UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

WESLEY LUCIANO KAIZER

Sequencing Operator Counts with State-Space Search

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. André Grahl Pereira

Porto Alegre
May 2020

CIP — CATALOGING-IN-PUBLICATION

Kaizer, Wesley Luciano

Sequencing Operator Counts with State-Space Search / Wesley Luciano Kaizer. – Porto Alegre: PPGC da UFRGS, 2020.

64 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2020. Advisor: André Grahl Pereira.

1. Artificial Intelligence. 2. Classical Planning. 3. State-Space Heuristic Search. 4. Integer Programming. 5. Operator-Counting Framework. I. Pereira, André Grahl. II. Título.

“You check out guitar George, he knows all the chords
Mind, he’s strictly rhythm he doesn’t want to make it cry or sing
And an old guitar is all he can afford
When he gets up under the lights to play his thing”

AGRADECIMENTOS

Agradeço à minha família e a todos os amigos da cidade de Cascavel/PR por todo o incentivo e apoio que recebi durante o mestrado.

Agradeço a todos os professores da UNIOESTE que contribuíram com minha formação profissional. Agradeço aos professores Rogério Rizzi, Claudia Rizzi e Guilherme Galante com os quais tive a oportunidade de trabalhar conjuntamente durante a gradução em quatro projetos de iniciação científica, no estágio obrigatório e no trabalho de conclusão de curso. Agradeço especialmente ao professor Rogério Rizzi pela oportunidade de participar, após formado, como pesquisador durante seu pós-doutorado. Sempre lembrarei de todo o conhecimento transmitido durante aquele ano, tanto no âmbito profissional quanto pessoal.

Agradeço imensamente ao professor André Grahl Pereira por ter me aceito como seu aluno, mesmo sem me conhecer anteriormente, e pela oportunidade de cursar um mestrado em uma universade de renome como a UFRGS. Agradeço por todo o conhecimento compartilhado, pela sua incondicional confiança e atenção, pelo apoio e incentivo em momentos de incerteza durante a pesquisa e por todas as conversas e reuniões que tivemos que resultaram na conclusão deste trabalho. Independentemente de qual caminho profissional trilharei, eu sempre levarei comigo toda a bagagem científica que me ajudou a construir. Agradeço a todos os professores e funcionários da UFRGS que possibilitaram o desenvolvimento e conclusão deste mestrado, especialmente aos professores com os quais cursei disciplinas: Álvaro Moreira e Marcus Ritt pela disciplina de Algoritmos e Teoria de Computação, Luciana Buriol pela disciplina de Otimização, e Bruno Castro da Silva e Mariana Mendonza pela disciplina de Aprendizagem de Máquina. Agradeço ainda ao professor Marcus Ritt por toda a sua ajuda como um coorientador durante o segundo ano do mestrado, e aos professores Rafael Coelho, Felipe Meneguzzi e Levi Lelis por aceitarem fazer parte da banca. Por fim, agradeço a todos os colegas do laboratório de algoritmos e otimização pela recepção, companhia e ajuda durante o mestrado, principalmente pelos vários *extended coffee-breaks* que participei.

ABSTRACT

A search algorithm with an admissible heuristic function is the most common approach to optimally solve classical planning tasks. Recently DAVIES et al. (2015) introduced the solver *OpSeq* using Logic-Based Benders Decomposition. In this approach to planning, the master problem is an integer program derived from the *operator-counting* framework that generates *operator counts*, i.e., an assignment of integer counts for each task operator. Then, the *operator counts sequencing subproblem* verifies if a plan satisfying these operator counts exists, or generates a violated constraint to strengthen the master problem. In *OpSeq*, the subproblem is solved by a SAT solver.

In this thesis, we show that this subproblem can be better solved by state-space search. We introduce *OpSearch*, an A*-based algorithm to solve the operator counts sequencing problem: it either finds an optimal plan, or uses the frontier of the search, i.e., the set of generated but yet unexpanded states, to derive a violated constraint. We show that using a standard search framework has three advantages: i) the search scales better than a SAT-based approach for solving the operator counts sequencing, ii) explicit information in the search frontier can be used to derive stronger constraints, and iii) stronger heuristics generate more informed constraints.

We present results using the benchmark of the International Planning Competition, showing that this approach solves more planning tasks, using less memory. On tasks solved by both methods, *OpSearch* usually requires solving fewer operator counts sequencing problems than *OpSeq*, evidencing the stronger constraints generated by *OpSearch*.

Keywords: Artificial Intelligence. Classical Planning. State-Space Heuristic Search. Integer Programming. Operator-Counting Framework.

Sequenciamento de Contagens de Operadores com Busca em Espaço de Estados

RESUMO

Um algoritmo de busca com uma função heurística admissível é a abordagem mais comum para resolver otimamente tarefas de planejamento. Recentemente DAVIES et al. (2015) introduziram o resolvedor *OpSeq* usando uma Decomposição de Benders Baseada em Lógica. Nesta abordagem para planejamento, o problema principal é um programa inteiro derivado da estrutura de *operator-counting* que gera *contagens de operadores*, isto é, uma atribuição de contagens inteiras para cada operador da tarefa. Em seguida, o *problema de sequenciamento de contagens de operadores* verifica se um plano satisfazendo estas contagens de operadores existe, ou gera uma restrição violada para fortificar o problema principal. Em *OpSeq*, o subproblema é resolvido por um resolvedor SAT.

Nesta dissertação, mostramos que este subproblema pode ser melhor resolvido por busca em espaço de estados. Introduzimos *OpSearch*, um algoritmo baseado em A* para resolver o problema de sequenciamento de contagens de operadores: ele encontra um plano ótimo, ou usa a fronteira da busca, isto é, o conjunto de estados gerados mas ainda não expandidos, para derivar uma restrição violada. Mostramos que utilizar uma estrutura de busca padrão tem três vantagens: i) a busca escala melhor que uma abordagem baseada em SAT para resolver o sequenciamento de contagens de operadores, ii) informação explícita na fronteira de busca pode ser usada para derivar restrições mais fortes, e iii) heurísticas mais fortes geram restrições mais informadas.

Apresentamos resultados utilizando o conjunto de instâncias da Competição Internacional de Planejamento, mostrando que esta abordagem resolve mais tarefas de planejamento, usando menos memória. Nas tarefas resolvidas por ambos os métodos, *OpSearch* geralmente requer resolver menos problemas de sequenciamento de contagens de operadores que *OpSeq*, evidenciando as restrições mais fortes geradas por *OpSearch*.

Palavras-chave: Inteligência Artificial, Planejamento Clássico, Busca Heurística em Espaço de Estados, Programação Inteira, Estrutura de Operator-Counting.

LIST OF ABBREVIATIONS AND ACRONYMS

BC	Branch and Cut
BIP	Binary Integer Program
CBD	Classical Benders Decomposition
DTG	Domain Transition Graph
GLC	Generalized Landmark Constraint
IP	Integer Program
IPC	International Planning Competition
LBBD	Logic-Based Benders Decomposition
LP	Linear Program
MIP	Mixed Integer Program
NP	Nondeterministic Polynomial Time
PDB	Pattern Database
SAT	Propositional Satisfiability Problem

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

In *optimal classical planning*, a solution for a *planning task* is a *plan*, a sequence of *operators*, that achieve some *goal state* from an *initial state*. Finding solutions to planning tasks is a PSPACE-complete problem (BÄCKSTRÖM; NEBEL, 1995), and hence intractable in general. However, *heuristic search algorithms* such as A* (HART; NILSSON; RAPHAEL, 1968) with automatically derived *heuristic functions* (heuristics), e.g., *pattern databases* h^{PDB} (EDELKAMP, 2014), h^{LMCut} (HELMERT; DOMSHLAK, 2009) and *merge and shrink* $h^{\text{M\&S}}$ (HELMERT et al., 2007), have achieved notable progress. A* with these strong heuristics can search large state-spaces efficiently, solving many hard planning tasks in practice.

Many recently proposed heuristics are based on linear programming. The *operator-counting framework* (POMMERENING et al., 2014) is of particular interest because it combines information from many admissible heuristics in terms of constraints of a linear program, that must be satisfied by every plan for the planning task. Thus, the optimal value of the objective function is an admissible estimate of the cost of an optimal plan, an *admissible heuristic*. Among the sources of admissible operator-counting constraints are: disjunctive action landmarks h^{LMC} (BONET; BRIEL, 2014), state equation h^{SEQ} (BONET, 2013), post-hoc optimization h^{PhO} (POMMERENING; RÖGER; HELMERT, 2013), and the optimal delete relaxation h^+ (IMAI; FUKUNAGA, 2014). The operator-counting framework and methods from *Operations Research* have also inspired the development of alternative views for the planning problem.

DAVIES et al. (2015) introduced a novel approach for cost-optimal planning, recognizing that the primal solution of the operator-counting linear program contains useful information that can be understood as a possibly incomplete and unordered plan. This approach interprets the operator-counting framework beyond its primary use as a heuristic function and decomposes the process of finding solutions to a planning task into two independent but related problems, using a *Logic-Based Benders Decomposition* (HOOKER; OTTOSSON, 2003).

In this decomposition for planning problems, there is a *master problem* and a combinatorial *subproblem* used to explain the infeasibility of a solution to the master. The master problem is modeled as an *integer program*, corresponding to an *operator-counting heuristic*. The subproblem is modeled as a *propositional satisfiability problem* (SAT) encoding the planning task and the *operator counts* obtained from the primal solution of the master. A SAT solver is then used to *sequence* the operator counts, i.e., to check if a plan with these counts exists. If there is no plan with the given operator counts, the SAT solver returns a *violated constraint* for the master problem.

In this thesis, we propose an algorithm to solve the operator counts sequencing subproblem using heuristic search instead of a SAT-based formulation. This new approach is based on an A* search that employs information unavailable to SAT solvers, such as the f-value of search nodes and the explicit structure of the search graph. We present a novel strategy to construct a violated constraint during the expansion of the search graph by considering the frontier of the search. We show that this strategy generates an admissible generalized landmark constraint. We experimentally show that the resulting algorithm *OpSearch* has better coverage and less memory requirements than a SAT-based approach and can generate smaller and more informative explanations of infeasibility, as shown by the total number of solved subproblems required to solve the planning tasks. We believe this approach is relevant because it opens new research directions towards specialized operator counts sequencing methods based on well-known classical planning technologies.

The paper *Sequencing Operator Counts with State-Space Search* containing the main results presented in this thesis was accepted for publication at the 30th International Conference on Automated Planning and Scheduling (ICAPS 2020).

2 BACKGROUND

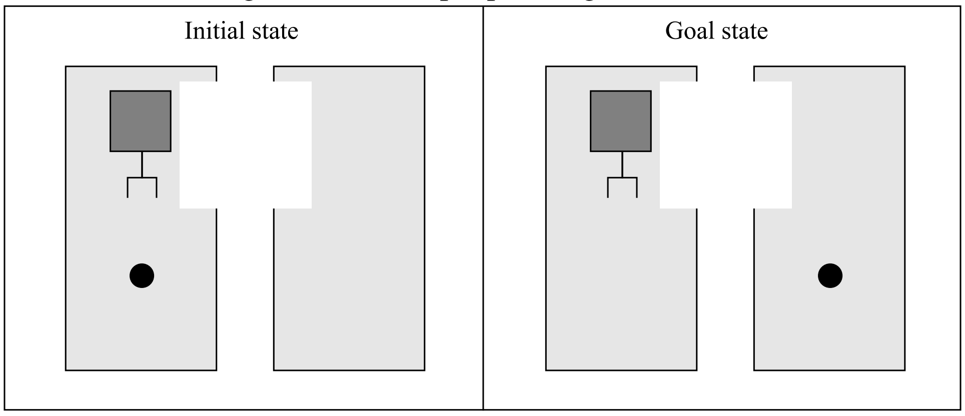
2.1 Automated Planning

2.1.1 Overview

Automated planning aims to find a sequence of operators, called *plan*, whose application achieves a goal state from an initial state. States describe currently valid conditions and operators modify them at some cost. There exists only one initial state but possibly many goal states. *Planning* is a general problem solving technique with many applications in industry; for example, robot navigation, activities scheduling, tasks automation, puzzle solving, and many others. Here, we focus on *classical optimal planning*, in which operators have discrete, deterministic and fully observable effects, the plan is computed offline and has the global minimal cost (GHALLAB; NAU; TRAVERSO, 2004).

As an illustrative example, consider the planning task Π_{robot} presented in Figure 2.1 in which a robot must transport one ball from the left to the right room. The robot starts in the left room and must return to it after transporting the ball. We can model this task using two variables: *ball_at* for the ball position and *robot_at* for the robot position. Variable *ball_at* can assume the value *left*, when the ball is in the left room, *right* when the ball is in the right room, and *robot* when the ball is in the robot's hand. Variable *robot_at* can assume the value *left* when the robot is in the left room and *right* when the robot is in the right room. In the initial state we have *ball_at=left* and *robot_at=left* and in goal states we want *ball_at=right* and *robot_at=left*.

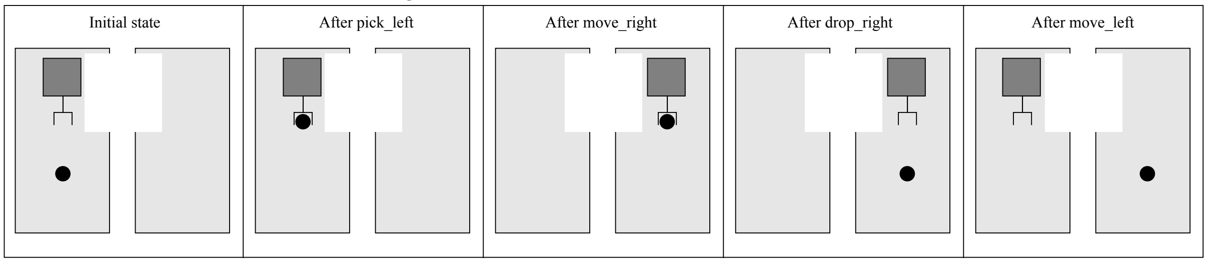
Figure 2.1: Example planning task Π_{robot}.



We define six operators to reach the goal by changing the values of *ball_at* and *robot_at*: *pick_left* and *pick_right* cause the robot to pick the ball at the left or right room, *drop_left* and *drop_right* cause the robot to drop the ball at left or right room, and *move_right* and *move_left* cause the robot to move from the left room to the right and from the right room to the left. To apply *pick_left* or *pick_right*, both the robot and the ball must be in the left or right room, respectively. To apply *drop_left* or *drop_right*, the ball must be in the robot's hand and the robot must be in the left or right room, respectively. To apply *move_left* or *move_right*, the robot must be in the right or left room, respectively.

We can define any non-negative costs for the operators. For example, if we want to minimize the total number of applied operators or steps, we can define all costs equal to one. If we want to minimize the movements made by the robot, possibly because they consume fuel or energy, we can define non-zero costs for the *move_left* and *move_right* operators and zero cost for the others. For instance, the optimal plan for task Π_{robot} considering that operators *pick_left* and *pick_right* cost 4, *drop_left* and *drop_right* cost 2, and *move_left* and *move_right* cost 10 is $\langle pick\_left, move\_right, drop\_right, move\_left\rangle$ with total cost of $4 + 10 + 2 + 10 = 26$. This plan is illustrated in Figure 2.2. Operator *pick_left* causes the robot to pick the ball, *move_right* causes the robot to move to the right room while holding the ball, *drop_right* causes the robot to drop the ball in the right room and *move_left* causes the robot to move back to its initial position.

Figure 2.2: Plan to task Π_{robot}.



Domain-independent planning aims to find a plan based only on a high-level specification of the task, having no other useful information about the specific domain or instance that humans could easily perceive. This allows *planners* to be flexible and try to solve any problem that could be described using this specification. A *domain-independent optimal planner* is a software that takes a planning task description written in some formalism and outputs a plan with optimal cost. This description allows the planner to compactly represent tasks internally since it contains only the initial state, the goal specification, the variables and the operators. Furthermore, it allows to implicitly represent state-spaces in a declarative way. One such high-level formalism for planning is SAS$^+$.

2.1.2 SAS$^+$ Formalism

Definition 1 (SAS$^+$ Task). *An* SAS$^+$ planning task Π *is a tuple* $\langle \mathcal{V}, O, s_0, s_*, c\rangle$ *where:*

- $\mathcal{V}$ *is the set of* variables*;*
- O *is the set of* operators*;*
- s_0 *is the* initial state*;*
- s_* *is the* goal condition*;*
- c *is the* cost function for operators.

Each variable $v \in \mathcal{V}$ has a finite domain $D(v)$ that describes the possible values v can assume. An *atom* can be written as $\langle v = x\rangle$ where $v \in \mathcal{V}$ and $x \in D(v)$, denoting the assignment of one of the values in the variable's domain to v. A *complete state* or *state* s is a complete assignment for all variables $v \in \mathcal{V}$ to values $x \in D(v)$ and a *partial state* s is a partial assignment over some subset of $\mathcal{V}$. We write $vars(s)$ for the set of variables defined in state s, $s[v]$ for the value of variable v in state s, and S for the set of all states of Π, also known as the *state-space*. State s_0 is complete and s_* is partial. We say that a state s is *consistent* with a partial state s' if $s(v) = s'(v)$ for all $v \in vars(s')$. A *goal* is a state consistent with s_*.

Each operator $o \in O$ is a pair of partial states $\langle pre(o), post(o)\rangle$. Partial state $pre(o)$ represents *preconditions*: operator o is applicable in all states s that are consistent with $pre(o)$. Partial state $post(o)$ represents *effects* of applying operator o to a state s, which produces a new state s' identical to s but with updated values for affected variables, i.e., for all $v \notin vars(post(o))$: $s'[v] = s[v]$ and for all $v \in vars(post(o))$: $s'[v] = post(o)[v]$. Function $c : O \to \mathcal{Z}_0^+$ assigns to each operator $o \in O$ a non-negative cost $c(o)$. We say that task Π is unit-cost if for all $o \in O$ we have $c(o) = 1$.

An s-plan π is a sequence of operators $\langle o_1, \ldots, o_n\rangle$ such that there exists a sequence of states $\langle s_1 = s, \ldots, s_{n+1}\rangle$ where o_i is applicable to s_i and produces state s_{i+1}, and s_{n+1} is consistent with s_*. The *plan length* of π is n, i.e., the total number of operators. The *plan cost* of π is defined as $cost(\pi) = \sum_{o \in \pi} c(o)$. Finally, an s_0-plan is simply called a *plan*, and solving a classical planning task optimally means to find a plan π for Π of minimal cost or prove that no plan exists.

To illustrate the SAS$^+$ formalism, we show the formulation of the example planning task Π_{robot} discussed earlier. The task Π_{robot} encoded using the SAS$^+$ formalism is a tuple $\Pi_{robot} = \langle \mathcal{V}, O, s_0, s_*, c\rangle$ in which:

The set of variables is $\mathcal{V} = \{ball\_at, robot\_at\}$ with:

$$D(ball\_at) = \{left, right, robot\};$$
$$D(robot\_at) = \{left, right\}.$$

The set of operators is $O = \{pick\_left, pick\_right, drop\_left, drop\_right, move\_left, move\_right\}$ in which:

$$\begin{aligned}
&pick\_left = \langle robot\_at = left \wedge ball\_at = left; ball\_at := robot\rangle;\\
&pick\_right = \langle robot\_at = right \wedge ball\_at = right; ball\_at := robot\rangle;\\
&drop\_left = \langle robot\_at = left \wedge ball\_at = robot; ball\_at := left\rangle;\\
&drop\_right = \langle robot\_at = right \wedge ball\_at = robot; ball\_at := right\rangle;\\
&move\_left = \langle robot\_at = right; robot\_at := left\rangle;\\
&move\_right = \langle robot\_at = left; robot\_at := right\rangle.
\end{aligned}$$

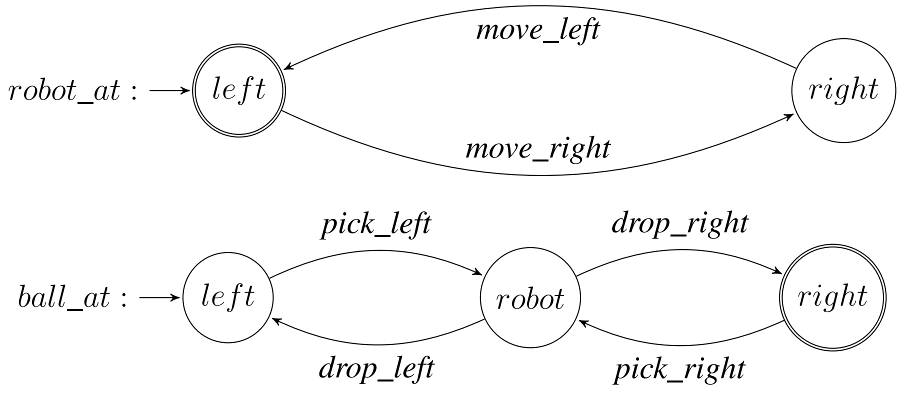
The initial state is $s_0 = \langle ball\_at = left \wedge robot\_at = left\rangle$ and the goal condition is $s_* = \langle ball\_at = right \wedge robot\_at = left\rangle$.

Finally, the cost function c for operators is:

$$c(o) = \begin{cases} 4, & \text{if } o \in \{pick\_left, pick\_right\},\\ 2, & \text{if } o \in \{drop\_left, drop\_right\},\\ 10, & \text{if } o \in \{move\_left, move\_right\}.\end{cases}$$

The *Domain Transition Graph* (DTG) for a planning task and some of its variables v is a common representation that describes the transitions of values of v by the application of task operators. A DTG for a variable $v \in \mathcal{V}$ is a directed graph with nodes for each value $x \in D(v)$ and a labeled arc for each transition over v, in which each label corresponds to an operator in the planning task. The arcs can contain more than one label, since several operators can change the value of a variable in the same way. Figure 2.3 shows the DTGs for the variables of the example planning task Π_{robot}. For each variable, the initial states are marked by an incoming arrow and goal states are double circled. Arcs between nodes correspond to transitions induced by applicable operators.

Figure 2.3: DTGs for the task Π_{robot}.



Planning tasks are usually solved using *heuristic search algorithms* to efficiently explore the state-space. These algorithms rely on *heuristic functions* to map each state to an estimate of the remaining cost to a goal. In the next section, we discuss the A* algorithm, that is widely applied in the planning field and expands the state-space preferring to extend the most promising plan prefixes found until the moment.

2.1.3 A* Algorithm

A* (HART; NILSSON; RAPHAEL, 1968) is the most prominent heuristic search algorithm in classical optimal planning and in the area of artificial intelligence in general. In the context of planning, A* aims to find a plan with minimal cost from the initial state s_0 to some goal state consistent with s_*, using three functions f, g and h to map states to numeric values. Function $g(s)$ is the cost of the best-known plan from the initial state s_0 to state s at the current step of A*. Function $h(s)$ is the *heuristic function* that estimates the cost of an optimal plan from s to some goal state. Finally, function $f(s) = g(s) + h(s)$ estimates the cost of an optimal plan from s_0 to a goal state, going through state s. A* is itself *admissible*, i.e., always returns a cost-optimal plan, when using an admissible heuristic function. In the next section we define relevant properties of heuristic functions.

A* maintains two lists of states during the search process: the *open* and *closed* lists. The *open list* stores the states yet to be expanded by A* and is usually implemented as a priority queue with elements ordered in ascending order by f-values. The *closed list* stores states already expanded and is implemented as a set. Both lists are populated during the *state expansion* step. In *state expansion*, the A* algorithm expands a state s by generating its successor states, i.e., it applies all the available operators to s. When state s

is expanded, it is removed from the *open list* and inserted in the *closed list* and each of its successors is inserted in the *open list* if it was not generated before, i.e., it is not in *open* or *closed*, or it has a better g-value than any equivalent state previously generated.

Initially, the *open list* contains only the initial state. A* iteratively chooses the state with the smallest f-value from the *open list* to expand. A* stops when it chooses a state s consistent with s_*. The plan is constructed by backtracking from state s to the initial state s_0 using the search graph constructed during the process. The time complexity of the A* algorithm is usually measured by the number of *state expansions* performed before reaching a goal state. Below we present the A* algorithm:

Algorithm 1: The A* algorithm.

```
open := priority queue ordered by ascending $f$-value
closed := $\emptyset$
push $s_0$ to open
while open $\neq \emptyset$ do
    pop a state $s$ from open
    if $s$ is consistent with $s_*$ then
        return plan from $s_0$ to $s$
    for each operator $o$ applicable in $s$ do
        apply $o$ in $s$ generating a state $s'$
        if $s' \notin$ closed $\cup$ open then
            push $s'$ to open
        else
            if $\exists\, s'' \in$ open s.t. $s'' = s'$ and $g(s') < g(s'')$ then
                replace $s''$ with $s'$ in open
            if $\exists\, s'' \in$ closed s.t. $s'' = s'$ and $g(s') < g(s'')$ then
                remove $s''$ from closed and push $s'$ to open
    push $s$ to closed
return Failure
```

An important component of the A* algorithm is the *heuristic function* h, that allows the algorithm to prune unpromising states during search and is the main responsible for its efficiency. In the next section we discuss heuristic functions in more detail.

2.1.4 Heuristic Functions

A *heuristic function* $h : S \to \mathcal{Z}_0^+ \cup \{\infty\}$ maps a state s to an h-value, an estimate of the cost of an s-plan. The *perfect heuristic* h^* maps a state s to the cost of an optimal s-plan or to ∞ if no plan exists. Below we summarize some properties of heuristic functions:

- h is *safe* if $h(s) = \infty$ implies $h^*(s) = \infty$ for all states $s \in S$;
- h is *goal-aware* if $h(s) = 0$ for all states $s \in S$ consistent with s_*;
- h is *admissible* if for all $s \in S$ we have $h(s) \leq h^*(s)$;
- h is *consistent* if for all transitions $s \xrightarrow{o} s'$, $h(s) \leq h(s') + c(o)$ is valid;
- If h is *admissible*, then it is also *goal-aware* and *safe*;
- If h is *consistent* and *goal-aware*, then it is also *admissible*.

A heuristic function is *safe* if it correctly detects *dead ends*. A *dead end* is a state s for which there does not exist an s-plan. However, this does not imply that a *safe* heuristic h detects all dead ends of the planning task, i.e., there could exist a state s in which $h(s) \neq \infty$ but $h^*(s) = \infty$. A heuristic is *goal-aware* if it correctly detects all *goal states*. If s is a goal state, it is easy to see that the cost of an s-plan should be 0. A heuristic is called *admissible* if it is a lower bound on the optimal plan cost for all states. Finally, a heuristic h is consistent if the application of an operator o in state s, generating state s', always decreases the heuristic estimate from $h(s)$ to $h(s')$ by at most its cost $c(o)$. A* with a consistent heuristic is guaranteed to expand a state at most once (HOLTE, 2010).

We can compare admissible heuristics using the concept of *dominance*: an admissible heuristic function h_2 is said to *dominate* other admissible heuristic h_1 if for all states $s \in S$ the inequality $h_2(s) \geq h_1(s)$ is valid and there exists at least one state s for which $h_2(s) > h_1(s)$. It seems intuitive to think that A* is guaranteed to expand fewer states using h_2 than h_1. Although there exist experimental results showing correlations between the use of more informed heuristics and the expansion of fewer states during search by A*, HOLTE (2010) shows that this is not always true, even for consistent heuristics.

It is possible to automatically derive heuristic functions by relaxing some aspects of the planning task through a process denominated *task relaxation*. The resulting heuristics are usually grouped in heuristic function classes, depending on the relaxation procedure applied. Currently, well-known heuristic function classes are *critical path*, *delete relaxation*, *abstraction* and *landmark* (HELMERT; DOMSHLAK, 2009).

Critical-path heuristics estimate plan costs using sets of atoms of cardinality m, representing sub-goals that must be achieved during the plan execution. The main example is the h^m heuristic family (GEFFNER; HASLUM, 2000). *Delete relaxation* heuristics assume that variables can preserve their past values through the application of operators. The *delete effects* of operators are ignored and they can only add new valid atoms. Some examples are the h^+, $h^{\max}$, h^{add}, and h^{FF} heuristics (HOFFMANN; NEBEL, 2001; BONET; GEFFNER, 2001). *Abstraction* heuristics use optimal costs of simpler tasks as heuristic estimates. The task is mapped to several abstract tasks that can be optimally solved more efficiently than the original. Examples of such heuristics are the *pattern database heuristics* h^{PDB} and the *merge-and-shrink* $h^{\text{M\&S}}$ (EDELKAMP, 2014; HELMERT et al., 2007). *Landmark* heuristics use atoms or operators that must occur in every s-plan to estimate plan costs. Although deciding if an atom or operator is in fact a task landmark is a PSPACE-hard problem, there are methods able to efficiently compute a subset of all landmarks at each search state. Examples are the h^{LMC}, h^{L}, h^{LA} and h^{LMCut} heuristics (KARPAS; DOMSHLAK, 2009; HELMERT; DOMSHLAK, 2009).

First uses of linear programming in cost-optimal planning relate to *cost-partitioning*, a method to admissibly combine several abstraction heuristics by partitioning operator costs among them. KATZ; DOMSHLAK (2008) is the seminal work that proposes a linear programming formulation to the *non-negative cost-partitioning* method in planning, that restricts the partitioned operator costs to be non-negative. POMMERENING et al. (2015) demonstrate that enforcing this constraint is not necessary and proposes a *general cost partitioning* scheme that allows negative costs and also introduces the family of *potential heuristics*. Other examples of cost-partitioning methods are the *saturated cost partitioning* (SEIPP; HELMERT, 2014) and *uniform cost-partitioning* (SEIPP; KELLER; HELMERT, 2017).

POMMERENING et al. present several admissible heuristics that can be expressed using linear programming such as disjunctive action landmarks, state equation and post-hoc optimization. Others heuristics are the IP formulation for the optimal delete relaxation heuristic h^{+} introduced by IMAI; FUKUNAGA (2014) and the *dynamic merging* method from (BONET; BRIEL, 2014) based on flow constraints. Due to the hardness of IP problems, generally the objective function value of the linear relaxation is used to guide a heuristic search algorithm. POMMERENING et al. (2014) also show that it can be advantageous to optimize a linear program at each search state and that some combinations of constraints originated from different heuristics can be more informative than each of its components alone.

2.2 Integer Programming

2.2.1 Overview

Integer programming is a technique originated from the *Operations Research* field that aims to encode a *combinatorial optimization problem* as an *Integer Program* (IP). In this class of problems, we usually have a finite set of elements N, a numeric weight for each element, and a set F of feasible subsets of N. The objective is to find a feasible subset of elements with minimum total weight. Examples of problems that can be formulated as integer programs are *train scheduling*, in which train travels are scheduled based on the time spent at each station and total travel time; *airline crew scheduling*, in which crews are assigned to serve flights weekly, satisfying constraints regarding to the total flying time, minimum rest, and others; and *telecommunications*, in which there is a demand to install new capacities to satisfy predicted demands for data transmission in certain areas while minimizing costs (WOLSEY, 1998).

A *Linear Program* (LP) is a mathematical formulation of some problem written in terms of a set of *variables*, a set of linear *constraints* modeled as linear inequalities and a linear *objective function*. Solving an LP means to find values for the variables that optimizes the objective function, subject to the constraints. Some subset of these variables are called the *decision variables* and usually encode decisions that must be taken, for instance, which train travels from station A to B or in which location L to install a facility F, or even quantities such as the cost budget available to compose a flight crew.

Definition 2 (**Linear Program**)**.** *A Linear Program (LP) consists of:*

- *a finite set of real-valued variables $\mathcal{V}$;*
- *a finite set of linear constraints over $\mathcal{V}$;*
- *an objective function Z, which is a linear combination of $\mathcal{V}$;*
- *an optimization objective, which should be to minimize or maximize Z.*

In IPs, the variables can assume only integer values and this requirement is usually called the *integrality constraints*. In a *binary integer program* (BIP), all variables can only assume two values: 0 or 1. In a *mixed integer program* (MIP), some variables can assume any real value while others can assume only integer values. The objective function *maximizes* or *minimizes* some linear combination of the variables.

Sometimes it is preferable to describe LPs using vector and matrix notations from

Linear Algebra. We only present the representation for minimization problems, since it is analogous for maximization.

Definition 3 (Matrix Notation for Linear Programs). *A Linear Program can be written using matrix notation with a column vector $b = \langle b_1, \ldots, b_m \rangle^T$, a column vector $c = \langle c_1, \ldots, c_n \rangle^T$, and an $n \times m$ matrix A. The problem is to find a column vector $y = \langle y_1, \ldots, y_m \rangle^T$ to minimize $b^T y$ subject to $Ay \geq c$ and $y \geq 0$.*

Note that a minimization problem can be converted to a maximization problem and vice versa by multiplying the objective function by -1 since $minimize(Z) = maximize(-Z)$ and converting constraints in the form $Ay \geq c$ to $-Ay \leq -c$.

A vector y is *feasible* if it satisfies the constraints. An LP is *feasible* if there exists a feasible vector, otherwise it is *infeasible*. A minimization problem is *unbounded* if the objective function can assume arbitrarily large negative values with feasible vectors, otherwise it it *bounded*. The *optimal objective value* of a bounded feasible minimization problem is the minimum value of the objective function with a feasible vector.

Next we illustrate the integer programming formulation of a particular instance of the general *set covering problem* (WOLSEY, 1998): given a certain number of cities, we must decide which set of emergency centers to install. The cost to install a center and the cities that it could serve are previously known. The objective is to choose a set of centers with minimum total installation cost, guaranteeing that all cities are covered.

Let $M = \{1, \ldots, m\}$ be the set of cities and $N = \{1, \ldots, n\}$ be the set of centers. Let S_j be the cities that could be served by the center j and c_j its installation cost. This problem can be formulated as a BIP with variables x_j, denoting if center j is installed or not.

The objective function can be expressed as $\sum_{j=1}^{n} c_j x_j$. The constraints in the form $\sum_{j \in [n] \,|\, i \in S_j} x_j \geq 1$ express the condition that each city i must be served by at least one emergency center. Finally, the constraints $x_j \in \{0, 1\}$ restrict the variables x_j to the binary domain $\{0, 1\}$. The complete BIP for this problem can be written as:

$$
\begin{aligned}
\text{minimize} & \sum_{j=1}^{n} c_j x_j & \\
\text{subject to} & \sum_{j \in [n] \,|\, i \in S_j} x_j \geq 1 & \text{for } 1 \leq i \leq m, \\
& x_j \in \{0, 1\} & \text{for } 1 \leq j \leq n.
\end{aligned}
$$

Finding an optimal solution to an IP is an NP-hard problem (GAREY; JOHNSON, 1990). However, its *linear program relaxation*, that ignores the integrality constraints and allows all variables to assume any real value, can be solved in polynomial time (KARMARKAR, 1984).

Definition 4 (Linear Relaxation). *The linear program relaxation or simply LP-relaxation of an integer program is the problem that arises by removing the requirement that variables are integer-valued. i.e., the variables can also assume real values.*

2.2.2 The Branch and Cut Algorithm

There are several techniques to optimally solve integer programs. Of particular interest to this thesis is the *Branch and Cut* (BC) algorithm (MITCHELL, 2002), that is an exact method for solving integer programming problems and arises from the combination of *cutting plane* methods and *Branch and Bound* algorithms.

Cutting plane methods iteratively refine a linear relaxation of an integer programming problem aiming to approximate the original problem solution. Initially the linear relaxation is solved and it is incrementally improved by adding discovered violated constraints called *cuts* (WOLSEY, 1998). Such methods were neglected for many years after conception due to slow convergence, but they have been used successfully to strengthen other general approaches, such as the *Branch and Bound* (MITCHELL, 2002).

The *Branch and Bound* algorithm solves an integer program in a *divide-and-conquer* manner, performing an implicit enumeration of the search space. An *LP-based Branch and Bound algorithm* solves several linear relaxations during the solving process to help in two important open decisions of the general algorithm: *how to branch* and *how to bound*. A simple way to decide *how to branch* is by choosing the integer variable with the most fractional value in the linear relaxation solution. There exist others more involved and effective branching strategies in the literature, such as the *pseudocost* and *strong* branching (ACHTERBERG; KOCH; MARTIN, 2005). The selected variable is used to create nodes and add subproblems to the *Branch and Bound tree*. Depending on the decision of *how to bound*, the algorithm is able to prune a significantly number of nodes from the tree. It can be tackled by solving linear relaxations of the nodes and comparing it to a global lower bound value. For minimization, if the solution to the linear relaxation is greater than the global lower bound, the node can be pruned since a better solution cannot be reached through it (MITCHELL, 2002; WOLSEY, 1998).

The BC algorithm is derived from the general Branch and Bound approach with the addition of cutting planes generated throughout the expansion of the Branch and Bound tree. This combination allows BC to efficiently solve many difficult IPs in practice (MÉNDEZ-DÍAZ; ZABALA, 2006; VASILYEV; KLIMENTOVA, 2010). Below we present the general BC algorithm for the minimization case:

Algorithm 2: The Branch and Cut Algorithm for minimization.

```
IP$^0$ is the original integer programming problem
$L := \{\text{IP}^0\}$ is the set of active nodes
$\bar{z} := +\infty$ is an upper bound on the optimal solution
while $L \neq \emptyset$ do
    choose and remove a problem $l^{\text{IP}}$ from $L$
    let $l^{\text{LP}}$ be the linear relaxation of $l^{\text{IP}}$
    search and add cutting planes to $l^{\text{LP}}$
    if $l^{LP}$ is feasible then
        $\underline{z}$ := the optimal objective value of $l^{\text{LP}}$
        $x$ := the primal solution of $l^{\text{LP}}$
        if $\underline{z} < \bar{z}$ then
            if $x$ is integral feasible then
                $\bar{z} := \underline{z}$
            else
                choose a variable $x_i$ from $x$ that is fractional
                choose an integer value $a$, for instance $a = \lfloor x_i \rfloor$
                add a new problem $l_1^{\text{IP}}$ with $x_i \leq a$ to $L$
                add a new problem $l_2^{\text{IP}}$ with $x_i \geq a + 1$ to $L$
return $\bar{z}$ and the associated primal solution $x$
```

2.2.3 Logic-Based Benders Decomposition

Classical Benders Decomposition (CBD) (BENDERS, 2005) is a technique to solve mixed integer linear programs by partitioning the original problem into two problems: i) a master programming problem which may be linear or not, and ii) a linear programming subproblem. Each problem uses a mutually exclusive subset of the variables. This decomposition aims to avoid the computation of the complete set of constraints for the original problem by designing two multi-step procedures that, in a finite number of steps, lead to a set of constraints for the optimal solution of the original problem. CBD can be applied to solve problems described as below (RAHMANIANI et al., 2017):

$$\begin{aligned} \text{minimize} \quad & f^T y + c^T x \\ \text{subject to} \quad & Ay = b, \\ & By + Dy = d, \\ & x \geq 0, \\ & y \in \mathcal{Z}_0^+. \end{aligned}$$

Variables $y \in \mathcal{R}^{n_1}$ can assume only positive integer values and must satisfy the constraint set $Ay = b$, where $A \in \mathcal{R}^{m_1 \times n_1}$ and $b \in \mathcal{R}^{m_1}$ are a given matrix and vector, respectively. The variables $x \in \mathcal{R}^{n_2}$ and y must satisfy the constraint set $By + Dx = d$, where $B \in \mathcal{R}^{m_2 \times n_1}$, $D \in \mathcal{R}^{m_2 \times n_2}$ and $d \in \mathcal{R}^{m_2}$. The objective function minimizes $f^T y + c^T x$, where $f \in \mathcal{R}^{n_1}$ and $c \in \mathcal{R}^{n_2}$. This model can be concisely rewritten as follows:

$$\min_{\bar{y} \in Y}\{f^T \bar{y} + \min_{x \geq 0}\{c^T : Dx = d - B\bar{y}\}\}.$$

Variables $\bar{y}$ denotes some given values for the variables y and originate from the set $Y = \{y | Ay = b, y \in \mathcal{Z}_0^+\}$. The inner minimization problem can be rewritten in terms of a maximization problem, using a set of dual variables κ associated with the constraint set $Dx = d - B\bar{y}$. The previous formulation can then be rewritten as:

$$\min_{\bar{y} \in Y}\{f^T \bar{y} + \max_{\kappa \in \mathcal{R}^{m_2}}\{\kappa^T (d - B\bar{y}) : \kappa^T D \leq c\}\}.$$

The feasible space F of the inner maximization problem is independent of the choice of $\bar{y}$. This inner maximization problem is called the *subproblem*. This problem can be either unbounded or feasible for any arbitrary choice of $\bar{y}$. If it is unbounded, given a set of extreme rays Q from F, we can add the following cuts indicating the infeasibility of the $\bar{y}$ solution:

$$r_q^T(d - B\bar{y}) \leq 0 \qquad \forall q \in Q.$$

If the solution is feasible, it is one of the extreme points $\kappa_e, e \in E$, where E is the set of extreme points of F. Considering Q and E, we can rewrite the model as follows:

$$\begin{aligned} \text{minimize}_{y,\eta} \qquad & f^T y + \eta \\ \text{subject to} \qquad & Ay = b, \\ \eta \geq \kappa_e^T(d - By) \qquad & \forall e \in E, \\ 0 \geq r_q^T(d - By) \qquad & \forall q \in Q, \\ & y \in \mathcal{Z}_0^+. \end{aligned}$$

This problem is denominated the *Benders Master Problem* (MP). The complete enumeration of the cuts from the sets Q and E is generally impractical. Therefore, the strategy adopted consists of iteratively generate relaxations of these cuts. The CBD approach repeatedly solves the MP that includes only a subset of the cuts from Q and E to obtain some value for $\bar{y}$. Then it solves the inner maximization problem, the *subproblem*, fixing the values of $\bar{y}$. If the subproblem is feasible and bounded, a cut from E is generated. If it is unbounded, a cut from Q is generated. In both cases, if the current solution violates the cuts, they are added to the current MP and the process repeats (RAHMANIANI et al., 2017).

As discussed by (HOOKER; OTTOSSON, 2003), the main idea of the CBD strategy is to assign some feasible values to variables and then try to find the best solution consistent with this assignment. During this process, the algorithm learns something about the quality of the solutions and uses this knowledge to reduce the number of feasible solutions generated before finding an optimal solution. In the literature, this strategy is referred to as *learning from one's mistakes*.

CBD works by deriving *Benders cuts* to remove uninteresting solutions. These cuts are formulated by solving the dual problem of the subproblem fixed with the assignment. Therefore, the subproblem must have an associated dual problem, such as in linear or non-linear programming problems (HOOKER; OTTOSSON, 2003).

The variables are partitioned in two mutually exclusive sets X and Y. Then, CBD fixes values for Y and defines a subproblem containing only variables in X. If the solution for the subproblem indicates that the values fixed for Y are infeasible, the solution of the dual problem associated to the subproblem is used to identify a number of other values of Y that are also infeasible. The next values used to fix Y must not have been excluded in previous iterations. Eventually the algorithm terminates after enumerating a few of the possible values of Y (HOOKER; OTTOSSON, 2003).

The solutions tested during the CBD solving process can be viewed as tuples (Y, z), in which Y are the fixed values and z is the objective function value. The Benders cuts that can be generated have the form $z \geq \beta(Y)$, where $\beta(Y)$ is a bound on the optimal solution value that depends on Y (HOOKER; OTTOSSON, 2003).

Logic-Based Benders Decomposition (LBBD) (HOOKER; OTTOSSON, 2003) generalizes the CBD approach by extending the concept of duality with the *inference dual*, whose solution is a proof of optimality written using some logical formalism. This generalization extends the class of problems in which CBD can be used. While LBBD can be applied to any class of optimization problems, a method that generates Benders cuts must be developed for each class.

Since the subproblem may be distinct from traditional linear or non-linear programming problems, LBBD allows to take advantage of the subproblem internal structure and specialized algorithms, that may not be possible in the CBD. An example is the application of LBBD as a framework for combining optimization and constraint programming proposed by HOOKER (2011).

A similar idea of CBD's *learns from mistakes* has been developed in the constraint programming field, in which it is referred to as *nogoods*. In constraint programming, when a partial solution to a problem cannot be completed to obtain a feasible solution, the solving algorithm can try to explain the motivating reason. The resulting explanation can be used to construct a constraint called *nogood* to invalidate a number of partial solutions, i.e., these solutions will not be generated later during the solving process. It is desirable to generate as strong as possible *nogood* constraints to invalidate as many as possible partial solutions (HOOKER; OTTOSSON, 2003).

The key aspect of the LBBD approach is the generalization of the dual problem used to construct the Benders cuts. As discussed by HOOKER; OTTOSSON (2003): *the dual must be definable for any type of subproblem, not just linear ones, and must provide an appropriate bound on the optimal value*. HOOKER; OTTOSSON (2003) formulate the dual observing that β can be directly inferred from the constraints of the subproblem, which is called the *inference dual*, that is the problem of *inferring a strongest possible bound β from the constraint set*. A solution to the *inference dual* problem provides a *logic-based Benders cut*, that is written using some logical formalism. HOOKER; OTTOSSON (2003) discuss the use of the *Propositional Satisfiability Problem* (SAT) to generate the *logic-based Benders cuts*, which is strongly related to the work of DAVIES et al. (2015) and this thesis.

2.3 The Operator-Counting Framework

2.3.1 Overview

Operator-counting (POMMERENING et al., 2014) unifies information from several different heuristics into a single integer program. The program contains an *operator-counting variable* Y_o for each operator $o \in O$, that counts the number of occurrences of operator o in some plan. Its objective function is to minimize the total cost of the selected operators while satisfying all *operator-counting constraints*. Operator-counting constraints and heuristics are defined below as in POMMERENING et al. (2014).

Definition 5 (**Operator-counting constraints**). *Let $\mathcal{Y}$ be a set of non-negative real-valued and integer variables, including an integer variable Y_o for each operator $o \in O$ along with any number of additional variables. Variables Y_o are called* operator-counting variables. *If π is a plan for planning task Π, we denote the number of occurrences of operator $o \in O$ in π with Y_o^π. A linear inequality over $\mathcal{Y}$ is called an* operator-counting constraint *for state s if for every s-plan there exists a feasible solution with $\mathsf{Y}_o = \mathsf{Y}_o^\pi$ for all $o \in O$. A* constraint set *for s is a set of operator-counting constraints where the only common variables between constraints are the operator-counting variables.*

Definition 6 (**Operator-counting IP/LP Heuristic**). *The* operator-counting integer program IP_C *for a set of operator-counting constraints C for state s is*

$$\begin{aligned}&\textit{minimize} \sum_{o \in O} c(o)\mathsf{Y}_o\\ &\textit{subject to } C,\\ &\mathsf{Y}_o \in \mathcal{Z}_0^+.\end{aligned}$$

The IP heuristic h_C^{IP} *is the objective value of* IP_C*, and the* LP heuristic h_C^{LP} *is the objective value of its linear relaxation. If the IP or LP is infeasible, the heuristic estimate is ∞.*

If π is a plan for Π then $\mathsf{Y}_o = \mathsf{Y}_o^\pi$ for all operators $o \in O$ is a solution for IP_C. Thus, the cost of an optimal plan π^* is an upper bound for the objective value of IP_C and therefore the IP heuristic is admissible. Since an integer solution for IP_C is also a solution for its linear relaxation, the LP heuristic is also admissible. Note also that adding more constraints to the model can only improve the heuristic estimates, possibly subject to a higher computational cost to optimize the model.

2.3.2 Sources of Operator-Counting Constraints

2.3.2.1 Action Landmarks

Action landmarks constraints (POMMERENING et al., 2014) are based on *disjunctive action landmarks* for the planning task. These landmarks encode necessary conditions for all plans by stating that *at least one of a set of operators must occur*. Computing all action landmarks given a planning task is a PSPACE-complete problem (HOFFMANN; PORTEOUS; SEBASTIA, 2004). However, some subset of action landmarks can be efficiently computed using the LMCut algorithm (HELMERT; DOMSHLAK, 2009) or with a *delete-relaxed exploration* (HOFFMANN; PORTEOUS; SEBASTIA, 2004).

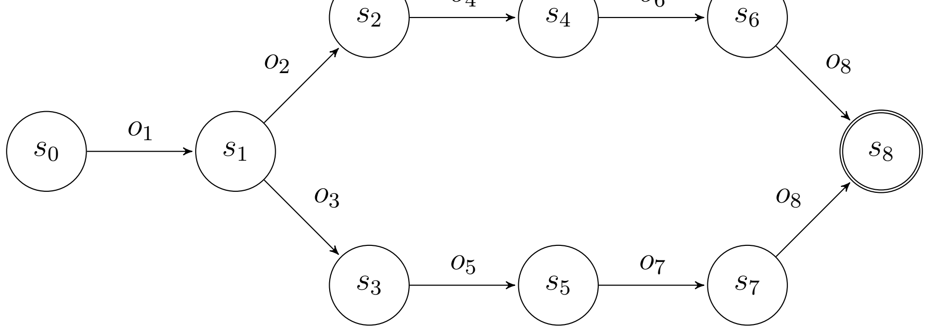
In the literature, these constraints are frequently applied to strengthen other heuristics, because they can enforce lower bounds on the number of times some operators are applied. The operator-counting constraint corresponding to a disjunctive action landmark is a set of operators for a state s such that every s-plan must contain at least one operator from the disjunctive action landmark:

Definition 7. *The operator-counting constraint corresponding to a disjunctive action landmark $L \subseteq O$ for a state s of a planning task Π is*

$$\sum_{o \in L} \mathsf{Y_o} \geq 1.$$

Figure 2.4 illustrates an example state-space with goal state s_8 where o_1 and o_8 are action landmarks and $\mathsf{Y}_{\mathsf{o}_1} \geq 1$ and $\mathsf{Y}_{\mathsf{o}_8} \geq 1$ are valid operator-counting constraints. Additionally, the operator-counting constraints $\mathsf{Y}_{\mathsf{o}_2} + \mathsf{Y}_{\mathsf{o}_3} \geq 1$, $\mathsf{Y}_{\mathsf{o}_4} + \mathsf{Y}_{\mathsf{o}_5} \geq 1$ and $\mathsf{Y}_{\mathsf{o}_6} + \mathsf{Y}_{\mathsf{o}_7} \geq 1$ can also be derived from valid disjunctive action landmarks.

Figure 2.4: Example state-space for action landmarks constraints.



2.3.2.2 Post-hoc Optimization

Post-hoc Optimization (POMMERENING; RÖGER; HELMERT, 2013) is a heuristic that exploits the fact that *sometimes the set of operators that effectively contribute to a heuristic estimate is known*. It is also a framework that can combine information from several admissible heuristics. The most prominent family of heuristics used in post-hoc optimization is the *pattern database heuristics* (PDB) (CULBERSON; SCHAEFFER, 1998; EDELKAMP, 2014).

PDB heuristics projects the original planning task to a subset of the original variables $P \subseteq V$ called *pattern*. This projection changes the semantics of the states and operators, resulting in a significantly easier abstract task that only accounts for variables $v \in P$ and ignore others. The entire abstract state-space for the projection is explicitly generated and the exact plan costs for all abstract states are computed and stored in a table. To compute the heuristic estimate during search, a state s is mapped to an abstract state s' and the table is consulted, returning the exact plan cost for s'.

All PDB heuristics are admissible, since plan costs for abstract states can never be greater than the plan costs for original states. We can use the heuristic estimate of some PDB heuristic h^P for some pattern P, as a lower bound on the total combined cost of some subset of operators relevant to P. Fortunately, it is easy to compute the set of *relevant operators* R to pattern P in PDB heuristics: only operators that affect some variable $v \in P$, i.e., change its value, effectively contribute to the heuristic estimate. Below we define the operator-counting constraint associated to post-hoc optimization heuristics:

Definition 8. *The operator-counting constraint corresponding to the post-hoc optimization for some state s and a PDB heuristic h^P for some pattern P with a set of relevant operators R is*

$$\sum_{o \in R} c(o)\mathsf{Y}_\mathsf{o} \geq h^P(s).$$

To illustrate these operator-counting constraints, consider the state-space illustrated in Figure 2.5. This state-space corresponds to a simple planning task with variables $\mathcal{V} = \{x, y\}$ and domains $D(x) = D(y) = \{0, 1, 2, 3\}$. Since each operator only induces one state transition in this task, they are presented directly in the figure. For instance, preconditions and effects of operator o_1 are $\langle x = 0, y = 0\rangle$ and $\langle x := 1, y := 1\rangle$, respectively. The initial state is s_0 and s_5 is the goal. The cost of operator o_i is i. In the figure, the notation $\langle i, j\rangle$ inside nodes denotes that variable x assumes the value $i \in D(x)$ and y assumes the value $j \in D(y)$.

Figure 2.5: Example state-space for post-hoc optimization constraints.

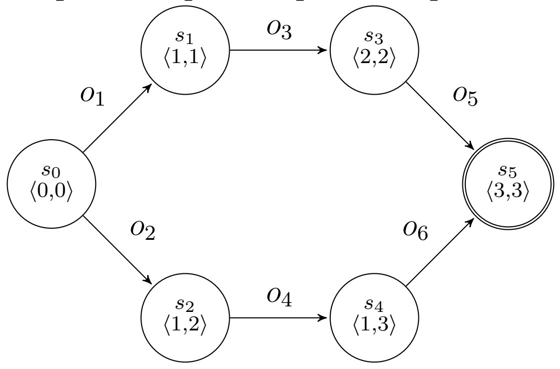
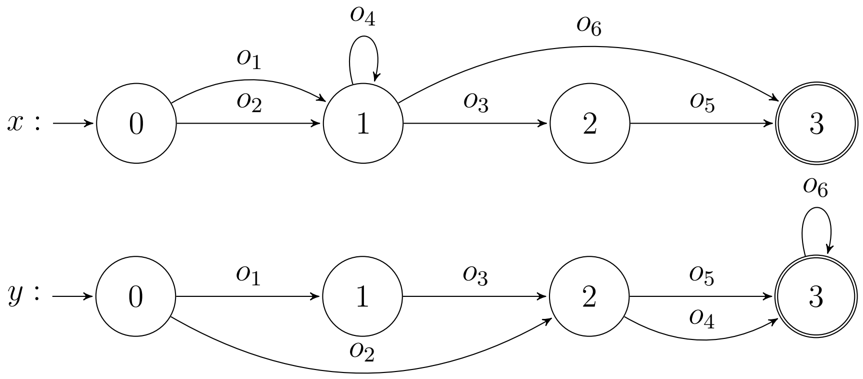


Figure 2.6 presents the DTG for atomic projections over the variables of this task: $P_1 = \{x\}$ and $P_2 = \{y\}$. The heuristic estimate for the initial state s_0 for pattern P_1 is $h^{P_1}(s_0) = 7$ since $\langle o_1, o_6 \rangle$ is a cost-optimal plan for the projected task. Analogously, $h^{P_2}(s_0) = 6$ since $\langle o_2, o_4 \rangle$ is a cost-optimal plan for the projected task using pattern P_2.

Figure 2.6: DTG for atomic projections on variables x and y.



The operator-counting constraints generated from these PDB heuristics and post-hoc optimization are presented below. For $h^{P_1}(s_0)$, operator o_4 is irrelevant because it does not change the value of x. Similarly, o_6 is not relevant for $h^{P_2}(s_0)$ since it does not change variable y.

$$Y_{o_1} + 2Y_{o_2} + 3Y_{o_3} + 5Y_{o_5} + 6Y_{o_6} \geq 7, \qquad \text{for } h^{P_1}(s_0),$$

$$Y_{o_1} + 2Y_{o_2} + 3Y_{o_3} + 4Y_{o_4} + 5Y_{o_5} \geq 6, \qquad \text{for } h^{P_2}(s_0).$$

2.3.2.3 State Equation and Network Flow

State equation constraints (BONET, 2013) are derived from the h^{SEQ} heuristic and obtained by mapping planning tasks to *Petri nets* and computing *net changes* for all possible atoms of the task. The net change $\Delta^s_{\langle v=x\rangle}$ of an atom $\langle v = x\rangle$ and some state s expresses the change in the atom's truth table from state s to some goal state s_*, i.e., $\Delta^s_{\langle v=x\rangle}$ is $+1$ if the atom $\langle v = x\rangle$ becomes true at s_*, -1 if it becomes false and 0 if it is unchanged. The net change $\Delta^s_{\langle v=x\rangle}$ can be precisely denoted as:

$$\Delta^s_{\langle v=x\rangle} = \begin{cases} +1, & \text{if } s[v] \neq x \text{ and } s_*[v] = x, \\ -1, & \text{if } s[v] = x \text{ and } s_*[v] \neq x, \\ 0, & \text{otherwise.} \end{cases}$$

Based on the net changes $\Delta^s_{\langle v=x\rangle}$, it is possible to construct a set of *state equation constraints*, one for each atom $\langle v = x\rangle$ of the planning task. These constraints express balance for variables values considering the goals and the current state s, and can even describe dependencies between operators. It is possible to model situations in which the application of an operator o_1 adds atoms required by another operator o_2, or when atoms are produced and consumed by symmetric but necessary operators. We say that an operator is a producer of $\langle v = x\rangle$ if makes it true. Conversely, an operator is a consumer of $\langle v = x\rangle$ if makes it false. Next we define the operator-counting constraints derived from state equation heuristics.

Definition 9. *Let $Prod(\langle v = x\rangle)$ and $Cons(\langle v = x\rangle)$ be the sets of producers and consumers of atom $\langle v = x\rangle$, respectively. Then, the operator-counting constraint corresponding to the state equation heuristic for atom $\langle v = x\rangle$ and state s is*

$$\sum_{o \in Prod(\langle v=x\rangle)} \mathsf{Y}_o - \sum_{o \in Cons(\langle v=x\rangle)} \mathsf{Y}_o \geq \Delta^s_{\langle v=x\rangle}.$$

To illustrate the state equation constraints, consider a planning task with variables $\mathcal{V} = \{x, y\}$ with $D(x) = D(y) = \{1, 2, 3\}$, unit-cost operators $O = \{o_1 = \langle x = 1 \wedge y = 2; x := 1 \wedge y := 3\rangle, o_2 = \langle x = 2 \wedge y = 2; x := 3 \wedge y := 3\rangle, o_3 = \langle x = 3 \wedge y = 3; x := 2 \wedge y := 3\rangle, o_4 = \langle x = 1 \wedge y = 3; x := 2 \wedge y := 2\rangle, o_5 = \langle x = 2 \wedge y = 2; x := 3 \wedge y := 1\rangle, o_6 = \langle x = 1 \wedge y = 2; x := 3 \wedge y := 2\rangle\}$, initial state $s_0 = \langle x = 1 \wedge y = 2\rangle$ and goal $s_* = \langle x = 3 \wedge y = 1\rangle$. The net changes $\Delta^s_{\langle v=x\rangle}$ considering $s = s_0$ are:

$$\Delta^s_{v=x} = \begin{cases} +1, & \text{if } \langle v = x \rangle \in \{\langle x = 3 \rangle, \langle y = 1 \rangle\}, \\ -1, & \text{if } \langle v = x \rangle \in \{\langle x = 1 \rangle, \langle y = 2 \rangle\}, \\ 0, & \text{if } \langle v = x \rangle \in \{\langle x = 2 \rangle, \langle y = 3 \rangle\}. \end{cases}$$

The producers and consumers sets for this illustrative planning task are:

$$\begin{aligned} &\mathit{Prod}(\langle x = 1 \rangle) = \{o_1\}; && \mathit{Cons}(\langle x = 1 \rangle) = \{o_1, o_4, o_6\}; \\ &\mathit{Prod}(\langle x = 2 \rangle) = \{o_3, o_4\}; && \mathit{Cons}(\langle x = 2 \rangle) = \{o_2, o_5\}; \\ &\mathit{Prod}(\langle x = 3 \rangle) = \{o_2, o_5, o_6\}; && \mathit{Cons}(\langle x = 3 \rangle) = \{o_3\}; \\ &\mathit{Prod}(\langle y = 1 \rangle) = \{o_5\}; && \mathit{Cons}(\langle y = 1 \rangle) = \emptyset; \\ &\mathit{Prod}(\langle y = 2 \rangle) = \{o_4, o_6\}; && \mathit{Cons}(\langle y = 2 \rangle) = \{o_1, o_2, o_5, o_6\}; \\ &\mathit{Prod}(\langle y = 3 \rangle) = \{o_1, o_2, o_3\}; && \mathit{Cons}(\langle y = 3 \rangle) = \{o_3, o_4\}. \end{aligned}$$

Finally, the operator-counting constraints derived from the state equation for state s_0 are presented below. Note that an operator that is simultaneously a producer and a consumer of an atom cancels out, for instance, operator o_1 in the constraint for atom $\langle x = 1 \rangle$.

$$\begin{aligned} \mathsf{Y}_{\mathsf{o}_1} - \mathsf{Y}_{\mathsf{o}_1} - \mathsf{Y}_{\mathsf{o}_4} - \mathsf{Y}_{\mathsf{o}_6} &\geq -1, && \text{for } \langle x = 1 \rangle, \\ \mathsf{Y}_{\mathsf{o}_3} + \mathsf{Y}_{\mathsf{o}_4} - \mathsf{Y}_{\mathsf{o}_2} - \mathsf{Y}_{\mathsf{o}_5} &\geq \ \ 0, && \text{for } \langle x = 2 \rangle, \\ \mathsf{Y}_{\mathsf{o}_2} + \mathsf{Y}_{\mathsf{o}_5} + \mathsf{Y}_{\mathsf{o}_6} - \mathsf{Y}_{\mathsf{o}_3} &\geq +1, && \text{for } \langle x = 3 \rangle, \\ \mathsf{Y}_{\mathsf{o}_5} &\geq +1, && \text{for } \langle y = 1 \rangle, \\ \mathsf{Y}_{\mathsf{o}_4} + \mathsf{Y}_{\mathsf{o}_6} - \mathsf{Y}_{\mathsf{o}_1} - \mathsf{Y}_{\mathsf{o}_2} - \mathsf{Y}_{\mathsf{o}_5} - \mathsf{Y}_{\mathsf{o}_6} &\geq -1, && \text{for } \langle y = 2 \rangle, \\ \mathsf{Y}_{\mathsf{o}_1} + \mathsf{Y}_{\mathsf{o}_2} + \mathsf{Y}_{\mathsf{o}_3} - \mathsf{Y}_{\mathsf{o}_3} - \mathsf{Y}_{\mathsf{o}_4} &\geq \ \ 0, && \text{for } \langle y = 3 \rangle. \end{aligned}$$

As POMMERENING (2017) discusses, *network flow* and *state equation* are very similar and the former can be seen as a generalization of the latter. The main difference is that network flow constraints are constructed considering abstract states and transitions and state equation constraints consider atoms and operators. Also, network flow constraints use one variable for each abstract transition and state equation constraints use one variable for each operator. Due to this similarity, we only discuss and detail the constraints for state equation.

2.3.2.4 Optimal Delete-Relaxation

Heuristics based on *delete relaxation* (MCDERMOTT, 1999; BONET; GEFFNER, 2001; HOFFMANN; NEBEL, 2001) change the semantics of planning tasks by allowing variables to maintain past values, discarding the *delete effects* from operators. Consequently, they produce relaxed tasks that are significantly easier than the original, since each operator is applied at most once in a feasible plan. In spite of that, finding cost optimal plans for delete-relaxed tasks is still an NP-hard problem (BYLANDER, 1997).

IMAI; FUKUNAGA (2014) introduce an IP formulation compatible with the operator-counting framework to compute h^+, the *optimal delete relaxation heuristic*. This formulation accounts for operators and atoms that are part of the optimal relaxed plan, as well as the order in which operators are executed and atoms are first reached. Below, we introduce the IP model variables, where F denotes the set of all atoms, π^+ is a delete-relaxed optimal plan, and $add(o)$ is the set of atoms added by the application of operator o:

$\forall f \in s_0, I_f \in \{0,1\} :\ I_f = 1$ if $f \in s_0$;
$\forall f \in F, R_f \in \{0,1\} :\ R_f = 1$ if f is reached by π^+;
$\forall o \in O, U_o \in \{0,1\} :\ U_o = 1$ if o is part of π^+;
$\forall o \in O, \forall f \in add(o), A_{o,f} \in \{0,1\} :\ A_{o,f} = 1$ if f is reached by π^+ and o adds f first;
$\forall f \in F, T_f \in \{0,\ldots,|O|\} :\ T_f = t$ when f is reached by π^+ and π^+_{t-1} adds f first;
$\forall o \in O, T_o \in \{0,\ldots,|O|-1\} :\ T_o = t$ when is the t-th applied operator in π^+.
$T_o = |O| - 1$ when o is not applied in π^+.

Finally, IMAI; FUKUNAGA (2014) introduce a set of counting and net change constraints. The authors experimentally show that these constraints improve the heuristic estimate of h^+, originating a new heuristic function. However, we do not use these constraints since the operator-counting heuristic usually dominates h^+. Next, we introduce the base model constraints without the enhancements presented in the paper. We add the *linking constraints* defined by Equation (7) to link the delete relaxation variables U_o to the operator-counting variables $\mathsf{Y_o}$.

Definition 10. *The operator-counting constraints corresponding to the delete relaxation h^+ heuristic are*

$$\forall f \in s_*, R_f = 1, \tag{1}$$

$$\forall f \in F, I_f + \sum_{o \in O \mid f \in add(o)} A_{o,f} \geq R_f, \tag{2}$$

$$\forall o \in O, \forall f \in add(o), U_o \geq A_{o,f}, \tag{3}$$

$$\forall o \in O, \forall f \in pre(o), R_f \geq U_o, \tag{4}$$

$$\forall o \in O, \forall f \in pre(o), T_f \leq T_o, \tag{5}$$

$$\forall o \in O, \forall f \in add(o), T_o + 1 \leq T_f + (|O| + 1)(1 - A_{o,f}), \tag{6}$$

$$\mathsf{Y_o} \geq U_o. \tag{7}$$

To illustrate the operator-counting constraints derived from the optimal delete relaxation heuristic h^+, consider a planning task Π_{h^+} with variables $\mathcal{V} = \{x, y\}$ with $D(x) = \{1, 2\}$ and $D(y) = \{0, 1, 2\}$, unit-cost operators $O = \{o_1 = \langle x = 1 \wedge y = 1; y := 0\rangle, o_2 = \langle x = 2 \wedge y = 2; y := 0\rangle, o_3 = \langle x = 1 \wedge y = 0; y := 1\rangle, o_4 = \langle x = 2 \wedge y = 0; y := 2\rangle, o_5 = \langle x = 1; x := 2\rangle, o_6 = \langle x = 2; x := 1\rangle\}$, initial state $s_0 = \langle x = 1 \wedge y = 1\rangle$ and goal $s_* = \langle x = 1 \wedge y = 2\rangle$. The optimal plan for Π_{h^+} is $\langle o_1, o_5, o_4, o_6\rangle$ with cost 4.

We use the task $\Pi^+_{h^+}$ resultant from the delete-relaxation of the original task Π_{h^+} to construct the respective operator-counting constraints. This relaxation causes changes to the variables' domains $D(x)$ and $D(y)$ to support sets of values. Effectively, $D(x) = \{\{1\}, \{2\}, \{1, 2\}\}$ and $D(y) = \{\{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}\}$. The semantics of operators are changed so they only add values to the variables when applied and are applicable if the variables contains the value specified in preconditions: $O = \{o_1 = \langle 1 \in x \wedge 1 \in y; y := y \cup \{0\}\rangle, o_2 = \langle 2 \in x \wedge 2 \in y; y := y \cup \{0\}\rangle, o_3 = \langle 1 \in x \wedge 0 \in y; y := y \cup \{1\}\rangle, o_4 = \langle 2 \in x \wedge 0 \in y; y := y \cup \{2\}\rangle, o_5 = \langle 1 \in x; x := x \cup \{2\}\rangle, o_6 = \langle 2 \in x; x := x \cup \{1\}\rangle\}$. The initial state is $s_0 = \langle x := \{1\} \wedge y := \{1\}\rangle$ and goal $s_* = \langle 1 \in x \wedge 2 \in y\rangle$. The optimal plan for $\Pi^+_{h^+}$ is $\langle o_1, o_5, o_4\rangle$ with cost 3.

Next, we present the operator-counting constraints generated for state s_0 considering Definition 10. For Equation (1) we have:

$$R_{\langle x=1\rangle} = 1; \qquad R_{\langle y=2\rangle} = 1.$$

For Equation (2):

$$1 + A_{o_6,\langle x=1\rangle} \geq R_{\langle x=1\rangle}; \qquad 1 + A_{o_3,\langle y=1\rangle} \geq R_{\langle y=1\rangle};$$

$$0 + A_{o_5,\langle x=2\rangle} \geq R_{\langle x=2\rangle}; \qquad 0 + A_{o_4,\langle y=2\rangle} \geq R_{\langle y=2\rangle};$$

$$0 + A_{o_1,\langle y=0\rangle} + A_{o_2,\langle y=0\rangle} \geq R_{\langle y=0\rangle}.$$

For Equation (3):

$$U_{o_1} \geq A_{o_1,\langle y=0\rangle}; \qquad U_{o_2} \geq A_{o_2,\langle y=0\rangle}; \qquad U_{o_3} \geq A_{o_3,\langle y=1\rangle};$$

$$U_{o_4} \geq A_{o_4,\langle y=2\rangle}; \qquad U_{o_5} \geq A_{o_5,\langle x=2\rangle}; \qquad U_{o_6} \geq A_{o_6,\langle x=1\rangle}.$$

For Equation (4):

$$R_{\langle x=1\rangle} \geq U_{o_1}; \qquad R_{\langle y=0\rangle} \geq U_{o_3};$$

$$R_{\langle y=1\rangle} \geq U_{o_1}; \qquad R_{\langle x=2\rangle} \geq U_{o_4};$$

$$R_{\langle x=2\rangle} \geq U_{o_2}; \qquad R_{\langle y=0\rangle} \geq U_{o_4};$$

$$R_{\langle y=2\rangle} \geq U_{o_2}; \qquad R_{\langle x=1\rangle} \geq U_{o_5};$$

$$R_{\langle x=1\rangle} \geq U_{o_3}; \qquad R_{\langle x=2\rangle} \geq U_{o_6}.$$

For Equation (5):

$$T_{\langle x=1\rangle} \leq T_{o_1}; \qquad T_{\langle y=0\rangle} \leq T_{o_3};$$

$$T_{\langle y=1\rangle} \leq T_{o_1}; \qquad T_{\langle x=2\rangle} \leq T_{o_4};$$

$$T_{\langle x=2\rangle} \leq T_{o_2}; \qquad T_{\langle y=0\rangle} \leq T_{o_4};$$

$$T_{\langle y=2\rangle} \leq T_{o_2}; \qquad T_{\langle x=1\rangle} \leq T_{o_5};$$

$$T_{\langle x=1\rangle} \leq T_{o_3}; \qquad T_{\langle x=2\rangle} \leq T_{o_6}.$$

For Equation (6):

$$\begin{aligned}
&T_{o_1} + 1 \leq T_{\langle y=0\rangle} + 7(1 - A_{o_1,\langle y=0\rangle}); && T_{o_4} + 1 \leq T_{\langle y=2\rangle} + 7(1 - A_{o_4,\langle y=2\rangle});\\
&T_{o_2} + 1 \leq T_{\langle y=0\rangle} + 7(1 - A_{o_1,\langle y=0\rangle}); && T_{o_5} + 1 \leq T_{\langle x=2\rangle} + 7(1 - A_{o_5,\langle x=2\rangle});\\
&T_{o_3} + 1 \leq T_{\langle y=1\rangle} + 7(1 - A_{o_3,\langle y=1\rangle}); && T_{o_6} + 1 \leq T_{\langle x=1\rangle} + 7(1 - A_{o_6,\langle x=1\rangle}).
\end{aligned}$$

Finally, for the linking constraints from Equation (7) we have:

$$\begin{aligned}
&\mathsf{Y}_{o_1} \geq U_{o_1}; && \mathsf{Y}_{o_4} \geq U_{o_4};\\
&\mathsf{Y}_{o_2} \geq U_{o_2}; && \mathsf{Y}_{o_5} \geq U_{o_5};\\
&\mathsf{Y}_{o_3} \geq U_{o_3}; && \mathsf{Y}_{o_6} \geq U_{o_6}.
\end{aligned}$$

2.3.3 Operator Counts

An *operator counts* $\mathcal{C} : O \rightarrow \mathcal{Z}_0^+$ is a function that assigns to each operator $o \in O$ the integer count Y_o obtained from the primal solution of the operator-counting IP$_C$ for state s. These counts can be seen as *an estimate on how often each operator is used in a feasible solution* for IP$_C$. We interpret operator counts as *multisets of operators* and say that $\mathcal{C}_1 \subseteq \mathcal{C}_2$ if $\mathcal{C}_1(o) \leq \mathcal{C}_2(o)$ for all $o \in O$.

For instance, suppose that we have a planning task with four operators o_1, o_2, o_3 and o_4. Then we solve the operator-counting heuristic corresponding to this planning task with some set of operator-counting constraints, obtaining the primal solution $\mathsf{Y}_{o_1} = 2$, $\mathsf{Y}_{o_2} = 1$, $\mathsf{Y}_{o_3} = 4$ and $\mathsf{Y}_{o_4} = 0$. Then the operator counts corresponding to this primal solution is $\mathcal{C} = \{o_1 \mapsto 2, o_2 \mapsto 1, o_3 \mapsto 4\}$. We only show counts for non-zero operators.

2.3.4 Generalized Landmarks

DAVIES et al. (2015) introduced a generalization of disjunctive action landmark constraints that *counts the number of occurrences of operators*. A *generalized landmark constraint* (GLC) is a disjunction of binary variables called *bounds literals*. A *bounds literal* $[\mathsf{Y}_o \geq k_o]$ is true if there are at least k_o occurrences of operator o in the solution of the IP$_C$. This generalization is compatible with operator-counting constraints and can be used to express constraints of the form $[\mathsf{Y}_{o_1} \geq k_{o_1}] + \ldots + [\mathsf{Y}_{o_n} \geq k_{o_n}] \geq 1$. To satisfy this constraint, at least one of the bounds literals must be true, enabling to encode necessary conditions of feasible plans directly over the operator counts.

Definition 11 (Generalized Landmark Constraint). *A generalized landmark constraint L for a state s and a planning task Π, corresponding to a disjunctive action landmark $A \subseteq O \times \mathcal{Z}^+$, is defined as*

$$\sum_{\forall \langle o,k \rangle \in A} [\mathsf{Y}_o \geq k] \geq 1.$$

Domain constraints are used to link bounds literals with the operator-counting variables Y_o: we have for all $k \geq 1$:

$$[\mathsf{Y}_o \geq k] \leq [\mathsf{Y}_o \geq k-1]\,; \tag{1}$$

$$\mathsf{Y}_o \geq \sum_{i=1}^{k} [\mathsf{Y}_o \geq i]\,; \tag{2}$$

$$\mathsf{Y}_o \leq M\,[\mathsf{Y}_o \geq k] + k - 1. \tag{3}$$

Constraint (1) ensures that bound $[\mathsf{Y}_o \geq k]$ is only valid when the next smallest bound $[\mathsf{Y}_o \geq k-1]$ is; (2) ensures that the total number of valid bounds literals for operator o is a lower bound on the number of operators Y_o; and (3) ensures that bound $[\mathsf{Y}_o \geq k]$ is set when $\mathsf{Y}_o \geq k$. Combined, (2) and (3) guarantee that Y_o is the number of occurrences of operator o.

For instance, if we had a disjunctive action landmark with counts $A = \{\langle o_1, 2\rangle, \langle o_2, 1\rangle\}$, the corresponding GLC would be $L = [o_1 \geq 2] + [o_2 \geq 1] \geq 1$. Consequently, the following domains constraints would have to be included in the model:

$$\begin{aligned}
&[\mathsf{Y}_{o_1} \geq 2] \leq [\mathsf{Y}_{o_1} \geq 1]\,; \\
&[\mathsf{Y}_{o_1} \geq 1] \leq [\mathsf{Y}_{o_1} \geq 0]\,; \\
&\quad \mathsf{Y}_{o_1} \geq [\mathsf{Y}_{o_1} \geq 1] + [\mathsf{Y}_{o_1} \geq 2]\,; \\
&\quad \mathsf{Y}_{o_1} \leq M\,[\mathsf{Y}_{o_1} \geq 2] + 1; \\
&\quad \mathsf{Y}_{o_1} \leq M\,[\mathsf{Y}_{o_1} \geq 1]\,.
\end{aligned}
\qquad
\begin{aligned}
&[\mathsf{Y}_{o_2} \geq 1] \leq [\mathsf{Y}_{o_2} \geq 0]\,; \\
&\quad \mathsf{Y}_{o_2} \geq [\mathsf{Y}_{o_2} \geq 1]\,; \\
&\quad \mathsf{Y}_{o_2} \leq M\,[\mathsf{Y}_{o_2} \geq 1]\,.
\end{aligned}$$

3 PLANNING WITH LOGIC-BASED BENDERS DECOMPOSITION

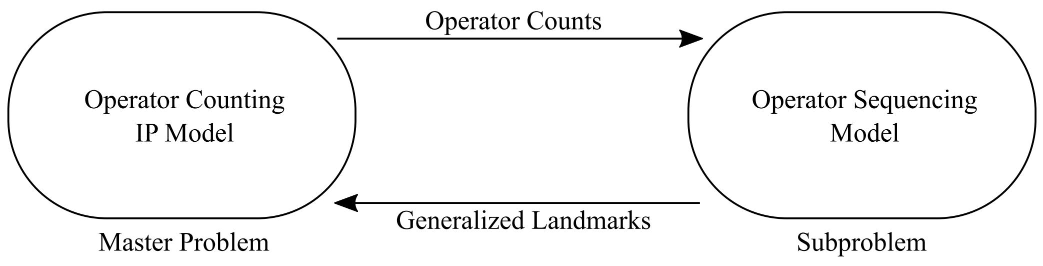
3.1 Overview

POMMERENING et al. present several heuristic functions to guide A* that are based on linear programming. Even though linear programs are polynomial-time solvable, one must use them as heuristic functions carefully, since the state-space expanded by A* grows exponentially in the number of states, according to the planning task specification, and the cost of solving a linear program to compute the heuristic for each state is not negligible. Of particular interest among these heuristics is the *operator-counting framework* discussed before, that estimates the optimal plan cost from the number of times the operators are used in some feasible plan.

Usually only the *objective value* of the linear relaxation of the operator-counting IP is used to guide the search to solve planning tasks. However, DAVIES et al. (2015) note that the *operator counts* associated to the IP's *primal solution* contains an interesting information that can be used to solve the problem. Specifically, it can be interpreted as a possibly *incomplete and unordered plan* to the planning task, with some missing necessary operators. It originates the idea to incrementally search for missing operators until a complete and ordered plan is found.

DAVIES et al. (2015) propose an LBBD to decompose the process of solving planning tasks into two related problems: i) a *master problem* that solves IP$_C$: a relaxation of the original planning task, which generates operator counts $\mathcal{C}$, and ii) a *subproblem* which tries to sequence $\mathcal{C}$, constructing and returning a violated constraint on failure. The main idea consists of incrementally strengthening the master problem relaxation with some learned knowledge about the infeasibility of its current solution. These constraints should be as informative as possible to decrease the number of total iterations between the master and the subproblem. The process stops when the BC algorithm from the master proves the optimality of the current incumbent plan. Figure 3.1 illustrates the overall process.

Figure 3.1: LBBD to cost-optimal planning (adapted from DAVIES et al. (2015)).



This decomposition establishes an interface between operator-counting heuristics and operator counts sequencing procedures. In the next section we discuss how DAVIES et al. (2015) solve the operator counts sequencing subproblem.

3.2 Operator Counts Sequencing with SAT

The solver *OpSeq* introduced by DAVIES et al. (2015) applies a SAT model that encodes the planning task limited to an operator counts $\mathcal{C}$ as a formula in the conjunctive normal form. They use this model to solve the sequencing operator counts subproblem. If the formula is satisfiable, *OpSeq* can directly extract a plan. If the operator counts does not correspond to a plan i.e., if the formula is not satisfiable, *OpSeq* uses assumptions to generate an explanation of its infeasibility. The assumptions are special variables relating to the current operator counts. The generated explanation is a disjunction of negated assumptions that can be directly translated to a generalized landmark constraint (GLC) and added to the master problem.

OpSeq does not solve the entire operator-counting IP_C at each step of the LBBD. Instead, it solves the linear relaxation and obtains a valid operator counts by rounding up the primal solution values to the nearest integers, only if its cardinality and objective value are within 20% of the fractional operator counts. Consequently, *OpSeq* is able to generate violated constraints that also remove fractional solutions.

Most IP solvers support the definition of *control callbacks* to dynamically interact with the BC algorithm. *OpSeq* uses this mechanism to heuristically construct plans using the round-up method and to add constraints to strengthen linear relaxations or invalidate integer solutions that cannot be sequenced to a feasible plan. These callbacks also allow *OpSeq* to call the sequencing procedure whenever an operator counts is available in a node of the BC tree, independently if it comes from a fractional or an integral primal solution.

OpSeq follows the main idea of planning using LBBD. The process begins by using a BC to solve the IP$_C$. If BC finds an integer solution it calls the SAT sequencing procedure to try to sequence the corresponding operator counts. If BC finds a relaxed solution it obtains a valid operator counts by rounding up the primal solution values to the nearest integers, and sequences it if it is within the 20% threshold or ignores it otherwise. If this operator counts is sequencable, *OpSeq* informs the BC that a new solution has been found. This process continues until BC proves that one of the found plans is optimal.

A restart occurs when a generated GLC contains more than one *weak bounds literal*. It results from a missing bounds literal $[v \geq k]$ that has not been allocated yet, for some $v \in \{Y_o,\ \forall o \in O\} \cup \{Y_f\}$ and some value of k. During the BC algorithm, *OpSeq* solves this issue by adding the weak bounds literal v/k corresponding to the missing bounds literal $[v \geq k]$. Initially, *OpSeq* allocates bounds literals up to $k = 2$. If a restart occurs, the IP solving process is reinitialized and the weak bounds literals are replaced with new conventional bounds literals allocated.

To illustrate this situation, assume that we have a planning task with three operators o_1, o_2 and o_3, with pre-allocated bounds literals up to $k = 1$. Also, o_1 and o_2 are action landmarks for this task and the IP contains the corresponding constraints $[Y_{o_1} \geq 1] \geq 1$ and $[Y_{o_2} \geq 1] \geq 1$. If we solve the linear relaxation of this IP, we obtain the primal solution $P_0 = \langle Y_{o_1} = 1, Y_{o_2} = 1, Y_{o_3} = 0, [Y_{o_1} \geq 1] = 1, [Y_{o_1} \geq 0] = 1, [Y_{o_2} \geq 1] = 1, [Y_{o_2} \geq 0] = 1, [Y_{o_3} \geq 1] = 0, [Y_{o_3} \geq 0] = 1\rangle$. Now, suppose that the corresponding operator counts is not sequencable and the GLC $L_1 = [Y_{o_1} \geq 2] + [Y_{o_3} \geq 1] \geq 1$ is learned. Since the bounds literal $[Y_{o_1} \geq 2]$ is not allocated in the model, the constraint that is effectively added is $\frac{1}{2}Y_{o_1} + [Y_{o_3} \geq 1] \geq 1$. Solving the linear relaxation of the model with this constraint would return a primal solution $P_1 = \langle Y_{o_1} = 1, Y_{o_2} = 1, Y_{o_3} = 0.5, [Y_{o_1} \geq 1] = 1, [Y_{o_1} \geq 0] = 1, [Y_{o_2} \geq 1] = 1, [Y_{o_2} \geq 0] = 1, [Y_{o_3} \geq 1] = 0.5, [Y_{o_3} \geq 0] = 1\rangle$. In this case, since $P_1 \neq P_0$, a GLC with only one weak bounds literal still invalidates the current solution P_0. On the other hand, suppose that instead we learn the GLC $L_2 = [Y_{o_1} \geq 2] + [Y_{o_2} \geq 2] + [Y_{o_3} \geq 1] \geq 1$. Since the bounds literals $[Y_{o_1} \geq 2]$ and $[Y_{o_2} \geq 2]$ are not allocated, the constraint would be $\frac{1}{2}Y_{o_1} + \frac{1}{2}Y_{o_2} + [Y_{o_3} \geq 1] \geq 1$. Solving the linear relaxation of the model with this constraint would return a primal solution $P_2 = P_0$, therefore, constraint L_2 does not invalidate the solution P_0.

The SAT model constructed by *OpSeq* is composed of layers and only one operator can be applied in each layer. *OpSeq* uses the variable Y_T to limit the total number of layers, computed as the total number of operators in the operator counts. It constructs a set of *assumptions* and informs the solver to use them. On failure, the SAT model is able to construct a GLC based on these assumptions, explaining *why the operator counts is not sequencable*. This constraint is derived by the *Conflict-Directed Clause Learning* algorithm (MARQUES-SILVA; SAKALLAH, 1999) implemented in SAT solvers, that backtracks until it reaches to the assumptions that cause the formula's unsatisfiability.

Below we present the SAT formulation proposed by DAVIES et al. (2015) for each layer l, where $v =_l x$ denotes that variable v holds the value x in layer l; o_l that operator o occurs in layer l; $L = Y_T = \sum_{o \in O} \mathcal{C}(o)$ is the total number of layers; $\leq_k (S)$ denotes the at-most-k constraint that enforces that k or fewer literals from a set S are simultaneously true (SINZ, 2005); and $prod(\langle v = x \rangle)$ denotes the set of operators that are producers of atom $\langle v = x \rangle$:

$$
\begin{aligned}
& & \leq_1 (\{o_l | o \in O\}); \qquad & (1) \\
\forall v \in \mathcal{V}: & & \leq_1 (\{v =_l x_i | x_i \in D(v)\}); \qquad & (2) \\
\forall \langle v = x \rangle \in s_0: & & v =_0 x; \qquad & (3) \\
\forall o \in O: & & \bigwedge_{v = x \in pre(o)} (\neg o_l \vee v =_{l-1} x); \qquad & (4) \\
\forall o \in O: & & \bigwedge_{v = x \in post(o)} (\neg o_l \vee v =_l x); \qquad & (5) \\
\forall \langle v = x \rangle \in \mathcal{V}: & & v =_{l+1} x \implies v =_l x \vee \bigvee_{o \in prod(\langle v = x \rangle)} o_{l+1}; \qquad & (6) \\
\forall \langle v = x \rangle \in s_*: & & v =_L x \vee [\Sigma \mathcal{C}(o) \geq L + 1]; \qquad & (7) \\
\forall o \in O: & & \leq_{\mathcal{C}(o)} (\{o_l | l \in [1, L]\}) \vee [Y_o \geq \mathcal{C}(o) + 1]. \qquad & (8)
\end{aligned}
$$

Part (1) ensures that at most one operator occurs by layer; (2) ensures that a variable can only assume one value from its domain at a time; (3) that the atoms in the initial state are valid at first layer $l = 0$; (4) that an operator can occur at layer l only if its preconditions are satisfied at the previous layer $l - 1$; (5) that the effects of an operator applied are valid at layer l; (6) that an atom can only be valid at layer l if it is valid at the previous layer $l - 1$ or an operator which is a producer of this atom is applied at l; (7) ensures that all atoms from the goal state are valid at the last layer L; and (8) that the total number of times each operator is applied is bounded by the number of operators available in the operator counts $\mathcal{C}$. Variables $[\Sigma \mathcal{C}(o) \geq L + 1]$ and $[Y_o \geq \mathcal{C}(o) + 1]$ are the *assumptions* informed to the SAT solver and used to express the formula's infeasibility explanation as a GLC.

4 PROPOSED APPROACH

4.1 Overview

We propose the solver *OpSearch*, which uses the A* search algorithm to solve the operator counts sequencing subproblem. *OpSearch* works in the same way as *OpSeq*: given an operator counts $\mathcal{C}$, it returns a plan π if $\mathcal{C}$ is sequencable, or a violated condition as a GLC L, otherwise. The master problem is unchanged. The presence of potentially useful information in the search graph, such as f-values, motivates the use of heuristic search algorithms as base for an alternative method. This approach could generate smaller and more informed constraints and, as observed by CIRÉ; COBAN; HOOKER (2013), eliminating irrelevant parts of constraints can significantly decrease the solving time of an integer program.

4.2 Extended State and Generation of Successors

Usually, only a set of atoms constitutes the default A* state representation in a search node, representing the current assignment of values $x \in D(v)$ to all variables $v \in \mathcal{V}$. Since the number of variables $|\mathcal{V}|$ and the domain sizes $|D(v)|$ are known, current implementations store states in a memory-friendly bitstring. This efficient representation enables planners to expand large state-spaces without running out of memory. However, it is insufficient to sequence operator counts by heuristic search, since a partial assignment of variables can be reached by applying distinct sequences of operators.

To address this issue, we extended the A* state representation with the operator counts. The number of distinct operators with counts greater than zero in the operator counts $\mathcal{C}$ gives the bound on the number of extra variables necessary to represent subsets of $\mathcal{C}$ and each variable corresponding to operator o is bounded by the initial operator counts $\mathcal{C}(o)$, since the operator counts of $\mathcal{C}$ can only decrease during search.

Using this extended state, *OpSearch* is able to distinguish between two states with identical values assignment for the original task variables $v \in \mathcal{V}$ but with different operator counts, i.e., different task operators budgets. Given the current operator counts $\mathcal{C}$, we extend the A* state representation with a variable v_o for each $o \in O$ if $\mathcal{C}(o) > 0$ and $c(o) > 0$. The domain of v_o is $D(v_o) = \{0, \ldots, \mathcal{C}(o)\}$.

This new state representation requires another modification in the behavior of A*, which needs to consider the extra operator variables and limit the number of times an operator o is applied. Effectively, if A* could generate s' from s using operator o, it will in fact generate s' in two situations. First, if $c(o) = 0$, i.e., we generate states freely for zero-cost operators. Second, if $v_o \in vars(s)$ and $s(v_o) > 0$ then s' is generated and the value of variable v_o in s' is set to $s'(v_o) = s(v_o) - 1$.

OpSearch applies zero-cost operators independently of $\mathcal{C}$ and only generates bounds literals for operators o with $c(o) > 0$. Zero-cost operators can be applied freely during the search, even if they are absent from the current operator counts. This is motivated by the observation that bounds literals for zero-cost operators do not directly force the operator-counting objective function to increase.

To illustrate these modifications to A*, we use a planning task Π_1 as an example, containing $\mathcal{V} = \langle v_1 \rangle$ with $D(v_1) = \{0, 1, 2\}$, $O = \{o_1, o_2, o_3, o_4\}$ with $o_1 = \langle v_1 = 1; v_1 := 2\rangle$, $o_2 = \langle v_1 = 0; v_1 := 2\rangle$, $o_3 = \langle v_1 = 1; v_1 := 2\rangle$, $o_4 = \langle v_1 = 1; v_1 := 3\rangle$, $c(o_1) = 2$, $c(o_2) = c(o_4) = 1$ and $c(o_3) = 0$. Note that, even though o_1 and o_3 have identical preconditions and effects, they have distinct costs and therefore are different operators. The initial state is $s_0 = \{v_1 = 1\}$ and the goal is $s_* = \{v_1 = 2\}$. Suppose the operator counts is $\mathcal{C} = \{o_1 \mapsto 1\}$.

In *OpSearch*, this task would be changed by including a variable v_{o_1} for operator o_1 with domain $D(v_{o_1}) = \{0, 1\}$ since $\mathcal{C}(o_1) = 1$, but no variables for o_2, o_3 or o_4, since o_2 and o_4 have counts equal to zero and o_3 has zero cost. The value of v_o in s_0 is $\mathcal{C}(o)$. Therefore, our final extended representation for state s_0 would be $\{v_1 = 0, v_{o_1} = 1\}$. Extended states are used to test for equality and for successor generation. However, for computing the heuristic function, only the original variables of the planning task are considered.

During the execution of the modified A* in this task, *OpSearch* would generate two states from s_0: state $s' = \{v_1 = 2, v_{o_1} = 0\}$ by applying operator o_1 and $s'' = \{v_1 = 2, v_{o_1} = 1\}$ by applying o_3. No state is generated from the application of operator o_4, since it is not contained in $vars(s)$.

4.3 Constraint Generation Strategy

We now explore the situation when $v_o \notin vars(s) \wedge c(o) > 0$ and when $s(v_o) = 0$ to derive some violated condition on the current operator counts $\mathcal{C}$ that is not sequencable. This condition is modeled as a GLC with bounds literals for operator o and can be interpreted as follows: if we had one more instance of o, we could further generate a state, that could possibly reach a goal state with optimal cost. Additionally, we can use other information available during A* to strengthen the generated constraints, such as the f-value of state s, since it is an estimate of the plan cost through s.

Next we present the strategy to generate violated constraints from non-sequencable operator counts. It incrementally generates bounds literals during A* search to compose the final learned GLC L, that includes at most one bounds literal for each operator. The strategy returns bounds for operators that currently have count equal to zero but might generate new states with f-value at most f_{max}. The f_{max} is the objective value of the relaxation of some node in the BC tree, that is found while solving operator-counting IP. This node calls the sequencing subproblem informing f_{max} and the operator counts $\mathcal{C}$. State s denotes a state expanded by A* and s' is a generated one.

$$L = \{\, [Y_o \geq \mathcal{C}(o) + 1] \mid \exists s \xrightarrow{o} s' : f(s') \leq f_{\text{max}} \wedge \\ ((v_o \notin vars(s) \wedge c(o) > 0) \vee s(v_o) = 0)\}.$$

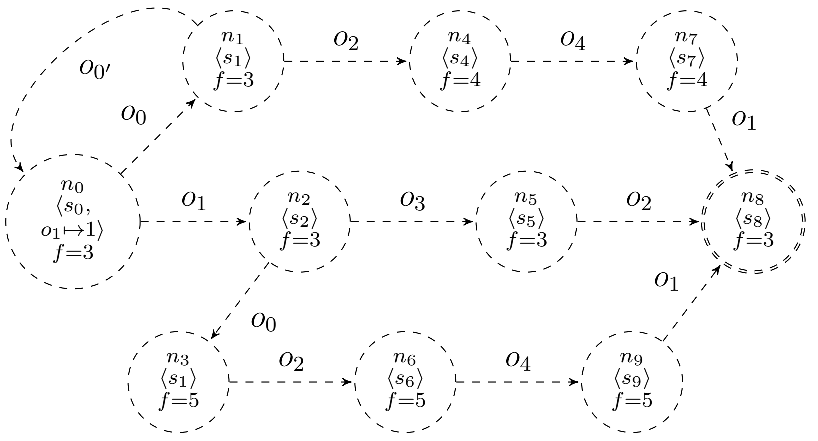
Further, if the f-value of a state s' found during search is greater than f_{max}, then we directly bound the plan cost. To this end, we introduce an auxiliary variable Y_f, which represents the objective function value to the operator-counting model and is defined as:

$$\mathsf{Y}_f = \sum_{o \in O} c(o)\mathsf{Y}_o.$$

Now let $f_{\text{min}} = \min_{s \mid f(s') > f_{\text{max}}} \{f(s')\}$, i.e., f_{min} is the minimum f-value for all states found during A* that have f-value greater than f_{max}. Then, if some state s' has $f(s') > f_{\text{max}}$, we add the bounds literal $[\mathsf{Y}_f \geq f_{\text{min}}]$ to L. Note that f_{max} is used during the A* sequencing procedure and f_{min} is used after A* to construct the GLC when $\mathcal{C}$ is not sequencable.

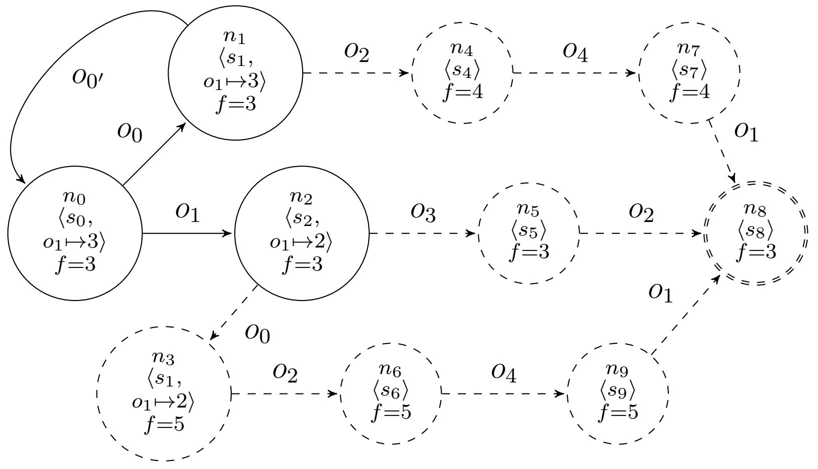
To illustrate the solving process of *OpSearch*, we define another example planning task Π_2 with $O = \{o_0, o_{0'}, o_1, o_2, o_3, o_4\}$ and costs $c(o_0) = c(o_{0'}) = 0$, $c(o_1) = c(o_2) = c(o_3) = 1$ and $c(o_4) = 2$. We assume that the initial operator-counting IP$_C$ contains the constraint $\mathsf{Y}_{o_1} \geq 1$ since o_1 is an action landmark for Π_2. The primal solution for this IP_C provides the operator counts $\mathcal{C} = \{o_1 \mapsto 1\}$ and the objective function value gives the f-value bound $f_{\max} = 1$. Figure 4.1 illustrates the state-space generated by A* with the perfect heuristic h^*, where vertices represent nodes and arcs the application of operators. Solid vertices and edges represent nodes and operators that are generated or applied according to $\mathcal{C}$. Nodes and operators that cannot be generated or applied during the search are dashed. Goals are indicated by double circled vertices.

Figure 4.1: State-space of example task Π_2, first iteration.


n_1 $\langle s_1 \rangle$ f=3
n_4 $\langle s_4 \rangle$ f=4
n_7 $\langle s_7 \rangle$ f=4
n_0 $\langle s_0,$ $o_1 \mapsto 1 \rangle$ f=3
n_2 $\langle s_2 \rangle$ f=3
n_5 $\langle s_5 \rangle$ f=3
n_8 $\langle s_8 \rangle$ f=3
n_3 $\langle s_1 \rangle$ f=5
n_6 $\langle s_6 \rangle$ f=5
n_9 $\langle s_9 \rangle$ f=5
$o_{0'}$
o_0
o_1
o_2
o_3
o_4

Since $f(n_0) > f_{\max}$, *OpSearch* generates the constraint $[\mathsf{Y}_f \geq 3] \geq 1$ informing that the f-value bound $f_{\max}$ must increase to 3 in the next iteration. Assume now that after adding this constraint the master returns $\mathcal{C} = \{o_1 \mapsto 3\}$ and $f_{\max} = 3$. The resulting state-space is illustrated in Figure 4.2:

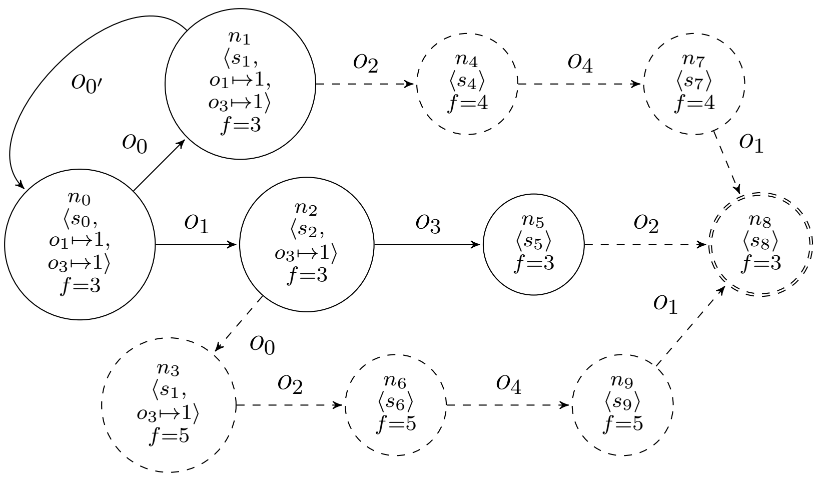
Figure 4.2: State-space of example task Π_2, second iteration.


n_1 $\langle s_1,$ $o_1 \mapsto 3 \rangle$ f=3
n_4 $\langle s_4 \rangle$ f=4
n_7 $\langle s_7 \rangle$ f=4
n_0 $\langle s_0,$ $o_1 \mapsto 3 \rangle$ f=3
n_2 $\langle s_2,$ $o_1 \mapsto 2 \rangle$ f=3
n_5 $\langle s_5 \rangle$ f=3
n_8 $\langle s_8 \rangle$ f=3
n_3 $\langle s_1,$ $o_1 \mapsto 2 \rangle$ f=5
n_6 $\langle s_6 \rangle$ f=5
n_9 $\langle s_9 \rangle$ f=5
$o_{0'}$
o_0
o_1
o_2
o_3
o_4

Now *OpSearch* expands n_0 and generates node n_2 by applying o_1. Since we apply zero-cost operators freely during search *OpSearch* also generates n_1 and n_3 by applying

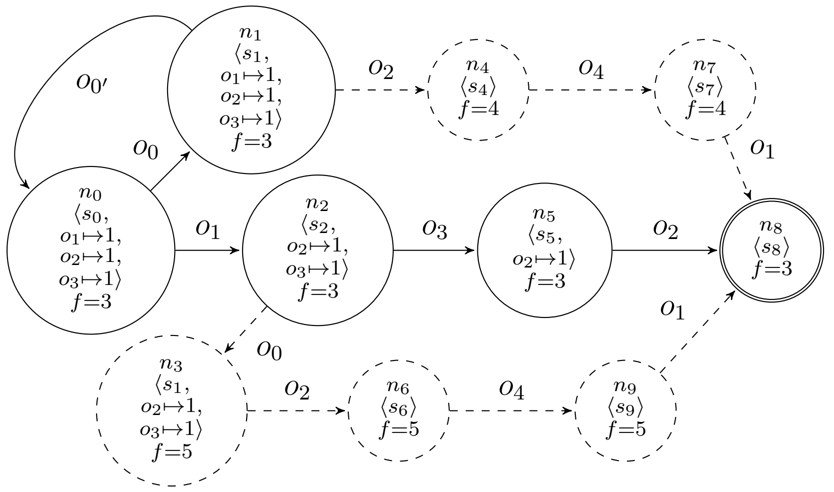
o_0 to n_0 and n_2. Note that n_1 and n_3 have the same variable assignment s_1 but different operator counts $\{o_1 \mapsto 3\}$ and $\{o_1 \mapsto 2\}$ and therefore are treated as different states. From this state-space, *OpSearch* returns the constraint $[\mathsf{Y}_{o_3} \geq 1] + [\mathsf{Y}_f \geq 4] \geq 1$. The bound $[\mathsf{Y}_{o_3} \geq 1]$ comes from the transition $n_2 \xrightarrow{o_3} n_5$ and $[\mathsf{Y}_f \geq 4]$ from $n_1 \xrightarrow{o_2} n_4$, since transition $n_2 \xrightarrow{o_0} n_3$ would generate the bound $[\mathsf{Y}_f \geq 5]$. Suppose that, after adding this constraint, the IP$_C$ returns $\mathcal{C} = \{o_1 \mapsto 2, o_3 \mapsto 1\}$ and $f_{\max} = 3$. The resulting state-space is shown in Figure 4.3:

Figure 4.3: State-space of example task Π_2, third iteration.


n_1 $\langle s_1, o_1\mapsto 1, o_3\mapsto 1\rangle$ f=3
$o_{0'}$
o_0
o_2
n_4 $\langle s_4\rangle$ f=4
o_4
n_7 $\langle s_7\rangle$ f=4
o_1
n_0 $\langle s_0, o_1\mapsto 1, o_3\mapsto 1\rangle$ f=3
o_1
n_2 $\langle s_2, o_3\mapsto 1\rangle$ f=3
o_3
n_5 $\langle s_5\rangle$ f=3
o_2
n_8 $\langle s_8\rangle$ f=3
o_1
o_0
n_3 $\langle s_1, o_3\mapsto 1\rangle$ f=5
o_2
n_6 $\langle s_6\rangle$ f=5
o_4
n_9 $\langle s_9\rangle$ f=5

From this state-space, *OpSearch* returns the constraint $[\mathsf{Y}_{o_2} \geq 1] + [\mathsf{Y}_f \geq 4] \geq 1$. The bound $[\mathsf{Y}_{o_2} \geq 1]$ comes from the transition $n_5 \xrightarrow{o_2} n_8$ and $[\mathsf{Y}_f \geq 4]$ from $n_1 \xrightarrow{o_2} n_4$. After adding this constraint, *OpSearch* returns a sequencable operator counts $\mathcal{C} = \{o_1 \mapsto 1, o_2 \mapsto 1, o_3 \mapsto 1\}$ and $f_{\max} = 3$, as illustrated in Figure 4.4.

Figure 4.4: State-space of example task Π_2, fourth iteration.


n_1 $\langle s_1, o_1\mapsto 1, o_2\mapsto 1, o_3\mapsto 1\rangle$ f=3
$o_{0'}$
o_0
o_2
n_4 $\langle s_4\rangle$ f=4
o_4
n_7 $\langle s_7\rangle$ f=4
o_1
n_0 $\langle s_0, o_1\mapsto 1, o_2\mapsto 1, o_3\mapsto 1\rangle$ f=3
o_1
n_2 $\langle s_2, o_2\mapsto 1, o_3\mapsto 1\rangle$ f=3
o_3
n_5 $\langle s_5, o_2\mapsto 1\rangle$ f=3
o_2
n_8 $\langle s_8\rangle$ f=3
o_1
o_0
n_3 $\langle s_1, o_2\mapsto 1, o_3\mapsto 1\rangle$ f=5
o_2
n_6 $\langle s_6\rangle$ f=5
o_4
n_9 $\langle s_9\rangle$ f=5

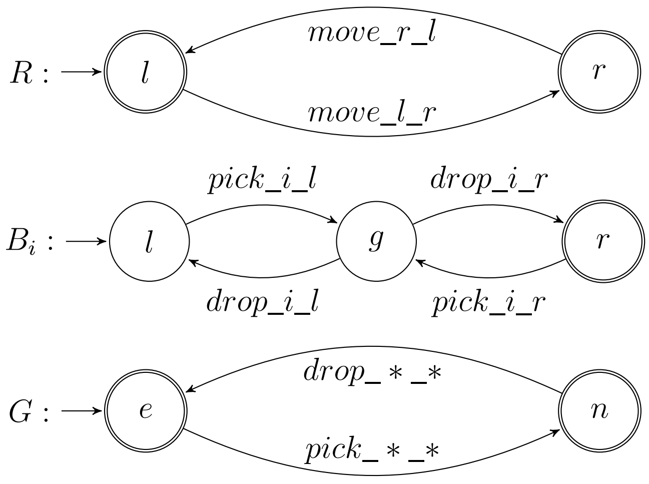
Theorem 1. *For a solvable SAS$^+$ planning task Π, an operator counts $\mathcal{C}_s$ with an associated f-bound value f_{max}, such that* OpSearch*'s modified A* with an admissible heuristic function h cannot sequence $\mathcal{C}_s$,* OpSearch *always returns an admissible constraint to the master integer program.*

Proof sketch. Consider an optimal plan $\pi^* = \langle o_1, \ldots, o_k \rangle$ with a corresponding state sequence $\langle s_0, s_1, \ldots, s_* \rangle$. Let L be a GLC generated by *OpSearch* with $\mathcal{C}_s$ and f-bound f_{max}, and S be the set of (extended) states expanded by *OpSearch*. Now, extend the state sequence $\langle s_0, s_1, \ldots, s_* \rangle$ to an (extended) state sequence $\langle s'_0, s'_1, \ldots, s'_* \rangle$ with operator-counting variables, such that the operator count of s'_0 is $\mathcal{C}_s$, and that of the subsequent states is decreased according to π^*. Since *OpSearch* failed to sequence $\mathcal{C}_s$ and maintains an extended state, there must be a first state $s'_i \notin S$. If $i = 0$, then $f(s_0) > f_{\text{max}}$ and the bounds literal $[\mathsf{Y}_f \geq f_{\text{min}}]$ must be satisfied since the heuristic h is admissible. Otherwise, there is a predecessor state $s'_{i-1} \in S$ with $s'_{i-1} \xrightarrow{o} s'_i$, and *OpSearch* did not generate s'_i. The reason for this is either $f(s'_i) > f_{\text{max}}$ or $s(o) = 0$ or $v_o \notin vars(s'_i) \wedge c(o) > 0$. But in the first case $f(s'_i) \geq f_{\text{min}}$ and by admissibility of h the bounds literal $[\mathsf{Y}_f \geq f_{\text{min}}]$ is satisfied, and in the second case the bounds literal $[\mathsf{Y}_o \geq \mathcal{C}_s(o) + 1]$ must be satisfied by π^*. □

4.4 Impact of Heuristic Functions in Generated Constraints

Using different heuristic functions to guide A* also plays an important role in the generation of constraints, since we expect more informed heuristics to generate smaller and stronger constraints by *OpSearch*. To illustrate this behaviour in more detail, we reintroduce the simple *gripper* example from (DAVIES et al., 2015): there are two balls, two rooms and a robot that can transport one ball at a time. The robot starts in the left room and the goal is to move the balls from left to right. Operators $pick\_i\_j$ and $drop\_i\_j$ causes the robot to hold or release ball i at room j and $move\_i\_j$ causes the robot to move from room i to j. All operators have unit-cost. An optimal plan with total cost of 7 is $\langle pick\_1\_l, move\_l\_r, drop\_1\_r, move\_r\_l, pick\_2\_l, move\_l\_r, drop\_2\_r \rangle$. The DTG for this example is illustrated in Figure 4.5. R represents the location of the robot (in left or right room); B_i the location of ball i (in left or right room or in the gripper); and G the state of the gripper (empty or non-empty). For each variable, the initial states are marked by an incoming arrow and goal states are double circled.

Figure 4.5: DTGs for the gripper example introduced by DAVIES et al. (2015).



We assume that the initial f-bound is $f_{\max} = 5$ and the initial operator counts is $\mathcal{C} = \{drop\_1\_r \mapsto 1, drop\_2\_r \mapsto 1, move\_l\_r \mapsto 1, pick\_1\_l \mapsto 1, pick\_2\_l \mapsto 1\}$. Since this operator counts is not sequencable, *OpSearch* learns some violated constraint. Next we show examples to illustrate that *OpSearch* with different heuristics generates different violated constraints, even if the base operator-counting master IP$_C$ initially contains the same set of constraints. To generate these examples, we use an operator-counting with initial constraints from disjunctive action landmarks h^{LMC} (BONET; BRIEL, 2014), state equation h^{SEQ} (BONET, 2013), and the optimal delete relaxation h^{+} (IMAI; FUKUNAGA, 2014).

OpSeq generates a constraint with five bounds: $[\mathsf{Y}_T \geq 6] + [\mathsf{Y}_{drop\_1\_l} \geq 1] + [\mathsf{Y}_{drop\_2\_l} \geq 1] + [\mathsf{Y}_{move\_r\_l} \geq 1] + [\mathsf{Y}_{pick\_1\_r} \geq 1] \geq 1$. *OpSearch* with the blind heuristic h^{blind}, that returns zero for goal states and one for others, also generates a constraint with five bounds, but replaces the bound for Y_T by $\mathsf{Y}_{pick\_2\_r}$: $[\mathsf{Y}_{drop\_1\_l} \geq 1] + [\mathsf{Y}_{drop\_2\_l} \geq 1] + [\mathsf{Y}_{move\_r\_l} \geq 1] + [\mathsf{Y}_{pick\_1\_r} \geq 1] + [\mathsf{Y}_{pick\_2\_r} \geq 1] \geq 1$. *OpSearch* with the h^{LMCut} heuristic generates a stronger constraint with only one bound for the plan cost: $[\mathsf{Y}_f \geq 6] \geq 1$. Finally, *OpSearch* with h^* generates a perfect constraint that forces the IP$_C$ objective value to directly increase up to the optimal plan cost: $[\mathsf{Y}_f \geq 7] \geq 1$.

5 EXPERIMENTS

5.1 Overview

The goals of the experiments are: i) to evaluate the performance of *OpSearch* compared to *OpSeq* in terms of the total number of sequencing subproblems and instances solved; ii) to contrast the computational resources required by both approaches in terms of total solving runtimes and memory consumption on average; and iii) to experimentally validate the hypothesis that *OpSearch* can generate stronger constraints in terms of the average number of operators included.

We use the same benchmarks from the *International Planning Competition* (IPC) of 2011 used by DAVIES et al. (2015), totaling 11 domains with 20 instances each. All experiments were conducted using an Intel Core i7 930 CPU (2.80 GHz) with a memory limit of 4 GB and a time limit of one hour for each planner execution. We implemented *OpSearch* and *OpSeq* inside the Fast Downward planning system, version 19.06 (HELMERT, 2006). For solving the satisfiability subproblems we use the MiniSat 2.2 solver (EÉN; SÖRENSSON, 2003). We opted to use MiniSat to facilitate comparisons with (DAVIES et al., 2015) but we could use any SAT solver with support to conflict analysis. We use CPLEX 12.8 to solve linear programs and implement the LBBD approach.

The initial operator-counting master IP$_C$ contains constraints from the disjunctive action landmarks heuristic h^{LMC} (BONET; BRIEL, 2014), the state equation h^{SEQ} (BONET, 2013) and the optimal delete relaxation h^{+} base formulation without enhancements from (IMAI; FUKUNAGA, 2014). We use h^{LMCut} (HELMERT; DOMSHLAK, 2009) as the primary heuristic function to guide the A* algorithm from *OpSearch* when sequencing operator counts.

Since *OpSeq*'s source code is not publicly available, we re-implemented its SAT-based approach strictly following the description presented in (DAVIES et al., 2015). The source code for *OpSearch* and our version of *OpSeq* is available[1] to facilitate further development of operator counts sequencing procedures. The data and scripts used to generate the tables and figures presented in this chapter are also publicly available[2].

[1]Available at <https://github.com/kaizerw/PlanningLP>
[2]Available at <https://github.com/kaizerw/icaps20>

5.2 The Benchmark Set

Table 5.1 presents information about the benchmark set, summarized by domain. $\overline{|\mathcal{V}|}$ denotes the mean number of variables; $\overline{|O|}$ is the mean number of operators; zco indicates the presence of zero-cost operators; $\overline{c_{min}}$ is the mean minimum operator cost, ignoring zero-cost operators; $\overline{c_{max}}$ is the mean maximum operator cost; $\overline{lb}$ is the mean best lower bound on the optimal plan cost[3]; $\overline{z}_0$ is the mean initial relaxed operator-counting solution of our initial operator-counting master problem; and $\overline{r}_0$ and $\overline{c}_0$ are the mean number of rows and columns in the initial IP$_C$ program, respectively.

Table 5.1: Information about the benchmark set.

domain	$\overline{\|\mathcal{V}\|}$	$\overline{\|O\|}$	zco	$\overline{c_{min}}$	$\overline{c_{max}}$	$\overline{lb}$	$\overline{z_0}$	$\overline{r_0}$	$\overline{c_0}$
`barman`	53.3	358.3	–	1	10	30.15	15.75	7408.2	3896.0
`elevators`	40.0	866.0	✓	6	32	3.75	1.00	12810.3	6265.0
`nomystery`	34.0	185.0	–	1	1	8.85	3.92	3701.9	1772.0
`openstacks`	108.2	663.2	✓	1	1	123.35	76.58	14456.3	6231.6
`parcprinter`	59.9	254.8	✓	987	217007	1223929.00	1223929.00	4340.6	2167.9
`pegsol`	12.2	572.5	✓	1	1	59.05	34.09	8201.0	4120.8
`scanalyzer`	9.7	1280.0	–	1	3	521.90	295.91	26130.9	12515.8
`sokoban`	7.1	1380.8	✓	1	1	24.85	21.60	47324.4	25688.1
`transport`	38.6	176.0	–	1	95	41.80	40.78	2096.2	1406.5
`visitall`	15.5	1659.5	–	1	1	36.90	30.62	189001.6	91734.2
`woodworking`	74.5	908.8	–	5	44	329.50	296.40	17438.5	7980.7

We see that the domains *elevators*, *parcprinter*, *openstacks*, *pegsol* and *sokoban* have zero-cost operators and the last three only have zero-cost and unit-cost operators. Two domains have only unit-cost operators: *nomystery* and *visitall*. Ignoring zero-cost operators, some domains have diverse operators costs such as *barman*, *elevators*, *parcprinter*, *scanalyzer*, *transport* and *woodworking*. Among these, *parcprinter* is notable due to the wide cost range of its operators.

We observe that some domains have few operators and variables, such as *nomystery* and *transport* and others have a large number of operators but few variables, such as *visitall*, *sokoban* and *scanalyzer*. We can also note that $\overline{z}_0$ is very close to $\overline{lb}$ in *parcprinter*, *sokoban*, *transport* and *visitall*. Some domains have huge initial IP$_C$ such as *visitall* and *sokoban* while others have very small ones, for instance, *nomystery*, *parcprinter* and *transport*.

[3]Obtained from <http://editor.planning.domains>

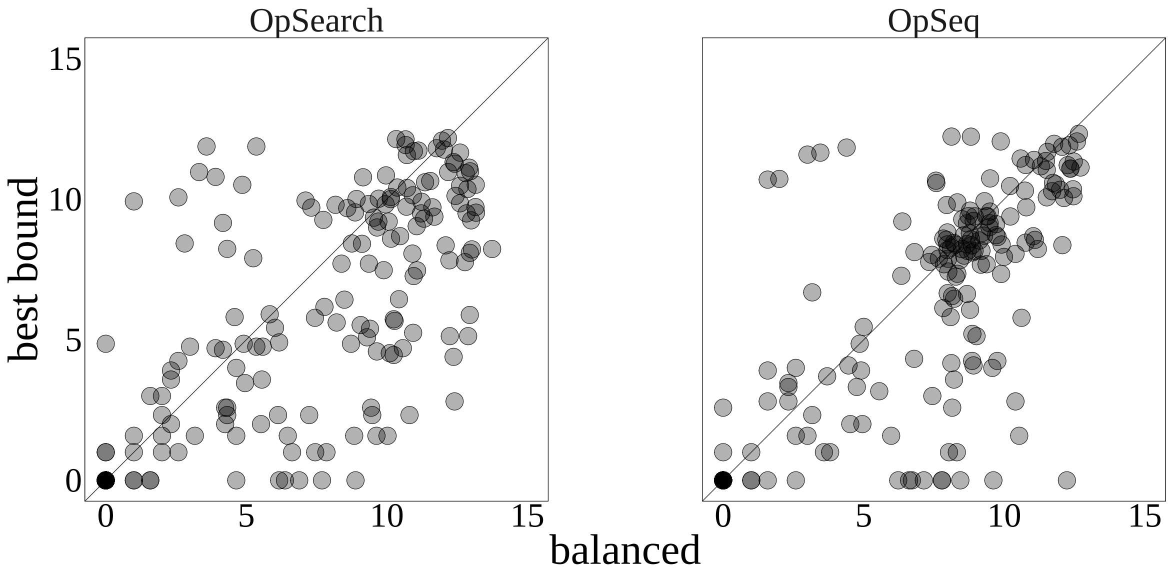
5.3 IP Solver Settings

We noticed that settings for IP solvers can change the BC process and interfere with the operator counts sequencing subproblem. In particular, some *primal heuristics* executed by the IP solver to construct feasible solutions can generate very large operator counts which are not useful to sequence, and which in *OpSeq* lead to memory problems when constructing the SAT models. We have turned off these heuristics in both approaches. We used legacy callbacks of the C++ interface in CPLEX to add the learned constraints through user cuts and lazy constraints.

Another relevant parameter is the *IP emphasis*. With the default setting of *balanced*, the solver tries to balance progress on good feasible solutions and a proof of optimality. When set to *best bound*, it prioritizes increasing the current best bound with low effort in detecting feasible integer solutions. Considering the incremental lower bounding technique used by *OpSeq*, we use the *best bound* setting in our experiments.

Figure 5.1 shows plots of the total number of sequence calls, comparing IP emphasis *balanced* to *best bound*. We can see that when the IP emphasis is set to *best bound*, both *OpSearch* and *OpSeq* require fewer sequencing calls than with the *balanced* setting.

Figure 5.1: IP emphasis (log2-log2 scale).



5.4 *OpSeq* and *OpSearch*

Tables 5.2 and 5.3 show results grouped by domain for *OpSeq* and *OpSearch*. Column C presents the coverage for that particular domain; S the total number of sequencing calls; R the total number of restarts; $\bar{T}_t$ the mean total solving time in seconds; $\overline{M}$ the mean memory usage in megabytes; $\overline{u}$ the mean percentage of operators included in the generated constraints; $\overline{p}$ the mean percentage of total sequencing times by total solving time; and *bb* the best bound found by the IP solver. Best results are highlighted.

Table 5.2: *OpSeq*.

domain	C	S	R	$\bar{T}_t$	$\bar{M}$	$\bar{u}$	$\bar{p}$	bb
`barman`	**0**	40556	16	**3417**	857	20	**0.1**	2484
`elevators`	**0**	**5922**	**0**	**3275**	2931	17	0.8	690
`nomystery`	0	**3660**	**0**	1459	736	44	0.6	437
`openstacks`	0	24383	**3**	1709	**433**	29	**0.1**	20
`parcprinter`	**20**	**21**	**0**	**1**	**126**	**0**	74.0	**8524162**
`pegsol`	**11**	22998	15	1964	175	47	**0.0**	154
`scanalyzer`	0	3377	**0**	1305	**955**	18	**0.0**	585
`sokoban`	0	**7907**	**0**	3208	2385	9	0.8	319
`transport`	**0**	5800	**0**	1879	265	8	**0.0**	**6251**
`visitall`	**15**	**5632**	**0**	**957**	298	**19**	0.2	**848**
`woodworking`	**17**	946	**0**	**437**	355	**4**	**0.5**	**6348**
Total	63	121202	**34**	1783	865	20	**0.4**	8542298

Table 5.3: *OpSearch*.

domain	C	S	R	$\bar{T}_t$	$\bar{M}$	$\bar{u}$	$\bar{p}$	bb
`barman`	**0**	**36565**	**1**	3548	**202**	**5**	**0.1**	**2496**
`elevators`	**0**	10802	7	3555	**254**	**4**	**0.2**	**865**
`nomystery`	**3**	10383	4	**1120**	**322**	**1**	**0.1**	**443**
`openstacks`	**13**	**266**	14	**966**	968	**0**	23.6	**67**
`parcprinter`	16	**21**	**0**	271	377	**0**	**55.9**	**8524162**
`pegsol`	10	**12906**	**2**	**1888**	**123**	**16**	**0.0**	**166**
`scanalyzer`	**1**	**700**	5	**1001**	1046	**2**	**0.0**	**592**
`sokoban`	**5**	17695	51	**2779**	**183**	**3**	**0.2**	**455**
`transport`	**0**	**910**	11	**1707**	**222**	**1**	**0.0**	6235
`visitall`	14	9078	10	1112	**119**	29	**0.0**	839
`woodworking`	11	**111**	**0**	974	**224**	5	43.7	6258
Total	**73**	**99437**	105	**1720**	**367**	**6**	4.4	**8542578**

We see that *OpSearch* has better coverage than *OpSeq*, solving 10 more planning tasks. *OpSearch* performs better on domains *nomystery*, *openstacks*, *scanalyzer* and *sokoban*. *OpSearch* on *openstacks* and *sokoban* solves 13 and 5 tasks not solved by *OpSeq*. We find that *OpSearch* uses 57% less memory and generates violated constraints that are on average 70% smaller than *OpSeq*. We also observe that *OpSearch* has a smaller total number of sequencing calls, approximately 18%, more restarts, and that it found higher best bounds than *OpSeq* in seven domains.

An important metric for comparing the solvers is the percentage of operators in the learned constraints. On average, constraints generated by *OpSeq* have 20% of the operators, while constraints generated by *OpSearch* have only 6% of the operators. Also, *OpSeq* learns constraints with more than 10% of the operators on seven domains, while *OpSearch* learns constraints with more than 10% of the operators on only two domains, which confirms the potential of search-based methods to solve the operator counts sequencing subproblem, generating smaller and potentially more informed constraints.

Figures 5.2, 5.3, 5.4, 5.5 and 5.6 show plots comparing the total number of sequencing calls S, memory usage M, mean percentage of operators by learned constraints $\bar{u}$, total sequence times S_t and total solving time T_t for *OpSearch* and *OpSeq*. Visually, we can confirm the results presented before: i) *OpSearch* solved fewer sequencing subproblems; ii) in most of the times *OpSearch* uses less memory than *OpSeq*; and iii) *OpSearch* usually generates smaller constraints than *OpSeq*.

Figure 5.2: Sequencing subproblem calls S (log2-log2 scale).

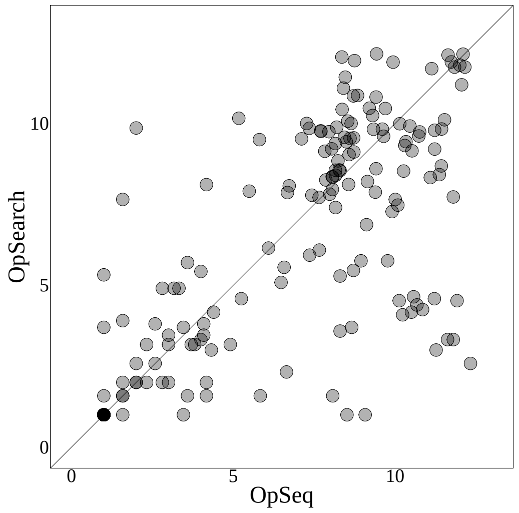


Figure 5.3: Memory usage M (log2-log2 scale).

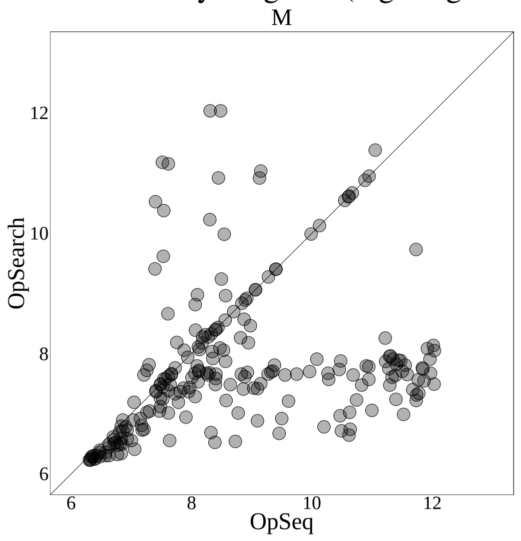


Figure 5.4: Mean constraint size $\bar{u}$ (log2-log2 scale).

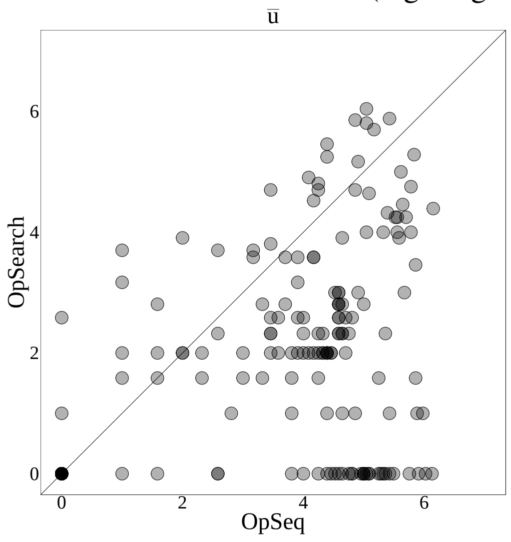


Figure 5.5: Sequencing runtimes S_t (log2-log2 scale).

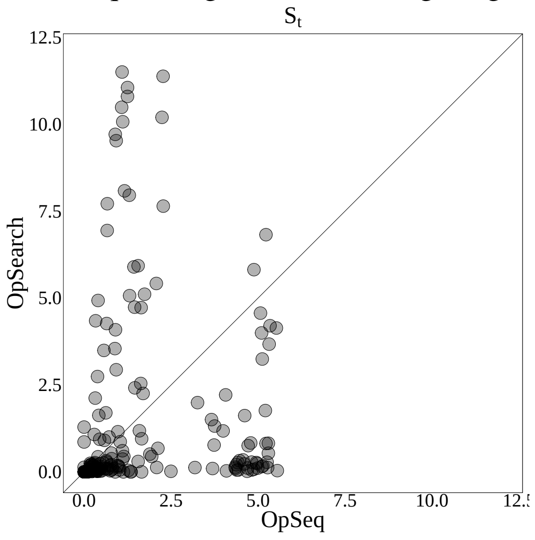


Figure 5.6: Total solving time T_t (log2-log2 scale).

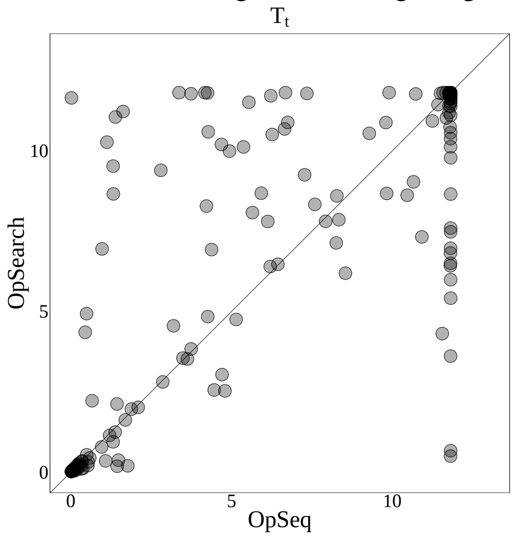


Table 5.4 summarises the results only considering the 50 instances solved by both *OpSearch* and *OpSeq*. This table shows that, when we compare *OpSearch* and *OpSeq* using the same set of instances, *OpSearch* in fact solves fewer sequencing subproblems, uses slightly less memory and generates smaller constraints than *OpSeq*, but spends more time sequencing, as indicated by $\bar{p}$, and has a higher total solving time, as indicated by $\bar{T}_t$.

Table 5.4: Summary for the 50 instances solved by both methods.

	S	R	$\bar{T}_t$	$\bar{M}$	$\bar{u}$	$\bar{p}$
OpSearch	**2169**	**1**	191	**118**	**9**	46.4
OpSeq	2738	6	**92**	122	15	**0.3**

5.5 Impact of *OpSearch*'s heuristic function

Table 5.5 shows results for *OpSearch* using different heuristic functions in A*. We have tested the heuristics h^{blind}, h^{LMCut} and operator-counting h^{OC} with constraints from state-equation and action landmarks. We have chosen these functions because h^{OC} dominates h^{LMCut} and h^{blind} is the least informative. The table only shows results for the 49 instances solved by *OpSeq* and *OpSearch* using each one of the heuristics cited before, except by the column C that considers all the 220 instances.

The table shows that using more informed heuristic functions in *OpSearch* results in: i) fewer sequencing subproblems solved, as indicated by S; ii) greater mean total solving times $\bar{T}_t$ since computing the heuristics are more expensive; iii) less mean memory usage, as indicated by $\bar{M}$; and iv) smaller constraints are generated, as indicated by $\bar{u}$.

Table 5.5: Summary for the 49 instances solved by all heuristics.

	C	S	R	$\bar{T}_t$	$\bar{M}$	$\bar{u}$	$\bar{p}$
h^{blind}	**79**	3717	**1**	93	171	10	21.5
h^{LMCut}	73	2161	**1**	183	116	9	22.9
h^{OC}	70	**1119**	3	141	**99**	**8**	**17.4**
OpSeq	63	2725	6	**90**	121	16	23.4

Table 5.6 shows results for *OpSearch* using all the 282 instances from IPC-1998 to IPC-2014[4] in which h^* can be computed by a full PDB using at most 4 GB of memory. Similarly to the previous test, we used h^{blind}, h^{LMCut}, operator-counting h^{OC} with constraints from state-equation and action landmarks, and h^*. The table only shows results for the 154 instances solved by all methods, except by the column C that considers all the 282 instances.

We can observe that: i) the total number of sequencing subproblems solved decreases as the heuristic function is more informed (S); ii) the total solving times $\bar{T}_t$ for h^{OC} is twice as much as for the other heuristics; iii) h^* uses much more memory $\bar{M}$ than the other heuristics due to the full PDB; and iv) on average, smaller constraints are generated by more informed heuristics, as indicated by $\bar{u}$.

Table 5.6: Summary for the 154 instances solved by all heuristics.

	C	S	R	$\bar{T}_t$	$\bar{M}$	$\bar{u}$	$\bar{p}$
h^{blind}	191	25059	57	**10**	82	18	11.2
h^{LMCut}	195	13304	75	11	82	11	2.5
h^{OC}	200	7215	40	39	**81**	10	13.3
h^*	**241**	**3214**	**19**	13	234	**8**	**1.3**
OpSeq	169	29106	53	37	95	18	12.4

[4]Obtained from <https://bitbucket.org/aibasel/downward-benchmarks>

6 CONCLUSION AND FUTURE WORK

In this thesis we introduced *OpSearch*, a technique inspired by LBBD that uses an A*-based algorithm to solve the problem of sequencing operator counts. We discussed the inner workings of the approach to incrementally explicate during search why operator counts are not sequencable.

As main results, we showed that heuristic search is able to sequence operator counts or to generate admissible constraints in the form of generalized landmarks, and that it can perform better than *OpSeq*, a SAT-based approach to sequencing, solving fewer subproblems and presenting a higher coverage. We analyzed the choice of heuristic functions inside the A* procedure, showing that more informed heuristics can produce smaller and more focused violated constraints. We also presented results indicating that an approach based on A* can scale better than *OpSeq* in terms of overall memory usage. Finally, we made the source code for *OpSearch* and our version of *OpSeq* publicly available.

Future work could address the development of specific heuristic functions to explore structural properties of the operator counts sequencing problem, possibly giving rise to specialized and more efficient algorithms. Other directions would be to investigate improvements on the master integer program, studying strategies to generate operator counts more likely to be sequencable earlier during the solving process; improving methods to deal with zero-cost operators to increase the integer program lower bound faster; or evaluate pruning methods (WEHRLE; HELMERT, 2014) in the operator counts sequencing context, that might help to derive smaller constraints.

REFERENCES

ACHTERBERG, T.; KOCH, T.; MARTIN, A. Branching rules revisited. **Operations Research Letters**, v. 33, n. 1, p. 42 – 54, 2005.

BÄCKSTRÖM, C.; NEBEL, B. Complexity results for SAS+ planning. **Computational Intelligence**, v. 11, n. 4, p. 625–655, 1995.

BENDERS, J. F. Partitioning procedures for solving mixed-variables programming problems. **Computational Management Science**, Springer, v. 2, n. 1, p. 3–19, 2005.

BONET, B. An admissible heuristic for SAS$^+$ planning obtained from the state equation. **International Joint Conference on Artificial Intelligence**, p. 2268–2274, 2013.

BONET, B.; BRIEL, M. van den. Flow-based heuristics for optimal planning: Landmarks and merges. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2014. p. 47–55.

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

BYLANDER, T. A linear programming heuristic for optimal planning. In: **National Conference on Artificial Intelligence and Conference on Innovative Applications of Artificial Intelligence**. [S.l.]: AAAI Press, 1997. (AAAI'97/IAAI'97), p. 694–699.

CIRÉ, A.; COBAN, E.; HOOKER, J. N. Mixed Integer Programming vs. Logic-Based Benders Decomposition for Planning and Scheduling. In: GOMES, C.; SELLMANN, M. (Ed.). **Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 325–331.

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

DAVIES, T. O. et al. Sequencing operator counts. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2015. p. 61–69.

EDELKAMP, S. Planning with pattern databases. In: **European Conference on Planning**. [S.l.: s.n.], 2014. p. 84–90.

EÉN, N.; SÖRENSSON, N. An extensible SAT-solver. In: SPRINGER. **International Conference on Theory and Applications of Satisfiability Testing**. [S.l.], 2003. p. 502–518.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability; A Guide to the Theory of NP-Completeness**. New York, NY, USA: W. H. Freeman & Co., 1990.

GEFFNER, P. H. H.; HASLUM, P. Admissible heuristics for optimal planning. In: **International Conference of AI Planning Systems**. [S.l.: s.n.], 2000. p. 140–149.

GHALLAB, M.; NAU, D.; TRAVERSO, P. **Automated Planning: Theory and Practice**. [S.l.]: Elsevier, 2004.

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

HELMERT, M. The Fast Downward planning system. **Journal of Artificial Intelligence Research**, v. 26, p. 191–246, 2006.

HELMERT, M.; DOMSHLAK, C. Landmarks, critical paths and abstractions: what's the difference anyway? In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2009. p. 162–169.

HELMERT, M. et al. Flexible abstraction heuristics for optimal sequential planning. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2007. p. 176–183.

HOFFMANN, J.; NEBEL, B. The ff planning system: Fast plan generation through heuristic search. **Journal of Artificial Intelligence Research**, v. 14, p. 253–302, 2001.

HOFFMANN, J.; PORTEOUS, J.; SEBASTIA, L. Ordered landmarks in planning. **Journal of Artificial Intelligence Research**, v. 22, p. 215–278, 2004.

HOLTE, R. C. Common misconceptions concerning heuristic search. In: **Third Annual Symposium on Combinatorial Search**. [S.l.: s.n.], 2010. p. 46–51.

HOOKER, J. **Logic-based methods for optimization: combining optimization and constraint satisfaction**. [S.l.]: John Wiley & Sons, 2011. v. 2.

HOOKER, J. N.; OTTOSSON, G. Logic-based Benders decomposition. **Mathematical Programming**, Springer, v. 96, n. 1, p. 33–60, 2003.

IMAI, T.; FUKUNAGA, A. A practical, integer-linear programming model for the delete-relaxation in cost-optimal planning. In: **European Conference on Artificial Intelligence**. [S.l.: s.n.], 2014. p. 459–464.

KARMARKAR, N. A new polynomial-time algorithm for linear programming. **Combinatorica**, v. 4, p. 373–395, 12 1984.

KARPAS, E.; DOMSHLAK, C. Cost-optimal planning with landmarks. In: **International Joint Conference on Artificial Intelligence**. [S.l.: s.n.], 2009. p. 1728–1733.

KATZ, M.; DOMSHLAK, C. Optimal additive composition of abstraction-based admissible heuristics. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2008. p. 174–181.

MARQUES-SILVA, J. P.; SAKALLAH, K. A. GRASP: A search algorithm for propositional satisfiability. **IEEE Transactions on Computers**, IEEE, v. 48, n. 5, p. 506–521, 1999.

MCDERMOTT, D. Using regression-match graphs to control search in planning. **Artificial Intelligence**, Elsevier, v. 109, n. 1-2, p. 111–159, 1999.

MÉNDEZ-DÍAZ, I.; ZABALA, P. A branch-and-cut algorithm for graph coloring. **Discrete Applied Mathematics**, v. 154, n. 5, p. 826 – 847, 2006. IV ALIO/EURO Workshop on Applied Combinatorial Optimization.

MITCHELL, J. E. Branch-and-cut algorithms for combinatorial optimization problems. **Handbook of Applied Optimization**, v. 1, p. 65–77, 2002.

POMMERENING, F. **New perspectives on cost partitioning for optimal classical planning**. Tese (Doutorado) — University of Basel, 2017.

POMMERENING, F. et al. From non-negative to general operator cost partitioning. In: **AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2015. p. 3335–3341.

POMMERENING, F.; RÖGER, G.; HELMERT, M. Getting the most out of pattern databases for classical planning. In: **International Joint Conference on Artificial Intelligence**. [S.l.: s.n.], 2013. p. 2357–2364.

POMMERENING, F. et al. LP-based heuristics for cost-optimal planning. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2014. p. 226–234.

RAHMANIANI, R. et al. The benders decomposition algorithm: A literature review. **European Journal of Operational Research**, Elsevier, v. 259, n. 3, p. 801–817, 2017.

SEIPP, J.; HELMERT, M. Diverse and additive cartesian abstraction heuristics. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2014. p. 289–297.

SEIPP, J.; KELLER, T.; HELMERT, M. A comparison of cost partitioning algorithms for optimal classical planning. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2017. p. 259–268.

SINZ, C. Towards an optimal CNF encoding of boolean cardinality constraints. In: SPRINGER. **International Conference on Principles and Practice of Constraint Programming**. [S.l.], 2005. p. 827–831.

VASILYEV, I. L.; KLIMENTOVA, K. B. The branch and cut method for the facility location problem with client's preferences. **Journal of Applied and Industrial Mathematics**, v. 4, n. 3, p. 441–454, Jul 2010.

WEHRLE, M.; HELMERT, M. Efficient stubborn sets: Generalized algorithms and selection strategies. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2014. p. 323–331.

WOLSEY, L. **Integer Programming**. [S.l.]: Wiley, 1998. (Wiley Series in Discrete Mathematics and Optimization).

APPENDIX A — SEQUENCIAMENTO DE CONTAGENS DE OPERADORES COM BUSCA EM ESPAÇO DE ESTADOS

Em *Planejamento Clássico*, uma solução para uma *tarefa de planejamento* é denominada como *plano*, consistindo em uma sequência de *operadores* tal que sua aplicação na sequência especificada pelo plano no *estado inicial* da tarefa de planejamento resulta em um *estado objetivo*. Os *estados* descrevem condições que são válidas em determinados momentos durante o processo de solução e são alterados de acordo com a aplicação de operadores. Os operadores da tarefa de planejamento possuem custo associados às suas aplicações e usualmente deseja-se resolver *otimamente* uma tarefa de planejamento, isto é, obter um *plano ótimo* que minimiza o custo total da aplicação de seus operadores.

Uma abordagem de sucesso e amplamente empregada à solução do problema de planejamento consiste na utilização de *algoritmos de busca* informados por *funções heurísticas*, que estimam o custo de uma solução ótima a partir de um estado. Estes algoritmos, comumente denominados *algoritmos de busca heurística*, garantem que a solução encontrada é ótima, ou seja, que ela apresenta o menor custo possível dentre todas as soluções factíveis, se uma função heurística admissível é empregada. Uma função heurística é *admissível* se ela sempre fornece limitantes inferiores sobre o custo da solução ótima em todos os estados. A admissibilidade de funções heurísticas é um objeto de estudo de interesse na área de Planejamento Clássico que já possibilitou progressos importantes na área, como por exemplo o desenvolvimento de poderosas heurísticas capazes de resolver difíceis tarefas de planejamento eficientemente.

A heurística de *operator-counting* é de particular interesse para o trabalho desenvolvido nesta dissertação. Esta heurística viabiliza que informações provenientes de diversas heurísticas admissíveis sejam combinadas por meio da definição de um *Programa Linear*, em que as condições que devem ser satisfeitas por todos os planos factíveis são expressas em termos de restrições lineares. A heurística de *operator-counting* também possilita o desenvolvimento de abordagens alternativas de solução para o problema de planejamento, como por exemplo aquele apresentado por DAVIES et al. (2015).

Em DAVIES et al. (2015), os autores apresentam o resolvedor *OpSeq* que utiliza uma *Decomposição de Benders Baseada em Lógica*: uma abordagem inédita na área de planejamento. O resolvedor proposto divide o problema original de planejamento em dois outros problemas relacionados: um *problema principal*, que é um programa inteiro que utiliza a estrutura de *operator-counting* e gera *contagens de operadores*, isto é, uma atribuição de contagens inteiras para cada operador da tarefa, e um *subproblema* denominado como o *problema de sequenciamento de contagens de operadores*, que verifica se um plano satisfazendo uma contagens de operadores gerada pelo problema principal existe, ou gera uma restrição violada por essa contagem que é utilizada para fortificar o problema principal. O resolvedor *OpSeq* resolve o subproblema de sequenciamento codificando-o em uma fórmula na forma normal disjuntiva e utilizando um resolvedor SAT.

Nesta dissertação, propomos um algoritmo alternativo para a solução do problema de sequenciamento de contagens de operadores, que não utiliza uma formulação SAT como *OpSeq*. A abordagem que apresentamos utiliza um algoritmo de busca em espaço de estados guiado por uma função heurística e abre novas questões de pesquisa relacionadas ao desenvolvimento de funções heurísticas específicas para o problema de sequenciamento de contagens de operadores. Efetivamente, nosso resolvedor denominado *OpSearch* emprega um algoritmo A* modificado que encontra um plano ótimo, ou usa a fronteira da busca, isto é, o conjunto de estados gerados mas ainda não expandidos, para derivar uma restrição violada que informa ao problema principal que a contagem de operadores não é *sequenciável*.

Os resultados obtidos por essa pesquisa mostram que problemas de planejamento, ou especificamente o subproblema de sequenciamento de contagens de operadores, podem ser melhor resolvidos utilizando uma abordagem baseada em busca heurística do que empregando resolvedores SAT. Em geral, nossa abordagem apresenta melhor escalabilidade do que uma abordagem SAT, pode utilizar informações explícitas presentes na fronteira de busca para derivar restrições mais informadas, e é diretamente beneficiada pela utilização de funções heurísticas mais informadas. Os experimentos executados utilizam o conhecido conjunto de instâncias da Competição Internacional de Planejamento, e mostram a superioridade de *OpSearch* em relação à *OpSeq*, já que resolve mais tarefas de planejamento, utiliza menos memória, gera menores restrições e em geral resolve menos problemas de sequenciamento de contagens de operadores.