UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

NÍCOLAS CASAGRANDE DURANTI

A Non-Admissible Heuristic Function Based on Synchronized Abstract Plans

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Science

Advisor: Prof. Dr. André Grahl Pereira
Co-advisor: Augusto B. Corrêa

Porto Alegre
April 2023

ABSTRACT

Classical Planning is a traditional Artificial Intelligence problem that consists of finding a sequence of actions, called a plan, to achieve some desired goal given an initial state. We say that the plan cost is the sum of the costs of performing each action that composes the plan. A classical planning task is a compact description of a planning problem. It induces a transition system: a graph where the vertices represent the possible states and the edges represent the actions that transform the states. To solve the planning problem, one has to determine a path from the initial state to a goal state in the transition system.

Search algorithms combined with domain-independent heuristic functions are the most popular method for solving classical planning tasks. Heuristics estimate how far a state is from the goal, indicating which state should be evaluated next. An approach taken by some heuristics is to use abstractions: a mapping of the concrete states into abstract states that results in the so-called abstract transition system, which is typically smaller than the original transition system. The cost of a plan in the abstract transition system is an estimate for the concrete plan cost.

This work introduces a new non-admissible heuristic for satisficing planning (where the goal is to find any plan, not necessarily the cheaper one). The main idea is: given an ordered list of abstract transition systems, compute the cheapest path from the source state to a goal state for each graph, considering the actions that compose previous paths as credit for the following ones. More precisely, an action can be used in a new path for free (with cost zero) up to the number of times it appears on the already determined path that uses it the most. In the end, the heuristic value is the sum of the cost of all paths found.

We evaluate the proposed heuristic with the Fast Downward planning system. We compare our heuristic with FF and Post-Hoc Optimization heuristics on the IPC11's benchmark suite. The results show that the new heuristic increases the coverage by 12% compared to the FF heuristic, while expanding fewer states on most domains.

Keywords: Artificial Intelligence, Heuristic Search, Classical Planning, Abstraction-based Heuristic Functions, Non-admissible Heuristic Functions.

Uma Heurística Não Admissível Baseada em Planos Abstratos Sincronizados

RESUMO

Planejamento clássico é um tradicional problema de Inteligência Artificial que consiste em encontrar uma sequência de ações, denominada de plano, para atingir um desejado objetivo dado um estado inicial. Dizemos que o custo do plano é a soma dos custos de realizar cada ação que compõem o plano. Uma tarefa de planejamento clássico é uma descrição compacta de um problema de planejamento. Ela induz um sistema de transição: um grafo onde os vértices representam os possíveis estados e as arestas representam as ações que transformam os estados. Para resolver o problema de planejamento, é preciso determinar um caminho do estado inicial para um estado objetivo no sistema de transição. Algoritmos de busca combinados com funções heurísticas independentes de domínio são o método mais popular para resolução de tarefas de planejamento clássico. Heurísticas estimam quão longe um estado está do objetivo, indicando qual estado deve ser avaliado a seguir. Uma abordagem adotada por algumas heurísticas é utilizar uma abstração: um mapeamento dos estados concretos em estados abstratos que resulta no chamado sistema de transição abstrato, que é tipicamente menor que o sistema de transição original. O custo de um plano no sistema de transição abstrato é uma estimativa do custo do plano concreto.

Este trabalho introduz uma nova heurística não admissível para "satisficing planning" (onde o objetivo é encontrar qualquer plano, não necessariamente o mais barato). A ideia principal é: dado uma lista ordenada de sistemas de transição abstratos, calcular o caminho mais barato do estado inicial até um estado objetivo para cada grafo, considerando as ações que compõem os caminhos anteriores como crédito para os seguintes. Mais precisamente, uma ação pode ser utilizada em um novo caminho gratuitamente (com custo zero) até a quantidade de vezes que aparece no caminho já encontrado que mais a utiliza. No final, o valor da heurística é a soma do custo de todos os caminhos encontrados.

Nós avaliamos a heurística proposta com o sistema de planejamento Fast Downward. Comparamos nossa heurística com as heurísticas FF e Post-Hoc Optimization no conjunto de benchmarks da IPC11. Os resultados mostram que a nova heurística aumenta a cobertura em 12% em comparação com a heurística FF, enquanto expande menos estados na maioria dos domínios.

Palavras-chave: Inteligência Artificial, Busca Heurística, Planejamento Clássico, Fun-

ções Heurísticas Baseadas em Abstração, Funções Heurísticas Não Admissíveis..

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

IP	Integer Program
IPC11	International Planning Competition 2011
LP	Linear Program
PDB	Pattern Database
PhO	Post-Hoc Optimization

CONTENTS

1 INTRODUCTION

Classical planning (Hoffmann, 2011) is an area of Artificial Intelligence focused on determining a sequence of actions, or a plan, that transforms an initial state into a goal state. It can be applied, for example, in the problem of solving the Rubik's Cube puzzle or deciding routes that each truck in a fleet can take to transport packages between different places. In the first scenario, the initial state can be any shuffled arrangement of the cube, the goal is to make every face contain only one color and the actions are all possible rotations over the cube. In turn, the trucks' logistics task starts with each vehicle and package in a specific location and the goal is to have all packages delivered to their destinations through the actions of loading and unloading the cargo and moving the truck. Each action has a cost associated with it. The cost of a plan is the sum of the costs of all actions in it. Finding a plan with the lowest cost possible is an optimization problem, while just finding any valid plan is a satisficing problem.

A classical planning task induces a transition system. A transition system is a directed weighted labeled graph whose vertices represent the possible world's states and the edges represent the actions (or operators). In this context, finding a plan is the same as determining a path in the graph from the vertex associated with the initial state to any vertex associated with the goal states. However, in most problems, the transition system of a task is considerably large, even for simple scenarios. Therefore, constructing the entire transition system of a task to solve it is intractable, and using brute-force approaches to find a plan is unfeasible in terms of time and memory.

The most efficient and common approach for solving planning tasks is heuristic search. A heuristic (Bonet & Geffner, 2001; Pommerening, Röger, & Helmert, 2013) is a function that maps a state s to the estimated distance between s and the closest goal state. The general approach taken by heuristics is to simplify the original task, solve (ideally optimally) this simplified task and use the solution cost as the estimate. Search algorithms use heuristics to guide the exploration through the state space. Given a state s, the search algorithm applies the operators whose preconditions are satisfied by s and identifies all states that can be reached from s. Then, the heuristic function computes the heuristic value of every generated state, indicating to the search algorithm which state is more promising to achieve the goal. The more accurate or informed a heuristic is, the more efficient the planning will be. However, good heuristics usually take considerable time to be computed, which makes the process of obtaining a heuristic that is both informed and fast to compute a challenge.

A particular class of heuristic functions is the abstraction heuristics (Edelkamp, 2001). An abstraction of a transition system is a mapping that merges several states into one. The resulting state space (typically smaller than the original) is called an abstract transition system. The heuristic value of an abstraction heuristic is usually the cost of an optimal plan in the abstract transition system.

Abstraction heuristics are commonly used for optimal planning. They apply several relaxations to the problem they want to solve. This makes such heuristics less informed, but ensures admissibility, which is essential for optimal planning - search algorithms like A* (Hart, Nilsson, & Raphael, 1968) find optimal solutions when using admissible heuristics. Meanwhile, abstraction heuristics have not been used for satisficing planning, where heuristics based on delete relaxation are commonly used due to their speed and efficiency. In this work, we want to bridge the gap between abstraction heuristics and satisficing planning. Since our aim is to find solutions fast, we renounce admissibility to have a more informed heuristic.

Hence, we propose a new non-admissible heuristic for satisficing planning. Given an ordered list of abstract transition systems, our heuristic finds an abstract plan for each of them, considering the usage of an operator in one abstract transition system as credit (cost zero) for the next ones. For instance, if an operator is used twice in an abstract plan found, it can be used up to two times in the next abstract plans without incurring its cost to the plan. To compute the heuristic value, we simply sum the costs of all abstract plans found. Since abstract transition systems are smaller than the original transition systems, they are faster to solve, but do not carry all the information of the task. This approach can produce an estimate in a reasonable time while combining the information covered by each abstract transition system and minimizing the over-counting of redundant operators.

To evaluate the heuristic proposed, we compare it with the FF heuristic and Post-Hoc Optimization heuristic on benchmark tasks from the International Planning Competition 2011 (IPC11). We run the experiments with the Greedy Best-First Search (Doran & Michie, 1966) algorithm in the Fast Downward planner (Helmert, 2006).

The results show that our heuristic can solve more tasks than FF and Post-Hoc Optimization heuristics. Furthermore, it expands fewer states than the other heuristics for most domains, while the costs of the solutions found are very close between them. Nevertheless, the heuristic proposed usually makes fewer expansions per second than the other two heuristics.

This work is organized as follows. Chapter 2 presents some background information, like the formal definition of tasks, transition systems and heuristic functions referenced along the text. Chapter 3 comprises the proposed heuristic explanation, an example of its behavior and the algorithm to compute it. Chapter 4 contains the outcomes of the experiments. It compares the metrics of coverage, expansions, expansions per second and plan cost obtained by our heuristic and some baseline heuristics. In Chapter 5 we discuss some ideas that we tested aiming to improve the heuristic performance, which results were not as expected. Finally, Chapter 6 closes with the conclusion and future works.

2 BACKGROUND

2.1 Classical Planning

We define a planning task as a tuple $\Pi = \langle V, I, O, \gamma, c\rangle$. V is a set of state variables v, each with a finite domain $\mathrm{dom}(v)$. Given a subset of variables $V' \subseteq V$, a partial state is an assignment of each variable $v \in V'$ to a value in its respective domain. A state is a complete variable assignment over V. I is the task's initial state and γ is a partial state that describes the task's goal. O is a set of operators $o = \langle \mathrm{pre}, \mathrm{eff}\rangle$, where pre and eff are partial states. pre is the precondition for applying the operator and eff is the operator effect. An operator o is applicable in a state s if all variables in pre are in s with the same respective value. Applying o in s generates the state s', which is the result of updating s with the values in eff. s is called the predecessor of s' and s' is called the successor of s. The function $c : O \to \mathbb{R}_0^+$ gives a cost to each operator. A plan is a sequence of operators that leads I to a goal state (where γ is satisfied) and the plan cost is the sum of the cost of every operator in the sequence. A plan is an optimal plan when there is no other plan with a lower cost.

For example, consider the following logistics task where one truck can load and unload one package and move between three locations:

$$
\begin{aligned}
V &= \{\text{truck-at, package-at}\} \\
dom(\text{truck-at}) &= \{\text{A, B, C}\} \\
dom(\text{package-at}) &= \{\text{A, B, C, truck}\} \\
I &= \{\text{truck-at = A, package-at = B}\} \\
\gamma &= \{\text{package-at = C}\} \\
c(o) &= 1 \text{ for all } o \in O \\
O &= \{\textit{move}(src, tgt), \textit{load}(loc), \textit{unload}(loc) \\
&\qquad \mid src, tgt, loc \in \{\text{A, B, C}\}, src \neq tgt\}
\end{aligned}
$$

where

$$
\begin{aligned}
move(src, tgt) &= \langle \text{truck-at} = src;\ \text{truck-at} = tgt\rangle \\
load(loc) &= \langle \text{truck-at} = loc, \text{package-at} = loc;\ \text{package-at=truck}\rangle \\
unload(loc) &= \langle \text{truck-at} = loc, \text{package-at} = \text{truck};\ \text{package-at} = loc\rangle
\end{aligned}
$$

The variables are the truck location and the package location. The truck can be at location A, B or C and the package can be in one of these locations or inside the truck. In the initial state, the truck is at A and the package is at B. The goal is to have the package at C. The operator $move$ updates the truck position to tgt since it is at src. $load$ has the precondition of having the truck and the package at the same location and the effect of changing the package location variable value to truck, while $unload$ assigns the current truck location to the variable package-at if the truck was carrying the package. All operators have a unitary cost.

A task Π induces a transition system, also called state space. A transition system $\mathcal{T} = \langle S, T, s_0, S^* \rangle$ is a weighted, labeled and directed graph where S is a set of states, $s_0 \in S$ is the initial state, $S^* \subseteq S$ is the set of goal states and $T \subseteq S \times S$ is a set of transitions. For every operator $o \in O$ that is applicable in a state s and results in a state s', there's a transition $t \in T$ from s to s' with weight equal to $c(o)$ and label "o". The states in S and the transitions in T are, respectively, the vertices and edges of the graph.

2.2 Heuristics

A heuristic is a function $h : S \to \mathbb{R}^+_0 \cup \{\infty\}$ that estimates the cost of an optimal plan from a given state $s \in S$ to a goal. We define $h(s)$ as the heuristic value for the state s, and $h(s) = \infty$ means that there is no possible way to reach a goal from s. h^* is the representation for a perfect heuristic, which maps any state to the optimal solution cost for it. We say that a heuristic is admissible when $h(s) \leq h^*(s)$ for all states $s \in S$.

Search algorithms use heuristics to guide the exploration of the state space in the process of finding a plan. A search algorithm usually expands a state, identifying its successor states, and has to decide which state to expand next. The heuristic value of each state indicates how close it is to the goal. Thus, the algorithm can select the most promising state already reached to continue the search.

2.2.1 FF Heuristic

Consider a planning task Π' where all variables in V are facts, i.e. $dom(v)$ = {true, false} for all $v \in V$, and the goal is a logical formula composed only by non-negated propositions. In this context, we say that a variable is added or deleted when an operator's effect changes its value to true or false, respectively. Any task Π can be transformed into a task like Π' in polynomial time and its benefit is that, since all variables in the goal have value true, delete effects can be considered as bad effects.

Given a task with the characteristics of Π', it's possible to derive a delete-relaxed planning task Π^+, which is identical to the original task but all assignments of variables to false (deletions) are removed from the operators' effects. We call a plan in Π^+ a relaxed plan and it can be used as a heuristic for Π', where the heuristic value of a state s is the cost of a relaxed plan starting from s. An easy way to determine a relaxed plan is the greedy strategy: identify the applicable operators in the initial state, select one of them randomly and apply it, repeating these steps until all variables in the goal are set to true. However, although simple and easy to compute, this approach can be very inaccurate. To get a more informed heuristic, the function could calculate the cost of the optimal relaxed plan. We define the h^+ heuristic as the perfect heuristic $h^*(s)$ of a state s in the delete-relaxed task Π^+. Unlike the greedy method, finding a relaxed optimal plan is NP-hard. Therefore, h^+ cannot be computed efficiently.

Since the two approaches presented so far contain limitations, Bonet and Geffner (2001) introduce the h^{add} heuristic, based on estimating plans for each variable in the goal independently and summing them up. In other words, it computes the minimal cost to make each proposition in the goal of the relaxed task true, with the calculation of one subgoal not influencing the other. Formally, the heuristic function is defined as

$$h^{add}(s) = \sum_{v \in G} f(v, s)$$

where G is the set of variables mentioned in the goal and

$$f(v,s) = \begin{cases} 0 & \text{if } v = true \text{ in s} \\ \underset{o \in A(v)}{argmin}\ cost(o) + \sum_{p \in Pre(o)} f(p,s) & \text{otherwise} \end{cases}$$

where $A(v)$ is the set of operators that adds v and $Pre(o)$ is the set of variables in the precondition of operator o.

h^{add} is more accurate than the greedy heuristic and can be computed more efficiently than h^+, but it still has a downside: if an operator is used to reach more than one subgoal, its cost will be counted multiple times, making the heuristic overestimates the real plan cost. To deal with this, Hoffmann and Nebel (2001) proposes the relaxed plan heuristic, denoted h^{FF}. It finds plans to achieve each subgoal separately like h^{add}, but keeps track of the operators occurring on each plan to ensure that their cost will be considered only once. This results in a non-admissible heuristic that can be computed in polynomial time.

2.2.2 Abstractions

An abstraction is a relaxation of the planning task that joins different states, generating a new and (typically) smaller transition system. Given the original transition system T, the abstraction is a function that maps the states s of T into abstract states $\alpha(s)$, resulting in the abstract transition system T_α. If s is a goal state in T, then $\alpha(s)$ is also a goal state in T_α. A transition from s_1 to s_2 in T is replicated as a transition from $\alpha(s_1)$ to $\alpha(s_2)$ in T_α. We call a plan in T_α an abstract plan. A special case of abstraction is the projection, where only a subset of the state variables, called pattern, is kept while the others are ignored. All original states that have identical values for the variables in the pattern became the same abstract state.

With patterns small enough, plans in the abstract transition system can be found very quickly and the cost of such a plan can be used as a heuristic for the original task. The pattern database (PDB) heuristic (Edelkamp, 2001), written h^P, returns the cost of the cheapest path to the goal in the abstract transition system resulting from projecting the task with the pattern P.

For example, consider the following planning task (from Pommerening et al., 2013):

$$\begin{aligned}
V &= \{\text{A, B, C}\} \\
dom(v) &= \{0, 1, 2, 3, 4\} \text{ for all } v \in V \\
I &= \{\text{A=0, B=0, C=0}\} \\
\gamma &= \{\text{A=3, B=3, C=3}\} \\
c(o) &= 1 \text{ for all } o \in O \\
O &= \{inc^v_x,\ jump^v \mid v \in V,\ x \in \{1, 2, 3, 4\}\} \\
&\text{where} \\
inc^v_x &= \langle v = x;\ v = x + 1\rangle \\
jump^v &= \langle v' = 4 \text{ for all } v' \neq v;\ v = 3\rangle
\end{aligned}$$

It contains three variables that can assume numbers between zero and four. In the initial state, all variables are assigned to the value 0, and in the goal state, all variables are assigned to the value 3. The operator inc increments the value of a variable by one and the $jump$ operator assigns the number 3 to a variable under the precondition that all other variables have the value 4. Both operators have a unitary cost, which means that the goal can be optimally achieved with a cost of nine, by using the inc operator three times for each variable.

For this example task, any pattern with just one variable results in the PDB heuristic estimating the cost of one for the initial state, since all other variables are ignored and the jump operator can be applied with no restrictions. If using the patterns with two variables, the function gives an estimate of six, considering the actions of incrementing the variables.

2.2.2.1 Systematic Pattern Generation

A simple method to obtain patterns is the Systematic Pattern Generation approach. It generates all possible combinations containing up to a determined number of variables. We define $SysX$ as the set of patterns with size between one and X (inclusive). Given the example task from the previous section, the systematic sets are:

$$Sys1 = \{\{\text{A}\}, \{\text{B}\}, \{\text{C}\}\}$$
$$Sys2 = Sys1 \cup \{\{\text{A, B}\}, \{\text{A, C}\}, \{\text{B, C}\}\}$$
$$Sys3 = Sys2 \cup \{\{\text{A, B, C}\}\}$$

2.2.3 Post-Hoc Optimization Heuristic

PDB heuristics take into account only some aspects of the problem (variables in the pattern) while completely ignoring others. This inspired the study of approaches that combine the information covered by different patterns with the aim of getting a more informed function. Consider the example task shown in the previous section and a pattern collection composed of the three possible patterns with two variables. Given that $h^{A,B}$ = 6 for the initial state and that all operators have unitary cost, it's possible to conclude that any solution for this task must contain at least six actions that modify the variable A or the variable B. The same is valid for $h^{A,C}$ and $h^{B,C}$ and the variables A, C and B, C, respectively. From the combination of these three constraints, we can induce that at least nine operators are required in any plan.

This strategy can be generalized with an integer program (IP) with one variable X_o for each operator $o \in O$, representing the total cost of the usages of the operator o. In other words, X_o is equal to the operator's cost times the number of times it appears in the estimated plan.

The IP's objective is to minimize

$$\sum_{o \in O} X_o$$

subject to the constraints

$$\sum_{o \in O_P} X_o \geq h^P(s) \text{ for all patterns } P \text{ in the patter collection}$$
$$X_o \geq 0 \text{ for all } o \in O$$

where O_P is the set of operators that cause a state change in the abstract transition system induced by P. The objective value of the IP presented above is an admissible heuristic. However, using an IP solver is intractable. Therefore, an LP-relaxation is used instead: the Post-Hoc Optimization heuristic (h_C^{PhO}), introduced by Pommerening et al. (2013), is defined as the objective value of the LP presented above solved with the pattern collection C. h_C^{PhO} is also admissible.

2.2.4 Satisficing Post-Hoc Optimization with a Greedy Constructive Algorithm

Satisficing Post-Hoc Optimization with a Greedy Constructive Algorithm (h_C^{PhOG}) is a work in progress of the Master's student Daniel Matheus Doebber based on the Post-Hoc Optimization heuristic. Its main idea is to generate the group of constraints exactly as h_C^{PhO} does but, instead of running an LP solver, it uses a greedy strategy to determine the value of each variable. Such method is simple: for each constraint, go through the variables in sequence incrementing their value by one until the expression is satisfied. Once all constraints are satisfied, the heuristic value is the sum of all variables.

To illustrate the heuristic computation, consider the constraints derived from the two-size patterns of the example task presented in section 2.2.2:

$$inc_1^A + inc_2^A + inc_3^A + inc_4^A + jump^A + inc_1^B + inc_2^B + inc_3^B + inc_4^B + jump^B \geq 6$$
$$inc_1^A + inc_2^A + inc_3^A + inc_4^A + jump^A + inc_1^C + inc_2^C + inc_3^C + inc_4^C + jump^C \geq 6$$
$$inc_1^B + inc_2^B + inc_3^B + inc_4^B + jump^B + inc_1^C + inc_2^C + inc_3^C + inc_4^C + jump^C \geq 6$$

Starting with the first constraint, the six leading variables will be incremented once, satisfying the expression. For the second constraint, all variables related to "A" already have value one, making it necessary only one more operator to achieve the requirement. Thus, inc_1^A is increased again. Next, the third constraint still requires five more operators to fulfill the expression, since only inc_1^B is not zero. Therefore, the first five variables are incremented. This results in $inc_1^A = inc_1^B = 2$ and $inc_2^A = inc_3^A = inc_4^A = jump^A = inc_2^B = inc_3^B = inc_4^B = jump^B = 1$, what makes the estimate for this task to be 12. As this prediction is higher than the optimal plan cost (nine), we can conclude that h_C^{PhOG} is not admissible. Nevertheless, being simple and fast to compute, this heuristic has demonstrated the potential to solve satisficing planning tasks very quickly.

3 SATISFICING GREEDY CONSTRUCTIVE HEURISTIC BASED ON ABSTRACT TRANSITION SYSTEMS

To compute the Post-Hoc Optimization heuristic with an LP solver can consume a considerable time. Besides the fact that finding an optimal solution for the LP is not trivial, the heuristic also has the overhead of communicating with the solver, which is a separate application. Therefore, the h_C^{PhOG} heuristic (presented in section 2.2.4) proposes to satisfy the constraints in a more efficient (but inadmissible) way, using a greedy strategy. Nevertheless, the LP used by h_C^{PhOG} is a relaxation of the real problem: an assignment that satisfies the constraints for a given abstract transition system might not be an abstract plan for it. In other words, the operators selected by h_C^{PhOG} just solve the LP problem numerically. Thus, we do not know whether these operators are important to reach the goal or not. The consequence of selecting operators that are not important is overestimating the plan cost, losing information. In contrast, selecting important operators reduces the heuristic value (making it closer to the optimal plan cost), since such operators probably are part of abstract plans for several abstract transition systems.

For example, consider the abstract transition systems in Figure 3.1. The constraints derived from them for the initial state are:

$$\begin{aligned} o_1 + o_2 + o_3 &\geq 2 \\ o_2 + o_3 &\geq 2 \end{aligned}$$

To satisfy the first constraint, h_C^{PhOG} assigns the value one to o_1 and o_2. To satisfy the second constraint, h_C^{PhOG} increments o_2 again, which now is assigned to the value two. Therefore, the heuristic value for this example is three. Note that the operators selected are not a plan for both abstract transition systems. Choosing the operator o_1 does not help to achieve the goal, but in h_C^{PhOG} such information was relaxed and ignored. Therefore, the heuristic overestimates the optimal abstract plan cost (two).

In this work, we consider this information ignored by h_C^{PhOG}. We achieve this by computing synchronized plans over the set of abstract transition systems. We find an optimal abstract plan for the first abstract transition system. Hence, we know that the operators in this plan are (probably) important to solve the task. Then, we consider this information to find an abstract plan in the next abstract transition system: transitions related to the operators already used do not incur a cost to the abstract plan up to the times

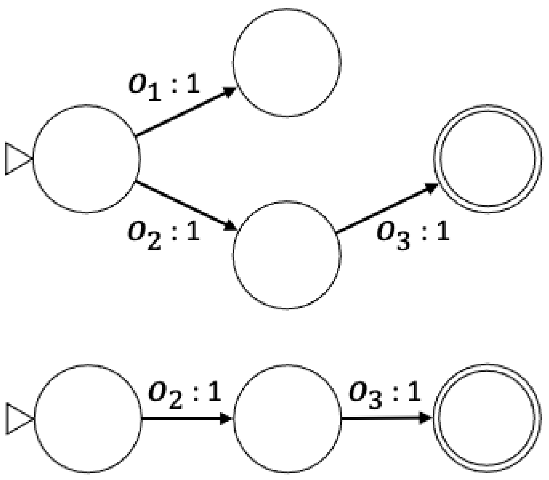

Figure 3.1 – Example transition systems. Triangles indicate the initial states, double circles are goal states and edge labels are the operators' names followed by the edge cost. Edges that are self-loops are not represented for simplicity.

they appear in previous plans. This is done iteratively for all abstract transition systems in the set. Ideally, we find a solution that minimizes the over-counting of operators and is closer to the task's optimal solution.

The main idea of our heuristic is described by the following steps, given an ordered list of abstract transition systems L as input and a state s:

- compute an optimal abstract plan for s in the first abstract transition system of L;
- collect the operators in this plan and save them, generating what we call the *credit*;
- compute an abstract plan for s in the next abstract transition system, but now transitions related to operators present in the credit can be used for free up to the times they appear in the credit;
- collect the operators from the new abstract plan that were not used for free and insert them in the credit;
- repeat the last two steps for all remaining abstract transition systems in L.

The heuristic value $h_L^{GP}(s)$ is the sum of the costs of all operators in the final credit.

For example, consider the graphs in Figure 3.2 as a list of abstract transition systems of a task, where the cost of each operator (o_1 to o_5) is one, and that the evaluation order is from the top to the bottom. The only possible abstract plan in the first abstract transition system has cost two and is composed of o_1 and o_2. Therefore, these two operators are saved as credit. For the second abstract transition system, the cheapest way to achieve the goal is through operators o_1 and o_4. Even though the alternative plan (o_3 and o_4) contains the same amount of transitions, operator o_1 can be used for free because it

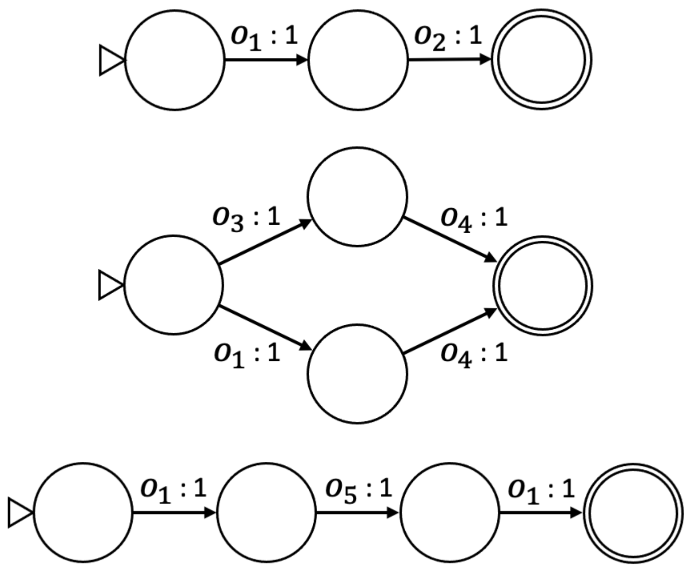

Figure 3.2 – Example transition systems. Triangles indicate the initial states, double circles are goal states and edge labels are the operators' names followed by the edge cost. Edges that are self-loops are not represented for simplicity.

appears in the preceding abstract plan, resulting in an abstract plan with cost one. Since o_4 was not used for free, it is added to the credit. The last abstract transition system also has a single plan, which costs two since one of the applications of the operator o_1 comes as credit from the previous usage. The third transition is still counted because none of the other abstract plans found had more than one occurrence of o_1 in the same plan. Operators o_1 and o_5 are included in the credit, resulting in the final credit that is formed by o_1, o_2, o_4, o_5 and o_1. Therefore, the heuristic gives the value five as an estimate for the concrete task plan.

Algorithm 1 shows the heuristic pseudocode. We use a function based on the Dijkstra algorithm (Dijkstra, 1959) to determine the cheapest path in a graph (lines 1-26). The main function *compute_heuristic* (lines 28-43) calls the *dijkstra* function for each abstract transition system in the *graphs* list (lines 30-31). The algorithm represents the credit as a vector (line 29) that stores at each index the usage count of the operator with the respective id. *dijkstra* function initializes a vector to count the operators used in the paths and insert it with the initial state in the priority queue Q with the value zero (lines 3-5). Lines 6-25 perform the search: line 6 removes the entry with the lower value from Q; lines 8-10 check if the entry corresponds to a goal state and, if so, return the operators present in the path found; lines 11-24 expand the state in case it was not expanded yet. For each outgoing transition of the state, the algorithm increments the usage count of the operator related to the transition in *new_ops_count* and checks whether the operator (still) can be used with no cost (lines 15-17). If that is the case, line 18 inserts the successor state and *new_ops_count* into Q with the same value as the predecessor state. Otherwise, line 20 increases the value by the cost of the operator and inserts the successor state and *new_ops_count* into Q. Once the *dijkstra* function returns the operators that compose the path found, *compute_heuristic* updates the current credit (line 36). After the algorithm determines a plan for every abstract transition system, it computes the heuristic value $h_L^{GP}(s)$ by summing the product of the operators' count in the final credit times their cost (lines 39-42).

The code developed for this work follows the pseudocode presented above. Nevertheless, it is worth mentioning some implementation details, which mainly aim at making the computation faster. The data structure used to represent the graph is an adjacency list, once it enables better performance to retrieve the outgoing transitions of a specific state to find its successors. In addition, the vector that stores the operators' count in the Dijkstra function only has the capacity for the abstract operators that are not self-loops. Since the algorithm inserts this vector into the priority queue, new memory is constantly allocated for it, which demands considerable time. Thus, having a smaller vector significantly reduces time consumption. Also, to avoid memory request operations, we declare the vector *expanded* only once, with a length equal to the number of states of the abstract transition system with most states. Then, when the Dijkstra function is called, we just (re)initialize the structure until the position required by the current abstract transition system, without being necessary to allocate new space.

It is important to note that, from the second abstract transition system on, our algorithm does not ensure that we will find an optimal abstract transition system, because we do not consider the credit when marking a state as expanded. For instance, suppose that we have only two entries in the priority queue, both related to the same state s and with the same value (let's say, 15, which is optimal until here), but one of them (e_1) still can use the operator o for free while the second (e_2) cannot. In addition, the only outgoing transition of s is relative to the operator o, and the successor state of s is a goal state. In case e_1 is removed from the queue, s is marked as expanded and operator o will not incur its cost to the plan. Therefore, we achieve the goal and the plan cost is 15. However, if e_2 is removed from the queue, the cost of o is added to the plan cost, which is not optimal anymore.

Algorithm 1: h^{GP} computation

```
Function dijkstra(graph, initial_state, credit)
    $expanded[v] \leftarrow False$    $\forall$ vertex $v \in graph.vertices$
    $ops\_count[o] \leftarrow 0$    $\forall$ operator $o \in graph.operators$

    $Q.push(initial\_state, 0, ops\_count)$
    while not Q.empty() do
        $entry \leftarrow Q.pop()$
        if entry.state is goal then
            return $entry.ops\_count$
        end
        if not expanded[entry.state] then
            $expanded[entry.state] \leftarrow True$
            foreach edge e from entry.state to t do
                if not expanded[t] then
                    $new\_ops\_count \leftarrow entry.ops\_count$
                    $new\_ops\_count[e.operator] ++$
                    if entry.ops_count[e.operator] $\leq$ credit[e.operator] then
                        $Q.push(t, entry.cost, new\_ops\_count)$
                    else
                        $Q.push(t, entry.cost + e.cost, new\_ops\_count)$
                    end
                end
            end
        end
    end
    return $Null$

Function compute_heuristic(graphs, state)
    $credit \leftarrow 0$    $\forall$ operator $o$
    foreach graph g in graphs do
        $graph\_credit \leftarrow dijkstra(g, state, credit)$
        if graph_ops_count is Null then
            return $INF$
        end
        foreach operator o in credit do
            $credit[o] \leftarrow max(credit[o], graph\_ops\_count[o])$
        end
    end
    $heuristic\_value \leftarrow 0$
    foreach operator o do
        $heuristic\_value \leftarrow heuristic\_value + credit[o] * cost(o)$
    end
    return $heuristic\_value$
```

3.1 Satisficing Heuristic Function based on Synchronized Plan Costs on Abstract Transition Systems Is Not Admissible

Consider the logistics task from section 2.1. For this example, we make one modification in the task, including in the goal the condition that the truck must be at location C:

$$\gamma = \{ \text{truck-at = C, package-at = C} \}$$

The optimal plan for this task is $move$(A,B), $load$(B), $move$(B,C) and $unload$(C), and its cost is four. Now, let's analyze h^{GP}computation for the initial state of this task with an input list composed of the abstract transition systems induced by $Sys2$ patterns ($Sys1$ patterns lead the list). Figure 3.3 illustrates the two abstract transition systems induced by patterns with only one variable, with the optimal abstract plans highlighted in red. The optimal abstract plan for the first abstract transition system, which considers only the variable truck-at, is just $move$(A,C). Meanwhile, the optimal abstract plan for the second abstract transition system, which considers only the variable package-at, is $load$(B) and $unload$(C). The last abstract transition system in the input list considers both variables, modeling the complete task. Given the operators in the credit ($move$(A,C), $load$(B) and $unload$(C)), there are two optimal abstract plans, both with cost two:

$$move\text{(A,B)}, load\text{(B)}, move\text{(B,C)}, unload\text{(B)}$$
$$move\text{(A,C)}, move\text{(C,B)}, load\text{(B)}, move\text{(B,C)}, unload\text{(B)}$$

Therefore, $h^{GP}_{Sys2}(s_0) = 5$. Since $h^{GP}_{Sys2}(s_0) > h^{*}(s_0)$, we conclude that h^{GP} is not admissible.

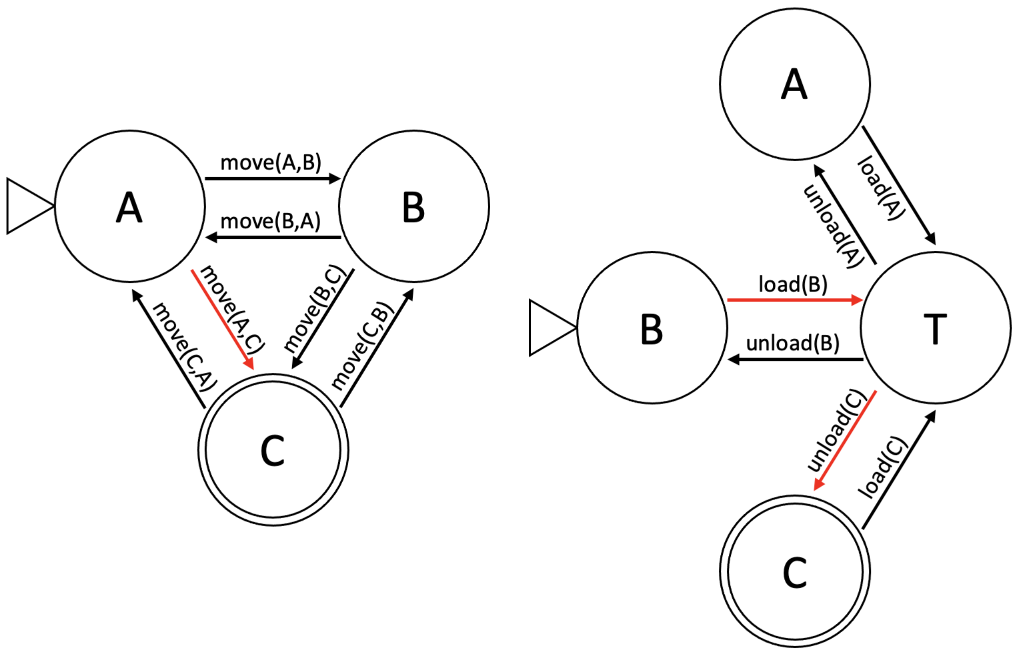

Figure 3.3 – Abstract transition systems of the logistics task induced by patterns {truck-at} (left) and {package-at} (right). Triangles indicate the initial states, double circles are goal states, identifiers in the vertices are the variable values on each state and edge labels are the operators' names. The optimal abstract plan is represented in red. Edges that are self-loops are not represented for simplicity.

4 EXPERIMENTS

To evaluate the proposed heuristic, we implemented it in Fast Downward (Helmert, 2006), a domain-independent classical planning system with different search algorithms and popular heuristic functions available. In addition, we did the experiments with Downward Lab (Seipp, Pommerening, Sievers, & Helmert, 2017), a tool that runs Fast Downward collecting metrics and enables the generation of reports and charts. We performed all experiments on a machine with Linux OS, 32GB of RAM and 12 cores. Furthermore, we configured Downward Lab to solve at most five tasks in parallel, with a memory limit of 4GB and a timeout of 30 minutes for each process.

We used a benchmark set formed by the 280 tasks from 14 domains that composed the sequential track of the International Planning Competition 2011 (IPC11). For each task, we ran the experiments using abstract transition systems projected from $Sys1$ and $Sys2$ patterns and, for the second case, graphs induced by one-variable patterns are evaluated first by the heuristic. We compare these two variations with h^{FF}, h^{PhO}_{Sys2}, h^{PhO-IP}_{Sys2} (h^{PhO} treated as an integer problem) and h^{PhOG}_{Sys2}. We run all heuristics with the search algorithm Greedy Best-First Search (Doran & Michie, 1966). The following metrics are analyzed:

- Coverage: number of tasks solved within time and memory limits;
- Expansion: number of states expanded by the search algorithm until finding a goal state;
- Plan cost: cost of the plan found;
- Total Time: amount of time taken to run preprocessing and search steps;
- Expansions per Second: number of states evaluated per second (only search time is considered)

Both tested versions of the heuristic proposed had interesting results. Nevertheless, because h^{GP}_{Sys2} is more informed than h^{GP}_{Sys1} and, thus, has more potential to solve harder tasks, the results presented in this Chapter are focused on h^{GP}_{Sys2}.

Figure 4.1 shows the number of expansions done by h^{GP}_{Sys2} compared to the other heuristics for all benchmark domains. It's possible to see that h^{GP}_{Sys2} makes fewer expansions than h^{PhO}_{Sys2}, h^{PhO-IP}_{Sys2} and h^{GP}_{Sys1}. This can be explained by the fact that h^{GP}_{Sys2} (a) considers the operators sequence (paths in transition systems) and not only the operators costs like h^{PhO} heuristics and (b) has more and larger patterns than h^{GP}_{Sys1}. In general,

Coverage	h^{PhO}_{Sys2}	h^{PhO-IP}_{Sys2}	h^{FF}	h^{PhOG}_{Sys2}	h^{GP}_{Sys1}	h^{GP}_{Sys2}
barman (20)	0	0	6	**20**	0	13
elevators (20)	**0**	**0**	**0**	**0**	**0**	**0**
floortile (20)	0	0	**8**	0	0	0
nomystery (20)	7	5	10	8	**14**	10
openstacks (20)	**0**	**0**	**0**	**0**	**0**	**0**
parcprinter (20)	16	7	7	15	14	**18**
parking (20)	3	0	**20**	**20**	**20**	14
pegsol (20)	17	15	**20**	17	17	**20**
scanalyzer (20)	13	12	18	**20**	**20**	17
sokoban (20)	**18**	12	**18**	**18**	13	16
tidybot (20)	17	12	16	**19**	**19**	**19**
transport (20)	0	0	0	0	**12**	0
visitall (20)	11	9	3	13	**20**	5
woodworking (20)	**18**	3	14	**18**	5	**18**
Sum (280)	120	75	140	**168**	154	150

Table 4.1 – Domain-wise coverage for all evaluated methods. The number of instances in each domain is shown in parenthesis next to the domain name. The best results over all algorithms are highlighted in bold.

h^{GP}_{Sys2} also expands fewer states than h^{PhOG}_{Sys2}, except for the `VisitAll`, `Barman` and `Sokoban` domains. In contrast, the comparison between h^{GP}_{Sys2} and h^{FF} looks more balanced. Considering only the tasks solved by both algorithms, h^{GP}_{Sys2} is better in 55 tasks, while h^{FF} makes fewer expansions in 54 tasks. Nonetheless, the domain-wise results show a superiority of h^{GP}_{Sys2}, which expands fewer states in seven of them, while h^{FF} has lower values only in `PegSol`, `Sokoban` and `TidyBot`.

Figure 4.2 compares the expansions per second for tasks that were solved in more than one second by both algorithms compared. It demonstrates that h^{GP}_{Sys2} is more expensive than the other heuristics, but h^{PhO-IP}_{Sys2}. This is the consequence of performing the search in multiple transition systems, which is responsible for making the heuristic informed at the cost of consuming considerable time.

Table 4.1 presents the coverage results on the benchmark suite. h^{GP}_{Sys1} and h^{GP}_{Sys2} solved, respectively, 154 and 150 tasks. They are only behind h^{PhOG}_{Sys2}, which completed a total of 168 tasks. Furthermore, analyzing domain by domain, h^{GP}_{Sys1} is the function leading the coverage in six cases (three isolated). Moreover, it was the only algorithm to solve tasks from the `Transport` domain. Meanwhile, h^{GP}_{Sys2} leads the coverage in four domains. These results reveal that, despite being expensive, h^{GP} accuracy makes it possible to solve difficult tasks that are not solved by less informed (although faster) heuristics.

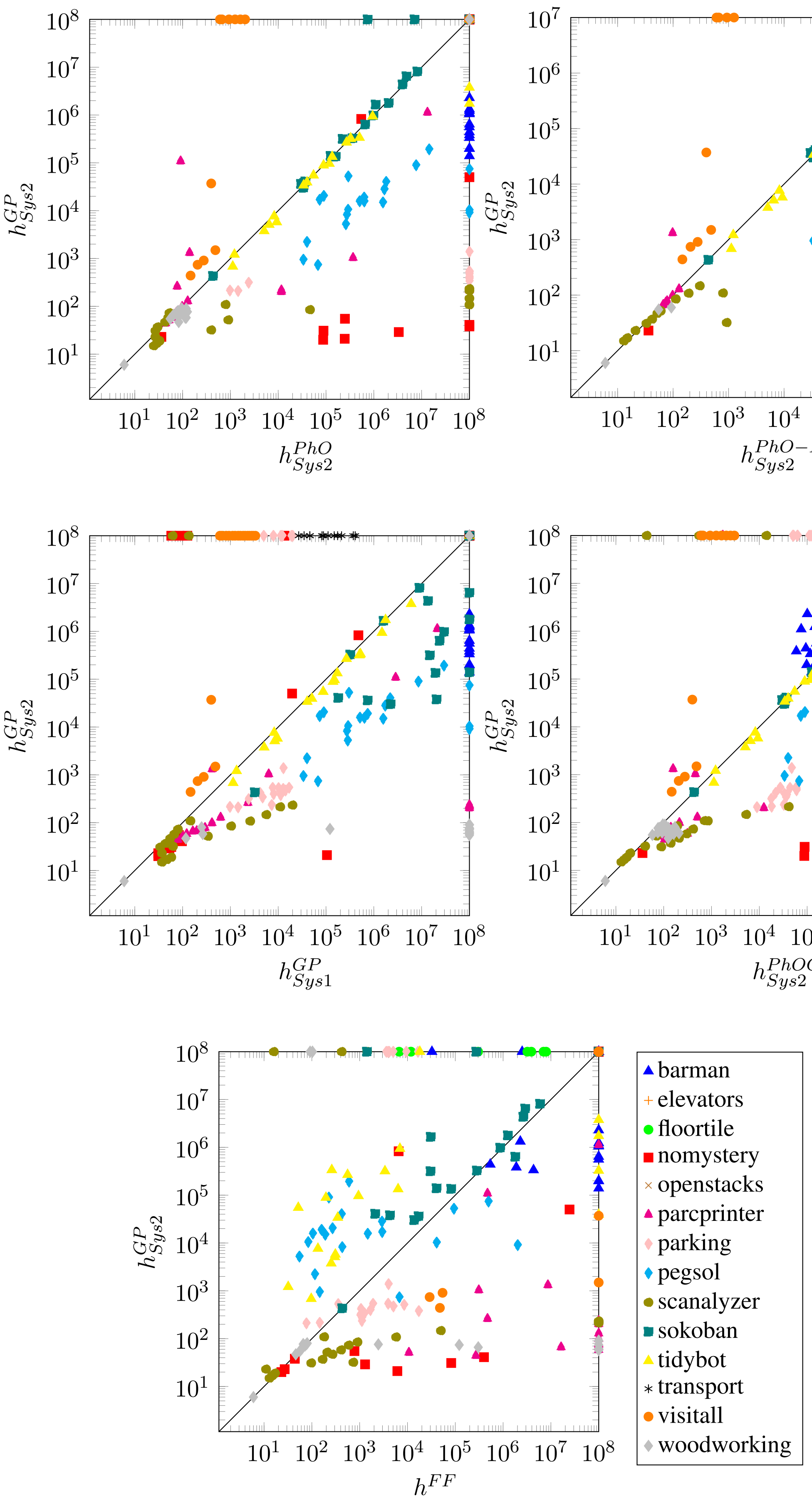

Figure 4.1 – Scatter plots of expansions: h^{GP}_{Sys2} (y-axis) vs. other heuristics. Unsolved tasks are plotted at 10^8.

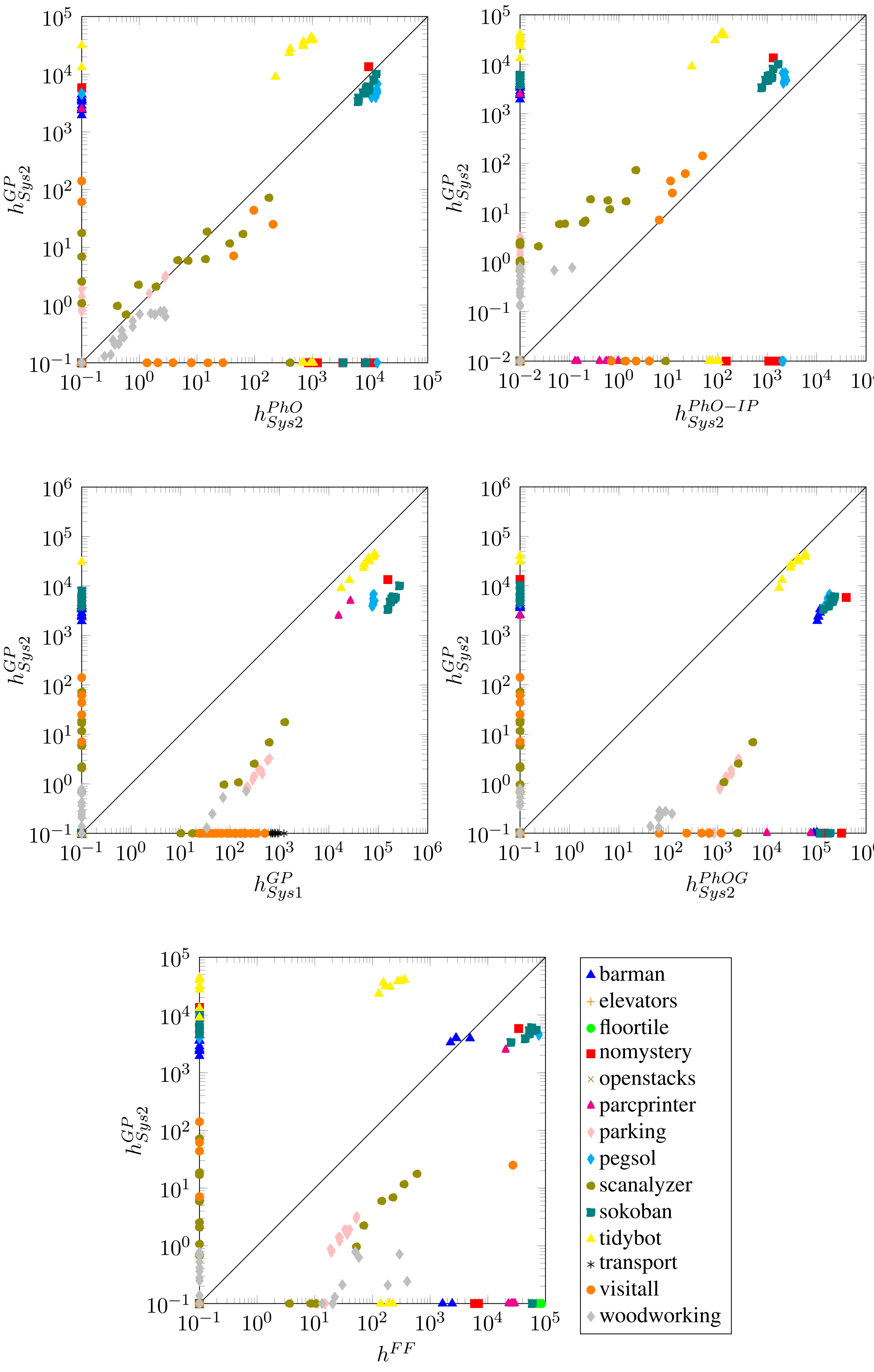

Figure 4.2 – Scatter plots of expansions per second: h^{GP}_{Sys2} (y-axis) vs. other heuristics. Unsolved tasks are plotted at the lower edge (10^{-1} or 10^{-2}).

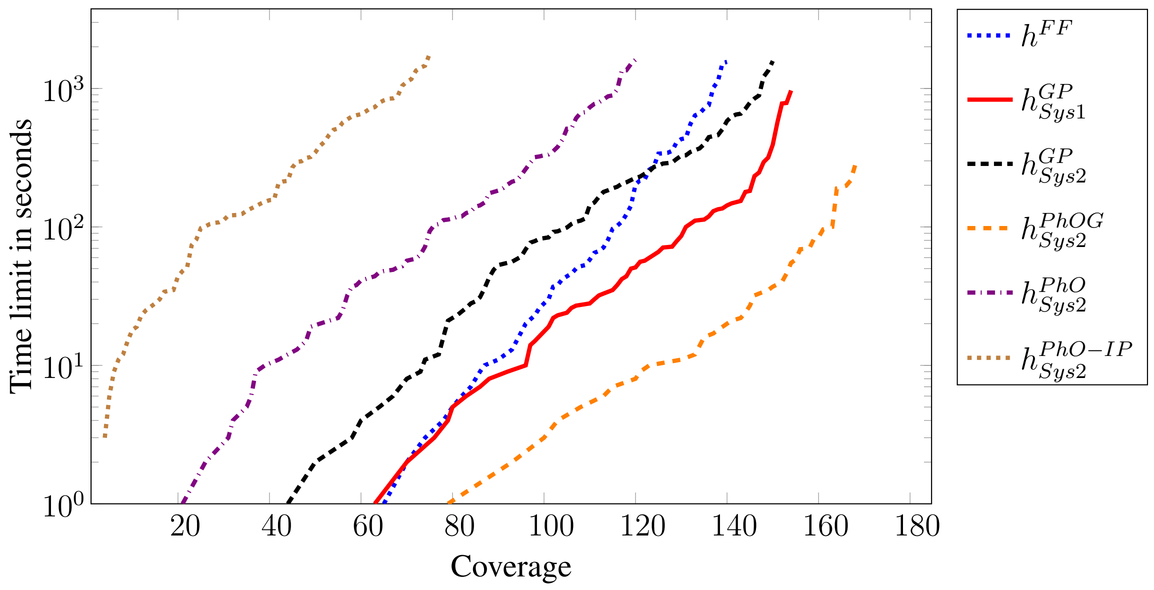

Figure 4.3 – Number of tasks solved (individually) in a determined amount of time

Figure 4.3 shows a cactus plot with the coverage of each heuristic until a determined computation time. It demonstrates that, when tasks need to be solved in a short period of time, h^{GP}_{Sys1} and h^{FF} are a better choice over h^{GP}_{Sys2}, as they are faster to solve easier tasks. However, with more time available, h^{GP}_{Sys2} performs better, as it overcomes h^{FF} coverage after 300 seconds and seems to also surpass h^{GP}_{Sys1} as the distance between the curves is decreasing as time increases.

The cost of the plan found by the planner with each heuristic is compared in Figure 4.4. It shows that the admissible heuristics h^{PhO}_{Sys2} and h^{PhO-IP}_{Sys2} were able to find cheaper plans more times than h^{GP}_{Sys2}. In contrast, h^{GP}_{Sys2} leads to better plans than h^{FF}. Considering only tasks solved by both heuristics, h^{GP}_{Sys2} solves 56 instances finding cheaper plans, while h^{FF} performs better in 24 tasks. Nevertheless, the costs of the plans are very close between all heuristics, making it not possible to state that one has a clear advantage over another.

In the next sections, we present some modifications in the heuristic computation (and their respective outcomes) that we tested during the development of this work.

4.1 Parallel Operators

While running the experiments, we observed that when more than one operator can be used to move between two abstract states (what we call *parallel operators* or *parallel transitions*) with the same cost, choosing one operator or another to compose the

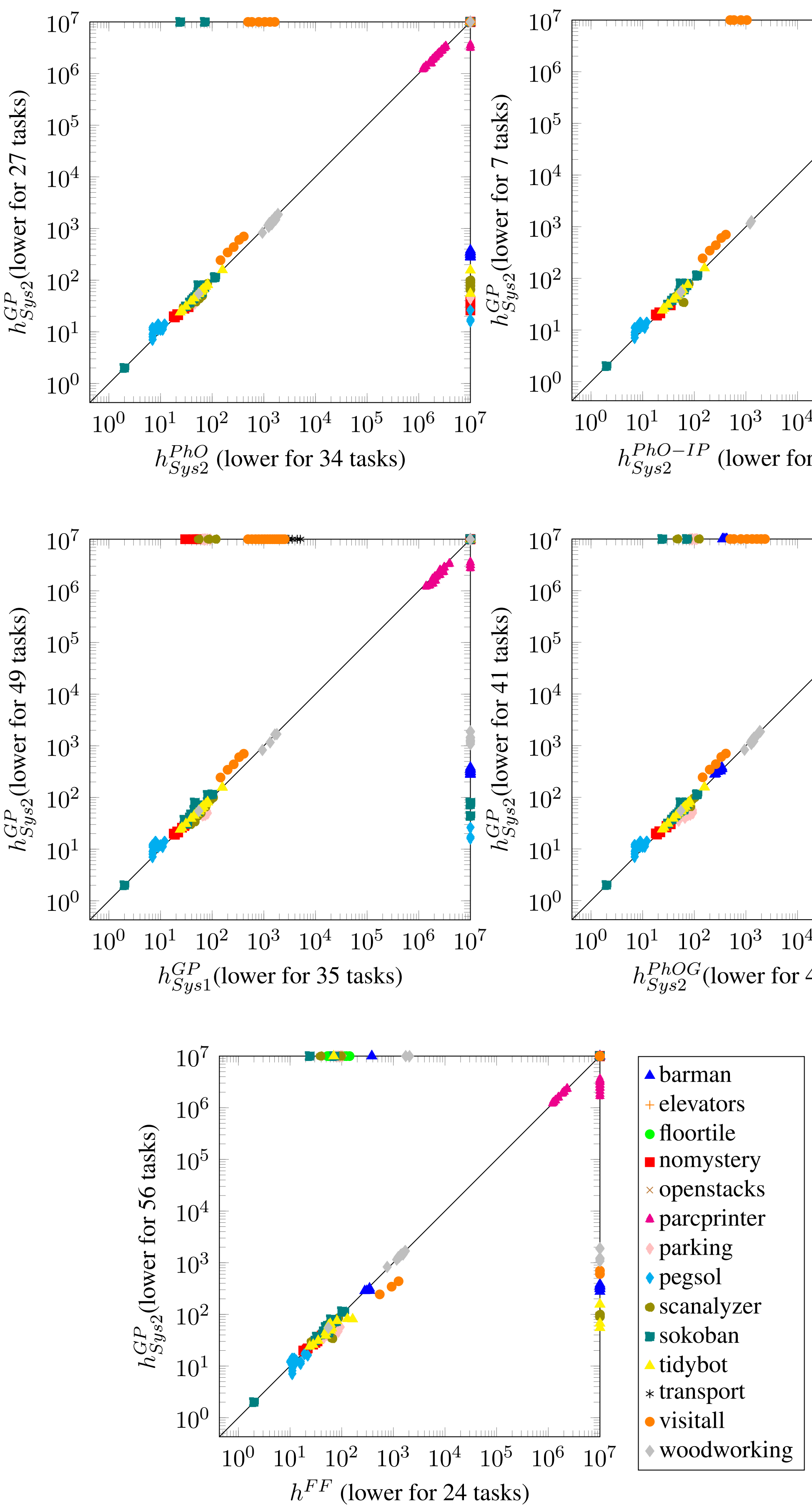

Figure 4.4 – Scatter plots of plan cost: h^{GP}_{Sys2} (y-axis) vs. other heuristics. Unsolved tasks are plotted at 10^8.

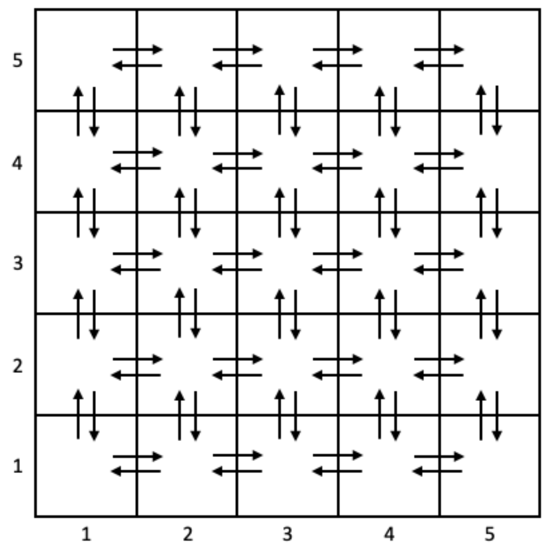

Figure 4.5 – Illustration of the 5x5 `VisitAll` task. Arrows represent all possible operators.

path can result in a significant difference in the heuristic value. One domain affected by this is `VisitAll`, which models tasks where an agent is in the middle of a square grid with size NxN and can move between adjacent cells with the goal of visiting all cells. The variables are the agent's current position and whether a cell was already visited or not (one variable for each cell). Also, there is one unitary-cost operator for every possible move. The precondition for applying the operators is that the agent is in the respective cell and the effect is to update its position and mark the target cell as visited.

As an example, consider the 5x5 `VisitAll` task represented in Figure 4.5, which illustrates the grid and all the operations that the agent can perform (arrows). Formally, we define the task's variables as follows (we refer to the cell at line l and row r as cell-lxr):

$$\begin{aligned} V &= \{\text{agent-at}\} \cup \{\text{visited-}l\text{x}r \mid l, r \in \{1,2,3,4,5\}\} \\ dom(\text{agent-at}) &= \{\text{cell-}l\text{x}r \mid l, r \in \{1,2,3,4,5\}\} \\ dom(\text{visited-}l\text{x}r) &= \{\text{true, false}\} \mid l, r \in \{1,2,3,4,5\}\} \end{aligned}$$

In addition, consider the abstract transition system induced by the pattern that only contains the variable visited-3x4. This abstract transition system has two states, the initial and the goal states, where the variable value is false and true, respectively. Operators that make the variable true connect the two states: the four (parallel) operators which have cell-3x4 as the target, as illustrated in Figure 4.6. All remaining transitions are self-loops that do not provoke a state change.

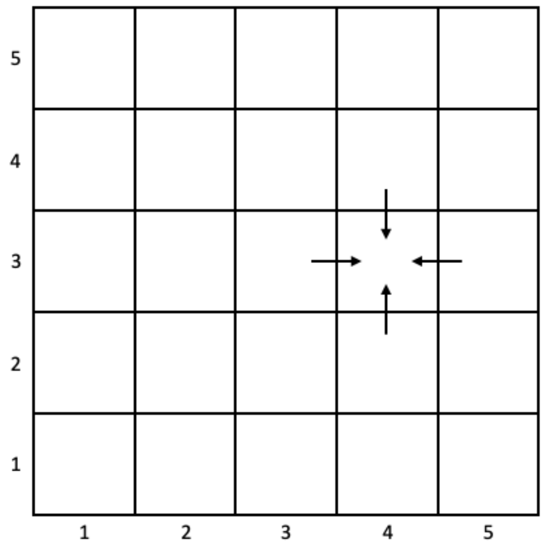

Figure 4.6 – Operators whose effect mark cell-3x4 as visited.

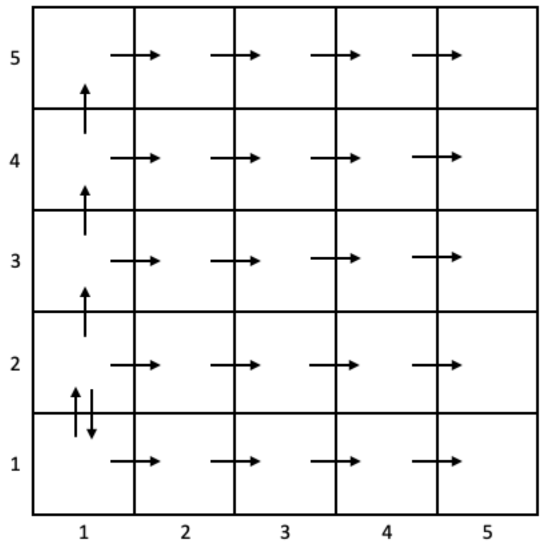

Figure 4.7 – Operators in the credit after all transition systems induced by $Sys1$ are evaluated.

In the code developed for the experiments, we store parallel transitions in a list and the tiebreak condition is selecting the first operator with the lower cost. For the abstract transition system example above, the order of the operators in the list is: arriving from left, arriving from down, arriving from right and arriving from up. Therefore, as all operators have the same cost, the transition relative to the first operator is chosen. The list of operators is ordered following this same sequence (when looking for the movement direction) for all abstract transition systems induced by $Sys1$ patterns. Thus, when h^{GP} finds an abstract plan for all abstract transition systems induced by $Sys1$ patterns, the result is a credit composed by the operators shown in Figure 4.7.

The next step is to find abstract plans for the abstract transition systems induced by patterns with two variables. Some of these patterns will be composed by the variable agent-at plus one of the "cell was visited" variables. For such pattern, since the agent position is relevant, the resulting transition system models the complete grid and all moves cause a state change. Now, let's analyze the operators required to achieve the goal when the pattern is composed by agent-at plus (a) visited-4x5 and (b) visited-2x1. In the first case, since the agent starts at cell-3x3, any path has to include at least two moves right and one move up. As the operators that move to the right are in the credit, the optimal abstract plan cost is one, corresponding to the movement upwards. On the other hand, to achieve cell-2x1, the agent has to move left twice and down once. However, none of these actions are in the credit, resulting in an abstract plan with cost three and adding three new operators to the credit. Hence, the credit contains more than one operator that arrives at the same cell. This happens with several cells on the left side of the grid, which incurs an unnecessary extra cost and makes the estimate less accurate. In some domains, this behavior can have an even worse consequence, while in others it can be opportunely good.

Aiming to mitigate this issue, we evaluated two modifications in the algorithm. The first one is to shuffle the lists of operators in the graph. This change adds randomness to the transition choice and makes the heuristic less sensitive to biases caused by predefined operator orders. Table 4.2 shows the results of the experiment using three distinct seeds. It illustrates that each domain is differently affected by the shuffle. For instance, the change is very beneficial for `VisitAll` and `Sokoban` domains, which had their coverage increased with the three seeds. In contrast, for the `Barman` domain, the heuristic can perform better or worse depending on the final arrangement of the operators. Nevertheless, the total coverage of 159 tasks from the experiment with seed 202 makes this an interesting variant to test when dealing with new domains.

The second approach tested to reduce the influence of the order of the operators is motivated by one of the causes of the issue: the necessity of choosing a transition. If one of the parallel operators is in the credit, it can be used for free and will not affect the plan cost. Thus, we modified the algorithm to sort the input list of abstract transition systems based on the amount of times that there's more than one edge with the same direction between two vertices. As a consequence, the credit may contain more operators when the abstract transition systems that require choices are evaluated, decreasing the chances of selecting unnecessary operators. Coverage results in table 4.3 shows the gain of seven new tasks solved by this variation over the original algorithm.

		Shuffle		
Coverage	Original	seed=48	seed=202	seed=500
barman (20)	13	9	**18**	13
elevators (20)	**0**	**0**	**0**	**0**
floortile (20)	**0**	**0**	**0**	**0**
nomystery (20)	**10**	**10**	**10**	**10**
openstacks (20)	**0**	**0**	**0**	**0**
parcprinter (20)	**18**	17	17	**18**
parking (20)	**14**	**14**	**14**	**14**
pegsol (20)	**20**	**20**	**20**	**20**
scanalyzer (20)	**17**	16	**17**	**17**
sokoban (20)	16	**18**	**18**	**18**
tidybot (20)	**19**	**19**	**19**	**19**
transport (20)	**0**	**0**	**0**	**0**
visitall (20)	5	**8**	**8**	**8**
woodworking (20)	**18**	**18**	**18**	**18**
Sum (280)	150	149	**159**	155

Table 4.2 – Domain-wise coverage for initial h^{GP}_{Sys2} propose and the variation shuffling operators lists with three different random seeds. The number of instances in each domain is shown in parenthesis next to the domain name. The best results over all algorithms are highlighted in bold.

Coverage	Original	Sorting
barman (20)	**13**	**13**
elevators (20)	**0**	**0**
floortile (20)	**0**	**0**
nomystery (20)	10	**13**
openstacks (20)	**0**	**0**
parcprinter (20)	**18**	**18**
parking (20)	**14**	**14**
pegsol (20)	**20**	**20**
scanalyzer (20)	**17**	**17**
sokoban (20)	16	**18**
tidybot (20)	**19**	**19**
transport (20)	**0**	**0**
visitall (20)	5	**7**
woodworking (20)	**18**	**18**
Sum (280)	150	**157**

Table 4.3 – Domain-wise coverage for initial h^{GP}_{Sys2} propose and the variation sorting graphs by the amount of pair of nodes connected by multiple edges. The number of instances in each domain is shown in parenthesis next to the domain name. The best results over all algorithms are highlighted in bold.

4.2 Unit-Cost Operators

Analyzing the results, we observed that the heuristic value of h^{GP} for the initial state of every task from the `OpenStacks` domain is zero. `OpenStacks` has only one operator with a non-zero cost, which does not affect the variables referenced in the goal. Therefore, we tested transforming all tasks of the benchmark set to make every operator have cost one (including operators that originally have cost zero). The results demonstrated an increase in coverage. However, this change does not have a specific impact on h^{GP}computation. Transforming the tasks tends to always increase the coverage of heuristics that do not perform well in domains with several cost-zero operators. Thus, we have this section to document this finding, but we do not focus on the experiment's results.

4.3 Partial Expansion

Memory allocation is one of the operations that consumes considerable time in the heuristic computation. It is mainly done when inserting new entries in the priority queue of the Dijkstra algorithm. Therefore, the more we can avoid these insertions, the faster the heuristic will be. The intuition for this modification is that (a) operators in the credit are more likely to compose good abstract plans since they were already used in previous graphs and (b) paths that contain operators in the credit probably are cheaper than others.

With this in mind, the idea is to not completely expand states that have both (a) transitions referent to operators that are in the credit and (b) referent to operators that are not. Instead, we only insert directly in the queue successors achieved for free, while the predecessor state is reinserted in the queue with the cost of its cheaper non-free successor. When the reinserted entry is removed from the queue again, the algorithm inserts into the queue the successor states that were not inserted before. Thus, all successor states that incur a cost increase will be only added to the queue if the predecessor is expanded again. However, if the algorithm achieves a goal state before the second expansion, those insertions will be saved.

The experiments with this modification show a slight improvement in the total time for most domains and the coverage stayed unchanged for all of them. Nevertheless, there was a significant reduction in the time required to solve tasks from `Parking` and `Scanalyzer` domains, with some tasks having up to 50% decrease in the total time.

5 FAILED IDEAS

During the development of this work we had several "good" ideas to improve the heuristic performance. Unfortunately, many of them did not produce the expected outcomes, and we don't currently know exactly why. In this Chapter, we present the most promising of these ideas that we tried to implement.

Even though the approaches discussed here did not lead to significant results, we still believe they are worth mentioning. Besides documenting what was already tried, they can be inspirations for related works.

Most of the ideas tested are related to the priority queue in the Dijkstra algorithm. Our aim was to reduce the number of insertions in the queue, make the search faster and/or improve the performance of operations made in the data structure. In addition, we also made experiments transforming the costs of the operators in the tasks.

5.1 Bucket Queue

In the initial version of the heuristic implementation, we use the standard priority queue from the C++ language (std.priority_queue) in the Dijkstra algorithm computation. The complexity of inserting an entry in this queue is logarithmic in relation to the number of elements in it. Therefore, aiming to improve the performance, we tested replacing the standard queue with a bucket queue: a data structure where the complexity of inserting a new entry is constant.

Our custom-implemented bucket queue is a dictionary that maps an integer (the priority) to a deck (entries with the respective priority). To insert a new entry in the queue, we simply add it to the end of the respective deck, which is retrieved with constant complexity from the dictionary.

Despite the theoretical advantage of the bucket queue, the results of the experiment using it did not show improvements in the average computation time. With the bucket queue, considering all solved tasks, the geometric mean of the total time metric is 11.72 seconds, while with the standard queue, the geometric mean is 10.82 seconds. Only in one domain the average total time was better using the bucket queue.

5.2 Using PDBs

The code we use to create the graphs (abstract transition systems) is from the Fast Downward package that computes the PDB heuristic. Once the PDB algorithm has to go through the complete abstract transition system to calculate the heuristic value of each state, we simply modified the code to also store the graph and make it available to the h^{GP}computation. Therefore, we can use the PDB heuristic without harming the execution time. In this section, we present two ideas that incorporate the PDB heuristic in our algorithm, with the intention of making the search in the abstract transition systems faster.

5.2.1 Using PDBs to Guide Dijkstra

Once the PDB heuristic is available, we can transform the Dijkstra algorithm into the A* algorithm (Hart et al., 1968), adding the heuristic value of a state to the cost required to achieve it when inserting the state into the queue. This aims to make the search more informed, which (ideally) enables the algorithm to find the abstract plans expanding fewer abstract states.

However, the PDB estimate does not consider the existence of the credit, which can reduce the remaining cost until the goal. To include this information, we subtract the sum of the cost of all operators still in the credit from the PDB estimate. Thus, the function to calculate the state's values $f(s)$ is

$$f(s) = g(s) + max(h^P(s) - cost(C),\ 0)$$

where $g(s)$ is the path cost and C is the credit.

The results of the experiment with this modification did not show improvements in most domains. On average, the new approach expanded the same or slightly fewer states than the original algorithm. However, the total time increased. This probably indicates that the gain of having the more informed search does not compensate for the extra cost added to execute this strategy.

5.2.2 Using PDBs to Early Stop Dijkstra

We observed that, when the Dijkstra algorithm already used all operators in the credit to achieve a determined state, we can simply stop the expansion of this state. Since the PDB heuristic already finds the cheaper paths from every state to a goal, it is possible to just save these plans and reuse them in h^{GP}computation when the credit is completely consumed.

Therefore, we extended the code to (a) store the optimal abstract plans identified by PDB heuristic computation and (b) check the credit before expanding a state s in the Dijkstra algorithm. If the credit is empty (considering only operators that affect the abstract transition system), s is reinserted in the priority queue with its value plus $h^P(s)$. In case this entry is removed from the queue again, we recover the optimal plan from s to the closest goal from the PDB computation and finish the Dijkstra search.

Although promising, the experiments with this modification also did not show improvements compared to the original algorithm. For almost all tasks, the total time increased.

6 CONCLUSION

This work introduced a new heuristic to satisficing planning that uses a greedy strategy to combine multiple abstract transition systems. We also presented some variations and improvement ideas that can be explored to increase the heuristic performance. The empirical evaluation of the heuristic demonstrated that both versions (h^{GP}_{Sys1} and h^{GP}_{Sys2}) have the potential to compete with traditional heuristic functions such as h^{FF}.

Our heuristic function synchronizes abstract plans for abstract transition systems in an ordered list. It starts by finding an optimal abstract plan for the first abstract transition system. The operators in this plan are now a credit: since they are already used in a plan, they can be used again for free. Considering this credit, the heuristic function finds an abstract plan for the next abstract transition system. The operators in this new plan that incurs a cost to it are also included in the credit list. Following this approach, we find an abstract plan for every abstract transition system in the list. The heuristic value is the sum of the costs of each operator in the final credit.

The main advantage of this heuristic is that it expands fewer states when compared with h^{FF} and h^{PhOG}_{C} in most domains. h^{GP}_{Sys2} is a good alternative when dealing with difficult tasks that require minutes (or more) to be solved. The $Sys2$ patterns require more time to be explored but provide more information, compensating in the long term. In contrast, h^{GP}_{Sys1} is a good alternative especially when the time available is limited since the heuristic is able to expand more states per second with small patterns.

The high computation time of the heuristic is one of its big disadvantages. Even using small patterns, the list of abstract transition systems and their size can be considerably large, and finding abstract plans for each of them is costly. Therefore, future works can investigate strategies to make the heuristic faster. For instance, we could try to use different (and smaller) pattern collections, possibly obtained with pattern selection approaches that intend to generate reduced pattern collection without losing information. Furthermore, the variations presented in Chapter 5 are another possible starting point, since there is space to continue investigating and experimenting with them. We could try to understand the reason why they do not work as expected and, possibly, improve the strategy or even combine the variations with each other looking for better results.

REFERENCES

Bonet, B., & Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, *129*(1), 5–33.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*, 269–271.

Doran, J. E., & Michie, D. (1966). Experiments with the graph traverser program. *Proceedings of the Royal Society A*, *294*, 235–259.

Edelkamp, S. (2001). Planning with pattern databases. In Cesta, A., & Borrajo, D. (Eds.), *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, pp. 84–90. AAAI Press.

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

Helmert, M. (2006). The Fast Downward planning system. *Journal of Artificial Intelligence Research*, *26*, 191–246.

Hoffmann, J. (2011). Everything you always wanted to know about planning (but were afraid to ask). In Bach, J., & Edelkamp, S. (Eds.), *Proceedings of the 34th Annual German Conference on Artificial Intelligence (KI 2011)*, Vol. 7006 of *Lecture Notes in Artificial Intelligence*, pp. 1–13. Springer-Verlag.

Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, *14*, 253–302.

Pommerening, F., Röger, G., & Helmert, M. (2013). Getting the most out of pattern databases for classical planning. In Rossi, F. (Ed.), *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, pp. 2357–2364. AAAI Press.

Seipp, J., Pommerening, F., Sievers, S., & Helmert, M. (2017). Downward Lab. `https://doi.org/10.5281/zenodo.790461`.