Marvin Abisrror Zarate

On Selecting Heuristic Function Subsets For Domain Independent-Planning.

Brasil

2016, v-1.9.5

Marvin Abisrror Zarate

On Selecting Heuristic Function Subsets For Domain Independent-Planning.

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para a obtenção do título de *Magister Scientiae*.

Universidade de Viçosa – UFV

Centro de Ciencias Exactas e Tecnologicas (CCE)

Programa de Pós-Graduação

Orientador: Levi Henrique Santana de Lelis

Co-orientador: Santiago Franco

Brasil

2016, v-1.9.5

Marvin Abisrror Zarate
On Selecting
Heuristic Function Subsets For Domain Independent-Planning./ Marvin Abisrror Zarate. – Brasil, 2016, v-1.9.5-
66 p. : il. (algumas color.) ; 30 cm.

Orientador: Levi Henrique Santana de Lelis

Tese (Mestrado) – Universidade de Viçosa – UFV
Centro de Ciencias Exactas e Tecnologicas (CCE)
Programa de Pós-Graduação, 2016, v-1.9.5.

1. Palavra-chave1. 2. Palavra-chave2. 2. Palavra-chave3. I. Orientador. II. Universidade xxx. III. Faculdade de xxx. IV. Título

Marvin Abisrror Zarate

On Selecting Heuristic Function Subsets For Domain Independent-Planning.

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para a obtenção do título de *Magister Scientiae*.

Trabalho aprovado. Brasil, 24 de novembro de 2012:

Levi Henrique Santana de Lelis
Orientador

Professor
Convidado 1

Professor
Convidado 2

Brasil
2016, v-1.9.5

This dissertation is dedicated to my Mother.

Acknowledgements

I would like to express my sincere gratitude to my advisor PhD. Levi Henrique Santana de Lelis, for the continuous support and guidance during the dissertation process. His valuable advice, patience and encouragement have been of great importance for this work.

Besides my advisor, I would like to thank to my co−advisor: PhD. Santiago Franco for his insightful feedback, interest and tough questions.

To the professors of the CCE, particularly the Master Degree program with all its members, played an invaluable role in my graduate education.

Last, but not least, I would like to thank my Mother.

"Não vos amoldeis às estruturas deste mundo,
mas transformai-vos pela renovação da mente,
a fim de distinguir qual é a vontade de Deus:
o que é bom, o que Lhe é agradável, o que é perfeito.
(Bíblia Sagrada, Romanos 12, 2)

Abstract

In this dissertation we present greedy methods for selecting a subset of heuristics functions from a large set of heuristics with the objective of reducing the running time of search algorithms. Holte et al. (2006) showed that search can be faster if several smaller pattern databases are used instead of one large pattern database. We introduce greedy methods for selecting a subset of the most promising heuristics from a large set of heuristic functions to guide A* search algorithm. If the heuristics are consistent, our method is able to make optimal subset selections with respect to an approximation of the A* search tree size. In addition to being consistent, if all heuristics have the same evaluation time, the subset our method selects is also good with respect to the resulting A* running time. We implemented our method in Fast Downward and showed empirically that it produces heuristics which outperform the state-of-the-art planners in the International Planning Competition benchmarks.

Key-words: Heuristics selection; A*

List of Figures

Figure 1 – 8 tile puzzle using DFS. (Bernard Chen, 2011) 22
Figure 2 – 8 tile puzzle using BFS. (Bernard Chen, 2011) 23
Figure 3 – The left tile-puzzle is one possible initial distribution of tiles and the right tile-puzzle is the goal distribution of tiles. Each one represents a state. 30
Figure 4 – Heuristic Search: I: Initial state, s: Some sate, G: Goal state 31
Figure 5 – Out of place heuristic 31
Figure 6 – Manhatham distance heuristic 32
Figure 7 – Type system and the search space representation. 41
Figure 8 – Search tree using type system 44
Figure 9 – Cartesian Plane with domain $\langle 0, 2\rangle$ and range $\langle 0, 2\rangle$ 57
Figure 10 – SS vs A* ratios for the optimal domains − The number of instances used in each domain are showed next to the name of the Domain. . . . 58

List of Tables

Table 1 – Ratios of the number of nodes expanded using $h_{max}(\zeta')$ to the number of nodes expanded using $h_{max}(\zeta)$ 47

Table 2 – Coverage of SS, CS and Max on the 2011 IPC benchmarks. For GHS using only GA-PDBs heuristics. 49

Table 3 – Coverage of different planning systems on the 2011 IPC benchmarks. For the GHS and Max approaches we also present the average number of heuristics GHS selects ($|\zeta'|$). 52

Table 4 – Comparison between SS and IDA* for 1, 10, 100, 1000 and 5000 probes using $hmax$ heuristic. 54

Table 5 – Poor prediction of SS against A* using ipdb, LM-Cut and M&S with 500 probes 55

Table 6 – Percentage of choices SS made correctly. 59

Contents

I	**INTRODUCTION**	**19**
1	**ABOUT THE PROBLEM**	**21**
1.1	**Background work**	**21**
1.2	**Aim and Objectives**	**24**
1.2.1	Aim	24
1.2.2	Objectives	24
1.3	**Scope, Limitations, and Delimitations**	**25**
1.4	**Justification**	**25**
1.5	**Hypothesis**	**25**
1.6	**Contribution of the Dissertation**	**25**
1.7	**Organization of the Dissertation**	**25**
II	**LITERATURE REVIEW**	**27**
2	**BACKGROUND**	**29**
2.1	**Similar Selection Systems**	**29**
2.2	**Problem definition**	**29**
2.3	**The 8-tile-puzzle case**	**30**
2.4	**Heuristics**	**30**
2.4.1	Out of place (O.P)	31
2.4.2	Manhatham Distance (M.D)	32
2.5	**Heuristic Generators**	**32**
2.6	**Take advantage of Heuristics**	**32**
2.7	**Heuristic Subset**	**33**
III	**APPROACH PROPOSAL**	**35**
3	**META-REASONING FOR SELECTION**	**37**
3.1	**Greedy Heuristic Selection (GHS)**	**37**
3.2	**Approximately Minimizing Search Tree Size**	**37**
3.3	**Approximately Minimizing A*'s Running Time**	**38**
3.4	**Estimating Tree Size and Running Time**	**38**
3.5	**Culprit Sampler (CS)**	**39**
3.6	**Stratified Sampling (SS)**	**40**
3.6.1	Type System	40

3.7 SS step by step 43

IV EMPIRICAL EVALUATION 45

4 EMPIRICAL EVALUATION 47
4.1 Empirical Evaluation of $\hat{J}$ and $\hat{T}$ 48
4.2 Comparison with Other Planning Systems 50
4.3 Systems Tested 50
4.4 Discussion of the Results 51
4.5 Comparison between SS and IDA* 53
4.6 Comparison between SS and A* 55
4.7 Approximation Analysis for SS and A* 56

V CONCLUSION 61

5 CONCLUDING REMARKS 63

Bibliography 65

Chapter I

Introduction

1 About the Problem

State-space search algorithms have been used to solve important real-world problems, such as problems arising in Robotics, domain-independent planning, chemical compounds discovery, bin packing, sequence alignment, automating layouts of sewers, and network routing, among others. In this dissertation we study methods for selecting a subset of heuristic functions while minimizing the search tree size and the running time of the A* search algorithm (Hart P. E. et al., 1968).

We are interested in selecting heuristics from a large set of possibilities because Holte et al., (2006) showed that search can be faster if several smaller pattern databases are used instead of one large pattern database. In fact, we believe that different heuristics can give us valuable information about the solution of the problem. For example, one heuristic can be helpful in a region of the search tree where other heuristics aren't. Then, instead of using one heuristic to find the solution, it would be best to use the most promising subset of heuristics from a possibly large set.

1.1 Background work

Search algorithms are used to solve Artificial Intelligence (`AI`) problems by finding sequence of actions that goes from the start state to the goal state in the search space. Two well know search algorithms are Depth-First Search (`DFS`) and Breadth-First Search (`BFS`). `DFS` looks for the solution by exploring the subtree rooted node n before exploring the substrees rooted at n's siblings while looking for a path from start to goal. In the Figure 1 we can see `DFS` expands 31 states until finding a solution to the problem. `BFS` looks for the solution by exploring the nodes in a given level, before exploring the nodes in the next level. In the Figure 2 we can see `BFS` expands 46 states to find the goal. In both Figure 1 and 2 the numbers above of each state represent the order in which the states are visited. Furthermore, both search algorithms usually visit a large number of states before finding a solution. We call brute-force-search tree (`BFST`) the tree expanded by `DFS` and `BFS`.

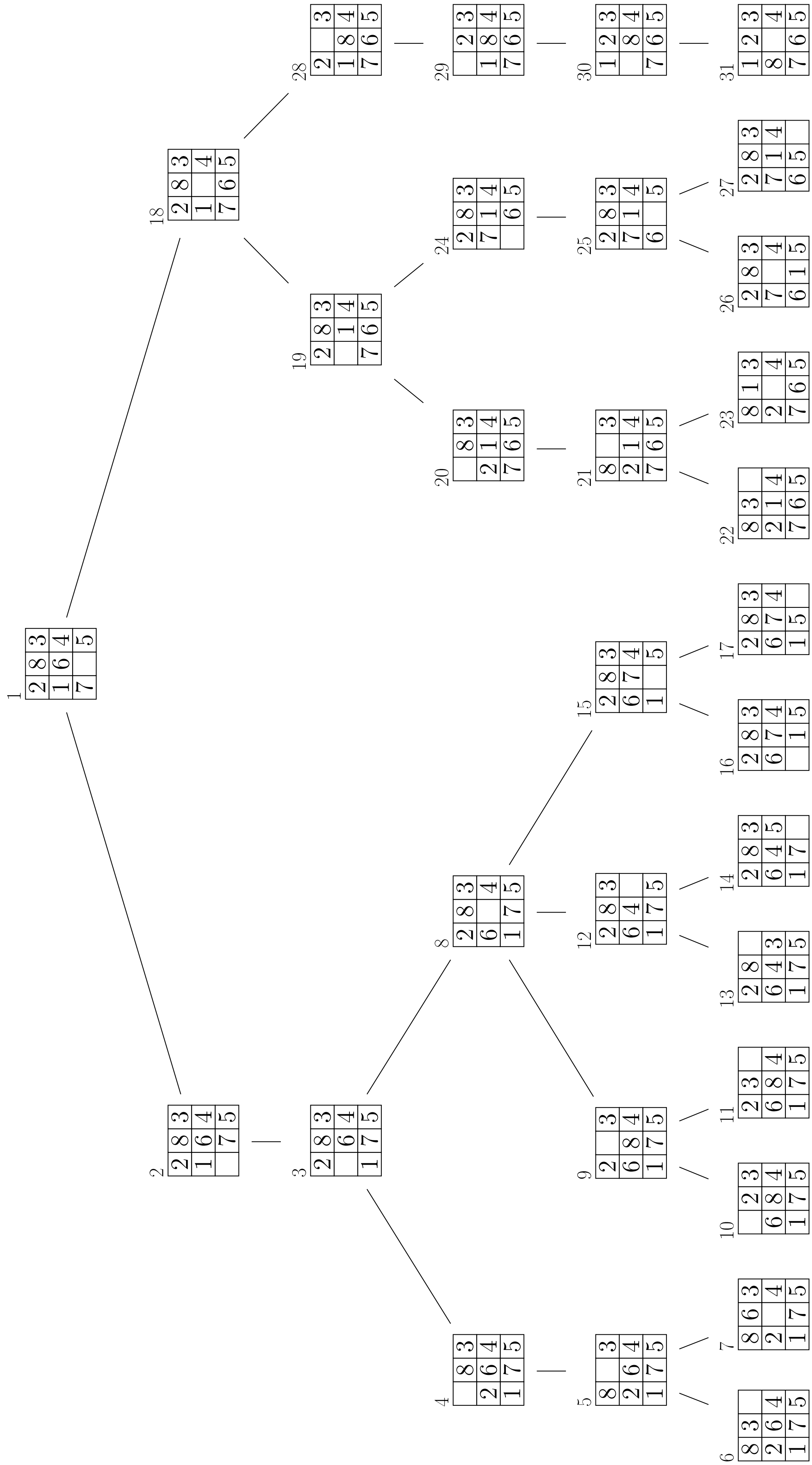

Figure 1 – 8 tile puzzle using DFS. (Bernard Chen, 2011)

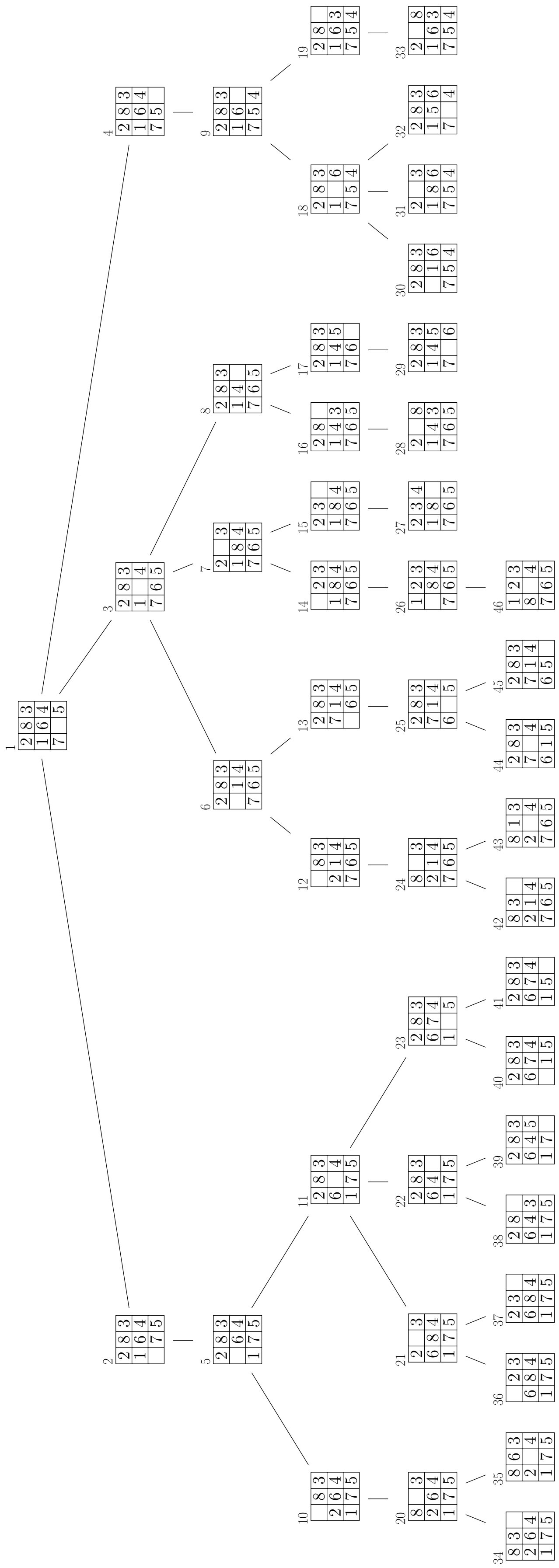

Figure 2 – 8 tile puzzle using BFS. (Bernard Chen, 2011)

There are other types of algorithms called heuristic search algorithms, which use one of the most important algorithm that use heuristic is A* (Hart P. E. et al., 1968), which also solves problems optimally. The heuristic is the estimation of the distance for one node in the search tree to get to a goal state. The heuristic search algorithms tend to generate smaller search trees in comparison to the `BFST`, because the heuristic guides the search to more promising parts of the state space. Also, by reducing the search tree size, the heuristic function guidance might also reduce the overall running time of the algorithm.

There are different approaches (Haslum et al., 2007; Edelkamp, 2007; Nissim et al., 2011) that have been developed to generate heuristics. The way to create heuristics could be using domain knowledge or without domain knowledge. The systems that generate a heuristic are called heuristic generators by Barley et al., (2014). Actually, the approache that have showed most successful results in heuristic generation is the Pattern Database (`PDB`) (Culberson and Schaeffer, 1998). The way how `PDBs` work is the following: The search space of the problem is abstracted into a smaller state space that can be enumerated with exhaustive search. The distance of all abstracted states to the abstracted goal state are stored in a lookup table, which can be used as a heuristic function for the original state space.

Holte et al., (2006) showed that search can be faster if several smaller `PDBs` are used instead of one large pattern database. In addition Domshlak et al., (2010) and Tolpin et al., (2013) showed that evaluating the heuristic lazily, only when they are essential to a decision to be made in the search process is worthy in comparison to take the maximum of the set of heuristics. Our meta-reasoning proposes to use selection of heuristics instead of the methods proposed already because we believe that we will outperformed other approaches using approximations.

1.2 Aim and Objectives

1.2.1 Aim

The objective of this dissertation is to develop meta-reasoning approaches for selecting a heuristic subset from a large set of heuristics with the goal of reducing the running time of the search algorithms employing these functions.

1.2.2 Objectives

- Develop an approach to find a subset of heuristics from a large pool of heuristics ζ that minimizes the number of nodes expanded by A* in the process of search.

- Develop an approach to find a subset of heuristics from a large pool of heuristics ζ that minimizes the A* running time.

1.3 Scope, Limitations, and Delimitations

We implemented our method in Fast Downward (Helmert, 2006) and we test our methods on the 2011 International Planning Competition (IPC) domain instances.

1.4 Justification

Good results have been obtained in domain-independent planning by using heuristic search approach (Haslum et al., 2007). The heuristic function used to guide the A* search can greatly affect the algorithm's running time.

We use heuristic generators in order to create a large set of heuristics and obtain heuristics to solve problems.

1.5 Hypothesis

We test the following hypothesis:

- **H1:** A greedy algorithm is effective for selecting a good subset of heuristics to guide the A* search.

1.6 Contribution of the Dissertation

The main contributions of this dissertation are:

- Provide a meta-reasoning approach for selecting heuristic functions while minimizing the number of nodes by A*.
- Provide a meta-reasoning approach for selecting heuristic functions while minimizing the running time of the A* search.

1.7 Organization of the Dissertation

The dissertation is organized as follows:

1. In Chapter I, the background of the dissertation is provided, which also includes our motivation and the scope definition.

2. In Chapter II, we review the state-of-the-art in selection of heuristic functions.
3. In Chapter III, we introduce Greedy Heuristic Selection (`GHS`).
4. In Chapter IV, we explain the results obtained by using `GHS` and compare it with other planner systems.
5. We conclude in Chapter V.

In the next chapter, the domain 8-tile-puzzle is used to understand the concepts that will be helpful for the other chapters.

Chapter II

Literature Review

2 Background

2.1 Similar Selection Systems

The system most similar to ours is RIDA* (Barley et al., 2014). RIDA* also selects a subset from a pool of heuristics to guide the A* search. In RIDA* this is done by starting with an empty subset and trying subsets of size one before trying subsets of size two and so on. RIDA* stops after evaluating a fixed number of subsets. While RIDA* is able to evaluate sets of heuristics with only tens of elements. By contrast, the method we propose in this dissertation is able to evaluate sets with thousands of elements.

Rayner et al., (2013) present an optimization procedure that is similar to ours. In contrast with our work, Rayner et al. limited their experiments to a single objective function that sought to maximize the sum of heuristic values in the state space. Moreover, Rayner et al.'s method performs an uniform sampling of the state space to estimate the sum of heuristic values in the state space. Thus, their method is not directly applicable to domain-independent planning. In this dissertation we adapt Rayner et al.'s approach to domain-independent planning by using Chen (1992) Stratified Sampling to estimate the sum of heuristic values in the state space. Our empirical results show that `GHS` minimizing an approximation of A*'s running time is able to substantially outperform Rayner et al.'s approach in domain-independent planning.

Our meta-reasoning requires a prediction of the number of nodes expanded by A* using any given subset. The prediction system we choose is Stratified Sampling (`SS` system (Lelis et al., 2013)). Even though, `SS` produce good predictions for Iterative Deepening−A*, it does not give us good predictions for A* because it is unable to detect duplicated nodes during search (Lelis et al., 2014). However, we use it as a estimator in our meta-reasoning because we showed in optimal domains that `SS` has good results.

2.2 Problem definition

A SAS^+*planning task* (Bäckström; Nebel, 1995) is a 4 tuple $\nabla = \{V, O, I, G\}$. V is a set of *state variables.* Each variable $v \in V$ is associated with a finite domain of possible D_v. A state is an assignment of a value to every $v \in V$. The set of possible states, denoted V, is therefore $D_{v_1} \times ... \times D_{v_2}$. O is a set of operators, where each operator $o \in O$ is triple $\{pre_o, post_o, cost_o\}$ specifying the preconditions, postconditions (effects), and non-negative cost of o. pre_o *and* $post_o$ are assignments of values to subsets os variables, V_{pre_o} *and* V_{post_o}, respectively. Operator o is applicable to state s if s and pre_o agree on the assignment of values to variables in V_{pre_o}. The effect of o, when applied to s, is to set the variables in V_{post_o} to the values specified in $post_o$ and to set all other variables to the value they have in s. G is the goal condition, an assignment of values to a subset of variables, V_G. A state is a goal state if it and G agree on the assignment of values to the variable in V_G. I is the initial state, and the planning task, ∇, is to find an optimal (least-cost) sequence of operators leading from I to a goal state. We denote the optimal solution cost of ∇ as C^*.

2.3 The 8-tile-puzzle case

The state space problem illustrated in the Figure 3 consists in a board with 8 squares named tiles numbered from 1 to 8 and one square without tile and number named empty tile. The goal of the game is to order the tiles in some order. For example: From left to right and up to bottom in the following order 1, 2, 3, 4, 5, 6, 7, 8 and empty tile or ordering the tiles in the following way: From left to right and up to down 1, 2, 3, 8, empty tile, 4, 7, 6, 5. This distribution looks like we are ordering around the tile of the center of the board. This game is called the 8 Sliding-tile puzzle and its objective is to place the tiles in order by moving the numbered tiles into the empty tile. For this case, the goal would be reached by placing the tiles 1, 2 and 3 in the first row, and 4, 5 and 6 in the following row, and 7, 8 and empty tile in the last row.

Initial

4	1	2
8		3
5	7	6

Goal

1	2	3
4	5	6
7	8	

Figure 3 – The left tile-puzzle is one possible initial distribution of tiles and the right tile-puzzle is the goal distribution of tiles. Each one represents a state.

Instead of using `DFS` or `BFS` that will analyze the states encountered, we can obtain heuristics for the 8-sliding-tile puzzle that will help us solve the problem quickly.

2.4 Heuristics

There exist many state-space algorithms, and one of the most important and well know is A* (Hart P. E. et al., 1968). Each node in the search tree generated by A* uses the $f(s) = g(s) + h(s)$ cost function to find the solution. $g(s)$ is the cost to go from the start state to state s, and $h(s)$ is the estimated cost to go from s to the goal.

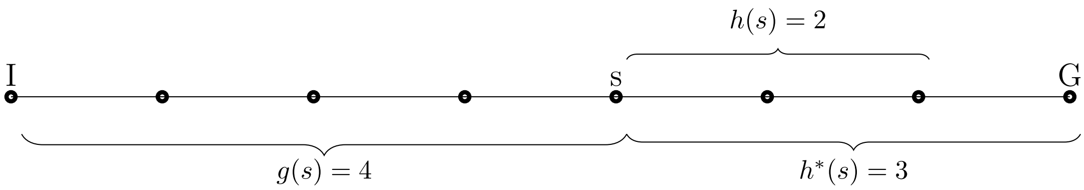

Figure 4 – Heuristic Search: I: Initial state, s: Some sate, G: Goal state

In the Figure 4 the optimal distance from the initial state I to the state s is 4 and is represented by $g(s)$. The $h^*(s) = 3$ represents the optimal distance from s to the goal state G, and $h(s) = 2$ is the estimation distance from s to G.

A heuristic function $h(s)$ estimates the cost of a solution path from s to a goal state. A heuristic is admissible if $h(s) \leq h^*(s)$ for all $s \in V$, where $h^*(s)$ is the optimal cost of s. A heuristic is consistent iff $h(s) \leq c(s,t) + h(t)$ for all states s and t, where $c(s,t)$ is the cost of the cheapest path from s to t. For example, the heuristic function provided by a pattern database (PDB) heuristic (Culberson and Schaeffer, 1998) is admissible and consistent.

Given a set of admissible and consistent heuristics $\zeta = \{h_1, h_2, \ldots, h_M\}$, the heuristic $h_{max}(s,\zeta) = \max_{h\in\zeta} h(s)$ is also admissible and consistent. When describing our method we assume all heuristics to be consistent. We define $f_{max}(s,\zeta) = g(s) + h_{max}(s,\zeta)$, where $g(s)$ is minimal when A* using a consistent heuristic expands s. We call an A* search tree the tree defined by the states generated by A* using a consistent heuristic while solving a problem ∇.

We now show two heuristics for the 8-puzzle.

2.4.1 Out of place (O.P)

This heuristic counts the number of tiles that are out of the goal position. If the tile is not in the goal position, then it counts as one, otherwise it counts as zero.

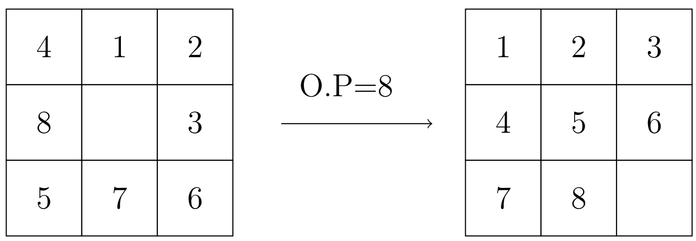

Figure 5 – Out of place heuristic

The tiles numbered with 4, 1, 2, 3, 6, 7, 5, and 8 are out of place then each tile count as 1 and the sum would be 8, then the heuristic value for this state is 8.

2.4.2 Manhatham Distance (M.D)

This heuristic counts the minimum number of operations that should be applied to any tile to place it in its goal position. Let us explain this with an example: Tile 5 is located on the bottom left of the tile puzzle shown on the left-hand side of Figure 6, then the minimum number of moves to get tile 5 to its goal position is 2 (either up and right or right and up), both movements equal to 2.

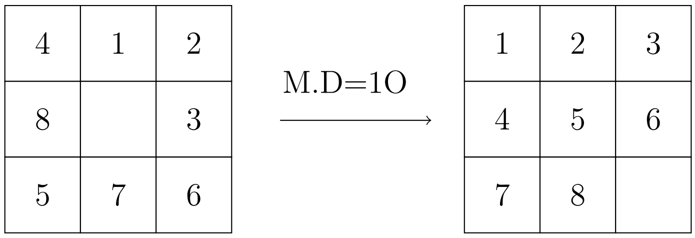

Figure 6 – Manhatham distance heuristic

Tile 4 requires one move to get to its goal position; Tile 1 requires one move to get to its goal position; Tile 2 requires one move to get to its goal position; Tile 3 requires one move to get to its goal position; Tile 6 requires one move to get to its goal position; Tile 7 requires one move to get to its goal position; Tile 5 requires two moves to get to its goal position; Tile 8 requires two moves to get to its goal position; Then the sum would be 10.

Out-of-place and Manhattan distance are heuristics that use domain knowledge. There are methods that create heuristics without using domain knowledge. Barley et al., (2014) call them heuristic generators.

2.5 Heuristic Generators

Some heuristic generators work by creating abstractions of the original problem space. `PDBs` (Culberson and Schaeffer, 1998) are an effective way of generating heuristics, `PDBs` work as follows: The search space of the problem is abstracted into a smaller state space that can be enumerated with exhaustive search. The distance of all abstracted states to the abstracted goal state are stored in a lookup table, which can be used as a heuristic function for the original state space.

2.6 Take advantage of Heuristics

One well know approach to take advantage of heuristics is taking the maximum of all heuristics ζ. For example, using three different heuristics $h1, h2$ and $\max(h1, h2)$. Heuristic $h1$ and $h2$ are based on domain abstractions and the $\max(h1, h2)$ is the maximum heuristic value of $h1$ and $h2$.

If we have to choose which heuristic to use between $h1$ or $h2$ or $\max(h1, h2)$. One of the answers would be $\max(h1, h2)$, because that value would allow to be near from the objective.

The heuristic generators can create hundreds or even thousands of heuristics and we need to figure it out how to take advantage of this large set of heuristics created. In fact, there exist different ways to take advantage of those heuristics, however we need to take into account that if we want to use all the heuristics created by the heuristic generator, the resulting heuristic would be too expensive to guide search. This is because, the main problem involved would be the time to evaluate all the heuristics for each node in the search tree.

In this dissertation we use the meta-reasoning based on the minimum of an approximation to the size of the search tree.

2.7 Heuristic Subset

Let's suppose we have to run our meta-reasoning considering that we have a fixed amount of memory M. Then, one question is raised: How many heuristics our system should handle in order to avoid a collapse of the memory and stop the system? Holte et al., (2006) observed that maximizing over N pattern databases of size M/N, for $N < M$, can produce a size of the search tree smaller than the search tree size generated by using a single pattern database of size M. Then, the number of heuristics that we need to select from a large set of heuristics must be suitable to make our system work without problems.

Heuristic generator systems can create a large number of heuristics. Let us suppose $|\zeta| = 1000$ heuristics were created considering the time and memory avaiable and we want to select the best $N = 50$ heuristics. This would imply count the following number of combinations one can get from solving the combinatorial problem:

$$\binom{1000}{50} \approx 10^{85} possibilities$$

In the next Chapter, we will introduce the meta-reasoning proposed for selecting heuristics.

Chapter III

Approach Proposal

3 Meta-Reasoning for selection

3.1 Greedy Heuristic Selection (GHS)

We present a greedy algorithm selecction for approximately solving the heuristic subset selection problem while optimizing different objective functions. We consider the following general optimization problem.

$$\textbf{minimize}_{\zeta' \subseteq \zeta} \Psi(\zeta', \nabla) \tag{3.1}$$

Where $\Psi(\zeta', \nabla)$ is an objective function that we want to minimize using a subset of heuristics ζ' that is selected from ζ. According to Rayner et al., (2013) it is unlikely that there is an efficient algorithm for solving Equation 3.1.We use an algorithm based on the local search we call Greedy Heuristic Selection (`GHS`) to approximately solve Equation 3.1 for different functions Ψ.

Algoritmo 1: Greedy Heuristic Selection

Input : problem ∇, set of heuristics ζ
Output : heuristic subset $\zeta' \subseteq \zeta$

1 $\zeta' \leftarrow \emptyset$
2 **while** Ψ *can be improved* **do**
3 $\quad h \leftarrow \arg\min_{h \in \zeta} \Psi(\zeta' \cup \{h\}, \nabla)$
4 $\quad \zeta' \leftarrow \zeta' \cup \{h\}$
5 return ζ'

Algorithm 1 shows `GHS`. `GHS` receives as input a problem ∇, a set of heuristics ζ, and it returns a subset $\zeta' \subseteq \zeta$. In each iteration `GHS` greedily selects from ζ the heuristics h which will result in the largest reduction of the value Ψ (line 3). `GHS` returns ζ' once the objective function can not be improved. In other words, the algorithm will halt when adding another heuristic does not improve the objective function.

3.2 Approximately Minimizing Search Tree Size

The first objective function Ψ we consider accounts for the number of expansions A^* performs while solving a given planning problem. The planning problem must be solvable, this means C^* can not be infinity. When solving ∇ using the consistent heuristic function $h_{max}(\zeta^{'})$ for $\zeta^{'} \subseteq \zeta$, A* expands in the upper bound $J(\zeta^{'}, \nabla)$ nodes, where

$$J(\zeta^{'}, \nabla) = |\{s \in V | f_{max}(s, \zeta^{'}) \leq C^*\}| \tag{3.2}$$

$$J(\zeta^{'}, \nabla) = |\{s \in V | h_{max}(s, \zeta^{'}) \leq C^* - g(s)\}| \tag{3.3}$$

We write $J(\zeta^{'})$ or simply J instead of $J(\zeta^{'}, \nabla)$. What's more, we assume that A* expands all nodes s with $f(s) \leq C^*$ solving ∇, as shown in Equation 3.2. GHS is able to find the solutions when we use J as the objective function Ψ.

3.3 Approximately Minimizing A*'s Running Time

Another objective function Ψ we consider accounts for the A* running time and is defined as follows. Let $T(\zeta^{'}, \nabla)$ be an approximation to the running time of A* when using $h_{max}(\zeta^{'})$ for solving ∇, defined as follows.

$$T(\zeta^{'}, \nabla) = J(\zeta^{'}, \nabla) \cdot t_{h_{max}}(\zeta^{'}) \tag{3.4}$$

where, for any heuristic function h, the term t_h refers to the running time used for computing the $h-$value of any state s.

We assume that t_h to be independent of s, which is a reasonable assumption for several heuristics such as PDBs.

In order to compute the running time of A* exactly we would also have to account for all nodes evaluated. Specifically, our objective function accounts for the generation time added to the heuristic evaluation time of all nodes generated, not only nodes expanded. In this way, $T(\zeta^{'}, \nabla)$ is reasonable approximation for A*'s running time for the heuristic subset selection problem.

3.4 Estimating Tree Size and Running Time

In practice GHS used approximations models of J, T, and $T^{'}$ instead of their exact values. This is because computing J, T, and $T^{'}$ exactly would require solving ∇, and this is what we obviously want to avoid. We denote the approximations of J as $\hat{J}$, and since both T and $T^{'}$ model A*'s running time, we denote the approximation for both as $\hat{T}$.

We use the Culprit Sampler (CS) introduced by Barley et al., (2014) and the Stratified Sampling (SS) algorithm introduced by Chen, (1992) for computing $\hat{J}$ and $\hat{T}$.

Each of the two algorithms has its strengths and weaknesses, which we explore in the experimental Chapter.

Both `CS` and `SS` must be able to quickly estimate the values of $\hat{J}(\zeta')$ and $\hat{T}(\zeta')$ for any subset ζ' of ζ so they can be used in `GHS`'s optimization process.

3.5 Culprit Sampler (CS)

`CS` is an approach introduced by Barley et al., (2014) and works by running in a time$-$bounded A* search while sampling $f-culprits$ and $b-culprits$. In this dissertation we use `CS` to estimate the values of $\hat{J}$ and $\hat{T}$.

Definition 3.5.1. *(f-culprit) Let $\zeta = \{h_1, h_2, ..., h_M\}$ be a set of heuristics. The f$-$culprit of a node n in an A* search tree is defined as the tuple F(n) = $\langle f_1(n), f_2(n), ..., f_M(n)\rangle$, where $f_i(n) = g(n) + h_i(n)$. For any n$-$tuple F, the counter C_F denotes the number of nodes n in the tree with F(n) = F.*

Definition 3.5.2. *(b-culprit) Let $\zeta = \{h_1, h_2, ..., h_M\}$ be a set of heuristics and b a lower bound on the solution cost ∇. The b$-$culprit of a node n in an A* search tree is defined as the tuple $B(n) = \langle y_1(n), y_2(n), ..., y_M(n)\rangle$, where $y_i(n) = 1$ if g(n) + $h_i(n) \leq b$ and $y_i(n) = 0$, otherwise. For any binary n$-$tuple B, the counter C_B denotes the number of nodes n in the tree with B(n) = B.*

`CS` works by running an A* search bounded by a user$-$specified time limit. Then, `CS` compresses the information obtained in the A* search (i.e., the $f-$values of all nodes expanded according to all heuristics h in ζ) in b$-$culprits, which are later used for computing $\hat{J}$. The f$-$culprits are generated as an intermediate step for computing the b$-$culprits, as we explain below. The maximum number of f$-$culprits and b$-$culprits in an A* search tree is equal to the number of nodes in the tree expanded by the time$-$bounded A* search. However, in practice the number of f$-$culprits is usually much lower than the number of nodes in the tree. Moreover, in practice, the total number of different b$-$culprits tends to be even lower than the total number of f$-$culprits. Given a planning problem ∇ and a set of heuristics ζ, `CS` samples the A* search tree as follows.

1.- `CS` runs A* using $h_{min}(s, \zeta) = min_{h \in \zeta} h(s)$ until reaching a user$-$specified time limit. A* using h_{min} expands node n if it were to expand n while using any of the heuristics in ζ individually. For each node n expanded in this time$-$bounded search we store n's f$-$culprit and its counter.

2.- Let f_{maxmin} be the largest $f-$value according to h_{min} encountered in the time$-$bounded A* search described above. We now compute the set $\mathbb{B}$ of b$-$culprits and their counters based on the f−culprits and on the value of f_{maxmin}. This is done by iterating over all f−culprits once.

The process described above is performed only once GHS's execution. The value of $\hat{J}(\zeta', \nabla)$ for any subset ζ' of ζ if then computed by iterating over all b−culprits **B** and summing up the relevant values of C_B. The relevant values of C_B represent the number of nodes A* would expand in a search bounded by b if using $h_{max}(\zeta')$. This computation can be written as follows.

$$\hat{J}(\zeta', \nabla) = \sum_{\mathbb{B} \in B} W(B) \tag{3.5}$$

Where $W(B)$ is 0 if there is a heuristic in ζ' whose y-value in B is zero (i.e., there is a heuristic in ζ' that prunes all nodes compressed into B), and C_B otherwise. If the time−bounded A* search with h_{min} expands all nodes n with $f(n) \leq C^*$, then $\hat{J} = J$. In practice, however, our estimate $\hat{J}$ will tend to be much lower than J.

The value of $\hat{T}$ is computed by multiplying $\hat{J}$ by the sum of the evaluation time of each heuristic in ζ'. The evaluation time of the heuristics in ζ' is measured in a separate process, before executing CS, by sampling a small number of nodes from ∇'s start state.

3.6 Stratified Sampling (SS)

Chen, (1992) presented a method for estimating the search tree size of backtracking search algorithms by using a stratification of the search tree to guide its sampling. We define Chen's stratification as a type system.

3.6.1 Type System

The type system is calculated based of any property of each node in the search space. In addition, this new search space that is created with the types is called partition of the search space and nodes are going to have the same type if they have the same f-value. Lelis et al., (2013). In the Figure 7 we can see that the type system is a partition of the type system.

Definition 3.6.1. *Type System Let S = (N,E) be a search tree, where N is its set of nodes and for each $n \in N$, $\{n' | (n, n') \in E\}$ is n's set of child nodes. $TS = \{t_1, ..., t_k\}$ is a type system for S if it is a disjoint partitioning of N. If $n \in N$ and $t \in TS$ with $n \in t$, we write $TS(n) = t$.*

According to Lelis et al., (2013), SS is an approximation function of the form $\varphi = \sum_{n \in S} z(n)$, where z is a function assigning a numerical value to a node, consequently

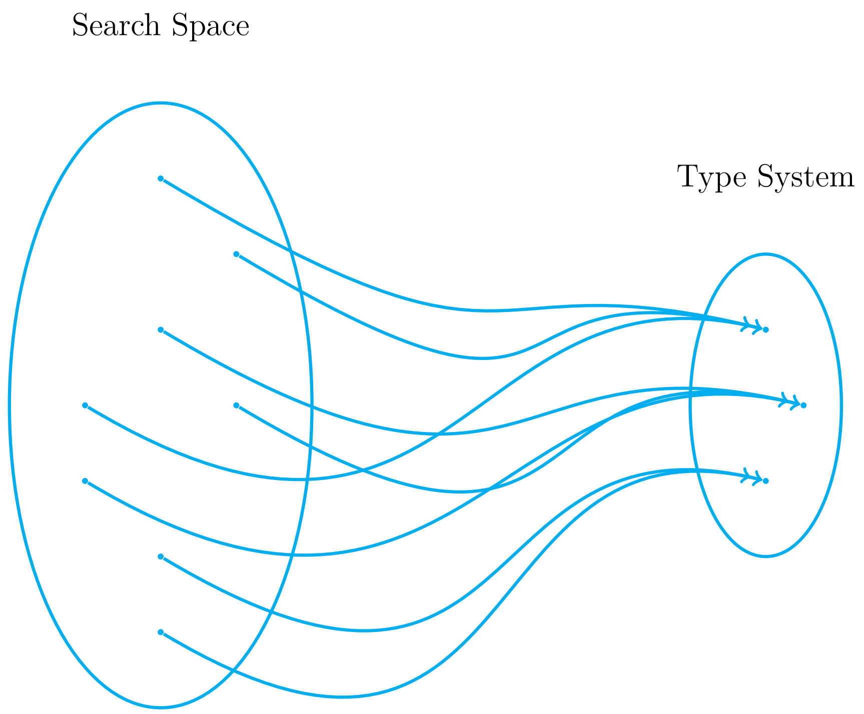

Figure 7 – Type system and the search space representation.

φ is a numerical property of the search tree rooted at n^*. Furthermore, if $z(n) = 1$ for all $n \in S$, then φ is the size of the tree. Instead of summing all the $z-$values from the tree, `SS` considers subtrees rooted at nodes of the same type will have equal values of φ and only one of each type, chosen at random, is expanded. This assumption is fundamental to `SS`'s efficiency due the fact that many nodes need to be examined intensely in the search tree. With input a search tree S and a type system TS, `SS` predicts φ in the following way: First, the search tree is sampled, returning the set of $representative - weight$ pairs, with one such pair for every unique type seen during sampling. Second, in the pair $\langle s, w \rangle$ in A for type $t \in TS$, n is the unique node of type t that was expanded during search and w is an estimate of the number of nodes type t in the tree. Finally, φ is then approximated by $\hat{\varphi}$, defined as, $\hat{\varphi} = \sum_{\langle s,w \rangle \in A} w \times z(n)$.

By making $z(n) = 1$ for all $n \in S$ `SS` prooduces an estimate $\hat{J}$ of J. Similarly to our approach with `CS`, we obtain $\hat{T}$ by multiplying $\hat{J}$ by the sum of heuristic evaluation time and generation time.

In `SS` the types are required to be partially ordered: a node's type must be strictly greater than the type of its parent. This can be guaranteed by adding the depth of a node to the type system and then sorting the types lexicographically. That is why in our implementation of `SS` types at one level are treated separately from types at another

Algoritmo 2: SS, a single probe

Input : root n^* of a tree and a type system TS, upper bound d, heuristic function h

Output : an array of sets A, where $A[i]$ is the set of (node, weight) pairs $< s, w >$ for the nodes n expanded at level i.

1 $A[0] \leftarrow \{\langle s^*, 1\rangle\}$

2 $i \leftarrow 0$

3 **while** *A[i] is not empty* **do**

4   **for** *each element* $\langle s, w\rangle$ *in* $A[i]$ **do**

5   **for** *each child* $\hat{n}$ *of* n **do**

6   **if** $g(\hat{n}) + h(\hat{n}) \leq d$ **then**

7   **if** $A[i+1]$ *contains an element* $\left\langle n^{'}, w^{'}\right\rangle$ *with* $TS(n^{'}) = TS(\hat{n})$ **then**

8                     $w^{'} \leftarrow w^{'} + w$

9   with probability $w/w^{'}$, replace $\left\langle n^{'}, w^{'}\right\rangle$ in $A[i+1]$ by $\left\langle \hat{n}, w^{'}\right\rangle$

10   **else**

11   insert new element $\langle \hat{n}, w\rangle$ in $A[i+1]$

12     $i \leftarrow i + 1$

level by the division of A into groups $A[i]$, where $A[i]$ is the set of representative−weight pairs for the types encountered at level i. If the same type occurs on differente levels the occurrences will be treated as if they were different types − the depth of search is implicitly included into all of our type systems.

Algorithm 2 shows SS in detail. Representative nodes from $A[i]$ are expanded to get representative nodes for $A[i+1]$ as follows. $A[0]$ is initialized to contain only the root of the search tree to be probed, with weight 1 (Line 1). In each iteration (Lines 4 through 11), all nodes in $A[i]$ are expanded. The children of each node in $A[i]$ are considered for inclusion in $A[i+1]$ if their $f-$value do not exceed an upper bound d provided as input to SS. If a child $\hat{n}$ has a type t that is already represented in $A[i+1]$ by another node $n^{'}$, then a *merge* action on $\hat{n}$ and $n^{'}$ is performed. In a merge action we increase the weight in the corresponding representative−weight pair of type t by the weight $w(n)$ of $\hat{n}$'s parent n (from level i) since there were $w(n)$ nodes at level i that are assumed to have children of type t at level $i+1$. $\hat{n}$ will replace $n^{'}$ according to the probability shown in Line 9. Chen, (1992) proved that this probability reduces the variance of the estimation. Once all the states in $A[i]$ are expanded, we move to the next iteration.

One run of the SS algorithm is called a *probe*. Chen, (1992) proved that the expected value of $\hat{\varphi}$ converges to φ in the limit as the number of probes goes to infinity. As Lelis et al., (2014), SS is not able to detect duplicated nodes in its sampling process. As a result, since A* does not expanded duplicates, SS usually overestimates the actual number of nodes A* expands. Thus, in the limit, as the number of probes grows large, SS's prediction converges to a number which is likely to overestimate the A* search tree size. We test empirically whether `SS` is able to allow `GHS` to make good subset selects despite being unable to detect duplicated nodes during sampling.

Similarly to `CS`, we also define a time−limit to run `SS`. We use `SS` with an iterative−deepening approach in order to ensure an estimate of $\hat{J}$ and $\hat{T}$ before reaching the time limit. We set the upper bound d to the heuristic value of the start state and, after performing p probes, if there is still time, we increase d to twice its previous value. The values of $\hat{J}$ and $\hat{T}$ is given by the prediction produced for the last d−value in which `SS` was able to perform all p probes.

`SS` must also be able to estimate the values of $\hat{J}(\zeta')$ and $\hat{T}(\zeta')$ for any subset ζ' of ζ. This is achieved by using `SS` to estimate b−culprits (See Definition 3.5.2) instead of the search tree size directly. Similarly to `CS,` `SS` used h_{min} of the heuristics in ζ to decide when to prune a node (See Line 6 of Algorithm 2) while sampling. This ensures that `SS` expands a node n if A* employing at least one of the heuristics in ζ would expand n according to bound d. The C_B counter of each b−culprit B encountered during SS's probe is given by,

$$C_B = \sum_{\langle n,w\rangle \in A \wedge B(n)=B} w \tag{3.6}$$

We recall that to compute $B(n)$ for node n one needs to define a bound b. Here we use the bound d used by `SS`. The average value of C_B across p probes is used to predict the search tree size for a given subset ζ'. As explained for `CS`, this can be done by traversing over all b−culprits once.

3.7 SS step by step

In the Figure 8, we can see how type system works. In this dissertation we use type system based only in heuristics. Lelis et al., (2013) use the type system where two nodes have the same type if they have the same heuristic value.

In the Level 1, we have the root node, and their w is initialized with one. Let's suppose that in the Level 2 three nodes are generated by the root node. The nodes in the Level 2 have the following types: red, blue and red from left to right respectively, and each node receive the same w from their father. In the Level 2 we apply `SS`'s assumption, which says that two nodes in the same level that have the same type (The same color) root subtrees of the same size and only one of them must be chosen randomly to be expanded. In Level 2 there are two nodes with type red. In that way, we choose randomly one of them. Let's suppose we choose the right red node. Then, we have to update the number of nodes with the type red using the w, both red node types have $w = 1$, then we sum the w

and the new $w = 2$. To sum up, in the Level 2 we will have two nodes of red type and one node with blue type.

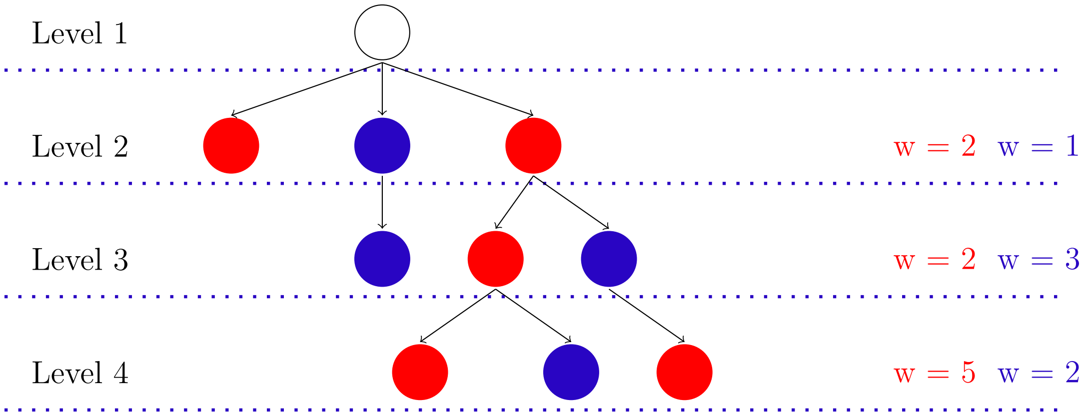

Figure 8 – Search tree using type system

Let's suppose that nodes in Level 2 are expanded. The blue node generates one node of type blue and the red node generates two nodes with the following types: red and blue from left to right. The question that raises here is how many nodes are generated in the Level 3. To answer this question, we only have to apply a simple multiplication between each type and their respectively weight w and then sum all the nodes with the same type. For Level 3 the answer would be: $1 \times \textit{blue} + 2 \times \textit{red} + 2 \times \textit{blue}$. Therefore, in Level 3 we will have 2 nodes of type red and 3 nodes of type blue. This process continues until reaching the bottom of the tree.

Finally, the number of nodes expanded in the search tree is obtained summing all w plus one (The root node). In summarize, the number of nodes expanded in the search tree would be $15 + 1 = 16$.

As we know the stochastic behavior of `SS`, if we run the algorithm again maybe in Level 2 instead of choose the right red node we choose the left red node and the history of the generation of nodes would be different in Level 3, as a consequence the number of nodes generated could be higher or lower than we have now. In other words, in each probe of the algorithm we could obtain a different value of the estimation of the search tree size, and in order to obtain very accurate results that represent the size of the search tree, we need to apply many probes and calculate the average.

Chapter IV

Empirical Evaluation

4 Empirical Evaluation

GHS is able to find a near-optimal and good heuristic subset to guide the A* search tree size and running time respectively, granted that it is able to compute the objective function of interest. Thus, the practical effectiveness of GHS depends on its ability of finding good approximations $\hat{J}$ and $\hat{T}$. In order to verify its practical effectiveness, we have implemented GHS in Fast Downward Helmert, (2006) and tested the A* performance using subsets of heuristics selected by GHS while minimizing different objective funcions.

We run two sets of experiments. In the first set we verify whether the approximations $\hat{J}$ and $\hat{T}$ provided by CS and SS allow GHS to make near-optimal and good subset selections for A* search tree size and running time respectively. In the second set of experiments we test the effectiveness of GHS by measuring the total number of problem instances solved by A* using a heuristic subset selected by GHS.

The GHS is executed up to adding another heuristic does not improve the objective function. In each iteration GHS greedily selects from ζ the heuristic h which will result in the largest reduction of the value of the objective function Ψ. We can not control the size of the resulting subset because, GHS stops if adding another heuristic h to the $\zeta^{'}$ does not improve the objective function and returns the current subset $\zeta^{'}$.

In all our experiments we use a type system that assigned the same type for a node with the same $f-$value. Such a type system has shown to be effective in guiding SS to produce accurate tree size predictions in other application domains Lelis et al., (2013), Lelis et al., (2014).

Table 1 – Ratios of the number of nodes expanded using $h_{max}(\zeta^{'})$ to the number of nodes expanded using $h_{max}(\zeta)$

Domain	SS		CS		$\|\zeta\|$	n
	Ratio	$\|\zeta^{'}\|$	Ratio	$\|\zeta^{'}\|$		
Barman	1.11	17.70	1.50	30.25	5168.50	20
Elevators	11.50	2.00	1.03	21.00	168.00	1
Floortile	1.02	43.07	1.01	42.35	151.28	14
Openstacks	1.00	1.00	1.00	1.00	390.69	13
Parking	1.00	5.52	1.01	7.26	21.73	19
Pegsol	1.00	31.00	1.00	57.00	90.00	2
Scanalyzer	1.22	30.57	1.56	19.42	72.85	7
Tidybot	1.00	2.35	1.00	8.58	3400.17	17
Transport	1.00	14.70	1.02	14.30	171.7	10
Visitall	1.02	99.33	1.18	48.66	256.33	3
Woodworking	32.42	3.00	199.65	5.00	1289.00	5

We ran our experiments on the 2011 International Planning Competition (`IPC`) instances. We used the 2011 instances instead of the 2014 instances because the former do not have problems with conditional effects, which are currently not handled by **PDB** heuristics. All experiments are run on 2.67 GHz machines with 4GB, and are limited to 1,800 seconds of running time.

4.1 Empirical Evaluation of $\hat{J}$ and $\hat{T}$

We test whether the approximation $\hat{J}$ provided by `CS` and `SS` allows `GHS` to make near-optimal subset selections. This test is made by comparing $J(\zeta^{'})$ with $J(\zeta)$, which is minimal. The condition $J(\zeta^{'}) \leq \alpha \cdot J(\zeta)$, is sufficient to show that $J(\zeta^{'})$ is within α times good with respect to all subsets of any size, for some constant α. We show empirically that $J(\zeta^{'})$ is within 10% of $J(\zeta)$.

In contrast with objective function J, there is no easy way to find the minimum of T for a subset in general. We experiment then with the special case in which all heuristics in ζ have the same evaluation time. This way we are able to test whether the estimates $\hat{T}$ are allowing `GHS` to make good subset selections while minimizing the A* running time. This is because by only selecting heuristics which have the same evaluation time, if `GHS` is making near-optimal subset selections with respect to J, then `GHS` must be making good subset selections with respect to T.

We collect values of $J(\zeta)$ and $J(\zeta^{'})$ as follows. For each problem instance ∇ in our test set we generate a set of `PDB` heuristics using the `GA-PDB` algorithm Edelkamp, (2007) as described by Barley et al., (2014) − we call each `PDB` generated by this method a `GA-PDB`. We chose to use `GA-PDBs` in this experiment because they all have nearly the same evaluation time and will allow us to verify whether `GHS` is making near-optimal and good selections when minimizing J and T respectively, as explained above. The number of `GA-PDBs` generated is limited in this experiment by 1,200 seconds and 1GB of memory. Also, all `GA-PDBs` we generate have 2 millions entries each. The `GA-PDBs` generated form our ζ set. `GHS` then selects a subset $\zeta^{'}$ of ζ. Finally, we use $h_{max}(\zeta^{'})$ and $h_{max}(\zeta)$ to independently try to solve ∇. We call the system which uses A* with $h_{max}(\zeta)$ the `Max` approach. For `GHS` we allow 600 seconds for selecting $\zeta^{'}$ and for running A* with $h_{max}(\zeta^{'})$, and for `Max` we allow 600 seconds for running A* with $h_{max}(\zeta)$. Since we used 1,200 seconds to generate the heuristics, both `Max` and `GHS` were allowed 1,800 seconds in total for solving each problem. In this experiment we test both `CS` and `SS`.

In this experiment we refer to the approach that runs A* guided by a heuristic subset selected by `GHS` using `CS` as `GHS+CS`. Similarly, we write `GHS+SS` when `SS` is used as predictor to make the heuristic subset selection.

Table 1 shows the average ratios of $J(\zeta^{'})$ to $J(\zeta)$ for both `SS` and `CS` in different

Table 2 – Coverage of SS, CS and Max on the 2011 IPC benchmarks. For GHS using only GA-PDBs heuristics.

Domain	SS	CS	Max
Barman	8	7	4
Elevators	19	19	19
Floortile	10	10	9
Nomystery	20	20	20
Openstacks	17	17	11
Parcprinter	17	15	14
Parking	1	1	1
Pegsol	19	19	19
Scanalyzer	10	10	10
Sokoban	20	20	20
Tidybot	14	13	11
Transport	14	14	14
Visitall	18	18	18
Woodworking	12	11	12
Total	199	194	182

problem domains. The value of J, for a given problem instance, is computed as the number of nodes expanded up to the largest $f-$layer which is fully expanded by all approaches tested (`Max, GHS` using SS and GHS using CS). We only present results for instances that are not solved during GHS's CS sampling process. The column "n" shows the number of instances used to compute the averages of each row. We also show the average number of GA-PDBs generated ($|\zeta|$) and the average number of GA-PDBs selected by GHS ($|\zeta'|$). This experiment shows that for most of the problems GHS, using CS or SS, is selecting near-optimal and a good subset of ζ for A* search tree size and running time respectively. For example, in Tidybot GHS selects only a few GA-PDBs out of thousands when using either SS or CS. Moreover, the resulting A* search tree size is on average at most 10% larger than optimal for GHS+SS, and is good for GHS+CS.

The exceptions in Table 1 are the ratios for Elevators, Scanalyzer and Woodworking. In Elevators SS has an average ratio of 11.50 and CS of 1.03. By looking at the ratios of SS for individual instances of Scanalyzer (results now show in Table 1), we noticed that SS is able to make good selections for all but 3 of the 7 instances considered in this experiment. Since we do not know a priori what is the instance's optimal solution cost, SS samples nodes with $f-$values much larger than the instance's optimal solution cost. We believe that, in this particular instance of Scanalyzer, by sampling a portion of the state space that is not expanded during the actual A*, SS is biasing the subset selection to select heuristics that do not contribute to reducing the actual A* search tree size.

The SS's ability of sampling deep into the search space is not always harmful. For example, SS allows GHS to make good selections for instances of the Woodworking domain. By contrast, CS's systematic approach to sampling only allow a shallow sample of the A* search tree. As a result, GHS makes a limited selection of heuristics to guide A* search. While GHS using SS selects an average of 3 heuristics in Woodworking instances, GHS using CS selects only an average of 5 heuristics. This difference on sampling strategies reflects on the number of problems solved by A*. While GHS+SS solves 12 instances of the Woodworking domain, GHS+CS solves only 11. In total, out of the 280 instances of the IPC 2011 benchmark set, GHS+SS solves 199 problem instances in this experiment, while GHS+CS only solves 194 problem instances. (The numbers of instances solved are shown in Table 2).

4.2 Comparison with Other Planning Systems

The objective of this second set of experiments is to test the quality of the subset of heuristics GHS selects while optimizing different objective functions. Our evaluation metric is coverage, I.E., number of problems solved within a 1,800 second time limit. We note that the 1,800−second limit includes the time to generate ζ, select ζ', and run A* using $h_{max}(\zeta')$. The ζ set of heuristics is composed of a number of different GA-PDBs, a PDB heuristic produced by the iPDB method Haslum et al., (2007) and the LM-Cut heuristic. The generation of GA-PDBs is limited by 600 seconds and 1GB of memory. We use one fourth of 600 seconds to genetate GA-PDBs with each of the following number of entries: $\{2 \cdot 10^3, 2 \cdot 10^4, 2 \cdot 10^5, 2 \cdot 10^6\}$. Our approach allows one to generate up to thousands of GA-PDBs. For every problem instance, we use exactly the same ζ set for Max and all GHS approaches.

4.3 Systems Tested

GHS is tested while minimizing the A* search tree size (Size) and the A* running time (Time). We also use GHS to maximize the sum of heuristic values in the state space (Sum), as suggested by Rayner et al., (2013). Rayner et al., (2013) assumed that one could uniformly sample states in the state space in order to estimte the sum of the heuristic values for a given heuristic subset. Since we are not aware of any method to uniformly sample the state space of domain−independent problems, we adapted the Rayner et al., (2013)'s method by using SS to estimate the sum of heuristic values in the search tree rooted at ∇'s start state. We write Size + SS to refer to the approach that used A* guided by a heuristic selected by GHS while minimizing an estimate of the search tree size provided by SS. We follow the same pattern to name the other possible combinations of objective functions and prediction algorithms (E.G.,Time+CS).

In addition to experimenting with all combination of prediction algorithms (CS and SS) and objective functions (Time, Size), we also experiment with an approach that minimizes both the search tree size and the running time as follows. First we create a pool of heuristics ζ composed solely of `GA-PDB` heuristics, then we apply `GHS` while minimizing tree size and using `SS` as predictor. As explained above, in this setting `GHS` minimizes J and T simultaneaously, as all heuristics in ζ have the same evaluation time. We call the selection of a subset of `GA-PDBs` as the *first selection*. Once the first selection is made, we test all possible combinations of the resulting $h_{max}(\zeta^{'})$ added to the `iPDB` and `LM-Cut` heuristics while minimizing the running time as estimated by `CS`—we call this step the *second selection*. We call the overall approach `Hybrid`.

The intuition behind `Hibrid` is that we apply `GHS` with its strongest settings. `GHS` makes near-optimal and good selections respect to J and T respectively when selecting from a pool of heuristics with the same evaluation time. After such a selection is made, we reduce the size of the pool of heuristics from possible thousands to only three (the maximum of a subset of the initial `GA-PDBs, iPDB`, and `LM-Cut`). With only three heuristics we are able to choose the exact combination that minimizes the A* running time the most. The reason we chose to use `SS` instead of `CS` for the first selection in `Hybrid` is that the former is able to make better subset selections in this setting, as suggested by the results discussed in the previous Chapter 3. Finally, as we show below, `CS` is more effective if one is interested in minimizing the A* running time while selecting from a pool of heuristic with different evaluation times. That is why we use `CS` as predictor for the second selection in `Hybrid`.

We compare the coverage of the `GHS` approaches with several other state—of—the—art planners. Namely, we experiment with RIDA* Barley et al., (2014), two variants of Stone-Soup (StSp1 and StSp2) as described by Nissim et al., (2011), two versions of Symba (SY1 and SY2), and A* being independently guided by the maximum of all heuristics in ζ (`Max`), `iPDB, LM-cut` and `Merge & Shrink`(`M&S`) Nissim et al., (2011).

The results are presented in Table 3. The results for the `GHS` approaches are averages computed over 10 independent runs of the planner; the average numbers are truncated to two decimal places in our table. The variance of the results is small, thus we omitted them from the table of results.

4.4 Discussion of the Results

The system that solves the largest number of instances is `Hybrid`— it solves 219 problems on average. As explained above, we combine in `Hybrid` the strengths of both `SS` and `CS` in a single system. `SS` is used to greedily select heuristics from a pool of heuristics with similar evaluation time, and only then `CS` is used for selecting heuristics with different evaluation times. This strategy has proven particularly effective on the Barman domain where `Hybrid`'s first selection is able to select near-optimal subsets of `GA-PDBs` and its second selection is able to recognize that it must not include the `iPDB` and `LM-Cut` heuristics to the subset selected by its first selection. As a result, `Hybrid` solves more problems on this domain than any other `GHS` approach.

`Time+CS` also performed well in our experiments—the approach solves 216 problems on average. Clearly `Hybrid` and `Time+CS` are far superior to all other approaches tested. For example, `Size + SS` and `Sum` solves only 206 and 207 problems, respectively. While minimizing the search tree size or maximizing the sum of heuristic values, `GHS` will tend to add accurate heuristics to the selected subset, independently of their evaluation time. As a result, if not minimizing the running time, `GHS` often adds the `LM-Cut` heuristic to ζ' as `LM-Cut` is often the heuristic that is able to reduce the most the search tree size and to increase the most the sum of heuristic values. However, `LM-Cut` is very computationally expensive, and in various cases the search is faster if `LM-Cut` is not in ζ'. Both `Hybrid` and `Time+CS` are able to recognize when `LM-Cut` should not be included in ζ' because they account for the heuristics' evaluation time.

Table 3 – Coverage of different planning systems on the 2011 `IPC` benchmarks. For the `GHS` and `Max` approaches we also present the average number of heuristics `GHS` selects ($|\zeta'|$).

Domains	`Hybrid`	`CS`		`SS`		`Sum`	RIDA*	SY1	SY2	StSp1	StSp2	`Max`	iPDB	LM-Cut	M&S
		`Time`	`Size`	`Time`	`Size`										
Barman	7	16	4	4	4	4	4	10	11	4	4	4	4	4	4
Elevators	19	14	19	19	19	19	19	20	20	18	18	19	17	18	12
Floortile	14	15	14	14	14	14	14	14	14	14	14	14	8	14	10
Nomystery	20	19	20	19	20	20	20	16	16	20	20	20	19	14	18
Openstacks	17	19	15	17	15	15	15	20	20	17	17	11	17	15	17
Parcprinter	18	14	15	16	16	19	18	17	17	18	18	18	16	17	16
Parking	7	20	2	7	2	2	7	2	1	5	5	2	7	2	7
Pegsol	19	20	19	19	19	19	19	19	20	19	19	19	20	17	19
Scanalyzer	14	18	13	11	14	14	14	9	9	14	14	14	10	12	11
Sokoban	20	19	20	20	20	20	20	20	20	20	20	20	20	20	20
Tidybot	16	4	16	16	17	16	17	15	17	16	16	15	14	16	9
Transport	14	14	13	11	13	11	10	10	11	7	8	9	8	6	7
Visitall	18	7	17	15	17	18	18	12	12	16	16	18	16	10	16
Woodworking	16	17	16	12	16	16	15	20	20	15	15	16	9	15	9
Total	219	216	203	200	206	207	210	204	208	203	204	199	185	180	175

Note that the difference on the number of problems solved by `Time+CS` and `Time+SS`: While the former solves 216 instances, the latter solved only 200. We conjecture that this happens because `SS` is not able to detect duplicated nodes during sampling. As a result, `SS` often overestimates by several orders of magnitude the actual A*'s running time. Similarly to the `Size` and `Sum` approaches, due to `SS`'s overestimations, `Time+SS` often mistakenly adds the accurate but expensive `LM-Cut` heuristic in cases where the A* search would be faster without `LM-Cut`'s guidence. For example, although `iPDB` tends to prune fewer nodes than `LM-Cut` in Parking instances, `iPDB` is the heuristic of choice in that domain. This is because its evaluation time is much smaller than `LM-Cut`'s. `Time+CS` solves 20 Parking instances on average as it correctly selects `iPDB` and leaves `LM-Cut` out of ζ'. By contrast, likely due to its prediction overestimation, `Time+SS` solves 7 parking instances because wrongly estimates `LM-Cut` that will reduce overall search time and adds the heuristic to its selected subset. Notice, that `Size+CS` and `Size+SS` also does poorly Parking instances as they also always select `LM-Cut`.

RIDA* is the most similar system to `GHS`, as it also selects a subset of heuristics from a pool of heuristics by using an evaluation method similar to `CS`. RIDA* uses a systematic approach for selecting a subset of heuristics. Namely, it starts with an empty subset and evaluates all subsets of size i before evaluating subsets of size $i + 1$. This procedure allows RIDA* to consider only tens of heuristics in their pool. By contrast, `GHS` is able to consider thousands of heuristics while making its selection.

The ability to handle large set of heuristics can be helpful, even if most of the heuristics in the set are redundant with each other—as is the case with the `GA-PDBs`. The process of generating `GA-PDBs` is stochastic, thus one increases the chances of generating helpful heuristic by generating a large number of them. `GHS` is an effective method for selecting a small set of informative heuristics from a large set of mostly uninformative ones. This is illustrated in Table 3 on the Transport domain. Compared to systems which use multiple heuristics (StSp1 and 2, and RIDA*), `Time+CS` solves the largest number of Transport instances, which is due to the selection of a few key `GA-PDBs`.

The best `GHS` approach, `Hybrid`, substantially outperforms the number of instances solved by `Max`−`Hybrid` solves on average more than 20 instances than `Max`. Finally, `Hybrid` and `Time+CS` substantially outperforms all other approaches tested, with RIDA* being the closest competitor with 210 instances solved.

4.5 Comparison between SS and IDA*

In this experiment `SS` estimates the search tree size generated by IDA* using a consistent heuristic. `SS` estimates the size of the search tree up to some defined deep f-layer in the tree.

We first ran IDA* for Fast-Downward benchmark for optimal domains. Our evaluation metric is coverage, `i.e.,` number of problems solved within 30 minutes time limit. We note that in 30 minutes non all the instances for a specific domain using a consistent heuristic can be solved. Afterwards, run `SS` using as a threshold the $f-$layer limited by the search time, this process is executed using different number of probes `i.e.,` 1, 10, 100, 1000 and 5000.

$$\frac{\sum_{s \in PI} \frac{Pred(s,d) - R(s,d)}{R(s,d)}}{|PI|} \tag{4.1}$$

Table 4 – Comparison between SS and IDA* for 1, 10, 100, 1000 and 5000 probes using *hmax* heuristic.

Domain	*hmax*												
	IDA*	time	relative−error					time					n
			1	10	100	1000	5000	1	10	100	1000	5000	
Barman	8835990.00	6016.38	0.60	0.45	0.20	0.07	0.04	0.06	0.32	3.21	32.57	214.59	20
Elevators	1012570.00	4987.57	0.84	0.42	0.23	0.13	0.10	1.40	9.85	96.37	994.33	4425.93	20
Floortile	30522300.00	3919.72	2.02	0.62	0.40	0.14	0.11	0.01	0.07	0.69	6.93	36.60	2
Nomystery	6565740.00	3256.86	0.53	0.26	0.07	0.03	0.01	0.07	0.38	3.63	36.35	181.03	20
Openstacks	80108.50	4017.19	0.03	0.03	0.03	0.03	0.03	94.79	774.86	1067.84	10929.00	11174.30	20
Parcprinter	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.04	0.35	3.48	17.29	20
Parking	374925.00	5607.50	0.17	0.04	0.01	0.00	0.00	1.79	11.36	114.28	1196.83	5835.03	20
Pegsol	68763.70	5.00	0.17	0.04	0.02	0.01	0.00	0.01	0.04	0.37	3.69	17.88	20
Scanalyzer	8449890.00	4920.58	0.43	0.25	18.63	0.02	0.01	3.13	28.79	273.74	3033.06	10254.00	20
Sokoban	3118530.00	3932.69	0.41	0.26	0.11	0.05	0.04	0.31	2.00	21.42	222.47	1056.61	20
Tidybot	444473.00	5632.08	300.86	1072.40	5.88	0.01	0.01	4.40	26.48	238.76	2747.10	11925.40	20
Transport	2622880.00	2253.51	0.63	0.54	0.24	0.15	0.11	0.09	0.61	5.89	59.37	290.31	20
Visitall	71032400.00	3704.78	0.12	0.04	0.01	0.00	0.00	0.00	0.05	0.56	5.77	28.07	20
Woodworking	5139070.00	4944.76	1.28	0.69	0.27	0.17	0.07	0.15	1.33	13.21	130.82	664.08	20

The Formula 4.1 is called *relative unsigned error* by Lelis et al., (2012) (we simplify it by relative-error). This help us to determinate the accuracy of the prediction of SS about IDA* given a state s and cost bound d. The parameters that this Formula require are:

- PI represent all the instances for a domain.
- $Pred(s, d)$ is the estimation of the search tree size expanded by IDA* for start state s and cost bound d.
- $R(s, d)$ is the real number of nodes expanded by IDA* for start state s and cost bound d.

The result from apply the relative-error is a number, that in order to obtain a good approximation, the result must be near to 0.00, which represents the perfect score.

The Table 4 shows how the relative-error behavies when SS makes prediction of the number of nodes expanded by IDA* when it is searching with a specific heuristic and cost threshold. The heuristic used in this experiment is *hmax*. Five probes were used: 1, 10, 100, 1000 and 5000. The average value of IDA* and time were used. The relative-error gets a perfect score while increasing the number of probes. For Barman, the relative-error goes from 0.60 for 1 probe to 0.45 for 10 probes, 0.20 for 100 probes, 0.07 for 1000 probes and 0.04 for 5000 probes. In the case of time, while the number of probes increase, SS need to spend more time calculating the size of the search tree. Then, the time increase. For Barman, the time goes from 0.06 *seconds* for 1 probe to 0.32 *seconds* for 10 probes, 3.21 *seconds* for 100 probes, 32.57 *seconds* for 1000 probes and 214.59 *seconds* for 5000 probes. There are domains such as: Parcprinter, Parking, Pegsol and Visitall that have perfect score using 5000 probes. In the case of Tidybot, the relative-error using 1 probe is smaller than using 10 probes. The reason might be the search tree generated for some instances or the stochastic behavior of SS that sometimes it will choose a node that expand a search tree that will be more expensive to expand. The last column `n` represent the number of instances where IDA* found the number of nodes expanded when it is searching with *hmax* and cost threshold. The 2011 `IPC` domains contains 20 instances per domain. Floortile only have 2 instances, it means that when running IDA* for all the instances of Floortile only two instances (opt-p01-001.pddl and opt-p03-006.pddl) have found number of nodes expanded under some threshold. In summary, we proved that for 2011 `IPC` domains, `SS` estimations converges to the real search tree size generated by IDA* when the number of probes goes to infinity.

4.6 Comparison between SS and A*

The Table 5 shows that `SS` is not a good estimator for A* and that is because `SS` does not count for duplicate nodes and A* does. `SS` overestimate the A* search tree size. As a result, `SS` often gets values by several orders of magnitude bigger than the actual A* search tree.

Table 5 – Poor prediction of SS against A* using ipdb, `LM-Cut` and M&S with 500 probes

Domain	ipdb		`LM-Cut`		M&S		n
	A*	SS−error	A*	SS−error	A*	SS−error	
Barman	1.72e+07	8.68e+31	7.45e+06	2.21e+30	6.67e+06	1.26e+36	4
Floortile	1.40e+07	1.74e+18	702435	4.68e+14	4.46e+06	1.90e+12	4
Nomystery	40169.7	6.71e+32	267100	6.14e+19	8236	1.20e+20	9
Openstacks	570099	0.61884	570099	0.677425	569984	0.672143	4
Parcprinter	1157	2.56e+22	1363.67	2.33e+21	766.333	6.36e+20	3
Pegsol	841693	2901.39	398221	6859.86	933430	779.017	16
Scanalyzer	337894	3.94e+33	334747	7.58e+31	337833	2.42e+31	3
Sokoban	376755	1.04e+07	45374	2.74e+06	739775	5.60e+08	9
Transport	1.89e+06	2.91e+38	1.49e+06	1.15e+25	1.73e+06	1.50e+29	2
Visitall	253710	1.69e+46	253195	1.69e+46	253521	1.71e+46	8
Woodworking	3.21e+06	2.53e+18	3.20e+06	2.76e+18	3.21e+06	2.48e+18	3

Three heuristics were used: ipdb, `LM-Cut` and M&S. The last column `n` represent the number of instances solved by A* using the three heuristics. For this experiment we decided to use only the instances that are solved by the three heuristics at the same time. The columns with A* represents the average of number of nodes expanded by A* for the instances solved using a specific heuristic. The column with `SS`-error represents the relative-error Formula 4.1 applied to the solved instances.

For Barman: Using M&S, A* expands in average 6.67e+06 which is less nodes than ipdb−1.72e+07 and `LM-Cut`−7.45e+06. However, using M&S, `SS`−error is 1.26e+36, ipdb−8.68e+31 and `LM-Cut`−2.21e+30, which indicates that the number of nodes expanded by `SS` in average is in the order of magnitude from 30 to 40. In Visitall, `SS`-error shows that `SS` overestimates A* highly, which represent a very bad prediction of `SS`. In Openstack: Using the three heuristics, A* expands almost the same number of nodes for the 4 instances solved, and `SS`-error shows a score near to the perfect and the reason is that `SS` expands less nodes than A*.

Looking in the instances solved, the number of nodes expanded by SS using ipdb, LM-Cut and M&S for the 4 instances of Openstack are close to the number of nodes expanded by A* using the same heuristics. For example, for the first instance (p01.pddl) using ipdb SS expands 10977 and A* expands 4862.23, using LM-Cut SS expands 10977 and A* expands 2729.12, using M&S SS expands 10977 and A* expands 2729.12. The same behavior is observed in other instances. In consequence, applying relative-error Formula the result will tend to be the perfect score for this domain.

4.7 Approximation Analysis for SS and A*

Here we show that SS is able to make near-optimal selection of heuristics ipdb, LM-Cut and GA-PDBs with respect to the A* search tree size.

Even though, SS has poor prediction system for A*, as the Table 5 says, we have used SS in our utility function and observed that we get to solve many instances for optimal domains. In this experiment we analize the approximation of SS against A*.

Moreover, in order to understand how SS and A* behaves we have created plots with the fixed range of 2. This way we are going to have 4 different regions as shown in the Figure 9 . The points represent the fraction between the number of nodes expanded by A* using a heuristic i ($J(h_i)$), and the estimate of the number of nodes expanded by SS ($\hat{J}(h_i)$). Points on regions II and III are heuristics that SS correctly choose to be used with A*. Points following on the other regions are those choices, SS made incorrectly. For abscissa y in the cartesian plane, which is represented by A* ratios: $J(h_1) > J(h_2)$ means that A* expands fewer nodes if using h_2.

Points that fall in each of the regions:

- I $J(h_1) > J(h_2)$ for A*, $\hat{J}(h_2) > \hat{J}(h_1)$ according to SS.
- II $J(h_1) > J(h_2)$ for A* and SS agrees.
- III $J(h_2) > J(h_1)$ for A* and SS agrees.
- IV $J(h_2) > J(h_1)$ for A*, $\hat{J}(h_1) > \hat{J}(h_2)$ according to SS.

In the Figure 10 we can see the distribution of the points in each domain. We use three different heuristics ratios: ipdb, LM-Cut(lmcut) and 10 GA-PDBs(gapdb) which are represented by the symbols ■ , ● and ▲ respectively. The ipdb ratio is the result of divide two search tree size generated by h_1 = ipdb and h_2 = ipdb. The LM-Cut ratio is the result of divide two search tree size generated by h_1 = LM-Cut and h_2 = LM-Cut. When at least one heuristic h_1 or h_2 is GA-PDB then the heuristic ratio is GA-PDB. This experiment was done using 5000 probes for SS and during 30 minutes.

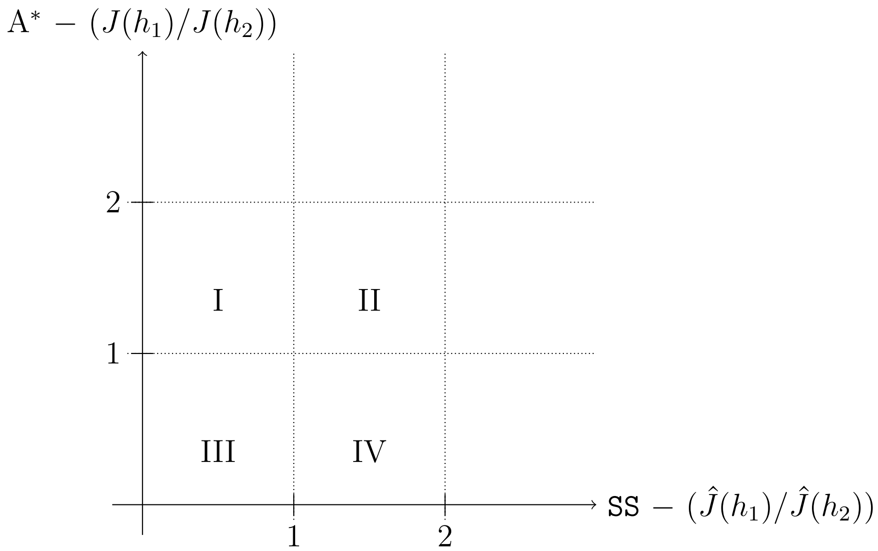

Figure 9 – Cartesian Plane with domain $\langle 0, 2\rangle$ and range $\langle 0, 2\rangle$

The eleven plots displayed show the distribution of the heuristic ratios in the four quadrants. We drew the function $y = x$ because SS yields a perfect fraction if the point fall in that function. That is why it is important to have such a line as a reference on the plot. We decided to use the same scale on both axis, otherwise it will be hard to see which points fall on the diagonal line ($y = x$) and which points don't. Furthermore, the scale is 2 for both axis. As a result, the points that are far away from the quadrants, are set to be in the limit. For example, if any of the ratios r is larger than $1,000$ then r will be on the 2 border of the plot.

The points $(0, 0), (1, 1), (2, 2)$ mean that SS made a perfect choice and these points represent the function $y = x$. As we are interested in the percentage of points that fall in the quadrants II and III we are going to consider the border only for those quadrants.

The points are dispersed in the positive quadrant of the cartesian plane because we do not have negative number of nodes generated by h_1 or h_2. In the case of Tidybot and Woodworking all the points are in the quadrant II or III.

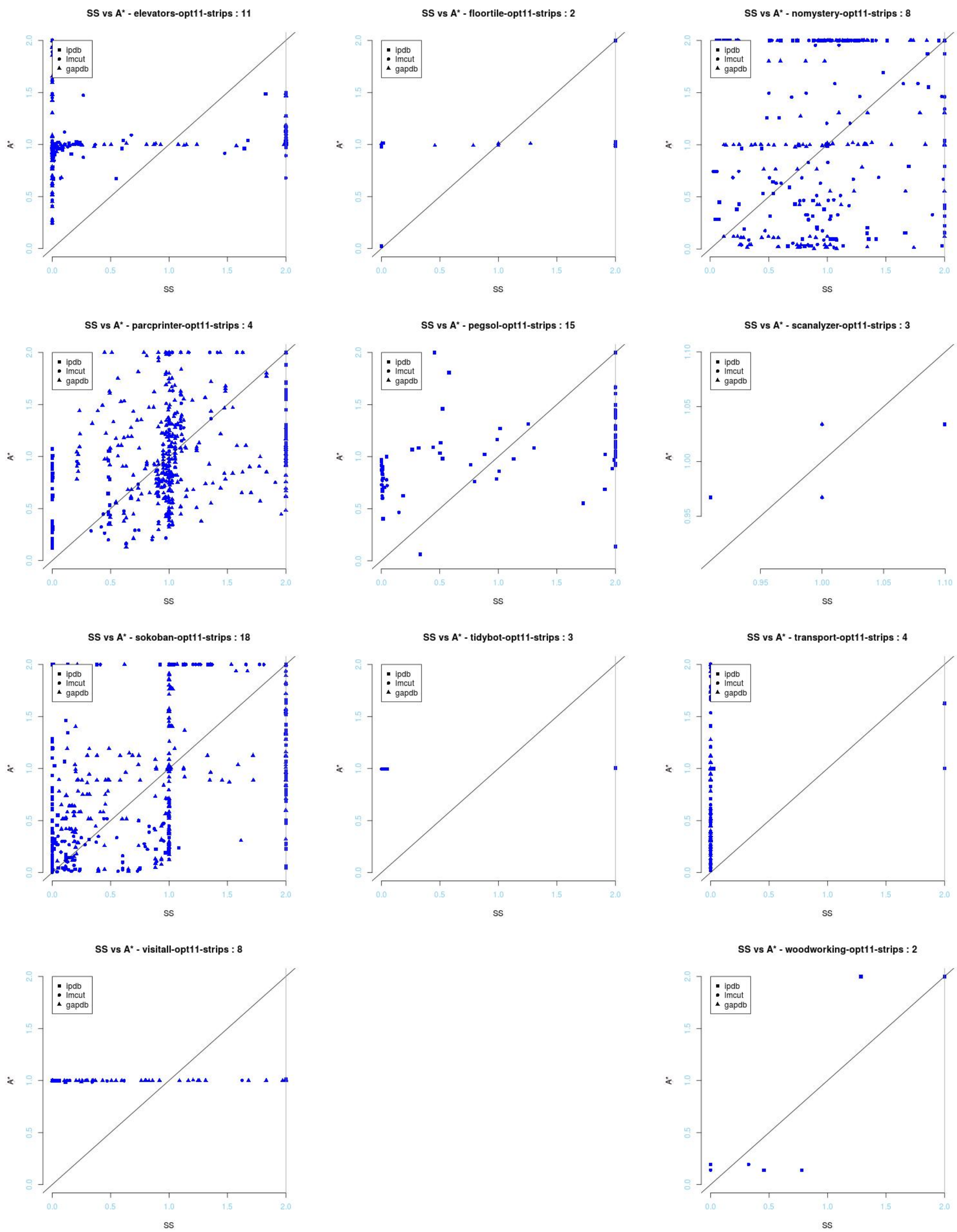

Figure 10 – SS vs A* ratios for the optimal domains – The number of instances used in each domain are showed next to the name of the Domain.

In Table 6 we present a single number for each domain representing the percentage of choices SS make correctly. From all the domains all are above of the 50% which means that at least the half of the points represent good relation of heuristics. With this experiment we prove that SS is not as bad as we thought it would be when we want to make selection.

Domain	II and III (%)
Elevators	78.57
Floortile	96.08
Nomystery	71.82
Parcprinter	70.50
Pegsol	96.83
Scanalyzer	100.00
Sokoban	89.31
Tidybot	100.00
Transport	51.78
Visitall	98.05
Woodworking	100.00

Table 6 – Percentage of choices SS made correctly.

Chapter V

Conclusion

5 Concluding Remarks

This dissertation showed that the problem of finding the optimal subset of a set of heuristics ζ for a given problem task is solved using the models of A* search tree size and, under mild assumptions, with respect to the A* running time. Thus, the `GHS` algorithm which selects heuristics from ζ one at a time is able to produce a subset ζ' such that the number of nodes expanded by A* while guided by the heuristics ζ' is 10% optimal. Furthermore, if all heuristics in ζ have the same evaluation time, then we have a good subset selection with respect to running time. In addition to minimizing the search tree size and the running time, we also experimented with an objective function that accounts for the sum of heuristic values in the state-space, as suggested by Rayner et al., (2013).

Since we cannot compute the values of the objective functions exactly, `GHS` effectiveness depends on the quality of the approximations we can obtain. We tested two prediction algorithms, `CS` and `SS`, for estimating the values of the objective functions and showed empirically that both `CS` and `SS` allow `GHS` to make near-optimal and good subset selections with respect to the search tree size and running time respectively.

Finally, experiments on optimal domain-independent problems showed that `GHS` minimizing approximations of the A* running time outperformed all the other approaches tested, which demostrates the effectiveness of our method for the heuristic subset selection problem.

Bibliography

BÄCKSTRÖM, C.; NEBEL, B. Complexity results for sas+ planning. *Computational Intelligence*, Wiley Online Library, v. 11, n. 4, p. 625–655, 1995. Cited on page 29.

BARLEY, M. W.; FRANCO, S.; RIDDLE, P. J. Overcoming the utility problem in heuristic generation: Why time matters. *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*, 2014. Cited 7 times in the pages 24, 29, 32, 38, 39, 48, and 51

CHEN, B. *Part2 AI as Representation and Search*. 2011. <http://www.slideshare.net/praveenkumar33449138/ai-ch2>. Accessed: 2016-24-01. Cited 3 times in the pages 13, 22, and 23

CHEN, P.-C. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, p. 295–315, 1992. Cited 4 times in the pages 29, 38, 40, and 42

CULBERSON, J. C.; SCHAEFFER, J. Pattern databases. *Computational Intelligence*, Wiley Online Library, v. 14, n. 3, p. 318–334, 1998. Cited 3 times in the pages 24, 31, and 32

DOMSHLAK, C.; KARPAS, E.; MARKOVITCH, S. To max or not to max: Online learning for speeding up optimal planning. In: *AAAI*. [S.l.: s.n.], 2010. Cited on page 24.

EDELKAMP, S. Automated creation of pattern database search heuristics. In: *Model Checking and Artificial Intelligence*. [S.l.]: Springer Berlin Heidelberg, 2007. p. 35–50. Cited 2 times in the pages 24 and 48

HART P. E.; NILSSON, N. J.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC*, IEEE, v. 4, n. 2, p. 100–107, 1968. Cited 3 times in the pages 21, 24, and 30

HASLUM, P. et al. Domain-independent construction of pattern database heuristics for cost-optimal planning. *AAAI*, v. 7, p. 1007–1012, 2007. Cited 3 times in the pages 24, 25, and 50

HELMERT, M. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)*, v. 26, p. 191–246, 2006. Cited 2 times in the pages 25 and 47

HOLTE, R. C. et al. Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence*, Elsevier, v. 170, n. 16, p. 1123–1136, 2006. Cited 4 times in the pages 11, 21, 24, and 33

LELIS, L.; ZILLES, S.; HOLTE, R. C. Fast and accurate predictions of ida*'s performance. In: *AAAI*. [S.l.: s.n.], 2012. Cited on page 54.

LELIS, L. H.; OTTEN, L.; DECHTER, R. Memory-efficient tree size prediction for depth-first search in graphical models. In: *Principles and Practice of Constraint Programming*. [S.l.]: Springer International Publishing, 2014. p. 481–496. Cited on page 47.

LELIS, L. H.; STERN, R.; STURTEVANT, N. R. Estimating search tree size with duplicate detection. In: *Seventh Annual Symposium on Combinatorial Search*. [S.l.: s.n.], 2014. Cited 2 times in the pages 29 and 42

LELIS, L. H.; ZILLES, S.; HOLTE, R. C. Predicting the size of ida* search tree. *Artificial Intelligence*, Elsevier, v. 196, p. 53–76, 2013. Cited 4 times in the pages 29, 40, 43, and 47

NISSIM, R.; HOFFMANN, J.; HELMERT, M. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In: *22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*. [S.l.: s.n.], 2011. Cited 2 times in the pages 24 and 51

RAYNER, C.; STURTEVANT, N.; BOWLING, M. Subset selection of search heuristics. *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, p. 637–643, 2013. Cited 4 times in the pages 29, 37, 50, and 63

TOLPIN, D. et al. Towards rational deployment of multiple heuristics in a*. In: AAAI PRESS. *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*. [S.l.], 2013. p. 674–680. Cited on page 24.