Sequencing Operator Counts with State-Space Search

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

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 to solve planning tasks optimally. In this approach, 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 necessary violated constraint to strengthen the master problem. In OpSeq the subproblem is solved by a SAT solver. In this paper we show that operator counts sequencing can be better solved by state-space search. We introduce OpSearch, an A*-based algorithm to solve the operator counts sequencing subproblem: it either finds an optimal plan, or uses the frontier of the search to derive a violated constraint. We show that using a standard search framework has two advantages: i) 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. We present results on the IPC-2011 benchmarks showing that this approach solves more planning tasks, using less memory. On tasks solved by both methods OpSearch usually requires to solve fewer operator counts sequencing problems than OpSeq, evidencing the stronger constraints generated by OpSearch.

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 and Nebel 1995). However, heuristic search algorithms such as A* (Hart, Nilsson, and Raphael 1968) with automatically derived heuristic functions (heuristics) – e.g., pattern databases (Edelkamp 2014) and merge-and-shrink (Helmert et al. 2007) – have achieved notable progress. A* with these strong heuristics can search large state-spaces efficiently, solving many planning tasks in practice.

Many recently proposed heuristics are based on linear programming optimization. The operator-counting framework (Pommerening et al. 2014) is of particular interest because it combines in a declarative form the information of many admissible heuristics by 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 and van den Briel 2014), state equation $h^{SEQ}$ (Bonet 2013), and the optimal delete relaxation $h^+$ (Imai and Fukunaga 2014).

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 possible 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 subproblems, in a similar way to Logic-Based Benders Decomposition (Hooker and Ottosson 2003). There is a master problem and a combinatorial subproblem used to explain the infeasibility of a solution of the master problem. The master problem is modeled as an integer program, corresponding to an operator-counting heuristic. The subproblem is modeled as a propositional satisfiability (SAT) problem encoding the planning task and the operator counts obtained from the 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 paper, we 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 operator-counting
constraint. We show experimentally that the resulting algorithm OpSearch has better coverage and less computational 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 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 technology.

Background

**SAS** Planning Task** An SAS** planning task of the form $\Pi = (V, O, s_0, s_*, c)$ is defined by a set of variables $V$, a set of operators $O$, an initial state $s_0$, a goal condition $s_*$, and a cost function $c$. Each variable $v \in V$ has a finite domain $D(v)$. A partial state $s$ is a partial assignment over $V$ and a state $s$ is a complete assignment over $V$. We write $vars(s)$ for the set of variables in state $s$, $s(v)$ for the value of variable $v$ in $s$, and $S$ for the set of all states of $\Pi$, 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 $(pre(o), post(o))$. 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 $v \in vars(post(o))$.

**Heuristic Search** $A^*$ is the most prominent heuristic search algorithm in classical planning (Hart, Nilsson, and Raphael 1968). 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 $h : S \rightarrow \mathbb{R} \cup \{\infty\}$ 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 $\infty$ 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 heuristic function $h$, if a plan exists.

**Integer Programming** Integer programming (Wolsey 1998) 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 (Karmarkar 1984). 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 (Katz and Domshlak 2008; Karpas and Domshlak 2009).

**The Operator-Counting Framework** Operator-counting (Pommerening et al. 2014) 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 Pommerening et al. (2014).

**Definition 1 (Operator-counting constraints).** Let $\Pi$ be a planning task with operator set $O$, and $s$ be a state of $\Pi$. Let $Y$ be a set of non-negative valued operators, 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 $\pi$ is an $s$-plan in $\Pi$ if it is a valid plan that leads from a state $s$ to a goal $s_*$. If $\pi$ is an $s$-plan, we denote the number of occurrences of operator $o \in O$ in $\pi$ with $Y^\pi_o$. A set of linear inequalities over $Y$ is called an operator-counting constraint for $s$ if for every $s$-plan there exists a feasible solution with $Y_o = Y^\pi_o$ 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

$$\begin{align*}
\text{minimize} & \quad \sum_{o \in O} c(o)Y_o \\
\text{subject to} & \quad C, \\
& \quad Y_o \in \mathbb{Z}^+.
\end{align*}$$

The IP heuristic $h^*_{IP}$ is the objective value of $IP_C$, and the LP heuristic $h^*_{LP}$ is the objective value of its linear relaxation. If the IP or LP is infeasible, the heuristic estimate is $\infty$.

If $\pi$ is a plan for $\Pi$ then $Y^\pi_o = Y^\pi_o$ is a solution for $IP_C$. Thus, the cost of an optimal plan $\pi^*$ is an upper bound for the objective value of $IP_C$, and the LP 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 (Bonet and van den Briel 2014; Pommerening, Röger, and Helmert 2013; Bonet 2013; Imai and Fukunaga 2014; van den Briel et al. 2007).
For example, the operator-counting constraint corresponding to a disjunctive action landmark is a set of operators for a state \( s \) such that every plan from \( s \) must contain at least one operator from the disjunctive action landmark:

**Definition 3.** The operator-counting constraint corresponding to a disjunctive action landmark \( L \subseteq O \) for a state \( s \) of planning task \( \Pi \) is

\[
\sum_{o \in L} Y_o \geq 1.
\]

Since every plan from \( s \) contains at least one operator from \( L \) the constraint is an operator-counting constraint.

**Operator Counts** An operator counts \( C_s : O \rightarrow \mathbb{Z}_0^+ \) is a function that assigns to each operator \( o \in O \) the integer count \( Y_o \) of the primal solution of the operator-counting IP\(_C\) for state \( s \). The total number of operators of an operator counts \( C_s \) is defined as \( |C_s| = \sum_{o \in O} C_s(o) \).

**Generalized Landmarks** Davies et al. (2015) introduced the generalized landmark constraint (GLC) that contains binary variables called "bounds literals" in the form \( \lceil Y_o \geq k_o \rceil \), 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 \( \lceil Y_{o_1} \geq k_{o_1} \rceil + \lceil Y_{o_2} \geq k_{o_2} \rceil + \cdots + \lceil Y_{o_n} \geq k_{o_n} \rceil \geq 1 \). To satisfy this constraint at least one of the bounds literals must be true.

**Definition 4 (Generalized Landmark Constraint).** A generalized landmark constraint \( L \) for \( A \subseteq O \times \mathbb{Z}^+ \) for a state \( s \) in planning task \( \Pi \) is defined as:

\[
\sum_{(o,k) \in A} \lceil Y_o \geq k \rceil \geq 1.
\]

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

\[
\lceil Y_o \geq k \rceil \leq \lceil Y_o \geq k - 1 \rceil, \quad (1)
\]

\[
Y_o \geq \sum_{i=1}^{k} \lceil Y_o \geq i \rceil, \quad (2)
\]

\[
Y_o \leq M \lceil Y_o \geq k \rceil + k - 1. \quad (3)
\]

Constraint (1) ensures that bound \( \lceil Y_o \geq k \rceil \) is only valid when the next smallest bound \( \lceil Y_o \geq k - 1 \rceil \) 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 \( \lceil Y_o \geq k \rceil \) is set when \( Y_o \geq k \). Combined, (2) and (3) guarantee that \( Y_o \) is the number of occurrences of operator \( o \).

**Planning using Logic-Based Benders Decomposition**

Usually in classical planning, only the objective function value of the operator-counting heuristic guides the search. However, variables \( Y_o \) in the primal solution of IP\(_C\) contains useful information. This suggests a novel approach to solve planning tasks optimally.

Davies et al. (2015) propose a Logic-Based Benders Decomposition that decomposes the process of solving planning tasks into two related problems: a master problem that solves IP\(_C\) – a relaxation of the original planning task, which generates operator counts \( C_s \), and a subproblem that tries to sequence \( C_s \), 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 master and the subproblem. The process stops when the Branch and Cut algorithm (BC) from master proves the optimality of the current incumbent plan. Figure 1 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 Davies et al. (2015) solve the sequencing subproblem.

**Sequencing Operators Counts 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 \( C_s \) 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 is not 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 a feasible plan.

The SAT model is composed of layers, and only one operator can be applied in each layer. Thus, its memory usage grows with the total number of layers. OpSeq uses the vari-
able \( Y_T \) 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 generalized landmark constraint based on these assumptions, explaining why the operator counts is not sequencable. This constraint is derived by the Conflict-Directed Clause Learning algorithm implemented in SAT solvers, that backtracks until it reaches to the assumptions that cause the formula’s unsatisfiability.

**Proposed Approach**

We propose a solver \( OpSearch \), which uses the \( A^* \) search algorithm to solve the operator counts sequencing subproblem. Given an initial operator counts \( C_{s_0} \), it returns a plan \( \pi \) if \( C_{s_0} \) is sequencable, or a violated condition as a generalized landmark constraint \( L \), otherwise. 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 Ciré, Coban, and Hooker (2013), 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. 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.

**Extended State and Generation of Successors**

In this section, we use a planning task \( \Pi_1 \) as an example, containing \( \mathcal{V} = \{ v_1 \} \) with \( D(v_1) = \{ 0, 1, 2 \} \). \( O = \{ o_1, o_2, o_3, o_4 \} \), \( o_1 = \langle v_1 = 1, v_1 = 2 \rangle \), \( o_2 = \langle v_1 = 0, v_1 := 2 \rangle \), \( o_3 = \langle v_1 = 1, v_1 := 2 \rangle \), \( o_4 = \langle v_1 = 1, v_1 := 3 \rangle \), \( c(o_1) = 2 \), \( c(o_2) = 0 \), and \( c(o_3) = 1 \), with initial state \( s_0 = \{ v_1 = 1 \} \) and goal state \( s = \{ v_1 = 2 \} \).

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 \( C_{s_0} = \{ o_1 \rightarrow 1 \} \) (we only list non-zero operator counts).

States generated through different sequences of operators are considered different states by \( OpSearch \). Given the current operator counts for the initial state \( C_{s_0} \), we extend the \( A^* \) state representation with a variable \( v_o \) for each \( o \in O \) if \( C_{s_0}(o) > 0 \) and \( c(o) > 0 \). The domain of \( v_o \) is \( D(v_o) = \{ 0, \ldots, C_{s_0}(o) \} \). The example task \( \Pi_1 \) 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 \( v_o \) with \( s_0 \) is \( C_{s_0}(o) \). 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.

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 \( C_{s_0} \) 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 \( \Pi_1 \), \( OpSearch \) would generate two states from \( s_0 \); state \( s' = \{ v_1 = 2, v_{o_1} = 0 \} \) with operator \( o_1 \) and state \( s'' = \{ v_1 = 1, v_{o_1} = 1 \} \) with operator \( o_3 \). No state is generated from the application of operator \( o_4 \), since it is not contained in \( vars(s) \).

**Constraint Generation Strategy**

We now explore the situation when \( v_o \notin vars(s) \land c(o) > 0 \) 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 \( f_{\text{max}} \), the objective value of the relaxation of the node in the BC tree that called the sequencing subproblem. State \( s \) denotes a state expanded by \( A^* \) and \( s' \) is a generated one.

\[
L = \{ [Y_o \geq C_{s_0}(o) + 1] \mid \exists s' \rightarrow s' : f(s') \leq f_{\text{max}} \land \\
(\forall o_o \notin vars(s) \land c(o) > 0) \lor s(v_o) = 0 \}
\]

Further, if the \( f \)-value is more than \( f_{\text{max}} \) 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

\[
Y_f = \sum_{o_o \in O} c(o)Y_o.
\]
Now let \( f_{\min} = \min_{s'(f(s')) > Y_f} \{ f(s') \} \). Then, if \( f_{\min} > -\infty \), we add the bounds literal \([ Y_f \geq f_{\min} \)] to \( L \).

To illustrate the solving process of OpSearch, we define an example planning task \( \Pi_2 \) with \( O = \{ o_1, o_2, o_3, o_4, o_5 \} \) and costs \( c(o_2) = 0, c(o_1) = c(o_2) = c(o_3) = 1, c(o_4) = 2 \) and \( c(o_5) = 0 \). We assume that \( o_1 \) is an action landmark for \( \Pi_2 \) and the initial operator-counting IP \( \Pi_0 \) contains the constraint \([ Y_{o_1} \geq 1 \] \). The primal solution for this IP \( \Pi_0 \) provides the initial operator counts \( C_{s_0} = \{ o_1 \rightarrow 1 \} \) and the objective function value gives the \( f_{\text{max}} \) is 1. Figure 2 illustrates the state-space generated by \( A^* \) with the perfect heuristic \( h^* \), where vertices represent states and arcs the application of operators. Solid vertices and edges represent nodes and operators that are generated during the search are dashed. Goals are indicated by doubly circled vertices.

Since \( f(n_0) > Y_f \), OpSearch generates the constraint \([ Y_f \geq 3 \] \) informing that the \( f \)-value bound \( f_{\text{max}} \) must increase to 3. Assume now that after adding this constraint the master returns \( C_{s_1} = \{ o_1 \rightarrow 3 \} \) and \( Y_f = 3 \). The resulting state-space is illustrated in Figure 3:

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 \( \{ o_1 \rightarrow 3 \} \) and \( \{ o_1 \rightarrow 2 \} \) and therefore are treated as different states. From this state-space, OpSearch returns the constraint \([ Y_{o_3} \geq 1 \] \) and \( Y_f \geq 4 \). The bound \([ Y_{o_3} \geq 1 \] \) comes from the transition \( n_1 \rightarrow o_1 \rightarrow n_3 \) and \([ Y_f \geq 4 \] \) from \( n_1 \rightarrow o_2 \rightarrow n_4 \), since transition \( n_2 \rightarrow o_3 \rightarrow n_3 \) would generate the bound \([ Y_f \geq 5 \] \). Suppose that, after adding this constraint, the IP \( \Pi_3 \) returns \( C_{s_0} = \{ o_1 \rightarrow 2, o_3 \rightarrow 1 \} \) and \( Y_f = 3 \). The resulting state-space is shown in Figure 4:

From this state-space, OpSearch returns the constraint \([ Y_{o_3} \geq 1 \] \) and \([ Y_f \geq 4 \] \). The bound \([ Y_{o_3} \geq 1 \] \) comes from the transition \( n_2 \rightarrow o_3 \rightarrow n_3 \) and \([ Y_f \geq 4 \] \) from \( n_1 \rightarrow o_2 \rightarrow n_4 \). After adding this constraint, OpSearch returns a sequencable operator counts \( C_{s_0} = \{ o_1 \rightarrow 1, o_2 \rightarrow 1, o_3 \rightarrow 1 \} \) and \( Y_f = 3 \), as illustrated in Figure 5.

Theorem 1. For a solvable SAS+ planning task \( \Pi \), an operator counts \( C_o \) with an associated \( f \)-bound value \( f_{\text{max}} \) such that OpSearch’s modified \( A^* \) with an admisible heuristic function \( h \) cannot sequence \( C_o \), OpSearch always returns an admissible constraint to the master integer program.

Proof sketch. Consider an optimal plan \( \pi^* = \{ o_1, \ldots, o_k \} \)
with a corresponding state sequence \((s_0, s_1, \ldots, s_n)\). Let \(L\) be an GLC generated by \(OpSearch\) with \(C_s\) and \(f\)-bound \(f_{\max}\), and \(S\) be the set of (extended) states expanded by \(OpSearch\). In other words, \(S\) contains only states \(s\) which are \(\pi\)-admissible. Otherwise, there is a predecessor state \(s'\) with \(s' \in S\) and \(\pi\)-admissible. The reason for this is either \(f(s'_i) > f_{\max}\) or \(s(o) = 0\) for \(v_o \notin \text{vars}(s'_i)\). In the first case, the bounds literal \([Y_f \geq f_{\min}]\) must be satisfied, and in the second case the bounds literal \([Y_o \geq C_s(o) + 1]\) must be satisfied by \(\pi^*\).

### Different Heuristic Functions

The heuristic function used in \(\pi^*\) plays an important role in GLCS generation and we expect that \(OpSearch\) with more informed heuristics generates smaller and stronger constraints. To illustrate this we use the 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 at the left room and the goal is to move the balls from the left to right. Operators \(p_{ij}\) and \(d_{ij}\) cause the robot to pick or drop ball \(i\) at room \(j\) and \(m_{ij}\) causes the robot to move from room \(i\) to \(j\). All operators have unit cost and the optimal plan with total cost of 7 is \((p_{11}, m_{12}, d_{12}, m_{21}, p_{21}, m_{12}, d_{21})\).

Assume that \(f_{\max} = 5\) and \(C_s = \{d_{12} \mapsto 1, d_{22} \mapsto 1, m_{12} \mapsto 1, p_{11} \mapsto 1, p_{21} \mapsto 1\}\). We use an IP\(_C\) with constraints from disjunctive action landmarks \(h_{\text{LMC}}\) (Bonet and van den Briel 2014), state equation \(h_{\text{SEQ}}\) (Bonet 2013), and the optional delete relaxation \(h^*\) (Imai and Fukunaga 2014).

### Experiments

The goals of the experiments are: i) to evaluate the performance of \(OpSearch\) compared to \(OpSeq\); ii) to contrast the computational resources required by both approaches; and iii) to experimentally validate the hypothesis that \(OpSearch\) can generate stronger GLCs.

We use the same benchmarks from IPC-2011 used by Davies et al. (2015), 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 (Helmert 2006). The SAT solver is MiniSat 2.2 (Eén and Sörensson 2003) and the IP solver is CPLEX 12.8. Since \(OpSeq\)’s is not publicly available, we re-implemented it. \(OpSearch\) and \(OpSeq\) are available to facilitate future work\(^1\).

The initial IP\(_C\) contains constraints from the disjunctive action landmarks \(h_{\text{LMC}}\) (Bonet and van den Briel 2014), state equation \(h_{\text{SEQ}}\) (Bonet 2013) and the optional delete relaxation \(h^*\) base formulation from (Imai and Fukunaga 2014). We use \(h_{\text{LMCut}}\) to guide \(OpSearch\) when sequencing.

### The Benchmark Set

Table 1 presents information about the benchmark set, summarized by domain. \(|\mathcal{V}|\) denotes the mean number of variables; \(|\mathcal{O}|\) is the mean number of operators; \(z_{\text{ceo}}\) indicates the presence of zero-cost operators; \(\overline{r}_{\text{min}}\) is the mean minimum operator cost, ignoring zero-cost operators; \(\overline{r}_{\text{max}}\) is the mean maximum operator cost; \(l_{\overline{B}}\) is the mean best lower bound on the optimal plan cost; \(\tau_0\) is the mean initial relaxed operator-counting solution of our initial operator-counting master problem; and \(\tau_0\) and \(\tau_0\) are the mean number of rows and columns in the initial IP\(_C\), respectively.

We see that 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 with 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 its very wide operator costs range.

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 also note that \(\tau_0\) is very close to \(l_{\overline{B}}\) in parcprinter, sokoban, transport and visitall. Some domains have huge IP\(_C\) such as visitall and sokoban and others have small ones, for instance, nomystery, parcprinter and transport.

### 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 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. Figure 6 shows plots of the total number of sequence

\(^1\)Available at https://github.com/kaizerw/PlanningLP

\(^2\)Obtained from http://editor.planning.domains
calls, comparing IP emphasis “balanced” to “best bound”. We can see that when the IP emphasis is set to “best bound”, both \textit{OpSearch} and \textit{OpSeq} require fewer sequencing calls than with the “balanced” setting.

**OpSeq and OpSearch**

Table 3 shows results grouped by domain for \textit{OpSeq} and \textit{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; $T_f$ the mean total solving time in seconds; $M$ the mean memory usage in MB; $t$ the mean percentage of operators included in the generated constraints; $\tau$ 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 \textit{OpSeq} and \textit{OpSearch} rebuild the model and restart BC.

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

An important comparison between the solvers is the percentage of operators in the learned constraints. On average, constraints generated by \textit{OpSeq} have 20\% of the operators, while constraints generated by \textit{OpSearch} have only 6\% of the operators. Also, \textit{OpSeq} learns constraints with more than 10\% of the operators on seven domains, while \textit{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.

Figure 6 shows plots comparing the total number of sequencing calls $S$, memory usage $M$, mean percentage of operators by learned constraints $u_i$, total sequence times $t_i$ and total solve time $T_f$ for \textit{OpSearch} and \textit{OpSeq}. Visually, we can confirm the results presented before: i) \textit{OpSearch} solved fewer sequencing subproblems; ii) in most of the times \textit{OpSearch} uses less memory than \textit{OpSeq}; and iii) \textit{OpSearch} usually generates smaller constraints than \textit{OpSeq}. Table 2 summarises the results only considering instances solved by both \textit{OpSearch} and \textit{OpSeq}, showing that \textit{OpSearch} solves fewer subproblems, uses slightly less memory, generates smaller constraints than \textit{OpSeq}, but spends more time sequencing, as indicated by $\bar{p}$.

**Impact of OpSearch’s Heuristic Function**

Table 4 shows results for \textit{OpSearch} using different heuristic functions in $A^*$. We have tested the $h_{\text{blind}}$, $h_{\text{LMCut}}$ and operator-counting $h_{\text{OC}}$ with constraints from state-equation and action landmarks. We have chosen these functions because $h_{\text{OC}}$ is usually more informed than $h_{\text{LMCut}}$ and $h_{\text{blind}}$ is the simplest one.

Using in general more informed heuristic functions in \textit{OpSearch} results in: i) fewer sequencing subproblems

---

Table 1: Information of benchmark set.

| domain     | $|V|$ | $|O|$ | $z_{co}$ | $t_{min}$ | $t_{max}$ | $\bar{t}$ | $\tau$   | $\bar{t}$ |
|------------|-----|-----|----------|-----------|-----------|----------|----------|----------|
| barman     | 53  | 358 | -        | 1         | 10        | 30.15    | 15.75    | 7408.2   |
| elevators  | 40  | 866 | ✓        | 6         | 32        | 3.75     | 1.00     | 12810.3  |
| nomyystery | 34  | 185 | -        | 1         | 1         | 8.85     | 3.92     | 3701.9   |
| openstacks | 108 | 663 | ✓        | 1         | 1         | 123.35   | 76.58    | 14456.3  |
| parcprinter| 59  | 254 | ✓        | 987       | 217007    | 1223929.0| 1223929.0| 4340.6   |
| pegsol     | 12  | 572 | ✓        | 1         | 1         | 59.05    | 34.09    | 8201.0   |
| scanalyzer | 9.7 | 128 | -        | 3         | 521.90    | 295.91   | 26130.9  |
| sokoban    | 7.1 | 130 | ✓        | 1         | 1         | 24.85    | 21.60    | 47324.4  |
| transport  | 38  | 176 | -        | 1         | 1         | 41.80    | 40.78    | 2096.2   |
| visitall   | 15.5| 165 | -        | 1         | 1         | 36.90    | 30.62    | 189001.6 |
| woodworking| 74  | 908 | -        | 5         | 44        | 329.50   | 296.40   | 17438.5  |

Table 2: Summary for 50 instances solved by both methods.

<table>
<thead>
<tr>
<th></th>
<th>$S$</th>
<th>$R$</th>
<th>$T_f$</th>
<th>$M$</th>
<th>$u$</th>
<th>$\bar{p}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{OpSearch}</td>
<td>2169</td>
<td>1</td>
<td>191</td>
<td>118</td>
<td>9</td>
<td>46.4</td>
</tr>
<tr>
<td>\textit{OpSeq}</td>
<td>2738</td>
<td>6</td>
<td>92</td>
<td>122</td>
<td>15</td>
<td>0.3</td>
</tr>
</tbody>
</table>
solved, as indicated by $S$; ii) greater mean total solving times $T_t$ since computing the heuristics are more expensive; iii) less mean memory usage, as indicated by $M$; and iv) smaller constraints are generated on average, as indicated by $\bar{u}$.

Table 5 shows results for $\text{OpSearch}$ 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 $h_{\text{blind}}$, $h_{\text{LMCut}}$, operator-counting $h_{\text{OC}}$ with constraints from state-equation and action landmarks, and $h^*$.

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 $T_t$ for $h_{\text{OC}}$ is twice as much as for the other heuristics; iii) $h^*$ uses much more memory $M$ than the other heuristics due to the full PDB; and iv) on average, smaller constraints are generated by more informed heuristics, as indicated by $\bar{u}$.

Conclusions and Future Work

In this work we introduced $\text{OpSearch}$, a technique inspired by Logic-Based Benders Decomposition that uses an $A^*$-based algorithm to solve the problem of sequencing operator counts. 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 $\text{OpSeq}$, a SAT-based approach to sequencing, solving fewer subproblems and presenting a higher coverage. We also presented results indicating that an approach based on $A^*$ can scale better than $\text{OpSeq}$ in terms of overall memory usage.

Future work could address the development of specific heuristic functions to explore structural properties of the operator counting 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; or improving methods to deal with zero-cost operators to increase the integer program lower bound faster.

Table 3: Results for $\text{OpSeq}$ and $\text{OpSearch}$.

<table>
<thead>
<tr>
<th>Domain</th>
<th>$C$</th>
<th>$S$</th>
<th>$R$</th>
<th>$T_t$</th>
<th>$M$</th>
<th>$\bar{u}$</th>
<th>$\bar{p}$</th>
<th>$bb$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barman</td>
<td>0</td>
<td>22</td>
<td>7</td>
<td>107</td>
<td>6</td>
<td>4</td>
<td>19.1</td>
<td>0.1</td>
</tr>
<tr>
<td>Elevators</td>
<td>0</td>
<td>29</td>
<td>9</td>
<td>106</td>
<td>8</td>
<td>21</td>
<td>22.7</td>
<td>0.5</td>
</tr>
<tr>
<td>Nomystry</td>
<td>0</td>
<td>36</td>
<td>10</td>
<td>106</td>
<td>8</td>
<td>24</td>
<td>27.3</td>
<td>0.2</td>
</tr>
<tr>
<td>Openstacks</td>
<td>20</td>
<td>4</td>
<td>1</td>
<td>106</td>
<td>8</td>
<td>18</td>
<td>19.3</td>
<td>0.1</td>
</tr>
<tr>
<td>Pparseprnt.</td>
<td>20</td>
<td>1</td>
<td>0</td>
<td>106</td>
<td>8</td>
<td>18</td>
<td>19.3</td>
<td>0.1</td>
</tr>
<tr>
<td>Pegsol</td>
<td>11</td>
<td>21</td>
<td>8</td>
<td>106</td>
<td>8</td>
<td>18</td>
<td>19.3</td>
<td>0.1</td>
</tr>
<tr>
<td>Scanalyzer</td>
<td>0</td>
<td>37</td>
<td>6</td>
<td>106</td>
<td>8</td>
<td>18</td>
<td>19.3</td>
<td>0.1</td>
</tr>
<tr>
<td>Sokoban</td>
<td>0</td>
<td>39</td>
<td>9</td>
<td>106</td>
<td>8</td>
<td>18</td>
<td>19.3</td>
<td>0.1</td>
</tr>
<tr>
<td>Transport</td>
<td>0</td>
<td>40</td>
<td>6</td>
<td>106</td>
<td>8</td>
<td>18</td>
<td>19.3</td>
<td>0.1</td>
</tr>
<tr>
<td>Visittall</td>
<td>0</td>
<td>41</td>
<td>6</td>
<td>106</td>
<td>8</td>
<td>18</td>
<td>19.3</td>
<td>0.1</td>
</tr>
<tr>
<td>Woodworking</td>
<td>0</td>
<td>42</td>
<td>6</td>
<td>106</td>
<td>8</td>
<td>18</td>
<td>19.3</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 4: Summary for 49 instances solved by all heuristics.

<table>
<thead>
<tr>
<th>Domain</th>
<th>$C$</th>
<th>$S$</th>
<th>$R$</th>
<th>$T_t$</th>
<th>$M$</th>
<th>$\bar{u}$</th>
<th>$\bar{p}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_{\text{blind}}$</td>
<td>79</td>
<td>37</td>
<td>1</td>
<td>93</td>
<td>171</td>
<td>10</td>
<td>21.5</td>
</tr>
<tr>
<td>$h_{\text{LMCut}}$</td>
<td>73</td>
<td>21</td>
<td>1</td>
<td>183</td>
<td>116</td>
<td>9</td>
<td>22.9</td>
</tr>
<tr>
<td>$h_{\text{OC}}$</td>
<td>70</td>
<td>119</td>
<td>3</td>
<td>141</td>
<td>99</td>
<td>8</td>
<td>17.4</td>
</tr>
<tr>
<td>$\text{OpSeq}$</td>
<td>63</td>
<td>2725</td>
<td>6</td>
<td>90</td>
<td>121</td>
<td>16</td>
<td>23.4</td>
</tr>
</tbody>
</table>

Table 5: Summary for 154 instances solved by all heuristics.
Acknowledgements
Wesley L. Kaizer thanks the financial support from Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq). André G. Pereira acknowledges support from FAPERGS with project 17/2551-0000867-7. Marcus Ritt thanks CNPq for support with grants 420348/2016-6 and 307522/2016-4. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

References
Bonet, B. 2013. An admissible heuristic for SAS+ planning obtained from the state equation. International Joint Conference on Artificial Intelligence 2268–2274.