UNIVERSITÄT BASEL

Empirical Evaluation of Search Algorithms for Satisficing Planning

Master's Thesis

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

Examiner: Malte Helmert
Supervisor: Jendrik Seipp

Patrick von Reth
patrick.vonreth@unibas.ch
08-052-508

01/21/2015

Acknowledgements

I would like to thank Prof. Dr. Malte Helmert for the chance to write my master's thesis in the field of artificial intelligence with the support of his research team. I would also like to thank Jendrik Seipp for his support and guidance on this thesis. In addition, I would like to thank Fan Xie, Tatsuya Imai amd Akihiro Kishimoto for answering all my emails. I am grateful to Jendrik Seipp, Prof. Dr. Helmert, Gabriele Röger for reviewing my code. Further, I would like to thank Tobias Hartmann for his proofreading and his feedback. Finally, I want to thank all my friends and family who supported during this thesis.

Abstract

Essential for the estimation of the performance of an algorithm in satisficing planning is its ability to solve benchmark problems. Those results can not be compared directly as they originate from different implementations and different machines.
We implemented some of the most promising algorithms for greedy best-first search, published in the last years, and evaluated them on the same set of benchmarks. All algorithms are either based on randomised search, localised search or a combination of both.
Our evaluation proves the potential of those algorithms.

Table of Contents

1 Introduction

1.1 Motivation

Recently, many new promising search algorithms for satisficing planning were proposed. The most promising approaches either focus on the addition of a certain level of randomness use of a more focused *local* search.

Random exploration is used to compensate weaknesses or errors in a heuristic. Instead of expanding the most promising state, a random state from the open list is chosen for exploration with a certain probability. The algorithms differ in how and how often those random states are selected.

Local exploration can be used to find a path out of plateaus, i.e. regions in the heuristic where all states have the same heuristic value. A subset of states is used for a faster, more focused exploration.

While all approaches show great potential, it is unclear when and why they perform best.

All algorithms were tested on different benchmark sets, in different implementations and on different machines. To have a fair and easy comparison, we implemented them in the same framework and in a similar manner. For the implementation we focused on abstraction and reusability to be able to allow as many combinations of algorithms and configurations as possible. The algorithms are implemented in the Fast Downward planning framework (Helmert, 2006).

1.2 Outline

Chapter 2 introduces the relevant background information. In Chapter 3 we introduce the algorithms we implemented into Fast Downward as part of this thesis or are part of the experimental comparison. In Chapter 4 we describe the actual design and implementation details for the algorithms, that were newly added to Fast Downward. In Chapter 5 we reproduce the experiments from the papers the algorithms where presented in. In Chapter 6 we compare all algorithms directly on the IPC 2011[1] benchmark set.

Finally in Chapter 7 we discuss the results and give an outlook on possible research.

[1] International Planning Competition 2011 http://www.plg.inf.uc3m.es/ipc2011-deterministic/.

2 Background

This chapter introduces some basic knowledge and terms relevant for this thesis.

2.1 Planning tasks

We use SAS^+ planning tasks as introduced by Bäckström and Nebel (1995). A planning task is defined as a tuple $\prod = \langle V, O, s_0, s_*, c \rangle$ where V is a finite set of state variables, each state variable $v \in V$ having its own finite domain D_v.
A partial state s is a variable assignment of a value $s(v) \in D_v$ on variables $vars(s) \subseteq V$. A partial state defined on all variables is called a state. The partial state s is consistent with state s', if there are no variables $v \in V$ for which both states are defined with different values. The state s_0 is called **initial state** and the partial state s_* is called **goal**.
O is a finite set of **operators**. Each operator has a **precondition** pre_o and **effect** eff_o which are both partial states. c is the **cost** function $c : O \to \mathbb{R}_0^+$ which assigns each operator $o \in O$ its cost. If the **precondition** of an operator $o \in O$ is consistent with s, the operator o is **applicable**.

2.2 Heuristic search

A heuristic is a function which assigns a non negative number or ∞ to every state $s \in S$, where S is defined as the finite set of all states in $\prod$.

$$h : S \to \mathbb{R}_0^+ \cup \{\infty\}$$

It estimates the cost needed to reach a goal state from S. We call h^* the heuristic that returns the optimal plan cost for every state.
A heuristic is:

- admissible if $h(s) \leq h^*(s)$ for every state $s \in S$.
- consistent if $h(s) \leq h(s') + c(o)$ for all transitions $s \xrightarrow{o} s'$.
- goal aware if $h(s) = 0$ for every state s consistent with s_*.

2.2.1 Greedy best-first search

Best-first search is class of heuristic search algorithms, which use a function $f(s)$ to determine the most promising state to expand, the node with the minimum f.
Greedy best-first Search (GBFS) only considers the heuristic value of a state:

$$f(s) = h(s).$$

A prominent example for best-first search is A^*, which also includes the cost from the initial state s_0 to s, $g(s)$:

$$f(s) = g(s) + h(s).$$

2.3 Plateaus and misleading heuristics

Heuristics often make errors and thus lead to the exploration of states that do not lie on the path to the goal. Plateaus are regions in the search space, where the heuristic values $h(s)$ are all the same. For that reason many states with the same values have to be evaluated, which will not directly lead to an improvement and only add more states with the same heuristic value to the open list.
To escape a plateau there are multiple strategies, two of them are random exploration and local exploration.

2.3.1 Random exploration

In random exploration random states from the open list are explored to eventually find states with a more promising heuristic value. This is useful to escape plateaus created by a misleading heuristic.

2.3.2 Local exploration

A local search is started not from the initial state s_0 but from a particular state s_p from the search space. Because the local search does not know about states evaluated before, it is focused on the exploration of the children of the state s_p. A local search has therefore a higher probability to find a state with a lower heuristic value faster. Local exploration has a great potential for escaping plateaus.

2.4 Fast Downward

Fast Downward (Helmert, 2006) is a planning framework supporting variety of search algorithms and heuristics. Fast Downward is written in `C++` and uses modern features from `C++11` and Gnu Compiler Collection (GCC) to obtain a good performance while keeping a maintainable code base. Fast Downward is an open source project and can be found ot http://www.fast-downward.org/.

2.5 Search Enhancements

While there are many ways to enhance a search, we will only introduce the most relevant to this thesis.

2.5.1 Deferred evaluation

The default behaviour of GBFS is to evaluate all child states with the heuristic directly after the parent state is expanded. In Fast Downward this behaviour is found in the so called **eager** search engines. One search enhancement is **deferred evaluation** (Richter and Helmert, 2009). The children of the expanded state are added to the open list with the heuristic value of their parent. The computation of their actual heuristic value is delayed until the states are expanded. In Fast Downward search engines using this approach are called **lazy**.

2.5.2 Preferred operators

Preferred operators were first introduced by Hoffmann and Nebel (2001) as **helpful actions**. These are operators which are more likely a part of a solution path. A common application of preferred operators is the use of dual queue search with two open lists (Richter and Helmert, 2009). One open list contains all open states, and the second open list contains only the states reached with preferred operators.

3 Algorithms

This chapter introduces the algorithms relevant for this thesis.

3.1 ϵ-greedy best-first search

ϵ-greedy best-first search (**ϵ-GBFS**) (Valenzano et al., 2014) is an extension of GBFS using random exploration. With a probability given by the parameter ϵ, a state is selected uniformly randomly from the open list. With a probability of $(1 - \epsilon)$ the state with the lowest heuristic value is selected, i.e. the default behaviour of GBFS.

3.2 Enforced hill climbing

Enforced hill climbing (**EHC**) (Hoffmann and Nebel, 2001) starts a search on the state $s = s_0$. The search ends, when a state with a lower heuristic value is found, a goal state is reached or the whole search fails. If a new state s' with $h(s') < h(s)$ is found the operator o that reached s', $s \xrightarrow{o} s'$, is appended to the solution path. A new search is started on s', $s = s'$.

3.3 Type-based exploration

Type-based exploration (**type-based-GBFS**) (Xie et al., 2014b) is similar to ϵ-GBFS. This search algorithm is based on uniform random state selection. It maintains two open lists, a standard open list and a bucket open list. In the bucket open list states are grouped in buckets by multiple keys, the types. Types can be multiple heuristic values, the $g(s)$ even a constant value. When a state is removed from the bucket open list, first a uniform random bucket is selected. Afterwards out of this bucket a state is selected uniformly at random. States are selected alternately from the standard open list and from the bucket open list.

3.4 Monte-Carlo random walks

In Monte-Carlo random walks (Nakhost and Müller, 2009) the search explores states by applying operators at random for a path of a pre defined length (**random walk**). Only the state at the end of a path is evaluated. This is repeated multiple times (**random exploration**) and only the path yielding the lowest heuristic value is appended to the global path.
The global path consists of all selected random paths. New random walks are started from the end state of the global path. This is repeated until a goal is found or the search failed. Monte-Carlo random walks explore the state space fast, as the heuristic is only evaluated for the end point of a path.

There are multiple enhancements to this algorithm:

- Acceptable progress: The random exploration ends as soon as a heuristic value is found with an improvement above a certain threshold.
- Iterative deepening: Instead of a fixed length for the **random walk**, the length of a walk is automatically increased, if enough **random walks** did not yield an improvement.
- Monte-Carlo-Helpful-Actions (**MHA**): When a explorations was successful, $h(s)$ was improved, the probability to select one of the preferred operators, applicable to the current end state if the path, is increased.
- Monte-Carlo-Dead-End-Avoidance (**MDA**): The number of times an operator leads to a dead end is counted and the probability to select that operator is decreased.

3.5 Local exploration

Local exploration (Xie et al., 2014a) performs a normal GBFS (**global search**). If the heuristic value does not improve after a certain number of search steps a local search (**GBFS-LS**) is started on the next state s from the open list. The depth of the local search is limited. The local search shares the closed list with the global search.
The local search ends if either:

- the configured depth is reached.
- a state s' with $h(s') < h(s)$ is found.
- the local search fails, the local open list is empty.

All remaining nodes from the local open list will be merged into the open list of the global search.
A second configuration (**GBFS-LRW**) for local explorations is using local random walks. Monte-Carlo random walks (Nakhost and Müller, 2009) are started on the next state s from the open list. If a state s' with $h(s') < h(s)$ is found, the path of the random walk is added to the close list and s' is added to the global open list.

3.6 Diverse best-first search

In diverse best-first search (**DBFS**) (Imai and Kishimoto, 2011) two open lists are used. The first is the global open list, which selects states probabilistic to their $h(s)$ and $g(s)$ values (see Imai and Kishimoto, 2011, Algorithm 2). States with a small $g(s)$ are preferred. The second list is a local open list, which can be of any open list implemented in Fast Downward. A state s is selected from the global open list and inserted into the local open list. Afterwards a search is performed on the local open list with the depth $d := h(s)$. If the depth is reached, the number of removed nodes equals d, or the local search fails, the remaining states are merged into the global open list. The local open list is cleared and a new local search is started, using the next state from the global open list.

DBFS uses both random exploration as well as local exploration.

4 Implementation

In this chapter we describe our implementations and the decisions leading to the actual design.

4.1 ϵ-Greedy best-first search

We implemented two versions of ϵ-GBFS (Valenzano et al., 2014). The first implementation, `RandomBucketOpenList`, is similar to the original implementation used by Valenzano et al. (2014) (personal communication with the authors). The second implementation, `RandomOpenList`, is heap-based and specialised for the requirements of ϵ-GBFS and should thus perform better. In Table 4.1 you can find a comparison of the complexities of both implementations.

4.1.1 RandomBucketOpenList

In `StandardScalarOpenList` the states are stored in buckets, based on their heuristic value and the buckets are stored in a `std::map` mapping the heuristic value to a bucket. The states are retrieved using FIFO tie breaking. This implementation is an extension of `StandardScalarOpenList`. We reimplemented the method `StandardScalarOpenList::remove_min`. While this method returns the state with the minimal heuristic value from the open list, `RandomBucketOpenList::remove_min` returns, with the probability of ϵ, a random state. With a probability of $(1 - \epsilon)$ the default implementation `StandardScalarOpenList::remove_min` is used. As the states are stored in buckets, we have to iterate over the buckets until we find the bucket containing the n^{th} random state, which has linear with respect to the number of buckets.

4.1.2 RandomOpenList

Instead of a `std::map` containing buckets as used in `RandomBucketOpenList`, we store the states in a `std::heap`. This enables us to remove the states with the lowest heuristic value in a fast manner, using `std::pop_heap` in case of the default behaviour of GBFS which happens with the probability $(1-\epsilon)$. To remove a random state we generate a random index for the state to be removed. We decrease the key to $-\infty$ of the state and readjust the heap. Finally, we can just use `std::pop_heap` to remove the selected state. In order to enable FIFO tie breaking, as it is done with buckets in `StandardScalarOpenList`, we not only use the heuristic value as key in our heap, but also assign a unique id to each state. The id is increased with every added state.

Action	**RandomBucketOpenList**	**RandomOpenList**
Insert state	$O(1)$	$O(log(n))$
Remove random state	$O(m)$	$O(log(n))$
Remove min state	$O(1)$	$O(log(n))$

Table 4.1: Complexity of ϵ-GBFS. n is the number of states and m the number of buckets.

In Table 4.1 we can see that we have a slightly higher complexity for the insertion of states and the removal of the minimum state $O(log(n)) > O(1)$ but its compensated by the reduced complexity for the removal of a random stae. For `RandomBucketOpenList` the complexity is linear to the number of buckets, which depending on the problem and the used heuristic can be enormous.

4.2 Type-based exploration

As Fast Downward already supports alternation between multiple open lists, we only had to implement the type-based open list. In the type-based open list states are stored in key-bucket pairs in a `std::vector` (**list**). In addition we maintain a `std::unordered_map` (**lookup table**) pointing to the index in the list. We use a `std::vector<int>` as key. As a result the combination of keys can be of any length.
The lookup table is only used to retrieve the index for the insertion of the state. In order to remove a state, we generate a uniform random index for the bucket. We access the list directly with this index without the use of the lookup table. Finally we generate a uniform random index for the state inside the bucket. We decided to use this two level approach as accessing the n^{th} random bucket in a map has linear complexity $O(n)$ whereas the complexity for accessing the n^{th} random bucket in a vector is constant $O(1)$.

4.3 Monte-Carlo random walks

Our implementation of Monte-Carlo random walks (Nakhost and Müller, 2009) includes all exploration algorithms from ARVAND. The support for multiple configurations is not implemented and thus we cannot switch between the different exploration algorithms (Pure-Random, Monte-Carlo-Helpful-Actions (**MHA**) and Monte-Carlo-Dead-End-Avoidance (**MDA**)). However this feature is not needed for this thesis.
While Fast Downward already provides a closed list, we had to implement something new for the random walks, to be able to prevent loops in the global path. We use a set of states as a close list, and a simple vector of state-operator pairs to save the path. If a path, resulting of a random walk, is selected to be part of the global path, we merge it with the global path and remove possible loops. After the search reaches the goal state, this path is converted to

the default representation used in Fast Downward.

4.4 Local exploration

Our implementation of local exploration (Xie et al., 2014a) is designed to be an abstract and non intrusive search engine. The actual search steps are delegated to existing search engines.

Our design enables us to use different search engines for the local search and for the global search as well as different heuristics or open lists.

To be able to run a new search, starting on any state instead of the initial state s_0, we extended the options parameter for the supported search types by the option **root**. If the option **root** is available, the search will start on that state instead of s_0. It is now also possible to pass the search space, which in Fast Downward represents the closed list, to a search engine using the options. We added a method `new_instance(const Options &options)`, to the supported search engines, to be able to create new instances of search engines of the same type for a local search.

Open lists in Fast Downward are template-based and the types for lazy (see Subsection 2.5.1) and eager search are different. For that reason it is not possible to combine a lazy with an eager search.

4.5 Diverse best-first search

We decided to implement DBFS with two levels of open lists. The `DiverseOpenList` receives two open lists as arguments. The first is the global open list, which is used to store the states between the local search steps. The second is the local open list. DBFS is implemented as open lists thus it support different search engines. The global open list is supposed to be a `ProbabilisticOpenList`, while the local open list can be of any supported type. Our `ProbabilisticOpenList` handles probabilistic selection of states based on Imai and Kishimoto (2011) (Algorithm 2). We modified the algorithm to only iterate over existing $g(s)$ and $h(s)$ values, see Algorithm 4.1.

The design of our `DiverseOpenList` implementation is basically a wrapper for the local open list. Most calls are directly forwarded to the local open list.

`DiverseOpenList::remove_min` counts the number of local steps. If the number of steps exceeds the limit or the local open list is empty, the local open list is reset. The limit is defined as the heuristic value of the last root state for the local open list. All remaining states from the local open lists are evaluated again, inserted into the global open list and the local open list is cleared. Finally, the next state from the global open list is fetched as new root state s, inserted into the local open list and the limit for the local steps is set to $h(s)$.

```
OL = the open list , mapping (h,g) keys to states
p_total = 0
h_map = sorted map of h values with reference count
g_map = sorted map of g values with reference count

if with probability of P:
    G = random(0, |g_map|)
else:
    G = |g_map|
end if

for h in h_map:
    for g in g_map[0:G]:
        if (h,g) in OL:
            p[h][g] = pow(T, h - h_min)
            p_total += p[h][g]
        end if
    end for
end for

select a pair of h and g with probability of p[h][g]/p_total
h_map[h]--
g_map[g]--

if the reference count is 0 the h or g value is removed from the maps
dequeue a node n with h(n)=h and g(n)=g in OL
return n
```

Algorithm 4.1: Modified version of Imai and Kishimoto (2011) Algorithm 2.

5 Experiments

5.1 Setup

For all search algorithms we have run experiments with the same configuration as in the original paper and on the same set of benchmarks. Results named **base** are always referring to a standard GBFS run on the benchmark set. The heuristic used depends on the experiments we reproduced.

The experiments were run on a cluster of Intel Xeon E5-2660 processors (2.2 GHz) running CentOS 6.5. Each task was executed on a single processor core, with a time limit of 30 minutes and a memory limit of 2GB.
The full configurations can be found in Appendix A.

5.2 Evaluation scores

The coverage charts show how many problems of a domain were solved.
Coverage (Sum): The sum over all domains.
Average score: The number of solved problems in relation to domains in percentage.
Coverage score: The average percentage of problems solved per single domain.
Domains better than base: The number of domains, where the coverage improved over the base line algorithm.
Domains worse than base: The number of domains, where the coverage could not reach the results of the base line algorithm.

In the bar charts we use the following colours:

- blue: The results of the original implementation.
- green: Our own results.
- red: The results of our second implementation.

5.3 ϵ-greedy best-first search

We have run the experiments on the bucket implementation (Subsection 4.1.1). In addition to the ϵ parameters proposed, we also evaluated the results for $\epsilon = 0$ and $\epsilon = 1$ to show the performance without any random evaluation and pure random evaluation. We have run the same benchmarks on our more specialised heap implementation from Subsection 4.1.2. A comparison of the original results with both `RandomBucketOpenList` and `RandomOpenList` can be found in Figure 5.1.
The benchmark set consists of 790 problems in 30 domains.

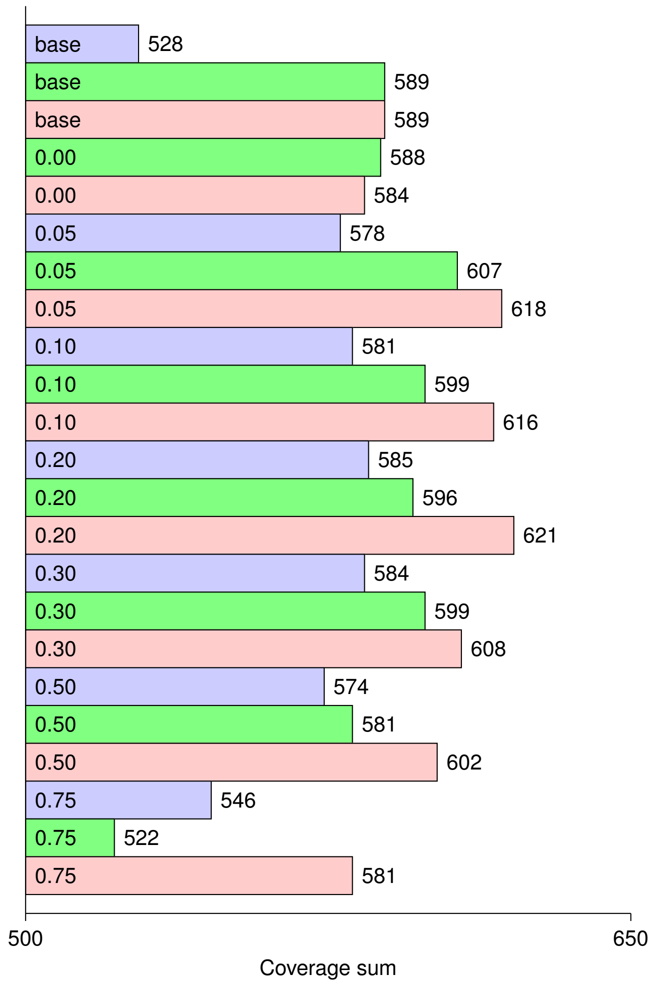

Figure 5.1: Comparison of ϵ-GBFS coverage with Valenzano et al. (2014) Table 1 for different values of ϵ. Blue are the results from Valenzano et al. (2014) Table 1, green are the results for `RandomBucketOpenList` and red are the results for `RandomOpenList`. The results for $\epsilon = 1$ are not included as they fall out of the comparison.

While we surpass most results from Valenzano et al. (2014) Table 1 even for the baseline configuration, our specialised implementation of ϵ-GBFS shows much better results than both, the original and our bucket approach. The result for $\epsilon = 0$, i.e. no random states, shows that our heap implementation can compete with the `StandardScalarOpenList`. For `RandomBucketOpenList` $\epsilon = 0.05$ achieves the best results, while in `RandomOpenList`

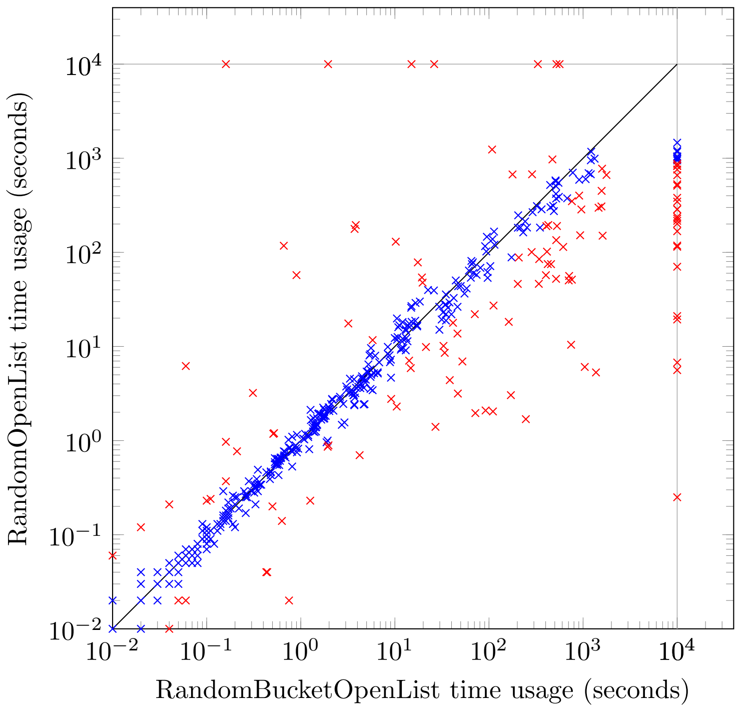

Figure 5.2: Comparison of time usage of `RandomOpenList` and `RandomBucketOpenList` for $\epsilon = 0.20$.

$\epsilon = 0.20$ achieves the best results.
This is due to the complexity overhead in `RandomBucketOpenList`. Using more random values leads to more overhead and impacts the run time.
In Figure 5.2 we compare the run-time of `RandomOpenList` with the run-time of `RandomBucketOpenList`. As expected the gain in coverage is directly related to the improved run-time.

The heap based implementation `RandomOpenList` consumes more memory than `RandomBucketOpenList`. The few cases were `RandomBucketOpenList` solved a problem and `RandomOpenList` did not, are those, where `RandomOpenList` exceeded the memory limit.

In Table 5.1 and Table 5.2 you can see that the improvement is evenly distributed over all domains.

Coverage	**base**	**0.00**	**0.05**	**0.10**	**0.20**	**0.30**	**0.50**	**0.75**	**1.00**
barman-sat11-strips (20)	**19**	**19**	**19**	16	14	15	11	9	0
elevators-sat08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	0
elevators-sat11-strips (20)	18	18	**20**	18	**20**	19	16	14	0
floortile-sat11-strips (20)	**6**	**6**	4	4	4	3	3	2	0
nomystery-sat11-strips (20)	9	9	8	9	**11**	9	9	8	3
openstacks (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	7
openstacks-sat08-adl (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	6
openstacks-sat08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	6
openstacks-sat11-strips (20)	**20**	**20**	**20**	**20**	**20**	**20**	19	19	0
parcprinter-08-strips (30)	26	26	26	27	26	**28**	27	26	12
parcprinter-sat11-strips (20)	12	12	12	**15**	**15**	14	14	13	0
parking-sat11-strips (20)	**18**	17	17	17	15	**18**	17	11	0
pathways (30)	11	11	**14**	12	13	10	9	9	4
pegsol-08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	25
pegsol-sat11-strips (20)	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	15
pipesworld-tankage (50)	23	23	27	25	26	28	**29**	28	9
rovers (40)	23	23	**26**	24	23	24	23	21	6
scanalyzer-08-strips (30)	27	27	**30**	**30**	**30**	**30**	29	27	7
scanalyzer-sat11-strips (20)	17	17	**20**	**20**	**20**	**20**	**20**	17	1
sokoban-sat08-strips (30)	**29**	**29**	**29**	**29**	**29**	**29**	**29**	28	22
sokoban-sat11-strips (20)	**19**	**19**	**19**	**19**	**19**	**19**	**19**	18	12
storage (30)	18	18	19	21	20	**22**	21	19	13
tidybot-sat11-strips (20)	14	14	16	15	15	**17**	16	16	3
tpp (30)	**23**	**23**	21	20	19	17	17	15	5
transport-sat08-strips (30)	16	16	17	16	**18**	16	16	16	6
transport-sat11-strips (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
trucks (30)	**17**	**17**	**17**	16	15	15	13	13	6
visitall-sat11-strips (20)	5	5	**9**	7	6	7	5	4	0
woodworking-sat08-strips (30)	**30**	**30**	29	**30**	29	**30**	**30**	**30**	5
woodworking-sat11-strips (20)	**19**	**19**	18	**19**	**19**	**19**	**19**	**19**	1
Sum (790)	589	588	**607**	599	596	599	581	552	174
Average Score in %	74.56	74.43	**76.84**	75.82	75.44	75.82	73.54	69.87	22.03
Coverage Score in %	74.67	74.51	**76.74**	75.83	75.43	75.76	73.07	68.95	20.71
Domains better than base	0	0	10	10	**11**	10	7	4	0
Domains worse than base	0	1	6	5	6	5	8	13	**29**

Table 5.1: ϵ-GBFS: Reproduction of Valenzano et al. (2014) Table 1. base is a standard GBFS, the other results are from RandomBucketOpenList implementation for different values of ϵ.

Coverage	**base**	**0.00**	**0.05**	**0.10**	**0.20**	**0.30**	**0.50**	**0.75**	**1.00**
barman-sat11-strips (20)	**19**	**19**	**19**	**19**	**19**	18	17	16	0
elevators-sat08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	1
elevators-sat11-strips (20)	18	18	19	**20**	**20**	18	19	17	0
floortile-sat11-strips (20)	6	6	6	6	**7**	6	6	6	2
nomystery-sat11-strips (20)	9	9	**10**	**10**	**10**	**10**	9	**10**	4
openstacks (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	7
openstacks-sat08-adl (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	6
openstacks-sat08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	6
openstacks-sat11-strips (20)	**20**	**20**	**20**	**20**	**20**	**20**	**20**	19	0
parcprinter-08-strips (30)	26	26	26	**28**	**28**	27	27	27	14
parcprinter-sat11-strips (20)	12	12	14	15	**16**	15	14	15	0
parking-sat11-strips (20)	18	18	**20**	**20**	19	18	18	15	0
pathways (30)	11	11	**12**	**12**	**12**	10	10	5	4
pegsol-08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	27
pegsol-sat11-strips (20)	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	17
pipesworld-tankage (50)	23	23	27	27	26	27	26	**30**	9
rovers (40)	23	22	**24**	23	**24**	23	22	21	6
scanalyzer-08-strips (30)	27	27	**30**	**30**	**30**	**30**	**30**	28	8
scanalyzer-sat11-strips (20)	17	17	**20**	**20**	**20**	**20**	**20**	18	1
sokoban-sat08-strips (30)	**29**	**29**	**29**	**29**	**29**	**29**	**29**	**29**	25
sokoban-sat11-strips (20)	**19**	**19**	**19**	**19**	**19**	**19**	**19**	**19**	15
storage (30)	18	17	22	21	21	22	**23**	20	14
tidybot-sat11-strips (20)	14	14	16	16	16	16	**17**	16	2
tpp (30)	**23**	21	22	22	22	19	18	16	6
transport-sat08-strips (30)	16	16	19	17	**20**	18	18	16	6
transport-sat11-strips (20)	0	0	0	0	**1**	0	0	0	0
trucks (30)	17	17	**18**	16	16	17	15	17	9
visitall-sat11-strips (20)	5	4	7	7	7	**8**	7	5	0
woodworking-sat08-strips (30)	**30**	**30**	**30**	**30**	**30**	29	29	29	6
woodworking-sat11-strips (20)	**19**	**19**	**19**	**19**	**19**	**19**	**19**	17	1
Sum (790)	589	584	618	616	**621**	608	602	581	196
Average Score in %	74.56	73.92	78.23	77.97	**78.61**	76.96	76.20	73.54	24.81
Coverage Score in %	74.67	74.09	78.41	78.33	**79.01**	77.22	76.51	73.36	23.54
Domains better than base	0	0	14	13	**16**	10	10	8	0
Domains worse than base	0	4	1	2	2	4	6	9	**29**

Table 5.2: ϵ-GBFS: Reproduction of Valenzano et al. (2014) Table 1. base is a standard GBFS, the other results are from RandomOpenList implementation for different values of ϵ.

5.4 Type-based exploration

We conducted the same experiment as Xie et al. (2014b) Table 1. A comparison of the results can be found in Figure 5.3.
The benchmark set consists of 2112 problems in 54 domains.

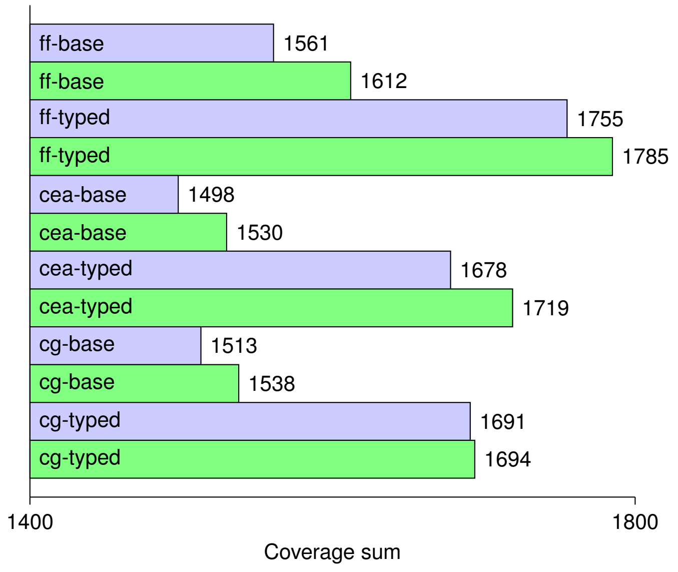

Figure 5.3: Comparison Type-based Exploration coverage with Xie et al. (2014b) Table 1 for the heuristics ff, cea, cg and with the type-based-GBFS(heuristic,$g(s)$).

As you can see in Figure 5.3 the results scale similarly to the original implementation.

Coverage	**cea-base**	**cea-typed**	**cg-base**	**cg-typed**	**ff-base**	**ff-typed**
airport (50)	**45**	41	28	29	36	36
assembly (30)	11	23	12	15	**30**	**30**
barman-sat11-strips (20)	0	0	0	1	**19**	17
blocks (35)	**35**	**35**	**35**	**35**	**35**	**35**
cybersec-sat08-strips (30)	12	**30**	0	17	26	**30**
depot (22)	12	14	12	17	15	**18**
driverlog (20)	18	18	18	**20**	18	19
elevators-sat08-strips (30)	28	29	28	29	**30**	**30**
elevators-sat11-strips (20)	9	10	9	10	**19**	16
floortile-sat11-strips (20)	6	7	0	2	6	**8**
freecell (80)	79	**80**	75	**80**	79	**80**
grid (5)	4	**5**	4	**5**	4	**5**
gripper (20)	**20**	**20**	**20**	**20**	**20**	**20**
logistics00 (28)	**28**	**28**	**28**	**28**	**28**	**28**
logistics98 (35)	**34**	**34**	**34**	**34**	30	29
miconic (150)	**150**	**150**	**150**	**150**	**150**	**150**
miconic-fulladl (150)	**139**	138	137	**139**	136	**139**
miconic-simpleadl (150)	**150**	**150**	**150**	**150**	**150**	**150**
movie (30)	**30**	**30**	**30**	**30**	**30**	**30**
mprime (35)	34	34	**35**	**35**	31	33
mystery (30)	17	**19**	17	18	17	**19**
nomystery-sat11-strips (20)	7	12	7	15	10	**19**
openstacks (30)	24	17	26	24	**30**	**30**
openstacks-sat08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**
openstacks-sat11-strips (20)	19	17	13	12	**20**	19
optical-telegraphs (48)	5	5	1	2	4	**6**
parcprinter-08-strips (30)	**30**	29	**30**	**30**	**30**	**30**
parcprinter-sat11-strips (20)	**20**	19	**20**	**20**	**20**	**20**
parking-sat11-strips (20)	14	10	**20**	17	**20**	17
pathways (30)	10	20	11	11	10	**21**
pegsol-08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**
pegsol-sat11-strips (20)	**20**	**20**	**20**	**20**	**20**	**20**
philosophers (48)	**48**	**48**	**48**	**48**	**48**	**48**
pipesworld-notankage (50)	24	41	28	41	33	**42**
pipesworld-tankage (50)	21	**29**	16	23	21	28
psr-large (50)	**31**	**31**	**31**	28	15	22
psr-small (50)	**50**	**50**	**50**	**50**	**50**	**50**
rovers (40)	26	**32**	27	**32**	23	29
satellite (36)	31	30	**36**	**36**	27	27
scanalyzer-08-strips (30)	29	29	**30**	**30**	27	**30**
scanalyzer-sat11-strips (20)	19	19	**20**	**20**	17	**20**
schedule (150)	20	87	20	87	37	**111**
sokoban-sat08-strips (30)	5	27	28	**29**	**29**	**29**
sokoban-sat11-strips (20)	3	17	18	**19**	**19**	**19**
storage (30)	16	24	18	**28**	18	23
tidybot-sat11-strips (20)	**18**	**18**	11	15	15	16
tpp (30)	**27**	24	26	24	23	21
transport-sat08-strips (30)	28	26	**30**	29	17	16
transport-sat11-strips (20)	9	7	**17**	14	1	0
trucks-strips (30)	15	20	14	14	17	**23**
visitall-sat11-strips (20)	4	7	4	**8**	3	**8**
woodworking-sat08-strips (30)	14	21	14	20	16	**27**
woodworking-sat11-strips (20)	2	8	2	4	3	**12**
zenotravel (20)	**20**	**20**	**20**	**20**	**20**	**20**
Sum (2112)	1530	1719	1538	1694	1612	**1785**
Average Score in %	72.44	81.39	72.82	80.21	76.33	**84.52**
Coverage Score in %	69.70	78.47	71.25	77.71	76.00	**83.12**
Domains better than base	0	22	0	26	0	25
Domains worse than base	0	11	0	7	0	8

Table 5.3: Type-Based Exploration: We compare the coverage of a standard GBFS with the heuristics ff, cea, cg and type-based-GBFS (heuristic,$g(s)$).

5.4.1 Multiple heuristics in type-based-GBFS

We also reproduced the experiment from Xie et al. (2014b) Table 4 see Figure 5.4. This experiment is important, as we are able to use any number of heuristics with our type based open list implementation while the original Fast Downward implementation from Xie et al. (2014b) only allowed a $h(s)$ and a $g(s)$ value. For that reason only a direct comparison to the results of the experiments with the keys $[const(1)], [g()], [h^{ff}()], [h^{ff}(), g()]$ is possible as the other results originate from the LAMA implementation of type-based exploration. The results for $[h^{ff}(), h^{cea}(), g()], [h^{ff}(), h^{cg}(), h^{cea}(), g()], [h^{ff}(), h^{cg}(), g()]$ are additions on our side to test how our implementation performs with longer keys.

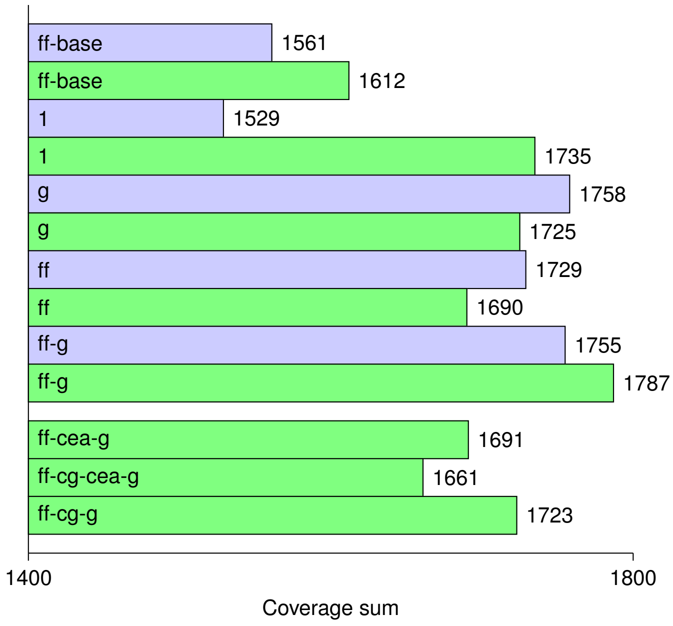

Figure 5.4: Type-based Exploration: Reproduction of Xie et al. (2014b) Table 4. We extended the table with the configurations for cg, and cea.

As expected using longer keys, which implies more heuristics, reduces the quality of the results as the number of the evaluations is greatly increased. In Table 5.4 you can see that the impact of the multiple evaluations is high enough, that even the simple randomisation without any type system ($[const(1)]$) yields better results.
In Table 5.4 you can see that the domain improvement is the best for $[const(1)]$ and $[h^{ff}(), g()]$.

Coverage	**ff-base**	**g**	**ff**	**ff-cea-g**	**ff-cg-cea-g**	**ff-cg-g**	**ff-g**	**one**
airport (50)	**36**	35	**36**	34	33	33	**36**	**36**
assembly (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
barman-sat11-strips (20)	19	17	17	14	13	17	17	**20**
blocks (35)	**35**	**35**	**35**	**35**	**35**	**35**	**35**	**35**
cybersec-sat08-strips (30)	26	**30**	**30**	**30**	**30**	**30**	**30**	**30**
depot (22)	15	17	16	15	14	17	**18**	17
driverlog (20)	18	18	18	18	18	18	**19**	**19**
elevators-sat08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
elevators-sat11-strips (20)	19	18	**20**	12	12	19	16	**20**
floortile-sat11-strips (20)	6	**8**	7	**8**	6	7	**8**	7
freecell (80)	79	**80**	78	77	75	79	**80**	**80**
grid (5)	4	**5**	**5**	**5**	**5**	**5**	**5**	**5**
gripper (20)	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
logistics00 (28)	**28**	**28**	**28**	**28**	**28**	**28**	**28**	**28**
logistics98 (35)	**30**	29	29	25	25	26	29	**30**
miconic (150)	**150**	**150**	**150**	**150**	**150**	**150**	**150**	**150**
miconic-fulladl (150)	136	**139**	137	138	138	**139**	**139**	138
miconic-simpleadl (150)	**150**	**150**	**150**	**150**	**150**	**150**	**150**	**150**
movie (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
mprime (35)	31	31	31	34	**35**	**35**	33	31
mystery (30)	17	**19**	**19**	**19**	**19**	**19**	**19**	18
nomystery-sat11-strips (20)	10	10	10	12	11	18	**19**	10
openstacks (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
openstacks-sat08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
openstacks-sat11-strips (20)	**20**	**20**	**20**	19	19	17	19	**20**
optical-telegraphs (48)	4	4	5	4	5	**6**	**6**	**6**
parcprinter-08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
parcprinter-sat11-strips (20)	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
parking-sat11-strips (20)	**20**	**20**	19	16	13	17	17	19
pathways (30)	10	18	17	17	13	15	**21**	16
pegsol-08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
pegsol-sat11-strips (20)	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
philosophers (48)	**48**	**48**	**48**	**48**	**48**	**48**	**48**	**48**
pipesworld-notankage (50)	33	41	39	39	39	**42**	**42**	41
pipesworld-tankage (50)	21	27	25	28	26	26	28	**29**
psr-large (50)	15	22	18	**23**	20	22	22	20
psr-small (50)	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**
rovers (40)	23	**30**	28	28	27	28	29	27
satellite (36)	**27**	**27**	26	26	26	26	**27**	**27**
scanalyzer-08-strips (30)	27	**30**	**30**	27	27	**30**	**30**	**30**
scanalyzer-sat11-strips (20)	17	**20**	**20**	17	17	**20**	**20**	**20**
schedule (150)	37	69	63	83	81	83	**111**	73
sokoban-sat08-strips (30)	**29**	**29**	**29**	28	**29**	**29**	**29**	**29**
sokoban-sat11-strips (20)	**19**	**19**	**19**	18	**19**	**19**	**19**	**19**
storage (30)	18	22	21	20	21	21	**23**	22
tidybot-sat11-strips (20)	15	16	16	**17**	15	**17**	**17**	16
tpp (30)	23	**25**	18	16	16	17	21	21
transport-sat08-strips (30)	17	16	17	16	16	17	17	**18**
transport-sat11-strips (20)	**1**	0	0	0	0	0	0	0
trucks-strips (30)	17	19	19	22	17	22	**23**	17
visitall-sat11-strips (20)	3	**11**	9	7	7	7	8	7
woodworking-sat08-strips (30)	16	22	20	22	19	22	27	**29**
woodworking-sat11-strips (20)	3	11	8	6	4	7	12	**17**
zenotravel (20)	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
Sum (2112)	1612	1725	1690	1691	1661	1723	**1787**	1735
Average Score in %	76.33	81.68	80.02	80.07	78.65	81.58	**84.61**	82.15
Coverage Score in %	76.00	81.53	79.85	78.19	76.31	80.63	**83.27**	82.15
Domains better than base	0	22	22	19	17	23	**25**	**25**
Domains worse than base	0	6	7	**13**	12	8	7	3

Table 5.4: Type-Based Exploration: Reproduction of Xie et al. (2014b) Table 4. We extended the table with the configurations for cg, and cea.

5.5 Monte-Carlo random walks

For Monte-Carlo random walks we have run the experiment from Nakhost and Müller (2009) Table 1, but also run the experiments for MDA and MHA on the same set of benchmarks. We reproduced the configuration from Nakhost and Müller (2009) Table 3.
The benchmark set consists of 382 problems in 8 domains.

To have an impression of the impact of the *acceptable progress* parameter, we also run the experiment with a basic pure random Monte-Carlo Random Walk search without *acceptable progress*, see Figure 5.5.

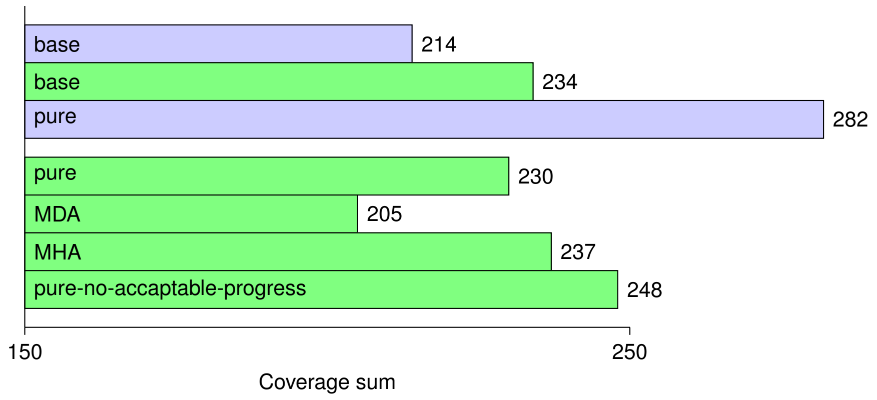

Figure 5.5: Monte-Carlo random walks: Reproduction of Nakhost and Müller (2009) Table 1. We extended the table with the configurations without acceptable progress. Numbers for the original results are estimates from percentage of coverage.

Coverage	**base**	**mda**	**mha**	**pure**	**pure-no-alpha**
airport (50)	36	37	43	44	**46**
optical-telegraphs (48)	4	0	**7**	6	**7**
philosophers (48)	**48**	17	14	13	13
pipesworld-notankage (50)	33	42	**46**	41	45
pipesworld-tankage (50)	21	39	40	41	**42**
psr-large (50)	15	9	**19**	18	18
psr-small (50)	**50**	33	40	40	**50**
satellite (36)	27	**28**	**28**	27	27
Sum (382)	234	205	237	230	**248**
Average Score in %	61.26	53.66	62.04	60.21	**64.92**
Coverage score in %	61.67	54.15	62.19	60.32	**64.83**
Domains better than base	0	4	**6**	5	5
Domains worse than base	0	**4**	2	2	1

Table 5.5: Monte-Carlo Random-Walks: Reproduction of Nakhost and Müller (2009) Table 1. We extended the table with the configurations without acceptable progress.

In Nakhost and Müller (2009) Table 1, ARVAND solved 74% of the problems in the average case. Our implementation with the same configuration only covers 60% of the problems, but without the use of acceptable progress at least 65%. Our implementation of MHA surpasses the results of the pure random walk. In Nakhost and Müller (2009) Table 3, MHA was only applied on the satellite set, where it also outperformed pure random walks.

5.6 Local exploration

We reproduced the basic experiment from Xie et al. (2014a) Table 1 for the heuristics h^{ff}, h^{cg}, h^{cea} in Table 5.6.
The benchmark set consists of 2112 problems in 54 domains.

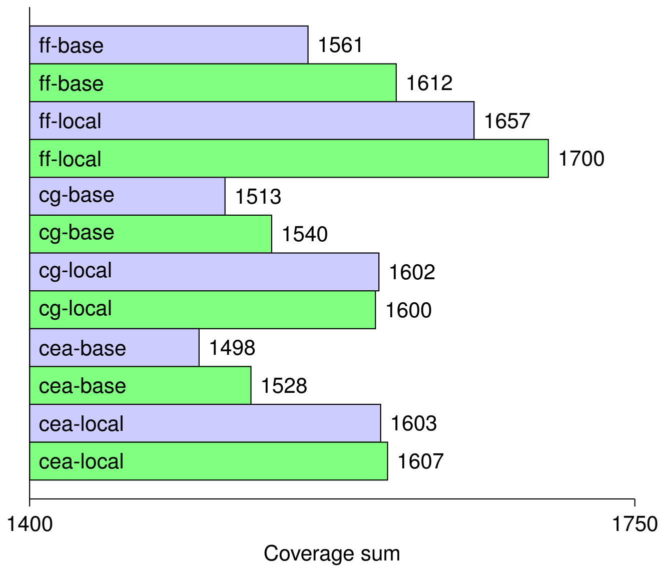

Figure 5.6: Reproduction of Xie et al. (2014a) Table 1 GBFS-LS.

As in the original implementation, the local search outperforms the standard GBFS in all three cases. The gained improvement to the base experiments is also at a similar scale.

Coverage	**cea-base**	**cea-local**	**cg-base**	**cg-local**	**ff-base**	**ff-local**
airport (50)	**45**	41	28	28	36	36
assembly (30)	11	23	12	13	**30**	**30**
barman-sat11-strips (20)	0	0	0	0	**19**	**19**
blocks (35)	**35**	**35**	**35**	**35**	**35**	**35**
cybersec-sat08-strips (30)	12	26	0	2	26	**30**
depot (22)	12	13	12	12	15	**17**
driverlog (20)	18	**20**	18	**20**	18	18
elevators-sat08-strips (30)	28	28	28	**30**	**30**	**30**
elevators-sat11-strips (20)	9	9	9	10	19	**20**
floortile-sat11-strips (20)	**6**	**6**	0	0	**6**	**6**
freecell (80)	79	**80**	75	76	79	**80**
grid (5)	4	4	4	**5**	4	**5**
gripper (20)	**20**	**20**	**20**	**20**	**20**	**20**
logistics00 (28)	**28**	**28**	**28**	**28**	**28**	**28**
logistics98 (35)	**34**	**34**	**34**	**34**	30	28
miconic (150)	**150**	**150**	**150**	**150**	**150**	**150**
miconic-fulladl (150)	**139**	**139**	137	137	136	136
miconic-simpleadl (150)	**150**	**150**	**150**	**150**	**150**	**150**
movie (30)	**30**	**30**	**30**	**30**	**30**	**30**
mprime (35)	34	34	**35**	**35**	31	30
mystery (30)	17	**19**	17	18	17	18
nomystery-sat11-strips (20)	7	7	7	7	**10**	9
openstacks (30)	24	9	26	12	**30**	**30**
openstacks-sat08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**
openstacks-sat11-strips (20)	19	19	13	15	**20**	**20**
optical-telegraphs (48)	**5**	2	1	1	4	4
parcprinter-08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**
parcprinter-sat11-strips (20)	**20**	**20**	**20**	**20**	**20**	**20**
parking-sat11-strips (20)	14	13	**20**	**20**	**20**	**20**
pathways (30)	10	**17**	11	10	10	**17**
pegsol-08-strips (30)	**30**	**30**	**30**	**30**	**30**	**30**
pegsol-sat11-strips (20)	**20**	**20**	**20**	**20**	**20**	**20**
philosophers (48)	**48**	**48**	**48**	**48**	**48**	**48**
pipesworld-notankage (50)	24	28	28	33	33	**35**
pipesworld-tankage (50)	21	23	16	20	21	**26**
psr-large (50)	31	31	31	**32**	15	16
psr-small (50)	**50**	**50**	**50**	**50**	**50**	**50**
rovers (40)	26	32	27	**35**	23	28
satellite (36)	31	31	**36**	**36**	27	28
scanalyzer-08-strips (30)	29	29	**30**	**30**	27	28
scanalyzer-sat11-strips (20)	19	19	**20**	**20**	17	18
schedule (150)	20	50	20	50	37	**81**
sokoban-sat08-strips (30)	5	5	28	28	**29**	**29**
sokoban-sat11-strips (20)	3	3	18	18	**19**	**19**
storage (30)	16	19	18	20	18	**22**
tidybot-sat11-strips (20)	18	**19**	13	16	15	15
tpp (30)	25	27	26	**29**	23	24
transport-sat08-strips (30)	28	28	**30**	**30**	17	18
transport-sat11-strips (20)	9	12	**17**	**17**	1	1
trucks-strips (30)	15	15	14	13	**17**	**17**
visitall-sat11-strips (20)	4	7	4	7	3	**8**
woodworking-sat08-strips (30)	14	**19**	14	17	16	18
woodworking-sat11-strips (20)	2	**6**	2	3	3	5
zenotravel (20)	**20**	**20**	**20**	**20**	**20**	**20**
Sum (2112)	1528	1607	1540	1600	1612	**1700**
Average Score in %	72.35	76.09	72.92	75.76	76.33	**80.49**
Coverage score in %	69.58	73.25	71.43	73.92	76.00	**79.56**
Domains better than base	0	18	0	20	0	21
Domains worse than base	0	4	0	3	0	3

Table 5.6: Reproduction of Xie et al. (2014a) Table 1 GBFS-LS.

5.7 Diverse best-first search

We reproduced the experiment from Imai and Kishimoto (2011) Table 1.
The benchmark set consists of 1612 problems in 32 domains.

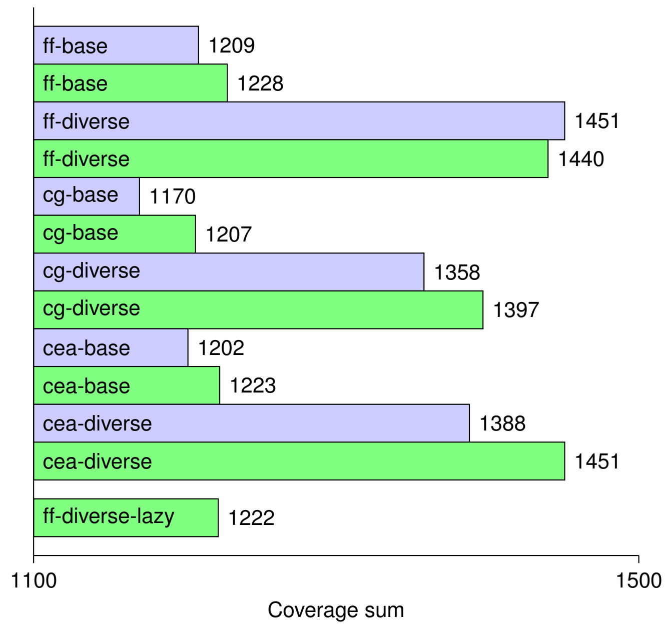

Figure 5.7: Diverse best-first search: Reproduction of Imai and Kishimoto (2011) Table 1.

While deferred evaluation (Subsection 2.5.1) is possible, our experimental results showed that in the current implementation of DBFS, the use of deferred evaluation decreased the quality of the results. This is due to the nature of deferred evaluation which, in case of DBFS, leads to a huge increase of local searches, ending after the first node. This again results in an increased number of evaluations.

Coverage	**cea-base**	**cea-diverse**	**cg-base**	**cg-diverse**	**ff-base**	**ff-diverse**	**ff-diverse-lazy**
airport (50)	**45**	44	28	34	36	43	33
assembly (30)	11	28	12	10	**30**	**30**	**30**
blocks (35)	**35**	**35**	**35**	**35**	**35**	**35**	**35**
depot (22)	12	**19**	12	**19**	15	18	17
driverlog (20)	18	**20**	18	**20**	18	**20**	18
freecell (80)	79	**80**	75	77	79	**80**	**80**
grid (5)	4	**5**	4	**5**	4	4	**5**
gripper (20)	**20**	**20**	**20**	**20**	**20**	**20**	**20**
logistics00 (28)	**28**	**28**	**28**	**28**	**28**	**28**	**28**
logistics98 (35)	**34**	**34**	**34**	**34**	30	28	12
miconic (150)	**150**	**150**	**150**	**150**	**150**	**150**	**150**
miconic-fulladl (150)	**139**	**139**	137	**139**	136	**139**	134
miconic-simpleadl (150)	**150**	**150**	**150**	**150**	**150**	**150**	**150**
movie (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**
mprime (35)	34	**35**	**35**	34	31	33	22
mystery (30)	17	**19**	17	**19**	17	**19**	17
openstacks (30)	24	22	26	22	**30**	**30**	20
optical-telegraphs (48)	5	6	1	1	4	**7**	6
pathways (30)	10	**30**	11	10	10	27	7
philosophers (48)	**48**	**48**	**48**	**48**	**48**	**48**	47
pipesworld-notankage (50)	24	**43**	28	**43**	33	**43**	**43**
pipesworld-tankage (50)	21	29	16	29	21	**34**	**34**
psr-large (50)	**31**	27	**31**	26	15	24	14
psr-middle (50)	**50**	**50**	**50**	**50**	43	49	38
psr-small (50)	**50**	**50**	**50**	**50**	**50**	**50**	**50**
rovers (40)	26	38	27	**40**	23	35	32
satellite (36)	31	31	**36**	35	27	28	12
schedule (150)	20	**142**	20	**142**	37	135	54
storage (30)	16	27	18	**30**	18	26	24
tpp (30)	26	**30**	26	**30**	23	**30**	24
trucks (30)	15	22	14	17	17	**27**	16
zenotravel (20)	**20**	**20**	**20**	**20**	**20**	**20**	**20**
Sum (1612)	1223	**1451**	1207	1397	1228	1440	1222
Average Score in %	75.87	**90.01**	74.88	86.66	76.18	89.33	75.81
Coverage score in %	75.74	**88.87**	75.28	83.99	76.46	87.96	74.92
Domains better than base	0	16	0	14	0	19	10
Domains worse than base	0	3	0	6	0	1	11

Table 5.7: Diverse best-first search: Reproduction of Imai and Kishimoto (2011) Table 1.

6 Comparison of all algorithms

In addition to the experiments from Chapter 5, we have also run the most promising configuration of each algorithm on the IPC 2011[2] benchmark set. Thus we will be able to compare the algorithms on the same set of benchmarks. We have run the experiments twice, once as *eager search* without any search enhancements and once as *lazy search* with deferred evaluation, provided that deferred evaluation is applicable.
We use unit costs for all experiments and as heuristic we use h^{ff}. The setup is the same used in Chapter 5. An explanation of the scores can be found in Section 5.2. The full configurations can be found in Appendix A.
The benchmark set consists of 280 problems in 14 domains.

6.1 Eager search

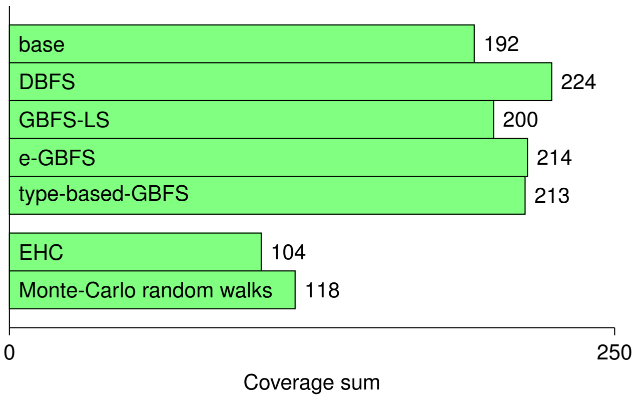

Figure 6.1: Comparison of the coverage of all search algorithms introduced in Chapter 3.

All newly implemented algorithms achieved an higher coverage compared to standard GBFS. Random walks are only included as they have been implemented as part of this thesis. They can not compete with the other algorithms presented here.

[2] International Planning Competition 2011 http://www.plg.inf.uc3m.es/ipc2011-deterministic/.

Coverage	base	**DBFS**	**EHC**	**GBFS-LS**	**Monte-Carlo random walks**	**e-GBFS**	**type-based-GBFS**
barman-sat11-strips (20)	19	**20**	0	19	0	19	19
elevators-sat11-strips (20)	19	**20**	18	**20**	19	**20**	19
floortile-sat11-strips (20)	6	**9**	0	6	2	7	8
nomystery-sat11-strips (20)	10	**19**	3	9	5	10	**19**
openstacks-sat11-strips (20)	**20**	**20**	**20**	**20**	**20**	**20**	**20**
parcprinter-sat11-strips (20)	**20**	**20**	11	**20**	9	**20**	**20**
parking-sat11-strips (20)	**20**	**20**	1	**20**	2	19	16
pegsol-sat11-strips (20)	**20**	**20**	2	**20**	19	**20**	**20**
scanalyzer-sat11-strips (20)	17	**20**	18	18	13	**20**	**20**
sokoban-sat11-strips (20)	**19**	**19**	1	**19**	1	**19**	**19**
tidybot-sat11-strips (20)	15	**17**	10	15	15	**17**	16
transport-sat11-strips (20)	1	**3**	0	1	2	1	0
visitall-sat11-strips (20)	3	**8**	0	**8**	7	6	**8**
woodworking-sat11-strips (20)	3	9	**20**	5	4	16	9
Sum (280)	192	**224**	104	200	118	214	213
Average Score in %	68.57	**80.00**	37.14	71.43	42.14	76.43	76.07
Domains better than base	0	**9**	2	4	3	6	6
Domains worse than base	0	0	**11**	1	8	1	2

Table 6.1: Comparison of the coverage of all search algorithms introduced in Chapter 3.

DBFS performs best. It not only has the biggest improvement for coverage, it also does not decrease the coverage for a single domain.
Typed-based exploration and ϵ-GBFS solve a similar amount of problems and improved on the same amount of domains.

6.2 Lazy search

As deferred evaluation is not applicable for EHC (Section 3.2) and Monte-Carlo random walks (Section 3.4), we did not include them in this section.

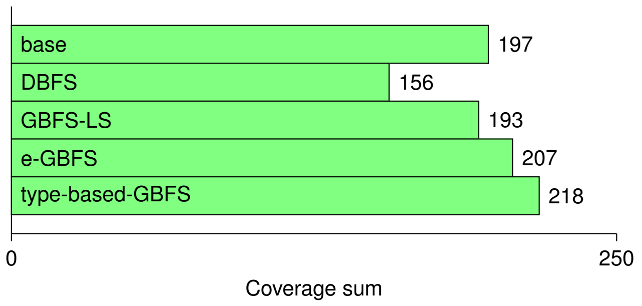

Figure 6.2: Comparison of the coverage of all search algorithms introduced in Chapter 3.

In Section 5.7 we already discovered that the performance of DBFS is massively decreased

Coverage	**base**	**DBFS**	**GBFS-LS**	**e-GBFS**	**type-based-GBFS**
barman-sat11-strips (20)	19	**20**	**20**	18	17
elevators-sat11-strips (20)	**18**	10	12	**18**	**18**
floortile-sat11-strips (20)	6	4	6	6	**7**
nomystery-sat11-strips (20)	9	11	9	10	**18**
openstacks-sat11-strips (20)	**20**	19	**20**	**20**	**20**
parcprinter-sat11-strips (20)	12	10	12	15	**20**
parking-sat11-strips (20)	**19**	6	13	18	17
pegsol-sat11-strips (20)	**20**	**20**	**20**	**20**	**20**
scanalyzer-sat11-strips (20)	17	14	18	**20**	**20**
sokoban-sat11-strips (20)	**19**	17	18	**19**	**19**
tidybot-sat11-strips (20)	14	16	16	16	**17**
transport-sat11-strips (20)	0	**2**	1	0	0
visitall-sat11-strips (20)	5	5	**9**	8	**9**
woodworking-sat11-strips (20)	**19**	2	**19**	**19**	16
Sum (280)	197	156	193	207	**218**
Average Score in %	70.36	55.71	68.93	73.93	**77.86**
Domains better than base	0	4	5	5	**6**
Domains worse than base	0	**8**	3	2	3

Table 6.2: Comparison of the coverage of all search algorithms introduced in Chapter 3 with deferred evaluation.

when deferred evaluation is applied. The deferred evaluation only resulted in a minimal improvement for the standard GBFS and type-based-GBFS. The coverage for all other algorithms is decreased.

In Table 6.2 you can see that for DBFS in 8 of 14 domains the coverage was decreased while it was only increased in 4 domains. Type-based-GBFS is the only algorithm where the number of domains with a higher coverage than the base algorithm did not decrease for deferred evaluation, see Table 6.2 and Table 6.1.

7 Conclusion and future work

7.1 Conclusion

We proved that all algorithms perform as proposed in the original papers. They all show improved coverage over GBFS. The combination of random exploration with a local search in DBFS (Imai and Kishimoto, 2011) showed the best results but already simple randomisation can greatly enhance the coverage. While DBFS performed best, simple randomisation performs also well as we where able to see for ϵ-GBFS (Valenzano et al., 2014) and type-based-GBFS (Xie et al., 2014b).
In Section 5.3 we were able to prove that our implementation of $\epsilon - GBFS$ performs better than the simple implementation and further effort on the improvement of these algorithms seems reasonable.

7.2 Future work

The implementation of those algorithms in the same framework and with this level of abstraction enables us to experiment with many more configurations of experiments than presented here. For example, it would be possible to try an open list other than the `ProbabilisticOpenList` as global open list for DBFS.

While we already tried to reduce the complexity of the DBFS algorithm (Section 4.5) there are still many possibilities for improvement. Our version of the algorithm should also be compared to linear implementation from Imai and Kishimoto (2011).
We also did not explore the full set of configurations for the algorithms and features mentioned in the original papers.

The results for type-based-GBFS with a const key ($[const(1)]$) show that it could be interesting to use the alternating open list in combination with a single bucket open list, basically a specialised version of type-based-GBFS.

There is also still space for optimisation. In Section 5.3 we discovered that the higher memory consumption of the heap implementation causes the search to fail in some cases.

There are many possibilities how we could reduce the memory usage.
In Section 6.1 the results for type-based-GBFS and ϵ-GBFS show that we probably should run experiments with both algorithms on a bigger set of benchmarks to see whether the type system gives any improvement over the simple randomisation in ϵ-GBFS.

Bibliography

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

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

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

Tatsuya Imai and Akihiro Kishimoto. A novel technique for avoiding plateaus of greedy best-first search in satisficing planning. In Wolfram Burgard and Dan Roth, editors, *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011)*, pages 985–991. AAAI Press, 2011.

Hootan Nakhost and Martin Müller. Monte-carlo exploration for deterministic planning. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 1766–1771, 2009.

Silvia Richter and Malte Helmert. Preferred operators and deferred evaluation in satisficing planning. In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 273–280. AAAI Press, 2009.

Richard Valenzano, Nathan R. Sturtevant, Jonathan Schaeffer, and Fan Xie. A comparison of knowledge-based GBFS enhancements and knowledge-free exploration. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, pages 375–379. AAAI Press, 2014.

Fan Xie, Martin Müller, and Robert C. Holte. Adding local exploration to greedy best-first search in satisficing planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*. AAAI Press, 2014a.

Fan Xie, Martin Müller, Robert C. Holte, and Tatsuya Imai. Type-based exploration with multiple search queues for satisficing planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*. AAAI Press, 2014b.

A
Configurations

```
# baseline
./downward --search lazy(single(ff(cost_type=one)),cost_type=one) < output
# the following experiments where run for epsilon = 0.00, 0.05, 0.10, 0.20, 0.30, 0.50,
    0.75, 1.00
# random open list
./downward --search lazy(random(ff(cost_type=one),epsilon=0.50),cost_type=one) < output
# random-buckets open list
./downward --search lazy(random_buckets(ff(cost_type=one),epsilon=0.50),cost_type=one) <
    output
```

Configuration A.1: ϵ-GBFS.

```
#ff-base
./downward --heuristic hff=ff(cost_type=one) --search eager(single(hff)) < output
#ff-typed
./downward --heuristic hff=ff(cost_type=one) --search eager(alt([single(hff),type_based([
    hff,g()])])) < output
#cg-base
./downward --heuristic hcg=cg(cost_type=one) --search eager(single(hcg)) < output
#cg-typed
./downward --heuristic hcg=cg(cost_type=one) --search eager(alt([single(hcg),type_based([
    hcg,g()])])) < output
#cea-base
./downward --heuristic hcea=cea(cost_type=one) --search eager(single(hcea)) < output
#cea-typed
./downward --heuristic hcea=cea(cost_type=one) --search eager(alt([single(hcea),type_based
    ([hcea,g()])])) < output
```

Configuration A.2: type-based-GBFS.

```
#ff-base
./downward --heuristic hff=ff(cost_type=one) --search eager(single(hff)) < output
#one
./downward --heuristic hff=ff(cost_type=one) --search eager(alt([single(hff),type_based([
    const(1)])])) < output
#g
./downward --heuristic hff=ff(cost_type=one) --search eager(alt([single(hff),type_based([g
    ()])])) < output
#hff
./downward --heuristic hff=ff(cost_type=one) --search eager(alt([single(hff),type_based([
    hff])])) < output
#hff-g
./downward --heuristic hff=ff(cost_type=one) --search eager(alt([single(hff),type_based([
    hff,g()])])) < output
#hff-cg-g
./downward --heuristic hff=ff(cost_type=one) --search eager(alt([single(hff),type_based([
    hff,cg(),g()])])) < output
#hff-cea-g
```

```
./downward --heuristic hff=ff(cost_type=one) --search eager(alt([single(hff),type_based([
     hff,cea(),g()])])) < output
#hff-cg-cea-g
./downward --heuristic hff=ff(cost_type=one) --search eager(alt([single(hff),type_based([
     hff,cg(),cea(),g()])])) < output
```

Configuration A.3: type-based-GBFS with different paramter.

```
#base
./downward --search eager(single(ff(cost_type=one))) < output
#pure-no-alpha
./downward --search random_walk(ff(cost_type=one),walk_type=pure(),
     max_explorations_without_improvement=7,walks_per_exploration=2000,default_walk_length
     =10,extending_period=300,extending_rate=1.5) < output
#pure
./downward --search random_walk(ff(cost_type=one),walk_type=pure(),
     max_explorations_without_improvement=7,walks_per_exploration=2000,default_walk_length
     =10,extending_period=300,extending_rate=1.5,weight_for_acceptable_progress=0.9) <
     output
#mha
./downward --search random_walk(ff(cost_type=one),walk_type=mha(tau=10),
     max_explorations_without_improvement=7,walks_per_exploration=2000,default_walk_length
     =10,extending_period=300,extending_rate=1.5,weight_for_acceptable_progress=0.9) <
     output
#mda
./downward --search random_walk(ff(cost_type=one),walk_type=mda(tau=0.5),
     max_explorations_without_improvement=7,walks_per_exploration=2000,default_walk_length
     =1,extending_period=300,extending_rate=2,weight_for_acceptable_progress=0.9) < output
```

Configuration A.4: Monte-Carlo random walks.

```
#ff-base
./downward --search eager(single(ff(cost_type=one))) < output
#ff-local
./downward --heuristic hff=ff(cost_type=one) --search eager_local(eager(single(hff),
     cost_type=one),eager(single(hff),cost_type=one),max_stalled=1000,max_local_tries=100,
     local_search_size=1000) < output
#cg-base
./downward --search eager(single(cg(cost_type=one)),cost_type=one) < output
#cg-local
./downward --heuristic hcg=cg(cost_type=one) --search eager_local(eager(single(hcg),
     cost_type=one),eager(single(hcg),cost_type=one),max_stalled=1000,max_local_tries=100,
     local_search_size=1000) < output
#cea-base
./downward --search eager(single(cea(cost_type=one)),cost_type=one) < output
#cea-local
./downward --heuristic hcea=cea(cost_type=one) --search eager_local(eager(single(hcea),
     cost_type=one),eager(single(hcea),cost_type=one),max_stalled=1000,max_local_tries
     =100,local_search_size=1000) < output
```

Configuration A.5: Local exploration.

```
#ff-base
./downward --search eager(single(ff(cost_type=one))) < output
#ff-diverse
./downward --heuristic hff=ff(cost_type=one) --search eager(diverse(probabilistic(hff,
     probability_for_random_g=0.1,h_probability_base=0.5),single(hff))) < output
#ff-diverse-lazy
./downward --heuristic hff=ff(cost_type=one) --search lazy(diverse(probabilistic(hff,
     probability_for_random_g=0.1,h_probability_base=0.5),single(hff))) < output
#cg-base
./downward --search eager(single(cg(cost_type=one))) < output
#cg-diverse
./downward --heuristic hcg=cg(cost_type=one) --search eager(diverse(probabilistic(hcg,
     probability_for_random_g=0.1,h_probability_base=0.5),single(hcg))) < output
#cea-base
./downward --search eager(single(cea(cost_type=one))) < output
#cea-diverse
./downward --heuristic hcea=cea(cost_type=one) --search eager(diverse(probabilistic(hcea,
     probability_for_random_g=0.1,h_probability_base=0.5),single(hcea))) < output
```

Configuration A.6: DBFS.

```
#base
./downward --search eager(single(ff(cost_type=one))) < output
#e-GBFS
./downward --search eager(random(ff(cost_type=one),epsilon=0.3),cost_type=one) < output
#type-based-GBFS
./downward --heuristic hff=ff(cost_type=one) --search eager(alt([single(hff),type_based([
      hff,g()])]),cost_type=one) < output
#GBFS_LS
./downward --heuristic hff=ff(cost_type=one) --search eager_local(eager(single(hff),
      cost_type=one),eager(single(hff),cost_type=one),max_stalled=1000,max_local_tries=100,
      local_search_size=1000,cost_type=one) < output
#DBFS
./downward --heuristic hff=ff(cost_type=one) --search eager(diverse(probabilistic(hff,
      probability_for_random_g=0.1,h_probability_base=0.5),single(hff)),cost_type=one) <
      output
#EHC
./downward --search ehc(ff(cost_type=one),cost_type=one) < output
#Monte-Carlo random walks
./downward --search random_walk(ff(cost_type=one),walk_type=pure(),
      max_explorations_without_improvement=7,walks_per_exploration=2000,default_walk_length
      =10,extending_period=300,extending_rate=1.5,weight_for_acceptable_progress=0.9) <
      output
```

Configuration A.7: Comparison eager.

```
#base
./downward --search lazy(single(ff(cost_type=one))) < output
#e-GBFS
./downward --search lazy(random(ff(cost_type=one),epsilon=0.3),cost_type=one) < output
#type-based-GBFS
./downward --heuristic hff=ff(cost_type=one) --search lazy(alt([single(hff),type_based([
      hff,g()])]),cost_type=one) < output
#GBFS-LS
./downward --heuristic hff=ff(cost_type=one) --search lazy_local(lazy(single(hff),
      cost_type=one),lazy(single(hff),cost_type=one),max_stalled=1000,max_local_tries=100,
      local_search_size=1000,cost_type=one) < output
#DBFS
./downward --heuristic hff=ff(cost_type=one) --search lazy(diverse(probabilistic(hff,
      probability_for_random_g=0.1,h_probability_base=0.5),single(hff)),cost_type=one) <
      output
```

Configuration A.8: Comparison lazy.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

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

Author — Autor
Patrick von Reth

Matriculation number — Matrikelnummer
08-052-508

Title of work — Titel der Arbeit
Empirical Evaluation of Search Algorithms for Satisficing Planning

Type of work — Typ der Arbeit
Master'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, 01/21/2015

[illegible]

Signature — Unterschrift