



# **Combining Novelty-Guided and Bounded Suboptimal Search**

Bachelor Thesis

Natural Science Faculty of the University of Basel  
Department of Mathematics and Computer Science  
Artificial Intelligence  
<http://ai.cs.unibas.ch>

Examiner: Prof. Dr. Malte Helmert  
Supervisor: Dr. Gabriele Röger

Gian-Andrea Wetten  
[gian-andrea.wetten@stud.unibas.ch](mailto:gian-andrea.wetten@stud.unibas.ch)  
12-720-777

14.07.2017

## **Acknowledgments**

I would like to thank Dr. Gabriele Röger for the guidance and helpful advice. Furthermore I would like to thank Prof. Dr. Helmert for allowing me to write this thesis in his research group. Thanks to the sciCORE team as well for maintaining the infrastructure that we used for our experiments. Last but not least I would like to thank my friends and family for their support.

## **Abstract**

The notion of adding a form of exploration to guide a search has been proven to be an effective method of combating heuristical plateaus and improving the performance of greedy best-first search. The goal of this thesis is to take the same approach and introduce exploration in a bounded suboptimal search problem. Explicit estimation search (EES), established by Thayer and Ruml, consults potentially inadmissible information to determine the search order. Admissible heuristics are then used to guarantee the cost bound. In this work we replace the distance-to-go estimator used in EES with an approach based on the concept of novelty.

# Table of Contents

<b>Acknowledgments</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>3</b>
2.1 Planning . . . . .	3
2.2 Open lists . . . . .	4
2.3 Heuristics . . . . .	4
2.4 Best-First Search . . . . .	5
<b>3 Novelty-Guided Search</b>	<b>6</b>
<b>4 Bounded Suboptimal Search</b>	<b>9</b>
<b>5 Combination</b>	<b>11</b>
<b>6 Evaluation</b>	<b>14</b>
6.1 Setup . . . . .	14
6.2 Comparing Novelty Bounds . . . . .	14
6.3 Comparing Weights . . . . .	18
6.4 Alternatives . . . . .	20
<b>7 Conclusion and Future Work</b>	<b>23</b>
<b>Bibliography</b>	<b>24</b>
<b>Appendix A Appendix</b>	<b>26</b>
<b>Declaration on Scientific Integrity</b>	<b>27</b>

# 1

## Introduction

*Classical Planning* plays a very important role in the domain of AI problem solving. The objective is to find a sequence of actions to apply which move the agent from a given initial state to a goal state. In contrast to other AI topics the environment in classical planning is known beforehand and static, which means it does not change while we decide on an action. Additional properties include being fully observable as well as deterministic.

*Best-first search (BFS)* is a forward state space search algorithm that expands the most promising node in each iteration by applying a heuristic evaluation function  $f$ . The node  $n$  to expand is the one with the minimum function value  $f(n)$ . It is thereby customary to use the priority queue data structure for the open list which keeps track of the candidates for an expansion. One subtype of BFS is the greedy approach. Greedy BFS or GBFS is an algorithm that considers only the heuristic estimates  $f(n) = h(n)$ . Another well-known variant of BFS is  $A^*$  [1]. Its evaluation function  $f(n) = g(n) + h(n)$  sets it apart from GBFS as  $A^*$  relies on a combination of path cost and heuristic estimation. This guarantees us an optimal solution given that the heuristic used is admissible, which means that it does not overestimate. Computing this solution however is often a very costly endeavour with respect to both the time it takes to find a solution and to memory requirements. Here is where bounded suboptimality comes into play. Some algorithms, for example weighted  $A^*$ , compute a solution of a cost that is within an adjustable bound from the optimal solution cost. This leads to fewer expansions and therefore a faster search time. Most approaches for bounded suboptimal search have severe drawbacks which limit their performance and thus they often have not seen wide adoption in the field. Thayer and Ruml [2] have shown that by guiding the search with a potentially inadmissible heuristic while ensuring that the costs are within the suboptimality bound with an admissible heuristic can lead to great results. In this paper we take their idea and apply it using the novelty of a state as the guiding function. The concept of *novelty* has been first introduced in 2012 by Lipovetzky and Geffner [3]. The novelty is a measurement for how “new” the state looks when compared to already encountered states. The authors demonstrated that state-of-the-art performance can be achieved when combining this idea of novelty with known techniques such as landmarks and heuristics. Our goal is to evaluate whether this concept can be efficiently applied to bounded suboptimal search. The structure of this thesis is as follows: In Chapter 2 we

---

present the background knowledge required for the understanding of this work as well as basic definitions. Chapter 3 introduces the novelty aspect and how it was implemented. Chapter 4 provides a closer look on bounded suboptimal search and the ways it has been approached so far. Chapter 5 explains how we have combined the two ideas and provides details on the implementation of the search engine which has then been evaluated in Chapter 6. Chapter 7 finishes with a conclusion of the results and an outlook.

# 2

## Background

### 2.1 Planning

Planning can be described as a search in general state spaces. Planning tasks can be described using a suitable problem description language (planning formalism). A variety of formalisms are used nowadays, each with their own advantages and disadvantages. The two most common ones are STRIPS and SAS+ [4]. SAS<sup>+</sup> is an extension of STRIPS allowing multi-valued variables and also the language used in the *Fast Downward* planning framework [5].

It is defined as follows, with the notation borrowed from Sievers et al.[6]:

**Definition 1.** *An instance of the SAS<sup>+</sup> planning problem is given by a tuple*

$$\Pi = \langle V, O, S_0, S_*, cost \rangle$$

- $V = \{v_1, \dots, v_m\}$  is the set of state variables. Each variable  $v \in V$  has a finite domain  $D(v)$ . A partial state is a variable assignment on a subset of  $V$ , denoted by  $vars(s)$ . We write  $s[v]$  for the value assigned to  $v \in vars(s)$ , which must satisfy  $s[v] \in D(v)$ . We say that  $s$  complies with partial state  $s'$  if  $s[v] = s'[v]$  for all  $v \in vars(s) \cap vars(s')$ . A partial state  $s$  is a state if  $vars(s) = V$ . An atom is a pair of a variable and one of its values.
- $O$  is a set of operators with preconditions  $pre(o)$  and effects  $eff(o)$ , which are both partial variable assignments over  $V$ . An operator  $o \in O$  is applicable in a state  $s$  if  $s$  complies with  $pre(o)$ , in which case  $o$  can be applied, resulting in the successor state  $s'$  that complies with  $eff(o)$  and satisfies  $s'[v] = s[v]$  for all  $v \notin vars(eff(o))$ .
- $s_0 \in S_V^+$  denotes the initial state and is a complete assignment over  $V$ . The goal  $s_*$  is a partial assignment. A plan or solution is an operator sequence  $\pi = \langle o_1, \dots, o_n \rangle$  that is applicable to  $s_0$  resulting in a state that complies with  $s_*$ .
- The function  $cost : O \rightarrow \mathbb{R}_0^+$  assigns a non-negative cost to each operator. We call the plan  $\pi = \langle o_1, \dots, o_n \rangle$  with the smallest  $cost(\pi) = \sum_{i=1}^n cost(o_i)$  an optimal solution.

Given a planning task we try to find either:

- A plan, which is a sequences of actions leading from the initial state to a goal state
- Or a proof that no plan exists

Classical planning is a sub-category of planning with a static, deterministic and fully observable environment. Planning is usually subdivided in optimal and suboptimal planning. The difference is that in the first case we have to guarantee that the returned plans have minimal overall cost while in the second case we trade in that optimality expecting a performance gain in return.

## 2.2 Open lists

Open lists are used to keep track of candidate nodes for expansion. They manage the nodes generated by the parent node (leaves of a search tree). With their help we decide which node to expand next. The choice of data structure depends on the search strategy but the interface for the methods needed usually remains the same:

- `is_empty()` returns true if the list has no values
- `pop()` removes and returns the next node to expand
- `insert(n)` inserts node `n` into the open list

## 2.3 Heuristics

A heuristic function or heuristic for  $S$  is a function  $h : S \rightarrow \mathbb{R}_0^+ \cup \{\infty\}$  mapping each state to a non-negative number (or  $\infty$ ). The purpose of this function is to provide distance estimates from a state  $s$  to the closest goal state. The perfect heuristic  $h^*(s)$  always returns the cost of an optimal solution for  $s$  (or  $\infty$  if no solution exists). A Heuristic  $h$  can have the following properties:

- admissibility:  $h(s) \leq h^*(s) \quad \forall s \in S$
- consistency:  $h(s) \leq cost(a) + h(s')$  for all transitions  $s \xrightarrow{a} s'$
- goal-awareness:  $h(s) = 0 \quad \forall s_* \in S_*$
- safeness:  $h^*(s) = \infty$  for all  $s \in S$  with  $h(s) = \infty$



## 2.4 Best-First Search

A search node is implemented as a data structure holding information about:

- the corresponding state
- the parent node
- the path cost necessary to reach it
- the operator applied to reach it

Best-first search is a forward state space search, which means that it starts of at the initial state and tries to find a path to a goal state by expanding the most promising (search) node first. Expanding a node means generating all its children by applying the operators with met preconditions. An expanded node is noted as closed. Oftentimes a closed list is used to keep track of them. This us to check if a state already exists as a search node, also known as duplicate detection. It is also necessary for the concept of reopening a node, where a duplicate node is expanded if we find a cheaper path to its state. Reopening is an important feature as it guarantees that algorithms such as  $A^*$  can find optimal solutions with inconsistent (but admissible) heuristics.

# 3

## Novelty-Guided Search

Novelty-guided search, also known as width-based search, is an approach that relies on a width parameter bounding the complexity of classical planning problems. Lipovetzky and Geffner [7] have shown that the width of many domains is small and bounded provided that the goals are restricted to single atoms. Furthermore they have proven that classical planning tasks can be solved in time exponential in the width of the problem.

One example of an implementation for such an algorithm would be the Iterated Width search (IW) [7]. IW( $k$ ) is a normal breadth-first search that prunes all newly generated states with a *novelty*  $> k$  where the novelty of a state is defined as follows [3]:

**Definition 2.** *The novelty  $w(s)$  of a state  $s$  is  $i$  iff there is a tuple  $t$  of  $i$  atoms such that  $s$  is the first state in the search that makes all the atoms in  $t$  true, and no tuple of smaller size has this property.*

Since our search algorithm is only using the novelty for exploration we do not need to implement every step from IW. All that is needed is a novelty bound  $n$  passed to the framework as a parameter of the search engine. This bound  $n$  specifies the level precision of the novelty measures. For example with a level precision of 3 the state has either a novelty of 1,2,3 or larger than 3. For each state generated we then calculate its novelty with the specified precision and if the novelty exceeds the bound we return the value  $n + 1$ . Nodes with a state that has a heuristic value which is bigger than the bound will not be inserted in the open list.

To check whether the novelty of a state is  $d$  or below we have to check all partial states of size  $d$ . This means that for every state we consider we have to check the subsets of size  $d$ ;  $d \in [1..n]$ . This process can be accelerated drastically if we cache all the possible subsets and create a lookup table where we keep track of the encountered partial states as described in [8]. An additional list containing the offsets of these subsets is kept in memory for faster lookups. The number of possible subsets is described as:

$$l = \sum_{x=1}^n \binom{m}{x}$$

where  $m$  is the number of variables. Both the list of subsets and the list of offsets are of size  $l$ . For the sake of convenience we define the list of subsets  $subs$  as a list of sets ordered by inclusion. A simple way to compute this list is to generate all the subsets  $s_u$  with size  $1 \leq |s_u| \leq n$  of the powerset  $P(V)$  with  $V = \{v_0, v_1, \dots, v_m\}$ . The actual offset values and lookup table size depend on domain sizes  $|D_v|$  of the variables as well. With  $x \in [0, l]$ , the offset of the  $x$ -th subset  $s_{u_x}$  can be calculated as follows:

$$offset(s_{u_x}) = \begin{cases} 0, & \text{if } x = 0 \\ offset(s_{u_x - 1}) + \prod_{z=0}^{|s_{u_x}|} |D_{v_z}|, & \text{otherwise} \end{cases}$$

The partial state  $s^+$  for a given subset  $s_u$  is defined as:  $s^+ = \{v \mapsto s[v] \mid \forall v \in s_u\}$ . To get the offset of a partial state we need the position of  $s^+$  with respect to all possible assignments. This means that if  $s^+$  is the  $i$ -th partial state possible in  $s_u$  we get  $calculate\_offset(s^+) = offset(s_u) + i$ . The novelty  $w(s)$  of a given state can be computed using a sample algorithm as follows:

---

**Algorithm 1** computeNovelty
 

---

```

novelty = n + 1
for subset in subsets do
  partial_state = get_partial(subset, state)
  offset = calculate_offset(partial_state)
  if !encountered_partial_states[offset] then
    novelty = min(novelty, subset.size())
    encountered_partial_states[offset] = true
  end if
end for

```

---

Since it is important that we update all entries in the lookup table we have to finish the loop even if we find a cell that holds a *false* value. We can however exploit the fact that a check of the subsets of size  $d$  guarantees that the novelty is  $x$  or below. This means that if we cycle through the subsets from biggest to smallest we can abort the check if we find no *false* entry for all subsets of a particular size  $d$  and set  $w(s) = d + 1$ .

**Example 1.** Look up whether a given partial state has been encountered

- Variables:  $V = \{v_0, v_1, v_2, v_3\}$
- Variable domain sizes:  $|D_{v_0}| = 4, |D_{v_1}| = 5, |D_{v_2}| = 3, |D_{v_3}| = 1$
- Variable domains:  $D_v = \{0, \dots, |D_v| - 1\} \quad \forall v \in V$
- Subsets:  $subs = \langle \{v_0, v_1\}, \{v_0, v_2\}, \dots, \{v_2, v_3\} \rangle$
- Novelty bound:  $n = 2$
- State:  $s = \{v_0 \mapsto 2, v_1 \mapsto 4, v_2 \mapsto 1, v_3 \mapsto 0\}$
- Example subset:  $s_u = \{v_1, v_2\}$

*In this example the offsets, calculated as described above, would be:  $off = \{0, 20, 32, 36, 51, 56\}$ . The example subset  $s_u$  produces the partial state  $s^+ = \{v_1 \mapsto 4, v_2 \mapsto 1\}$ . We now look up the offset value of the subset  $\{1,2\}$  which is 36 and then we add  $s[v_1] \cdot |D_{v_2}| + s[v_2]$  to get the real offset 49 of our partial state. If cell 49 in `encountered_partial_states` holds false we know that the novelty is 2 or below.*

In our implementation of the novelty computation we add an extended version of the novelty  $w_h(s)$ , which takes an additional heuristic  $h$  into account. The computation of  $w_h(s)$  is exactly the same as above except that we initialize a lookup table for each distinct  $h(s)$  value. This modification helps avoiding thrashing of the lookup table which would cause the search to have a lot of low novelty values at the start and barely any states with  $w(s) \leq n$  in the later stages.

# 4

## Bounded Suboptimal Search

The main idea of a bounded suboptimal search is to sacrifice the optimality of the plan and get a solution more quickly in return. The solution cost is guaranteed to be within a certain range from the optimal cost, adjustable by a weighting parameter  $w$ . Solutions that give us this guarantee are named  $w$ -admissible solutions. A well known example of an algorithm returning such solutions is the weighted  $A^*$  algorithm [9]. As the name suggests it is a modified version of the optimal  $A^*$  with the difference that its evaluation function  $f'(n) = g(n) + w \cdot h(n)$  places additional emphasis on the estimated cost-to-go  $h(n)$ . Usually a single admissible heuristic is used for both objectives, the determination of the search order and the guarantee of the cost bound. It is however possible to use different heuristics for these two aspects. This approach is the essence of the *Explicit Estimation Search (EES)*, introduced by Thayer and Ruml [2]. In EES a potentially inadmissible heuristic is chosen for guiding the search while admissible information ensures the suboptimality bound.

In this paper we build upon the idea of EES to obtain a  $w$ -admissible solution but use a slightly modified implementation. The algorithm uses a potentially inadmissible distance-to-go estimator  $\hat{d}(n)$ , which tries to approximate the number of actions along a minimum-cost path from  $n$  to a goal. This idea is based on  $A_\epsilon^*$  [10], where a *focal* list sorted on  $\hat{d}(n)$  is kept up-to-date. The next nodes are then chosen from that list if they are within the specified suboptimality bound  $f(n) \leq w \cdot f(best_f)$ , where  $best_f$  is the node on the open list with the smallest  $f(n)$ . The tendency of  $f(n)$  rising along the path consequently leads to children of an expanded node not being inserted in *focal* as well as  $best_f$  having a low depth and high  $\hat{d}$ . This can cause a serious thrashing problem, where the *focal* list is repeatedly emptied until  $best_f$  is expanded. That expansion raises the bound condition and thus more children of the expanded node are included in *focal*.

EES combats this issue with introducing another list into the mix, which sorts the generated nodes on  $\hat{f}(n) = g(n) + \hat{h}(n)$ , where  $\hat{h}$  is a potentially inadmissible cost-to-go estimator. Because the algorithm relies on the unbiased  $\hat{h}$  to form its focal list the tendency for estimated costs to always rise is tempered and thus the thrashing behaviour observed in  $A_\epsilon^*$  is avoided.

For the node selection strategy we define  $best_{\hat{f}}$  and  $best_{\hat{d}}$  as follows:

$$best_{\hat{f}} = \arg \min_{n \in open} \hat{f}(n)$$

$$best_{\hat{d}} = \arg \min_{n \in open \wedge \hat{f}(n) \leq w \cdot \hat{f}(best_{\hat{f}})} \hat{d}(n)$$

We also have to keep track of the following lists:

- *open*: Contains all open nodes, sorted on  $\hat{f}(n)$
- *cleanup*: Contains all open nodes, sorted on  $f(n)$
- *focal*: Contains all open nodes with  $\hat{f}(n) \leq w \cdot \hat{f}(best_{\hat{f}})$  sorted on  $\hat{d}$

With these terms defined we select our next node using the following conditions:

---

**Algorithm 2** selectNode

---

```

if  $\hat{f}(best_{\hat{d}}) \leq w \cdot f(best_f)$  then
     $best_{\hat{d}}$ 
else if  $f(best_{\hat{f}}) \leq w \cdot f(best_f)$  then
     $best_{\hat{f}}$ 
else
     $best_f$ 
end if

```

---

First  $best_{\hat{d}}$  is considered provided that the focal list is not empty. If its expected solution cost is not within the bound we check if  $best_{\hat{f}}$  satisfies the condition. The expansion of  $best_{\hat{f}}$  can potentially relax the bound  $w \cdot \hat{f}(best_{\hat{f}})$  raising the number of possible candidates for the focal list. This can lead to a new  $best_{\hat{d}}$ , which may have a lower  $f$  value and consequently be chosen in the next iteration of the node selection. Worst case we choose  $best_f$  and therefor potentially raise the lower bound  $w \cdot f(best_f)$ . which might lead to choosing  $best_{\hat{d}}$  or  $best_{\hat{f}}$  in the next iteration. As proven by Thayer and Ruml [2], EES guarantees that  $f(n) \leq w \cdot g(opt)$  for all expanded nodes  $n$ , where  $g(opt)$  is the cost of an optimal solution. This conclusion means that EES provides  $w$ -admissible solutions. We must make sure that the constraint  $\hat{h}(n) \geq h(n)$  is validated which therefor limits our options in choice of heuristics.

# 5

## Combination

The central idea of this thesis was to combine the novelty measures with the bounded sub-optimal search, specifically the EES algorithm. This is done using the procedure presented in the previous chapter with the novelty approach in place of the distance-to-go estimator  $\hat{d}$ . We need a way too handle the nodes which are not in the bound restriction for the focal list, since it is still possible that  $best_{\hat{f}}$  rises along the path. One way to solve this issue is to introduce one more list that holds exactly these nodes. With this *backup* list in place we have the possibility to move nodes over to the focal list once we expand a new  $best_{\hat{f}}$  with a higher solution cost estimate. Furthermore because the novelty of a state does not change, the *focal* and *backup* list can just ignore all nodes with a novelty that exceeds our novelty bound  $n_b$ . Looking at those requirements a possible lineup for our lists would be:

- *open*: Contains all generated nodes, sorted on  $\hat{f}(n)$
- *cleanup*: Contains all generated nodes, sorted on  $f(n)$
- *focal*: Contains all generated nodes with  $\hat{d}(n) \leq n_b \wedge \hat{f}(n) \leq w \cdot \hat{f}(best_{\hat{f}})$  sorted on  $\hat{d}$
- *backup*: Contains all generated nodes with  $\hat{d}(n) \leq n_b \wedge \hat{f}(n) > w \cdot \hat{f}(best_{\hat{f}})$  sorted on  $\hat{f}(n)$

Most implementation details for the search engine are the same or similar to the eager search, which is already included in the Fast Downward framework. Each node is inserted in *open*, *cleanup* and *backup*. Major differences occur mainly in the function *fetch\_next\_node* which is, as the name already suggests, responsible for choosing the node we want to expand next.

**Algorithm 3** fetchNextNode

---

```

1: procedure FETCH_NEXT_NODE
2:   if open.is_empty() then return notsolvable
3:   end if
4:   bestf ← fetch_bestf()           ▷ From cleanup
5:   bestf̂ ← fetch_bestf̂()         ▷ From open
6:   while true do
7:     if backup.is_empty() then
8:       break
9:     end if
10:    node ← backup.remove_min()
11:    if  $\hat{f}(\textit{node}) \leq w \cdot \hat{f}(\textit{best}_{\hat{f}})$  then           ▷ Move nodes from backup to focal
12:      focal.insert(node)
13:    else
14:      backup.insert(node)           ▷ Reinsert node
15:      break
16:    end if
17:  end while
18:  bestâ ← fetch_bestâ()           ▷ From focal
19:  if  $\hat{f}(\textit{best}_{\hat{a}}) \leq w \cdot f(\textit{best}_f)$  then
20:    cleanup.insert(bestf)           ▷ Reinsert bestf
21:    open.insert(bestf̂)           ▷ Reinsert bestf̂
22:    return bestâ
23:  else
24:    focal.insert(bestâ)           ▷ Reinsert bestâ
25:  end if
26:  if  $\hat{f}(\textit{best}_{\hat{f}}) \leq w \cdot f(\textit{best}_f)$  then
27:    cleanup.insert(bestf)           ▷ Reinsert bestf
28:    return bestf̂
29:  else
30:    open.insert(bestf̂)           ▷ Reinsert bestf̂
31:    return bestf
32:  end if
33: end procedure

```

---

Notice that we have to explicitly fetch *best<sub>f</sub>*, because it is possible that the cleanup list contains closed nodes because it would be too costly to go through all 3 lists every time a node is removed from one. This means we have to iterate through the list discarding nodes until we hit one that is not closed. The same thing happens when we fetch *best<sub>â</sub>* and *best<sub>f̂</sub>*. Furthermore we need to pay attention to moving the nodes from *backup* to *focal*. This is a cheap operation since *backup* is ordered on  $\hat{f}$  and thus we can abort as soon as one node has a heuristic value over the bound. Open lists in Fast Downward do not provide a peek operation. However we can get the function values of the nodes on removal, which means we do not have to recompute the values (lines 11,19,26). Since they get cached computation on reinserting is not necessary either (lines 14,20,21,24,27,30). The same applies for the calculation of the novelty  $w_h(s)$  where we use  $\hat{h}$  as our heuristic for the extension of  $w(s)$ . Looking at the algorithms 2 things become apparent:



1. The lower the weight  $w$  the more often we disregard the *focal* and *backup* lists and choose *best<sub>f</sub>*. That means the search behaves more like  $A^*$ .  
A high  $w$  emphasizes the exploration aspect of the search.
2. The higher  $n_b$  the novelty bound the more we have to compute while gaining “only” 1 level of precision. Since determining that  $w(s)$  is  $i$  is exponential in  $i-1$  only 2 or 3 level precision is used in the best-first width search. As a consequence thereof setting the novelty bound too high can have a negative effect on the performance.

From these 2 points we can draw the conclusion that a high  $w$  in combination with a low  $n_b$  will probably not be a good idea.

# 6

## Evaluation

In this chapter we present the results as well as an evaluation of the conducted experiments. Goal of this work was to determine the usefulness of the of the integration of a novelty guided approach in bounded suboptimal search. First we compare different runs of our search algorithm presented in Chapter 5 with the objective of determining the best parameter choice. Thereafter we compare these findings to weighted  $A^*$ , using different weighting parameters in the process. Finally we proceed to evaluate the results of the two approaches introduced in Chapter 3 and 4 separately with the intention of observing their impact on already implemented search engines. The search options used in our experiments can be found in Appendix A.

### 6.1 Setup

The experiments were run using the *lab* Python module on a cluster of Intel<sup>®</sup> Xeon<sup>®</sup> processors running CentOS 6.5 at 2.2GHz at the center for scientific computing at the University of Basel (*sciCORE*). A memory limit of 2 GB and a time limit of 3 minutes has been set for all tasks. For the compilation of our planning system Fast Downward we used GCC 4.8.1 targeting a 32 bit release. The benchmark consists of 1,667 problems from 40 different domains.

### 6.2 Comparing Novelty Bounds

As mentioned in Chapter 5 the choice of a novelty bound heavily impacts the number of operations that have to be performed for the calculation of the novelty. A higher bound may lead to more nodes being inserted in our *focal* list and may improve the results. For this reason we ran the benchmarks with 3 different novelty bounds  $n_b = \{1, 2, 3\}$ , so we could find out the most suitable choice for our domain. Our own implementation based on the description in Chapter 5 will from now on be labeled as  $bsw\langle h, \hat{h}, nov_{n_b} \rangle$  where  $h$  is the admissible,  $\hat{h}$  the (potentially) inadmissible and  $nov_{n_b}$  our novelty approach with the bound  $n_b$  and tie-breaking on  $g(n)$ .

Table 6.1: Summary of the search attributes of the runs using *bsw* with different novelty values

Summary	bsw( $n_b = 1$ )	bsw( $n_b = 2$ )	bsw( $n_b = 3$ )	wA*
Plan length - Sum	26962	<b>26936</b>	26945	27348
Memory - Sum	<b>8337552</b>	8491108	20315652	10079064
Generated - Geometric mean	9383.61	9379.55	<b>9375.64</b>	12664.49
Expansions - Geometric mean	<b>1098.68</b>	1100.06	1099.75	1493.06
Coverage - Sum	947	947	882	<b>1027</b>
Search time - Geometric mean	0.80	0.85	1.44	<b>0.25</b>

In our experiments we used  $h = h^{LM}$ , the *landmark cut heuristic* [11] and  $\hat{h} = h^{FF}$ , the *FF-heuristic* [12], which is also used as the bucket heuristic in the determination of  $w_h(s)$ . This configuration meets the requirements for the  $w$ -admissibility of solutions, since  $h^{FF} \geq h^{LM}$ . We expected better results in terms of number of expansions the higher we set our  $n_b$ . We also forecasted a higher memory usage on the basis of the lookup tables growing at an exponential rate and having to keep track of more subsets. Based on these conditions we projected a lower expansion per search time rate. This consequently could lead to lower coverage due to time constraints.

Surprisingly changing  $n_b$  from one to two does not seem to have a very big effect on any attribute (see Figure 6.1). This indicates that in almost all problem-domains we meet one or both of these scenarios:

- We often have a node  $n$  with state  $s$  and  $w_h(s) = 1$  at the front of our *focal* queue,
- or often fall back to the open lists.

In both cases runs with different bounds fetch the same node for their next expansion. A different weight  $w$  could change that as more nodes from focal are allowed to be selected.

The second step from  $n_b = 2$  to  $n_b = 3$  has significantly higher impact on the search attributes and proves us right in our predictions. Coverage and expansion rate dropped considerably as we can see in Figure 6.2. The sum of memory used more than doubled in this step. These are all direct consequences of the overhead introduced by a higher precision. It should also be noted that some tasks could not be solved with  $n_b = 3$  because the number of variables and/or the size of the variable domains makes the size of the lookup tables too big. An example thereof would be problem 20 from the domain *logistics98*. It has 50 variables and each has a big domain size. In this particular instance we would have to store 8,339,805,951 booleans in our tables, which is inconceivable. On the other hand with a  $n_b = 2$  this number is reduced down to 7,150,018 booleans.

Because  $n_b = 2$  does not introduce much overhead when compared to  $n = 1$  and the additional precision may be useful when operating with different weights, we used it for the comparison against weighted  $A^*$  in the next step.

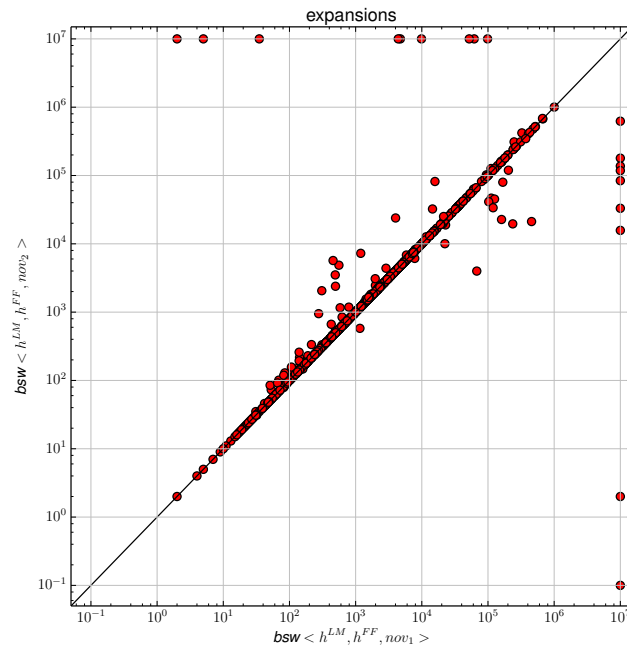


Figure 6.1: Comparison of expansions between  $bsw(n_b = 1)$  and  $bsw(n_b = 2)$  with  $w = 2$

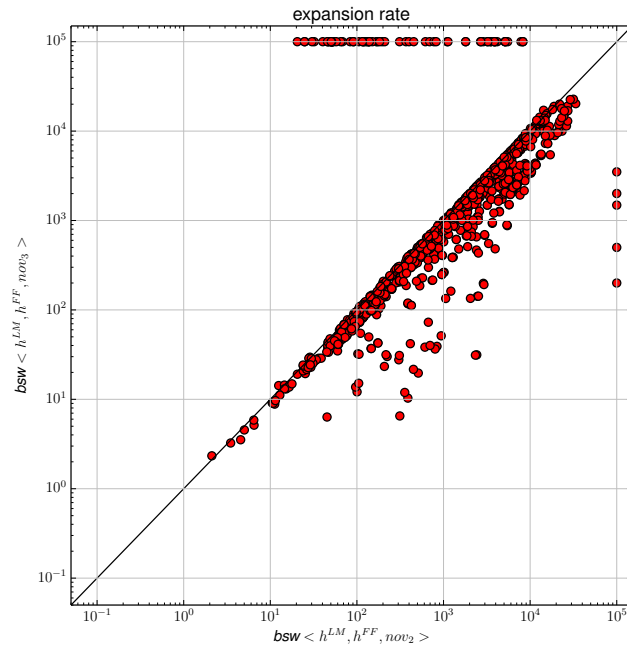


Figure 6.2: Comparison of the expansion rate between  $bsw(n_b = 2)$  and  $bsw(n_b = 3)$  with  $w = 2$

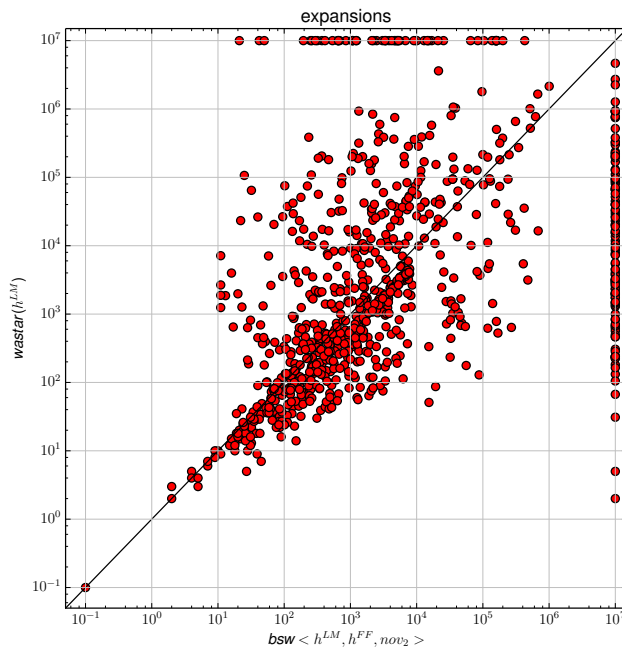


Figure 6.3: Comparison of expansions between  $\text{bsw}(n_b = 1)$  and weighted  $A^*$  with  $w = 2$

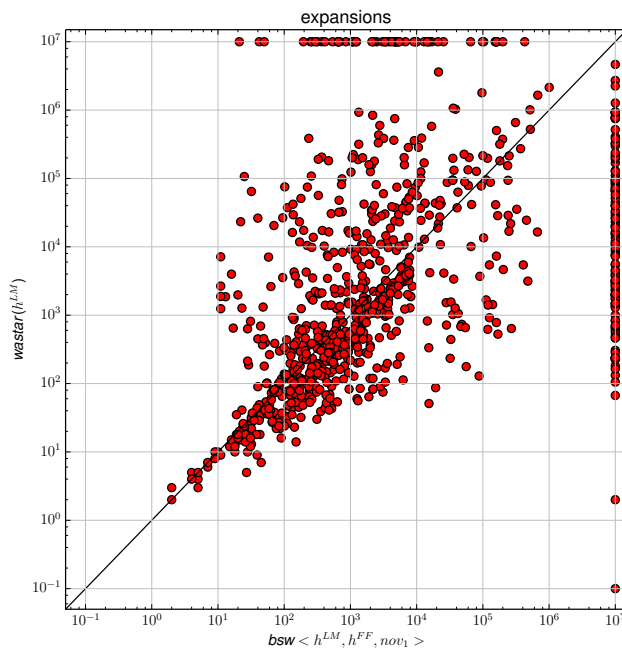


Figure 6.4: Comparison of expansions between  $\text{bsw}(n_n = 2)$  and weighted  $A^*$  with  $w = 2$

### 6.3 Comparing Weights

In this section we compare the impact of the weighting parameter  $w$  on our algorithms. Since we decided based on the results from above to set our bound to  $n_b = 2$  we are going to contrast  $bsw\langle h^{LM}, h^{FF}, nov_2 \rangle$  and weighted  $A^*$  ( $wA^*$ ) with  $h = h^{LM}$  with the weights  $w = \{2, 3\}$ . In Fast Downward the weight option, also used by  $wA^*$ , is implemented as an integer value. In  $bsw$  however it is also possible to specify a  $w$  as a decimal number, because it is only used in the bound checking and not in the evaluation functions themselves. As a consequence we additionally evaluated  $bsw$  with  $w = 1.5$ .

We confirm when looking at Figure 6.4 and Figure 6.3 that the number of expanded nodes appears to be lower in most domains which speaks for the guiding aspect of our search. In addition to that Table 6.1 confirms that the total memory used by  $bsw$  is lower than in  $wA^*$ .

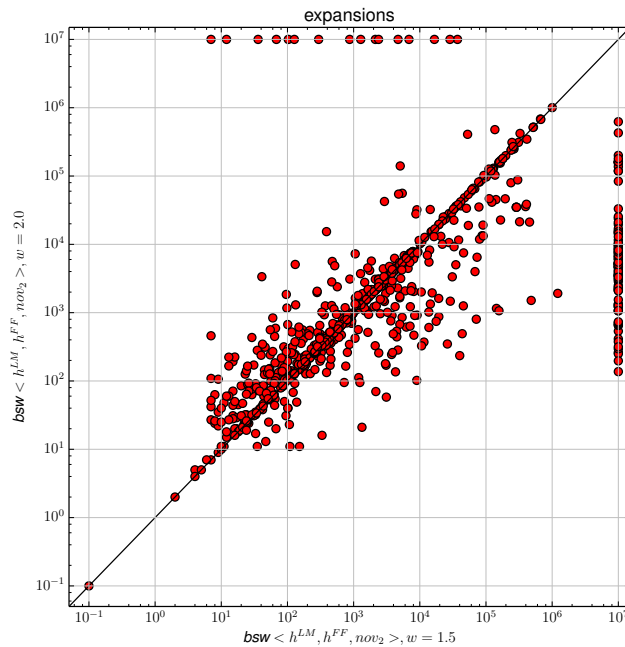


Figure 6.5: Comparison of expansions between  $bsw(n_b = 2)$  with  $w = 1.5$  and  $bsw(n_b = 2)$  with  $w = 3.0$

Coverage	$w = 1.5$	$w = 2.0$		$w = 3.0$	
	bsw( $n_b = 2$ )	bsw( $n_b = 2$ )	weighted $A^*$	bsw( $n_b = 2$ )	weighted $A^*$
airport (50)	20	21	<b>31</b>	20	<b>31</b>
barman-opt11-strips (20)	0	1	<b>4</b>	<b>4</b>	<b>4</b>
barman-opt14-strips (14)	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>
blocks (35)	26	30	32	32	<b>35</b>
childsnack-opt14-strips (20)	0	0	0	<b>2</b>	0
depot (22)	9	9	9	9	<b>12</b>
driverlog (20)	14	14	<b>15</b>	14	<b>15</b>
elevators-opt08-strips (30)	24	25	22	25	<b>26</b>
elevators-opt11-strips (20,20,20,19,20)	<b>19</b>	<b>19</b>	18	18	<b>19</b>
floortile-opt11-strips (20)	6	6	<b>13</b>	10	9
floortile-opt14-strips (20)	5	5	<b>16</b>	10	12
freecell (80)	15	24	13	<b>41</b>	20
ged-opt14-strips (20)	15	<b>19</b>	15	<b>19</b>	15
grid (5)	2	2	2	<b>3</b>	2
gripper (20)	7	<b>20</b>	<b>20</b>	<b>20</b>	<b>20</b>
hiking-opt14-strips (20)	9	12	9	<b>19</b>	11
logistics00 (28)	20	25	<b>28</b>	27	<b>28</b>
logistics98 (35)	7	7	<b>16</b>	9	15
miconic (150)	147	147	<b>150</b>	147	<b>150</b>
movie (30)	<b>30</b>	<b>30</b>	<b>30</b>	<b>30</b>	<b>30</b>
mprime (35)	<b>27</b>	24	21	22	20
mystery (30)	<b>18</b>	17	15	16	15
nomystery-opt11-strips (20)	15	<b>18</b>	<b>18</b>	<b>18</b>	14
openstacks-opt08-strips (30)	14	14	<b>16</b>	14	<b>16</b>
openstacks-opt11-strips (20)	9	9	<b>11</b>	9	<b>11</b>
openstacks-opt14-strips (20)	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>
openstacks-strips (30)	7	7	7	<b>15</b>	7
parcprinter-08-strips (30)	20	17	<b>30</b>	14	<b>30</b>
parcprinter-opt11-strips (20)	14	13	<b>20</b>	10	<b>20</b>
parking-opt11-strips (20)	6	<b>15</b>	9	6	9
parking-opt14-strips (20)	4	<b>16</b>	7	8	10
pathways-noneg (30)	5	6	8	6	<b>13</b>
pegsol-08-strips (30)	27	<b>30</b>	27	<b>30</b>	28
pegsol-opt11-strips (20)	17	<b>20</b>	17	<b>20</b>	18
pipesworld-notankage (50)	20	25	21	<b>26</b>	25
pipesworld-tankage (50)	11	13	10	13	<b>14</b>
psr-small (50)	48	48	<b>49</b>	48	<b>49</b>
rovers (40)	7	13	18	18	<b>19</b>
satellite (36)	11	11	13	11	<b>14</b>
scanalyzer-08-strips (30)	14	13	19	15	<b>20</b>
scanalyzer-opt11-strips (20)	11	10	15	11	<b>16</b>
sokoban-opt08-strips (30)	27	28	<b>29</b>	<b>29</b>	<b>29</b>
sokoban-opt11-strips (20)	19	<b>20</b>	<b>20</b>	<b>20</b>	<b>20</b>
storage (30)	15	15	17	<b>19</b>	18
tetris-opt14-strips (17)	3	3	5	5	<b>7</b>
tidybot-opt11-strips (20)	14	13	15	14	<b>16</b>
tidybot-opt14-strips (20)	6	6	10	8	<b>12</b>
tpp (30)	6	10	8	<b>12</b>	10
transport-opt08-strips (30)	<b>12</b>	11	11	11	11
transport-opt11-strips (20)	<b>8</b>	7	7	7	7
transport-opt14-strips (20)	<b>7</b>	<b>7</b>	6	6	<b>7</b>
trucks-strips (30)	10	15	<b>16</b>	<b>16</b>	15
visitall-opt11-strips (20)	12	12	17	15	<b>19</b>
visitall-opt14-strips (20)	8	8	10	9	<b>12</b>
woodworking-opt08-strips (30)	18	16	27	21	<b>30</b>
woodworking-opt11-strips (20)	13	10	19	15	<b>20</b>
zenotravel (20)	10	10	<b>15</b>	11	<b>15</b>
<b>Sum (1667)</b>	869	947	1027	1008	<b>1071</b>

As expected we achieve a higher coverage with a higher  $w$  for both algorithms. Figure 6.6 again confirms that when comparing number of expanded and generated nodes we can see a trend of them being lower in  $bsw$  for most domains. In domains like *elevators*, *freecell*, *parking* and *pegsol* there is a big improvement over  $wA^*$ . The only 3 domains where  $wA^*$  is substantially lower of  $bsw$  in terms of expansions are *floortile*, *sokoban* and *visitall*. The low correlation on *floortile* and *visitall* are likely due to  $h^{FF}$  not performing very well in these domains and  $bsw$  being misled when a node from the open list is expanded, while in the third domain *sokoban* novelty measures seem to be a poor choice of guiding method. These conclusions coincide with the results obtained in the experiments run by Lipovetzky and Geffner [13] where they set different algorithms using novelty measures in contrast to state-of-the-art algorithms like FF [12].

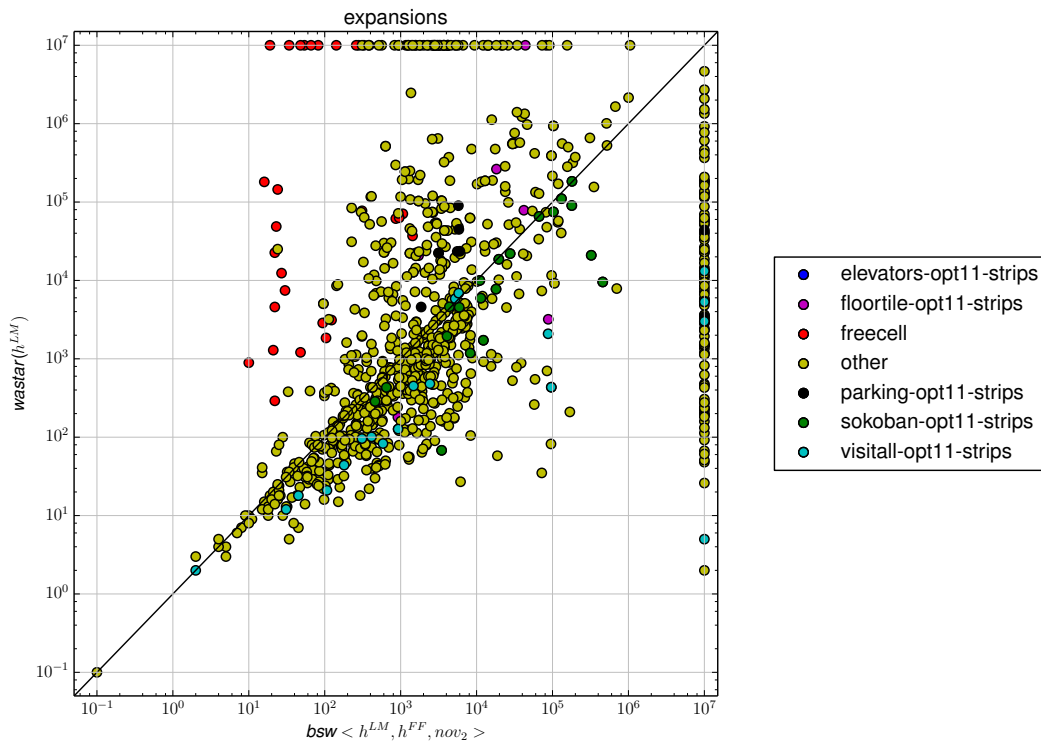


Figure 6.6: Comparison of expansions between  $bsw(n_b = 2)$  and weighted  $A^*$  with  $w = 3$

## 6.4 Alternatives

When using our implementations of EES and novelty measures separately we can observe an improvement over weighted  $A^*$  in almost all statistical attributes. Two algorithms are compared to weighted  $A^*$  with the *landmark and cut* heuristic. All three are using a weight of  $w = 2$ . The impact of the novelty guided search was tested using a weighted  $A^*$  algorithm that breaks ties using the novelty of states. EES was run with  $ees\langle h = h^{LM}, \hat{h} = h^{FF}, \hat{d} = h^{FF'} \rangle$ . The heuristic  $h^{FF'}$  is the *FF-heuristic* with the cost adapted to one which acts as a distance-to-go estimator. For the tie-breaking approach we



set the novelty bound to 2 and used  $h^{LM}$  as the bucket heuristic to avoid the need for computation of additional heuristic values. Both configurations reach a higher coverage than weighted  $A^*$ . The tie-breaking results are very close to weighted  $A^*$  in terms of coverage and mean search time while the geometric mean of expanded nodes is slightly lower as seen on Figure 6.7. A larger improvement was observed when analyzing the outcome of EES. The mean search time as well as the sum of memory used is lower. Figure 6.8 and Table 6.2 show that the number of generated and expanded nodes are much lower across almost all domains with the exception of the two domains *floortile* and *visitall*. The negative results in these domains can be explained with the same reasoning as above, since  $h^{FF}$  is being used here as well. Measured on memory consumption *ees* is by a factor 0.65 more efficient than the best of the other two algorithms. This can be explained with the lower number of generated nodes, which leads to less memory usage.

Table 6.2: Summary of the search attributes for the runs using the implementations of our search engine components separately

<b>Summary</b>	<b>wA*</b>	<b>tie&lt;wA*, nov<sub>2</sub>&gt;</b>	<b>ees</b>
Plan length - Sum	<b>35999</b>	37096	36504
Memory - Sum	14876532	15126440	<b>9523920</b>
Generated - Geometric mean	20926.90	19148.35	<b>2576.71</b>
Expansions - Geometric mean	2360.67	2164.33	<b>298.77</b>
Coverage - Sum	1027	1029	<b>1070</b>
Search time - Geometric mean	0.46	<b>0.37</b>	0.47

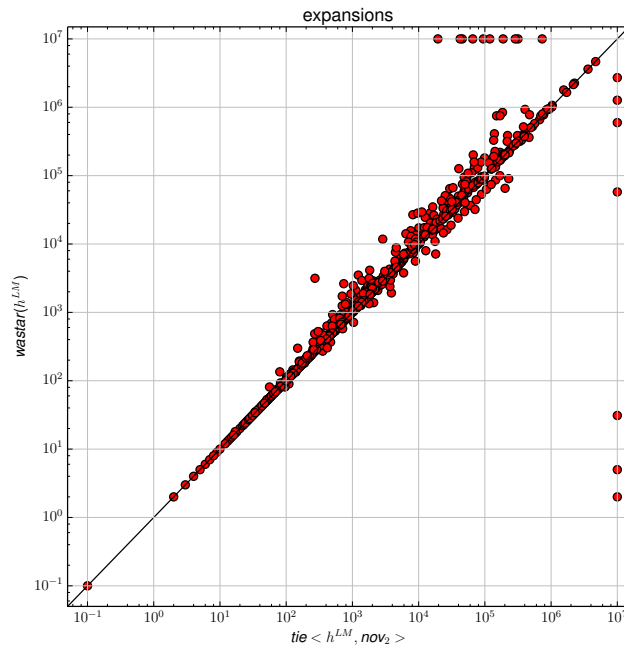


Figure 6.7: Comparison of expansions between weighted  $A^*$  and weighted  $A^*$  with novelty measures on tie break and  $w = 2$

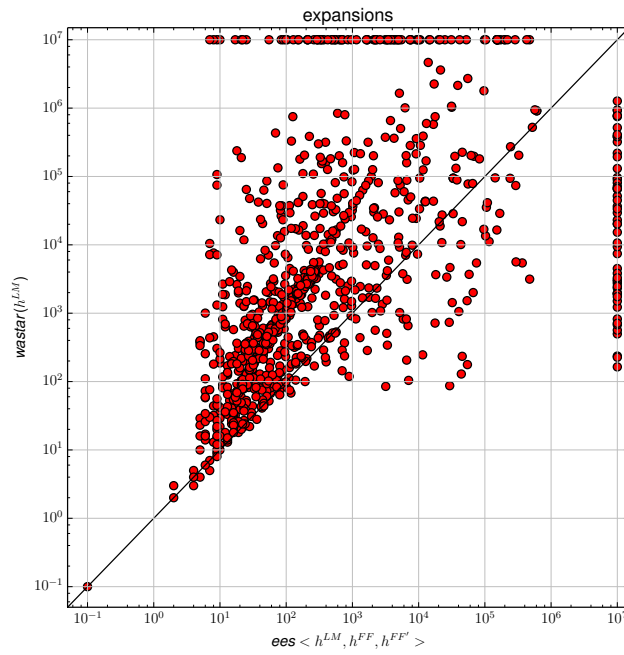


Figure 6.8: Comparison of expansions between ees and weighted  $A^*$  with  $w = 2$

# 7

## Conclusion and Future Work

We started off this thesis by introducing the novelty of a state. We then went on to describe the EES algorithm and explain how it is able to outperform previous bounded suboptimal search approaches. As a next step we looked at how novelty guided search can be used in place of the distance-to-go estimator in EES, combining the two ideas. We elaborated how the different novelty bounds affect the overall performance of the algorithm. The findings from this section were then used to compare our algorithm against weighted  $A^*$  with different weights. As a final step we looked at both components of the search engine separately.

An evaluation of the results illustrated a tendency of a lower number of expanded and generated nodes when comparing *bsw* to weighted  $A^*$  in most problem-domains. The mean search time on the other hand was considerably higher. This means we got a lower expansion rate as a result of our approach being computationally intensive which in turn lead to the coverage being lower. A higher time limit may have a positive effect for *bsw* on the comparison of coverage between the two algorithms. This speculation is based upon the fact that there are still problems open in domains where *bsw* showed promising results, for example in *freecell*, but would need further testing to be confirmed. When examining all algorithms used in our evaluation pure EES clearly returned us the best results. Although the overhead that EES introduced when compared to  $wA^*$  is noticeable we still get better results across almost all statistical attributes.

One possibility for future work would be to implement multiple heuristic functions for  $w_h$  instead of just one as described in [3]. As the amount of subsets we check in our calculation of novelty remains static during the search it would also be pretty easy to group them in chunks and parallelize this calculation. With a decreased overhead in calculations the algorithm may turn out to be more effective than weighted  $A^*$ .

## Bibliography

- [1] Hart, P. E., Nilsson, N. J., and Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107 (1968).
- [2] Thayer, J. T. and Ruml, W. Bounded suboptimal search: A direct approach using inadmissible estimates. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, IJCAI'11, pages 674–679 (2011).
- [3] Lipovetzky, N. and Geffner, H. Best-First Width Search: Exploration and Exploitation in Classical Planning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, AAAI, pages 3590–3596 (2017).
- [4] Bäckström, C. and Nebel, B. Complexity results for SAS+ Planning. *Computational Intelligence*, 11(4):625–655 (1995).
- [5] Helmert, M. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246 (2006).
- [6] Sievers, S., Wehrle, M., and Helmert, M. Generalized Label Reduction for Merge-and-Shrink Heuristics. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, AAAI, pages 2358–2366 (2014).
- [7] Lipovetzky, N. and Geffner, H. Width and Serialization of Classical Planning Problems. In *Proceedings of the 20th European Conference on Artificial Intelligence*, ECAI'12, pages 540–545 (2012).
- [8] Maggi, D. *Combining Novelty-Guided and Heuristic-Guided Search*. Master's thesis, University of Basel (2016).
- [9] Pohl, I. The Avoidance of (Relative) Catastrophe, Heuristic Competence, Genuine Dynamic Weighting and Computational Issues in Heuristic Problem Solving. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 12–17 (1973).
- [10] Pearl, J. and Kim, J. H. Studies in Semi-Admissible Heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(4):392–399 (1982).
- [11] Helmert, M. and Domshlak, C. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, ICAPS'09, pages 162–169 (2009).

- 
- [12] Hoffman, J. and Nebel, B. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302 (2001).
- [13] Lipovetzky, N. and Geffner, H. Width-based Algorithms for Classical Planning: New Results. In *Proceedings of the 21st European Conference on Artificial Intelligence*, ECAI'14, pages 1059–1060 (2014).

# A

## Appendix

Search options used with the Fast Downward planning system:

Weighted  $A^*$  with  $h = h^{LM}$  and  $w = 2$ :

- `-search "lazy_wastar(lmcut(),w=2)"`

*bsw* with  $h = h^{LM}, \hat{h} = h^{FF}$  and  $nov_2, w = 2.0$

- `-heuristic "hff=ff()" -search "bsw(eval=lmcut(),inad=hff,exp=novelty(n=2,bh=hff),w=2.0)"`

*ees* with  $h = h^{LM}, \hat{h} = h^{FF}, \hat{d} = h^{FF'}$  and  $w = 2.0$

- `-search "bsw(eval=lmcut(),inad=ff(),exp=ff(transform=adapt_costs(one)),w=2.0)"`

Weighted  $A^*$  with  $h = h^{LM}$ ,  $w = 2$ , ties broken with  $nov_2$

- `-heuristic "hlm=lmcut()"`  
• `-search "lazy(tiebreaking([sum([g(),weight(hlm,2)]),  
novelty(n=2,bh=hlm)],pref_only=false,unsafe_pruning=false),reopen_closed=true)"`

# Declaration on Scientific Integrity

## Erklärung zur wissenschaftlichen Redlichkeit

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

**Author — Autor**

Gian-Andrea Wetten

**Matriculation number — Matrikelnummer**

12-720-777

**Title of work — Titel der Arbeit**

Combining Novelty-Guided and Bounded Suboptimal Search

**Type of work — Typ der Arbeit**

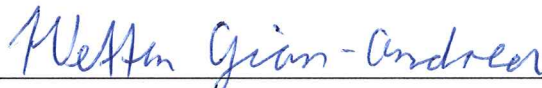
Bachelor Thesis

**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, 14.07.2017



Signature — Unterschrift