCP	26
5.1.4	More complex prediction features	26

Bibliography**27**

1

Introduction

To solve a classical planning problem deterministically, the common approach involves exploring the state-space of the problem starting from the initial state until we reach a goal state. Often we also desire a problem to be solved optimally, that is we seek a plan that solves the problem with the least number of actions or with the lowest cost. In all cases, as the numbers of variables and domains grow, the state space grows exponentially, meaning for a timely solution of complex problems we require a heuristic to guide our search.

We then find two contradictory desiderata:

1. The problem solver should be general, being capable of solving any problem that can be expressed in a format such as SAS+ or PDDL. This means we cannot use heuristics that are well-designed for any specific problem or domain.
2. We want a heuristic which best fits the problem, allowing for the least expansion of states and the fastest search. Because of generality, we do not have an intuition of how to solve the problem, and must extrapolate heuristic values based only on the state space's description.

The goal of an abstraction is to best represent the relevant parts of a state space, while remaining much smaller to ensure the abstraction can be created and solved quickly. For this thesis, we will only look at Cartesian abstractions, which form abstract states by assigning each variable a set of values (instead of only a single value as with concrete states). The abstractions are generated using counterexample-guided abstraction refinement[4] (CEGAR), which improves abstractions by finding solutions that are not applicable in the concrete search problem (resulting in a *flaw*). Specifically, we will be looking at different methods of picking a split for a given flaw in an abstraction (called *refinement strategies*). These strategies are similar to the common heuristics, except that their evaluation function does not estimate the goal distance of a state, and instead assigns each split a value in \mathbb{R} depending on how desirable the split is, to then pick the best one during abstraction refinement. In addition to simply using abstractions as heuristic functions for the planner, we will evaluate our refinement strategies when generating abstractions from subtasks, as well as applied in saturated cost partitioning.

2

Background

Definition 2.1. State spaces

A state space or transition system $\mathcal{T} = \langle S, \mathcal{O}, cost, T, s_0, S_* \rangle$ is a tuple of a finite set of states S , a finite set of operators \mathcal{O} , a cost function $cost : \mathcal{O} \rightarrow \mathbb{R}_0^+$, a set of transitions $T \subseteq S \times \mathcal{O} \times S$, an initial state $s_0 \in S$ and a set of goal states $S_* \subseteq S$.

In this thesis, every state space has an associated set of variables \mathcal{V} and corresponding domains $dom(v)$ which its states are based on:

$$S = \{ \langle v_1 \rightarrow d_1, \dots, v_n \rightarrow d_n \rangle \mid \{v_1, \dots, v_n\} = \mathcal{V} \wedge d_i \in dom(v_i) \}$$

We only consider state spaces that are deterministic, that is for a given $\langle s, o \rangle \in S \times \mathcal{O}$ there exists at most one $\langle s, o, s' \rangle \in T$.

Definition 2.2. Variables, domains and facts

A variable is an atomic element used to build the states of a state space.

The domain $dom(v)$ of a variable v is the set of values this variable can be assigned.

A fact is a partial assignment $f = \{v \rightarrow d\}$ which assigns a value $d \in dom(v)$ to a single variable v .

Definition 2.3. States and cartesian states

A state is a total assignment over the variables of its state space. Concrete states are atomic and exhaustive, meaning an agent will always be in exactly one state and transitions will always be between two states (which need not be different).

A cartesian state is an abstract state whose variables can take a set of values:

$$s \in \mathcal{P}(dom(v_1)) \times \dots \times \mathcal{P}(dom(v_n))$$

Where $\{v_1, \dots, v_n\} = \mathcal{V}$ are the variables of the concrete planning problem and \mathcal{P} denotes the power-set. It is possible for a cartesian state's domain set to only contain a single value per variable (but no less).

We denote $[s]$ the abstract state which contains the concrete state s . Every concrete state is always contained in exactly one abstract state, namely $s \in [s] \forall s \in S$. It is valid for an abstract state to only contain a single concrete state.

Definition 2.4. Operators and paths

An operator or action is an abstract representation of a transition. It has an associated set of preconditions and effects which define the transitions induced by the operator. An operator must induce at least one transition (it's preconditions must not be contradictory). Every operator is assigned a cost by the state space's cost function.

A path is a tuple of operators leading from one state into another:

$$\pi = \langle s_1 \xrightarrow{o_1} s_2, \dots, s_{n-1} \xrightarrow{o_{n-1}} s_n \rangle$$

A path is associated with a cost based on the sequence of operators it applies:

$$\text{cost}(\pi) = \sum_{i=1}^{n-1} \text{cost}(o_i)$$

We call a path π a solution of a planning task iff it starts in the task's initial state $s_1 = s_0$ and ends in any of its goal states $s_n \in S_*$. Furthermore, it is called optimal if there is no solution π' with a lower cost $\text{cost}(\pi') < \text{cost}(\pi)$.

Definition 2.5. Landmarks

A fact landmark $L = \{f_1, \dots\}$ is a non-empty set of facts of a state space. At least one $f \in L$ must be reached (but unlike goal facts need not be retained) in every solution of the state space at some point. While in SAS+ every goal fact constitutes a landmark we ignore those for simplicity.

An operator landmark $L = \{o_1, \dots\}$ is a non-empty set of operators of a state space. At least one $o \in L$ must be applied at some point in every solution of the state space.

2.1 Counterexample-guided abstraction refinement (CEGAR)

The CEGAR-algorithm works as follows:

1. Initialize the abstraction by creating a single abstract state which contains the entire state-space: $S_1^A = \{\text{dom}(v_1) \times \dots \times \text{dom}(v_n)\}$
2. Compute an optimal path from the abstract initial state to an abstract goal state.
3. If no path was found (implying that the task is unsolvable), abort and use the current abstraction to compute heuristic values.
4. Find a flaw explaining why this path is not a solution of the concrete search problem.
5. Find a split of an abstract state which prevents this flaw from happening in later iterations. Some flaws imply multiple splits we have to pick from.
6. If the abstraction should be refined further, continue at step 2.

To refine the abstraction optimally we use a refinement strategy (also called splitting strategy) which evaluates a split in the context of an abstraction and assigns it a value in \mathbb{R} . We then pick the split with the highest refinement value according to our strategy.

The resulting abstraction is then used as a heuristic for the planning problem by computing the goal distances in the abstraction and assigning it to every state s as $h^A(s) = h^*([s])$.

2.1.1 Building the abstraction on subtasks

The simplest application of CEGAR only requires solving a single task: reaching a goal state. Because we are solving a planning problem in SAS+ format, this means reaching a partial assignment α . Sometimes, this single task is not enough to efficiently guide abstraction refinement. To solve this issue, we can introduce subtasks that the abstractions are built on, without affecting the eventual solving of the planning task (we still receive a single admissible heuristic). This means that we now have to generate one abstraction per subtask, which are eventually combined using saturated cost partitioning (SCP). However, we only generate exactly one abstraction per subtask, which results in a single-order SCP (which cannot profit from diversification beyond the combining of subtasks).

In our case, we replace the original task with two subtasks:

- A subtask for every goal fact. This is important so our abstraction keeps targeting the goal, but allows achieving every goal fact on it's own. Ideally this helps to detect alternate paths and allows the abstraction to only focus on a few of them.
- A subtask for every landmark we have to pass to get to the goal. For simplicity, we only use fact landmarks.

2.1.2 Saturated cost partitioning

The third experiment uses saturated cost partitioning (SCP), a method of combining multiple abstractions into a single, admissible heuristic. This allows capturing multiple aspects of a problem by varying the tasks the abstractions are generated with.

For SCP, we generate a number of abstractions (as before with CEGAR) while manipulating the costs of operators, so that the sum of a number of heuristic estimates (which we get from multiple abstractions) remains additive. We employ subtasks for this once more, but generate multiple orders of abstractions. Normally, SCP uses a fixed number of sub-heuristics, which it generates by distributing a cost function onto the different runs, but we instead keep generating abstractions until a resource limit is reached (200 seconds in this case).

The important difference, why this yields better results than the previous methods, is because it generates multiple different orders of the same abstractions. There are many ways to achieve this, for example:

- Randomly reordering variables or the items in their domain. This does not change the planning task itself, as both the variables \mathcal{V} and their domains $\text{dom}(v)$ are (unordered) sets. However, most planners (particularly the Fast Downward planner used in this thesis) represent those as lists and then prefer variables or values of lower index to arrive at a deterministic solution.
- The CEGAR-algorithm allows certain choices, such as which split to resolve, to be made arbitrarily. We do not apply this in our thesis, instead we generate abstractions using a single splitting strategy only. This is to allow comparing different strategies over all three experiments (single task, single SCP order over subtasks, multiple SCP orders over subtasks).

- We can again vary the subtasks that the abstraction is guided to represent. While before, it was important to keep all goal facts (just splitting them into subtasks) we can now generate an abstraction whose only purpose is to reach a single such goal. By generating an abstraction for every goal and combining them within SCP, we can once again reach a heuristic which is aware of all goals of the original planning task.

2.2 Refinement strategies of the Fast Downward planner

The Fast Downward implementation this thesis is based on already contains three refinement strategies. It also supports randomly picking a split. It should be noted that this is not the same as building a random abstraction, it is still guided by resolving the flaws between the initial state and the goal states.

In the implementation, to evaluate a split we give 3 arguments: The abstraction \mathcal{T} , the abstract state s_δ which is being split, and the split δ which is to be evaluated (consisting of the variable v at which the state is split and the wanted values $w \subset \text{dom}(v, s_\delta)$ which are split off). The value for s_δ is the same for all splits, and is obtained from the flaw $\langle s, c \rangle$ as $s_\delta = [s]$. Each flaw may produce multiple splits, one for each variable, which can be obtained by capturing all variables and their assignments from $[s] \cap c = s_\delta \cap c$. It is impossible for a flaw to generate no splits, as that would imply $[s] \cap c = \emptyset$, contradicting the existence of a flaw at this point.

The formulae in this thesis then use modified versions of REFINE and SPLIT (based on the definitions by Seipp and Helmert [4]) which, instead of the flaw itself, take the s_δ and δ resulting from this flaw (thus expressing the choice which would normally be part of the splitting process). This allows a refinement strategy to judge a split based on the abstraction it causes.

Further, all strategies except RANDOM come in two variants: MIN and MAX. Most are functionally the same, except for (as the name implies) seeking a maximal or minimal value of their base evaluation function. In this thesis, we name the MAX-variant the default, as it is the one which seeks to maximize the quality which the strategy is named after. The two refinement functions available to the planner (for a hypothetical X-strategy) then become $r_{\text{MAX},X} = r_X$; $r_{\text{MIN},X} = -r_X$.

2.2.1 The UNWANTED-strategies

These refinement strategies evaluate a split based on the number of unwanted values it implies. More specifically, if there was a flaw $\langle s, c \rangle$ in the abstraction, the set of wanted values w for a variable v becomes $w = \text{dom}(v, c) \cap \text{dom}(v, s)$ (all values from the domain of v which are in both $[s]$ and in the desired state c) with there only being a split for v if $s[v] \notin c[v]$.

The set of unwanted values is then $\bar{w} = \text{dom}(v, s_\delta) \setminus w$ and the evaluation function is:

$$r_{\text{UNWANTED}}(\delta = \langle v, w \rangle) = |\text{dom}(v, s_\delta)| - |w|$$

The strategy ignores the state which is to be split.

2.2.2 The REFINED-strategies

The REFINED-strategies are the ones originally proposed by Seipp and Helmert [4] (more specifically, the MAX-variant). They assign each split a value corresponding to the degree of refinement of the variable being split on (that is, how much of its domain was split off in the current abstract state):

$$r_{\text{REFINED}}(s_\delta, \delta = \langle v, w \rangle) = -\frac{|\text{dom}(v, s_\delta)|}{|\text{dom}(v)|}$$

The strategy ignores the wanted values which are being split off.

2.2.3 The HADD-strategies

Based on h^{add} by Bonet and Geffner [1], these strategies pick the split which contains a fact with the highest or lowest h^{add} value in the abstract planning graph, calculated from the initial state $[s_0]$.

This strategy is the only one where the MIN-variant and the MAX-variant differ functionally. In this case the evaluation functions are:

$$r_{\text{MIN.HADD}}(s_\delta, \delta = \langle v, w \rangle) = -\min_{f \in w} h^{\text{add}}([s_0], \langle v, f \rangle)$$

$$r_{\text{MAX.HADD}}(s_\delta, \delta = \langle v, w \rangle) = \max_{f \in w} h^{\text{add}}([s_0], \langle v, f \rangle)$$

Where $h^{\text{add}}(s, \langle v, f \rangle)$ is similar to the basic function $h^{\text{add}}(s)$ except for two major differences:

- The value assignments are reversed. Instead of marking the cost of s with zero and returning the value of the goal state (as h^{add} would) we mark the goal state with zero.
- We do not return the h^{add} value of the initial or goal state and instead that of the planning fact $\alpha = \{v \rightarrow f\}$.

3

New refinement strategies

In each case, the split selector selects the split with the highest refinement value. This works fine for the MAX-variant, and the MIN-variant simply negates the output of the evaluation function to arrive at the split with the lowest refinement value. In the case that multiple splits have the same refinement value, we utilize random tie-breaking by first shuffling the list of possible splits.

Most of the refinement strategies below require the following parameters (which are the same as those of the existing refinement strategies outlined in the previous chapter), terms and functions:

1. The parameters, consisting of the state at which the flaw occurred s_δ , the split being evaluated δ and the current, flawed transition system

$$\langle S, \mathcal{O}, \text{cost}, T, [s_0], S_* \rangle = \mathcal{T}$$

2. The transition system implied by the parameters:

$$\langle S', \mathcal{O}, \text{cost}, T', [s_0], S_* \rangle = \mathcal{T}' = \text{REFINE}(s_\delta, \delta)$$

The operators and cost function remain unchanged from \mathcal{T} .

3. The perfect heuristic $h_{\mathcal{T}}^*(s)$ to determine the goal distance of a state s in the transition system \mathcal{T} (in the implementation, this value is calculated for all states, generating additional overhead).

3.1 The CG-strategies

The CG-strategies pick a split based on the variable which is first or last in the variable ordering Fast Downward generates based on the causal graph of the planning task. Because each variable can only induce a single split for a given flaw, this refinement strategy does not depend on tie-breaking. Additionally, because the causal graph is generated by the planner beforehand, each variable's index is already known when applying the evaluation function (making it very fast). The goal of this strategy is to prioritize variables which appear early or late in the causal graph, with the hope that they allow the planner to solve the starting or finishing part of the task more effectively.

3.2 The GOAL_DIST-strategies

The GOAL_DIST-strategies compute the average goal distance of all solvable abstract states. The algorithm first obtains the transition system induced by the split, to then find all the states which can reach the goal and compute the refinement value as the average over those states' goal distances:

$$R = \{s \in S \mid h_{\mathcal{T}'}^*(s) < \infty\}$$

$$h_{\text{GOAL_DIST}} = \frac{\sum_{s \in R} h_{\mathcal{T}'}^*(s)}{|R|}$$

The algorithm completely ignores all abstract states which cannot reach the goal to avoid degenerate cases which depend entirely on tie-breaking. This value is always defined, because the concrete planning task (and thus every Cartesian abstraction) must have at least one goal state.

The goal of this strategy is to create an abstraction where most states have a high goal distance, and assuming that this results in a heuristic which is most informed. It is easy to see why an abstraction with the least possible average goal distance of zero can only be achieved if every abstract state contains at least one concrete goal state, and is reasonable to assume that the inverse will then result in a good abstraction. Because the algorithm does not consider states which cannot reach the goal, it is possible for the average goal distance to become smaller, even though this can never be the case for any single abstract state.

3.3 The HIGHER_DIST-strategies

The HIGHER_DIST-strategies count the abstract states which have a higher goal distance after a split. Again, the algorithm obtains the transition system induced by the split, to then directly count the states whose goal distance increases:

$$h_{\text{HIGHER_DIST}} = |\{s \in S \mid h_{\mathcal{T}'}^*(s) > h_{\mathcal{T}}^*(S(s))\}|$$

Where $S(s)$, given $s \in S'$, is the state's equivalent in \mathcal{T} :

$$S' = (S \setminus s_\delta) \cup \text{SPLIT}(s_\delta, \delta) \Rightarrow S(s \in S') = \begin{cases} s_\delta & \text{if } s \in \text{SPLIT}(s_\delta, \delta) \\ s & \text{otherwise} \end{cases}$$

The abstract state which gets split is considered twice, once for each state it splits into. However, it is impossible for both resulting states to have a goal distance different from the initial abstract state.

It should be noted that, while similar to the above strategy, MAX_HIGHER_DIST prefers splits which increase the goal distance of many nodes slightly (such as landmarks close to the goal) as opposed to increasing the average (which yields a diminishing effect as the abstraction grows).

3.4 The ACTIVE_OPS-strategies

The ACTIVE_OPS-strategies count the operators which are active in the abstraction resulting from a split. An operator is considered active if it induces at least one transition between

two abstract states, instead of only inducing self-loops. Again, we obtain the induced transition system, and count the active operators from the transitions:

$$h_{\text{ACTIVE_OPS}} = |\{o \in \mathcal{O} \mid \exists (s, o, s') \in T \wedge s \neq s'\}|$$

The MAX-variant of this strategy attempts to create a most diverse abstraction without requiring the use of subtasks and SCP. It's inverse attempts to keep the abstraction narrow, which partially contradicts the purpose of abstraction refinement. As the refinement continues, it becomes increasingly less likely for a split to activate any operators. Additionally, it is not possible for operators to become inactive during refinement, meaning that this refinement strategy will degenerate to tie-breaking as refinement advances. The refinement value cannot decrease after a split, but it can increase arbitrarily fast because multiple (or all) operators can induce a transition between two given states.

3.5 Predicting the best refinement strategy

To obtain the most suited abstraction for a problem, the ideal refinement strategy must be chosen in advance based on certain characteristics of the planning problem. Because Fast Downward converts a PDDL-problem into SAS+ before solving it, we only have access to characteristics of the converted problem. Specifically, we attempt to predict the number of expansions caused by an abstraction generated from a refinement strategy or how many problems of a domain it solves best using the number of operators, variables and facts in the planning problem.

We attempt such a prediction in the following ways:

1. The first method attempts to find a linear correlation via matrix operations.

$$\text{Expansions} = \text{Parameters} * M$$

$$M = \text{Parameters}^{-1} * \text{Expansions}$$

Where `Expansions` and `Parameters` are each a matrix of sample rows.

We solve this equation using the singular value decomposition of `Parameters` to construct its pseudo-inverse. With the resulting matrix `M` we predict the number of expansions of each strategy, selecting the one which has the lowest result. Because there may be multiple strategies which result in the least number of expansions, we measure performance of the algorithm only in how many expansions it causes, and not how many it predicts or the exact error of this prediction.

2. The second method uses a multivariate normal distribution (MVND) for every refinement strategy, which should model the domain of problems in which it performs best. The parameters are again the variables of the problem, and the MVND is conditioned to include all problems where the corresponding strategy has achieved the best results (the least number of expansions). We then predict the probability that a problem with a given set of parameters is solved with the least number of expansions by a specific strategy:

$$P(p \mid r) = \frac{1}{(2\pi)^k * \det C} \exp\left(-\frac{1}{2}(x - \mu)^T C^{-1}(x - \mu)\right)$$

The parameters are calculated assuming optimality for different strategies is independent, then applying maximum-likelihood estimation [3]:

$$\mu = \frac{1}{n} \sum_{i=1}^n \vec{x}_i; C = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^T (x_i - \mu)$$

Where \vec{x}_i is the i -th parameter sample expressed as a row vector. Then the posterior probability is used to pick the best refinement strategy:

$$\arg \max_r P(r | p) = \arg \max_r P(p | r) * P(r)$$

Again, we do not make use of the true predicted value or its error and measure only the outcome of our predictor function.

3. An additional method which is used as a baseline for the performance of the other methods is to pick the strategy which causes the least number of total expansions:

$$\text{BestGuess} = \arg \min_r \left\{ \sum_{p \in P} \text{Expansions}(p, r) \right\}$$

Where $\text{Expansions}(p, r)$ is the expansions measured in problem p of the set of all problems tested P when using strategy r to build the abstraction. In other words, the best guess is the strategy which, if used exclusively, would have resulted in the least amount of expansions.

We also compare these numbers to the least or most possible expansions, if we had a perfect predictor:

$$\text{Best} = \sum_{p \in P} \min_r \text{Expansions}(p, r)$$

$$\text{Worst} = \sum_{p \in P} \max_r \text{Expansions}(p, r)$$

Like the best guess, these values are not known before solving the problem with all strategies, thus serving as evaluation context only.

We conduct this analysis over the entire set of problems by splitting off a random 10% into a test set and exclusively using the remaining 90% as training data. While this introduces some randomness into the results, it outweighs the issues of testing the algorithm on the training set and avoids accidentally training on test data after a number of iterations.

4

Results

To evaluate the refinement strategies we have used two indicators of search performance:

1. The number of expansions before the last jump of the heuristic value during planning. This criterion is similar to the total number of expansions, but does not depend on the tie-breaking strategy of A*. The resources used to find the optimal path typically correspond linearly to this value.
2. The amount of time necessary to create the abstraction. Because the abstraction is only created once, before solving the problem, the former criterion is more important. In most cases, abstraction refinement will not terminate early because no concrete solution (or lack thereof) was found. Instead, the time to construct the abstraction strongly depends on the resource limits in place.

4.1 Running the planner

The evaluation of our refinement strategies is performed by running the Fast Downward planner [2] on a set of various planning problems[6] in PDDL format. The invocation of the planner and the handling of most parameters, as well as the later parsing and processing of the results is performed using Downward Lab [5].

To run the planner, we supply two additional arguments to Lab:

1. The search function, which differs by experiment.

Basic search function:

```
astar(cegar(subtasks=[original()], max_states=10K,  
            max_transitions=infinity, max_time=infinity, pick=<pick>))
```

This invocation results in generating a single abstraction based on the original task. Refinement will only stop when either a concrete solution (or lack thereof) was found or the abstraction has reached 10'000 states.

Search function using subtasks:

```
astar(cegar(subtasks=[landmarks(random_seed=0),  
                    goals(random_seed=0)], max_states=10K,  
            max_transitions=infinity, max_time=infinity, pick=<pick>))
```


Search function using saturated cost partitioning:

```
astar(saturated_cost_partitioning(
    [cartesian([landmarks(order=random, random_seed=0),
                goals(order=random, random_seed=0)], pick=<pick>)],
    max_time=200, max_orders=infinity,
    diversify=true, max_optimization_time=0))
```

2. The algorithm used in the search function (denoted by the 'pick' parameter in the above formulae) which is changed to ensure every refinement strategy is tested on every problem.

One of: [RANDOM, MIN_UNWANTED, MAX_UNWANTED, MIN_REFINED,
 MAX_REFINED, MIN_HADD, MAX_HADD, MIN_CG, MAX_CG,
 MIN_GOAL_DIST, MAX_GOAL_DIST, MIN_HIGHER_DIST,
 MAX_HIGHER_DIST, MIN_ACTIVE_OPS, MAX_ACTIVE_OPS]

4.2 Original task

First we examine the time used by the strategies to generate abstractions. Because the GOAL_DIST, HIGHER_DIST and ACTIVE_OPS strategies require a rewiring of the abstraction per split evaluation, they are noticeably slower than all other strategies (Figure 4.1). The remaining strategies show little difference as the time required by the refinement process overshadows the time to pick a split. The noticeable variance is because, for simple problems, one strategy sometimes completes refinement before reaching a state or transition limit. An additional source is the varying number of average splits during refinement (the planner does not invoke the refinement function if there is only one split to pick).

Next we compare the MAX-variants of our refinement strategies to the MIN-variants. In Figure 4.2 we can see that the existing strategies' MAX-variant generally performs better than the inverse. For the new strategies (Figure 4.3) we get a mostly similar picture, with the performance difference between MAX.CG and MIN.CG being only very small (in spite of them preferring opposite splits), and for the ACTIVE_OPS strategies we see the inverse, where the MIN-variant performs significantly better. This is likely because refinement forces operators to become active, while the MIN-variant avoids the degenerate case when all operators are active.

In Figure 4.4 we, again, compare a specific refinement strategy to RANDOM, this time considering expansions. Unfortunately, none of the new strategies managed to perform better than the previous best strategy, MAX.HADD. We can also observe that the strategies which performed badly have a more narrow distribution, indicating that they might be constrained in some way. Another interesting observation is that the strategies performing well do so more likely for difficult problems (closer to the right) while the strategies performing badly appear to have a higher tendency to do so for easy problems (closer to the left).

Finally, we put the performance of all algorithms into perspective in Figure 4.5. It can be seen how MAX.HADD is the best strategy, immediately followed by MAX.CG and MIN.CG. After a small gap we find the other strategies by Seipp and Helmert [4] (MAX_REFINED and MAX_UNWANTED), followed by the remaining strategies. We can also observe a trend in most

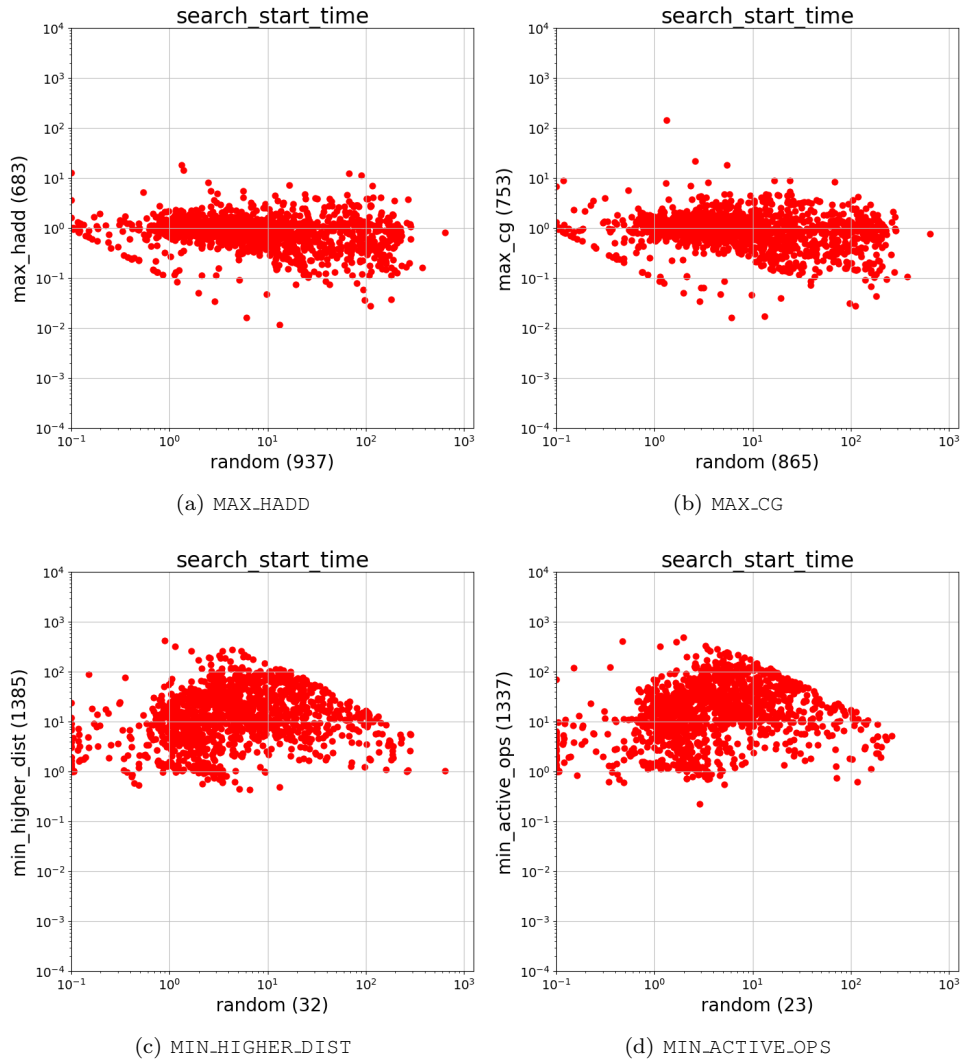


Figure 4.1: Scatter plots comparing the time to construct the abstraction for some strategies, compared to RANDOM. For most abstractions, the split selection requires minimal effort, as can be seen in the top row. The plots for variants of the same strategy vary little as they require the same effort.

strategies, where the MAX-variant is better, except for MAX_CG (which is only very slightly better than MIN_CG) and MAX_ACTIVE_OPS, which is considerably worse than its inverse.

4.2.1 Number of distinct splits

Next, to measure the impact of our refinement strategy we also analyse 2 additional parameters:

1. `options`: The average number of possible splits to choose from. This value indicates the freedom of the evaluation function to pick a split. One might expect this value to not depend on the refinement strategy because it is already fixed before the strategy decides which split to use. However, empirically this value varies based on the strategy,

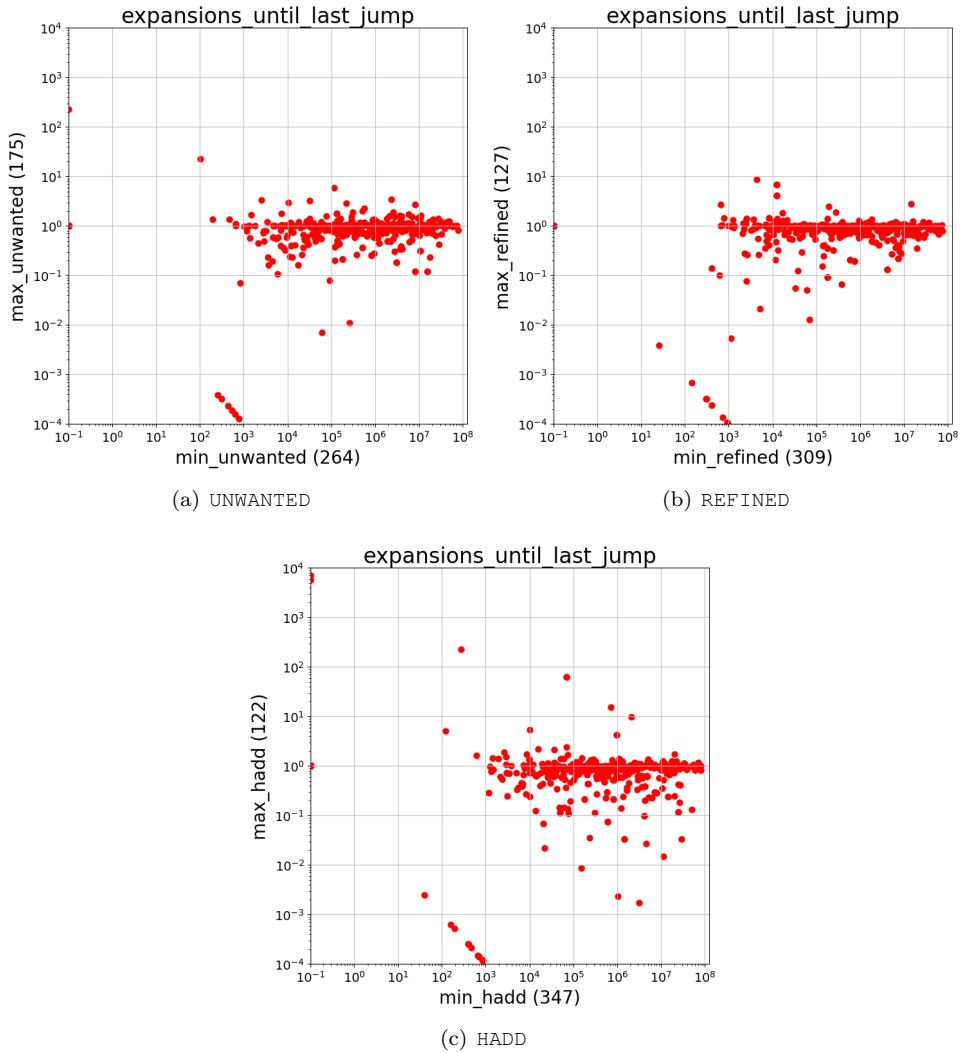


Figure 4.2: Comparison between variants of the existing refinement strategies. Points below 1 indicate that the MAX-variant is better. We can see that this is the case for all strategies, especially so for HADD.

though not as much as the number of distinctly valued splits.

2. *distinct*: The average number of distinct splits (distinct in the sense that the evaluation function assigns them different values). This indicates how much the split selection depends on the tie-breaking strategy. If the value reaches one, this means that the strategy is mostly random (the tie-breaking strategy used in all experiments).

To simplify calculations, the implementation reports these attributes, which it measures once for every flaw found, via the arithmetic mean. A more accurate measure of the entropy of the decision would be given by the geometric mean, as it more closely represents the number of possible abstractions that could have been constructed assuming no two distinct sequences of splits result in the same abstraction.

Table 4.1 shows the means and standard deviations over all problems, for each refinement

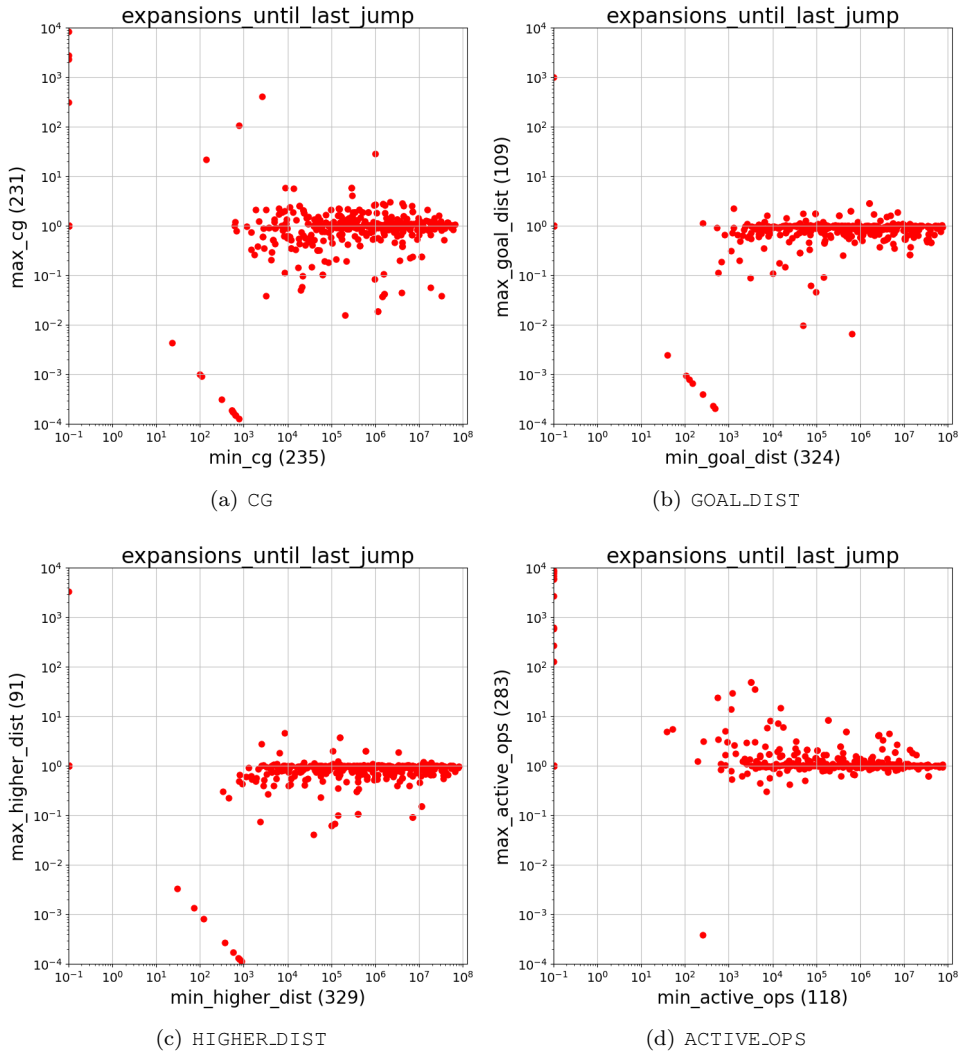


Figure 4.3: Comparison between variants of the existing refinement strategies. Points below 1 indicate that the MAX-variant is better. Unlike the existing strategies, both CG-variants perform similarly well while for ACTIVE_OPS the MIN-variant is better.

strategy and measure. While they do vary by strategy, they vary little (except for occasional outliers) by problem domain. Ideally the values on the left should match those on the right for the refinement strategy to have the most influence. Note that this is always the case for RANDOM, MAX_CG and MIN_CG as they are unable to value different splits equally. The last four strategies instead show the opposite, indicating that they degenerate to RANDOM because of tie-breaking.

It is interesting to see how MAX_HADD achieves less freedom than MAX_CG (and certainly less distinct results) yet performs better. A possible explanation for this is that the splits, which cause the abstraction to ultimately be more useful, are made only during a short period of refinement, while the average number of distinct values is low at other times.

In Figure 4.6 we can see the histogram of both attributes. The distribution appears linear

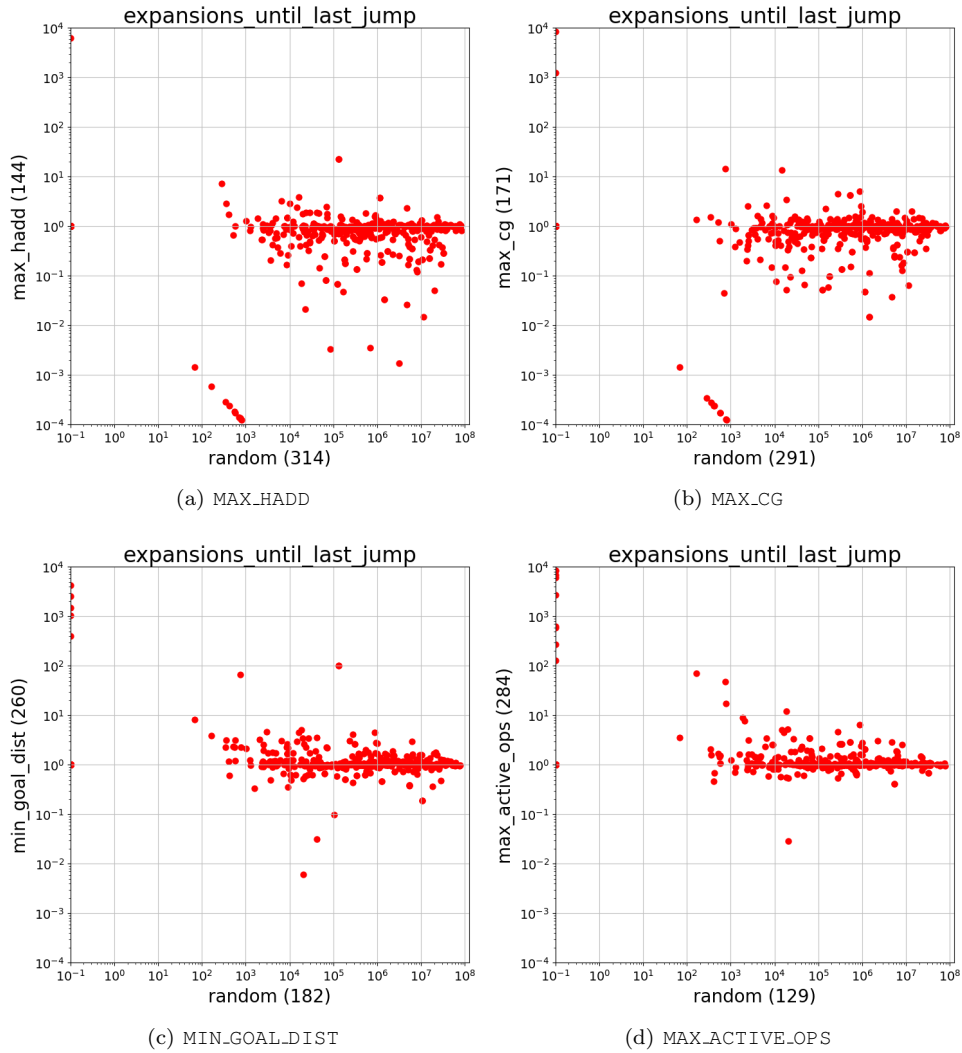


Figure 4.4: Scatter plots comparing the expansions for a selection of refinement strategies performing best (top) and worst (bottom), compared to RANDOM. Values below 1 indicate that the strategy is better than RANDOM.

in a logarithmic plot, starting at $x = 1$. The exponential distribution is:

$$p(x) = \begin{cases} \lambda e^{-\lambda(x-1)} & \text{if } x \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

Where we can obtain λ given μ from Table 4.1 via $\lambda = \frac{1}{\mu - 1}$.

However, although the general trend follows an exponential distribution, the distributions obtained from only a single refinement strategy each sometimes do not. Especially the distribution of the number of split options available tends to peak between 1.05 and 1.15, but still decays linearly in log-space.

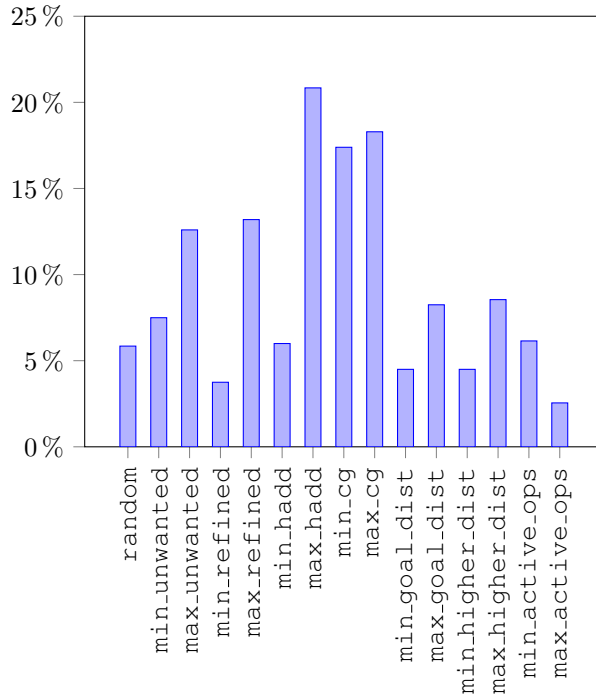


Figure 4.5: Percentage of optimal results (by refinement strategy) for the original task. In most cases, the MAX-variant is significantly better than the inverse, which matches the observations in Figure 4.2 and Figure 4.3.

4.2.2 Progress of average goal distance

To gain more insight into the progress of the average goal distance during abstraction refinement, specifically for the GOAL_DIST-strategy, we track the progress of the average goal distance during refinement. As expected, the average goal distance follows a logarithmic curve, stagnating near the end. We get this same picture, even when examining problems

	μ_{distinct}	σ_{distinct}	μ_{options}	σ_{options}
RANDOM	1.18	0.167	1.18	0.167
MIN_UNWANTED	1.10	0.119	1.18	0.180
MAX_UNWANTED	1.12	0.154	1.21	0.238
MIN_REFINED	1.11	0.143	1.20	0.188
MAX_REFINED	1.10	0.148	1.20	0.198
MIN_HADD	1.14	0.171	1.21	0.249
MAX_HADD	1.14	0.150	1.22	0.215
MIN_CG	1.19	0.191	1.19	0.191
MAX_CG	1.24	0.256	1.24	0.256
MIN_GOAL_DIST	1.06	0.059	1.19	0.177
MAX_GOAL_DIST	1.06	0.058	1.18	0.160
MIN_HIGHER_DIST	1.02	0.032	1.19	0.176
MAX_HIGHER_DIST	1.02	0.021	1.18	0.163
MIN_ACTIVE_OPS	1.00	0.022	1.18	0.168
MAX_ACTIVE_OPS	1.01	0.069	1.18	0.169

Table 4.1: Statistics on the number of distinctly rated splits and total available splits to pick from, showing the achieved freedom of the splitting strategies in the original task.

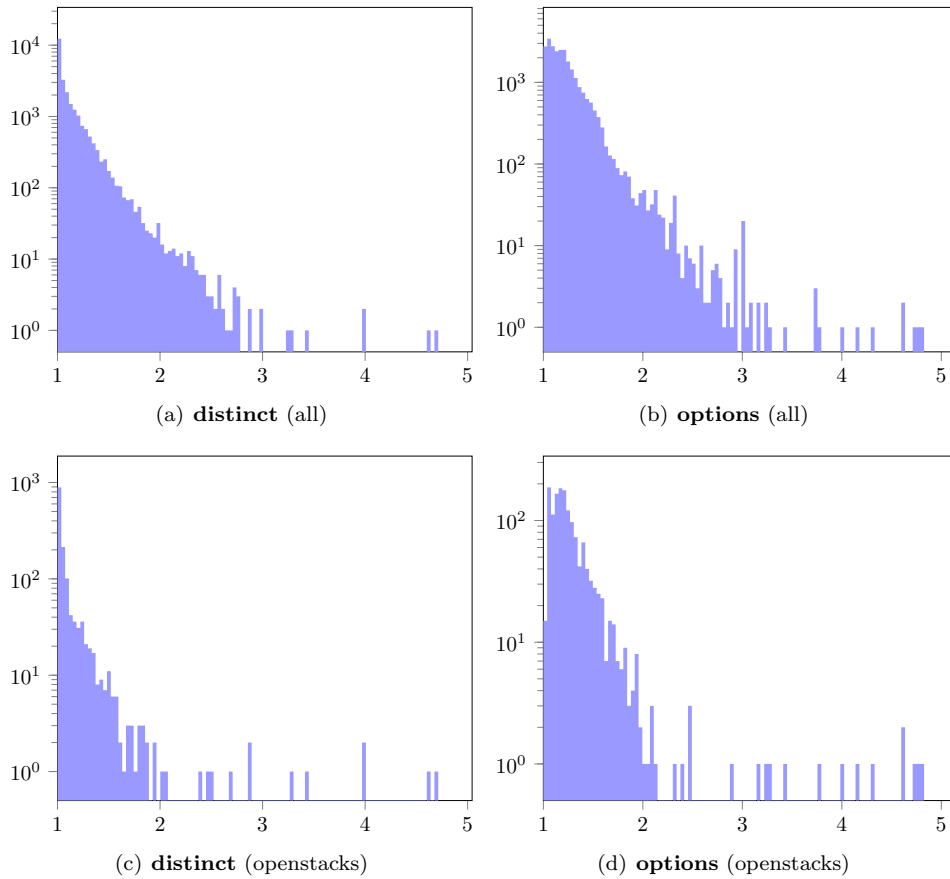


Figure 4.6: Histograms showing the distribution of the number of distinct and available splits to choose from each refinement. As the vertical scale is logarithmic, the distribution can be recognized as exponential in $\mathbb{D} = [1, \infty[$.

that are solved badly by an abstraction built using this refinement strategy. Surprisingly enough, the same can be observed for `MAX_GOAL_DIST` as well as `MIN_GOAL_DIST`. Figure 4.7 shows the progress of the refinement value for the two tested strategies, selecting two problems from different domains in which the strategy was best or worst. This does not necessarily imply that these problems weren't solved well or badly (respectively) by any other strategy.

4.3 Including subtasks

In Figure 4.8 we compare the expansions caused by the strategies (relative to `RANDOM`) when using subtasks. We can see that the distribution of the comparison has become significantly wider. This correlates with an increase in the degree of freedom seen from Table 4.1 to Table 4.2, which means that the impact of the refinement strategy has increased. The results also start off lower, corresponding to a general increase in search performance resulting from the use of subtasks and SCP. As before, the plots imply that more difficult problems are more likely to show improvement compared to using the `RANDOM` refinement strategy, but simpler problems are not noticeably more likely to show deterioration.

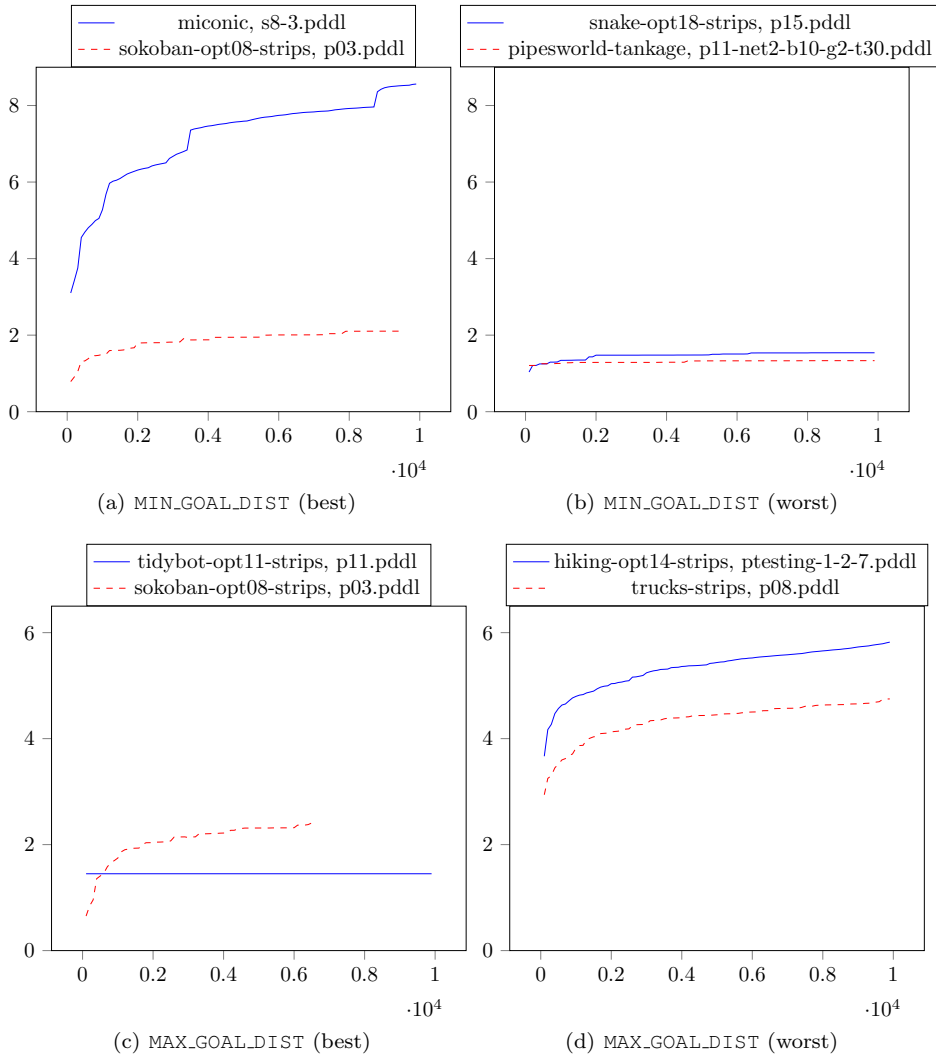


Figure 4.7: Plot showing the progress of average goal distance during refinement using GOAL_DIST, showing little correlation to the overall performance of the strategy. Instead, it appears that achieving the desired (least or highest) average goal distance results in a worse performance. Each plot shows the average goal distance for two problems which were solved best (or worst) according to expansions.

Figure 4.9 shows the fraction of problems which were optimally solved by each refinement strategy. When compared to the original task (rescaled so that RANDOM yields the same results for both experiments) we notice a sharp decline in the performance of the previous best strategies MAX_HADD, MIN_CG and MAX_CG. There is an additional, small decline for the MAX_UNWANTED-strategy which is compensated for by an increase of the MAX_REFINED-strategy (making it the new best refinement strategy by a fair margin). Incidentally, this is also the strategy which is outlined in the paper[4] (Section 4.3.2).

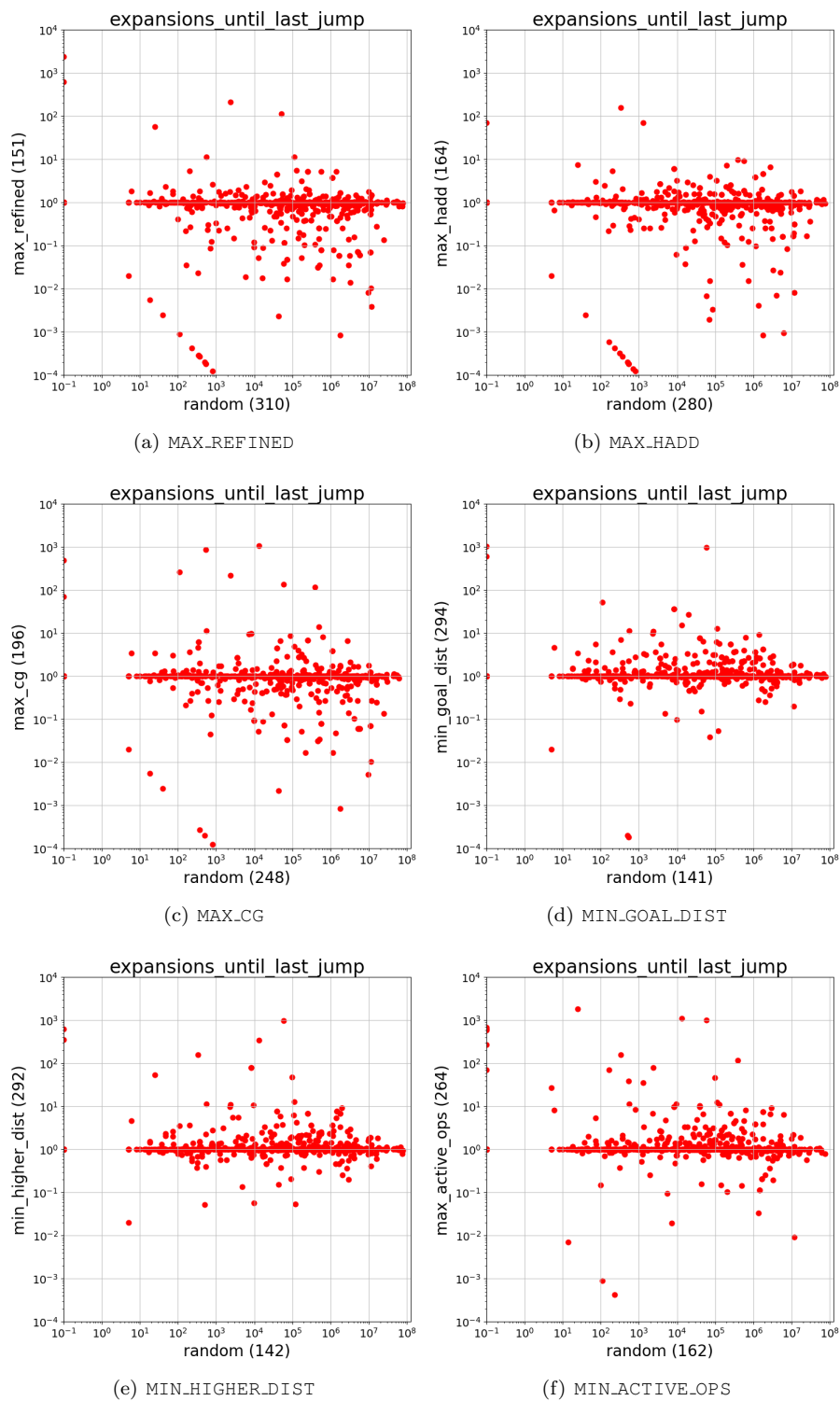


Figure 4.8: Scatter plots comparing the expansions for a selection of refinement strategies, compared to RANDOM, when using subtasks. Values below 1 indicate that the strategy is better than RANDOM.

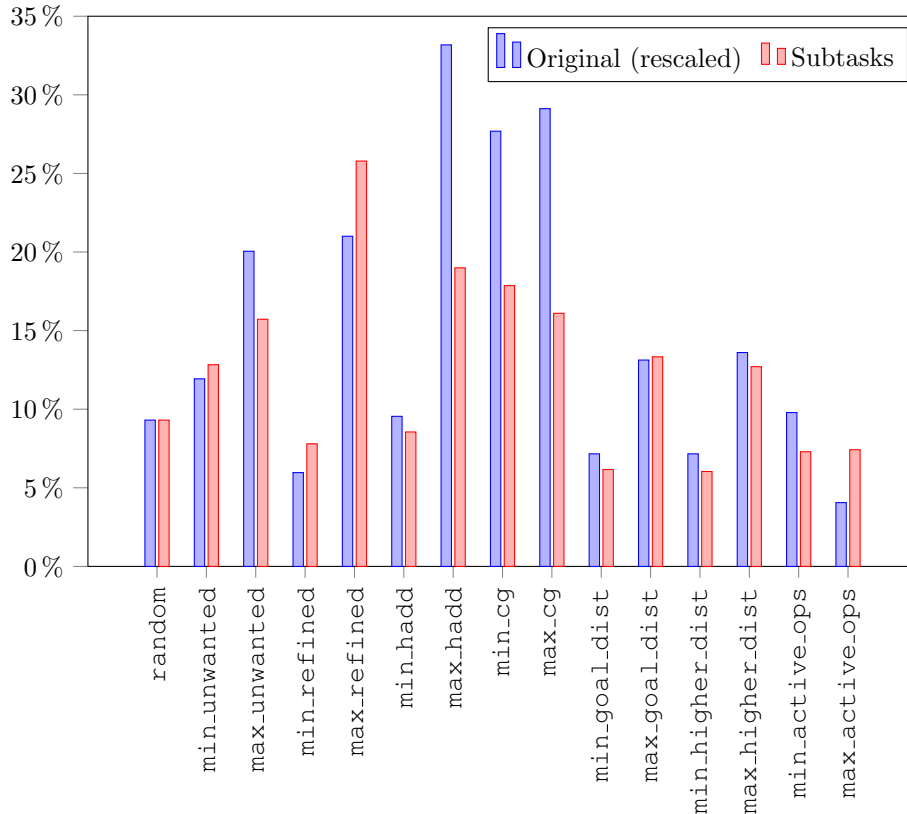


Figure 4.9: Percentage of optimal results for subtasks compared to the original task (the original results are rescaled to have the same value for RANDOM). We notice a sharp decline in the performance of the previous best strategies MAX_HADD and CG (both variants).

4.3.1 Number of distinct splits

The analysis results in Table 4.2, performed on the problems solved using subtasks shows an increase in the degree of freedom available to all algorithms, from roughly 1.2 to roughly 1.3, and a comparatively large increase in the standard deviation. Because there is always at least one split available, this implies that the chance for an unusually high degree of freedom in a single problem has increased.

At the same time, the number of distinct splits has risen by a small factor, except for MAX_ACTIVE_OPS which shows a large jump, placing it above most of the other new refinement strategies (GOAL_DIST, HIGHER_DIST and MIN_ACTIVE_OPS). When using subtasks, we generate one abstraction per subtask and reduce the size of each abstraction accordingly. If all operators are active, it is impossible for MAX_ACTIVE_OPS to produce distinct ratings for splits, because operators cannot become inactive again. This period of stagnation always appears at the end of refinement, and is thus cut off first when reducing the size of the abstraction. When combining this result with Figure 4.9, we again fail to see a correlation, except that the improvement of MAX_ACTIVE_OPS appears in both figures.

	μ_{distinct}	σ_{distinct}	μ_{options}	σ_{options}
RANDOM	1.30	0.520	1.30	0.520
MIN_UNWANTED	1.19	0.241	1.29	0.527
MAX_UNWANTED	1.21	0.287	1.33	0.546
MIN_REFINED	1.19	0.271	1.32	0.532
MAX_REFINED	1.16	0.236	1.30	0.531
MIN_HADD	1.21	0.261	1.30	0.526
MAX_HADD	1.23	0.298	1.35	0.573
MIN_CG	1.32	0.565	1.32	0.565
MAX_CG	1.33	0.546	1.33	0.546
MIN_GOAL_DIST	1.12	0.173	1.32	0.536
MAX_GOAL_DIST	1.09	0.129	1.28	0.514
MIN_HIGHER_DIST	1.05	0.095	1.31	0.528
MAX_HIGHER_DIST	1.02	0.047	1.28	0.513
MIN_ACTIVE_OPS	1.03	0.107	1.30	0.535
MAX_ACTIVE_OPS	1.16	0.262	1.31	0.531

Table 4.2: Statistics on the number of available splits and splits which are valued differently by a refinement strategy, showing the achieved freedom of the strategies when using subtasks.

4.4 Applying SCP

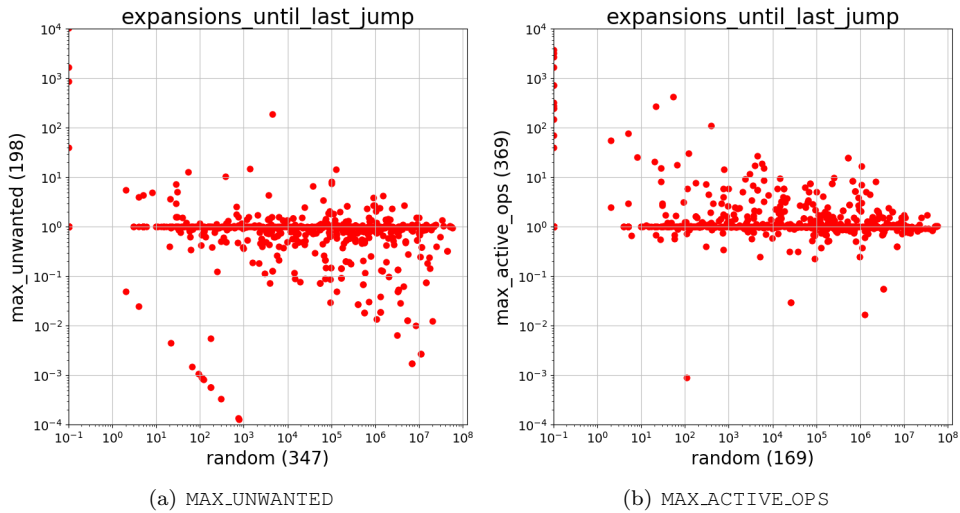


Figure 4.10: Scatter plots comparing the expansions for a selection of refinement strategies, compared to RANDOM, when using SCP. Values below 1 indicate that the strategy is better than RANDOM.

In the final experiment, involving SCP, we notice little difference in the distribution compared to when using subtasks (Figure 4.10). This makes sense, because we combine the subtasks in the same way, the change being that we generate different orders of abstractions. The tendency for difficult problems to be solved better than RANDOM (or simple problems being solved worse) has also increased again.

In general, we notice slight improvements to all refinement strategies compared to only using subtasks. In Figure 4.11 we can see that the MAX_UNWANTED-strategy now performs much

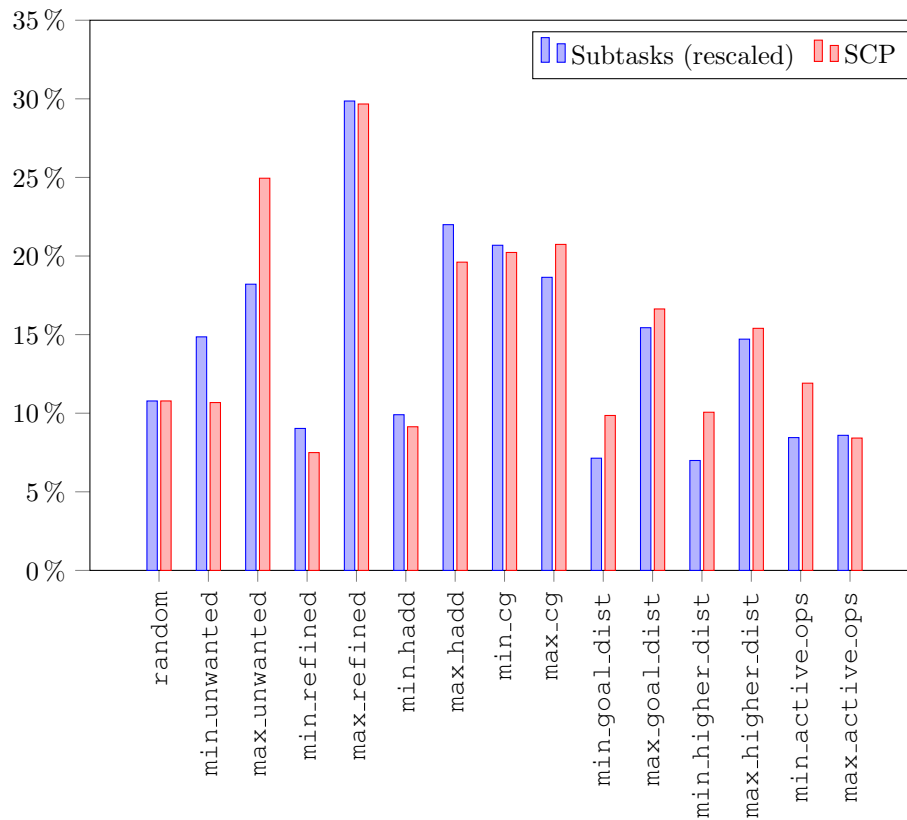


Figure 4.11: Percentage of optimal results for SCP compared to subtasks (the subtasks' results are rescaled to have the same value for RANDOM). The new strategies' MIN-variants see some improvements again.

better than before (with a corresponding but smaller drop in its inverse variant). While it too is now better than MAX_HADD and the CG-strategies, MAX_REFINED is still superior.

4.5 Predicting the best refinement strategy

	least exp.	most exp.	best strategy	MVND	linear
original task	46038868	131558133	64292474	70275225	116803152
	1	2.858	1.396	1.526	2.537
			1	1.093	1.817
subtasks	30570863	64358053	36005074	60324594	61799436
	1	2.105	1.178	1.973	2.022
			1	1.675	1.716
SCP	67968544	85726053	71164310	77393698	74462557
	1	1.261	1.047	1.139	1.096
			1	1.088	1.046

Table 4.3: Prediction accuracy for MVND and matrix approximation for all three experiments.

Unfortunately, prediction of the best refinement strategy a-priori was only moderately successful. Both prediction methods have a chance of outperforming the best guess (typically MAX_HADD) but cannot consistently do so. However, the MVND-predictor is frequently superior to the linear predictor. Table 4.3 shows the results for all problems, split by experiment. Note that these values depend on the random pick of the test set. This table was generated using a seed of zero. Additionally, to make the numbers more readable, every block of 3 lines contains the same data, but with different scaling.

5

Conclusions

As a part of this thesis we have extended the Fast Downward planner[2] with eight new refinement strategies. We have evaluated how well they perform compared to the existing implementations in three different circumstances (original tasks, with subtasks and saturated cost partitioning). Unfortunately we have not managed to create a refinement strategy which consistently performs better than the existing ones. We have attempted to find an explanation by examining the freedom of the decision process, and while it does not seem well suited to predict how often a strategy performs well, it might still be possible to enhance search behaviour of other strategies. Finally, we have attempted to use these results to predict which strategy should be employed to solve a problem. This was partially successful, showing a partial improvement over the "best guess" which is otherwise not known a-priori. We expect that these results can be further improved using more parameters or more advanced classifiers.

5.1 Future work

In addition to the refinement strategies examined in this thesis, there are some different approaches that could show improvement to both the new and old refinement strategies.

5.1.1 Tie-breaking strategies

A subject touched on in this thesis but which was not applied is a variation of tie-breaking strategies. The original implementation of Fast Downward simply picked the first split with the same lowest refinement value (which generally ends up having the lowest causal graph index) while we employed random picks among the set of best splits. More generally, it is possible to use multiple strategies, one which picks first and combined with a second one which breaks ties.

5.1.2 A strategy with the most freedom

We have seen that some strategies have a tendency to end up with an abstraction which does not allow them to make meaningful decisions any more, because they are only presented with

a single remaining split to "choose" from. Perhaps it is possible to produce a strategy which aims to maximize the number of splits it can pick from during refinement. Such a strategy might not directly produce good abstractions (as can be seen by the lack of correlation between optimality and achieved freedom), but when combined with others (perhaps as a tie-breaker itself) it might improve the generated abstraction.

5.1.3 Varying split selection strategy during SCP

One of the major advantages of SCP is the fact that it can combine multiple heuristics, each unsuitable to efficiently solve the problem, into a single heuristic which captures the dependencies between all its parts. This requires a diverse set of abstractions. Currently, we only use a single split selection strategy during a run of SCP. Ideally, we would be using multiple to get a most diverse abstraction. Selecting which refinement strategies to use, and how often compared to the others, is a big problem itself. A possible start is to use a predictor for the best split strategy, and then use strategies in fractions equal to the probability of them solving the problem optimally on their own.

5.1.4 More complex prediction features

To predict which split selection strategy yields the best abstraction we have only used three parameters: variable count, fact count (the number of possible partial assignments with only one variable) and operator count. While these may give an overview of the size of the state space, they offer little information about the connectivity of the state space. It is possible that, when using additional parameters, the accuracy of these predictions could be increased. Examination of how a strategy builds its abstraction, and consequently which features a planning problem requires to have an ideal abstraction generated, could provide a better first set of parameters.

Bibliography

- [1] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129 (1-2):5–33, 2001.
- [2] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.
- [3] Alvin C Rencher. *Methods of multivariate analysis*, volume 492. John Wiley & Sons, 2003.
- [4] Jendrik Seipp and Malte Helmert. Counterexample-guided Cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research*, 62:535–577, 2018.
- [5] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. <https://doi.org/10.5281/zenodo.790461>, 2017. URL <https://doi.org/10.5281/zenodo.790461>.
- [6] Various. STRIPS PDDL benchmarks from sequential optimization tracks of IPC 1998-2018, March 2019. URL <https://doi.org/10.5281/zenodo.2616479>.

Declaration on Scientific Integrity

Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor

Martin Zumsteg

Matriculation number — Matrikelnummer

2016-056-111

Title of work — Titel der Arbeit

Refinement Strategies for Counterexample-Guided Cartesian Abstraction Refinement

Type of work — Typ der Arbeit

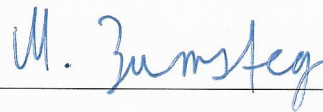
Bachelor's thesis

Declaration — Erklärung

I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, July 20, 2019



Signature — Unterschrift