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.

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

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* $\Pi = \langle \mathcal{V}, O, s_0, s_*, c\rangle$ 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 $D(v)$. A *partial state* s is a partial assignment over $\mathcal{V}$ and a *state* s is a complete assignment over $\mathcal{V}$. 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 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 $v \in vars(post(o))$. Function $c : O \rightarrow \mathcal{Z}_0^+$ assigns a non-negative cost $c(o)$ to each operator $o \in O$. 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.

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 ∞ 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 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 (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 Π is*

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

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

Operator Counts An operator counts $\mathcal{C}_s : O \rightarrow \mathcal{Z}_0^+$ 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 $\mathcal{C}_s$ is defined as $|\mathcal{C}_s| = \sum_{o \in O} \mathcal{C}_s(o)$.

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

Definition 4 (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 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.

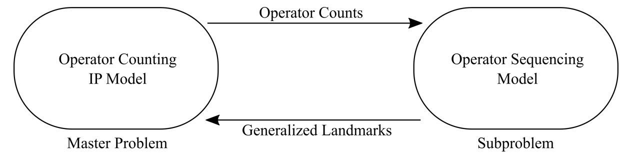

Figure 1: Logic-Based Benders Decomposition to cost-optimal planning (adapted from Davies et al. (2015)).

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 $\mathcal{C}_s$, 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 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 $\mathcal{C}_s$ 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 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 $\mathcal{C}_{s_0}$, it returns a plan π if $\mathcal{C}_{s_0}$ is sequencable, or a violated condition as a generalized landmark constraint L, otherwise. The 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 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 $\mathcal{C}_{s_0} = \{o_1 \mapsto 1\}$ (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 $\mathcal{C}_{s_0}$ 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 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 $\{v_1 = 0, v_{o_1} = 1\}$. 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 $\mathcal{C}_{s_0}$ 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 $s'' = \{v_1 = 1, v_{o_1} = 1\}$ 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 $v_o \notin vars(s) \wedge c(o) > 0$ 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 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.

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

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 $f_{\min} = \min_{s' | f(s') > \mathsf{Y}_f} \{f(s')\}$. Then, if $f_{\min} > -\infty$, we add the bounds literal $[\mathsf{Y}_f \geq f_{\min}]$ to L.

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 $f_{\max} = 1$. Figure 2 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.

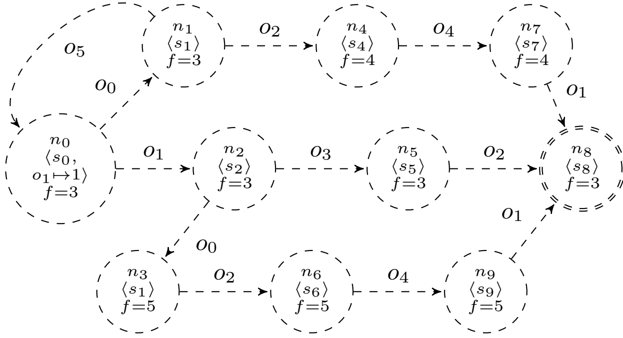

Figure 2: State-space of example problem Π_2, 1st iteration.

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 Figure 3:

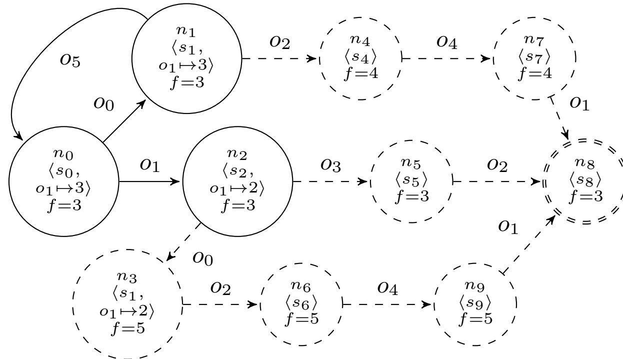

Figure 3: State-space of example problem Π_2, 2nd iteration.

Now *OpSearch* expands n_0 and generates node n_2 by applying o_1. Since we apply zero-cost operators freely during search *OpSearch* also generates n_1 and n_3 by applying o_0 to n_0 and n_2. Note that n_1 and n_3 have the same variable assignment s_1 but different operator counts $\{o_1 \mapsto 3\}$ and $\{o_1 \mapsto 2\}$ and therefore are treated as different states. From this state-space, *OpSearch* returns the constraint $[\mathsf{Y}_{o_3} \geq 1] + [\mathsf{Y}_f \geq 4] \geq 1$. The bound $[\mathsf{Y}_{o_3} \geq 1]$ comes from the transition $n_2 \xrightarrow{o_3} n_5$ and $[\mathsf{Y}_f \geq 4]$ from $n_1 \xrightarrow{o_2} n_4$, since transition $n_2 \xrightarrow{o_0} n_3$ would generate the bound $[\mathsf{Y}_f \geq 5]$. Suppose that, after adding this constraint, the IP$_C$ returns $\mathcal{C}_{s_0} = \{o_1 \mapsto 2, o_3 \mapsto 1\}$ and $\mathsf{Y}_f = 3$. The resulting state-space is shown in Figure 4:

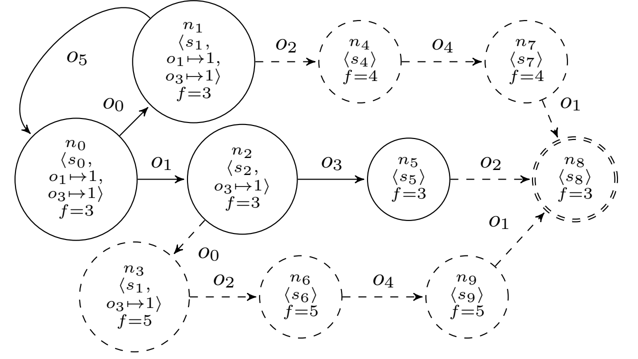

Figure 4: State-space of example problem Π_2, 3rd iteration.

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 Figure 5.

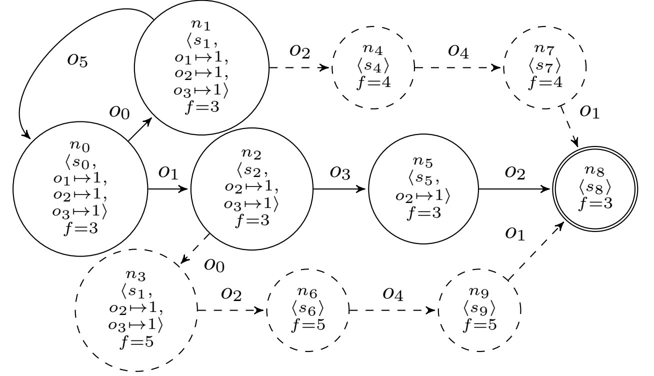

Figure 5: State-space of example problem Π_2, 4th iteration.

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

Proof sketch. Consider an optimal plan $\pi^* = \langle o_1, \ldots, o_k \rangle$

with a corresponding state sequence $\langle s_0, s_1, \ldots, s_* \rangle$. Let L be an 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^*. □

Different Heuristic Functions The heuristic function used in A* 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 pij and dij causes the robot to pick or drop ball i at room j and mij 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 $\langle p1l, mlr, d1r, mrl, p2l, mlr, d2r \rangle$.

Assume that $f_{\text{max}} = 5$ and $\mathcal{C}_s = \{d1r \mapsto 1, d2r \mapsto 1, mlr \mapsto 1, p1l \mapsto 1, p2l \mapsto 1\}$. We use an IP$_C$ with constraints from 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). *OpSeq* generates an GLC with five bounds: $[\mathsf{Y}_T \geq 6] + [\mathsf{Y}_{d1l} \geq 1] + [\mathsf{Y}_{d2l} \geq 1] + [\mathsf{Y}_{mrl} \geq 1] + [\mathsf{Y}_{p1r} \geq 1] \geq 1$; *OpSearch* with h^{blind} also generates an GLC with five bounds, but replaces the bound Y_T by Y_{p2r}: $[\mathsf{Y}_{d1l} \geq 1] + [\mathsf{Y}_{d2l} \geq 1] + [\mathsf{Y}_{mrl} \geq 1] + [\mathsf{Y}_{p1r} \geq 1] + [\mathsf{Y}_{p2r} \geq 1] \geq 1$; *OpSearch* with the h^{LMCut} heuristic generates an GLC with only one bound: $[\mathsf{Y}_f \geq 6] \geq 1$; Finally, *OpSearch* with h^* generates a perfect GLC that forces the IP$_C$ objective value to increase up to the cost of π^*: $[\mathsf{Y}_f \geq 7] \geq 1$.

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^{LMC} (Bonet and van den Briel 2014), state equation h^{SEQ} (Bonet 2013) and the optimal delete relaxation h^+ base formulation from (Imai and Fukunaga 2014). We use h^{LMCut} to guide *OpSearch* when sequencing.

The Benchmark Set

Table 1 presents information about the benchmark set, summarized by domain. $\overline{|\mathcal{V}|}$ denotes the mean number of variables; $\overline{|O|}$ is the mean number of operators; zco indicates the presence of zero-cost operators; $\overline{c_{min}}$ is the mean minimum operator cost, ignoring zero-cost operators; $\overline{c_{max}}$ is the mean maximum operator cost; $\overline{lb}$ is the mean best lower bound on the optimal plan cost[2]; $\overline{z}_0$ 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$_C$, 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 can also note that $\overline{z}_0$ is very close to $\overline{lb}$ 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

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

Table 1: Information of benchmark set.

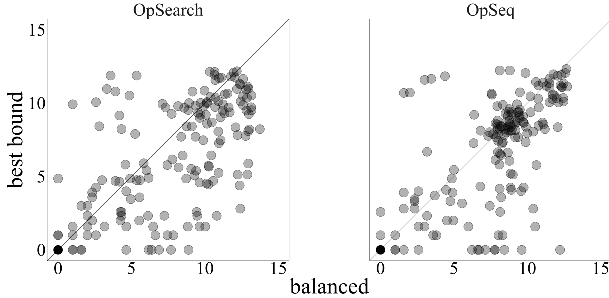

Figure 6: IP emphasis (log2-log2 scale).

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.

OpSeq and *OpSearch*

Table 3 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; $\bar{T}_t$ 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 Υ_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 comparison between 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.

Figure 7 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 2 summarises the results only considering instances solved by both *OpSearch* and *OpSeq*, showing that *OpSearch* solves fewer subproblems, uses slightly less memory, generates smaller constraints than *OpSeq*, but spends more time sequencing, as indicated by $\bar{p}$.

	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 2: Summary for 50 instances solved by both methods.

Impact of *OpSearch*'s Heuristic Function

Table 4 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 h^{OC} is usually more informed than h^{LMCut} and h^{blind} is the simplest one.

Using in general more informed heuristic functions in *OpSearch* results in: i) fewer sequencing subproblems

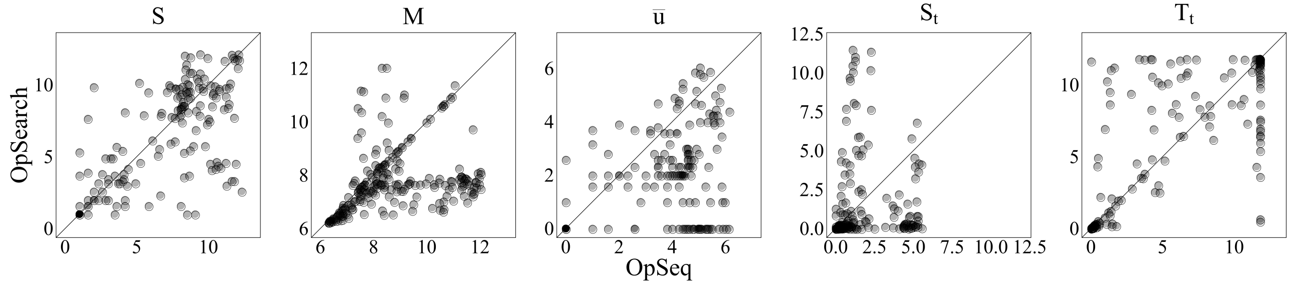

Figure 7: Detailed comparison between *OpSeq* and *OpSearch* (log2-log2 scale).

	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**

Table 3: Results for *OpSeq* and *OpSearch*.

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

Table 4: Summary for 49 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

Table 5: Summary for 154 instances solved by all heuristics.

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 5 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^*.

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

Conclusions and Future Work

In this work we introduced *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 *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 *OpSeq* in terms of overall memory usage.

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

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., 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.

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

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

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.

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.

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

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

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.

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.

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

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

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.

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

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

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

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.

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.

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.

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