Operator-Counting Heuristics for Domain-Independent Dynamic Programming

Anubhav Singh,[1] **Florian Pommerening,**[2] **Tanja Schindler,**[2] **J. Christopher Beck,**[1] **Malte Helmert** [2]

[1]University of Toronto, Canada
[2]University of Basel, Switzerland
anubhav.singh@utoronto.ca, {florian.pommerening,tanja.schindler,malte.helmert}@unibas.ch, jcb@mie.utoronto.ca

Abstract

Domain-independent Dynamic Programming (DIDP) recently emerged as a model-and-solve framework. Its performance heavily depends on a user-specified dual bound function. We introduce a way to automatically derive such bounds generalizing operator-counting heuristics from classical planning. The method approximates by how much an operator can change the value of a feature, which we compute using SAT modulo theories (SMT). We tighten the operator-counting LP using invariants derived by Abstract Interpretation. The dual bounds derived this way match well-known domain-specific dual bounds in TSP and Bin Packing. We evaluate our new dual bounds empirically in an established set of benchmarks.

Introduction

Domain-independent dynamic programming (DIDP) (Kuroiwa 2024) is a framework to model and solve combinatorial optimization problems. DIDP solvers internally use heuristic search methods like cost-algebraic A* (Edelkamp, Jabbar, and Lluch Lafuente 2005) or complete anytime beam search (Zhang 1998), whose performance critically depends on high-quality admissible heuristics. In DIDP, such heuristics are specified as a dual bound in the model. To obtain high-quality bounds, a domain expert needs to come up with a good idea and prove its admissibility. We propose a method to automatically derive high-quality admissible heuristics in a domain-independent way.

Our heuristics reason about changes in numeric state features and the number of operator applications required to induce a certain change. This idea is not new, early work in classical and numeric planning already considered *producers* and *consumers* of numerical resources or individual atoms (Kautz and Walser 1999; van den Briel, Vossen, and Kambhampati 2005; Coles et al. 2008; Piacentini et al. 2018a). The reasoning there was restricted to the change between two time steps of a Graphplan encoding. This was later generalized to reasoning about changes from the current state to a goal state, ignoring the order of operators and only considering the number of times they should be used. *Net-change constraints* (van den Briel et al. 2007; Bonet 2013; Pommerening et al. 2015) in the operator-counting framework categorize operators into producers and consumers and balance their number to match the total change.

Recent work adapts operator-counting and net-change constraints from classical to numeric planning (Piacentini et al. 2018b; Kuoriwa et al. 2021) to also track changes in numeric variables. They assume a fragment of simple numeric planning where numeric effects are limited to increase variables by a constant ($v \mathrel{+}= k$), which again yields a rigid categorization into producers and consumers.

We generalize this concept to cover not only the numeric but also the set variables of DIDP models. Instead of seeing an operator as a producer or consumer of a feature, we compute an interval bounding the possible changes this operator can induce. The interval is derived from an encoding using Satisfiability Modulo Theories (SMT, Barrett et al. 2021) and strengthened using an abstract interpretation (Cousot and Cousot 1977) of the DIDP model.

We first introduce the background for DIDP, operator-counting, SMT, and abstract interpretations. We then derive a generalized version of net-change constraints for labeled state spaces and apply them to DIDP. Studying two concrete problems shows that the derived heuristics match or dominate well-known hand-crafted bounds. We confirm this with an empirical evaluation and see the dominating heuristic translate to a higher coverage for a TSP model.

Background

We start with introducing the notation and mathematical background for the techniques used in this paper.

Domain-Independent Dynamic Programming

DIDP is a general framework to represent and solve dynamic programming problems. At the core of this framework is the Dynamic Programming Description Language (DyPDL, Kuroiwa and Beck 2023a), offering a standardized way to model DP problems. A DyPDL model is a tuple $\langle \mathcal{V}, S_0, \mathcal{K}, \mathcal{O}, B, C, h \rangle$ with the following components.[1]

Each *variable* $v \in \mathcal{V}$ is associated with a domain $dom(v)$ and can be either a *numeric variable* (with $dom(v) = \mathbb{N}$ or $dom(v) = \mathbb{R}$), an *element variable* for a type $\tau(v)$ (with

[1]We ignore forced transitions, resource variables, and logarithms, and assume the cost algebra to be commutative. We also simplify some notation for space and consistency reasons.

dom$(v) = E_{\tau(v)} = \{0, \dots, n_{\tau(v)} - 1\}$), or a *set variable* for a type $\tau(v)$ (with *dom*$(v) = 2^{E_{\tau(v)}}$). A variable assignment of all variables to values in their domains is called a *state*.

The *target state* S_0 is the state for which the *value* $V(S_0)$ should be computed.

The set of *constants* $\mathcal{K}$ contains numeric, element, set, and boolean constants which can be used in expressions. Expressions e are also grouped by type (numeric, element, set, boolean) and can be evaluated in a state S to a value in their domain $S(e) \in$ *dom*(e). Constants may be multi-dimensional *tables* that can be indexed with element expressions. Numeric and element expressions support the typical mathematical operations $\{+, -, *, /, \%, \min, \max\}$, numeric expressions also support $\{-(unary), |\cdot|, \sqrt{\cdot}, \exp\}$ and different casting operators to convert between integers and reals. Set expressions support set operations $\{\cap, \cup, \setminus, \bar{\cdot}\}$. Finally, boolean expressions can be built using comparators $\{=, \neq, <, \leq, \geq, >, \subseteq, \in\}$ and boolean connectives $\{\wedge, \vee, \neg\}$. We say a boolean expression b is *satisfied* in a state S (written as $S \models b$) if $S(b) = \top$. If-then-else expressions ITE(b, x_1, x_2) evaluate to $S(x_1)$ if $S \models b$ and to $S(x_2)$ otherwise. Reduce expressions reduce$(T, \circ, s_1, \dots, s_k)$ evaluate to

$$\bigcirc_{x_1 \in S(s_1), \dots, x_k \in S(s_k)} T[x_1] \cdots [x_k].$$

for operators $\circ \in \{+, *, \max, \min, \cup, \cap\}$. For a full list, see the supplementary material (Singh et al. 2025).

An *operator* $o \in \mathcal{O}$ is a tuple $\langle$*pre*(o), *eff*(o), *cost*$(o)\rangle$ where *pre*(o) is a boolean expression, *eff*(o) is a total function mapping variables to expressions of matching type, and *cost*(o) is a numeric expression. We also treat *eff*(o) as a set of tuples $\langle v, e_v\rangle$. The *applicable* operators in a state S are those with satisfied preconditions $\mathcal{O}(S) = \{o \in \mathcal{O} \mid S \models$ *pre*$(o)\}$. Applying an operator o in a state S results in the successor state $S[\![o]\!] = \{v \mapsto S(e_v) \mid \langle v, e_v\rangle \in$ *eff*$(o)\}$.

The *base case* B is a boolean expression. In the literature, DyPDL models can have multiple base cases that each have an associated cost expression. Those can be compiled into operators that achieve a new single base case.

The *state constraint* C is a boolean expression that must be satisfied by all visited states.

The costs in DIDP are determined by a *cost algebra* (Edelkamp, Jabbar, and Lluch Lafuente 2005), where

- $\langle A, \times, \mathbf{1}\rangle$ is a monoid ($\times$ is associative, $\mathbf{1}$ is neutral).
- $\preceq$ is a total order on A with *least* operation $\sqcup$.
- $\bigsqcup A = \mathbf{1}$ and $\bigsqcap A = \mathbf{0}$ (**1**/**0** are least/greatest elements).
- A is isotone ($a \preceq b$ implies both $a \times c \preceq b \times c$ and $c \times a \preceq c \times b$ for all $a, b, c \in A$).

The value of a state S then is $\mathbf{0}$ if $S \not\models C$, $\mathbf{1}$ if $S \models C \wedge B$, and $\bigsqcup_{o \in \mathcal{O}(S)}$ *cost*$(o)) \times V(S[\![o]\!])$ otherwise.

The *dual bound* h is a lower bound to the value of each state ($h(S) \preceq V(S)$ for all states S). In the literature, h is a numeric expression, however we define it as a general function $h : \mathcal{S} \to A$ here to accommodate lower bounds that cannot easily be expressed as DIDP expressions.

A DyPDL model $\Pi = \langle \mathcal{V}, S_0, \mathcal{K}, \mathcal{O}, B, C, h\rangle$ induces a labeled transition system $\mathcal{T}_\Pi = \langle \mathcal{S}, L, T, I, G,$ *cost*$\rangle$ with states $\mathcal{S}$ consisting of all states of Π, labels $L = \mathcal{O}$, transitions $T = \{S \xrightarrow{o} S' \mid o \in \mathcal{O}(S), S' = S[\![o]\!], S' \models C\}$, initial state $I = S_0$, goal states $G = \{S \in \mathcal{S} \mid S \models C \wedge B\}$ and cost *cost*$(S \xrightarrow{o} S') =$ *cost*$(o))$. A solution is a sequence $\pi = \langle t_1, \dots, t_n\rangle$ from I to a state in G. Its cost is $\times_i$ *cost*(t_i), and $V(S_0)$ is the cost of a $\preceq$-minimal solution.

Operator-Counting

Operator-counting heuristics (Pommerening et al. 2015) were originally developed for classical planning. The core idea is to minimize the total cost of used operators subject to constraints that describe necessary properties of solutions in terms of how often each operator is used. They use one *operator-counting variable* x_ℓ for each label ℓ of a transition system $\mathcal{T}$. A linear constraint over such variables is an *operator-counting constraint* for $\mathcal{T}$ if it is satisfied by $x_\ell =$ *occur*(ℓ, π) for every solution π of $\mathcal{T}$ where *occur*(ℓ, π) counts the number of times ℓ occurs in π. Many operator-counting constraints have been proposed, but for us the *net-change constraints* (Bonet 2013; Pommerening et al. 2015) are relevant. They trace how often an atom $a = \langle v, d\rangle$ is *produced* (made true) and *consumed* (made false):

$$L \leq \sum_{\substack{\ell \text{ always} \\ \text{produces } a}} x_\ell + \sum_{\substack{\ell \text{ sometimes} \\ \text{produces } a}} x_\ell - \sum_{\substack{\ell \text{ always} \\ \text{consumes } a}} x_\ell \tag{1}$$

$$\sum_{\substack{\ell \text{ always} \\ \text{produces } a}} x_\ell - \sum_{\substack{\ell \text{ always} \\ \text{consumes } a}} x_\ell - \sum_{\substack{\ell \text{ sometimes} \\ \text{consumes } a}} x_\ell \leq U \tag{2}$$

where L and U bound the *net change* of a, i.e., the change between the initial state and a goal state.

Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) denotes the problem of deciding whether a first-order logic formula is satisfiable with respect to given background theories (Barrett et al. 2021). Theories restrict the interpretation of some symbols, enabling reasoning about objects such as integers or sets.

A quantifier-free *SMT formula* is a formula in many-sorted first-order logic with equality. It is built from variables, constant, function and predicate symbols, that are either restricted by the theory or *uninterpreted*, the usual logical connectives ($\neg, \wedge, \vee$), and the equality symbol $=$. Each term has an associated sort (e.g., Int). The formula can contain ternary ITE functions with the same meaning as in DIDP. A *model* of an SMT formula is an interpretation that satisfies the formula and is consistent with the theories.

In this work, we use the theory of nonlinear integer and real arithmetic and the theory of finite sets with cardinality. The former defines the sorts Int and Real, operators $-$ (unary), $+$, $-$, $*$, $/$, $\%$, $|\cdot|$ for integer operands, $-$ (unary), $+$, $-$, $*$, $/$, $\lfloor\cdot\rfloor$ for real operands, and the comparators $<$, $\leq$, $=$, $\geq$, $>$ with their usual mathematical meaning. We use an extension of the theory supported by the SMT solver CVC5 (Barbosa et al. 2022) that also defines exponentiation $\cdot^k$ for $k \in \mathbb{N}$ and $\sqrt{\cdot}$. The set theory (Bansal et al. 2018) defines sorts Set_T (with elements of sort T), the usual set operations and predicates ($\cup$, $\cap$, $\setminus$, $\in$, $\subseteq$), functions $|\cdot|$ for the cardinality, $\emptyset$ and $\{\cdot\}$ for creating an empty or singleton set.

Interval Arithmetic

Closed intervals are important for several aspect of our work. Given intervals $X = [X_L, X_U]$, $Y = [Y_L, Y_U]$, a constant $c \in \mathbb{R}$, and a set $S \subseteq \mathbb{R}$, we define the following symbols:

$$X + Y = [X_L + Y_L, X_U + Y_U] \qquad \min(X) = X_L$$
$$cX = Xc = [cX_L, cX_U] \qquad \max(X) = X_U$$
$$c \in X \text{ iff } X_L \leq c \leq X_U \qquad X = \emptyset \text{ iff } X_L > X_U$$
$$S \subseteq X \text{ iff } x \in X \text{ for all } x \in S$$
$$X \cap Y = [\max(X_L, Y_L), \min(X_U, Y_U)]$$
$$X \cup Y = [\min(X_L, Y_L), \max(X_U, Y_U)]$$
$$(X \pitchfork Y) \text{ iff } X \cap Y \neq \emptyset$$

Abstract Interpretation

Abstract Interpretation (Cousot and Cousot 1977) is a method for static analysis of software. It analyzes programs in terms of the possible variable values at each program point, but instead of considering their concrete values in a domain D, it uses an abstract domain D^α, where each value represents a set of concrete values. A fixpoint in an abstract execution is an overapproximation of all possible program behaviours. We refer to Miné (2017) for a detailed explanation.

Formally, an abstract domain D^α comes with a partial order $\sqsubseteq^\alpha$ on D^α with least element $\bot^\alpha$ and greatest element $\top^\alpha$, a concretization function $\gamma : D^\alpha \to D$, abstractions of union $\cup^\alpha$ and intersection $\cap^\alpha$ to combine abstract values such that they overapproximate their concrete counterparts, and abstract semantics for assignments and conditions.

We represent programs by program graphs. To compute an abstract interpretation, their edges are associated with abstract values that are repeatedly updated until a fixpoint is reached. They consist of the following types of nodes:

- The single *entry* node has no incoming edges. Their outgoing edge is initialized to the least element $\bot^\alpha$.
- *Assignment* nodes correspond to simultaneous assignments $v \leftarrow$ *expr* to several variables. Their outgoing edge is updated with the abstract assignment semantics.
- *Assumption* nodes contain statements **assume** *cond*, where program execution cannot continue unless *cond* is satisfied. Their outgoing edge is updated by intersecting the incoming value with the abstract condition semantics.
- *Non-deterministic choice* nodes may have multiple outgoing edges, corresponding to branching in the program. They copy the incoming value to all outgoing edges.
- *Junction* nodes have an arbitrary finite number of incoming edges, merging several paths. Their outgoing edge is updated as the abstract union of incoming edges.
- The single *exit* node has no outgoing edges.

Each update results in a superset of the previous value. To guarantee termination, in each program loop a *widening* operator ∇ is applied that combines old and new abstract values into a superset of the two in such a way that any repeated application converges in finite time. The final abstract values at an edge contain all possible values at the respective program point and represent *invariants* of the program.

We use several numeric domains. The interval domain (Cousot and Cousot 1976) expresses bounds on the values of single variables. Its abstract values are all closed intervals $[a, b] \subseteq \mathbb{Z} \cup \{-\infty, +\infty\}$ (or $\mathbb{R} \cup \{-\infty, +\infty\}$) with least element $\bot^\alpha$ representing the empty interval. The partial order $\sqsubseteq^\alpha$ is $\subseteq$, $\cap^\alpha$ is $\cap$, and $\cup^\alpha$ is $\cup$ on intervals. Arithmetic operators $+, -, *, /$ are abstracted by performing the computation on the bounds of the intervals corresponding to the operands. Widening $[a, b] \nabla [c, d]$ replaces a with $-\infty$ if $c < a$ and b with ∞ if $b < d$. The octagon domain (Miné 2006) expresses differences between two variables using inequalities $\pm x \pm y \leq c$ and the polyhedra domain (Cousot and Halbwachs 1978) expresses linear dependencies between variables using linear inequalities $\sum_{i=1}^{n} a_i x_i \geq c$.

Net-Change Constraints for DIDP

We now introduce a general form of net-change constraints for transition systems and then apply them to DIDP.

State Features Instead of tracking how often an atom is produced and consumed, we consider *state features* $f : \mathcal{S} \to \mathbb{R}$, e.g. a numeric DIDP expression. In the experiments, we consider one feature $f_v(S) = S(v)$ for every numeric and element variable. In addition, we consider one indicator feature $f_{v=d}$ for each combination of an element variable v and one of its values d that is 1 in states S with $S(v) = d$. The reason is that treating changes to element variables as numerical changes can be misleading. For example, moving from location 4 to location 7 in a TSP is a change of 3 but this difference is meaningless in the TSP. For set variables v, we consider two kinds of features: the cardinality of the set $f_{|v|}(S) = |S(v)|$ and an indicator feature for each element x of the correspond type $f_{x \in v}(S)$ that maps to 1 iff $x \in S(v)$.

Interval-based net-change constraints Along path $\pi = \langle \ell_1, \ldots, \ell_n \rangle$ via states $S_0, S_1, \ldots, S_n$, the value of a feature f starts at $f(S_0)$, then changes to $f(S_1)$, $f(S_2)$ and so on until it reaches $f(S_n)$. In step i label ℓ_i causes the change $\Delta_i = f(S_i) - f(S_{i-1})$ and overall, we have $\sum_i \Delta_i = f(S_n) - f(S_0)$. We would like to group the elements of this sum by label and write $\sum_{\ell \in L} \Delta_\ell \mathit{occur}(\ell, \pi) = f(S_n) - f(S_0)$. However, labels can occur multiple times in π and induce different changes each time, so Δ_ℓ cannot be represented as a constant.

For each label ℓ in a transition system $\langle \mathcal{S}, L, T, I, G, \mathit{cost} \rangle$ and each feature f, let $D_{\ell,f} = \{f(S') - f(S) \mid S \xrightarrow{\ell} S' \in T\}$ be the possible changes that ℓ can cause for f and let $\Delta_{\ell,f}$ be an interval with $D_{\ell,f} \subseteq \Delta_{\ell,f}$. Further, define $D_f = \{f(s_\star) - f(I) \mid s_\star \in G\}$ and Δ_f as an interval containing all values in D_f.

Definition 1. *Let $\mathcal{T} = \langle \mathcal{S}, L, T, I, G, \mathit{cost} \rangle$ be a transition system and let $f : \mathcal{S} \to \mathbb{R}$ be a state feature. The* interval-based net-change constraint *for f in $\mathcal{T}$ is*

$$\sum_{\ell \in L} \Delta_{\ell,f} x_\ell \pitchfork \Delta_f$$

This constraint uses intervals as coefficients and is nonlinear, so it is not an operator-counting constraint. But similar to one, it is satisfied by operator occurrences in solutions.

Proposition 1. *Let f be a state feature for a transition system $\mathcal{T}$ and π a solution of $\mathcal{T}$. The interval-based net-change constraint for f is satisfied by $x_\ell = occur(\ell, \pi)$.*

Proof. Let $\pi = \langle \ell_1, \dots, \ell_n \rangle$ and $S_0, S_1, \dots, S_n$ the states that π passes through (with $S_0 = I$). For $1 \leq i \leq n$ we have $(f(S_i) - f(S_{i-1})) \in D_{\ell_i,f} \subseteq \Delta_{\ell_i,f}$. Thus $nc = f(S_n) - f(S_0) = \sum_{1 \leq i \leq n} (f(S_i) - f(S_{i-1})) \in \sum_{1 \leq i \leq n} \Delta_{\ell_i,f} = \sum_{\ell \in L} \Delta_{\ell,f} x_\ell$. Likewise, we have $nc \in D_f \subseteq \Delta_f$, so nc is a witness for $\sum_{\ell \in L} \Delta_{\ell,f} x_\ell \pitchfork \Delta_f$. □

General net-change constraints We can derive linear operator-counting constraints from an interval-based constraint using that $(X \pitchfork Y)$ iff $X \neq \emptyset$, $Y \neq \emptyset$, $\min(X) \leq \max(Y)$, and $\min(Y) \leq \max(X)$. The first two of these constraints are trivially satisfied if all involved intervals are non-empty. An interval $\Delta_{\ell,f}$ can only be empty if there is no transition labeled with ℓ and Δ_f can only be empty if the transition system has no goal state. In such cases, we can fix $x_\ell = 0$ or report $\mathcal{T}$ unsolvable in a preprocessing step.

Definition 2. *Let $\mathcal{T} = \langle \mathcal{S}, L, T, I, G, cost \rangle$ be a transition system and f a state feature. The* general net-change constraint *for f in $\mathcal{T}$ consists of the following two inequalities.*

$$\sum\nolimits_{\ell \in L} \min(\Delta_{\ell,f}) x_\ell \leq \max(\Delta_f) \tag{3}$$

$$\sum\nolimits_{\ell \in L} \max(\Delta_{\ell,f}) x_\ell \geq \min(\Delta_f) \tag{4}$$

If every label occurs on at least one transition and there is at least one goal state, constraints (3)–(4) are equivalent to the interval-based formulation, as $\min(\sum_{\ell \in L} \Delta_{\ell,f} x_\ell) = \sum_{\ell \in L} \min(\Delta_{\ell,f}) x_\ell$ (and analogously for maximization).

Connection to classical planning The net-change constraint for an atom a (equation (1)–(2)) can be seen as a special case of general net-change constraints using a feature f_a that is 1 in states satisfying a and 0 otherwise. In that case Δ_{o,f_a} is $[1, 1]$ if o always produces a, $[0, 1]$ if o sometimes produces a, $[-1, -1]$ if o always consumes a, and $[-1, 0]$ if o sometimes consumes a. The bounds $[L, U]$ on the net change used in constraints (1)–(2) directly correspond to Δ_{f_a}. Using this in (3)–(4) shows the equivalence.

Connection to numeric planning The LP-RPG heuristic for numerical planning (Coles et al. 2013) identifies an operator o as a producer or consumer of a numeric variable v if applying o always increases or decreases the value of v by some constant $\delta(v, o)$. This is used in a relaxed plan graph (RPG) as a constraint $v' = v + \sum_o C_o \delta(v, o)$ where v and v' are the values of the variable in two subsequent layers and C_o is a variable representing the number of times o is used in between those two layers. The constraint matches the general net-change constraint in cases where all operators are producers, consumers, or irrelevant for v. The main difference is that the RPG-constraint encodes the change between two layers of the RPG, while net-change constraints encode the total change throughout the plan.

The numeric state equation constraints by Piacentini et al. (2018b) consists of constraints $I(v) + \sum_{o \in O} \delta(v, o) x_o \leq ub_v$ (their equation 18, adapted to our notation) where $\delta(v, o)$ is the constant that o adds to v, and ub_v is an upper bound for the value of v in any (goal) state. This constraint matches (3) and the corresponding lower-bound constraint (their equation 19) matches (4) with $\Delta_v = [lb_v - I(v), ub_v - I(v)]$. Piacentini et al. also include a constraint that considers multiple features that occur together in a goal condition (their equation 17). This is not covered by our constraints but could be added for sufficiently simple goal conditions (e.g. linear expressions) by replacing each variable v with $I(v) + \sum_{\ell \in L} \Delta_{\ell,v} x_\ell$ and then minimizing/maximizing over the resulting interval equation.

In contrast to the two methods, general net-change constraints also deal with features that are not numeric variables and cases where operators do not induce a constant change.

Use in DIDP To use general net-change constraints in DIDP, we still have to define which labels to use, how to minimize the total cost of used operators subject to the constraints, and how to compute good overapproximations $\Delta_{\ell,f}$ and Δ_f. We now discuss these steps in more detail.

Splitting DIDP Operators

When mapping a DIDP instance to a labeled transition system, it can be beneficial to treat one DIDP operator as multiple labels. For example, in a TSP, *move*(j) moves the agent from its current location to j. This clearly produces the atom that the agent is at j but without knowing where the operator was applied, it is unclear which atom is consumed. Using labels *move*(i, j) for moves from i to j simplifies this. Formally, we can split a DIDP operator by creating copies of it and adding jointly-exhaustive preconditions to the copies (Röger, Pommerening, and Helmert 2014).

In the experiments our strategy is to instantiate element variables which occur in cost expressions. For example, in TSP, the cost of *move*(j) is *dist*$[l, j]$, the distance from the current location l to j. Note that j is a constant here, but l is a variable. We create one copy of *move*(j) for each value $i \in$ *dom*(l) and add the precondition $(l = i)$ to it. This creates an operator that moves from i to j (for constants i and j).

Optimizing Total Cost

To derive a lower bound on the solution cost from a set of operator-counting constraints *OCC* in a transition system with labels L, we have to solve the following problem:

$$\underset{\pi \in L^*}{\text{Minimize}}\ cost(\pi) \text{ subject to } OCC \text{ and}$$
$$x_\ell = occur(\ell, \pi) \text{ for all } \ell \in L$$

Since any optimal solution of the transition system must satisfy *OCC* with $x_\ell = occur(\ell, \pi)$, minimization over all label sequences yields a lower bound to the optimal solution cost.

For DIDP, a complication comes from the fact that $cost(\pi)$ is defined by a cost algebra $\langle A, \sqcup, \times, \preceq, \mathbf{0}, \mathbf{1} \rangle$, and that the cost function can be state-dependent. That is, for a plan $\pi = \langle o_1, \dots, o_n \rangle$ passing through states $\langle S_0, \dots, S_n \rangle$, the cost of π is defined as $cost(\pi) = S_0(cost(o_1)) \times \cdots \times S_{n-1}(cost(o_n))$. We underestimate the state-dependent cost function with the state-independent cost function $cost(o) = \sqcup_s cost(o, s)$. Splitting DIDP operators as discussed above

can mitigate the effect of this underestimation. In the following we can assume that operator costs are state-independent and $cost(\pi) = cost(o_1) \times \cdots \times cost(o_n)$.

We made the further assumption that $\times$ is commutative. This is true in all our benchmarks as well as in all cost algebras implemented in the DIDP library so far. The total cost can then be written as $cost(\pi) = \bigtimes_{o \in \mathcal{O}} cost(o)^{occur(o,\pi)}$ where a^n is interpreted as $(a \times \cdots \times a)$ with n repetitions of a. This simplifies the operator-counting problem from an optimization over all operator sequences to an optimization of operator counts:

$$\bigsqcup_{x \in \mathbb{N}^{|\mathcal{O}|}} \bigtimes_{o \in \mathcal{O}} cost(o)^{x_o} \text{ subject to } OCC \tag{5}$$

For some special cases of operator $\times$, the expression a^n has a well-defined interpretation. For example, when $\times$ denotes addition, $a^n = na$; and when $\times$ denotes the maximization, $a^n = a$. For both the cases, the optimization problem can be solved as an LP.

Even in the general case, with no formal paradigm defining a^n, optimization problem (5) can be solved using Logic-Based Benders Decomposition (Hooker and Ottosson 2003). This decomposes the problem into two parts, the master: Minimize$_{x \in \mathbb{N}^{|\mathcal{O}|}, z \in \mathbb{R}}$ z subject to OCC and χ, and the subproblem: $z \geq f(\bigtimes_{o \in \mathcal{O}} cost(o)^{x_o})$, where $z \in \mathbb{R}$ represents the cost, constraints χ connect operator-counting variables x to z, and f is a morphism mapping A into $R \subseteq \mathbb{R}$, preserving the relation, $a \preceq b \iff f(a) \leq f(b)$.

The master and subproblem are solved iteratively starting with the master using any Mixed Integer Linear Programming solver. The subproblem is solved by evaluating $\lambda(z) \geq f(\bigtimes_{o \in \mathcal{O}} cost(o)^{\lambda(x_o)})$ for the assignment λ to master variables x and z. If the subproblem is satisfied, λ is optimal, otherwise the conflict resolution clause

$$\bigwedge\nolimits_{o \in \mathcal{O}} (x_o \geq \lambda(x_o)) \rightarrow (z \geq f(\bigtimes\nolimits_{o \in \mathcal{O}} cost(o)^{\lambda(x_o)}))$$

is added to the master as a Benders cut. Master and subproblem are repeatedly solved until the subproblem is satisfied.

Bounding the Possible Change using SMT

We now discuss how the intervals $\Delta_{o,f}$ (overapproximating the possible changes to f induced by o) and Δ_f (overapproximating the net change in f) can be computed for an operator o and a feature f encoded as a numeric expression.

Our approach uses an encoding $\eta(e, \mathcal{V})$ of a DyPDL expression e over variables $\mathcal{V}$ as an SMT formula. Most DyPDL expressions can be straight-forwardly encoded, as the languages are similar. However, for set expressions, the semantics are subtly different: DyPDL sets s are always subsets of some finite set $E_{\tau(s)}$. We could create a sort T for each type $\tau(s)$ and use sort Set$_T$ for s. However, elements of these sets would no longer be compatible with the numeric operations in the theory of integers. We thus use sort Set$_{\text{Int}}$ for all sets. When computing the complement of a set we then have to make the set $E_{\tau(x)}$ explicit, to compute the complement relative to the correct domain. For DyPDL tables T, we use uninterpreted function symbols f_T of the correct arity k and add clauses $f_T(x_1, \ldots, x_k) = T[x_1] \cdots [x_k]$ to fix the table entries. Reduce expressions are rewritten as a sequence of if-then-else expressions. For example $\sum_{x \in s} T[x]$ is rewritten as $\sum_x \text{ITE}(x \in s, T[x], 0)$. Some valid DyPDL expressions (x^y where y is not an integer constant, $x \% y$ for continuous terms) cannot be encoded in the SMT theories we use but these do not occur in our benchmarks. For a full list, see the supplementary material.

To compute bounds on Δ_f, we evaluate f in the target state and then encode $\eta(B, \mathcal{V}) \wedge \eta(C, \mathcal{V}) \wedge (\eta(f, \mathcal{V}) \leq d)$ for a constant d to express that f can have a value of at most d in a base case B. A binary search on the domain of a 32-bit integer can find the lowest value of d for which the formula is satisfiable in at most 32 satisfiability checks. For integer-valued features, this is a lower bound to Δ_f. For real-valued features we have to use the infimum instead of the minimum and thus pick the first value for which the formula is unsatisfiable. An upper bound can be found with a second binary search on an analogous formula. In case one of the binary searches does not yield a bound, the instance is unsolvable.

For $\Delta_{o,f}$ we use two copies of the DyPDL variables $\mathcal{V}$ and $\mathcal{V}'$ to encode a transition $S \xrightarrow{o} S'$, where variables in $\mathcal{V}$ are used for statements about S and variables in $\mathcal{V}'$ for S'. The formula is the conjunction of four parts, representing in turn that $pre(o)$ is satisfied in S, that the effects of o are applied in S', that S and S' satisfy properties of reachable states, and that the difference in the feature value of f is bounded by d. The subformula invariants$(\mathcal{V}, \mathcal{V}')$ consists of clauses $0 \leq v$ and $v < n_{\tau(v)}$ for every element variable, $v \subseteq E_{\tau(v)}$ for every set variable, analogous clauses for S', as well as $\eta(C, \mathcal{V}')$ to ensure that the state constraint is satisfied in S'. We will later show how the formula can be strengthened with additional invariants.

$$\eta(pre(o), \mathcal{V}) \wedge \Big(\bigwedge_{\langle v, e_v \rangle \in eff(o)} \eta(v, \mathcal{V}') = \eta(e_v, \mathcal{V})\Big) \tag{6}$$

$$\wedge \text{ invariants}(\mathcal{V}, \mathcal{V}') \wedge (\eta(f, \mathcal{V}') - \eta(f, \mathcal{V}) \leq d) \tag{7}$$

A model of (6)–(7) describes a state where o is applicable and changes the value of f by at most d. As before, we use two binary searches to determine $\Delta_{o,f}$. If one of them does not yield a bound, $\Delta_{o,f} = \emptyset$ and we fix $x_o = 0$.

The size of the SMT formula in (6)–(7) is linear in the size of the DIDP expressions of pre_o, eff_o for a single operator o, and the relevant tables and invariants.

Example 1. *Consider a TSP encoding with a set variable U that tracks the set of unvisited locations and an element variable L for the current location of the agent. For a specific city c, the feature $f_{c \in U}$ is 1 if c is unvisited and 0 otherwise. Let $n = n_{\tau(U)}$, $E = E_{\tau(U)}$, and let $o = \text{move}_c$ be an operator that moves to c. Its precondition is $(c \in U)$ and its effect is $\{(L, c), (U, U \setminus \{c\})\}$. The constructed SMT formula for difference d is*

$$\begin{aligned}
&(c \in U) \wedge (L' = c) \wedge (U' = U \setminus \{c\}) \\
&\wedge (0 \leq L) \wedge (L < n) \wedge (U \subseteq E) \\
&\wedge (0 \leq L') \wedge (L' < n) \wedge (U' \subseteq E) \\
&\wedge (\text{ITE}(c \in U', 1, 0) - \text{ITE}(c \in U, 1, 0) \leq d)
\end{aligned}$$

The lowest value of d for which the formula is satisfiable is -1. Replacing $\leq$ with $\geq$ shows that the lowest upper bound is -1 as well, fixing $\Delta_{o,f_{c\in U}} = [-1,-1]$.

In the example above, the discovered interval is tight. Unfortunately, this is not guaranteed. In the next example, we see how the SMT encoding can fail to find a bound.

Example 2. *A bin-packing problem for items N can be encoded with a set variable U ($E_{\tau(U)} = N$) representing the set of unpacked items and an integer variable r for the amount of remaining space in the last open bin (initially 0 assuming no bin is open). There is an operator* pack_i *to pack item $i \in N$ into the current bin if i is unpacked ($i \in U$) and there is sufficient space left ($w_i \leq r$). Its effect is $\{\langle U, U \setminus \{i\}\rangle, \langle r, r - w_i\rangle\}$. Furthermore, there is an operator* open *that is only applicable if there is at least one item to pack ($U \neq \emptyset$) and no item fits in the current bin anymore ($\bigwedge_{i\in N}((i \notin U) \vee (w_i > r))$). Its effect is to open a new bin with capacity q: $\{\langle U, U\rangle, \langle r, q\rangle\}$.*

Consider the feature f_r that evaluates to the value of r. The SMT formula for the change induced by $o =$ open *(ignoring some irrelevant invariants) is*

$$(U \neq \emptyset) \wedge \bigwedge\nolimits_{i\in N}((i \notin U) \vee (w_i > r))$$
$$\wedge\, (U' = U) \wedge (r' = q) \wedge (U \subseteq N) \wedge (r' - r \leq d)$$

As expected, the formula establishes a lower bound of $q - \max_{i\in I}(w_i - 1)$ for the change in f_r. However, if we replace $\leq$ with $\geq$ to look for an upper bound of the change, the formula becomes satisfiable for any value of d with $r = \min(\{q - d\} \cup \{w_i - 1 \mid i \in U\})$.

The reason why the formula does not yield a useful upper bound in the example above is that it considers each operator in isolation. If r could become negative, setting it to q could increase its value by an arbitrary amount. To see that this is not possible, more global reasoning is necessary. Operator pack_i can only reduce r but never below 0 while open sets r to q. Since the value initially is in the interval $[0, q]$, it will remain in this interval in all reachable states. We now show how to automate this argument.

Invariants

We discuss two methods for deriving invariants of a DyPDL model using abstract interpretation and SMT. Any invariants we discover can be encoded in an SMT expression and added to invariants$(\mathcal{V}, \mathcal{V}')$ in formula (7). This strengthens the formula and can lead to tighter bounds on $\Delta_{o,f}$

Abstract Interpretation Invariants

The reachable part of the DIDP state space can be described as a non-deterministic program shown in Algorithm 1. The program initializes the variables to the values they have in S_0. Afterwards, it repeatedly selects an operator, checks that it is applicable, applies the operator's effects, and checks that the resulting state does not violate the state constraint.

Program invariants for the start of line 8 are invariants for reachable states in the DIDP model. We can represent the program as the program graph shown in Figure 1. Since

Algorithm 1: Program to compute any reachable state.

1: $v \leftarrow S_0(v)$ for all $v \in \mathcal{V}$
2: **while not** terminate() **do**
3: $\quad o \leftarrow$ non-deterministic choice from $\mathcal{O}$
4: $\quad$ **assume**$(\mathit{pre}(o))$
5: $\quad v \leftarrow e_v$ for all $\langle v, e_v\rangle \in \mathit{eff}(o)$
6: $\quad$ **assume**(C)
7: **end while**
8: **return** state stored in variables $v \in \mathcal{V}$

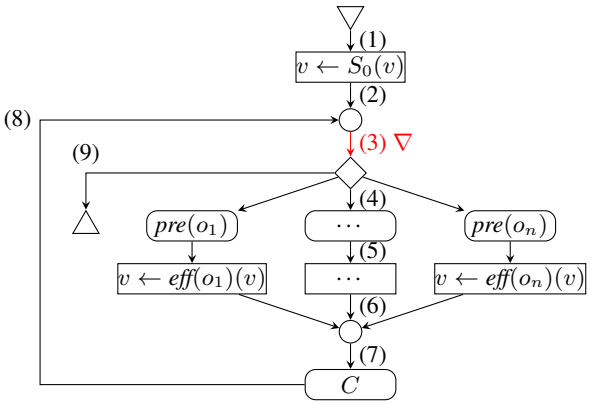

Figure 1: Program graph for Algorithm 1 for the special case of a single variable. With multiple variables, assignments to all variables must be done in the same node. Widening is performed on edge (3).

Apron (Jeannet and Miné 2009), the library we use to compute abstract interpretations, only supports numeric variables and a limited set of expressions, we now discuss how we can express assumptions and assignments of DIDP expressions in this language.

Apron natively supports integer and floating point numbers, expressions built from the common mathematical operators $\{+, -, *, /, \%, \sqrt{}, \exp\}$, as well as all rounding modes that DyPDL supports. Expressions $|n|$ can be replaced with ITE$(n \geq 0, n, -n)$, $\max(e_1, e_2)$ with ITE$(e_1 \geq e_2, e_1, e_2)$, and $\min(e_1, e_2)$ with ITE$(e_1 \leq e_2, e_1, e_2)$.

If an ITE(b, x_1, x_2) expression occurs in any assignment or assumption node n of the program graph, a fresh variable v is added and we insert a non-deterministic choice with branches (**assume**$(b); v \leftarrow x_1$) and (**assume**$(\neg b); v \leftarrow x_2$) before n, We then replace the ITE expression in n with v. Afterwards, boolean expressions only occur in assumptions.

Apron does not support boolean operators $\{\wedge, \vee, \neg\}$, so in assumption nodes we first transform the condition to negation normal form, moving the negations in front of basic numeric relations, then use complement relations to get rid of them, e.g. transforming $\neg(x < y)$ to $(x \geq y)$. We then repeatedly split assumption nodes for conjunctions $a \wedge b$ into a sequence of two assumption nodes for a and b, and assumption nodes for disjunctions $a \vee b$ into a non-deterministic choice of assuming a or assuming b.

We compile expressions accessing tables with element expressions $T[e_1]\cdots[e_k]$ into a series of ITE expressions

that check for each combination of indices $x_i \in E_{\tau(e_i)}$ whether $x_i = e_i$ and assign a temporary variable to the matching table entry. We rewrite reduce expressions $\text{reduce}(T, \circ, s_1, \ldots, s_k)$ with ITE expressions just as in our SMT encoding.

For set variables s with domain $2^{E_{\tau(s)}}$, we introduce one integer variable s_i for each $i \in E_{\tau(s)}$, that is 1 if $i \in s$ and 0 otherwise. Set cardinality $|s|$ is then simply $\sum_i s_i$ and set expressions can be seen as a sequence of numeric expressions, one for each entry. Set operations are encoded as their influence on each s_i: $(\bar{s})_i = 1 - s_i$, $(s_1 \cap s_2)_i = s_{1i} s_{2i}$, $(s_1 \cup s_2)_i = s_{1i} + s_{2i} - s_{1i} s_{2i}$, and $(s_1 \setminus s_2)_i = s_{1i}(1 - s_{2i})$. Set equality $s_1 = s_2$ can be expressed as $\bigwedge_i s_{1i} = s_{2i}$, subset relations $s_1 \subseteq s_2$ as $\bigwedge_i s_{1i} \leq s_{2i}$, and inequality $s_1 \neq s_2$ as $\sum_i (s_{1i}(1 - s_{2i})) + ((1 - s_{1i}) s_{2i}) \geq 1$. Finally, $e \in s$ can be written as $(\prod_i (e + 1) - (i + 1) s_i) = 0$ (after shifting the indices up by one they all are different from 0).

The result of this encoding is a program graph as the one in Figure 1. We use Apron with different abstract domains to extract invariants from the abstract value of the edge leading to the exit node which represents a reachable state. We perform widening starting from the second iteration through the loop which means the computation will terminate.

Example 3. *Consider the Bin Packing model from Example 2 again. We now show how abstract interpretation with the interval domain computes an invariant for r (we ignore U for simplicity) referring to edge labels in Figure 1.*

The abstract value of r is initialized to $\bot^\alpha$ at edge (1), then updated to $[0, 0]$ at (2), and propagated to (3) and (4).

In the branch for open*, the value of r is not changed in (5) and set to $[q, q]$ on (6), while in the branch for* pack_i*, it is set to $\bot^\alpha$ in (5) based on the precondition $r \geq w_i$. The values are joined to $[q, q] \cup \bot^\alpha = [q, q]$ on edge (7). There is no state constraint, so $[q, q]$ propagates to (8). Joining it with $[0, 0]$ from (2) updates the value of (3) to $[0, 0] \cup [q, q] = [0, q]$.*

Widening at this point would widen $[0, q]$ to $[0, \infty]$ because the upper bound increased in the loop. This occurs in the first iteration of any problem, because operators change the abstract values set in the initial state, which is why we skip the first widening. In the second iteration, the branch for open *again yields $[q, q]$ at (6), while in the branch for* pack_i*, assuming the precondition $r \geq w_i$ this time changes $[0, q]$ to $[w_i, q]$ at (5). Executing the effect then yields $[0, q - w_i]$ at (6). Joining the values gives $[0, q]$ at (7) and (8). At (3) we reach a fixpoint. The extracted invariant is $0 \leq r \leq q$. If we add this invariant to the formula in Example 2, we can now find an upper bound of q for the change* open *induces in f_r.*

Templated Invariants

When extracting invariants with abstract interpretation, the form of the discovered invariant is limited by the abstract domain we are using. For example, in the TSP, the current location L is always visited, so $L \notin U$ is an invariant but none of our abstract domains could find it: the space it describes in terms of the variables L and U_i is non-convex.

An alternative way of computing invariants is to fix a mathematical form, generate candidates of this form, and test if they are indeed invariants. We restrict the form to $e \in s$ and $e \notin s$ for element variables e and set variables s, since these cannot be found with abstract interpretation and there is a small finite pool of such candidates. The method for testing an invariant is more general, though.

To test whether a boolean DyPDL expression b is an invariant, we have to confirm that it is satisfied in the initial state ($S_0 \models b$) and cannot become false along a transition. We can verify the latter by testing the following SMT formula for satisfiability which closely follows (6)–(7).

$$\eta(\mathit{pre}(o), \mathcal{V}) \wedge \Big(\bigwedge_{\langle v, e_v \rangle \in \mathit{eff}(o)} \eta(v, \mathcal{V}') = \eta(e_v, \mathcal{V}) \Big) \\ \wedge \eta(C, \mathcal{V}) \wedge \eta(C, \mathcal{V}') \wedge \eta(b, \mathcal{V}) \wedge \neg \eta(b, \mathcal{V}')$$

If the formula is satisfiable, there is a state where b holds, o is applicable and invalidates b. If this is not the case for all operators $o \in \mathcal{O}$, then b is an invariant.

Case Studies

We now put the pieces of the previous sections together to derive lower bounds for our running examples. Interestingly, these lower bounds match well-known handcrafted bounds.

TSP

As shown earlier, in the TSP, our context splitting strategy splits $\text{move}(j)$ into operators $\text{move}(i, j)$ for all locations i with the additional precondition $(L = i)$ and a constant cost. Invariant $L \notin U$ is found by the templated invariant synthesis and shows that $\text{move}(i, i)$ is never applicable. Compiling away the base cases introduces operators $\text{move}(i, 0)$ that move back from some location i to the initial location 0 once the set of unvisited locations is empty $U = \emptyset$.

For the feature $f_{k \in U}$ operator $\text{move}(i, k)$ for $k \neq 0$ changes the value by exactly -1 which our methods detect as shown in Example 1. Other operators do not change the value of $f_{k \in U}$. The value of $f_{k \in U}$ in any base case is correctly determined as 0 making the net change $\Delta_{f_{k \in U}}$ equal to -1 if $k \in S(U)$ and 0 otherwise. Since all intervals are tight, the general net-change constraints (3) and (4) form an equation $\sum_i x_{\text{move}(i,k)} = [k \in S(U)]$ expressing that every unvisited location has to be entered exactly once.

For feature $f_{L=k}$ the change induced by any move is 1 if it enters k, -1 if it leaves k and 0 otherwise. The value in any base case is 1 for $k = 0$ and 0 otherwise. In state S the value is 1 for $k = S(L)$ and 0 otherwise. Again, all bounds are tight and can be determined automatically. The resulting constraint is $[S(L) = k] + \sum_i x_{\text{move}(i,k)} = \sum_j x_{\text{move}(k,j)} + [k = 0]$ expressing that the number of moves entering and leaving a location have to balance out (with correction terms for the current and the target location).

Minimizing the total cost over these constraints computes a network flow through the unvisited locations starting at $S(L)$ and ending in 0. It visits every location once but might have isolated subtours. This is a common LP relaxation of the TSP. The user-specified bound in the DyPDL benchmark computes the minimal costs of entering/leaving each unvisited location and maximizes over those two values. The net-change constraints also ensure that each unvisited location is entered and left once, so they dominate this bound.

Bin Packing

In our Bin Packing model, we derive a constraint $x_{\text{pack}(k)} = [k \in S(U)]$ for the feature $f_{k \in U}$ analogous to the TSP case.

The bounds on f_r depend on deriving the invariant $0 \leq r \leq q$ as we have seen in Examples 2 and 3. With this invariant, net change can be determined to be $\Delta_{f_r} = [0, q] - S(r)$. Operator pack$(k)$ deterministically changes the value by $-w_k$ while open induces a change in $[q - \max_i(w_i - 1), q]$ as shown in the examples. The resulting constraint is

$$(q - \max_i(w_i - 1))x_{\text{open}} - \sum\nolimits_i w_i x_{\text{pack}(i)} \leq q - S(r)$$
$$q x_{\text{open}} - \sum\nolimits_i w_i x_{\text{pack}(i)} \geq -S(r)$$

Plugging in the first constraint removes the dependency on variables $x_{\text{pack}(i)}$ and shows that these constraints provide upper and lower bounds on the number of bins to open. As we minimize costs, the upper bound is redundant. The lower bound is $x_{\text{open}} \geq (\sum_{i \in S(U)} w_i - S(r))/q$, which is one of three handcrafted lower bounds in the benchmark model.

Experiments

We evaluate our operator-counting framework on established OR problems that serve as benchmarks in existing DIDP literature (Kuroiwa and Beck 2023b; Narita, Kuroiwa, and Beck 2025), including the traveling salesperson problem (TSP), bin-packing problem (BPP), multi-commodity pick and delivery TSP (m-PDTSP), and the talent scheduling problem (TS). For TSP, we use TSPLIB instances with fewer than 100 cities (Reinelt 1991). For BPP, m-PDTSP, and TS, we use the same datasets as Kuroiwa and Beck.

Each benchmark was run on a single core of a 64-core AMD EPYC 7742 processor with a memory limit of 3.5 GB. Execution time was limited to 30 minutes for the search and to 2 hours for the operator-counting preprocessing. Operator-counting heuristics were evaluated with five different ways of computing invariants: no-invariant (ONI), template-based invariants (OTI), and abstract interpretation invariants using interval (OII), octagon (OOI), and polyhedra (OPI) domains. The heuristics were used in a cost-algebraic A* solver and their performance compared against the zero heuristic (ZO) and user-defined heuristics (USR).

Table 1 shows that the number of expansions and hence the informedness in operator-counting heuristic depends on the computed invariants. Compared to ONI, interval-based invariants (OII) reduce the required expansions by an order of magnitude in BPP. For TSP, we see this with the template-based approach (OTI). This is consistent with our case study and translates to better coverage in TSP.

In BPP, OII solves more instances to optimality than other operator-counting strategies, but unlike in TSP, it solves significantly less instances than the user-defined heuristic, pointing to an accuracy-time trade-off: the additional computational effort required to automatically derive a stronger heuristic does not yield performance gains in this domain.

For m-PDTSP, the operator-counting heuristic solves the largest number of problems with an order of magnitude fewer expansions. However, in contrast to TSP and BPP, automatic invariant computation does not contribute to tightening the LP in this problem. All solvers exhibit similar performance on Talent Scheduling, except for abstract interpretation using the polyhedra domain, which fails to complete.

Expansions	ONI	OII			OTI	USR	ZO
BPP (101)	210k	25k			210k	**15k**	210k
m-PDTSP (855)	**37k**	**37k**			**37k**	195k	292k
TS (183)	**948k**	**948k**			**948k**	**948k**	1173k
TSP (4)	223k	223k			**33k**	315k	326k
Coverage	**ONI**	**OII**	**OOI**	**OPI**	**OTI**	**USR**	**ZO**
BPP (1615)	109	210	1	0	111	**900**	217
m-PDTSP (1178)	**979**	975	869	327	978	899	891
TS (1000)	**184**	**184**	**184**	0	**184**	**184**	183
TSP (39)	4	4	4	0	**10**	4	4

Table 1: Average expansions on commonly solved instances, excluding OOI and OPI to get meaningful averages (top) and optimal coverage (bottom).

When considering preprocessing time in the 30 minutes time limit, OTI solves four fewer TSP instances, still outperforming other methods. In contrast, ONI solves a hundred fewer m-PDTSP instances than USR, losing its advantage.

Preprocessing itself is costly: computing possible changes consumes the majority of the time in ONI, OII, and OTI. Also, as the abstract domain becomes more expressive, invariant synthesis grows increasingly expensive, with OPI, using the most expressive polyhedra domain, spending 95% of its time on this; see the results in Supplementary material.

Preliminary tests on other domains show worse results. Our experiments show the method's potential and we see them as a proof of concept. More experiments are needed to establish average performance.

Conclusion and Future Work

We introduced operator-counting heuristics based on net-change constraints for DIDP. These heuristics achieve strong lower bounds, matching or exceeding some manually specified bounds without requiring input from a domain expert.

Preprocessing is the main bottleneck for our current approach as each combination of an operator and a feature requires solving up to 64 SMT problems. Generating features incrementally up to a threshold would reduce the load. We also plan to study a per-domain lifted computation of the intervals $\Delta_{o,f}$ to amortize the cost over multiple instances.

In classical planning, potential heuristics can quickly approximate net-change constraints with high accuracy. Adopting this idea to our setting could speed up the dual bound computation. Conversely it would be interesting to try our heuristic in a (non-simple) numeric planning setting.

Our framework can handle any commutative cost algebra using logic-based Benders decomposition. Extending it to non-commutative algebras is an interesting problem.

We use abstract interpretations to derive invariants for DIDP which offers other directions for future work. For example, we could introduce explicit variables for $\Delta_{o,f}$ inside our non-deterministic program and derive invariants on them without an SMT solver.

Acknowledgments

This work was supported by the Swiss National Science Foundation (SNSF) and the Natural Sciences and Engineering Research Council of Canada (NSERC) as part of the Alliance International Collaboration grant "Problem Solving with Domain-Independent Dynamic Programming: Theory and Practice". We would also like to thank Gidon Ernst for helpful discussions in the early stages of this project.

References

Bansal, K.; Barrett, C. W.; Reynolds, A.; and Tinelli, C. 2018. Reasoning with Finite Sets and Cardinality Constraints in SMT. *Logical Methods in Computer Science*, 14(4): 1–31.

Barbosa, H.; Barrett, C. W.; Brain, M.; Kremer, G.; Lachnitt, H.; Mann, M.; Mohamed, A.; Mohamed, M.; Niemetz, A.; Nötzli, A.; Ozdemir, A.; Preiner, M.; Reynolds, A.; Sheng, Y.; Tinelli, C.; and Zohar, Y. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In Fisman, D.; and Rosu, G., eds., *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2022)*, volume 13243 of *Lecture Notes in Computer Science*, 415–442. Springer-Verlag.

Barrett, C. W.; Sebastiani, R.; Seshia, S. A.; and Tinelli, C. 2021. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, 1267–1329. IOS Press, 2nd edition.

Bonet, B. 2013. An Admissible Heuristic for SAS$^+$ Planning Obtained from the State Equation. In Rossi, F., ed., *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, 2268–2274. AAAI Press.

Coles, A.; Coles, A.; Fox, M.; and Long, D. 2013. A Hybrid LP-RPG Heuristic for Modelling Numeric Resource Flows in Planning. *Journal of Artificial Intelligence Research*, 46: 343–412.

Coles, A.; Fox, M.; Long, D.; and Smith, A. 2008. A Hybrid Relaxed Planning Graph-LP Heuristic for Numeric Planning Domains. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E., eds., *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, 52–59. AAAI Press.

Cousot, P.; and Cousot, R. 1976. Static determination of dynamic properties of programs. In Robinet, B., ed., *Proceedings of the 2nd international symposium on Programming*, 106–130. Dunod.

Cousot, P.; and Cousot, R. 1977. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL 1977)*, 238–252. ACM.

Cousot, P.; and Halbwachs, N. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL 1978)*, 84–96. ACM.

Edelkamp, S.; Jabbar, S.; and Lluch Lafuente, A. 2005. Cost-Algebraic Heuristic Search. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2005)*, 1362–1367. AAAI Press.

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

Jeannet, B.; and Miné, A. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In Bouajjani, A.; and Maler, O., eds., *Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009)*, volume 5643 of *Lecture Notes in Computer Science*, 661–667. Springer-Verlag.

Kautz, H. A.; and Walser, J. P. 1999. State-space Planning by Integer Optimization. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI 1999)*, 526–533. AAAI Press.

Kuoriwa, R.; Shleyfman, A.; Piacentini, C.; Castro, M. P.; and Beck, J. C. 2021. LM-Cut and Operator Counting Heuristics for Optimal Numeric Planning with Simple Conditions. In Goldman, R. P.; Biundo, S.; and Katz, M., eds., *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*, 210–218. AAAI Press.

Kuroiwa, R. 2024. *Domain-Independent Dynamic Programming*. Ph.D. thesis, University of Toronto.

Kuroiwa, R.; and Beck, J. C. 2023a. Domain-Independent Dynamic Programming: Generic State Space Search for Combinatorial Optimization. In Koenig, S.; Stern, R.; and Vallati, M., eds., *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling (ICAPS 2023)*, 236–244. AAAI Press.

Kuroiwa, R.; and Beck, J. C. 2023b. Solving Domain-Independent Dynamic Programming Problems with Anytime Heuristic Search. In Koenig, S.; Stern, R.; and Vallati, M., eds., *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling (ICAPS 2023)*, 245–253. AAAI Press.

Miné, A. 2006. The octagon abstract domain. *Higher Order and Symbolic Computation*, 19(1): 31–100.

Miné, A. 2017. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Foundations and Trends in Programming Languages*, 4(3–4): 120–372.

Narita, M.; Kuroiwa, R.; and Beck, J. C. 2025. Reinforcement Learning-based Heuristics to Guide Domain-Independent Dynamic Programming. In Tack, G., ed., *Proceedings of the 22nd International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2025)*, 137–154. Springer-Verlag.

Piacentini, C.; Castro, M. P.; Cire, A. A.; and Beck, J. C. 2018a. Compiling Optimal Numeric Planning to Mixed Integer Linear Programming. In de Weerdt, M.; Koenig, S.; Röger, G.; and Spaan, M., eds., *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018)*, 383–387. AAAI Press.

Piacentini, C.; Castro, M. P.; Cire, A. A.; and Beck, J. C. 2018b. Linear and Integer Programming-Based Heuristics for Cost-Optimal Numeric Planning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*, 6254–6261. AAAI Press.

Pommerening, F.; Helmert, M.; Röger, G.; and Seipp, J. 2015. From Non-Negative to General Operator Cost Partitioning. In Bonet, B.; and Koenig, S., eds., *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, 3335–3341. AAAI Press.

Reinelt, G. 1991. TSPLIB—A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3(4): 376–384.

Röger, G.; Pommerening, F.; and Helmert, M. 2014. Optimal Planning in the Presence of Conditional Effects: Extending LM-Cut with Context Splitting. In Schaub, T.; Friedrich, G.; and O'Sullivan, B., eds., *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, 765–770. IOS Press.

Singh, A.; Pommerening, F.; Schindler, T.; Beck, J.; and Helmert, M. 2025. Code, benchmarks, experiment data, and additional results for the ICAPS 2026 paper "Operator-Counting Heuristics for Domain-Independent Dynamic Programming". https://doi.org/10.5281/zenodo.19114180.

van den Briel, M.; Benton, J.; Kambhampati, S.; and Vossen, T. 2007. An LP-Based Heuristic for Optimal Planning. In Bessiere, C., ed., *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *Lecture Notes in Computer Science*, 651–665. Springer-Verlag.

van den Briel, M.; Vossen, T.; and Kambhampati, S. 2005. Reviving Integer Programming Approaches for AI Planning: A Branch-and-Cut Framework. In Biundo, S.; Myers, K.; and Rajan, K., eds., *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, 310–319. AAAI Press.

Zhang, W. 1998. Complete Anytime Beam Search. In Rich, C.; and Mostow, J., eds., *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI 1998)*, 425–430. AAAI Press.