Exploration in Greedy Best-First Search for Satisficing Planning

by

Fan Xie

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

This thesis proposes, analyzes and tests different exploration-based techniques in *Greedy Best-First Search* (GBFS) for satisficing planning. First, we show the potential of exploration-based techniques by combining GBFS and random walk exploration locally. We then conduct deep analysis on how flaws in heuristics impact GBFS's performance. *Uninformative Heuristic Regions* (UHRs) and *Early Mistakes* (EMs) for GBFS are analyzed and illustrated on a number of International Planning Competition (IPC) benchmarks. Corresponding solutions, namely *Greedy Best-First Search with Local Exploration* (GBFS-LE) and *Type-based Greedy Best-First Search* (Type-GBFS), are proposed and shown to outperform GBFS substantially. While this thesis mainly focuses on improving coverage (number of problems solved) with exploration-based techniques, we also introduce the *Diverse Anytime Search* (DAS) framework, which reduces *unproductive time* and improves plan quality by randomized exploration. Finally, we integrate these techniques and build the new satisficing planner *Jasper*, which ranked 4th of 20 planners in the Sequential Satisficing track of IPC-2014 and solved the largest number of problems among non-portfolio-based planners.

Preface

Most of the research described in this thesis has been previously published. The following paragraph lists the original publications and describes my contribution to each project if the project is a collaboration with others.

Chapter 3 is based on a publication in ICAPS 2012 [66]. In the paper, I implemented all the code and ran all results. Hootan Nakhost collaborated on developing the algorithm, designing the experiments and preparing the paper. Chapter 4, which is all my original work, is based on a publication that appeared in AAAI 2014 [68]. Chapter 5, which is also all my original work, is based on a publication that appeared in ICAPS 2015 [72]. Chapter 6 is based on a publication in AAAI 2014 [70]. In this paper, Tatsuya Imai implemented the DBFS planner for experimental comparison. I designed the algorithm, and implemented all the experiments. Chapter 7 is based on a publication in ICAPS 2013 [67]. In the paper, Rick Valenzano contributed the idea on how to randomize operators for plan diversity. I designed the algorithm and implemented all the experiments. Chapter 8 is based on a publication in IPC-2014 [69]. I implemented the planner *Jasper*, which is based on the LAMA-2011 [50] code.

To my mother,

Xueqin Chen

Acknowledgements

First of all, I would like to thank my supervisors, Martin Müller and Robert Holte. I am so lucky having both of them as my friends and teachers over my past 5 years. They always provide me freedom to explore what I found interesting, no matter it is a step-out research topic or an industrial internship. In the meantime, they are always there providing guidance and help when I got stumbled. I feel so blessed to have them with me over the past 5 years, and I am so grateful for how much I have learned from them.

I would also like to thank all the committee members, Vadim Bulitko, Zachary Friggstad, and Malte Helmert for their insights and support on my work. Besides being in my committee, Malte was always supportive and helpful of my work over the years, and I am very grateful for that. I would also like to thank Adi Botea and Akihiro Kishimoto for their guidance and support during my internship in IBM Research. I have grown a lot during this industrial research experience, and it is impossible without their help. I would also like to acknowledge the financial support from AITF, NSERC and GRAND.

We have a very active research group on Planning and Heuristic Search in our department. Much of the work in this dissertation is done via collaboration within the group. Here, I would especially thank Hootan Nakhost, Rick Valenzano, and Levi Lelis for being both great coauthors and friends. I would also like to thank Jordan Thayer, Tatsuya Imai and Gabi Röger for their support and help on my work. Many other friends have been there and sharing memories with me over the years. They include Ruitong Huang, Shuoyi Xie, Xiaochao Fan, Dan Han, Zengben Hao, Jing Zhang, Xiaoxiao Li, Gaojian Fan, and many others. It is so wonderful to have them with me, and I will always appreciate their support and help over the years.

I want to say thanks to my dear love, Yanxi. You lighted up my life from the dark. You made me understand what love is. Having you here, my love, explains all. I think I am the luckiest man in the world because I got the chance to know you, approach you, and fall in love with you.

Finally, I want to thank my parents and my brother. It is impossible to get through all these challenges and hard time without your supports. Thank you so much for your unconditional love and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 AI Planning and Heuristic Search

Planning exists in real life. Activities that need planning vary from simple tasks, such as shopping and routing, to complicated tasks, such as logistics problems and even automatically controlling rovers on Mars. Naturally, planning a task efficiently becomes an interesting research question. In Artificial Intelligence (AI), AI Planning is such a research tool that solves the real-life planning problems.

AI Planning systems compute a sequence of actions that achieve a goal. Figure 1.1 shows the basic idea of an AI planning task. The task is to move the package in City B to City C with a truck which is initially located at City C. A valid sample plan would be:

- Move the truck from City C to City B;
- Load the package to the Truck;
- Move the truck from City B to City C;
- Unload the package from the Truck.

Note, the loading and unloading actions are not shown in the diagram for simplicity. The bold states in the diagram illustrate the above solution.

This example is very simple, however it can be made much harder (or more realistic) by adding more cities, packages and trucks. It very quickly becomes hard for human beings to compute the desired solutions, and AI planning techniques start

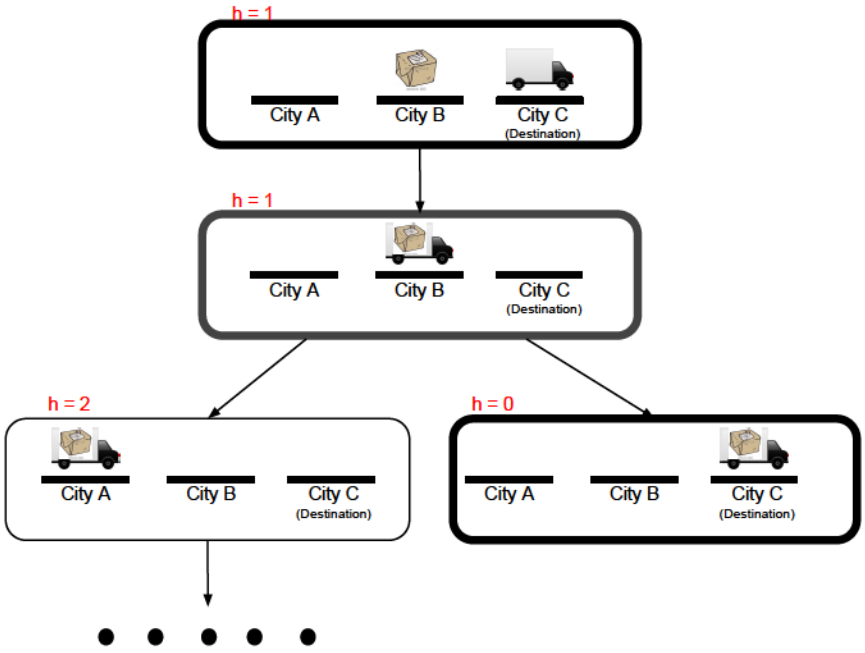


Figure 1.1: An toy logistic example.

being useful. There are several approaches in AI planning, including SAT planning [29], Graphplan planning [3], Hierarchical Task Networks (HTN) [12] and Heuristic Search [4]. Heuristic search has been proven one of the most effective [15, 19, 46, 61].

In heuristic search, there is a function $h(.)$, such that given a state s, $h(s)$ represents the estimated cost of a sequence of actions that transform s to one target state. The high level idea of heuristic search is to selectively search the state space with the help of the heuristic function, in order to find the desired solutions faster. For example, in Figure 1.1, where there are roads between City A and B, and City B and C, assume that the distance between the package and the destination is used as the heuristic value. The heuristic values of these states in Figure 1.1 are shown to the left top of these states. In this example, the heuristic function suggests to take the right branch instead of the left branch, which moves further away from the goal state. Recently, there has been great progress in AI planning during recent International Planning Competitions [15, 61]. Many planning problems are inspired by real applications, such as robot control, transportation and molecular biology.

1.2 Flaws in Heuristic Functions and Exploration

During the past decades, the AI planning community spent much effort in developing strong domain-independent heuristics, such as FF [25], causal graph (CG) [18] and context-enhanced additive (CEA) [21] heuristics. Heuristic search algorithms, which selectively search the state space, can be misled by the flaws in a heuristic. As an example, Greedy Best First Search (GBFS), which plays a core role in modern satisficing planning algorithms, always searches a state with lowest known heuristic value, and is very easily misled by flawed heuristics.

Figure 1.2 illustrates one such example. The task is to move an agent from start location S to goal location G. Black locations are obstacles that the agent can not go through. A state in this problem is modeled by the current location of the agent and the map itself. Assume that the heuristic function estimates the cost from a state s to G by the euclidean distance. States at shaded locations are more promising (lower heuristic value) than 9.8. Therefore, Greedy Best First Search starting from S has to search all these states before it turns to the correct direction to the right of S.

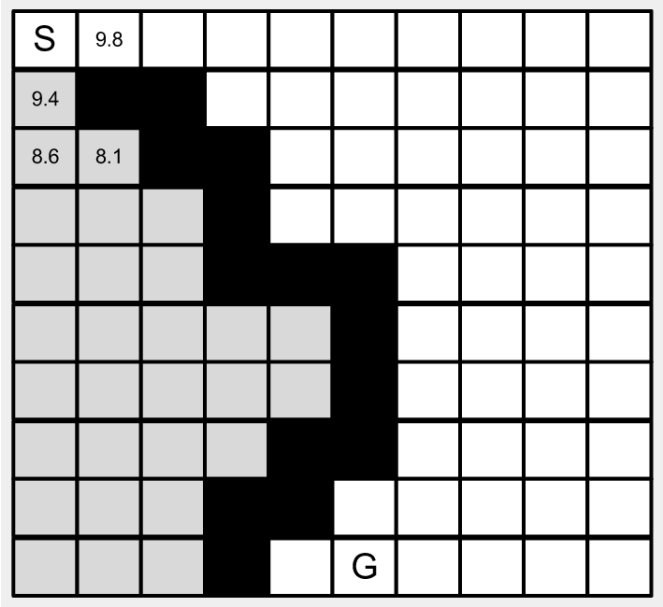


Figure 1.2: The effect of misleading heuristics.

Adding exploration (not following heuristic values) can be a good answer to flawed heuristics. Balancing exploitation and exploration has been well researched

in bandit algorithms [1] and Monte Carlo Tree Search [31], which has major applications in computer games [11] and probabilistic planning [30]. The high level idea is that while the search algorithms are focusing on the current most promising states (exploitation), they also spend a certain effort in trying currently not most promising states (exploration). For the example in Figure 1.2, adding an exploration component means we will also have some chance in searching heuristically less preferred states, such as the state that the one with heuristic value 9.8.

1.3 Contributions of this Thesis

This thesis analyzes some search algorithms on specific planning instances in detail, but proposes exploration-based solutions that can improve heuristic search algorithms in general. Intensive case studies are performed in this dissertation, such as visualizing the *Multiple Uninformative Heuristic Region* problem in 10 different planning domains in Chapter 5, illustrating the biased h-value distribution over open list for one resource constrained domain in Chapter 6, and analyzing the *Unproductive Time* for the famous planner LAMA-2011 [46] in Chapter 7. These case studies are important contributions of this dissertation, which provide strong clues on how exploration should be applied in heuristic search. Besides, different exploration-based techniques are proposed. They combine GBFS with Monte Carlo Random Walk Planning (MRW), add local exploration, trade off frequency and size of local exploration, add type-based exploration, and diversify search for better plan quality. The following sections briefly describe these techniques. Note, most experiments in this dissertation were conducted before the IPC-2014 competition [61]. Since LAMA-2011 was the well-recognized state of the art during that time, these techniques were mostly compared with LAMA-2011.

Chapter 3: Combining GBFS and RWS locally for Better Scaling Behavior

Chapter 3 introduces *Random Walk-Driven Local Search* (RW-LS). RW-LS balances exploration and exploitation by running both local GBFS and random walks, which are sequences of actions which are selected randomly. The algorithm has

been implemented in the system *Arvand with Local Search (Arvand-LS)*. Its performance is evaluated experimentally against other top planners from IPC-2011 on IPC-2011 benchmarks, and on scaled-up instances from several IPC domains. The results show significant improvements in both coverage and plan quality over IPC-2011 planners. This chapter is based on the following publication:

- *F. Xie, H. Nakhost and M. Müller. Planning via Random Walk-Driven Local Search. In Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS-12), 315–322, 2012.* [66].

In the above paper, I implemented all the code and ran all results. Hootan Nakhost collaborated on developing the algorithm, designing the experiments and preparing the paper.

Chapter 4: Adding Local Exploration to Greedy Best First Search

Chapter 4 analyzes the problem of uninformative heuristic regions (UHRs), where heuristic functions provide no guidance, and proposes a two level search framework as a solution. In *Greedy Best-First Search with Local Exploration (GBFS-LE)*, a local exploration is started from within a global GBFS whenever the search seems stuck in UHRs. Two different local exploration strategies are developed and evaluated experimentally: Local GBFS (LS) and Local Random Walk Search (LRW). The two new planners LAMA-LS and LAMA-LRW integrate these strategies into the GBFS component of LAMA-2011. Both are shown to yield improvements in terms of both coverage and search time on benchmarks from the first 7 International Planning Competition, especially for domains that are proven to have large or unbounded UHRs [24, 23]. This chapter is based on the following publication:

- *F. Xie, M. Müller and R. 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), 2388-2394, 2014* [68].

Chapter 5: Understanding and Improving Local Exploration in Greedy Best First Search

Chapter 5 analyzes, quantifies and improves the performance of the Greedy Best-First Search with Local Search (GBFS-LS) algorithm. GBFS-LS adds exploration using a local GBFS to a global GBFS. This substantially improves performance for domains that contain large Uninformative Heuristic Regions (UHRs), such as plateaus or local minima. Planning problems with a mix of small and large UHRs are shown to be hard for GBFS but easy for GBFS-LS. In three standard IPC planning instances analyzed in detail, adding exploration using local GBFS gives more than three orders of magnitude speed up. This chapter also discusses an improved GBFS-LS algorithm, which replaces larger-scale local GBFS explorations with a greater number of smaller explorations. This section is based on the following publication:

- *F. Xie, M. Müller and R. Holte. Understanding and Improving Local Exploration for GBFS. In Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS-2015), 244–248, 2015* [72].

Chapter 6: Type Based Exploration for Early Mistakes in GBFS

Chapter 6 explores how *early mistakes*, which lead the search into a unpromising region of the state space where no easy-to-find solution exists, influence GBFS' performance. This chapter proposes exploration via type systems [35] and multiple queues [19] as one solution.

Utilizing multiple queues [19] in GBFS has been proven to be a very effective approach to satisficing planning. Successful techniques include extra queues based on Helpful Actions (or Preferred Operators), as well as using Multiple Heuristics. One weakness of all standard GBFS algorithms is their lack of exploration. All queues used in these methods work as priority queues sorted by heuristic values. Therefore, misleading heuristics, especially early in the search process, can cause the search to become ineffective.

Chapter 6 introduces a search algorithm that utilizes type systems [35] in a new

way: for exploration within a GBFS multi-queue framework in satisficing planning. A careful case study shows the benefits of such exploration for overcoming deficiencies of the heuristic. The proposed new baseline algorithm Type-GBFS solves almost 200 more problems than baseline GBFS over all 2112 problems from the first 7 International Planning Competitions. This chapter is based on the following publication:

- *F. Xie, M. Müller, R. Holte and T. Imai. Type-based Exploration for Satisficing Planning with Multiple Search Queues. In Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI-2014), 2395-2401, 2014* [70].

In the above paper, Tatsuya Imai implemented the DBFS planner for experimental comparison. I designed the algorithm, and implemented all the experiments.

Chapter 7: Better Anytime Search for Plan Quality with Post-processing

Chapter 7 explores how to generate better quality solutions via randomized exploration. Most of the satisficing planners, which are based on heuristic search, iteratively improve their solution quality through an anytime approach. Typically, the lowest-cost solution found so far is used to constrain the search. This avoids areas of the state space which cannot directly lead to lower cost solutions. However, when used in conjunction with a post-processing plan improvement system such as ARAS [39] this bounding approach can harm a planner's performance since the bound may prevent the search from ever finding additional plans for the post-processor to improve. The new anytime search framework of Diverse Anytime Search (DAS) addresses this issue through the use of restarts, randomization, and by not bounding as strictly as is done by previous approaches.

Chapter 7 shows that by using these techniques, the framework is able to generate a more diverse set of "raw" input plans for the post-processor to work on. We then show that when adding both Diverse Anytime Search and the ARAS post-processor to LAMA-2011, the winner of IPC-2011, the performance according to

the IPC scoring metric improves from 511 points to over 570 points when tested on the 550 problems from IPC 2008 and IPC 2011. Performance gains are also seen when these techniques are added to Anytime Explicit Estimation Algorithm (AEES) [56], as the performance improves from 440 points to over 513 points on the same problem set. This chapter is based on the following publication:

- *F. Xie, R. Valenzano and M. Müller. Better Time Constrained Search via Randomization and postprocessing. In Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS-2013), pages 269-277, 2013* [67].

In the above paper, Rick Valenzano contributed the idea on how to randomize operators for plan diversity. I designed the algorithm and implemented all the experiments.

Chapter 8: The Jasper Planner: the Art of Exploration with GBFS

Chapter 8 integrates all techniques developed in this dissertation into a strong satisficing planner. Jasper is a satisficing planner that builds on LAMA-2011. It adds two modifications. First, it replaces the GBFS algorithm in LAMA-2011 with Type Exploration based Greedy Best-First Search with Local Search (Type-GBFS-LS), which combines GBFS-LS from Chapter 4 and Type-GBFS from Chapter 6. Type-GBFS-LS is an improved version of GBFS that is less sensitive to flaws in heuristic functions. Second, it implements the DAS system as in Chapter 7 for solution improvement, which takes the modified LAMA-2011 as the anytime planner and Aras [39] as the post-processing system.

Jasper ranked the 4th (out of 21 planners) in IPC-2014 Deterministic Satisficing Track. It solved the largest number of problems among all non-portfolio-based planners. This chapter is based on the following publication:

- *F. Xie, M. Müller and R. Holte. Jasper: the Art of Exploration in Greedy Best First Search. In M. Vallati, L. Chrpa, L. and T. McCluskey, The Eighth International Planning Competition, University of Huddersfield, 39–42, 2014* [69].

Other Publications

Except for publications discussed in the dissertation, the other publications created during my PhD time are listed as follows:

- *F. Xie, H. Nakhost, and M. Müller. A local Monte Carlo tree search approach in deterministic planning (abstract). In Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, pages 1832–1833, 2011 [65].*

- *H. Nakhost, M. Müller, R. Valenzano, and F. Xie. Arvand: the art of random walks. In Á. García-Olaya, S. Jiménez, and C. Linares López, editors, The 2011 International Planning Competition, pages 15–16, 2011 [38].*

- *F. Xie. Exploration and combination: Randomized and multi-strategies search in satisficing planning. In Doctoral Consortium of ICAPS-13, 2013 [64].*

- *R. Valenzano, J. Schaeffer, N. Sturtevant, and F. 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, pages 375–379, 2014 [60].*

- *F. Xie, A. Botea, and A. Kishimoto. Heuristic-aided compressed distance databases. In Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence, pages 85–91, 2015 [71].*

- *R. Valenzano and F. Xie. On the Completeness of Best-First Search Variants that Use Random Exploration. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, 2016 [57].*

1.4 Chapter Summary

In this chapter, I described how flaws in heuristics can lead to bad performance for Greedy Best First Search, and proposed adding exploration components to handle this problem. I then outlined our contributions in this direction that use different

kinds of exploration in order to solves these problems and improve the performance of GBFS-based algorithms. Each of these contributions will be described in a separate chapter below. Before doing so, I will first introduce the background of my research as well as the related works that frequently appear in the dissertation in the next chapter.

Chapter 2

Background

AI Planning is a generic tool to solve real life planning problems. Its purpose is to find a sequence of actions starting from the initial state to reach a target state. First, this chapter introduces the representations of a classical planning problem. Second, it introduces the most popular approach for classical planning: *Planning as Heuristic Search* [4].

2.1 Model and Representation

A classical planning task, such as a STRIPS [14] or SAS+ [2] planning task, can be interpreted as a path finding problem in a directed graph, in which nodes represent possible states of the problem and edges represent actions that transit from one state to another. A plan can also be understood as a path in the corresponding graph. It starts from the initial node to a node that represents one of the goal states of the problem. In classical planning, which is the focus of this dissertation, it is assumed that the environment is static and all the available actions have deterministic outcomes. This section reviews the STRIPS [14] and *SAS+* representations [2] for planning problems.

2.1.1 STRIPS Representation

The STRIPS representation uses boolean variables, known as *fluent, fact or atom*, whose domain contains only two values: *True* and *False*. The formal definition of a STRIPS planning problem [14] is:

Definition 1. *(STRIPS).* A STRIPS planning task *is a tuple* $\pi = \langle F, I, G, A \rangle$ *where*

1. F *contains a set of fluents;*
2. $I \subseteq F$ *is the initial state;*
3. $G \subseteq F$ *describes a set of goal states;*
4. A *contains a set of actions, which have the form* $\langle Pre(a), Add(a), Del(a) \rangle$*, where* $Pre(a), Add(a), Del(a) \subseteq F$*.*

The STRIPS representation implicitly describes the state space S, which equals the power set of F, of a planning problem. Each state $s \in S$ is a set of fluents $F(s) \subseteq F$ such that all fluents $f \in F(s)$ have the value ***True*** while fluents $f' \notin F(s)$ have the value ***False***. $I \subseteq F$ represents the initial state. G presents a set of fluents that a goal state must satisfy, namely $S_G = \{s | G \subseteq F(s), s \in S\}$. An action a is only *applicable* in a given state s if $Pre(a) \subseteq F(s)$, and is executed by a transition function $f(s, a) = (F(s) \cup Add(a) \setminus Del(a))$ that sets fluents in $Add(a)$ to be ***True*** and fluents in $Del(a)$ to be ***False***.

2.1.2 SAS+ Representation

Unlike the boolean variables in the STRIPS representation, SAS+ [2] describes states using *multivalued variables*, variables with domains of arbitrary finite size. The formal definition of a SAS+ planning problem [2] is:

Definition 2. *(SAS+).* A SAS+ planning task *is a tuple* $\pi = \langle V, I, G, A \rangle$ *where*

1. V *is a set of state variables. Each variable* $v \in V$ *has an associated finite size domain* D_v*.* $D_v^+ = D_v \cup u$ *is an extended domain, where* u *denotes the* ***undefined*** *value.* $s[v]$ *represents the value of state variable* v *in state* s*. Implicitly, the complete state space is defined as* $S_V = D_1 \times D_2 \times ... \times D_{|V|}$*, and the partial state space is defined as* $S_V^+ = D_1^+ \times D_2^+ \times ... \times D_{|V|}^+$*;*
2. $I \in S_V^+$ *is the initial state;*
3. $G \in S_V^+$ *describes the goal state;*

4. *A contains a set of actions. Each action a has the form $\langle Pre(a), Post(a), Prevail(a)\rangle$, where $Pre(a), Post(a), Prevail(a) \subseteq S_V^+$. For every action a, there are two constraints: 1), for all $v \in V$, if $Pre(a)[v] \neq u$, then $Post(a)[v] \neq u$ and $Pre(a)[v] \neq Post(a)[v]$; 2), for all $v \in V$, $Post(a)[v] = u$ or $Prevail(a)[v] = u$.*

Each state $s \in S^+$ is a complete assignment of state variables in V. Each state variable v can take an defined value from D_v or the *undefined* value, which means unknown or "does not matter". A *complete state* assigns defined values to all state variables, while a *partial state* also assigns the *undefined* value. We say a state s_1 satisfies a state s_2 if for all $v \in V$, $s_2[v] = s_1[v]$ or $s_2[v] = u$. $I \in S^+$ and $G \in S^+$ represent the initial state and goal state. An action a is modeled by $\langle Pre(a), Post(a), Prevail(a)\rangle$, where $Pre(a), Post(a), Prevail(a)$ are all partial states:

- $Pre(a)$ specifies the defined values of these changed variables before the action a is applied;
- $Post(a)$ specifies the defined values of these changed variables after the action a is applied;
- $Prevail(a)$ specifies the defined values of required unchanged variables before the action a is applied.

An action a is *applicable* in a given state s if s satisfies both $Pre(a)$ and $Prevail(a)$. The resulting state is computed by updating the changed variables to the values specified in $Post(a)$.

Optimal Planning and Satisficing Planning

In a classical planning task, every action a has a positive cost $cost(a)$. A solution (or plan) is a sequence of actions $a_0, ..., a_n$ that transform the initial state to one goal state. The plan is optimal if it minimizes sum of action costs $\sum_{i=0}^{n} a_i$. Optimal Planning only looks for optimal plans, but does not scale to large problems because of the hardness to prove plan optimality. Satisficing Planning does not require plan

optimality in order to scale to larger problems. The Satisficing track in the International Planning Competitions (IPCs) [15, 61] measures both coverage (number of problems solved) and plan quality (sum of action cost).

2.2 Planning As Heuristic Search

Heuristic search is the most effective approach in classical planning [15, 19, 46, 61]. A heuristic estimator guides search in the corresponding graph of the state space. Since the seminal paper by Bonet and Geffner [4], which introduced two domain-independent heuristics h^{add} and h^{max}, much research applied heuristic search to classical planning. Many top planners are based on heuristic search techniques. Examples are FF [25], Fast Downward [19] and LAMA [47]. The remainder of this section introduces popular domain-independent heuristics, heuristic search algorithms, and commonly used search enhancements.

2.2.1 Domain Independent Heuristics

Since the introduction of h^{add} and h^{max}, there has been remarkable progress in developing domain-independent heuristics, including the Fast Forward (FF) heuristic h^{FF} [25], the Causal Graph (CG) heuristic h^{cg} [18], the Context Enhanced Additive heuristic h^{cea} [21], and the Landmark Count (LM) heuristic h^{lm} [48]. In this section, we briefly discuss the most popular family of domain-independent heuristic, the delete relaxation based heuristics, namely h^{add}, h^{max} and h^{FF}. While these heuristics are discussed based on the STRIPS representation, for which they were initially introduced, they can also be derived for the SAS+ representation.

Delete-Relaxation Based Heuristics

Domain-independent heuristics are often derived from solving a simpler version of the problem, called the *relaxation* π^+ of the original problem π. The relaxation abstracts some conditions, such as the delete list in STRIPS actions, so that any solution in the problem is also a solution in the relaxation. The most popular way of relaxation is the *delete-relaxation*, which removes all delete lists $Del(a)$ of a

problem. Since no fluent can become *False* from *True*, the number of fluents that have the value *True* increases monotonically. Let A^+ denote the actions set without delete list. A sequence of actions $a_0^+, ..., a_n^+ \in A^+$ is an optimal plan for π^+ if it transforms the initial state to a goal state and minimizes the sum of actions $\sum_{i=0}^{n} a_i^+$. Let p^+ denote one optimal plan for π^+, then $cost(p^+)$ is an admissible heuristic known as h^+, which never overestimates the cost. $cost(p^+)$ is also called the optimal delete relaxation heuristic. Although the computational complexity of finding a solution (if any) is bounded by the number of fluents in F, computing an optimal solution for π^+ is NP-Complete [5]. An approximate solution for π^+ can be calculated in polynomial time and quickly enough for solving planning problems in practice [4, 25].

The Max and Additive Heuristics: both h^{add} and h^{max} approximate the optimal delete relaxation heuristic h^+ by aggregating the costs of making a set of fluents *True*. The cost of making one fluent *True* is decided by the action a that achieves it with the minimal estimated cost, which is calculated from the costs of making all fluents in $Pre(a)$ *True* plus the cost of a itself. h^{max} approximates h^+ by the minimal estimated cost of the most expensive single fluent, assuming that the cost of achieving a set of fluents is equal to that of the most expensive fluent. The formal definition of h^{max} for fluents, actions and sets of fluents is:

$$h^{max}(f;s) = \begin{cases} 0 & iff\, f \in s \\ \infty & A(f) = \emptyset \\ \min_{a \in A(f)} h^{max}(a;s) & otherwise \end{cases} \tag{2.1}$$

$$h^{max}(a;s) = cost(a) + h^{max}(Pre(a);s) \tag{2.2}$$

$$h^{max}(Q;s) = \max_{f \in Q} h^{max}(f;s) \tag{2.3}$$

$$h^{max}(s) = h^{max}(G;s) \tag{2.4}$$

$A(f)$ denotes the set of actions that can achieve fluent f. If there exist zero cost actions, there might be multiple value assignments that satisfy the above equations. This issue can be solved by computing the heuristic values of fluents and actions in a specific order. First, compute heuristic value of fluents in s, which are all 0.

Then, compute heuristic values of actions that are applicable from the current state, followed by computing heuristic values of new fluents that are achieved by these actions and adding these fluents into the current state. At the same time, update the heuristic value of already achieved fluents if they are achievable by lower heuristic values. The process is repeated until heuristic values of all fluents in G are computed.

h^{max} is an *admissible* heuristic [4] and $h^{max}(s) \leq h^{+}(s)$ given any s [4]. However, in practice h^{max} is less informative than h^{add}, which is not admissible. h^{add} assumes costs of achieving a set of fluents is the sum of achieving each independently and approximates h^{+} in this way. The formal definition of h^{add} for fluents, actions and sets of fluents is:

$$h^{add}(f;s) = \begin{cases} 0 & iff\ f \in s \\ \infty & A(f) = \emptyset \\ \min_{a \in A(f)} h^{add}(a;s) & otherwise \end{cases} \tag{2.5}$$

$$h^{add}(a;s) = cost(a) + h^{add}(Pre(a);s) \tag{2.6}$$

$$h^{add}(Q;s) = \sum_{f \in Q} h^{add}(f;s) \tag{2.7}$$

$$h^{add}(s) = h^{add}(G;s) \tag{2.8}$$

Similar to h^{max}, if there exist zero cost actions, there might be multiple value assignments that satisfy the above equations. The same process described for h^{max} can be used to resolve this issue. $h^{add}(s) \geq h^{+}(s)$ given any state s [4]. Very often, one action achieves more than one desired fluent, which causes the problem of over-counting for h^{add}.

The Fast Forward Heuristic: h^{FF} attacks the over-counting issue in h^{add} by returning the cost of a solution for the delete-relaxed problem π^{+}. There are two steps in computing such a plan [25]: 1) build a relaxed planning graph, which alternates between fluent layers, containing current *True* fluents, and action layers, containing actions that lead to new *True* fluents; 2) extract a solution from the graph. Unlike h^{max} and h^{add}, h^{FF} computes a real plan for problem π^{+}. In this plan, an action can

Algorithm 1 Best First Search

Input Initial state I, goal states G
Output A solution plan

```
1: open.insert(I, 0, h(I))
2: while open ≠ ∅ do
3:    n ← open.remove_min() {The open list is sorted differently for different
      algorithms.}
4:    if n ∈ G then
5:       return plan from I to n
6:    end if
7:    closed.insert(n)
8:    for each v ∈ successors(n) do
9:       if v ∉ closed then
10:         open.insert(v, g(n) + cost(n, v), h(v))
11:      else
12:         if g(n) + cost(n, v) < g(v) then
13:            closed.remove(v)
14:            open.insert(v, g(n) + cost(n, v), h(v))
15:         end if
16:      end if
17:   end for
18: end while
```

make multiple fluents *True*. The cost of the plan is returned as the heuristic value. h^{FF} is not admissible but it is more informative in practice than h^{add} [25]. Overall, for any state s, we have $h^{max}(s) \leq h^{+}(s) \leq h^{FF}(s) \leq h^{add}(s)$ [4, 25].

2.2.2 Heuristic Search Algorithms

A heuristic function is important for planning systems since it provides guidance for future search. However, a search algorithm is also essential for the performance of planners. Two main types are used in satisficing planners: Best First Search [19, 36, 46] and Local Search [25, 38].

Best First Search Algorithms

Best First Search algorithms always expand the most promising state according to a function called $f(.)$, and keep all successors of the state into in a priority queue called called *Open List*. All states that are already expanded are stored in a data

Algorithm 2 Enforced Hill Climbing

Input Initial state I, goal states G
Output A solution plan

```
1:  open.insert(I)
2:  h_best ← h(I)
3:  while open ≠ ∅ do
4:      cur ← open.remove()
5:      succs ← cur.succssors()
6:      while succs ≠ ∅ do
7:          n ← succs.remove()
8:          if n ∈ G then
9:              return plan from I to n
10:         end if
11:         if h(n) < h_best then
12:             open.clear()
13:             succs.clear()
14:             h_best ← h(n)
15:         end if
16:         open.insert(n)
17:     end while
18: end while
```

structure called *Closed List*. Given a state s, $f(s)$ is usually a linear combination of $g(s)$, the best known total cost of a sequence of actions that reaches s from the start state, and $h(s)$, the estimated cost to reach a goal state from s. For example, the best first search algorithm *A** [17] takes $f(s) = g(s) + h(s)$. In this case, when $h(s)$ is *admissible*, never overestimating the cost to reach a goal state, solutions found by A* are optimal. Greedy Best First Search (GBFS), where $f(s) = h(s)$, is one of the most important best first search algorithms for satisficing planning. GBFS always expands a state that is closest to a goal state according to $h(.)$ without considering $g(s)$. GBFS can often find a solution quickly. Many state-of-the-art satisficing planners, such as *LAMA* [46] and *Fast Downward* [19], run GBFS to quickly find a solution first before looking for better quality solutions. Flaws in a heuristic can significantly decrease the performance of GBFS, which will be discussed in detail in Chapters 4, 5 and 6. Another variant of Best First Search is Weighted A* [44], in which $f(s) = g(s) + w * h(s)$. Here, $w \geq 1$ is a parameter that controls the trade-off between plan quality and search time. The larger w is, the more emphasis is put on

expanding states that are closer to goal states. $w = 1$ corresponds to A*. Algorithm 1 shows the pseudo code of the Best First Search framework. A*, GBFS and WA* differ from each other in how the $open$ list is ordered.

Local Search Algorithms

Local search algorithms have also been well explored in the planning community. One famous such algorithm is *Enforced Hill Climbing* (EHC) [25]. As in the standard Hill Climbing algorithm [53], EHC expands a current state s and chooses a successor state with lowest h value (must be smaller than $h(s)$) as the new current state. This process iterates until either the current state is a goal state or no such successor state exists. EHC differs from Hill Climbing in local minima, when none of the successor states has a lower h value than s. In this case, EHC switches to exhaustive breadth first search until finding a state s' with $h(s') < h(s)$. EHC fails when no such exit state s' is found after EHC exhausts all states reachable from s. Algorithm 2 shows the pseudo code of Enforced Hill Climbing. The planner FF [25] starts with an EHC search. If it fails, FF restarts with GBFS from the initial state.

Monte Carlo Random Walk Planning (MRW) [38] is another local search based algorithm that applies the Monte Carlo sampling ideas from computer game research in classical planning. Algorithm 3 shows the pseudo code. In Line 5, *IsDeadEnd(s)* returns *True* if the heuristic function think there is no solution node in the subtree under s. In MRW, quick random walks are performed to explore the neighborhood of the current search state s. A random walk starting from s_0 ($s_0 = s$) is a sequence of states $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n$, in which each transition action $s_k \rightarrow s_{k+1}$ is randomly selected from among the applicable actions for s_k. Only the end state s_n of the random walk is evaluated by a heuristic function h such as h^{FF} [25]. After a number of random walks from the current start state s, the search greedily updates s to an end state of minimum h-value. This process of performing a set of random walks, then jumping to a best endpoint is called a *search step*. The algorithm terminates when a goal state is found. If the minimal heuristic value is not improved for *MAXSTEPS* steps, the algorithm restarts from I.

2.2.3 Planning Enhancements

Besides the basic search algorithms and heuristics, successful heuristic search based satisficing planners use many search enhancements. This section introduces the three most commonly used search enhancements: *Deferred Evaluation* [45], *Preferred Operators* [19] and *Multiple Heuristics* [51]. The latter two enhancements are typically used under the *Multiple Queue Search* framework, which is also described in this section.

Deferred Evaluation

Domain independent heuristics are generally expensive to compute. Most of the run time of heuristic search based planners is spent in computing heuristic values. Once a node is expanded, its successors are evaluated by $h(n)$ before being put into the open list. Only a small part of the nodes will eventually be expanded. Helmert [19] proposed a technique called *Deferred Evaluation*, in which nodes are evaluated by

Algorithm 3 Monte-Carlo Random Walk Search

Input Initial state I, goal states G
Paramter *MAXSTEPS* **Output** A solution plan

1: $s \leftarrow I$
2: $h_{min} \leftarrow h(I)$
3: $counter \leftarrow 0$
4: **while** $s \notin G$ **do**
5: **if** $counter > MAXSTEPS$ **or** $IsDeadEnd(s)$ **then**
6: $s \leftarrow I$ {restart from the initial state}
7: $h_{min} \leftarrow h(I)$
8: $coutner \leftarrow 0$
9: **end if**
10: $s \leftarrow RandomWalkSearch(s, G)$ {return the endpoint with the lowest heuristic value of a set of random walks.}
11: **if** $h(s) < h_{min}$ **then**
12: $h_{min} \leftarrow h(s)$
13: $counter \leftarrow 0$
14: **else**
15: $counter \leftarrow counter + 1$
16: **end if**
17: **end while**
18: **return** the plan from I to s

the heuristic value of their parent. The nodes in the open list are not evaluated until they are expanded, which substantially reduces the number of node evaluations, especially when combined with *Preferred Operators* (see next section).

Multiple Queue Search

Two of the most important enhancements for satisficing planning, Multi-Heuristics and Preferred Operators, are based on the *Multiple Queue Search* [19, 52] framework. When more than one heuristic is used in the search algorithm, Multiple Queue Search uses one priority queue for each heuristic, and selects the next node to expand from these queues by a strategy, such as round-robin. Queue boosting [19] can be used to assign priorities to different queues. Once a node is expanded, all its successors are evaluated by all heuristics, and put into every queue with the corresponding value. *Preferred Operators* is a term coined by Helmert [19] in the spirit of *Helpful Actions* [25], to represent all operators that are promising for some reason. For *Preferred Operators*, one additional priority queue per heuristic is used, which contains only these successors. Usually the preferred operator queue(s) are also boosted. The search benefits both from focusing more on usually relevant actions, and from reaching greater depths more quickly because of the smaller effective branching factor in the preferred queues.

2.3 Chapter Summary

In this section, we first described the model and representation we use to present planning problems. We then focus on the heuristic search based solution for satisficing planning, by describing the three major components: Planning Heuristic, Search Algorithms and Planning Enhancements.

Chapter 3

A Taste of Exploration: Combining Greedy Best First Search and Stochastic Random Walk

This chapter describes a new search algorithm called *Random Walk-Driven Local Search* (RW-LS). RW-LS combines the systematic greedy algorithm Greedy Best-First Search (GBFS) and the exploration-based algorithm Random Walk Search (RWS) in a local search manner. This new algorithm is compared with the two closely related algorithms: GBFS and RWS. An new set of benchmarks IPC-2011-LARGE are generated to show the limit of top planners in IPC-2011 as well as the better scaling behavior of the proposed algorithm. This chapter is based on the following publication:

- *F. Xie, H. Nakhost and M. Müller. Planning via Random Walk-Driven Local Search. In Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS-12), 315–322, 2012 [66].*

3.1 Introduction

Most successful current satisficing planners combine several complementary search algorithms [15, 46, 61]. Examples range from portfolio planners such as Fast Downward Stone Soup [22] and loosely coupled parallel planners such as ArvandHerd [58] to systems which cycle through several search strategies, such as FF [25], FD [19], and ArvandHerd, and dual queue search algorithms as in LAMA [46]. Other

examples are Probe [36], which combines explorative probes with standard search techniques, Arvand [38], a planner which alternates exploration by Random Walks with a greedy hill-climbing strategy, and Roamer [37], which combines a best-first strategy with the use of random walks to escape from plateaus and local minima. Another successful planning technique has been post-processing for plan improvement, as in the Aras system [39].

The main contribution of this chapter is *Random Walk-Driven Local Search* (RW-LS), a new planning method which combines local search with random walks. In RW-LS, a greedy best-first search is driven by two kinds of evaluations: direct evaluation of tree nodes, and evaluation of random walk endpoints as in Arvand.

Another contribution is *IPC-2011-LARGE*, a set of scaled up test instances for several of these domains. Such scaled up instances are useful to show the limit of these top planners in IPC-2011.

This chapter is organized as follows: After the new planning algorithm RW-LS is introduced and compared with related work, the RW-LS planner Arvand-LS is described next, followed by a section about the generation and selection of harder problems from existing IPC domains for which scalable problem generators are available. The experimental results for Arvand-LS show strong improvements compared to Arvand and LAMA-2011 in both coverage and plan quality for hard problems from several IPC domains.

3.2 Monte Carlo Random Walks Local Greedy Best-First Search

The MRW algorithm emphasizes fast exploration by random walks, while leaving plan improvement to a postprocessor such as Aras. In contrast, RW-LS focuses on plan quality during search by performing a local Greedy Best First Search in each step. Figure 3.1 compares the search strategies of MRW and RW-LS. Both algorithms use random walks to explore the search space near a starting point s_0. After each exploration phase, both algorithms update s_0 to an explored state with minimum h-value and start the next search step from this new s_0. Unlike MRW, RW-LS

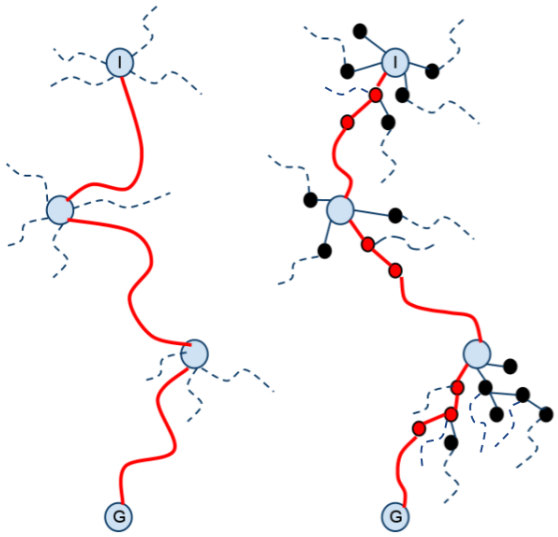


Figure 3.1: The search strategies of MRW (left) and RW-LS (right).

performs a local Greedy Best-First Search starting from state s_0 during exploration. The search uses well-known enhancements such as Deferred Evaluation and a second open list containing only states reached via preferred operators [19]. RW-LS evaluates the best state s retrieved from an open list, and also performs a random walk starting from s ending in a state r. A linear combination of the heuristic values $h(s)$ and $h(r)$ is used to order the nodes in the open list.

Algorithm 4 The MRW-like main loop of RW-LS

Input Initial State I and goal condition G
Output A solution plan

$(s, h_{min}) \leftarrow (I, h(I))$
while $G \not\subseteq s$ **do**
  $(s, status) \leftarrow$ IteratedRW-LS(s, G)
  if $status =$ NO_PROGRESS **or** IsDeadEnd(s) **then**
    $(s, h_{min}) \leftarrow (I, h(I))$ {restart from initial state}
  else
    $h_{min} \leftarrow h(s)$
  end if
end while
return the plan from I to s

Algorithm 4 shows an outline of the MRW framework [38], adapted to RW-LS. A state s is DeadEnd if there is no solution under the subtree of s. The only

difference in the top-level algorithm is the call to *IteratedRW-LS* instead of doing pure exploration by random walks. A successful search returns the sequence $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n$ of states along a plan, with $s_0 = I$ the initial state, $s_n \supseteq G$ a goal state and each partial path $s_k \rightarrow s_{k+1}$ obtained by a search step starting from s_k. The main search loop fails if *IteratedRW-LS* reports no progress, or if the current search state becomes a dead end. In these cases, the algorithm simply restarts from the initial state I.

IteratedRW-LS, shown in Algorithm 5, attempts to find an improved state s' by calling RW-LS a limited number of times. Each call constitutes one search step. If the minimum h-value does not improve, *IteratedRW-LS* returns no progress.

Unlike MRW, RW-LS performs a random walk-guided greedy best-first search as its search step. This method is shown in Algorithm 6.

$RandomWalk(s, G)$ performs one random walk from a state s. As in MRW [38], each random walk either returns the end state r reached after a given maximum number of random actions, or terminates early on encountering a goal or dead end.

Algorithm 5 Iterated RW-LS

Input State s and goal condition G
Parameter MAXSTEPS
Output New s and progress status

```
open ← [s]
closed ← []
for i = 1 to MAXSTEPS do
   s' ← RW-LS(s, G, open, closed)
   if IsNotDeadEnd(s') and h(s') < h(s) then
      return (s', PROGRESS)
   end if
end for
return (s, NO_PROGRESS)
```

3.2.1 Random Walk-Driven Local Search

Algorithm 6 shows the random walk-guided greedy best-first search algorithm. Open and closed lists are shared between iterations. RW-LS performs a random walk from each state retrieved from the open list, and evaluates the endpoint r. The

Algorithm 6 *RW-LS*, random walk-driven local search

Input state s_0, goal condition G, open, closed
Parameter NUMWALKS
Output Best state s_{min} or DEAD_END

```
(s_min, h_min) ← (s_0, h(s_0))
for i = 1 to NUMWALKS do
  if open.empty() then
    return DEAD_END
  end if
  s ← open.remove_min() {best open node in tree}
  if G ⊆ s then
    return s
  end if
  r ← RandomWalk(s, G)
  if G ⊆ r then
    return r
  end if
  if h(s) < min(h_min, h(r)) then
    (s_min, h_min) ← (s, h(s))
  else if h(r) < h_min then
    (s_min, h_min) ← (r, h(r))
  end if
  closed.insert(s)
  open.insert(s.children() − closed, w × h(s) + h(r))
end for
return s_min
```

algorithm keeps track of, and returns the best state seen overall, either from *closed* or from a random walk endpoint. States in $open$ are ordered by a linear combination $w \times h(s) + h(r)$. The parameter w controls the trade-off between exploration and exploitation. The variable $open$ in the algorithm manages both the normal and the preferred operator open lists. The method also keeps track of the path (action sequence) leading from s to the best state s_{min}. In case s_{min} was a random walk endpoint, this path consists of an in-tree prefix followed by the random walk. For simplicity, the details of path handling are omitted in the pseudocode.

If RW-LS fails to find an improvement, it returns the input state. Unlike MRW, there is no forced transition to an end point in this case, and *IteratedRW-LS* repeatedly calls RW-LS from the same start state, while keeping the open and closed lists

and increasing the parameter *NUMWALKS*.

3.2.2 Comparison to Other Related Algorithms

The RW-LS algorithm has some similarities to Enforced Hill Climbing [25]. Both algorithms perform a local search before transitioning to the next search state. The major differences are: 1) EHC uses a uniform, breadth first exploration strategy, while RW-LS combines a selective GBFS with random walks to explore a larger neighborhood; 2) EHC does not use heuristic values to guide the search, until it finds a state with improved h-value, while RW-LS uses a linear combination of $h(s)$ and $h(r)$ to guide the greedy best-first search.

Random-Walk Assisted Best-First Search (RWA-BFS) [37] is more closely related to RW-LS, as it combines best-first search with random walks. However, random walks in RWA-BFS are added to a global, complete best-first search. In contrast, RW-LS is a stochastic local search algorithm. As a complete global search algorithm, RWA-BFS outperforms RW-LS in domains which seem to require some exhaustive search, such as Sokoban and Parking. RW-LS shines in solving large problems for which its combination of local search and exploration is effective. Furthermore, while RW-LS runs random walks from every expanded node, RWA-BFS uses them only when the search is considered to be stuck in a local minimum.

3.3 Arvand-LS: A Simple Planner based on RW-LS

Arvand-LS is an implementation of the RW-LS algorithm based on the IPC-2011 version of Arvand [41], called Arvand-2011 here. As outlined in Algorithm 4, Arvand-LS replaces the pure exploration by random walks of Arvand-2011 by RW-LS. For most other parts of the planner, the Arvand-2011 default settings are used, such as the maximal number of local searches before restarting from the initial state, $MAXSTEPS = 7$ in Algorithm 5. Arvand-LS first runs each search step with $NUMWALKS = 100$ random walks. For each restart from I, the number of random walks is doubled, up to a maximum of 3200. There are two reasons for this doubling strategy: as long as no plan to a goal state is found, wider exploration increases the

probability of finding a solution; after a solution is found, larger searches increase the fraction of in-tree actions, as opposed to random walk actions, in the solution, which helps improve plan quality. The weight w in the combined heuristic function of RW-LS is set to a large value, $w = 100$, after some initial experiments on IPC-2011. $h(r)$ is often used only for tie-breaking, but it seems to be very successful in that because it makes the search more informed in large plateaus. In our testing, the algorithm was robust against changes to $MAXSTEPS$ in the interval [1,14]. Performance declined slowly for larger values of this parameter.

How random walks of Arvand-LS are performed is controlled by a tuple of parameters:

- $(len\_walk, e\_rate, e\_period, WalkType)$

Random walk length scaling is controlled by an initial walk length of len_walk, an extension rate of e_rate and an extension period of $NUMWALKS * e\_period$. The algorithm performs random walks with length cur_len_walk, which initially equals len_walk. If h_{min} does not improves over $NUMWALKS * e\_period$ random walks, cur_len_walk will be updated to $cur\_len\_walk * (1 + e\_rate)$. The choices for $WalkType$ are MHA and MDA [38], which use online statistics of $helpful\ actions$ and $deadend\ states$ respectively to bias the action selection in random walks. For example, a configuration of $(1, 2, 0.1, MHA)$ means that all random walks use the MHA enhancement, and if h_{min} does not improve for $NUMWALKS * 0.1$ random walks, then the length of walks, len_walk, which starts at 1, will be doubled. The same three configurations used in Arvand are used:

- *config-1*: $(10, 2, 0.1, MHA)$
- *config-2*: $(1, 2, 0.1, MDA)$
- *config-3*: $(1, 2, 0.1, MHA)$

The algorithm cycles through these configurations, starting with *config-1* and changing at each restart.

Integration with the Aras postprocessor is identical to Arvand-2011: Aras is run on each new plan found until it reaches the memory limit. Compared to the original postprocess-once method [39], this applies Aras to a larger variety of input plans.

One major difference between Arvand-2011 and Arvand-LS is less pruning: Arvand-LS turns off the pruning in the search or random walks for states whose cost so far exceeds the cost of the best found solution. This pruning does not work well for Arvand-LS since its high quality initial plans often prevent the algorithm from finding any other alternate plans. This pruning strategy is further discussed in Chapter 7.

3.4 IPC-2011-LARGE: Scaling up IPC-2011 Domains

In several domains, the test instances used at IPC-2011 have become too easy for the current top planners. Harder instances are needed to show that a new approach advances the state of the art in coverage as well as in plan quality. The collection *IPC-2011-LARGE* represents a new, more challenging test set.

Four out of fourteen IPC-2011 domains have been scaled up to harder problems in *IPC-2011-LARGE* to help illustrate the performance of Arvand-LS: *Woodworking*, *Openstacks*, *Elevators* and *Visit-All*. In *Woodworking*, wood must be processed using different tools to produce parts. Planners should minimize the processing cost. *Openstacks* is a combinatorial optimization problem where products must be temporarily stacked. Planners need to minimize the maximum number of stacks in simultaneous use. In *Elevators*, passengers need to be transported using two types of elevators with different cost characteristics. In *Visit-All*, an agent located in the center of a $n \times n$ grid must visit all cells of the grid while minimizing the number of moves.

Regarding the other IPC-2011 domains, *Transport*, *Tidybot*, *Nomystery* and *Floortile* seem hard enough for current planners, so scaling can wait. In *Parking*, *Sokoban* and *Barman*, neither Arvand nor Arvand-LS scale very well - see the discussion in the IPC-2011 results section below. Depending on the needs of the community, *IPC-2011-LARGE* could be expanded in the future to include harder instances from

those domains. For the remaining domains of *Parcprinter*, *Pegsol* and *Scanalyzer*, no scalable generators were available.

3.4.1 IPC-2011-LARGE Parameter Settings

Parameters used in most *IPC-2011-LARGE* instances are larger than the hardest problems in IPC-2011. To describe a range of parameters, the notation $p \in [L..U; I]$ is used to indicate that parameter p varies from a lower limit of L to an upper limit of U in increments of I. For example, $p \in [10..30; 5]$ means that the generated instances have values of $p \in \{10, 15, 20, 25, 30\}$. For these domains, the parameters are selected so that none of the planners can solve all of them.

Elevators The hardest IPC-2011 problem has $f = 40$ floors and $p = 60$ passengers. The new problems also fix $f = 40$, and vary $p \in [20..305; 15]$.

Openstacks The largest number of products in IPC-2011 is $p = 250$. New problems use $p \in [250..630; 20]$.

Visit-All The hardest IPC-2011 problem has grid size $g = 50$. New problems use $g \in [48..86; 2]$.

Woodworking The largest number of parts in IPC-2011 is $p = 23$. For new problems, $p \in [25..105; 4]$.

The new test set is designed to be a clear step up in difficulty from IPC-2011. Only planners that were able to solve most of the IPC-2011 instances for a given domain are likely to solve some of the new *IPC-2011-LARGE* instances.

3.5 Experiments

Experiments include tests on all IPC-2011 and IPC-2011-LARGE benchmarks. They were run on an 8 core 2.8 GHz machine with a time limit of 30 minutes and memory limit of 2 GB per problem. Results for planners which use randomization are averaged over 5 runs.

3.5.1 Planners and Evaluation

The experiments compare Arvand-LS with the best known version of seven other planners: LAMA-2011, FDSS-2, Probe, Arvand, LPG-td, FF, FD-Autotune-2 (FD-AT-2) and Roamer. LAMA-2011, FDSS-2 and Probe are the three top performers from IPC-2011. Arvand and LPG-td [16] are well-konwn local search planners. FF and FD-AT-2 both use Enforced Hill Climbing as the first try to solve the problem. Roamer is included for comparison with Random-Walk Assisted Best-First Search. LAMA-2011-FF is LAMA-2011 in FF-only mode, with the Landmark Heuristic disabled. Since Arvand and Arvand-LS use h^{FF} only, this version of LAMA-2011 helps to make a better comparison by removing the influence of the h^{LM} heuristic.

The IPC-2011 versions of Roamer and Arvand were affected by a bug in the PDDL-to-SAS+ translator used. Arvand also had a memory management problem. These issues have been fixed in the versions of the planners used in the current experiments. All planner codes can be found in the official IPC-2011 SVN repository[1].

The evaluation method is the same as in IPC-2011: the score for an unsolved problem is 0. For each solved problem, the planner receives a score between 0 and 1, scaled relative to the best plan that these planners find in the experiments.

3.5.2 Results on IPC-2011 Benchmarks

The coverage and quality results on IPC-2011 are shown in Tables 3.1 and 3.2. In addition, Figure 3.2 plots the number of problems solved as a function of the search time first for the first 5 minutes, and then for the full 30 minutes. Arvand-LS solves more problems quickly than the other tested planners. The overall performance of Arvand-LS shows a significant improvement compared to the baseline planner Arvand and exceeds that of several other top planners.

The coverage of both Arvand and Arvand-LS is weak in the domains of *Sokoban*, *Parking* and *Barman*. The main reason for the bad performance of random walks in these domains is that specific action sequences need to be discovered in order to

[1] svn://svn@pleiades.plg.inf.uc3m.es/ipc2011/

make progress. The probability of repeatedly discovering such sequences through random walks is very small. In contrast, best-first search planners work much better in these domains, since they store the states that they have already tried, and can discover such paths by a complete search. The local search in Arvand-LS improves upon Arvand's pure exploration: The average *Sokoban* coverage improves from 2.2 to 8.4, *Parking* improves from 3.8 to 8.6 and *Barman* from 0 to 9.6.

Another interesting data point shown is LAMA-2011-FF, the version of LAMA without the h^{LM} heuristic. It solves 39 fewer problems than LAMA-2011. *Visit-All* shows the biggest drop, from 20 to 4. This domain is also very hard for many other planners that don't use the LM heuristic, such as FD-AT-2 and FDSS-2. Interestingly, the random walks planners Arvand and Arvand-LS, which use only the FF heuristic, solve all problems in *Visit-All*. It is because h^{FF} has the multiple *Uninformative Heuristic Regions* (URHs) problem in *Visit-All*, where multiple small and large UHRs create a large "virtual" UHR over the open list. The multiple UHRs problem will be further discussed in Chapter 5.

While local search improves the coverage of random walk planners most of the time, it fails for *Nomystery* [42]. This is a truck-package transportation domain similar to *Transport* in IPC-2011. Its key feature is that trucks only have a limited amount of fuel. In such resource constrained planning domains, delete-relaxation heuristics are not informative, since they ignore resource consumption. Random walk-based methods work very well here [42]. In Arvand-LS, the local greedy best first search guided by the h^{FF} heuristic seems to suffer from similar problems as other GBFS planners, following paths of great heuristic improvement but running out of fuel in the process. While Arvand-LS is weaker than Arvand here because of its greedy best-first search, thanks to random walks, it still achieves the third best score among all tested planners.

Compared to Arvand, Arvand-LS shows a significant improvement on plan quality in *Elevator*, *Openstacks*, *Scanalyzer* and *Visit-All*. In these domains, the local search finds much shorter plans. In *Elevator*, the plan quality of Arvand-LS is even better than that achieved with systematic search planners, such as LAMA-2011.

Domains	Arvand-LS	Arvand	Roamer	Probe	LAMA-2011	LAMA-2011-FF	FD-AT-2	FDSS-2
barman(20)	9.6	0	16.8	**20**	**20**	17	4	14
elevators(20)	**20**	**20**	13.4	**20**	**20**	18	16	18
floortile(20)	1.2	1.8	1	4	5	4	**9**	6
nomystery(20)	14	**19**	10	7	11	8	16	13
openstacks(20)	**20**	**20**	**20**	12	**20**	17	19	14
parcprinter(20)	**20**	**20**	7	13	**20**	**20**	14	**20**
parking(20)	8.6	3.8	5	15	15	11	4	**19**
pegsol(20)	**20**	**20**	19	**20**	**20**	**20**	**20**	**20**
scanalyzer(20)	**20**	17.6	**20**	18	**20**	**20**	17	**20**
sokoban(20)	8.4	2.2	12.4	15	**18**	**18**	15	**18**
tidybot(20)	17	17.8	15	**18**	16	11	16	16
transport(20)	**18**	14	16	13	14	12	9	13
visitall(20)	**20**	**20**	18	18	**20**	4	4	6
woodworking(20)	**20**	**20**	18.6	**20**	**20**	**20**	13	**20**
Total(280)	216.8	196.2	192.2	213	**239**	200	176	217

Table 3.1: Number of tasks solved in IPC-2011. Total number of tasks shown in parentheses after each domain name.

Domains	Arvand-LS	Arvand	Roamer	Probe	LAMA-2011	LAMA-2011-FF	FD-AT-2	FDSS-2
barman(20)	9.56	0.00	14.22	**17.98**	16.78	14.12	2.55	13.15
elevators(20)	**18.79**	7.63	9.57	7.57	9.20	8.87	13.99	11.76
floortile(20)	1.20	1.80	0.89	2.13	4.22	3.42	**8.96**	5.21
nomystery(20)	13.16	**18.11**	9.65	6.79	10.84	7.89	15.61	12.28
openstacks(20)	15.25	9.78	17.20	10.61	**18.59**	15.93	18.08	11.94
parcprinter(20)	18.56	17.61	6.05	11.29	**19.82**	18.34	13.64	18.24
parking(20)	6.63	2.85	3.04	7.32	12.82	6,79	3.84	**17.14**
pegsol(20)	**19.96**	19.91	17.69	18.41	19.90	19.23	19.78	14.46
scanalyzer(20)	**18.94**	15.41	16.63	14.83	17.60	16.59	15.14	17.81
sokoban(20)	7.89	2.17	12.15	11.83	**16.64**	17.61	14.92	16.08
tidybot (20)	15.01	15.85	12.95	**16.40**	14.04	9.63	13.78	13.19
transport(20)	**16.26**	10.54	13.36	6.76	11.02	7.10	7.51	7.67
visitall (20)	11.51	7.59	17.43	17.57	**18.07**	1.40	3.35	1.73
woodworking(20)	16.36	16.09	11.05	17.87	15.26	15.28	9.71	**19.14**
Total(280)	189.08	145.34	161.86	167.36	**204.78**	162.21	160.87	179.80

Table 3.2: Score in IPC-2011. The maximum possible score is shown in parentheses after each domain name.

Domains	Arvand-LS	Arvand	Roamer	Probe	LAMA-2011	FD-AT-2	FDSS-2	FF	LPG
large-elevator(20)	**20.0**	18.8	1.2	4.0	4.0	1.0	2.0	3.0	0.4
large-openstacks (20)	16.0	**17.6**	10.0	0.0	8.0	0.0	0.0	1.0	0.0
large-visitall(20)	**20.0**	12.6	0.0	0.0	13.0	0.0	0.0	0.0	0.0
large-woodworking(20)	**18.0**	10.4	5	4.0	11.0	3.0	11.0	3.0	8.6
total(80)	**74.0**	59.4	16.2	8.0	36.0	4.0	13.0	7.0	9.0

Table 3.3: Number of tasks solved in IPC-2011-LARGE. Total number of tasks shown in parentheses after each domain name.

3.5.3 Results on *IPC-2011-LARGE* Problems

Table 3.3 shows the coverage of all tested planners on the four scaled domains. The key observations are:

- Arvand-LS significantly outperforms the other tested planners in all domains except *Large-Openstacks*, where Arvand has a slight advantage. Arvand-LS solves twice as many problems as LAMA-2011, the winner of IPC-2011. Many other planners have problems scaling up to these large instances.
- Both RW-LS and MRW scale better than EHC implemented both in FF and FD-AT-2.
- Relying on random walks to escape from the plateaus within a global best-first search framework is not enough to scale to these large problems: both Arvand and Arvand-LS solve between 5 to 20 problems more than Roamer in each of the domains.
- In three out of four domains, the local search in Arvand-LS increases coverage compared to Arvand.

Figure 3.3 shows IPC-style scores for each domain. To keep the figures readable, only the results of the four planners with highest score are shown in each domain [2]. The results show:

- Arvand-LS always generates better quality plans than Arvand. In a few cases, such as problems 17 to 20 in *Large-Openstacks*, Arvand-LS gets lower scores since it is not able to consistently solve the problem in all 5 runs.
- Except for *Large-Visit-All*, Arvand-LS creates plans of better (*Large-Elevators*) or competitive quality compared to more systematic planners such as LAMA-2011.

[2] In *Large-Visit-All*, only three planners solve any problem.

Focusing on the *Large-Elevators* results in Figure 3.3(a), the difference in plan quality of Arvand-LS and Arvand decreases with problem size. There are two reasons for this: for problems 10 to 20, solutions are found only by the *config-1* configuration using long random walks, so the generated solutions consist mostly of random actions. Furthermore, these solutions are so long that Aras, which grows a neighborhood graph along the solutions, has memory problems. The best solution for problem 1 is found by FD-Autotune-2. However, this planner did not scale to larger instances.

In *Large-Openstacks*, Figure 3.3(b), both LAMA-2011 and Arvand-LS generate good quality solutions initially, but Arvand-LS scales better. Arvand-LS finds better quality solutions than Arvand until problem 16, and then Arvand takes over for larger instances. Beyond problem 16, Arvand-LS does not solve the problems in all 5 trial runs.

For smaller *Large-Visit-All* instances, both Arvand and Arvand-LS generate lower quality plans than LAMA. Only Arvand-LS scales beyond problem 13. To avoid making $bubbles$ - unvisited squares surrounded by visited squares - is a good strategy to generate good quality plans in this domain. Random walks visit unvisited squares quickly in the beginning, but create a lot of $bubbles$ on the way, leading to bad plan quality.

The *Large-Woodworking* instances, Figure 3.3(d), seem relatively easier since more planners can solve problems here. The plan quality on the first 10 problems is very close for the tested planners. Only Arvand-LS scales beyond problem 13. The low score for problem 17 is due to 3 out of 5 trial runs of Arvand-LS failing.

To summarize, Arvand-LS is the strongest planner overall in these tests in terms of both coverage and quality. In some problems with moderate plan length, Arvand has better performance than Arvand-LS. Because of the post-processing system Aras, for small problems, the diversity of input plans for Aras is more important than the quality of initial plans. Arvand produces more, and also more diverse, plans than Arvand-LS because of its faster pure exploration strategy. For larger problems, the very long plans generated by Arvand often deviate too far from good plans, and the local optimization of Aras is not sufficient to substantially improve these.

3.6 Conclusions

This chapter introduces the new algorithm RW-LS, which uses a local Greedy Best-First Search driven by both direct node evaluation and Random Walks. The planner Arvand-LS improves both coverage and quality significantly over the IPC-2011 version of Arvand. In domains which contain sufficiently many paths from the initial state to a goal, the algorithm scales better than other tested planners.

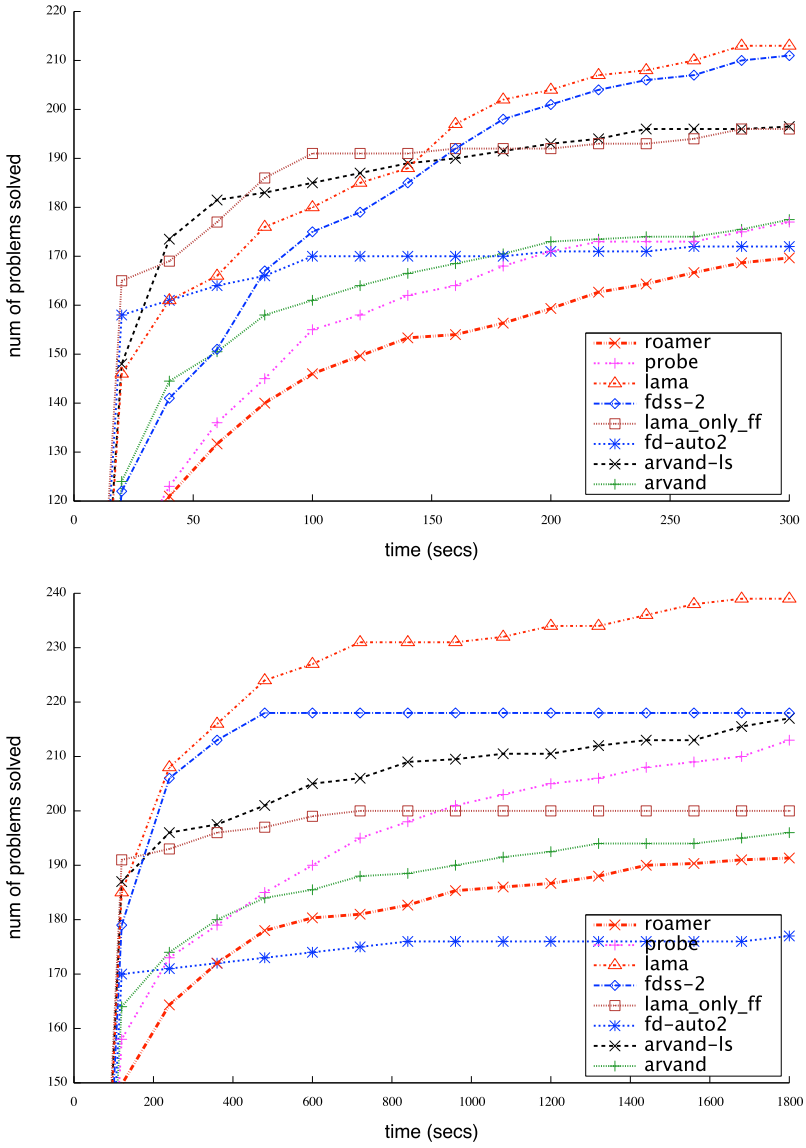
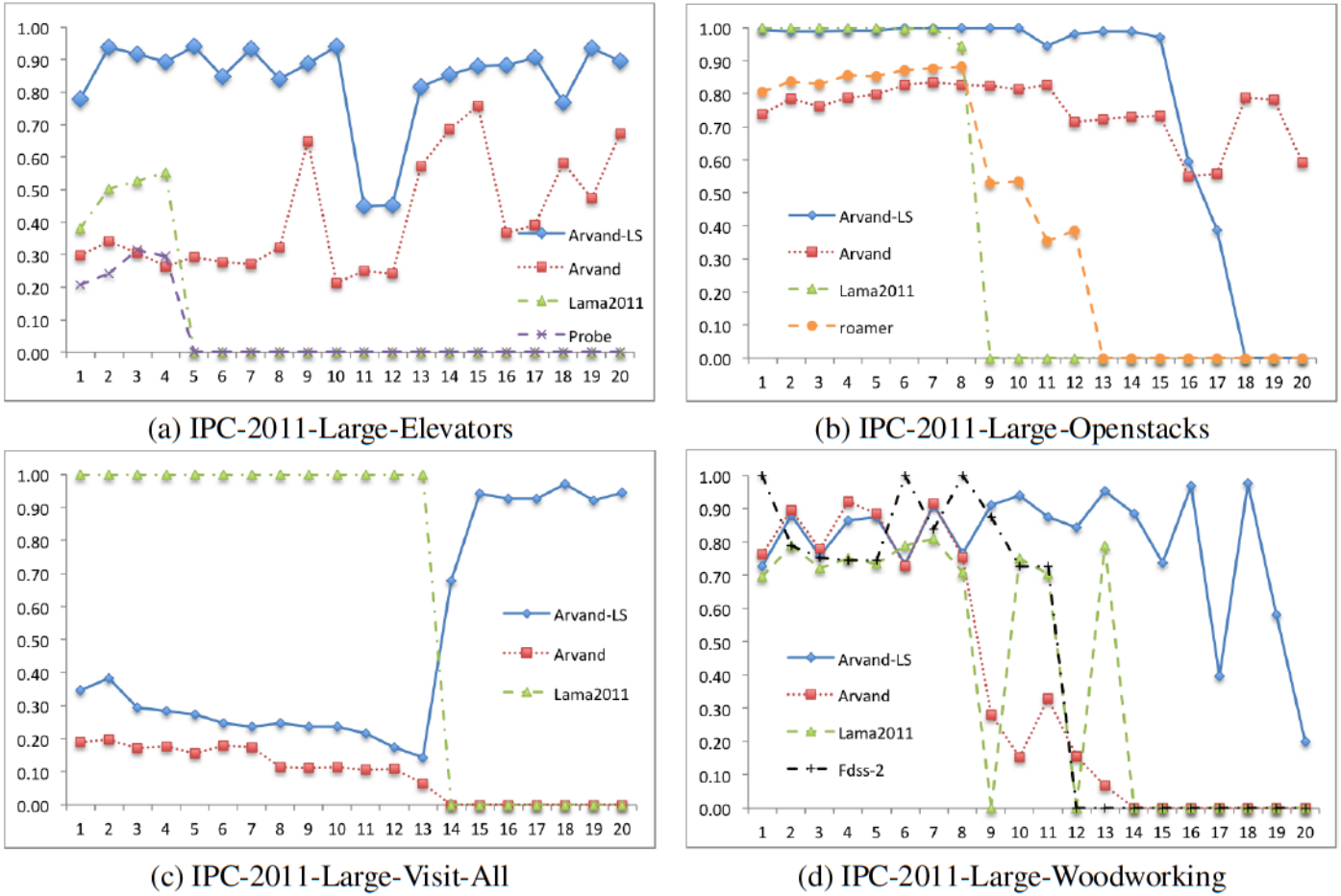


Figure 3.2: The number of problems solved in the first 5 minutes and 30 minutes. The x-axes represent the running time, and the y-axes represent the number of problems solved in the given time.



(a) IPC-2011-Large-Elevators

(b) IPC-2011-Large-Openstacks

(c) IPC-2011-Large-Visit-All

(d) IPC-2011-Large-Woodworking

Figure 3.3: Scores of top four planners on IPC-2011-LARGE. The x-axis presents the problem numbers. The size of problems grows from problem 1 to problem 20.

Chapter 4

Adding Local Exploration to Greedy Best First Search

Greedy Best-First Search (GBFS) is a powerful algorithm at the heart of many state of the art satisficing planners. One major weakness of GBFS is its behavior in so-called uninformative heuristic regions (UHRs) - parts of the search space in which its heuristic provides no guidance towards states with improved heuristic values. To address this, one can try to develop better heuristics. An alternative approach is to modify the search algorithm in order to make the algorithm less sensitive to UHRs. This chapter proposes a framework based on GBFS, named GBFS with local exploration, which automatically switches from the global GBFS into local exploration mode once the search algorithm gets stuck in UHRs. This chapter is based on the following publication:

- *F. Xie, M. Müller and R. 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), 2388-2394, 2014* [68].

4.1 Introduction

Uninformative heuristic regions (UHRs) includes *local minima* and *plateaus*. A *local minimum* is a state with minimum h-value within a local region which is not a global minimum. A *plateau* is an area of the state space where all states have

the same heuristic value. GBFS, because of its open list, can get stuck in multiple UHRs at the same time.

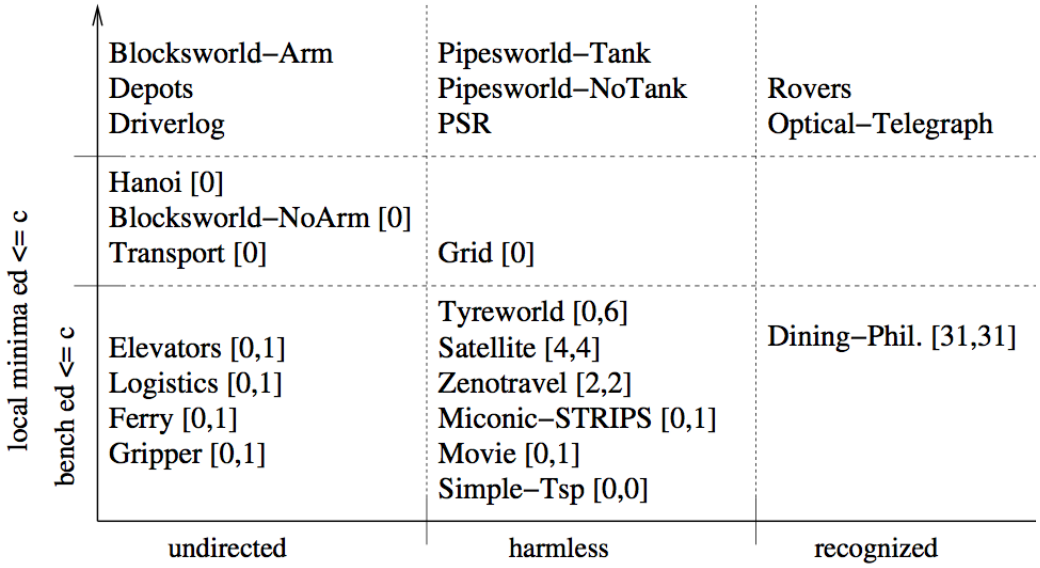


Figure 4.1: Overview of h^+ topology, from Hoffmann [24]. Domains with unrecognized dead ends are not shown. Assume we denote the state space by a graph $G(V, E)$, and $gd(s)$ denotes the goal distance of node s. On the x-axis, ***undirected*** means that all actions can be undone, namely for any node s and s', both (s, s') and $(s', s) \in E$; ***harmless*** means that no action that can not be undone leads to a dead end, namely if there exists $(s, s') \in E$ such that $(s', s) \notin E$, and, for all $s \in V$, $gd(s) < \infty$; ***recognized*** means that all dead ends are recognizable by $h^+(s)$, namely if there exists $s \in V$ such that $gd(s) = \infty$, and, for all $s \in V$, if $gd(s) = \infty$ then $h^+(s) = \infty$. The y-axis represents the existence or non-existence of constant upper bounds on the ***exit distance*** [23] of local minima and benches (called plateau in this dissertation).

Hoffmann has studied the problem of UHRs for the case of the optimal relaxation heuristic h^+ [24, 23]. He classified a large number of planning benchmarks, shown in Figure 4.1, according to their *maximum exit distance* from plateaus and local minima, and by whether dead ends exist and are recognized by h^+. This chapter proposes local exploration to improve GBFS. The focus of the analysis is on domains with a large or even unbounded maximum exit distance for plateaus and local minima, but without unrecognized dead ends. In these domains, there exists a plan from each state in an UHR (with $h^+ < \infty$).

As an example, the IPC domain 2004-notankage has no dead ends, but contains

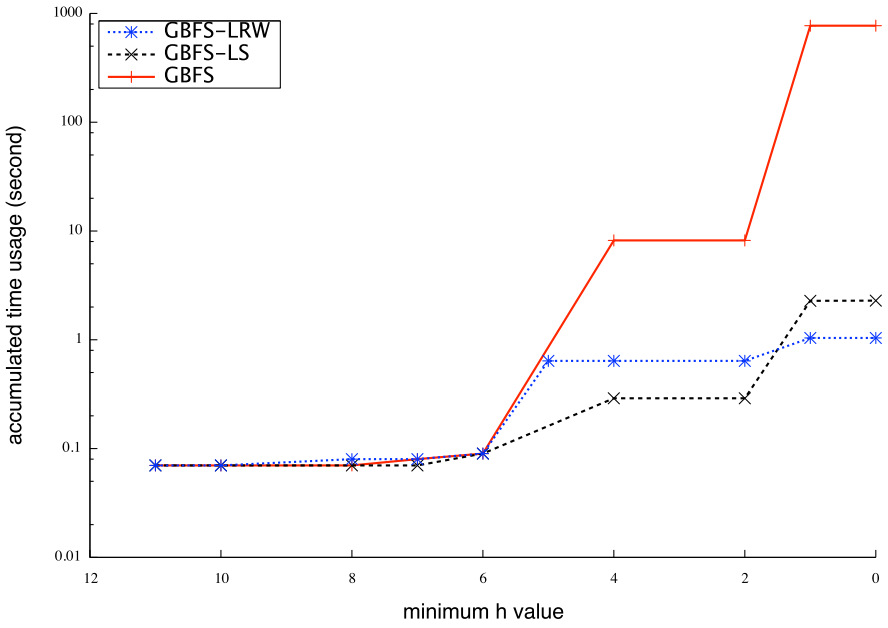


Figure 4.2: Cumulative search time (in seconds) of GBFS, GBFS-LS and GBFS-LRW with h^{FF} for first reaching a given h_{min} in 2004-notankage #21.

unbounded plateaus and local minima [24]. Instance #21 serves to illustrate a case of bad search behavior in GBFS due to UHRs. For this instance, Figure 4.2 plots the current minimum heuristic value h_{min} in the closed list on the x-axis against the log-scale cumulative search time needed to first reach h_{min}. The solid line is for GBFS with h^{FF}. The two huge increases in search time, with the largest (763 seconds) for the step from $h_{min} = 2$ to $h_{min} = 1$, correspond to times when the search is stalled in multiple UHRs. Since the large majority of overall search time is used to inefficiently find an escape from UHRs, it seems natural to try switching to a secondary search strategy which is better at escaping. Such ideas have been tried several times before. This related work is reviewed and compared in the next section.

This chapter introduces a framework which adds a local search algorithm to GBFS in order to improve its behavior in UHRs. Two such algorithms, *local GBFS (LS(n))* and *local random walks (LRW(n))*, are designed to find quicker escapes from UHRs, starting from a node n within an UHR. The main contributions of this work are:

- An analysis of the problem of UHRs in GBFS, and its consequences for limiting the performance of GBFS in current benchmark problems in satisficing planning.

- A new search framework, Greedy Best-First Search with Local Exploration (GBFS-LE), which runs a separate local search whenever the main global GBFS seems to be stuck. Two concrete local search algorithms, local GBFS (*LS*) and local random walks (*LRW*), are shown to be less sensitive to UHRs than GBFS and when incorporated into GBFS are shown to outperform the baseline by a substantial margin over the IPC benchmarks.

- An analysis of the connection between Hoffmann's theoretical results on local search topology [24, 23] and the performance of adding local exploration into GBFS.

The remainder of this chapter is organized as follows: after a review of previous work on strategies for escaping from UHRs, the new search framework GBFS-LE is introduced, compared with related work, and evaluated experimentally on IPC domains.

4.2 Search Strategies for Escaping UHRs

There are several approaches to attack the UHR problem. Better quality heuristics [18, 21, 25] can shrink the size of UHRs, as can combining several heuristics [46, 51]. Additional knowledge from heuristic computation or from problem structure can be utilized in order to escape from UHRs. Examples are helpful actions [25] and explorative probes [36]. The third popular approach is to develop search algorithms that are less sensitive to flaws in heuristics. Algorithms which add a global exploration component to the search, which is especially important for escaping from unrecognized *dead ends*, include restarting [10, 38, 49] and non-greedy node expansion [26, 60, 70]. This chapter focuses on another direction: adding a local exploration component to the globally greedy GBFS algorithm.

The planner *Marvin* adds machine-learned plateau-escaping macro-actions to

enforced hill-climbing [9]. *YAHSP* constructs macro actions from *FF*'s relaxed planning graph [62]. *Identidem* adds exploration by expanding a sequence of actions chosen probabilistically, and proposes a framework for escaping from local minima in local-search forward-chaining planning [10]. *Arvand* [38] uses random walks to explore quickly and deeply. *Arvand-LS* in Chapter 3 combines random walks with local greedy best-first search, while *Roamer* [37] adds exploration to *LAMA-2008* by using fixed-length random walks. Analysis from Nakhost and Müller [40] shows that while random walks outperform GBFS in escaping many kinds of plateaus, but they fail badly in domains such as Sokoban, where a precise action sequence must be found to escape. In *Real-Time Heuristic Search* [33], the idea of *Heuristic Depression* [27] is very similar to UHR. Intuitively, a heuristic depression is a bounded region of the search space containing states whose heuristic value is much lower compared to states in the border of the depression. LSS-LRTA* [32] uses extensive look-ahead and heuristic correction to handle heuristic depressions. Although extensive look-ahead and heuristic correction might not be applied directly in satisficing planning because of the more expensive to compute heuristic computing and much large state spaces, the ideas are very similar to what we have seen in these local search based planning algorithms. However, while escaping from UHRs has been well studied in the context of these local search based planners, there is comparatively little research on how to use search for escaping UHRs in the context of GBFS. This chapter begins to fill this gap.

Algorithm 7 GBFS-LE

Input Initial state I, goal states G
Parameter STALL_SIZE, MAX_LOCAL_TRY
Output A solution plan

```
 1: (open, h_min) ← ([I], h(I))
 2: stalled ← 0; nuLocalTry ← 0
 3: while open ≠ ∅ do
 4:    n ← open.remove_min()
 5:    if n ∈ G then
 6:       return plan from I to n
 7:    end if
 8:    closed.insert(n)
 9:    for each v ∈ successors(n) do
10:       if v ∉ closed then
11:          open.insert(v, h(v))
12:          if h_min > h(v) then
13:             h_min ← h(v)
14:             stalled ← 0; nuLocalTry ← 0
15:          else
16:             stalled ← stalled + 1
17:          end if
18:       end if
19:    end for
20:    if stalled = STALL_SIZE
       and nuLocalTry < MAX_LOCAL_TRY then
21:       n ← open.remove_min()
22:       LocalExplore(n){local GBFS or random walks}
23:       stalled ← 0; nuLocalTry ← nuLocalTry + 1
24:    end if
25: end while
```

4.3 GBFS-LE: GBFS with Local Exploration

The new technique of *Greedy Best-First Search with Local Exploration (GBFS-LE)* uses local exploration whenever a global GBFS (G-GBFS) seems stuck. If G-GBFS fails to improve its minimum heuristic value h_{min} for a fixed number of node expansions, then GBFS-LE runs a small local search for exploration, *LocalExplore(n)*, from the best node n in a global-level open list. Algorithm 7 shows GBFS-LE. *STALL_SIZE* and *MAX_LOCAL_TRY*, used in Line 22, are parameters which control the tradeoff between global search and local exploration.

Algorithm 8 $LS(n)$, *local GBFS*

Input state n, initial state I, goal states G, h_{min} {global variable}, $open$, $closed$
Parameter LSSIZE

```
1:  local_open ← [n]
2:  h_improved ← false
3:  for i = 1 to LSSIZE do
4:     if local_open = ∅ then
5:        return
6:     end if
7:     n ← local_open.remove_min() {FIFO tie-breaking}
8:     if n ∈ G then
9:        return plan from I to n
10:    end if
11:    closed.insert(n)
12:    for each v ∈ successors(n) do
13:       if v ∉ closed then
14:          if h(v) < ∞ then
15:             local_open.insert(v, h(v))
16:             if h_min > h(v) then
17:                h_min ← h(v)
18:                h_improved ← true
19:             end if
20:          end if
21:       end if
22:    end for
23:    if h_improved then
24:       break
25:    end if
26: end for
27: merge(open,local_open)
28: return
```

The main change from GBFS is the call to *LocalExplore(n)* at Line 24 whenever there has been no improvement in heuristic value over the last *STALL_SIZE* node expansions.

Two local exploration strategies were tested. The first is local GBFS search starting from node n, *LocalExplore(n)* = *LS(n)*, which shares the closed list of G-GBFS, but maintains its own separate open list *local_open* that is cleared before each local search. $LS(n)$, as shown in Algorithm 8, succeeds if it finds a node v with $h(v) < h_{min}$ at Line 16 before it exceeds the *LSSIZE* limit. In any case,

Algorithm 9 $LRW(n)$, *local random walk*

Input state n, goal states G, h_{min} {global variable}, $open$
Parameter LSSIZE

```
1: for i = 1 to LSSIZE do
2:    s ← n
3:    for j = 1 to LENGTH_WALK do
4:       A ← ApplicableActions(s)
5:       if A = ∅ then
6:          break
7:       end if
8:       a ← SelectAnActionFrom(A)
9:       s ← apply(s, a)
10:      if s ∈ G then
11:         open.insert(s, h(s))
12:         return
13:      end if
14:   end for
15:   if h(s) < h_min then
16:      open.insert(s, h(s))
17:      break
18:   end if
19: end for
20: return
```

the remaining nodes in *local_open* are merged into the global open list. A local search tree grown from a single node n is much more focused and grows deep much more quickly than the global open list in G-GBFS. It also restricts the search to a single plateau, while G-GBFS can get stuck when exploring many separate plateaus simultaneously, which will be further discussed in Chapter 5. Both G-GBFS and *LS(n)* use a first-in-first-out tie-breaking rule.

The second local exploration strategy tested is local random walk search, *LocalExplore(n) = LRW(n)*, as shown in Algorithm 9. The implementation of random walks from the Arvand planner [38, 41] is used. $LRW(n)$ runs up to a pre-set number of random walks starting from node n, and evaluates only the endpoint of each walk using h^{FF}. All intermediate states are checked for whether they are goal states. Like $LS(n)$, $LRW(n)$ succeeds if it finds a node v with $h(v) < h_{min}$ within its exploration limit at Line 15. In this case, v is added to the global open list, and the path from n to v is stored for future plan extraction. In case of failure, unlike *LS(n)*,

no information is kept.

Random walks are controlled by a tuple of parameters:

- $(len\_walk, e\_rate, e\_period, walk\_type)$

Random walk length scaling is controlled by an initial walk length of len_walk, an extension rate of e_rate and an extension period of $NUMWALKS * e\_period$. The algorithm performs random walks with length cur_len_walk, which initially equals len_walk. If h_{min} does not improves over $NUMWALKS * e\_period$ random walks, cur_len_walk will be updated to $cur\_len\_walk * (1 + e\_rate)$. $walk\_type$ decides how actions are selected in Line 8. The choices are pure random (***PURE***) and Monte Carlo Helpful Actions (***MHA***), which bias random walks to ***Helpful Actions*** [25]. For example, in configuration $(1, 2, 0.1, MHA)$ all random walks use the MHA walk type, and if h_{min} does not improve for $NUMWALKS * 0.1$ random walks, then the length of walks, len_walk, which starts at 1, will be doubled. This is very different from Roamer, which uses fixed length random walks. LRW was tested with the following two configurations: $(1, 2, 0.1, MHA)$, which is used with preferred operators, and $(1, 2, 0.1, PURE)$.

The example of Figure 4.2 is solved much faster, in around 1 second, by both GBFS-LS and GBFS-LRW, while GBFS needs 771 seconds. The three algorithms built exactly the same search trees when they first achieved the minimum h-value 6. The local GBFS in GBFS-LS, focusing on one branch, found a 5 step path that decreases the minimum h-value using only 10 expansions. The ***h***-values along the path were 6, 7, 7, 6 and 4, showing an initial increase before decreasing. h-values along GBFS-LRW's path also increased before decreasing. In contrast, GBFS gets stuck in multiple separate h-plateaus since it needs to expand over 10000 nodes with h-value 6, which were distributed in many different parts of the search tree. Only after exhausting these, it expands the first node with $h = 7$. This is called the ***Multiple Uninformative Heuristic Region*** problem, which will be further discussed in Chapter 5. In this example, the local explorations, which expand or visit higher h-value nodes earlier, massively speed up the escape from UHRs.

There are several major differences between GBFS-LS and GBFS-LRW. GBFS-LS keeps all the information gathered during local searches by copying its nodes

into the global open list at the end. GBFS-LRW keeps only endpoints that improve h_{min} and the paths leading to them. This causes a difference in how often the local search should be called. For GBFS-LS, it is generally safe to do more local search, while over-use of local search in GBFS-LRW can waste search effort[1]. This suggests using more conservative settings for the parameters *MAX_LOCAL_TRY* and *LSSIZE* in *LRW(n)*. The two algorithms also explore UHRs very differently. *LS(n)* systematically searches the subtree of n, while *LRW(n)* samples paths leading from n sparsely but deeply.

4.4 Experimental Results

Experiments were run on a set of 2112 problems in 54 domains from the first seven International Planning Competitions which are publicly available[2], using one core of a 2.8 GHz machine with 4 GB memory and 30 minutes per instance. Results for planners which use randomization are averaged over five runs. All planners are implemented on the Fast Downward code base FD-2011 [19]. The translation from PDDL to SAS+ was done only once, and this common preprocessing time is not counted in the 30 minutes. Parameters were set as follows: *STALL_SIZE = 1000* for both algorithms. (*MAX_LOCAL_TRY*, *LSSIZE*) = (100, 1000) for GBFS-LS and (10, 100) for GBFS-LRW.

4.4.1 Local Search Topology for h^+

For the purpose of experiments on UHRs, the detailed classification by h^+ of Figure 4.1 can be coarsened into three broad categories:

- *Unrecognized-Deadend*: 195 problems from 4 domains with unrecognized dead ends: Mystery, Mprime, Freecell and Airport.

- *Large-UHR*: 383 problems from domains with UHRs which are large or of unbounded exit distance, but with recognized dead ends: column 3 in Figure

[1] Each step in a random walk generates all children and randomly picks one, which is only slightly cheaper than one expansion by LS when Deferred Evaluation is applied.

[2] The current IPC test set does not include Blocksworld, Hanoi, Ferry and Simple-Tsp from Figure 4.1.

4.1, plus the top two rows of columns 1 and 2.

- *Small-UHR*: 669 problems from domains without UHRs, or with only small UHRs, corresponding to columns 1 and 2 in the bottom row of Figure 4.1.

The problems from these three categories are only a subset of the total 2112 problems, since not all the 54 domains were analyzed by Hoffmann [24].

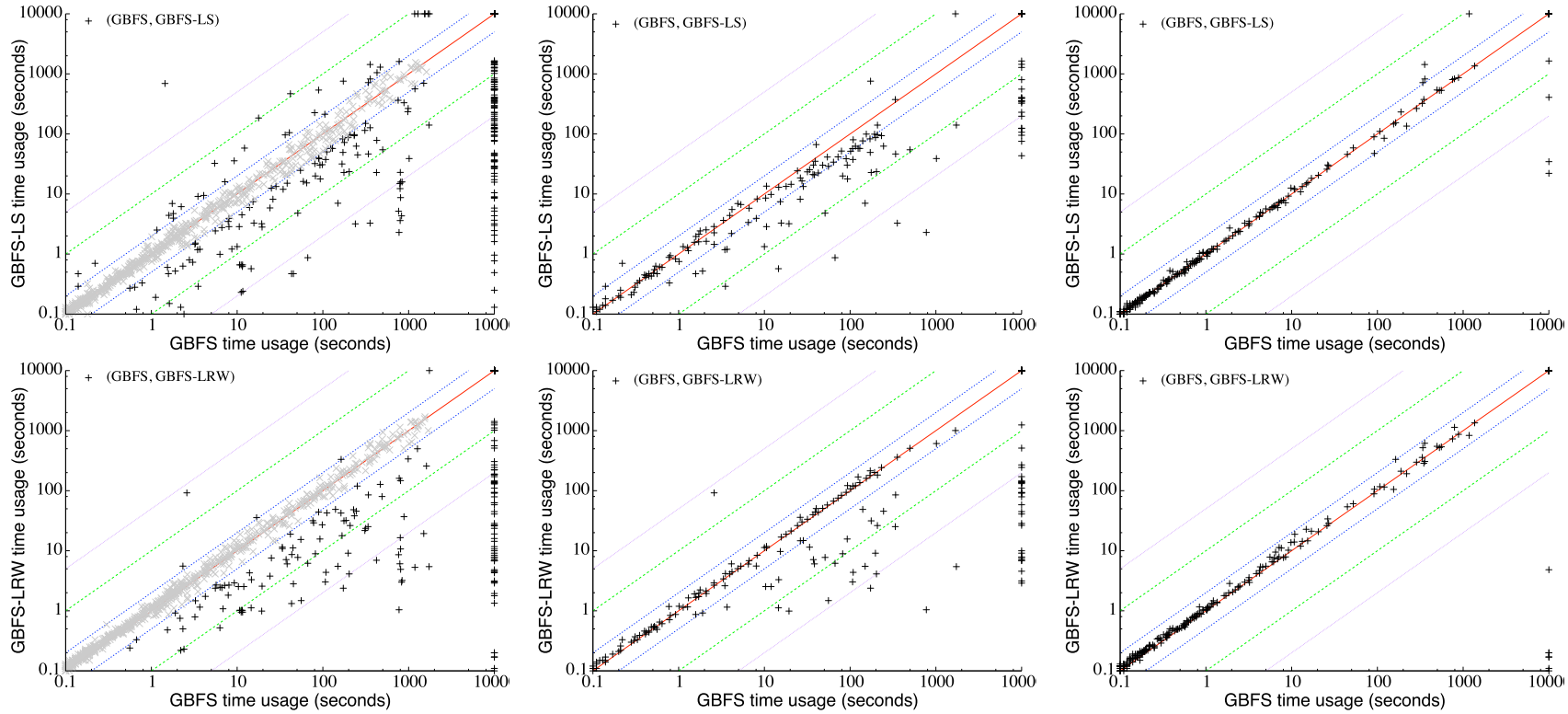
4.4.2 Performance of Baseline Algorithms

Heuristic	GBFS	GBFS-LS	GBFS-LRW
FF	1561	**1641**	1619.4
CG	1513	**1608**	1573.2
CEA	1498	1592	**1615.2**

Table 4.1: IPC coverage out of 2112 for GBFS with and without local exploration, with three standard heuristics: FF, CG and CEA.

The baseline study evaluates GBFS, GBFS-LS and GBFS-LRW without the common planning enhancements of preferred operators, deferred evaluation and multi-heuristics. Three widely used planning heuristics are tested: FF [25], causal graph (CG) [18] and context-enhanced additive (CEA) [21]. Table 4.1 shows the coverage on all 2112 IPC instances. Both GBFS-LS and GBFS-LRW outperform GBFS by a substantial margin for all 3 heuristics.

Figure 4.3(a) compares the time usage of the two proposed algorithms with GBFS using h^{FF} over the IPC benchmarks. Every point in the figure represents one instance, plotting the search time for GBFS on the x-axis against GBFS-LS (top) and GBFS-LRW (bottom) on the y-axis. Only problems for which both algorithms need at least 0.1 seconds are shown. Points below the main diagonal represent instances that the new algorithms solve faster than GBFS. For ease of comparison, additional reference lines indicate $2\times$, $10\times$ and $50\times$ relative speed. Data points within a factor of 2 are shown in grey in order to highlight the instances with substantial differences. Problems that were only solved by one algorithm within the 1800 second time limit are included at $x = 10000$ or $y = 10000$. The above plot setting, which is referred as ***IPC time comparison setting***, is used frequently in this



(a) All IPC (b) Large-UHR (c) Small-UHR

Figure 4.3: Comparison of time usage of the three baseline algorithms. 10000 corresponds to runs that timed out or ran out of memory. Results shown for one typical run of GBFS-LRW, which is selected by comparing all 5 runs. All runs are very similar.

dissertation. Both new algorithms show substantial improvements in search time over GBFS.

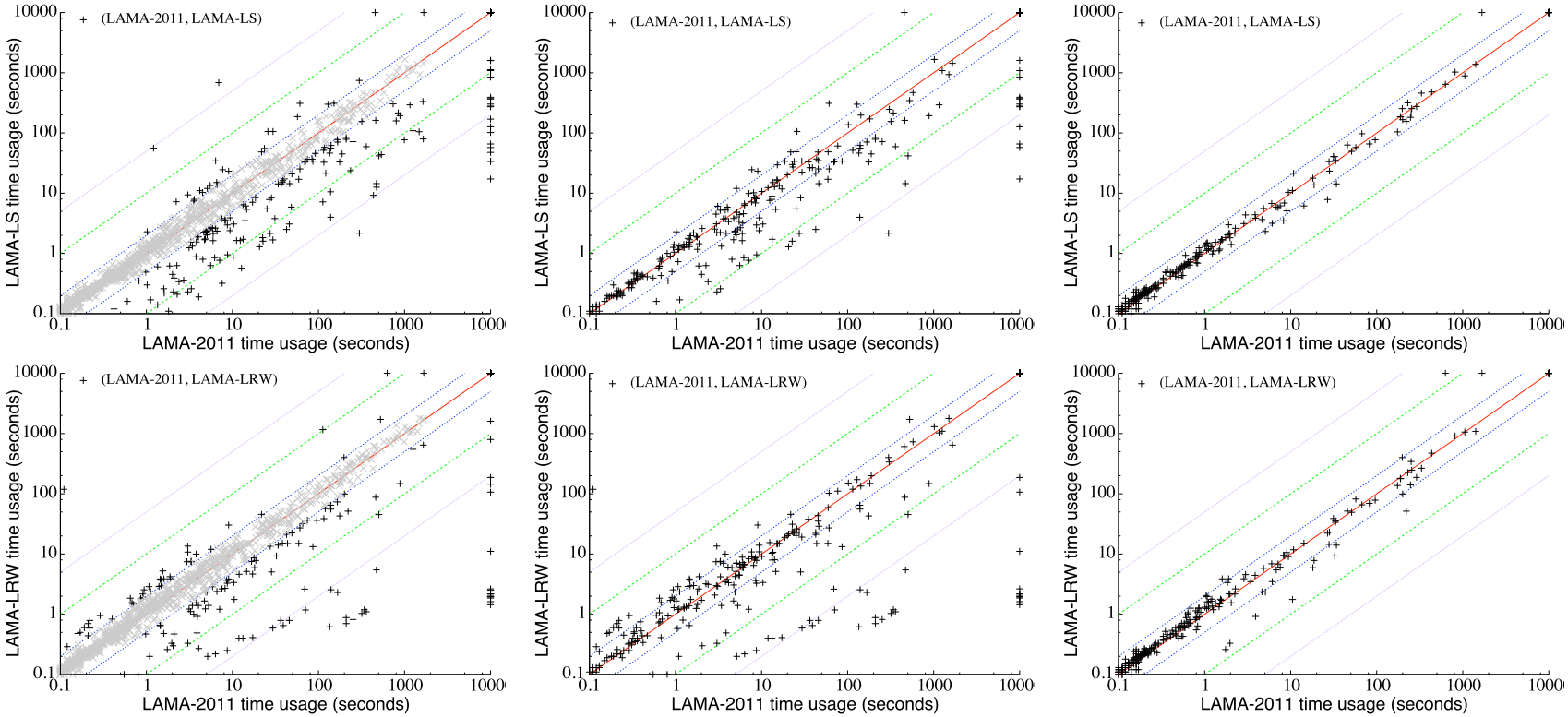
Figures 4.3(b) and (c) restrict the comparison to Large-UHR and Small-UHR domains, respectively. In Large-UHR domains, GBFS-LS and GBFS-LRW solve 19 (+9.7%) and 30 (+15.3%) more problems than GBFS (195/383) respectively. Both outperform GBFS in search time. However, in Small-UHR domains, GBFS-LS and GBFS-LRW only solve 3 (+0.5%) and 7 (+1.1%) more problems than GBFS (634/669), and there is very little difference in search time among the three algorithms. This result clearly illustrates the relationship between the size of UHRs and the performance of the two local exploration techniques. For Unrecognized-Deadend, GBFS-LS is slightly slower than GBFS with the same coverage (162/195), while GBFS-LRW is slightly faster and solves 7 (+3.7%) more problems. The effect of local exploration on the performance in the case of unrecognized dead-ends is not clear-cut.

4.4.3 Performance with Search Enhancements

Experiments in this section test the two proposed algorithms when three common planning enhancements are added: Deferred Evaluation, Preferred Operators and Multiple Heuristics. h^{FF} is used as the primary heuristic in all cases.

- *Deferred Evaluation* delays state evaluation and uses the parent's heuristic value in the priority queue [45]. This technique is used in G-GBFS and *LS(n)*, but not in the endpoint-only evaluation of random walks in *LRW(n)*.

- The *Preferred Operators* enhancement keeps states reached via a preferred operator, such as helpful actions in h^{FF}, in an additional open list [45]. An extra preferred open list is also added to $LS(n)$. Boosting with the default parameter value (1000) is used, and Preferred Operator first ordering is used for tie-breaking as in LAMA-2011 [46]. In $LRW(n)$, preferred operators are used in form of the *Monte Carlo with Helpful Actions* (MHA) technique [38], which biases random walks towards using operators which are often preferred.



(a) All IPC (b) Large-UHR (c) Small-UHR

Figure 4.4: Comparison of search time of LAMA-2011 with LAMA-LS and LAMA-LRW. A typical single run is used for LAMA-LRW.

- The *Multi-Heuristics* approach maintains additional open lists in which states are evaluated by other heuristic functions. Because of its proven strong performance in LAMA, the *Landmark count* heuristic h^{lm} [48] is used as the second heuristic. Both G-GBFS and *LS(n)* use a round robin strategy for picking the next node to expand. In Fast Downward, h^{lm} is calculated incrementally from the parent node. When Multi-Heuristics is applied to GBFS-LRW, the $LRW(n)$ part still uses h^{FF} because the path-dependent landmark computation was not implemented for random walks. When $LRW(n)$ finds a heuristically improved state s, GBFS-LRW evaluates and expands all states along the path to s in order to allow the path-dependent computation of $h^{lm}(s)$ in G-GBFS. Without Multi-Heuristics, only s itself is inserted to the open list.

Table 4.2 shows the experimental results on all IPC domains. Used as a single enhancement, Preferred Operators improves all three algorithms. Deferred Evaluation improves GBFS-LS and GBFS-LRW, but fails for GBFS, mainly due to plateaus caused by the less informative node evaluation [45]. In GBFS-LS and GBFS-LRW, the benefit of faster search outweighs the weaker evaluation. Multi-Heuristics strongly improves GBFS and GBFS-LS, but is only a modest success in GBFS-LRW. This is not surprising since *LRW(n)* does not use h^{lm}, and in order to evaluate the new best states generated by *LRW(n)* with h^{lm} in G-GBFS, all nodes on the random walk path need to be evaluated, which degrades performance.

Enhancement	GBFS	GBFS-LS	GBFS-LRW
(none)	1561	1641	1619.4
PO	1826	1851	1827.4
DE	1535	1721	1635
MH	1851	1874	1688.4
PO + DE	1871	1889	1880.6
PO + MH	1850	1874	1854.2
DE + MH	1660	1764	1730.2
PO + DE + MH	**1913**	**1931**	**1925.4**

Table 4.2: Number of instances solved with search enhancements, out of 2112. PO = Preferred Operators, DE = Deferred Evaluation, MH = Multi-Heuristic.

4.4.4 Comparing with LAMA-2011 in terms of Coverage and Search Time

The final row in Table 4.2 shows coverage results when all three enhancements are applied. The performance comparisons in this section use this best known configuration in terms of coverage for three algorithms based on GBFS, GBFS-LS and GBFS-LRW, which closely correspond to the "coverage-only" first phase of the LAMA-2011 planner:

- **LAMA-2011:** only the first GBFS iteration of LAMA is run, with deferred evaluation, preferred operators and multi-heuristics with h^{FF} and h^{lm} [46].
- **LAMA-LS:** Configured like LAMA-2011, but with GBFS replaced by GBFS-LS.
- **LAMA-LRW:** GBFS in LAMA-2011 is replaced by GBFS-LRW.

Table 4.3 shows the coverage results per domain. LAMA-LS has the best overall coverage, 18 more than LAMA-2011, closely followed by LAMA-LRW. LAMA-LS solves more problems in 7 of the 10 domains where LAMA and LAMA-LS differ in coverage. This number for LAMA-LRW is 7 out of 11. Although LAMA-LRW uses a randomized algorithm, our 5 runs for LAMA-LRW had quite stable results: 1927, 1924, 1926, 1924 and 1926. By comparison, adding the landmark count heuristic, which differentiates LAMA-2011 from other planners based on the Fast Downward code base, improves the coverage of LAMA-2011 by 42, from 1871 to 1913.

Using the same format as Figure 4.3 for baseline GBFS, Figure 4.4 compares the search time of the three planners over the IPC benchmark. Both LAMA-LS and LAMA-LRW show a clear overall improvement over LAMA-2011 in terms of speed. The benefit of local exploration for search time in Large-UHR still holds even with all enhancements on. Both LAMA-LS and LAMA-LRW solve 12 more problems (4.1%) than LAMA-2011's 290/383 in Large-UHR, while in Small-UHR they solve 1 and 2 fewer problems respectively than LAMA-2011's 646/669.

For further comparison, the coverage results of some other strong planners from IPC-2011 on the same hardware are: FDSS-2 solves 1912/2112, Arvand 1878.4/2112,

Domain	Size	LAMA-2011	LAMA-LS	LAMA-LRW
00-miconic-ful	150	**136**	**136**	135.6
02-depot	22	**20**	**20**	19.6
02-freecell	80	78	**79**	78 .2
04-airport-str	50	32	**34**	32.8
04-notankage	50	**44**	43	**44**
04-optical-tel	48	4	**6**	4
04-philosoph	48	39	47	**47.8**
04-satellite	36	**36**	35	35
06-storage	30	18	**23**	21
06-tankage	50	41	41	**42**
08-transport	30	29	**30**	**29.6**
11-floortile	20	**6**	5	**6**
11-parking	20	18	**20**	16.8
11-transport	20	16	16	**17**
Total	2112	1913	1931	1925.4
Unsolved		199	181	186.6

Table 4.3: Domains with different coverage for the three planners. 33 domains with 100% coverage and 7 further domains with identical coverage for all planners are not shown.

Lama-2008 1809/2112, fd-auto-tune-2 1747/2112, and Probe 1706/1968 (failed on the ":derive" keyword in 144 problems).

Although the local explorations are inclined to increase the solution length[3], the influence is not clear-cut since they also solve more problems. The IPC-2011 plan quality scores for LAMA-2011, LAMA-LS and LAMA-LRW are 1898.0, 1899.6 and 1900.5.

4.5 Chapter Summary

This chapter investigates how local exploration facilitates escaping from UHRs for greedy best-first search. The new framework of GBFS-LE, GBFS with Local Exploration, has been tested successfully in two different realizations, adding local greedy best-first search in GBFS-LS and random walks in GBFS-LRW.

[3]Local GBFS intends to search deeper than global GBFS, and local rand walks tend to search even deeper.

Chapter 5

Understanding and Improving Local Exploration in Greedy Best First Search

Chapter 4 has shown that the local exploration in GBFS-LE can dramatically improve the performance of GBFS. One implementation of the GBFS-LE framework, GBFS with local GBFS (GBFS-LS), is the focus of this chapter. GBFS-LS does not add new search algorithms, it only differs from GBFS in the addition of a local GBFS, and was experimentally shown to yield a substantial improvement for IPC planning domains that have large UHRs. This chapter analyzes the reasons for this improvement in detail.

The analysis will illustrate that a search method such as GBFS, which uses a global open list, can become stuck in the union of many distinct UHRs from different parts of the search space, which combine to form a large virtual UHR over the open list. This is called the *Multiple Uninformative Heuristic Region* problem, which can be overcome by local exploration. This section is based on the following publication:

- *F. Xie, M. Müller and R. Holte. Understanding and Improving Local Exploration for GBFS. In Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS-2015), 244–248, 2015* [72].

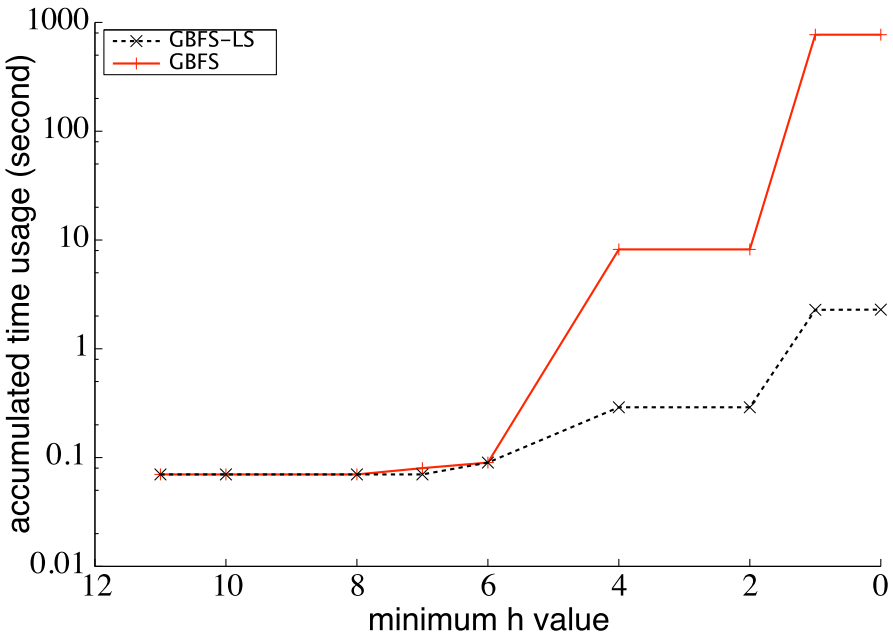


Figure 5.1: Search time (in seconds) of GBFS and GBFS-LS with h^{FF} for a given h-value (x-axis) to first become the minimum of all h-values generated up to that time. 2004-pipesworld-notankage instance #21.

# of node expansions needed for escaping (NEE)	number of nodes
[1,10]	456 (7.1%)
(10, 100]	66 (1.0%)
(100, 1000]	0 (0%)
>1000	5874 (91.8%)

Table 5.1: Number of nodes with different number of escaping node expansions.

5.1 Introduction

Instance #21 from the IPC-2004 pipesworld-notankage domain shows a clear case of such a large virtual UHR. Figure 5.1 plots the accumulated search time that GBFS and GBFS-LS need to reach the first node with a given minimum h^{FF}-value. GBFS requires almost 1000 seconds to decrease its minimum h-value from 2 to 1. GBFS-LS solves the whole problem in 2 seconds.

Analyzing the search tree of GBFS shows that search stalls in a large virtual UHR as shown in Figure 5.2. Since this virtual UHR is surrounded by a barrier of nodes with heuristic value 3, GBFS has to expand all nodes with heuristic value

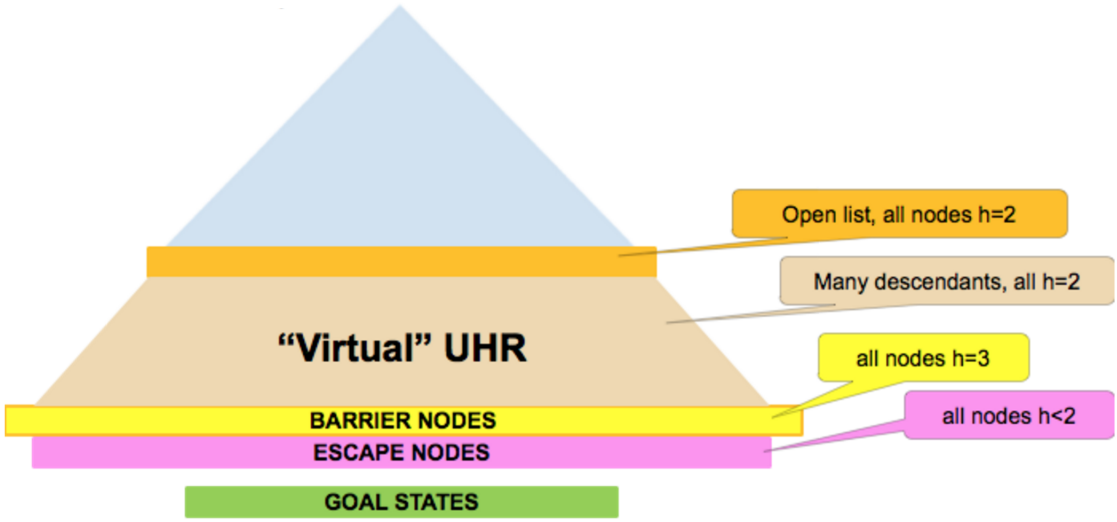


Figure 5.2: Abstract structure of the search tree of GBFS when it has stalled in heuristic value 2.

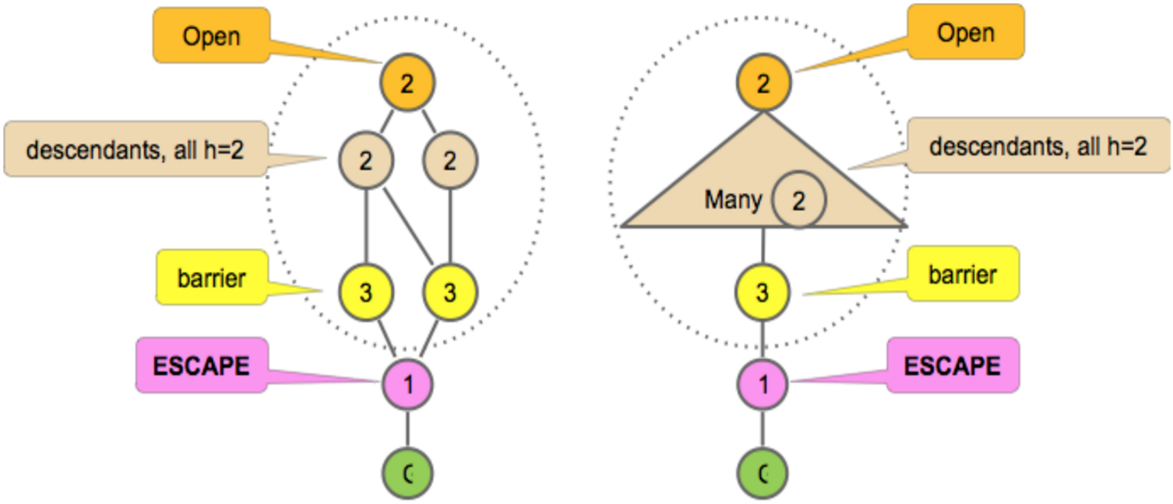


Figure 5.3: Small and large UHRs.

2 in the virtual UHR before extending any node in the barrier. However, the good news is that the virtual UHR consists of both small and large UHRs from different regions of the state space, such as the two UHRs shown in Figure 5.3. If a local GBFS starts from the root node of a small UHR as in Figure 5.3 (left), it only needs 4 node expansions to escape the UHR. Given this analysis, the question then becomes whether there is a sufficient number of small UHRs to make local GBFS effective?

Consider the following snapshot of GBFS: the minimum h-value (h_{min}) on the Open list is 2, there are 6396 nodes n on the Open list with $h(n) = 2$, and, since none of them has a child c with $h(c) < 2$, GBFS expands all 6396 nodes. In contrast, GBFS-LS expands only a small fraction of these nodes since some have a relatively quick local escape to a node n' with $h(n') < 2$. To quantify this, a small local GBFS was started from each of these nodes n to determine NEE(n), the number of nodes expanded by a local GBFS search starting at n before a node with $h(n) < 2$ is encountered. Table 5.1 lists the order of magnitude of NEE(n) for these nodes. 7.1% of nodes, a non-negligible fraction, allow for a very quick escape, with $NEE(n) \leq 10$. In contrast, standard GBFS without local exploration requires 1.8 million node expansions to reach a node with $h(n) \leq 1$. This analysis illustrates the existence of small UHRs within a large virtual UHR and gives a hint of why local GBFS can substantially reduce the time required to reach a goal state.

5.1.1 Contributions and Organization of this Chapter

The contributions of this chapter can be summarized as follows:

- Illustrate that there are both small UHRs and large UHRs in the open list of GBFS;
- Explain why adding local GBFS improves the performance;
- Show how to further improve the performance of GBFS based on the distribution of NEE values as in the example above.

The remainder of the chapter is organized as follows. A more detailed analysis is presented to illustrate the existence of multiple UHRs and why GBFS-LS out-

perform the GBFS on three IPC instances in detail. A modification of GBFS-LS, which further improves performance, is proposed and tested experimentally.

5.2 The Problem of Simultaneous Expansion of Multiple Uninformative Heuristic Regions

This section investigates the problem of GBFS with multiple UHRs by analyzing search behaviour in three IPC planning instances: 2000-Schedule #10-0, 2004-pipesworld-notankage #21, and 2008-Cyber-security #01. h^{FF} is used as the heuristic. Experiments use one core of a 2.8 GHz Intel Xeon CPU machine with 4 GB memory and 30 minutes per instance.

In all three instances, GBFS gets stuck at $h_{min} = 2$. It fails to find a lower h-value in 30 minutes for 2000-Schedule #1, needs 1.8 million node expansions (around 1000 seconds) in 2004-pipesworld-notankage #21 to decrease h_{min} from 2 to 1, and needs 2.5 million node expansions (nearly 800 seconds) to achieve the same step in 2008-Cybersecurity #1. GBFS-LS's search time for completely solving these three instances is 28.26, 2.29 and 4.32 seconds respectively.

5.2.1 Small UHRs and Large UHRs

Given an expansion limit L (1000 in our experiments), a node n on the global open list is said to be a *small UHR* if a local GBFS search from n generates a node v with $h(v) < h_{min}$ after expanding L or fewer nodes. i.e. $NEE(n) \leq L$. Otherwise, n is a *large UHR*.

The following experiments investigate the frequency of small and large UHRs in the open list of GBFS for the three planning instances above. The experiment begins when the first node n_1 with $h(n_1) = 2$ is added to the open list. The experiment has the following three steps:

1. Keep GBFS running for 10,000 *initializing expansions* in order to add more nodes to the open list.

2. Define set S as the first 5000 nodes in the open list at this point in time. Use random tie-breaking to choose among nodes with equal h-value.

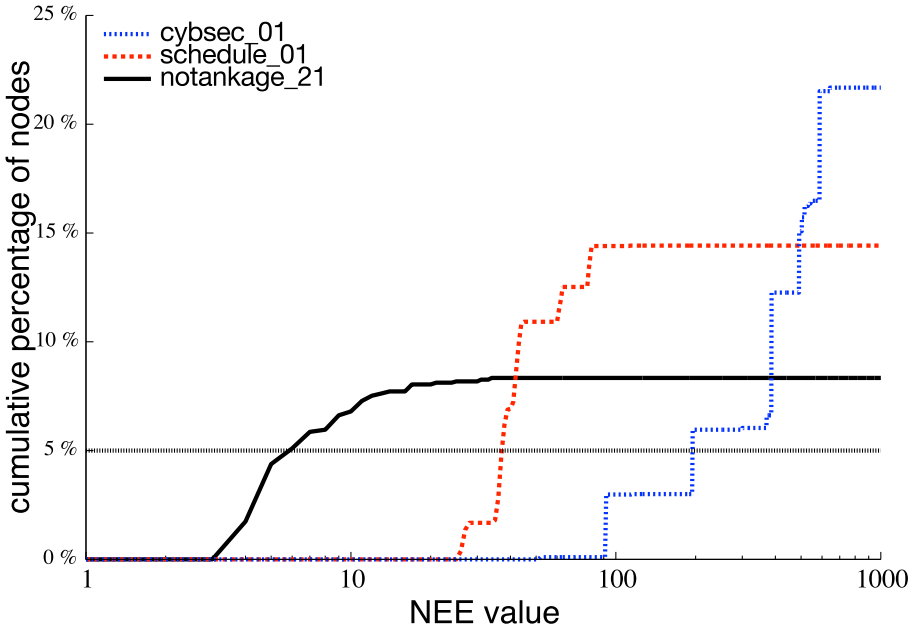


Figure 5.4: The distribution of NEE values over the 5000 picked nodes in 2000-Schedule #01, 2004-pipesworld-notankage#21, and 2008-cybersecurity #01.

3. Run local GBFS from each node $n \in S$, setting $h = h^{FF}$, $h_{min} = 2$, $L = 1000$. Each local search uses an initially empty local open list and a local closed list initialized with the (fixed) global closed list. Both the local open list and the local closed list are discarded afterwards. The NEE of all nodes in S is recorded.

For the three test instances, Figure 5.4 shows the percentage of nodes with a NEE value less than or equal to the value on the x-axis. A non-negligible percentage of nodes with a relatively small NEE are found in all three instances. A local search starting from any of these nodes succeeds in reducing h_{min}. This result helps explain why GBFS-LS can quickly make progress, and can be orders of magnitude faster than GBFS.

The distribution of NEE varies among the three instances. For example, the value of x such that 5% of the nodes have a $NEE(n) \leq x$ is 6 for 2005-pipesworld-notankage #21, but is 38 for 2000-Schedule #01, and 195 for 2008-cybersecurity #01. For ease of comparison, a reference line with y=5% is shown in the figure. In each case, while the majority of the 5000 analyzed nodes seem to be located in

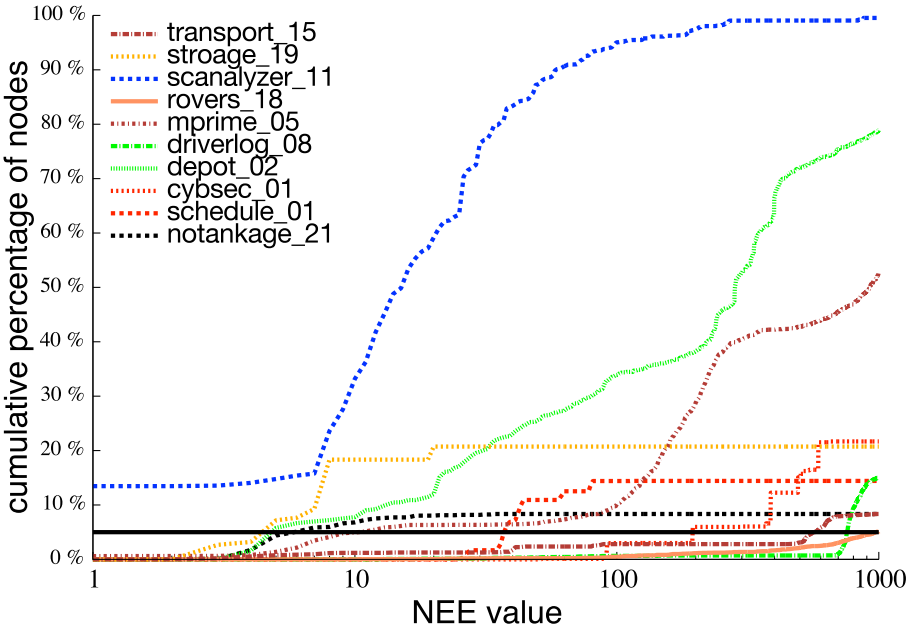


Figure 5.5: The distribution of NEE values over the 5000 picked nodes in 10 different planning instances.

large UHRs and cause the local search to fail, there is a sufficient number of nodes in small UHRs that local GBFS can make quick progress.

Besides the three domains for which examples were analyzed above, co-occurring small and large UHRs were detected in 7 further IPC domains: 1998-Mystery-Prime, 2002-Depot, 2002-Driverlog, 2006-Rovers, 2006-Storage, 2008-Scanalyzer and 2008-Transport. All these domains contain some instances where the GBFS expands a larger number of nodes than the GBFS-LS in improving some h_{min} and meanwhile the GBFS-LS only expands a small number of nodes (less than 5,000) in finding such an improvement. In these instances, co-occurring small and large UHRs can be easily detected using the same method above. Figure 5.5 illustrate the NEE value distribution over 10 instances from these 10 domains.

5.2.2 Why does Global GBFS not explore Small UHRs?

In the cases analyzed above, enough small UHRs exist, from which local GBFS can easily escape. Why does global GBFS not find an escape path from these small UHRs? Is GBFS just unlucky, and always picks nodes with high NEE? The answer is no. Further analysis shows that all the escape paths found by local GBFS in 2000-

Schedule #10-0, 2004-pipesworld-notankage #21 and 2008-Cyber-security #1 go through at least one *barrier* node n with $h(n) > h_{min}$. This means that all these small UHRs are local minima, not plateaus from which global GBFS could escape. While local GBFS can expand across such barrier nodes very quickly, GBFS is forced to exhaustively expand all nodes n' in all UHRs with $h(n') = h_{min}$, before it can expand any nodes with larger h.

As an extreme example, assume that all but one of the UHRs are local minima needing only 2 node expansions to escape from and that the other UHR contains a large number of nodes n' with $h(n') = h_{min}$. GBFS has to expand all the nodes in the large UHR before it can find an escape path from one of the others. The discussion here matches the observation in Chapter 4 that GBFS-LS only improves the performance in domains that contain large UHRs. Such barrier nodes also exist for other two commonly used heuristics: causal graph (CG) [18] and context-enhanced additive (CEA) [21].

However, not all small UHRs must be local minima. As an example, in 2006-Pipesworld-Tankage instance #32, GBFS needs 4706 node expansions in 13 seconds to improve h_{min} from 6 to 5, while it takes GBFS-LS 6.6 seconds and 1621 node expansions starting from the same open list. The virtual UHR over the open list contains both local minima and plateaus. Because the global GBFS can escape from the plateaus, GBFS-LS does not improve the performance as dramatically as it does in the three instances above.

5.3 More Exploration with Smaller Local GBFS

Greedy Best First Search with Local Search (GBFS-LS) [68] is the same as GBFS except it executes a local GBFS whenever the global GBFS (G-GBFS) seems stalled. G-GBFS is considered *stalled* if it fails to improve its minimum heuristic value h_{min} for a specified number *STALL_SIZE* of node expansions, set to 1000 by default. In this case GBFS-LS runs a small local GBFS for exploration, from a best node *n* in the (global) open list. After each local exploration, the mechanism for detecting a stalled global search is reset.

Local GBFS can dramatically improve the time to solution if used for nodes in small UHRs. In the original GBFS-LS algorithm, a single local GBFS is called and expands up to 1000 nodes whenever G-GBFS did not improve h_{min} over its last 1000 expansions. Therefore each single local GBFS is relatively expensive.

The experiments above suggest that using more frequent but smaller local searches may be a good tradeoff. To investigate this with minimal changes to the algorithm, the proposed new scheme GBFS-LS-$X \times Y$, where $X \times Y = 1000$, runs X local searches with Y expansions each from X random nodes in the best (minimum h) bucket in the open list. If there are fewer than X nodes in this bucket, the remaining nodes are chosen from the next-best bucket(s). Our experiments include the following pairs of (X, Y): (1, 1000), (10, 100), (100, 10) and (1000, 1).

Experiments were run on the same set of 2112 problems as in Chapter 4, in 54 domains from the first seven International Planning Competitions (IPC 1 to 7), using one core of a 2.8 GHz Intel Xeon CPU machine with 4 GB memory and 30 minutes per instance. Results for planners which use randomization are averaged over five runs. All planners are implemented on the Jasper [69] code base, which is downloaded from the IPC-8 website.[1] The translation from PDDL to SAS$^+$ was done only once, and this common preprocessing time is not counted in the 30 minutes.

Heuristic	GBFS	LS	LS-1×1000	LS-10×100	LS-100×10	LS-1000×1
FF	1561	1641	1641.0	**1678.2**	1659.2	1576.4
CG	1513	1608	1600.2	**1618.4**	1595.2	1516.4
CEA	1498	1592	1577.2	**1612.4**	1609.8	1497.0

Table 5.2: IPC coverage out of 2112 for GBFS, GBFS-LS, GBFS-LS-1×1000, GBFS-LS-10×100, GBFS-LS-100×10 and GBFS-LS-1000×1 under three standard heuristics. LS is short for GBFS-LS.

Table 5.2 compares the new algorithms with GBFS and GBFS-LS. Three widely used planning heuristics are tested: FF [25], causal graph (CG) [18] and context-enhanced additive (CEA) [21]. Table 5.2 shows the coverage on all 2112 IPC instances. Overall, GBFS-LS-10×100 outperforms GBFS, GBFS-LS and other configurations for all three heuristics.

[1] http://helios.hud.ac.uk/scommv/IPC-14/

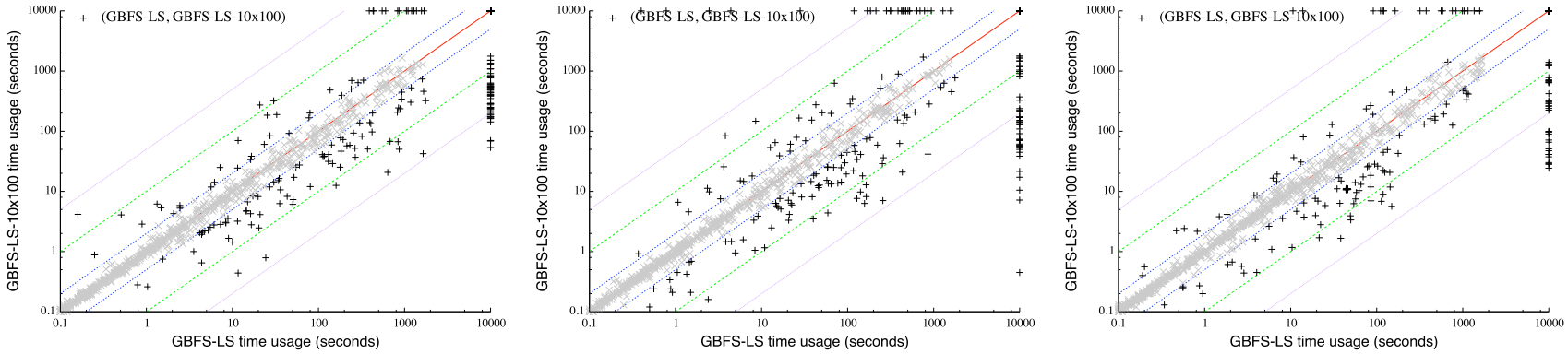
GBFS-LS-1×1000 and GBFS-LS-1000×1 are added for evaluating the influence of randomness. While GBFS-LS applies a deterministic first-in-first-out approach in picking the starting node for local GBFS, GBFS-LS-1×1000 applies tie-breaking uniformly at random. However, these two algorithms achieve very similar coverage results. Similarly, GBFS-LS-1000×1 is very close to a GBFS version that applies the random tie-breaking, which also results in a very similar coverage result to GBFS. These two data points show that the superior performance of GBFS-LS-10×100 over GBFS-LS is due to running a larger number of small local searches and not due to the randomness in the node selection process.

Figure 5.6 compares the search time of GBFS-LS-10×100 (y-axis) with GBFS-LS (x-axis) in *IPC time comparison setting* (see Chapter 4 Section 4.4.2) over the IPC benchmarks. For all the heuristics tested, besides its improved coverage, GBFS-LS-10×100 also shows a substantial improvement in search time over GBFS-LS, with many more results in the factor 2 to 10 speed up region favouring the new algorithm.

The same modification was also tested with LAMA-LS, which replaces the GBFS component of LAMA-2011 with GBFS-LS from Chapter 4. Unfortunately, there is no noticeable improvement here: LAMA-LS and LAMA-LS-10×100 are very similar in both coverage and search time. One possible reason is that the major enhancements in LAMA-2011 such as deferred evaluation, preferred operators [45] and multiple heuristics [46], already cover some bad scenarios for GBFS-LS. This is a topic for future study.

5.4 Chapter Summary

This chapter illustrates the multiple UHRs problem of GBFS using three IPC examples, and explains why adding local GBFS improves the performance of GBFS. As suggested by the analysis, it is confirmed that running a larger number of smaller local searches further improves the performance.



(a) FF heuristic (b) CG heuristic (c) CEA heuristic

Figure 5.6: Comparison of search time of GBFS-LS-10×100 with GBFS-LS for three different heuristics. For each heuristic, we compared all 5 runs results of GBFS-LS-10×100 with GBFS-LS, and one typical run is selected.

Chapter 6

Adding Global Exploration with Type Buckets to Greedy Best First Search

Chapter 3 and 4 have described how and why local exploration improves the performance of GBFS. This chapter proposes a framework that adds global exploration into GBFS via a type system and the multiple queue approach. This chapter is based on the following publication:

- *F. Xie, M. Müller, R. Holte and T. Imai. Type-based Exploration for Satisficing Planning with Multiple Search Queues. In Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI-2014), 2395-2401, 2014* [70].

6.1 Introduction

The popular enhancements to planning systems such as Preferred Operators [45] and Multi-Heuristic [46] are implemented in a *Multiple Queue Search* framework [19]. Separate priority queues are used to hold different sets of nodes, or keep them sorted according to different heuristics.

Still, each queue is sorted based on some heuristic h, and is used in a greedy fashion by the search, which always expands a node with minimum h-value from one of the queues. This makes search vulnerable to the *misleading heuristic* problem, where it can stall in *bad subtrees*, which contain large local minima or plateaus

but do not lead to an easy solution. Adding exploration to a search algorithm is one way to attack this problem.

Previous approaches to this problem of GBFS with misleading heuristics include K-BFS [13], which expands the first k best nodes in a single priority queue and adds all their successors, and Diverse-BFS [26], which expands extra nodes with non-minimal h-values or at shallow levels of the search tree. Another simple algorithm is ϵ-GBFS [60], which expands a node selected uniformly at random from the open list with probability ϵ. All these algorithms add an element of exploration.

This chapter proposes and evaluates a simple yet very effective way of adding exploration based on a *type system* [35]. The major contributions are:

1. An analysis of the weaknesses of previous simple exploration schemes. and a non-greedy approach to exploration based on a simple type system.

2. A search algorithm in the framework of multiple-queue search named *Type-GBFS* which uses a type system for exploration, and the corresponding planner *Type-LAMA*, which replaces the GBFS component of LAMA-2011 by Type-GBFS.

3. Detailed experiments on IPC benchmarks, which demonstrate that baseline Type-GBFS solves substantially more problems than baseline GBFS, and that this superiority also holds when adding most combinations of standard planning enhancements. Type-LAMA with all such enhancements outperforms LAMA-2011.

6.2 Early Mistakes in GBFS

Early mistakes are mistakes in search direction at shallow levels of the search tree caused by sibling nodes being expanded in the wrong order. This happens when the root node of a *bad subtree*, which contains no solution or only hard-to-find solutions, has a lower heuristic value than a sibling that would lead to a quick solution.

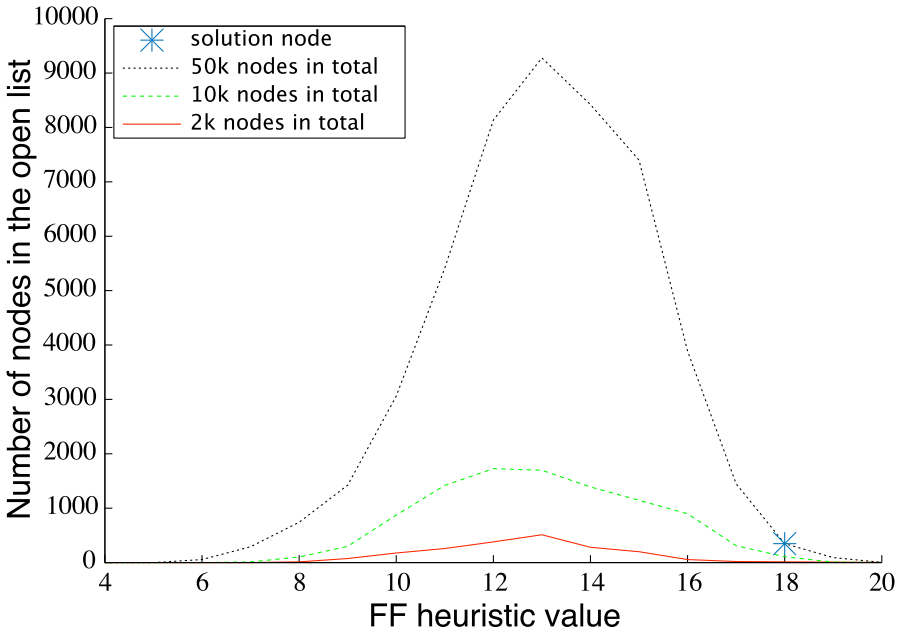
The 2011-Nomystery domain from IPC-2011 is a typical example where delete-relaxation heuristics systematically make early mistakes [42]. In this transporta-

tion domain with limited non-replenishable fuel, delete-relaxation heuristics such as h^{FF} ignore the crucial aspect of fuel consumption, which makes the heuristic overoptimistic and misleading, and results in large unrecognized dead-ends in the search space. Bad subtrees in the search tree, which over-consume fuel early on, are searched exhaustively, before any good subtrees which consume less fuel and can lead to a solution are explored. As a result, while the random walk-based planner Arvand with its focus on exploration solved 19 out of 20 nomystery instances in IPC-2011, LAMA-2011 solved only 10.

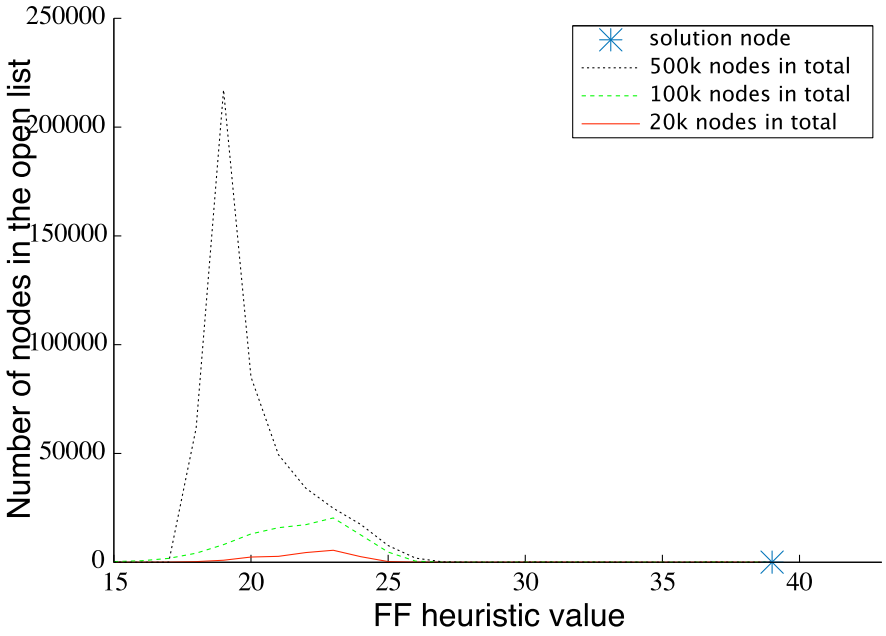
6.3 Exploration bias in the Open List: Two Case Studies

Previous exploration methods in GBFS suffer from biasing their exploration heavily towards the neighborhood of nodes in the open list. In the case of early mistakes, the large majority of these nodes is in useless regions of the search space. Consider the nodes in the regular h^{FF} [25] open list of LAMA-2011 while solving the problem 2011-nomystery #12. Figure 6.1(a) shows snapshots of their h-value distribution after 2,000, 10,000 and 50,000 nodes expanded. In the figure, the x-axis represents different heuristic values and the y-axis represents the number of nodes with a specific h value in the open list. The solution eventually found by LAMA-2011 goes through a single node n in this 50,000 node list, with $h(n) = 18$. This node is marked with an asterisk in the figure. Over 99% of the nodes in the open list have lower h-values, and will be expanded first, along with much of their subtrees. However, in this example, none of those nodes leads to a solution. The open list is flooded with a large number of useless nodes with undetected dead ends.

ϵ-GBFS [60] samples nodes uniformly over the whole open list. This is not too useful when entries are heavily clustered in bad subtrees. In the example above, ϵ-GBFS has a less than 1% probability to pick a node with h-value 18 or more in its exploration step, which itself is only executed with probability ϵ. Furthermore, the algorithm must potentially select several good successor nodes before making measurable progress towards a solution by finding an exit node with a lower h-

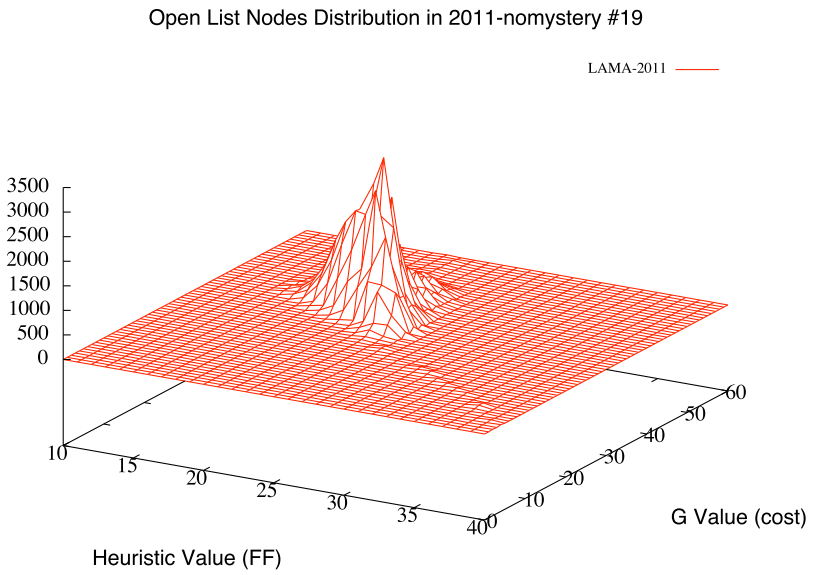


(a) h-values in nomystery #12



(b) h-values in nomystery #19

Figure 6.1: h-value distribution in the regular h^{FF} open list of LAMA-2011.



(h, g)-distribution in nomystery #19

Figure 6.2: Distribution of types in the regular h^{FF} open list of LAMA-2011 after 100,000 nodes in 2011-nomystery #19.

value.

The instance 2011-nomystery #12, with 6 locations and 6 packages, has a relatively small search space, and both GBFS and ϵ-GBFS eventually solve it after exhaustively enumerating the dead ends. However, a larger problem like 2011-nomystery #19, with 13 locations and 13 packages, is completely out of reach for GBFS or ϵ-GBFS. This instance was solved by only 2 planners in IPC-2011. Figure 6.1(b) shows the h-value distribution in LAMA-2011's regular h^{FF} queue after 20,000, 100,000 and 500,000 nodes. The node with $h = 39$ from a solution found by Arvand-2011 [38] is marked at the far right tail of the distribution in the figure.

6.4 Adding Exploration via a Type System

Can the open list be sampled in a way that avoids the over-concentration on a cluster of very similar nodes? A *type system* [35], which is based on earlier ideas of stratified sampling [7], is one possible approach.

6.4.1 Type System

A *type system* is defined as follows:

Definition 3. *[35] Let S be the set of nodes in a given search space. $T = \{t_1, \ldots, t_n\}$ is a type system for S if T is a disjoint partitioning of S. For every $s \in S$, $T(s)$ denotes the type $t \in T$ for s.*

Types can be defined using any property of nodes. The simple type system used here defines the type of a node s in terms of its h-value for different heuristics h, and its g-value. A simple and successful choice is the pair $T(s) = (h^{FF}(s), g(s))$. The intuition behind such type systems is that they can roughly differentiate between nodes in different search regions, and help explore away from the nodes where GBFS gets stuck.

Figure 6.2 views a LAMA-2011 search of instance 2011-nomystery #19 through the lens of a (h^{FF}, g) type system. The horizontal x- and y-axes represent h^{FF}-values and g-values respectively. The number of nodes in the open list with a specific (h^{FF}, g) type is plotted on the vertical z-axis. The graph shows the frequency of each type in the regular h^{FF} open list of LAMA-2011 at the time when the open list first reaches 100,000 nodes. After initial rapid progress, search has stalled around a single peak. Most of the open list is filled with a large number of useless nodes, which lead to no solution.

6.4.2 Type-GBFS: Adding a Type System to GBFS

Type-GBFS uses a simple two level *type bucket* data structure tb which organizes its nodes in buckets according to their type. Type bucket-based node selection works as follows: First, pick a bucket b uniformly at random from among all the non-empty buckets. Then pick a node n uniformly at random from all the nodes in b. Type-GBFS alternately expands a node from from the regular open list O and from tb, and each new node is added to both O and tb.

Multi-Heuristic type systems $(h_1(s), h_2(s), ...)$ have been explored before using a *Pareto set* approach [52]. The main differences between their approach and ours are: 1) only *Pareto Optimal* buckets are selected in their approach, while all

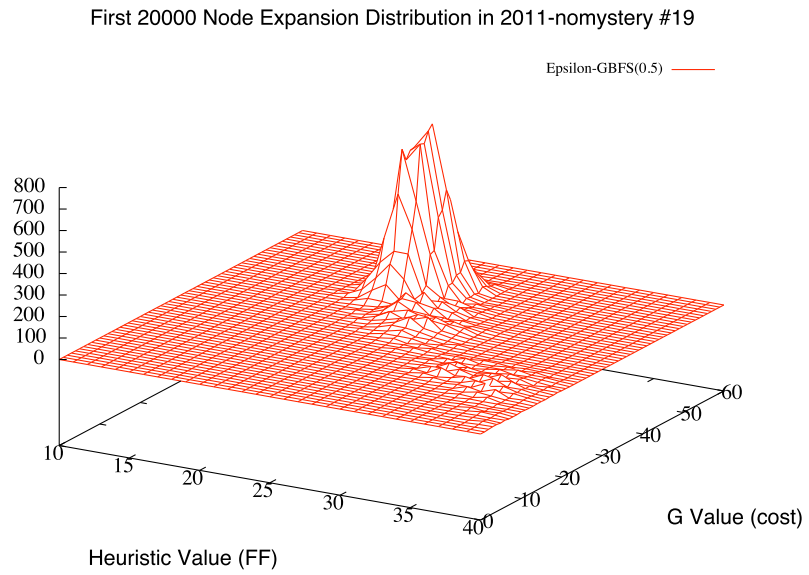
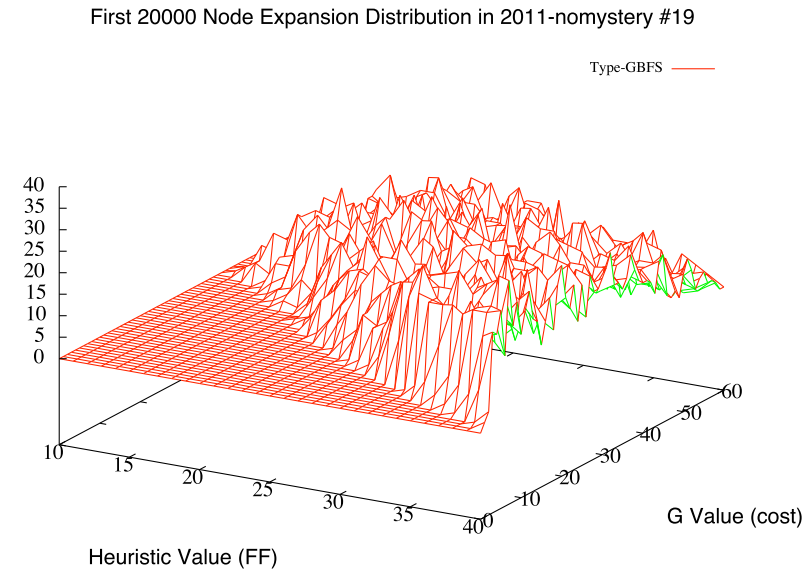


Figure 6.3: Distribution of types over the first 20,000 nodes expanded in the exploring phase (ϵ-exploration or type buckets) of ϵ-GBFS($\epsilon = 0.5$).

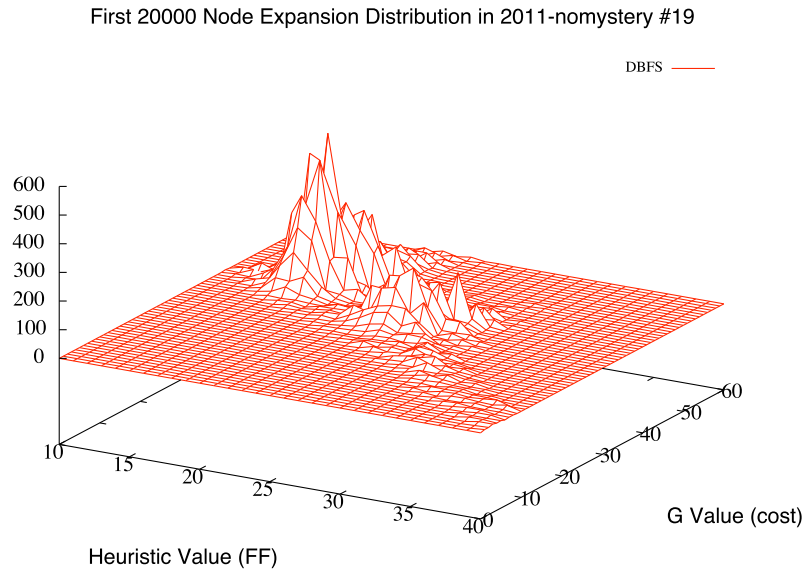
buckets can be select in Type-GBFS; 2) the probability of selecting each Pareto optimal bucket is proportional to the number of nodes it contains in their approach, while Type-GBFS selects all bucket uniformly random ; 3) only heuristics are used to define types in their approach, whereas our approach also considers g and potentially any other relevant information; and 4) nodes in a bucket are selected deterministically in FIFO order, not uniformly at random.

Diverse Best-First Search (DBFS) [26] is another closely related high performance search algorithm which includes an exploration component. This two-level search algorithm uses a global open list O_G, a local open list O_L and a shared closed list.

Like Type-GBFS with the $(h_{FF}(s), g(s))$ type system, DBFS picks nodes based on their h- and g-values. There are three major differences between these algorithms. 1) DBFS performs a sequence of local searches while Type-GBFS defines a single global search; 2) DBFS uses g to restrict its node selection, while Type-GBFS can use g as part of its type system; 3) DBFS biases its node selection using h, while Type-GBFS samples uniformly over all types.



(a) Type-GBFS



(b) DBFS

Figure 6.4: Distribution of types over the first 20,000 nodes expanded in the exploring phase of Type-GBFS and DBFS.

6.4.3 Exploration in Type-GBFS, ϵ-GBFS and DBFS

Type-GBFS and ϵ-GBFS with $\epsilon = 0.5$ both spend half their search effort on exploration. However, the distribution of types of the explored nodes is very different. In Nomystery-2011 #19, GBFS in LAMA-2011 grows the single peak shown in Figure 6.2. Figure 6.3 and 6.4 shows the frequency of explored node types for ϵ-GBFS with $\epsilon = 0.5$, Type-GBFS[1] and DBFS after 20,000 node expansions. Note that while Figure 6.1 shows the distribution of all nodes in the open list, Figure 6.3 and 6.4 shows the types of only those nodes that were chosen in the exploration step[2].

ϵ-GBFS mainly explores nodes close to the GBFS peak types, while Type-GBFS explores much more uniformly over the space of types. DBFS explores more types than ϵ-GBFS. Unlike Type-GBFS, which samples types uniformly, DBFS is still biased towards low h and high g values.

Note that the z-axis scales are different for the three plots. The single most explored type contains around 800 nodes for ϵ-GBFS and 600 for DBFS, but only 40 for Type-GBFS. The presence or absence of exploration helps explain the relative performance in 2011-Nomystery. The coverage for the 20 instances of this domain for one typical run with 4 GB memory and 30 minutes per instance is 9 for GBFS, 11 for ϵ-GBFS with $\epsilon = 0.5$, 17 for Type-GBFS and 18 for DBFS.

6.5 Experiments

The experiments use a set of 2112 problems (54 domains) from the first seven International Planning Competitions, and were run on an 8-core 2.8 GHz machine with 4 GB memory and 30 minutes per instance. Results for planners which use randomization are averaged over five runs. All algorithms are implemented using the Fast Downward code base FD-2011 [19]. The translation from PDDL to SAS+ was done only once, and this common preprocessing time is not counted in the 30 minutes.

[1] Some explored types are outside the (h, g) range shown in Figure 6.4(a).

[2] Unlike ϵ-GBFS and Type-GBFS, there is no clear exploration step in DBFS. All visited nodes are shown in the figure.

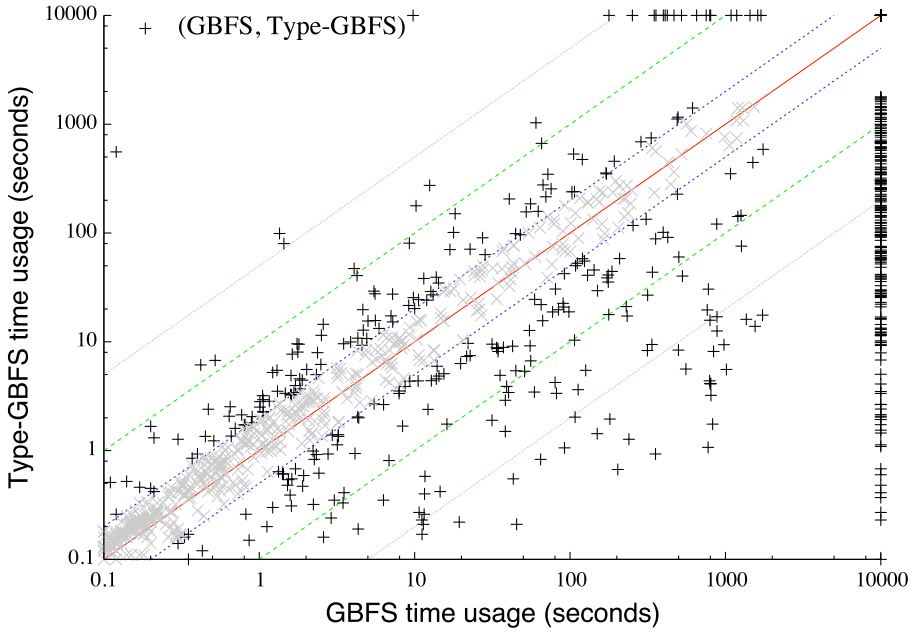


Figure 6.5: Comparison of search time of GBFS and Type-GBFS with the h^{FF} heuristic on IPC.

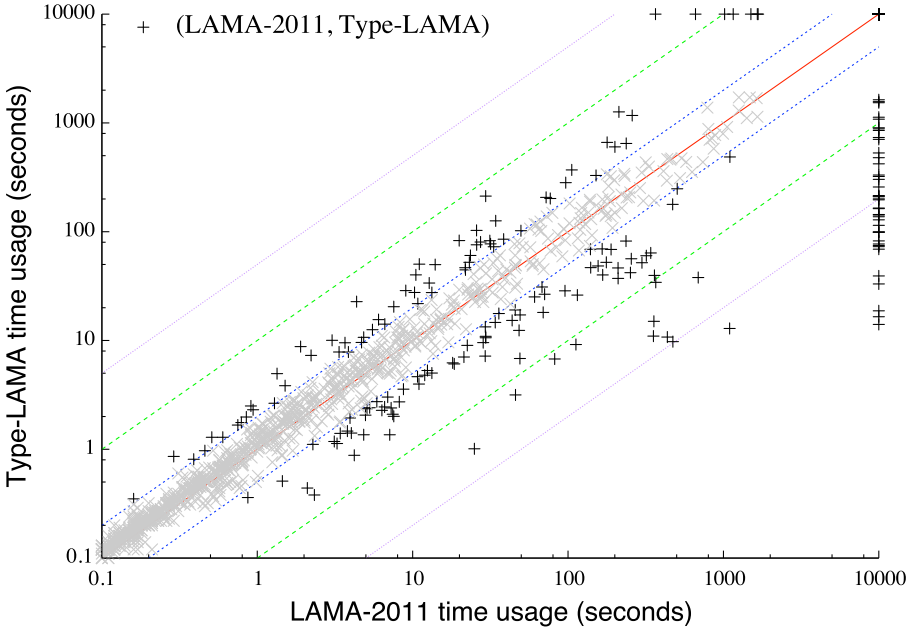
6.5.1 Performance of Baseline Algorithms

The baseline study evaluates the two algorithms GBFS and Type-GBFS without the common planning enhancements of preferred operators, deferred evaluation and multi-heuristics. It uses three popular planning heuristics - FF [25], causal graph (CG) [18] and context-enhanced additive (CEA) [21]. Table 6.1 shows the coverage results. Type-GBFS outperforms GBFS by a substantial margin for each tested heuristic.

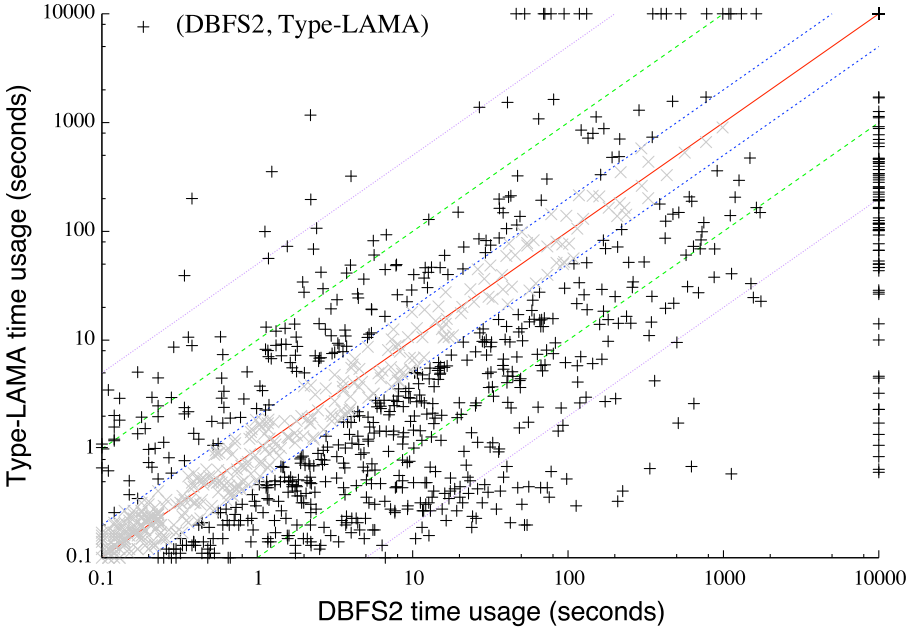
Heuristic	GBFS	Type-GBFS
FF	1561	**1755.6**
CG	1513	**1691.4**
CEA	1498	**1678.8**

Table 6.1: Baseline GBFS vs. Type-GBFS - coverage of 2112 IPC instances.

Figure 6.5 compares the time performance of the baseline algorithms with h^{FF} in *IPC time comparison setting* (see Chapter 4 Section 4.4.2). Beyond solving almost 200 more problems, Type-GBFS shows a substantial advantage over GBFS in search time for problems solved by both planners. There are many more problems where Type-GBFS outperforms GBFS by more than a factor of 10 or 50 than vice



(a) Lama-2011 (x) vs. Type-LAMA (y)



(b) DBFS2 (x) vs. Type-LAMA (y)

Figure 6.6: Comparison of search time. (a): GBFS and Type-GBFS with the h^{FF} heuristic on IPC. (b)(c): Type-LAMA vs. Lama-2011 (left) and DBFS2 (right). using typical single runs of Type-GBFS, Type-LAMA and DBFS2.

versa. Still, while Type-GBFS outperforms GBFS overall, it does not dominate it on a per-instance basis. Sometimes the extra exploration of Type-GBFS wastes time or even leads the search astray for a while.

6.5.2 Performance with Different Enhancements

How do GBFS and Type-GBFS compare when common planning enhancements are added? All combinations of Deferred Evaluation, Preferred Operators and Multiple Heuristics are tested, with h^{FF} as the primary heuristic. As the enhancements are applied the same as in Chapter 4 Section 4.4.3, we only list the special settings for Type-GBFS:

- With **Preferred Operators**, Type-GBFS uses only a single set of type buckets for all nodes. There are no separate type buckets containing preferred nodes only.
- With **Multi-Heuristics** Type-GBFS uses two open lists, one for each heuristic, plus type buckets for the (h^{FF}, g) type system.

When both Multi-Heuristic and Preferred Operators are applied, GBFS uses four queues, two regular and two preferred ones. Type-GBFS uses the same queues plus (h^{FF}, g) type buckets.

Enhancement	GBFS	Type-GBFS
(none)	1561	1755.6
PO	1826	1848.6
DE	1535	1834.6
MH	1851	1789.8
PO + DE	1871	1906.4
PO + MH	1850	1846.2
DE + MH	1660	1729.0
PO + DE + MH	**1913**	**1949.8**

Table 6.2: Number of IPC tasks solved out of 2112. PO = Preferred Operators, DE = Deferred Evaluation, MH = Multi-Heuristic.

Table 6.2 shows the experimental results on IPC domains for all 8 combinations of enhancements. When used as a single enhancement, Preferred Operators

and Multiple Heuristic improve both algorithms. Deferred Evaluation also strongly improves Type-GBFS, but causes a slight decrease in coverage for GBFS, mainly due to plateaus caused by the less informative node evaluation [45]. Apparently, Type-GBFS gets stuck in such plateaus less often.

When combining any two enhancements, both algorithms achieve their best performance with Preferred Operators plus Deferred Evaluation, as observed for GBFS in Richter and Helmert's work [45]. Multiple Heuristics have a negative effect on Type-GBFS when combined with either Preferred Operators or Deferred Evaluation, but work very well when combined with both. Finding an explanation for this surprising behavior is left as future work. The last row in Table 6.2 lists coverage results when all three enhancements are applied as in LAMA.

6.5.3 Comparing with LAMA-2011 and DBFS2 in Terms of Coverage and Search Time

The performance comparison in this section includes the following planners:

- **LAMA-2011:** only the first iteration of LAMA using GBFS is run, with deferred evaluation, preferred operators and multi-heuristics (h^{FF}, h^{lm}) [46].
- **Type-LAMA:** LAMA-2011 with GBFS replaced by Type-GBFS, uses the same four queues as LAMA-2011, plus (h^{FF}, g) type buckets.
- **DBFS2:** DBFS2 [26], an enhanced version of DBFS which adds a second global open list for preferred operators only, was re-implemented by its original author on the LAMA-2011 code base, which is stronger than the LAMA-2008 code base used in the original experiment [26]. The parameters $P = 0.1, T = 0.5$ from the same paper are used.

LAMA-2011 and Type-LAMA correspond to PO+DE+MH in Table 6.2.

Table 6.3 shows detailed coverage differences of these three planners. Type-LAMA outperforms the other two planners, solving 36.8 more problems than LAMA-2011 and 54.4 more than DBFS2. The percentage of unsolved problems is reduced from 9.4% for LAMA to 7.6%. This is comparable to the improvement from adding

domain	size	LAMA-2011	Type-LAMA	DBFS2
98-logistics	35	**35**	**35**	34
98-mystery	30	**19**	**19**	**19**
00-miconic-full	150	136	**139**	138.6
02-depot	22	20	**21.6**	18.2
02-freecell	80	78	77.8	**79.8**
04-airport-strips	50	32	34.6	**41**
04-notankage	50	**44**	**44**	43.2
04-optical-tele	48	4	**5.4**	5
04-philosopher	48	39	**48**	**48**
04-psr-large	50	**32**	31.6	15.6
04-satellite	36	**36**	35.2	27
06-pathways	30	**30**	**30**	28.4
06-storage	30	18	**23.8**	23.4
06-tankage	50	41	**42**	36.8
06-trucks-strips	30	14	20.8	**24**
08-scanalyzer	30	**30**	**30**	29.6
08-sokoban	30	**28**	27	**28**
08-transport	30	29	**30**	29.8
08-woodworking	30	**30**	29.8	**30**
11-elevators	20	**20**	**20**	18.8
11-floortile	20	6	5.6	**7.6**
11-nomystery	20	10	**17.8**	**17.8**
11-openstacks	20	**20**	**20**	12.8
11-parking	20	**18**	**17.6**	12.6
11-scanalyzer	20	**20**	**20**	19.8
11-sokoban	20	**19**	18.2	18
11-tidybot	20	16	**16.4**	16.2
11-transport	20	**16**	15.6	12.4
11-visitall	20	**20**	**20**	7
Total	2112	1913	**1949.8**	1895.4
Unsolved	-	199	**162.2**	216.6

Table 6.3: Number of instances solved. 25 domains with 100% coverage for all three planners omitted.

the *Landmark count* heuristic [46] - from 11.4% to 9.4% unsolved. Considering that only the hardest problems were left unsolved, adding the type system makes the planner substantially stronger.

Figures 6.6 compares the search time of Type-LAMA against LAMA-2011 and DBFS2 in the manner described for Figure 6.5. Between Type-LAMA and LAMA-2011, many results are very close, presumably for instances where exploration plays

only a small role. Type-LAMA has a large time advantage of over $10\times$ more often. Results for Type-LAMA and DBFS2 are much more diverse. Besides its coverage advantage, Type-LAMA also wins the time comparison over DBFS for a large number of instances by factors of over $2\times$, $10\times$ and $50\times$.

For further comparison, the coverage results of some other strong planners from IPC-2011 on the same hardware are: FDSS-2 solves 1912/2112, Probe 1706/1968 (failed on DERIVE keyword in 144 problems), Arvand 1878.4/2112, fd-auto-tune-2 1747/2112, and Lama-2008 1809/2112.

Although type buckets cause some change in solution costs, the influence is not clear-cut. If we compare LAMA-2011 and one typical run of Type-LAMA's results, there are 1907 problems solved by both planners. The IPC plan quality scores for LAMA-2011 and Type-LAMA are 1895.1 vs 1892.5, only 2.6 difference over the 1907 problems. For 1698 problems, both planners get the same cost solutions.

6.5.4 Effect of Different Type Systems

The results above for both Type-GBFS and Type-LAMA are for the (h^{FF}, g) type system. Table 6.4 summarizes results for these two planners when using several other simple type systems. $Type(1)$ is the trivial single-type system $T(s) = 1$.

Among single-element type systems, $Type(g)$ performs better than either heuristic, and $Type(h^{lm})$ solves around 10 more problems than $Type(h^{FF})$. Type-GBFS is only tested with one heuristic h^{FF}, while Type-LAMA is tested for two heuristics h^{FF} and h^{lm}.

Compared to GBFS, $Type(g)$ explores many more nodes with low g-values, which are more likely to be at shallow levels of the search tree. Many such nodes will be ignored or expanded very late in GBFS. In contrast, an (h)-only type system focuses on exploring different estimated goal distances and ignores g. Interestingly, $Type(g)$ is slightly better than $Type(h^{FF}, g)$ in Type-GBFS, but $Type(h^{FF}, g)$ is better in Type-LAMA. Among two-element type systems, $Type(h^{FF}, g)$ and $Type(h^{lm}, g)$ are the top two configurations, while $Type(h^{FF}, h^{lm})$ is just slightly better than $Type(h^{lm})$. The three-element type system $Type(h^{FF}, h^{lm}, g)$ is worse and might be too fine-grained for this test set. The question of the right granularity

is important and needs further study.

Type	T-GBFS	T-LAMA	Type	T-LAMA
none	1561	1913	(h^{lm})	1921.6
(1)	1529.6	1916.2	(h^{lm}, g)	1942.4
(g)	**1758.2**	1935.0	(h^{FF}, h^{lm})	1925.6
(h^{FF})	1729.0	1918.6	(h^{FF}, h^{lm}, g)	1939.0
(h^{FF}, g)	1755.6	**1949.8**		

Table 6.4: Coverage of Type-GBFS (T-GBFS) and Type-LAMA (T-LAMA) with simple type systems.

6.5.5 Type-LAMA Works Better as an Integrated System than as a Simple Portfolio

This experiment compares Type-LAMA, which integrates type-based exploration directly into LAMA's search process, with a portfolio which independently runs LAMA and a simple type-based planner ST. ST selects nodes exclusively from type buckets, using a (h^{FF}, g) type system as defined above. For consistency with LAMA, Deferred Evaluation is used for type buckets as well.

By itself, ST solves 1266 out of 2112 IPC problems. Consider a portfolio planner that uses x seconds for LAMA-2011, followed by $1800 - x$ seconds for ST. The best coverage of 1926 is achieved for $x = 1279$. Figure 6.7 illustrates the performance over different x value. Type-LAMA solves 23.8 problems more than this best portfolio. This shows the synergy between exploitation-based search in LAMA and exploration using a type system.

6.6 Chapter Summary

This chapter investigates the problem of inefficient exploration in previous GBFS-type planners, and proposes a solution of using type-based exploration. The new algorithm Type-GBFS samples nodes uniformly over a type system instead of uniformly over all nodes in an open list. By replacing GBFS with Type-GBFS, the planner Type-LAMA can solve 36.8 more problems than LAMA-2011 on average, decreasing the number of unsolved problems over the first 7 IPC event from 199 to 162.2.

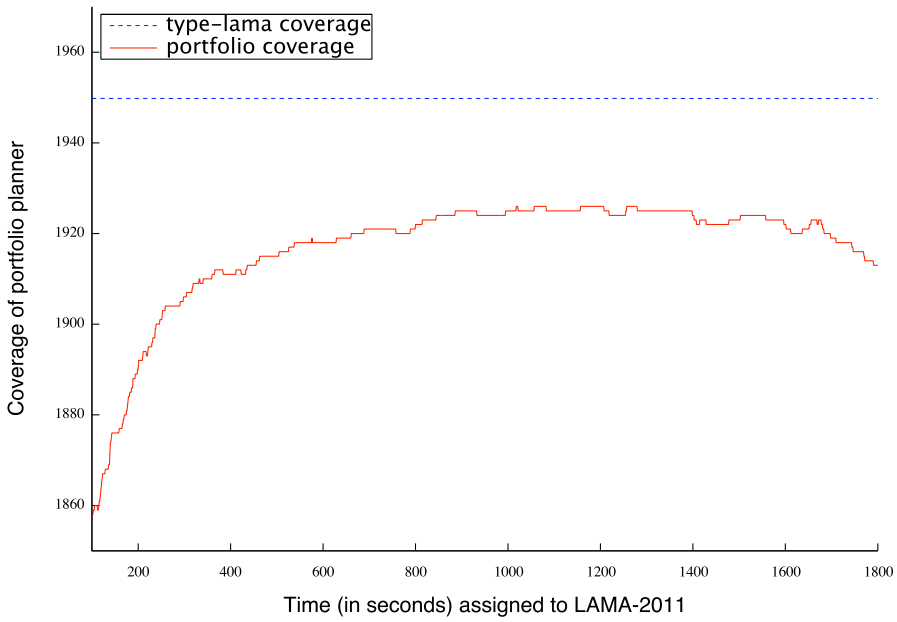


Figure 6.7: Coverage of LAMA+ST portfolio planner with varying time allocation.

Chapter 7

Adding Exploration for Better Quality Solutions

Chapter 4, 5 and 6 have discussed how to use exploration to improve the coverage (number of problems solved) of GBFS. This chapter shows that exploration can also be used for generating better quality solutions.

- *F. Xie, R. Valenzano and M. Müller. Better Time Constrained Search via Randomization and postprocessing. In Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS-2013), pages 269-277, 2013* [67].

7.1 Introduction

Since IPC-2008, the satisficing planning community has been using the IPC scoring function to evaluate planners. This function emphasizes both plan quality and coverage simultaneously. Many satisficing planners such as LAMA [46] and Fast Downward [19] use an *anytime approach*: they attempt to quickly find an initial plan of possibly low quality, then use the remaining time to improve upon this plan. *Post-processing*, as implemented in the ARAS system [39], is another plan quality improvement technique. This approach takes an existing valid plan as input and tries to improve it by removing unnecessary actions and by finding shortcuts with a local search. Another *post-processing* technique, discussed by Chrpa et al. [8], analyzes action dependencies and independencies in order to identify redundant actions or non-optimal sub-plans.

Originally, the ARAS post-processor was applied as the final step of the planning process, after the planner had completed or was considered unlikely to find a better plan. However, since post-processing systems decouple plan improvement from plan discovery, post-processors like ARAS can also be used in an anytime fashion: by running them with a relatively tight time or memory limit on *every* plan produced by an anytime planner. This approach has been used by several planners [41, 59, 66].

Tests with these planners, described below, show that there is a large variance in the amount of improvement achieved with post-processing. As such, a lower quality input plan can often yield a higher quality final plan through post-processing than an initially better quality input plan. This behaviour can be exploited by planning systems which use post-processing. Currently, most anytime satisficing planners use the cost of the best incumbent solution to bound their future search. This avoids wasting effort in areas of the state space that cannot *directly* lead to a better solution. However, such bounding greatly decreases the number and variety of plans that an anytime system finds and we will show that because of it, bounding can have a negative impact on performance, when using post-processing.

The main contributions of this chapter are as follows:

- Introduce the concept of *unproductive time*, which measures the amount of time after the best solution is found, to help explain the impact of bounding.
- Present evidence that bounding in an anytime system is detrimental when used in conjunction with a post-processing system.
- Develop the meta-algorithm Diverse Anytime Search (DAS), which uses restarting to generate a more diverse set of plans.
- Implement DAS in Fast Downward [19] and show that it leads to significant plan quality improvements for two planning systems: LAMA-2011 [46] and AEES [56].
- Show that the improvements from DAS and from post-processing are independent, and can even be synergetic: with LAMA-2011, the improvement

from using both techniques together is slightly larger than the sum of the improvements when applied individually.

The remainder of this chapter is organized as follows: after introducing the concept of *unproductive time* and measuring it in LAMA for IPC planning problems, the new meta-algorithm DAS is introduced. DAS is tested both with and without the ARAS post-processor on LAMA-2011 and AEES. The experimental results show strong improvements in plan quality on IPC-2008 and IPC-2011 domains.

7.2 Unproductive Time in Anytime Satisficing Planning

As mentioned above, most planners use an anytime strategy to improve solution quality over time. However, there has been little investigation into how much time is actually being used to find the final solution. Let *unproductive time* be defined as the amount of time remaining, out of the total time given, when the planner's best solution[1] is found. For example, if an anytime planner A finds its best solution on a planning instance B at 13 minutes given a 30 minute time limit, and A does not improve upon this plan in the remaining 17 minutes, then the unproductive time for planner A on problem B is 17 minutes. The amount of unproductive time can be used to evaluate how efficiently an anytime planner is using the given search time, since that unproductive time could be spent doing something more useful, such as plan post-processing. Below, we will show that one of the consequences of using bounding in an anytime system such as LAMA-2011 [46] is that it often leads to large amounts of unproductive time.

LAMA-2011 won the sequential satisficing track of the International Planning Competition in 2011 (IPC 2011) after, in its previous incarnation as LAMA-2008, it won the same track at IPC 2008. LAMA-2011 starts its search with two runs of greedy best-first search: first with a distance-to-go heuristic and then with a cost-to-go heuristic. Next, LAMA improves the quality of its solutions through the anytime procedure of Restarting Weighted A* (RWA*) [49]. This procedure starts a new

[1] the current best plan must not be the optimal plan.

WA* search with a lower weight w whenever a new best solution is found. Only cost-to-go heuristics are used in this phase.

Whenever a new best solution with cost C is found, this cost is used to *bound* the rest of the search. This means that only nodes with g-cost (cost of best known path to the node) less than C are added to the open list. This prunes states that cannot lead directly to a better solution than before. Figure 7.1 shows that this approach also leads to a very large fraction of unproductive time on IPC benchmarks. Among the total of 244 problems solved in IPC-2011 with an 1800 second (30 minute) time limit, in more than 45% (111) of the problems, LAMA-2011 is unproductive for more than 1700 seconds. Table 7.1 shows the amount of unproductive time separately for each IPC-2011 domain. In the four domains of 2011-barman, 2011-elevators, 2011-parcprinter and 2011-woodworking, unproductive time exceeds 90%. In these domains, the planner is able to quickly find an initial solution, but fails to improve upon it.

As a typical example, Table 2 shows the number of solutions and the amount of unproductive time for the 20 instances of 2011-elevators. With the exception of instance 04, LAMA-2011 finds only a single solution to each problem. This does not at all imply that the first solution found by LAMA-2011 is optimal. Subsequent postprocessing with ARAS yields improved solutions for these problems. In these cases, it is much more difficult to find a second solution when using cost-to-go heuristics and the bound from the first solution, than to generate the initial solution using distance-to-go heuristics with no bound.

7.3 Post-Processing with ARAS

Since even a strong planner such as LAMA-2011 spends the large majority of its execution time being unproductive, a natural question becomes: How to use this time more effectively?

One possibility is to feed the solutions found by a planner into a post-processing plan improvement system such as ARAS [39]. ARAS consists of two components. The first, Action Elimination (AE), examines the plan for unnecessary actions. This

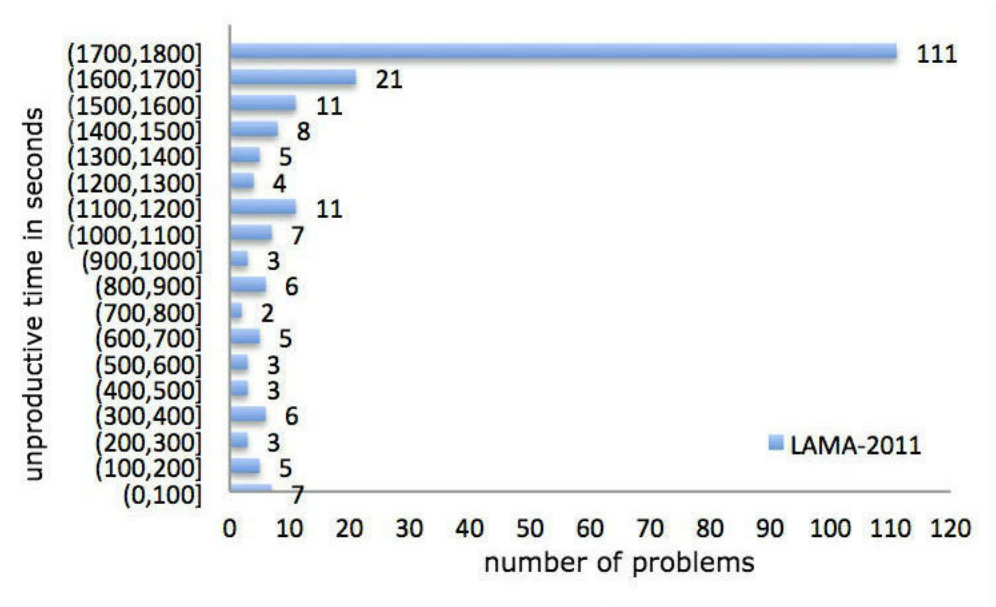


Figure 7.1: *Unproductive Time* of LAMA-2011 on IPC-2011.

Domain	solved	total time	UT	percentage
parcprinter	20	36000	35997	99.99%
barman	20	36000	35901	99.73%
woodworking	20	36000	35357	98.21%
elevators	20	36000	33890	94.14%
visitall	20	36000	31280	86.89%
pegsol	20	36000	31107	86.41%
scanalyzer	20	36000	29844	82.90%
nomystery	10	18000	14682	81.57%
transport	15	27000	20554	76.13%
openstacks	20	36000	24902	69.17%
floortile	6	10800	7032	65.11%
tidybot	16	28800	18099	62.84%
sokoban	19	34200	20377	59.58%
parking	18	32400	13948	43.05%

Table 7.1: *Unproductive Time* (UT) of LAMA-2011 on different IPC-2011 domains.

involves scanning the plan from front to back, removing each action and its dependent actions in turn, and testing if the resulting plan is still valid and reaches a goal. If so, the unnecessary actions are discarded and the scan continues on the resulting shorter plan. The second and main component of ARAS is Plan Neighbourhood Graph Search (PNGS). This technique builds a neighbourhood graph in the state space around the trajectory of the input plan by performing a breadth-first search

Instance	#s	UT	Instance	#s	UT
01	1	1799	11	1	1760
02	1	1798	12	1	1704
03	1	1797	13	1	1679
04	3	1794	14	1	1725
05	1	1799	15	1	1659
06	1	1796	16	1	1649
07	1	1798	17	1	1576
08	1	1791	18	1	1510
09	1	1789	19	1	1152
10	1	1783	20	1	1531

Table 7.2: Number of solutions (#s) and *Unproductive Time* (UT) of LAMA-2011 in the 20 instances of 2011-elevators.

to some depth d. Once this plan neighbourhood is constructed, a lowest-cost plan from the start to a goal state is extracted from this graph. Intuitively, the algorithm identifies local shortcuts along the path.

Typically, ARAS is run with a time and memory limit. Until the first of these limits is reached, AE and PNGS are alternated (starting with AE). PNGS is run iteratively, with an increasing depth bound. The best plan found within the limits is returned.

When ARAS was first introduced, it was used in a *process-once* fashion. This involved splitting the available planning time into two phases: plan finding and plan post-processing. In the first phase, a planner is used to find some plan (or a series of plans). When the plan finding phase is up, the best plan found so far is handed to the post-processor which is then allotted all of the remaining time. Several IPC-2011 planners [41, 59] used ARAS in an anytime fashion. This involves interleaving the plan finding and post-processing phase by using the post-processor in between iterations of an anytime planner. Each new plan found is immediately post-processed by ARAS. When ARAS hits one of its limits, it returns the best solution it has found and allows the anytime planner to continue looking for more plans.

There are two motivations for using ARAS in this way. First, for most anytime satisficing planners, it is difficult to determine if any new plans will be found during the remaining time or if the planner will be unproductive. Second, even if the

anytime planner can find a better plan with more search, there is no guarantee that post-processing the best plan found by an anytime planner will have a lower cost than the post-processed version of a weaker plan found earlier. The following experiment illustrates this. There are 151 problems from IPC-2011 for which LAMA-2011 generates more than one plan (without any assistance from a post-processor). If ARAS is run with a 2 GB memory limit on all these plans, then on 44 (29%) of these problems, the lowest cost solution is generated by post-processing an earlier plan, not by post-processing the final and best input plan found by LAMA-2011. The instance 2011-transport-07 is typical: LAMA-2011 generates two plans, one of cost 8396 and the other of cost 7222. Post-processing with ARAS improves the weaker input plan to 6379, and the stronger one to 6402. The weaker initial plan leads to the better final result.

This suggests that generating a greater diversity of input plans can be important for post-processing. If the anytime planner is only able to generate a small number of similar plans, then ARAS can only search within a very restricted set of neighbourhoods. This likely means that all its output plans will be similar, and of similar quality. When ARAS is handed a larger and more diverse set of input plans, it has a greater chance of finding significant improvements for at least one of them.

7.4 The Diverse Anytime Search Meta-Algorithm

The *Diverse Anytime Search (DAS)* meta-algorithm, shown in Algorithm 10, uses restarts with no bounds and post-processing in order to improve a given anytime planner. DAS divides the given total planning time into N equal time segments, where N is a user-supplied constant. In the first segment, the anytime planner P is run normally, except that ARAS is used to post-process each plan generated by P. At any time during this first segment, P can use its best plan found so far, *excluding* post-processing, to bound its search.

When time runs out on the first time segment, the best plan found so far, *including* post-processing, is saved. The anytime planner is restarted without any knowledge of previous solutions, and with a planner-specific randomization which will

vary the planner's search. At the end of each search segment, the best overall plan is updated, and P restarts from scratch with a new random seed.

As the early, greedier iterations of P typically find plans much more quickly and frequently, by restarting from scratch (and thus performing these greedy iterations again), DAS increases the number of plans available to the post-processor. This increases the number of opportunities for finding a strong improvement. **In our experiments, if P cannot find any solution by the end of the first time segment, then it is does not restart. This ensures that the coverage of the planners using DAS is the same as those that do not use it, and so we can directly focus on the impact that DAS and other anytime approaches have on plan quality.** These details are included in Algorithm 11. The algorithm uses two time limits: a soft time limit t for each segment, used for restarting if at least one solution was found, and a hard time limit T for the whole search. If in the first segment, P cannot find a solution within the soft time limit t, then restarts are disabled. In Algorithm 11, in the assignment $\langle p1, t1\rangle \leftarrow P.search(conf, bound, rand, time)$, *conf* is the current search configuration (such as the weight for RWA*), *bound* is the plan quality bound, and *rand* is the random seed. If the search finds a solution within time limit $time$, it returns the plan $p1$ and time used $t1$. Otherwise, the search terminates with $p1 = NULL$ and $t1 = time$.

The expression $\langle p2, t2\rangle \leftarrow PP.process(p1, time)$ indicates a call to the post-processor *PP* with $p1$ as the input plan. The post-processor *PP* returns when it either reaches the $time$ limit or a pre-set memory limit. When PP terminates because of the time limit, it returns the best plan found and $t2 = time$. When it terminates because of memory, it returns the best plan found and the time used. In each case, the best plan returned could still be the input plan $p1$ if no improvements are found.

7.5 Experiments

In this section, we describe five sets of experiments which show the utility of DAS and help enhance our understanding of this meta-algorithm. We begin with experiments that show that bounding can be harmful when used in conjunction with a

Algorithm 10 Diverse Anytime Search

Input Initial State I, goal condition G, given search time T, planner P, post-processor PP
Parameter N
Output A solution plan

$t \leftarrow T/N$
$\langle plan_{best}, cost_{best} \rangle \leftarrow \langle [], \infty \rangle$
for i from 1 to N **do**
  $rand \leftarrow generate\_random\_seed()$
  isSolved $\leftarrow i > 1$ {If the first iteration is successfully terminated, it means there is at least one solution found. }
  $plan \leftarrow AnytimeSearchPostprocess\ (I, G, T, t, rand, P, PP, isSolved)$
  if $cost(plan) < cost_{best}$ **then**
    $\langle plan_{best}, cost_{best} \rangle \leftarrow \langle plan, cost(plan) \rangle$
  end if
end for
return $plan_{best}$

post-processor. This is followed by a look at the performance of DAS without post-processing in LAMA-2011. We then experiment with combining DAS in LAMA-2011 with the ARAS system. The fourth set of experiments then looks at the impact of parameterization on DAS. Finally, we show that DAS is also effective when added to AEES.

All experiments in this section were run on an 8 core 2.8 GHz machine with a time limit of 30 minutes and memory limit of 4 GB per problem. Unless otherwise noted, the test set is made of all 550 problems from IPC-2008 and IPC-2011. Results for planners which use randomization are averaged over 5 runs (unless otherwise noted). All planners are implemented on the same version of the Fast Downward code base [19] and so the translation from PDDL to SAS+ is not included against the time limit since it is the same for all planners. For the first four experiments, the scores shown use the IPC metric with LAMA-2011's plans as the baseline. This means that if L is the best plan found by LAMA-2011 for a problem A, then the score of a given plan P for problem A is given by $cost(L)/cost(P)$. A planner can get > 1 score for problem A if it can find a better plan than L. For the post-processor, we use the ARAS system with a 2 GB memory limit.

Algorithm 11 Anytime Search with Post-processing

Input Initial State I, goal condition G, hard timelimit T,
soft timelimit t, random seed $rand$, planner P,
post-processor PP, first solution found flag *solved*
Parameter N
Output A solution plan

$conf \leftarrow P.GetFirstConf()$
$bound \leftarrow \infty$
$plan_{best} \leftarrow []$
$isSolved \leftarrow solved$
$totalTime \leftarrow 0$
restart $\leftarrow true$
while have time **do**
  if not $isSolved$ **or** not *restart* **then**
    $time \leftarrow T - totalTime$
    $\langle p1, t1 \rangle \leftarrow P.search(conf, bound, rand, time)$
    if not $isSolved$ **and** $t1 > t$ **then**
      restart $\leftarrow false$
    end if
  else
    $time \leftarrow t - totalTime$
    $\langle p1, t1 \rangle \leftarrow P.search(conf, bound, rand, time)$
  end if
  if $p1 == []$ **then**
    return []
  end if
  $isSolved \leftarrow true$
  $totalTime \leftarrow totalTime + t1$
  $conf \leftarrow P.nextConf(conf)$
  $bound \leftarrow cost(p1)$
  if *restart* **then**
    $time \leftarrow t - totalTime$
  else
    $time \leftarrow T - totalTime$
  end if
  $\langle p2, t2 \rangle \leftarrow PP.process(p1, time)$
  $totalTime \leftarrow totalTime + t2$
  if $cost(p2) < cost(plan_{best})$ **then**
    $plan_{best} \leftarrow p2$
  end if
end while
return $plan_{best}$

Domain	# of problems	LAMA-2011	LAMA-Aras	LAMA-Aras-B
barman	20	20	**23.99**	**23.99**
elevators	20	20	**26.01**	25.87
floortile	20	6	**6.77**	**6.77**
nomystery	20	**10**	**10.00**	9.83
openstacks	20	**20**	19.98	19.89
parcprinter	20	20	**20.10**	19.78
parking	20	18	**18.93**	17.46
pegsol	20	**20**	**20.00**	**20.00**
scanalyzer	20	20	**23.35**	21.26
sokoban	20	19	**20.23**	19.05
tidybot	20	16	**16.77**	**16.77**
transport	20	15	**17.70**	16.38
visitall	20	20	**20.45**	20.37
woodworking	20	20	**20.96**	20.85
Total	280	244	**265.24**	258.27

Table 7.3: Plan Quality of LAMA-2011, LAMA-Aras and LAMA-Aras-B on IPC-2011.

7.5.1 Experiment 1: Using Post-Processing and Bounding in LAMA-2011

Table 7.3 compares three planners on IPC-2011 domains:

- **LAMA-2011** is the IPC-2011 version of LAMA.
- **LAMA-Aras** is an implementation of *Diverse Anytime Search (DAS)* with the input planner being *LAMA-2011*, the input post-processor being ARAS, and *N=1*. This means that there are no restarts, and that the improved plans found by ARAS are *not* used for bounding, but LAMA-2011 *still* does its own bounding internally.
- **LAMA-Aras-B** is like LAMA-Aras except all plans, including the improved plans found by Anytime ARAS, are used to bound the subsequent iterations of WA*.

Table 7.3 shows that combining bounding with post-processing can be harmful, LAMA-Aras's plan quality scores are higher than LAMA-Aras-B's in almost all domains. Among the three planners, LAMA-Aras almost always gets the best plan quality score, except by a small margin in openstacks and tidybot. ARAS is known

to be ineffective on openstacks problems [39], and the time wasted running it causes a slight decrease in plan quality in that domain.

7.5.2 Experiment 2: DAS without Postprocessing

This section examines the impact of using Diverse Anytime Search in terms of unproductive time and the number of solutions it generates when used with LAMA-2011. These tests do not use any post-processing. They show that the new meta-algorithm increases the number of plans found and even improves solution quality in a number of domains.

When DAS is added to LAMA-2011, RWA* is being used within each time segment (as is bounding). When a time segment ends, RWA* starts again from scratch with a greedy best-first search iteration, though it does not bound using information from previous time segments. The source of diversity is *random operator ordering* [59]. This involves randomly shuffling the order of the generated set of successors of an expanded node before they are added to the open list. Random operator ordering affects the search by changing how ties are broken. **However, to ensure that competing algorithms have the same coverage, we use the default operator ordering during the first time segment.**

We refer to the new planner as **Diverse-LAMA(N)**. **N** is the parameter which affects the length of the time segments. Table 7.4 compares this planner, setting $N = 4$, with LAMA-2011 on the 2011-elevators domain. The new planner exhibits much less unproductive time[2]. In particular:

- The average number of plans increases from 1.1 to 4.3. On 17 of the problems, we see an increase from 1 plan with standard LAMA-2011 to 4 plans with Diverse-LAMA(4), since Diverse-LAMA(4) finds one plan per segment. In the cases of elev02 and elev03, Diverse-LAMA(4) sometimes finds more than 1 plan per segment, depending on the random seed, whereas there is a segment in which no plan is found on elev05.

- The amount of unproductive time decreases in all but one instance (elev11),

[2]Here, we show only one run instead of the average over 5 runs.

often drastically. The problem elev11 is the only exception as **Diverse-LAMA(4)** is unable to improve the plan it finds during the first segment when it finds the same plan as LAMA-2011. However, in all other problems there was at least one plan found in a later segment that was better than the first plan found.

Due to LAMA-2011's high amount of unproductive time in this domain, Diverse-LAMA(4) is also often able to find better solutions. This is because LAMA-2011 rarely finds a new solution after the first $1800/4 = 450$ seconds. In contrast, Diverse-LAMA(4) continues to find solutions by restarting and returning to a greedier search, some of which are better than the solutions found in the first 450 seconds.

Table 7.5 compares the plan quality of LAMA-2011 and Diverse-LAMA(4) on IPC-2011 domains and shows that this behaviour is also not limited to the elevators domain. In total, Diverse-LAMA(4) improves by a score of 6.1 though this improvement is not uniform over all domains. Instead, Diverse-LAMA(4) improves its solution quality over LAMA-2011 in 8 domains, while it is worse in 5 domains. These improvements are mainly made in domains in which LAMA-2011 has a high percentage of *unproductive time* for the same reasons as was the case in the elevators domain. However, in those domains in which LAMA-2011 is more productive later in the search, the restarts prevent Diverse-LAMA(4) from following through on one search long enough to find the best solutions. This is more apparent in Table 7.6, which shows the number of problems on which each of LAMA-2011 and Diverse-LAMA(4) found the best plan. In those domains in which LAMA-2011 is mostly unproductive, Diverse-LAMA(4) rarely generates worse final solutions, while for those domains in which LAMA-2011 is more productive later on — such as Floortile, Tidybot, Sokoban and Parking — Diverse-LAMA(4) will occasionally find weaker plans.

7.5.3 Experiment 3: Combining DAS with ARAS

This section tests DAS when used with LAMA and ARAS. The system is denoted as Diverse-LAMA-Aras(N) The four planners **LAMA-2011**, **LAMA-2011-Aras**[3],

[3]LAMA-2011-Aras is equivalent to Diverse-LAMA-Aras(1).

Instance	#s1	UT1	#s2	UT2	Instance	#s1	UT1	#s2	UT2
elev01	1	1799	4	1350	elev11	1	1760	4	1762
elev02	1	1798	6	330	elev12	1	1704	4	863
elev03	1	1797	4	1349	elev13	1	1679	4	354
elev04	3	1794	9	835	elev14	1	1725	4	793
elev05	1	1799	3	900	elev15	1	1659	4	713
elev06	1	1796	4	446	elev16	1	1649	4	795
elev07	1	1798	4	448	elev17	1	1576	4	1262
elev08	1	1791	4	895	elev18	1	1510	4	294
elev09	1	1789	4	1342	elev19	1	1152	4	1009
elev10	1	1783	4	440	elev20	1	1531	4	296

Table 7.4: Number of solutions and *Unproductive Time* of LAMA-2011 (#s1 and UT1) and Diverse-LAMA(4) (#s2 and UT2) in the 20 instances of 2011-elevators.

domain	UT	LAMA-2011	Diverse-LAMA(4)
2011-parcprinter	99.99%	20	**20.08**
2011-barman	99.73%	20	**21.76**
2011-woodworking	98.21%	20	**20.48**
2011-elevators	94.14%	20	**25.20**
2011-visitall	86.89%	20	**20.10**
2011-pegsol	86.41%	**20**	19.79
2011-scanalyzer	82.90%	20	**20.75**
2011-nomystery	81.57%	**10**	**10**
2011-transport	76.13%	15	**15.69**
2011-openstacks	69.17%	20	**20.22**
2011-floortile	65.11%	**6**	5.05
2011-tidybot	62.84%	**16**	15.30
2011-sokoban	59.58%	**19**	18.59
2011-parking	43.05%	**18**	17.21
total		244	**250.10**

Table 7.5: Plan Quality of LAMA-2011, Diverse-LAMA(4) on IPC-2011. Domains are sorted by decreasing fraction of *Unproductive time* (**UT**) shown in Table 7.1.

Diverse-LAMA(4), and **Diverse-LAMA-Aras(4)** are tested on all 550 problems from IPC-2008 and IPC-2011.

Table 7.7 shows a comparison of the plan quality of these planners in each of the domains tested. Diverse-LAMA-Aras(4) is the best (or tied for best) in 18 of the 23 domains and achieves the highest overall score, improving over the baseline planner LAMA-2011 by **59** units.

domain	UP	better	worse	total
2011-parcprinter	99.99%	**2**	0	20
2011-barman	99.73%	**19**	0	20
2011-woodworking	98.21%	**8**	0	20
2011-elevators	94.14%	**19**	0	20
2011-visitall	86.89%	**6**	3	20
2011-pegsol	86.41%	0	**1**	20
2011-scanalyzer	82.90%	**6**	2	20
2011-nomystery	81.57%	**0**	**0**	10
2011-transport	76.13%	**6**	0	15
2011-openstacks	69.17%	**4**	**4**	20
2011-floortile	65.11%	0	**2**	6
2011-tidybot	62.84%	2	**7**	16
2011-sokoban	59.58%	1	**4**	19
2011-parking	43.05%	**7**	4	18

Table 7.6: Plan Comparison between Diverse-LAMA(4) and LAMA-2011 on different domains. The columns **better** indicates in how many problems Diverse-LAMA(4) generates better plans than LAMA-2011 (**worse** means how many worse). Domains are ordered according to the percentages of *Unproductive time* (**UP**) shown in Table 7.1.

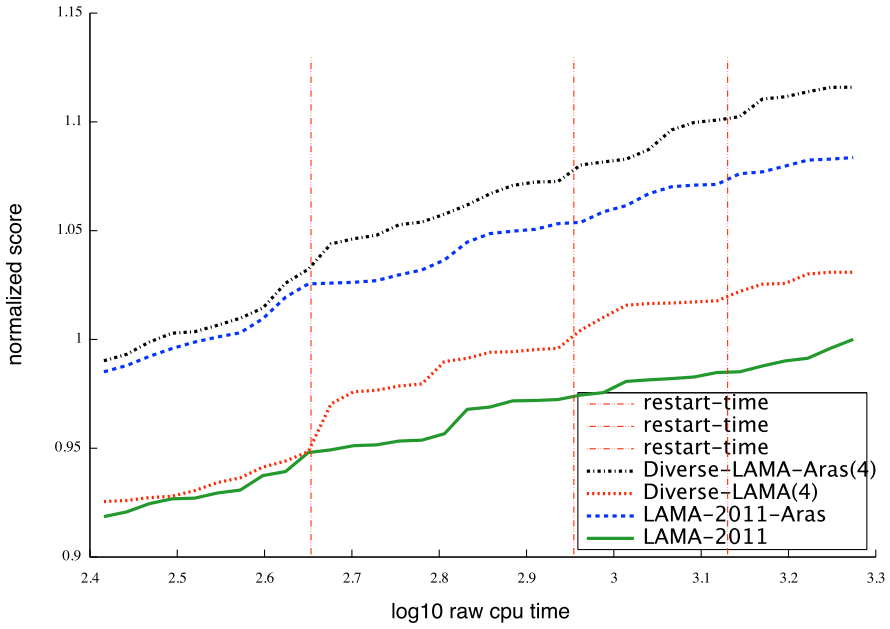


Figure 7.2: Normalized Score Curve of the 4 tested planners.

Figure 7.2 shows the normalized score [4] curve over 30 minutes of the 4 tested

[4]It equals the current score divided by 511, which is the final score by LAMA-2011 as shown in Table 7.7.

domain	# of problems	LAMA	DL(4)	LAMA-Aras	DL-Aras(4)
08-cybersec	30	**30**	**30.00**	**30.00**	**30.00**
08-elevators	30	30	35.82	38.50	**43.34**
08-openstacks	30	30	30.25	29.97	**30.35**
08-parcprinter	30	30	30.00	30.09	**30.10**
08-pegsol	30	**30**	29.78	**30.00**	**30.00**
08-scanalyzer	30	30	31.35	34.00	**34.18**
08-sokoban	30	**28**	27.15	27.66	27.48
08-transport	30	29	31.51	35.14	**36.73**
08-woodworking	30	30	30.92	33.10	**34.28**
11-barman	20	20	21.76	23.99	**24.20**
11-elevators	20	20	25.20	26.01	**31.16**
11-floortile	20	6	5.01	**6.77**	**6.77**
11-nomystery	20	**10**	**10.00**	**10.00**	9.89
11-openstacks	20	20	**20.22**	19.98	20.11
11-parcprinter	20	20	20.08	**20.10**	20.05
11-parking	20	18	17.21	18.93	**19.54**
11-pegsol	20	20	19.79	20.00	**20.01**
11-scanalyzer	20	20	20.75	23.35	**23.46**
11-sokoban	20	19	18.59	20.23	**20.54**
11-tidybot	20	16	15.21	**16.77**	16.17
11-transport	20	15	15.69	17.70	**19.68**
11-visitall	20	20	20.10	20.45	**20.53**
11-woodworking	20	20	20.48	20.96	**21.78**
total	550	511	526.89	553.69	**570.35**

Table 7.7: Plan Quality of LAMA-2011 (LAMA), LAMA-2011-Aras (**LAMA-Aras**), Diverse-LAMA(4) (**DL(4)**) and Diverse-LAMA-Aras(4) (**DL-Aras(4)**) on all 550 problems from IPC-2008 and IPC-2011. Extra experiments data can be found: *www.cs.ualberta.ca/research/theses-publications/technical-reports/2013/TR13-02*.

planners over all test domains. The three vertical lines indicate the restart points for Diverse-LAMA(4) and Diverse-LAMA-Aras(4) of 450, 900 and 1350 seconds. Notice that the time axis is in log scale. Before the first restart, DAS and non-DAS versions of the same planner are nearly the same[5]. Immediately after the first restart, the DAS planners show a quick jump in solution quality. This is because for many problems, restarting allows the planner to find new, sometimes better solutions. A

[5]To fully utilize our computational resources, we run several processes simultaneously on a multi-core machine. While all versions use the same memory limit, the restarts cause DAS to use less memory. The resulting decrease in memory contention accounts for the small differences in planner performance.

similar but less pronounced jump is also visible after the second restart.

By producing more plans, Diverse-LAMA-Aras(4) also provides more input plans for ARAS. Compared to Diverse-LAMA-Aras(4), the improvement from the first restart is more pronounced in Diverse-LAMA(4). Using ARAS smooths out some of the variance in solution quality between different runs.

As shown by the previous two experiments, using either ARAS or DAS improves plan quality. In Table 7.8, we show that the performance improvements from these techniques are independent, and sometimes even synergetic. The score for each planning system is split into two components: the raw scores of the best plans produced by the planning system ignoring the impact of ARAS (*ie.* ARAS is being run, but the plans it outputs are not counted towards the score of the planner), and the independent contribution of ARAS when it is used in the planning system. For comparison, the scores of the baseline planners (*ie.* those that do not run ARAS at all) are also shown. The raw scores are slightly worse than the baseline scores, since the planner producing the raw scores uses less time for the main search because of using ARAS (though it is not counted in the score). The improvement of Aras over Diverse-LAMA(4) is slightly larger than the improvement of Aras over LAMA-2011. This demonstrates that the improvements from Diverse-LAMA(4) finding substantially different overall plans seems to be largely independent from the local plan improvements found by Aras. The following two examples help to explain this behaviour:

- In 2011-floortile 05, the best raw plan generated by Diverse-LAMA(4) has cost 132, while LAMA-2011 can find a cost 63 plan. ARAS can improve the cost 132 plan to a cost of 63 as well. This suggests that ARAS can help DAS in cases where restarting prevents the search from running long enough to find good plans.

- In 2011-woodworking 01, plans of cost 1600, 1630 and 1620 are produced in time segments 1, 3 and 4, while LAMA-2011 only finds one solution of cost 1600. ARAS improves the three solutions as follows: $1600 \rightarrow 1460$, $1630 \rightarrow 1290$ and $1620 \rightarrow 1380$. The worst input plan turns out to be the

best after post-processing, while the best input plan becomes the worst after post-processing. Out of the 244 problem instances solved by LAMA-2011, in 44 cases the best final plan produced from Anytime Aras does not come from LAMA's best plan. In Diverse-LAMA-Aras(4), this ratio increases dramatically, to **86/244 problems**. This demonstrates how increased plan diversity from DAS can improve overall performance.

domain	LAMA	DL(4)	LAMA-Aras			DL-ARAS(4)		
			raw$_{LA}$	Δ_{ARAS}	Final	raw$_{DL}$	Δ_{ARAS}	Final
barman	20.00	21.76	20.00	3.99	23.99	21.70	2.51	24.20
elevators	20.00	25.20	20.21	5.80	26.01	24.82	6.35	31.16
floortile	6.00	5.01	5.21	1.55	6.77	4.67	2.10	6.77
nomystery	10.00	10.00	10.00	0.00	10.00	9.86	0.03	9.89
openstacks	20.00	20.22	19.98	0.00	19.98	19.94	0.17	20.11
parcprinter	20.00	20.08	20.00	0.10	20.10	20.00	0.05	20.05
parking	18.00	17.21	16.84	2.09	18.93	16.33	3.21	19.54
pegsol	20.00	19.79	19.57	0.43	20.00	17.65	2.36	20.01
scanalyzer	20.00	20.75	18.89	4.46	23.35	20.39	3.07	23.46
sokoban	19.00	18.59	16.58	3.65	20.23	16.10	4.44	20.54
tidybot	16.00	15.21	15.15	1.62	16.77	14.75	1.42	16.17
transport	15.00	15.69	14.00	3.70	17.70	16.32	3.37	19.68
visitall	20.00	20.10	20.00	0.45	20.45	20.06	0.47	20.53
woodwork	20.00	20.48	20.00	0.96	20.96	20.48	1.30	21.78
Total	244.0	250.1	236.44	28.80	265.24	243.06	30.84	273.90

Table 7.8: Combined effect of DAS and post-processing in IPC-2011 domains.

7.5.4 Experiment 4: Testing DAS with Different Numbers of Segments

Recall that DAS is parameterized by the number of segments, N, for which it runs, with $N = 1$ corresponding to LAMA-ARAS, and $N = 4$ to the algorithm used in the previous experiments. Figure 7.3 shows how the behaviour of this meta-algorithm changes when varying N in the range from 1 to 120 on IPC 2011 domains. Overall, the plan quality score differences are small, with the best results for N from 3 to 6. For $N \leq 3$, plan quality increases with N, taking advantage of the diversity and number of plans generated. For $N > 6$, the solution quality slowly

decreases as the runtime for both LAMA and ARAS becomes ever shorter.[6] The trade-off is that a too large N does not leave the planner enough time to find high quality input plans, while small N hurt diversity.

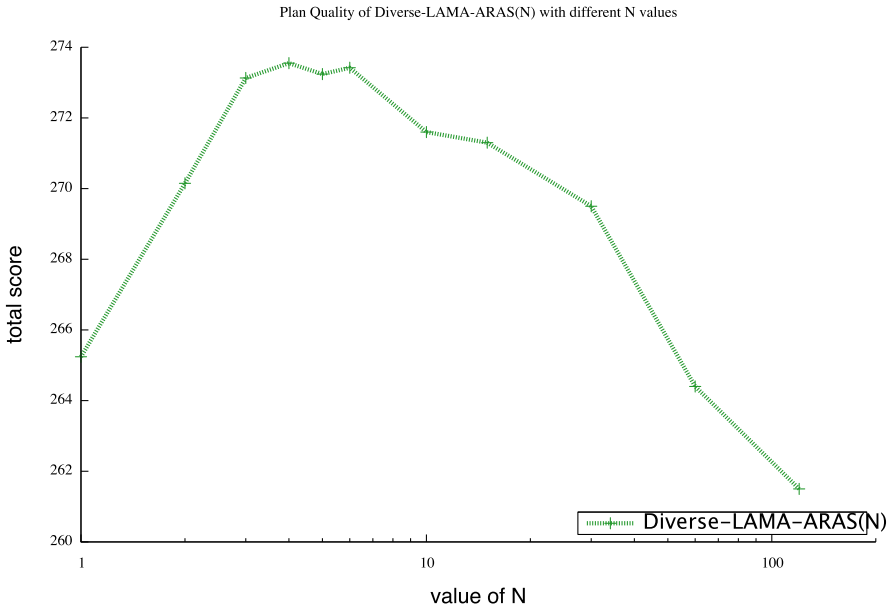


Figure 7.3: Plan Quality of Diverse-LAMA-Aras(N) with different N values. The x-axis represents the value of N. The y-axis represents the quality score.

7.5.5 Experiment 5: Combining DAS with Anytime Explicit Estimation Search

Anytime explicit estimation search (AEES) is an anytime search algorithm introduced by Thayer et al. [56]. AEES uses explicit estimation search (EES) [55] as its main search component. EES is a suboptimal search algorithm which is able to use both inadmissible and admissible heuristics while still satisfying a given solution cost bound. It does so by focusing its search on nodes that the inadmissible heuristic estimates will lead to solutions that are within the bound. It has been shown to be particularly effective in domains with non-unit action costs due to its ability to use both distance-to-go and cost-to-go heuristics [55]. AEES is the anytime version of EES, which lowers the suboptimality bound whenever a new best solution is found,

[6]The time of each time segment is given by T/N, where T is the total time (30 minutes in our experiments).

by using the latest solution as the new bound. The AEES algorithm's goal is to minimize the time between solutions, and generate more solutions. This makes it a good test case for DAS.

This section tests AEES on IPC-2008 and IPC-2011 domains. DAS with AEES is configured as follows: it uses the two planning-specific enhancements of deferred evaluation and preferred operators, and the three heuristics Landmark-cut (admissible cost-to-go heuristic) [20], FF-cost (inadmissible cost-to-go heuristic) and FF-distance (distance-to-go heuristic) [25]. The scores shown use the IPC metric with AEES as a baseline. If L is the plan computed by AEES, then the score of a given plan P is calculated by $cost(L)/cost(P)$. The experimental results are shown in Table 7.9. Similar to the LAMA-2011 experiments, Diverse-AEES-ARAS(4) gets the highest score, improving the baseline planner AEES by 73.7 units from 440 to 513.7, and achieving the best score in 14 of 23 domains.

7.6 Chapter Summary

This chapter has shown that the search performance of LAMA-2011 suffers from a large amount of *unproductive time*, time which can be used in other ways such as post-processing. The new meta-algorithm of Diverse Anytime Search tries to utilize this unproductive time with randomized restarts to generate a larger and more diverse set of plans for a post-processing system such as ARAS to improve upon. Experimental results show that the new framework leads to substantial improvements on IPC-2008 and IPC-2011 domains, for both LAMA-2011 and the AEES algorithm. The best parameter N for DAS depends on factors such as the planning domain, randomizing method and search algorithm. However, in the experiments the performance was robust for small values of N between 3 and 6.

domain	# of problems	AEES	DE(4)	AEES-Aras	DE-Aras(4)
08-cybersec	30	29	31.20	29.00	**32.36**
08-elevators	30	30	33.80	41.67	**45.20**
08-openstacks	30	30	**31.35**	30.00	31.15
08-parcprinter	30	25	25.16	25.70	**25.82**
08-pegsol	30	30	29.96	**30.11**	30.05
08-scanalyzer	30	30	30.53	**34.35**	34.16
08-sokoban	30	**27**	26.47	26.71	26.73
08-transport	30	28	31.09	38.43	**40.86**
08-woodworking	30	20	20.25	21.14	**21.32**
11-barman	20	20	20.88	22.74	**22.85**
11-elevators	20	19	23.03	25.03	**28.06**
11-floortile	20	**6**	5.50	**6.00**	**6.00**
11-nomystery	20	**10**	9.94	**10.00**	9.91
11-openstacks	20	20	20.92	20.00	**21.00**
11-parcprinter	20	11	11.14	11.88	**11.92**
11-parking	20	15	15.15	16.61	**18.40**
11-pegsol	20	20	19.92	20.11	20.04
11-scanalyzer	20	20	21.00	**24.73**	24.43
11-sokoban	20	**17**	16.73	16.60	16.96
11-tidybot	20	13	13.56	**16.21**	15.41
11-transport	20	13	14.60	18.22	**19.51**
11-visitall	20	3	3.32	7.30	**7.39**
11-woodworking	20	4	3.98	4.12	**4.13**
total	550	440	459.50	496.67	**513.67**

Table 7.9: Plan Quality of AEES, Diverse-AEES(4) (**DE(4)**), AEES-Aras, and Diverse-AEES-Aras(4) (**DE-Aras(4)**) on all 550 problems from IPC-2008 and IPC-2011.

Chapter 8

The Jasper Planner: Combining Exploration in Greedy Best First Search

Chapters 3 to 7 propose techniques that improve the performance of GBFS in both coverage and plan quality. This chapter integrates these techniques and builds the new satisficing planner *Jasper*. This chapter is based on the following publication:

- *F. Xie, M. Müller and R. Holte. Jasper: the Art of Exploration in Greedy Best First Search. In M. Vallati, L. Chrpa, L. and T. McCluskey, The Eighth International Planning Competition, University of Huddersfield, 39–42, 2014* [69].

8.1 The Jasper Planner

Jasper is a satisficing planner that builds on LAMA-2011 [47]. It adds two modifications. First, it replaces the GBFS algorithm in LAMA-2011 with an improved GBFS variant, *Type Exploration based Greedy Best-First Search with Local Search (Type-GBFS-LS)*. GBFS-LS (see Chapter 4) and Type-GBFS (see Chapter 6) are designed for two different problems in GBFS. Jasper applies both enhancements to GBFS. Like GBFS-LS, Type-GBFS-LS uses a local search when the global search gets stuck. However, Type-GBFS-LS replaces GBFS with Type-GBFS in both the global level search and the local level search. Type-GBFS-LS is an improved version of GBFS that is less sensitive to flaws in heuristic functions. Second, it imple-

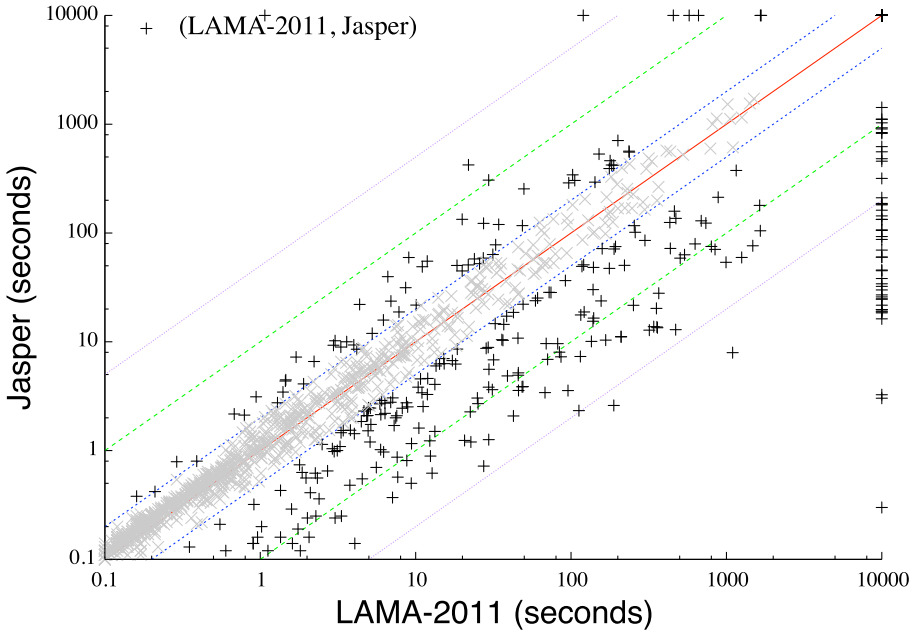


Figure 8.1: Comparison of search time: LAMA-2011 vs. Jasper.

ments the DAS (see Chapter 7) system for solution improvement, which takes the modified LAMA-2011 as the anytime planner and ARAS [39] as the post-processing system, as shown in Chapter 7.

8.2 Experiments on the First 7 IPC domains

Experiments were run on a set of 2112 problems in 54 domains from the first seven International Planning Competitions, using one core of a 2.8 GHz machine with 4 GB memory and 30 minutes per instance. Results for planners which use randomization are averaged over five runs.

The performance comparison in this section includes the following planners: LAMA-2011, LAMA-LS (from Chapter 4), Type-LAMA (from Chapter 6), and Jasper.

Table 8.1 shows the coverage results for the four planners. All three new planners get better results than LAMA-2011, with the best result of 1953.0 for Jasper.

Figures 8.1, 8.2 and 8.3 compare each planner with Jasper in terms of their search time of finding the first solution in *IPC time comparison setting* (see Chapter 4 Section 4.4.2). Jasper shows a clear improvement over the other 3 planners in

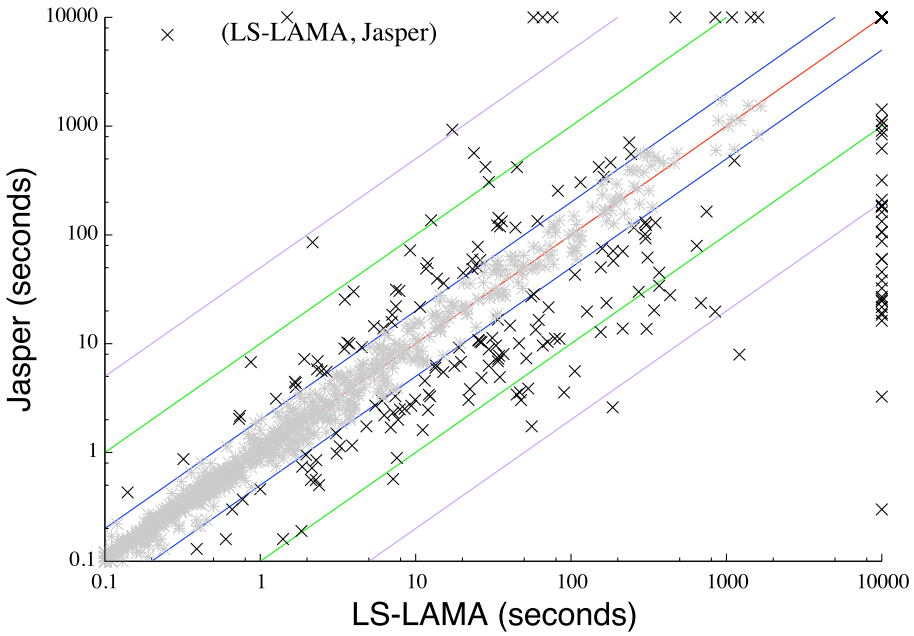


Figure 8.2: Comparison of search time: LAMA-LS vs. Jasper.

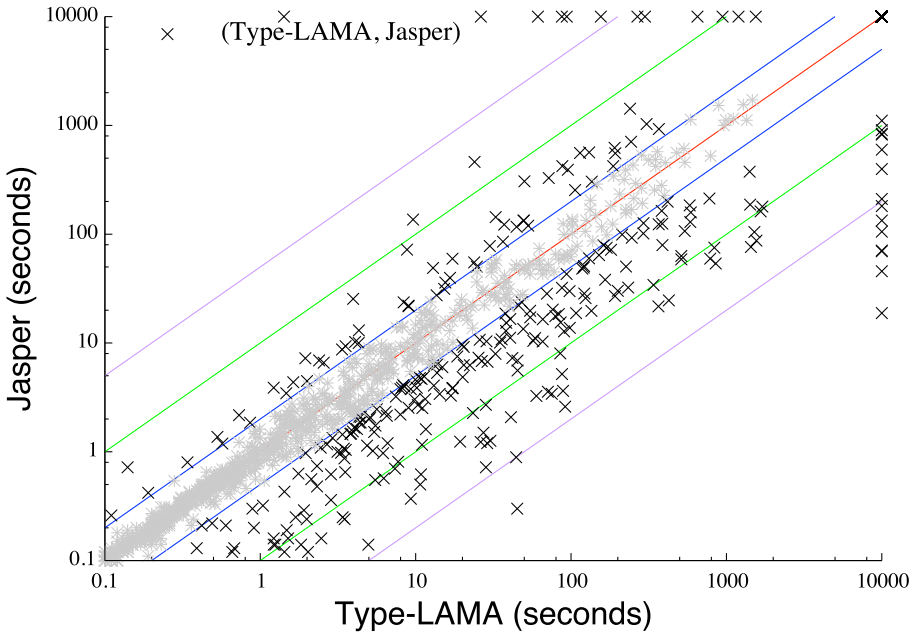


Figure 8.3: Comparison of search time: Type-LAMA and Jasper.

terms of speed. It solves more problems than LAMA-LS. Besides its advantage in coverage, it wins the time comparison with Type-LAMA for a larger number of instances by factors $2\times$ and $10\times$.

Planner	LAMA-2011	LAMA-LS	Type-LAMA	Jasper
Coverage	1913	1931	1949.8	**1953.0**

Table 8.1: IPC coverage out of 2112.

8.3 Experiments on IPC 2014

Jasper is developed and tuned towards the first 7 IPC benchmarks. In order to show its robustness in new domains, this section compares Jasper with LAMA-2011 over the IPC-2014 domains.

8.3.1 Overview of IPC-2014 Sequential Satisficing Track

There are 14 domains and 280 problems (20 problems each domain) in the Sequential Satisficing track of IPC-2014. The score of a plan is computed according the IPC metric with best found solution as the baseline. This means that if L_{best} is the plan with minimum cost found by all planners, then the score of a given plan P is given by $cost(L_{best})/cost(P)$. The score is 0 if no plan is found. A planner's score is computed by summing up scores of their best solutions in each problem.

There are 20 planners attending the Sequential Satisficing track. Jasper ranked 4th (considering both coverage and plan quality) behind IBaCoP2/IBaCoP [6], Mercury [28] and MIPlan [43]. Portfolio based planners achieved remarkable success in this competition. Mercury and Jasper are the only two not portfolio based planners that ranked in top 10. Jasper solved the second largest number of problems only behind IBaCoP2/IBaCoP [6], which is a powerful planner based on 28 different planners.

8.3.2 Comparing Coverage and Plan Quality with LAMA-2011

Experiments were run on all 280 problems from IPC-2014, using one core of a 2.8 GHz machine with 4 GB memory and 30 minutes per instance. Results for Jasper

which use randomization are averaged over five runs.

		Coverage		Plan Quality	
Domain	# of ins	LAMA-2011	Jasper	LAMA-2011	Jasper
2014-Barman	20	**20**	**20**	14.84	**19.54**
2014-CaveDiving	20	**7**	**7**	**7.00**	**7.00**
2014-Childsnack	20	0	0	0.00	0.00
2014-CityCar	20	4	**10**	3.29	**9.92**
2014-Floortile	20	**2**	**2**	1.96	**2.00**
2014-GED	20	**20**	**20**	17.59	**18.27**
2014-Hiking	20	17	**19.6**	14.17	**18.27**
2014-Maintenance	20	7	**11**	6.07	**10.88**
2014-Openstacks	20	19	**19.6**	17.36	**18.62**
2014-Parking	20	12	**19.8**	10.05	**18.19**
2014-Tetris	20	4	**7.2**	3.34	**4.42**
2014-Thoughtful	20	**17**	**17**	16.02	**16.68**
2014-Transport	20	**7**	**10**	4.69	**9.58**
2014-Visitall	20	**20**	**20**	19.14	**19.93**
Total	280	156	**180.8**	135.52	**173.30**

Table 8.2: Coverage and quality score for LAMA-2011 and Jasper.

The first two columns in Table 8.2 compare the coverage result between LAMA-2011 and Jasper. Jasper solved 180.8 (average over 5 runs) problems, 24.8 more than LAMA-2011. Jasper solved 182, 180, 180, 180 and 182 problems in 5 different runs respectively.

Figure 8.4 compares the time usage of LAMA-2011 and Jasper over the IPC-2014 benchmarks in *IPC time comparison setting* (see Chapter 4 Section 4.4.2). Jasper shows substantial improvements in search time over LAMA-2011.

The last two columns in Table 8.2 compare LAMA-2011 and Jasper in IPC scores. Jasper's IPC scores are quite stable in 5 runs. It achieved 174.39, 173.10, 173.31, 172.25 and 173.45 in 5 different runs respectively. Jasper only solved 24.8 more problems than LAMA-2011, however, its IPC score is 37.7 (average over 5 runs) higher than LAMA-2011. Jasper not only solved more problems but also produced better solutions because of the Diverse Anytime Search framework.

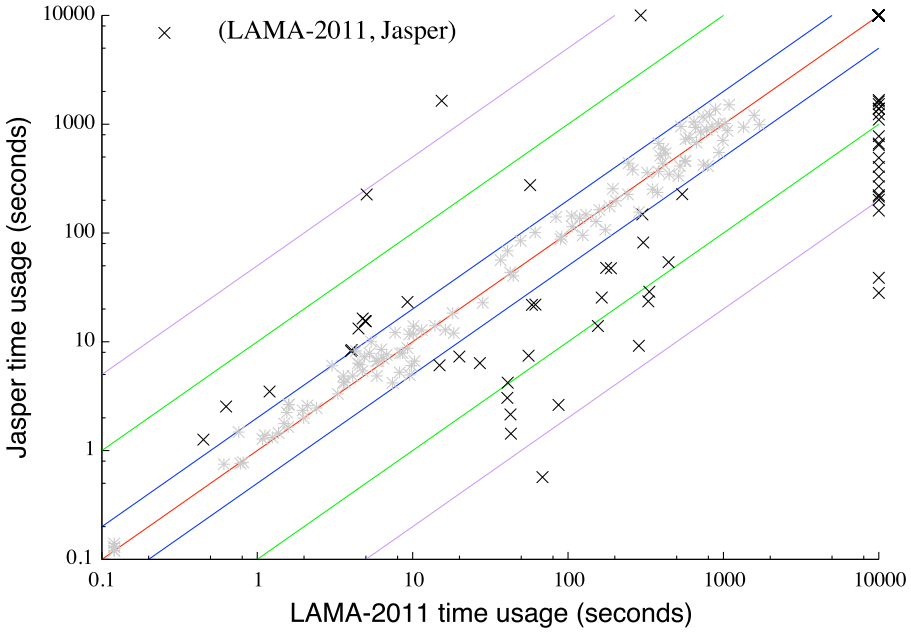


Figure 8.4: Comparison of search time: LAMA-2011 vs. Jasper on IPC-2014 benchmarks.

8.4 Chapter Summary

This chapter describes the new planner Jasper, which integrates techniques developed in this dissertation into LAMA-2011. It compares LAMA-2011 and Jasper on both the first 7 IPC benchmarks and the latest IPC-214 benchmarks. Jasper shows a substantial improvement over LAMA-2011 in both coverage and plan quality.

Chapter 9

Conclusion

This chapter summarizes the contributions and lessons we learned during this dissertation, followed by a discussion on the limitations and possible future work in this line of research.

9.1 Contributions and Lessons

AI Planning and Heuristic Search are fundamental research areas for Artificial Intelligence. Heuristic Search is proven to be one of the most effective methods in solving planning problems. Despite the success in this direction, flawed heuristics are still some of the main bottlenecks that prevents heuristic search from scaling to harder problems.

This dissertation proposed exploration as a solution to reduce the influence of flaws in heuristics. It investigates how exploration works from four different aspects, and develops a strong planner based on these techniques:

- **Investigating the scalability of current satisficing planners in large IPC domains**: In Chapter 3, we show the limit of the current satisficing planners by generating a set of large problems from IPC domains. We also demonstrate the combination of local GBFS and random walks, which do not evaluate intermediate nodes, can scale to much larger planning instances.

- **Understanding and using local exploration in GBFS**: Chapters 4 and 5 introduce the Greedy Best First Search with Local Exploration (GBFS-LE) framework along with two concrete algorithms GBFS-LS and GBFS-LRW

to attack the UHR problem for GBFS. We also conduct a deep analysis on some instances of GBFS, which shows that GBFS often gets stuck in a big "virtual" UHR that consists of both small and big UHRs over the open list. Moreover, we show that the existence of barrier nodes is the main reason that GBFS fails badly in some problems, while GBFS-LS performs much better in this case.

- **Exploring the usage of type systems in adding exploration to GBFS**: Chapter 6 demonstrates that early mistakes can make GBFS grow the search tree deep into a region of state space without any easy-to-find solutions. In the case of early mistakes, the nodes on some solution path that do exist in the open list are not open until very late because they have high heuristic values. Moreover, we show that the straightforward exploration by selecting a node from the open list uniformly at random like ϵ-GBFS has some limitations when the majority of the nodes in the open list might come from the bad region. The novel idea of type buckets is used to change the distribution of the nodes that Type-GBFS samples from to improve the quality of exploration.

- **Discovering unproductive time and reducing it by combining randomized restart and post-processing**: It is commonly accepted that in an anytime satisficing planner, all solutions with higher plan cost than the current best solution should be pruned. Chapter 7 shows that such a bounding strategy is harmful for plan improvement when post-processing is applied. We introduce the concept of *Unproductive Time*, and analyze it on LAMA-2011. Later, we propose the new meta algorithm of *Diverse Anytime Search*, which takes an anytime satisficing planner and a post-processing system to reduce the unproductive time and generates better quality solutions.

- **Developing a strong satisficing planner**: Chapter 8 introduces the new satisficing planner Jasper. It integrates both the local exploration from GBFS-LS and the global exploration from Type-GBFS into a single search algorithm. It also integrates Diverse Anytime Search for better quality solutions. It ranked the 4th out of 21 planners in the deterministic satisficing track of IPC-2014,

and solved the largest number of problems among all non-portfolio-based planners.

There are two major lessons that I learned during my past 5 years. First, balancing between exploration and exploitation is important for overcoming the problems caused by flaws in heuristics. Second, deep analysis of specific instances can lead to algorithm improvement in general[1]. The second lesson is the fundamental methodology behind much of my work, which keeps evolving the process. In my first few years in PhD, guess-and-try manner was my major strategy. For example, during building Arvand-LS, I first asked the question "can we combine systematic search and random walks?" and then I conducted different combinations and reported the one that worked best. In later years, more analysis on specific instances provided solid reasoning on why and how the proposed techniques will work. For example, Chapter 5 illustrates that small UHRs take up a sufficiently large part of the open list, leading to the idea of breaking a large local exploration into multiple smaller ones. Chapter 7 analyzes unproductive time and the relationship between an input raw solution and an output improved solution, which leads to the idea of randomization and multiple restarts in that chapter.

9.2 Challenge and Future Work

This section discusses some of the challenges of the current research and proposes interesting directions for future work:

- **Combining Greedy Best First Search and Stochastic Random Walk:** One remaining weakness of Arvand-LS is in problems that require exhaustive search of large regions of the state space, such as Sokoban and Parking. Since portfolio based planners have achieved remarkable results [6, 59], building a portfolio planner that contains Arvand-LS as well as other planners such as LAMA-2011, which performs well in such problems, is an interesting direction.

[1]This sentence is partially borrowed from one AAAI review. The original sentence is "This work is an excellent example of directed analysis leading to algorithm improvement...".

- **Adding and Understanding Local Exploration in GBFS:** Determining whether the *Multiple Uninformative Heuristic Region* problem occurs in classical heuristic search domains, such as sliding tile puzzles [34], is an interesting research direction. Valenzano et al. [60] show that replacing preferred operators with random actions can achieve about half the improvement of preferred operators. Similarly, replacing the secondary heuristic in multiple heuristics with a purely random heuristic achieves about half the improvement of multiple heuristics. The *Multiple Uninformative Heuristic Region* problem might be a contributing cause of these two phenomena.

- **Adding Global Exploration with Type Buckets to Greedy Best First Search:** One potential problem of Type-GBFS is that the type system might not be able to explore deeply enough when the distance from current nodes in the open list to heuristically promising nodes is large. One potential solution for this problem is to use a larger local search. As I show in Chapter 8, combining Type-GBFS with local GBFS performs better than Type-GBFS itself. However, more research,on how to combine global level and local level exploration properly, is needed.

- **Adding Exploration for Better Quality Solutions**: The Diverse Anytime Search framework introduces solution diversity by randomizing operators. However, it is detected in many problems that the neighborhood graphs of "different" input plans have a large degree of overlap. Finding more diverse plans, such as Srivastava et al.'s work [54], can be a promising direction to explore in the future.

- **The Jasper planner**: Jasper has shown its strong performance in IPC-2014. It solved the second largest number of problems in the competition. It solved more problems than many portfolio base planners, only behind IBaCoP2/IBaCoP [6]. It naturally raises the question of whether we can include Jasper in a portfolio planner to further improve the coverage?

Bibliography

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002. doi: 10.1023/A:1013689704352. URL http://dx.doi.org/10.1023/A:1013689704352.

[2] C. Bäckström and B. Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11:625–656, 1995. doi: 10.1111/j.1467-8640.1995.tb00052.x.

[3] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artif. Intell.*, 90(1-2):281–300, 1997. doi: 10.1016/S0004-3702(96)00047-1. URL http://dx.doi.org/10.1016/S0004-3702(96)00047-1.

[4] B. Bonet and H. Geffner. Planning as heuristic search. *Journal Artificial Intelligence*, 129(1-2):5–33, 2001. doi: 10.1016/S0004-3702(01)00108-4.

[5] T. Bylander. The computational complexity of propositional STRIPS planning. *Artif. Intell.*, 69(1-2):165–204, 1994. doi: 10.1016/0004-3702(94)90081-7. URL http://dx.doi.org/10.1016/0004-3702(94)90081-7.

[6] I. Cenamor, T Rosa, and F. Fernández. IBACOP and IBACOP2 planner. In M. Vallati, L. Chrpa, and T. McCluskey, editors, *The Eighth International Planning Competition (IPC-2014)*, pages 35–38, 2014.

[7] P. Chen. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM J. Comput.*, 21(2):295–315, 1992.

[8] L. Chrpa, T. McCluskey, and H. Osborne. Optimizing plans through analysis of action dependencies and independencies. In Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet, editors, *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS-2012)*, 2012. ISBN 978-1-57735-562-5.

[9] A. Coles and A. Smith. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, pages 28:119–156, 2007.

[10] A. Coles, M. Fox, and A. Smith. A new local-search algorithm for forward-chaining planning. In Mark S. Boddy, Maria Fox, and Sylvie Thiébaux, editors, *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS-2007)*, pages 89–96, 2007.

[11] M. Enzenberger, M. Müller, B. Arneson, and R. Segal. Fuego - an open-source framework for board games and go engine based on monte carlo tree search. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):259–270, 2010. doi: 10.1109/TCIAIG.2010.2083662. URL http://dx.doi.org/10.1109/TCIAIG.2010.2083662.

[12] K. Erol, J. Hendler, and D. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 2.*, pages 1123–1128, 1994. URL http://www.aaai.org/Library/AAAI/1994/aaai94-173.php.

[13] A. Felner, S. Kraus, and R. E. Korf. KBFS: K-best-first search. *Ann. Math. Artif. Intell.*, 39(1-2):19–39, 2003.

[14] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971. doi: 10.1016/0004-3702(71)90010-5.

[15] Á. García-Olaya, S. Jiménez, and C. Linares López, editors. *The 2011 International Planning Competition*, 2011.

[16] A. Gerevini, A. Saetti, and I. Serina. Planning through stochastic local search and temporal action graphs in LPG. *Journal of Artificial Intelligence Research*, 20:239–290, 2003.

[17] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968. doi: 10.1109/TSSC.1968.300136. URL http://dx.doi.org/10.1109/TSSC.1968.300136.

[18] M. Helmert. A planning heuristic based on causal graph analysis. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-2004)*, pages 161–170, 2004.

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

[20] M. Helmert and C. Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-2009)*, 2009.

[21] M. Helmert and H. Geffner. Unifying the causal graph and additive heuristics. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric A. Hansen, editors, *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS-2008)*, pages 140–147, 2008. ISBN 978-1-57735-386-7.

[22] M. Helmert, G. Röger, and E. Karpas. Fast Downward Stone Soup: A baseline for building planner portfolios. In *Proceedings of the ICAPS-2011 Workshop on Planning and Learning (PAL)*, pages 28–35, 2011.

[23] J. Hoffmann. Where 'ignoring delete lists' works: Local search topology in planning benchmarks. *Journal of Artificial Intelligence Research*, 24:685–758, 2005.

[24] J. Hoffmann. Where ignoring delete lists works, part II: Causal graphs. In Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert, editors, *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS-2011)*, pages 98–105, 2011.

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

[26] T. Imai and A. 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, 2011.

[27] T. Ishida. Moving target search with intelligence. In *Proceedings of the 10th National Conference on Artificial Intelligence. San Jose, CA, July 12-16, 1992.*, pages 525–532, 1992. URL http://www.aaai.org/Library/AAAI/1992/aaai92-081.php.

[28] M. Katz and J. Hoffmann. Mercury planner: Pushing the limits of partial delete relaxation. In M. Vallati, L. Chrpa, and T. McCluskey, editors, *The Eighth International Planning Competition (IPC-2014)*, pages 43–47, 2014.

[29] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the tenth European Conference on Artificial Intelligence*, pages 359–363, 1992.

[30] T. Keller and P. Eyerich. PROST: probabilistic planning based on UCT. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*, 2012. URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4715.

[31] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, pages 282–293, 2006. doi: 10.1007/11871842_29. URL http://dx.doi.org/10.1007/11871842_29.

[32] S. Koenig and X. Sun. Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems*, 18(3):313–341, 2009. doi: 10.1007/s10458-008-9061-x. URL http://dx.doi.org/10.1007/s10458-008-9061-x.

[33] R. Korf. Real-time heuristic search. *Artif. Intell.*, 42(2-3):189–211, 1990. doi: 10.1016/0004-3702(90)90054-4. URL http://dx.doi.org/10.1016/0004-3702(90)90054-4.

[34] R. E. Korf and L. Taylor. Finding optimal solutions to the twenty-four puzzle. In William J. Clancey and Daniel S. Weld, editors, *The Thirteenth National Conference on Artificial Intelligence*, pages 1202–1207, 1996. ISBN 0-262-51091-X.

[35] L. Lelis, S. Zilles, and R. Holte. Stratified tree search: a novel suboptimal heuristic search algorithm. In *Proceeding of the 12th International Conference on Autonomous Agents and Multi-Agent Systems*, pages 555–562, 2013.

[36] N. Lipovetzky and H. Geffner. Searching for plans with carefully designed probes. In Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert, editors, *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS-2011)*, pages 154–161, 2011.

[37] Q. Lu, Y. Xu, R. Huang, and Y. Chen. The Roamer planner random-walk assisted best-first search. In Á. García-Olaya, S. Jiménez, and C. Linares López, editors, *The 2011 International Planning Competition*, pages 73–76, 2011.

[38] H. Nakhost and M. Müller. Monte-Carlo exploration for deterministic planning. In Walsh [63], pages 1766–1771. ISBN 978-1-57735-516-8.

[39] H. Nakhost and M. Müller. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In R. Brafman, H. Geffner, J. Hoffmann, and H. Kautz, editors, *Proceeedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS-2010)*, pages 121–128, Toronto, Canada, 2010. AAAI Press.

[40] H. Nakhost and M. Müller. A theoretical framework for studying random walk planning. In Daniel Borrajo, Ariel Felner, Richard E. Korf, Maxim Likhachev, Carlos Linares López, Wheeler Ruml, and Nathan R. Sturtevant, editors, *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SOCS-2012)*, pages 57–64, 2012.

[41] H. Nakhost, M. Müller, R. Valenzano, and F. Xie. Arvand: the art of random walks. In Á. García-Olaya, S. Jiménez, and C. Linares López, editors, *The 2011 International Planning Competition*, pages 15–16, 2011.

[42] H. Nakhost, J. Hoffmann, and M. Müller. Resource-constrained planning: A Monte Carlo random walk approach. In Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet, editors, *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS-2012)*, pages 181–189, 2012.

[43] S. Núñez, D. Borrajo, and C. López. MIPLAN and DPMPLAN. In M. Vallati, L. Chrpa, and T. McCluskey, editors, *The Eighth International Planning Competition (IPC-2014)*, pages 13–16, 2014.

[44] I. Pohl. Heuristic search viewed as path finding in a graph. *Artif. Intell.*, 1(3): 193–204, 1970. doi: 10.1016/0004-3702(70)90007-X. URL `http://dx.doi.org/10.1016/0004-3702(70)90007-X`.

[45] S. Richter and M. Helmert. Preferred operators and deferred evaluation in satisficing planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-2009)*, pages 273–280, 2009.

[46] S. Richter and M. Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

[47] S. Richter and M. Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *CoRR*, abs/1401.3839, 2014.

[48] S. Richter, M. Helmert, and M. Westphal. Landmarks revisited. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-2008)*, pages 975–982, 2008. ISBN 978-1-57735-368-3.

[49] S. Richter, J. Thayer, and W. Ruml. The joy of forgetting: Faster anytime search via restarting. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS-2010)*, pages 137–144, 2010.

[50] S. Richter, M. Westphal, and M. Helmert. LAMA 2008 and 2011. In García-Olaya et al. [15], pages 50–54.

[51] G. Röger and M. Helmert. The more, the merrier: Combining heuristic estimators for satisficing planning. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS-2010)*, pages 246–249, 2010.

[52] G. Röger and M. Helmert. The more, the merrier: Combining heuristic estimators for satisficing planning. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS-2010)*, pages 246–249, 2010.

[53] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003. ISBN 0137903952.

[54] B. Srivastava, T. Nguyen, A. Gerevini, S. Kambhampati, M. Do, and I. Serina. Domain independent approaches for finding diverse plans. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2016–2022, 2007. URL `http://dli.iiit.ac.in/ijcai/IJCAI-2007/PDF/IJCAI07-325.pdf`.

[55] J. Thayer and W. Ruml. Bounded suboptimal search: A direct approach using inadmissible estimates. In Walsh [63], pages 674–679. ISBN 978-1-57735-516-8.

[56] J. Thayer, J. Benton, and M. Helmert. Better parameter-free anytime search by minimizing time between solutions. In *SOCS*, 2012.

[57] R. Valenzano and F. Xie. On the completeness of best-first search variants that use random exploration. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI 2016, Phoenix, Arizona USA, February 12âĂŞ17, 2016*, 2016.

[58] R. Valenzano, H. Nakhost, M. Müller, J. Schaeffer, and N. Sturtevant. ArvandHerd: Parallel planning with a portfolio. In García-Olaya et al. [15], pages 113–116.

[59] R. Valenzano, H. Nakhost, M. Müller, J. Schaeffer, and N. Sturtevant. Arvandherd: Parallel planning with a portfolio. In Luc De Raedt, Christian Bessière, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter J. F. Lucas, editors, *ECAI*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 786–791. IOS Press, 2012. ISBN 978-1-61499-097-0.

[60] R. Valenzano, J. Schaeffer, N. Sturtevant, and F. 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*, pages 375–379, 2014.

[61] M. Vallati, L. Chrpa, and T. McCluskey, editors. *The 2014 International Planning Competition*, 2014.

[62] V. Vidal. A lookahead strategy for heuristic search planning. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-2004)*, pages 150–160, 2004.

[63] Toby Walsh, editor. *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, 2011. ISBN 978-1-57735-516-8.

[64] F. Xie. Exploration and combination: Randomized and multi-strategies search in satisficing planning. In *Doctoral Consortium of ICAPS-13*, 2013.

[65] F. Xie, H. Nakhost, and M. Müller. A local Monte Carlo tree search approach in deterministic planning (abstract). In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*, pages 1832–1833, 2011.

[66] F. Xie, H. Nakhost, and M. Müller. Planning via random walk-driven local search. In Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet, editors, *Proceeedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS-2012)*, pages 315–322, 2012.

[67] F. Xie, R. Valenzano, and M. Müller. Better time constrained search via randomization and postprocessing. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*, pages 269–277, 2013.

[68] F. Xie, M. Müller, and R. Holte. Adding local exploration to Greedy Best-First Search in satisficing planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2388–2394, 2014.

[69] F. Xie, M. Müller, and R. Holte. Jasper: the art of exploration in Greedy Best First Search. In M. Vallati, L. Chrpa, and T. McCluskey, editors, *The Eighth International Planning Competition (IPC-2014)*, pages 39–42, 2014.

[70] F. Xie, M. Müller, R. Holte, and T. Imai. Type-based exploration with multiple search queues for satisficing planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2395–2402, 2014.

[71] F. Xie, A. Botea, and A. Kishimoto. Heuristic-aided compressed distance databases. In *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 85–91, 2015.

[72] F. Xie, M. Müller, and R. Holte. Understanding and improving local exploration for gbfs. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11*, pages 244–248, 2015.