On Selecting Heuristics Functions for Domain-Independent Planning.

Student: Marvin Abisrror Zarate
Advisor: Levi Lelis
Departamento de Informática
Universidade Federal de Vicosa
Viçosa, Brazil

July 2015

0.1 Introduction

Selecting a subset of heuristics functions in order to solve Domain-Indepentent Planning problems using A* search algorithm is an approach that selects the most promising heuristics from a bunch of heuristics that were generated based on the concepts of reducing the search tree and combination of primitive heuristics [**?**]
This project is concerned with cost-optimal state-space planning using the A* algorithm and finding strategies to make the A* search algorithm finishes quickly is our main aim. The approach can be applied for solving optimal domain problems in the Fast-Downward Planner. A few of these are: Elevators opt08 and opt11, Floortile opt11, Nomystery opt11, etc.
The approach we choose to select the most promising heuristics is Deterministic and is based on the ranking of heuristics according the number of nodes generated running the Stratification Sampling algorithm (SS). The SS system (Lelis 2013) is a Stocastic approach that produces accurate results predicting the number of nodes expanded by IDA*. Nevertheless, we are aware that SS can not be easily adapted to A* because A*'s duplicate pruning makes it very difficult to predict how many nodes will occur at depth $\boldsymbol{d}$ of A* search tree (the tree of nodes expanded by A*). That is the principal reason why we decided to create the concept of *regret* which is the cost from not to choose the heuristic that generates the lowest number of nodes generated or the best heuristic. If the *regret* for one heuristic is very close to zero when running A* then it is a promising heuristic.

This project is concerned with cost-optimal state-space planning using the A* algorithm [Hart and 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 ζ' of those heuristics ($h_{max}(s, \zeta') = \max_{h \varepsilon \zeta'} h(s)$). The choice of the subset ζ' can greatly affect the efficiency of A*. For a given size N and planning task $\triangledown$, a subset containing N heuristics from ζ is optimal if no other subset containing N heuristics from ζ result in A* expanding fewer nodes when solving $\triangledown$ (in the worst case-see the formal definition below).

The first contribution of this research is to show that the problem of finding the optimal subset of ζ of size N for a given problem task is supermodular, and therefore a greedy algorithm, which we call Greedy Heuristic Selection (GHS), that selects heuristics from ζ one at a time is guaranteed to produce a subset ζ' such that the number of nodes expanded by A* is no more than approximately 1.36 times optimal. An optimization procedure which is similar to ours is presented by citaRayner, but their procedure maximizes the average heuristic value. By contrast, GHS minimizes the search three size. GHS 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*) (e.g the SS system [Lelis et al., 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*). In this research we use a method for predicting the size of A*'s search tree which is based on the prediction method presented by BarleySantiagoOver
The second contribution of this research is an emperical evaluation of GHS in optimal domain-independent planning problems. Namely, there are experiments in the 2011 International Planning Competition (IPC) where the subset chosen by the GHS can be far superior, in terms of coverage, to defining $h_{\max}$ over the entire collection ζ and competitive with other state-of-the-art methods.
The system most similar to GHS is 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 combinations of size one before trying the combinations 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, GHS is able to evaluate a set of heuristics with thousand of elements.

0.2 Background

An SAS^+ [Bäckström and Nebel, 1995] is a 4-tuple $\triangledown = \langle vNO \rangle$

We use as experiments the competitions domains problems from the fast-downward benchmarks. The planner contains satisficing and optimization track problems, the satisficing problem are so hard to work and also is difficicult to found an optimal solutions, so they could be used to research because we pretend to predict the number of nodes expanded in the tree but we won't found an optimal solutions and also it would skip our analysis using consistent heuristics. In this research we will use the problems that require optimization in order to test our approach.

0.2.1 Problem Domains

Two of the optimal domain problems we have worked are barman and parking. The block domain contains easy instances that are very good to test our approach quickly.

Barman

This domain was created for the *International Planning Competition (IPC)* in *2011*. In this domain there is a robot barman that manipulates drink dispensers, glasses and shaker. The goal is to find a plan of the robot's actions that serves a desired set of drinks.

Parking

This domain was created for learning track *IPC-2008*. 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.

Blocks World

This domain had origin in the *IPC-2000*. 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. The problem probBLOCKS-4-0 is going to be our instance that will be used to explain our results in this Project. We chosen that because is a simple problem that can be resolved in less than few seconds and also contains the characteristics of duplicate nodes.

0.3 Problem Formulation

Let the *underlying search tree* (*UST*) be the full brute-force tree created from a connected, undirected and implicitly defined *underlying search graph* (*USG*) describing a state space. Some search algoritms 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 guidence.); we call this subtree the *expanded search tree* (*UST*). In this paper we want to estimate the size of the subgraph expanded by an algorithm searching the (*USG*); we call this subgraph the *expanded search graph* (*ESG*)

Definition 1 *(Canonical and Duplicate Nodes). A node n in the UST is called a duplicate node if there exists another node n' in the UST such that state(s) = state(n') and n' < n. A node that is not duplicate is called a canonical node.*

We define three graphs: The first one is the graph generated by dijkstra algorithm (*subtree of the UST*), which contains the tree without duplicate nodes. The second is the Brute Force Search tree generated by the Stratified Sampling algorithm which generate duplicate nodes and the last one is the tree generated by A* using a consistent heuristic which is also the tree we want to predict.

Formally, Let *G = (N, E)* be a graph representing an expanded search graph (ESG) of each graph explained above 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. We use the words edges and operators interchangeably. The prediction is to estimate the size of *N* without fully generate the ESG from A*.

0.4 Predicting the size of the Search Graph

We now explain how we pretend to achieve the prediction of the number of nodes expanded by A*.

0.4.1 Dijkstra Algorithm

Dijkstra Algorithm is an uninformed search algorithm. In our cost function the value of ***h*** is ***0*** for all the nodes of the search tree and the value ***g*** increase according to the level of the tree. It means that there is no guide to search the goal, in each iteratation the nodes from a level expand blindly according to the value of the level up to find the goal node. We collect the total number of nodes expanded by level without prunning. So, for our exemple in the level ***0*** we have one node which is initial state, in the level ***1*** we have ***4*** nodes, level ***2*** contains ***12*** nodes, etc. This information is collected in the planner executing A* using the dijkstra heuristic which assign the value zero for all the nodes in the search tree.

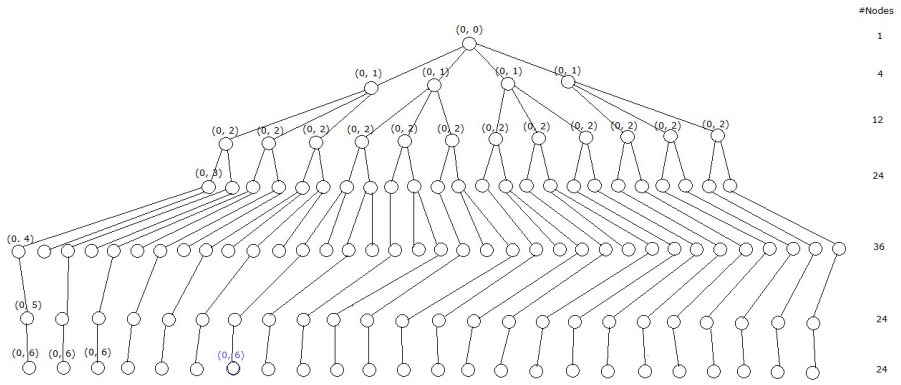

Figure 1: The par value $(\boldsymbol{h}, \boldsymbol{g})$ represent the heuristic value $\boldsymbol{h}$ and the cost to go from the start state to the current node $\boldsymbol{g}$. The tree generated by Dijkstra Search Algorithm to the problem of blocks worlds probBLOCKS-4-0.

In the Figure 1 are showed the total number of nodes expanded by level and also how the tree is distributed.

We also, present the Table 1 which explain the distribution of nodes expanded by level during dijkstra search algorithm.

Level	#Nodes expanded
0	1
1	4
2	12
3	24
4	36
5	24
6	24

Table 1: Number of nodes expanded by *Level* using Dijkstra algorithm.

0.4.2 *f-value* Distribution using Dijkstra and collecting *f-value* with consistent heuristic - Assumption

In order to keep the data consistent and validate the process of the prediction we make assumptions about what we expect to obtain, for example we make comparison between *f-value* distributions generated by two algorithms with the objective to see which algorithm is superior and fits fits well to our approach.

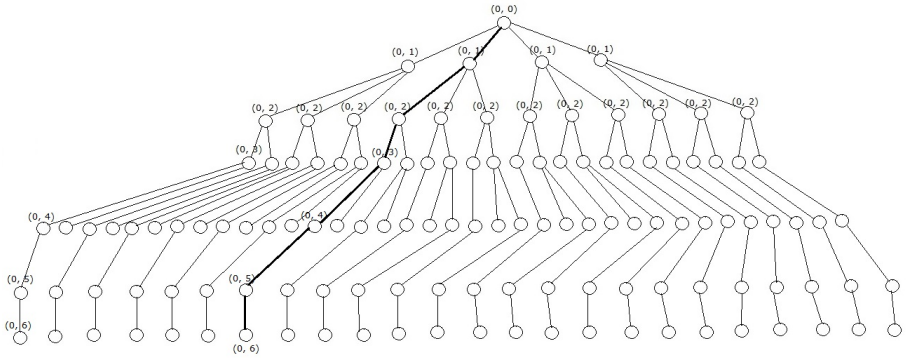

Figure 2: The tree generated by Dijkstra Search Algorithm and the solution route expressed by bold solid line.

Starting with our assumptions we present the distribution of *f-values* in the tree generated by dijkstra, but collecting the *f-values* using a consistent heuristic.

The search tree generated by Dijkstra Search Algorithm expand each node blindly with heuristic value zero. In the Figure 2 the solid line represents the connection from the parent nodes to their childs nodes. Each node is represented with the par ***(h, g)*** which represent the heuristic value $\boldsymbol{h}$ and the cost to go from the origin node to the current node $\boldsymbol{g}$.

Dijkstra Algorithm iteratively expand each node in each level up to found the solution. The initial node is given by ***(0, 0)*** and is located at level ***0*** and the goal node is ***(0, 6)*** and is at level ***6***. We expanded each node and collected the ***f-value*** based on a consistent

	12	15
4	6	1
	12	2
	14	6
	16	15
5	6	1
	12	2
	14	6
	16	15
6	6	1
	14	2
	16	6
	18	15

Table 2: *f-value* distribution expanding the graph using dijkstra and collecting the *f-value* using a consistent heuristic.

The Table 3 represent the distribution of *f-value* in the tree search expanded by dijkstra and using a consistent heuristic to collect the *f-value*. The first column represent the level in which the node is located. The second column is the *f-value* and the third column is the quantity, which represent the number of nodes in the level that have the same *f-value*.

In order to make a prediction we will require the Formula (2) on page 9. P_ex is the percentage of nodes expanded by level with *f-value* $\boldsymbol{d}$ less than to the threshold.

Level	Ni	$P_ex(s,d=6,Level)$
0	1	1
1	4	0.250
2	12	0.083
3	24	0.041
4	36	0.027
5	24	0.041
6	24	0.041

Table 3: *f-value* distribution expanding the graph using dijkstra and collecting the *f-value* using a consistent heuristic.

Applying the Formula (1) on page 9 we obtain the prediction.

	threshold	#nodes expanded	Prediction
probBLOCKS-4-0	6	7	7
probBLOCKS-4-1	10	11	11
probBLOCKS-4-2	6	7	7
probBLOCKS-5-0	12	16	16
probBLOCKS-5-1	10	14	14
probBLOCKS-5-2	16	29	29
probBLOCKS-6-0	12	13	13
probBLOCKS-6-1	10	11	11
probBLOCKS-6-2	18	44	44
	19	97	97
	20	399	399
probBLOCKS-7-0	17	8	8
	18	30	30
	19	35	35
	20	243	243

Table 4: Prediction of the number of nodes expanded by A* for the first 10 blocks world domain problems.

The Table 4 contains four columns: In the first column we have the blocks world domain problems chosen from fast-downward benchmark. We chosen 10 problems, as well as the number of the problem increases it turns more complex in terms of duplicate and expansions nodes. The second column in the threshold which represent the branches where the nodes are expanded. The third column is the number of nodes expanded by A* in each threshold, and the last one column is the prediction.

The results in the Table 4 are 100% accurate, and this predictions are the expected, because we are testing with the same data the two terms of the prediction formula (1).

0.4.3 *f-value* Distribution using DFS - Assumption

We have an assumption that if the distribution of the *f-values* of the tree generated by dijkstra have similar structure to the distribution of the *f-values* generated by Depth First Search (DFS). It would mean that the prediction using the number of nodes expanded by level with dijkstra will generate accurate values conforme to the number of nodes by *f-value* generated by A*.

Our assumption about the similarity of the *f-value* distribution of Dijkstra and Depth First Search was tested with the same problem mentioned above obtaining the following behaviors: The *f-value* distribution of Dijkstra was quickly elaborated, three steps were necesary, first the nodes expansions, the collection of the *f-value* of the nodes and then the generation of the *f-value* distribution file. Nevertheless, DFS last approximately less than 5 seconds to expand all the nodes and generate the *f-value* which make sense because it expands just 2581 nodes. For other instances for example the probBLOCKS-6-2 just expand the nodes takes approximitely 6 minutes.

So, try to generate the *f-value* distribution is expensive in computational time because expand nodes the way how DFS expand the nodes is a iterative task that could take much time. So we are going to present the distribution obtained from probBLOCKS-4-0. We found that the nodes ***(6, 0), (5, 1), (4, 2), (3, 3), (2, 4), (5, 1), (0, 6)*** were expanded in the same way as expanded by dijkstra algorithm, see the Table 3 on page 5. It means that the distribution of *f-value* equal to 6 in each level generated by DFS is the same as the *f-value* distribution with *f-value* equal to 6 in each level generated by Dijkstra.

According to our expience testing DFS, it could take much time to execute one instance if we set the depth of the search greater than the *f-value* we are going to produce prediction. Then, after analyze the consistent heuristic we are using we set up the depth to be equal to the initial heuristic value. In this way for the problem probBLOCKS-5-0 the depth is going to be ***6***.

Level	*f-value*	quantity
0	6	1
1	6	1
	8	3
2	6	1
	8	7
	10	8
3	6	1
	8	9
	10	27
	12	15
4	6	1
	8	11
	10	59
	12	88
	14	25
5	6	1
	8	12
	10	79
	12	239
	14	186
	16	15
6	6	1
	8	13
	10	103
	12	507
	14	832
	16	321
	18	15

Table 5: *f-value* distribution expanding the graph using depth first search and collecting the *f-value* using a consistent heuristic and threshold equal to the initial heuristic value.

The Table 5 represent the distribution of *f-value* in the tree search expanded by depth first search using a consistent heuristic and threshold twice the initial heuristic value. Depth first search found seven nodes with *f-value* equal to six in the levels ***0, 1, 2, 3, 4, 5***, and ***6*** which are the same distribution of *f-value* equal to six in Table 3 on page 5.

This result encourage to continue with our research, because the distribution of dijkstra is more quickly to obtain and also make prediction over the distribution in comparison with the depth first search tree generated that is more expensive to obtain.

	Threshold	#nodes expanded	Prediction
probBLOCKS-4-0	6	7	3.54892

probBLOCKS-4-1	4	1	1
	5	2	2
	6	3	3
	8	4	6
	10	14	6
probBLOCKS-4-2	6	10	5.12235
probBLOCKS-5-0	6	3	74.6
	7	4	75.6
	8	8	80.2
	10	12	94.2254
	11	13	97.6516
	12	30	730
probBLOCKS-5-1	6	3	2.7
	7	5	4.10741
	8	11	13.598
	10	27	60.2934
probBLOCKS-5-2	8	2	36
	10	4	38
	12	11	43.3429
	14	29	64.7618
	15	36	65.6421
	16	119	866
probBLOCKS-6-0	10	6	3.65083
	11	7	4.65083
	12	33	19.3541
probBLOCKS-6-1	10	21	17.1216
probBLOCKS-6-2	10	2	187
	12	3	1877.667
	13	4	189
	14	10	193.056
	15	11	194.343
	16	51	212.526
	17	59	217.676
	18	239	432.239
	19	263	481.725
	20	762	7057
probBLOCKS-7-0	12	1	1
	13	2	2
	14	4	3.33333
	16	8	7.61538
	17	9	9
	18	43	24.5351
	19	46	28.3212
	20	236	133.908

Table 6: Prediction of the number of nodes expanded by A* using the number of nodes by level from dijkstra and the *f-value* distribution obtained from DFS.

The Table 7 displays the results of apply the Formula (1) using two terms: The first one, the number of nodes expanded by dijkstra, and the other is the *f-value* distribution obtained from DFS. For the first row the terms are being obtained from the Table 1 and the Table 5.

0.4.4 Stratification Sampling

Knuth [1975] did experiments to predict the number of nodes expanded by IDA* under the assumption that all branches contained the same structure. He realized that the method was not effective when the branches were unbalanced. Chen [1992] addressed the problem with stratification of the search tree through a *type system.* He assumed that nodes of the same type at a level of the search tree would generate subtrees of the same size. Then, only one node of each type **SS** estimates the size of the tree.

The distribution the *f-values* is going to be obtained from the Brute Force Search, in this case the **SS**.

Type System

We require a property that allow us to represent each node in the state space search and this property can be defined using any information about the node. For example we could use a *type system* which counts how many children a node generates, or how many children the parent of the current node generates or the parent of the parent of the current node, the *f-value*, the $\boldsymbol{g}$ value or the

heuristic value ***h*** of the node, etc.
This property is used in **SS** to distinguish the nodes or stratified the state space. In our research we present a *type system* that use the heuristic value of the node ***h*** and the level in which the node is located ***L***. We use the following *type system*:

$$T(s) = (h(s), L(s))$$

Two nodes at a level of the search tree will have the same *type system T* and randomly one of them will be chosen because the asumption is that both generate the same subtree. Then, the subtree not chosen is removed from the tree and the number of nodes in the chosen node is upated withh the sum of nodes of the subtree of the current node and the nodes of the subtree removed.

	Threshold	#nodes expanded	Prediction
probBLOCKS-4-0	6	7	3.61117
probBLOCKS-4-1	4	1	1
	5	2	2
	6	3	3
	8	4	6
	10	14	6
probBLOCKS-4-2	6	10	5.12235
probBLOCKS-5-0	6	3	2.6
	7	4	3.6
	8	8	8.2
	10	12	21.8861
	11	13	25.3694
	12	30	459
probBLOCKS-5-1	6	3	2.7
	7	5	4.14
	8	11	13.8467
	10	27	59.4738
probBLOCKS-5-2	8	2	2
	10	4	4
	12	11	9.36077
	14	29	32.0507
	15	36	32.7296
	16	119	730
probBLOCKS-6-0	10	6	3.64713
	11	7	4.64713
	12	33	19.1604
probBLOCKS-6-1	10	21	17.311
probBLOCKS-6-2	10	2	2
	12	3	4
	13	4	4
	14	10	8.04615
	15	11	9.34474
	16	51	25.9412
	17	59	30.6944
	18	239	239.922
	19	263	281.115
	20	762	6317
probBLOCKS-7-0	12	1	1
	13	2	2
	14	4	3
	16	8	7.63636
	17	9	9
	18	43	23.8428
	19	46	27.6623
	20	236	196.257

Table 7: Prediction of the number of nodes expanded by A* using the number of nodes by level from dijkstra and the *f-value* distribution obtained from SS.

0.5 Prediction Formula

For a given state ***s*** and A* threshold ***d***, **N(s, d)** is the prediction of the number of nodes that A* will expand if it use ***s*** as its start state and does a complete search with an A* threshold of ***d***.

$$N(s,d) = \sum_{i=1}^{d} N_i(s,d) \tag{1}$$

Where *N(s, d)* is the number of nodes expanded by A* at level i when its threshold is d. One way to decompose $N_i(s,d)$ is as the product of two terms. [Zahavi et al., 2010]

$$N_i(s,d) = N_i(s) \times P_ex(s,d,i) \tag{2}$$

Where N_i is the number of nodes in level i of *BFS*, the brute-force search tree (*i.e.,* the tree created by dijkstra search without heuristic pruning.) of depth $\boldsymbol{d}$ rooted at start state $\boldsymbol{s}$, and $P_ex(s,d,i)$ is the percentage of nodes in level i of BFS(*i.e,* the distribution with threshold two times the heuristic of the initial state generated by the Stratified Sampling.) that are expanded by A* when its threshold is $\boldsymbol{d}$.

0.6 Prediction the number of nodes expanded by IDA*

The research also includes the prediction of the number of nodes expanded by the Iterative Deepening astar IDA*.

We have run IDA* in order to obtain the number of nodes expanded by bound up to find the solution. Then, for each bound obtained by IDA* we ran SS using each bound as a threshold and compared the results. The results are very accurate for all the IPC domains.

Time	bound	exp
0.32s	6	7

Table 8: Running IDA* for probBLOCKS-4-0.pddl we get the number of nodes expanded by bound.

The number of nodes expanded by bound are displayed in the table 8, the first column is the time spent to find all the nodes in a certain bound. The second column is the bound and the third column is the number of nodes expanded.

bound	ida* (exp)	ss (exp)
6	7	7

Table 9: Running SS for probBLOCKS-4-0.pddl we get the number of nodes expanded by bound.

In the table 9 is displayed the comparison of number of nodes expanded by ida* and ss given a certain bound.

0.7 Comparison between IDA* and SS

We say that a prediction system *V* dominates another prediction system *V'* if *V* is able to produce more accurate predictions in less time than *V'*. In our tables of results we highlight the runtime and error of a prediction system if it dominates its competitor. The results presented in this section experimentally show that SS employing greather number of probes dominates SS employing less number of probes.

In our experiments, prediction accuracy is measured in terms of the ***Relative Unsigned Error***, which is calculated as,

$$\frac{\sum_{s \in PI} \frac{|Pred(s,d) - R(s,d)|}{R(s,d)}}{|PI|} \tag{3}$$

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

In this experiment we also aim to show that SS produces accurate predictions when an inconsistent heuristic is employed. We show results for SS using the ***Th***, which is the type system based on the heuristics value.

In the table 10 we display the average values for the ida* value, ida* time, ss-error (***Relative Unsigned Error***), ss-time and the number of instances solved in each domain. The number of probes used in this table is 1000.

Domain	ida*	ida* time	ss-error	ss-time	n
barman-opt11-strips	—	—	—	—	0
blocks	2.09693e+09	10018.2	0.742198	1.36583	24
elevators-opt08-strips	9.5651e+07	20223.9	1.58258	858.46	4

elevators-opt11-strips	1.54128e+08	36205.9	2.84576	1178.09	2
floortile-opt11-strips	—	—	—	—	0
nomystery-opt11-strips	3.60878e+08	2514.16	0.474518	1.10286	14
openstacks-opt08-adl	—	—	—	—	0
openstacks-opt08-strips	2.0436e+06	15633	0.293452	772.126	7
openstacks-opt11-strips	4.16623e+06	35539	0.44249	1678.33	3
parcprinter-opt11-strips	3065.69	337.491	0.000228272	0.735385	13
parking-opt11-strips	3.06957e+08	7448.66	0.13508	20.748	5
pegsol-opt11-strips	54945.3	1181.48	0.326574	39.435	20
scanalyzer-opt11-strips	1.07152e+09	8163.32	0.342058	8.32571	7
sokoban-opt08-strips	280429	9.52462	0.00357773	3.65385	13
sokoban-opt11-strips	404338	13.4578	0.00236228	4.86889	9
tidybot-opt11-strips	1.13122e+06	355.4	0.0955455	84.1267	3
transport-opt08-strips	15	0.12	0	0.04	1
transport-opt11-strips	—	—	—	—	0
visitall-opt11-strips	7.60463e+06	31.6927	0.0745269	0.84	11
woodworking-opt08-strips	9.37402e+07	6484.69	0.0924333	27.8425	8
woodworking-opt11-strips	2.49863e+08	16954.9	0.247914	59.68	3

Table 10: Using ipdb heuristic and employing 1000 probes for SS algorithm

In the table 11 we display the average values for ss-error obtained after obtain the table 10 using 1, 10, 100, 1000 and 5000 probes.

	Probes					
	1	10	100	1000	5000	
Domain	ss-error	ss-error	ss-error	ss-error	ss-error	ida*
barman-opt11-strips	—	—	—	—	—	—
blocks	4.789	3.543	1.678	0.742	0.485	2096930000.000
elevators-opt08-strips	8.909	5.724	4.030	1.583	1.183	95651000.000
elevators-opt11-strips	13.495	9.043	3.898	2.846	1.771	154128000.000
floortile-opt11-strips	—	—	—	—	—	—
nomystery-opt11-strips	1429.530	1.759	1.065	0.475	0.167	360878000.000
openstacks-opt08-adl	—	—	—	—	—	—
openstacks-opt08-strips	0.376	0.301	0.279	0.293	0.290	2043600.000
openstacks-opt11-strips	0.316	0.449	0.457	0.442	0.437	4166230.000
parcprinter-opt11-strips	0.040	0.011	0.006	0.000	0.077	3065.690
parking-opt11-strips	3.584	1.488	0.399	0.135	0.288	306957000.000
pegsol-opt11-strips	1.515	0.662	0.342	0.327	0.376	54945.300
scanalyzer-opt11-strips	1.353	49.072	0.167	0.342	0.036	1071520000.000
sokoban-opt08-strips	0.112	0.026	0.005	0.004	0.003	280429.000
sokoban-opt11-strips	0.073	0.001	0.004	0.002	0.001	404338.000
tidybot-opt11-strips	1.759	0.637	0.416	0.096	0.036	1131220.000
transport-opt08-strips	0.000	0.000	0.000	0.000	0.000	15.000
transport-opt11-strips	—	—	—	—	—	—
visitall-opt11-strips	1.136	0.784	0.234	0.075	0.055	7604630.000
woodworking-opt08-strips	2.541	0.950	0.384	0.092	0.054	93740200.000
woodworking-opt11-strips	3.093	1.808	0.354	0.248	0.116	249863000.000

Table 11: Using ipdb heuristics and employing 1, 10, 100, 1000 and 5000 probes for SS algorithm

According to the results of the table 11 while the number of probes increases then the ss-error (***Relative Unsigned Error***) is very close to 0.00 (zero) which is the the perfect score.

0.8 Conclusions

In this Project we presented approaches to predict the number of nodes expanded by A* and IDA*, we take as assumption that the expansion of the number of nodes by level that dijkstra produces can be usefull to our prediction because according to the behavior of the distribution of nodes that dijkstra give us we can extend it using linear regression. Another information we get from the *f-value* distribution that Stratified Sampling give us when search the solution of the problem. Joining both informations we can use a mathematical formula of prediction.

Our approach of predicting the number of nodes expanded by A* produce good results for unit cost domains. And the approach of predicting the number of nodes expanded by A* using SS produce also good results.

Bibliography

Richard E Korf, Michael Reid, and Stefan Edelkamp. Time complexity of iterative-deepening-a*. *Artificial Intelligence*, 129(1): 199–218, 2001.

N. J.; Hart, P. E.; Nilsson and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC*, 4(2):100–107, 1968.

Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

Levi HS Lelis, Sandra Zilles, and Robert C Holte. Predicting the size of ida* search tree. *Artificial Intelligence*, 196:53–76, 2013.

Christer Bäckström and Bernhard Nebel. Complexity results for sas+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematic of Computation*, 29(129):121–136, 1975.

P.-C. Chen. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, pages 295–315, 1992.

Uzi Zahavi, Ariel Felner, Neil Burch, and Robert C Holte. Predicting the performance of ida* using conditional distributions. *Journal of Artificial Intelligence Research*, 37(1):41–84, 2010.