MARVIN ABISRROR ZARATE

ON SELECTING HEURITIC FUNCTION SUBSET FOR DOMAIN INDEPENDENT-PLANNING

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

VIÇOSA
MINAS GERAIS - BRASIL
2016

This dissertation is dedicated to 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)

Acknowledgements

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

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

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

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

Contents

List of Figures

List of Tables

RESUMO

Abisrror, Marvin, M.Sc., Universidade Federal de Viçosa, Fevereiro 2016. **Selecção de um subconjunto de funções heurísticas para o planejamento de domínio independente**. Orientador: Levi Henrique Santana de Lelis.

Nesta dissertação apresentamos métodos gulosos para a seleção de um subconjunto de funções heurísticas de um grande conjunto de possibilidades com o objetivo de reduzir o tempo de execução de algoritmos de busca. Trabalhos anteriores mostraram que a busca pode ser mais rápido se vários bancos de dados padrão menores são usados em vez de um grande banco de dados padrão. Nossos métodos são capazes de selecionar boas heurísticas de um grande conjunto de funções heurísticas para guiar uma A*. Implementamos nosso método em Fast Downward e mostrou empiricamente que produz heurísticas que superam o estado-da-arte de outros planejadores na Competição Internacional de Planejamento.

ABSTRACT

Abisrror, Marvin, M.Sc., Universidade Federal de Viçosa, Febrary 2016. **On selecting heuristic function subset for domain independent-planning**. Adviser: Levi Henrique Santana de Lelis.

In this dissertation we present greedy methods for selecting a subset of heuristic functions from a large pool of possibilities with the objective of reducing the running time of search algorithms. Previous works showed that search can be faster if several smaller pattern databases are used instead of one large pattern database. Our methods are able to select good heuristics from a large set of heuristic functions to guide A* search. We implemented our method in Fast Downward and showed empirically that it produces heuristics which outperform the state-of-the-art planners in the International Planning Competition benchmarks.

1 Introduction

State-space search algorithms have been used to solve important real-world problems, such as problems arising in Robotics (BADRUDDIN; ALI, 2015), chemical compounds discovery (HEIFETS; JURISICA, 2012), number partitioning (KORF, 1998), bin packing (KORF, 2002), sequence alignment (KORF; ZHANG, 2000), automating layouts of sewers (BURCH et al., 2010), network routing (LI; HARMS; HOLTE, 2005), and domain-independent planning (BONET; GEFFNER, 2001a), among others.

In this dissertation we study methods for selecting a subset of heuristic functions while minimizing approximations of the search tree size and of the running time of the A* (HART P. E.; NILSSON; RAPHAEL, 1968) while solving state-space search problems.

We are interested in selecting heuristics from a large set of possibilities. This is because previous works have shown that using the maximum of multiple heuristics can reduce the search time (HOLTE et al., 2006). Intuitively, one heuristic can be helpful in a region of the search tree where another heuristic isn't. Then, instead of using one heuristic to find the solution, it would be best to use the most promising subset of heuristics from a possibly large set.

1.1 Background work

State-space search algorithms are used to solve certain class of Artificial Intelligence (AI) problems by finding a sequence of actions from the start state to a goal state in the search space. Two well known search algorithms for solving state-space search problems are Depth-First Search (DFS) and Breadth-First Search (BFS). DFS looks a the solution by exploring the subtree rooted at node n before exploring the subtrees rooted at n's siblings while looking for a path from start to goal.

Figure 1 shows the ordering in which DFS expands nodes while solving the 8-tile puzzle (explained in detail in Chapter 2), a state-space problem. DFS expands 31 states before finding a goal state (represented by the bottom right state). BFS looks for the solution by exploring all nodes in a given level before exploring nodes in the next level. In Figure 2 we can see BFS expands 46 states to find the goal. In both Figure 1 and Figure 2 the numbers above of each state represent the order in which the states are visited. DFS and BFS are brute-force search algorithms, as they visit all states encountered during its search before finding a solution. We call brute-force-search tree (BFST) the tree expanded by DFS and BFS.

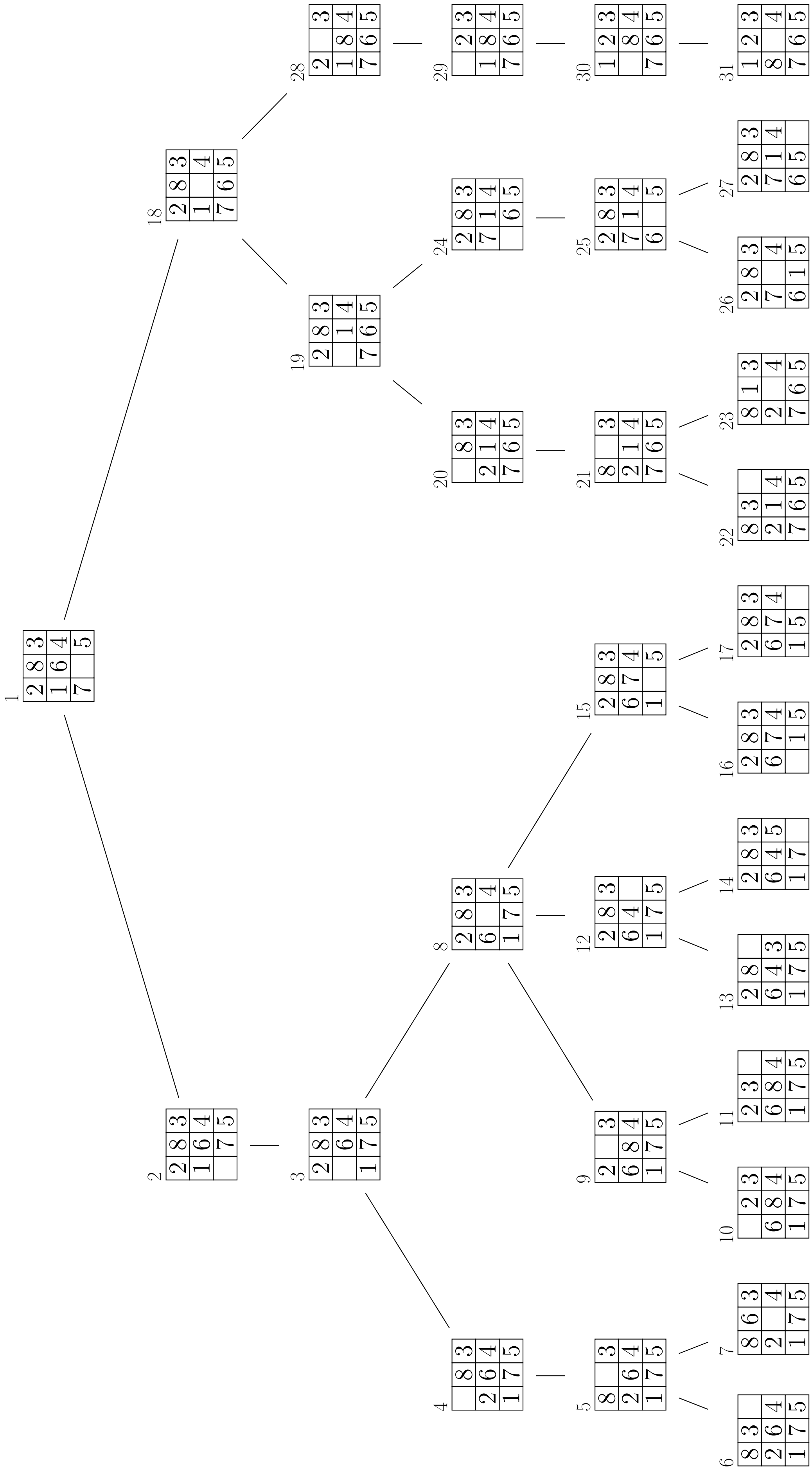

Figure 1 – 8 tile puzzle using DFS. (CHEN, 2011)

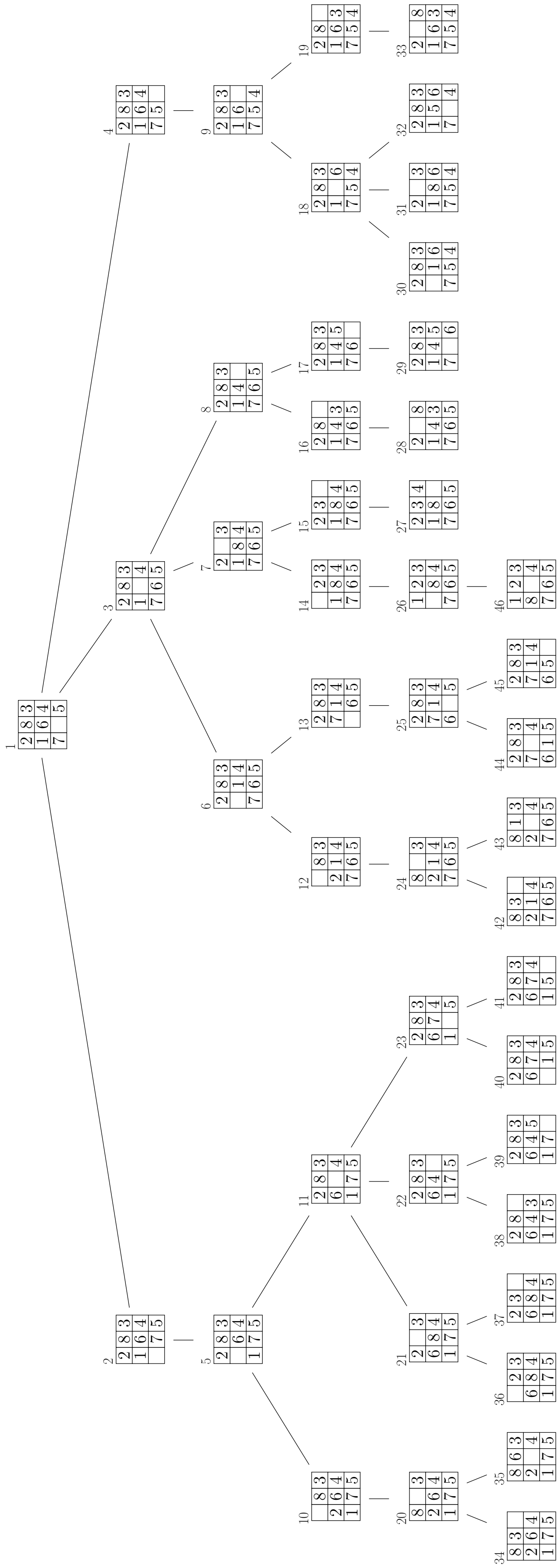

Figure 2 – 8 tile puzzle using BFS. (CHEN, 2011)

There are other types of algorithms called heuristic search algorithms, and the most representative algorithm of this type is A* (HART P. E.; NILSSON; RAPHAEL, 1968). Heuristic search algorithms use a heuristic function for estimating the distance of a node in the search tree to a goal state. Heuristic search algorithms tend to generate smaller search trees than the BFST because the heuristic guides the search to more promising parts of the state space. Also, by reducing the search tree size, the guidance of the heuristic function might also reduce the overall running time of the algorithm.

There are different approaches (HASLUM et al. 2007; EDELKAMP 2007; NISSIM; HOFFMANN; HELMERT 2011) for creating heuristics. One approach that has shown successful results in heuristic generation is Pattern Database (PDB) (CULBERSON; SCHAEFFER, 1998). The way PDBs work is as follows: The search space of the problem is abstracted into a smaller state space that can be enumerated with exhaustive search. The distances of all abstracted states to the abstracted goal state are stored in a lookup table, which can be used as a heuristic function for the original state space.

1.2 Objectives

The general objective of this dissertation is to develop algorithms for selecting a heuristic subset from a large set of heuristics with the goal of solving domain-independent planning problems. Specifically our objectives are the following.

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

1.3 Scope, Limitations, and Delimitations

We implemented our method in Fast Downward (HELMERT, 2006) and we tested our methods on the 2011 International Planning Competition (IPC) problem instances.

1.4 Justification

Good results have been obtained in domain-independent planning by using a heuristic search approach (BONET; GEFFNER, 2001b). The heuristic function used to guide the A* search are known to greatly affect the algorithm's running time. That is why it is important to have methods for selecting a good subset of heuristics to guide the A* search.

1.5 Hypothesis

We test the following hypothesis:

- A greedy algorithm is effective for selecting a good subset of heuristics to guide the A* search while solving domain-independent planning problems.

1.6 Contributions

The main contributions of this dissertation are:

- An approach for selecting heuristic functions while minimizing the number of nodes generated by A*.
- An approach for selecting heuristic functions while minimizing the running time of the A* search.
- Detailed experiments on domain-independent planning showing the strengths and weaknesses of the proposed approaches. Our experiments also show that one of our proposed approaches is able to outperform all other systems tested.

1.7 Organization

This dissertation is organized as follows:

1. In Chapter I, the background of the dissertation is provided, which also includes the objectives and the scope definition.
2. In Chapter II we review the state-of-the-art in selection of heuristic functions.
3. In Chapter III we introduce Greedy Heuristic Selection (`GHS`).
4. In Chapter IV compare `GHS` with other planner systems.
5. We conclude in Chapter V.

2 Background

The system most similar to the one we present in this dissertation 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 subsets of size one before trying subsets of size two and so on. RIDA* stops after evaluating a fixed number of subsets. While RIDA* is able to evaluate sets of heuristics with only tens of elements, the method we propose in this dissertation is able to evaluate sets with thousands of elements.

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

Our method requires a prediction of the number of nodes expanded by A* using any given subset. One of the prediction methods we use is SS. Although, SS produces good predictions of the Iterative-Deepening A* (IDA*) (KORF, 1985) search tree, it does not produce good predictions of A*'s search tree. This is because SS is unable to detect duplicated nodes during sampling (LELIS; STERN; STURTEVANT, 2014). Although SS does not produce good predictions of the number of nodes generated by A*, we show empirically that SS allows the algorithm we introduce in this dissertation to make good subset selections.

2.1 Planning Task

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 $D_{v_1} \times \cdots \times D_{v_2}$. 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 of 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, resulting in a new state, which we call a *child* of s. We define as $children(s)$ the set of child nodes of s. G is the goal condition, an assignment of values to a subset of variables, V_G. A state is a goal state if it and G agree on the assignment of values to the variable in V_G. I is the initial state, and the planning task, ∇, is to find an optimal (least-cost) sequence of operators leading from I to a goal state. We denote the optimal solution cost of ∇ as C^*.

2.2 The 8-tile-puzzle case

The domain 8-tile-puzzle is used to illustrate a few concepts that will be helpful in the other chapters of this dissertation. The 8-tile-puzzle, illustrated in the Figure 3, consists of a board with 8 tiles numbered from 1 to 8 and one empty square. The goal of this puzzle is to order the tiles as shown by the goal configuration in Figure 3. The goal can be achieved by sequentially moving the numbered tiles into the empty tile.

Initial

4	1	2
8		3
5	7	6

Goal

1	2	3
4	5	6
7	8	

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

Instead of using `DFS` or `BFS` that will analyze all states encountered during search to solve instances of the 8-tile-puzzle, we can obtain heuristics from the domain, which will allow search algorithms such as A* and IDA* to find a solution quicker.

2.3 Heuristics

There exist many state-space search algorithms, and one of the most important and well known is A* (HART P. E.; NILSSON; RAPHAEL, 1968). A* uses the $f(s) = g(s)+h(s)$ cost function to guide its search to more promising parts of the state space. $g(s)$ is the cost-to-go from the start state to state s, and $h(s)$ is the estimated cost-to-go from s to the goal.

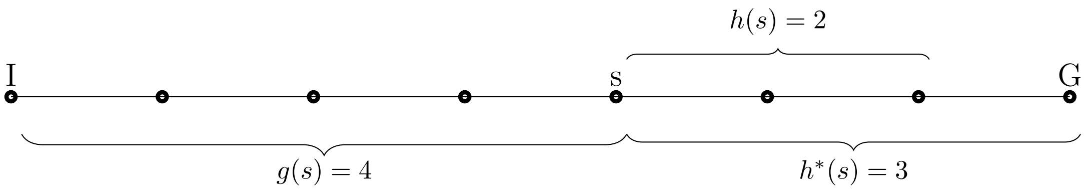

Figure 4 – Heuristic Search: I: Initial state, s: Some sate, G: Goal state

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

A heuristic is admissible if $h(s) \leq h^*(s)$ for all $s \in V$. A heuristic is consistent iff $h(s) \leq c(s, t) + h(t)$ for all states s and t, where $c(s, t)$ is the cost of the cheapest path from s to t. For example, the heuristic function provided by a `PDB` (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 minimal when A* using a consistent heuristic expands s. We call an A* search tree the tree defined by the states generated by A* using a consistent heuristic while solving a problem ∇.

We now show two heuristics for the 8-tile-puzzle.

2.3.1 Out-of-place Heuristic (OP)

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

4	1	2
8		3
5	7	6

$\xrightarrow{\text{O.P=8}}$

1	2	3
4	5	6
7	8	

Figure 5 – Out-of-place heuristic

The tiles numbered with 4, 1, 2, 3, 6, 7, 5, and 8 are out of place, then each tile counts as 1, yielding a total of 8, which is heuristic value for this state.

2.3.2 Manhatham Distance Heuristic (MD)

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

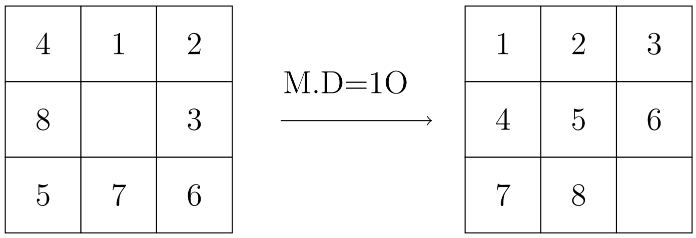

Figure 6 – Manhatham distance heuristic

Tile 4 requires one move to get to its goal position; tile 1 through 7 require one move each; tiles 5 and 8 require two moves each; the sum of all minimum required moves is 10, which is the MD estimate of the cost-to-go to this particular state.

OP and MD are heuristics that use domain knowledge to estimate the cost to go to different states. Other methods, such as `PDBs`, do not require domain knowledge to create a heuristic function. In this dissertation we use a set of domain-independent heuristics to guide search.

2.4 Using a Set of Heuristics

A* uses a structure called `OPEN` to store all nodes encountered during search that were not yet explored. In each iteration A* expands the state with the lowest cost function in `OPEN`. In Figure 7, we have two figures representing the same graph. The problem is to find the least-cost path from `A` to `O`. Above of each node we have the heuristic value assigned by a heuristic h_1. A* adds the state `A` in `OPEN` which is immediately expanded, generating states `B,C` and `D`. State `A` is removed from `OPEN` and `B,C,D` are added to `OPEN`, In every iteration, A* expands the state with the lowest cost function in `OPEN`. After `A` is expanded we have that $f(\texttt{B}) = 8, f(\texttt{C}) = 9$, and $f(\texttt{D}) = 12$. Thus, state `B` is chosen to be expanded next. The state `E` is generated and added to `OPEN` with $f(\texttt{E}) = 10$. For the next iteration, `C` is chosen for expansion because it has the lowest f-value and `C`'s child `F` is added to `OPEN`. The state `F` has the lowest cost value in `OPEN` because $f(\texttt{F}) = 9$. `F`'s child `O` is added to `OPEN`. `O` is expanded next, and that is when A* stops because `O` is the goal.

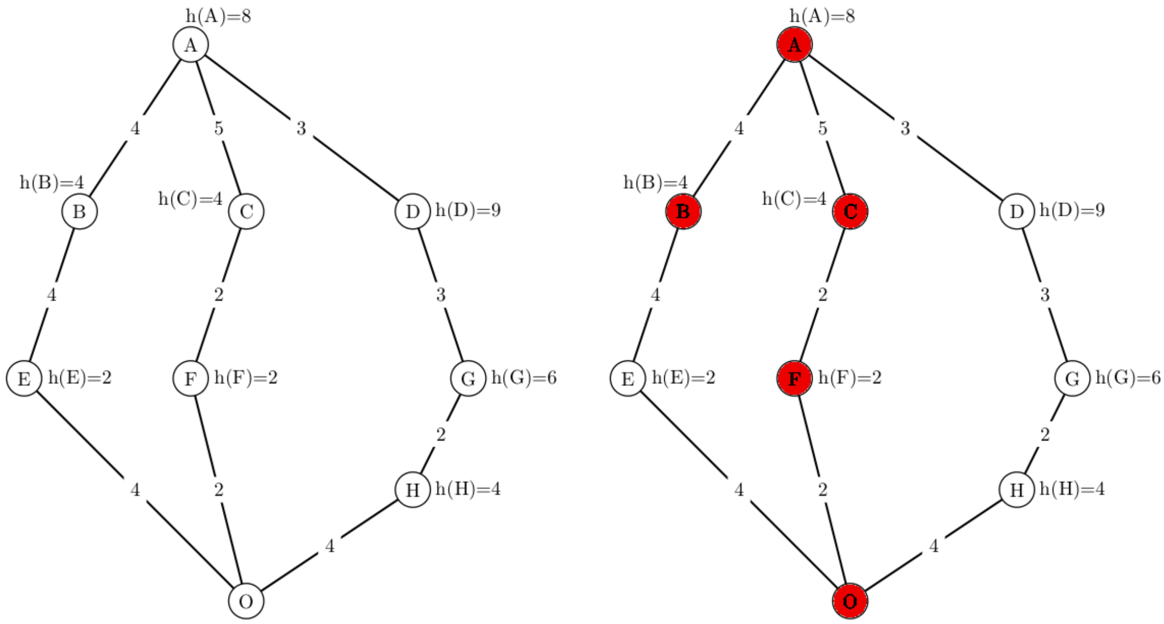

Figure 7 – The left figure shows a state space graph where the number above of each node is the heuristic value assigned by a heuristic h_1. The initial state is `A` and the goal state is `O`. The right figure shows a state space where the highlighted states are those expanded by A*.

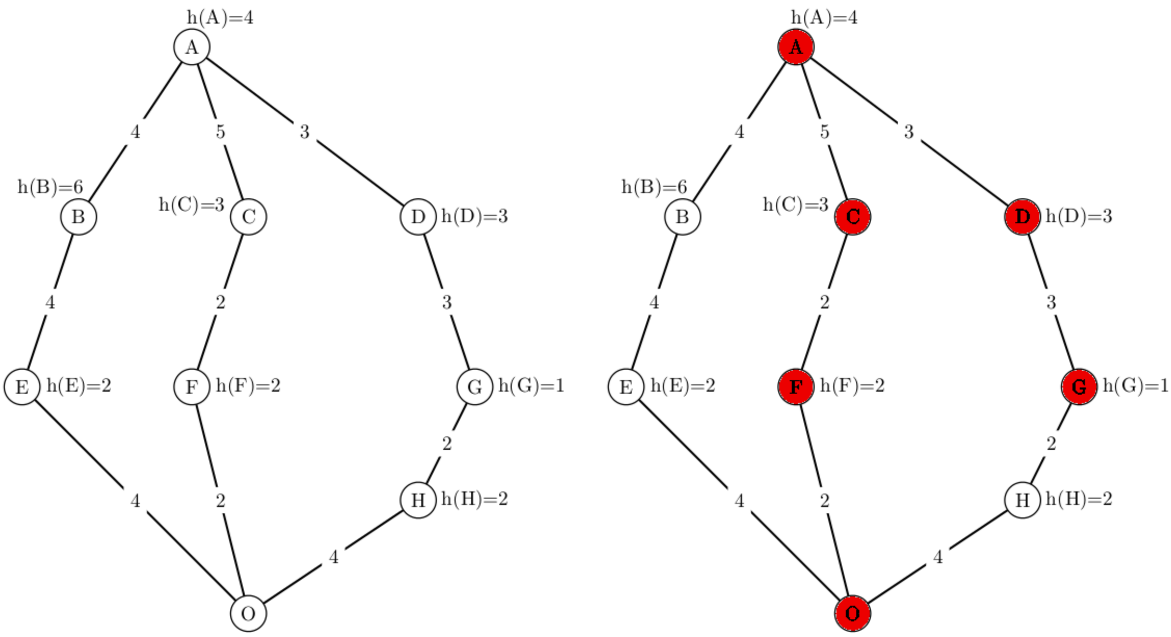

Figure 8 – The left figure shows a state space graph where the number above of each node is the heuristic value assigned by a heuristic h_2. The initial state is `A` and the goal state is `O`. The right figure shows a state space where the highlighted states are those expanded by A*.

Consider that we replace h_1 by another heuristic h_2, resulting in Figure 8. The right figure shows that A* finds the optimal cost path for the problem, which has cost 9. We also observe that A* using h_2 expands a different set of nodes than A* using h_1.

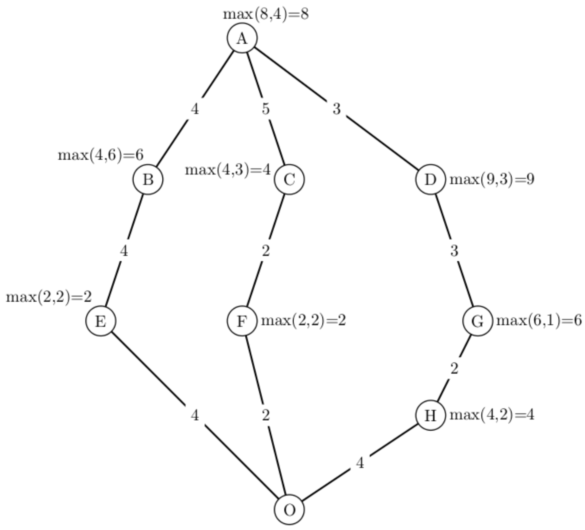

Figure 9 – The left figure shows a state space graph where the number above of each node is the maximum heuristic value of h_1 and h_2. The initial state is `A` and the goal state is `O`. The right figure shows a state space where the highlighted states are those expanded by A*.

One approach to take advantage of a set of heuristics ζ is to compute the maximum of all heuristics in ζ. For example, given the two heuristics h_1 and h_2, the maximum of h_1 and h_2 will tend to yield a more informed heuristic than h_1 or h_2 alone. In Figure 9, we show that A* using the maximum of heuristics values of h_1 and h_2 expands fewer nodes than each heuristic individually.

One can easily create thousands of heuristics for a problem instance. For example, each different abstraction of a problem domain results in a different `PDB` heuristic. It is possible to prove that the maximum of all heuristics in ζ cannot be less informed than the maximum of any subset of ζ. However, if ζ is too large, then the resulting heuristic obtained by taking the maximum of all heuristics in ζ can be too expensive to be effective to guide search. That is why we select a subset of ζ to guide the A* search. This way we try to select the most informative heuristics in ζ while preventing the resulting maximum heuristic to be computationally expensive.

In the next Chapter, we will introduce a meta-reasoning for selecting a subset of ζ to guide the A* search.

3 Greedy Heuristic Selection

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

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

Where $\Psi(\zeta', \nabla)$ is an objective function we want to minimize for a subset of heuristics ζ' of ζ. We use a hill-climbing algorithm we call Greedy Heuristic Selection (`GHS`) to solve Equation 3.1 for different functions Ψ.

Algoritmo 1: Greedy Heuristic Selection

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

```
1 ζ' ← ∅
2 while Ψ(ζ', ∇) can be improved do
3     h ← arg min_{h∈ζ} Ψ(ζ' ∪ {h}, ∇)
4     ζ' ← ζ' ∪ {h}
5 return ζ'
```

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

3.1 Minimizing Search Tree Size

The first objective function Ψ we consider is the number of node generations A* performs while solving a given planning problem. The planning problem must be solvable, this means C^* cannot be infinity. When solving ∇ using the consistent heuristic function $h_{max}(\zeta')$ for $\zeta' \subseteq \zeta$, A* generates a number of nodes that is bounded above by $J(\zeta', \nabla)$, defined as,

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

We write $J(\zeta')$ or simply J instead of $J(\zeta', \nabla)$ whenever ζ' and ∇ are free from the context. What's more, we assume that A* expands all nodes s with $f(s) \leq C^*$ while solving ∇.

3.2 Minimizing A*'s Running Time

Another objective function Ψ we consider is an approximation of the A^* running time and is defined as follows.

$$T(\zeta', \nabla) = J(\zeta', \nabla) \cdot (t_{h_{max}(\zeta')} + t_{gen}) \,, \tag{3.3}$$

where, for any heuristic function h, the term $t_{h_{max}(\zeta')}$ refers to the running time used for computing the $h_{max}(\zeta')$ of any state s. We assume that $t_{h_{max}(\zeta')}$ to be constant in the state space, which is a reasonable assumption for several heuristics such as `PDBs`. t_{gen} is the node generation time, which we also assume to be constant.

3.3 Estimating Tree Size and Running Time

In practice `GHS` uses approximations of J and T instead of their exact values. This is because computing J and T exactly would require solving ∇, and this is what we obviously want to avoid.

We use the Culprit Sampler (`CS`) introduced by Barley et al., (2014) and the Stratified Sampling (`SS`) algorithm introduced by Chen (1992), for computing $\hat{J}$ and $\hat{T}$. Each method has its strengths and weaknesses, which we explore in Chapter 4, where we make an empirical comparison of GHS with other approaches.

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

3.4 Culprit Sampler (CS)

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

Definition 3.4.1. *(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 tuple F, the counter C_F denotes the total number of children of nodes n in the search tree with $F(n) = F$.*

Definition 3.4.2. *(b-culprit) Let $\zeta = \{h_1, h_2, ..., h_M\}$ be a set of heuristics and b a lower bound on the solution cost of ∇. 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 tuple B, the counter C_B denotes the total number* *of children of nodes n in the search tree with $B(n) = B$. The vector $B(n)$, and thus the values of C_B for any B, depend on the bound b, which we assume to be fixed.*

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

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

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

The process described above is performed only once `GHS`'s execution. The value of $\hat{J}(\zeta', \nabla)$ for any subset ζ' of ζ if then computed by iterating over all b-culprits $\mathbf{B}$ 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.

$$\hat{J}(\zeta', \nabla) = \sum_{\mathbb{B} \in B} W(B) \tag{3.4}$$

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

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

3.5 Stratified Sampling (SS)

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

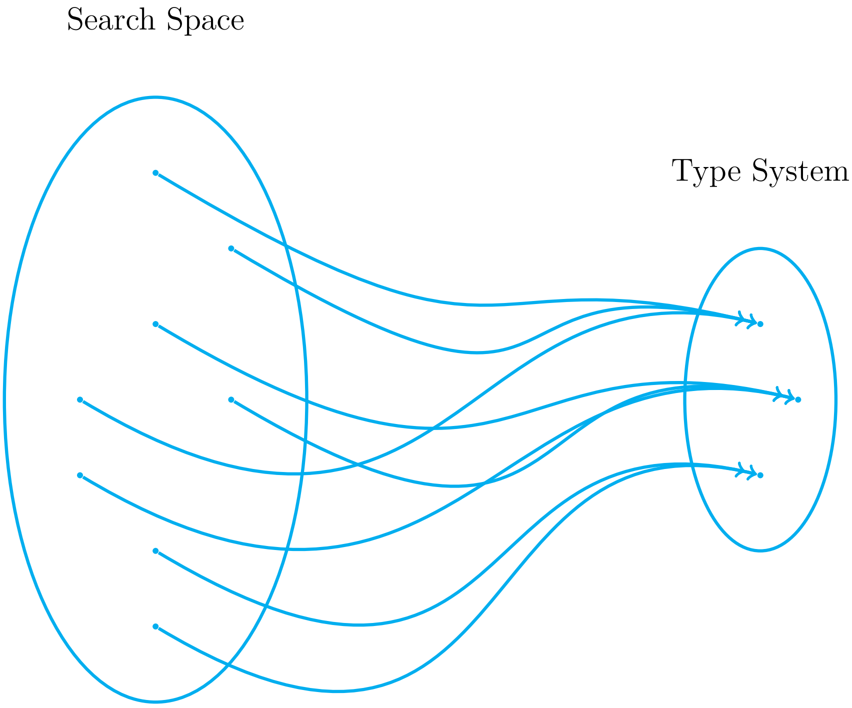

Figure 10 – Type system and the search space representation. Type system is a partition of the original search space.

3.5.1 Type System

A type system can be defined accounting for any property of nodes in the search tree. A common definition is the one that assigns two nodes to the same type if they have the same f-value (LELIS; ZILLES; HOLTE, 2013). A type system is effectively compressing the state space, as depicted in Figure 10, by mapping a set of states in the original state space to a single type.

Definition 3.5.1. *Type System Let $S = (N, E)$ be a search tree, where N is its set of nodes and for each $n \in N$, $\{n' | (n, n') \in E\}$ is n's set of child nodes. $TS = \{t_1, ..., t_k\}$ is a type system for S if it is a disjoint partitioning of N. If $n \in N$ and $t \in TS$ with $n \in t$, we write $TS(n) = t$.*

SS is able to approximate functions of the form $\varphi = \sum_{n \in S} z(n)$, where z is a function assigning a numerical value to a node, consequently φ is a numerical property of S. For example, if $z(n) = |children(n)|$ for all $n \in S$, then φ is the size of the tree. Instead of summing all the z-values from the tree, SS considers subtrees rooted at nodes of the same type will have equal values of φ and only one of each type, chosen at random, is expanded. Given a search tree S and a type system TS, SS predicts φ in the following way: First, the search tree is sampled, returning the set of *representative-weight* pairs, with one such pair for every unique type seen during sampling. Second, in the pair $\langle s, w \rangle$ in A for type $t \in TS$, n is the unique node of type t that was expanded during search and w is an estimate of the number of nodes of type t in the tree. Finally, φ is then approximated by $\hat{\varphi}$, defined as, $\hat{\varphi} = \sum_{\langle s,w \rangle \in A} w \times z(n)$.

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

Algoritmo 2: SS, a single probe

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

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

```
A[0] ← {⟨s*, 1⟩}
i ← 0
while A[i] is not empty do
    for each element ⟨s, w⟩ in A[i] do
        for each child n̂ of n do
            if g(n̂) + h(n̂) ≤ d then
                if A[i+1] contains an element ⟨n', w'⟩ with TS(n') = TS(n̂) then
                    w' ← w' + w
                    with probability w/w', replace ⟨n', w'⟩ in A[i+1] by ⟨n̂, w'⟩
                else
                    insert new element ⟨n̂, w⟩ in A[i+1]
    i ← i + 1
```

In SS the types are required to be partially ordered: a node's type must be strictly greater than the type of its parent. This can be guaranteed by adding the depth of a node to the type system and then sorting the types lexicographically. That is why in our implementation of SS types at one level are treated separately from types at another level by the division of A into groups $A[i]$, where $A[i]$ is the set of *representative-weight* pairs for the types encountered at level i. If the same type occurs on differente levels the occurrences will be treated as if they were different types − the depth of search is implicitly included into all of our type systems.

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

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

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

SS must also be able to estimate the values of $\hat{J}(\zeta')$ and $\hat{T}(\zeta')$ for any subset ζ' of ζ. This is achieved by using SS to estimate b-culprits (see Definition 3.4.2) instead of the search tree size directly. Similarly to CS, SS used h_{min} of the heuristics in ζ to decide when to prune a node (see Line 6 of Algorithm 2) while sampling. This ensures that SS expands a node n if A* employing at least one of the heuristics in ζ would expand n according to bound d. The C_B counter of each b-culprit B encountered during SS's probe is given by,

$$C_B = \sum_{\langle n,w \rangle \in A \wedge B(n)=B} w \tag{3.5}$$

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

3.6 An Example of an SS Execution

In Figure 11 we see an example of an SS run. Nodes have different types if they have different colors or are in different levels of the search tree.

In level 1 we have the root node, and its w is set to one. In level 2 three nodes are generated by the root node. The nodes in level 2 have the following types: red and blue, and initially each node receives the same w from their parent (in this case $w = 1$). Since SS assumes that nodes of the same type root subtrees of the same size, then only one node of each type needs to be expanded. In level 2 there are two nodes with the red type. In that way, we choose randomly one of them to expand. Let us suppose we choose the right red node. Then, we have to update the number of nodes with the red type by summing their w-values. Since both nodes of the red type have $w = 1$, then we sum the w-values results in $w = 2$.

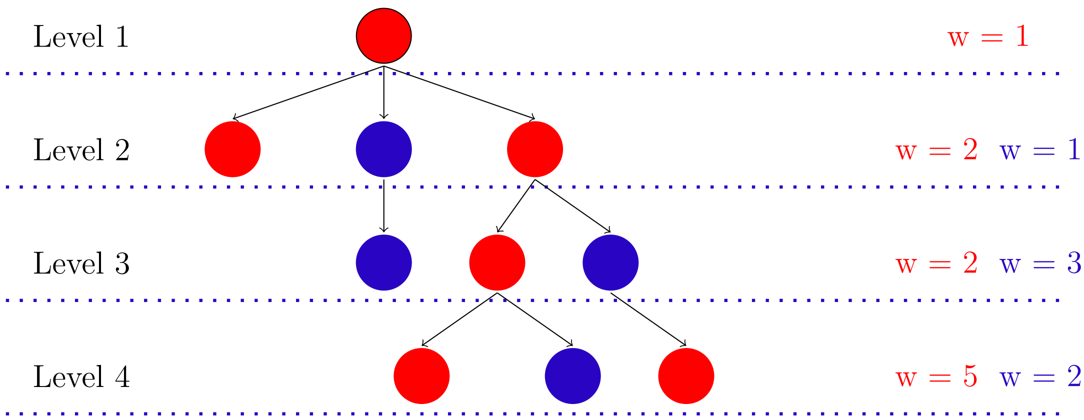

Figure 11 – SS uses the type system to predict the size of the search tree. Different color in each level represent a different type. w represents the number of nodes of any type in some level.

Let us now suppose that nodes in level 2 are expanded. The blue node generates one node of the blue type and the red node generates two nodes with the following types: red and blue. To compute the w-values of the nodes at level 3 we do the following. The w-value of nodes of the red type is computed by summing up the w-values of all parents that generated a node of the red type at level 3. In this case only one node at level 2 generated a node of the red type at level 3. Since this node has a w-value of 2, then SS estimates that there is 2 nodes of the red type at level 3 (red type's w-value). Two nodes at level 2 generate a blue node at level 3, thus we have to sum their w-values, resulting in $w = 2 + 1 = 3$ for blue nodes.

Finally, the number of nodes expanded in the search tree is obtained summing all w. In this example, `SS` estimates that the size of the search tree sampled is $1+2+1+2+3+5+2=16$.

If we run the algorithm again maybe in level 2 instead of choosing the right red node we could choose the left red node and the nodes generated in level 3 would be different. As a consequence, the predicted number of nodes generated by `SS` could be different than the current estimate. In other words, each probe of the algorithm could result in a different value of the estimated tree size. In order to obtain accurate estimates of the search tree size one usually performs several probes of `SS` and average the results.

4 Empirical Evaluation

This chapter is organized as follows. First we show that SS is able to produce very accurate estimates of the number of nodes generated by IDA*, a search algorithm that does not detect duplicated nodes during search, while solving domain-independent planning problems. Next, we show that SS produces very poor estimates of the number of nodes generated by A* on the same problem domains. Although SS is unable to produce good predictions for A*, we show empirically that the SS's predictions allow one to select, from a pool of heuristics, the one that will result in the smallest A* search tree size.

After testing the effectiveness of SS, we test whether the heuristic subsets selected by GHS (using both SS and CS) are near optimal. That is, we are interested in knowing how $J(\zeta', \nabla)$ compares with $J(\zeta, \nabla)$, which is provably optimal.

Finally, we compare the effectiveness of GHS with other state-of-the-art planners.

4.1 SS for Predicting the IDA* Search Tree Size

SS was shown to produce good predictions of the IDA* search tree size (LELIS; ZILLES; HOLTE, 2013). IDA* is a heuristic search algorithm that uses the same cost function used by A* but it searches the state space in a depth-first manner. Moreover, in contrast with A*, IDA* does not detect duplicated nodes.

In this section we show that SS is able to produce good predictions for IDA* also in domain-independent planning, and we will show that it produces very inaccurate predictions for A*. In this experiment SS estimates the search tree size generated by IDA* using a consistent heuristic. SS estimates the size of the search tree up to some defined f-layer in the tree.

We first run IDA* in domain-independent planning benchmark problems with a 30-minute time limit. Then we run SS limited by different f-layers encountered in the IDA* searches. We experiment with the following number of probes: 1, 10, 100, 1000 and 5000.

We use the *relative unsigned error* (LELIS; ZILLES; HOLTE, 2012), which we call relative-error, for short, to measure prediction accuracy. The relative-error is shown in Equation 4.1 below.

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

Table 1 – Precision of SS against IDA* for 1, 10, 100, 1000 and 5000 probes using *hmax* heuristic.

Domain	*hmax*												n
	IDA*−exp	IDA*−time	relative−error					time					
			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

- PI is the set of all instances of a problem domain.
- $Pred(s, d)$ is SS's estimated search tree size for start state s and cost bound d.
- $R(s, d)$ is the actual number of nodes expanded by IDA* for start state s and cost bound d.

A relative error of 0.0 means that SS was able to produce perfect predictions.

Table 1 shows the average number of nodes expanded by IDA* represented by the column "IDA*−exp", the average running time of IDA* represented by "IDA*−time". SS's relative-error and running time for different number of probes. The column "n" shows the number of problems that were used to compute the relative-error. The heuristic used in this experiment is *hmax*. Under each number of probes we show their respective relative-error and time performed by the algorithm.

The relative-error tends to reduce as one increases the number of probes. For example, in 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 running time, while the number of probes increase, SS needs to spend more time approximating the size of the search tree. That is why the overall running time of SS increases as one increases the number of probes. For example, in 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, which happens due to the stochastic nature of SS.

Each domain of the 2011 IPC benchmark contains 20 instances. The results of Floortile in Table 1 were computed using only 2 instances ($n = 2$). This is because IDA* was able to fully complete an iteration for its initial cost bound for only two instances (opt-p01-001 and opt-p03-006). In summary, we showed that for 2011 IPC domains, SS is able to produce good predictions of the search tree expanded by IDA*.

4.2 SS for Predicting the A* Search Tree Size

Table 2 shows the relative-error of SS while predicting the number of nodes generated by A*. In this experiment we tested different heuristic functions: iPDB (HASLUM et al., 2007), LM-Cut (POMMERENING; HELMERT, 2013) and M&S (NISSIM; HOFFMANN; HELMERT, 2011). Column "n" shows the number of instances solved by A* using the three heuristics. For this experiment we only use the instances that are independently solved by A* using each of the three heuristics within a 30-minute time limit. The columns with A* represents the average of number of nodes expanded by A*. The column with relative-error represents the relative-error for the solved instances. We used 500 probes in this experiment.

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*	relative-error	A*	relative-error	A*	relative-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

As can be observed in Table 2, SS usually overestimates the number of nodes expanded by A* by several orders of magnitude, for example, in Barman SS's relative-error is 1.26×10^{36} (M&S), 8.68×10^{31} (iPDB), and 2.21×10^{30} (LM-Cut). The exception in Table 2 is Openstacks, where SS produces accurate predictions. This is probably because the A* search tree does not have a large number of duplicated nodes in this domain. That is, the A* search tree is similar to the IDA* search tree for Openstack problems.

4.3 Approximation Analysis for SS and A*

Although SS is not able to produce good predictions of the A* search tree size, here we show empirically that SS allows one to make good heuristic subset selections. In this experiment we use the following heuristics: iPDB, LM-Cut, and 10 independently generated GA-PDBs (EDELKAMP, 2007).

Our goal with this experiment is to show that in order to select heuristics, the predictions produced by `SS` do not have to be accurate in absolute terms, but they have to be accurate in relative terms. That is, if A* using heuristic h_1 expands fewer nodes than when using h_2, then `SS`'s predictions could be arbitrarily inaccurate for both h_1 and h_2, as long as the prediction estimates that A* expands fewer node when using h_1.

In order to investigate whether `SS` is able to allow one to select the best heuristic in a set of size two: $\{h_1, h_2\}$, we plot the ratio $\hat{J}(h_1)/\hat{J}(h_2)$ (the predicted search tree size of h_1 to the predicted search tree size of h_2) and $J(h_1)/J(h_2)$ (the actual number of nodes generated by A* using h_1 to the number of nodes generated by A* using h_2); The plot is divided in four quadrants, as shown in Figure 12. If a data point falls in quadrants II and III it means that `SS` is able to correctly select the heuristic that will yield the smallest search tree size. If the point falls in quadrants I or IV it means that `SS` selects the heuristic that will yield the largest tree size.

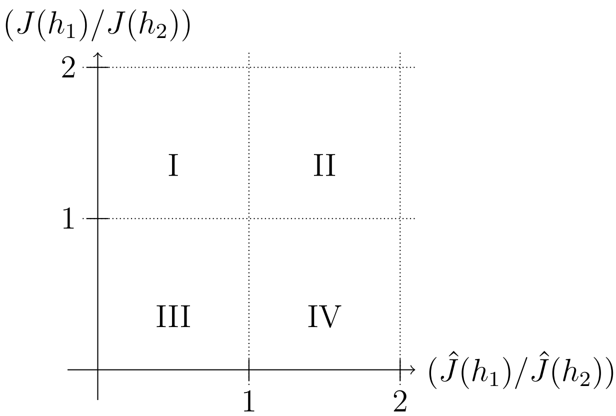

Figure 12 – Cartesian Plane with domain $\langle 0, 2\rangle$ and range $\langle 0, 2\rangle$. Note that the ratio values could be larger than 2, and we assume that any ratio larger than 2 is automatically set to 2 in this experiment.

A data point falls in each quadrant (I, II, III, and IV) if the corresponding boolean expression is true.

I - $J(h_1) > J(h_2) \land \hat{J}(h_2) \geq \hat{J}(h_1)$

II - $\Big(J(h_1) > J(h_2) \land \hat{J}(h_1) > \hat{J}(h_2)\Big) \lor J(h_1) = J(h_2)$.

III - $J(h_2) > J(h_1) \land \hat{J}(h_2) > \hat{J}(h_1)$.

IV - $J(h_2) > J(h_1) \land \hat{J}(h_1) \geq \hat{J}(h_2)$.

We assign a data point to quadrant II if $J(h_1) = J(h_2)$ because in this situation A* generates the same number of nodes when using either h_1 or h_2.

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.

For each problem instance of the `IPC` 2011 benchmark we generate 12 heuristics: iPDB, `LM-Cut`, and 10 independently generated `GA-PDBs`, and we generate a data point ($\hat{J}(h_1)/\hat{J}(h_2)$ and $J(h_1)/J(h_2)$) for each possible pairwise combination of heuristics. In Table 3 we present the percentage of data points in quadrants II and III. This percentage is above 50% for all domains, meaning that `SS` is able to correctly select the heuristic that yields smaller search trees in at least half of the binary decisions tested (i.e., should one choose h_1 or h_2?). This experiment indicates that although `SS` produces inaccurate estimates of the A* search tree size, `SS` is capable of selecting the heuristic that will allow A* to generate fewer nodes in the majority of the cases tested.

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

In the previous experiment we tested whether `SS` allows one to correctly select the heuristic, from a set of size two, that yield the smallest A* search tree size. Next, we test whether the approximations of $\hat{J}$ provided by `CS` and `SS` allow `GHS` to make good subset selections from larger sets, with thousands of heuristics. This test is made by comparing $J(\zeta^{'})$ with $J(\zeta)$, which is provably minimal. We restrict our experiment to J because, in contrast with J, there is no easy way to find the minimum of T for a subset in general.

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 as described by Barley et al., (2014)—we call each `PDB` generated by this method a `GA-PDB`. 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 million entries each. The `GA-PDBs` are generated for our ζ set. `GHS` then selects a subset $\zeta^{'}$ of ζ. Finally, we use $h_{max}(\zeta^{'})$ and $h_{max}(\zeta)$ to independently try to solve ∇. We call the system which uses A* with $h_{max}(\zeta)$ the `Max` approach. For `GHS` we allow 600 seconds for selecting $\zeta^{'}$ and for running A* with $h_{max}(\zeta^{'})$, and for `Max` we allow 600 seconds for running A* with $h_{max}(\zeta)$. Since we use 1,200 seconds

Table 4 – Ratios of the number of nodes generated using $h_{max}(\zeta')$ to the number of nodes generated using $h_{max}(\zeta)$.

Domain	SS		CS		$\|\zeta\|$	n
	Ratio	$\|\zeta'\|$	Ratio	$\|\zeta'\|$		
Barman	1.11	17.70	1.50	30.25	5,168.50	20
Elevators	11.50	2.00	1.04	21.00	168.00	1
Floortile	1.02	43.07	1.02	42.36	151.29	14
Openstacks	1.00	1.00	1.00	1.00	390.69	13
Parking	1.00	5.53	1.01	7.26	21.74	19
Parcprinter	3.62	1.00	2.21	13.00	1,189.00	1
Pegsol	1.00	31.00	1.00	57.00	90.00	2
Scanalyzer	1.23	30.57	1.57	19.43	72.86	7
Sokoban	1.32	7.00	1.01	24.00	341.00	1
Tidybot	1.00	2.35	1.00	8.59	3,400.18	17
Transport	1.00	14.70	1.03	14.30	171.70	10
Visitall	1.03	99.33	1.19	48.67	256.33	3
Woodworking	32.43	3.00	199.66	5.00	1,289.00	5

to generate the heuristics, both `Max` and `GHS` are allowed 1,800 seconds in total for solving each problem. In this experiment we test both `CS` and `SS`.

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

Table 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 generated up to the largest f-layer which is fully expanded by all approaches tested (`Max`, `GHS+SS` and `GHS+CS`). We only present results for instances that are not solved during `GHS`'s `CS` sampling process. Column "n" shows the number of instances used to compute the averages of each row. We also show the average number of `GA-PDBs` generated ($|\zeta|$) and the average number of `GA-PDBs` selected by `GHS` ($|\zeta^{'}|$). This experiment shows that for most of the problems `GHS`, using `CS` or `SS`, is selecting good subsets of ζ. For example, in Tidybot `GHS` selects only a few `GA-PDBs` out of thousands when using either `SS` or `CS`.

The exceptions in Table 4 are the ratios for Elevators, Parcprinter, and Woodworking. In all three domains `GHS+SS` failed to make good heuristic selections because `SS` was not able to sample "deep enough" into the problem's search tree. For instance, for the Elevators instance `SS` was able to perform only 245 probes with the initial f-boundary within the time limit of 300 seconds. `GHS+CS` is able to make better selections for this instance because it is able to sample deeper into the search tree. For the Parcprinter instance as well as the Woodworking instances neither `SS` nor `CS` are able to sample deep enough to make good heuristic selections. These results suggest that `SS` could benefit from

Table 5 – Coverage of SS, CS and Max on the 2011 IPC benchmarks. For GHS using only GA-PDBs heuristics.

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

an adaptive approach for choosing SS's number of probes. For example, for the Elevators instance SS could have performed better by sampling deeper into the tree with fewer probes. We intend to investigate this direction in future works.

In total, out of the 280 instances of the IPC 2011 benchmark set, GHS+SS solved 199 problem instances in this experiment, while GHS+CS only solved 194 problem instances. The numbers of instances solved are shown in Table 5.

4.5 Comparison with Other Planning Systems

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

4.6 Systems Tested

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

In addition to experimenting with all combinations of prediction algorithms (`CS` and `SS`) and objective functions (`Time`, `Size`), we also experiment with an approach that minimizes both the search tree size and the running time as follows. First we create a pool of heuristics ζ composed solely of `GA-PDB` heuristics, then we apply `GHS` while minimizing tree size and using `SS` as predictor. 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 `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`.

We compare the coverage of the `GHS` 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 − Under the principle of the folk tale where the collaboration are better ingredients than stones in the preparation of a soup is created a Portafolio planner, where the whole is greather than the sum of its parts. This parts are Fast-Downward, the sequential optimization and sequential satisficing tracks of `IPC` 2011.) (HELMERT; RöGER; KARPAS, 2011), two versions of Symba (SY1 and SY2 − The symbolic paradigm is used instead of Heuristic search.) (TORRALBA, 2015), 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 6.

4.7 Discussion of the Results

The system that solves the largest number of instances is `Hybrid` (219 in total). Its combination of the strengths of both `SS` and `CS` has proven particularly effective on the Barman domain where `Hybrid`'s first selection contains good subsets of GA-PDBs and its

Table 6 – Coverage of different planning systems on the 2011 IPC benchmarks.

Domains	Hybrid	CS		SS		Sum	Max	RIDA*	SY1	SY2	StSp1	StSp2	iPDB	LM-Cut	M&S
		Time	Size	Time	Size										
Barman	7	5	4	4	4	4	4	4	10	11	4	4	4	4	4
Elevators	19	19	19	19	19	19	19	19	20	20	18	18	18	17	12
Floortile	15	14	14	14	14	14	14	14	14	14	14	14	14	8	10
Nomystery	20	20	20	19	19	20	20	20	16	16	20	20	14	19	18
Openstacks	17	17	15	17	15	15	11	15	20	20	17	17	15	17	17
Parcprinter	18	18	18	16	15	19	18	18	17	17	18	18	17	16	16
Parking	7	7	2	7	2	2	2	7	2	1	5	5	2	7	7
Pegsol	18	18	19	19	19	19	19	19	19	20	19	19	17	20	19
Scanalyzer	13	14	12	11	14	14	14	14	9	9	14	14	12	10	11
Sokoban	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
Tidybot	17	16	16	16	16	16	15	17	15	17	16	16	16	14	9
Transport	14	13	10	11	13	11	9	10	10	11	7	8	6	8	7
Visitall	18	18	18	15	18	18	18	18	12	12	16	16	10	16	16
Woodworking	16	15	15	12	16	16	16	15	20	20	15	15	15	9	9
Total	**219**	214	202	200	204	207	199	210	204	208	203	204	180	185	175

second selection recognizes when it must not add the iPDB and LM-Cut heuristics to the first selection. As a result, Hybrid solves more problems on this domain than any other GHS approach.

Time+CS also performs well—it solves 214 problems. Hybrid and Time+CS are superior to all other approaches tested. When minimizing $\hat{J}$ or maximizing Sum, GHS tends to add accurate heuristics to the selected subset, irrespective of their evaluation time. Thus, GHS frequently selects LM-Cut which is often the heuristic that most reduces the search tree size and most increases the sum of heuristic values. However, LM-Cut is computationally expensive, and often the search is faster if LM-Cut is not in ζ'. Both Hybrid and Time+CS often recognize when LM-Cut should not be in ζ' because they account for the heuristics' evaluation time.

Interestingly, while Time+CS solves 214 instances, Time+SS solves only 200. We conjecture that this is due to SS not detecting duplicate nodes during sampling and thus substantially overestimating A*'s running time. As a result, similarly to the Size and Sum approaches, Time+SS often mistakenly adds the accurate but expensive LM-Cut heuristic in cases where A* would be faster without LM-Cut.

RIDA* is the most similar system to GHS; it selects a subset of heuristics by using an evaluation method similar to CS. Starting with an empty subset, it evaluates all subsets of size i before evaluating subsets of size $i + 1$. This limits RIDA* to considering only tens of heuristics in its pool. Specifically, RIDA* uses 42 GA-PDBs, iPDB, and LM-Cut in its pool. By contrast, GHS may consider thousands of heuristics.

Selecting from large sets 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 a helpful heuristic by generating a large number of them. GHS is an effective method for selecting a small set of informative heuristics from a large set of mostly uninformative ones. This is illustrated in Table 6 on the Transport domain. Compared to systems which use multiple heuristics (StSp 1 and 2, and RIDA*), `Hybrid` solves the largest number of Transport instances, which is due to the selection of a few key GA-PDBs.

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

5 Concluding Remarks

This dissertation showed that the problem of finding the subset of a set of heuristics ζ for a given problem task is solved using models of the A* search tree size and models of the A* running time. We introduced the GHS algorithm which selects heuristics from ζ one at a time and is able to select good subsets ζ'. In addition to minimizing the search tree size and the running time, we also experimented with an objective function that accounts for the sum of heuristic values in the state-space, as suggested by Rayner et al., (2013).

Since we cannot compute the values of the objective functions exactly, GHS effectiveness depends on the quality of the approximations we can obtain. We tested two prediction algorithms, CS and SS, for estimating the values of the objective functions and showed empirically that both CS and SS allow GHS to make good subset selections with respect to the search tree size and running time. As a future work, we think that applying an adaptive number of probes for each problem is going to help us to enhance the selection of heuristics while using SS.

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

Bibliography

BÄCKSTRÖM, C.; NEBEL, B. Complexity results for sas+ planning. *Computational Intelligence*, Wiley Online Library, v. 11, n. 4, p. 625–655, 1995. Cited on page 6.

BADRUDDIN, S. A.; ALI, S. M. D. Recent developments in the optimization of space robotics for perception in planetary exploration. *CoRR*, abs/1505.00496, 2015. Disponível em: <http://arxiv.org/abs/1505.00496>. Cited on page 1.

BARLEY, M. W.; FRANCO, S.; RIDDLE, P. J. Overcoming the utility problem in heuristic generation: Why time matters. *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*, 2014. Cited 4 times in the pages 6, 13, 24, and 27

BONET, B.; GEFFNER, H. Planning as heuristic search. *Artif. Intell.*, Elsevier Science Publishers Ltd., Essex, UK, v. 129, n. 1-2, p. 5–33, jun. 2001. ISSN 0004-3702. Disponível em: <http://dx.doi.org/10.1016/S0004-3702(01)00108-4>. Cited on page 1.

BONET, B.; GEFFNER, H. Planning as heuristic search. *Artificial Intelligence*, Elsevier, v. 129, n. 1, p. 5–33, 2001. Cited on page 4.

BURCH, N. et al. Automating layouts of sewers in subdivisions. In: *Proceedings of the 2010 Conference on ECAI 2010: 19th European Conference on Artificial Intelligence*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2010. p. 655–660. ISBN 978-1-60750-605-8. Disponível em: <http://dl.acm.org/citation.cfm?id=1860967.1861096>. Cited on page 1.

CHEN, B. *Part2 AI as Representation and Search*. 2011. <http://www.slideshare.net/praveenkumar33449138/ai-ch2>. Accessed: 2016-24-01. Cited 3 times in the pages , 2, and 3

CHEN, P.-C. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, p. 295–315, 1992. Cited 4 times in the pages 6, 13, 15, and 17

CULBERSON, J. C.; SCHAEFFER, J. Pattern databases. *Computational Intelligence*, Wiley Online Library, v. 14, n. 3, p. 318–334, 1998. Cited 2 times in the pages 4 and 8

EDELKAMP, S. Automated creation of pattern database search heuristics. In: *Model Checking and Artificial Intelligence*. [S.l.]: Springer Berlin Heidelberg, 2007. p. 35–50. Cited 2 times in the pages 4 and 22

HART P. E.; NILSSON, N. J.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC*, IEEE, v. 4, n. 2, p. 100–107, 1968. Cited 3 times in the pages 1, 4, and 7

HASLUM, P. et al. Domain-independent construction of pattern database heuristics for cost-optimal planning. *AAAI*, v. 7, p. 1007–1012, 2007. Cited 2 times in the pages 4 and 22

HEIFETS, A.; JURISICA, I. Construction of new medicines via game proof search. In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. AAAI Press, 2012. (AAAI'12), p. 1564–1570. Disponível em: <http://dl.acm.org/citation.cfm?id=2900929.2900950>. Cited on page 1.

HELMERT, M. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)*, v. 26, p. 191–246, 2006. Cited on page 4.

HELMERT, M.; RöGER, G.; KARPAS, E. Fast downward stone soup: A baseline for building planner portfolios. In: *In Proceedings of the Workshop on Planning and Learning*. [S.l.]: AAAI, 2011. p. 28–35. Cited on page 27.

HOLTE, R. C. et al. Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence*, Elsevier, v. 170, n. 16, p. 1123–1136, 2006. Cited on page 1.

KORF, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, v. 27, p. 97–109, 1985. Cited on page 6.

KORF, R. E. A complete anytime algorithm for number partitioning. *Artif. Intell.*, Elsevier Science Publishers Ltd., Essex, UK, v. 106, n. 2, p. 181–203, dez. 1998. ISSN 0004-3702. Disponível em: <http://dx.doi.org/10.1016/S0004-3702(98)00086-1>. Cited on page 1.

KORF, R. E. A new algorithm for optimal bin packing. In: *Eighteenth National Conference on Artificial Intelligence*. Menlo Park, CA, USA: American Association for Artificial Intelligence, 2002. p. 731–736. ISBN 0-262-51129-0. Disponível em: <http://dl.acm.org/citation.cfm?id=777092.777205>. Cited on page 1.

KORF, R. E.; ZHANG, W. Divide-and-conquer frontier search applied to optimal sequence alignment. In: *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. AAAI Press, 2000. p. 910–916. ISBN 0-262-51112-6. Disponível em: <http://dl.acm.org/citation.cfm?id=647288.721444>. Cited on page 1.

LELIS, L.; ZILLES, S.; HOLTE, R. C. Fast and accurate predictions of ida*'s performance. In: *AAAI*. [S.l.: s.n.], 2012. Cited on page 20.

LELIS, L. H.; STERN, R.; STURTEVANT, N. R. Estimating search tree size with duplicate detection. In: *Seventh Annual Symposium on Combinatorial Search*. [S.l.: s.n.], 2014. Cited 2 times in the pages 6 and 17

LELIS, L. H.; ZILLES, S.; HOLTE, R. C. Predicting the size of ida* search tree. *Artificial Intelligence*, Elsevier, v. 196, p. 53–76, 2013. Cited 2 times in the pages 15 and 20

LI, Y.; HARMS, J. J.; HOLTE, R. Ida*_mcsp: a fast exact MCSP algorithm. In: *Proceedings of IEEE International Conference on Communications, ICC 2005, Seoul, Korea, 16-20 May 2005*. [s.n.], 2005. p. 93–99. Disponível em: <http://dx.doi.org/10.1109/ICC.2005.1494327>. Cited on page 1.

NISSIM, R.; HOFFMANN, J.; HELMERT, M. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In: *22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*. [S.l.: s.n.], 2011. Cited 3 times in the pages 4, 22, and 27

POMMERENING, F.; HELMERT, M. Incremental lm-cut. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. [S.l.: s.n.], 2013. p. 162–170. Cited on page 22.

RAYNER, C.; STURTEVANT, N.; BOWLING, M. Subset selection of search heuristics. *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, p. 637–643, 2013. Cited 3 times in the pages 6, 27, and 30

TORRALBA, Á. *Symbolic Search and Abstraction Heuristics for Cost-Optimal Planning*. Tese (Doutorado) — Universidad Carlos III de Madrid, 2015. Cited on page 27.