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.

Paper presented to the Federal University of Viçosa, as part of the requirements of Graduate Computer Science program, for obtaining the title of 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-
77 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

Errata sheet

Elemento opcional da ABNT (2011, 4.2.1.2). Exemplo:

FERRIGNO, C. R. A. **Tratamento de neoplasias ósseas apendiculares com reimplantação de enxerto ósseo autólogo autoclavado associado ao plasma rico em plaquetas**: estudo crítico na cirurgia de preservação de membro em cães. 2011. 128 f. Tese (Livre-Docência) - Faculdade de Medicina Veterinária e Zootecnia, Universidade de São Paulo, São Paulo, 2011.

Folha	**Linha**	**Onde se lê**	**Leia-se**
1	10	auto-conclavo	autoconclavo

Marvin Abisrror Zarate

On Selecting of heuristics functions for Domain−Independent planning.

Paper presented to the Federal University of Viçosa, as part of the requirements of Graduate Computer Science program, for obtaining the title of 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 a greedy method 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 the search algorithms.

(HOLTE et al., 2006) showed that search can be faster if several smaller pattern databases are used instead of one large pattern database. We introduce a greedy method for selecting a subset of the most promising heuristicss from a large set of heuristics 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.

List of Figures

List of Tables

Contents

Introduction

This thesis is concerned with cost−optimal state−space planning using the A* algorithm (HART P. E.; NILSSON; RAPHAEL, 1968). We assume that a pool, ζ, of hundreds or even thousands of heuristics is available, and that the final heuristic used to guide A*, h_{max}, will be defined as the maximum over a subset $\zeta^{'}$ of those heuristics ($h_{max}(s, \zeta^{'}) = max_{h \in \zeta^{'}} h(s)$). The choice of the subset $\zeta^{'}$ can hugely affect the efficiency of A*. For a given size N and planning task ∇, a subset containing N heuristics from ζ is optimal if no other subset containing N heuristics from ζ results in A* expanding fewer nodes when solving ∇.

Exists many problems of Artificial Intelligent (`AI`), such as: Finding the shortest path from one point to another in a game map, 8−tile−puzzle, Rubick's cube, etc. The level of difficulty to solve the problems mentioned are linked with the size of the search space generated.

State−space search algorithms have been used to solve the problems mentioned above. And in this dissertation we study the approach to solve problems in order to reduce the size of the search tree generated and the running time of the search algorithm using the best subset of heuristics selected from a large set of heuristics.

Part I

Preparation of the research

1 About the Problem

The purpose of this section if to motivate the problem.

1.1 Problem Statement and Motivation

Every problem of Artificial Intelligent can be cast as a state space problem. The state space is a set of states where each state represent a possible solution to the problem and each state is linked with other states if exists a function that goes from one state to another. In the search space there are many solutions that represent the same state, each of this solutions are called node. So, many nodes can be represented as one state. To find the solution of the problem is required the use of search algorithms such as: Depth First Search (`DFS`), which looks the solution of the problem traversing the search space exploring the nodes in each branch before backtracking up to find the solution. Another search algorithm is Breadth First Search (`BFS`), which looks for the solution exploring the neighbors nodes first, before moving to the next level of neighbors. The mentioned algorithms have the characteristic that when they do the search, they generate a larger search space. The search space that these algorithms generate are called Brute force search tree (`BFST`).

There are other types of algorithms called heuristics informed search, 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 to the near solution. The heuristic informed search generates smaller search tree in comparison to the `BFST`, because the heuristic guides the search exploring the nodes that are in the solution path and prunes the nodes which are not. Also, the use of heuristics reduce the running time of the search algorithm.

There are different approaches to create heuristics, such as: Pattern Databases (`PDBs`), Neural Network, and Genetic Algorithm. These systems that create heuristics receive the name of Heuristics Generators. And one of the approaches that have showed most successfull results in heuristic generation is the PDBs, which is memory-based heuristic functions 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.

Exists many ways to take advantage of all the heuristics that can be created, for

example: (HOLTE et al., 2006) showed that search can be faster if several smaller pattern databases are used instead of one large pattern database. In addition (DOMSHLAK; KARPAS; MARKOVITCH, 2010) and (TOLPIN et al., 2013) results showed that evaluating the heuristic lazily, only when they are essensial 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 algorithm 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 approaches to obtain the cardinality of the subsets of heuristics found.
- 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 heuristic functions based on the minimum evaluation cost of each heuristic.
- Develop an strategy to drop heuristics during the sampling that do not improve the objective function.
- Use Stratified Sampling (SS) algorithm for predicting the search tree size of Iterative-Deepening A* (IDA*). And use SS as our utility function.

1.3 Scope, Limitations, and Delimitations

We implemented our method in Fast Downward (HELMERT, 2006) and the problems we want to solve are the optimal domains benchmarks. Our meta−reasoning described in this thesis are going to try to solve the major number of problems using the most promising heuristics from a large set of heuristics. The exact way to create the large set of heuristics is beyond the scope of this thesis.

1.4 Justification

In the last few decades, Artificial Intelligence has made significant strides in domain−independent planning. The use of heuristics 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.

That is why we propose a meta−reasoning that will try to solve the major number of problems without relying on domain knowledge, to guide the A* search algorithm.

1.5 Hypothesis

This thesis will intend to prove the hypotheses listed below:

- **H1:** Probe that our objective function of selection is related with two mathematical properties: Monotonicity and Submodularity.
- **H2:** Reducing the size of the search tree generated helps to solve more problems.

1.6 Contribution of the Thesis

The main contributions of this Thesis are:

- Provide a prediction method to estimate the size of the search tree generated.
- Provide a meta−reasoning approach based on the size of the search tree generated.
- Provide a meta−reasoning approach based on the evaluation cost of each heuristic.

1.7 Organization of the Thesis

The Thesis is organized as follows:

1. In Part 1, the background of the thesis is provided which also includes our motivation and define the scope.
2. In Part 2, we review the State of the Art.
3. In Part 3, we introduce our meta−reasoning approach.
4. In Part 4, we introduce.
5. In Part 5, we .

6. We conclude in Part 6 by discussing further improvements and future work.

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

Part II

Literature Review

2 Background

The purpose of this section is to understand the problem.

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.

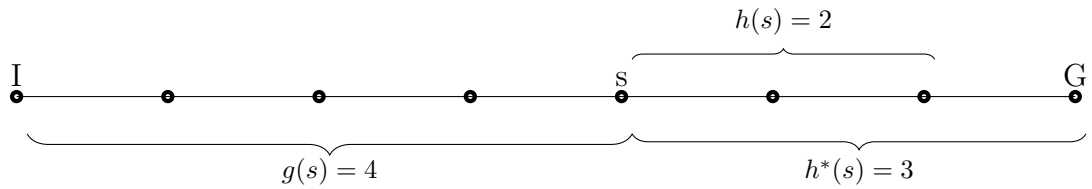

Figure 2: Heuristic Search: I: Initial State, s: Some Sate, G: Goal State

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.

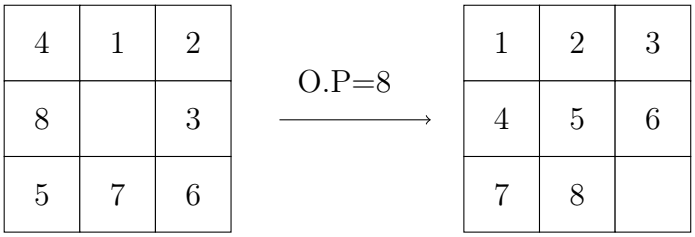

Figure 3: Out of place heuristic

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.

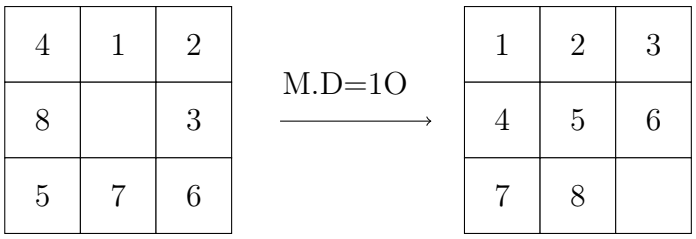

Figure 4: Manhatham distance heuristic

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.

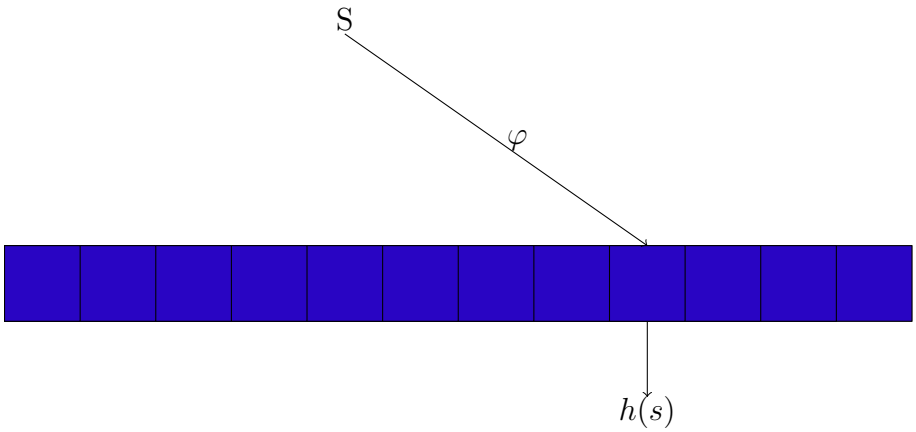

Figure 5: One heuristic of size M

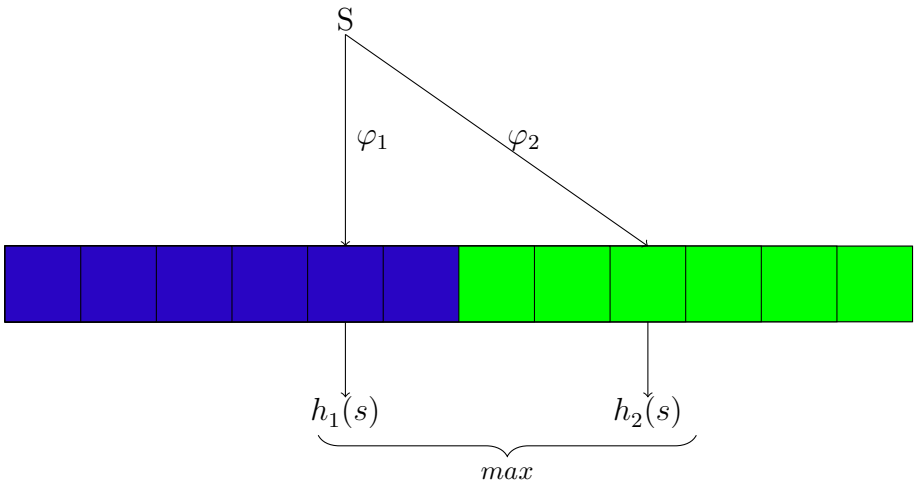

Figure 6: Two heuristics of size M/2

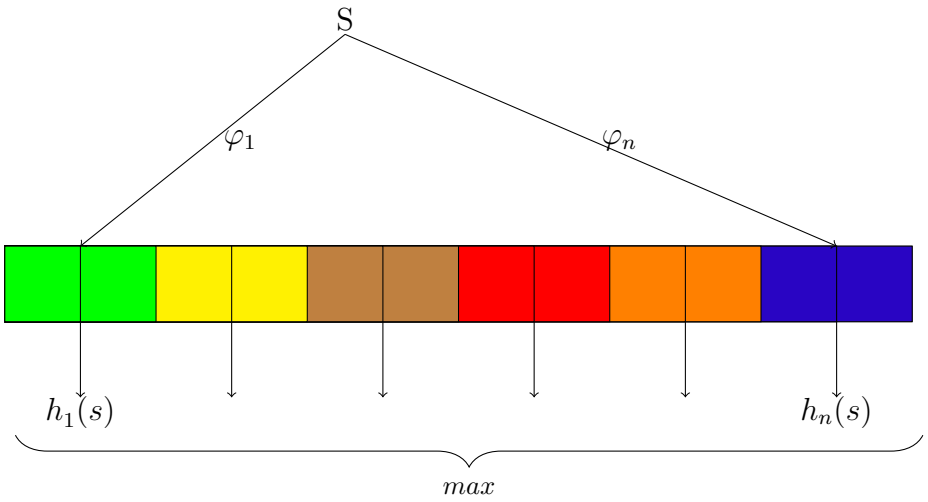

Figure 7: N heuristics of size M/N

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 Part.

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.

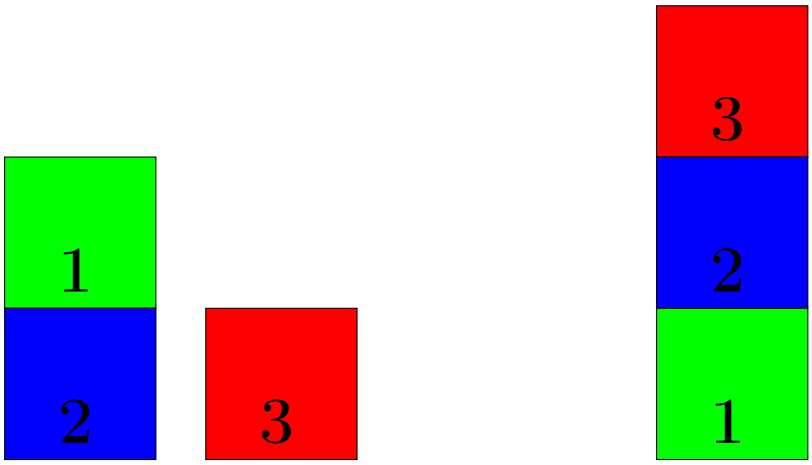

Figure 8: Blocks world with three blocks.

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.

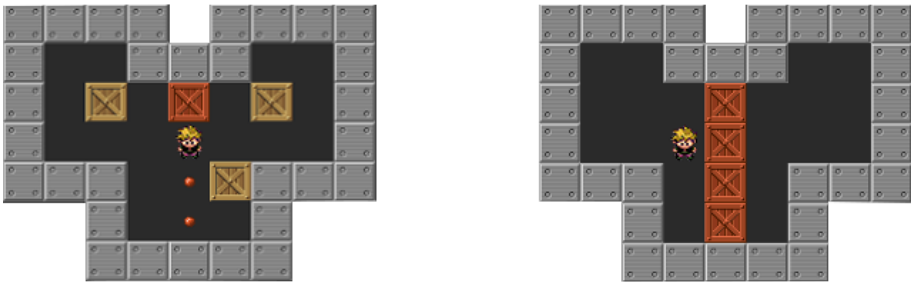

Figure 9: Sokoban with four blocks solved

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

Part III

Approach Proposal

3 Meta-Reasoning for selection

The purpose of this section is to introduce the meta−reasoning proposed.

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 ζ'_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 ζ'. 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 ζ'_{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 ζ'_{i-1} minimizes Ψ the most. `RGHS` returns ζ' 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.

3.1.1 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 3.1.1. *$J(\zeta^{'} \cup \{h\}) \leq J(\zeta^{'})$ 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 3.1.2. *J is supermodular.*
Proof. Let $A \subset B \subset \zeta$ and $h \in \zeta \backslash B$. By Lemma 3.1.1, $J(A) - J(A \cup \{h\}) \geq 0$ 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 3.1.1 and 3.1.2 are sufficient for using a result by (BUCHBINDER et al., 2014) to conclude the following.

Theorem 3.1.3. *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.*

3.2 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.

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

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 3.2.1. *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 $\zeta^{'}$ 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 $\zeta^{'}$. 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 3.1.3.

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 $\zeta^{'}$.
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 $\zeta^{'}$. 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 $\zeta^{'}$ sets.

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

Lemma 3.2.2. *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 3.1.2 is supermodular.

Since $T^{'}$ is supermodular, another result by (BUCHBINDER et al., 2014) and Lemma 3.2.2 allow us to conclude the following.

Theorem 3.2.3. *Let $\zeta^{'}$ be a subset selected by* ***RGHS****. Then $T(\zeta^{'}, \nabla)$ is within a factor of $\frac{e+1}{e} \approx 1.63$ of optimal.*

3.3 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 and$T^{'}$ model A*'s running time, we denote the approximation for both as $\hat{T}$.

We use the Culprit Sampler (CS) introduced by (BARLEY; 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 Part.

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

3.3.1 Culprit Sampler

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 3.3.1. *(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 3.3.2. *(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 **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.

3.4 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*

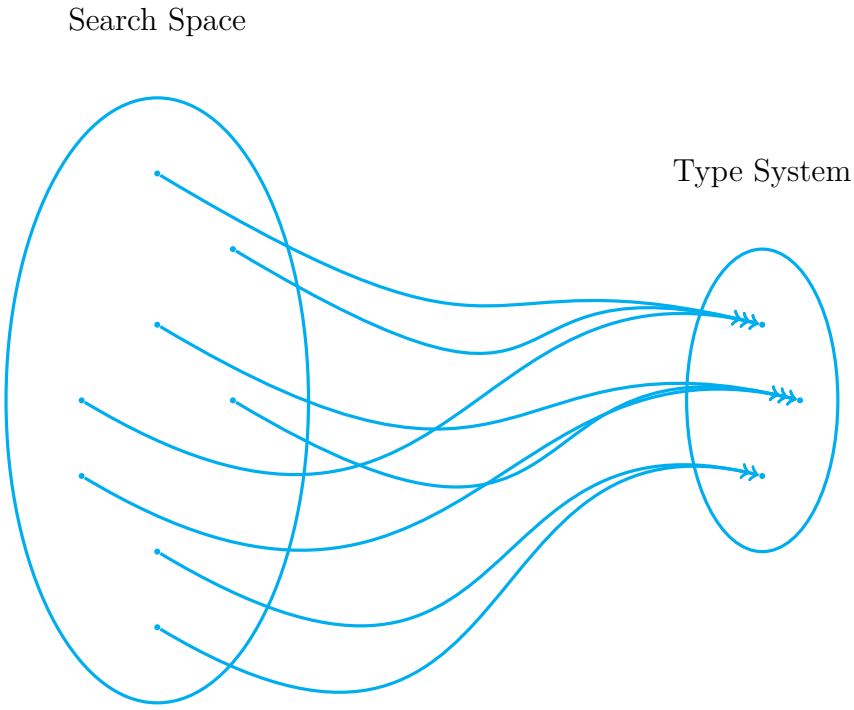

Figure 10: Type system and the search space Representation.

3.4.1 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*.

3.5 Search Space and Search Tree

(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)*.

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.

Definition 3.5.1. *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 so 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.

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
```

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 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 3.3.2) 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.

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.

Figure 11: The heuristic value is the position of the empty tile in a Specific state.

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` and the courners are labeled with `C` and the mediums with `E`. For this problem, for example, 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 blank 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, down left and right down. Then, the type would be (2,M). In the next board the empty tile is in the left top corner, 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).

In the Figure 13, we can see how *Type System* works. In the Level 1, we have the root node, we add the property called weight or (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 respectively, and each node recive the same W of the father. In the Level 2 we apply the concept of *Type System*, two states in the same level that have the same type (The same color) root subtrees of the same type and only one state per state must be explored. There are two nodes with type red in Level 2. So, we choose randomly one of them. Let's suppose we choose the right red node. Then, we

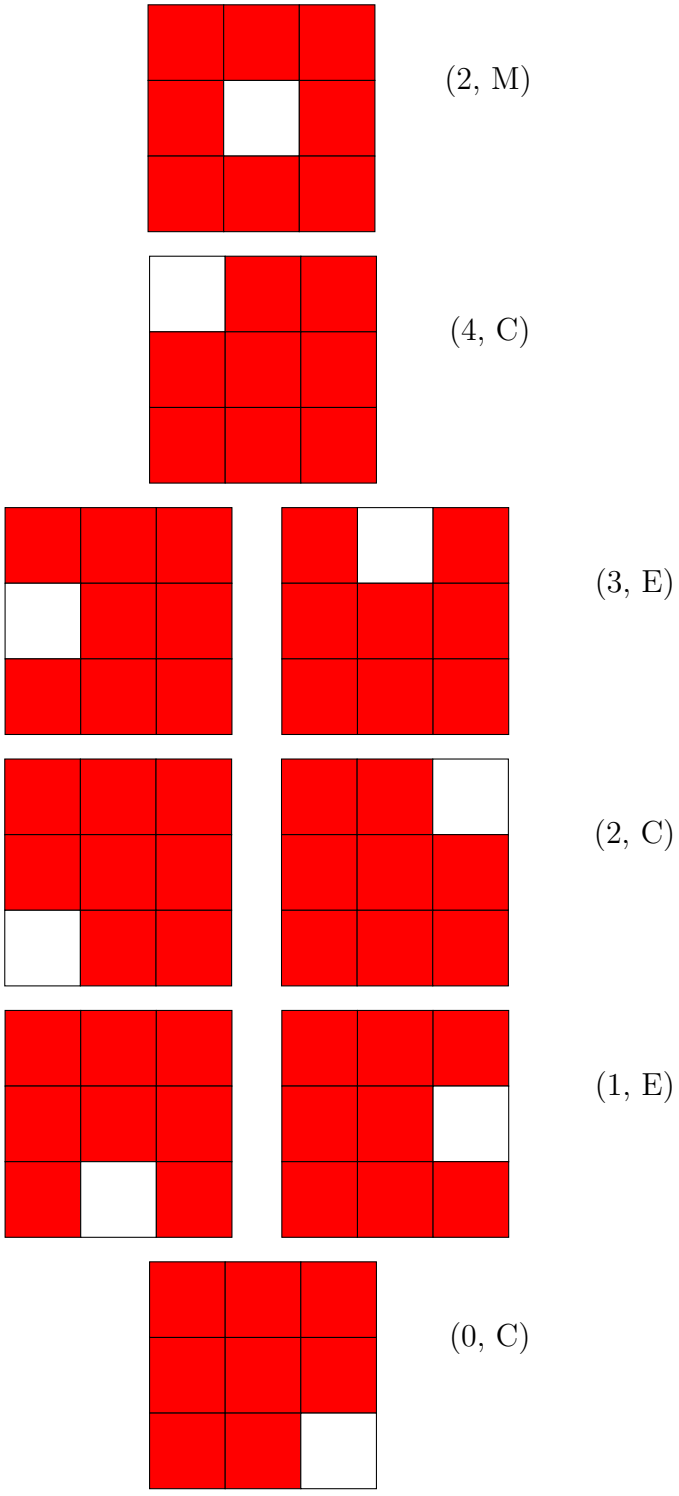

Figure 12: Using *type system*

have to update the number of nodes with the type red using the W, both red node types have $W = 1$, then we sum the W and the the new $W = 2$. So, 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 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 be the same W of the father. The father has the $W = 1$, then the child has the $W = 1$. 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 concept of *Type System* again. There are two nodes with type blue. So, we choose randomly one of them and update the 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. The red node expands two nodes of types red and blue and the blue node expands one of the red. How many nodes would be generated at Level 4? The answer is: $2 \times red + 3 \times red + 2 \times blue$. So, in the Level 4 we will have five nodes of type red and two nodes of type blue.

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

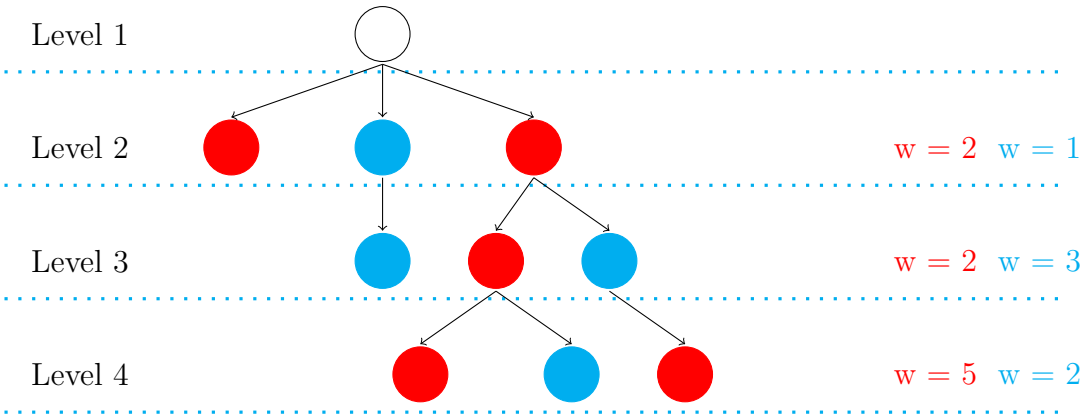

Figure 13: Search tree using Type System

4 Conclusão

Sed consequat tellus et tortor. Ut tempor laoreet quam. Nullam id wisi a libero tristique semper. Nullam nisl massa, rutrum ut, egestas semper, mollis id, leo. Nulla ac massa eu risus blandit mattis. Mauris ut nunc. In hac habitasse platea dictumst. Aliquam eget tortor. Quisque dapibus pede in erat. Nunc enim. In dui nulla, commodo at, consectetuer nec, malesuada nec, elit. Aliquam ornare tellus eu urna. Sed nec metus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

Phasellus id magna. Duis malesuada interdum arcu. Integer metus. Morbi pulvinar pellentesque mi. Suspendisse sed est eu magna molestie egestas. Quisque mi lorem, pulvinar eget, egestas quis, luctus at, ante. Proin auctor vehicula purus. Fusce ac nisl aliquam ante hendrerit pellentesque. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi wisi. Etiam arcu mauris, facilisis sed, eleifend non, nonummy ut, pede. Cras ut lacus tempor metus mollis placerat. Vivamus eu tortor vel metus interdum malesuada.

Sed eleifend, eros sit amet faucibus elementum, urna sapien consectetuer mauris, quis egestas leo justo non risus. Morbi non felis ac libero vulputate fringilla. Mauris libero eros, lacinia non, sodales quis, dapibus porttitor, pede. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi dapibus mauris condimentum nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Etiam sit amet erat. Nulla varius. Etiam tincidunt dui vitae turpis. Donec leo. Morbi vulputate convallis est. Integer aliquet. Pellentesque aliquet sodales urna.

Bibliography

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. *NBR 14724*: Informação e documentação — trabalhos acadêmicos — apresentação. Rio de Janeiro, 2005. 9 p. Citado na página 61.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. *NBR 14724*: Informação e documentação — trabalhos acadêmicos — apresentação. Rio de Janeiro, 2011. 15 p. Substitui a Ref. ABNT (2005). Citado na página 3.

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 33.

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 2 vezes nas páginas 33 and 49.

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 45, 46, 47, and 48.

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 49, 51, and 54.

CULBERSON, J. C.; SCHAEFFER, J. Pattern databases. *Computational Intelligence*, Wiley Online Library, v. 14, n. 3, p. 318–334, 1998. Citado na página 35.

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 28.

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 2 vezes nas páginas 23 and 34.

HASLUM, P. et al. Domain-independent construction of pattern database heuristics for cost-optimal planning. *AAAI*, v. 7, p. 1007–1012, 2007. Citado na página 48.

HELMERT, M. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)*, v. 26, p. 191–246, 2006. Citado na página 28.

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 48.

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 13, 28, and 37.

KNUTH, D. E. Estimating the efficiency of backtrack programs. *Mathematic of Computation*, v. 29, n. 129, p. 121–136, 1975. Citado na página 51.

KORF, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, v. 27, p. 97–109, 1985. Citado na página 33.

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 54.

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 3 vezes nas páginas 33, 51, and 52.

LELIS, L. H. Santana de. *Cluster-and-Conquer: A Paradigm for Solving State-Space Problems*. Tese (Doutorado) — University of Alberta, 2013. Citado na página 51.

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 46.

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 51.

RAYNER, C.; STURTEVANT, N.; BOWLING, M. Subset selection of search heuristics. *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, p. 637–643, 2013. Citado 3 vezes nas páginas 33, 45, and 46.

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 28.

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 54.

Appendix

APPENDIX A – Quisque libero justo

Quisque facilisis auctor sapien. Pellentesque gravida hendrerit lectus. Mauris rutrum sodales sapien. Fusce hendrerit sem vel lorem. Integer pellentesque massa vel augue. Integer elit tortor, feugiat quis, sagittis et, ornare non, lacus. Vestibulum posuere pellentesque eros. Quisque venenatis ipsum dictum nulla. Aliquam quis quam non metus eleifend interdum. Nam eget sapien ac mauris malesuada adipiscing. Etiam eleifend neque sed quam. Nulla facilisi. Proin a ligula. Sed id dui eu nibh egestas tincidunt. Suspendisse arcu.

APPENDIX B – Nullam elementum urna vel imperdiet sodales elit ipsum pharetra ligula ac pretium ante justo a nulla curabitur tristique arcu eu metus

Nunc velit. Nullam elit sapien, eleifend eu, commodo nec, semper sit amet, elit. Nulla lectus risus, condimentum ut, laoreet eget, viverra nec, odio. Proin lobortis. Curabitur dictum arcu vel wisi. Cras id nulla venenatis tortor congue ultrices. Pellentesque eget pede. Sed eleifend sagittis elit. Nam sed tellus sit amet lectus ullamcorper tristique. Mauris enim sem, tristique eu, accumsan at, scelerisque vulputate, neque. Quisque lacus. Donec et ipsum sit amet elit nonummy aliquet. Sed viverra nisl at sem. Nam diam. Mauris ut dolor. Curabitur ornare tortor cursus velit.

Morbi tincidunt posuere arcu. Cras venenatis est vitae dolor. Vivamus scelerisque semper mi. Donec ipsum arcu, consequat scelerisque, viverra id, dictum at, metus. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut pede sem, tempus ut, porttitor bibendum, molestie eu, elit. Suspendisse potenti. Sed id lectus sit amet purus faucibus vehicula. Praesent sed sem non dui pharetra interdum. Nam viverra ultrices magna.

Aenean laoreet aliquam orci. Nunc interdum elementum urna. Quisque erat. Nullam tempor neque. Maecenas velit nibh, scelerisque a, consequat ut, viverra in, enim. Duis magna. Donec odio neque, tristique et, tincidunt eu, rhoncus ac, nunc. Mauris malesuada malesuada elit. Etiam lacus mauris, pretium vel, blandit in, ultricies id, libero. Phasellus bibendum erat ut diam. In congue imperdiet lectus.

Annex

ANNEX A – Morbi ultrices rutrum lorem.

Sed mattis, erat sit amet gravida malesuada, elit augue egestas diam, tempus scelerisque nunc nisl vitae libero. Sed consequat feugiat massa. Nunc porta, eros in eleifend varius, erat leo rutrum dui, non convallis lectus orci ut nibh. Sed lorem massa, nonummy quis, egestas id, condimentum at, nisl. Maecenas at nibh. Aliquam et augue at nunc pellentesque ullamcorper. Duis nisl nibh, laoreet suscipit, convallis ut, rutrum id, enim. Phasellus odio. Nulla nulla elit, molestie non, scelerisque at, vestibulum eu, nulla. Ut odio nisl, facilisis id, mollis et, scelerisque nec, enim. Aenean sem leo, pellentesque sit amet, scelerisque sit amet, vehicula pellentesque, sapien.

ANNEX B – Cras non urna sed feugiat cum sociis natoque penatibus et magnis dis parturient montes nascetur ridiculus mus

Sed consequat tellus et tortor. Ut tempor laoreet quam. Nullam id wisi a libero tristique semper. Nullam nisl massa, rutrum ut, egestas semper, mollis id, leo. Nulla ac massa eu risus blandit mattis. Mauris ut nunc. In hac habitasse platea dictumst. Aliquam eget tortor. Quisque dapibus pede in erat. Nunc enim. In dui nulla, commodo at, consectetuer nec, malesuada nec, elit. Aliquam ornare tellus eu urna. Sed nec metus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

ANNEX C – Fusce facilisis lacinia dui

Phasellus id magna. Duis malesuada interdum arcu. Integer metus. Morbi pulvinar pellentesque mi. Suspendisse sed est eu magna molestie egestas. Quisque mi lorem, pulvinar eget, egestas quis, luctus at, ante. Proin auctor vehicula purus. Fusce ac nisl aliquam ante hendrerit pellentesque. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi wisi. Etiam arcu mauris, facilisis sed, eleifend non, nonummy ut, pede. Cras ut lacus tempor metus mollis placerat. Vivamus eu tortor vel metus interdum malesuada.

Index