Marvin Abisrror Zarate

On Selecting Of Heuristics Functions For Domain Independent Planning.

Brasil

2015, v-1.9.5

Marvin Abisrror Zarate

On Selecting Of Heuristics Functions For Domain Independent Planning.

Dissertação apresentada á 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

Supervisor: Levi Henrique Santana de Lelis

Co-supervisor: Santiago Franco

Brasil

2015, v-1.9.5

Marvin Abisrror Zarate
On Selecting Of
Heuristics Functions For Domain Independent Planning./ Marvin Abisrror Zarate. – Brasil, 2015, v-1.9.5-
76 p. : il. (algumas color.) ; 30 cm.

Supervisor: Levi Henrique Santana de Lelis

Tese (Mestrado) – Universidade de Viçosa – UFV
Centro de Ciencias Exactas e Tecnologicas (CCE)
Programa de Pós-Graduação, 2015, 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 Of Heuristics Functions For Domain Independent Planning.

Dissertação apresentada á 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
2015, v-1.9.5

This thesis 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 thesis 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 DTI, 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 based on the theory of supermodular optimization 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. (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 the A* search algorithm. If the heuristics are consistent, our method selects a subset which is guaranteed to be near optimal with respect to the resulting A* search tree size. In addition to being consistent, if all heuristics have the same evaluation time, our subset is guaranteed to be near optimal 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 heuristics in the International Planning Competition benchmarks.

Key-words: Heuristics. selection. Submodularity. A*

List of Figures

List of Tables

Contents

Introduction

State space search algorithms have been used to solve important real−world problems, such as: Robotics, domain-independent planning, chemical compounds discovery, bin packing, sequence alignment, automating layouts of sewers, and network routing, amount 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 search algorithm.

Chapter I

Preparation of the research

1 About the Problem

1.1 Search Space and Search Tree Formulation

(LELIS; ZILLES; HOLTE, 2013) defines the search space and search tree in the following way: Let the *underlying search tree* (*UST*) be the full brute−force tree created from a connected, undirected and implicitely defined *underlying search graph* (*USG*) describing a state space. Some search algorithms expand a subtree of the *UST* while searching for a solution (e.g., a portion of the *UST* might not be expanded due to heuristic guidance); we call this subtree the *expanded search tree (EST)*.

In this dissertation our methods require to estimate the size of the subgraph expanded by an algorithm searching the *USG*; and the subgraph expanded recieve the name of expanded search graph. *ESG*.

Let $G = (N, E)$ be a graph representing an *ESG* where N is its set of states and for each $n \in N$ $op(n) = op_i|(n, n_i) \in E$ is its set of operators. The term edges and operators are used interchangeably and work as a successor function which recieves as input a state and action and returns another state.

Many search algorithms, such as IDA* (KORF, 1985), expand a tree while searching the state space. By contrast, A* using a consistent heuristic expands a graph while searching the state space. The distinction between trees and graphs is important: Multiple nodes in a search tree might represent the same state in a search graph. In consequence, the tree might be substantially larger than the graph.

The purpose of using a search algorithm is to solve the problem or find the solution. We are insterested in finding a sequence of actions that goes from the start state to the goal state. To solve the problem is required the use of search algorithms. Two well know search algorithms are Depth First Search (`DFS`) and Breadth First Search (`BFS`). `DFS` looks for the solution traversing the search space exploring the nodes in each branch before backtracking up to find the solution. `BFS` looks for the solution exploring the neighbors nodes first, before moving to the next level of neighbors. Both search algorithms have the characteristic that generate a larger search space during the search. The search space that these algorithms generate are called Brute force search tree (`BFST`).

There are other type of algorithms called heuristic search algorithms, which are algorithms that requires the use of heuristics. The heuristic is the estimation of the distance for one node in the search tree to get the goal state. The heuristic search algorithms generate smaller search tree 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 to create heuristics, such as: Pattern Databases (PDBs) (HASLUM et al., 2007), Neural Network, and Genetic Algorithm (EDELKAMP, 2007). We call these systems Heuristic Generators. And one of the approaches that have showed most successfull results in heuristic generation is the PDBs, which are memory−based heuristic functions obtained by abstracting away certain problem variables, so that the remaining problem "patten‘' is small enough to be solved optimally for every state by blind exhaustive search. The results of such a blind search are stored in a table which represents a heuristic function for the original problem.

There exists many ways to take advantage of a large set of heuristic functions. For example: (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; KARPAS; MARKOVITCH, 2010) and (TOLPIN et al., 2013) showed that evaluating the heuristic lazily, only when they are essencial to a decision to be made in the search process is worthy in comparison to take the maximum of the set of heuristics. Then, using all the heuristics do not guarantees to solve the major number of problems in a limit time.

1.2 Aim and Objectives

1.2.1 Aim

The objective of this dissertation is to develop meta-reasoning approaches for selecting heuristics functions 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

- Demostrate that the problem of finding the optimal subset of ζ of size N for a given problem task is supermodular respect the size of the search tree.
- Develop an approach to find a subset of heuristics from a large pool of heuristics that optimize the number of nodes expanded in the process of search.

- Develop an approach for selecting a subset of heuristics functions based on the size of the search tree and running time.
- Compare SS algorithm for predicting the search tree size of Iterative-Deepening A* (IDA*).
- Use SS as our utility function.

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

In the last few decades, Artificial Intelligence has made significant strides in domain-independent planning. The use of heuristic search approach have contribuited to problem solving, where the use of an appropriate heuristic often means substantial reduction in the time needed to solve hard problems.

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

We believe that our idea of selecting heuristics using the size of the search tree and the running time will help us to solve more problems.

1.5 Hypothesis

We test the following hypothesis:

- **H1:** Test that the greedy algorithms are effective for selecting a subset of heuristics to guide search.
- **H2:** Test that SS is an effective prediction method to our meta-reasoning.

1.6 Contribution of the Dissertation

The main contributions of this Thesis are:

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

1.7 Organization of the Dissertation

The Thesis is organized as follows:

1. In Chapter I, the background of the thesis is provided which also includes our motivation and define the scope.
2. In Chapter II, we review the state of the art in selection of heuristic functions.
3. In Chapter III, we introduce Random Greedy Heuristic Selection (`RGHS`) and the prediction methods.
4. In Chapter IV, we explain the results obtained by using `GRHS` 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

An optimization procedure which is similar to ours is presented by (RAYNER; STURTEVANT; BOWLING, 2013), but their procedure maximizes the average heuristic value. By contrast, the meta−reasoning we are proposing minimizes the search tree size.

Our meta−reasoning requires a prediction of the number of nodes expanded by A* using any given subset. Although there are methods for accurately predicting the number of nodes expanded by Iterative Deepening−A* (KORF, 1985) (IDA^*). (SS system (LELIS; ZILLES; HOLTE, 2013)), these methods can't be easily adapted to A* because A*'s duplicate pruning makes it very difficult to predict how many nodes will occur at depth d of A^*'s search tree (the tree of nodes expanded by A^*). As a part of our proposal, we present SS for predicting the size of the search tree.

The system most similar to ours is RIDA* (BARLEY; FRANCO; RIDDLE, 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 all combination of size one before trying the combination of size two and so on. RIDA* stops after evaluating a fixed number of subsets. While RIDA* is able to evaluate a set of heuristics with tens of elements, our meta−reasoning is able to evaluate a set of heuristics with thousands of elements.

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^*

The state space problem illustrated in the figure 1 is a game that consists of a frame of numbered square tiles in random order with one tile missing. The puzzle also exists in other sizes, particularly the smaller 8−puzzle. If the size is 3×3 tiles, the puzzle is called the 8−puzzle or 9−puzzle, and if 4×4 tiles, the puzzle is called the 15−puzzle or 16−puzzle named, respectively, for the number of tiles and the number of spaces. The object of the puzzle is to place the tiles in order by making sliding moves that use the empty space.

The legal operators are to slide any tile that is horizontally or vertically adjacent to the blank into the blank position. The problem is to rearrange the tiles from some random initial configuration into a particular desired goal configuration. The 8−puzzle contains 181,440 reachable states, the 15−puzzle contains about 10^{13} reachable states, and the 24−puzzle contains almost 10^{25} states.

Initial

4	1	2
8		3
5	7	6

Goal

1	2	3
4	5	6
7	8	

Figure 1 – The left tile−puzzle is the initial distribution of tiles and the right tile−puzzle is the goal distribution of tiles. Each one represent a state.

Instead of using an algorithm of Brute force search that will analyze all the possible solutions. We can obtain heuristics from the problem of the slide tile puzzle that will help us to solve the problem.

2.3 Heuristics

State−space algorithms, such as A* (HART P. E.; NILSSON; RAPHAEL, 1968), are important in many `AI` applications. A* uses the $f(s) = g(s) + h(s)$ cost function to guide its search. Here, $g(s)$ is the cost of the path from the start state s, and $h(s)$ is the estimated cost−to−go from s to a gial; $h(.)$ is known as the heuristic function. The heuristic is the mathematical concept that represent to the estimate distance from the node s to the nearest goal state.

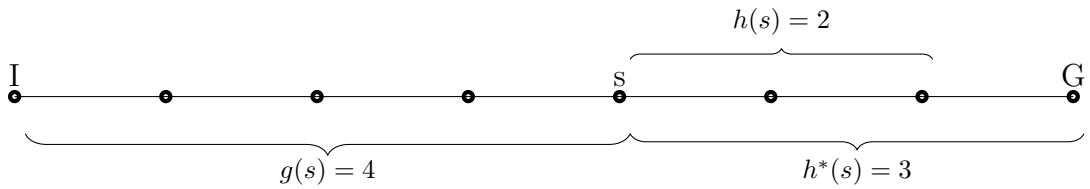

Figure 2 – Heuristic Search: I: Initial State, s: Some Sate, G: Goal State

In the figure 2 the optimal distance from the Initial State I to the state s is 4 and represented by $g(s)$. The $h^*(s)$ represent the optimal distance from s to the Goal State G. And the $h(s)$ 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 consisten 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; 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 the cost of the path expanded from I to s. $g(s)$ is minimal when A* using a consistent heuristic expands s. We call an A* search tree the tree defined by the states expanded by A* using a consistent heuristic while solving a problem ∇.

The heuristics can be obtained from each state of the problem. For example, for the problem of the 8−tile−puzzle figure 1 we can get two heuristics.

2.3.1 Out of place (O.P)

Counts the number of objects out of place.

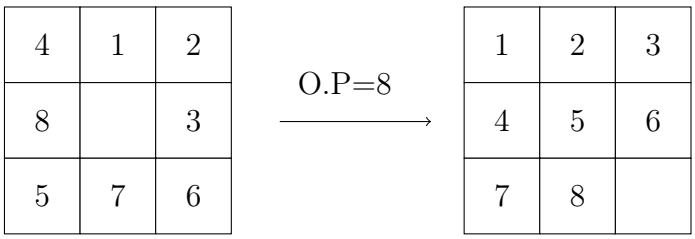

Figure 3 – Out of place heuristic

The tiles numbered with 4, 1, 2, 3, 6, 7, 5, 8, and 4 are out of place then each object count as 1 and the sum would be 8.

2.3.2 Manhatham Distance (M.D)

Counts the minimum number of operations to get to the goal state.

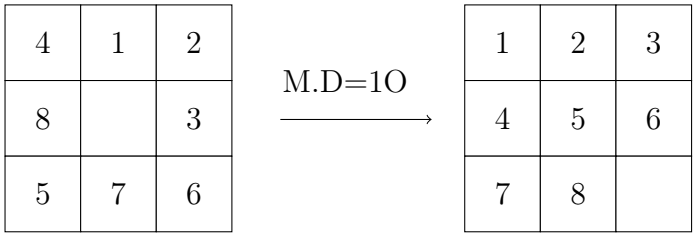

Figure 4 – Manhatham distance heuristic

The tile 4 count 1 to get to the goal position. The tile 1 count 1 to get to the goal position. The tile 2 count 1 to get to the goal position. The tile 3 count 1 to get to the goal position. The tile 6 count 1 to get to the goal position. The tile 7 count 1 to get to the goal position. The tile 5 count 1 to get to the goal position. The tile 8 count 1 to get to the goal position. Then the sum would be 10.

In order to solve the problem, we get the heuristics, which are information from the problem to solve the problem. Exists systems that can create heuristics for each problem. Those systems are called Heuristic Generators.

2.4 Heuristic Generators

Heuristic Generators works by creating abstractions of the original problem space. The approach that has showed more successful results lately is `PDB`.

2.4.1 Pattern Database (PDB)

It's obtained by abstracting away certain problem variables, so that the remaining problem ("pattern") is small enough to be solved optimally for every state by blind exhaustive search. The results stored in a table, represent a PDB for the original problem. The abstraction of the search space gives an admissible heuristic function, mapping states to lower bounds.

2.5 Take advantage of Heuristics

The heuristics generators can create hundreds or even thousand of heuristics. In fact, exists different ways to take advantage of those heuristics. For example: If we want to use all the heuristics created by the heuristic generator. It would not be a good idea to use all of them because the main problem involved would be the time to evaluate each

heuristic in the search tree, it could take too much time.

One way to take advantage of heuristics would be to take the maximum of the set of heuristics. 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 $h2$.

Exists different approaches to take advantage from a large set of heuristics. In this dissertation we use the meta−reasoning based on the minimum size of the search tree generated and the minimum evaluation time.

2.6 Number of heuristics created

Let's suppose we have to run our meta−reasoning using M amount of memory available. The question would be: How many heuristics our system should handle in order to avoid out of memory errors? So. one of the objectives of this tesis is to find the number of heuristics that our subset $\zeta^{'}$ should have.

(HOLTE et al., 2006) observed that maximizing over N pattern databases of size M/N, for a suitable choice N, produces a significant reduction in the number of nodes generated compared to using a single pattern database of size M.

In the Figures 6 and 7 we are taking advantange of the heuristics doing the maximization of all the heuristics created. In this thesis we are interested in heuristics that generate smaller search tree.

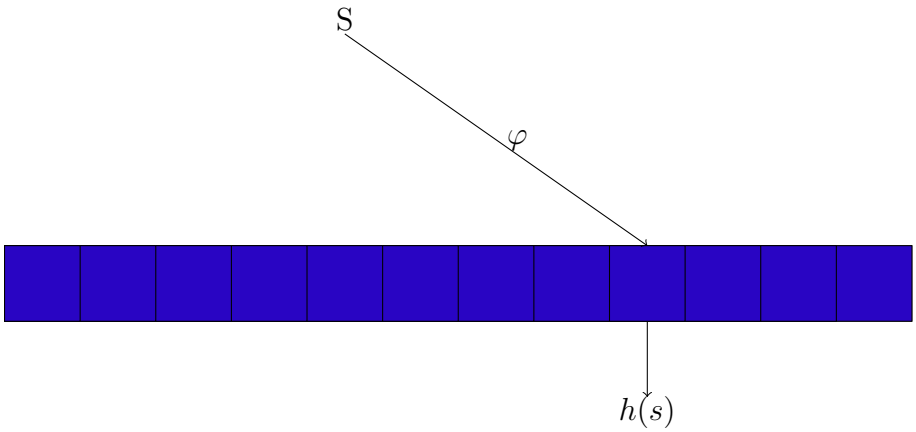

Figure 5 – One heuristic of size M

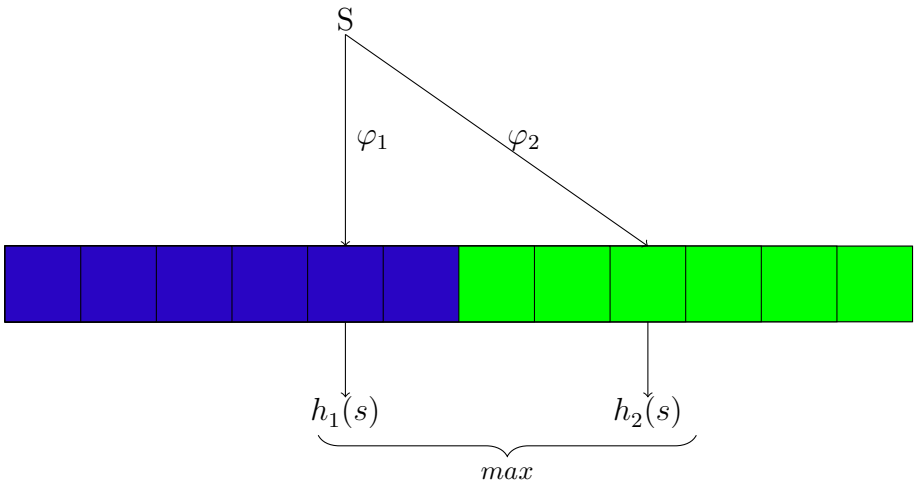

Figure 6 – Two heuristics of size M/2

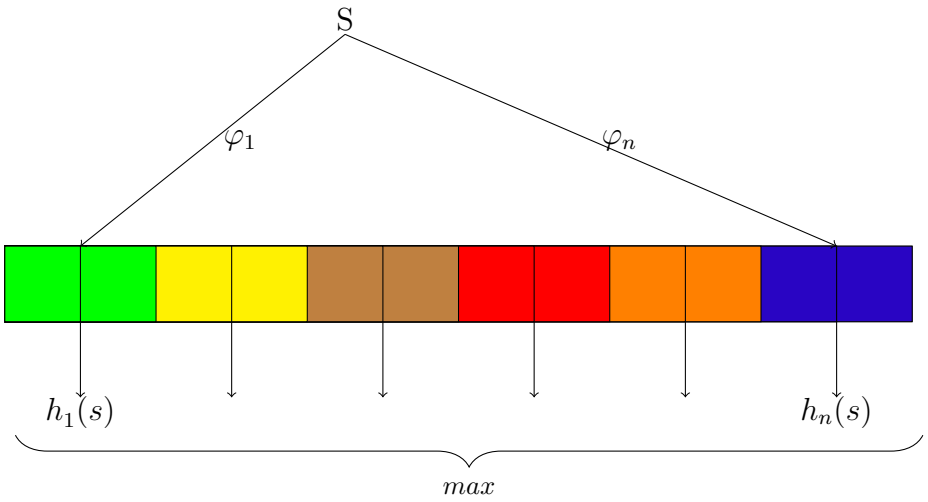

Figure 7 – N heuristics of size M/N

2.7 Heuristic Subset

The heuristics generator systems can create a large number of heuristics. Let's suppose $|\zeta| = 1000$ heuristics were created considering the time and memory avaiable and we want to select the best N = 100 heuristics. This would be:

$$\binom{1000}{100} = 10^{138} possibilities$$

So, try to select heuristics from a large set of heuristics are going to be treated as an optimization problem. Then, in order to obtain a good selection of subset of heuristics, our objective function should guarantee two properties: Monotonicity and Submodularity, that would be explained in the next Chapter.

2.8 Problem Domains

Some of the problems we are trying to solve are the optimal domains for International Planning Competition (IPC).

2.8.1 Blocks world

This domain consists of a set of blocks, a table and a robot hand. The blocks can be on top of other blocks or on the table; a block that has nothing on it is clear; and the robot hand can hold one block or be empty. The goal is to find a plan to move from one configuration of blocks to another.

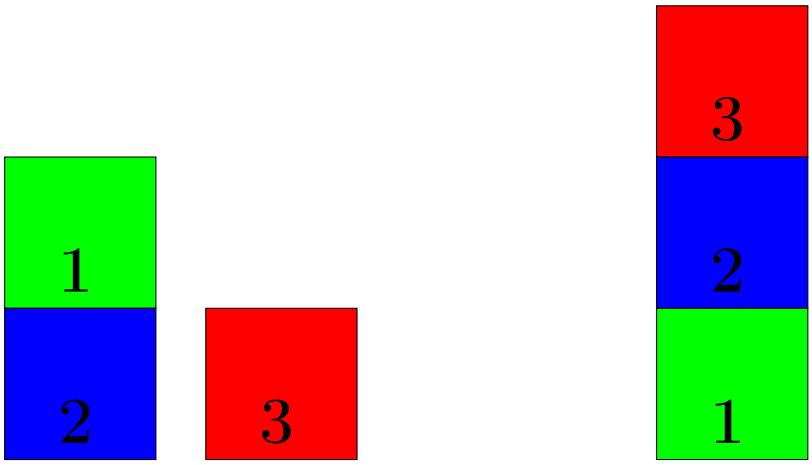

Figure 8 – Blocks world with three blocks.

The solution shown in Figure 8 is to unblock number 1 from block number 2; stack block number 2 on block number 1; finally, stack blocl number 3 on block number 2. We are interestred in optimal or near−optimal solutions for this kind of problem.

Depending on the number n of blocks this domain can have very large state spaces. For example, using $n = 20$ this domain has approximately 10^{20} different states.

2.8.2 Barman

In this domain there is a robot barman that manipulates drink dispensers, glasses and a shaker. The goal is to find a plan of the robot's actions that serves a desired set of drinks.

2.8.3 Elevators

The idea for this domain came up from the Miconic domain of IPC2, however the domain has been designed from scratch. The scenario is the following: There is a building

with N+1 floors, numbered from 0 to N. The building can be separated in blocks of size M+1, where M divides N. Adjacent blocks have a common floor. For example, suppose N=12 and M=4, then we have 13 floors in total (ranging from 0 to 12), which form 3 blocks of 5 floors each, being 0 to 4, 4 to 8 and 8 to 12.

The building has K fast (accelarating) elevators that stop only in floors that are multiple of M/2 (so M has to be an even number). Each fast elevator has a capacity of X persons. Furthermore, within each block, there are L slow elevators, that stop at every floor of the block. Each slow elevator has a capacity of Y persons (usually Y<X).

There are costs associated with each elavator starting/stoping and moving. In particular, fast (accelarating) elevators have negligible cost of starting/stoping but have significant cost while moving. On the other hand, slow (constant speed) elevators have significant cost when starting/stoping and negligible cost while moving. Travelling times between floors are given for any type of elevator, taking into account the constant speed of the slow elevators and the constant acceleration of the fast elevators.

There are several passengers, for which their current location (i.e. the floor they are) and their destination are given. The planning problem is to find a plan that moves the passangers to their destinations while it maximizes some criterion.

2.8.4 Floortile

A set of robots use different colors to paint patterns in floor tiles. The robots can move around the floor tiles in four directions (up, down, left and right). Robots paint with one color at a time, but can change their spray guns to any available color. However, robots can only paint the tile that is in front (up) and behind (down) them, and once a tile has been painted no robot can stand on it.

For the IPC set, robots need to paint a grid with black and white, where the cell color is alternated always. This particular configuration makes the domain hard because robots should only paint tiles in front of them, since painting tiles behind make the search to reach a dead-end.

2.8.5 Nomystery

In this domain, a truck moves in a weighted graph; a set of packages must be transported between nodes; actions move along edges, and load/unload packages; each

move consumes the edge weight in fuel. In brief, Nomystery is a straightforward problem similar to the ones contained in many IPC benchmarks.

2.8.6 Openstacks

The openstacks domain is based on the "minimum maximum simultaneous open stacks" combinatorial optimization problem, which can be stated as follows: A manufacturer has a number of orders, each for a combination of different products, and can only make one product at a time.

The total required quantity of each product is made at the same time (because changing from making one product to making another requires a production stop). From the time that the first product included in an order is made to the time that all products included in the order have been made, the order is said to be "open" and during this time it requires a "stack" (a temporary storage space). The problem is to order the making of the different products so that the maximum number of stacks that are in use simultaneously, or equivalently the number of orders that are in simultaneous production, is minimized (because each stack takes up space in the production area).

2.8.7 ParcPrinter

This domain models the operation of the multi-engine printer, for which one prototype is developed at the Palo Alto Research Center (PARC). This type of printer can handle multiple print jobs simultaneously. Multiple sheets, belonging to the same job or different jobs, can be printed simultaneously using multiple Image Marking Engines (IME). Each IME can either be color, which can print both color and black and white images, or mono, which can only print black and white image. Each sheet needs to go through multiple printer components such as feeder, transporter, IME, inverter, finisher and need to arrive at the finisher in order.

2.8.8 Parking

This domain involves parking cars on a street with N curb locations, and where cars can be double-parked but not triple-parked. The goal is to find a plan to move from one configuration of parked cars to another configuration, by driving cars from one curb location to another. The problems in the competition contain 2*(N-1) cars, which allows one free curb space and guarantees solvability.

2.8.9 Sokoban

This domain is inspired by the popular Sokoban puzzle game where an agent has the goal of pushing a set of boxes into specified goal locations in a grid with walls. The competition problems are generated in a way that guarantees solvability and generally are easy problem instances for humans.

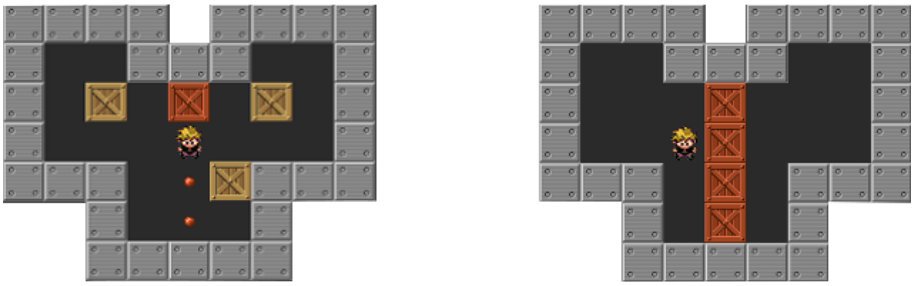

Figure 9 – Sokoban with four blocks solved

In the next Chapter, we will introduce the meta−reasoning proposed for selecting heuristics and will expand on the properties of our objective functions.

Chapter III

Approach Proposal

3 Meta-Reasoning for selection

3.1 Random Greedy Heuristic Selection (RGHS)

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

$$\begin{aligned} &\textbf{minimize}_{\zeta^{'} \in 2^{|\zeta|}} \Psi(\zeta^{'}, \nabla) \\ &\textbf{subject to} |\zeta^{'}| = N \end{aligned} \tag{3.1}$$

Where $\Psi(\zeta^{'}, \nabla)$ is an objective function and N is the desired subset size. N could be determined by a hard constraint such as the maximum number of `PDBs` one can store in memory. According to (RAYNER; STURTEVANT; BOWLING, 2013) it is unlikely that there is an efficient algorithm for solving Equation 3.1. We use an algorithm based on the work of (BUCHBINDER et al., 2014) we call Random Greedy Heuristic Selection (`RGHS`) to approximately solve Equation 3.1 for different functions Ψ.

Algoritmo 1: Random Greedy Heuristic Selection

Input : problem ∇, set of heuristics ζ, cardinality N
Output : heuristic subset $\zeta^{'} \subseteq \zeta$ of size N

$\zeta^{'}_0 \leftarrow \emptyset$
for *i = 1 to N* **do**
  Let $M_i \subseteq \zeta \setminus \zeta^{'}_{i-1}$ be a subset of size N minimizing $\sum_{h \in M_i} \Psi(\zeta^{'}_{i-1} \cup \{h\}) - \Psi(\zeta^{'}_i)$
  Let h_i be a uniformly random element from M_i
  $\zeta^{'}_i \leftarrow \zeta^{'}_{i-1} \cup \{h\}$
return $\zeta^{'}_N$

Algorithm 1 shows `RGHS`. `RGHS` receives as input a problem ∇, a set of heuristics ζ, a cardinality size N, and it returns a subset $\zeta^{'} \subseteq \zeta$. In each iteration i `RGHS` randomly selects a heuristic h_i from a pool M_i of "good" heurisitcs and adds h_i to $\zeta^{'}$. M_i is defined as follows. $M_i \subseteq \zeta \setminus \zeta^{'}_{i-1}$ is a subset of size N minimizing $\sum_{h \in M_i} \Psi(\zeta^{'}_{i-1} \cup \{h\}) - \Psi(\zeta^{'}_i)$, where $\zeta^{'}_{i-1}$ is the subset of heuristics `RGHS` selects prior to iteration i of the algorithm. M_i contains the N heuristics that when individually combined with $\zeta^{'}_{i-1}$ minimizes Ψ the most. `RGHS` returns $\zeta^{'}$ once it reaches the desired size N.

The work of (RAYNER; STURTEVANT; BOWLING, 2013) uses a greedy algorithm introduced by (NEMHAUSER; WOLSEY; FISHER, 1978) for approximately solving the heuristic subset selection problem. In contrast with the RGHS presented above, which is based on the work of (BUCHBINDER et al., 2014), (NEMHAUSER; WOLSEY; FISHER, 1978)'s approach does not offer near−optimality guarantees when the problem's objective functions is non−monotone. As mentioned before A*'s running time is a non−monotone objective function. While RGHS also offers guarantees for non−monotone objective functions, it retains the same guarantees offered by (NEMHAUSER; WOLSEY; FISHER, 1978)'s algorithm when optimizing monotone objective functions (BUCHBINDER et al., 2014). That is why we only consider (BUCHBINDER et al., 2014)'s approach in our theoretical and experimental analyses.

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. When solving ∇ using the consistent heuristic function $h_{max}(\zeta^{'})$ for $\zeta^{'} \subseteq \zeta$, A* expands in the worse case $J(\zeta^{'}, \nabla)$ 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 $J(\zeta^{'}, \nabla)$. RGHS is guaranteed to find near−optimal solutions when we use J as the objective function Ψ, as we now demostrate. In the following analysis all heuristic functions are assumed to be consistent. We also assume that A* expands all nodes n with $f(n) \leq C^{*}$ while solving ∇, as shown in Equation 3.2.

Lemma 3.2.1. *$J(\zeta^{'} \cup \{h\}) \leq J(\zeta^{'})$ for any $\zeta^{'}$ and any h.*

Proof. Fix $\zeta^{'}$ and h. Then

$$\begin{aligned} J(\zeta^{'} \cup \{h\}) &= |\{s \in V | h_{max}(s, \zeta^{'} \cup \{h\}) \leq C^{*} - g(s)\}| \\ &\leq |\{s \in V | h_{max}(s, \zeta^{'}) \leq C^{*} - g(s)\}| \\ &= J(\zeta^{'}) \end{aligned}$$

Where the inequality follows from the fact that $h_{max}(s, \zeta^{'} \cup \{h\}) \geq h_{max}(s, \zeta^{'})$ for all s.

Let S be a set and ϕ a function over 2^S. ϕ is supermodular if for any A, B, x with $A \subseteq B \subseteq S$ and $x \in S \backslash B$:

$$\phi(A) - \phi(A \cup \{x\}) \geq \phi(B) - \phi(B \cup \{x\}) \tag{3.4}$$

Intuitively, Equation 3.4 captures the idea of disminishing returns. In the context of search tree size, if we add a heuristic function h to a set of heuristics A strictly contained in a set B, then we would expect, then we would expect $J(A) - J(A \cup \{h\})$ to be larger than $J(B) - J(B \cup \{h\})$ as h would "contribute more" to A than to B.

Lemma 3.2.2. *J is supermodular.*
Proof. Let $A \subset B \subset \zeta$ and $h \in \zeta \backslash B$. By Lemma 3.2.1, $J(A) - J(A \cup \{h\}) \geq 0$ and $J(B) - J(B \cup \{h\}) \geq 0$. We consider two cases.
Case 1. $J(B) - J(B \cup \{h\}) = 0$. Then $J(A) - J(A \cup \{h\}) \geq 0$ yields $J(A) - J(A \cup \{h\}) \geq J(B) - J(B \cup \{h\})$.
Case 2. $J(B) - J(B \cup \{h\}) = k > 0$. Let $s_1, ..., s_k$ be all the states in V that satisfy $h_{max}(s_i, B) \leq C^* - g(s_i)$ and $h_{max}(s_i, B \cup \{h\}) > C^* - g(s_i)$. This implies $h(s_i) > C^* - g(s_i)$, for all $i \in \{1, ..., k\}$ Further, since $A \subset B$, we have $h_{max}(s_i, A) \leq h_{max}(s_i, B) \leq C^* - g(s_i)$, for all $i \in \{1, ..., k\}$. Consequently, $J(A) - J(A \cup \{h\}) \geq k = J(B) - J(B \cup \{h\})$.

Lemma 3.2.1 and 3.2.2 are sufficient for using a result by (BUCHBINDER et al., 2014) to conclude the following.

Theorem 3.2.3. *Let ζ' be a subset selected by **GHS**. Then $J(\zeta', \nabla)$ is within a factor of $\frac{e+1}{e} \approx 1.36$ of optimal.*

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.

$$T(\zeta', \nabla) = J(\zeta', \nabla) \cdot t_{h_{max}}(\zeta') \tag{3.5}$$

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 the time required for node generation and for the operations on A*'s `OPEN` and `CLOSED` lists. However, these two factors do not depend directly on the heuristic employed. Thus, $T(\zeta', \nabla)$ is reasonable approximation for A*'s running time for the heuristic subset selection problem.

Theorem 3.3.1. *Suppose $t_{h_i} = t_{h_j}$ for any h_i and $h_j \in \zeta$. Then for any fixed subset size N, **RGHS** yields a subset ζ' that is within a factor $\frac{e+1}{e}$ of optimal with respect to $T(\zeta', \nabla)$*

Proof. Since t_h is constant over $h \in \zeta$, the value $t_{h_{max}}(\zeta')$ is independent of ζ'. Hence the value $T(\zeta', \nabla)$ is a constant factor of $J(\zeta', \nabla)$. The latter is within a factor of $\frac{e+1}{e}$ of optimal by Theorem 3.2.3.

The assumption that $t_{h_i} = t_{h_j}$ for any $h_i, h_j \in \zeta$ often does not hold in domain−independent planning. For example, the `iPDB` heuristic (HASLUM et al., 2007) can be few order of magnitude faster than the Incremental `LM-Cut` (HELMERT; DOMSHLAK, 2009) heuristic. In order to lift such an assumption we first define A*'s running time in terms of its computational cost as follows.

$$T(\zeta', \nabla) = J(\zeta', \nabla) + \beta(\zeta') \tag{3.6}$$

Here $\beta(\zeta')$ is the computational cost incurred by using $h_{max}(\zeta')$. $T(\zeta', \nabla)$ accounts for the computational cost of expanding $J(\zeta', \nabla)$ nodes added of the computational cost of employing a set of heuristics ζ'.
We assume that the computational cost $\beta(\zeta' \cup \{h\}) = \beta(\zeta') + \beta(h)$, i.e., the computational cost of employing heuristic h during search is constant and independent of other heuristics in ζ'. Although this assumption is unlikely to hold in practice as $\beta(h)$ depends on the number of times A* uses h to evaluate nodes during search, which in turn depends on the heuristics in $\zeta' \setminus h$, we expect that the differences of $\beta(h)$−values to be negligible for different ζ' sets.

T' is clearly non−monotone as the reduction of $J'(\zeta', \nabla)$ caused by the addition of a heuristic to ζ' might not compensate for the increase in $\beta(\zeta')$. We now show that T' is supermodular.

Lemma 3.3.2. *T is supermodular.*
Proof. Let $A \subset B \subseteq \zeta$ and $h \in \zeta \setminus B$. We need to show that $T'(A) - T'(A \cup \{x\}) \geq T'(B) - T'(B \cup \{x\})$. We have that $T'(A) - T'(A \cup \{x\}) = J(A) - J(A \cup \{x\}) - \beta(x)$. and

$T'(B) - T'(B \cup \{x\}) = J(B) - J(B \cup \{x\}) - \beta(x)$. *Thus, we have that* $J(A) - J(A \cup \{x\}) \geq J(B) - J(B \cup \{x\})$, *which according to Lemma 3.2.2 is supermodular.*

Since T' is supermodular, another result by (BUCHBINDER et al., 2014) and Lemma 3.3.2 allow us to conclude the following.

Theorem 3.3.3. *Let* ζ' *be a subset selected by* ***RGHS****. Then* $T(\zeta', \nabla)$ *is within a factor of* $\frac{e+1}{e} \approx 1.63$ *of optimal.*

3.4 Estimating Tree Size and Running Time

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

We use the Culprit Sampler (CS) introduced by (BARLEY; FRANCO; RIDDLE, 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 RGHS's optimization process.

3.5 Culprit Sampler (CS)

CS runs a time−bounded A* search while sampling $f-culprits$ and $b-culprits$ 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 b$f-$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 RGHS's execution. The value of $\hat{J}(\zeta^{'},\nabla)$ for any subset $\zeta^{'}$ of ζ if then computed by iterating over all b−culprits $\mathbf{B}$ 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.

$$\hat{J}(\zeta^{'},\nabla) = \sum_{\mathbb{B}\in B} W(B) \tag{3.7}$$

Where $W(B)$ is 0 if there is a heuristic in $\zeta^{'}$ whose y-value in B is zero (i.e., there is a heuristic in $\zeta^{'}$ 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 $\zeta^{'}$. The evaluation time of the heuristics in $\zeta^{'}$ 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)

Stratified Sampling is a prediction algorithm that estimate the number of nodes expanded by some heuristic.

(KNUTH, 1975) created a method to estimate the size of the search tree such as IDA*. It works doing random walk from the root of the tree. Knuth's assumption is that all branches have the same structure. So, performing a random walk down one branch is enought to estimate the size of the search tree. However, the method does not work well for unbalanced search tree. (CHEN, 1992) solved this problem with a stratification of the search tree through a *type system* to reduce the variance of the sampling process (LELIS; ZILLES; HOLTE, 2013). In the figure 10 each state of the search space is mapped to the *Type System*

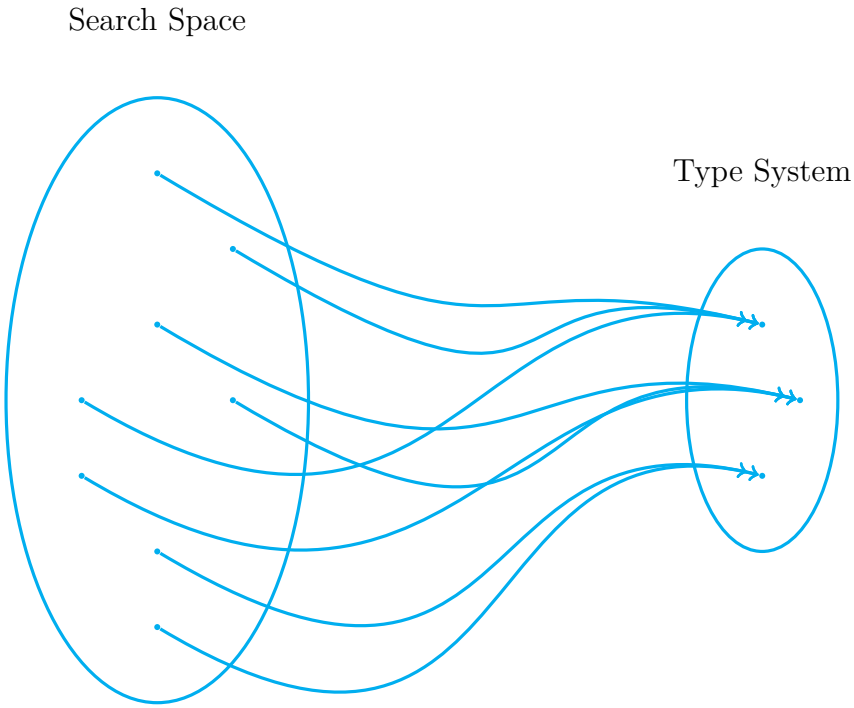

Figure 10 – Type system and the search space representation.

3.6.1 Type System

The *Type System* is a partition of the states in the state space. It is calculated based of any property of each node in the search tree. (LELIS, 2013)

A common misconception is think of *type system* as state−space abstractions. (PRIEDITIS, 1993) defines a state−space abstraction as a simplified version of the problem

in which:

- The cost of the least−cost path between two abstracted states must less than or equal to the cost of the least−cost path between the corresponding two states in the original state−space.
- Goal states in the original state−space must be goal states in the abstracted state−space.

In contrast with state−space abstractions, a *type system* does not have these two requeriments. A *type system* is just a partition of the nodes in the search tree.

The *type system* can not be represented as a graph since *type system* does not necessarily define relation between the types.

The relation between *type system* and abstractions is the following: The *type system* can not necessarily be used as abstractions, abstractions can always be used as *type system.*

Definition 3.6.1. *Type System Let S = (N,E) be a UST.* $T = \{t_1, ..., t_n\}$ *is a type system for S if it is a disjoint partitioning of N. If* $n \in N$ *and* $t \in T$ *with* $n \in t$*, we write* $T(n) = t$*.*

SS is a general method for approximating any function of the form $\varphi = \sum_{n \in S} z(n)$, where z is any function assigning a numerical value to a node. φ represents a numerical property of the search tree rooted at n^*. For instance, if $z(n) = 1$ for all $n \in S$, then φ is the size of the tree.

Instead of traversing the entire tree and summing all z−values, SS assumes subtrees rooted at nodes of the same type will have equal values of φ and only one node of each type, chosen randomly, is expanded. This is the key to SS's efficiency since the search trees of practical interest have far too many nodes to be examined examined exhaustively.

Given a search tree S and a type system T, SS estimates φ as follows. First, it samples the tree and returns a set A of $representative - weight$ pairs, with one such pair for every unique type seen during sampling. In the pair $\langle s, w \rangle$ in A for type $t \in T$, 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. 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 heuristic evaluation 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

Algoritmo 2: SS, a single probe

Input : root n^* of a tree and a type system T
Output : an array of sets A, where $A[i]$ is the set of pairs $< n, w >$ for the nodes n expanded at level i.

```
1  A[0] ← {⟨n*, 1⟩}
2  i ← 0
3  while A[i] is not empty do
4      for each element ⟨n, w⟩ in A[i] do
5          for each child n̂ of n do
6              if g(n̂) + h(n̂) ≤ d then
7                  if A[i + 1] contains an element ⟨n′, w′⟩ with T(n′) = T(n̂) then
8                      w′ ← w′ + w
9                      with probability w/w′, replace ⟨n′, w′⟩ in
10                     A[i + 1] by ⟨n̂, w′⟩
11                 else
12                     insert new element ⟨n̂, w′⟩ in A[i + 1]
13     i ← i + 1
```

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 ate another 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; STERN; STURTEVANT, 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 RGHS 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 $\zeta^{'}$ 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 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,

$$C_B = \sum_{\langle n,w\rangle \in A \wedge B(n)=B} w \tag{3.8}$$

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 $\zeta^{'}$. As explained for CS, this can be done by traversing over all b−culprits once.

3.7 8-tile-puzzle Case Using type system

In this thesis we use *type system* based only in heuristics.(ZAHAVI et al., 2010) use the simpliest heuristic−based *type system* in which two nodes n and $n^{'}$ are of the same type if they have the same heuristic value.

Let's explain how *type system* works through an example. The problem of 8−tile−puzzle in the Figure 11, the center tile is labeled with the letter M, the corners labeled with C

Figure 11 – The heuristic value is the position of the empty tile in a specific state.

and the mediums with `E`. For this problem, we can define the *type system* based on the position of the empty tile regarding the position of the empty tile in the goal state. For this case, two nodes n and n' would be of the same type if n and n' have the empty tile with the same letter and with the same distance to the empty tile of the goal state.

In the Figure 12, each row represent a type. The first board shows the empty tile in the center with distance to the empty tile in the goal state equal to 2. The shortest path to get to the empty tile in the goal state would be doing: down−left or right−down. Which both moves represent the same cost. Then, the type would be (2,`M`). In the next board, the empty tile is in the left top, and use the letter `C`. The minimum distance is 4 because to get the goal empty tile is necessary to do: down−down−right−right or right−right−down−down, etc. So the type would be (4, `C`). In the next board, there are two tiles that have the same type, because both have the same letter `M` with distance equal to 3. Then, both have the type (3,E). The fourth row have two boards with the same type, because both have the same letter `C` with distance equal to 2. Then, both have the type (2, `C`). The fifth row have two boards with the same type, because both have the same letter `E` with distance equal to 1. Then, both have the type (1, `E`). In the last row the empty tile is in the goal empty tile. So, the distance is zero and the letter is `C`. The type would be (0, `C`).

3.8 SS step by step

In the Figure 13, we can see how *Type System* works. In the Level 1, we have the root node, the w initialized with one. Let's suppose that three nodes are generated by the root node in the Level 2. 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 of the father. In the Level 2 we apply `SS`'s assumption, two nodes in the same level that have the same type (The same color) root subtrees of the same size and only one node of each type must be chosen randomly to be expanded. There are two nodes with type red in Level 2. 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

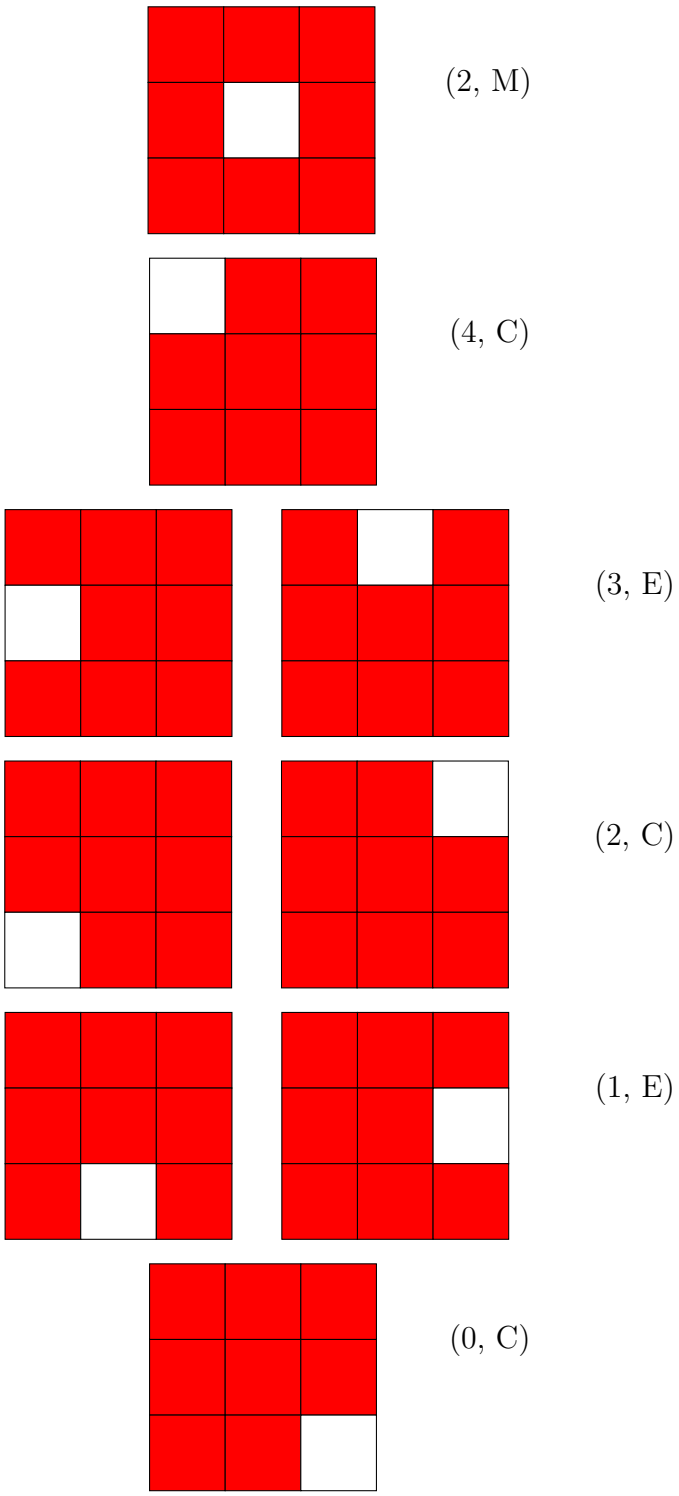

Figure 12 – Each row has one or two states that represent the same type.

$w = 1$, then we sum the w and the new $w = 2$. As a result, in the Level 2 we will have two nodes of red type and one node with blue type.

When nodes in the Level 2 are expanded. The blue node expands one node of type blue and the red node expands two nodes of type red and blue. The question here is how

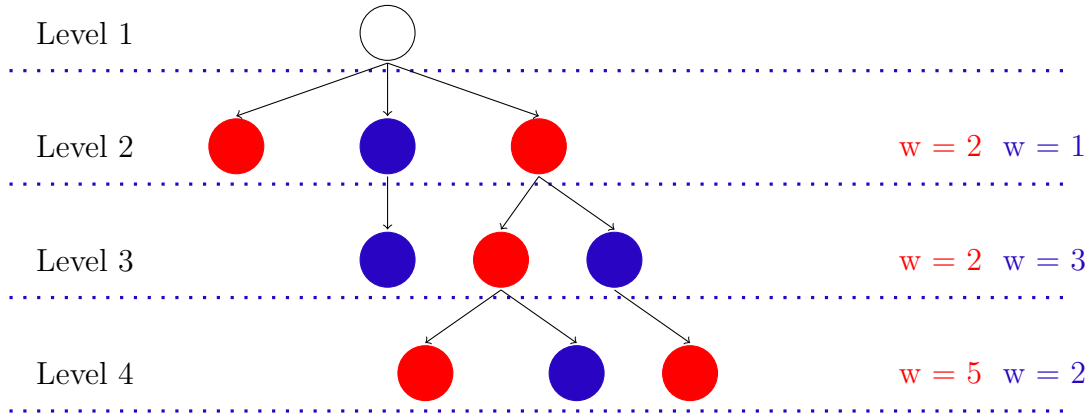

Figure 13 – Search tree using Type System

many nodes would be generated in the Level 3? The answer is: $1 \times blue + 2 \times red + 2 \times blue$. So, in the Level 3 we will have 2 nodes of red type and 3 nodes of type blue.

In the Level 3 the w of the node blue would have the same w of the father. The father has $w = 1$, then the child has $w = 1$. The w of the red type and blue type would be 2. Once the w has been updated for each node in the Level 3 we apply the SS's assumption again. There are two nodes with type blue. So, we choose randomly one of them and update their w. Let's choose the right blue type and the updated w would be 3 because 1 from the left blue type plus the 2 from the right blue type.

When nodes in the Level 3 are expanded. They are expanded in the following way: The red node expands two nodes of types red and blue and the blue node expands one of red type. How many nodes would be generated at Level 4, then? The answer is: $2 \times red + 3 \times red + 2 \times blue$. Therefore, in the Level 4, we will have five nodes of type red and two nodes of type blue.

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

3.9 Comparison between SS and IDA*

SS is an algorithm that estimates the number of nodes expanded performed by heuristic search algorithm seeking solutions in state space. We apply SS to predict the number of nodes expanded by IDA* in a given $f-$layer when using a consistent heuristics.

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 for each instance of each domain, this process is executed using different number of probes `i.e.,` 1, 10, 100, 1000 and 5000.

$$\frac{\sum_{s \in PI} \frac{Pred(s,d)-R(s,d)}{R(s,d)}}{|PI|} \tag{3.9}$$

Where PI is the set of problem instances, $Pred(s,d)$ and $R(s,d)$ are the predicted and actual number of nodes expanded by IDA* for start state s and cost bound d. A perfect score according to the measure is 0.00.

The Table 1 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 thresold. 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.

Table 1 – 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

3.10 Comparison between SS and A*

The Table 2 shows that SS is not a good predictor 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 overestimates by several orders of magnitude the actual A* search tree.

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* using a specific heuristic. The column with SS−error represents the relative−error formula 3.9.

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 of 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 because SS expands less nodes than A*.

Table 2 – 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

3.11 Approximation Analysis for SS and A*

Here we show that `SS` is able to make good selection of heuristics ipdb, `LM-Cut` and `GA-PDBs`.

So that, 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 14 . 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 chose to be used with A*. Points following on the other regions are those choices, `SS` made incorrectly.

Points that fall in each of the regions:

I $J(h_2) > J(h_1)$ for A*, $\hat{J}(h_1) > \hat{J}(h_2)$ according to `SS`.

II $J(h_2) > J(h_1)$ for A* and `SS` agrees.

III $J(h_1) > J(h_2)$ for A* and `SS` agrees.

IV $J(h_1) > J(h_2)$ for A*, $\hat{J}(h_2) > \hat{J}(h_1)$ according to `SS`.

In the Figure 15 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.

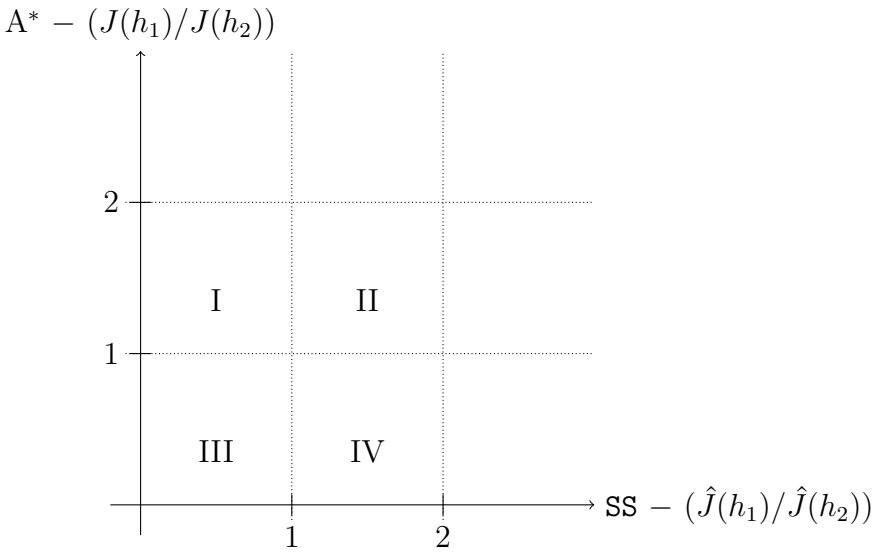

Figure 14 – Cartesian Plane with domain $\langle 0, 2\rangle$ and range $\langle 0, 2\rangle$

The eleven plots displayed show the distribution of the heuristic ratios in the four quadrants. We draw 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. So, 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, and for example just for Tidybot and Woodworking all the points are in the quadrant II or III.

In Table 3 we present a single number for each domin 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.

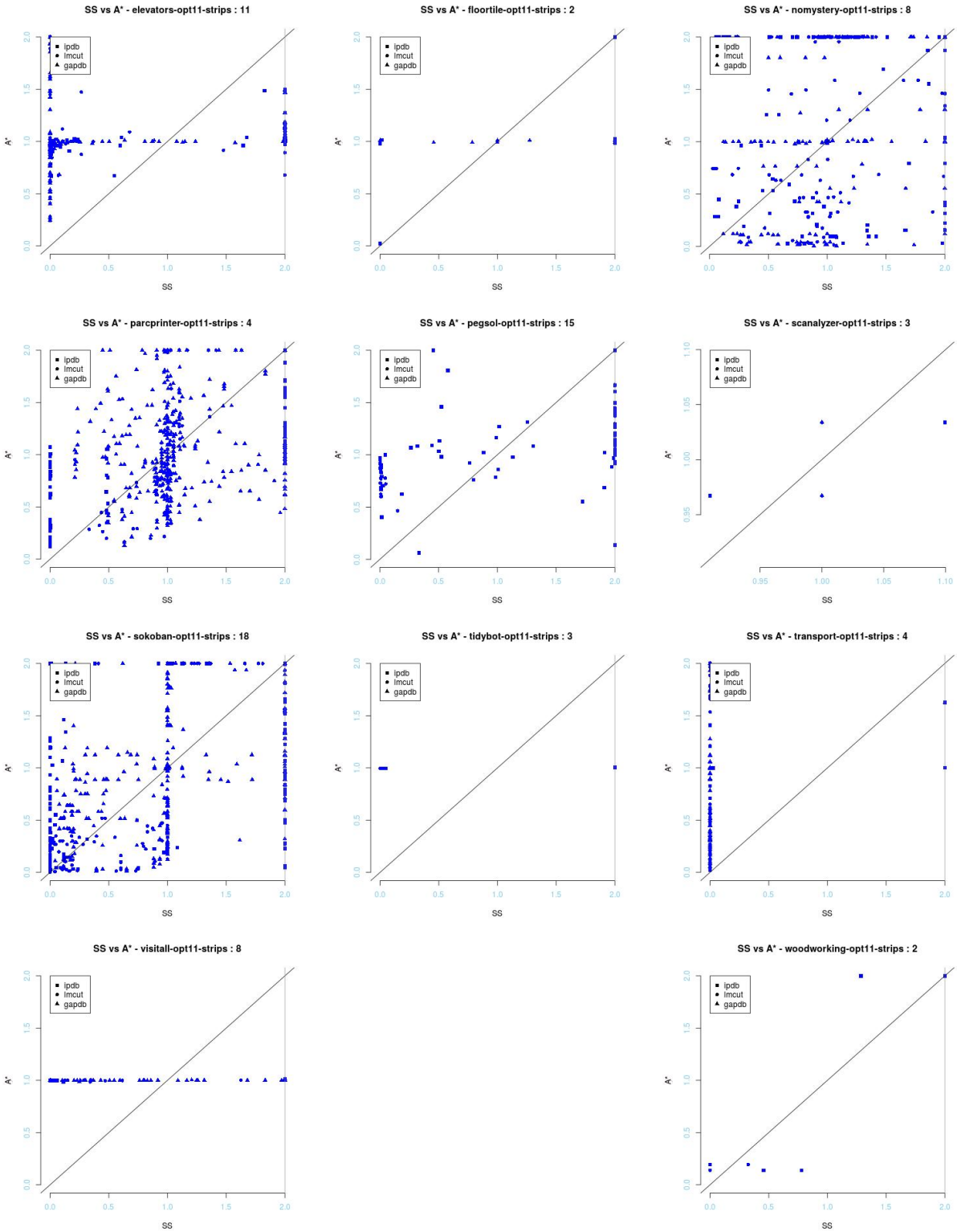

Figure 15 – SS vs A* ratios for the optimal domains – The number of instances used in each domain are showed next to the name of the Domain.

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 3 – Percentage of choices `SS` made correctly.

Chapter IV

Empirical Evaluation

4 Empirical Evaluation

RGHS is guaranteed to find a near−optimal heuristic subset to guide the A* search granted that it is able to compute the objective function of interest. Thus, the practical effectiveness of RGHS depends on its ability of finding good approximations $\hat{J}$ and $\hat{T}$. Moreover, RGHS's effectiveness depends on the value of N and on the quality of the set of heuristics ζ. In order to verify its practical effectivess, we have implemented RGHS in Fast Downward (HELMERT, 2006) and tested the A* performance using subsets of heuristics selected by RGHS 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 RGHS to make near−optimal subset selections. In the second set of experiments we test the effectiveness of RGHS by measuring the total number of problem instances solved by A* using a heuristic subset selected by RGHS.

In our experiments we implemented an approximation procedure to automatically choose the subset size selected by RGHS, which we now explain. Initially we fix N to a value (in our experiments we use $N = 25$). Then, in every iteration, when computing M_i, RGHS removes from ζ the heuristics that cannot help reducing φ. That is, in iteration i, all heuristics in $\zeta \backslash \zeta'_{i-1}$ that cannot reduce φ when combined with ζ'_{i-1} are removed from ζ. RGHS stops if ζ is empty and returns the current subset ζ' with $|\zeta'| < N$. Likewise, should heuristic h improves the objective function being optimized, then we also allow RGHS to add h to ζ' even if $|\zeta'| > N$. We observed in preliminary experiments that RGHS is robust to the initial value of N, as the total number of problems solved by A* while guided by a heuristic subset selected by RGHS varied little with N.

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; ZILLES; HOLTE, 2013; LELIS; OTTEN; DECHTER, 2014).

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.

Table 4 – 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	2.15	27.26	2.34	30.68	4790.16	20
Elevators	1.09	18.00	1.04	20.00	155.00	1
Floortile	1.01	45.86	1.01	41.14	136.57	13
Openstacks	1.00	1.00	1.00	1.00	316.62	12
Parking	1.01	7.05	1.01	7.37	17.53	18
Pegsol	1.00	32.00	1.00	48.33	78.00	2
Scanalyzer	1.01	37.29	1.07	21.43	52.14	6
Tidybot	1.03	4.06	1.00	8.19	3259.00	15
Transport	7.33	9.70	1.45	12.90	156.50	9
Visitall	1.00	95.00	1.02	48.00	218.00	2
Woodworking	2.39	83.00	611.72	32.00	346.00	5

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

We test whether the approximation $\hat{J}$ provided by `CS` and `SS` allows `RGHS` 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 optimal with respect to all subsets of any size, for some constant α. We are particularly interested in observing if $J(\zeta')$ is within 1.36 of $J(\zeta)$.

In contrast with objective function J, there is no easy way to find the minimum of T for a subset of fixed size 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 `RGHS` to make near−optimal subset selections while minimizing the A* running time. This is because by only selecting heuristics which have the same evaluation time, if `RGHS` is making near−optimal subset selections with respect to J, then `RGHS` must also be making near−optimal subset selectioons with respect to T for a fixed subset size (See Theorem 3.3.1).

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; FRANCO; RIDDLE, 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 `RGHS` is making near−optimal selections not only when minimizing J but also when minimizing T, 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 out ζ set. RGHS 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 RGHS 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 RGHS 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 RGHS using CS as RGHS+CS. Similarly, we write RGHS+SS when SS is used as predictor to make the heuristic subset selection.

Table 4 shows the average ratios of $J(\zeta^{'})$ to $J(\zeta)$ for both SS and CS in different 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, RGHS using SS and RGHS using CS). We only present results for instances that are not solved during RGHS'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 RGHS ($|\zeta^{'}|$). This experiment shows that for most of the problems RGHS, using either CS or SS, is selecting a near$-$optimal subset of ζ. For example, in Tidybot RGHS 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 3% larger than optimal for RGHS+SS, and is optimal for RGHS+CS.

The exceptions in Table 4 are the ratios for Barman, Transport and Woodworking. In Transport SS has an average ratio of 7.33 and CS of 1.45. By looking at the ratios of SS for individual instances of SS for individual instances of Transport (results now show in Table 4), we noticed that SS is able to make optimal selections for all but one of the 9 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 Transport, 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 RGHS 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, RGHS makes a limited selection of heuristics to guide A* search. While RGHS using SS selects an average of 83 heuristics in Woodworking instances, RGHS using CS selects only an average of 32 heuristics. This difference on sampling strategies reflects on the number of problems solved by A*. While RGHS+SS solves 15 instances of the Woodworking domain, RGHS+CS solves only 11. In total, out of the 280 instances of the IPC 2011 benchmark set, RGHS+SS solves 200 problem instances in this experiment, while RGHS+CS only solves 193 problem instances. (The numbers of instances solved are not shown in Table 4).

4.2 Comparison with Other Planning Systems

The objective of this second set of experiments is to teset the quality of the subset of heuristics RGHS 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 $\zeta^{'}$, 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 RGHS approaches.

4.3 Systems Tested

RGHS is tested while minimizing the A* search tree size (Size) and the A* running time (Time). We also use RGHS to maximize the sum of heuristic values in the state space (Sum), as suggested by (RAYNER; STURTEVANT; BOWLING, 2013). (RAYNER; STURTEVANT; BOWLING, 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; STURTEVANT; BOWLING, 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 RGHS 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 RGHS while minimizing tree size and using SS as predictor. As explained above, in this setting RGHS 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 RGHS with its strongest settings, I.E., according to Theorem 3.3.1 RGHS makes selections that are within the 1.36 factor of optimal with respect to both J and T 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 tree (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 RGHS approaches with several other state—of—the—art planners. Namely, we experiment with RIDA* (BARLEY; FRANCO; RIDDLE, 2014), two variants of StoneSoup (StSp1 and StSp2) as described by (NISSIM; HOFFMANN; HELMERT, 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; HOFFMANN; HELMERT, 2011).

The results are presented in Table 5. The results for the RGHS 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 218.4 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 good 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 RGHS approach.

Time+CS also performed well in our experiments−the approach solves 215 problems on average. Clearly Hybrid and Time+CS are far superior to all other approaches tested. For example, Size + SS and Sum solves only 204 and 203 problems, respectively. While minimizing the search tree size or maximizing the sum of heuristic values, RGHS will tend to add accurate heuristics to the selected subset, independently of their evaluation time. As a result, if not minimizing the running time, RGHS 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 5 – Coverage of different planning systems on the 2011 IPC benchmarks. For the RGHS and Max approaches we also present the average number of heuristics RGHS selects ($|\zeta'|$).

Domains	Hybrid	CS		SS		Sum	RIDA*	SY1	SY2	StSp1	StSp2	Max	iPDB	LM-Cut	M&S
		Time	Size	Time	Size										
Barman	6.9	4.75	4	4	4	4	4	10	11	4	4	4	4	4	4
Elevators	19	19	19	19	19	19	19	20	20	18	18	19	17	18	12
Floortile	14	14	14	14	14	14	14	14	14	14	14	14	8	14	10
Nomystery	20	20	20	20	20	19	20	16	16	20	20	20	19	14	18
Openstacks	17	17	15	17	15	15	15	20	20	17	17	11	17	15	17
Parcprinter	17.7	18	19	15	16	16	18	17	17	18	18	18	16	17	16
Parking	7	7	1	2	2	2	7	2	1	5	5	2	7	2	7
Pegsol	19	19	19	19	19	19	19	19	20	19	19	19	20	17	19
Scanalyzer	14	13.87	13	12	13	13	14	9	9	14	14	14	10	12	11
Sokoban	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
Tidybot	15.9	16	16	16	16	16	17	15	17	16	16	15	14	16	9
Transport	14	13.37	11	13	13	13	10	10	11	7	8	9	8	6	7
Visitall	18	18	17	17	17	17	18	12	12	16	16	18	16	10	16
Woodworking	15.7	15	15	12	16	16	15	20	20	15	15	16	9	15	9
Total	218.4	215	203	200	204	203	210	204	208	203	204	199	185	180	175

Note tat the difference on the number of problems solved by `Time+CS` and `Time+SS`: While the former solves 215 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 7 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` wrongly estimates that `LM-Cut` 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 `RGHS`, 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, `RGHS` 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. `RGHS` 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 5 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 `RGHS` approach, `Hybrid`, substantially outperforms the number of instances solved by `Max`—`Hybrid` solves on average more than 19 instances than `Max`. Finally, `Hybrid` and `Time+CS` substantially outperforms all other approaches tested, with RIDA* being the closest competitor with 210 instances solved.

Chapter V

Conclusion

5 Concluding Remarks

This dissertation showed that the problem of finding the optimal subset of size N of a set of heuristics ζ for a given problem task is supermodular with respect to the A* search tree size and, under mild assumptions, with respect to the A* running time. Thus, the `RGHS` algorithm which selects heuristics from ζ one at a time is guaranteed to produce a subset $\zeta^{'}$ such that the number of nodes expanded by A* while guided by the heuristic $\zeta^{'}$ is no more than approximately 1.36 times optimal. Furthermore, if all heuristics in ζ have the same evaluation time, then we have the same near−optimal guarantee with respect to running time. If the heuristics in ζ have different evaluation times, then the resulting A* running time is no more than approximately 1.63 times optimal. 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; STURTEVANT; BOWLING, 2013).

Since we cannot compute the values of the objective functions exactly, `RGHS` 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 `RGHS` to make near−optimal subset selections with respect to the search tree size and running time.

Finally, experiments on optimal domain−independent problems showed that `RGHS` 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. Citado na página 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. Citado 4 vezes nas páginas 29, 45, 64, and 67.

BUCHBINDER, N. et al. Submodular maximization with cardinality constraints. In: SIAM. *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*. [S.l.], 2014. p. 1433–1452. Citado 4 vezes nas páginas 41, 42, 43, and 45.

CHEN, P.-C. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, p. 295–315, 1992. Citado 3 vezes nas páginas 45, 47, and 49.

CULBERSON, J. C.; SCHAEFFER, J. Pattern databases. *Computational Intelligence*, Wiley Online Library, v. 14, n. 3, p. 318–334, 1998. Citado na página 31.

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. Citado na página 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. Citado 2 vezes nas páginas 24 and 64.

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. Citado na página 30.

HASLUM, P. et al. Domain-independent construction of pattern database heuristics for cost-optimal planning. *AAAI*, v. 7, p. 1007–1012, 2007. Citado 3 vezes nas páginas 24, 44, and 66.

HELMERT, M. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)*, v. 26, p. 191–246, 2006. Citado 2 vezes nas páginas 25 and 63.

HELMERT, M.; DOMSHLAK, C. Landmarks, critical paths and abstractions: what's the difference anyway? In: *ICAPS*. [S.l.: s.n.], 2009. p. 162–169. Citado na página 44.

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. Citado 3 vezes nas páginas 11, 24, and 33.

KNUTH, D. E. Estimating the efficiency of backtrack programs. *Mathematic of Computation*, v. 29, n. 129, p. 121–136, 1975. Citado na página 47.

KORF, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, v. 27, p. 97–109, 1985. Citado 2 vezes nas páginas 23 and 29.

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. Citado na página 63.

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. Citado na página 50.

LELIS, L. H.; ZILLES, S.; HOLTE, R. C. Predicting the size of ida* search tree. *Artificial Intelligence*, Elsevier, v. 196, p. 53–76, 2013. Citado 4 vezes nas páginas 23, 29, 47, and 63.

LELIS, L. H. Santana de. *Cluster-and-Conquer: A Paradigm for Solving State-Space Problems*. Tese (Doutorado) — University of Alberta, 2013. Citado na página 47.

NEMHAUSER, G. L.; WOLSEY, L. A.; FISHER, M. L. An analysis of approximations for maximizing submodular set functions—i. *Mathematical Programming*, Springer-Verlag, v. 14, n. 1, p. 265–294, 1978. Citado na página 42.

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. Citado na página 67.

PRIEDITIS, A. Machine discovery of effective admissible heuristics. *Machine Learning*, v. 12, p. 117–141, 1993. Disponível em: <http://dx.doi.org/10.1007/BF00993063>. Citado na página 47.

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. Citado 5 vezes nas páginas 29, 41, 42, 66, and 73.

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. Citado na página 24.

ZAHAVI, U. et al. Predicting the performance of ida* using conditional distributions. *Journal of Artificial Intelligence Research*, v. 37, n. 1, p. 41–84, 2010. Citado na página 50.