Sequencing Operator Counts with State-Space Search

Wesley L. Kaizer, Advisor: André G. Pereira and Marcus Ritt
Institute of Informatics
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
{wlkaizer,agpereira,marcus.ritt}@inf.ufrgs.br

***Abstract*—Heuristic search is the most common approach to solve classical planning tasks optimally. *OpSeq* is an innovative approach to solve planning tasks that decomposes the planning task into a *master problem* and a *subproblem*. The master problem generates an assignment of integer counts for each task operator, and the subproblem verifies if a plan satisfying these counts exists. If a plan does not exist, *OpSeq* learns a new constraint to inform the master problem. *OpSeq* solves the subproblem using a SAT solver. In this dissertation, we propose a new solver *OpSearch*: an A*-based approach to sequence operator counts that uses the frontier of the search to learn a constraint on failure. We show that *OpSearch* solves the subproblem better than *OpSeq*. It solves more planning tasks, scales better, and learns constraints more informative than *OpSeq*. We prove that *OpSearch* only learns constraints that maintain all optimal solutions. *OpSearch* extends the research on an entirely new method to solve planning tasks. Also, our solver opens many new lines of research based on decomposition, such as fast and stronger anytime lower bounds, new methods for agile planning, and new approaches to solve diverse planning.**

***Index Terms*—State-Space Search, Operator Counting Framework, Classical Planning, Artificial Intelligence.**

I. About this Thesis

a) *Student Level:* MSc.
b) *Date of Conclusion:* 11/03/2020.
c) *Examining Board Members:*

- Prof. PhD. Felipe Meneguzzi, Catholic Pontifical University of Rio Grande do Sul, Brazil.
- Prof. PhD. Levi Lelis, Federal University of Viçosa, Brazil.
- Prof. PhD. Rafael Coelho, Federal University of Rio Grande do Sul, Brazil.

d) *Full Thesis:* https://bit.ly/2N2dfiM
e) *Publication*: Kaizer, W. L., Pereira, A. G., Ritt, M. Sequencing Operator Counts with State-Space Search. International Conference on Automated Planning and Scheduling, pp. 166-174, 2020 (Qualis A2).
https://aaai.org/ojs/index.php/ICAPS/article/view/6658/6512

II. Introduction

Automated planning is a general problem solving technique that aims to find a *sequence of operators* called *plan*, whose the ordered application of its operators in the *initial state* achieves a *goal state* that satisfies the goal condition. There is only one initial state but possibly many goal states. States describe currently valid conditions and consist of a complete assignment of discrete values to every task variable, and operators with non-negative costs describe actions that can be applied in states to modify the value assigned to the variables, generating a new state. A *planning task* consists of a set of variables, a set of operators, an initial state, and a goal condition. In *optimal classical planning* the operators have discrete, deterministic and fully observable effects and the objective is to find a plan with global minimal cost or show that such a plan does not exist. The cost of a plan is computed as the sum of the costs of its operators [24].

As an illustrative example, consider the planning task Π_{robot} presented in Fig. 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*.

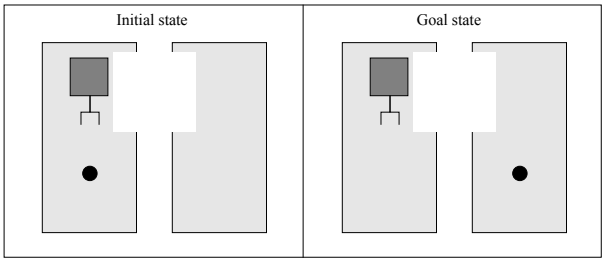

Fig. 1. Example planning task Π_{robot}.

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 Fig. 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.

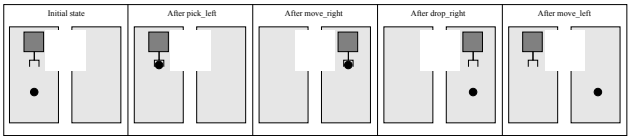

Fig. 2. Plan to task Π_{robot}.

Optimal classical planning can be solved by *domain-independent* algorithms that use as input only a high-level compact specification of the task, since in this setting the planner does not have any additional information about the specific task. The expansion of this compact representation of the planning task generates state-spaces exponentially larger than the input, which requires intelligent and efficient algorithms to find solutions in a feasible time. Planning aims to create one planner that performs sufficiently well on many application domains, including future ones. Many computational problems can be represented as planning tasks, which motivates its use in many application domains, such as robot navigation, activities scheduling, tasks automation, puzzle solving, content generation, goal recognition, and others [1], [2].

The main technologies used by planners to solve planning tasks rely on *heuristic search algorithms* to efficiently expand the state-space aiming to find an optimal solution. A* is one of the most prominent heuristic search algorithm in classical optimal planning and in the area of artificial intelligence in general. It uses three functions f, g and h to map states to numerical values during the search: function $g(s)$ is the cost of the best-known plan from the initial state to state s, function $h(s)$ is the *heuristic function* that estimates the cost of an optimal plan from s to some goal state, and function $f(s) = g(s) + h(s)$ estimates the cost of an optimal plan from the initial state to a goal state, going through state s. Finding optimal solutions to planning tasks is a PSPACE-complete problem and hence intractable in general. However, heuristic search algorithms with automatically derived heuristic functions, e.g., *pattern databases* h^{PDB}, h^{LMCut} and h^{SEQ}, have achieved notable progress by solving many hard planning tasks in practice.

One such heuristic function is the *operator-counting* that uses as heuristic value the objective value of the linear relaxation of an integer program [19]. The integer program contains one integer variable for each task operator and a set of *operator-counting constraints*. The counts assigned by the integer program solution to each operator, denominated *operator counts*, can be understood as an estimate of the number of times the operators are applied to solve the task. [8] introduces a novel approach observing that the operator counts of the operator-counting integer program contain useful information that can be understood as a possibly incomplete and unordered plan. The proposed method decomposes the process of solving a planning task into two independent but related problems using *Logic-Based Benders Decomposition*. The *master problem* corresponds to the operator-counting heuristic integer program. The *subproblem* is modeled as a *propositional satisfiability problem* (SAT) encoding the planning task, and the operator counts obtained from the master. A SAT solver is then used to try to *sequence* the operator counts, i.e., to check if a plan with these counts exists. If the operator counts is not sequenceable, the SAT solver returns a constraint for the master problem.

In this dissertation, 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.

III. Background

A. SAS$^+$ Planning Task

An *SAS$^+$ planning task* $\Pi = \langle \mathcal{V}, O, s_0, s_*, c \rangle$ is defined by a set of *variables* $\mathcal{V}$, a set of *operators* O, an *initial state* s_0, a *goal* condition s_*, and a cost function c. Each variable $v \in \mathcal{V}$ has a finite domain $D(v)$ that describes the possible values v can assume. A *partial state* s is a partial assignment over some subset of $\mathcal{V}$ and a *state* s is a complete assignment over all variables in $\mathcal{V}$. 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 a state and s_* is a partial state. We call a state s consistent with 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' with updated values for $v \in vars(post(o))$, 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 a non-negative cost $c(o)$ to each operator $o \in O$. 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 cost of an s-plan 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 planning task optimally means to find a plan π for Π of minimal cost or prove that no plan exists.

B. Heuristic Search

A* is the most prominent heuristic search algorithm in classical planning [11]. It systematically expands nodes from a set of *open* nodes in order of non-decreasing f-values. The f-value of a state s estimates the cost of a plan going through s and is defined as $f(s) = g(s) + h(s)$, where $g(s)$ is the current cost from s_0 to s and $h(s)$ is a heuristic estimate of the remaining cost to some goal state. Expanded nodes are stored in a *closed* set. A *heuristic function* $h : S \to \mathbb{R} \cup \{\infty\}$ maps a state s to its h-value, an estimate of the cost of an s-plan. The *perfect heuristic* h^* maps a state s to its optimal plan cost or to ∞ if no plan exists. A heuristic is *admissible* if it is a lower bound on the optimal plan cost, i.e., $h(s) \leq h^*(s)$ for all $s \in S$. A* is itself admissible, i.e., always returns a cost-optimal plan, when using an admissible h, if a plan exists.

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* [17], [25]–[29].

C. Integer Programming

Integer programming [22] is an optimization technique aiming to find feasible values for a set of decision variables that optimizes some linear objective function, subject to a set of linear constraints, where some variables can assume only integer values. The problem of finding an optimal solution to an integer program (IP) is NP-complete, but its linear program (LP) relaxation, which ignores the integrality constraints, can be solved in polynomial time. Early uses of linear programming in cost-optimal planning relate to *cost-partitioning*, a method to admissibly combine several heuristics by partitioning operator costs among them [18].

[19] presents 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 [15] and the *dynamic merging* method from [4] based on flow constraints. [19] 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.

D. The Operator-Counting Framework

Operator-counting [19] is a recently proposed framework that unifies information from several conceptually different heuristics into a single integer program. The program contains a variable Y_o, for each operator $o \in O$, that counts the number of occurrences of the operator o in some plan. Its objective function is to minimize the total operator costs while satisfying all its operator-counting constraints. Operator-counting constraints and heuristics are defined below as in [19].

Definition 1 (Operator-counting constraints). *Let Π be a planning task with operator set O, and s be a state of Π. 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*. We say that π is an s-plan in Π if it is a valid plan that leads from a state s to a goal s_*. If π is an s-plan, we denote the number of occurrences of operator $o \in O$ in π with Y_o^π. A set of linear inequalities over $\mathcal{Y}$ is called an* operator-counting constraint *for 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 for s where the only common variables between constraints are the operator-counting variables.*

Definition 2 (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 $\mathsf{Y}_o = \mathsf{Y}_o^\pi$ 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 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 can only improve the heuristic estimates at a possibly higher computational cost. There are many available sources of operator-counting constraints proposed in the literature, such as action landmarks, post-hoc

optimization, state equation, network flow, and optimal delete-relaxation h^+ [4], [5], [15], [21].

E. Operator Counts

An operator counts $\mathcal{C}_s : O \to \mathcal{Z}_0^+$ 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.

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 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.

F. Generalized Landmarks

The generalized landmark constraint (GLC) introduced by [8] contains binary variables called *bounds literals* in the form $[\mathsf{Y}_o \geq k_o]$, being 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 $[\mathsf{Y}_{o_1} \geq k_{o_1}] + \cdots + [\mathsf{Y}_{o_n} \geq k_{o_n}] \geq 1$. To satisfy this constraint at least one of the bounds literals must be true.

Definition 3 (Generalized Landmark Constraint). *A generalized landmark constraint L for $A \subseteq O \times \mathcal{Z}^+$ for a state s in planning task Π is defined as:*

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

Domain constraints are used to link bounds literals with 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 o.

G. Planning using Logic-Based Benders Decomposition

Usually in classical planning, only the objective function value of the operator-counting heuristic guides the search. Even though linear programs are polynomial-time solvable, one must use them as heuristic function carefully, since the state-space expanded by A* grows exponentially in the number of states, according to the planning task specification. Besides the objective function value, the operator counts obtained from the primal solution of IP_C can also contain useful information to solve the problem. Specifically, these counts can be interpreted as a possibly *incomplete and unordered plan* to the planning task, with some missing necessary operators. This observation suggests a novel approach to solve planning tasks optimally, by incrementally search for missing operators until a complete and ordered plan is found.

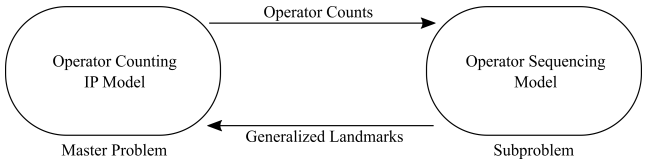

Fig. 3. Logic-Based Benders Decomposition to cost-optimal planning (adapted from [8]).

The Logic-Based Benders Decomposition proposed by [8] decomposes the process of solving planning tasks into two related problems: a master problem that solves IP_C, that is a relaxation of the original planning task and generates operator counts $\mathcal{C}_s$, and a subproblem that tries to sequence $\mathcal{C}_s$, constructing 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 Branch and Cut algorithm (BC) [30] used to solve the IP_C from master proves the optimality of the current incumbent plan. Fig. 3 illustrates the overall process.

This decomposition establishes an interface between operator-counting heuristics and operator counts sequencing procedures. In the next section we discuss how [8] solve the sequencing subproblem.

H. Sequencing Operators Counts with SAT

The solver *OpSeq* introduced by [8] applies a SAT model that encodes the planning task limited to an operator counts $\mathcal{C}_s$ as a formula in 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 that relates to the current operator counts. The generated explanation is a disjunction of negated assumptions that can be directly translated to a generalized landmark constraint and added to the master problem.

OpSeq does not solve the entire operator-counting IP_C at each step of their Logic-Based Benders Decomposition. 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 and ignoring it otherwise. Consequently, it is able to generate violated constraints that also remove relaxed solutions. Most IP solvers

support the definition of control callbacks to dynamically interact with the optimization procedure. *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 derive feasible plans.

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 BC, *OpSeq* solves this by adding the weak bounds literal v/k corresponding to the missing $[v \geq k]$. 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.

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 available in the operator counts. It constructs a set of assumptions about a feasible plan using the current operator counts and Y_T and informs the solver to use these assumptions while searching for a solution. 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 [31] 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 [8] 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 [23]; and $prod(\langle v = x \rangle)$ denotes the set of operators that are producers of atom $\langle v = x \rangle$:

$$\leq_1 (\{o_l | o \in O\}); \tag{1}$$

$$\forall v \in \mathcal{V}: \quad \leq_1 (\{v =_l x_i | x_i \in D(v)\}); \tag{2}$$

$$\forall \langle v = x \rangle \in s_0: \quad v =_0 x; \tag{3}$$

$$\forall o \in O: \quad \bigwedge_{v=x \in pre(o)} (\neg o_l \vee v =_{l-1} x); \tag{4}$$

$$\forall o \in O: \quad \bigwedge_{v=x \in post(o)} (\neg o_l \vee v =_l x); \tag{5}$$

$$\forall \langle v = x \rangle \in \mathcal{V}: \quad v =_{l+1} x \implies v =_l x \vee \bigvee_{o \in prod(\langle v=x \rangle)} o_{l+1}; \tag{6}$$

$$\forall \langle v = x \rangle \in s_*: \quad v =_L x \vee [\Sigma \mathcal{C}(o) \geq L + 1]; \tag{7}$$

$$\forall o \in O: \quad \leq_{\mathcal{C}(o)} (\{o_l | l \in [1, L]\}) \vee [Y_o \geq \mathcal{C}(o) + 1]. \tag{8}$$

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.

IV. Contributions of this dissertation

We propose a solver denominated *OpSearch*, which uses the A* search algorithm to solve the operator counts sequencing subproblem. Given an initial operator counts $\mathcal{C}_{s_0}$, it returns a plan π if $\mathcal{C}_{s_0}$ is sequencable, or a violated condition as a generalized landmark constraint L, otherwise. The master problem IP$_C$ is unchanged. The presence of potentially useful information in the search graph, such as f-values, motivates its use as base for an alternative algorithm. This approach could generate smaller and more informed constraints and, as observed by [7], eliminating irrelevant parts of constraints can significantly decrease solving time of an integer program.

Our approach follows the main idea of planning using Logic-Based Benders Decomposition. We initiate the process using a BC to solve the IP$_C$. If BC finds an integer solution it calls *OpSearch* and we try to sequence the corresponding operator counts. If BC finds a relaxed solution we obtain a valid operator counts by rounding up the primal solution values to the nearest integers, and sequencing only if its cardinality and objective value are within 20% of the linear count. This process is also applied in *OpSeq*. If the operator counts provided is sequencable, *OpSearch* 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. Extended State and Generation of Successors

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

States generated through different sequences of operators are considered different states by *OpSearch*, i.e. it is able to distinguish between two states with identical values assignment for the original task variables but with different operator budgets. Given the current operator counts for the initial state

$\mathcal{C}_{s_0}$ we extend the A* state representation with a variable v_o for each $o \in O$ if $\mathcal{C}_{s_0}(o) > 0$ and $c(o) > 0$. The domain of v_o is $D(v_o) = \{0, \ldots, \mathcal{C}_{s_0}(o)\}$. The number of distinct operators with counts greater than zero in the operator counts $\mathcal{C}_{s_0}$ gives the bound on the number of extra variables necessary to represent subsets of $\mathcal{C}_{s_0}$ and each variable corresponding to operator o is bounded by the initial operator counts $\mathcal{C}_{s_0}(o)$, since the operator counts can only decrease during search.

The example task Π_1 would be changed by including a variable v_{o_1} with domain $D(v_{o_1}) = \{0, 1\}$, but no variable for o_2 or o_4 since their counts are zero, or for o_3 since $c(o_3) = 0$. The value of v_o in s_0 is $\mathcal{C}_{s_0}(o)$. Therefore, our final extended representation for state s_0 would be $\{v_1 = 1, v_{o_1} = 1\}$. 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.

This new state representation requires another modification in the behavior of A*, which needs to consider the extended 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 $s'(v_o) = s(v_o) - 1$. Our approach applies zero-cost operators independently of $\mathcal{C}_{s_0}$ 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. In the example task Π_1, *OpSearch* would generate two states from s_0: state $s' = \{v_1 = 2, v_{o_1} = 0\}$ with operator o_1 and state $s'' = \{v_1 = 1, v_{o_1} = 2\}$ with operator o_3. No state is generated from the application of operator o_4, since it is not contained in $vars(s)$.

B. Constraint Generation Strategy

We now explore the situation when $v_o \notin vars(s) \wedge c(o) > 0$ and $s(v_o) = 0$ to derive some violated condition on the current operator counts. This condition is modeled as a generalized landmark constraint with bounds literals for operator o and can be interpreted as follows: if we had one more instance of o, we could further expand 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 generalized landmark constraint L, that includes at most one bounds literal for each operator. The strategy returns bounds for operators that currently have count 0 but might generate new states with an f-value at most f_{max}. The f_{max} is the objective value of the relaxation of the node in the BC tree, that is found while solving the operator-counting IP. This node calls the the sequencing subproblem informing f_{max} and the operator counts $\mathcal{C}_{s_0}$. State s denotes a state expanded by A* and s' is a generated one.

$$L = \{\, [Y_o \geq \mathcal{C}_{s_0}(o) + 1] \mid \exists s \xrightarrow{o} s' : f(s') \leq f_{\text{max}} \wedge ((v_o \notin vars(s) \wedge c(o) > 0) \vee s(v_o) = 0)\}$$

Further, if the f-value is more 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 $f_{\text{min}} = \min_{s' | f(s') > \mathsf{Y}_f}\{f(s')\}$, 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 $f_{\text{min}} > -\infty$, 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 the operator counts is not sequencable.

To illustrate the solving process of *OpSearch*, we define an example planning task Π_2 with $O = \{o_0, o_1, o_2, o_3, o_4, o_5\}$ and costs $c(o_0) = 0$, $c(o_1) = c(o_2) = c(o_3) = 1$, $c(o_4) = 2$ and $c(o_5) = 0$. We assume that o_1 is an action landmark for Π_2 and the initial operator-counting IP$_C$ contains the constraint $\mathsf{Y}_{o_1} \geq 1$. The primal solution for this IP$_C$ provides the initial operator counts $\mathcal{C}_{s_0} = \{o_1 \mapsto 1\}$ and the objective function value gives the $f_{\text{max}} = 1$. Fig. 4 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 $\mathcal{C}_{s_0}$. Nodes and operators that cannot be generated or applied during the search are dashed. Goals are indicated by doubly circled vertices.

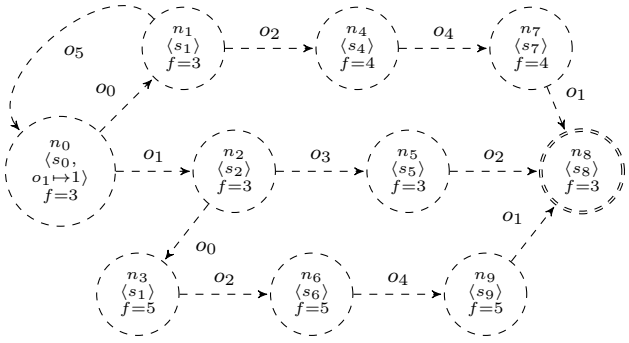

Fig. 4. State-space of example problem Π_2, 1st iteration.

Since $f(n_0) > \mathsf{Y}_f$, *OpSearch* generates the constraint $[\mathsf{Y}_f \geq 3] \geq 1$ informing that the f-value bound f_{max} must increase to 3. Assume now that after adding this constraint the master returns $\mathcal{C}_{s_0} = \{o_1 \mapsto 3\}$ and $\mathsf{Y}_f = 3$. The resulting state-space is illustrated in Fig. 5:

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

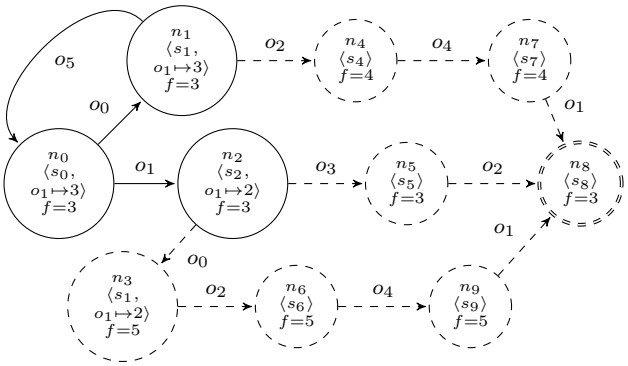

Fig. 5. State-space of example problem Π_2, 2nd iteration.

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 $\mathcal{C}_{s_0} = \{o_1 \mapsto 2, o_3 \mapsto 1\}$ and $\mathsf{Y}_f = 3$. The resulting state-space is shown in Fig. 6:

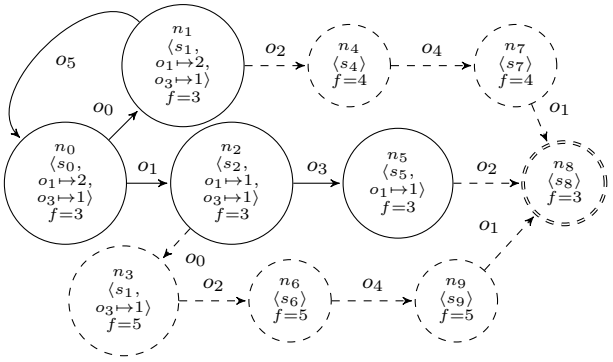

Fig. 6. State-space of example problem Π_2, 3rd iteration.

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 $\mathcal{C}_{s_0} = \{o_1 \mapsto 1, o_2 \mapsto 1, o_3 \mapsto 1\}$ and $\mathsf{Y}_f = 3$, as illustrated in Fig. 7.

In this dissertation, we also prove that our approach *OpSearch* generates admissible constraints, i.e., that do not remove feasible solutions, from non-sequencable operator counts. The resulting theorem is presented below.

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.*

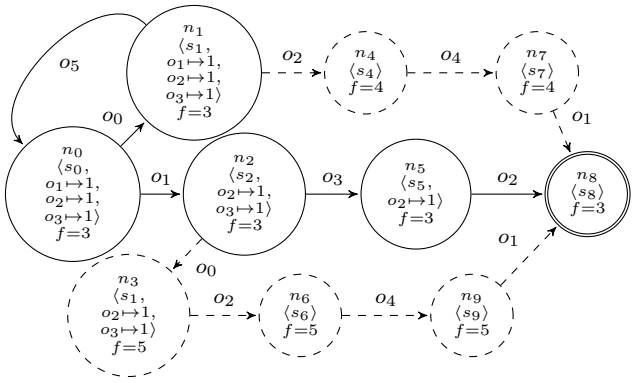

Fig. 7. State-space of example problem Π_2, 4th iteration.

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^*. □

C. 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 [8]: 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 Domain Transition Graph (DTG) for this example is illustrated in Figure 8. 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.

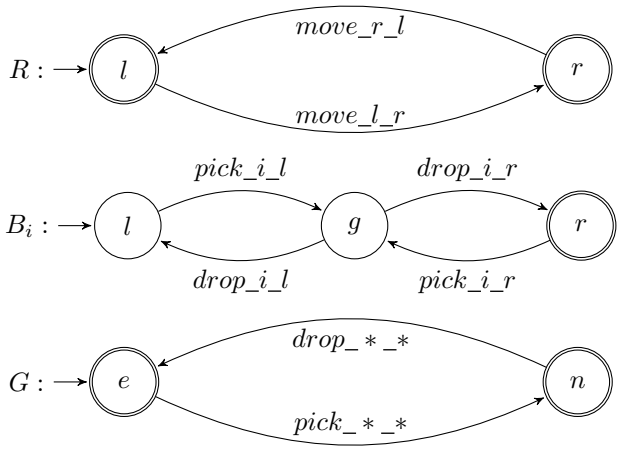

Fig. 8. DTGs for the gripper example introduced by [8].

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} [4], state equation h^{SEQ} [5], and the optimal delete relaxation h^+ [15].

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: $[\mathsf{Y}_f \geq 7] \geq 1$.

V. Experimental Evaluation

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 GLCs in terms of average number of operators included.

We use the same benchmarks from IPC-2011 used by [8], totaling 11 domains with 20 instances each. We used 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 [13]. For solving the satisfiability subproblems we use the MiniSat 2.2 solver [10]. We opted to used MiniSat to facilitate comparisons with [8] but we could use any SAT solver with support to conflict analysis. The IP solver used to solve linear programs is CPLEX 12.8.

The initial IP$_C$ contains constraints from the disjunctive action landmarks h^{LMC} [4], state equation h^{SEQ} [5] and the optimal delete relaxation h^+ base formulation from [15]. We use h^{LMCut} to guide *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 [8]. The source code for *OpSearch* and our version of *OpSeq* is publicly available to facilitate further development of operator counts sequencing procedures. The data and scripts used to generate the tables and figures presented are also publicly available.

A. The Benchmark Set

Table I 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; $\overline{z}_0$ 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.

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*.

B. 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 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

TABLE I
INFORMATION ABOUT THE BENCHMARK SET.

domain	$\overline{\lvert\mathcal{V}\rvert}$	$\overline{\lvert O\rvert}$	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

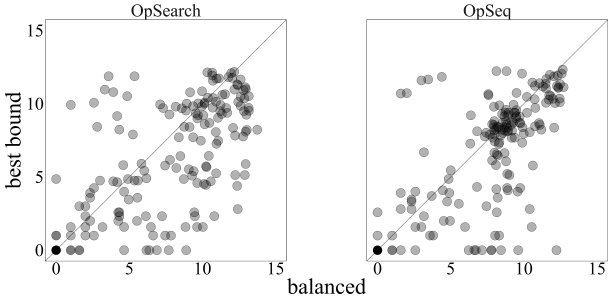

Fig. 9. IP emphasis (log2-log2 scale).

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 default setting "balanced" the solver tries to balance progress on good feasible solution 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. Fig. 9 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.

C. OpSearch *is Better than* OpSeq

Table II shows results grouped by domain for *OpSeq* and *OpSearch*. Best results are highlighted. Column C presents the coverage for that particular domain; S the total number of sequencing calls; R the total number of restarts; $\overline{T_t}$ the mean total solving time in seconds; $\overline{M}$ the mean memory usage in MB; $\overline{u}$ 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 is the best bound found by the IP solver. Since it is not possible to dynamically allocate new variables during the BC, the linear model IP_C has a limited number of bounds literals, up to $k = 2$, for each operator $o \in O$ and for Y_f. However, it can be necessary to add new bounds literals during BC due to the learned GLCs. In this case, both *OpSeq* and *OpSearch* rebuild the model and restart BC.

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.

Fig. 10 shows 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 solve 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*.

Table III summarises the results only considering 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, generates smaller constraints than *OpSeq*, but spends more time sequencing, as indicated by $\bar{p}$.

D. Better Heuristics Improve OpSearch *Performance*

Table IV shows results for *OpSearch* using different heuristic functions in A*. We have tested the h^{blind}, h^{LMCut} and operator-counting h^{OC} with constraints from state-equation and action landmarks. We have chosen these functions because

TABLE II
RESULTS FOR *OpSeq* AND *OpSearch*.

	OpSeq								*OpSearch*							
domain	C	S	R	$\bar{T}_t$	$\bar{M}$	$\bar{u}$	$\bar{p}$	bb	C	S	R	$\bar{T}_t$	$\bar{M}$	$\bar{u}$	$\bar{p}$	bb
barman	**0**	40556	16	**3417**	857	20	**0.1**	2484	**0**	**36565**	**1**	3548	**202**	**5**	**0.1**	**2496**
elevators	**0**	**5922**	**0**	**3275**	2931	17	0.8	690	**0**	10802	7	3555	**254**	**4**	**0.2**	**865**
nomystery	0	**3660**	**0**	1459	736	44	0.6	437	**3**	10383	4	**1120**	**322**	**1**	**0.1**	**443**
openstacks	0	24383	**3**	1709	**433**	29	**0.1**	20	**13**	**266**	14	**966**	968	**0**	23.6	**67**
parcprinter	**20**	**21**	**0**	**1**	**126**	**0**	74.0	**8524162**	16	**21**	**0**	271	377	**0**	**55.9**	**8524162**
pegsol	**11**	22998	15	1964	175	47	**0.0**	154	10	**12906**	**2**	**1888**	**123**	**16**	**0.0**	**166**
scanalyzer	0	3377	**0**	1305	**955**	18	**0.0**	585	**1**	**700**	5	**1001**	1046	**2**	**0.0**	**592**
sokoban	0	**7907**	**0**	3208	2385	9	0.8	319	**5**	17695	51	**2779**	**183**	**3**	**0.2**	**455**
transport	**0**	5800	**0**	1879	265	8	**0.0**	**6251**	**0**	**910**	11	**1707**	**222**	**1**	**0.0**	6235
visitall	**15**	**5632**	**0**	**957**	298	**19**	0.2	**848**	14	9078	10	1112	**119**	29	**0.0**	839
woodworking	**17**	946	**0**	**437**	355	**4**	**0.5**	**6348**	11	**111**	**0**	974	**224**	5	43.7	6258
Total	63	121202	**34**	1783	865	20	**0.4**	8542298	**73**	**99437**	105	**1720**	**367**	**6**	4.4	**8542578**

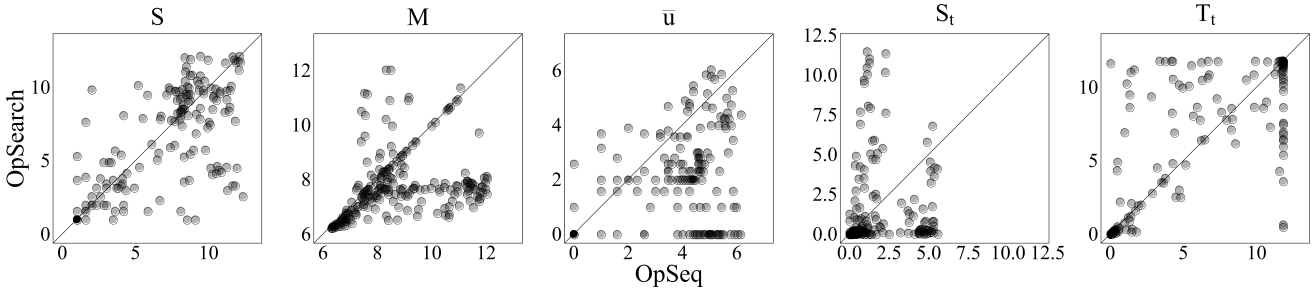

Fig. 10. Detailed comparison between *OpSeq* and *OpSearch* (log2-log2 scale).

TABLE III
SUMMARY FOR 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**

TABLE IV
SUMMARY FOR 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

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.

In general 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 on average, as indicated by $\bar{u}$.

Table V shows results for *OpSearch* using all the 282 instances from IPC-1998 to IPC-2014 in which h^* can be computed by a full *pattern database* (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

TABLE V
SUMMARY FOR 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

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, as indicated by 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 $\bar{u}$.

VI. FINAL REMARKS

A. Publication

The paper *Sequencing Operator Counts with State-Space Search* describing the details of the proposed approach and presenting the main results of the dissertation was accepted for publication at the International Conference on Automated Planning and Scheduling (ICAPS 2020) (Qualis A2). ICAPS the most important and internationally recognized conferences on planning and search.

B. Research

Our work contributes to the advancement in planning research by extending several awarded papers published in important international journals and conferences such as [19], that received the *ICAPS 2014 Outstanding Paper Award*; the doctoral thesis [3], that received the awards *ICAPS 2018 Best Dissertation Award* and *EurAI Artificial Intelligence Dissertation Award 2017*; and [8], that received the *ICAPS 2015 Outstanding Paper Award*.

C. Impact

This dissertation presents theoretical and experimental contributions to the operator counts sequencing problem, which is a novel research problem with great potential of development in both areas of *operations research* and *artificial intelligence*. Our approach opens new research directions towards specialized methods or heuristics to the operator counts sequencing problem and establishes an interface to the use of well-known technologies from the planning community and combinatorial optimization. Finally, we made the source code for *OpSearch* and our version of *OpSeq* publicly available to facilitate further research in this topic.

D. Conclusion and Future Research

We showed that our approach is better than the previous state-of-the-art method based on SAT in several metrics of practical interest, such as the solving time and the memory consumption. We are jointly researching in this topic with the most recognized planning group (University of Basel). We aim to submit our new contributions to relevant journals and conferences.

E. Applications

Our research results are relevant in practical applications besides the further development of automated planning, that is a general technique to problem solving. Furthermore, *OpSearch* can be used as an anytime method to obtain lower-bounds on plan costs, in *agile planning* to solve solve planning tasks for which informative heuristics are already known. Another practical application of our approach is for *diverse planning*, used for example by IBM, that aims to find several plans while guaranteeing diversity. With our approach, it is possible to iteratively generate constraints from non-sequencable operator counts that guarantee a minimum of diversity.

REFERENCES

[1] Pereira, R. F., Pereira, A. G., and Meneguzzi, F. 2019. Landmark-Enhanced Heuristics for Goal Recognition in Incomplete Domain Models. In *International Conference on Automated Planning and Scheduling*, 329–337.

[2] Bento, D. S., Pereira, A. G., and Lelis, L. H. S. 2019. Procedural Generation of Initial States of Sokoban. In *International Joint Conferences on Artificial Intelligence Organization*, 4651–4657.

[3] , Pommerening, F. 2017. New perspectives on cost partitioning for optimal classical planning. In University of Basel

[4] Bonet, B., and van den Briel, M. 2014. Flow-based heuristics for optimal planning: Landmarks and merges. In *International Conference on Automated Planning and Scheduling*, 47–55.

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

[6] Bäckström, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11(4):625–655.

[7] Ciré, A.; Coban, E.; and Hooker, J. N. 2013. Mixed Integer Programming vs. Logic-Based Benders Decomposition for Planning and Scheduling. In Gomes, C., and Sellmann, M., eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 325–331. Berlin, Heidelberg: Springer Berlin Heidelberg.

[8] Davies, T. O.; Pearce, A. R.; Stuckey, P. J.; and Lipovetzky, N. 2015. Sequencing operator counts. In *International Conference on Automated Planning and Scheduling*, 61–69.

[9] Edelkamp, S. 2014. Planning with pattern databases. In *European Conference on Planning*, 84–90.

[10] Eén, N., and Sörensson, N. 2003. An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*, 502–518. Springer.

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

[12] Helmert, M.; Haslum, P.; Hoffmann, J.; et al. 2007. Flexible abstraction heuristics for optimal sequential planning. In *International Conference on Automated Planning and Scheduling*, 176–183.

[13] Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

[14] Hooker, J. N., and Ottosson, G. 2003. Logic-based Benders decomposition. *Mathematical Programming* 96(1):33–60.

[15] Imai, T., and Fukunaga, A. 2014. A practical, integer-linear programming model for the delete-relaxation in cost-optimal planning. In *European Conference on Artificial Intelligence*, 459–464.

[16] Karmarkar, N. 1984. A new polynomial-time algorithm for linear programming. *Combinatorica* 4:373–395.

[17] Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *International Joint Conference on Artificial Intelligence*, 1728–1733.

[18] Katz, M., and Domshlak, C. 2008. Optimal additive composition of abstraction-based admissible heuristics. In *International Conference on Automated Planning and Scheduling*, 174–181.

[19] Pommerening, F.; Röger, G.; Helmert, M.; and Bonet, B. 2014. LP-based heuristics for cost-optimal planning. In *International Conference on Automated Planning and Scheduling*, 226–234.

[20] Pommerening, F.; Röger, G.; and Helmert, M. 2013. Getting the most out of pattern databases for classical planning. In *International Joint Conference on Artificial Intelligence*, 2357–2364.

[21] van den Briel, M.; Benton, J.; Kambhampati, S.; and Vossen, T. 2007. An LP-based heuristic for optimal planning. In *International Conference on Principles and Practice of Constraint Programming*, 651–665. Springer.

[22] Wolsey, L. 1998. *Integer Programming*. Wiley Series in Discrete Mathematics and Optimization. Wiley.

[23] Sinz, C. 2005. Towards an optimal CNF encoding of boolean cardinality constraints. In *International Conference on Principles and Practice of Constraint Programming*, 827–831. Springer.

[24] Ghallab, M.; Nau, D.; and Traverso, P. 2004 *Automated Planning: Theory and Practice* Elsevier

[25] Geffner, P.; Haslum H.; and Haslum, P. 2000 Admissible heuristics for optimal planning *International Conference of AI Planning Systems*, 140–149

[26] Hoffmann, J.; and Nebel, B. 2001 The FF planning system: Fast plan generation through heuristic search *Journal of Artificial Intelligence Research*, 253–302

[27] Bonet, B.; Geffner, H. 2001 Planning as heuristic search *Artificial Intelligence*, 5–33

[28] Edelkamp, S. 2014 Planning with pattern databases *European Conference on Planning*, 84–90

[29] Helmert, M.; and Domshlak, C. 2009 Landmarks, critical paths and abstractions: what's the difference anyway? *International Conference on Automated Planning and Scheduling*, 162–169

[30] Mitchell, J. 2002 Branch-and-cut algorithms for combinatorial optimization problems *Handbook of Applied Optimization*, 65–77

[31] Marques-Silva, J.; and Sakallah, K. 1999 GRASP: A search algorithm for propositional satisfiability *IEEE Transactions on Computers*, 506–521 IEEE